Rice University

Design Space Exploration of Parallel Algorithms and Architectures for Wireless Communication and Mobile Computing Systems

by

Guohui Wang

A Thesis Submitted
in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

[Signatures of committee members]

Houston, Texas
December 2014
ABSTRACT

Design Space Exploration of Parallel Algorithms and Architectures for Wireless Communication and Mobile Computing Systems

by

Guohui Wang

During past several years, there has been a trend that the modern mobile SoC (system-on-chip) chipsets start to incorporate in one single chip the functionality of several general purpose processors and application-specific accelerators to reduce the cost, the power consumption and the communication overhead. Given the ever-growing performance requirements and strict power constraints, the existence of different types of signal processing workloads have posed challenges to the mapping of the computationally-intensive algorithms to the heterogeneous architecture of the mobile SoCs. Heterogeneous accelerators have been used to improve the system performance. Specifically, because of the highly parallel architecture and high memory bandwidth, the GPGPU (general-purpose computing on graphics processing units) has attracted much attention and has been considered as universal parallel accelerators for many high performance applications. However, due to the inherent limitations of algorithms, such as the parallelism level and memory access pattern, not all algorithms are suitable for GPGPU acceleration. Partitioning the workloads, and transforming the algorithms to adapt to the hardware architecture of the corresponding accelerators (especially GPU) on mobile chipsets
are crucial for high performance and good efficiency on mobile platforms.

The goal of this thesis is to study parallel algorithms for several representative application workloads covering two of the most important application areas (wireless communication and mobile computer vision). According to the properties of these algorithms, we study how to map them onto GPU’s architecture with necessary algorithm transformation and optimization. We explore the design space of the parallel algorithms and architectures, and highlight the workload partitioning and architecture-aware optimization schemes including algorithmic optimization, data structure optimization, and memory access optimization to improve the throughput performance and hardware (or energy) efficiency. Based on the implementation results of the case studies, we found that the speedup provided by the GPGPU implementation varies significantly depending on the characteristics of the algorithms. Therefore, GPGPU is not always the best candidate to implement an accelerator. Some use cases do require custom logic to achieved required performance; while in some other use cases, we need to distribute the workloads between CPU and GPU heterogeneous architecture to achieve peak performance. The experimental results show that the heterogeneous architecture of mobile systems is the key to efficiently accelerating parallel algorithms in order to meet the growing requirements of performance, efficiency, and flexibility.
First of all, I would like to acknowledge my adviser, Professor Joseph R. Cavallaro, for his constant help, advice, support, and above all patience over the whole period of my graduate study. He encouraged me to think and explore freely and deeply, so I was able to investigate many interesting research topics and finally found out my interests. I would like to thank my thesis committee members, Professor Vivek Sarkar, Professor Lin Zhong and Professor Markku Juntti for their constructive comments and suggestions. I want to thank Professor Behnamm Aazhang, Professor Ashutosh Sabharwal, Professor Peter Varmann, Professor David Johnson, Professor John Mellor-Crummey, and Professor Mark Embree for their inspirations and guidance through discussions and courses.

The research for this thesis has been financially supported by Renesas Mobile, Texas Instruments, Xilinx, Samsung, Huawei, and the US National Science Foundation (under grants CNS-1265332, ECCS-1232274, EECS-0925942 and CNS-0923479).

I would like to thank my parents Yuqing Wang and Huiying Yao for their continuous support and encouragement throughout all these years. I also want to say "thank you" to my wife, Hui, for being supportive and considerate. We are both grateful to ECE Department and our advisers for giving us the chance to study together, in the same university, in the same department, and even in the same building!
I always feel lucky that I can spent five and a half years in such a beautiful and peaceful campus with many wise people. I am thankful for all my friends who have made the time at Rice so memorable. I cannot list all of them, but I have to mention a few: Yang Sun, Kia Amiri, Michael Wu, Bei Yin, Chris Studer, Xiaozhu Lin, Minyan Zhu, Hang Yu, Lei Tang, Samantha Summerson, Zhen Wang, Xu Liu, Hao Shen, Xiaoye Sun, Mr. Lan and Kaipeng Li.

