Acceleration Techniques for Sparse Recovery Based Plane-wave Decomposition of a Sound Field

Mahendra Samarawickrama
School of Electrical & Information Engineering
University of Sydney

A thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

October 2017
I dedicate this thesis to my lovely wife who has been a constant source of love, motivation, inspiration and support during my PhD.

This work is also dedicated to my parents and I hope this achievement will complete the dream they had for me all those many years ago when they chose to give me the best education they could.
Acknowledgements

First and foremost, I would like to express my sincere thanks to Professor Craig Jin for his continuous support and guidance during this study. His unique ideas, experience, and encouragement were immensely helpful to drive this thesis, and I am grateful for his inspirational patience, and insightful advice. Further, I would also like to thank my co-supervisor, Dr. Nicolas Epain for this guidance and constructive feedback on my work. His help, particularly in proof-reading my thesis and publications, are much appreciated.

I am deeply indebted to my dear wife Prasadi Domingo for the tremendous support, love, and understanding over past few years that encouraged me to accomplish this milestone. She has been a great pillar of my success from my undergraduate period. My daughter Janudi Samarawickrama and son Thejayu Samarawickrama played a significant part of my life during this period by keeping me cheered up with their adorable ways of love in a period I wanted them the most. My loving parents Vinson Samarawickrama and Dharmi Ranaweera helped to complete this journey by providing their extensive support morally and inspiring me every day to gain this achievement. They help me to succeed in life, not only educationally, but emotionally, mentally, and spiritually. I also thank my in-laws Premasiri Domingo and Sujatha Dodangoda who played invaluable roles in this journey by providing their extensive support and encouragement.

Many people helped me along the way and I want to thank them for all their help, support, interest, and valuable hints. I take this opportunity to thank all my friends and colleagues at Computing and Audio Research Lab [CARLAB] and Computer Engineering Lab [CEL] in The University of Sydney for creating a friendly atmosphere to conduct my research. I have met so many wonderful friends during my studies whose friendships will last a lifetime.

I like to thank School of Information Technologies [SIT] and Computing and Audio Research Lab [CARLAB] in The University of Sydney for enabling access to their HPC machines to perform my experiments. I was also supported by:
• Multi-modal Australian ScienceS Imaging and Visualisation Environment (MASSIVE)\textsuperscript{[52]},

• National Computational Infrastructure (NCI) by providing access to (RAI-JIN) HPC centre, and

• The National eResearch Collaboration Tools and Resources (NECTAR).

Further, I want to appreciate and acknowledge The University of Sydney for offering me very competitive full scholarships: University of Sydney International Scholarship (USydIS), University of Sydney Postgraduate Awards (UPA), and Norman I Price Supplementary Scholarship to pursue my PhD, without which I would not have been able to pursue this study at such a prestigious university. Last but not least, I will make this an opportunity to look back and thank my teachers from The University of Moratuwa, Royal College Colombo and Sri Sumangala College Panadura in Sri Lanka for laying a strong foundation of my life and academic adventures without which I would not have come this far.
Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged. Specifically, I would like to acknowledge that the sections of 3.2, 3.3, 3.4, 3.4.1 and 3.4.2 in the background are taken from publications of Professor Craig Jin.
Abstract

Plane-wave decomposition by sparse recovery is a reliable and accurate technique for plane-wave decomposition which can be used for source localization, beamforming, etc. In this work, we introduce techniques to accelerate the plane-wave decomposition by sparse recovery. The method consists of two main algorithms which are spherical Fourier transformation (SFT) and sparse recovery. Comparing the two algorithms, the sparse recovery is the most computationally intensive. We implement the SFT on an FPGA and the sparse recovery on a multithreaded computing platform. Then the multithreaded computing platform could be fully utilized for the sparse recovery. On the other hand, implementing the SFT on an FPGA helps to flexibly integrate the microphones and improve the portability of the microphone array.

For implementing the SFT on an FPGA, we develop a scalable FPGA design model that enables the quick design of the SFT architecture on FPGAs. The model considers the number of microphones, the number of SFT channels and the cost of the FPGA and provides the design of a resource optimized and cost-effective FPGA architecture as the output. Then we investigate the performance of the sparse recovery algorithm executed on various multithreaded computing platforms (i.e., chip-multiprocessor, multiprocessor, GPU, manycore). Finally, we investigate the influence of modifying the dictionary size on the computational performance and the accuracy of the sparse recovery algorithms. We introduce novel sparse-recovery techniques which use non-uniform dictionaries to improve the performance of the sparse recovery on a parallel architecture.
# Contents

1 Introduction .......................... 1
   1.1 Thesis Outline .......................... 4

2 Literature Review ......................... 6
   2.1 FPGA-based Audio Acquisition and Transmission System .................. 6
   2.2 FPGA-based Beamforming System .................................. 12
   2.3 Solving Linear-Algebra Problems on Multithreaded Platforms .......... 23

3 Background ............................. 30
   3.1 Introduction .................................... 30
   3.2 Spherical Microphone Arrays .......................... 30
   3.3 Spherical Fourier Transformation of the Audio Signals ............... 31
   3.4 Plane-wave Decomposition .....................................
      3.4.1 Sparse Plane-wave Decomposition ......................... 34
      3.4.2 The Iteratively-reweighted Least-square Algorithm ......... 35
   3.5 Introduction to FPGAs ............................... 37
      3.5.1 CLB Resources on FPGAs .......................... 37
      3.5.2 Block Memory Resources on FPGAs ...................... 39
      3.5.3 DSP Resources on FPGAs ................................. 40
      3.5.4 Other Important Features of FPGAs ..................... 41
      3.5.5 FPGA Design Flow .................................. 42
   3.6 The Concepts of FFT and Its Implementation on FPGAs ............. 42
      3.6.1 FPGA-based FFT implementation ......................... 48
         3.6.1.1 The Precision of FFTs .......................... 50
         3.6.1.2 The Configuration of Complex Multipliers in FFTs .... 51
   3.7 Introduction to Multithreaded Computing Architectures ........... 52
      3.7.1 Processor Performance and Memory Latency ................. 54
      3.7.2 Introduction to Memory Models of Computations .......... 55
3.7.3 Introduction to Multi-processor, Multi-core and Many-thread Architectures ........................................ 58

4 Development of a FPGA-based Audio Preprocessing System ................................................................. 65

4.1 Introduction ............................................................................................................................................. 65
4.2 FPGA Platform ..................................................................................................................................... 67
4.3 Audio-acquisition Board ....................................................................................................................... 67
4.4 ADC-configuration Subsystem ............................................................................................................. 69
  4.4.1 Implementation of the ADC-configuration Subsystem ...................................................................... 71
4.5 Audio-acquisition Subsystem .............................................................................................................. 74
4.6 UDP/IP Data Transmission Subsystem ................................................................................................ 78
4.7 DDR3-memory Subsystem ................................................................................................................... 81
4.8 Spherical Fourier Transformation (SFT) Subsystem ............................................................................ 86

5 Implementation Model for FPGA-based Spherical Fourier Transformation (SFT) ...................................... 88

5.1 Introduction ............................................................................................................................................. 88
5.2 Implementation of the Spherical Fourier Transformation (SFT) ........................................................ 88
5.3 SFT Architecture .................................................................................................................................. 95
  5.3.1 Microphone Data Buffer ..................................................................................................................... 95
  5.3.2 Implementation of the FFT Process .................................................................................................. 96
    5.3.2.1 Block RAM requirement of FFT implementation ......................................................................... 96
    5.3.2.2 DSP block requirement of FFT implementation ........................................................................... 97
    5.3.2.3 Latencies of different FFT configurations ................................................................................. 98
    5.3.2.4 Summary FFT configurations .................................................................................................. 99
  5.3.3 FFT Output Buffer ............................................................................................................................. 99
  5.3.4 Parallel SFT Filters ......................................................................................................................... 100
  5.3.5 Filter Output Buffer .......................................................................................................................... 103
  5.3.6 SFT Coefficient Buffer .................................................................................................................... 104
  5.3.7 IFFT of the SFT signals .................................................................................................................. 105
  5.3.8 Overlap and Add the IFFT Output .................................................................................................... 107
5.4 Evaluate the architectures against the timing constraints ....................................................................... 108
5.5 Evaluate the resource consumption against the architectural parameters .............................................. 111
5.6 Results and Discussion ........................................................................................................................ 114
5.7 Conclusions .......................................................................................................................................... 121
### 6 Analysis of the Performance of the Sparse Recovery on Multithreaded Platforms

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction</td>
<td>123</td>
</tr>
<tr>
<td>6.2 Computational Complexity of the IRLS Computation</td>
<td>126</td>
</tr>
<tr>
<td>6.3 Methodology</td>
<td>133</td>
</tr>
<tr>
<td>6.3.1 Implementation of the IRLS algorithm using OpenMP</td>
<td>133</td>
</tr>
<tr>
<td>6.3.2 Implementation of the IRLS algorithm using CUDA</td>
<td>135</td>
</tr>
<tr>
<td>6.3.3 Specifications of the Computing Platforms</td>
<td>138</td>
</tr>
<tr>
<td>6.3.4 Simulation Paradigm</td>
<td>138</td>
</tr>
<tr>
<td>6.4 Results</td>
<td>139</td>
</tr>
<tr>
<td>6.5 Discussion</td>
<td>142</td>
</tr>
<tr>
<td>6.6 Conclusion</td>
<td>148</td>
</tr>
</tbody>
</table>

### 7 Sparse Recovery Using Non-Uniform Spatial Dictionaries

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction</td>
<td>151</td>
</tr>
<tr>
<td>7.2 The proof-of-concept of using non-uniform spatial dictionary for sparse recovery</td>
<td>152</td>
</tr>
<tr>
<td>7.2.1 Influence of spatial resolution</td>
<td>156</td>
</tr>
<tr>
<td>7.2.2 Robustness to diffuse noise</td>
<td>156</td>
</tr>
<tr>
<td>7.2.3 Robustness to a back hemisphere source</td>
<td>157</td>
</tr>
<tr>
<td>7.3 Sparse plane-wave decomposition on streaming frequency-domain SFT signals</td>
<td>159</td>
</tr>
<tr>
<td>7.4 Algorithms of non-uniform spatial dictionaries based sparse recovery</td>
<td>160</td>
</tr>
<tr>
<td>7.4.1 Dictionary refining method</td>
<td>161</td>
</tr>
<tr>
<td>7.4.2 Dictionary subdividing method</td>
<td>161</td>
</tr>
<tr>
<td>7.4.3 Combined method</td>
<td>163</td>
</tr>
<tr>
<td>7.5 Evaluation of non-uniform dictionary based sparse plane-wave decomposition</td>
<td>168</td>
</tr>
<tr>
<td>7.5.1 Visual comparison of the quality of the results</td>
<td>168</td>
</tr>
<tr>
<td>7.5.2 The description of the three experiment conditions</td>
<td>170</td>
</tr>
<tr>
<td>7.5.3 The quality evaluation metrics</td>
<td>171</td>
</tr>
<tr>
<td>7.5.3.1 Energy-map mismatch</td>
<td>172</td>
</tr>
<tr>
<td>7.5.3.2 Angular-error estimation</td>
<td>173</td>
</tr>
<tr>
<td>7.6 Results and Discussion</td>
<td>173</td>
</tr>
<tr>
<td>7.7 Conclusion</td>
<td>202</td>
</tr>
</tbody>
</table>

### 8 Conclusions

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Summary</td>
<td>203</td>
</tr>
</tbody>
</table>

### A I2C Protocol

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 I2C Protocol</td>
<td>207</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The system of sound-field analysis which we studied in this thesis</td>
</tr>
<tr>
<td>2.1</td>
<td>32 Channel USB2.0 spherical microphone array for 3D audio recording and playback</td>
</tr>
<tr>
<td>2.2</td>
<td>An FPGA based system to transmit microphone array data to a computer [164]</td>
</tr>
<tr>
<td>2.3</td>
<td>FPGA-based beamforming and recording system for spherical microphone arrays</td>
</tr>
<tr>
<td>2.4</td>
<td>Block diagram of a FPGA and DSP based audio acquisition and transmission system</td>
</tr>
<tr>
<td>2.5</td>
<td>External view of the state-of-the-art custom HOA recording system</td>
</tr>
<tr>
<td>2.6</td>
<td>Dante Brooklyn II PDK</td>
</tr>
<tr>
<td>2.7</td>
<td>Configuration of the helmet-mounted microphone array and the apparatus [162]</td>
</tr>
<tr>
<td>2.8</td>
<td>Platform and its schematic which prototype FPGA-based wireless microphone array [110]</td>
</tr>
<tr>
<td>2.9</td>
<td>Experiment setup of the hat-type hearing system</td>
</tr>
<tr>
<td>2.10</td>
<td>The hardware blocks of SoundCompass [123]</td>
</tr>
<tr>
<td>2.11</td>
<td>The platform of 52 microphone array with a Virtex-4 FPGA [126]</td>
</tr>
<tr>
<td>2.12</td>
<td>64 microphone array which is interfaced with Xilinx Zynq 7010 FPGA based platform [64]</td>
</tr>
<tr>
<td>2.13</td>
<td>The FPGA platform which performs real-time beamforming using 20 MHz 64-channel [20]</td>
</tr>
<tr>
<td>2.14</td>
<td>The multiline delay-sum beamformer which consists of DDR memory for store beamforming coefficients [83]</td>
</tr>
<tr>
<td>2.15</td>
<td>The hardware-software codesign implemented on Zynq-7000 SoC for beamforming [77]</td>
</tr>
<tr>
<td>2.16</td>
<td>The source localization and speech enhancement system which is implemented on Zynq 7020 SoC using direct memory access (DMA) IP and onboard DDR memory [71]</td>
</tr>
<tr>
<td>2.17</td>
<td>CUDA grid of thread blocks and block of threads</td>
</tr>
</tbody>
</table>
3.1 The dual-radius SMA prototype built at CARLab [68].
3.2 The notations used to describe the geometry of an SMA are illustrated.
3.3 A segment of the CLB in Virtex-6 FPGA which consist of an LUT and 2 FFs.
3.4 The schematic diagram of Xilinx CLB which consist of 2 Slices.
3.5 Timing diagram of the dual-port block memory data access protocol.
3.6 Schematic diagram of Xilinx 25×18 bits DSP block.
3.7 The symmetry and repetition of twiddle factors when N is 2, 4 and 8.
3.8 The basic radix-2 architecture.
3.9 The basic radix-2 butterfly unit.
3.10 The 8-input butterfly diagram.
3.11 The basic radix-4 butterfly unit.
3.12 Radix-2 based 4-input butterfly diagram.
3.13 The radix-2 burst I/O configuration.
3.14 The radix-2 lite burst I/O configuration.
3.15 The radix-4 burst I/O configuration.
3.16 IEEE-754 32-bit floating point data format.
3.17 Different memory models which can be used to understand parallel computations.
3.18 Architecture of the multiprocessor platform.
3.19 Memory architecture of a multiprocessor platform.
3.20 Architecture of Intel Xeon-Phi coprocessor platform.
3.21 Architecture of Nvidia GPU platform.
3.22 Memory architecture of Intel Xeon-Phi 5110P coprocessor.
3.23 Memory architecture of Nvidia K40 GPU.

4.1 The block diagram of the audio-preprocessing system.
4.2 Xilinx ML605 FPGA-based development platform.
4.3 The highlighted section in the system is the audio-acquisition board.
4.4 Typical connections of TLV320ADC3101 ADC.
4.5 The connection between audio-acquisition board and ML605-FPGA platform.
4.6 The highlighted section in the system is the ADC-configuration subsystem.
4.7 The interfacing of 4 ADCs with common I2C master.
4.8 Images of the ADC module and the ADC motherboard.
4.9 ADC-configuration subsystem.
4.10 The ADC-configuration subsystem which consists of 8 I2C IP cores and a Microblaze processor.
6.2 MUSIC performance on (a) Closely spaced signals; (b) Effect of coherent signals .......................................................... 124
6.3 Functional block diagram and data-flow diagram of a plane-wave decomposition system ................................................. 126
6.4 Schematic of thread affinity types compact, scatter and balanced .......................................................... 135
6.5 Effective computational rate (ECR) against the number of IRLS problems solved on different architectures .................. 140
6.6 Curve fit the peak ECR measurements of different dictionaries with the reciprocal function of the dictionary resolution .......................................................... 141
6.7 The performance of batch Cholesky implementation on Nvidia K40 GPU [34] .......................................................... 142
6.8 Increase of the cache-miss rate when increasing the number of IRLS problems .......................................................... 144
6.9 The balance of block allocation on the GPU against the requested number of IRLS problems .......................................................... 145

7.1 Acoustic energy maps for the front hemisphere of space are shown. These maps were obtained using sparse recovery with non-uniform spatial dictionaries. $N_r$ and $N_b$ indicate the size of the front hemisphere and back hemisphere dictionaries, respectively. Source 5 is not shown as it is located in the back hemisphere; it had an amplitude of 0 dB. Diffuse noise was added to the SFT signals at a level of -20 dB. Diffuse noise was added to the SFT signals at a level of -20 dB. .......................................................... 154
7.2 Influence of the spatial resolution of the dictionary in the front and back hemisphere of space on the accuracy of the energy map. Source 5 (located at the back) has an amplitude of 0 dB and the SFT signals have an SNR of 20 dB. .......................................................... 155
7.3 Influence of diffuse noise on the accuracy of the energy map. This figure compares the results obtained with: a) a uniform dictionary ($N_f = 401$ and $N_b = 369$); and b) a non-uniform dictionary with fewer directions in the back hemisphere ($N_f = 401$ and $N_b = 49$), as a function of the SNR. Note that no source is present in the back hemisphere. .......................................................... 157
7.4 Influence of a source in the back hemisphere on the accuracy of the energy map. This figure compares the results obtained with: a) a uniform dictionary ($N_f = 401$ and $N_b = 369$); and b) a non-uniform dictionary with fewer directions in the back hemisphere ($N_f = 401$ and $N_b = 49$), as a function of the amplitude of Source 5. Note that no noise is present. .......................................................... 158
7.5 The dictionary-refining method .......................................................................................................................... 162
7.6 The subdivision of the high resolution dictionary .......................................................................................................................... 164
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.7</td>
<td>The positioning of the sound sources to visually compare the quality of the sparse recovery performed using uniform and non-uniform dictionaries... 169</td>
</tr>
<tr>
<td>7.8</td>
<td>The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the energy threshold of refining the dictionary. 191</td>
</tr>
<tr>
<td>7.9</td>
<td>The change of the accuracy and acceleration against the energy threshold of refining the dictionary and the number of sound sources in the environment 192</td>
</tr>
<tr>
<td>7.10</td>
<td>The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the number of sound sources in the environment 193</td>
</tr>
<tr>
<td>7.11</td>
<td>The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the energy threshold of refining the dictionary. 195</td>
</tr>
<tr>
<td>7.12</td>
<td>The change of the accuracy and acceleration against the energy threshold of refining the dictionary and the number of sound sources in the environment 196</td>
</tr>
<tr>
<td>7.13</td>
<td>The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the number of sound sources in the environment 197</td>
</tr>
<tr>
<td>7.14</td>
<td>The energy map of uniform and non-uniform dictionary based sparse recovery when applied for real measurements of the sound scene. 198</td>
</tr>
<tr>
<td>7.15</td>
<td>The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the energy threshold of refining the dictionary. 199</td>
</tr>
<tr>
<td>7.16</td>
<td>The change of the accuracy and acceleration against the number of sound sources. 200</td>
</tr>
<tr>
<td>7.17</td>
<td>The change of the accuracy and acceleration against the number of subdivisions of the dictionary. 201</td>
</tr>
<tr>
<td>A.1</td>
<td>The timing diagram of the Data Transfer on the I2C bus. 208</td>
</tr>
<tr>
<td>B.1</td>
<td>I2S timing diagram. 209</td>
</tr>
<tr>
<td>B.2</td>
<td>The ADC is driving in slave mode with independent MCLK. 210</td>
</tr>
<tr>
<td>B.3</td>
<td>The timing diagram related to an I2S slave transmitter having independent MCLK. 210</td>
</tr>
<tr>
<td>B.4</td>
<td>Some possible integrations of I2S clocks in slave mode. 211</td>
</tr>
<tr>
<td>B.5</td>
<td>The integration of the ADC I2S clocks in master mode. 211</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The specifications of Nvidia K40 GPU which are required to calculate the</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>number of active-thread blocks and the GPU occupancy.</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>The permutation of bit-revered order of FFT input.</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>Resource utilization and performance of different complex-multiplier config-</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>urations implemented with different resources.</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison of different market standards for data transmission [90]</td>
<td>79</td>
</tr>
<tr>
<td>4.2</td>
<td>The resource utilization of the UDP/IP transmission architecture on Virtex-6</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>Utilization of 18k block memory primitives in the 3 FFT configurations</td>
<td>97</td>
</tr>
<tr>
<td>5.2</td>
<td>Utilization of DSP-blocks in different FFT configurations</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>An estimation of latencies associated with 3 FFT configurations</td>
<td>99</td>
</tr>
<tr>
<td>5.4</td>
<td>Typical latencies of different FFT configurations and their maximum oper-</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>ating frequencies</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Resource utilization and performance of different FFT configurations</td>
<td>100</td>
</tr>
<tr>
<td>5.6</td>
<td>The resource utilization of the complex multiplier and complex adder</td>
<td>103</td>
</tr>
<tr>
<td>5.7</td>
<td>Resource utilization of the filter bank which consists of $p$ filters each</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>having $q$ parallel data paths. The 2 filter configurations are correspond-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ing to the 2 configurations of the complex multiplier which are highlighted in Table 5.6</td>
<td>103</td>
</tr>
<tr>
<td>5.8</td>
<td>The memory requirement to store the filter coefficients of 3$^{rd}$ order</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>SFT</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>Available block memories in different device versions of Virtex-6 FPGA.</td>
<td>105</td>
</tr>
<tr>
<td>5.10</td>
<td>Resource utilization of the overlap and add stage which having $v$ data</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>paths</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>Measured DDR3 memory bandwidth in some FPGA-based applications.</td>
<td>110</td>
</tr>
<tr>
<td>5.12</td>
<td>Arithmetic and logic resource utilization of the SFT architecture.</td>
<td>112</td>
</tr>
<tr>
<td>5.13</td>
<td>Block RAM utilization of the SFT architecture</td>
<td>112</td>
</tr>
<tr>
<td>5.14</td>
<td>Resource utilization of some additional modules of the SFT architecture.</td>
<td>113</td>
</tr>
<tr>
<td>5.15</td>
<td>FPGA feature summary by device</td>
<td>113</td>
</tr>
</tbody>
</table>
5.16 Comparison of the calculated resource utilization against the post-implementation results of selected architectures on Xilinx Artix and Kintex FPGAs. The maximum operating frequency of the architecture and its implementation time are also given. Note, in the system, \( n \) is the number of microphones, and \( m \) is the number of SFT signals.

5.17 The configuration of the resource optimized SFT architectures on the selected FPGAs.

5.18 The cost effective FPGA implementations of different SFT specifications.

6.1 The Number of FLOPs involves in the three methods of BLAS routines for computing the IRLS.

6.2 Specifications of multi-threaded architectures which have been used for analyses the performance of sparse recovery process.

6.3 The curve-fit coefficients (\( \kappa \)) related to different platforms.

6.4 The relative performance of the architectures against Intel Core-i3 370M processor.

6.5 The rate of solving the IRLS problems on the selected architectures for an arbitrary dictionary resolution.

6.6 The relative performance of the architectures against Intel Core-i3 370M processor.

7.1 Position and the strength of the sources.

7.2 The positions of the sound sources which are used for visually comparison of the quality of the sparse recovery.

7.3 The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different reverberant sound scenes containing 2 to 12 sources. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The dictionary is subdivided into 4 in the subdivision and combined methods. The DRR is 0.7 and SNR is 30dB.

7.4 The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different anechoic sound scenes by varying the threshold of the weights for refining the dictionary as -60dB, -45dB, -30dB, -15dB and -2dB. The dictionary is subdivided into 4 in the subdivision and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is 30dB.
7.5 The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different anechoic sound scenes by varying the subdivisions of the dictionary into 2, 4 and 8 in the subdivision and combined methods. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is 30dB.

7.6 The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different anechoic sound scenes. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The subdivision of the dictionary is 4 in the subdivision and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is varied as 60dB, 30dB, 15dB, 5dB and 0dB.

7.7 The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different reverberant sound scenes. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The subdivision of the dictionary is 4 in the subdivision and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is 30dB. The DRR is varied as 0.9, 0.7, 0.5 and 0.3.

7.8 The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for real sound scenes generated using the measured room impulse responses. There are 1 to 3 sources located in the room. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The dictionary is subdivided into 4 in the subdivision and combined methods.
Listings

C.1 The structure of the I2C program runs on Microblaze processor to program the ADCs. .................................................. 212
C.2 The bare-metal C program which runs on Microblaze processor to configure the ADC Modules using Xilinx I2C modules. ......................... 215
D.1 The method of solving multiple IRLS problems in parallel using C and OpenMP. The code performs Algorithm 6 in each OpenMP thread. The code demonstrates the sparse recovery for order-3 SFT signals with a dictionary having 230 plane-wave resolution. ......................... 219
E.1 The method of solving multiple IRLS problems in parallel using CUDA. The presented case demonstrates the sparse recovery for order-3 SFT signals with a dictionary having 230 plane-wave resolution. ......................... 224
Abbreviations

ADC       Analog to Digital Converter
BRAM      Block RAM (Random-access Memory)
CLB       Configurable Logic Block
CMP       Chip Multiprocessor
DDR       Double Data Rate
DSP       Digital Signal Processor
ECR       Effective Computational Rate
EDK       Embedded Development Kit
EEPROM    Electrically Erasable Programmable ROM
FFT       Fast Fourier Transform
FPGA      Field-Programmable Gate Array
GMII      Gigabit Media Independent Interface
GPGPU     General-purpose Graphics Processing Unit
HDL       Hardware Description Language
HLL       High-Level Language
IFFT      Inverse Fast Fourier Transform
IRLS      Iteratively Reweighted Least Squares
LMB       Local Memory Bus
LUT       Lookup Table
MAC       Media Access Control
MII       Media Independent Interface
NPI       Native Port Interface
PHY       Physical-side interface (Ethernet)
PLB       Processor Local Bus
PRAM      Parallel Random-access Machine
ROM       Read-Only Memory
SFT       Spherical Fourier Transform
SMA       Spherical Microphone Array
SNR       Signal to Noise Ratio
SVD       Singular Value Decomposition
SoC       System on Chip
STFT      Short-time Fourier Transform
TEMAC     Tri-mode Ethernet MAC (Media Access Control)
XPS       Xilinx Platform Studio
Chapter 1

Introduction

Real-time sound source localization is used in many practical applications. Some of them are, applications in acoustic cameras [74, 76, 82], road traffic monitoring [6, 124, 161], hands-free teleconference systems [22, 88], locating threats (i.e., snipers, UAVs) in warfare [110, 162], augmented-reality applications [66, 89], voice-based human-computer interfaces [59, 109], and various assisted listening applications [32, 51, 91]. These applications need real-time responses. In acoustic cameras, sound needs to be processed synchronously with the video. The real-time localization of sound sources in a visual-sonic scene improves the practical usability of the system. In applications of road traffic monitoring, real-time sound source localization can be used to identify noise vehicles and record them. In hands-free teleconference systems, it is important to quickly identify the speaker and perform beamforming to improve the quality of the voice recording. In an application like locating threats in warfare, it is critical to do it as fast as possible. In augmented-reality applications, audio and video are required to be processed synchronously in real-time. In future, the machines will be much capable of interacting with humans and the environment. To interact, machines need real-time localization of the action, moment or an incident with an audio input [14]. Therefore, real-time sound source localization is an important task to be achieved.

The sparse recovery based plane-wave decomposition of a sound field has been explored in several recent works [37, 38, 92, 132, 134, 136, 142, 143], which can be used for sound-source localization with high accuracy. It provides us an acoustic energy map which is an image showing the acoustic energy incoming from each direction in space. However, the associated algorithms and processors are computationally intensive, hence challenging to perform in real-time.

The focus of this thesis is to accelerate the sparse recovery based plane-wave decomposition of a sound field. In this thesis, we studied a particular system of sound-field analysis (see Fig 1.1) which consists of spherical microphone array (SMA), an embedded system for audio acquisition, preprocessing and transmission, and multithreaded computing platform.
Figure 1.1: The system of sound-field analysis which we studied in this thesis. It consists of spherical microphone array (SMA), an embedded system for audio acquisition, preprocessing and transmission, and multithreaded computing platform (e.g., GPU, Intel-Phi, CPU) for sparse recovery-based plane-wave decomposition.

(e.g., GPU, Intel-Phi, CPU) for sparse recovery-based plane-wave decomposition. We studied how to improve the embedded system for effectively preprocessing, types of a computing platform and how effective they are in implementing the sparse recovery-based plane-wave decomposition algorithm. Then we studied some methods to improve the effectiveness and the performance of the sparse recovery-based plane-wave decomposition algorithm.

Three contributions can be identified. In Chapter 5, a designing model of a scalable FPGA-based spherical Fourier transform (SFT) architectures is presented. The model is useful,
• To identify the feasibility of implementation of a given SFT system on a selected FPGA,

• To identify a resource optimized FPGA architecture for SFT,

• To identify a cost-effective FPGA for a given SFT requirement,

• To identify the bottleneck (resource or I/O bandwidth) of implementation of an SFT on a given FPGA,

• To find the maximum number of supported microphones by a FPGA for a given SFT-order,

• To find the highest supported SFT-order by a FPGA for given number of microphones.

Since the SFT algorithm is highly parameterizable, this model makes the design process easy and fast facilitates the FPGA design process. We presented an algorithm to calculate the model parameters and summarized the results in a table.

In Chapter 6, acceleration techniques of the sparse-recovery algorithm on multithreaded architectures are presented. At the beginning of the chapter, we analyzed contemporary beamforming techniques (i.e., Delay-and-Sum, MVDR, MUSIC) and compared and contrast their performance. Particularly, we focused on resolving proximity sources and coherent sources. Then we presented the sparse recovery as an efficient technique for super-resolution source localization even with the presence of coherent sources. Next, we analyzed a state-of-the-art GPU implementation of the MVDR beamformer and used that implementation as a benchmark to analyze the sparse-recovery technique and understood its relative complexity and suitability for multithreaded implementation. Then, we comprehensively studied the computational complexities and parallel processing opportunities of the sparse-recovery process. We analyzed the sparse-recovery process using relevant linear-algebraic methods and multithreaded-architectural features to understand the most computationally efficient ways to perform it on multithreaded architectures.

In Chapter 7, a novel non-uniform dictionary based sparse-recovery technique is presented. Unlike the general method of sparse recovery, in the non-uniform dictionary-based method, the dictionary has high resolution in the regions of interest and low resolution in the other regions. The non-uniform dictionaries are used to either reduce the computational complexity and/or subdivide the problem such that they can be accelerated on parallel computing architectures. Three dictionary refining methods are described. The first method reduces the dictionary size by progressively refining the dictionary used. The second method reduces the dictionary size by subdividing the dictionary and then solving
multiple plane-wave decomposition problems in parallel. The third method combines the previous two methods. The performance and the accuracy of the dictionary refining methods are evaluated with varying number of sources, signal-to-noise ratio, reverberation and algorithm parameters.

1.1 Thesis Outline

Following this introduction, the remainder of the thesis is organized as follows:

- **Chapter 2** presents the relevant literature review covering signal processing algorithms for sound fields and implementation of parallel algorithms on FPGAs and multithreaded platforms.

- **Chapter 3** presents the relevant background knowledge related to the research fields covered in this thesis. We present an introduction to spherical microphone arrays and the mathematics of the spherical Fourier transform. We also present a sparse-recovery method for plane-wave decomposition. We then provide a description of FPGA hardware and the implementation of an FFT on FPGA hardware to support the implementation of beamforming using an FPGA.

- **Chapter 4** presents a development of the FPGA-based audio preprocessing system. The main motivations of the embedded audio preprocessing system are to fully allocate the resources of the main computing platform to sparse plane-wave decomposition and improve the portability of the spherical microphone array (SMA). The preprocessing system is re-configurable. It also helps to acquire SMA data and transmit the preprocessed data to a distant computer.

- **Chapter 5** presents a method to determine the design of the spherical Fourier transform (SFT) architecture on a Field Programmable Gate Array (FPGA). The method accounts the number of microphones and desired number of SFT signals to calculate the required amount of FPGA resources for feasible designs. Then a resource optimized architecture is identified and the FPGA is determined.

- **Chapter 6** presents an analysis of the computational complexity of the sparse-recovery algorithm and the performance of the sparse-recovery algorithm executed on selected parallel computing platforms (i.e., Chip-multiprocessor, Multiprocessor, GPU, Manycore). Sparse recovery is performed in the frequency domain and frequency-specific sparse-recovery problems are assigned to the individual thread. We investigated possible techniques to accelerate the algorithm by reducing the computational complexity.
• Chapter 7 presents the development of methods to reduce the size of the plane-wave dictionary and improve the performance of the sparse recovery algorithm while maintaining accuracy in the acoustic image map. Three dictionary refining methods are described. A first method reduces the dictionary size by progressively refining the dictionary used. A second method reduces the dictionary size by subdividing the dictionary and then solving multiple plane-wave decomposition problems in parallel. A third method combines the previous two methods. The performance and the accuracy of the dictionary refining methods are evaluated with varying number of sources, signal-to-noise ratio, reverberation and algorithm parameters.

• Chapter 8 provides a general discussion and summary of the main results of the thesis. Perspectives for future research are described.
Chapter 2

Literature Review

In this chapter, we present the relevant literature review covering signal processing algorithms for sound fields and implementation of parallel algorithms on FPGAs and multi-threaded platforms.

2.1 FPGA-based Audio Acquisition and Transmission System

There are various kinds of SMA audio acquisition and transmission systems found in the literature. O’Donovan et al. [98] described a 32 microphone spherical array based system implementation for spatial audio capture and reproduction. The array is portable and can be plugged into a USB port on a computer. The array consists of FPGA-based custom hardware (Xilinx Spartan-3) which collects sampled data from two ADC chips in parallel followed by buffer data in FIFO queue and sends over USB to a computer. The computer side acquisition software is based on FrontPanel library provided by Opal Kelly [99]. It streams data and saves it to the hard disk in raw form. Their data sample-width is 12-bit with a sampling rate of 39.0625 kHz.

In paper [164], an FPGA-based microphone array is presented which can be used to capture the acoustic sound scene. It is lightweight, low power and scalable. Fig. 2.2 shows the data-flow block diagram of the system. The system consists of Spartan-3A FPGA which downsample and PCM convert the microphone data. A cascaded 3 FIR filters are used to reduce the sample rate by a factor of 2 in each stage. The FPGA is interfaced with external USB streaming controller via an AC97 interface to transmit the down sampled data to a computer. The FPGA also implements audio buffer, volume controller and required clock signals.

There is an attempt in [102], to measure the spatial and timbral characteristics of a legacy recording microphone and the characteristics of a 120-channel spherical microphone
Figure 2.1: 32 Channel USB2.0 SMA for 3D audio recording and playback [98]. Note how FPGA-based audio acquisition and transmission system is embedded into SMA. System is powered separately and USB interface is used for data acquisition.

Figure 2.2: An FPGA based system to transmit microphone array data to a computer [164].
array in an anechoic chamber (Fig. 2.3(a)). Using a least-square matching approach, the measured frequency responses were used to calculate the set of filters that synthesized the desired legacy recording microphone characteristic from the 120-channel spherical microphone array. By referring the system in [16], the paper stated that all microphone pre-amps and AD-converters are embedded inside SMA, and it could distribute all 120 audio channels over an Ethernet connection in 24 Bit/96 kHz. Further, it stated there is an additional FPGA available for onboard signal processing. The FPGA system which is referred [16], consists of 100BaseT Ethernet interface that supports up to 10 channels of 24-bit audio, 64 channels of sample-synchronous control-rate gesture data, and 4 precisely time-stamped MIDI I/O streams (Fig. 2.3(b)).

The paper [139] presents a multichannel audio processing evaluation platform which can interface 12 analog audio channels and transfer data via Gigabit Ethernet. The system consists of Virtex-6 FPGA and a DSP chip for audio processing. Analog channels are first converted to digital followed by interfacing to the FPGA. FPGA then manipulates digital audio channels and transfer data to DSP chip which handle Gigabit Ethernet for transmission. Data transmission between FPGA and DSP chip is performed via TI external memory interface (EMIF). Fig. 2.4 shows how audio channels, FPGA and DSP are interfaced.

Yonghao et al. [140] propose a Digital Signal Processor (DSP) and FPGA based multichannel audio processing latency evaluation system, which could provide an effective test bed for evaluating the latency problem. The evaluation system supports 12 channels of
In order to reduce the system latency, it would be useful to evaluate and investigate the latency of different stages such as I/O, ADC/DAC [11], buffer setting, scheduler [12] and the DSP algorithms [13] to identify the components contributing most latency, the cause of the latency and the possible solutions. Therefore this paper proposes a DSP and FPGA based multichannel audio processing latency evaluation system, which could provide an effective test bed for evaluating the latency problem. A first version of hardware prototype has been made. The platform is depicted in Figure 1.

In addition, the platform has auxiliary modules, Ethernet ports, and onboard Flash and SDRAM memories. It is mainly designed for multichannel audio signal low latency processing or latency measurement. The detailed functionalities are discussed in Section 3.

![Figure 2.4: Block diagram of a 12 Channel FPGA and DSP based audio acquisition and transmission system](image)

Abdallah et al. [4] proposed an FPGA-based data acquisition and processing system which has a network control module to transmit the acquired signals to authenticated destinations via the Internet. The scope of the system is not limited to audio and can acquire different types of channels in time division multiplexing. Further, characteristics of amplitude and frequency of the subjected channels can vary in a large range (i.e., amplitude with a range from millivolts to volts and frequency range from hertz to megahertz). The same author had published [3] an FPGA-based system for audio acquisition. This system is also capturing channel data in time division multiplexing where switching time is about 5 \( \mu s \). This is a system-on-chip (SoC), which is capable of preprocessing multi-channel audio data and storing them into a storage device.

The platform presented by Okamoto et al. [120] implemented an HOA recording system using a 121 SMA. The system consists of a 121 SMA, FPGA board and a custom computer. The audio signals are first converted to digital form by ADCs mounted in the SMA. These ADCs oversample the audio at 3.072 MHz frequency and 16-bit samples. The FPGA board is used to resample the 121 microphone channels at 48 kHz in the same sample width. The
The entire HOA encoding process is implemented using a computer. Fig. 2.5 shows the external view of the system and its block diagram.

Figure 2.5: External view of the state-of-the-art custom HOA recording system [120]. Note the size of the processing system.

Regarding multi-channel audio stream handling, Dante Brooklyn II PDK is a sophisticated device [15]. It consists of Spartan-6 FPGA as the main controller. It can interface 64×64 audio channels and transfer the audio via Ethernet. On the FPGA, Ethernet protocol is implemented as an application running on Linux kernel. The platform can be connected to computer/network-switch via Ethernet. The audio can be controlled using a proprietary software which communicates with the FPGA. Paper [114] describes an implementation of Ethernet-based synchronous audio playback system using such Audinate platforms. It shows scalability of an FPGA-based system for multi-channel audio stream controlling and transmission.
Figure 2.6: Dante Brooklyn II PDK.
2.2 FPGA-based Beamforming System

In the previous section we described recent FPGA-based implementations to interface microphones, control audio channels and transmit audio data. In this section, we describe recent FPGA-based implementations which can perform microphone array based beamforming and sound-source localization. Many beamforming and sound-source localization techniques which are suitable to implement on FPGAs can be found. In the paper [130], the well-known delay-and-sum beamforming is presented. It works by appropriately compensating the delays of the microphone signals followed by combining them using an additive operation. This method reinforce the desired signal while reduce noise by destructive interference among noises from different channels. The delay, attenuation and noise of the received signal by $M$ omni-directional microphones at time $t$ can be expressed:

$$x_m(t) = a_m s(t - \tau_m) + v_m(t),$$  \hspace{1cm} (2.1)

where $s(t)$ is the original source signal, $\tau_m$ is the delay of the signal received by the $m^{th}$ microphone, $a_m$ is the attenuation of the signal received by the $m^{th}$ microphone and $v_m(t)$ is the noise in the signal received by the $m^{th}$ microphone. In the frequency domain, this can be expressed:

$$X(\omega) = S(\omega)d + V(\omega),$$  \hspace{1cm} (2.2)

where $X(\omega) = [X_1(\omega), X_2(\omega), \cdots, X_M(\omega)]^T$, $V(\omega) = [V_1(\omega), V_2(\omega), \cdots, V_M(\omega)]^T$. The vector $d$ represents the array steering vector which depends on the location of the microphone and the source. In the near-field the receiving wavefront are spherical waves while in the far-field they can be approximated to plane waves. To recover the desired signal by delay-and-sum beamforming, each microphone output is weighted by frequency-domain coefficients $w_m(\omega)$. Therefore, for $M$ microphones, the delay-and-sum beamformer output $Y(\omega)$ is calculated:

$$Y(\omega) = \sum_{m=1}^{M} w_m^* (\omega) X_m(\omega).$$  \hspace{1cm} (2.3)

The delay-and-sum beamforming is basically a convolution in time domain with FIR filter coefficients. In the paper [70], audio-source separation in convolutive mixtures using Independent Component Analysis (ICA) algorithm is presented. Using this method, sources can be separated one at a time by placing nulls to the other sources presents in the mixture. This technique does not give any geometry information like direction of arrival (DOA). In the paper [86], the sources’ DOA is used to identify the permutations along the frequency axis. The sources are permuted along frequency, such that the directivity pattern of each
beamformer is aligned. The directive pattern is defined:

$$F_i(f, \theta) = \sum_{k=1}^{M} W_{ik}^{\text{phase}}(f)e^{j2\pi f d_k \sin(\theta_i)/c},$$  \hspace{1cm} (2.4)$$

where $W_{ik}^{\text{phase}}(f) = \frac{W_{ik}(f)}{|W_{ik}(f)|}$ is the phase of the unmixing filter coefficient between the $k^{th}$ microphone and the $i^{th}$ source at frequency $f$, $d_k$ is the distance of the $k^{th}$ microphone from the origin, $\theta$ is the DOA of the $i^{th}$ source and $c$ is the velocity of sound.

In the paper [71], MVDR beamforming is presented, which is performed in the time-frequency domain with the normalized frequency $\omega$. The signals of the microphone array having $M$ microphones are expressed:

$$x(\omega) = \sum_{l=1}^{L} g(\omega, \theta_l) S_l(\omega) + x_d(\omega) + x_n(\omega)$$  \hspace{1cm} (2.5)$$

where the vector $x(\omega) = [X_1(\omega), X_2(\omega), \cdots, X_M(\omega)]^T$ contains the $M$ microphone signals in the time-frequency domain. The noise and diffuse vectors are denoted as $x_d(\omega)$ and $x_n(\omega)$ respectively in similar notation. The speech component of the $l^{th}$ source at the reference microphone is denoted as $S_l(\omega)$ while $g(\omega, \theta_l)$ is the array propagation vector from the $m^{th}$ microphone to the reference microphone, for the source located at $\theta_l$ with respect to the linear array. Then the desired source is considered as $l = 1$ without the loss of generality. Then, the MVDR beamforming is performed on the microphone signals:

$$Y(\omega) = w^H(\omega)x(\omega)$$  \hspace{1cm} (2.6)$$

where $w(\omega) = [W_1(\omega), C, \cdots, W_M(\omega)]^H$ denotes MVDR beamforming weights. Note that $H$ denotes the conjugate transpose. MVDR beamformer ensures that the desired sound remains undistorted. The filter weights can be found by solving the following constrained optimization problem:

$$\arg\min_w w^H \Phi_u(\omega) w \quad \text{subject to} \quad w^H g(\omega, \theta_1) = 1,$$  \hspace{1cm} (2.7)$$

where $\Phi_u(\omega) = \Phi_n(\omega) + \Phi_d(\omega)$ denotes the undesired power spectral density (PSD) matrix to be minimized at beamformer output. The noise PSD matrix $\Phi_n(\omega) = E\{x_n(\omega)x_n^H(\omega)\}$ can be estimated when all the sources are silent, while $\Phi_d(\omega) = E\{x_d(\omega)x_d^H(\omega)\}$ expresses the diffuse PSD matrix which can be estimated as in the paper [122]. The resulting MVDR beamformer weights are given by

$$w(\omega) = \frac{\Phi_n^{-1}(\omega)g(\omega, \theta_1)}{g^H(\omega, \theta_1)\Phi_n^{-1}(\omega)g(\omega, \theta_1)},$$  \hspace{1cm} (2.8)$$

where, $w(\omega)$ is calculated for each direction.
In order to efficiently calculate the desired spatial filter coefficients, the direction of arrival (DOAs) for the sources need to be estimated \[71\]. The DOA is calculated using the well-known steered-response power-phase transform (SRP-PHAT) method. In this method, the PHAT weighting removes the magnitude spectrum from the computation of cross-correlations. In order to find the DOAs of \( L \) sources \( \Theta = [\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_L] \), the SRP-PHAT algorithm searches for \( L \) distinct local maxima in the so-called spatial pseudo-spectrum \( P(\tau_{m,m'}(\theta)) \) given by

\[
P(\tau_{m,m'}(\theta)) = \sum_{m=1}^{M} \sum_{m'=m+1}^{M} \mathcal{F}^{-1}\left\{ \frac{\phi_{m,m'}(\omega)}{|\phi_{m,m'}(\omega)|} \right\},
\]

where \( \phi_{m,m'} = E\{X_m(\omega)X_{m'}^*(\omega)\} \) denotes the cross power spatial density between the microphone pair \( (m, m') \). Further, \( E\{\cdot\} \) denotes mathematical expectation, and \( \mathcal{F}^{-1}\{\cdot\} \) denotes an inverse Fourier transform. The function \( \tau_{m,m'}(\theta) \) relates the source location to the relative delay between the pair of microphones \( (m, m') \) and is computed using the geometry of the microphone array.

The computational complexity of microphone-array based beamforming and sound-source localization increases with the number of microphones and many other factors depend on the methodology. Therefore, to achieve real-time performance, parallel processing is widely applied. In the paper \[162\] an FPGA-based system interfaced with helmet-mounted microphone array is discussed. Fig. 2.7 shows the configuration of the microphone array and the apparatus. On the FPGA, the microphone array signals are filtered based on the frequency and applied SRP-PHAT (steered response power and phase transform) algorithm \[69\] to calculate the trajectory of the bulletin reverberant condition. A similar system is implemented using an FPGA-based wireless microphone array \[110\]. The FPGA
is interfaced with the helmet-mounted microphone array which identified properties of the shockwaves to identify the direction of arrival of the bullet. Fig. 2.8 shows the FPGA-based wireless microphone array. The platform consists of Xilinx XC3S1000 FPGA with various standard peripherals. The FPGA calculates the shooter position by approximating the sound scene to far-field. Therefore, the receiving wave-fronts can be approximated to plane waves. The wireless network of spatially-distributed many such small microphone arrays makes large-aperture acoustic microphone array which improves the range of the system. Another system in paper [61] presents a hat-type hearing system using microphone array. Fig. 2.9 presents the experiment setup of the system. There are 48 microphones around the hat, which are interfaced with Altera-EP4CE15F17C8N FPGA. The output signals are calculated by the delay-and-sum beamforming technique performed on the FPGA to emphasize up to 10 dB of the sound coming from a chosen direction. The system can be operated on 9.0-volt batteries and its weight is about 500 g.

In the paper [111] presents a noise source localization system which consists of a microphone array and an FPGA platform. The source localization is performed by using conventional delay-and-sum beamforming algorithm. The microphone array consists of 33 microphones. It is interfaced to an FPGA which performs the beamforming in real-time. The paper [123] presents a similar system called SoundCompass for noise source localization. Fig. 2.10 shows the hardware blocks of SoundCompass. The microphone array consists of
52 microphones and interfaced to the FPGA via a bus of pulse density modulated (PDM) signals. The FPGA is also connected to a host platform via an I2C interface. In here also the source localization is performed on the FPGA using delay-and-sum beamforming algorithm.

In the paper [125][126] presents a system of 52 microphone array and an FPGA embedded with it. Fig. 2.11 shows the top and bottom view of the platform. The source localization is performed using Independent Component Analysis (ICA). The system is capable to beam steering on the horizontal and vertical planes in real-time. The FPGA used in the system is Virtex-4 XC4VFX12. The FPGA platform consist of Gigabit Ethernet and 32MB EPROM. Using the EPROM, audio data are stored and processed on the platform. The platform can be connected to a router or computer using Ethernet. Therefore, by using a router, multiple such microphone array boards can be interconnected [127]. A similar scalable system is presented in the paper [64] for microphone-array data acquisition and processing. Fig. 2.12 shows the typical system. It can be used to generate acoustic images. The platform consists of Xilinx Zynq 7010 FPGA which contains a dual-core ARM Cortex-A9 processor. Audio pre-processing stages of deinterlacing, decimation and filtering are performed on the FPGA while wideband beamforming is performed on ARM Cortex-A9 processor as frequency-domain phase-shift-and-sum beamforming. The spherical microphone array consists of 64 microphones which are interfaced to the FPGA via 32 ports by multiplexing 2 microphones in each port. The Zynq platform consists of 256 MB of DDR3 RAM, 512 MB of built-in storage space, USB Host port, and Wi-Fi interface. The memories are used to buffer acoustic images and Wi-Fi interface is used to control the platform using a computer.
Figure 2.10: The hardware blocks of SoundCompass [123].
In the paper [20], a 20 MHz 64-channel real-time beamforming system for antenna array is presented. In the acquisition system, the channels are sampled at 50 MHz and bandpass filtered using 64 ADCs. The sample width is 12-bit. These samples are then transferred to an FPGA-based platform which is shown in blue in Figure 3. The beamforming is performed in the frequency domain using a 64-channel 512-point FFT module and filters on the FPGA. The FPGA is interfaced with a processor-based system which loads beamformer weights into FPGA local block RAMs (BRAMs). The system is also capable of recording the FFT output to a disk via 10-GbE link.

In the paper [83], dual-port block RAM and dynamic memory based delay-sum beamformer are explained. In the implementation, the dual port block memory is used for buffer the receiving data while the beamforming coefficients are stored in a dynamic DDR memory, which is accessed through a dedicated controller. The system is implemented on an ULA-OP 256 front-end board which contains Altera Arria-V FPGA. To implement 32 channels, the system utilizes 8% of DSP48 blocks, 8% of logic registers and 35% of memory blocks. The system maximum operating frequency is 234.375 MHz.

In the paper [10], a 1024-channel 3D ultrasound digital delay and sum beamformer are presented. The implementation computes $32 \times 32$-channel on a Kintex KU040 FPGA. The critical resource for the implementation is BRAM which is utilized 71%. The BRAM are used mostly for buffers and delay coefficients of the beamformer. On the other hand, 30% of the FPGA lookup tables and 15% of the flip-flops are used. In the paper, calculations predict $90 \times 90$ channels can be processed on a large FPGA like Virtex XCVU190. This implementation is motivated by low power requirement where the computations are performed
Figure 2.12: 64 microphone array which is interfaced with Xilinx Zynq 7010 FPGA based platform [64].
Figure 2.13: The FPGA platform which performs real-time beamforming using 20 MHz 64-channel. The blue board is the FPGA platform, and the green board is the channel data acquisition system which consists of ADCs [20].

Figure 2.14: The multiline delay-sum beamformer which consists of DDR memory for store beamforming coefficients [83].
within 5-Watt power budget.

In the paper [24], passive SONAR beamforming and Low-Frequency Analysis and Recording (LOFAR) techniques are presented. The beamforming is performed using the conventional delay and sum method while LOFAR is implemented using conventional FFT which was configured as 1024 points. The systems were implemented on a Nexys-4 FPGA board having Artix-7 Xilinx FPGA. The implementation utilized only 22% of the FPGA resources except for BRAM which was used 90.37% where most of them were used for data memories.

In the paper [77], a fixed-point architecture for Frosts adaptive beamforming is presented. The architecture is designed such a way that optimizes resource consumption by utilizing lookup tables. The presented architecture can be customized based on the number of sensors, input bit-width, data-path width, bit-width of Frosts parameters, and the desired beam pattern. The adaptive beamformer can be dynamically updated when new beamforming pattern is required. Figure 1 shows the system implementation on Zynq-7000 SoC. A software executes on ARM processor while transmitting data in and out to Frost beamformer which is implemented as a coprocessor in the programmable logic (PL) side. The system utilizes 46% of DSP48 blocks, 32% flip-flops (FFs) and 61% of 6-input lookup tables (LUTs).

Figure 2.15: The hardware-software codesign implemented on Zynq-7000 SoC for beamforming [77]. A software is executed on ARM processor while transmitting data in and out to Frost beamformer which is implemented as a coprocessor in the programmable logic (PL) side.

In the paper [71], a source localization using steered response power phase-transform
(SRP-PHAT) and speech enhancement using the MVDR beamformer is presented. The system could effectively suspend noise and interferences in real-time. The block diagram of the architecture is given in Figure 2. The sound field is recorded using a linear 7-microphone array. The signal processing system is implemented on Zynq 7020 system on chip (SoC). The captured data from the microphone array are parallel-to-serial (P2S) converted followed by decimation (Decim.) and compensation (Comp) for pulse code modulation (PCM). Next, the data are transferred to DDR memory and enhanced by a program executed on the on-chip ARM microprocessor.

![Figure 2.16: The source localization and speech enhancement system [71]. The signal processing system is implemented on Zynq 7020 system on chip (SoC). The data buffers are implemented using direct memory access (DMA) IP and onboard DDR memory.](image)

In the paper [113], FPGA based novel approach to perform delay-sum beamforming is presented. To convert RF channel data into beamformed images in 3D systems, over 100 billion round-trip delay calculations are required. This is challenging, and the conventional 2D beamforming approach which computes delays and storing them in storage (LUTs or BRAMs) is infeasible due to the requirement of many Giga-byte storages. Alternatively, in the paper, the delay values are computed on-the-fly using iteratively calculated quadratic formula with add operations and a small table of pre-computed coefficients. Relative to a fully pre-computed method, this method significantly reduces the utilization of LUTs or BRAMs. The system is tested in real-time for single RF channel using Cyclone-2 FPGA.

In the paper [138], an FPGA-based MVDR (minimum variance directionless response) beamformer design for ultrasound imaging is presented. By using auto-correlation matrix approximation and Schur matrix decomposition scheme, the computational complexity of the associated matrix inversion process is reduced by an order. The system is implemented
on a Xilinx Vertex-7 FPGA. The maximum operating frequency of the system is 98.133 MHz. The design utilizes 26% of the logic cells and 72% of the DSP48E1 blocks of the FPGA.

In the paper [7], development of a 128-channel FPGA-based delay-and-sum beamformer for synthetic aperture (SA) imaging is presented. A well-known OpenCL image processing library is used to code the beamformer followed by compiled into register transfer level (RTL) using high-level synthesis (HLS) technique. The OpenCL SIMD kernel that we developed for SA beamforming is imported into the Altera software-development kit for OpenCL to perform high-level synthesis. Using the HLS tool, it was possible to configure SIMD vectorization width, for-loop unrolling and number of computing units. The system is implemented on Altera Stratix-V D5 FPGA which is mounted on PCIe-385N platform. The system which supports highest frame-processing throughput utilized 86.9% of the FPGA fabric and operated at a 196.5 MHz clock frequency. As per the development experience, the overall development effort was accelerated significantly with the HLS tool.

2.3 Solving Linear-Algebra Problems on Multithreaded Platforms

The most computationally intense algorithm in our method of source localization is sparse recovery algorithm. We perform the sparse recovery in frequency domain due to many advantages. Consequently, sparse recovery is required to be performed on each frequency. The sparse recovery algorithm is based on iteratively reweighted least squares (IRLS) algorithm. Therefore, the entire sparse recovery process is computationally intensive and require parallelism to accelerate the computation. In general, FPGAs and multithreaded platforms are widely used in high-performance parallel processing. In the paper [19], a performance comparison of FPGAs and GPUs was presented. Evidently, GPUs often perform faster than FPGAs for streaming applications, while enjoying a higher floating-point performance and memory bandwidth than FPGA-based systems. Even though FPGAs have advantages in sparse or mixed precision algorithms [26][129], our algorithm is fixed precision (single or double) and a dense linear-algebra problem. Therefore, the sparse recovery algorithm is implemented on multithreaded platforms.

In GPUs, a thread is the basic instruction execution process. There are many resources (i.e., ALUs, FPUs, registers, shared memory, etc.) on the GPU to execute many threads simultaneously. In CUDA framework, a kernel is a multithreaded program which is subjected to execute on the GPU [95]. At the time of launching a kernel, it is required to create block of threads as required by the algorithm and execute them as a grid of thread blocks. The hierarchy of thread, block and grid is illustrated in Fig. 2.17. The kernel is
executed using thread blocks which are assigned to the streaming multiprocessors (SMs). These thread blocks are arranged as a grid in CUDA. Different thread blocks of the grid can be assigned to the same or different SMs. Any thread in a grid can be uniquely indexed as:

\[
\text{int } i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}
\]  

(2.10)

where, \text{blockDim} is the dimension of the thread block. The \text{blockIdx} is indexed related to grid while \text{threadIdx} is indexed related to block.

![CUDA grid of thread blocks and block of threads.](image)

Figure 2.17: CUDA grid of thread blocks and block of threads. The kernel is executed using thread blocks which are assigned to the streaming multiprocessors (SMs). These blocks of threads are called a grid in CUDA. Different thread blocks of the grid can be assigned to the same or different SMs. Any thread in a grid can be uniquely indexed when using the CUDA computing model.

Each thread block requires certain amount of resources in a SM. The thread blocks which perform computations at a given time in a SM are called active thread blocks \( B_a \). The performance of a CUDA program executed on a GPU depends on the number of active-thread blocks (i.e., \( B_a \)) on the SMs and the CUDA kernel thread occupancy. The number
of active thread blocks per SM \(B_a\) can be calculated by Eq. 2.11 \[80\] s.t.,

\[
B_a = \min \left( \left\lfloor \frac{S}{S_B} \right\rfloor, \left\lfloor \frac{R}{R_T \cdot T_r} \right\rfloor, \left\lfloor \frac{B_r}{N_p} \right\rfloor, \left\lfloor \frac{T_{\text{max}P}}{T_r} \right\rfloor \right),
\]

(2.11)

where

- \(S\)  
  Shared memory per SM (in Bytes),

- \(S_B\)  
  Shared memory used per block (in Bytes),

- \(R\)  
  Number of registers per SM,

- \(R_T\)  
  Number of registers per thread,

- \(T_r\)  
  Requested number of threads per block,

- \(N_p\)  
  Number of SMs in the device,

- \(B_r\)  
  Requested number of blocks (= Number of IRLS problems),

- \(T_{\text{max}P}\)  
  Maximum number of threads per SM.

The parameters \(S\), \(R\), \(N_p\) and \(T_{\text{max}P}\) are specific to the GPU which can be found in the GPU user manual. The parameters \(S_B\) and \(R_T\) depend on the design of the kernel, which need to be either calculated by analyzing the kernel or evaluated by a program profiler such as NVIDIA Visual Profiler. The parameters \(T_r\) and \(B_r\) are specific to the kernel which need to be determined at the launch of the kernel on the GPU.

The GPU occupancy of a kernel is a measure of its effectiveness in utilizing the resources on the GPU to hide the memory access latency. Therefore, when increasing the number of active thread blocks on the GPU, it should ensure that the GPU occupancy is high. Otherwise, the increase of the number of active blocks will not be effective. The occupancy of the GPU can be calculated by Eq. 6.35 \[1\] s.t.,

\[
\text{GPU Occupancy} = \left\lfloor \frac{T_r \cdot B_a}{T_{\text{max}W}} \right\rfloor \frac{T_{\text{max}W}}{W_{\text{max}P}},
\]

(2.12)

where

- \(T_r\)  
  Requested number of threads per block,

- \(B_a\)  
  Active thread blocks per SM,

- \(T_{\text{max}W}\)  
  Maximum threads per warp,

- \(W_{\text{max}P}\)  
  Maximum warps per SM.

The parameter \(T_r\) is specific to the kernel which needs to be determined at the launch of the kernel on the GPU. The parameter \(B_a\) should be calculated by Eq. 2.11. The parameters \(T_{\text{max}W}\) and \(W_{\text{max}P}\) are specific to the GPU which can be found in the GPU user manual. Table 2.1 presents the specifications of Nvidia K40 GPU which are required to calculate the number of active-thread blocks and the GPU occupancy.

25
Table 2.1: The specifications of Nvidia K40 GPU which are required to calculate the number of active-thread blocks and the GPU occupancy.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Nvidia K40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of SMs on the GPU ($N_p$)</td>
<td>15</td>
</tr>
<tr>
<td>Maximum shared memory per SM ($S$)</td>
<td>48 KB</td>
</tr>
<tr>
<td>Maximum registers per SM ($R$)</td>
<td>65536</td>
</tr>
<tr>
<td>Maximum threads per SM ($T_{\text{max}P}$)</td>
<td>2048</td>
</tr>
<tr>
<td>Maximum threads per warp ($T_{\text{max}W}$)</td>
<td>32</td>
</tr>
<tr>
<td>Maximum warps per SM ($W_{\text{max}P}$)</td>
<td>64</td>
</tr>
</tbody>
</table>

When analysing the algorithmic performance on multithreaded architectures, understanding of the cache performance is important. According to TMM model [79], the performance of algorithms depend on the number of threads, when the number of threads is small. The performance converges to PRAM performance [44] with sufficient number of threads, which only depends on the problem size and the number of processors. This happens due to reduction of the cache efficiency when increasing the number of threads. The cache performance of a program is related to cache hit/miss rate which is a function of size of the cache and data localities of the algorithm. Smith [116] presented a 30% rule based on their observation of cache performance which stated that every doubling of cache size $x$ should reduce the cache miss rate $f(x)$ by 30%. This recurrence relation can be formulated as

$$0.7f(x) = f(2x).$$  \hspace{1cm} (2.13)

This relationship was generalized as one-term polynomial function s.t.,

$$f(x) = \beta x^\alpha,$$  \hspace{1cm} (2.14)

where $\alpha$ and $\beta$ are cache miss rate function constants which depend on the temporal locality of the application data [65]. Note that $\alpha$ is negative, $\beta$ is positive and $f(x) \in (0, 1)$. Regarding the shared cache in a particular parallel computing architecture, the cache allocated to a problem can be defined s.t.,

$$C_s = \frac{C}{N_{\text{prob}}},$$  \hspace{1cm} (2.15)

where $C$ is size of the shared cache and $N_{\text{prob}}$ is the number of problems. Then the cache miss rate of the program can be expressed by Eq. 6.30 s.t.,

$$f(C_s) = \beta \left( \frac{C}{N_{\text{prob}}} \right)^\alpha.$$  \hspace{1cm} (2.16)

Eq. 6.32 describes the cache miss rate when varying the number of problems. For clarity, Eq. 6.32 can be reformulated s.t.,

$$f(C_s) = \beta \cdot C^\alpha \cdot N_{\text{prob}}^{-\alpha}.$$  \hspace{1cm} (2.17)
where $\alpha$ is positive. For given algorithm and architecture $\alpha$, $\beta$ and $C$ are constant. Since the cache miss rate increases when increasing the number of problems, it decreases the rate of increasing the performance. Eventually, the performance is converged to PRAM performance where the cache performance is very low.

Now we discuss some of the iterative algorithms similar to IRLS algorithm, which are implemented on multithreaded platforms. With the increase of popularity of compressed sensing algorithms, a number of accelerated $l_1$-minimization algorithms have been proposed by explicitly take advantage of the special structure of the $l_1$-minimization problems. Therefore, these algorithms are similar to sparse recovery by the implementation. In the paper [115], $l_1$-minimization for face recognition is performed by using augmented Lagrangian method (ALM) which solves Lagrange multipliers in an iterative fashion. The algorithm is implemented on a GPU by mapping most of the operations to cuBLAS library [97]. The programs were written in CUDA. To minimize the CPU-GPU communication latency, data were initially transferred to GPU DRAM. The results show the performance on GPU is twice the performance of CPU. The paper [47] presents a similar implementation where $l_1$-minimization problem is solved by merging fast iterative shrinkage-thresholding (FISTA) algorithm and augmented Lagrangian multiplier method (ALM). The new GPU kernel was implemented and tested on GTX980 GPU. The paper states that the merged method is more robust than cuBLAS and always have higher performance. In the paper [117], Bregman iterative algorithm for $l_1$-minimization is implemented on a GPU for MRI image reconstruction. The MRI image reconstruction is also a compressed sensing problem which is solved iteratively as a sparse recovery problem. The results state that GPU acceleration could gain 27 times acceleration compared to single thread CPU implementation. A Nvidia Tesla C2050 GPU and a six-core Xeon X5650 CPU is used in this analysis. In this implementation also, the initial data are stored on the GPU DRAM to minimise the data transfer latencies. In compressed sensing, Orthogonal Matching Pursuit (OMP) is a sparse signal recovery algorithm which achieves good performance with low complexity. The algorithm recovers the sparse solution using iteratively reweighted least square (IRLS) algorithm. The bottlenecks of the OMP are matrix inverse and matrix-vector multiplication. In the paper [43], GPU implementation of OMP by adopting 2 algorithms was discussed. Fujimotos matrix-vector multiplication algorithm [45] is adopted to speed up the matrix-vector operations, while the matrix-inverse-update algorithm [57] is adopted to speed up the least-squares module. As per the evaluation of the implementation, over 40 times speedup is achieved by GTX480 GPU over Intel Core-i7 CPU. Another compressed sensing MRI reconstruction method is presented in the paper [105], which is based on convolutional sparse coding and a temporal
Total Variation (TV) regularization. In this method, the sparse solution is iteratively reconstructed using the sparse codes found during the reconstruction process. The algorithm is accelerated by a GPU implementation. Based on the evaluation, Nvidia GTX Geforce 980 Ti GPU is seven to nine times faster than Intel Core-i7 CPU.

The performance of the sparse recovery algorithm implemented on the multithreaded platform depends on how efficiently the underlying linear algebra computations are performed. The proposed sparse recovery based source-localization algorithm (i.e., iteratively reweighted least square - IRLS) mainly consist of matrix-matrix multiplication, matrix-vector multiplication, and a triangular solver. A good description about the linear computation of general least square (GLS) method is given in the paper \[40\]. It provides what BLAS and LAPACK routines are required to perform the least squares solver. Based on the literature, it is possible to replace the BLAS/LAPACK routines by adopting better algorithms. Now we discuss some of the new techniques to perform those underlying linear algebra computations.

In the paper \[118\], a scalable method of solving a dense linear problem is presented. In this method, the performance is improved by overlapping the computations and communications through dynamic scheduling of instructions. As a result, it was possible to achieve scalable performance for the double precision Cholesky factorization and QR factorization. This performance is comparable to Intel MKL on shared-memory multicore systems and better than GPU platforms running with Intel MKL or open source libraries. This paper also elaborates, to attain high performance, it is required to (1) minimize communication, (2) maximize the degree of task parallelism, (3) accommodate the processor heterogeneity, (4) overlap communication, and (5) keep load balance.

In the paper \[17\], solving of dense symmetric indefinite systems on hybrid CPU and GPU is presented. The performance bottleneck of this type of algorithm is the requirement of frequent synchronizations for symmetric pivoting to maintain the numerical stability of the factorization. This process has irregular memory accesses, which is inefficient on a GPUs. In this implementation, the matrix is factorized in single precision without pivoting. This algorithm only has a probabilistic proof of the numerical stability. However, it complements with GPU systems. Furthermore, the paper discussed techniques to minimise data transfers between CPU and GPU which hinders the performance. As per results, the algorithm without pivoting outperformed general LU factorization and achieved twice speed up.

In the paper \[1\], a batched Gauss-Jordan elimination CUDA kernel for matrix inversion is presented. In this method, an implicit pivoting technique is introduced while performing the entire inverse process on the GPU registers. This kind of inversion technique is very effective when there are many small matrices to inverse where a matrix can be fitted into
GPU shared memory or registers. The results show that the presented batched Gauss-Jordan elimination outperforms the standard LU-based approach by more than an order of magnitude. However, recent paper [33] shows that LU decomposition also can be improved to solve many small size linear problems. In the implementation, batched LU achieves up to 2.5-fold speedup when compared to the counterpart CUBLAS solution on a K40c GPU. This speedup is mainly achieved by improving the data layout and pivoting techniques.

In the paper [72], a GPU implementation of solving a large number of small symmetric positive definite systems of linear equations is described. Because of symmetric positive definite nature, Cholesky factorization followed by the forward and backward substitution provide the best performance. Our sparse recovery algorithm has the same characteristics, where the linear systems are small and symmetric positive definite. The implementation presented in the paper can perform thousands of linear systems of the same size where the matrix-dimension can vary from 5 to 100. Since the counterpart cuBLAS solves the linear systems using LU or QR decomposition, the new method is efficient when solving symmetric positive definite systems. The results show that Cholesky factorization method exceeds 120 Gflop/s on state-of-the-art GPUs.

Based on the above literature related to the implementation of $l_1$-minimization, compressed sensing and efficient linear-algebra solvers, we can identify following key facts:

1. Iterative algorithms can be accelerated on multithreaded platforms,

2. BLAS library [21] can be effectively used to implement sparse recovery operations on multithreaded platforms,

3. To overcome the computational bottlenecks in sparse recovery algorithm (i.e., matrix-vector operations, matrix-matrix operations and matrix inverse operation), different algorithms and techniques can be adopted.
Chapter 3

Background

3.1 Introduction

In this chapter first, we describe the background of spherical microphone arrays (SMA) and spherical Fourier transformation of SMA data. Then sparse-recovery technique is explained to perform plane-wave decomposition of SMA data using spherical Fourier transformation. We propose to implement spherical Fourier transformation using a FPGA and plane-wave decomposition using multi-core/many-core architecture. Therefore, the background of FP-GAs and multi-core/many-core architectures is present for clarity.

In spherical Fourier transformation of SMA data, implementation of fast-Fourier transformation (FFT) on a FPGA is highly parameterizable. Therefore, implementation concepts of FFT module on a FPGA are discussed in detail. Furthermore, the introduction of FPGA resources such as configurable-logic blocks, DSPs, memories, etc. is given. In the analysis of multi-core/many-core architectures, several memory access models are discussed. These models are helpful to understand the performance of sparse recovery based plane-wave decomposition on multi-core/many-core architectures.

3.2 Spherical Microphone Arrays

Spherical microphone arrays (SMAs) have been the focus of considerable recent research \[5, 15, 25, 36, 53, 60, 68, 84, 85, 106, 108, 119, 133, 141, 160\] and are especially useful for recording panoramic sound scenes. SMAs provide a natural framework for analyzing sound fields in the spherical harmonic domain because of their spherical symmetry. Over the last decade, spherical harmonic audio signal processing has been used in various applications, including sound field reproduction \[18, 133, 141\], beamforming \[25, 107, 160\], source localization and separation \[36, 119\] and room acoustics analysis \[53, 60, 84\].

The performance of a particular SMA is dictated by its physical characteristics \[68, 106, 108\], which include its dimensions, the number and positions of the microphones, the
presence of a baffle and the quality of the sensors employed. An image of SMA used in our work is shown in Figure 3.1 [68]. The rigid inner array consists of 32 omnidirectional microphones distributed over the surface of a 28 mm-radius hard sphere. The outer array consists of 32 omnidirectional microphones located on the surface of an open sphere of radius 95.2 mm. Both the rigid sphere and the structure supporting the outer array microphones were constructed from nylon using a laser selective sintering technique. The wires running to the microphones were run along the frame supporting the microphones and inside the supporting cylinder.

3.3 Spherical Fourier Transformation of the Audio Signals

Transforming audio signals into the spherical-harmonic domain is called spherical Fourier transform (SFT). We derive the mathematical model for the behavior of an SMA and use it to develop a framework for SFT. Let’s consider the case of an SMA consisting of \( N \) omnidirectional microphones located at various positions around a perfectly rigid sphere with radius \( R \). As illustrated in Figure 3.2, we define the position of the microphones by their spherical coordinates \((r, \theta, \phi)\). For simplicity, the mathematical expressions are derived
Figure 3.2: The notations used to describe the geometry of an SMA are illustrated.

in the frequency domain as a function of the dimensionless frequency $kR$, where $k$ denotes the wavenumber, $k = \frac{2\pi f}{c}$, $f$ denotes the frequency and $c$ denotes the speed of sound. As well, for a given radial distance, $r$, we introduce the dimensionless radius, $\rho$, defined by: $\rho = r/R$.

Consider the $n$-th microphone of the SMA, whose spherical coordinates are $(\rho_n R, \theta_n, \phi_n)$. In the case where the incident sound field consists of incoming waves, the acoustic pressure measured by this sensor is given by:

$$p_n = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} w_l(kR, \rho_n) Y_l^m(\theta_n, \phi_n) h_{l,m},$$  \hspace{1cm} (3.1)

where

- $h_{l,m}$ is a complex coefficient depending only on the incident sound field, which we denote as the order-$l$ and degree-$m$ spherical harmonic component.

- $Y_l^m$ denotes the order-$l$ and degree-$m$ real-valued spherical harmonic function:

$$Y_l^m(\theta, \phi) = \sqrt{\frac{2l+1}{4\pi}} \frac{(l-m)!}{(l+m)!} P_l^m(\sin \theta) \cdots$$

$$\times \begin{cases} 
\cos m\phi & \text{for } m \geq 0 \\
\sin |m| \phi & \text{for } m < 0 
\end{cases},$$  \hspace{1cm} (3.2)

where $P_l^m$ is the order-$l$, degree-$m$ associated Legendre polynomial. Note that the $\sin \theta$ term arises from the spherical coordinate convention chosen in this paper (see Figure 3.2).
- $w_l(kR, \rho_n)$ is the ‘modal strength’ of the order-$l$ spherical harmonic modes at the microphone position and is given by:

$$w_l(kR, \rho_n) = i^l \left( j_l(kR \rho_n) - \frac{j_l'(kR)}{\zeta_l^{(2)}(kR)} \zeta_l^{(2)}(kR \rho_n) \right), \quad (3.3)$$

where $j_l$ and $\zeta_l^{(2)}$ denote the order-$l$ spherical Bessel function and spherical Hankel function of the second kind, respectively.

We refer to Equation (3.1) as a Bessel-weighted spherical harmonic expansion of the acoustic pressure. In the audio engineering literature, this equation is sometimes referred to as a spherical Fourier transform (SFT).

According to Equation (3.1), the exact value of the pressure is determined by the summation of an infinite number of terms. This sum must be truncated for the pressure to be estimated numerically. Eq. (3.4) expresses the summation over a finite number of terms.

$$p_n \approx \sum_{l=0}^{L} \sum_{m=-l}^{l} w_l(kR, \rho_n) Y_l^m(\theta_n, \phi_n) h_{l,m}. \quad (3.4)$$

It can therefore be rewritten as the following vector product:

$$p_n = t^\top \Lambda, n h, \quad (3.5)$$

where

$$t_{\Lambda, n} = [t_{0,0,n}, t_{1,-1,n}, t_{1,0,n}, ..., t_{\Lambda,\Lambda,n}]^\top,$n$$

$$t_{l,m,n} = w_l(kR, \rho_n) Y_l^m(\theta_n, \phi_n),$$

$$h = [h_{0,0}, h_{1,-1}, h_{1,0}, ..., h_{\Lambda,\Lambda}]^\top. \quad (3.6)$$

Similarly, the vector of the acoustic pressures received by the $N$ microphones of the SMA can be expressed as:

$$p = T_\Lambda h, \quad (3.7)$$

where $T_\Lambda$ is the transfer matrix between the SFT components up to order-$\Lambda$ and the pressure received by the $N$ microphones, given by:

$$T_\Lambda = [t_{\Lambda,1}, t_{\Lambda,2}, ..., t_{\Lambda,N}]^\top. \quad (3.8)$$

We refer to the process of retrieving the up-to-order-$\Lambda$ SFT components from the microphone signals as order-$\Lambda$ SFT. SFT has a strong interest as it enables to configure the playback system independently from the microphone array used to capture the spatial sound field.
3.4 Plane-wave Decomposition

In the previous section, we presented how to calculate SFT signals from a SMA sound scene. In this section we discuss the calculation of plane-wave decomposition signals based on the SFT signals. Assume that we observe the sound field as a set of $K$ spherical harmonic expansion signals. In the time-frequency domain, these observation signals corresponding to a given time window $t$ and frequency bin $f$ can be expressed as a complex vector, $h(t, f)$:

$$h(t, f) = [h_1(t, f), h_2(t, f), ..., h_K(t, f)]^T.$$ (3.9)

where, different observations are indexed from 1 to $K$. The value of the $K$ depends only on the highest order of the spherical Fourier transformation (i.e., $\Lambda$) and $K = (\Lambda + 1)^2$.

Assuming all sound sources are sufficiently far from the microphone array, we can model the sound field as a sum of $N$ plane waves incoming from many directions in space ($N \gg K$). In other words, we assume that there exists a set of plane-wave signals, $x(t, f)$, satisfying:

$$Dx = h,$$ (3.10)

where $x$ is defined similarly to $h$:

$$x(t, f) = [x_1(t, f), x_2(t, f), ..., x_N(t, f)]^T.$$ (3.11)

Note the set of plane waves are indexed from 1 to $N$ in a given time window $t$ and frequency bin $f$. $D$ is a $K \times N$ matrix expressing the contribution of the different plane waves to the observation signals. We refer to $D$ as a dictionary because we are expressing the observation signals as a sum of plane-wave contributions. The dictionary can be complex or real. In this thesis we are referred to real dictionaries. In summary, the plane-wave decomposition problem consists in solving Equation (3.10) to find plane-wave signals $x(t, f)$ for given observation signals $h(t, f)$.

3.4.1 Sparse Plane-wave Decomposition

Since number of columns in the dictionary is larger than the number of rows (i.e., $N \gg K$), there is an infinite number of solutions to Equation (3.10). The classic way to solve this problem is to choose the solution with the least energy which is known as the least-norm solution. Analytically the least-norm solution is given by:

$$\bar{x} = D^T (DD^T)^{-1} h,$$ (3.12)

where $h \in \mathbb{Z}^{(\Lambda+1)^2 \times 1}$, $x \in \mathbb{Z}^{N \times 1}$ and $D \in \mathbb{R}^{(\Lambda+1)^2 \times N}$. The matrix $D^T (DD^T)^{-1}$ is referred as the Moore-Penrose pseudo-inverse of $D$. The issue with the least-norm solution is that it
tends to distribute the energy evenly across plane-wave directions. This leads to spatially blurry energy map, which is generally undesirable.

An alternative to the least-norm solution is the *sparsest* solution, that is, the solution that employs the smallest possible number of dictionary columns. Mathematically, this solution can be defined as:

\[
\begin{align*}
\text{minimize} & \quad \|x\|_0 \\
\text{subject to} & \quad Dx = h,
\end{align*}
\]

where \(\|\cdot\|_0\) denotes the \(\ell_0\) norm of vector \(x\), that is, the number of non-zero coefficients in \(x\).

Two reasons make this solution interesting. First, this solution is spatially sparse, therefore, unlike the least-norm solution, it is sharp. Second, it is likely that there are only a limited number of dominant sources at a given time, hence generally this solution makes more sense than the least-norm solution.

### 3.4.2 The Iteratively-reweighted Least-square Algorithm

In practice it is extremely difficult to solve Equation (3.13) for the \(\ell_0\)-norm solution. Instead, one may solve the problem for the solution with the least \(\ell_p\) norm, where \(0 < p \leq 1\), which also promotes sparsity across directions. This can be done using the iteratively-reweighted least-square algorithm (IRLS) [30], also referred to as the focal underdetermined system solver (FOCUSS) [28]. The basic idea of the IRLS algorithm is that the \(\ell_p\) norm of the solution can be expressed as a weighted \(\ell_2\) norm (the Frobenius norm). We have:

\[
\|x\|_p = \left( \frac{1}{N} \sum_{n=1}^{N} |x_n|^p \right)^{\frac{1}{p}} \quad \forall \ x_n \in x,
\]

\[
= \|W^{-\frac{1}{2}}x\|_2^2,
\]

where \(W\) is the diagonal matrix given by:

\[
W = \text{diag}(w_1, w_2, \ldots, w_N)
\]

\[
w_n = \left( x_n^2 \right)^{\frac{p-1}{p}} \quad \forall \ x_n \in x.
\]

Thus, finding the solution with the least \(\ell_p\) is equivalent to solving the weighted least-norm problem:

\[
\begin{align*}
\text{minimize} & \quad \|W^{-\frac{1}{2}}x\|_2^2 \\
\text{subject to} & \quad Dx = h.
\end{align*}
\]
Given a fixed $W$, this problem has a closed-form solution, $x_W$, given by:

$$x_W = W D^T \left( D W D^T \right)^{-1} h .$$

(3.16)

Note that this result can be easily demonstrated using the method of the Lagrange multipliers. If the matrix $D W D^T$ is ill-conditioned, it is not invertible. Then the matrix can be conditioned by regularizing the $D W D^T$ such that:

$$x_W = W D^T \left( D W D^T + \lambda I \right)^{-1} h ,$$

(3.17)

where $\lambda$ is a regularization parameter. The regularization parameter $\lambda$ is calculated such that:

$$\lambda = \frac{\beta}{1 - \beta} \left( \frac{\text{tr}(D W D^T)}{N} \right),$$

(3.18)

where, $\lambda$ represents the power of the noise signals $N$ which are incorporated with the plane-wave signals. The relationship of the observations, plane-wave signals and the noise signals can be formulated such that:

$$H = DX + N .$$

(3.19)

Note that $\lambda$ is updated in each iteration with $W$ as the power of the noise signals varies relative to the plane-wave signals at each iteration. The term $\beta$ is the relative energy of noise signals. Therefore, the term $\frac{\beta}{1 - \beta}$ represents the total power of the noise signals relative to the total power of the observation signals in the absence of noise. The term $\frac{\text{tr}(D W D^T)}{N}$ represents the average power of the observation signals in the absence of noise.

**Algorithm 1** The IRLS algorithm for sparse plane-wave decomposition.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \in \mathbb{R}^{(\Lambda+1)^2 \times N}$</td>
<td>$x \in \mathbb{Z}^{N \times 1}$</td>
</tr>
<tr>
<td>$h \in \mathbb{Z}^{(\Lambda+1)^2 \times 1}$</td>
<td>$\beta : \text{relative energy of noise}$</td>
</tr>
<tr>
<td>$\beta : \text{relative energy of noise}$</td>
<td>$\epsilon \leftarrow \min \left( \epsilon, \frac{e_{\text{max}}}{N^2} \right)$</td>
</tr>
<tr>
<td>$W = I_N$</td>
<td>$W \leftarrow \text{diag} \left( w_1, w_2, ..., w_N \right)$</td>
</tr>
</tbody>
</table>

**Initialization**

- $W = I_N$

**Until convergence, do**

- $\lambda \leftarrow \frac{\beta}{1 - \beta} \left( \frac{\text{tr}(D W D^T)}{N} \right)$
- $x \leftarrow W D^T \left( D W D^T + \lambda I \right)^{-1} h$
- for $n = \{1, 2, ..., N\}$, $e_i \leftarrow |x_n|^2$
- $e_{\text{max}} \leftarrow \max \{ e_i, i = 1, ..., N \}$
- $\epsilon \leftarrow \min \left( \epsilon, \frac{e_{\text{max}}}{N^2} \right)$
- for $i = \{1, 2, ..., N\}$, $w_i \leftarrow (e_i + \epsilon)^{\frac{1}{2}}$
- $W \leftarrow \text{diag} \left( w_1, w_2, ..., w_N \right)$
Because the weights in Equation (3.15) depend on vector $x$, the IRLS algorithm consists in calculating the solution iteratively, as summarized in Algorithm 1. First, the weights are all initialized to 1. Then, until convergence is reached, the algorithm alternates the two following steps: 1) given the weighting matrix, $W$, update the solution $x$ (line 7 of the algorithm); and 2) given the solution, update the weighting matrix (line 12 of the algorithm). Note that the term $\epsilon$ in the algorithm is a regularization term that prevents the weights from being equal to 0 [30].

3.5 Introduction to FPGAs

We have explained the process of SFT of the SMA sound scene followed by performing the plane-wave decomposition using the SFT signals. To implement the SFT process, we used a field-programmable gate arrays (FPGA). FPGAs provide great flexibility and scalability when design and implementation of SFT for SMAs. Once SFT is implemented on a FPGA which is associated with a microphone array, the host computer only needs to perform signal processing on the SFT signals.

Since their invention in the mid-1980s, field-programmable gate arrays (FPGAs) have grown significantly in popularity due to their effective programmability and reconfigurability. These advantages allow different design choices to be evaluated and adopted in a very short time. Unlike custom application-specific integrated circuit (ASIC) implementations, FPGAs are readily available at reasonable cost and allow a great reduction in a development cycle.

FPGAs consist of an array of configurable logic blocks (CLBs), static memories, digital-signal-processing (DSP) blocks, high-speed I/O pins, clock resources and distributed reconfigurable interconnects. The distributed reconfigurable interconnects allow different resources to be interconnected as required. Following are some of the important characteristics of FPGA resources and its development tools which are required to understand our FPGA design model.

3.5.1 CLB Resources on FPGAs

In digital system design using FPGAs, CLBs are configured to generate logic functions. The combinatorial logic circuits are implemented using look-up tables (LUTs) in CLBs. Each CLB has LUTs which can be configured and reconfigured according to the required function. Other than the LUTs, a CLB also has flip-flops (FFs) for sequential logic designing. A segment of the CLB in Virtex-6 FPGA is shown in Fig. 3.3. This segment consists of a 6-input 2-output LUT and 2 FFs is shown in Fig. 3.3. A CLB is a combination of several such segments. For example, a CLB in Xilinx Spartan-6, Virtex-6, Artix-7, Kintex-7 and
Virtex-7 family FPGAs consists of 8 of these segments. Therefore, a CLB consists of 8 LUTs and 16 FFs.

Figure 3.3: A segment of the CLB in Virtex-6 FPGA which consist of an LUT and 2 FFs. A CLB consists of 8 of these segments.

Further, Xilinx CLBs organize their resources as Slices. A CLB consists of 2 Slices. Therefore, a slice consists of 4 LUTs and 8 FFs. The schematic diagram of Xilinx CLB is shown in Fig. 3.4.

Figure 3.4: The schematic diagram of Xilinx CLB which consist of 2 Slices.

For an LUT the address is the function input, and the value at the address is the function output. Therefore, the number of inputs determines the complexity of the function which
can be generated by the LUT. However, regardless of the number of inputs, the LUT
queries only a single address. Therefore, increasing the number of inputs does not increase
the propagational delay of the combinatorial logic. However, it will consume more silicon
area. In modern FPGA families like Spartan-6, Virtex-6, Artix-7, Kintex-7 and Virtex-7
have 6-input LUTs.

3.5.2 Block Memory Resources on FPGAs

FPGAs have on-chip static memory resources which are called Block RAMs (BRAMs). The
block memories are consists of memory primitives which can vary from FPGA to FPGA.
Regarding Xilinx Virtex and Spartan FPGA series, BRAMs are fundamentally 36 Kb in
size where each block can also be used as two independent 18 Kb blocks. The 18 Kb
memory blocks can be cascaded to implement deeper and wider memories. Therefore, when
evaluating the BRAM utilization, calculation of 18 Kb block utilization is appropriate.

The block memories on the FPGA can be used to implement dual or single port RAM
modules, ROM modules, synchronous FIFOs. These memories can be easily generated by
Xilinx Block Memory Generator IP core [159]. Regarding a dual-port 18 Kb block RAM,
it consists of an 18 Kb storage area and two mutually independent access ports A and B.
Similarly, arbitrary size dual-port memory consists of the storage area and two mutually
independent access ports. The two ports permit shared access to its memory. Both ports
are functionally identical and providing read and write access. Simultaneous reads from
the same memory location may occur, but all other simultaneous, same location operations
should be avoided. Simultaneously reading-from and writing-to the same location results
in the correct data being written into the memory, but invalid data being presented at
the reading port. Each port has its own address (ADDR[A/B]), data in (DI[A/B]), data
out (DO[A/B]), clock (CLK[A/B]), port enable (EN[A/B]), and write enable (WE[A/B]).
The read and write operations are synchronous and require a clock edge. The data access
protocol of the dual-port block memory can be described using the timing diagram in
Fig. 3.5. As per the figure, to enable the data access the port enable (EN) should be set
high. Then as per the write enable (WE) the data can be read or write synchronously with
the clock (CLK). If the write enable (WE) is high, then the data present at data in (DI)
will be written into the memory location specified by address (ADDR). Otherwise, the data
in the address (ADDR) will be presented at data out (DO).

Block memory is a fast and limited resource which needs to be conserved when it may
become critical resources for the implementation. The Block Memory Generator core can
arrange block RAM primitives according to one of three algorithms: the minimum area
algorithm, the low power algorithm and the fixed primitive algorithm. The minimum area
algorithm provides a resource optimized solution, resulting in a minimum number of block RAM primitives. Dual-port memories usually operate beyond 300 MHz when they are configured as minimum area configuration.

3.5.3 DSP Resources on FPGAs

In digital systems, the period of the maximum operating frequency should be larger than the propagation delay of the critical path. When the FPGA architecture is dense, high propagational delays can occur in arithmetic circuits implemented by CLBs. This is due to routing delays between connected CLBs which are far apart. If this delay cannot be avoided by pipelining the critical path (by adding registers), the other option would be to use DSP blocks. The DSP blocks are ASIC in FPGA fabric, which can operate at higher frequencies than CLB-based modules. Typically they can be operated at a frequency higher than 400 MHz.

As an example, Xilinx DSP48E block consists of 25×18 bits 2’s complement multiplier and 3-input 48-bit adder/subtractor. Its basic block diagram is presented in Fig. 3.6. The multiplier inputs are asymmetric and accept an 18-bit 2’s complement operand and a 25-bit 2’s complement operand. It produces a 43-bit 2’s complement result which is sign-extended to 48 bits. If the processing data have higher width than the specified, DSP blocks are required to be cascaded which consumes additional DSPs. A number of available DSP blocks is very low compared to CLBs. Further, FPGAs consisting of more DSP blocks can be expensive. Therefore, DSP blocks should be conserved.
### 3.5.4 Other Important Features of FPGAs

One of the main advantages of using an FPGA is its large number of I/Os. In general, FPGAs have hundreds (some have thousands) of general purpose I/Os which allow easy integration with external systems such as high capacity dynamic memories and sensor arrays. Further, these I/Os provide very high bandwidth in digital communication.

The FPGA resources are interconnected via reconfigurable interconnects \[151\]. Placement and interconnection of resources are optimized by FPGA design tools. The design tools place and interconnect associated logic within a logic block or adjacent logic blocks to optimize speed performance and area efficiency.

FPGA is a high-speed parallel processing device which performs arithmetic/logic operations throughout the device in parallel. If the operations are required to be synchronous, the FPGA should have reliable distributed clock network. High-speed clock signals are not recommended to be connected via general interconnects. Further, clock synthesis is not accurate using logic resources due to propagational delays. FPGAs have dedicated clock synthesis and distribution resources to accurately synthesize and distribute the clock. In Xilinx FPGAs, the clock management tile (CMT) includes a mixed-mode clock manager (MMCM) \[157\] which provides clock frequency synthesis, deskew, and jitter filtering functionality. Further, the global clock trees allow clocking of synchronous elements across the device.

When designing FPGA architectures, developers do not need to develop everything
from scratch. The IP (intellectual property) cores which are available with the design tools, enable integration of ready-made system components to the design. Some of the important IP cores which will be discussed in this thesis are FFT, multi-port memory controller, tri-mode Ethernet, I2C and Microblaze soft-processor. With the evolution of designing tools, FPGA-based system developments become quick and easy. Developers can use high level languages for coding and they can be easily mapped to hardware by high-level synthesis tools. The high-level simulation tools such as Matlab has been integrated to FPGA synthesis tools, which enable accurate simulations while designing. Xilinx System Generator for DSP (XSG) and Xilinx Vivado Design Suite are good examples for high-level synthesis tools for FPGA designing.

3.5.5 FPGA Design Flow

The design entry of the design flow is creating the project file using a software tool. Assigning constraints such as timing constraints, pin assignments, and area constraints to the project are also considered in the design entry. In the coding phase, the model of the system is coded using high-level program (e.g., C, Matlab) or low-level program (e.g., Verilog, VHDL). Once coded, the system is synthesized which constructs a gate-level netlist from the code. After synthesis, the resource constraints should be met in order to implement the design in a target FPGA. If the resource constraints are not met, then the system needs to be re-coded to meet the requirements.

Once synthesis is successful, there might be many netlist files related to different modules of the system. The translating process merges all of the input netlists and design constraint information and generates a single file. Then the translated file is mapped to the targeted device which fits the design into the available resources on the device. Next, the place and route (PAR) process is performed which place and interconnects the mapped resources on the grid of the target FPGA. Place and route are performed off-chip and produces a file which is used to generate the bitstream. The placed and routed design should meet the timing constraints. Place and route can be run iteratively to find the best result. If the timing constraints are not met, then the system needs to be re-coded to meet the requirements. Finally, the FPGA can be programmed by downloading the bitstream into the device. The bitstream configures the interconnections and implements the system on the actual FPGA device.

3.6 The Concepts of FFT and Its Implementation on FPGAs

We now discuss the FFT computation in detail as it relates to the design considerations that form part of the thesis work. Fast Fourier transform (FFT) is a fundamental operation
in frequency domain filtering and beamforming. The transformation of time domain signal into frequency domain is implemented by FFT. FFT is a computationally efficient technique to perform discrete Fourier transform (DFT). If the transforming sample length is integer power of 2, the DFT function of $X(k)$ can be formulated s.t.:

$$X(k) = \sum_{i=0}^{N-1} x(i) \cdot e^{-j2\pi \frac{ki}{N}}, \quad 0 \leq k \leq N - 1 \quad (3.20)$$

where, $x(i)$ is time-domain sampled signal and $N$ is the FFT length. The FFT is computationally efficient because it takes the advantage of symmetry and periodicity of the complex sequence $e^{-j2\pi \frac{ki}{N}}$. Therefore, the DFT function can be reformulated s.t.:

$$X(k) = \sum_{i=0}^{\frac{N}{2}-1} \left(x(i) + (-1)^k \cdot x(i + \frac{N}{2})\right) W_N^{ki}, \quad 0 \leq k \leq N - 1 \quad (3.21)$$

where $W_N^{ki} = e^{-j2\pi \frac{ki}{N}}$ and referred to as twiddle factors.

The twiddle factors $W_N^{ki}$ describes a rotation vector which rotates in increments according to the number of samples $N$. Fig. 3.7 shows the symmetry and repetition of twiddle factors when $N$ is 2, 4 and 8.

As shown in the Fig. 3.7, $e^{-j2\pi \frac{ki}{N}}$ has redundant values which can be represented by a twiddle factor. Further, 180 degrees out of phase twiddle factors can be represented by the negative value of corresponding twiddle factor. Therefore, only $\frac{N}{2}$ twiddle factors are unique and required to be used when calculating DFT by FFT. The butterfly architectures which are used in FFTs are inspired by the symmetry and periodicity of the twiddle factors.

The basic FFT architecture which has 2-input and 2-output can be illustrated as in Fig. 3.8. The butterfly diagram represents the FFT architecture which is a visual aid to understand the algorithm. Fig. 3.9 shows the basic butterfly diagram which is shown in Fig. 3.8. In butterfly diagram, it can only flow on one path from input to output without doubling back. The value on the path should be multiplied with the input. The two lines merged at the output will be added together.

When number of inputs are increased, the butterfly diagram is expanded. The 8-input butterfly diagram is shown in Fig. 3.10. The butterfly diagram can be expanded while keeping the number of inputs to integer power of 2. Regardless the size, the two input butterfly diagram is the basic building block of larger butterfly diagrams. Basic butterfly diagrams are interconnected according to a certain pattern to make a larger butterfly diagram for FFT.

The basic butterfly unit discussed previously has only two inputs. However, it is possible to define a basic butterfly unit having input equivalent to integer power of 2 (i.e., 4, 8, 16,
Figure 3.7: The symmetry and repetition of twiddle factors when $N$ is 2, 4 and 8. The values in the bracket corresponding to $k_i$. Note that the unique number of twiddle factors are only $\frac{N}{2} \ll k_i$ for large $N$.

Figure 3.8: The basic radix-2 architecture

Figure 3.9: The basic radix-2 butterfly unit
The number of inputs in the basic butterfly unit is presented by radix of the butterfly diagram. The butterfly diagrams in Fig. 3.9 and Fig. 3.10 are radix-2 diagrams. Radix-4 basic butterfly diagram is shown in Fig. 3.11.

The data passing through butterfly diagrams are processed in stages as shown in Fig. 3.12. Each of the basic butterfly belongs to a particular stage. The number of stages is equivalent to $\log_r(N)$ where, $r$ is the radix and $N$ is the number of I/Os. Therefore, comparing radix-2 and radix-4 butterflies for same number of I/O,

$$\frac{\log_2(N)}{\log_4(N)} = \frac{\log_2(N)}{\left(\frac{\log_2(N)}{\log_2(4)}\right)} = \log_2(4) = 2$$

radix-2 has two times stages. When increasing the number of I/Os, butterfly diagram requires more basic butterflies. Each stage requires $\frac{N}{r}$ butterflies where $r$ is the radix and $N$ is the number of I/Os. Therefore, number of butterflies in the butterfly diagram can be
calculated by,

\[
\text{Number of stages} \times \text{Basic butterflies per stage} = \log_r(N) \cdot \left(\frac{N}{r}\right). \tag{3.22}
\]

The FFT architectures can be implemented as burst or streaming I/O fashion. In the burst mode data input need to be halted while FFT is being calculated. The computation is done in-place which uses the input buffer for storing results of each stage. Therefore, neither input nor output is continuous which describes as a burst. In contrast, in streaming mode input can be continuously loaded to the architecture while output can be continuously unloaded.
In burst mode, FFT implementation only utilizes the basic butterfly diagram of required radix. Consequently, FFT is performed by reusing the basic butterfly log₂(N) · \( \left( \frac{N}{2} \right) \) times as stated by Eq. 3.22. The FFT module is designed in such a way that it can load the relevant twiddle factors from a ROM in each new computation. Since there are \( \log_r(N) \cdot \left( \frac{N}{2} \right) \) repetitions of basic radix-\( r \) butterfly operations, it can be seen that higher radix butterflies have less latency when performing FFT computation. Further, latency is proportional to number of basic radix computations. Regarding the radix-2 and radix-4 butterflies, the ratio is,

\[
\frac{\log_2(N) \cdot \left( \frac{N}{2} \right)}{\log_4(N) \cdot \left( \frac{N}{4} \right)} = \frac{\log_2(N) \cdot \left( \frac{N}{2} \right)}{\log_2(N) \cdot \left( \frac{N}{2} \right) \cdot \left( \frac{N}{4} \right)} = \frac{\log_2(4)}{\left( \frac{N}{2} \right)} = 4. \quad (3.23)
\]

For arbitrary radix-\( r_1 \) and radix-\( r_2 \), the latency ratio can be evaluated s.t.,

\[
\frac{\log_{r_1}(N) \cdot \left( \frac{N}{r_1} \right)}{\log_{r_2}(N) \cdot \left( \frac{N}{r_2} \right)} = \frac{\log_{r_1}(N) \cdot \left( \frac{N}{r_2} \right) \cdot \left( \frac{N}{r_2} \right)}{\log_{r_1}(N) \cdot \left( \frac{N}{r_2} \right)} = \log_{r_1}(r_2) \cdot \left( \frac{r_2}{r_1} \right). \quad (3.24)
\]

In contrast, in streaming I/O mode, a butterfly is implemented in each stage. Therefore, \( \log_r(N) \) butterflies are required to implement streaming I/O FFT architecture which consumes more computational resources and memory over burst mode. Note that the butterfly computations within a stage still need to be performed by reusing the butterfly in the respective stage. In streaming mode, a channel data stream requires a dedicated FFT module. In SFT, we anticipate sharing the FFT modules between microphone channels to save resources. Therefore, streaming FFT mode is not a probable option.

FFT can be performed in two methods named decimation-in-time (DIT) and decimation-in-frequency (DIF). In DIT, the FFT is performed in such a way that equivalent DFT is computed by decomposing the function to even and odd samples. In contrast, DIF considers a first-half/second-half approach in equivalent DFT. The two methods can be illustrated based on the Eq. 3.21 s.t.,

\[
\text{DIT : } X(2k) = \sum_{i=0}^{\frac{N}{2}-1} \left( x(i) + x(i + \frac{N}{2}) \right) W_N^{ki} \quad 0 \leq k \leq N - 1,
\]

\[
X(2k+1) = \sum_{i=0}^{\frac{N}{2}-1} \left\{ \left( x(i) - x(i + \frac{N}{2}) \right) W_N^{ki} \right\} W_N^{ki} \quad 0 \leq k \leq N - 1,
\]

\[
\text{DIF : } X(k) = \sum_{i=0}^{\frac{N}{2}-1} \left( x(i) W_N^{ki} \right) + \sum_{i=\frac{N}{2}}^{N-1} \left( x(i) W_N^{ki} \right) \quad 0 \leq k \leq N - 1. \quad (3.25)
\]

The DIT requires input to be bit-revered order to decompose the input to even and odd samples. The calculation of bit-revered order is a permutation of binary value which
Table 3.1: The permutation of bit-revered order of FFT input.