Finally, I want to thank my son Alex. He brings us indefinite joy and made the past year even more wonderful and unforgettable.
Contents

Abstract ii

Acknowledgements iv

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Scope of the Thesis ........................................... 8
    1.2.1 Thesis Statement ...................................... 9
    1.2.2 Thesis Contributions ................................... 9
    1.2.3 Thesis Outline ...................................... 15
  1.3 List of Symbols and Abbreviations .......................... 15

2 Background and Related Work 18
  2.1 Related Work ............................................. 19
  2.2 Heterogeneous Accelerator Architecture for Wireless and Mobile Computing Systems ........................................... 23
    2.2.1 General-Purpose Computing on Graphics Processing Units (GPGPU) 23
    2.2.2 Other Accelerators ...................................... 29
  2.3 Methodology .............................................. 34
    2.3.1 Algorithm Mapping Guidelines .......................... 34
    2.3.2 Capability of Different Accelerators .................... 36
    2.3.3 Algorithm Analysis Steps ................................ 37
  2.4 Summary and Challenges ................................... 38
3 Case Study 1: Parallel LDPC Decoding for SDR Systems using GPU

3.1 Introduction to Low-Density Parity-Check (LDPC) Codes 41
3.2 Algorithmic Optimization to LDPC Decoding 47
  3.2.1 Loosely Coupled LDPC Decoding Algorithm 48
  3.2.2 Two-Pass Recursion CTV Message Update 48
  3.2.3 Two-min Min-Sum LDPC Decoding Algorithm 49
3.3 CUDA Architecture 52
3.4 Mapping LDPC Decoding Algorithm onto GPU 53
  3.4.1 Implementation of the LDPC Decoder Kernel on GPU 54
  3.4.2 Multi-codeword Parallel Decoding 55
  3.4.3 Implementation of Early Termination Scheme 57
3.5 Memory Access Optimization 58
  3.5.1 Data Structure and Storage Optimization 58
  3.5.2 Fully Coalesced Memory Access For CNP and VNP 59
  3.5.3 Data and Thread Alignment for Irregular Block Size 61
3.6 Reducing Decoding Latency 62
  3.6.1 Asynchronous Memory Transfer 62
  3.6.2 Multi-stream Scheduling for Concurrent Kernels 63
3.7 Experimental Results 65
3.8 VLSI Implementation of LDPC Decoding 67
  3.8.1 Algorithm and Architecture 68
  3.8.2 VLSI Implementation 69
3.9 Design Space Exploration 70
  3.9.1 Analysis and Discussion 76
3.10 Summary 78

4 Case Study 2: Parallel Multi-mode Turbo decoding 80
4.1 Error-Correction Codes 82
  4.1.1 Turbo Codes 84
4.2 Parallel Turbo Decoding Algorithm and Memory Conflict Problem 91
  4.2.1 Parallel Turbo Decoding Architecture 91
  4.2.2 Design Challenges 94
4.3 Related Work 96
4.4 Eliminating Memory Reading Conflicts via Balanced Turbo Decoding Scheduling ........................................... 98
4.4.1 Properties of Memory Reading and Writing Conflicts .......... 98
4.4.2 Unbalanced and Balanced Scheduling Schemes ................. 99
4.4.3 Further Discussions ........................................... 101
4.5 Double-buffer Contention-free Architecture for Interconnection Network 101
4.5.1 Methodology and System Model ............................... 102
4.5.2 Statistical Analysis of Memory Conflicts ........................ 103
4.5.3 Double-buffer Contention-free (DBCF) Architecture ............ 105
4.6 Efficient Unified Interleaver/Deinterleaver Architecture ........... 111
4.7 VLSI Implementation .............................................. 114
4.7.1 Implementation Results for the Contention-free HSPA+ Inter-
      leaver ............................................................. 116
4.7.2 Implementation Results of Multi-Standard Turbo Decoder ....... 120
4.8 GPU Implementation of Parallel Turbo Decoding .................. 124
4.8.1 Shared Memory Allocation ...................................... 126
4.8.2 Forward Traversal .............................................. 127
4.8.3 Backward Traversal and LLR Computation .................... 128
4.8.4 Early Termination Scheme ...................................... 130
4.8.5 Interleaver .................................................... 133
4.8.6 max* Function ................................................ 134
4.8.7 Experimental Results ........................................... 135
4.9 Design Space Exploration of High Performance Turbo decoder Design 137
4.9.1 Analysis and Discussion ........................................ 144
4.10 Summary ......................................................... 147