<table>
<thead>
<tr>
<th>Natural Order Index</th>
<th>Natural Order Binary</th>
<th>Bit-revered Binary</th>
<th>Bit-revered Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

can be illustrated by Table 3.1. When input the sample data, they are not required to be reordered as the FFT architecture is implemented in such a way that it can access the data in bit-revered order from the input RAM. However, FFT operations cannot be started until all the input data are loaded into the RAM. Therefore, when the FFT is configured for DIT, it operates in burst mode.

In contrast, the output of the DIF FFT configuration is in the natural order. Therefore, an output of a particular stage can be subjected to process by the next stage when it is being generated. Therefore, the FFT architectures configured in streaming mode uses DIF method. Nevertheless, we do not use streaming mode FFTs as they consume more resources.

3.6.1 FPGA-based FFT implementation

In this section, we describe configurations of the burst FFT module, which we considered in the model analysis. In burst mode, 3 types of FFT configurations are widely used: 1) Radix-2 burst I/O, 2) Radix-2 lite burst I/O and 3) Radix-4 burst I/O [50]. The radix-2 burst I/O configuration is shown in Fig. 3.13. The main component of the FFT module is the radix-2 butterfly. The FFT is performed in-place which save on-chip memory. The switches are used to access and store data in appropriate memory locations. The memory words of RAMs and ROM contain real and imaginary parts of the complex data. The \( \frac{N}{2} \) complex twiddle factors are stored in ROM \( W_0^{\frac{N}{2}} \) which is \( \frac{N}{2} \) words deep. At the start, RAM contains audio input data and they are over-written during the FFT computation. At the end, RAM contains the result of FFT. In \( N \)-point FFT, the size of the RAM is \( N \) complex words which is implemented as 2 blocks of memories each \( \frac{N}{2} \) words. Even though the input data is real, since the intermediate results and the output are complex, the RAM requires to store complex words for in-place computation when FFT.
Figure 3.13: The radix-2 burst I/O configuration. The radix-2 butterfly consists of a complex multiplier, complex adder and complex subtractors. The twiddle ROM $W^0_2$ is a $\frac{N}{2}$ words single block of memory where memory word consists of real and imaginary part of the complex data. The I/O buffer consists of 2 blocks of memories each $\frac{N}{2}$ complex words in size.

Regarding the radix-2 butterfly, the real and imaginary parts of the FFT can be computed in two steps if the increase of latency can be acceptable. This minimizes the resource requirement as complex data computation can be transformed to a sequence of real data computation. This is the concept of radix-2 lite burst I/O method. Fig. 3.14 illustrates the radix-2 lite butterfly architecture. As shown in the figure, the twiddle factor ROM remains same as in radix-2 configuration. However, the input-sample RAM is implemented as a single memory having the same width and twice the size of RAM in radix-2 configuration.

For high performance FFT, radix-4 burst I/O configuration is better than radix-2 configuration. As described previously, when increasing the radix, the number of butterfly stages to be computed in FFT decreases. Therefore, radix-4 configuration has lower latency (higher throughput) than radix-2. The FPGA architecture for radix-4 burst I/O configuration is shown in Fig. 3.15. As shown in the figure, the radix-4 butterfly requires more arithmetic resources compared to radix-2. Regarding the memory requirement, each memory in the figure is $\frac{N}{4}$.

In the implementation of an FFT butterfly LUTs, FFs, DSPs and BRAMs are the main resources in use. There are 2 important FFT parameters which are important when
analyzing the required resources by an FFT: 1) the data width of the FFT samples, and 2) the configuration of multipliers and adders of the butterfly.

### 3.6.1.1 The Precision of FFTs

In audio signal processing, IEEE-754 32-bit floating point precision is acceptable and commonly used. The IEEE-754 floating point format is shown in Fig. 3.16 which can be defined s.t.,

$$\text{Value} = (-1)^{sign} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right) \cdot 2^{e-127},$$

where, $sign = b_{31}$ and $e = b_{30}b_{29} \ldots b_{23}$ [100]. As it can be seen, the fractional data is represented by 23-bit binary value. Since audio data exists between -1 and 1, 8-bit exponent is not important. Therefore, 24-bit data format is sufficient to represent audio data in a precision similar to IEEE-754 floating point. The ADCs which are used with the microphones can generate the audio data in 24-bit 2’s complement binary format. Further, Xilinx FFT module can be configured to support the I/O for 24-bit 2’s complement. Therefore, we used 24-bit 2’s complement binary format throughout the SFT architecture. In 24-bit 2’s complement format, the MSB is the sign bit and other 23 bits represent the fractional magnitude of audio data in 2’s complement.
Figure 3.15: The radix-4 burst I/O configuration. The radix-4 butterfly consists of 3 complex multipliers, 4 complex adders and 4 complex subtractors. The twiddle memory is implemented by 3 memory blocks each provide twiddle factors for respective multiplier. The I/O buffer consists of 4 memory blocks. Each memory block is \( \frac{N}{4} \) complex words in size.

Figure 3.16: IEEE-754 32-bit floating point data format.

### 3.6.1.2 The Configuration of Complex Multipliers in FFTs

The butterfly architecture consists of complex multipliers and adders which consumes substantial resources of the whole architecture. Regarding the complex multipliers, fundamentally there are 2 configurations \([163]\) which are,

1. 4-multipliers and 2-add/sub configuration,
2. 3-multipliers and 5-add/sub configuration.
Assuming \( x \) and \( y \) are complex numbers, mathematically they can be illustrated s.t.,

\[
\begin{align*}
    x &= a, bi \\
    y &= c, di \\
    xy &= (a, bi)(c, di) \\
    &= (ac - bd), (ad + bc)i \\
    &= \{a(c + d) - (a + b)d\}, \{a(c + d) + (b - a)c\}i
\end{align*}
\]  

(3.28) 

(3.29)

Eq. 3.28 and Eq. 3.29 describe 4-multiplier and 3-multiplier complex multipliers respectively. Note that the strikeout term in Eq. 3.29 is a repetition which can be omitted.

Therefore in FPGA-based FFT implementation, the complex multipliers in the butterfly can be configured 1) using CLBs to save DSP resources 2) using DSP and 3-multiplier structure to optimize DSP utilization, or 3) using DSP and 4-multiplier structure to optimize performance. Table 3.2 shows the resource utilization and performance of different complex-multiplier configurations implemented with different resources.

Table 3.2: Resource utilization and performance of different complex-multiplier configurations implemented with different resources.

<table>
<thead>
<tr>
<th>Resource and Performance</th>
<th>3-multiplier option (DSP-based)</th>
<th>4-multiplier option (DSP-based)</th>
<th>3-multiplier option (Slice-based)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>FF</td>
<td>100</td>
<td>84</td>
<td>1545</td>
</tr>
<tr>
<td>LUT</td>
<td>76</td>
<td>84</td>
<td>1631</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>6</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>F_{\text{max}} (MHz)</td>
<td>590</td>
<td>590</td>
<td>321</td>
</tr>
</tbody>
</table>

3.7 Introduction to Multithreaded Computing Architectures

In the previous section, we have explained the background related to the implementation of SFT architecture on an FPGA. In this section, we explained the background related to multithreaded computing architectures which can be used for plane-wave decomposition by utilizing a thread to a plane-wave decomposition problem which is solved using the IRLS algorithm. This section is important to understand the work we did in Chapter 6 using multithreaded computing architectures.

Basically two main computer architectures have been evolved to increase the performance of computing. They are sequential and parallel computer architectures. Before
2005, as described by the Dennard scaling \cite{31}, the power density of a processor die remained constant even when the size of the transistor got small. Consequently, according to Moore’s law \cite{87}, when transistors are doubled in a given area, the performance of the processor also doubled since power density remains same. In other words, this means performance per watt increases at the same rate of Moore’s law. This trend inspired the computer architectures to rely on single core sequential computations.

The concept of the sequential computation was also backed by the famous Amdahl’s law \cite{8} which stated that the ultimate speedup can be achieved only by speeding up the sequential computation of the algorithm. Assuming that computation problem size does not change when running on parallel architecture, Amdahl’s law stated the achievable speedup \( S(N_p) \):

\[
S(N_p) = \frac{T(1)}{T(N_p)} = \frac{T_s + T_p}{T_s + \frac{T_p}{N_p}},
\]

where, \( T_p \) is the time taken to execute perfectly parallelizable portion in the algorithm with no communication and synchronization overhead while \( T_s \) is the time taken to execute sequential portion. The \( N_p \) is the number of parallel cores in the architecture. Therefore, as per the equation even when the fraction of serial work in a given problem is small, the maximum speedup obtainable on an infinite number of parallel processors is only \( \frac{T_s + T_p}{T_s} \) which is highly pessimistic towards parallel processing multicore architectures.

Therefore, with the inspiration of Dennard scaling and Amdahl’s law, before 2005 processors were designed with a single core and operated by high clock frequencies to speed up the sequential computation. The evolution of Intel Pentium and AMD Athlon processor series during this period are good examples of this trend. They operated in multiple Gigahertz of clock speed with a single core. After 2005, Dennard scaling breaks down and single-core processors failed to increase the performance according to Moore’s law. This was due to power density increment with increasing transistor count in a unit area. Once the power density increases above a certain limit, to keep the chip temperature in the safe operating range the power of the transistors should be limited even though the transistor density increases with Moore’s law. This unavoidable condition is called as dark silicon which refers the amount of silicon that cannot be powered on at the nominal operating power \cite{39}. It is predicted that in 8 nm semiconductor fabrications, the dark silicon may reach up to 50%-80% depending on the processor architecture, cooling technology, and application workloads.

To efficiently use the silicon area, processor architectures are moved into multicore designs. Even though the increase of die area is inevitable to minimize the dark silicon or power density, the multicore architectures offer better performance exploiting options than
counterpart single-core architectures. Pollack’s rule [104] states that performance increases due to microarchitecture advances are proportional to the square root of the increase of area. In contrast, the multicore architectures increase the performance potential proportional to the area.

Even though Amdahl’s equation undermines the parallel processing architectures, John Gustafson [54] argued that the assumption made by Amdahl’s law that the computation problem size does not change when running on parallel machines is not valid in most practical scenarios. According to Gustafson argument, when a number of cores are increased the problem size also can be increased. He stated that it should be runtime, not the problem size which has to be assumed as constant. Consequently, when the speedup is measured by scaling the problem to the number of processors, significant speedup can be achieved. However, Gustafson considers a machine with greater parallel computation ability while workloads are also fully scaled with parallel nodes which is unreasonable to be generalized. Similarly, Paul and Meyer developed few models [101] which question the validity of Amdahl’s law when it is applied to single-chip heterogeneous multiprocessor designs. In their analysis, they argued Amdahl’s law just impede the potential of parallel computing systems. Moreover, in 2008 Hill and Marty [58] showed that robust general-purpose multicore designs can gain speedup under Amdahl’s law.

3.7.1 Processor Performance and Memory Latency

Development of the processor architectures before and after Dennard scaling [31] requires much higher dynamic-memory bandwidth compared to the achievable bandwidth. Typically the memory access delay is roughly a factor of 100 from processor cycles. If this is the case, the performance of a processor is bound by the memory bandwidth and increase of the performance is useless. However, due to localities in memory access of various programs, it can be observed that some data can be copied to a small fast memory and subjected to many computations. This kind of memory access inspired to introduce on-die cache to the computer architecture.

There are two types of memory access localities. The multiple accesses to the same data within a short period is called temporal locality and accessing data in the close neighborhood of an earlier access is called spatial locality. Both localities are advantages to improve the performance. Once the data are cached using the temporal locality, many computations can be performed to increase the computation to memory access ratio. In addition, not only the required data but also neighborhood up to some specific size can be transferred into the cache to take advantage of the spatial locality. The neighborhood data are loaded simultaneously via multiple cache-links.
In practice data caching is done in several stages to optimize the performance. The modern processor cache hierarchy consists of level 1, 2 and 3 caches which are built on the processor die. L1 cache is the fastest and smallest cache. It is built by most expensive memory cells closest to the processor core. When moving away from the core from L1 to L3 caches, the size and the latency both increased. In multicore processor architectures, L1 caches are private to hardware threads while L2 and L3 caches are shared among hardware threads and cores respectively. The mapping of data between different caches is done automatically by the hardware. If data is required but not present in the L1 cache, then the data will be searched in L2 and continue to search sequentially until it is found down the memory hierarchy. The missing of data in a cache is called cache miss and availability of data in the cache is known as a cache hit. A cache hit is desirable as it reduces the latency of program execution.

Both hardware and software contributions are required to increase the cache hit rate. Regarding the software implementations, programmers’ awareness on the temporal and spatial localities of the data structures executed on the processor can increase the cache hit rate. Regarding the hardware architecture, the cache can be designed in different configurations such as direct-mapping, associative or set-associative. These cache configurations differ by the address decoding/mapping when reading or writing data between cache and main memory. Therefore, these configurations trade-off either cache hit rate of the memory access or hardware-implementation complexity. The associative cache has the highest hit rate and complexity, while direct-mapping caches are the simplest with low hit rate. The set-associative cache is a combination of direct-mapping and associative cache which trades off both simplicity and cache hit rate of each. The size of the cache is also important to increase the cache hit rate and down the cache hierarchy, cache sizes are also increased. Consequently, the data access latencies from the cache also increased, which requires a trade-off between cache size and speed. In Intel processors, generally L1 (32 kB) and L2 (256 kB) caches are set-associative while L3 (8 MB) cache is associative.

### 3.7.2 Introduction to Memory Models of Computations

The parallel architectures require supportive memory hierarchy and inter-processor communication techniques. When parallel algorithms are developed for parallel architectures, the configuration of the caches and inter-processor communication techniques significantly impact on the performance of the algorithms. There are many memory models available in the parallel computing literature. Here we focus on 3 memory models which are widely used to explain the performance of multithreaded computing architectures. They are:

1. Parallel Random-Access Machine (PRAM)
2. Parallel External Memory (PEM)  
3. Bulk Synchronous Parallel (BSP)  

which are graphically presented in Fig. 3.17.

PRAM model consists of processors which are attached to coherent external shared memory [44]. The shared memory is used to access data and inter-processor communication. Each processor can run a single thread at a time. The model does not have caches or any memory hierarchy, and each processor owns set of registers to compute data. Because of no memory hierarchy, all processors take the same amount of time to access data from shared memory. Because of this simplicity, many works in parallel algorithms have used PRAM model. Further, this model is effective when studying how parallel access of shared data are handled. However, lack of caches and a memory hierarchy fails to accurately model the execution time of the algorithms on modern parallel architectures. Further, data access patterns in shared memory are too random to utilize spatial locality [12].

As an extension to PRAM model, PEM model is proposed by Arge et al. [13] to model I/O efficient external memory with caches. The model consists of \( P \) processors each having a private cache of size \( M \) and all the processors share a common external memory. The caches are partitioned into blocks of size \( B \) and data is transferred between main memory and the cache in blocks of size \( B \). The processors can compute only the data in their caches. Processors cannot access other processors’ caches and inter-processor communication is possible only via common shared memory. PEM model is widely applied for parallel algorithms which are implemented on private-cache chip multiprocessor (CMP) architectures. Modern multi-core processors are examples for CMP architectures.

The performance of CMP architectures can be described by PEM model since each thread can efficiently use the associated caches. When the number of parallel problems is lesser than the number of threads, then the performance increases with the number of problems. However, when increasing the problems, the cache efficiency decreases. This is because the cache is sharing with multiple problems. Consequently, the rate of increasing the performance of computation decreases. Once the numbers of problems reach the point where data caching is ineffective, the performance can be described by PRAM model. We have used this behavior to explain the results in chapter 6.

Using the multi-threading concepts, some architectures have been developed to utilize a large number of threads. They are called many-thread architectures. In them, the threads hide memory-access latency by interleaving the memory access when threads stall for a resource or data. Their performance depends on how well the latency is hidden. This
Figure 3.17: Different memory models which can be used to understand parallel computations. In these figures, CPU represents the processing resources which can be a hardware thread, core or processor depending on the architecture.
performance can be described by the Threaded Many-core Memory (TMM) model\cite{79}. GPGPUs are examples for many-thread architectures.

In contrast to CMP or TMM, with the reduction of processing cost, systems are evolved by networking many processing units. The usability of this kind of parallel computer is well described by BSP model\cite{128}. BSP model is applied to computer architectures which have no shared memory. Such architectures consist of a network structure (i.e., router) where, the inter-processor communication is done via message passing through the network. The model assumes each processor has its internal memory for computation which avoids the burden of memory management in parallel processing. Further, as implied by the name, the model facilitates synchronizing of all or subset of the processors at regular intervals. In fact, a computation consists of a sequence of *supersteps* wherein each superstep a processor is allocated a task of local computations, message transmission and reception. After each synchronizing period, a global check is made to determine whether the superstep has been completed by all the processors. If the previous superstep is completed, the system proceeds to the next superstep, else next synchronizing time period is allocated to the unfinished superstep. By this way, the model accounts the options of assigning communication and performing low-level synchronization. BSPlib and message-passing interface (MPI) libraries are tools which can be used in parallel computing with network infrastructure. They are widely used in distributed-memory parallel computing.

From the above discussion, we have understood how important the underlying architecture is to achieve high-performance computing. However, the computing architecture alone will not make the computation efficient. Even though parallel processing architectures have been evolved with many threads, exploiting parallelism within the algorithm also important to achieve high performance by parallel computers\cite{58}.

### 3.7.3 Introduction to Multi-processor, Multi-core and Many-thread Architectures

In the previous section, we have presented few memory models used which are used to describe parallel-processing architectures. In this section, we describe characteristics of typical computing architectures which are used in this thesis to analyze the performance of sparse-recovery with the multithreaded environment. Firstly, we explain following terms which we used to describe parallel-processing architectures.

- **Hardware (H/W) threads**: The number of streams of execution supported by Hyper-threading technology. Each hardware thread should have a dedicated execution unit.

- **Core**: Physical hardware that works on the hardware threads. In a core, there may be 1 or more hardware threads.
- Processor/Device: Collection of cores which interface with the system motherboard via a physical socket.

- Platform: A system consists of 1 or more processors.

- Multithreaded architecture/platform: A system performs its computations using multiple hardware threads.

Regarding the CMP architecture, two types of platforms can be discussed. They are single processor platform and multiprocessor platform. In single processor platform, there is only 1 CMP processor in the platform (i.e., on the motherboard). In contrast, in multiprocessor platform, there are more than 1 CMP processor in the platform. Irrespective of the number of processors in the platform, each processor is interfaced with the dynamic memory via dedicated memory channels. Therefore, PEM and PRAM memory models can be applied to multiprocessor platforms as well. The basic block diagram of the multiprocessor architecture is given in Fig. 3.18. In multiprocessor platforms, the number of processors is equivalent to the number of occupied processor sockets on the motherboard. In each CMP processor, there are few cores and each core can handle one or more hardware threads. Therefore in multiprocessor platforms, the number of H/W threads can be calculated as:

Figure 3.18: Architecture of the multiprocessor platform. There are 2 processors in the diagram each connected to a socket on the motherboard. Each processor has its local dynamic memory which is connected via memory channels. These processors are CMP type which has multiple cores on the processor die. Each core contains 1 or more threads.
Number of H/W threads = Number of processors × Number of cores per processor
× Number of H/W threads per core , \hspace{1cm} (3.31)

where, the number of cores per processor and the number of hardware threads per core can be found in the processor user manual.

Regarding the integration of multiple processors to the dynamic memory, two main architectures are commonly used. They are 1) NUMA (Non-uniform memory access) and 2) SMP (Symmetric multiprocessing). In NUMA the memories interfaced with different processors are not shared equally between the processors. Therefore, memory access latency to different memory regions vary depending on if the accessed region is local to the processor or not. In contrast, SMP shares the dynamic memory space between all the processors equally which has same memory access latency to any memory region from any processor.

The multiprocessor platforms are generally NUMA architectures as shown in Fig. 3.19. The dynamic memories which are interfaced with different processors are not uniform. These

![Diagram of Memory Architecture](https://via.placeholder.com/150)

**Figure 3.19:** Memory architecture of a multiprocessor platform. This is a NUMA architecture which consists of 2 interconnected processors. The processors are interfaced via point-to-point processor interconnect (e.g., Intel-QPI, AMD-HT). A processor has lower memory access latency to its local-dynamic memory compared to the dynamic memory attached to the other processor.

memories are physically separated on the motherboard. If a processor needs to access data in a memory which belongs to a different processor, they need to be accessed via point-to-point processor interconnect. In Intel, this interconnect is named as QPI (QuickPath Interconnect) and in AMD this is called HT (HyperTransport) \([56]\). In the sparse recovery algorithm, we assume a thread which executes an IRLS problem on a particular processor
only access data from the dynamic memory local to that processor and avoids interprocessor
data transfers.

In GPGPUs and Intel many-core coprocessors, the memory integration architecture is
different to CMP architectures. In fact, they have a dynamic memory in the device, which
is interfaced with the processors. In Intel Xeon Phi coprocessor, the processor architecture
is similar to Intel 32-bit multithreaded core (i.e., Intel multithreaded x86 architecture).
Each of these processors can handle 4 hardware threads. There are many such processors
integrated to the device memory (e.g., 59, 60 or 61 which depend on the device) via mem-
ory channels \[42\]. The basic block diagram of the multiprocessor architecture is given in
Fig. 3.20. As per the figure, the number of H/W threads can be calculated s.t.,

![Block Diagram](image)

Figure 3.20: Architecture of Intel Xeon-Phi coprocessor platform. This is Phi 5110P device
which consists of 60 processors. Each processor has 4 threads. Therefore, there are 240
threads in the coprocessor. The processors are connected to the device dynamic memory
via a ring bus. The Phi device is connected to the Host (i.e., processor platform) via PCI
Express.
Number of H/W threads = Number of processors
\times\text{Number of hardware threads per processor}, \quad (3.32)

where, the number of processors and the number of hardware threads per processor can be found in the coprocessor user manual.

In GPUs, the number of processors is equivalent to the number of streaming multiprocessors (SMs) in the device. The basic block diagram of the GPU architecture is given in Fig. 3.21. Similar to other processing architectures, a GPU thread is the basic instruction execution process on the GPU processor. There are many resources (i.e., ALUs, FPUs, registers, shared memory, etc.) on the processor to execute many threads simultaneously. Even though there are many ALUs and FPUs on a GPU processor (e.g., 192 ALUs and FPUs per SM in Nvidia K40 GPU), the fast memory allocated to each processor is small (e.g., 64 KB per SM in Nvidia K40 GPU) [96]. Note that in here the fast memory meant the

Figure 3.21: Architecture of Nvidia GPU platform. This is Nvidia K40 device which consists of 15 processors named Streaming Multiprocessors (SMs). Each processor consists of 192 ALUs and FPUs which can carry out simultaneous 192 hardware threads. All the processors are connected to the device dynamic memory via a cache-coherent L2 cache between processors. The GPU is connected to the Host via PCI Express.
combination of L1 cache and shared memory on a GPU processor (see Fig. 3.21). Therefore, all the threads in a processor should share this small fast memory which is in similar scale to a thread cache (i.e., L1 cache) in a CMP.

Regarding the interfacing with the dynamic memory, GPU and Intel Phi architectures are fallen into SMP (Symmetric multiprocessing) category where, the device dynamic memory is shared between all the processors equally. Fig. 3.22 and Fig. 3.23 show the device architectures of the two devices which express the integration of processors and the dynamic memory via memory channels.

As per Fig. 3.22, Intel Xeon-Phi 5110P coprocessor has 60 processors which are integrated with the dynamic memory via 1024-bit ring network. The device contains 8 memory controllers, and each connected to 4 memory chips via dedicated memory channels. Therefore, there are 32 memory channels in the coprocessor. Each memory channel is 16-bit wide and can be operated at 5 GT/s speed.

Similar to Intel Phi coprocessor, Fig. 3.22 shows Nvidia K40 GPU which consists of
15 processors which are integrated with the dynamic memory via the crossbar. The device contains 6 memory controllers, and each connected to 4 memory chips via dedicated memory channels. Therefore, there are 24 memory channels in the GPU. Each memory channel is 16-bit wide and can be operated at 6 GT/s speed.

Figure 3.23: Memory architecture of Nvidia K40 GPU. The processors and memories on the device are interconnected via the crossbar. The device contains 6 memory controllers, and each connected to 4 memory chips via a dedicated memory channel. Therefore, there are 24 memory channels in the GPU. Each memory channel is 16-bit wide and can be operated at 6 GT/s speed.
Chapter 4

Development of a FPGA-based Audio Preprocessing System

4.1 Introduction

The audio acquisition, preprocessing and transmission using a dedicated computing platform helps to spare the resources of the main computing platform for sparse plane-wave decomposition. Further, implementation of the SMA data acquisition using an embedded system improves the portability of the SMA. This motivated to perform the spherical Fourier transformation (SFT) of microphones data on an FPGA-based embedded platform. FPGA enables designing of flexible and reconfigurable embedded architectures which require parallel processing and high I/O count. In this chapter, we describe a development of an FPGA-based embedded platform which can integrate to an analog microphone array, acquire microphone data, perform SFT on acquired data and transmit the preprocessed data to a distant computer. The system consists of:

- Audio-acquisition board which consists of analog-to-digital converters (ADCs) to integrate an analog microphone array to an FPGA to acquire data,
- ADC-configuration subsystem to configure the ADCs to acquire microphone data,
- Audio-acquisition subsystem to drive ADC audio interfaces to acquire microphone data,
- UDP/IP data transmission subsystem to transmit the filtered output via Ethernet cable,
- DDR3-memory subsystem to store filter coefficients for audio pre-processing,
- Spherical Fourier transformation (SFT) subsystem.
The block diagram of the system is shown in Figure 4.1. The system is flexible and scalable where the number of microphones and the order of the SFT can be changed.
4.2 FPGA Platform

We used Xilinx ML605 FPGA development board to implement the FPGA-based audio pre-processing system. Figure 4.2 shows the ML605 board which consist of Virtex-6 XC6VLX240T-1FFG1156 FPGA. The ML605 is versatile embedded system development platform having many resources. We used the on-board DDR3 SODIMM memory, the tri-mode Ethernet physical interface (PHY) and the high-speed VITA-57 FMC connector for implementing the system.

4.3 Audio-acquisition Board

Now we describe the audio-acquisition board. The audio-acquisition board is highlighted in Fig 4.3. The audio-acquisition board mainly consists of 10-32 female connectors and ADCs. The 10-32 female connectors are used to connect the analog microphones of the SMA. The analog microphone signals are converted to digital by the ADCs. Texas Instruments (TI) TLV320ADC3101 low-power stereo ADC chips [121] are used in the board. Each ADC chip consists of 2 ADCs. It supports sampling rates from 8 kHz to 96 kHz and maximum of 24-bit sample width. It has an inbuilt programmable phase-locked loop (PLL) for flexible audio clock generation which can be driven by an external master clock (MCLK). It supports single-ended or differential microphone signals. The gains of the connected microphones can be configured using on-chip programmable gain controllers. The ADC can be configured
using I2C interface. Each chip has 2-bit configurable I2C address which can be used when integrating multiple chips (i.e., maximum of 4). The audio serial output can be programmed to support I2S and many other modes. The I2S output can be operated in either master or slave mode. The stereo output has 92-dBA signal-to-noise ratio (SNR). Typical connections to the ADC chip are shown in Fig. 4.4.

Figure 4.3: The highlighted section in the system is the audio-acquisition board.

Figure 4.4: The typical connections of TLV320ADC3101 ADC.
The audio-acquisition board and ML605 platform are connected using a FMC-ADC adapter as shown in Fig. 4.5. The FMC-ADC adapter is connected to high-pin-count (HPC) FMC connector (Samtec ASP-134486-01) on ML605 platform. The FMC connector can connect 160 single-ended signals. SAMTEC QSH-060-01-F-D-A connector on the FMC-ADC adapter is connected to the audio-acquisition board using a cable. The I2C signals which are used to configure the ADCs and the I2S signals which are used to capture the ADC data are connected via this interface. The audio-acquisition board is scalable such that the number of ADCs can be increased based on the number of microphones. The scalability of the audio-acquisition board is inherited by the design of FPGA-based ADC-configuration subsystem and the audio-acquisition subsystem. Following is the description of the ADC-configuration subsystem and the audio-acquisition subsystem.

4.4 ADC-configuration Subsystem

TLV320ADC3101 is a programmable ADC which can be set up by configuring its registers. In the audio-acquisition board, the ADCs are configured via I2C interfaces by a processor-based subsystem on the FPGA. The ADC-configuration subsystem is highlighted in Fig. 4.6. The description of the I2C interface and the protocol is given in Appendix A. In our design, the I2C interface operates in standard mode and uses 7-bit addressing scheme. The ADC chip has a configurable 7-bit I2C address. The 5 MSBs of the chip address are fixed and
Figure 4.6: The highlighted section in the system is the ADC-configuration subsystem.

cannot be changed, while the 2 LSBs can be set by external pull-up/down. Therefore, a single I2C master can communicate with 4 ADC chips which having distinguish addresses. The schematic of the 4 ADC chip connection is shown in Fig. 4.7. Since 4 ADC chips can

Figure 4.7: The interfacing of 4 ADCs with a common I2C master. The 2 LSBs of the I2C address can be set by external pull-up/down. The other 5 MSBs of the chip address are fixed and common.

be interfaced and programmed via a single I2C master, we integrated 4 ADC chips into a PCB module having a small footprint. We call this ADC module. Depending on the number of microphones, ADC modules are deployed using a baseboard. The baseboard is referred as ADC motherboard which is the same audio-acquisition board which is described
in the previous section. ADC modules can be plugged into ADC motherboard. ADC motherboard consists of microphone jacks, ADC-module sockets, connector to interface the FPGA platform and ADC power supply.

We implemented an ADC motherboard which facilitates 8 ADC modules to interface 64 analog microphones (Note each ADC module consists of 4 ADC chips and each chip consists of 2 ADCs). Fig. 4.8 shows the images of ADC modules, ADC motherboard and the way they are connected. ADC-configuration subsystem contains a dedicated I2C master to program an ADC module of 4 ADC chips. The I2C masters are implemented using Xilinx I2C IP core. Each I2C IP core is an I2C master which programs a specific ADC module. They are connected to Xilinx Microblaze processor. The ADCs are configured using a bare-metal C program running on the Microblaze processor which is described in Appendix C. The schematic diagram of the ADC-configuration subsystem is shown in Fig. 4.9. The system is developed using Xilinx Embedded Development Kit (EDK).

4.4.1 Implementation of the ADC-configuration Subsystem

Now we describe the implementation of the ADC-configuration subsystem. ADC-configuration subsystem is implemented using Xilinx Embedded Development Kit (EDK). The implemented audio-acquisition board consists of 8 ADC modules which we programmed using 8 I2C masters. We implemented the I2C masters using Xilinx I2C IP cores. Xilinx I2C IP cores are integrated to Microblaze processor as shown in Fig. 4.10 using EDK. Processor Local Bus (PLB) is used to interface the I2C cores to Microblaze processor. Each I2C module
Figure 4.9: ADC-configuration subsystem. It contains a dedicated I2C master to program an ADC module. The ADCs are configured using a bare-metal C program running on the Microblaze processor.

Figure 4.10: The ADC-configuration subsystem which consists of 8 I2C IP cores and a Microblaze processor.
produces SCL and SDA lines which will be connected to ADC module. The interface of the I2C core is shown in Fig. 4.11. Each I2C core is mapped to a particular address space in the processor. The start address of the address space is called base-address and the last address is called high address. The I2C cores are referred by the processor using the base-address. The allocated address spaces to the I2C cores are shown in Fig. 4.12.

Figure 4.11: The configuration of the I2C interfaces. The I2C core is attached to the Microblaze using PLB. The SCL and SDA wires are configured as external ports to connect with an ADC module.

Figure 4.12: The allocation of the Microblaze processor’s address space to I2C cores. Each I2C core is uniquely referred by their base-address.
In this architecture, the I2C cores are operated in standard mode and use 7-bit addressing scheme. The I2C core can be configured with different hold times for the I2C bus lines. It helps to filter glitches in the I2C interface which can be caused by imperfect PCB wiring. The hold time is assigned by setting values to C_SCL_INERTIAL_DELAY and/or C_SDA_INERTIAL_DELAY parameters of the I2C core. Consequently, it delays the I2C signals internally by hold time. If the glitches are shorter than the hold time, they will be filtered. In this design 25 ns hold time is assigned to SCL and SDA lines to filter the glitches. Fig. 4.13 shows the configuration of the I2C core in EDK.

![Figure 4.13: The EDK GUI which is used to configure the I2C core](image)

### 4.5 Audio-acquisition Subsystem

In the audio-acquisition subsystem, the ADCs transfer audio data using I2S interface. The audio-acquisition subsystem is highlighted in Fig 4.14. The I2S protocol and different configurations of the I2S interface are described in Appendix B. In our design, all ADCs and the FPGA I2S receivers are operated in slave mode, which receive I2S clocks from FPGA-based I2S master clock module as shown in Fig. 4.15. The advantages of having a central FPGA-based I2S master clock module are,

- FPGA clock module can easily generate phase-locked I2S clock signals,
- It is easy to manage the clock centrally,
- FPGA clock resources can distribute the clock signals with low jitter and skew.
Figure 4.14: The highlighted section in the system is the audio-acquisition subsystem.

Figure 4.15: Generation of the master clock and I2S clocks for the ADC and FPGA I2S slave interfaces.

Fig. 4.16 shows the implemented I2S architecture on the FPGA. The I2S clock signals which are generated by the FPGA clock module are distributed to the I2S core and the ADCs. The ADCs are synchronized to the distributed master clock (MCLK). The FPGA I2S core buffers the serial data receiving from the I2S data lines to 24-bit registers. Note that the audio sample width is 24-bits. Since only a single ADC in a given chip produces a sample in a particular polarity of the WCLK, a single register is sufficient to buffer the data receiving...
from an ADC chip. The buffered data in the registers are copied to a sample buffer.

Figure 4.16: Overview of the data flow and the clock network in the audio-acquisition subsystem. The I2S clock signals which are generated by the FPGA clock module are distributed to the I2S core and the ADCs. The ADCs are synchronized to the distributed master clock (MCLK).

Now we discuss the frequency synthesizer in the master clock module which is implemented using Xilinx Mixed-Mode Clock Manager (MMCM) [157]. Xilinx MMCM is a voltage control oscillator (VCO) based clock synthesizer. The configuration related to I2S clock generation is shown in Fig. 4.17. The MMCM synthesizes the clock signals using a reference clock. It is a 100 MHz buffered clock signal in this system as shown in the figure. The clock input path has a programmable counter (D) which can be used to divide the input clock. The output of the D is fed to a phase-frequency detector (PFD) which compares phase and frequency of the rising edges of both the input (reference) and a feedback clock. Then the PFD generates a signal proportional to the phase and frequency between the two clocks. This signal drives a charge pump (CP) and a loop filter (LF) to generate a reference voltage to the VCO. The VCO produces eight output phases which can be selected when configuring the clock output. The counter M controls the feedback clock while the counter O divides the frequency at the output of the VCO, allowing a wide range of frequency synthesis. To minimize the clock skew, the generated clock signals are connected to the ADC
Figure 4.17: I2S clock synthesis by cascaded MMCM. The D, O and M are configurable clock dividers which are used for frequency synthesize. Note that CLKOUT6 and CLKOUT4 are cascaded to generate the 48 kHz WCLK.

motherboard via ODDR output registers on the FPGA.

Now we describe the mathematical expressions which govern the frequency synthesis. When calculating the counter values related to generation of the output clock signals, the VCO should be operated within a valid frequency range. For speed-grade (-1) Virtex-6 FPGAs, this range is 600 to 1200 MHz [156]. When the MMCM input clock signal is within the range of 10 to 700 MHz, the VCO is operated at a valid frequency. The relationship between the VCO and input frequencies can be expressed with D and M counter values such that,

$$F_{VCO} = F_{CLKIN} \times \frac{M}{D}.$$  \hspace{1cm} (4.1)

Then the output clock frequency can by derived using the O counter as

$$F_{OUT} = F_{CLKIN} \times \frac{M}{D \times O}.$$  \hspace{1cm} (4.2)

Regarding the configuration in Fig. 4.17 and the Eq. 4.1, the VCO operates at

$$F_{VCO} = 100 \text{ MHz} \times \frac{29}{4} = 725 \text{ MHz},$$

which is in the valid range. The generated MCLK and BCLK can be expressed using Eq. 4.2 such that,

$$F_{MCLK} = 100 \text{ MHz} \times \frac{29}{4 \times 59} = 12.288 \text{ MHz},$$

$$F_{BCLK} = 100 \text{ MHz} \times \frac{29}{4 \times 118} = 6.144 \text{ MHz}.$$
Although the counter values can be calculated by using Eq. 4.1 and Eq. 4.2, the Xilinx clock wizard automatically generate them when input and output frequencies are specified.

The generation of WCLK which is 48 kHz is impossible with the 100 MHz clock input, due to the applicable integer ranges to the divide counters. The valid ranges of the D, M and O dividers are [1,80], [5,64] and [1,128] respectively. Therefore, the minimum output frequency which can be generated by 100 MHz input is,

\[ F_{OUT,MIN} = 100 \text{ MHz} \times \frac{5}{80 \times 128} = 48.828 \text{ kHz} > 48 \text{ kHz} \]

To generate low-frequency WCLK while keeping a fixed-phase relationship with MCLK, the clock cascading technique is used. The clock cascading is a special configuration of the MMCM which allows CLKOUT6 O divider to be cascaded with the CLKOUT4 O divider. This increases the effective range of output clock division higher than 128. As shown in Fig. 4.17, 5.664 MHz intermediate clock is generated using the maximum O divider of CLKOUT6 (i.e., 128) and the result is passed through CLKOUT4 O divider (i.e., 118) which make the effective output division 128 × 118 and generates 48 kHz WCLK clock.

4.6 UDP/IP Data Transmission Subsystem

Now we describe the UDP/IP Data Transmission Subsystem. The UDP/IP Data Transmission Subsystem is highlighted in Fig 4.18. The supported transmission bandwidth is the most important parameter when choosing an interface for real-time audio transmission. The required transmission bandwidth of an audio system is determined by the number of microphones, the sampling rate of the microphones, sample width and compression techniques. The 64 microphones in our SMA are sampled at 48 kHz with 24-bit sample width, which requires 73.728 Mb/s bandwidth. Once the bandwidth of the transmission interface is sufficient, the data transmission distance is important. The higher distance between the SMA and the recording platform minimizes the noise and obstruct caused by the recording platform. The data transmission distance is limited by the attenuation of voltage levels of the interface signals due to impedance of the transmission medium. Some of the popular market standards for data transmission are compared in Table 4.1 [90]. We selected Gigabit Ethernet which enables long data transmission distance with sufficient bandwidth.

The data transmission over Ethernet is standardized by Open Systems Interconnection (OSI) model (see Fig. 4.19). The OSI model describes a standard protocol stack which can be followed to comply with standard networking devices such as switches and routers. In the OSI model, the physical layer explains the electrical and mechanical characteristics of the port, cable, network card and other physical aspects. ML605 platform has a Marvell M88E1111 chip which provides Ethernet physical-side interface (PHY). The onboard 1000
Figure 4.18: The highlighted section in the system is the UDP/IP data transmission subsystem.

<table>
<thead>
<tr>
<th>Standards</th>
<th>Bandwidth</th>
<th>Max. Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDMI 1.4</td>
<td>10.2 Gbps</td>
<td>15 m</td>
</tr>
<tr>
<td>USB 2.0</td>
<td>480 Mbps</td>
<td>5 m</td>
</tr>
<tr>
<td>FireWire 800</td>
<td>800 Mbps</td>
<td>4.5 m</td>
</tr>
<tr>
<td>Thunderbolt</td>
<td>10 Gbps</td>
<td>3 m</td>
</tr>
<tr>
<td>USB 3.0</td>
<td>5 Gbps</td>
<td>3 m</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>1 Gbps</td>
<td>100 m</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of different market standards for data transmission [90]

Base-T Ethernet port supports 1 Gbps data rate up to 100 m on four-pair category 5 (CAT5) shielded twisted-pair cable. This is a favorable length when transmitting audio data to a distant computer.

The data-link layer is responsible for the services of media access control (MAC), physical device addressing (i.e., MAC addresses of the FPGA and PC) and device-to-device delivery of frames. Fig. 4.20 shows the block diagram of the data-link and physical layer implementations. Xilinx Tri-mode EMAC (TEMAC) wrapper is used to implement the MAC protocol. MAC and PHY interfaces are independent from each other and they are interfaced using Media Independent Interface (MII). PHY device on ML605 platform (i.e., Marvell M88E1111 chip) supports Gigabit Media Independent Interface (GMII). Therefore,
MAC and PHY are interfaced using GMII interface. GMII can be generated by TEMAC wrapper by encapsulating MAC and GMII. Then the input interface of the wrapper communicates with the network layer while the output interface of the wrapper communicates with the physical layer. TEMAC wrapper handles data in bytes when communicating with the network and physical layers. It receives the data from the network layer via 8-bit FIFO and transmits the data to PHY via an 8-bit bus using serial transceivers. There are 8 serial transceivers to transmit the bytes in parallel. Each serial transceiver is driven by a 125 MHz clock which results $8 \times 125 \text{ MHz} = 1 \text{ Gbps}$ data rate. MMCM is used to generate the 125 MHz clock with low jitter.

We used UDP/IP stack for audio transmission over Ethernet. The open-source UDP/IP stack [41] is used to implement the IP and UDP layers (i.e., network and transport layers in OSI model) on the FPGA. UDP is a connectionless transport-layer protocol which implements on IP. There is no handshake or setup in UDP which is simpler and has less overhead. If the network connection is lossless point-to-point, UDP can provide light weight solution for audio transmission. The UDP/IP stack encapsulates the audio data into UDP decagrams which will be encapsulated into IPv4 packets. Then the IPv4 packets are transferred to MAC to generate Ethernet frames. The block diagram of the implementation

---

**Figure 4.19:** The OSI model for UDP/IP audio transmission over network.

<table>
<thead>
<tr>
<th>Application (audio data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport (UDP)</td>
</tr>
<tr>
<td>Network (IP)</td>
</tr>
<tr>
<td>Data Link (EMAC and GMII)</td>
</tr>
<tr>
<td>Physical (PHY)</td>
</tr>
</tbody>
</table>

**Figure 4.20:** The block diagram of the data-link and physical layer implementations for the Ethernet communication.
is shown in Fig. 4.21. The UDP/IP audio-transmission architecture is flexible and light

![Diagram of the FPGA-based UDP/IP audio transmission architecture.]

Figure 4.21: The FPGA-based UDP/IP audio transmission architecture.

weight. The resource utilization of the architecture on Virtex-6 (XC6VLX240T) FPGA is
given in Table 4.2. We used a finite state machine (FSM) to transmit the preprocessed data

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers</td>
<td>4,196 (1%)</td>
</tr>
<tr>
<td>Number of Slice LUTs</td>
<td>5,426 (3%)</td>
</tr>
<tr>
<td>Number of DSP48E1s</td>
<td>3 (1%)</td>
</tr>
<tr>
<td>Number of RAMB36E1s</td>
<td>18 (4%)</td>
</tr>
<tr>
<td>DDR3 Memory Usage</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.2: The resource utilization of the UDP/IP transmission architecture on Virtex-6 (XC6VLX240T) FPGA.

via the transmission subsystem.

4.7 DDR3-memory Subsystem

In this section, we describe the implementation of DDR3 memory subsystem which is used
to store the filter coefficients and read them when performing the spherical Fourier trans-
formation. The utilization of external memory spares the on-chip block memories. The
external memory subsystem is highlighted in Fig. 4.22. The filter coefficients are written
once to the DDR3 memory prior to the filtering and read them as required during the fil-
tering. We designed the DDR3-memory subsystem such that the filter coefficients can be
written to the DDR3 memory by a software application which runs on Microblaze processor. The filter coefficients can be emended without interrupting the filtering process. Even
though the writing of the coefficients is not time critical, reading should meet the timing
constraints of the real-time SFT.
Figure 4.22: The highlighted section in the system is the DDR3-memory subsystem.

Now we explain the implementation of the DDR3 memory subsystem. We used Xilinx Multi-Ported Memory Controller (MPMC) [147] to implement the DDR3 memory interfaces. The MPMC provides independent ports which can be configured as arbitrary interfaces to access the DDR3 memory. We configured 2 ports in an MPMC as a Processor Local Bus (PLB) interface and a Native Port Interface (NPI) to write and read the filter coefficients to and from the DDR3 memory. The MPMC can be configured using a GUI as shown in Fig 4.23. The PLB interface is used to connect Microblaze processor to the DDR3 memory via PLB bus. The NPI interface is a direct connect interface to the DDR3 memory instead of a connection to a shared bus like PLB. Because the interface is not shared, it does not require arbitration and allows low-latency copying of coefficients to the on-chip block memory. The block diagram of the DDR3 memory subsystem is shown in Fig. 4.24. The DDR3 memory subsystem is designed in EDK environment. Fig 4.25 shows the high-level design of the DDR3 memory subsystem in EDK. In the architecture, the DDR3 memory space is mapped to the Microblaze processor’s address space. NPI interface uses the same address space which is mapped to Microblaze to access the DDR3 memory. The NPI port is accessed by the SFT architecture to read the data. Therefore, NPI port is configured as an external port.

Now we explain the reading protocol of the filter coefficients via NPI port. The NPI data bus is 64-bit which burst 64 32-bit words in a single read. We implemented a finite state
Figure 4.23: The generation of PLB and NPI interfaces to the DDR3 memory using MPMC GUI.

Figure 4.24: The block diagram of the DDR3 memory subsystem. The DDR3 memory is shared between Microblaze processor and the SFT architecture via PLB and NPI interfaces respectively. These interfaces are generated as 2 ports on MPMC. The filter coefficients are written once to the DDR3 memory via PLB interface and read via NPI interface during the filtering.

machine to read data from DDR3 memory via NPI port. The burst data are written into block memories which can be accessed by the preprocessing subsystem. Fig. 4.26 presents how 16-bit coefficients can be written to the memory buffer. Since each burst is 64-bits, 2 complex coefficients can be written to the buffer in a single burst cycle. If there are 8 filter paths, 8 complex coefficients can be written in 4 burst cycles.
Figure 4.25: The integration of Microblaze processor and the MPMC in EDK.
Figure 4.26: Writing the filter coefficients to block memory while reading them from NPI port.
4.8 Spherical Fourier Transformation (SFT) Subsystem

In this section we describe the overview of the SFT subsystem. The SFT subsystem is highlighted in Fig 4.27. The SFT subsystem is implemented using Xilinx Integrated Software Environment (ISE) [150], Embedded Development Kit (EDK) [149] and System Generator for DSP (XSG) [153] software designing tools. The block diagram of the implemented architecture is shown in Fig. 4.28. This model file can be used as a framework for designing of different SFT architectures.

Figure 4.27: The highlighted section in the system is the spherical Fourier transformation (SFT) subsystem.
Figure 4.28: Overview of the 3rd-order 64 microphones SFT architecture. The architecture consists of 8 parallel computational data paths to achieve the required throughput. It is implemented using Xilinx ISE, EDK and System Generator for DSP tools. The highlighted section is the SFT subsystem which performs the preprocessing task. The complex data paths from FFT to IFFT are coloured in purple. The \( w \) is the memory word length which is 24-bit in this design.
Chapter 5

Implementation Model for FPGA-based Spherical Fourier Transformation (SFT)

5.1 Introduction

In the previous section, we described a stand-alone FPGA-based spherical Fourier transformation (SFT) system. As a preprocessing system, it can spare general computational resources for post-signal processing (i.e., sparse plane-wave decomposition). The configuration of the SFT system depends on the requirements such as the number of microphones and the order of the SFT. These requirements determine the number of ADCs, memory requirement, bandwidth requirement and computational resources. To enable fast deployment of an SFT system, we developed a modeling algorithm which determines the configuration of the SFT architecture based on the requirements. It takes the number of microphones and the order of the SFT as inputs and provides the configuration of the SFT architecture.

5.2 Implementation of the Spherical Fourier Transformation (SFT)

In this section, we describe the implementation of the spherical Fourier transformation algorithm for microphone-arrays. The spherical Fourier transformation is a beamforming process which can be illustrated by Eq. 5.1

\[
\begin{pmatrix}
    h_1 \\
    h_2 \\
    \vdots \\
    h_m
\end{pmatrix} =
\begin{pmatrix}
    e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\
    e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\
    \vdots & \vdots & \ddots & \vdots \\
    e_{m,1} & e_{m,2} & \cdots & e_{m,n}
\end{pmatrix} \ast
\begin{pmatrix}
    s_1 \\
    s_2 \\
    \vdots \\
    s_n
\end{pmatrix},
\]

where \( h \) represents SFT signals, \( s \) represents microphone signals and \( e \) represents FIR filters. The SFT requires \( n \times m \) convolutions of microphone signals and filters where \( n \) is the number
of microphones and \( m \) is the number of SFT signals. The number of SFT signals is equal to \((\Lambda + 1)^2\) where \( \Lambda \) is the highest order of SFT. The sample length of the microphone signals which are subjected to convolution is selected as same length as the FIR filters.

The SFT process is more computationally efficient in frequency domain than in the time domain. The time domain convolution has \( O(N^2) \) computational complexity where \( N \) is the length of convolving signals. If the computations of Eq. 5.1 are performed in the frequency domain, the time domain convolutions become multiplications and accumulation, which reduces the computational complexity to \( O(N) \) where \( N \) is the frequency-domain signal length. However, for doing this, time domain microphone signals and filters need to be transformed to frequency domain first by Fourier transformation and once filtering is completed the filtered signal need to be transformed back to the time domain by inverse Fourier transformation. Since filter coefficients are a predefined dataset, the transformation of filter coefficients to frequency domain does not effect on the computational complexity of the filtering process. However, transformation of microphone signals to the frequency domain and inverse transformation of the filtered signal to time domain increase the computational complexity. Fourier transformation can be performed efficiently as FFT (Fast Fourier Transform) which has \( O(N \log N) \) where \( N \) is the FFT length. Similarly, IFFT (Inverse FFT) has the same computational complexity, which is efficient in inverse Fourier transformation. Therefore by using FFT and IFFT techniques, the SFT can be performed more efficiently in the frequency domain than in the time domain.

To avoid the circular convolution in time domain during the filtering, the windowed microphone signal must be zero padded. Since the FFT length should be an integer power of 2, both window length and zero-padded length should be selected as identical and an integer power of 2. Therefore, FIR filter length should be half of the FFT length which is \( N \). Eq. 5.2 shows the mathematical expression for the frequency domain SFT.

\[
\begin{pmatrix}
\hat{h}_1 \\
\hat{h}_2 \\
\vdots \\
\hat{h}_m
\end{pmatrix}
= 
\begin{pmatrix}
\hat{e}_{1,1} & \hat{e}_{1,2} & \cdots & \hat{e}_{1,n} \\
\hat{e}_{2,1} & \hat{e}_{2,2} & \cdots & \hat{e}_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
\hat{e}_{m,1} & \hat{e}_{m,2} & \cdots & \hat{e}_{m,n}
\end{pmatrix}
\otimes
\begin{pmatrix}
\hat{s}_1 \\
\hat{s}_2 \\
\vdots \\
\hat{s}_n
\end{pmatrix}.
\]  

(5.2)

The symbol \( \otimes \) represents multiplications and additions of the corresponding frequency-domain samples of microphone signals and filters.

In the FPGA implementation, the microphone signals are first double buffered. Even though there are \( n \) microphones in the system, FFT is performed only \( u \) microphone signals in parallel to save resources, where \( \frac{n}{u} \in \mathbb{Z}^+ \). Therefore from the microphone-signal buffer,
$u$ signals are subjected to padding followed by FFT in parallel s.t.,

$$
\begin{pmatrix}
\hat{s}_1 \\
\hat{s}_2 \\
\vdots \\
\hat{s}_u
\end{pmatrix} = \text{FFT} \begin{pmatrix}
\text{zero-pad}(s_1) \\
\text{zero-pad}(s_2) \\
\vdots \\
\text{zero-pad}(s_u)
\end{pmatrix}.
$$

(5.3)

The FFT output is double buffered during the FFT process, which has Hermitian symmetry. Therefore, only $\frac{N}{2} + 1$ samples of an output need to be buffered where the rest can be calculated by the symmetric property. Similarly, frequency-domain filters also has Hermitian symmetry where, only $\frac{N}{2} + 1$ samples required to be stored/buffered.

The FFT of the microphone signal is multiplied by the FFT of the encoding filter. Filtering is the most computationally complex part of the SFT depending on the number of microphones and the order of the SFT. It requires a large number of multiplications and additions and full parallelism is not possible due to FPGA resource constraints. Since $u$ microphone signals are FFT and buffered, in the filtering stage, computations corresponding to the buffered $u$ FFT output are completed before the move to next $u$ FFT output. However, filtering of $u$ output at a time might not be possible due to resource constraints. Therefore, $p$ SFT signals are computed at a time by multiplying the filter coefficients with $q$ FFT outputs, where $\frac{m}{p} \in \mathbb{Z}^+$ and $\frac{u}{q} \in \mathbb{Z}^+$.

The computation of the $p$ SFT signals by multiplying the filter coefficients with $q$ FFT microphone signals are expressed in Eq. (5.4) s.t.,

$$
\begin{pmatrix}
\hat{h}_1 \\
\hat{h}_2 \\
\vdots \\
\hat{h}_p
\end{pmatrix} = \begin{pmatrix}
\hat{h}_1 \\
\hat{h}_2 \\
\vdots \\
\hat{h}_p
\end{pmatrix} \times \begin{pmatrix}
e_{1,1} & e_{1,2} & \cdots & e_{1,q} \\
e_{2,1} & e_{2,2} & \cdots & e_{2,q} \\
\vdots & \vdots & \ddots & \vdots \\
e_{p,1} & e_{p,2} & \cdots & e_{p,q}
\end{pmatrix} \times \begin{pmatrix}
\hat{s}_1 \\
\hat{s}_2 \\
\vdots \\
\hat{s}_q
\end{pmatrix}.
$$

(5.4)

$$
\begin{pmatrix}
\hat{h}_p+1 \\
\hat{h}_p+2 \\
\vdots \\
\hat{h}_{2p}
\end{pmatrix} = \begin{pmatrix}
\hat{h}_p+1 \\
\hat{h}_p+2 \\
\vdots \\
\hat{h}_{2p}
\end{pmatrix} \times \begin{pmatrix}
e_{p+1,1} & e_{p+1,2} & \cdots & e_{p+1,q} \\
e_{p+2,1} & e_{p+2,2} & \cdots & e_{p+2,q} \\
\vdots & \vdots & \ddots & \vdots \\
e_{2p+1,1} & e_{2p+1,2} & \cdots & e_{2p+1,q}
\end{pmatrix} \times \begin{pmatrix}
\hat{s}_1 \\
\hat{s}_2 \\
\vdots \\
\hat{s}_q
\end{pmatrix}.
$$

(5.5)

$$
\begin{pmatrix}
\hat{h}_{m-p+1} \\
\hat{h}_{m-p+2} \\
\vdots \\
\hat{h}_m
\end{pmatrix} = \begin{pmatrix}
\hat{h}_{m-p+1} \\
\hat{h}_{m-p+2} \\
\vdots \\
\hat{h}_m
\end{pmatrix} \times \begin{pmatrix}
e_{m-p+1,1} & e_{m-p+1,2} & \cdots & e_{m-p+1,q} \\
e_{m-p+2,1} & e_{m-p+2,2} & \cdots & e_{m-p+2,q} \\
\vdots & \vdots & \ddots & \vdots \\
e_{m,1} & e_{m,2} & \cdots & e_{m,q}
\end{pmatrix} \times \begin{pmatrix}
\hat{s}_1 \\
\hat{s}_2 \\
\vdots \\
\hat{s}_q
\end{pmatrix}.
$$

(5.6)

In the first stage corresponding to Eq. 5.4, SFT signals are partially calculated using $q$ FFT microphone signals. When more microphone signals are filtered in later stages as shown in Eq. 5.5 and Eq. 5.6, the partial results will be updated by addition. At a given time $p \times q$ filters are stored in the FPGA for filtering. During the filtering stage, $p$ SFT signals
are calculated in parallel using dedicated filters while individual SFT signal is calculated in streaming fashion. Once computation is completed, next \( p \) SFT signals are computed with different \( p \times q \) filters. This process repeat \( \frac{m}{p} \) times until all SFT signals are partially computed with \( q \) FFT microphone signals.

Once all SFT signals are partially computed as in Eq. (5.4–5.6), the next \( q \) FFT microphone signals are subjected to filtering and update the partial result of the SFT signals. The process is similar to Eq. (5.4–5.6). This must be repeated \( \frac{u}{q} \) times until all \( u \) FFT microphone signals are subjected to filtering. During this process, the partial calculation of the SFT signals is updated. Once \( u \) FFT microphone signals are filtered, the next \( u \) FFT microphone signals would be ready, which will be subjected to filtering in the same fashion. This will repeat \( \frac{m}{u} \) times until all SFT signals are calculated.

After completing the filtering, the buffered SFT signals are the frequency-domain SFT signals. They can be transformed back to time domain using an IFFT. Similar to the FFT process, the IFFT of SFT signals is also performed in stages to save resources. The IFFT process can be expressed as:

\[
\begin{bmatrix}
\hat{h}_1 \\
\hat{h}_2 \\
\vdots \\
\hat{h}_v
\end{bmatrix} = \text{IFFT}\left(\begin{bmatrix}
\tilde{h}_1 \\
\tilde{h}_2 \\
\vdots \\
\tilde{h}_v
\end{bmatrix}\right),
\]

where, \( v \) is number of SFT signals which are subjected to IFFT in parallel. Therefore, \( \frac{m}{v} \in \mathbb{Z}^+ \).

The final stage of the SFT process is overlap and add the IFFT output (i.e., \( \hat{h} \)) to eliminate the effect caused by zero padding at the beginning (see Eq. 5.3). Since half the length of microphone signal window is zero padded when performing the \( N \)-point FFT, \( \frac{N}{2} \) IFFT output samples are subjected to be overlapped and added with the previous \( \frac{N}{2} \) samples. The overlap and add process is performed in streaming fashion after the IFFT process which can be expressed as:

\[
\begin{bmatrix}
h_1 \\
h_2 \\
\vdots \\
h_v
\end{bmatrix} = \begin{bmatrix}
\hat{h}_1(t) + \hat{h}_1(t - \frac{N}{2} - 1) \\
\hat{h}_2(t) + \hat{h}_2(t - \frac{N}{2} - 1) \\
\vdots \\
\hat{h}_v(t) + \hat{h}_v(t - \frac{N}{2} - 1)
\end{bmatrix},
\]

Each output signal of the overlap and add stage is a time-domain SFT signal which is \( \frac{N}{2} \)-sample in size. The output of the overlap and add stage is double buffered followed by transmitted out.

We have created a scalable and resource-optimized model for implementing SFT on FPGAs. This is an arbitrary model which we believe it can provide reasonable resource-optimized architecture for SFT. A schematic of the model is shown in Fig. 5.1. The ar-
chitectural variables, parameters and constraints of the analytical model are shown in the
figure. There are \( n \) microphones and \( m \) SFT signals in the architecture. Each microphone
channel samples at \( F_s \) sampling frequency. The FFT is performed on microphone signals
using \( u \) \( N \)-point FFT modules. \( \frac{N}{F_s} \) sample are windowed and zero padded prior to FFT.
Therefore, the critical delay \( (T_c) \) allowed in the architecture is \( \frac{N}{F_s} \). The sample/data word
length is \( w \)-bit. The FFT output are filtered by \( p \) filters each having \( q \) parallel data paths.
The required SFT filter coefficients are copied from the external memory. The IFFT is per-
formed on filtered data using \( v \) \( N \)-point IFFT modules. To remove the zero-padding effect,
the IFFT output is overlapped and added with the previous signal. The buffer specifications
are specified in parenthesis as single(1)/double(2), real(1)/imaginary(2), buffer length
(in samples), sample width (in bits) and number of buffers respectively. To define a SFT
architecture for given microphones and SFT signals, \( N, u, p, q, v \) and \( w \) must be specified
with their respective FFT, filter, IFFT and overlap-add configurations.

Now we explain the method we used to identify the parameters used in the methodology
and schematic diagram. To define the SFT architecture,

- \( u \) : Number of FFTs,
- \( p \) : Number of Filters,
- \( q \) : Number of parallel data paths in a filter, and
- \( v \) : Number of IFFT and overlap-add data paths

need to be identified with their respective FFT, filter, IFFT and overlap-add configurations. Since several variables must be determined to define the resource optimized SFT
architecture, we formulated following optimization problem which

\[
\text{minimizes } \| \max \{ R_{FF}, R_{LUT}, R_{DSP}, R_{BRAM} \} \| \quad (5.9)
\]

with guaranteeing real-time SFT performance

\[
R_r = \frac{\text{utilized } r\text{-resource}}{\text{available } r\text{-resource}} = f(u, p, q, v)
\]

where \( n \) and \( m \) are number of microphones and SFT signals respectively. For a given re-
source, \( R \) represents the utilized resources as a ratio to the available resource on a FPGA.
We call this normalized resource utilization. The critical resource which constraints the
design would be the resource corresponding to the maximum normalized resource utilization
from FF, LUT, DSP, BRAM. The normalized resource utilization \( R \) is a function of
\( \{ u, p, q, v \} \) and decreasing of any parameter reduces the resource utilization while making
the computation more sequential which decreases the performance. If this continues, the performance will be degraded below the real-time performance which is not acceptable. Therefore in this optimization problem, we minimize \( u, p, q \) and \( v \) while satisfying the timing constraint. Once \( \{u, p, q, v\} \) is identified, the SFT architecture can be defined. The main benefits of this framework is that it can be used:

1. To identify the feasibility of implementation of a given SFT system on a selected FPGA.

2. To identify a resource optimized FPGA architecture for SFT.

3. To identify a cost-effective FPGA for a given SFT requirement.

4. To identify the bottleneck (resource or I/O bandwidth) of implementation of an SFT on a given FPGA.

5. To find the maximum number of supported microphones by a FPGA for a given SFT-order.

6. To find the highest supported SFT-order by a FPGA for given number of microphones.

Since the SFT algorithm is highly parameterizable, this framework makes the design process easy and fast facilitates the FPGA design process.
Figure 5.1: The schematic diagram of the SFT architecture. There are $n$ microphones, $m$ SFT signals, $u$ $N$-point FFT modules, $p$ filters, $q$ parallel data paths per filter and $v$ $N$-point IFFT modules. The sample/data word length is $w$-bit. The critical delay $T_c$ is $\frac{N}{F_s}$, where $F_s$ is the sampling rate. The buffer specifications are specified in parenthesis as single(1)/double(2), real(1)/imaginary(2), buffer length (in samples), sample width (in bits) and number of buffers respectively.
5.3 SFT Architecture

In this section, we present a design of different stages of the SFT architecture. The stages are

- Microphone data buffer
- FFT stage
- FFT output buffer
- SFT stage
- Filter buffer
- SFT coefficient buffer
- IFFT stage
- Overlap and add stage

In each stage, we evaluate the utilization of LUTs, FFs, DSPs and BRAMs which are the FPGA resources which make design constraints. At the end of this section, we present the complete SFT architecture and the combined resources utilization which will be useful to calculate the SFT design constraints.

5.3.1 Microphone Data Buffer

The first step of the SFT is to buffer the microphone data on FPGA block memory (BRAM). The SMA microphones are sampled at $F_s$ sampling rate which is the Nyquist frequency of audible bandwidth. The sample width is $w$-bits. Since there are $n$ microphones, the streaming rate of SMA audio data is $n \cdot F_s \cdot w$ bits per second. In our implementation $n$, $F_s$ and $w$ are 64, 48 kHz and 24-bit respectively.

The FFT is performed on the streaming microphone data. The FFT transform length is assumed to be $N$ which is an integer power of 2. To avoid the circular convolution in time domain during the filtering, the audio samples must be zero padded. Therefore, FFT is performed on $\frac{N}{2}$ sample window with an equivalent length of zero padding at the end.

Since there are $n$ microphones, $n \cdot \frac{N}{2}$ samples must be buffered for FFT. We implement double buffers to buffer the audio as it support higher throughput. Fig. 5.2 illustrates the implemented double buffers to buffer the microphone data. Even though all $n$ microphone signals are buffered, FFT is performed only on $u$ microphone signals where $\frac{n}{a} \in \mathbb{Z}^+$ (see Eq. 5.3). Using multiplexers, $u$ microphone signals are selected at a time.
5.3.2 Implementation of the FFT Process

The FFT is performed on the buffered microphone signals. An introduction to the FFT and its implementation is presented in the background chapter (see section 3.6.1). In the background section we explained 3 different FFT modules which are referred to as (a) radix-2 lite burst I/O module, (b) radix-2 burst I/O module and (c) radix-4 burst I/O module. In the model, \(u\) FFTs are operated in parallel where \(\frac{n}{u} \in \mathbb{Z}^+\). The parameter \(u\) must be calculated based on timing and resource utilization. Following sections present resource utilization and the performance of FFT on a FPGA.

5.3.2.1 Block RAM requirement of FFT implementation

Regarding the memory requirement, an FFT module contains complex-data I/O buffers and twiddle memories. These memories are constructed by the basic FPGA memory primitives which are 18k-bits in size. A memory primitive can be configured to different widths as required by the data width. In the FFT memories, real and imaginary data are stored in single memory word which can be accessed in a single clock cycle. Therefore, the memory width is \(2 \times 24\)-bit. The utilization of the block memory primitives depends on the size of the memory and configuration of the memory (i.e., minimum area, low power and fixed
primitive configurations). When memories are required to be implemented as separate blocks, each memory requires dedicated full 18k primitives.

Table 5.1 presents the utilization of block memory primitives in the 3 FFT configurations. In an FFT, there are random-access memories (RAMs) for I/O buffers and read-only memories (ROMs) for twiddle factors. Both types of memories are constructed using block memory primitives. Three FFT configurations discuss here are described in section 3.6.1.

Table 5.1: Utilization of 18k block memory primitives in the 3 FFT configurations.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Radix-2 lite</th>
<th>Radix-2</th>
<th>Radix-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM (I/O buffers)</td>
<td>$\lceil \frac{48N}{18k} \rceil u$</td>
<td>$\lceil \frac{48(\frac{N}{2})}{18k} - 2u \rceil$</td>
<td>$\lceil \frac{48(\frac{N}{4})}{18k} - 4u \rceil$</td>
</tr>
<tr>
<td>ROM (Twiddle factors)</td>
<td>$\lceil \frac{48(\frac{N}{2})}{18k} \rceil$</td>
<td>$\lceil \frac{48(\frac{N}{2})}{18k} \rceil$</td>
<td>$\lceil \frac{48(\frac{N}{4})}{18k} - 3 \rceil$</td>
</tr>
</tbody>
</table>

5.3.2.2 DSP block requirement of FFT implementation

Regarding the computational resources utilized in an FFT module, the FFT butterfly can be implemented by DSPs and Slices. Irrespective of the type of the resources, an FFT module can be operated at 300 MHz with a slight variation of the latency. However, when the complex multipliers are implemented by Slices, significantly more LUTs and FFs are consumed which is not desirable. Therefore, only DSP-based complex multipliers are used for implementing the FFT architecture. Since a multiplier requires more resources than adder/subtractor, 3-multiplier complex multiplication is resource optimized than the 4-multiplier option. When the multipliers are implemented using DSP blocks, the 3-multiplier option can save DSP blocks. Even though the 3-multiplier option saves DSPs, it requires slightly higher LUTs and FFs compared to the 4-multiplier option which is significant when implementing many complex multipliers. Therefore, appropriate configuration needs to be chosen based on the critical resources from DSPs and Slices of the entire design.

In radix-2 lite burst I/O module, the real and imaginary values of the FFT output are computed one after the other. Therefore, real multipliers and adders can be used in the radix-2 lite configuration. In resource optimized version, multipliers are implemented using DSP blocks. Since multiplication is done for 24-bit operands, each multiplier requires 2 DSP blocks. The adder/subtractor is implemented using Slice resources. Therefore, altogether 4 DSP blocks are required for radix-2 lite butterfly architecture. In contrast, high-speed version uses DSP blocks for adder/subtractor which requires additional 2 DSP blocks. It requires 6 DSP blocks in total.
In radix-2 burst I/O module, there are a complex multiplier, complex adder and complex subtracter in the butterfly. The resource optimized method uses 3-multiplier configuration of complex multiplier while adder and subtracter are implemented using Slice resources. Therefore, 6 DSP blocks are required for the resource optimized configuration. In contrast, in high-speed method complex multiplier is implemented in 4-multiplier configuration. Therefore, 8 DSP blocks are required for implementing the complex multiplier. In addition, the complex adder and subtracter consist of 2 DSP blocks in each making altogether 12 DSP blocks in the butterfly.

In radix-4 burst I/O method, there are 3 complex multipliers, 4 complex adders and 4 complex subtracters. In resource optimized version, complex multipliers are implemented in 3-multiplier configuration which requires 18 DSP blocks. All the complex adders and subtracters are implemented using Slice resources. In high-speed method, the complex multipliers are implemented in 4-multiplier configuration which requires 24 DSP blocks. Each complex adder/subtracter requires 2 DSP blocks which require 16 DSP blocks. Hence, the fast method requires 40 DSP blocks in total. As a summary, Table 5.2 shows the DSP-block utilization in 3 FFT configurations.

<table>
<thead>
<tr>
<th>FFT configuration</th>
<th>Optimization</th>
<th>DSP-blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix-2 lite</td>
<td>resource</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>6</td>
</tr>
<tr>
<td>Radix-2</td>
<td>resource</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>12</td>
</tr>
<tr>
<td>Radix-4</td>
<td>resource</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>40</td>
</tr>
</tbody>
</table>

5.3.2.3 Latencies of different FFT configurations

The latency (in clock cycles) of the FFT module is considered when evaluating the timing constraints of the SFT process. The computational delay (in time) of the FFT module can be expressed as:

$$FFT_{\text{delay}} = \frac{\text{Latency of the FFT}}{\text{FFT operating frequency}}.$$ 

(5.10)

In the background chapter we have explained the FFT butterfly architectures (see 3.6). The FFT latency depends on the radix of the butterfly for given FFT-length. In \(N\)-point FFT, there are \(\log_r(N) \cdot \left(\frac{N}{r}\right)\) repetitions of basic radix-\(r\) butterfly operations. In the expression, \(\log_r(N)\) is the number of stages and \(\left(\frac{N}{r}\right)\) is the basic butterflies per stage in the complete
butterfly. Table 5.3 shows an estimation of latencies for radix-2 lite, radix-2 and radix-4 FFT modules. In the formulas, the value $N$ at the beginning and end is the latency caused due to I/O buffering.

Table 5.3: An estimation of latencies associated with 3 FFT configurations. The latencies are presented in clock cycles.

<table>
<thead>
<tr>
<th>FFT configuration</th>
<th>Latency evaluation formula</th>
<th>Estimated latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix-2 lite</td>
<td>$N + 2 \left{ \log_2(N) \cdot \left( \frac{N}{2} \right) \right} + N$</td>
<td>5632</td>
</tr>
<tr>
<td>Radix-2</td>
<td>$N + \left{ \log_2(N) \cdot \left( \frac{N}{2} \right) \right} + N$</td>
<td>3328</td>
</tr>
<tr>
<td>Radix-4</td>
<td>$N + \left{ \log_4(N) \cdot \left( \frac{N}{4} \right) \right} + N$</td>
<td>1600</td>
</tr>
</tbody>
</table>

As per Table 5.3, when the radix increases, the latency decreases. The resources used in the butterfly may also slightly differ the latency which is insignificant. Table 5.4 shows the typical latencies of FFT configurations and their maximum operating frequencies [145]. As per the table, maximum operating frequency is same for all the configurations. Therefore, the delay of the FFT computation is directly proportional to the latency of the FFT configuration.

Table 5.4: Typical latencies of different FFT configurations and their maximum operating frequencies [145]. The latencies are presented in clock cycles.

<table>
<thead>
<tr>
<th>FFT configuration</th>
<th>Resource Optimized</th>
<th>Performance Optimized</th>
<th>Freq. (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix-2 lite</td>
<td>5664</td>
<td>5666</td>
<td>395</td>
</tr>
<tr>
<td>Radix-2</td>
<td>3505</td>
<td>3487</td>
<td>395</td>
</tr>
<tr>
<td>Radix-4</td>
<td>1770</td>
<td>1770</td>
<td>395</td>
</tr>
</tbody>
</table>

5.3.2.4 Summary FFT configurations

Table 5.5 presents a summary of study. These resource and performance parameters are used to analyze the resource and timing constraints of the SFT on FPGA.

5.3.3 FFT Output Buffer

The output of the FFT is buffered prior to filtering. The FFT output is complex and due to Hermitian symmetry, $\frac{N}{2} + 1$ output samples are buffered from each FFT output. Since there are $u$ FFTs, the size of the double buffer is $2\{2(\frac{N}{2} + 1) \cdot w \cdot u\}$ where $w$ is the data width.
Even though FFT is performed on \( u \) microphone signals, only \( q \) FFT output are subjected to filtering where \( \frac{u}{q} \in \mathbb{Z}^+ \) (see Eq. 5.4). Using multiplexers, \( q \) FFT output are selected at a stage. The filtering of \( u \) FFT output should be completed within the duration of performing FFT on \( u \) microphone samples. Due to double buffering the FFT output, the filtering process can be overlapped with FFT process. The two buffers can be accessed in different speeds as long as the timing constraint is satisfied.

### 5.3.4 Parallel SFT Filters

In the frequency domain, filtering is multiplied accumulation of the FFT output with frequency-domain filters, which is the most computationally expensive part of the SFT. Therefore, filtering process needs to be performed with sufficient parallelism to meet the timing constraint while minimizing the resource utilization.

In Eq. 5.4, the filtering process has been described. For clarity, Eq. 5.4 is given below which describes a stage of the filtering process.

\[
\begin{pmatrix}
\hat{h}_1 \\
\hat{h}_2 \\
\vdots \\
\hat{h}_p
\end{pmatrix}
= \begin{pmatrix}
\hat{h}_1 \\
\hat{h}_2 \\
\vdots \\
\hat{h}_p
\end{pmatrix}
+ \begin{pmatrix}
\tilde{e}_{1,1} & \tilde{e}_{1,2} & \cdots & \tilde{e}_{1,q} \\
\tilde{e}_{2,1} & \tilde{e}_{2,2} & \cdots & \tilde{e}_{2,q} \\
\vdots & \vdots & \ddots & \vdots \\
\tilde{e}_{p,1} & \tilde{e}_{p,2} & \cdots & \tilde{e}_{p,q}
\end{pmatrix}
\otimes
\begin{pmatrix}
\hat{s}_1 \\
\hat{s}_2 \\
\vdots \\
\hat{s}_q
\end{pmatrix}
\]

The filtering stage consists of \( p \) filters each having \( q \) parallel data path. Each filter corresponding to processing of a SFT signal in a given stage. Therefore, \( p \) SFT signals are processed in a given stage. Fig. 5.4 shows the proposed filter bank to perform the SFT. The input of the filter bank are FFT output and filter coefficients. During the operation, all filters process same FFT output with different filter coefficients. The output of the filters are double buffered and dedicated double buffers are maintained for each SFT signal.

To generate an SFT signal, it is required to filter all microphone channels. However, at a time only \( u \) FFT output are available where \( \frac{n}{u} \in \mathbb{Z}^+ \) (see Eq. 5.3). On the other hand,
from the available $u$ FFT output, only $q$ output are subjected to filtering where $\frac{u}{q} \in \mathbb{Z}^+$. This means to generate an SFT signal, a filter has to repeat its operation $\frac{u}{q}$ times.

Regarding the resource utilization of the SFT filter bank, each filter has $q$ complex multipliers and $q$ complex adders. In section 3.6.1.2 we explained a complex multiplier can be implemented in 2 methods as 3-real-multiplier or 4-real-multiplier. If the complex multipliers are implemented in DSPs, the 3-real-multiplier method is the resource optimized option for DSPs while 4-real-multiplier method is the performance optimized option. Alternately, the complex multipliers can be implemented with Slice resources to save more DSPs. Regarding implementation of the complex adder, the only DSP-based configuration is considered since its resource consumption and performance are in acceptable range.

The resource utilization of the complex multiplier and complex adder are shown in Table 5.6. Since operating frequency of all 3 options in complex multipliers are acceptable, the DSP-based 3-multiplier configuration and Slice-based configuration are considered for filter implementation (see highlighted columns in Table 5.6). The DSP-based 3-multiplier
configuration is optimized for low DSP utilization with low slice utilization. On the other hand, the slice-based configuration does not utilize DSPs but consumes many slices. The appropriate configuration should be selected based on the critical resource related to the selected FPGA.

Table 5.7 shows the resource utilization of the filter bank which consists of \( p \) filters each having \( q \) parallel data paths. The 2 filter configurations are corresponding to the 2 configurations of the complex multiplier which are highlighted in Table 5.6. As shown in the table, significant amount of FFs and LUTs can be saved by utilizing DSPs. Depending on the selected FPGA device, appropriate filter configuration should be applied.
Table 5.6: The resource utilization of the complex multiplier and complex adder. The highlighted configurations are considered in the analysis.

<table>
<thead>
<tr>
<th></th>
<th>Resource and Performance</th>
<th>Performance Optimize</th>
<th>Resource Optimize</th>
<th>Slice-based Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DSP-based Implementation</td>
<td>DSP-based Implementation</td>
<td>(No DSPs)</td>
</tr>
<tr>
<td>Complex Multiplier</td>
<td>LUT</td>
<td>41</td>
<td>47</td>
<td>2168</td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>96</td>
<td>228</td>
<td>2091</td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>8</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Latency</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Freq. (MHz)</td>
<td>590</td>
<td>306</td>
<td>240</td>
</tr>
<tr>
<td>Complex Adder</td>
<td>LUT</td>
<td>0</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>0</td>
<td>-</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>2</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Latency</td>
<td>2</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Freq. (MHz)</td>
<td>590</td>
<td>-</td>
<td>590</td>
</tr>
</tbody>
</table>

Table 5.7: Resource utilization of the filter bank which consists of $p$ filters each having $q$ parallel data paths. The 2 filter configurations are corresponding to the 2 configurations of the complex multiplier which are highlighted in Table 5.6.

<table>
<thead>
<tr>
<th></th>
<th>Resource</th>
<th>DSP-based Implementation</th>
<th>Slice-based Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUT</td>
<td>$p \cdot q(47 + 96) = 143pq$</td>
<td>$p \cdot q(2168 + 96) = 2264pq$</td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>$p \cdot q(228 + 74) = 302pq$</td>
<td>$p \cdot q(2091 + 74) = 2165pq$</td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>$p \cdot q(6 + 0) = 6pq$</td>
<td>$p \cdot q(0 + 0) = 0$</td>
</tr>
</tbody>
</table>

5.3.5 Filter Output Buffer

The output of a filter is an accumulative double buffer which adds up filter output and eventually generates the frequency domain SFT signal. Each SFT signal has a dedicated buffer. The filter output is in complex format and due to Hermitian symmetry, it is required to store only $\frac{N}{2} + 1$ samples per SFT signal. Therefore, the size of the buffer is $2\{2(\frac{N}{2} + 1) \cdot w \cdot m\}$ where, $w$ is 24-bit word length and $m$ is the number of SFT signals. The configuration of the buffer of a single filter output is shown in Fig. 5.5.

Since the number of SFT signals are comparatively less than the number of microphones, implementation of a double buffer to each SFT signal is feasible. At the end of the filtering stage, IFFT is performed on the buffered data which are the frequency domain SFT signal. The filter output buffers separate the FFT and filtering stages from the IFFT stage. Therefore, the FFT and filtering stages can be overlapped with the IFFT stage which allows the time constraints for each stage to be critical latency (i.e., $\frac{N}{2Fs}$) of the SFT. However, it will increase the total latency of the SFT architecture by twice the critical latency which is
Figure 5.5: The configuration of the accumulative double buffer of a filter output. If there are \( p \) filters and \( m \) SFT signals, each filter generates \( \frac{m}{p} \) SFT signals.

\[ \frac{N}{F_s}, \]

5.3.6 SFT Coefficient Buffer

In filtering, the filter coefficients need to be multiplied with the FFT output. These coefficients are pre-defined constants which can be stored on-chip or off-chip. The filters are in the frequency domain, and memory can be saved by using Hermitian symmetry. In S-FT, \( 2nm(\frac{N}{2} + 1)w \) bits of memory space is required to store the complex filter coefficients where, \( n \) is the number of microphones, \( m \) is the number of SFT signals and \( w \) is 24-bit word length. Table 5.8 shows the memory requirement to store the filter coefficients of 3\(^{rd}\) order SFT. The number of microphones is varied for analysis. For comparison, Table 5.9 presents available block memories in different device versions of Virtex-6 FPGA. In analysis,

<table>
<thead>
<tr>
<th>Number of microphones</th>
<th>Coefficients memory (Kbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>6168</td>
</tr>
<tr>
<td>64</td>
<td>12336</td>
</tr>
<tr>
<td>128</td>
<td>24672</td>
</tr>
<tr>
<td>256</td>
<td>49344</td>
</tr>
</tbody>
</table>
it can be seen that the required block memory size for coefficients is of similar magnitude compared to the available on-chip memory. Therefore to conserve block memories, filter coefficients are stored in off-chip memory. The coefficients are fetched appropriately to the FPGA when filtering is being performed.

Table 5.9: Available block memories in different device versions of Virtex-6 FPGA.

<table>
<thead>
<tr>
<th>FPGA device</th>
<th>Block Memory (Kbits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC6VLX75T</td>
<td>5,616</td>
</tr>
<tr>
<td>XC6VLX130T</td>
<td>9,504</td>
</tr>
<tr>
<td>XC6VLX195T</td>
<td>12,384</td>
</tr>
<tr>
<td>XC6VLX240T</td>
<td>14,976</td>
</tr>
<tr>
<td>XC6VLX365T</td>
<td>14,976</td>
</tr>
<tr>
<td>XC6VLX550T</td>
<td>22,752</td>
</tr>
<tr>
<td>XC6VLX760</td>
<td>25,920</td>
</tr>
<tr>
<td>XC6VSX315T</td>
<td>25,344</td>
</tr>
<tr>
<td>XC6VSX475T</td>
<td>38,304</td>
</tr>
<tr>
<td>XC6VHX250T</td>
<td>18,144</td>
</tr>
<tr>
<td>XC6VHX255T</td>
<td>18,576</td>
</tr>
<tr>
<td>XC6VHX380T</td>
<td>27,648</td>
</tr>
<tr>
<td>XC6VHX565T</td>
<td>32,832</td>
</tr>
</tbody>
</table>

When SFT coefficients are stored in off-chip memory, they must be accessed by the filter bank which consists of $p$ filters each having $q$ parallel data paths. To satisfy the timing constraints, the coefficients are double buffered for fast access. Consequently, $4\left(\left(\frac{N}{2} + 1\right) \cdot p \cdot q \cdot w\right)$ bits double buffer is implemented to access the filter coefficients from off-chip memory.

Fig. 5.6 illustrates an implementation of the coefficient double buffer for $p$ filters each having $q$ parallel data paths. Each data path requires unique coefficient set for filtering. Therefore, the output of the coefficient buffer should have $p \cdot q$ output ports. During the filtering, each port feeds the coefficients to its corresponding data path in the filter bank. Therefore, by the completion of filtering, the coefficients are copied in $\frac{n \cdot m}{p \cdot q}$ sequential stages to the double buffer from the DDR3 memory. Since the coefficients are accessed in multiple stages, the block memory can be conserved when implementing the double buffer.

5.3.7 IFFT of the SFT signals

Once filtering process is completed, $m$ frequency-domain SFT signals are generated. The IFFT should be performed on these channels to obtain the time-domain SFT signals. In
the model, we assumed there are \( v \) IFFT modules where \( \frac{m}{v} \in \mathbb{Z}^+ \). In practice, IFFT can be performed using the FFT architecture which we have already discussed. Mathematically, the fundamental equations of the DFT and IDFT can be presented s.t.,

\[
\text{DFT} : \quad X(k) = \sum_{i=0}^{N-1} x(i) \cdot e^{-j2\pi \frac{ki}{N}}, \quad 0 \leq k \leq N - 1 \tag{5.11}
\]

\[
\text{IDFT} : \quad x(i) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) \cdot e^{j2\pi \frac{ki}{N}}, \quad 0 \leq n \leq N - 1 \tag{5.12}
\]

where, \( N \) is the length of the FFT/IFFT and \( j = \sqrt{-1} \). \( X(k) \) is the frequency-domain sample of the FFT at \( k^{th} \) point and \( x(i) \) is the time-domain sample at \( i^{th} \) point. As per the equations, by changing the twiddle factors and scale down the output by \( \frac{1}{N} \) factor, the same FFT butterfly architecture can be used for IFFT calculation. In binary when \( N \) is an integer power of 2, the scale down is just shift right the binary data by \( \log_2^N \) bit positions. Regarding the twiddle factors, the FFT twiddle factors can be replaced by the IFFT twiddle.
factors. Therefore, as the IFFT configuration options in the scalable model, we consider the same FFT configurations. Table 5.5 shows the area and performance of the considered FFT configurations in this SFT model.

5.3.8 Overlap and Add the IFFT Output

The output of the IFFT is the time domain SFT signal. However, because of zero padding the FFT input, the IFFT output must be overlapped and added with the previous signal to construct the valid SFT signals. For clarity, Eq. 5.8 is given below which gives the mathematical expression for the overlap and add stage.

\[
\begin{pmatrix}
  \hat{h}_1 \\
  \hat{h}_2 \\
  \vdots \\
  \hat{h}_v
\end{pmatrix} = \begin{pmatrix}
  \hat{h}_1(t) + \hat{h}_1(t - \frac{N}{2} - 1) \\
  \hat{h}_2(t) + \hat{h}_2(t - \frac{N}{2} - 1) \\
  \vdots \\
  \hat{h}_v(t) + \hat{h}_v(t - \frac{N}{2} - 1)
\end{pmatrix}.
\]

The first half of the IFFT output needs to be added with the last half of the previous output in sequential order. Fig. 5.7 illustrates the overlap and add stage which constructs the valid SFT signal. The previous output signal is stored in delay buffers which is \( \frac{N}{2} \) samples each. Therefore, the size of the all delay buffers in overlap-add stage is \( m \cdot \frac{N}{2} \cdot w \) bits which saves half of the IFFT output in every SFT signal.

The overlap-add stage uses a dedicated real adder in each data path which requires all together \( v \) real adders. These adders can be implemented by DSPs or slices depending on

![Diagram of overlap and add stage](image-url)
the available resources on the selected FPGA. Table 5.10 shows the resource utilization of both DSP and slice-based configurations of the overlap-add stage.

Table 5.10: Resource utilization of the overlap and add stage which having $v$ data paths.

<table>
<thead>
<tr>
<th>Resource</th>
<th>DSP based</th>
<th>Slice based</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>0</td>
<td>96·$v$</td>
</tr>
<tr>
<td>FF</td>
<td>0</td>
<td>74·$v$</td>
</tr>
<tr>
<td>DSP</td>
<td>$v$</td>
<td>0</td>
</tr>
</tbody>
</table>

The output of the overlap-add stage is double buffered in SFT buffer which is transferred out via an appropriate interface. The interface should support the required bandwidth and distance requirements. The size of the SFT double buffer is $2 \cdot v \cdot \frac{N}{2} \cdot w$ bits.

5.4 Evaluate the architectures against the timing constraints

In this section, we formulate the timing constraints of the SFT architecture and present an algorithm to determine the unknown model parameters $u$, $p$, $q$ and $v$ which satisfy the real-time performance. Once $u$, $p$, $q$ and $v$ parameters are determined, the calculation of the resource utilization of the corresponding configurations is straightforward as we already discuss the implementation of the SFT architecture. This will be discussed in detail in next section.

The FFT and filtering process in SFT can be overlapped with IFFT process. Consequently, the critical delay of the both processes become same which is $\frac{N/2}{F_s}$ where $N$ is the FFT length and $F_s$ is the sampling rate. Further, the total delay of the architecture becomes twice the critical latency (i.e., $2T_c$) which is $\frac{N}{F_s}$. This can be graphically represented by Fig. 5.8.

![Figure 5.8: The timing constraints in the SFT architecture. The total delay of the architecture can be extended up to $2T_c$.](image)

During the FFT and filtering stage, the operations of (1) FFT the microphone data, (2) filtering the FFT output and (3) copying the SFT filter coefficients into the FPGA,
can be performed by overlapping. Each of these tasks spend $T_{fft}$, $T_{fliter}$ and $T_{coe}$ duration respectively. Then to meet the real-time performance, the timing constraint is

$$\max\{T_{fft}, T_{fliter}, T_{coe}\} < T_c = \frac{N/2}{F_s}.$$  \hfill (5.13)

Similarly, the timing constraint to complete the IFFT process is

$$T_{ifft} < T_c = \frac{N/2}{F_s}.$$  \hfill (5.14)

where, $T_{ifft}$ is the time spends for IFFT stage.

Based on the model, $T_{fft}$, $T_{fliter}$, $T_{coe}$ and $T_{ifft}$ can be formulated s.t.,

$$T_{fft} = \left(\frac{L_{fft}}{F_{fft}}\right)\left(\frac{n}{u}\right)$$  \hfill (5.15)

$$T_{filter} = \left(\frac{n}{q}\right)\left(\frac{m}{p}\right)\left(\frac{N}{2} + 1\right)\frac{F_{filter}}{F_{filter}}$$  \hfill (5.16)

$$T_{coe} = \frac{2nm(N/2 + 1)w}{B_{mem}}$$  \hfill (5.17)

$$T_{ifft} = \left(\frac{L_{ifft}}{F_{ifft}}\right)\left(\frac{m}{v}\right)$$  \hfill (5.18)

where

$L_x$ Latency of the $x$ where $x \in \{fft, filter, ifft\}$,  
$F_x$ Operating frequency of $x$ where $x \in \{fft, filter, ifft\}$,  
$B_{mem}$ External memory bandwidth,  
$u$ Number of FFTs,  
$p$ Number of Filters,  
$q$ Number of parallel data paths in a filter,  
$v$ Number of IFFT and overlap-add data paths,  
$N$ FFT length,  
$n$ Number of microphones,  
$m$ Number of SFT signals, and  
$w$ Sample/Data width.

The latencies and the corresponding operating frequencies of the FFT configurations were given in Table 5.4. The same are valid for IFFT configurations as well. Regarding the maximum operating frequency of the filter, it can be operated at the lesser operating frequency from the complex multiplier and adder in use. The maximum operating frequency of different configurations of complex multiplier and adder were given in Table 5.6. The external memory bandwidth $B_{mem}$ for accessing the SFT coefficients depends on the implemented memory controller. The memory controller features the bus width, operating frequency,
burst length and many more parameters \cite{147}. The memory bandwidth also depends on the data access pattern of the application. Table 5.11 shows some measured DDR3 memory bandwidth in applications implemented on Spartan-6, Virtex-6, Artix-7, Kintex-7 and Virtex-7 FPGAs.

Table 5.11: Measured DDR3 memory bandwidth in some FPGA-based applications.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>Interface</th>
<th>Physical interface</th>
<th>Max. throughput (Gbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan-6</td>
<td>NPI/2-port</td>
<td>16-bit 400MHz</td>
<td>9.88</td>
</tr>
<tr>
<td>Virtex-6</td>
<td>NPI/1-port</td>
<td>32-bit 400MHz</td>
<td>11.73</td>
</tr>
<tr>
<td>Artix-7</td>
<td>AXI VDMA</td>
<td>64-bit 800MHz</td>
<td>74.84</td>
</tr>
<tr>
<td>Kintex-7</td>
<td>AXI VDMA</td>
<td>64-bit 800MHz</td>
<td>74.84</td>
</tr>
<tr>
<td>Virtex-7</td>
<td>AXI VDMA</td>
<td>64-bit 800MHz</td>
<td>74.84</td>
</tr>
</tbody>
</table>

The unknown model parameters \(u, p, q\) and \(v\) in Eq. (5.15–5.18) can be identified with their respective FFT, filter and IFFT configurations using the brute-force Algorithm\cite{2}. The ratio of delay of the FFT and filter stages to the critical delay and the ratio of delay of the IFFT to the critical delay can be saved during the algorithm to have an estimation about achieved timing margins. After completion of the algorithm, the resource consumption of the architectures corresponding to the identified parameters can be calculated.
Algorithm 2 Identify the \(u\), \(p\), \(q\) and \(v\) parameter sets with their respective FFT, filter and IFFT configurations.

**Input:** \(n\): number of microphones, \(m\): number of SFT signals, \(N\): FFT length, \(w\): data width, \(T_c\): critical delay \((\frac{N}{2}F_s)\), \(L_{fft}\): latency of the FFT/IFFT module, \(F_{fft}\): operating frequency of the FFT/IFFT module, \(F_{filter}\): operating frequency of the filter, \(B_{mem}\): external memory bandwidth.

**Output:** (1) \(u\), \(p\), \(q\) and \(v\) parameter sets with their respective FFT, filter and IFFT configurations, (2) The ratio of delay of the FFT and filter stages to the critical delay, and the ratio of delay of the IFFT to the critical delay.

1. \textbf{for} \(u \leq n\) and \(\frac{u}{n} \in \mathbb{Z}^+\) \textbf{do}
2. \hspace{1em} \textbf{for} \(p \leq m\) and \(\frac{m}{p} \in \mathbb{Z}^+\) \textbf{do}
3. \hspace{2em} \textbf{for} \(q \leq u\) and \(\frac{u}{q} \in \mathbb{Z}^+\) \textbf{do}
4. \hspace{3em} \textbf{for} each FFT configuration \textbf{do}
5. \hspace{4em} \textbf{for} each filter configuration \textbf{do}
6. \hspace{5em} Calculate \(T' = \max\{T_{fft}, T_{filter}, T_{coe}\}\) by Eq. \(5.15–5.17\)
7. \hspace{5em} if \(T' < T_c\) then
8. \hspace{6em} save \(u\), \(p\) and \(q\) parameters
9. \hspace{6em} save FFT and Filter types
10. \hspace{6em} save \(T'/T_c\) ratio
11. \hspace{5em} \textbf{end if}
12. \hspace{4em} \textbf{end for}
13. \hspace{3em} \textbf{end for}
14. \hspace{2em} \textbf{end for}
15. \hspace{1em} \textbf{end for}
16. \textbf{end for}
17. \textbf{for} \(v \leq m\) and \(\frac{m}{v} \in \mathbb{Z}^+\) \textbf{do}
18. \hspace{1em} \textbf{for} each IFFT configuration \textbf{do}
19. \hspace{2em} Calculate \(T'' = T_{iff}\) by Eq. \(5.18\)
20. \hspace{2em} if \(T'' < T_c\) then
21. \hspace{3em} save \(v\) parameter
22. \hspace{3em} save IFFT type
23. \hspace{3em} save \(T''/T_c\) ratio
24. \hspace{2em} \textbf{end if}
25. \hspace{1em} \textbf{end for}
26. \textbf{end for}

5.5 Evaluate the resource consumption against the architectural parameters

In the previous section, we formulated the timing constraints of the SFT architecture and presented an algorithm to identify the unknown model parameters \(u\), \(p\), \(q\) and \(v\) which satisfy the real-time performance. Once \(u\), \(p\), \(q\) and \(v\) parameters with their respective FFT, filter and IFFT configurations are determined, the evaluation of the resource consumption of the corresponding architectures is straightforward using the model. Tables 5.12 and 5.13 are prepared using the model to evaluate FF, LUT, DSP and BRAM resource consumption.
by substituting $u, p, q$ and $v$ parameters of known FFT, filter and IFFT configurations.

Table 5.12: Arithmetic and logic resource utilization of the SFT architecture.

<table>
<thead>
<tr>
<th>SFT stages</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix-2 lite</td>
<td>resource</td>
<td>759·$u$</td>
<td>1095·$u$</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>674·$u$</td>
<td>1097·$u$</td>
</tr>
<tr>
<td>Radix-2</td>
<td>resource</td>
<td>1032·$u$</td>
<td>1469·$u$</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>821·$u$</td>
<td>1199·$u$</td>
</tr>
<tr>
<td>Radix-4</td>
<td>resource</td>
<td>2385·$u$</td>
<td>3291·$u$</td>
</tr>
<tr>
<td></td>
<td>speed</td>
<td>2385·$u$</td>
<td>3291·$u$</td>
</tr>
<tr>
<td>Filter</td>
<td>Slice based</td>
<td>2264·$p$·$q$</td>
<td>2165·$p$·$q$</td>
</tr>
<tr>
<td></td>
<td>DSP based</td>
<td>143·$p$·$q$</td>
<td>302·$p$·$q$</td>
</tr>
<tr>
<td>IFFT</td>
<td>Same as FFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overlap-Add</td>
<td>Slice based</td>
<td>96·$v$</td>
<td>74·$v$</td>
</tr>
<tr>
<td></td>
<td>DSP based</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.13: Block RAM utilization of the SFT architecture.

<table>
<thead>
<tr>
<th>SFT stage</th>
<th>BRAM (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microphone Input Buffer</td>
<td>$N \cdot w \cdot n$</td>
</tr>
<tr>
<td>FFT</td>
<td>Radix-2 lite</td>
</tr>
<tr>
<td></td>
<td>Radix-2</td>
</tr>
<tr>
<td></td>
<td>Radix-4</td>
</tr>
<tr>
<td></td>
<td>FFT Output Buffer</td>
</tr>
<tr>
<td>Filter Output Buffer</td>
<td>$4\left(\frac{N}{2}+1\right) \cdot w \cdot m$</td>
</tr>
<tr>
<td>SFT Coefficient Buffer</td>
<td>$4\left(\frac{N}{2}+1\right) \cdot w \cdot p \cdot q$</td>
</tr>
<tr>
<td>Overlap Add Buffer</td>
<td>$\frac{N}{2} \cdot w \cdot m$</td>
</tr>
<tr>
<td>SFT Output Buffer</td>
<td>$N \cdot w \cdot v$</td>
</tr>
</tbody>
</table>

The implementation of the SFT architecture requires supporting FPGA modules to perform I/O control, memory control, data communication, etc. The resource utilization to these modules is also accounted when evaluating the resource utilization of the entire system. These modules consume relatively low resources and not scale with the size of the SFT.

Regarding the FPGA modules presented in Table 5.14, the DDR3 memory controller is required if the SFT coefficients are stored in the external DDR3 memory. The Ethernet module is required for transmitting the SFT output from the FPGA. I2S interface is a
Table 5.14: Resource utilization of some additional modules of the SFT architecture.

<table>
<thead>
<tr>
<th>FPGA Module</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR3 memory controller</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparten-6</td>
<td>560</td>
<td>360</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Virtex-6</td>
<td>3600</td>
<td>2600</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7 Series</td>
<td>15164</td>
<td>10595</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ethernet</td>
<td>2080</td>
<td>1510</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Microblaze</td>
<td>716</td>
<td>299</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I2C</td>
<td>339</td>
<td>228</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12S</td>
<td>525</td>
<td>601</td>
<td>0</td>
<td>(36-1024)</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sparten-6</td>
<td>4220</td>
<td>2998</td>
<td>0</td>
<td>(36-1024)</td>
</tr>
<tr>
<td>Virtex-6</td>
<td>7260</td>
<td>5238</td>
<td>0</td>
<td>(36-1024)</td>
</tr>
<tr>
<td>7 Series</td>
<td>18824</td>
<td>13233</td>
<td>0</td>
<td>(36-1024)</td>
</tr>
</tbody>
</table>

A popular interface which can be used to acquire digital microphone data from ADCs. Microblaze is a low resource soft processor which can be implemented on Xilinx FPGAs. It can be used to configure external audio components/devices and perform general tasks where acceleration is not a concern. I2C is a popular interface commonly used to communicate and configure external components such as ADCs and micro-controllers.

Table 5.15 presents different FPGA devices with their costs and available resources. Once the full resource utilization is calculated, it is compared against Table 5.15 to select an appropriate FPGA device which can be used to implement the desired architecture.