5 Case Study 3: Computer Vision Accelerator using GPU on Mobile Devices 149
5.1 GPGPU on Mobile Devices ........................................ 151
5.2 Parallel Implementation of Exemplar-based Object Removal Algorithm 154
5.2.1 Overview of Object Removal Algorithm ........................ 154
5.2.2 Algorithm Workflow Analysis and CPU-GPU Partitioning .... 157
5.2.3 Algorithm Optimizations and Implementation Trade-offs ....... 159
5.2.4 Experimental Results .......................................... 174
5.2.5 Design Space Exploration ..................................... 180
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.6 Analysis of Algorithm and Discussion</td>
<td>184</td>
</tr>
<tr>
<td>5.3 OpenCL-based Accelerator of SIFT Algorithm using Mobile GPU</td>
<td>186</td>
</tr>
<tr>
<td>5.3.1 Overview of Scale-invariant Feature Transform (SIFT)</td>
<td>187</td>
</tr>
<tr>
<td>5.3.2 Algorithm Profiling and Workload Partitioning</td>
<td>189</td>
</tr>
<tr>
<td>5.3.3 Optimization of Parallel Gaussian Pyramid Generation</td>
<td>194</td>
</tr>
<tr>
<td>5.3.4 Experimental Results</td>
<td>196</td>
</tr>
<tr>
<td>5.3.5 Design Space Exploration</td>
<td>197</td>
</tr>
<tr>
<td>5.3.6 Analysis and Discussion</td>
<td>201</td>
</tr>
<tr>
<td>6 Design Space Exploration and Comparison on Heterogeneous Architecture</td>
<td>204</td>
</tr>
<tr>
<td>6.1 Comparison and Design Tradeoffs</td>
<td>205</td>
</tr>
<tr>
<td>6.1.1 Performance Requirements</td>
<td>205</td>
</tr>
<tr>
<td>6.1.2 System Performance Specifications for Accelerators</td>
<td>210</td>
</tr>
<tr>
<td>6.1.3 Comparisons of Accelerator Architectures</td>
<td>213</td>
</tr>
<tr>
<td>6.2 Summary</td>
<td>217</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>223</td>
</tr>
<tr>
<td>7.1 Conclusion of the Current Results</td>
<td>223</td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>225</td>
</tr>
<tr>
<td>Bibliography</td>
<td>227</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Diagram of mobile SoC architecture. ........................................... 6
1.2 Mapping computationally-intensive algorithms onto a heterogeneous
architecture. .................................................................................. 7

2.1 Diagram of a modern GPU architecture. ................................. 25
2.2 Methodology of algorithm analysis and mapping to heterogeneous ar-
chitecture. .................................................................................... 35
2.3 Steps to identify the algorithm bottleneck based on architecture spec
of an accelerator. ....................................................................... 37

3.1 An example parity-check matrix $\mathbf{H}$ and corresponding tanner
graph. ............................................................................................. 44
3.2 Parity-check matrix $\mathbf{H}$ for a block length 1944 bits, code rate 1/2, IEEE
802.11n (1944, 972) QC-LDPC code. $\mathbf{H}$ consists of $M_{\text{sub}} \times N_{\text{sub}}$ sub-
matrices ($M_{\text{sub}} = 12$, $N_{\text{sub}} = 24$ in this example). ...................... 45
3.3 Code structure of the GPU implementation of LDPC decoder by using
two CUDA kernels. ................................................................. 54
3.4 Multi-codeword parallel decoding algorithm. The 802.11n (1944, 972)
code is assumed. The $\mathbf{H}$ matrix has 12 layers. $Z = 81$. $N_{\text{CW}}$ repres-
ts the number of codewords in one Macro-codeword (MCW). $N_{\text{MCW}}$
represents the number of MCWs. .................................................. 56
3.5 The compact representation for $\mathbf{H}$ matrix. ............................... 58
3.6 Optimized coalesced memory access. A shifted identity matrix from WiMAX code ($Z = 96$) with shift value 43 is shown. Combining CNP from (a) and VNP from (c), we achieve fully coalesced memory accesses.