Table 5.15: FPGA feature summary by device.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>FPGA Device</th>
<th>Cost (AU$)</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XC6SLX4</td>
<td>21</td>
<td>2400</td>
<td>4800</td>
<td>8</td>
<td>216</td>
</tr>
<tr>
<td>2</td>
<td>XC6SLX9</td>
<td>31</td>
<td>5720</td>
<td>11440</td>
<td>16</td>
<td>576</td>
</tr>
<tr>
<td>3</td>
<td>XC6SLX16</td>
<td>42</td>
<td>9112</td>
<td>18224</td>
<td>32</td>
<td>576</td>
</tr>
<tr>
<td>4</td>
<td>XC6SLX25</td>
<td>78</td>
<td>15032</td>
<td>30064</td>
<td>38</td>
<td>936</td>
</tr>
<tr>
<td>5</td>
<td>XC6SLX45</td>
<td>118</td>
<td>27288</td>
<td>54576</td>
<td>58</td>
<td>2088</td>
</tr>
<tr>
<td>6</td>
<td>XC6SLX75</td>
<td>179</td>
<td>46648</td>
<td>93296</td>
<td>132</td>
<td>3096</td>
</tr>
<tr>
<td>7</td>
<td>XC6SLX100</td>
<td>181</td>
<td>63288</td>
<td>126576</td>
<td>180</td>
<td>4824</td>
</tr>
<tr>
<td>8</td>
<td>XC6SLX150</td>
<td>298</td>
<td>92152</td>
<td>184304</td>
<td>180</td>
<td>4824</td>
</tr>
<tr>
<td>9</td>
<td>XC6SLX25T</td>
<td>83</td>
<td>15032</td>
<td>30064</td>
<td>38</td>
<td>936</td>
</tr>
<tr>
<td>10</td>
<td>XC6SLX45T</td>
<td>132</td>
<td>27288</td>
<td>54576</td>
<td>58</td>
<td>2088</td>
</tr>
<tr>
<td>11</td>
<td>XC6SLX75T</td>
<td>190</td>
<td>46648</td>
<td>93296</td>
<td>132</td>
<td>3096</td>
</tr>
<tr>
<td>12</td>
<td>XC6SLX100T</td>
<td>261</td>
<td>63288</td>
<td>126576</td>
<td>180</td>
<td>4824</td>
</tr>
<tr>
<td>13</td>
<td>XC6SLX150T</td>
<td>392</td>
<td>92152</td>
<td>184304</td>
<td>180</td>
<td>4824</td>
</tr>
<tr>
<td>14</td>
<td>XC6VLX75T</td>
<td>1085</td>
<td>46560</td>
<td>93120</td>
<td>288</td>
<td>5616</td>
</tr>
<tr>
<td>15</td>
<td>XC6VLX130T</td>
<td>2519</td>
<td>80000</td>
<td>160000</td>
<td>480</td>
<td>9504</td>
</tr>
<tr>
<td>16</td>
<td>XC6VLX195T</td>
<td>3424</td>
<td>124800</td>
<td>249600</td>
<td>640</td>
<td>12384</td>
</tr>
<tr>
<td>17</td>
<td>XC6VLX240T</td>
<td>4631</td>
<td>150720</td>
<td>301440</td>
<td>768</td>
<td>14976</td>
</tr>
<tr>
<td>18</td>
<td>XC6VLX365T</td>
<td>8118</td>
<td>227520</td>
<td>455040</td>
<td>576</td>
<td>14976</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Ref.</th>
<th>FPGA Device</th>
<th>Cost (AU$)</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>XC6VLX550T</td>
<td>8658</td>
<td>343680</td>
<td>687360</td>
<td>864</td>
<td>22752</td>
</tr>
<tr>
<td>20</td>
<td>XC6VLX760</td>
<td>31677</td>
<td>474240</td>
<td>948480</td>
<td>864</td>
<td>25920</td>
</tr>
<tr>
<td>21</td>
<td>XC6VSX315T</td>
<td>4325</td>
<td>196800</td>
<td>393600</td>
<td>1344</td>
<td>25920</td>
</tr>
<tr>
<td>22</td>
<td>XC6VSX475T</td>
<td>16244</td>
<td>297600</td>
<td>595200</td>
<td>2016</td>
<td>38304</td>
</tr>
<tr>
<td>23</td>
<td>XC6VHX250T</td>
<td>6456</td>
<td>157440</td>
<td>314880</td>
<td>576</td>
<td>18144</td>
</tr>
<tr>
<td>24</td>
<td>XC6VHX255T</td>
<td>7424</td>
<td>158400</td>
<td>316800</td>
<td>576</td>
<td>18576</td>
</tr>
<tr>
<td>25</td>
<td>XC6VHX380T</td>
<td>9278</td>
<td>239040</td>
<td>478080</td>
<td>864</td>
<td>27648</td>
</tr>
<tr>
<td>26</td>
<td>XC6VHX565T</td>
<td>11839</td>
<td>354240</td>
<td>708480</td>
<td>864</td>
<td>32832</td>
</tr>
<tr>
<td>27</td>
<td>XC7A15T</td>
<td>71</td>
<td>10400</td>
<td>20800</td>
<td>45</td>
<td>900</td>
</tr>
<tr>
<td>28</td>
<td>XC7A35T</td>
<td>114</td>
<td>20800</td>
<td>41600</td>
<td>90</td>
<td>1800</td>
</tr>
<tr>
<td>29</td>
<td>XC7A50T</td>
<td>120</td>
<td>32600</td>
<td>65200</td>
<td>120</td>
<td>2700</td>
</tr>
<tr>
<td>30</td>
<td>XC7A75T</td>
<td>259</td>
<td>47200</td>
<td>94400</td>
<td>180</td>
<td>3780</td>
</tr>
<tr>
<td>31</td>
<td>XC7A100T</td>
<td>284</td>
<td>63400</td>
<td>126800</td>
<td>240</td>
<td>4860</td>
</tr>
<tr>
<td>32</td>
<td>XC7A200T</td>
<td>394</td>
<td>134600</td>
<td>269200</td>
<td>740</td>
<td>13140</td>
</tr>
<tr>
<td>33</td>
<td>XC7K70T</td>
<td>260</td>
<td>41000</td>
<td>82000</td>
<td>240</td>
<td>4860</td>
</tr>
<tr>
<td>34</td>
<td>XC7u60T</td>
<td>511</td>
<td>101400</td>
<td>202800</td>
<td>600</td>
<td>11700</td>
</tr>
<tr>
<td>35</td>
<td>XC7K325T</td>
<td>2205</td>
<td>203800</td>
<td>407600</td>
<td>840</td>
<td>16020</td>
</tr>
<tr>
<td>36</td>
<td>XC7K355T</td>
<td>2611</td>
<td>222600</td>
<td>445200</td>
<td>1440</td>
<td>25740</td>
</tr>
<tr>
<td>37</td>
<td>XC7K410T</td>
<td>4407</td>
<td>254200</td>
<td>508400</td>
<td>1540</td>
<td>28620</td>
</tr>
<tr>
<td>38</td>
<td>XC7K420T</td>
<td>4688</td>
<td>260600</td>
<td>521200</td>
<td>1680</td>
<td>30060</td>
</tr>
<tr>
<td>39</td>
<td>XC7K480T</td>
<td>7879</td>
<td>298600</td>
<td>597200</td>
<td>1920</td>
<td>34380</td>
</tr>
<tr>
<td>40</td>
<td>XC7V585T</td>
<td>8661</td>
<td>364200</td>
<td>728400</td>
<td>1260</td>
<td>28620</td>
</tr>
<tr>
<td>41</td>
<td>XC7V2000T</td>
<td>35854</td>
<td>1221600</td>
<td>2443200</td>
<td>2160</td>
<td>46512</td>
</tr>
<tr>
<td>42</td>
<td>XC7VX330T</td>
<td>4219</td>
<td>204000</td>
<td>408000</td>
<td>1120</td>
<td>27000</td>
</tr>
<tr>
<td>43</td>
<td>XC7VX415T</td>
<td>5417</td>
<td>257600</td>
<td>515200</td>
<td>2160</td>
<td>31680</td>
</tr>
<tr>
<td>44</td>
<td>XC7VX485T</td>
<td>5879</td>
<td>303600</td>
<td>607200</td>
<td>2800</td>
<td>37080</td>
</tr>
<tr>
<td>45</td>
<td>XC7VX550T</td>
<td>9578</td>
<td>346400</td>
<td>692800</td>
<td>2880</td>
<td>42480</td>
</tr>
<tr>
<td>46</td>
<td>XC7VX690T</td>
<td>12797</td>
<td>433200</td>
<td>866400</td>
<td>3600</td>
<td>52920</td>
</tr>
<tr>
<td>47</td>
<td>XC7VX980T</td>
<td>16824</td>
<td>612000</td>
<td>1224000</td>
<td>3600</td>
<td>54000</td>
</tr>
<tr>
<td>48</td>
<td>XC7VX1140T</td>
<td>23277</td>
<td>712000</td>
<td>1424000</td>
<td>3360</td>
<td>67680</td>
</tr>
<tr>
<td>49</td>
<td>XC7VH580T</td>
<td>21994</td>
<td>362800</td>
<td>725600</td>
<td>1680</td>
<td>33840</td>
</tr>
<tr>
<td>50</td>
<td>XC7VH870T</td>
<td>31950</td>
<td>547600</td>
<td>1095200</td>
<td>2520</td>
<td>50760</td>
</tr>
</tbody>
</table>

5.6 Results and Discussion

We use the presented SFT model to find the configuration parameters of the SFT architecture having 8, 16, 32, 64, 128, 256, 512 and 1024 microphones. The number of microphones is selected as an integer power of 2 which are popular configurations of the microphone arrays. The number of SFT signals are 4, 9, 16, 25, 36, 49, 64, 81 and 100 which is a square of an integer and a number less than the number of microphones.
We verified the model by comparing the calculated resource utilization against the post-implementation results of selected architectures on Xilinx Artix and Kintex FPGAs. Table 5.16 presents the verification. The selected FPGAs are free to analysis on Xilinx Vivado Design Suite [158]. As per the calculated model error, our model predicts the resource utilization for reasonable accuracy. In the table, we also presented the time taken to post-implementation of the architectures. We compiled the FPGA designs using Vivado Design Suite on a laptop having Intel Core-i7-6600U CPU and 20 GB RAM. Four threads were allocated to synthesis and implementation of the design. During the implementation, we did not experience any shortage or congestion of routing resources on the FPGAs. We have presented maximum-operating frequency of different implementations which is inversely proportional to the critical-path delay of the architecture. If the FPGA is congested during the place and routing of the components, the critical-path delay increases, hence the maximum-operating frequency decreases. As per the presented maximum-operating frequencies of the implemented architectures, our design does not congest the FPGA. When analyzing the maximum-operating frequencies in Table 5.16, note that Kintex devices are intrinsically faster than Artix devices.

Table 5.16: Comparison of the calculated resource utilization against the post-implementation results of selected architectures on Xilinx Artix and Kintex FPGAs. The maximum operating frequency of the architecture and its implementation time are also given. Note, in the system, $n$ is the number of microphones, and $m$ is the number of SFT signals.

<table>
<thead>
<tr>
<th>System</th>
<th>Resource</th>
<th>Calculated Util. (%)</th>
<th>Actual Util. (%)</th>
<th>Model Error (%)</th>
<th>Max. Operating Freq. (MHz)</th>
<th>Implementation Time (Min:Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC7A50T</td>
<td>LUT</td>
<td>64</td>
<td>69</td>
<td>-5</td>
<td>124.8</td>
<td>7:48</td>
</tr>
<tr>
<td>$n = 32$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m = 25$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>39</td>
<td>27</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BRAM</td>
<td>53</td>
<td>51</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XC7A75T</td>
<td>LUT</td>
<td>53</td>
<td>57</td>
<td>-4</td>
<td>122.9</td>
<td>5:12</td>
</tr>
<tr>
<td>$n = 64$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m = 64$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>21</td>
<td>25</td>
<td>-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BRAM</td>
<td>78</td>
<td>80</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XC7A200T</td>
<td>LUT</td>
<td>29</td>
<td>20</td>
<td>9</td>
<td>121.0</td>
<td>6:15</td>
</tr>
<tr>
<td>$n = 256$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m = 100$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BRAM</td>
<td>50</td>
<td>35</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XC7K70T</td>
<td>LUT</td>
<td>61</td>
<td>65</td>
<td>-4</td>
<td>160.1</td>
<td>5:50</td>
</tr>
<tr>
<td>$n = 64$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m = 64$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>24</td>
<td>29</td>
<td>-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BRAM</td>
<td>61</td>
<td>63</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XC7K160T</td>
<td>LUT</td>
<td>38</td>
<td>26</td>
<td>12</td>
<td>172.7</td>
<td>5:12</td>
</tr>
<tr>
<td>$n = 256$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m = 100$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>17</td>
<td>12</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSP</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BRAM</td>
<td>56</td>
<td>40</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We generated Table 5.17 and Table 5.18 using the model and Algorithm 2. In Table 5.17, the configuration of the SFT architecture which can be implemented on each FPGA with highest number of microphones and SFT signals is presented. In Table 5.18, the cost effective SFT architectures for different number of microphones are presented. The cost is evaluated based on the price of the FPGA device. In Table 5.17 and Table 5.18, the column ‘FPGA’ is the selected FPGA devices which are given in Table 5.15. The price of each FPGA device is given in the column ‘AU$’. The columns $n$ and $m$ present the number of microphones and SFT signals respectively. The model parameters are given in $u$, $p$, $q$ and $v$ columns while the configurations of FFT, filter, IFFT and overlap-add are given in FFT, Fil, IFFT and add columns. The columns LUT, FF, DSP and BRAM present the resource utilization of each resource as a ratio to the available resource. $T_1$ column presents the ratio of delay of the FFT and filter stages to the critical delay while $T_2$ column presents the ratio of delay of the IFFT stage to the critical delay. Note,

$$T_1 = \frac{\max\{T_{ffft}, T_{fil}, T_{coe}\}}{T_c},$$

$$T_2 = \frac{T_{iffft}}{T_c}.\quad (5.20)$$

The column $\alpha$ presents the critical resource type (1:LUT, 2:FF, 3:DSP, 4:BRAM), $\beta$ presents the critical delay of the FFT and filtering stage (1:$T_{ffft}$, 2:$T_{fil}$, 3:$T_{mem}$) and $\gamma$ presents the constraining factor of the SFT architecture (i.e., R:Resource or S:Speed). Based on the results from the tables, we can observe:

1. The FPGA devices XC6SLX4, XC6SLX9 and XC7A15T cannot be used for SFT as their resources are not sufficient (compare with Table 5.15 and find the missing devices).

2. In most SFT architectures, the highest utilized resource is BRAMs. However, in low-cost small FPGAs (e.g., XC6SLX16, XC6SLX25, XC6SLX25T, XC7A35T, XC7A50T) flip-flops (FFs) becomes the highest utilized resource (see column $\alpha$ in Table 5.17).

3. The constraining factor of the SFT can be either resource or speed (see column $\gamma$ in Table 5.17). The architectures implemented on Virtex-6 FPGAs are limited by the memory bandwidth of the filter coefficient access (see column $\beta$ in Table 5.17). In contrast, the architectures implemented on Artix/Kintex/Virtex-7 FPGAs are limited by the available BRAM resources (see column $\alpha$ in Table 5.17). Since Virtex-7 FPGAs support higher memory bandwidth than Virtex-6 FPGAs (see Table 5.11), the SFT on Artix/Kintex/Virtex-7 is not limited by the memory bandwidth of the filter coefficient access.

116
4. The maximum number of microphones Virtex-6 devices can support is 64 while Artix/Kintex/Virtex-7 devices can support up to 256. As stated, Virtex-6 SFT architectures are I/O bound while Virtex-6 architectures are resource bound.

5. The maximum number of SFT signals Virtex-6 devices can support is 64 (7th-order) which is limited by the maximum number of supported microphones (i.e., 64). Note that the number of SFT signals should be less than the number of microphones which is limited due to the constraint of the memory bandwidth of the filter coefficient access. In contrast, Artix/Kintex/Virtex-7 devices can support up to 100 SFT signals (9th-order). Note that Artix/Kintex/Virtex-7 devices can support up to 256 microphones.

6. Even though FPGAs expand in a wide price range, SFT can be implemented cost effectively (see Table 5.18).
Table 5.17: The configuration of the resource optimized SFT architectures on the selected FPGAs (see Table 5.15) which consist of highest number of supporting microphones and SFT signals. Refer the footnotes for terms and abbreviations used in the table. Fig. 5.1 provides an overview of the SFT model, which has been referred by Algorithm to generate this table.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>AU$</th>
<th>n</th>
<th>m</th>
<th>p</th>
<th>q</th>
<th>v</th>
<th>FFT</th>
<th>Fil</th>
<th>IFFT</th>
<th>Add</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
<th>α</th>
<th>T₁</th>
<th>β</th>
<th>T₂</th>
<th>γ</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC6SLX16</td>
<td>42</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.88</td>
<td>0.41</td>
<td>0.25</td>
<td>0.71</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.02</td>
<td>R</td>
</tr>
<tr>
<td>XC6SLX25</td>
<td>78</td>
<td>16</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.54</td>
<td>0.25</td>
<td>0.21</td>
<td>0.98</td>
<td>4</td>
<td>0.08</td>
<td>1</td>
<td>0.08</td>
<td>R</td>
</tr>
<tr>
<td>XC6SLX48</td>
<td>118</td>
<td>32</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.30</td>
<td>0.14</td>
<td>0.14</td>
<td>0.66</td>
<td>4</td>
<td>0.19</td>
<td>2</td>
<td>0.13</td>
<td>R</td>
</tr>
<tr>
<td>XC6SLX75</td>
<td>179</td>
<td>64</td>
<td>64</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.22</td>
<td>0.10</td>
<td>0.06</td>
<td>0.96</td>
<td>4</td>
<td>0.89</td>
<td>3</td>
<td>0.34</td>
<td>R</td>
</tr>
<tr>
<td>XC6SLX100</td>
<td>181</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.16</td>
<td>0.08</td>
<td>0.04</td>
<td>0.61</td>
<td>4</td>
<td>0.89</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6SLX150</td>
<td>298</td>
<td>64</td>
<td>64</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.11</td>
<td>0.05</td>
<td>0.04</td>
<td>0.61</td>
<td>4</td>
<td>0.89</td>
<td>3</td>
<td>0.34</td>
<td>R</td>
</tr>
<tr>
<td>XC6SLX25T</td>
<td>83</td>
<td>16</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.54</td>
<td>0.25</td>
<td>0.21</td>
<td>0.98</td>
<td>4</td>
<td>0.08</td>
<td>1</td>
<td>0.08</td>
<td>R</td>
</tr>
<tr>
<td>XC6SLX45T</td>
<td>132</td>
<td>32</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.30</td>
<td>0.14</td>
<td>0.14</td>
<td>0.66</td>
<td>4</td>
<td>0.19</td>
<td>2</td>
<td>0.13</td>
<td>R</td>
</tr>
<tr>
<td>XC6SLX75T</td>
<td>190</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.22</td>
<td>0.10</td>
<td>0.06</td>
<td>0.96</td>
<td>4</td>
<td>0.89</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6SLX100T</td>
<td>261</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.16</td>
<td>0.08</td>
<td>0.04</td>
<td>0.61</td>
<td>4</td>
<td>0.89</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6SLX150T</td>
<td>392</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.11</td>
<td>0.05</td>
<td>0.04</td>
<td>0.61</td>
<td>4</td>
<td>0.89</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VLX75T</td>
<td>1085</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.29</td>
<td>0.13</td>
<td>0.03</td>
<td>0.53</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VLX130T</td>
<td>2519</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.17</td>
<td>0.07</td>
<td>0.02</td>
<td>0.31</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VLX195T</td>
<td>3424</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.11</td>
<td>0.05</td>
<td>0.01</td>
<td>0.24</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VLX240T</td>
<td>4632</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.09</td>
<td>0.04</td>
<td>0.01</td>
<td>0.20</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VLX365T</td>
<td>8118</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>0.03</td>
<td>0.01</td>
<td>0.20</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VLX550T</td>
<td>8658</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.04</td>
<td>0.02</td>
<td>0.01</td>
<td>0.13</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VLX760</td>
<td>31677</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
<td>0.11</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VSX315T</td>
<td>4325</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.07</td>
<td>0.03</td>
<td>0.01</td>
<td>0.12</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VSX475T</td>
<td>16242</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.04</td>
<td>0.02</td>
<td>0.00</td>
<td>0.08</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VHX250T</td>
<td>6457</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.08</td>
<td>0.04</td>
<td>0.01</td>
<td>0.16</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VHX255T</td>
<td>7424</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.08</td>
<td>0.04</td>
<td>0.01</td>
<td>0.16</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VHX380T</td>
<td>9278</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>0.02</td>
<td>0.01</td>
<td>0.11</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
<tr>
<td>XC6VHX565T</td>
<td>11839</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.04</td>
<td>0.02</td>
<td>0.01</td>
<td>0.09</td>
<td>4</td>
<td>0.75</td>
<td>3</td>
<td>0.34</td>
<td>S</td>
</tr>
</tbody>
</table>

* FPGA: FPGA devices in Table 5.15. AU$: price of the FPGAs in Australian Dollars (in Nov-2015).
* n: number of microphones, m: number of SFT signals, p: number of FFTs, q: number of parallel data paths in the filter, v: number of FFTs.
* FFT: configuration of the FFT, Fil: configuration of the filter, IFFT: configuration of the IFFT, Add: configuration of the adders in overlap-add. (refer Table 5.12)
* LUT, FF, DSP, BRAM are the resource utilization of each resource as a ratio to the available resource. (refer Tables 5.12, 5.13, and 5.15)
* α: ratio of delay of the FFT and filter stages to the critical delay, T₁: ratio of delay of the FFT stages to the critical delay (refer Fig. 5.1)
Table 5.17 – Continued from previous page

<table>
<thead>
<tr>
<th>FPGA</th>
<th>AU$</th>
<th>n</th>
<th>m</th>
<th>u</th>
<th>p</th>
<th>q</th>
<th>v</th>
<th>FFT</th>
<th>Fil</th>
<th>IFFT</th>
<th>Add</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
<th>α</th>
<th>T₁</th>
<th>β</th>
<th>T₂</th>
<th>γ</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC7A35T</td>
<td>114</td>
<td>32</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0.98</td>
<td>0.38</td>
<td>0.19</td>
<td>0.76</td>
<td>1</td>
<td>0.19</td>
<td>2</td>
<td>0.13</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>XC7A50T</td>
<td>120</td>
<td>32</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.69</td>
<td>0.27</td>
<td>0.07</td>
<td>0.51</td>
<td>1</td>
<td>0.19</td>
<td>2</td>
<td>0.15</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>XC7A75T</td>
<td>259</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.53</td>
<td>0.21</td>
<td>0.04</td>
<td>0.78</td>
<td>4</td>
<td>0.49</td>
<td>2</td>
<td>0.34</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>XC7A100T</td>
<td>284</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.39</td>
<td>0.16</td>
<td>0.03</td>
<td>0.61</td>
<td>4</td>
<td>0.49</td>
<td>2</td>
<td>0.34</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>XC7A200T</td>
<td>394</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.29</td>
<td>0.12</td>
<td>0.02</td>
<td>0.50</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7K70T</td>
<td>260</td>
<td>64</td>
<td>64</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.61</td>
<td>0.24</td>
<td>0.03</td>
<td>0.61</td>
<td>4</td>
<td>0.49</td>
<td>2</td>
<td>0.34</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>XC7K160T</td>
<td>511</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.38</td>
<td>0.17</td>
<td>0.02</td>
<td>0.56</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7K325T</td>
<td>2205</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.19</td>
<td>0.08</td>
<td>0.01</td>
<td>0.41</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7K355T</td>
<td>2611</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.17</td>
<td>0.08</td>
<td>0.01</td>
<td>0.26</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7K410T</td>
<td>4407</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.15</td>
<td>0.07</td>
<td>0.01</td>
<td>0.23</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7K420T</td>
<td>4688</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.15</td>
<td>0.06</td>
<td>0.01</td>
<td>0.22</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7K480T</td>
<td>7897</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.13</td>
<td>0.06</td>
<td>0.01</td>
<td>0.19</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7V585T</td>
<td>8661</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.11</td>
<td>0.05</td>
<td>0.01</td>
<td>0.23</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7V2000T</td>
<td>35854</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
<td>0.14</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VX330T</td>
<td>4219</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.19</td>
<td>0.08</td>
<td>0.01</td>
<td>0.24</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VX415T</td>
<td>5417</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.15</td>
<td>0.07</td>
<td>0.01</td>
<td>0.21</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VX485T</td>
<td>5879</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.13</td>
<td>0.06</td>
<td>0.01</td>
<td>0.18</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VX550T</td>
<td>9578</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.11</td>
<td>0.05</td>
<td>0.01</td>
<td>0.16</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VX690T</td>
<td>12797</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.09</td>
<td>0.04</td>
<td>0.01</td>
<td>0.12</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VX980T</td>
<td>16824</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>0.03</td>
<td>0.01</td>
<td>0.12</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VX1140T</td>
<td>23277</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.05</td>
<td>0.02</td>
<td>0.01</td>
<td>0.10</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VH580T</td>
<td>21994</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.11</td>
<td>0.05</td>
<td>0.01</td>
<td>0.19</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>XC7VH870T</td>
<td>31950</td>
<td>256</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.07</td>
<td>0.03</td>
<td>0.00</td>
<td>0.13</td>
<td>4</td>
<td>0.77</td>
<td>2</td>
<td>0.53</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

a FPGA: FPGA devices in Table 5.15. AU$: price of the FPGAs in Australian Dollars (in Nov-2015).
b n: number of microphones, m: number of SFT signals, u: number of FFTs, p: number of filters, q: number of parallel data paths in the filter, v: number of IFFT

c FFT: configuration of the FFT, Fil: configuration of the filter, IFFT: configuration of the IFFT, Add: configuration of the adders in overlap-add. (refer Table 5.12)
d LUT, FF, DSP, BRAM are the resource utilization of each resource as a ratio to the available resource. (refer Tables 5.12, 5.13 and 5.15)
e T₁: ratio of delay of the FFT and filter stages to the critical delay, T₂: ratio of delay of the IFFT stage to the critical delay (refer Fig. 6.1)
Table 5.18: The cost effective FPGA implementations of different SFT specifications. Refer the footnotes\textsuperscript{a-f} for terms and abbreviations used in the table. Fig. 5.1 provides an overview of the SFT model, which has been referred by Algorithm 2 to generate this table.

| $n$ | $m$ | FPGA    | AU$ | $u$ | $p$ | $q$ | $v$ | FFT | Fil | IFFT | Add | LUT | FF | DSP | BRAM | $\alpha$ | $T_1$ | $\beta$ | $T_2$ | $\gamma$ |
|-----|-----|---------|-----|-----|-----|-----|-----|-----|-----|------|-----|-----|-----|------|-------|------|-------|------|-------|
| 4   | 4   | XC6SLX16| 41.75 | 1   | 1   | 1   | 1   | 1   | 1    | 1    | 0.89| 0.41| 0.25| 0.71| 1    | 0.02| 1    | 0.02| R     |
| 9   | 9   | XC6SLX25| 77.83 | 1   | 1   | 1   | 1   | 1    | 1    | 1    | 0.54| 0.25| 0.21| 0.66| 4    | 0.05| 1    | 0.05| R     |
| 16  | 16  | XC6SLX25| 77.83 | 1   | 1   | 1   | 1   | 1    | 1    | 1    | 0.54| 0.25| 0.21| 0.98| 4    | 0.08| 1    | 0.08| R     |
| 32  | 25  | XC6SLX45| 118.18| 1   | 1   | 1   | 1   | 1    | 1    | 1    | 0.30| 0.14| 0.14| 0.66| 4    | 0.19| 2    | 0.13| R     |
| 64  | 64  | XC6SLX75| 178.82| 1   | 2   | 1   | 1   | 1    | 1    | 1    | 0.22| 0.10| 0.06| 0.96| 4    | 0.89| 3    | 0.34| R     |
| 128 | 100 | XC7A200T| 394.05| 1   | 4   | 1   | 1   | 1    | 1    | 1    | 0.22| 0.09| 0.01| 0.37| 4    | 0.77| 2    | 0.53| S     |
| 256 | 100 | XC7A200T| 394.05| 2   | 4   | 2   | 1   | 1    | 1    | 1    | 0.29| 0.13| 0.02| 0.50| 4    | 0.77| 2    | 0.53| S     |

\textsuperscript{a} FPGA: FPGA devices in Table 5.15, AU$: price of the FPGAs in Australian Dollars (in Nov-2015).
\textsuperscript{b} $n$: number of microphones, $m$: number of SFT signals, $u$: number of FFTs, $p$: number of filters, $q$: number of parallel data paths in the filter, $v$: number of IFFTs.
\textsuperscript{c} FFT: configuration of the FFT, Fil: configuration of the filter, IFFT: configuration of the IFFT, Add: configuration of the adders in overlap-add. (refer Table 5.12)
\textsuperscript{d} LUT, FF, DSP, BRAM are the resource utilization of each resource as a ratio to the available resource. (refer Tables 5.12, 5.13 and 5.15)
\textsuperscript{e} $T_1$: ratio of delay of the FFT and filter stages to the critical delay, $T_2$: ratio of delay of the IFFT stage to the critical delay (refer Fig. 5.1).
\textsuperscript{f} $\alpha$: critical resource type (1:LUT, 2:FF, 3:DSP, 4:BRAM), $\beta$: critical delay of the FFT and filtering stage (1:$T_{fft}$, 2:$T_{filter}$, 3:$T_{mem}$), $\gamma$: constraining factor (R:Resource, S:Speed).
5.7 Conclusions

This chapter presented the development of a scalable FPGA design model for SFT. The model considers the number of microphones, SFT signals and affordable cost of FPGA as the input and provides the design of resource optimize and cost-effective FPGA architecture as the output. Using the model we could identify:

1. The FPGA devices XC6SLX4, XC6SLX9 and XC7A15T cannot be used for SFT as their resources are not sufficient (compare with Table 5.15, and find the missing devices).

2. In most SFT architectures, the highest utilized resource is BRAMs. However, in low cost small FPGAs (e.g., XC6SLX16, XC6SLX25, XC6SLX25T, XC7A35T, XC7A50T) flip-flops (FFs) becomes the highest utilized resource (see column $\alpha$ in Table 5.17).

3. The constraining factor of the SFT can be either resource or speed (see column $\gamma$ in Table 5.17). The architectures implemented on Virtex-6 FPGAs are limited by the memory bandwidth of the filter coefficient access (see column $\beta$ in Table 5.17). In contrast, the architectures implemented on Artix/Kintex/Virtex-7 FPGAs are limited by the available BRAM resources (see column $\alpha$ in Table 5.17). Since Virtex-7 FPGAs support higher memory bandwidth than Virtex-6 FPGAs (see Table 5.11), the SFT on Artix/Kintex/Virtex-7 is not limited by the memory bandwidth of the filter coefficient access.

4. The maximum number of microphones Virtex-6 devices can support is 64 while Artix/Kintex/Virtex-7 devices can support up to 256. As stated, Virtex-6 SFT architectures are constrained by the I/O bandwidth while Virtex-6 architectures are constrained by the available resources.

5. The maximum number of SFT signals Virtex-6 devices can support is 64 ($7^{th}$-order) which is limited by the maximum number of supported microphones (i.e., 64). Note that the number of SFT signals should be less than the number of microphones which is limited due to the constraint of the memory bandwidth of the filter coefficient access. In contrast, Artix/Kintex/Virtex-7 devices can support up to 100 SFT signals ($9^{th}$-order). Note that Artix/Kintex/Virtex-7 devices can support up to 256 microphones.

6. Even though FPGAs expand in a wide price range, SFT can be implemented cost effectively (see Table 5.18).
The model can be used to determine the design parameters of the resource-optimized SFT architecture. Since the SFT algorithm is highly parameterizable, the model makes the design process easy and fast facilitates the FPGA design process.
Chapter 6

Analysis of the Performance of the Sparse Recovery on Multithreaded Platforms

6.1 Introduction

The super-resolution source-localization techniques are an emerging requirement for many applications, such as radar, speech signal processing, mobile communication and so on [27, 55, 137]. In the paper [46], a comparison of different source localization techniques is presented. Based on the results, separation of coherent sources and closely spaced sources is challenging. The performance of the widely used MVDR and MUSIC algorithms were compared when separating two sources located at 20° and 24° which are separated by 4° of resolution of the signals. Further, performance was compared when separating two independent narrowband signals with equal amplitude. As per results (see Fig. 6.1 and Fig. 6.2), the MUSIC algorithm is showing good performance when separating closely spaced signals compared to the MVDR. However, both algorithms are not impressive when separating coherent sources.

One good way to address the coherent-source localization is sparse-recovery technique [35]. However, the sparse-recovery technique is computationally intensive, hence difficult to use in real-time applications. An optimized implementation for MVDR beamforming on a GPU is presented in the paper [23]. Using that implementation as a benchmark, following analysis was done for the sparse-recovery technique to understand its relative complexity and suitability for GPU implementation. Let’s consider an $M$ element uniform linear array. The $m^{th}$ channel can be expressed as $x_m[\theta, n]$ where, at the time sample $n$, it is digitally focusing the direction having azimuth angle $\theta$. To simplify the notation, $\theta$ will be omitted.

$$z[n] = w^H[n] \sum_{l=0}^{M-L} x_l[n] , \quad (6.1)$$
The effect of signal to noise ratio and number of array elements on MUSIC
The effect of closely spaced signals on MUSIC algorithm with DOA is 200 and 240 i.e. DOA of 14°, 230°, 35°, and 55°. The performance of each independent signal has been evaluated.

The simulation has been carried out for four independent narrow band signals with a spectral beamwidth of 40° resolution of the signals. It is clear from Fig. 5(a), as the signals are very close, the MUSIC gets resolve the signal but with less spectral beamwidth and observes satisfactorily performance in upper power levels in output spectrum. Now, if the two closely spaced signals are coherent in nature, and the MUSIC shows poor performance. It requires the computation of the matrix inverse which can be expensive for large arrays.

Increasing SNR. Also, the three different values of the number of array elements are M1 = 10, M2 = 50, M3 = 100. Figure 4(b), shows that as the value of array elements increases, the spectral beam width becomes narrower.

\[ \hat{R}[n] = \hat{R}[n] + \frac{d}{L} \text{tr}\{\hat{R}[n]\} \]

\[ \hat{R}[n] = 1 \frac{1}{N_K N_L} \sum_{l=0}^{M-L} \sum_{n'=n-K}^{n+K} x'_l[n'] x'_l[n']^H \in \mathbb{C}^{L \times L} \]

The MVDR beamformer output z[n] in Eq. (6.1) is calculated by applying the \( w^H[n] \) weight set to the sum of all the subarrays \( x_t[n] \in x_m[n] \):

\[ x_t[n] = [x_t[n] + x_{t+1}[n] + \cdots + x_{t+L-1}[n]]^T \]

where, \( L \) is the length of the subarray. The \( w^H[n] \) weight set is calculated as in Eq. (6.2) where \( a \) is the steering vector for each direction. The step in Eq. (6.3) is performed for conditioning.
the intermediate sample covariance matrix $\tilde{R}[n]$, which is required for accurately inverting the covariance matrix $\hat{R}[n]$. Further, note that $\hat{R}[n]$ is a Hermitian positive semidefinite matrix similar to $\tilde{R}[n]$ with a small dimension of $L$ (i.e., the length of the subarray). Therefore, the weights can be calculated for all the directions using a batch inversion with all $\hat{R}[n]$. In Eq. 6.4, $N_L = M - L + 1$ is the number of subarrays and $N_K = 2K + 1$ is the number of temporal samples to perform averaging over, where $K$ represents the temporal averaging. In the paper [23], $L \in [1, M]$, $K \in \{0, 1, 2\}$, and $M \in \{8, 16, 32\}$ are considered.

Regarding the computational complexity of the MVDR beamforming, the most complex operation is the computation of $\tilde{R}[n]$ than $\hat{R}^{-1}[n]$, since $N_KN_L > L$. We can approximately calculate the computational complexity of solving Eq. 6.3 and 6.4:

$$O_R = O_mN_kN_L^2 + O_a(N_k + N_l - 2)L^2,$$  \hspace{1cm} (6.6)

where $O_m$ and $O_a$ are the required floating point operations for complex multiplication and addition respectively.

Now, we discuss the IRLS algorithm which performs the sparse plane-wave decomposition for source localization. The main computational bottleneck of the IRLS algorithm is the linear computation:

$$x^{(i+1)} \leftarrow W^{(i)} D^T \left( DW^{(i)} D^T + \lambda^{(i)} I \right)^{-1} h,$$  \hspace{1cm} (6.7)

which is repeated iteratively to update the plane-wave solution. The vector $x^{(i+1)}$ is the sparse intermediate result solved for $h$ spherical Fourier transform (SFT) signal. The $i$ and $(i + 1)$ terms express the update of the corresponding value from one iteration to another. Note that $D$ is the dictionary, $\lambda$ is the regularization parameter and $W$ is the diagonal matrix of weights (see Section 3.4.2). By analysing the Eq. 6.7 (we will present the detailed analysis in Section 6.2), we have identified the following which are related to MVDR beamforming.

1. Similar to MVDR beamforming, the matrix inversion in Eq. 6.7 is computationally less expensive as the symmetric positive definite $[DW^{(i)} D^T + \lambda^{(i)} I]$ matrix has the dimension equivalent to the length of $h \in h$. Therefore, in the sparse recovery process, the matrix inversion can be performed as a small matrix inversion batch process. In the paper [23], the batch inversion process is performed using highly optimized Nvidia’s Gauss Jordan based batch linear equation solver.

2. Once the matrix is inverted, the rest of the computations in both algorithms are matrix and vector multiplications which are highly efficient on the GPU,
3. Unlike MVDR beamforming, the sparse recovery does not require inversion of the data covariance matrix and the sources localization is performed by solving a single optimization problem iteratively. Even though the number of iterations is one of the leading factors for the computational complexity, practically IRLS algorithm converges fast [30]. Therefore, in super-resolution source localization, sparse recovery technique can be computationally efficient than MVDR beamforming [49]. Therefore, we assume a similar or better optimization can be achieved once sparse recovery is performed on a GPU/multithreaded-platform.

In the frequency-domain plane-wave decomposition, the sparse-recovery algorithm can be applied for each frequency or frequency band independently [133, 135]. Fig. 6.3 shows the data-flow diagram for the plane-wave decomposition problem. There are two options to accelerate the frequency domain sparse recovery calculations. The first option is to accelerate the individual IRLS computation given in Eq. 6.7. We will discuss some techniques to speed up an individual IRLS problem in the next chapter. The second option is to simultaneously solve each separate IRLS problem in parallel. Ideally, this should provide a speedup equal to the number of IRLS problems as they can be processed in parallel [54]. In this chapter, we explore parallelization of the IRLS problem using modern processors such as CPUs, GPUs, and multicore computers.

6.2 Computational Complexity of the IRLS Computation

In order to explore the parallelization of the IRLS algorithm, we first need to understand the computational complexity of the critical calculations. There are two fundamental calcula-
tions that are repeated on every iteration of the ILRS algorithm. These two calculations are updating the plane-wave solution and weights. The computational complexity associated with updating the weights is negligible compared to updating the solution. In this section, we explore the computational complexity involved in updating the plane-wave solution (refer to equation 6.7). We used Basic Linear Algebra Subprograms (BLAS) library [21] to implement the IRLS algorithm. In fact, most of the computing tools use BLAS library for linear computations (e.g., Matlab, LAPACK, etc.). When using the BLAS library, different routines are available for the same computation which depends on the properties of the data structures. For example when solving a system of linear equations \( AX = B \), the BLAS routines differ depends on the properties of the matrix \( A \) (e.g., singular, positive definite, symmetric, etc.) . The most efficient BLAS routines are required to explicitly specify using the properties of the computation.

The BLAS library consists of standard routines which provide building blocks for performing basic vector and matrix operations. There are 3 set of BLAS routines called as levels. The level-1 BLAS performs scalar, vector and vector-vector operations. The level-2 BLAS performs matrix-vector operations, and the level-3 BLAS performs matrix-matrix operations. The BLAS library is efficient, portable (i.e., it can be tuned for different platforms) and widely available. According to the naming convention of the BLAS routines, the first letter indicates the supporting precision of the routine where S: SINGLE-REAL, D: DOUBLE-REAL, C: SINGLE-COMPLEX and, Z: DOUBLE-COMPLEX. As an example, matrix-matrix multiplication routine \( xGEMM \) can be categorized as \( SGEMM \), \( CGEMM \), \( DGEMM \) and \( ZGEMM \).

Now we present the implementation of the IRLS computation using the BLAS routines. Regarding updating the plane-wave solution in Eq. 6.7, we can define the matrix \( M \) such that,

\[
M = [WD^T][DWD^T + \lambda I]^{-1}, \quad (6.8)
\]

which is called the dimixing matrix in our context. Then the solution to each iteration of the IRLS computation can be calculated by multiplying the dimixing matrix \( M \) with the SFT signal vector \( h \). Note that \( M \) is updated in each iteration of the IRLS computation. Since \( M \) is a real matrix, even though \( h \) is complex vector, the multiplication of \( M \cdot h \) can be performed as 2 real matrix-vector multiplications for real and imaginary values. In single precision, matrix-vector multiplication is performed by \( SGEMV \) BLAS routine. In the calculation of \( M \), it is required to calculate \( WD^T \) and \( (DWD^T + \lambda I)^{-1} \) matrices beforehand. The xSCAL BLAS routine is used repeatedly to calculate \( WD^T \), which scale each dictionary column (note that \( D^T \) is the transpose of the dictionary) by the corresponding diagonal
element s.t.,

\[
\begin{pmatrix}
  w_1 \\
  w_2 \\
  \vdots \\
  w_U 
\end{pmatrix}_{U \times U} \begin{pmatrix}
  d_{1,1} & d_{1,2} & \cdots & d_{1,m} \\
  d_{2,1} & d_{2,2} & \cdots & d_{2,m} \\
  \vdots & \vdots & \ddots & \vdots \\
  d_{U,1} & d_{U,2} & \cdots & d_{U,m}
\end{pmatrix}_{U \times m} \Rightarrow \begin{pmatrix}
  w_1 \otimes d_{1,1} & w_2 \otimes d_{1,2} & \cdots & w_U \otimes d_{1,m} \\
  w_2 \otimes d_{2,1} & w_2 \otimes d_{2,2} & \cdots & w_2 \otimes d_{2,m} \\
  \vdots & \vdots & \ddots & \vdots \\
  w_U \otimes d_{U,1} & w_U \otimes d_{U,2} & \cdots & w_U \otimes d_{U,m}
\end{pmatrix}_{U \times m}.
\]

In xSCAL BLAS routine, the scaling of a vector is performed s.t.,

\[ y = \alpha y . \] (6.10)

where, \( \alpha \) is the scalar and \( y \) is the vector. The calculation of \( DWD^T \) matrix is performed using xGEMM which is matrix-matrix multiplication operations:

\[ C = \alpha AB + \beta C , \] (6.11)

where \( \alpha \) is 1 and \( \beta \) is 0. As per the notation, \( A \equiv D, B \equiv WD^T \). Notice that at the time of calculating \( DWD^T \) matrix, \( WD^T \) is already computed by xSCAL routine. The computed \( DWD^T \) matrix is then regularized. Since \( DWD^T \) is calculated by multiplying the dictionary with its weighted transpose, it is a square matrix having dimension \( m \times m \).

It is regularized by updating the diagonal elements:

\[ r_i = r_i + \frac{k}{m} \sum_{i=1}^{m} r_i \Rightarrow r_i + \lambda , \] (6.12)

where \( r_i \) is the \( i^{th} \) diagonal element and \( k \) is a constant related to regularizing. Therefore, regularizing can be done by xSAXPY BLAS routine s.t.,

\[ y = \alpha x + y . \] (6.13)

where, \( x \) and \( y \) are vectors and \( \alpha \) is a scalar. When using the xSAXPY for regularizing, the diagonal of \( DWD^T \) matrix should be applied as the vector \( y \) and \( x \) is the unit vector corresponding to the diagonal of the unit matrix. The scaler \( \alpha \) is corresponding to the regularizing parameter \( \lambda \). Once \( WD^T \) and \((DWD^T + \lambda I)\) matrices are computed, the rest of the IRLS computation can be performed in three methods using BLAS.

The first method solves \( AX = B \) system of linear equations by LU decomposition followed by forward and backward substitution. The LU decomposition is performed by xGETRF and forward-backward substitution is performed by xGETRS. The combined operation of xGETRF and xGETRS is similar to xGESV routine which is specially developed for solving system of linear equations by LU decomposition. Therefore to find the demixing
matrix, Eq. 6.8 should be formulated s.t.,

\[ M = [WD^T][(DWD^T + \lambda I)^{-1}] \], \quad (6.14) 

\[ M(DWD^T + \lambda I) = WD^T, \quad (6.15) \]

\[ (DWD^T + \lambda I)^T M^T = (WD^T)^T \Rightarrow AX = B. \quad (6.16) \]

Then \( LU \) decomposition by xGETRF:

\[ LU \cdot M^T = (WD^T)^T, \quad (6.17) \]

and by forward and backward substitution by xGETRS:

forward substitution \( LM' = (WD^T)^T, \quad (6.18) \)

backward substitution \( UM^T = M'. \quad (6.19) \)

Consequently, the solution of xGETRS routine would be \( M^T \). The FLOP involved in \( LU \) decomposition method is presented in Algorithm 3. The FLOP count for each BLAS routine is calculated based on the LAPACK working note 18 [9].

**Algorithm 3** BLAS routines applied for LU decomposition based IRLS computation. The number of FLOPs for each BLAS routine is calculated based on the LAPACK working note 18 [9]. In notations, \( U \) is the resolution of the dictionary and \( m \) is the number of SFT signals.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D \in \mathbb{R}^{(\Lambda+1)^2 \times U} )</td>
<td>( x \in \mathbb{Z}^U )</td>
</tr>
<tr>
<td>( h \in \mathbb{Z}^{(\Lambda+1)^2} )</td>
<td></td>
</tr>
<tr>
<td>( W \in \mathbb{Z}^U )</td>
<td></td>
</tr>
<tr>
<td>( \lambda ) is the regularization factor</td>
<td></td>
</tr>
</tbody>
</table>

**Step of the IRLS algorithm**

<table>
<thead>
<tr>
<th>FLOPs in a single iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSCAL</strong> : ( WD^T )</td>
</tr>
<tr>
<td><strong>SGEMM</strong> : ( DWD^T )</td>
</tr>
<tr>
<td><strong>SAXPY</strong> : ( (DWD^T + \lambda I)^T )</td>
</tr>
<tr>
<td><strong>SPOTRF</strong> : ( LL^T = (DWD^T + \lambda I)^T )</td>
</tr>
<tr>
<td><strong>SPOTRS</strong> : ( LU \cdot M^T = (WD^T)^T )</td>
</tr>
<tr>
<td><strong>SGEMV</strong> : ( Re(x) = M \cdot Re(h) )</td>
</tr>
<tr>
<td><strong>SGEMV</strong> : ( Im(x) = M \cdot Im(h) )</td>
</tr>
</tbody>
</table>

In second method, xPOTRF and xPOTRS routines which perform Cholesky decomposition followed by forward and backward substitution are used. xPOTRF computes the
Cholesky factorization of the positive-definite matrix $(DWD^T + \lambda I)$. Then xPOTRS solves the system of linear equations $AX = B$ by forward and backward substitution using the Cholesky factorization computed by xPOTRF. The reformulation of the linear system in Eq. 6.16 for Cholesky decomposition is similar to LU decomposition s.t.,

$$(DWD^T + \lambda I)^T M^T = (WD^T)^T \Rightarrow AX = B.$$ 

Then Cholesky decomposition by xPOTRF:

$$LL^T \cdot M^T = (WD^T)^T,$$

and by forward and backward substitution by xPOTRS:

forward substitution

$$LM' = (WD^T)^T,$$

backward substitution

$$L^T M^T = M'.$$

The FLOP involved in Cholesky decomposition method is presented in Algorithm 4. The FLOP count for each BLAS routine is calculated based on the LAPACK working note 18 [9].

Algorithm 4 BLAS routines applied for Cholesky decomposition based IRLS computation. The number of FLOPs for each BLAS routine is calculated based on the LAPACK working note 18 [9]. In notations, $U$ is the resolution of the dictionary and $m$ is the number of SFT signals.

<table>
<thead>
<tr>
<th>Input</th>
<th>Step of the IRLS algorithm</th>
<th>FLOPs in a single iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \in \mathbb{R}^{(\Lambda+1)^2 \times U}$</td>
<td>SSCAL : $WD^T$</td>
<td>$mU$</td>
</tr>
<tr>
<td>$h \in \mathbb{Z}^{(\Lambda+1)^2}$</td>
<td>SGEMM : $DWD^T$</td>
<td>$2m^2U$</td>
</tr>
<tr>
<td>$W \in \mathbb{Z}^U$</td>
<td>SAXPY : $(DWD^T + \lambda I)^T$</td>
<td>$2U$</td>
</tr>
<tr>
<td>$\lambda$ is a regularization factor</td>
<td>SPOTRF : $LL^T = (DWD^T + \lambda I)^T$</td>
<td>$\frac{1}{2}m^3 + \frac{1}{2}m^2 + \frac{1}{6}m$</td>
</tr>
<tr>
<td>$x \in \mathbb{Z}^U$</td>
<td>SPOTRS : $LL^T \cdot M^T = (WD^T)^T$</td>
<td>$2m^2U$</td>
</tr>
<tr>
<td>$x \in \mathbb{Z}^U$</td>
<td>SGEMV : $Re(x) = M \cdot Re(h)$</td>
<td>$2mU$</td>
</tr>
<tr>
<td></td>
<td>SGEMV : $Im(x) = M \cdot Im(h)$</td>
<td>$2mU$</td>
</tr>
</tbody>
</table>
In 1-st and 2-nd methods, the forward and backward substitution technique is used to solve the system of linear equations. The forward and backward substitution is widely applied in solving triangular systems of linear equations by LU and Cholesky decompositions. Alternatively, by using the properties of Cholesky decomposition, the demixing matrix $\mathbf{M}$ can be calculated without applying forward and backward substitution routine s.t.,

$$
\mathbf{M} = \left[\mathbf{WD}^T\right]\left[\left(\mathbf{DWD}^T + \lambda\mathbf{I}\right)^{-1}\right], \quad (6.23)
$$

Cholesky decomposition

$$
\mathbf{M} = \left[\mathbf{WD}^T\right]\left[\left(\mathbf{LL}^T\right)^{-1}\right], \quad (6.24)
$$

$$
\mathbf{M} = \left[\mathbf{WD}^T\right]\left[\left(\mathbf{L}^T\right)^{-1}\mathbf{L}^{-1}\right], \quad (6.25)
$$

$$
\mathbf{M} = \left[\mathbf{WD}^T\right]\left[\left(\mathbf{L}^{-1}\right)^T\mathbf{L}^{-1}\right], \quad (6.26)
$$

where, Cholesky decomposition is performed by xPOTRF routine. However, it requires to inverse the lower triangular matrix $\mathbf{L}$ which is calculated by Cholesky decomposition. The inverse of $\mathbf{L}$ can be calculated s.t.,

$$
\mathbf{LL}^{-1} = \mathbf{I} \quad (6.27)
$$

where, $\mathbf{I}$ is the identity matrix. Since the inverse of a non-singular lower triangular matrix is lower triangular, the calculation of $\mathbf{L}^{-1}$ is straight forward. Therefore, as the third method, xZPOTRI routine can be used to compute the demixing matrix $\mathbf{M}$ which can be calculated after Cholesky decomposition by xPOTRF. In the calculation, $\mathbf{WD}^T$, $(\mathbf{L}^{-1})^T$ and $\mathbf{L}^{-1}$ matrices are subjected to multiply using xGEMM BLAS routine. The xGEMM should be performed 2 times to compute $\mathbf{M}$. The FLOP involved in this matrix-inverse method is presented in Algorithm 5. The FLOP count for each BLAS routine is calculated based on the LAPACK working note 18 [9].