3.7 Asynchronous data transfer. H2D: host to device data transfer. D2H: device to host data transfer.

3.8 Multi-stream LDPC decoding.

3.9 Multi-GPU LDPC decoding managed by multiple CPU threads.

3.10 Memory access conflict problem for layered LDPC decoding.

3.11 Diagram of double-layer LDPC decoding architecture.

3.12 Analysis of performance bottleneck of LDPC decoding algorithm.

4.1 Block diagram of a typical modern wireless communication system with channel coding.

4.2 Structure of rate 1/3 turbo encoder in 3GPP LTE.

4.3 Block diagram for a turbo encoder (left) and a turbo decoder (right).

4.4 Forward and backward recursions for trellis step $k$ in the MAP decoding algorithm.

4.5 (a) Parallel turbo decoding architecture with $P$ SISO decoders and $M$ extrinsic memory modules; (b) A 2-way memory conflict happens when SISO decoder 1 and SISO decoder $P$ access (read from or write to) memory 2 at the same time.

4.6 Diagram of the cross-MAP (XMAP) decoding architecture with NII.

4.7 (a) Memory writing conflict, in which 3 SISO decoders try to write to the same memory; (b) Memory reading conflict, in which 3 SISO decoders try to read data from the same memory. $L_1$, $L_2$ and $L_3$ represent LLR data; $D$ represents a clock cycle’s delay.

4.8 (a) The unbalanced turbo decoder scheduling; (b) The balanced scheduling scheme. (The numbers indicate the order of scheduling steps.)

4.9 Cycle-accurate simulation used to study the memory conflict problem.

4.10 Memory conflict ratio for different parallelism degrees and block sizes for the HSPA+ turbo code interleaver.

4.11 The percentage of $n$-way memory conflicts (for $P_{LLR} = 16$ and $P_{LLR} = 32$). Interleaving algorithm from 3GPP HSPA+ standard is employed. Turbo code block size is $K = 5114$. 
4.12 Block diagram of the proposed DBCF buffer architecture to solve memory writing conflict. (Note: wp and rp in the circular buffer represent write pointer and read pointer, respectively.)

4.13 Dynamic balance of data pressure on inter-connection network via “spring-like” structure of the DBCF architecture.

4.14 Architecture of unified parallel interleaving address generation (IAG) supporting both interleaving and deinterleaving. Note that we need $P_{LLR}$ copies of each duplicated block in this figure.

4.15 Overall architecture of the proposed multi-standard (HSPA+/LTE/LTE-Advanced) turbo decoder.

4.16 Bit error rate (BER) performance with fixed-point simulation for turbo decoder (code rate 1/3) with the proposed parallel architecture, using BPSK modulation over an additive white Gaussian noise (AWGN) channel. The parallelism degrees for HSPA+ and LTE mode are 8 and 16, respectively. The number of iterations is 5.5.

4.17 VLSI chip layout for an HSPA+/LTE multi-standard turbo decoder with the architecture shown in Fig. 4.15. The TSMC 45nm standard cell library is used. This turbo decoder contains 16 Radix-4 XMAP decoders, 64 LTE QPP interleavers, and 32 HSPA+ interleavers.

4.18 Hardware efficiency (energy/area) comparison with related work.

4.19 To decode $N$ codewords, we divide each codeword into $P$ sub-blocks. Each thread-block has 128 threads and handles 16 codeword sub-blocks.

4.20 High bandwidth requirement of turbo decoding algorithm, due to the irregular memory access pattern.

4.21 Analysis of turbo decoding algorithm with the proposed methodology.

5.1 Architecture of a typical mobile platform.

5.2 OpenCL programming model and hierarchical memory architecture.

5.3 An example of the object removal algorithm. The mask image indicates the object to be removed from the original image.

5.4 Major steps of the exemplar-based image inpainting algorithm for object removal.
5.5 Algorithm workflow of the exemplar-based object removal algorithm.
Please note that one OpenCL block might be mapped to one or multiple OpenCL kernels depending on the computation and memory data access patterns. ................................................................. 157

5.6 Algorithm mapping of findBestPatch kernel function using OpenCL. 160

5.7 The best patch mapping found by a full search and the result images.
The 1st row: original images. The 2nd row: masks covering the unwanted objects. The 3rd row: best patch mapping; the small squares indicate the best patches found by the findBestPatch() function. The 4th row: result images. ................................................................. 161

5.8 Diagram of reduced search area with search factor α. .................. 163

5.9 The effect of reducing the search area. The search area factor α is defined as in Fig. 5.8. ................................................................. 166

5.10 The relationship among search area, object area and the overlap area for the best patch searching. ..................................................... 167

5.11 Impact of increased patch size 13 × 13 and 17 × 17. “WalkingMan” test image. The processing time for 13 × 13 and 17 × 17 patches is normalized by the time of the 9 × 9 patch. (The 9 × 9 patch size is suggested by the original algorithm proposed by Criminisi et al.). ... 168

5.12 The experimental results for increased patch size. (α = 0.05 in this experiment.) ................................................................. 169

5.13 A detailed diagram showing how 8 × 8 work items in a work group compute SSD values in parallel. Most of the pixels in the dashed box are accessed multiple times by parallel work items. ................. 172

5.14 Performance comparison between using the local memory and without using the local memory. The results for the “WalkingMan” are shown. The experiments performed on other test images generate similar results. ................................................................. 174

5.15 An interactive object removal demo on Android with the OpenCL acceleration. (a) original image; (b) a mask indicating the object area; (c) intermediate result; (d) final result image after iterative editing. ... 175

5.16 Analysis of object removal algorithm with the proposed methodology. 184

5.17 Diagram of SIFT keypoint detection algorithm showing one octave with 6 Gaussian image layers. ..................................................... 187

5.18 Algorithm partitioning for heterogeneous implementation. ........ 193
5.19 Efficient Gaussian pyramid generation. The procedure for octaves 0 ∼ 2 is shown. For octaves 3 ∼ 4, the process is similar to octave 2.

5.20 Keypoint detection results for “graf”, “ubc”, and “boat” images. Each circle represents a keypoint, its radius indicating the scale of the feature. The lines in the circle represent orientation of the keypoints. Numbers of keypoints are 269, 133, and 192, respectively.

5.21 Analysis of SIFT algorithm with the proposed methodology.

6.1 Algorithm properties and architecture offloading to mobile accelerators.
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The evolution of mobile telecommunication standards.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>The key features of 5G wireless communication systems.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Terminology comparison between CUDA and OpenCL.</td>
<td>25</td>
</tr>
<tr>
<td>2.2</td>
<td>Summary of compute capability of accelerators.</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Complexity comparison for CNP using a “native” implementation, the FBA and the TMA.</td>
<td>52</td>
</tr>
<tr>
<td>3.2</td>
<td>Achievable throughput. $N_s = 16$, $N_{CW} = 2$, $N_{MCW} = 40$.</td>
<td>65</td>
</tr>
<tr>
<td>3.3</td>
<td>Lowest achievable latency for different throughput goals ($N_{iter} = 10$). WiMAX (2304, 1152) code.</td>
<td>66</td>
</tr>
<tr>
<td>3.4</td>
<td>Decoding latency comparison with other works. ($N_C$: number of codewords; $T$: throughput; $L$: latency)</td>
<td>67</td>
</tr>
<tr>
<td>3.5</td>
<td>Throughput performance of the proposed LDPC decoder.</td>
<td>70</td>
</tr>
<tr>
<td>3.6</td>
<td>Notations for parallel LDPC decoding.</td>
<td>72</td>
</tr>
<tr>
<td>3.7</td>
<td>Memory accesses for each iteration in LDPC decoding.</td>
<td>72</td>
</tr>
<tr>
<td>3.8</td>
<td>Memory bandwidth for different throughput requirements for LDPC decoding.</td>
<td>74</td>
</tr>
<tr>
<td>3.9</td>
<td>Flexibility requirements for a reconfigurable LDPC decoder.</td>
<td>75</td>
</tr>
<tr>
<td>3.10</td>
<td>Comparison between ASIC and GPU implementations of LDPC decoding.</td>
<td>76</td>
</tr>
<tr>
<td>4.1</td>
<td>FEC codes used by major wireless communication standards.</td>
<td>83</td>
</tr>
<tr>
<td>4.2</td>
<td>Simulation results for parameter determination.</td>
<td>109</td>
</tr>
</tbody>
</table>
5.9 Speedup for OpenCL-based heterogeneous implementations with the proposed algorithmic and memory optimizations. 180
5.10 Profiling results for the major processing steps of SIFT, using the $320 \times 256$ “graf” image. 269 keypoints are detected. 192
5.11 Processing time on the whole dataset with 48 images. 196
5.12 SIFT algorithm breakdown and analysis. 198