Now we compare the computational complexities of the three methods. Table 6.1 presents the summary of the total floating-point operations performed in each method. We have formulated the computational complexity against the resolution of the dictionary and the number of SFT signals. As per the table, the Cholesky decomposition based forward and backward substitution method has the lowest asymptotic computational complexity. Therefore, Cholesky decomposition based solver is used to implement the IRLS algorithm. Further, if the number of SFT signals are being constant, the the computational complexity is linearly related to the resolution of the dictionary which is the higher dimension of the dictionary matrix.
Algorithm 5 BLAS routines applied for matrix inverse based IRLS computation. The number of FLOPs for each BLAS routine is calculated based on the LAPACK working note 18 [9]. In notations, $U$ is the resolution of the dictionary and $m$ is the number of SFT signals.

**Input**
- $D \in \mathbb{R}^{(\Lambda+1)^2 \times U}$
- $h \in \mathbb{Z}^{(\Lambda+1)^2}$
- $W \in \mathbb{Z}^{U}$
- $\lambda$ is a regularization factor

**Output**
- $x \in \mathbb{Z}^U$

**Step of the IRLS algorithm**

<table>
<thead>
<tr>
<th>Method</th>
<th>Computational Complexity (FLOPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAL</td>
<td>$mU$</td>
</tr>
<tr>
<td>SGEMM</td>
<td>$2m^2U$</td>
</tr>
<tr>
<td>SAXPY</td>
<td>$2U$</td>
</tr>
<tr>
<td>SPOTRF</td>
<td>$\frac{1}{3}m^3 + \frac{1}{2}m^2 + \frac{1}{6}m$</td>
</tr>
<tr>
<td>SPOTRI</td>
<td>$\frac{2}{3}m^3 + \frac{1}{2}m^2 + \frac{5}{6}m$</td>
</tr>
<tr>
<td>SGEMM</td>
<td>$2m^3$</td>
</tr>
<tr>
<td>SGEMV</td>
<td>$2mU$</td>
</tr>
<tr>
<td>SGEMV</td>
<td>$2mU$</td>
</tr>
</tbody>
</table>

**Table 6.1:** The Number of FLOPs involves in the three methods of BLAS routines for computing the IRLS. Computational complexity is formulated as a polynomial of the resolution of the dictionary and the number of SFT signals. In notations, $U$ is the resolution of the dictionary and $m$ is the number of SFT signals.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Method</th>
<th>Computational Complexity (FLOPs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>LU solver</td>
<td>$\frac{2}{3}m^3 + (4U - \frac{1}{2})m^2 + (5U + \frac{5}{6})m + 2U$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$U(4m^2 + 5m + 2) + (\frac{2}{3}m^3 - \frac{1}{2}m^2 + \frac{5}{6}m)$</td>
</tr>
<tr>
<td>4</td>
<td>Cholesky solver</td>
<td>$\frac{1}{3}m^3 + (4U + \frac{1}{2})m^2 + (5U + \frac{1}{6})m + 2U$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$U(4m^2 + 5m + 2) + (\frac{1}{3}m^3 + \frac{1}{2}m^2 + \frac{1}{6}m)$</td>
</tr>
<tr>
<td>5</td>
<td>Cholesky based inverse</td>
<td>$3m^3 + (4U + 1)m^2 + (5U + 1)m + 2U$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$U(4m^2 + 5m + 2) + (3m^3 + m^2 + m)$</td>
</tr>
</tbody>
</table>
6.3 Methodology

In the previous section, we identified Cholesky decomposition is the most computationally efficient way to implement updating the plane-wave solution. Therefore, Cholesky decomposition based solver is used to implement the IRLS algorithm. The implementation of the IRLS algorithm is done using C language with OpenMP [29] or CUDA [48]. OpenMP (Open Multi-Processing) is an open-source application programming interface (API) which supports multithreaded programming on parallel architectures. The sparse recovery on CMPs, Multiprocessors and Manycore devices are implemented in C with OpenMP. Similarly, CUDA (Compute Unified Device Architecture) is a parallel programming framework invented by Nvidia to program CUDA-enabled GPUs in C. The sparse recovery on Nvidia K40 GPU is implemented in CUDA. Following sections describe the implementation of the IRLS algorithm in OpenMP and CUDA.

6.3.1 Implementation of the IRLS algorithm using OpenMP

In the implementation of sparse recovery with OpenMP, an OpenMP thread is assigned to compute each IRLS problem. In OpenMP, it is possible to create threads equivalent to the number of IRLS problems. These threads are executed independently on the platform. Algorithm 6 describes the IRLS computation which solves multiple IRLS problems in parallel. The C-programme of the algorithm is presented in Appendix D.

Now we describe the thread affinity of an OpenMP programme. Thread affinity is an important parameter when assigning IRLS problems to a multi-threaded architecture. It determines in which pattern the OpenMP threads are bound to the hardware threads in cores [67]. For clarity, the thread affinities of scatter, compact and balanced are presented in Fig. 6.4. They are the most common thread affinities supported by many compilers. Compact affinity utilizes the hardware threads which are close to each other when progressively assigning software threads to hardware threads. Consequently, in compact affinity, hardware threads local to a processor/core are first utilized. If two threads share the same data in a cache, putting them near can give advantages. Scatter affinity utilizes the hardware threads on cores in round robin fashion. Therefore, the threads are evenly distributed among cores and suitable when adjacent threads are not sharing cache data and are required to exploit the memory bandwidth effectively. Balanced affinity is for using good aspects of compact and scatter affinities. It distributes the thread on all available cores evenly, while preserving thread locality. The thread affinity can be set by `export PHI_KMP_AFFINITY = scatter` for an OpenMP program compiled with Intel compiler [67]. In GNU-Linux systems, the available hardware-thread information can be
Algorithm 6  The IRLS algorithm which solves multiple IRLS problems in parallel [30].
The C-programme of the algorithm is presented in Appendix D.

Input
D ∈ N^{(Λ+1)^2×U} : dictionary
h ∈ C^{(Λ+1)^2} : SFT signal
p ∈ (0,1] : norm used for sparse recovery
K ∈ Z^+ : a constant depends on the expected sparsity of x
ε : weighting parameter
n : number of iterations
n : maximum number of iterations allowed
N_p : Number of parallel threads

Output
x ∈ C^U : the recovered sparse signal

Algorithm

n ← 0
W^{(0)} ← diag ([1,1,...,1])
ε^{(0)} ← 1
parfor (1 : N_p) do
  while (n ≤ n) do
    W^{(n)} ← diag ([w_1^{(n)}, w_2^{(n)}, ..., w_U^{(n)}])
    x^{(n+1)} ← W^{(n)} D^T (D W^{(n)} D^T + λ^{(n)} I)^{-1} h
    ε^{(n+1)} ← min (ε^{(n)}, \frac{r(x^{(n+1)})}{K})
    w_i^{(n+1)} ← (x_i^{(n+1)})^2 + (ε^{(n+1)})^2
    n ← n + 1
  end while
end parfor

Notes
r(x) is a function that returns the components of x sorted in descending order according to the absolute values of the entries in x. Hence, r(x)_i is the i-th largest element of the set \{|x_j|, j = 1,2,...,U\}. λ is the regularization parameter calculated as in Eq. 3.18.

retrieved from the CLI command `numactl -H`. This information can be used to set the thread affinity by `export GOMP_CPU_AFFINITY = '...the choice of thread pattern...'` for an OpenMP program compiled with GNU compiler. Since different IRLS problems process different data sets, no advantage can be realized by solving IRLS problems using closer threads. In fact, utilization of closer threads may adversely affect on thread performance due to load imbalance across the threads. Therefore regarding the thread assignment, scatter affinity is desirable for sparse recovery. We used scatter affinity to bind the IRLS problems to hardware-threads.
6.3.2 Implementation of the IRLS algorithm using CUDA

Now we describe the implementation of IRLS algorithm on the GPU. In our implementation, the shared memory in the streaming multiprocessor (SM) becomes the critical resource to determine the number of active thread blocks. There are only 64 KB per SM in Nvidia K40 GPU \[96\]. Since each IRLS problem has to process private data, the available shared memory is insufficient to implement a single kernel to solve multiple IRLS problems. If the thread cannot use the shared memory efficiently, then the performance of the thread would affect by memory latency. Therefore, an IRLS problem is solved by partitioning the problem into subproblems and solve them using a thread block rather a single thread. Using a grid in CUDA, it is possible to launch many thread blocks which is equivalent to the number of IRLS problems. The different thread blocks can be executed independently on the GPU. Algorithm 7 presents the implemented multi-kernel IRLS algorithm. The algorithm is performed as a sequence of several kernels which are executed with barrier synchronization. Each kernel performs a particular step of the IRLS algorithm. Remarks that the kernels are developed such that each IRLS problem is mapped to a unique thread block which performs the computations independently. The CUDA-programme of the algorithm is presented in Appendix E.
Algorithm 7 Multi-kernel IRLS algorithm which solves multiple IRLS problems in parallel [30]. The CUDA-programme of the algorithm is presented in Appendix E.

**Input**

- $D \in \mathbb{N}^{(\Lambda+1)^2 \times U}$: dictionary is stored in the constant memory
- $h \in \mathbb{C}^{(\Lambda+1)^2}$: SFT signal is stored in device memory
- $p \in (0, 1]$: norm used for sparse recovery
- $K \in \mathbb{Z}^+$: a constant depends on the expected sparsity of $x$
- $\epsilon$: weighting parameter
- $n$: number of iterations
- $\hat{n}$: maximum number of iterations allowed

**Output**

- $x \in \mathbb{C}^U$: the recovered sparse signal

**Algorithm**

\[
\begin{align*}
n &\leftarrow 0 \\
W^{(0)}_{\text{dev}} &\leftarrow \text{diag}([1, 1, \ldots, 1]) \\
\epsilon^{(0)} &\leftarrow 1 \\
\text{Kernel:1} &\\
W^{(n)}_{\text{sm}} &\leftarrow W^{(n)}_{\text{dev}} \\
\Gamma^{(n+1)}_{\text{sm}} &\leftarrow W^{(n)}_{\text{sm}} D^T \\
\Gamma^{(n+1)}_{\text{dev}} &\leftarrow \Gamma^{(n+1)}_{\text{sm}} \\
\Upsilon^{(n+1)}_{\text{sm}} &\leftarrow D \Gamma^{(n+1)}_{\text{sm}} \\
\Upsilon^{(n+1)}_{\text{dev}} &\leftarrow \text{do vectorDot \ (rows (D), columns (\Gamma^{(n+1)}_{\text{sm}}))} \\
\Upsilon^{(n+1)}_{\text{sm}} &\text{is subject to regularization} \\
\Upsilon^{(n+1)}_{\text{dev}} &\leftarrow \text{Gauss-Jordan matrix inverse (\Gamma^{(n+1)}_{\text{sm}})} \\
\text{Kernel:2} &\\
\Upsilon^{(n+1)}_{\text{sm}} &\leftarrow \Upsilon^{(n+1)}_{\text{dev}} \\
\Psi^{(n+1)}_{\text{sm}} &\leftarrow (\Upsilon^{(n+1)}_{\text{sm}})^{-1} \\
\Psi^{(n+1)}_{\text{dev}} &\leftarrow \text{Gauss-Jordan matrix inverse (\Upsilon^{(n+1)}_{\text{sm}})} \\
\text{Kernel:3} &\\
\Gamma^{(n+1)}_{\text{sm}} &\leftarrow \Gamma^{(n+1)}_{\text{dev}} \\
\Psi^{(n+1)}_{\text{sm}} &\leftarrow \Psi^{(n+1)}_{\text{dev}} \\
h^{(n+1)}_{\text{sm}} &\leftarrow h \\
M^{(n+1)}_{\text{sm}} &\leftarrow \Gamma^{(n+1)}_{\text{sm}} \Psi^{(n+1)}_{\text{sm}} \\
\text{Re} \left( x^{(n+1)}_{\text{sm}} \right) &\leftarrow M^{(n+1)}_{\text{sm}} \text{Re} (h^{(n+1)}_{\text{sm}}) \\
\text{Im} \left( x^{(n+1)}_{\text{sm}} \right) &\leftarrow M^{(n+1)}_{\text{sm}} \text{Im} (h^{(n+1)}_{\text{sm}}) \\
\text{Re} \left( x^{(n+1)}_{\text{dev}} \right) &\leftarrow \text{Re} \left( x^{(n+1)}_{\text{sm}} \right) \\
\text{Im} \left( x^{(n+1)}_{\text{dev}} \right) &\leftarrow \text{Im} \left( x^{(n+1)}_{\text{sm}} \right) \\
\text{Kernel:4} &\\
sq\_abs \left( x^{(n+1)}_{\text{dev}} \right) &\leftarrow \text{perform as vector operations} \\
&\left( \text{Re} \left( x^{(n+1)}_{\text{dev}} \right) \cdot \text{Re} \left( x^{(n+1)}_{\text{dev}} \right) + \text{Im} \left( x^{(n+1)}_{\text{dev}} \right) \cdot \text{Im} \left( x^{(n+1)}_{\text{dev}} \right) \right)
\end{align*}
\]
Algorithm 7 Multi-kernel IRLS algorithm which solves multiple IRLS problems in parallel (continued).

Kernel: 5

\[
\text{sorted sq.abs } \mathbf{x}_{\text{dev}}^{(n+1)} \leftarrow \text{sort.descending}\left(\text{sq.abs } \mathbf{x}_{\text{dev}}^{(n+1)}\right)
\]

Kernel: 6

\[
\kappa_{\text{dev}}^{(n+1)} \leftarrow \frac{r\left(\mathbf{x}_{\text{dev}}^{(n+1)}\right)_K}{U^2} \quad \Rightarrow \quad \text{k}^{\text{th}} \text{ element of } \text{sorted sq.abs } \mathbf{x}_{\text{dev}}^{(n+1)}
\]

\[
\epsilon_{\text{dev}}^{(n+1)} \leftarrow \min\left(\epsilon_{\text{dev}}^{(n)}, \kappa_{\text{dev}}^{(n+1)}\right)
\]

Kernel: 7

\[
\mathbf{W}_{\text{dev}}^{(n+1)} \leftarrow \text{diag}\left(\left\{\mathbf{x}_{\text{dev}}^{(n+1)} \cdot \mathbf{x}_{\text{dev}}^{(n+1)} + \epsilon^{(n+1)} \cdot \epsilon^{(n+1)}\right\}^{\frac{2-p}{2}}\right)
\]

Start

while \((n \leq \hat{n})\) do

Perform Kernel: 1 for all the IRLS problems
Barrier Synchronization

Perform Kernel: 2 for all the IRLS problems
Barrier Synchronization

Perform Kernel: 3 for all the IRLS problems
Barrier Synchronization

Perform Kernel: 4 for all the IRLS problems
Barrier Synchronization

Perform Kernel: 5 for all the IRLS problems
Barrier Synchronization

Perform Kernel: 6 for all the IRLS problems
Barrier Synchronization

Perform Kernel: 7 for all the IRLS problems
Barrier Synchronization

end while

Notes

\(r(\mathbf{x})\) is a function that returns the components of \(\mathbf{x}\) sorted in descending order according to the absolute values of the entries in \(\mathbf{x}\). Hence, \(r(\mathbf{x})_i\) is the \(i^{\text{th}}\) largest element of the set \(\{|x_j|, j = 1, 2, \ldots, U\}\). \(\lambda\) is the regularization parameter calculated as in Eq. 3.18.

The subscript terms dev and sm are used to represents data in device/global and shared memories respectively. The data that do not explicitly specify their memory type exist in the global memory. The most computationally intensive kernels are kernel 1 and 3.
6.3.3 Specifications of the Computing Platforms

We implemented the sparse recovery algorithm on multi-threaded architectures which are presented in Table 6.2. An introduction to the multi-threaded architectures is given in Appendix 3.7 to understand the architectures presented in this chapter. We used CMP (Chip Multiprocessor), MP (Multiprocessor), GPU and Manycore architectures in this analysis.

Table 6.2: Specifications of multi-threaded architectures which have been used for analyses the performance of sparse recovery process.

<table>
<thead>
<tr>
<th>System†</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>CMP</td>
<td>CMP</td>
<td>CMP</td>
<td>MP</td>
<td>MP</td>
<td>Manycore</td>
<td>GPU</td>
</tr>
<tr>
<td>Number of Processors</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cores per Processor</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>60</td>
<td>15</td>
</tr>
<tr>
<td>H/W Threads per Core</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>192</td>
</tr>
<tr>
<td>Thread Frequency</td>
<td>2.4</td>
<td>3.2-3.4†</td>
<td>2.1-3.3†</td>
<td>2.1</td>
<td>1.4</td>
<td>1.05</td>
<td>0.745</td>
</tr>
<tr>
<td>Platform Memory Bandwidth</td>
<td>17.1</td>
<td>25.6</td>
<td>25.6</td>
<td>76.8</td>
<td>204.8</td>
<td>320</td>
<td>288</td>
</tr>
</tbody>
</table>

† The system index is corresponding to (1) Intel Core-i3 370M (2) Intel Core-i5 4460 (3) Intel Core-i7 4600U (4) Intel Xeon E5-2450 (5) AMD Opteron 6378 (6) Intel Xeon Phi 5110P Coprocessor and (7) Nvidia K40 GPU.

‡ Operates with Intel Turbo-Boost technology

6.3.4 Simulation Paradigm

Now we describe the simulation paradigm. In the experiment, we increased the number of IRLS problems assigned to the platform at a time and calculated the rate of solving the IRLS problems. The number of iterations of the IRLS algorithm is set to 100 for making the computational complexity constant for all the IRLS problems. The rate of solving the IRLS problems is calculated such that:

\[
\text{Rate of solving the IRLS problems} = \frac{\text{Number of IRLS Problems}}{\text{Time Spent to Solve All the IRLS Problems}},
\]

which is called effective computational rate (ECR) of the architecture for performing the sparse recovery. Further, we measured the ECR against different dictionary resolutions on different platforms.
6.4 Results

In this section we discuss the results of solving the sparse recovery problem on the specified architectures in Table 6.2. We measured the ECR of the IRLS problems on different platforms. Fig. 6.5 presents ECR plots corresponding to the architectures presented in Table 6.2. Regarding the plots, the IRLS problems are solved with a dictionary having 230 resolution. We have identified resolution of the dictionary is linearly proportional to the computational complexity of the IRLS algorithm when the order of the SFT signal remain constant (see Table 6.1). In Fig. 6.6 we present the variation of the peak ECR against the resolution of the dictionary on the considered platforms. Therefore, a reciprocal model of the dictionary resolution is applied on the results as the dictionary resolution is inversely proportional to the computational complexity of the IRLS algorithm. The reciprocal function $\frac{\kappa}{U}$ is fitted on the peak ECR measurements where, $\kappa$ is the curve-fit coefficient and $U$ is the dictionary resolution. The curve-fit coefficient $\kappa$ is the parameter of the reciprocal model (i.e., $\frac{\kappa}{U}$), which is calculated using Matlab lsqcurvefit function by minimising the sum of squares of the error in the model. The curve-fit coefficients related to different platforms are given in Table 6.3.

Table 6.3: The curve-fit coefficients ($\kappa$) related to different platforms.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Curve-fit coefficient ($\kappa$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core-i7 4600U</td>
<td>$1.1862 \times 10^4$</td>
</tr>
<tr>
<td>Intel Core-i5 4460</td>
<td>$3.0497 \times 10^4$</td>
</tr>
<tr>
<td>Intel Core-i3 370M</td>
<td>$7.2283 \times 10^3$</td>
</tr>
<tr>
<td>AMD Opteron 6378</td>
<td>$1.6585 \times 10^5$</td>
</tr>
<tr>
<td>Intel Xeon E5-2450</td>
<td>$8.9250 \times 10^4$</td>
</tr>
<tr>
<td>Intel Xeon Phi 5110P Coprocessor</td>
<td>$3.8218 \times 10^5$</td>
</tr>
<tr>
<td>Nvidia K40 GPU</td>
<td>$2.5282 \times 10^4$</td>
</tr>
</tbody>
</table>

Now we validate our performance results against literature. In the paper [34], a fast batch Cholesky decomposition on GPU is discussed. To verify our results, we compare the performance of our implementation on GPU against the results of that paper. The both algorithms were implemented on a Nvidia K40 GPU while Cholesky decomposition is performed on similar small size positive-definite matrices. First, we calculate the performance
Figure 6.5: Effective computational rate (ECR) against the number of IRLS problems solved on different architectures.
Figure 6.6: Curve fit the peak ECR measurements of different dictionaries with the reciprocal function of the dictionary resolution.
of our method as follow,

\[ \text{Performance} = \text{The calculated number of FLOP per iteration} \times \text{The executed number of iterations per problem} \times \text{Number of solved problems per second} \]

The calculated performance is 0.9±1 GFLOPS for all the dictionaries. Note, that the dimension of the positive-definite matrix in our problem is 9, which is equivalent to the number of SFT signals. The performance of our implementation is of similar magnitude compared to the performance presented in Fig. 6.7 which is taken from the paper [34]. Similarly, we calculate and compare the performance of Cholesky decomposition on Intel Xeon-Phi architecture. The performance of our implementation is 13.77±1 GFLOPS which is comparable with the results presented in the paper [75]. In this comparison, we compare our results against the scaler version as we have not implemented any vector operations on Intel Phi. In the scaler version, the acceleration is achieved by utilizing many threads instead of utilizing vector processing unit (VPU).

### 6.5 Discussion

In the previous sections, we tested the performance of the sparse recovery when it is performed on multi-threaded architectures. In this section, we analyze and discuss the results. As per the results, ECR increases steadily when the number of IRLS problems are lower than
the cores. In some instances, this steadiness remains until the number of problems exceeding the available hardware threads. When the architectures have cores/processors which have incoherent caches and possibly independent memory access to dynamic memory, the thread efficiency is not degrading when the threads are being bound to the hardware with scatter affinity. This is because the allocated resources remain constant for each problem. In NUMA and Intel Phi architectures this characteristic is more visible. This is because NUMA has cache incoherent threads associated with independent memory channels and Intel Phi has cache incoherent cores which associated with many memory channels. According to Gustafson’s law [54], if the number of problems increases while keeping the allocated resources for a problem same, then the performance increases with the similar rate of increase in the number of problems.

When the number of IRLS problems increases beyond a certain limit, the rate of increasing ECR gradually decreases and converge to a peak. Then the ECR cannot be increased further by increasing the number of IRLS problems. According to TMM model [79], the performance of algorithms depends on the number of threads, when the number of threads is small. The performance converges to PRAM performance [44] with a sufficient number of threads, which only depends on the problem size and the number of processors. Since the problem size and the number of processors are fixed, the ECR remains constant. This happens due to the reduction of the cache efficiency when increasing the number of threads.

The cache performance of a program is related to cache hit/miss rate which is a function of size of the cache and data localities of the algorithm. Smith [116] presented a 30% rule based on their observation of cache performance which stated that every doubling of cache size $x$ should reduce the cache miss rate $f(x)$ by 30%. This recurrence relation can be formulated as

$$0.7f(x) = f(2x).$$

This relationship was generalized as one-term polynomial function s.t.,

$$f(x) = \beta x^\alpha,$$

where $\alpha$ and $\beta$ are cache miss rate function constants which depend on the temporal locality of the application data [65]. Note that $\alpha$ is negative, $\beta$ is positive and $f(x) \in (0, 1)$. Regarding the shared cache in a particular parallel computing architecture, the cache allocated to a problem can be defined s.t.,

$$C_s = \frac{C}{N_{irls}},$$

(6.31)
where \( C \) is size of the shared cache and \( N_{irls} \) is the number of IRLS problems. Then the cache miss rate of the program can be expressed by Eq. (6.30) s.t.,

\[
f(C) = \beta \left( \frac{C}{N_{irls}} \right)^\alpha.
\]  

(6.32)

Eq. (6.32) describes the cache miss rate when varying the number of problems. For clarity, Eq. (6.32) can be reformulated s.t.,

\[
f(C) = \beta \cdot C^\alpha \cdot N_{irls}^{-\alpha}.
\]  

(6.33)

where \( \beta \cdot C^\alpha \) and \( -\alpha \) positive constants. Therefore, the cache miss rate increases when increasing the number of problems. We assume this leads to gradually decreasing the rate

![Graph](image)

**Figure 6.8:** Increase of the cache-miss rate when increasing the number of IRLS problems.

of increasing the ECR. Once the performance is converged to PRAM performance (i.e., when the cache performance is low), the ECR remains constant.

The zig-zag pattern is related to the instantaneous imbalance of load of the hardware threads due to thread affinity. Since the applied thread affinity is Scatter, when the beginning of a new round of assigning IRLS problems to the cores/threads there would be load imbalance of the cores. Because of this load imbalance ECR may be dropped. Because of round-robin assignment of problems to the cores/threads, this repeats at starting of new rounds of assigning IRLS problems and cause zig-zag pattern. However, when the number of threads increases within a round, ECR increases again beyond the previous peak ECR. Nevertheless, when the hardware threads of the processing platform are fully utilized the load imbalance of hardware threads due to thread affinity becomes less noticeable.
We assume that zig-zag shape of the ECR on the GPU is due to imbalance of block allocation on the GPU resources. The balance of the block allocation on the GPU can be defined:

\[
\text{Balance of the Block Allocation} = \left\lfloor \frac{B_r}{B_a \cdot N_p} \right\rfloor, \quad (6.34)
\]

where

- \(B_r\) Requested number of thread blocks (= Number of IRLS problems),
- \(B_a\) Active number of thread blocks on a SM,
- \(N_p\) Number of processors in the device (=15 for K40 GPU).

The balance of block allocation on the GPU against the requested number of IRLS problems can be expressed by Fig. 6.9. As per the figure, balance of block allocation becomes optimum when the number of IRLS problems are integer multiply of 15 (i.e., Number of SMs on the GPU). The zig-zag shape in Fig. 6.6(g) is due to the balance of block allocation on the GPU. The reason of increasing the ECR with the number of IRLS problems is more IRLS problems increase the occupancy of the GPU. When the occupancy of the GPU increases, it may hide memory access latency more efficiently and improve the performance. The occupancy of the GPU can be calculated by Eq. 6.35 such that,

\[
\text{GPU Occupancy} = \left\lfloor \frac{T \cdot B_a}{T_{\max} \cdot W} \right\rfloor, \quad (6.35)
\]

Figure 6.9: The balance of block allocation on the GPU against the requested number of IRLS problems.
where

\( T_r \)  
Requested number of threads per block,

\( B_a \)  
Active thread blocks per processor,

\( T_{maxW} \)  
Maximum threads per warp,

\( W_{maxP} \)  
Maximum warps per processor.

The parameter \( T_r \) is specific to the kernel which needs to be determined at the launch of the kernel on the GPU. The parameters \( T_{maxW} \) and \( W_{maxP} \) are specific to the GPU which can be found in the GPU user manual. The number of active thread blocks per processor \( B_a \) can be calculated by Eq. 6.36 \[80\] such that,

\[
B_a = \min \left( \left\lfloor \frac{S}{S_B} \right\rfloor, \left\lfloor \frac{R}{R_T \cdot T_r} \right\rfloor, \left\lfloor \frac{B_r}{N_p} \right\rfloor, \left\lfloor \frac{T_{maxP}}{T_r} \right\rfloor \right),
\]

(6.36)

where

\( S \)  
Shared memory per processor (in Bytes),

\( S_B \)  
Shared memory used per block (in Bytes),

\( R \)  
Number of registers per processor,

\( R_T \)  
Number of registers per thread,

\( T_r \)  
Requested number of threads per block,

\( N_p \)  
Number of processors in the device,

\( B_r \)  
Requested number of blocks (= Number of IRLS problems),

\( T_{maxP} \)  
Maximum number of threads per processor.

The parameters \( S, R, N_p \) and \( T_{maxP} \) are specific to the GPU which can be found in the GPU user manual. The parameters \( S_B \) and \( R_T \) depend on the design of the kernel, which needs to be either calculated by analyzing the kernel or evaluated by a program profiler such as NVIDIA Visual Profiler. The parameters \( T_r \) and \( B_r \) are specific to the kernel, which need to be determined at the launch of the kernel on the GPU.

Now we analyze Fig. 6.6 which describes how peak ECR behaves against the dictionary resolution. As per Table 6.1 if the number of SFT signals are being constant, the Flop per IRLS iteration is linearly proportional to the resolution of the dictionary. It is possible to
derive a relationship between peak ECR and the resolution of the dictionary as follows.

\[
\text{Peak ECR} = \frac{\text{Maximum number of IRLS problems}}{\text{Total time spent}}, \quad (6.37)
\]

\[
\text{Peak ECR} = \frac{\text{Maximum number of IRLS problems}}{\text{Total flop performed} \cdot \frac{\text{Total flop performed}}{\text{Total time spent}}}, \quad (6.38)
\]

\[
\text{Peak ECR} = \frac{\text{Maximum number of IRLS problems}}{\frac{\text{Total flop performed}}{\text{Total time spent}}}, \quad (6.39)
\]

\[
\text{Peak ECR} = \frac{\text{Peak attainable performance}}{\text{Flop per problem}}, \quad (6.40)
\]

\[
\text{Peak ECR} = \frac{\text{Peak attainable performance}}{(\text{Average number of iterations}) \cdot (\text{Flop per IRLS iteration})}, \quad (6.41)
\]

\[
\text{Peak ECR} = \frac{\text{Peak attainable performance}}{(\text{Average number of iterations}) \cdot (c \cdot U)}, \quad (6.42)
\]

\[
\text{Peak ECR} = \frac{\kappa}{U}, \quad (6.43)
\]

where, \( U \) is the resolution of the dictionary and \( c \) is a constant which depends only on the order of the SFT (see Table 6.1). The parameter \( \kappa \) would be,

\[
\kappa = \frac{\text{Peak attainable performance}}{c \cdot (\text{Average number of iterations})}. \quad (6.44)
\]

Assuming the peak attainable performance and average number of iterations are constant for solving IRLS problems on a given architecture, the Peak ECR is inversely proportional to the dictionary resolution. Eq.6.43 is the same function which we used to curve fit the peak ECR measurements. Therefore, curve-fitting coefficient \( \kappa \) is proportional to the peak attainable performance of the architecture.

Now we describe the relative performance of the selected architectures by comparing the curve-fitting coefficient \( \kappa \). We showed that the curve-fitting coefficient \( \kappa \) is proportional to the peak attainable performance of the architecture. The relative performance of the architectures are presented in Table 6.1 by calculating the ratio of the curve-fitting coefficients against the curve-fitting coefficient corresponding Intel Core-i3 370M processor. The Intel Core-i3 370M processor is selected as the reference since it has the lowest curve-fitting coefficient from the selected architectures. As per the table, Intel Xeon Phi 5110P coprocessor delivered the best performance which is 52.87 faster than the Intel Core-i3 370M processor.
Table 6.4: The relative performance of the architectures against Intel Core-i3 370M processor. The relative performance is the ratio of the curve-fitting coefficients against the curve-fitting coefficient corresponding Intel Core-i3 370M processor.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Relative Performance against Intel Core-i3 370M processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core-i7 4600U</td>
<td>1.64</td>
</tr>
<tr>
<td>Intel Core-i5 4460</td>
<td>4.16</td>
</tr>
<tr>
<td>AMD Opteron 6378</td>
<td>22.94</td>
</tr>
<tr>
<td>Intel Xeon E5-2450</td>
<td>12.34</td>
</tr>
<tr>
<td>Intel Xeon Phi 5110P Coprocessor</td>
<td>52.87</td>
</tr>
<tr>
<td>Nvidia K40 GPU</td>
<td>3.50</td>
</tr>
</tbody>
</table>

6.6 Conclusion

In this chapter first, we analyzed the computational complexity of the IRLS algorithm. Then we implemented the sparse recovery algorithm in OpenMP and CUDA to perform multiple IRLS problems in parallel using software threads. Finally, we investigated the performance of solving the IRLS problems on multi-threaded architectures. In this investigation, we increased the number of assigned problems on the architecture and analyzed the rate of solving the IRLS problems. Based on the results, following conclusions are made.

1. Analytically the least computational complex method of implementing the IRLS algorithm is Cholesky decomposition based method.

2. To achieve the best performance on multithreaded architectures, the number of IRLS problems should be an integer multiple of the total number of hardware threads in the architecture. Regarding the GPU, the number of IRLS problems should be an integer multiple of the active-thread blocks. In summary, when assigning IRLS problems on an architecture, the workload on hardware resources of the architecture should be balanced for the peak performance.

3. The peak ECR is inversely proportional to the dictionary resolution. We estimated the proportionality constant of the peak ECR and the dictionary resolution for the selected architectures. Using the proportionality constant, it is possible to estimate
the peak ECR of solving the IRLS problems for an arbitrary dictionary resolution (see Table 6.5).

Table 6.5: The rate of solving the IRLS problems on the selected architectures for an arbitrary dictionary resolution $U$.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Number of IRLS problems per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core-i7 4600U</td>
<td>$1.1862 \times 10^4 / U$</td>
</tr>
<tr>
<td>Intel Core-i5 4460</td>
<td>$3.0497 \times 10^4 / U$</td>
</tr>
<tr>
<td>Intel Core-i3 370M</td>
<td>$7.2283 \times 10^3 / U$</td>
</tr>
<tr>
<td>AMD Opteron 6378</td>
<td>$1.6585 \times 10^5 / U$</td>
</tr>
<tr>
<td>Intel Xeon E5-2450</td>
<td>$8.9250 \times 10^4 / U$</td>
</tr>
<tr>
<td>Intel Xeon Phi 5110P Coprocessor</td>
<td>$3.8218 \times 10^5 / U$</td>
</tr>
<tr>
<td>Nvidia K40 GPU</td>
<td>$2.5282 \times 10^4 / U$</td>
</tr>
</tbody>
</table>

4. The curve-fitting coefficient $\kappa$ is proportional to the peak attainable performance of the architecture. The relative performance of an architecture can be estimated by calculating the ratio of the curve-fitting coefficient against the curve-fitting coefficient corresponding a reference architecture. Table 6.1 presents relative performances of the selected architectures against Intel Core-i3 370M processor which are calculated using curve-fitting coefficients. As per the table, Intel Xeon Phi 5110P coprocessor delivers the best performance which is 52.87 faster than the Intel Core-i3 370M processor.

In summary, multi-threaded architectures are useful to accelerate the sparse recovery by solving IRLS problems in parallel. Further, the reduction of the dictionary resolution increases the rate of solving IRLS problems. In the next chapter we investigate possible changes to the sparse-recovery algorithm to reduce the resolution of the dictionary used in the IRLS computation.
Table 6.6: The relative performance of the architectures against Intel Core-i3 370M processor. The relative performance is the ratio of the curve-fitting coefficients against the curve-fitting coefficient corresponding Intel Core-i3 370M processor.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Relative Performance against Intel Core-i3 370M processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core-i7 4600U</td>
<td>1.64</td>
</tr>
<tr>
<td>Intel Core-i5 4460</td>
<td>4.16</td>
</tr>
<tr>
<td>AMD Opteron 6378</td>
<td>22.94</td>
</tr>
<tr>
<td>Intel Xeon E5-2450</td>
<td>12.34</td>
</tr>
<tr>
<td>Intel Xeon Phi 5110P Coprocessor</td>
<td>52.87</td>
</tr>
<tr>
<td>Nvidia K40 GPU</td>
<td>3.50</td>
</tr>
</tbody>
</table>
Chapter 7

Sparse Recovery Using Non-Uniform Spatial Dictionaries

7.1 Introduction

In the previous section, we have analyzed acceleration of the sparse recovery process when solving multiple IRLS problems in parallel on a parallel architecture. In parallel processing architectures, each IRLS problem is solved as a thread and many threads are executed simultaneously to accelerate the process. We have observed, when the number of IRLS problems are exceeding a certain limit, the effective number of simultaneous IRLS problems which is executed on the architecture reaches a peak and remains constant. In a resource-constrained architecture, the performance does not always increase as per Gustafson’s law which states the performance can be scaled-up with the number of processors when each problem is assigned to a dedicated processor [54]. Instead, it can be explained by famous Amdahl’s law [8] on parallel computing.

Amdahl’s law considers a serial and parallel portion of the computation when calculating the acceleration on parallel architectures. In sparse recovery, the solving of individual IRLS problems in parallel using parallel resources on the platform is the parallel portion of the computation. However, the rest of the computation has to be performed in serial because the computation cannot be further subdivided to execute in parallel as there are dependencies in the computation and/or the resources are not sufficient to assign each problem to dedicated resources. Then Amdahl’s law states the achievable acceleration $S(N)$:

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + \frac{T_p}{N}},$$  \hspace{1cm} (7.1)

where, $T_p$ is the time taken to execute perfectly parallelizable portion in the algorithm while $T_s$ is the time taken to execute totally serial portion. The $N$ is the number of parallel processors in the architecture. As per Amdahl’s law, the maximum acceleration
obtainable on an infinite number of parallel processors is only $T_s + T_p T_s$. This means the overall acceleration is limited by the serial workload which cannot be benefited by the parallel processing. Therefore, one possible technique to increase the acceleration is reducing the time complexity of the serial portion of the computation which is $T_s$. Regarding the sparse recovery process which has been discussed, the IRLS solver is the serial section of computation. Even though different IRLS problems are solved in parallel, an individual IRLS problem is solved serially due to data dependencies within the algorithm.

The time complexity of the IRLS problem is linearly proportional to the size of the dictionary. The issue of the resolution of the spatial dictionary versus the computational cost leads us to consider the use of spatial dictionaries with non-uniform resolution. If one is only interested in a specific region of space, it would seem to be advantageous to locally increase the resolution of the dictionary in this region. Conversely, one can speed-up solving the IRLS problem by strategically reducing the resolution of the spatial dictionary in spatial regions that are not of interest. Moreover, if there are many regions of interest, ability to subdivide the sparse recovery problem using multiple non-uniform spatial dictionaries helps to reduce the computational complexity of individual problems while enabling to solve them in parallel.

Therefore, the concept of using non-uniform spatial dictionaries for sparse recovery is beneficial to accelerate the sparse recovery process. In this chapter, we explore several methods of applying non-uniform spatial dictionaries for the sparse recovery.

### 7.2 The proof-of-concept of using non-uniform spatial dictionary for sparse recovery

In our previous sparse recovery work, we have always run the sparse recovery algorithm using a spatial dictionary that has constant resolution across space. In this work, we explore the consequences of running the sparse recovery algorithm with a non-uniform spatial dictionary. For simplicity, our non-uniform spatial dictionary uses two different resolutions: a higher resolution for the front hemisphere of space and a lower resolution for the back hemisphere of space. Conceptually, the front hemisphere of space is treated as the *region of interest*, while the back hemisphere of space is ignored. In other words, in sparse recovery, we are interested in obtaining a high-resolution acoustic energy map for the front hemisphere and are not concerned with the quality of the energy map for the back hemisphere.

This work examines how the different spatial resolutions applied to the dictionary for the front and back hemisphere influence the quality of the front-hemisphere energy map. A question of particular interest is how much can we decrease the spatial resolution of the dictionary in the back-hemisphere and still obtain a robust, high-resolution, front-hemisphere...
energy map. An answer to this question will indicate the effective speedup of solving an individual IRLS problem and effectiveness of subdividing the sparse recovery problem which are complimenting factors for increasing the peak ECR of parallel architectures. For brevity, we assume the objective of obtaining a robust, high-resolution, front-hemisphere acoustic energy map is implicitly understood. In this context, we explore the interplay between the level of diffuse background noise and the dictionary resolution required for the front and back hemisphere of space. We also explore how the intensity level of a source located in the back hemisphere of space influences the dictionary resolution required for the front and back hemisphere of space.

Table 7.1: Position and the strength of the sources.

<table>
<thead>
<tr>
<th>Source</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azimuth (deg.)</td>
<td>-3.28</td>
<td>4.91</td>
<td>-23.30</td>
<td>68.57</td>
<td>-160.56</td>
</tr>
<tr>
<td>Elevation (deg.)</td>
<td>25.63</td>
<td>35.00</td>
<td>-8.67</td>
<td>-21.92</td>
<td>-8.07</td>
</tr>
<tr>
<td>Intensity (dB)</td>
<td>-6.02</td>
<td>-6.02</td>
<td>-20</td>
<td>-1.94</td>
<td>variable</td>
</tr>
</tbody>
</table>

We now describe the details of the simulation paradigm. In this work, the non-uniform spatial dictionaries are created using Lebedev grids \[73\]. Lebedev grids provide a means to flexibly generate relatively uniform spatial grids across a sphere with varying number of grid points. The non-uniform spatial dictionary uses a Lebedev grid of higher spatial resolution for the front hemisphere of space and lower spatial resolution for the back hemisphere of space. We place four test sources (Sources 1 to 4) in the front hemisphere region of interest to examine the resolution of the obtained acoustic energy maps. There is also a possible fifth interfering source placed in the back hemisphere. A table listing the source positions and their relative amplitudes in dB is shown in Table 7.1. None of the source positions are included in the non-uniform spatial dictionaries. Sources 1 and 2 are set close to each other with a separation angle of approximately 12 degrees in order to challenge the resolving power of the acoustic energy maps. Source 3 is a weak source that may easily go undetected in an acoustic energy map. Source 4 is a strong source that should easily be detected, but may dominate the acoustic energy map. Source 5 is an interfering source placed in the back hemisphere that can distort the front hemisphere acoustic energy map. The acoustic energy maps are obtained by applying the sparse-recovery algorithm to order-3 SFT signals. Diffuse noise is simulated by adding uncorrelated white noise to the source signals. The level of the diffuse noise and the interfering source could be changed independently.

There are a number of significant issues that arise in the acoustic imaging problem when using a non-uniform spatial dictionary. To clarify these issues, consider Fig. 7.1. In this figure, we show four different acoustic energy maps for the front hemisphere of space. For
Figure 7.1: Acoustic energy maps for the front hemisphere of space are shown. These maps were obtained using sparse recovery with non-uniform spatial dictionaries. \( N_f \) and \( N_b \) indicate the size of the front hemisphere and back hemisphere dictionaries, respectively. Source 5 is not shown as it is located in the back hemisphere; it had an amplitude of 0 dB. Diffuse noise was added to the SFT signals at a level of -20 dB.

As well, diffuse noise with an amplitude of -20 dB, relative to the signals originating from the sources, was added to the SFT signals. For brevity, we use \( N_f \) to indicate the number of Lebedev grid directions in the front hemisphere of space and \( N_b \) to similarly indicate the number of directions in the back hemisphere of space. Consider now Fig. 7.1d for which the four test sources are clearly resolved. This acoustic energy map was obtained using \( N_f = 401 \) and \( N_b = 37 \). There is a drastic reduction in the number of directions in the back hemisphere compared to the front hemisphere, which nevertheless still results in a high-resolution acoustic energy map. We cannot really reduce the number of directions in the back hemisphere any further. As shown in Fig. 7.1c, reducing \( N_b \) to 21 results in an
energy map which misses the relatively weak Source 3. Furthermore, as shown in Fig. 7.1b, reducing $N_b$ to zero simply results in a spurious energy map. The resolution of the energy map clearly depends on $N_f$. In Fig. 7.1c, $N_f = 105$ and the resolution of the energy map is clearly reduced compared to Fig. 7.1d for which $N_f = 401$.

In order to quantitatively explore the accuracy of an acoustic energy maps, we define and calculate an angular spread for each test source. To calculate the angular spread for a given test source, we first examine the test source direction and its twenty nearest neighbor directions and identify which of these directions has the maximum energy in the energy map. We then find the nearest direction to the direction of maximum energy which has less than 80% of the maximum energy value. The angular spread is defined as the angle between the direction of maximum energy and the direction with 80% of the maximum energy. We measure the quality of a given acoustic energy map by the angular spread values for the four test sources in the front hemisphere. Generally, the smaller the angular spread, the greater is the accuracy of the acoustic energy map.

Figure 7.2: Influence of the spatial resolution of the dictionary in the front and back hemisphere of space on the accuracy of the energy map. Source 5 (located at the back) has an amplitude of 0 dB and the SFT signals have an SNR of 20 dB.
7.2.1 Influence of spatial resolution

We begin by examining the influence of $N_f$ and $N_b$ on the acoustic energy map for the front hemisphere. Figure 7.2 shows the value of the mean angular spread obtained for non-uniform dictionaries with various values for $N_f$ and $N_b$, in the presence of diffuse noise and with a 0 dB source located in the back hemisphere. Clearly, dictionaries with greater $N_f$ result in a lower mean angular spread. This indicates that increasing the spatial resolution in the front hemisphere leads to more accurate source localization. This is consistent with the concept that a dictionary with greater $N_f$ can provide a sparser explanation of the observed signals, and thus the obtained image is cleaner and more accurate.

As well, it can be observed that the accuracy of the energy map is also influenced by $N_b$. It appears that $N_b$ must be larger than some critical value to obtain the most accurate acoustic energy map. For example, for $N_f = 401$, the lowest angular spread value is reached for $N_b$ values greater than 49. It is not clear what determines the critical value for $N_b$. However, it is clear that the dictionary requires some directions in the back hemisphere in order to account for the presence of a source in the back hemisphere. The reason that some directions are required in the back hemisphere can be understood as follows. The plane-wave decomposition obtained from the sparse recovery algorithm has to explain the observed sound field and it stands to reason that when there is a source in the back hemisphere, the plane-wave decomposition requires some directions in the back hemisphere. However, it is not clear to us what exactly specifies the number of directions required in the back hemisphere to maintain an accurate acoustic image of the front hemisphere of space.

7.2.2 Robustness to diffuse noise

An important characteristic of sound field imaging techniques is robustness in the presence of diffuse noise. In the context of acoustic imaging, noise can occur in the shape of electronic noise (produced by the microphones) or in that of diffuse reverberation. Figure 7.3 compares the angular spread obtained for Sources 1 and 3 with: a) a dictionary with a uniform resolution ($N_f = 401$ and $N_b = 369$; note that $N_f \neq N_b$ because $N_f$ includes locations with an azimuth of $\pm \frac{\pi}{2}$); and b) a non-uniform dictionary with fewer directions in the back hemisphere ($N_f = 401$ and $N_b = 49$), as a function of the SNR of the SFT signals. Note that when a source was completely missed the angular spread value was limited to 45$^\circ$ to make the plot easier to read. Clearly, noise has a very large influence on the accuracy of the map. Source 3 was missed for SNR values lesser than 20 dB and both sources were missed when the SNR was -20 dB. However, no significant difference can be observed between the angular spread values obtained with the uniform and non-uniform dictionaries. In other
7.2.3 Robustness to a back hemisphere source

Lastly, we examine the robustness of the imaging in the presence of an interfering source located in the back hemisphere. This scenario is quite different from that of the previous section because diffuse noise appears to be originating from every direction in space. Figure 7.4 compares the angular spread for Sources 1 and 3 obtained with: a) a dictionary with uniform spatial resolution ($N_f = 401$ and $N_b = 369$); and b) a non-uniform dictionary with fewer directions in the back hemisphere ($N_f = 401$ and $N_b = 49$), as a function of the amplitude of Source 5 (located in the back hemisphere). Similarly to the presence of noise, the presence of Source 5 has an important influence on the accuracy of the acoustic energy map: the greater the amplitude of Source 5, the greater the angular spread. As well, Source 3 was missed when the amplitude of Source 5 was 20 dB. However, the results obtained with the two different dictionaries are almost identical. Hence, the use of a dictionary with fewer directions in the back hemisphere does not seem to make the imaging more sensitive to the presence of a source in the back hemisphere of space.

As per the simulations, even using dictionaries with relatively low spatial resolution in the back hemisphere, accurate energy maps for the front hemisphere of space could still be obtained. Our success with using non-uniform spatial dictionaries indicates that it is worth
Figure 7.4: Influence of a source in the back hemisphere on the accuracy of the energy map. This figure compares the results obtained with: a) a uniform dictionary ($N_f = 401$ and $N_b = 369$); and b) a non-uniform dictionary with fewer directions in the back hemisphere ($N_f = 401$ and $N_b = 49$), as a function of the amplitude of Source 5. Note that no noise is present.

exploring algorithms which spatially subdivide or refine the dictionary used in the sparse recovery problem for acoustic imaging.
7.3 Sparse plane-wave decomposition on streaming frequency-domain SFT signals

In section 3.4 we explained a SFT signal $h$ can be represented in time-frequency domain such that:

$$h(t, f) = [h_1(t, f), h_2(t, f), ..., h_K(t, f)]^T.$$ \hspace{1cm} (7.2)

where, $t$ is the time window and $f$ is the frequency bin. Note that $h$ consists of $K$ spherical harmonic expansion signals which depends on the order of the SFT. In practice $h$ signal is generated by short-time Fourier transform (STFT). Assuming STFT is performed over $N_t$ time windows, we can construct a matrix $H$ for a given frequency such that:

$$H(f) = [h(t_1, f), h(t_2, f), ..., h(t_{N_t}, f)],$$ \hspace{1cm} (7.3)

where, $H$ represents streaming frequency-domain SFT signals. Regarding streaming frequency-domain SFT signals, it is inefficient to perform the sparse recovery for each frequency bin and time window sequentially. Following this section, we perform the frequency-domain IRLS problems by accounting multiple time windows for each frequency simultaneously.

Now we describe the multiple time window frequency domain IRLS algorithm. The IRLS solution in Eq. 3.17 can be transformed such that,

$$x = WD^T (DWD^T + \lambda I)^{-1} h,$$ \hspace{1cm} single frequency single time window \hspace{1cm} (7.4)

$$X = WD^T (DWD^T + \lambda I)^{-1} H,$$ \hspace{1cm} single frequency multiple time windows \hspace{1cm} (7.5)

In Eq. 7.4 the column vector $h$ is a SFT signal corresponding to a particular frequency bin and a time window. In Eq. 7.5 the matrix $H$ is constructed using multiple $h$ signals which are belongs to same frequency bin and multiple consecutive time windows. Consequently the resulting matrix $X$ in Eq. 7.5 would be similar to combination of $x$ column vectors where each $x$ is a sparse plane-wave decomposed signal. Assume there are $N_t$ time windows in $H$ matrix. Then the computational complexity of the IRLS solution depends on the size $N_t$. If the matrix $H$ has more columns than rows (i.e. $N_t > (\Lambda + 1)^2$), it implies that the columns of $H$ are not linearly independent (the rank cannot exceed $(\Lambda + 1)^2$). This fact, along with the assumption that the locations of the sound sources remain stationary in all the time windows, enables an approach based on Singular Value Decomposition (SVD) \cite{81}. This method is called as dimension reduction which reformulates the Eq. 7.5 to an equivalent but computationally simpler problem. Following this approach, SVD is applied to $H$ in order to express it in the subspace defined by its first $(\Lambda + 1)^2$ singular vectors. Thus the
matrix of SFT signals is reduced from a \( N_t \)-dimensional to a \((\Lambda + 1)^2\)-dimensional matrix. Mathematically, this can be expressed such that:

\[
H = UV^T, \quad (7.6)
\]

\[
H_{\text{red}} = U\Gamma, \quad (7.7)
\]

where \( H_{\text{red}} \) is the reduced version of \( H \). With the reduce form, the optimization problem in Eq. 7.5 can be transformed to equivalent but significantly smaller problem:

\[
X_{\text{red}} = WD^T \left( DWD^T + \lambda I \right)^{-1} H_{\text{red}}. \quad (7.8)
\]

where \( X_{\text{red}} \) is the reduced form of \( X \). The matrix \( X \) can be calculated from the reduced form \( X_{\text{red}} \) such that:

\[
X = X_{\text{red}} V^T. \quad (7.9)
\]

### 7.4 Algorithms of non-uniform spatial dictionaries based sparse recovery

In the previous section, we have noticed non-uniform spatial dictionaries can be used to accurately construct the energy maps using IRLS algorithm. In this section, we discuss 3 non-uniform spatial dictionary based sparse recovery methods. They are 1) dictionary-refining method, 2) dictionary-subdividing method and 3) combined method of first and second methods. During the analysis of three methods, we follow the general steps given in Algorithm 8 at the beginning of each method. In our experiments, we used 512-point STFT.

**Algorithm 8** The general steps of starting the frequency-domain IRLS process.

1. Calculate short-time Fourier transform (STFT) of the SFT signals.
2. Band-pass filter the STFT signals based on the SNR of the frequency-domain signals.
3. Singular value decomposed (SVD) the band-pass signals for dimensionality reduction.
4. Perform the non-uniform dictionary based sparse recovery on dimensionality reduced signals.

The window length is 256 and the hop size is 64 which is a quarter of the window length. Each window is zero-padded with 256 zeros. Our spherical microphone array has high SNR between 900 Hz and 9000 Hz which are the settings we used for band-pass filtering. Following is the description of three non-uniform dictionary based sparse recovery methods.
7.4.1 Dictionary refining method

In dictionary refining method (first method), the initial dictionary is a high-resolution uniform dictionary. It contains a uniform low-resolution dictionary which is unchanged during the sparse recovery. The high-resolution dictionary is progressively refined based on the weights calculated in each iteration. The dictionary directions corresponding to low weights are omitted from the dictionary when progressing. The sparse recovery is initiated with unit weights in all directions. Then, the weights are recalculated in each iteration of the IRLS algorithm such that,

\[ W^{(n)} \leftarrow \text{diag} \left( \left[ w_1^{(n)}, w_2^{(n)}, \ldots, w_N^{(n)} \right] \right), \]  

(7.10)

where, \( N \) is the number of directions in the dictionary. These weights are normalized by dividing by maximum weight. The normalized weights are used to identify the higher weights compared to a predefined threshold \( W_{th} \). This process can be described such that,

\[ W_{\text{norm}, \text{db}}^{(n)} \leftarrow 10 \log_{10} \left( \frac{W^{(n)}}{\max(W^{(n)})} \right), \]  

(7.11)

\[ \mathbf{v} \leftarrow \text{find} \left( W_{\text{norm}, \text{db}}^{(n)} \geq W_{th} \right). \]  

(7.12)

In here the find function is used to determine the indexes \( \mathbf{v} \) of the weights which are higher than the predefined threshold \( W_{th} \). Since the weights are corresponding to the dictionary directions, the dictionary directions related to the higher weights can be identified using the weight indexes \( \mathbf{v} \). The IRLS process is continued until reaching the maximum number of iterations or the length of \( \mathbf{v} \) is less than a predefined minimum value which indicates the dictionary is refined substantially. In each iteration, the new dictionary is abstracted from the existing dictionary based on the new \( \mathbf{v} \) indexes. Even though the dictionary is changed in each iteration, the new dictionary directions related to the original dictionary are tracked so the solution can be reconstruct at the end. The rest of the algorithm is similar to the general IRLS algorithm which was described in Section 3.4.2. Algorithm 9 presents the steps of the dictionary refining method. Figure 7.5 graphically presents an example of the dictionary refinement.

7.4.2 Dictionary subdividing method

In dictionary subdividing method (second method), the sparse recovery process is conducted using multiple dictionaries in parallel to reduce the computational complexity of the individual problem. The motivation is, if the parallelism is effective, the overall time should be less than the time spends to solve a single IRLS problem using an uniform dictionary for similar quality. In this method, a dictionary consists of high and low-resolution partitions in
Figure 7.5: The initial dictionaries and the way they are changed in the dictionary-refining method. In the dictionary-refining method the high and low resolution dictionaries are used. The low-resolution dictionary is kept unchange while the high-resolution dictionary is refined based on the source locations. Note that high resolution is maintained around the source while reducing the resolution in less interested regions.

space. The dictionary is segmented to low and high-resolution partitions in space based on an azimuth angle which determines the size of each partition. This idea is graphically presented in Figure 7.6. We call the azimuth angle which partitions the dictionary as slice angle ($\theta_m$). The range of the slice angle is less than or equal $\pi$-radians and an integer fraction of $2\pi$-radians (i.e., $\theta_m = \frac{2\pi}{k}$ where $k \geq 2$ and $k \in \mathbb{Z}^+$. The sparse recovery is performed using
multiple dictionaries such that the high-resolution segment of every dictionary covers the entire space without intersecting (see Fig. 7.6). In geometry, each dictionary corresponds to rotation of the constructed dictionary around azimuth axis by an integer multiple of slice angle. Since solving of different IRLS problems corresponding to different dictionaries are mutually independent, each IRLS problem can be solved simultaneously in parallel on a parallel architecture. On the other hand, if the sparse recovery is performed for the region of interest, the number of dictionaries can be limited to cover only the interested region by the high-resolution segment of the dictionaries. Assuming sparse recovery is performing on the entire region of space, the steps of the dictionary subdividing method is presented in Algorithm 10. The algorithm is developed to construct the number of dictionaries equivalent to the number of parallel threads in the application environment. Then the IRLS algorithm is performed in parallel using each dictionary. In the algorithm, genDict function is used to generate the first dictionary. Then using rotateGenDict function the other dictionaries are derived by rotating the coordinates of the generated dictionary around the azimuth axis by an integer multiple of slice angle $\theta_m$. At the end of the parallel algorithm, there exist solutions related to each dictionary. These solutions can be merged to an uniform solution by using combin function.

### 7.4.3 Combined method

In the combined method (third method), the sparse recovery techniques in method 1 and 2 are used together. In other words, the dictionaries which are constructed as high and low-resolution partitions in method 2 are refined by the IRLS algorithm as in method 1 when performing the sparse recovery. We assume the method 1 (i.e., dictionary refining method) can complement to method 2 (dictionary subdividing method) to achieve higher performance. Algorithm 11 presents the steps of the combined algorithm of Algorithm 9 and Algorithm 10.
Figure 7.6: The subdivision of the high resolution dictionary into 4. The viewpoint of the dictionary is from the top of the azimuth axis. Each dictionary is constructed using a high and a low resolution dictionaries. Note that each dictionary corresponds to rotation of the first dictionary around azimuth axis by an angle equivalent to integer multiplied of $\frac{2\pi}{4}$-radians (i.e., the slice angle).
Algorithm 9 Dictionary refining algorithm.

Input
$N_f$ : number of frequencies, $N_t$ : (time windows×hop size), $U$ : dictionary resolution,
$\Lambda$ : SFT Order
$D \in \mathbb{N}^{(\Lambda+1)^2 \times U}$ : uniform dictionary
$\hat{n}$ : maximum number of iterations allowed
$U_{th}$ : minimum dictionary resolution allowed
$W_{th}$ : minimum weight carry forward by the sparse recovery (threshold of the weights)
$v$ : indexes of the weights higher than the predefined threshold $W_{th}$
$\tilde{v}$ : reference indexes of $v$ to the original dictionary
$I$ : Identity matrix
$\lambda$ : regularization parameter calculated as in Eq. 3.18
$H_{red}$ : dimension reduced form of a time-frequency domain SFT signal

Output
$X \in \mathbb{C}^{N_f \times U \times N_t}$ : the sparse plane-wave solution

Algorithm
Perform Algorithm 8 for $n_f = 1 : N_f$

$n \leftarrow 0$

$D^{(0)} \leftarrow D$
$W^{(0)} \leftarrow \text{diag}([1,1,\ldots,1])$

while $(n \leq \hat{n})$

$W^{(n)} \leftarrow \text{diag}([w_1^{(n)}, w_2^{(n)}, \ldots, w_{\text{length}(v)}^{(n)}])$
$W_{\text{norm,db}}^{(n)} \leftarrow 10 \log_{10} \left( \frac{W^{(n)}}{\max(W^{(n)})} \right)$
$(v, \tilde{v}) \leftarrow \text{find_and_track} \left( W_{\text{norm,db}}^{(n)} \geq W_{th} \right)$

if length($v$) < $U_{th}$ then

break

else

$W^{(n)} \leftarrow W^{(n)}(v)$ : modify the weighting vector
$D^{(n)} \leftarrow D^{(n)}(v)$ : refine the dictionary

end if

$X^{(n+1)}_{\text{red}} \leftarrow W^{(n)} D^{T(n)} (D^{(n)} W^{(n)} D^{T(n)} + \lambda^{(n)} I)^{-1} H_{\text{red}}$
$x^{(n+1)} \leftarrow \text{energy} \left( X^{(n+1)}_{\text{red}} \right)$

Continue the IRLS algorithm and calculate the new weights $w_i^{(n+1)}$

$n \leftarrow n + 1$

end while

$\hat{X}^{(n_f)} \leftarrow X^{(n_f)}_{\text{red}} V^{T(n_f)}$ : transform back to the original dimension. See Eq. 7.6–7.9
$\tilde{X}^{(n_f)} \leftarrow \text{reconstruct} \left( \hat{X}^{(n_f)}, \tilde{v}^{(n_f)} \right)$

$X[n_f,:,:] \leftarrow \tilde{X}^{(n_f)}$

end for

Notes

\texttt{find_and_track} function is used to determine the indexes $v$ of the weights which are higher than the predefined threshold $W_{th}$ and track their reference indexes $\tilde{v}$ to the original dictionary. \texttt{reconstruct} function is used to reconstruct the final solution using the indexes $\tilde{v}$ which are related to the original dictionary. \texttt{energy} function returns the energy of the signals.
Algorithm 10 Dictionary subdividing algorithm.

Input
\[ N_f : \text{number of frequencies, } N_t : \text{(time windows} \times \text{hop size)}, \Lambda : \text{SFT Order} \]
\[ D_h \in \mathbb{N}^{(A+1)^2 \times U_h} : \text{high-resolution dictionary } (U_h : \text{dictionary resolution}) \]
\[ D_l \in \mathbb{N}^{(A+1)^2 \times U_l} : \text{low-resolution dictionary } (U_l : \text{dictionary resolution}) \]
\[ D_c \in \mathbb{N}^{(A+1)^2 \times U_c} : \text{composite-resolution dictionary } (U_c : \text{dictionary resolution}) \]
\[ \hat{n} : \text{maximum number of iterations allowed} \]
\[ I : \text{Identity matrix} \]
\[ \lambda : \text{regularization parameter calculated as in Eq. 3.18} \]
\[ H_{\text{red}} : \text{dimension reduced form of a time-frequency domain SFT signal} \]
\[ N_p : \text{Number of parallel threads} \]

Output
\[ X \in \mathbb{C}^{N_f \times U_h \times N_t} : \text{the sparse plane-wave solution} \]

Algorithm
\[ \theta_m = \frac{2\pi}{N_p} : \text{slice angle} \]
\[ D_c = \text{genDict}(D_h, D_l, \theta_m) \]
\[ \left[ D_{c}^{(1)}, \ldots, D_{c}^{(N_p)} \right] = \text{rotateGenDict}(D_c, \theta_m) \]

Perform Algorithm 8 for \( n_f = 1 : N_f \)
\[ \text{parfor} \ (n_p = 1 : N_p) \text{ do} \]
\[ n \leftarrow 0 \]
\[ D^{(0)} \leftarrow D_c^{(n_p)} \]
\[ W^{(0)} \leftarrow \text{diag } ([1, 1, \ldots, 1]) \]
\[ \text{while } (n \leq \hat{n}) \text{ do} \]
\[ W^{(n)} \leftarrow \text{diag } \left( \left[ w_1^{(n)}, w_2^{(n)}, \ldots, w_{U_c}^{(n)} \right] \right) \]
\[ X_{\text{red}}^{(n+1)} \leftarrow W^{(n)} D^{T(n)} (D^{(n)} W^{(n)} D^{T(n)} + \lambda^{(n)} I)^{-1} H_{\text{red}} \]
\[ x^{(n+1)} \leftarrow \text{energy } \left( X_{\text{red}}^{(n+1)} \right) \]
\[ n \leftarrow n + 1 \]
\[ \text{end while} \]
\[ \hat{X}^{(n_f)} \leftarrow X_{\text{red}}^{(n_f)} V^{T(n_f)} : \text{transform back to the original dimension. See Eq. 7.6–7.9} \]
\[ \text{end parfor} \]
\[ X_c^{(n_f)} \leftarrow \text{combin } \left( \hat{X}^{(1)}, \hat{X}^{(2)}, \ldots, \hat{X}^{(N_p)} \right)^{(n_f)} \]
\[ X_{[n_f, \ldots]} \leftarrow X_c^{(n_f)} \]

end for

Notes
\text{genDict} \text{ function is used to generate the first combined-resolution dictionary.} \\
\text{rotateGenDict} \text{ function is used to derive the other dictionaries by rotating the coordinates of the generated dictionary around the azimuth axis by an integer multiple of slice angle } \theta_m. \text{ energy function returns the energy of the signals. combin function is used to merge solutions based on individual dictionaries to an uniform high-resolution solution.}
Algorithm 11 The combined algorithm of sparse recovery.

Input

$N_f$ : number of frequencies, $N_t$ : (time windows×hope size), $\Lambda$ : SFT Order

$D_h \in \mathbb{N}^{(\Lambda+1)^2 \times U_h}$ : high-resolution dictionary ($U_h$ : dictionary resolution)

$D_l \in \mathbb{N}^{(\Lambda+1)^2 \times U_l}$ : low-resolution dictionary ($U_l$ : dictionary resolution)

$D_c \in \mathbb{N}^{(\Lambda+1)^2 \times U_c}$ : composite-resolution dictionary ($U_c$ : dictionary resolution)

$\hat{n}$ : maximum number of iterations allowed

$U_{th}$ : minimum dictionary resolution allowed

$W_{th}$ : minimum weight carry forward by the sparse recovery (threshold of the weights)

$\lambda$ : regularization parameter calculated as in Eq. 3.18

$H_{red}$ : dimension reduced form of a time-frequency domain SFT signal

$N_p$ : Number of parallel threads

Output

$X \in \mathbb{C}^{N_f \times U_h \times N_t}$ : the sparse plane-wave solution

Algorithm

$\theta_m = \frac{2\pi}{N_p}$ : slice angle

$D_c = \text{genDict}(D_h, D_l, \theta_m)$

$[D_c^{(1)}, \ldots, D_c^{(N_p)}] = \text{rotateGenDict}(D_c, \theta_m)$

Perform Algorithm 8 for $n_f = 1 : N_f$

parfor ($n_p = 1 : N_p$) do

$n \leftarrow 0$

$D^{(0)} \leftarrow D_c^{(n_p)}$

$W^{(0)} \leftarrow \text{diag}([1, 1, \ldots, 1])$

while ($n \leq \hat{n}$) do

$W^{(n)} \leftarrow \text{diag}([w_1^{(n)}, w_2^{(n)}, \ldots, w_{\text{length}(v)}^{(n)}])$

$W_{\text{norm db}}^{(n)} \leftarrow 10 \log_{10} \left( \frac{\max(W^{(n)})}{\text{max}(W^{(n)})} \right)$

$(v, \tilde{v}) \leftarrow \text{find_and_track} \left( W_{\text{norm db}}^{(n)} \geq W_{th} \right)$

if length($v$) < $U_{th}$ then

break

else

$W^{(n)} \leftarrow W^{(n)}(v)$ : modify the weighting vector

$D^{(n)} \leftarrow D^{(n)}(v)$ : refine the dictionary

end if

$X_{\text{red}}^{(n+1)} \leftarrow W^{(n)} D^{T(n)} \left( D^{(n)} W^{(n)} D^{T(n)} + \lambda^{(n)} I \right)^{-1} H_{\text{red}}$

$x^{(n+1)} \leftarrow \text{energy} \left( X_{\text{red}}^{(n+1)} \right)$

$n \leftarrow n + 1$

end while

$\tilde{X}^{(n_f)} \leftarrow X_{\text{red}}^{(n_f)} V^{T(n_f)}$ : transform back to the original dimension. See Eq. 7.6–7.9

$\tilde{X}^{(n_f)} \leftarrow \text{reconstruct} \left( \tilde{X}^{(n_f)}, \tilde{v}^{(n_f)} \right)$

end parfor
Algorithm 11 The combined algorithm of sparse recovery (continued).

\[
\begin{align*}
X_c^{(n_f)} & \leftarrow \text{combin} \left( \tilde{X}^{(1)}, \tilde{X}^{(2)}, \ldots, \tilde{X}^{(N_p)} \right)^{(n_f)} \\
X[n_f,:,:] & \leftarrow X_c^{(n_f)}
\end{align*}
\]

end for

Notes

**find_and_track** function is used to determine the indexes \( v \) of the weights which are higher than the predefined threshold \( W_{th} \) and track their reference indexes \( \tilde{v} \) to the original dictionary. **reconstruct** function is used to reconstruct the final solution using the indexes \( \tilde{v} \) which are related to the original dictionary. **energy** function returns the energy of the signals. **genDict** function is used to generate the first combined-resolution dictionary. **rotateGenDict** function is used to derive the other dictionaries by rotating the coordinates of the generated dictionary around the azimuth axis by an integer multiple of slice angle \( \theta_m \). **combin** function is used to merge solutions based on individual dictionaries to an uniform high-resolution solution.

7.5 Evaluation of non-uniform dictionary based sparse plane-wave decomposition

In this section, we describe the quality and the speed measuring paradigms which we used. First, we describe an experiment which is done for visually compare the results obtained by the general and new sparse recovery methods. We tested the algorithms in both simulated and real sound scenes. Next, we describe 3 sets of experiments which are conducted in the anechoic, reverberant and real conditions. Finally, we describe 2 evaluation metrics to measure the quality/accuracy of the sparse plane-wave decomposition methods.

7.5.1 Visual comparison of the quality of the results

Now we describe an experiment which is done for visually compare the results obtained by the general and new sparse recovery methods. We set 6 sources in the space as shown in the Fig. 7.7. The locations of each source are given in Table 7.2. As shown in Fig. 7.7, source 1 is isolated which should be easy to localize. The sources 2 and 3 are located close to each other such that it is challenging to separate the sources. Similarly, sources 4, 5 and 6 also located close to each other as triangular positioning such that energies may localize closer to centroid in consequence of poor localization. Further, the cluster of sources 2-3 and 4-5-6 are located around -90° and 90° azimuth respectively where it will be a challenge for dictionary subdividing and combined methods for source localizing as the sources are located on the boundary of the high-resolution dictionary. The speech sound sources used in this experiment are selected from a subset of the TIMIT database. The speech signals are sampled at 16 kHz and the length of the signals is 3 seconds (i.e., 48000 samples).
Figure 7.7: The positioning of the sound sources to visually compare the quality of the sparse recovery performed using uniform and non-uniform dictionaries.

Table 7.2: The positions of the sound sources which are presented in Fig. 7.7

<table>
<thead>
<tr>
<th>Sound source</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azimuth (deg.)</td>
<td>-120.20</td>
<td>-68.40</td>
<td>-93.56</td>
<td>109.76</td>
<td>76.11</td>
<td>100.64</td>
</tr>
<tr>
<td>Elevation (deg.)</td>
<td>-44.69</td>
<td>33.60</td>
<td>34.45</td>
<td>-15.75</td>
<td>-20.52</td>
<td>-52.80</td>
</tr>
</tbody>
</table>
7.5.2 The description of the three experiment conditions

Now we describe 3 sets of experiments which are conducted in the anechoic, reverberant and real conditions. The first set of experiments is conducted in the anechoic conditions. In these experiments, we examine the robustness of the proposed technique in terms of parameters: 1) the number of the sound sources, 2) the signal to noise ratio (SNR), 3) the threshold of the weights for omitting the directions, and 4) the number of subdivisions of the dictionary. The number of sources is varied from 2 to 6. As the number of SFT signals (order-2 SFT signals) is 9, only the under-determined cases (i.e., number of SFT signals < 9) are considered. The test sources are placed in fixed positions randomly in space to examine the accuracy of the obtained acoustic energy maps. The source directions are not in-lined with the dictionary directions. The speech sound sources used in this experiment are selected from a subset of the TIMIT database. The speech signals are sampled at 16 kHz and the length of the signals is 3 seconds (i.e., 48000 samples). The measurement noise is modeled using uncorrelated Gaussian white noise and the SNR is set as 30dB. The threshold of the weights for refining the dictionary is varied as -45dB, -30dB, -15dB and -2dB. The algorithms are executed on a multiprocessor platform which has 2 Intel Xeon E5-2643 processors. Intel Xeon E5-2643 processor has 4 cores which make altogether 8 cores in the platform. Mathlab is used as the parallel-processing environment which can call parfor loop to execute independent programs in parallel. Therefore, the number of subdivisions of the dictionary is varied from 2 to 8 in steps of 2. We conduct 50 trials for each combination of these parameters. In each trial, new observation signals (the order-2 SFT signals) are generated. The dictionaries are created using Lebedev grids [73]. The initial high-resolution dictionary in the dictionary-refining method is created using a Lebedev grid of 770 directions. The same Lebedev grid is used to create the high-resolution dictionary in the dictionary-subdividing method. The low-resolution dictionary in the dictionary-subdividing method is created using a Lebedev grid of 86 directions.

The second set of experiments is conducted using simulated reverberant conditions. The experiment procedures under reverberant conditions are similar to the experiments under anechoic conditions. However, in here we consider a room having reverberation as the surrounding environment/space. The room with different direct-to-reverberant energy ratios (DRR) was simulated using MCRoomSim [131] tool, a multichannel room acoustics simulator. The DRR is varied as 0.3, 0.5, 0.7 and 0.9 where higher DRR means higher energy in the direct component or lower reverberation. Fig shows the size and reverberant characteristics of the room and the location of the spherical microphone array in it to generate the SFT signals. The number of sources is varied from 2 to 6 in steps of 2. The sources surround the SMA at a distance of 1 m and their directions are chosen randomly. Similar to
anechoic conditions the measurement noise is modeled using uncorrelated Gaussian white noise and the SNR is set as 30dB. The threshold of the weights for refining the dictionary is varied as -45dB, -30dB, -15dB and -2dB. The algorithms are performed on the same platform described in anechoic experiments. The number of subdivisions of the dictionary is varied from 2 to 8 in steps of 2. We conducted 50 trials for each combination of the number of sources, DRR values, thresholds and subdivisions of the dictionary. In each trial, new observation signals (the order-2 SFT signals) are generated. The dictionaries are generated and used similar to the anechoic experiments. The initial high-resolution dictionary in the dictionary-refining method is created using a Lebedev grid of 770 directions. The same Lebedev grid is used to create the high-resolution dictionary in the dictionary-subdividing method. The low-resolution dictionary in the dictionary-subdividing method is created using a Lebedev grid of 86 directions.

The third set of experiments is conducted using measured room impulse responses. The measured microphone impulse responses are used to generate the SFT signals. The room has the dimensions of 14×8×3 m. The dual-concentric microphone array which is shown in Figure 3.1 is used. In the microphone array, each sphere contains 32 microphones. The inner sphere has radius 28 mm, and the outer sphere has radius 95.2 mm. The impulse responses were measured using a Talkbox that was moved to different locations in a room. TalkBox is an acoustic signal generator for speech intelligibility measurements and simulates a human talker. In the experiment, impulse responses are measured for 3 sound sources located at (0°,0°), (0°,45°), and (0°,225°), respectively. The threshold of the weights for refining the dictionary is varied as -45dB, -30dB, -15dB and -2dB. The algorithms are performed on the same platform described in anechoic experiments. The number of subdivisions of the dictionary is varied from 2 to 8 in steps of 2. The dictionaries are generated and used similarly to the anechoic experiments. The initial high-resolution dictionary in the dictionary-refining method is created using a Lebedev grid of 770 directions. The same Lebedev grid is used to create the high-resolution dictionary in the dictionary-subdividing method. The low-resolution dictionary in the dictionary-subdividing method is created using a Lebedev grid of 86 directions.

7.5.3 The quality evaluation metrics

In this section we present two evaluation metrics to measure the quality/accuracy of the sparse plane-wave decompositions methods:

1. Measure of mismatching between the estimated acoustic energy map and the true acoustic energy map (energy-map mismatch),
2. Measure of angular error between the estimated source locations and the true source locations (angular-error estimation).

Following is the detailed description of these metrics.

### 7.5.3.1 Energy-map mismatch

In order to quantitatively evaluate the accuracy of the acoustic energy maps, we compare the true locations of the sound sources and the corresponding powers (map one) with the map obtained by the plane-wave decomposition of non-uniform dictionary method, which consists of dictionary direction and the corresponding power values (map two). This introduces two types of mismatch errors:

1. Error in the estimation of the source powers
2. Error in the estimation of the source directions

In [93], the authors define a measure of the mismatch between two energy maps that is inspired from tools used in differential geometry. The energy map mismatch, $E$, between maps 1 and 2, is given by:

$$ E = \frac{K_{11} + K_{22} - 2K_{12}}{K_{11} + K_{22}} $$

(7.13)

where $K_{ij}$ is given by:

$$ K_{ij} = \sum_{q=1}^{Q} \sum_{p=1}^{P} \rho_q^{(i)} \rho_p^{(j)} k\left((\theta_q, \phi_q)^{(i)}, (\theta_p, \phi_p)^{(j)}\right), $$

(7.14)

where $(\theta_q, \phi_q)^{(i)}$ and $\rho_q^{(i)}$ denote the $q$-th direction in map $i$ and the corresponding power value, respectively. The function $k(\cdot, \cdot)$ is a spatial kernel function defined as:

$$ k\left((\theta_q, \phi_q)^{(i)}, (\theta_p, \phi_p)^{(j)}\right) = \max\left(1 - \frac{\angle\left((\theta_q, \phi_q)^{(i)}, (\theta_p, \phi_p)^{(j)}\right)}{\pi/12}, 0\right), $$

(7.15)

where $\angle\left((\theta_q, \phi_q)^{(i)}, (\theta_p, \phi_p)^{(j)}\right)$ denotes the angle between directions $(\theta_q, \phi_q)^{(i)}$ and $(\theta_p, \phi_p)^{(j)}$. The value of $k\left((\theta_q, \phi_q)^{(i)}, (\theta_p, \phi_p)^{(j)}\right)$ decreases linearly from 1 to 0 when the angular distance between directions $(\theta_q, \phi_q)^{(i)}$ and $(\theta_p, \phi_p)^{(j)}$ increases from 0 to $\pi/12$ radians ($15^\circ$), and is equal to 0 when the angular distance is larger than $\pi/12$. Note that the value $\pi/12$ was chosen arbitrarily.

The value of mismatch error is between 0 and 1. The smaller values indicate a better estimation of acoustic energy maps. The values $K_{11}$ and $K_{22}$ in Equation 7.13 are the autocorrelations of map 1 and 2, respectively. $K_{12}$ represents the crosscorrelation between
maps 1 and 2. When the two maps are exactly identical (i.e., the directions and powers are the same), the maps are perfectly correlated and $K_{11} = K_{22} = K_{12}$. Thus, $E$ is equal to 0 (no mismatch). On the contrary, when there is no overlap between the two maps, which happens when every direction in map 2 is more than $\pi/12$ radians away from every direction on map 1, $K_{12} = 0$ and thus $E$ is equal to 1 (complete mismatch). A value of $E$ close to 0 can be obtained only if the peaks and their power values in the estimated map are sufficiently close to that in the true map. Experimentally we have found, the energy maps having mismatch error less than 0.4 provide acceptable quality. Therefore in the simulations, 0.4 mismatch error is kept as the reference for minimum acceptable quality.

7.5.3.2 Angular-error estimation

The angular error is defined as the angle between the direction $(\theta_i, \phi_i)$ of the true source and the resolved source direction $(\hat{\theta}_i, \hat{\phi}_i)$ given by:

$$\text{Angular Error} = \angle((\theta_i, \phi_i), (\hat{\theta}_i, \hat{\phi}_i)).$$  \hspace{1cm} (7.16)

To resolve the source direction, first neighborhood dictionary directions which having angle to the true direction $(\theta_i, \phi_i)$ less than 20° are scanned. Then the source direction $(\hat{\theta}_i, \hat{\phi}_i)$ is selected as the nearest direction from the true direction having 80% of the energy of the highest energy direction in this neighborhood. If a source direction cannot be resolved, the angular error is set to 20°. When the angular error of a particular resolved source is increasing, the source localization in the energy-map getting erroneous.

7.6 Results and Discussion

In this section, we discuss the quality and the performance of the 3 non-uniform dictionary based sparse recovery methods.
Table 7.3: The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different reverberant sound scenes containing 2 to 12 sources. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The dictionary is subdivided into 4 in the subdivision and combined methods. The DRR is 0.7 and SNR is 30dB.

Reverberant, Threshold: -30dB, Subdivisions: 4, DRR: 0.7, SNR: 30dB

Sources: 2

Uniform Dictionary

Dictionary Refining

Dictionary Subdividing

Combined Method

Sources: 4

Uniform Dictionary

Dictionary Refining

Continued on next page
Table 7.3 – Continued from previous page

Reverberant, Threshold: -30dB, Subdivisions: 4, DRR: 0.7, SNR: 30dB

<table>
<thead>
<tr>
<th>Sources</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Uniform Dictionary</td>
<td>Dictionary Refining</td>
</tr>
<tr>
<td>8</td>
<td>Uniform Dictionary</td>
<td>Dictionary Refining</td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.3 – Continued from previous page

Reverberant, Threshold: -30dB, Subdivisions: 4, DRR: 0.7, SNR: 30dB

<table>
<thead>
<tr>
<th>Sources</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Uniform Dictionary</td>
<td>Dictionary Refining</td>
</tr>
<tr>
<td></td>
<td>Dictionary Subdividing</td>
<td>Combined Method</td>
</tr>
<tr>
<td>12</td>
<td>Uniform Dictionary</td>
<td>Dictionary Refining</td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.3 – Continued from previous page

Reverberant, Threshold: -30dB, Subdivisions: 4, DRR: 0.7, SNR: 30dB

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
</tr>
</thead>
<tbody>
<tr>
<td>-30dB</td>
<td><a href="#">Graph</a></td>
<td><a href="#">Graph</a></td>
</tr>
</tbody>
</table>

Dictionary Subdividing  Combined Method

Table 7.4: The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different anechoic sound scenes by varying the threshold of the weights for refining the dictionary as -60dB, -45dB, -30dB, -15dB and -2dB. The dictionary is subdivided into 4 in the subdivision and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is 30dB.

Anechoic, Sources: 6, Subdivisions: 4, SNR: 30dB

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
</tr>
</thead>
<tbody>
<tr>
<td>-60dB</td>
<td><a href="#">Graph</a></td>
<td><a href="#">Graph</a></td>
</tr>
</tbody>
</table>

Dictionary Subdividing  Combined Method

Continued on next page
<table>
<thead>
<tr>
<th>Threshold</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-45dB]</td>
<td><img src="image1" alt="Graph" /></td>
<td><img src="image2" alt="Graph" /></td>
<td><img src="image3" alt="Graph" /></td>
<td><img src="image4" alt="Graph" /></td>
</tr>
<tr>
<td>[-30dB]</td>
<td><img src="image5" alt="Graph" /></td>
<td><img src="image6" alt="Graph" /></td>
<td><img src="image7" alt="Graph" /></td>
<td><img src="image8" alt="Graph" /></td>
</tr>
</tbody>
</table>

*Continued on next page*
Table 7.4 – Continued from previous page

Anechoic, Sources: 6, Subdivisions: 4, SNR: 30dB

<table>
<thead>
<tr>
<th>Method</th>
<th>Threshold</th>
<th>Power [dB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Dictionary</td>
<td>-15dB</td>
<td></td>
</tr>
<tr>
<td>Dictionary Refining</td>
<td>-2dB</td>
<td></td>
</tr>
<tr>
<td>Dictionary Subdividing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combined Method</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graphs showing power distribution for different methods](image_url)
Table 7.5: The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different anechoic sound scenes by varying the subdivisions of the dictionary into 2, 4 and 8 in the subdivision and combined methods. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is 30dB.

Anechoic, Sources: 6, Threshold: -30dB, SNR: 30dB

<table>
<thead>
<tr>
<th>Subdivisions</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><img src="image1" alt="Uniform Dictionary" /></td>
<td><img src="image2" alt="Dictionary Refining" /></td>
<td><img src="image3" alt="Dictionary Subdividing" /></td>
<td><img src="image4" alt="Combined Method" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image5" alt="Uniform Dictionary" /></td>
<td><img src="image6" alt="Dictionary Refining" /></td>
<td><img src="image7" alt="Dictionary Subdividing" /></td>
<td><img src="image8" alt="Combined Method" /></td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.5 – Continued from previous page

Anechoic, Sources: 6, Threshold: -30dB, SNR: 30dB

<table>
<thead>
<tr>
<th>Subdivisions</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Dictionary</td>
<td>Dictionary Refining</td>
<td>Dictionary Subdividing</td>
</tr>
</tbody>
</table>
Table 7.6: The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different anechoic sound scenes. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The subdivision of the dictionary is 4 in the subdivision and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is varied as 60dB, 30dB, 15dB, 5dB and 0dB.

<table>
<thead>
<tr>
<th>SNR</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>60dB</td>
<td><img src="image1" alt="Uniform Dictionary" /></td>
<td><img src="image2" alt="Dictionary Refining" /></td>
<td><img src="image3" alt="Dictionary Subdividing" /></td>
<td><img src="image4" alt="Combined Method" /></td>
</tr>
<tr>
<td>30dB</td>
<td><img src="image5" alt="Uniform Dictionary" /></td>
<td><img src="image6" alt="Dictionary Refining" /></td>
<td><img src="image7" alt="Dictionary Subdividing" /></td>
<td><img src="image8" alt="Combined Method" /></td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.6 – Continued from previous page

Anechoic, Sources: 6, Threshold: -30dB, Subdivisions: 4

<table>
<thead>
<tr>
<th>SNR</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>15dB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5dB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.6 – Continued from previous page

Anechoic, Sources: 6, Threshold: -30dB, Subdivisions: 4

<table>
<thead>
<tr>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>184</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
</tr>
</thead>
<tbody>
<tr>
<td>184</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>184</td>
<td></td>
</tr>
</tbody>
</table>
Table 7.7: The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for different reverberant sound scenes. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The subdivision of the dictionary is 4 in the subdivision and combined methods. There are 6 sources in the environment as described in Section 7.5.1. The SNR is 30dB. The DRR is varied as 0.9, 0.7, 0.5 and 0.3.

Reverberant, Sources: 6, Threshold: -30dB, Subdivisions: 4, SNR: 30dB

<table>
<thead>
<tr>
<th>DRR</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td><img src="Uniform_Dictionary_0.9" alt="Image" /></td>
<td><img src="Dictionary_Refining_0.9" alt="Image" /></td>
<td><img src="Dictionary_Subdividing_0.9" alt="Image" /></td>
<td><img src="Combined_Method_0.9" alt="Image" /></td>
</tr>
<tr>
<td>0.7</td>
<td><img src="Uniform_Dictionary_0.7" alt="Image" /></td>
<td><img src="Dictionary_Refining_0.7" alt="Image" /></td>
<td><img src="Dictionary_Subdividing_0.7" alt="Image" /></td>
<td><img src="Combined_Method_0.7" alt="Image" /></td>
</tr>
</tbody>
</table>

*Continued on next page*
Table 7.7 – Continued from previous page

Reverberant, Sources: 6, Threshold: -30dB, Subdivisions: 4, SNR: 30dB

<table>
<thead>
<tr>
<th>Method</th>
<th>DRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Dictionary</td>
<td>0.5</td>
</tr>
<tr>
<td>Dictionary Refining</td>
<td></td>
</tr>
<tr>
<td>Dictionary Subdividing</td>
<td></td>
</tr>
<tr>
<td>Combined Method</td>
<td></td>
</tr>
</tbody>
</table>

Dictionary Subdividing

Combined Method

Uniform Dictionary

Dictionary Refining

Dictionary Subdividing

Combined Method

Continued on next page
Table 7.7 – Continued from previous page

Reverberant, Sources: 6, Threshold: -30dB, Subdivisions: 4, SNR: 30dB

<table>
<thead>
<tr>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Image" /></td>
<td><img src="image2" alt="Image" /></td>
</tr>
</tbody>
</table>

Table 7.8: The acoustic energy maps obtained by uniform and non-uniform dictionary-based sparse recovery methods for real sound scenes generated using the measured room impulse responses. There are 1 to 3 sources located in the room. The threshold of the weights for refining the dictionary is -30dB in dictionary-omitting and combined methods. The dictionary is subdivided into 4 in the subdivision and combined methods.

<table>
<thead>
<tr>
<th>Real, Threshold: -30dB, Subdivisions: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3" alt="Image" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sources: 1</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5" alt="Image" /></td>
<td><img src="image6" alt="Image" /></td>
<td></td>
</tr>
</tbody>
</table>

| ![Image](image7) | ![Image](image8) |

<table>
<thead>
<tr>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image9" alt="Image" /></td>
<td><img src="image10" alt="Image" /></td>
</tr>
</tbody>
</table>

Continued on next page
Table 7.8 – Continued from previous page

Real, Threshold: -30dB, Subdivisions: 4

<table>
<thead>
<tr>
<th>Sources</th>
<th>Uniform Dictionary</th>
<th>Dictionary Refining</th>
<th>Dictionary Subdividing</th>
<th>Combined Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><img src="image1" alt="Image" /></td>
<td><img src="image2" alt="Image" /></td>
<td><img src="image3" alt="Image" /></td>
<td><img src="image4" alt="Image" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image5" alt="Image" /></td>
<td><img src="image6" alt="Image" /></td>
<td><img src="image7" alt="Image" /></td>
<td><img src="image8" alt="Image" /></td>
</tr>
</tbody>
</table>

188
Now we examine the possible acceleration of sparse recovery with the non-uniform dictionary-based methods without compromising the quality. The quality of the results is evaluated using the energy map mismatch and angular error estimation which are presented in Section 7.5.3.1 and Section 7.5.3.2 respectively. In this context, we would like to answer the following questions in terms of quality and performance (i.e., acceleration).

1. What will happen when increasing the subdivisions of the dictionary?

2. What will happen when increasing the threshold of the energy of refining the dictionary?

3. What will happen when increasing the number of sound sources?

First in Figure 7.8, we examine the effects on the accuracy and acceleration as a cause of changing the number of subdivisions of the dictionary and the energy threshold of refining the dictionary. The figure is related to the results of the experiments performed under anechoic conditions. There are 3 subfigures and in each subfigure, the number of subdivisions of the dictionary is increased from 2 to 8 along the x-axis. For a given number of subdivisions, several thresholds of the dictionary refining are tested from -45dB to -2dB. The SNR is set constant as -30dB. The number of sources in the environment is 4. Let’s focus on the subfigures 7.8(a) and 7.8(b) which present estimations about the accuracy of the energy map. When increasing the number of subdivisions of the dictionary or/and the energy threshold of refining the dictionary both degrade the quality of the energy map as it increases the mismatch and angular error. This is because, in both methods, the information is reduced in the sparse recovery by omitting the directions. Now we focus the subfigure 7.8(c) which is about the performance of the sparse recovery. As per the results, when increasing both the number of subdivisions of the dictionary and the energy threshold of refining the dictionary, the algorithm accelerates. Therefore, the combined method is effective, and the dictionary refining and subdividing methods complement each other for increasing the performance. Then we examine the effects on the accuracy and acceleration as a cause of changing the number of sound sources in the environment and the energy threshold of refining the dictionary. As shown in subfigures 7.9(a) and 7.9(b) the quality of the energy map degrades when increasing the number of sound sources. We have already discussed the degradation of quality when increasing the energy thresholds of the dictionary refining. Regarding the subfigure 7.9(c) the acceleration decreases when increasing the number of sources. Therefore, the increase the number of sound sources reduces the quality and the performance both. Next, we examine the effects on the accuracy and acceleration
as a cause of changing the number of sound sources in the environment and the number of subdivisions of the dictionary. The results in Figure 7.10 can be interpreted similarly to Figure 7.9 where the increment of the number of sound sources reduces the quality and the performance. Further, the acceleration gradually increases with the number of subdivisions of the dictionary. Therefore, apart from the number of subdivisions of the dictionary and the energy threshold of refining the dictionary, the number of sound sources also impact on the accuracy and the performance of the sparse recovery.
Figure 7.8: The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the energy threshold of refining the dictionary. The evaluation is done in anechoic conditions. Along the x-axis in a selected subfigure, the number of dictionaries is increased from 2 to 8. For a given number of dictionaries, several thresholds of the dictionary refining are tested from -45dB to -2dB. The SNR is set constant as -30dB. The number of sources in the environment is 4.
Figure 7.9: The change of the accuracy and acceleration against the energy threshold of refining the dictionary and the number of sound sources in the environment. The evaluation is done in anechoic conditions with the dictionary-refining method. Along the x-axis in a selected subfigure, several thresholds of the dictionary refining are tested from -45dB to -2dB. For each threshold, the number of sources in the environment is changed from 2 to 6. The SNR is set constant as -30dB.
Figure 7.10: The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the number of sound sources in the environment. The evaluation is done in anechoic conditions with the dictionary-subdividing method. Along the x-axis in a selected subfigure, the number of dictionaries is increased from 2 to 8. For a given number of dictionaries, the number of sources in the environment is changed from 2 to 6. The SNR is set constant as -30dB.
Now we examine the same scenarios under reverberant conditions. We present Figure 7.11, Figure 7.12, and Figure 7.13 which are similar to Figure 7.8, Figure 7.9, and Figure 7.10 in anechoic conditions. When analyzing the figures, it can be seen the accuracy and acceleration behave in similar pattern as under anechoic conditions. Therefore, we do not describe the same patterns again. However, few exceptions can be seen which are described now. First focus on Figure 7.11. The accuracy of the resulting energy map under reverberant conditions is lower than the result under anechoic conditions. In other words, the reverberation degrades the quality of the source localization in the energy map. It is obvious that the energy maps generated with reverberant energy are not accurate as it is done under anechoic conditions. However, unlike in anechoic conditions, when the energy threshold of refining the dictionary is increased to -2dB, the accuracy of the energy map increases compared to the result corresponding to the energy threshold of -15dB (see subfigure 7.12(a)). Therefore, we assume with a higher energy threshold of refining the dictionary, the reverberant energy which causes inaccuracies is omitted. Furthermore regarding the acceleration, the reverberant results shows slightly better acceleration compared to the anechoic results (compare subfigures 7.8(c) and 7.11(c)). The behavior of the quality and performance presented in Figure 7.12 and Figure 7.13 can be understood in same way.
Figure 7.11: The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the energy threshold of refining the dictionary. The evaluation is done in reverberant conditions. Along the x-axis in a selected subfigure, the number of dictionaries is increased from 2 to 8. For a given number of dictionaries, several threshold of the dictionary refining are tested from -45dB to -2dB. The SNR is set constant as -30dB. The direct-to-reverberant energy ratios (DRR) is set constant as 0.5. The number of sources in the environment is 4.
Figure 7.12: The change of the accuracy and acceleration against the energy threshold of refining the dictionary and the number of sound sources in the environment. The evaluation is done in reverberant conditions with dictionary-refining method. Along the x-axis in a selected subfigure, several threshold of the dictionary refining are tested from -45dB to -2dB. For each threshold, the number of sources in the environment is changed from 2 to 6. The SNR is set constant as -30dB. The direct-to-reverberant energy ratios (DRR) is set constant as 0.5.
Figure 7.13: The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the number of sound sources in the environment. The evaluation is done in reverberant conditions with dictionary-subdividing method. Along the x-axis in a selected subfigure, the number of dictionaries is increased from 2 to 8. For a given number of dictionaries, the number of sources in the environment is changed from 2 to 6. The SNR is set constant as -30dB. The direct-to-reverberant energy ratios (DRR) is set constant as 0.5.
Now we examine the same scenarios for real measurements of the sound scene. We present Figure 7.15, Figure 7.16 and Figure 7.17 which are similar to Figure 7.11, Figure 7.12 and Figure 7.13 in reverberant conditions. In here we only focus on three sound sources which are located at \((0^\circ,0^\circ)\), \((0^\circ,45^\circ)\), and \((0^\circ,225^\circ)\). The experiment setup is described in Section 7.5.2. The visual inspection of the corresponding energy maps is done in Figure 7.8. The same results which corresponding to 3 sound sources are presented in Figure 7.14. As per the results (i.e., both energy maps and the graphs) the non-uniform dictionary based sparse recovery is effective for source localization.

(a) Sparse recovery using an uniform dictionary

(b) Sparse recovery using the dictionary-refining method. Threshold: -30dB.

(c) Sparse recovery using the dictionary-subdividing method. Subdivisions: 4.


Figure 7.14: The energy map of uniform and non-uniform dictionary based sparse recovery when applied for real measurements of the sound scene. There are 3 sound sources present in the environment. Threshold: -30dB, Subdivisions: 4.
Figure 7.15: The change of the accuracy and acceleration against the number of subdivisions of the dictionary and the energy threshold of refining the dictionary. The evaluation is done for real measurements of the sound scene. Along the x-axis in a selected subfigure, the number of dictionaries is increased from 2 to 8. For a given number of dictionaries, several threshold of the dictionary refining are tested from -45dB to -2dB. The number of sources in the environment is 3.
Figure 7.16: The change of the accuracy and acceleration against the number of sound sources in the environment. The evaluation is done for real measurements of the sound scene with dictionary-refining method. Along the x-axis in a selected subfigure, several threshold of the dictionary refining are tested from -45dB to -2dB. For each threshold, the number of sources in the environment is changed from 1 to 3.
Figure 7.17: The change of the accuracy and acceleration against the number of subdivisions of the dictionary. The evaluation is done for real measurements of the sound scene with dictionary-subdividing method. Along the x-axis in a selected subfigure, the number of dictionaries is increased from 2 to 8. For a given number of dictionaries, the number of sources in the environment is changed from 1 to 3.
7.7 Conclusion

In this chapter, we explored non-uniform spatial dictionaries based sparse recovery techniques for plane-wave decomposition. We set the front hemisphere of space as the region of interest and ignored the back hemisphere of space. Then the spatial dictionary had a much greater resolution in the front hemisphere of space compared to the back hemisphere of space. We discovered that even using dictionaries with relatively low spatial resolution in the back hemisphere, accurate energy maps for the front hemisphere of space could still be obtained. We identified the spatial resolution of the back hemisphere of space cannot be reduced below the resolution of a uniform dictionary having 110 directions, without degrading the accuracy of the energy map for the front hemisphere. Based on the non-uniform dictionary based concept, we presented 3 non-uniform dictionary based sparse-recovery algorithms which can be used for accelerating the sparse recovery process. They are 1) dictionary refining method, 2) dictionary subdividing method and 3) combined method of 1st and 2nd methods. The non-uniform dictionaries can be used to either reduce the computational complexity and/or subdivide the problem such that they can be accelerated on parallel computing architectures. The simulations proved the non-uniform dictionaries based sparse recovery can provide 2 to 3 times acceleration over the uniform dictionary based sparse recovery with acceptable quality. As per the simulations, the new methods are robust to the reverberation and the presence of diffuse noise.
Chapter 8

Conclusions

This thesis considers the design, development and evaluation of technologies to support sparse plane-wave decomposition for spherical microphone arrays. The motivation for the research was to accelerate the sparse recovery plane-wave decomposition. In the following, we recapitulate the outcomes for this thesis.

8.1 Summary

- Chapter 5 presented the development of a scalable FPGA design model for SFT. The model considers the number of microphones, SFT signals and affordable cost of FPGA as the input and provides the design of resource optimize and cost-effective FPGA architecture as the output. Since the SFT algorithm is highly parameterizable, the model makes the design process easy and fast facilitates the FPGA design process. Using the model we could identify:

1. The FPGA devices XC6SLX4, XC6SLX9 and XC7A15T cannot be used for SFT as their resources are not sufficient.

2. In most SFT architectures, the highest utilized resource is BRAMs. However, in low-cost small FPGAs (e.g., XC6SLX16, XC6SLX25, XC6SLX25T, XC7A35T, XC7A50T) flip-flops (FFs) becomes the highest utilized resource.

3. The constraining factor of the SFT can be either resource or speed. The architectures implemented on Virtex-6 FPGAs are limited by the memory bandwidth of the filter coefficient access. In contrast, the architectures implemented on Artix/Kintex/Virtex-7 FPGAs are limited by the available BRAM resources. Since Virtex-7 FPGAs support higher memory bandwidth than Virtex-6 FPGAs, the SFT on Artix/Kintex/Virtex-7 is not limited by the memory bandwidth of the filter coefficient access.
4. The maximum number of microphones Virtex-6 devices can support is 64 while Artix/Kintex/Virtex-7 devices can support up to 256. As stated, Virtex-6 SFT architectures are I/O bound while Virtex-6 architectures are resource bound.