6.1 Comparisons of computation and memory access properties of different algorithms. 207
6.2 Technical specification of GTX 690 and GTX Titan. GTX 690 contains two GK104 GPU cores. The table shows the numbers for one GK104 core. 212
6.3 Technical specifications of the major mobile GPUs. 213
6.4 Summary of design space exploration and accelerator choice for the presented case studies. 219
1.1 Motivation

The rapid development of mobile computing technology has increased the computing capability as well as the quality of services to enable a rich user experience in many wireless and mobile applications. To fulfill the ever-growing demand for information retrieval, social networks and ubiquitous computing, high speed connectivity and computation capability for image processing and computer vision have become two of the most important features of modern mobile computing systems, such as smartphones, tablets and numerous emerging wearable devices [1–3].

High Speed Connectivity

Many emerging technologies and related applications, such as teleconference, multimedia streaming, remote surveillance, online gaming, mobile cloud computing and so on, have been driving the evolution of high speed wireless communication systems to support higher data rates and more users [4–7]. During the past decades, the mobile telecommunication systems have quickly evolved from 1G (first generation) to 4G (the forth generation), with the data rates increasing from several Kbps to
Table 1.1: The evolution of mobile telecommunication standards.

<table>
<thead>
<tr>
<th>Generation</th>
<th>Year</th>
<th>Technology</th>
<th>Data rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G</td>
<td>~1981</td>
<td>AMPS, TACS</td>
<td>14.4 Kbps</td>
</tr>
<tr>
<td>2G</td>
<td>~1995</td>
<td>GSM, CDMA</td>
<td>144 Kbps</td>
</tr>
<tr>
<td>2.5G</td>
<td>~2000</td>
<td>GPRS, EDGE, CDMA2000</td>
<td>~200 Kbps</td>
</tr>
<tr>
<td>3G</td>
<td>~2002</td>
<td>W-CDMA, CDMA2000 1xEV-DO</td>
<td>384 Kbps</td>
</tr>
<tr>
<td>3.5G</td>
<td>~2007</td>
<td>HSDPA, HSPA+, LTE, WiMAX</td>
<td>10-100 Mbps</td>
</tr>
<tr>
<td>4G</td>
<td>~2015</td>
<td>IMT-Advanced, LTE-Advanced, HSPA+ Advanced</td>
<td>1 Gbps</td>
</tr>
<tr>
<td>5G</td>
<td>2020~2030</td>
<td>Under research</td>
<td>&gt; 10Gbps</td>
</tr>
</tbody>
</table>

several hundred Mbps [8–10]. Mobile-broadband subscriptions have climbed from 268 million in 2007 to 2.1 billion in 2013 [11]. Recently, modern 3G/4G wireless communication systems such as 3GPP (3rd Generation Partnership Project) UMTS/HSPA+ (Universal Mobile Telecommunications System/High-Speed Packet Access Evolution) [12], 3GPP LTE (Long Term Evolution) and LTE-Advanced [13] have been deployed to meet the ever-growing demand for higher data rates and better quality of service. High throughput is one of the most important requirements for emerging wireless communication standards. For instance, the 3GPP UMTS standard Release 11 extends HSPA+ with several key enhancements including increased bandwidth and number of antennas. These enhancements lead to 336 Mbps peak data rate with $2 \times 2$ MIMO (multiple-input multiple-output) and 40MHz bandwidth (or $4 \times 4$ MIMO and 20MHz bandwidth) [12]. Recently, up to 672 Mbps data rate has been proposed for the future release of 3GPP standards [14, 15]. As a 4G standard, the 3GPP LTE-Advanced promises up to 1 Gbps data rate as its long term goal. Table 1.1 summaries the technologies and data rate requirements for major wireless telecommunication standards.

The 4G wireless communication systems have been deployed in many countries and will be soon deployed around the globe. However, with the explosion of the demand for data services and the emerging internet-of-things (IoT), 4G systems will
Table 1.2: The key features of 5G wireless communication systems.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System capacity</td>
<td>1000 times higher capacity compared to 4G</td>
</tr>
<tr>
<td>Spectral efficiency</td>
<td>10x more efficiency compared to 4G</td>
</tr>
<tr>
<td>Peak data rate for low mobility</td>
<td>10 Gb/s</td>
</tr>
<tr>
<td>Peak data rate for high mobility</td>
<td>1 Gb/s</td>
</tr>
<tr>
<td>Average cell throughput</td>
<td>25 times higher compared to 4G</td>
</tr>
<tr>
<td>Major technologies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Massive MIMO</td>
</tr>
<tr>
<td></td>
<td>Millimeter wave (mmWave)</td>
</tr>
<tr>
<td></td>
<td>Cognitive radio networks</td>
</tr>
<tr>
<td></td>
<td>Mobile femtocell</td>
</tr>
<tr>
<td></td>
<td>Visible light communication</td>
</tr>
<tr>
<td></td>
<td>Energy efficiency (green)</td>
</tr>
</tbody>
</table>

not meet the requirements of the data rates and mobile network capacity. Therefore, research works toward the 5G wireless communication systems have started recently and such systems are expected to be deployed beyond 2020. Although the definition of the 5G system is still uncertain yet, some key features compared to the 4G systems are widely agreed [16–18]. These key features are shown in Table 1.2.

Many research works have been conducted towards the development of the 4G and 5G wireless technologies. However, there are still many challenges, such as reducing signal processing complexity, performance metrics optimization, realistic channel modeling, interference management for cognitive radio networks and so on [16]. Among them, the high complexity of the advanced signal processing algorithms are always one of the major performance bottlenecks. This becomes even more challenging as the peak data throughput grows up to 10 Gb/s in the upcoming 5G systems. That being said, the design and implementation of the parallel algorithms and accelerator architectures will play an even more important role to support very high performance requirements as the wireless communication systems keeps evolving.

As a necessary module in modern wireless communication systems, error-correction codes are widely employed in all major wireless communication standards to ensure reliable communication between base-stations and mobile terminals [19, 20]. Due to
their superior error-correction performance, turbo codes and low-density parity-check (LDPC) codes are two of the most important classes of error-correction codes and are widely adopted by many wireless communication standards [21, 22]. For example, turbo codes are employed by 3GPP UMTS, HSPA+ and LTE/LTE-Advanced, and LDPC codes are adopted by 802.11ac WiFi, 802.16e WiMAX, DVB-S2, deep space satellite communication and so on. However, the fact that the decoders for turbo codes and LDPC codes utilize iterative decoding algorithms lead to very long decoding latency and therefore prevent the decoders from achieving high data rates. Therefore, parallel algorithms and architectures are required to implement turbo or LDPC decoders to reach high data rate requirements.

**Image Processing and Computer Vision**

We are witnessing an exponential growth of more advanced image processing and computer vision use cases, such as image editing, augmented reality, object recognition, real-time navigation and so on [23, 24]. Computer vision brings together multiple fields such as machine learning, image processing, probability, and artificial intelligence to allow computer systems to analyze the real-world scenes and respond to human behaviors.

As mobile processors are gaining more computing capability, developers have started to implement computer vision algorithms on mobile devices [25–27]. Since mobile computer vision applications are able to provide exceptional user experiences and enhanced user interfaces, they have significantly stimulated the popularization of the mobile devices such as smartphones and tablets during the past few years. However, due to the relatively limited compute resources, advanced computer vision algorithms still suffer from long processing time due to the inherently high computational complexity, which prevents advanced computer vision algorithms from
being practically used in mobile applications.

Recently, with the advent of general-purpose computing on graphics processing unit (GPGPU) and corresponding programming models on mobile systems, exploiting the parallelism in the computer vision algorithms in different levels and wisely distributing the partitioned workloads between mobile CPU and GPU bring us new opportunities to enable advanced computer vision use cases in mobile systems [28, 29]. However, the algorithm partitioning and architecture-aware optimizations are still quite challenging problems due to the differences in the programming models and hardware architectures between the traditional desktop/workstation environments and mobile computing systems.