5. The maximum number of SFT signals Virtex-6 devices can support is 64 ($7^{th}$-order) which is limited by the maximum number of supported microphones (i.e., 64). In contrast, Artix/Kintex/Virtex-7 devices can support up to 100 SFT signals ($9^{th}$-order) and 256 microphones.

- **Chapter 6** presented an analysis of the computational complexity of the sparse-recovery algorithm and investigation of the performance of the sparse-recovery algorithm executed on selected parallel computing platforms (i.e., Chip-multiprocessor, Multiprocessor, GPU, Manycore). Sparse recovery is performed in the frequency domain and frequency-specific sparse-recovery problems are assigned to thread resources to solve them in parallel. We examined the scalability and the acceleration when performing the sparse recovery in the frequency domain by assigning frequency-related problems to threads. Based on the results, following conclusions are made.

1. Analytically, the least computational complex method of solving an IRLS problem is by Cholesky decomposition.

2. To achieve the best performance on multithreaded architectures, the number of IRLS problems should be an integer multiple of the total number of hardware threads in the architecture. Regarding the GPU, the number of IRLS problems should be an integer multiple of the active-thread blocks.

3. When assigning IRLS problems on an architecture, the workload on cores and hardware threads should be balanced for efficiency.

4. Reducing the dictionary resolution improves the rate of solving the IRLS problems by reducing the computational complexity of the algorithm.

5. The throughput (i.e., peak effective computational rate) of solving frequency-domain IRLS problems is inversely proportional to the dictionary resolution. Using the proportionality constant of a given architecture, it is possible to estimate the throughput of solving the IRLS problems for an arbitrary dictionary resolution. In Table 6.5 proportionality constants of the selected architectures are presented.

6. Base on the throughput, Table 6.4 presents relative performances of the selected architectures against Intel Core-i3 370M processor. As per the table, Intel Xeon
Phi 5110P coprocessor delivers the best performance which is 52.87 faster than the Intel Core-i3 370M processor.

In summary, multi-threaded architectures are useful to accelerate the sparse recovery by solving IRLS problems in parallel. The reduction of the dictionary resolution increases the rate of solving IRLS problems.

- **Chapter 7** explored non-uniform spatial dictionaries based sparse recovery techniques for plane-wave decomposition. We set the front hemisphere of space as the region of interest and ignored the back hemisphere of space. Then the spatial dictionary had a much greater resolution in the front hemisphere of space compared to the back hemisphere of space. We discovered that even using dictionaries with relatively low spatial resolution in the back hemisphere, accurate energy maps for the front hemisphere of space could still be obtained. We identified that the spatial resolution of the back hemisphere of space cannot be reduced below the resolution of a uniform dictionary having 110 directions, without degrading the accuracy of the energy map for the front hemisphere. Based on the non-uniform dictionary based concept, we presented 3 non-uniform dictionary based sparse-recovery algorithms which can be used for accelerating the sparse recovery process. They are 1) dictionary refining method, 2) dictionary subdividing method and 3) combined method of 1st and 2nd methods. The non-uniform dictionaries can be used to either reduce the computational complexity and/or subdivide the problem such that they can be accelerated on parallel computing architectures. The simulations proved the non-uniform dictionaries based sparse recovery can provide 2 to 3 times acceleration over the uniform dictionary based sparse recovery with acceptable quality. As per the simulations, the new methods does not seem less robust to the reverberation and the presence of diffuse noise.

Based on the outcomes, the sparse plane-wave decomposition for spherical microphone arrays can be accelerated by using an appropriate FPGA and a multithreaded architecture. The spherical Fourier transformation (SFT) should be performed on an FPGA which spare full resources of the multithreaded architecture to perform the sparse recovery. The FPGA also helps to integrate the microphones for data acquisition. The sparse recovery should be performed in the frequency domain and different frequencies should be processed in parallel using threads in the multithreaded architecture. The speed of solving the sparse recovery problem can be further improved using non-uniform dictionary based sparse recovery algorithms which reduce the computational complexity and exploit the parallelism of the computation. We presented dictionary refining and dictionary subdividing techniques to produce non-uniform dictionaries. The non-uniform dictionaries based sparse recovery can
provide 2 to 3 times acceleration over the uniform dictionary based sparse recovery with acceptable quality.
Appendix A

I2C Protocol

I2C is a popular communication protocol in embedded systems, because of two-wire interface which provides easy and flexible integration with multiple devices. It is used in inter-chip and intra-chip low bandwidth communication. I2C protocol can be operated in five modes with different clock speeds. They are standard-mode (max. 100 kHz), fast-mode (max. 400 kHz), fast-mode-plus (max. 1MHz), high-speed-mode (max. 3.4 MHz) and ultra-fast-mode (5 MHz). In this implementation I2C interface is operated in standard-mode.

The two wires of the I2C protocol are SCL and SDA which transfer serial clock and data respectively. Important specifications of these two wires are, they are bidirectional open-drain lines. Therefore, devices which communicate via I2C cannot drive the bus high and when the high state occurs the bus line will be floating. To overcome this issue, two external pull-up resistors should be used to drive the bus high when floating. When determining the required pull-up resistors, I2C bus impedance should be considered to keep the proper shape of the digital signals. The number of devices connected to the bus is only limited by the total allowed bus capacitance of 400 pF.

The standard communication on the I2C bus between a master and slave is composed of four steps. They are START, Slave address, Data transfer and STOP. The timing diagram of the data transfer on the bus is shown in Fig. A.1. A device can start the data transmission with a START condition and terminate with a STOP condition. To initiate the START condition master should stay until the bus is free. The START condition is defined by a high-to-low transition on SDA while SCL is high, and STOP is a low-to-high transition on the SDA line while SCL is high. Between the START and STOP conditions of the bus, data are transferred synchronously with the SCL clock. In the transmission, the most significant bit transmits first and 8-bits of data followed by an acknowledge bit.

Regarding the addressing of communicating devices, the I2C protocol defines 7-bit and 10-bit addressing schemes for connected devices. Depending on the addressing technique, a slight modification is done to the protocol as it is not possible to transmit 10 bits in a
Figure 2 illustrates how the definitions of: (a) the bus free state, and (b) the times when SDA and SCL may change relative to each other, ensure that the START and STOP conditions are not confused as data.

Each transfer on the IIC bus consists of nine clock pulses on SCL to move eight bits of data and one acknowledge bit. Master and slave transmitter send data with the most significant bit first (MSB).

After providing data for the eight clock period, the (master | slave) transmitter releases the SDA line during the acknowledgement clock period to permit the receiver to transfer a 1-bit acknowledgment.

If a slave-receiver issues a not-acknowledge (by releasing the SDA signal during the acknowledgement clock period) this indicates that the slave-receiver was unable to accept the prior 8-bits transferred (consisting of address or data bits.) Note that after a byte of data is transferred the slave (receiver | transmitter) has the unique capability to throttle the transfer by keeping the SCL line in its low state by actively pulling the SCL line low for an arbitrary period of time. This ability allows it time to determine internally what value it should place on the SDA line for the acknowledgement.

Note:
1. The wired-and nature of the bus signals and each device’s pull-low or release output capability permit bi-directional data transfer.
2. This means the master and slaves in the system cooperatively determine the speed of data transfers. The masters set the maximum speed and the slaves (and/or masters) can arbitrarily slow it down as needed. It also means, since the master may only release the SCL line that it must check to see that SDA in fact went high before proceeding with the next clock period.

If the master -receiver signals a not-acknowledge, this indicates to the slave -transmitter that this byte was the last byte of the transfer.

Standard communication on the bus between a Master and a Slave is composed of four parts: START, Slave address, Data transfer, and STOP. The IIC protocol defines a transfer format for both 7-bit and 10-bit addressing.

A seven bit address is initiated as follows. After the START condition, a Slave address is sent. This address is seven bits long followed by an eighth-bit which is the read/write bit. A High indicates a request for data (read) and a Low indicates a data transmission (write).

Only the Slave with the calling address that matches the address transmitted by the Master (that won arbitration) responds by sending back an acknowledge bit by pulling the SDA line Low on the ninth clock.

For 10-bit addressing, two bytes are transferred to set the 10-bit address. The transfer of the first byte contains the following bit definition. The first five bits (bits 7:3) notify the slave that this is a 10-bit transfer followed by the next two bits (bits 2:1), which set the slave address bits 9:8, and the LSB bit (bit 8) is the R/W bit. The second byte transferred sets bits 7:0 of the slave address.

Figure A.1: The timing diagram of the Data Transfer on the I2C bus.

single transmission of data. Once slave addressing is successful, the data are transferred between the master and slave byte-by-byte to the specified direction. When the master is required to transmit data continually, it can transmit the data without generating the STOP condition. This is called a repeated START. A repeated start enables the master to change the direction of data transfer or address a different slave without giving up the bus.
Appendix B

I2S Protocol

The ADC provides an I2S interface which can transfer non-buffered audio data. The I2S data are in PCM 2’s complement format which are generated by ∆Σ modulation. The I2S interface consists of BCLK, WCLK and data line. WCLK is also called word select (WS) which specifies whether the data belongs to left or right channels of the stereo ADC chip. When the WCLK is low, channel 1 or left channel is being transmitted and when it is high, channel 2 or right channel is being transmitted.

In the I2S protocol, serial audio data is transmitted in 2’s complement with the MSB first. The MSB is transmitted first because the transmitter and receiver may have different word lengths. As per the specifications, it is not necessary for the transmitter to know how many bits the receiver can handle, nor does the receiver need to know how many bits are being transmitted. When the receivers word length is greater than transmitter’s word length, least significant data bits of the receiver are set to ‘0’. If the receiver received more bits than its word length, the bits after the LSB are ignored. The timing diagram of the I2S interface is shown in Fig. B.1. It expresses how the left and right channel data are transmitted synchronously with WCLK and BCLK.

![I2S Timing Diagram](image)

Figure B.1: I2S timing diagram for transmitting of the left and right channel 24-bit 2’s complement data synchronously with WCLK and BCLK.

Based on the I2S clocks direction, an I2S device can be identified as a slave or master. The device which receives the I2S clocks is a slave while a device which transmits the clocks
is a master. To generate audio samples, ∆Σ modulator needs to be operated in much higher clock rate than the sample rate or WCLK. This operational clock rate is provided by a master clock (MCLK). When the ADC I2S interface is operating as a slave transmitter, MCLK and WCLK should be phase-locked. The ∆Σ modulator internally generates the samples and transmits them out according to the WCLK. If the MCLK is independent from the WCLK, there is a possibility that generated samples on the MCLK are drifted with respect to WCLK over time and cause repetition of samples. Fig. B.2 shows a configuration which needs to be avoided because of this fact. The timing diagram in Fig. B.3 shows how the repetition of samples occur when drifting the MCLK with respect to WCLK.

Figure B.2: The ADC is driving in slave mode with independent MCLK. In here, there is a possibility that generated samples on the MCLK are drifted with respect to WCLK over time and cause repetition of samples.

Figure B.3: The timing diagram related to an I2S slave transmitter having independent MCLK. Note that generated samples on the MCLK are drifted with respect to WCLK over time and cause repetition of samples.
Fig. B.4 shows the integration of an ADC in slave mode. The FPGA as a master provides the phased-locked synchronized clocks to the ADC. In case master does not have capabilities to provide MCLK, the TLV320ADC3101 ADC is capable of deriving the internal master clock from the external BCLK, as shown in Fig. B.4(b). However, in this purpose BCLK must be within the codec PLL input frequency range which is 512 kHz to 50 MHz. When

![Diagram](image)

Figure B.4: Some possible integrations of I2S clocks in slave mode. (a) The I2S clocks and the MCLK are provided by the I2S master device. (b) The I2S slave ADC deriving the internal master clock from the external BCLK.

When the ADC I2S interface is operating in master-transmit mode, the I2S clocks are transmitted from the ADC chip. These clocks can be derived from the external MCLK using on-chip PLL as shown in Fig. B.5

![Diagram](image)

Figure B.5: The integration of the ADC I2S clocks in master mode.
Appendix C

Programming the ADCs

The TLV320ADC3101 is a programmable ADC which can be set up by configuring its registers. Once hardware reset the chip after power-up, the register can be read/written. The reset pin is active low which requires holding low at least 10 ns after stabilizing the power supply. The registers in the ADC are organized into memory pages. To programme the required registers, the related memory page needs to be selected. Once the memory page is selected, the follow-on register addressing is corresponding to the selected memory page. The ADCs are programmed via an I2C interface of the chip. A C-program is developed to program the ADCs using Xilinx I2C read/write library functions. Xilinx Software Development Kit (SDK) provides C libraries which support Microblaze processor. The program runs on Microblaze processor as a bare-metal program which controls the Xilinx I2C modules attached to the Microblaze processor.

Code C.1 describes the structure of the C program for ADC programming via I2C interface. The program writes a byte 0x01 to address 0x00 of the I2C slave device (i.e., ADC chip) having the device address 0x18. The I2C master is Xilinx I2C module which has address 0x88018000. The configurable addresses and their related values are specified in WriteBuffer. XIic_Send API can repeatedly configure the register’s with their related values by using WriteBuffer. The XIic_Send function returns the status indicating the number of bytes sent. Using the return value, it can be verified whether configuring all registers is successful or not. Further, XIic_Recv API can be used to verify whether the correct values are written to the specified addresses. To read the data which is written into address 0x00, firstly byte 0x00 should be sent using XIic_Send API. Then the value in the sent address can be assigned to the ReadBuffer by passing its pointer using XIic_Recv API.

Code C.1: The structure of the I2C program runs on Microblaze processor to program the ADCs.

```c
// This file contains system parameters for the Xilinx device driver environment.
```
#include "xparameters.h"

// These files contains low-level driver functions of I2C
#include "xiic.h"
#include "xiic_1.h"

// The driver instance for I2C Device
XIic I2c;

/* Specify the I2C master base-address and I2C slave device.
The I2C master base-address is the address related to I2C linker script address */
# I2C_MASTER_BASE_ADDRESS = 0x88018000;
I2C_DEVICE_ID = 0;
u8 IIC_SLAVE_DEVICE_ADDRESS = 0x18;

// Initialize the I2C driver.
Status = XIic_Initialize(&I2c, I2C_DEVICE_ID);

 (*)( I2C_Master base, Address, u8* BufferPtr, unsigned ByteCount, u8 Option )
unsigned XIic_Send ( u32 BaseAddress, u8 Address, u8 *BufferPtr, unsigned ByteCount, u8 Option )
unsigned XIic_Recv ( u32 BaseAddress, u8 Address, u8 *BufferPtr, unsigned ByteCount, u8 Option )

BaseAddress : contains the base address of the IIC device.
Address : contains the 7 bit I2C address of the device to send/receive
BufferPtr : points to the data to be sent/received.
ByteCount : is the number of bytes to be sent/received.
Option : is to hold or free the bus after transmitting/reception the data.

// Send address and data to the I2C device
XIic_Send(I2C_MASTER_BASE_ADDRESS, I2C_SLAVE_DEVICE_ADDRESS, WriteBuffer, 2, XIIC_STOP);

// Send address (to receive data of that address) to the I2C device
XIic_Send(I2C_MASTER_BASE_ADDRESS, I2C_SLAVE_DEVICE_ADDRESS, WriteBuffer, 1, XIIC_STOP);

// Receive data (corresponding to previously sent address) from the I2C device
XIic_Recv(I2C_MASTER_BASE_ADDRESS, I2C_SLAVE_DEVICE_ADDRESS, ReadBuffer, 1, XIIC_STOP);

// This function stops the I2C device and driver such that data is no longer sent or received on the I2C bus.
// This function stops the device by disabling interrupts.
XIic_Stop(&I2c);

Now we explain the complete Microblaze ADC program which is given in Code C.2. The program controls 8 Xilinx I2C master modules which having the base addresses 0x88018000, 0x88012000, 0x8800A000, 0x8801A000, 0x8800C000, 0x8801C000, 0x8800E000 and 0x88014000.
These addresses are referred in the program as IIC_MASTER_BASE_ADDRESS. Each I2C module programs an ADC board which consists of 4 slave ADCs chips. The I2C addresses of these chips are 0x18, 0x19, 0x1A and 0x1B. These addresses are referred in the program as IIC_SLAVE_DEVICE_ADDRESS. An I2C master module programs each slave chip sequentially. The I2C program is similar to Code C.1 which we described. Each relevant register is first configured by writing the data followed by verified by reading and compared with what has written. Regarding the verification of ADC programming, XIic_Send function returns the status indicating the number of bytes send and XIic_Recv function is used to verify whether the correct values are written to the registers.

Now we explain the configurations of the registers which we used to program the ADCs. For clarity we use the notation \((p,r,v)\) to describe the configuration which set the hexadecimal value \(v\) in register \(r\) in page \(p\). When programming the ADC, first it is required to software reset the ADC. Then the default values are not required to be programmed. This can be done by the configuration \((0,0,00h)\) followed by \((0,1,01h)\). Consequently, the page 0 is selected followed by soft reset the ADC. Then ADC clock settings should be set before powering up the ADC channels. The registers related to ADC clock settings are also on page 0 which has already been selected. Even though the ADC has PLL functionality for clock generation, in this design it uses only NADC and MADC dedicated on-chip clock dividers to derive the internal 48 kHz sampling frequency from the MCLK. This is possible due to the availability of phased-locked and synchronized FPGA-based ASI clock signals. To select the MCLK as the ADC clock input, the configuration \((0,4,00h)\) is used which is set by default. The master clock is 12.288 MHz which requires to be divided by 256 in order to generate 48 kHz internal sampling frequency. This can be achieved by setting \(\text{NADC} = 1, \text{MADC} = 2\) and \(\text{AOSR} = 128\) with the configurations of \((0,18,81h)\), \((0,19,82h)\) and \((0,20,80h)\) respectively (i.e., \(\text{NADC}\times\text{MADC}\times\text{AOSR} = 256\)). The AOSR register does not configure as it is set by default. The dividers are powered up by the same configuration. Once the clock settings are set, the ADC channels can be powered up. This can be done by the configuration \((0,81,C2h)\). By default, the ADC channels are muted and after powering up the ADC channels, they should be unmuted with the configuration \((0,82,00h)\). Next, the audio serial interface (ASI) should be configured. In this design, I2S interface is used to transmit the audio out. Both BCLK and WCLK are provided by the FPGA so they should be configured as inputs to the ADC. The selected I2S word length is 24-bits. These can be configured by the configuration \((0,27,20h)\).

Once the ADC clock settings and ASI are configured, the audio inputs should be configured. This includes configuration of microphone bias voltage, interface and gain control. These settings are contained in page 1. As the currently selected page is 0, it should be
changed to page 1 by the configuration (0,0,01h). Because of the specification of the microphones, microphone bias voltage is set as same as analog voltage supply of the ADC. This can be set by the configuration (1,51,78h). The microphone input can be set as single-ended or differential. Since the microphones in the SMA are functioning in differential mode, the microphone inputs should also be set to the differential. For this, left and right inputs need to be configured separately by the configurations (1,52,3Fh) and (1,55,3Fh) respectively. Then the left and right microphone gains can be configured by configuring the gain control registers. The resolution of the gain control is 0.5dB and with the configurations (0,59,Hh) and (0,60,Hh), left and right microphone gains can be varied. The actual gain is half of the set value (i.e., $\frac{H}{2}$) as resolution is 0.5dB. Once gain values are set, the ADC gain flag need to be set such that \textit{applied gain} = \textit{programmed gain} by the configuration (1,62,03h). Once completed the aforesaid configuration of the registers, the digital audio can be captured via I2S interfaces of each chip.

Code C.2: The bare-metal C program which runs on Microblaze processor to configure the ADC Modules using Xilinx I2C modules.

```c
#include <stdio.h>
#include <stdlib.h>
#include "xparameters.h"
#include "xutil.h"
#include "xbasic_types.h"
#include "xil_io.h"
#include "xiic.h"
#include "xiic_1.h"

Xilic Iic; /* The driver instance for I2C Device */

int main()
{
    u32 IIC_MASTER_BASE_ADDRESS;
    u16 IIC_DEVICE_ID;

    /* Initialize the I2C Master Devices */
    IIC_MASTER_BASE_ADDRESS = 0x88018000;
    IIC_DEVICE_ID = 0;
    initialize_iic(IIC_MASTER_BASE_ADDRESS, IIC_DEVICE_ID);

    IIC_MASTER_BASE_ADDRESS = 0x88012000;
    IIC_DEVICE_ID = 1;
    initialize_iic(IIC_MASTER_BASE_ADDRESS, IIC_DEVICE_ID);

    IIC_MASTER_BASE_ADDRESS = 0x8800A000;
    IIC_DEVICE_ID = 2;
    initialize_iic(IIC_MASTER_BASE_ADDRESS, IIC_DEVICE_ID);

    IIC_MASTER_BASE_ADDRESS = 0x8801A000;
    IIC_DEVICE_ID = 3;
    initialize_iic(IIC_MASTER_BASE_ADDRESS, IIC_DEVICE_ID);

    IIC_MASTER_BASE_ADDRESS = 0x8800C000;
    IIC_DEVICE_ID = 4;
    initialize_iic(IIC_MASTER_BASE_ADDRESS, IIC_DEVICE_ID);
```

215
int initialize_iic(IIC_MASTER_BASE_ADDRESS, IIC_DEVICE_ID)
{
    int Status;
    int i, j = 0;

    u8 IIC_SLAVE_DEVICE_ADDRESS;

    /* Initialize the I2C driver and perform a self-test */
    Status = XIic_Initialize(&Iic, IIC_DEVICE_ID);
    Status = XIic_SelfTest(&Iic);

    /* Initialize write buffer */
    u8 WriteBuffer[32];

    WriteBuffer[0] = 0x00; //Page control register
    WriteBuffer[1] = 0x00; //page 0 select
    WriteBuffer[2] = 0x01; //Software reset register
    WriteBuffer[3] = 0x01; //Software reset
    WriteBuffer[4] = 0x12; //ADC NADC clock divider
    WriteBuffer[5] = 0x81; //Power up + divide by 1
    WriteBuffer[6] = 0x13; //ADC NADC clock divider
    WriteBuffer[7] = 0x82; //Power up + divide by 2
    WriteBuffer[8] = 0x51; //Power up/down of the right/left ADC channels
    WriteBuffer[9] = 0xC2; //Left+Right:0xC2, Right-Only:0x42, Left-Only:0x82
    WriteBuffer[10] = 0x52; //Mute ADC channels
    WriteBuffer[11] = 0x00; //All-channel-working:0x00, Left-muted:0x80, Right←mute:0x08
    WriteBuffer[12] = 0x1B; //ADC audio interface settings
    WriteBuffer[13] = 0x20; //I2S, 24-bit, BCLK(input), WCLK(input)

    //-------------
    WriteBuffer[14] = 0x00; //Page control register
    WriteBuffer[15] = 0x01; //page 1 select
    WriteBuffer[16] = 0x33; //MICBIAS Control
    WriteBuffer[17] = 0x78; //MICBIAS1 and MICBIAS2 are connected to AVDD
    WriteBuffer[18] = 0x34; //Left ADC Input Selection for Left PGA
    WriteBuffer[19] = 0x3F; //LCH_SEL4: Differential Pair Using the IN2L(P) as PLUS← and IN3L(M) as MINUS Inputs (0–dB setting is chosen).
    WriteBuffer[20] = 0x37; //Right ADC Input Selection for Left PGA
    WriteBuffer[21] = 0x3F; //RCH_SEL4: Differential Pair Using the IN2R(P) as PLUS← and IN3R(M) as MINUS Inputs (0–dB setting is chosen).
    WriteBuffer[22] = 0x3B; //Left Analog PGA Settings
WriteBuffer[23] = 0x3C; //0x3C = 60dec --> 60*0.5 = 30dB
WriteBuffer[24] = 0x3C; //Right Analog PGA Settings
WriteBuffer[25] = 0x3C; //0x3C = 60dec --> 60*0.5 = 30dB
WriteBuffer[26] = 0x3E; //ADC Analog PGA Flags
WriteBuffer[27] = 0x03; //Left and Right ADC PGA : applied gain = programmed

void setup()
{
    // Initialize ADC chip
    // ADC chip 1
    ADC1_Init(ADC1); // ADC chip 1
    // ADC chip 2
    ADC2_Init(ADC2); // ADC chip 2
    // ADC chip 3
    ADC3_Init(ADC3); // ADC chip 3
    // ADC chip 4
    ADC4_Init(ADC4); // ADC chip 4

    // Start the I2C device
    Status = XIic_Start(&Iic); // ADC chip 1
    if (Status != XST_SUCCESS)
    {
        xil_printf("Failed to start the IIC device\n");
        return XST_FAILURE;
    }
    for (j=0;j<4;j++)
    {
        if (j==0)
        {
            IIC_SLAVE_DEVICE_ADDRESS = 0x18; // ADC chip 1
        } else if (j==1)
        {
            IIC_SLAVE_DEVICE_ADDRESS = 0x19; // ADC chip 2
        } else if (j==2)
        {
            IIC_SLAVE_DEVICE_ADDRESS = 0x1A; // ADC chip 3
        } else if (j==3)
        {
            IIC_SLAVE_DEVICE_ADDRESS = 0x1B; // ADC chip 4
        }
        for (i=0;i<28;i++)
        {
            // Send data (address + data) as a master on the I2C bus
            Status = XIic_Send(IIC_MASTER_BASE_ADDRESS, IIC_SLAVE_DEVICE_ADDRESS, WriteBuffer+i, 2, XIIC_STOP);
            if (Status != 2)
            {
                goto flag1;
            }
            xil_printf("%d , ", Status);
            // Send address (to receive data of that address) as a master on the I2C bus
            Status = XIic_Send(IIC_MASTER_BASE_ADDRESS, IIC_SLAVE_DEVICE_ADDRESS, WriteBuffer+i, 1, XIIC_STOP);
            if (Status != 1)
            {
                goto flag2;
            }
            xil_printf("%d , ", Status);
            // Receive data (in previously sent address) as a master on the I2C bus
            Status = XIic_Recv(IIC_MASTER_BASE_ADDRESS, IIC_SLAVE_DEVICE_ADDRESS, ReadBuffer, 1, XIIC_STOP);
            if (Status != 1)
            {
                goto flag3;
            }
            xil_printf("%d ; ", Status);
        }
        xil_printf("REG_0x%X : 0x%X\n", (WriteBuffer+i), ReadBuffer[0]);
    }
}
/* Stop the I2C device */
Status = XIic_Stop(&Iic);
If (Status != XST_SUCCESS) {
    xil_printf("Failed to stop the I2C device\r\n");
    return XST_FAILURE;
}
Appendix D

Implementation of the IRLS algorithm in OpenMP

Code D.1: The method of solving multiple IRLS problems in parallel using C and OpenMP. The code performs Algorithm 6 in each OpenMP thread. The code demonstrates the sparse recovery for order-3 SFT signals with a dictionary having 230 plane-wave resolution.

```c
#include <omp.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
//#include <numa.h>
#define dimDM 9
#define dimDN 230
#define dimON 1
#define nT 192

// Power values used in the weight calculation
const float norm_pow = 0.65; // 0.5*(2−0.7) ;

// IRLS parameters
const int  c_kpi = 0;
const float paramReg = 0.01;
const float c_enThresh = 1e-6;
const float c_convThresh = 1e-6;

// Dictionary
const float constDict[dimDM*dimDN] = { ... };

void IRLS_solver( float *reXn, float *imXn, float *sftReMatrix, float *sftImMatrix, float *dictSizeVar, float *zDemixMat, float *R, float *Wn, float *EN, float *sortEN, int c_maxIter ){
    float maxEN, tempEN, c_epsilon, sqNormWn, sqNormDiff;
    int numIter;
    int i, j, k;
    float sum, temp;

    // wn = ones(N,1) :
    memset(Wn, 0, nT*dimDN*sizeof(float));
}
```
for (k=0;k<nT*dimDN;k++){
    Wn[k] = 1.f;
}

c_epsilon = 1.f; // initial value

// A team of threads giving them their own copies of variables */
int tid, nthreads;
#pragma omp parallel private(tid, nthreads, i, j, numIter, maxEN, tempEN, c_epsilon, sqNormWn, sqNormDiff, sum, temp) shared(reXn, imXn, sftReMatrix, sftImMatrix, dictSizeVar, zDemixMat, R, Wn, EN, sortEN, c_maxIter)
{
    // Obtain thread number
    tid = omp_get_thread_num();
    if (tid == 0){
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }*/
    if (tid < nT){
        for (numIter = 0; numIter < c_maxIter; numIter++){
            // Weight the dictionary : in C the dictionary is 642*9
            for (i=0; i<dimDN; i++){
                for (j=0; j<dimDN; j++){
                    dictSizeVar[(tid*dimDN+i)+(j*dimDN+j)] = Wn[(tid*dimDN+i)← +i] * constDict[i*dimDM+j];
                }
            }

            // Calculate Positive Definite Symmetric Matrix
            sum = 0.f;
            for (i=0; i<dimDN; i++){
                sum += R[(tid*dimDM+i)+i];
            }
            for (i=0; i<dimDN; i++){
                R[(tid*dimDM+i)+i] += (paramReg / (1-paramsReg)) ← + (sum / dimDM) ;
            }

            // Regularizing the positive definite symmetric matrix
            sum = 0.f;
            for (i=0; i<dimDN; i++){
                for (j=i; j<dimDN; j++){
                    if (i == j) {  // R[(tid*dimDM+i)+(i*dimDM+i)] = sqrt(sum);  
                        } else R[(tid*dimDM+i)+(j*dimDM+j)] = sum/R[(tid*dimDM+i)+(i*dimDM+i)];
                    }
            }
            for (i=0; i<dimDN; i++) for (j=0; j<i; j++) R[(tid*dimDM+i)+(j*dimDM+j)] = R[(tid*dimDM+i)+(i*dimDM+j)];

            for (i=0; i<dimDN; i++) for (j=0; j<i; j++)
                    }
// Solve AsX = B where A = L*L';
// www.math.utah.edu/software/lapack/lapack-d/dpotrs.html

#define A(tid, i, J) B[(tid*dimDN+dimDM)+(i−1 + (J−1)*dimDM)]
#define B(tid, i, J) dictSizeVar[(tid*dimDM+dimDN)+(i−1 + ((J−1)*dimDN)+i)]

for (j = 0; j < dimDN; j++) {
    for (k = 0; k < dimDN; k++) {
        if (B(tid, k, j) != 0.f) {
            B(tid, k, j) /= A(tid, k, k);
            for (i = k+1; i < dimDM; i++) {
                B(tid, i, j) -= B(tid, k, j) * A(tid, i, k);
            }
        }
    }
}

// Real to complex conversion by memory interleave.
for (i = 0; i < dimDN; i++) {
    zDemixMat[(tid*2*dimDM+dimDN)+(i*2)] = dictSizeVar[(tid*dimDN+dimDN)+i];
    zDemixMat[(tid*2*dimDM+dimDN)+(i*2)+1] = 0;
}

// Multiply the demixing matrix with observation
for (i = 0; i < dimDN; i++) {
    for (j = 0; j < dimDN; j++) {
        reXn[(tid*dimDN+dimON)+i/2] += zDemixMat[(tid*2*dimDM+dimDN−i*2)+j/2]*sftReMatrix[(tid*dimDN)+j/2];
        imXn[(tid*dimDN+dimON)+i/2] += zDemixMat[(tid*2*dimDM+dimDN−i*2)+j/2]*sftImMatrix[(tid*dimDN)+j/2];
    }
}

// en = \sum |\text{abs}(x_n)|^2, 2)
for (i = 0; i < dimDN; i++) {
    //EN[(tid*dimDN)+i] = reXn[(tid*dimDN+dimON)+i]*reXn[(tid*−dimON+dimON)+i] + imXn[(tid*dimDN+dimON)+i]*imXn[(tid*−dimON+dimON)+i];
    EN[(tid*dimDN)+i] = reXn[(tid*dimDN+dimON)+i]*reXn[(tid*dimDN+dimON)+i] + imXn[(tid*dimDN+dimON)+i]*imXn[(tid*dimDN+dimON)+i];
    if (EN[(tid*dimDN)+i] > maxEN) {
        maxEN = EN[(tid*dimDN)+i];
    }
}

// en = en/\max(\text{en})
for (i = 0; i < dimDN; i++) {
    EN[(tid*dimDN)+i] = EN[(tid*dimDN)+i] / maxEN;
}
sortEN[(tid+dimDN)+i] = EN[(tid+dimDN)+i];
}

// enSort = sort(en.1,'descend');
for(i=0; i<c_kp1; i++){
    for(j=i; j<dimDN; j++){
        if(sortEN[(tid+dimDN)+i] < sortEN[(tid+dimDN)+j]){
            tempEN = sortEN[(tid+dimDN)+i];
            sortEN[(tid+dimDN)+i] = sortEN[(tid+dimDN)+j];
            sortEN[(tid+dimDN)+j] = tempEN;
        }
    }
}

// epsilon = min(epsilon, enSort(k+1)/N) ;
if(c_epsilon > sortEN[(tid+dimDN)+c_kp1]/dimDN){
    c_epsilon = sortEN[(tid+dimDN)+c_kp1]/dimDN;
}
for(i=0; i<dimDN; i++){
    EN[(tid+dimDN)+i] = pow(EN[(tid+dimDN)+i] + (c_epsilon),norm_pow);
}

// Update weights
for(i=0; i<dimDN; i++){
    Wn[(tid+dimDN)+i] = EN[(tid+dimDN)+i];
}

int main (int argc, char *argv[])
{
    float sftReMatrix[dimDM*nT] = { ... } ;
    float sftImMatrix[dimDM*nT] = { ... } ;
    float *pwdReMatrix = malloc(nT*dimDN*sizeof(float));
    float *pwdImMatrix = malloc(nT*dimDN*sizeof(float));
    int c_maxIter = 100 ;
    float *reXn, *imXn; //PLD result
    float *EN, *sortEN;
    R = malloc(nT*dimDM*dimDM*sizeof(float));
    Wn = malloc(nT*dimDN*sizeof(float));
    dictSizeVar = malloc(nT*dimDM*dimDN*sizeof(float));
    EN = malloc(nT*dimDM*sizeof(float));
    sortEN = malloc(nT*dimDM*sizeof(float));
    zDemixMat = malloc(nT*dimDM*dimDN*sizeof(float));
    reXn = malloc(nT*dimDM*dimDN*sizeof(float));
    imXn = malloc(nT*dimDM*dimDN*sizeof(float));

    struct timeval tval_before, tval_after, tval_result;
    omp_set_num_threads(nT);
    gettimeofday(&tval_before, NULL);
    IRLS_solver( reXn, imXn, sftReMatrix, sftImMatrix, dictSizeVar, zDemixMat, R, ←
memcpy(pwdReMatrix, reXn, nT*dimDN*sizeof(float));
memcpy(pwdImMatrix, imXn, nT*dimDN*sizeof(float));

gmtimeofday(&tval_after, NULL);
timersub(&tval_after, &tval_before, &tval_result);
printf("Time Spent : %ld.%06ld \n", (long int)tval_result.tv_sec, (long int)tval_result.tv_usec);

free(R);
free(Wn);
free(dictSizeVar);
free(EN);
free(sortEN);
free(zDemixMat);
free(reXn);
free(imXn);"}
Appendix E

Implementation of the IRLS algorithm in CUDA

Code E.1: The method of solving multiple IRLS problems in parallel using CUDA. The presented case demonstrates the sparse recovery for order-3 SFT signals with a dictionary having 230 plane-wave resolution.

```c
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "cuda.h"
#include "cuda_runtime.h"
#include "cuComplex.h"
#include "inverse.h"
#include "mex.h"

#define M 9 // Number of SFT Channels
#define BATCH 192 // Number of Frequencies
#define dictN 230 // Dictionary resolution
#define dictM M
#define sizeDict dictM*dictN
#define tpb 256 // Threads per block
#define MIN(a,b) ((a)<(b))?(a):(b)

// Dictionary
__constant__ float constDict[dictM*dictN] = { ... };

// Kernel:1
__global__ void irlsDataStructCompute( float *w, float *WYt, float *YWYt ) {
    int i,j,k,symN;
    __shared__ float smem_uDict[dictN*dictM];
    __shared__ float smem_weight[dictN];
    __shared__ float smem_sum[tpb];
    __shared__ float smem_symMat[dictM*dictM];
    __shared__ float r;
    __shared__ float lamda;

    r = 0.01 ;
    symN = 16 ; //pow(ceil(log(dictM)/log(2)),2) : threads for reduction
```
int tid = threadIdx.x;
int bid = blockIdx.x;

if (tid < dictN) {
    smem_weight[tid] = w[(bid*dictN)+tid];
    __syncthreads();

    smem_wDict[(0+dictN)+tid] = smem_weight[tid] * constDict[(0+dictN)+tid];
    smem_wDict[(1+dictN)+tid] = smem_weight[tid] * constDict[(1+dictN)+tid];
    smem_wDict[(2+dictN)+tid] = smem_weight[tid] * constDict[(2+dictN)+tid];
    smem_wDict[(3+dictN)+tid] = smem_weight[tid] * constDict[(3+dictN)+tid];
    smem_wDict[(4+dictN)+tid] = smem_weight[tid] * constDict[(4+dictN)+tid];
    smem_wDict[(5+dictN)+tid] = smem_weight[tid] * constDict[(5+dictN)+tid];
    smem_wDict[(6+dictN)+tid] = smem_weight[tid] * constDict[(6+dictN)+tid];
    smem_wDict[(7+dictN)+tid] = smem_weight[tid] * constDict[(7+dictN)+tid];
    smem_wDict[(8+dictN)+tid] = smem_weight[tid] * constDict[(8+dictN)+tid];
    __syncthreads();
}

for (i=0; i<dictM*dictN; i+=dictN) {
    for (j=i; j<dictM*dictN; j+=dictN) {
        if (tid < tpb) {
            smem_sum[tid] = 0;
        }
        __syncthreads();

        if (tid < dictN) {
            smem_sum[tid] = constDict[i+tid] * smem_weight[tid] * constDict[j+j+tid];
        }
        __syncthreads();

        for (k=2; k<tpb; k=k+2) {
            if (tid < (tpb/k)) {
                smem_sum[tid] = smem_sum[tid] + smem_sum[(tpb/k)+tid];
            }
            __syncthreads();
        }

        *(smem_symMat+(((i/dictN)*dictM)+(j/dictN))) = smem_sum;
    }
    __syncthreads();
}

__syncthreads();
if (tid < dictM*dictM) {
    YWT[(bid*dictM*dictM)+tid] = smem_symMat[tid];
} __syncthreads();

/////////// Regularizing /////////////
if (tid < symN) {
    smem_sum[tid] = 0;
} __syncthreads();
if (tid < dictM) {
    smem_sum[tid] = YWT[(bid*dictM*dictM)+(tid*dictM)+tid] + tid;
} __syncthreads();
for (unsigned int stride = symN/2; stride >= 1; stride >>= 1){
    __syncthreads();
    if (tid < stride) {
        smem_sum[tid] += smem_sum[tid + stride];
    }
} __syncthreads();

lamda = (r/(1-r))*(smem_sum[0]/dictM);

if (tid<dictM) {
    YWT[(bid*dictM*dictM)+(tid*dictM)+tid] = lamda + YWT[(bid*dictM*dictM)+(tid*dictM)+tid];
} __syncthreads();

// Kernel: 2
// This is smatinv_batch kernel which performs batch inverse on small matrices.
// This is implemented by Nvidia.
// The code can be downloaded from here.

// Kernel: 3
__global__ void genPwdVec(float *invYWT, float *WYt, float *SFT, float *SFI, float *PWDr, float *PWDi) {
    int i, j, k;
    float sum1, sum2, sum3;
    __shared__ float smem_invYWT[dictM*dictM];
    __shared__ float smem_WYt[dictM*dictN];
    __shared__ float smem_DMIX[dictM*dictN];
    __shared__ float smem_SFTr[dictM];
    __shared__ float smem_SFI[dictN];
    __shared__ float smem_PWDr[dictN];
    __shared__ float smem_PWDi[dictN];

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    if (tid < dictM*dictM) {
        smem_invYWT[tid] = invYWT[(bid*dictM*dictM)+tid];
    } __syncthreads();

    for (i=0; (dictM*dictN)-(i*1024)>0; i++) {
        if (tid < MIN(1024,(dictM*dictN)-(i*1024))) {
            smem_WYt[i*1024]+tid = WYt[(bid*dictM*dictN)+(i*1024)+tid];
        } __syncthreads();
    }
for (i=0; i<dictM; i++) {
    for (j=0; j<dictN; j++) {
        sum1 = 0.0;
        for (k=0; k<dictM; k++){
            sum1 += smem_invYWyT[i * dictM + k] * smem_WYt[k * dictN + j];
        }
        smem_DMIX[j * dictM + i] = sum1;
    }
}
__syncthreads();

if (tid < dictM) {
    smem_SFT[tid] = SFTr[(bid*dictM)+tid];
    __syncthreads();
    smem_SFTi[tid] = SFTi[(bid*dictM)+tid];
    __syncthreads();
}
__syncthreads();

for (i=0; i<dictN; i++) {
    sum2 = 0.0;
    for (k=0; k<dictM; k++){
        sum2 += smem_DMIX[i * dictM + k] * smem_SFTi[k];
    }
    smem_PWDr[i] = sum2;
}
__syncthreads();

for (i=0; i<dictN; i++) {
    sum3 = 0.0;
    for (k=0; k<dictM; k++){
        sum3 += smem_DMIX[i * dictM + k] * smem_SFTi[k];
    }
    smem_PWDi[i] = sum3;
}
__syncthreads();

if (tid < dictN) {
    PWDr[(bid*dictN)+tid] = smem_PWDr[tid];
    __syncthreads();
    PWDi[(bid*dictN)+tid] = smem_PWDi[tid];
    __syncthreads();
}
__syncthreads();

// Kernel: 4
__global__ void abs_gpu(float *Xgpu_re, float *Xgpu_im, float *ENgpu, float *←maxENgpu)
{
    __shared__ int i;
    __shared__ float smemMaxEn;
    int k = threadIdx.x;
    int j = blockIdx.x;

    float temp1, temp2, temp3, temp4 = 0;
    smemMaxEn = 0.0;

    if(k<dictN){
        temp1 = Xgpu_re[(j*dictN)+k];
        temp2 = Xgpu_im[(j*dictN)+k];
        temp3 = temp1*temp1 + temp2*temp2;
        ENgpu[(j*dictN)+k] = temp3;
    }
}
__syncthreads();

for (i=0; i<dictN; i++){
    temp4 = smemMaxEn;
    if (ENgpu[ (j*dictN)+i ] > temp4){
        smemMaxEn = ENgpu[ (j*dictN)+i ];
    }
}

maxENgpu[ j ] = smemMaxEn;

// Kernel: 5
__global__ void normalize_gpu( float *ENgpu, float *sortENgpu, float *maxENgpu )
{
    int k = threadIdx.x;
    int j = blockIdx.x;

    float temp1, temp2 = 0;
    temp1 = *(maxENgpu+j);
    if (k<dictN){
        temp2 = *(ENgpu+(j*dictN)+k);
        *(ENgpu+(j*dictN)+k) = temp2/temp1;
        *(sortENgpu+(j*dictN)+k) = temp2/temp1;
    }
}

// Kernel: 6
__global__ void sort_gpu( float *sortENgpu )
{
    int j = blockIdx.x;

    __shared__ int k1, k2;
    __shared__ int c_kp1;
    c_kp1 = 0;

    float temp1, temp2 = 0;
    for (k1=0; k1<=c_kp1; k1++){
        for (k2=k1; k2<dictN; k2++){
            temp1 = *(sortENgpu+(j*dictN)+k1);
            temp2 = *(sortENgpu+(j*dictN)+k2);
            if (temp1 < temp2){
                *(sortENgpu+(j*dictN)+k1) = temp2;
                *(sortENgpu+(j*dictN)+k2) = temp1;
            }
        }
    }
}

// Kernel: 7
__global__ void pow_gpu( float *ENgpu, float *sortENgpu, float *dev_w )
{
    float temp1;
    __shared__ float norm_pow;
    __shared__ float c_epsilon;
    __shared__ int c_kp1;
    norm_pow = 0.5*(2-0.7);   // 0 0.5
    c_epsilon = le-6;
    c_kp1 = 0;
    int j = blockIdx.x;
    int k = threadIdx.x;
}
// \( \epsilon = \min(\epsilon, \text{enSort}(k+1)/N) \):
if \( \epsilon > (\text{enSort}(j\cdot\text{dictN}+c_{xp1})/\text{dictN}) \) {
    \( \epsilon = (\text{enSort}(j\cdot\text{dictN}+c_{xp1})/\text{dictN}) \);
}

if \( k < \text{dictN} \) {
    temp1 = pow(EN_gpu + (j\cdot\text{dictN}) + k, \text{c_epsilon}) + (\text{c_epsilon}\cdot\text{c_epsilon})
}

--syncthreads();

// Update the weights
if \( k < \text{dictN} \) {
    \text{dev} = EN_gpu + (j\cdot\text{dictN}) + k;
}

--global__ void memSetVector(float *vec, int size) {
    __shared__ int constant;
    constant = 1;
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;

    if (idx < size) {
        vec[idx] = constant;
    }
    __syncthreads();
}

void process_irls_cuda(float *dev_sftReMatrix, float *dev_sftImMatrix, float *host_pwdReMatrix, float *host_pwdImMatrix, int c_maxIter) {
// Increase the shared memory to 64kB
    cudaDeviceSetCacheConfig(cudaFuncCachePreferShared);
    cudaFuncSetCacheConfig(irlsDataStructCompute, cudaFuncCachePreferShared);
    cudaFuncSetCacheConfig(genPwdVec, cudaFuncCachePreferShared);
    cudaFuncSetCacheConfig(normalize_gpu, cudaFuncCachePreferShared);
    cudaFuncSetCacheConfig(sort_gpu, cudaFuncCachePreferShared);
    cudaFuncSetCacheConfig(pow_gpu, cudaFuncCachePreferShared);

    float *Ainv_d;
    float *Xgpu_re, *Xgpu_im;
    float *dev_a, *dev_a2, *dev_a3;
    float *dev_c, *dev_w;
    float *EN_gpu, *maxEN_gpu;
    float *sortEN_gpu;
    clock_t start;
    clock_t end;
    float gpuTime;

    cudaMalloc((void**)&dev_w, BATCH*dictN*sizeof(float));
    cudaMalloc((void**)&dev_c, BATCH*dictM*sizeof(float));
    cudaMalloc((void**)&dev_a, BATCH*dictM*dictN*sizeof(float));
    cudaMalloc((void**)&dev_a2, BATCH*dictM*dictN*sizeof(float));
    cudaMalloc((void**)&dev_a3, BATCH*dictM*dictN*sizeof(float));
    cudaMalloc((void**)&EN_gpu, BATCH*dictN*sizeof(float));
    cudaMalloc((void**)&maxEN_gpu, BATCH*sizeof(float));

cudaMalloc((void**)&sortENgpu, BATCH*dictN*sizeof(float));
cudaMalloc((void**)&Ainv_d, BATCH*dictM*dictM*sizeof(float));
memsetVector<<<512,1024>>>(dev_w, BATCH*dictN);
cudaDeviceSynchronize();

const dim3 gridSize(BATCH, 1, 1);

int numIter;

start = clock();

for(numIter = 0; numIter < c_maxIter; numIter++){

cudaMemset(dev_c, 0, BATCH*dictM*dictM*sizeof(float));
cudaMemset(Ainv_d, 0, BATCH*dictM*dictM*sizeof(float));
cudaMemset(dev_a3, 0, BATCH*dictM*dictN*sizeof(float));
cudaDeviceSynchronize();

// Kernel: 1
irlsDataStructCompute<<<gridSize,tpb>>>(dev_w, dev_a2, dev_c);
cudaDeviceSynchronize();

// Kernel: 2
smatinv_batch(dev_c, Ainv_d, M, BATCH);
cudaDeviceSynchronize();

// Kernel: 3
genPwdVec<<<gridSize,1024>>>(Ainv_d, dev_a2, dev_sftReMatrix, dev_sftImMatrix, Xgpu_re, Xgpu_im);
cudaDeviceSynchronize();
cudaMemset(ENgpu, 0, BATCH*dictN*sizeof(float));

// Kernel: 4
abs_gpu<<<gridSize,dictN>>>(Xgpu_re, Xgpu_im, ENgpu, maxENgpu);
cudaDeviceSynchronize();

// Kernel: 5
normalize_gpu<<<gridSize,dictN>>>(ENgpu, sortENgpu, maxENgpu);
cudaDeviceSynchronize();

// Kernel: 6
sort_gpu<<<gridSize,1>>>(sortENgpu);
cudaDeviceSynchronize();

// Kernel: 7
pow_gpu<<<gridSize,dictN>>>(ENgpu, sortENgpu, dev_w);
cudaDeviceSynchronize();
}
cudAmemcpy(host_pwdReMatrix, Xgpu_re, BATCH*dictN*sizeof(float), host_pwdReMatrixToHost);
cudAmemcpy(host_pwdImMatrix, Xgpu_im, BATCH*dictN*sizeof(float), host_pwdImMatrixToHost);

dtime = clock();
gpuTime = (float)(end - start) / CLOCKS_PER_SEC;
cudAfree(dev_c);
cudAfree(dev_w);
cudAfree(dev_a2);
cudAfree(dev_a3);
cudAfree(Ainv_d);
cudAfree(Xgpu_re);
cudAfree(Xgpu_im);
cudAfree(ENgpu);
cudaFree(maxENgpu);
cudaFree(sortENgpu);
}
Bibliography


[41] Peter Fall. Gigabit Ethernet UDP/IP stack (LGPL License), October 2014.


[52] Wojtek James Goscinski, Paul McIntosh, Ulrich Felzmann, Anton Maksimenko, Christopher Hall, Timur Gureyev, Darren Thompson, Andrew Janke, Graham Gal-
loway, Neil Killeen, Parnesh Raniga, Owen Kaluza, Amanda Ng, Govinda Poudel, David Barnes, Toan Nguyen, Paul Bonnington, and Gary Egan. The multi-modal Australian ScienceS Imaging and Visualization Environment (MASSIVE) high per-


Nils Peters and Andrew Schmeder. Beamforming using a spherical microphone array based on legacy microphone characteristics. In International Conference on Spatial Audio, Detmold, Germany, November 2011.