**GPGPU and Heterogeneous Mobile Computing**

With rapid advances in semiconductor processing technologies, the density of gates on the die has increased dramatically during the past decades [30]. This allows the realization of complicated designs on the same chip [31]. Over the last few years, to meet the demand for better mobility and computation capability on the emerging mobile devices such as smartphones and tablets, we have witnessed a trend that heterogeneous components including the traditional microprocessor, GPU (graphics processing unit), memories, DSP (digital signal processor), ISP (image signal processor), communication modem, and other accelerators have been integrated on a single mobile SoC (system-on-chip) chip, as is shown in Fig. 1.1.

The emergence of mobile SoC technology brings both opportunities and challenges to the algorithm and architecture design of the mobile and wireless communication applications. The opportunities come from the fact that the mobile SoCs are gaining more powerful computation capabilities with their heterogeneous architecture, a whole spectrum of applications are enabled on practical mobile devices via com-
Figure 1.1: Diagram of mobile SoC architecture.

Computation offloading to accelerators [32]. Meanwhile, the challenges come from the system specifications or performance requirements such as total throughput, power consumption constraints, flexibility, configurability and so on [31].

Moreover, many signal processing workloads in the aforementioned systems such as channel decoding for wireless communication systems or mobile computer vision applications exhibit very high computational complexity, which require parallel algorithms and architectures to be explored to meet the performance and power efficiency requirements.

In addition, due to the fast evolution of the wireless communication standards and computer vision algorithms, it is highly desirable to implement flexible and configurable modules, so that mobile devices can easily support future extensions of standards or more powerful and accurate algorithms with low development and deployment cost. Therefore, configurability, scalability, and flexibility have become more essential, and this has posed new challenges to the design and implementation of accelerators for signal processing algorithms.

In these cases, GPUs are typically considered good platforms to meet the above requirements including performance, efficiency and flexibility, and improve the overall system performance. However, for wireless and mobile computing applications,
limited research work has been done with GPGPU. It is important for us to explore this very specific area and see if GPGPUs can be good accelerators across different applications, how GPGPU can be used in such scenarios, and most importantly, how the algorithms and architecture should be co-designed to make GPU more suitable for these workloads. Therefore, it is crucial for system designers and application developers to exploit algorithm-level and architecture-level optimizations to fully utilize the GPU in the heterogeneous environment of a mobile platform.

By analyzing the core algorithms in wireless and mobile computing applications, we can learn whether GPGPU is suitable to accelerate these algorithms. Furthermore, if GPGPU is not the best platform or cannot meet the performance requirement, how
to efficiently map the different types of workloads onto heterogeneous architecture
to achieve a specific performance requirement has become a major challenge, given
memory access pattern (regular or irregular) and parallelism degree (low or high) of
an algorithm, as is shown in Fig. 1.2.

1.2 Scope of the Thesis

In this thesis, we focus on parallel algorithms and architecture for mobile computing
systems. To be more specific, we exploit the parallelism in algorithms and study
whether the algorithms can be mapped efficiently onto GPU’s parallel architecture.
We also study the methodology to map the algorithms onto other parallel architec-
tures including VLSI architecture, multi-core architecture, and heterogeneous mobile
processors if we found that GPU cannot meet the performance goal. Our goal is to
achieve high performance parallel signal processing systems with reduced complexity,
while meeting the requirements of configurability and flexibility.

The central part of this thesis is algorithm optimization, workload partitioning
and design space exploration of a few key algorithms in the fields of wireless commu-
nication and computer vision. We present a GPU-based turbo decoding implementa-
tion, followed by a low-complexity high-throughput parallel interleaver architecture
for multi-standard turbo decoding, first enabling very high data rate for an HSPA+
turbo decoder. We present high throughput LDPC decoding algorithms based on
the massively parallel architecture of GPU, achieving high throughput close to VLSI
implementations, and for the first time, the long decoding latency issue is studied
and solved with the proposed scheduling method. We also propose heterogeneous
implementations to accelerate two representative computer vision algorithms (object
removal algorithm and SIFT algorithm) using both mobile CPU and GPU. We further
explore the design space and analyze several key aspects, including performance re-
quirement, power consumption, memory architecture, flexibility, design effort of the proposed optimized accelerator designs and implementations based on customized hardware accelerator and massively parallel architecture on mobile platforms.

1.2.1 Thesis Statement

GPGPU is typically considered as a powerful parallel computing tool to speed up many computationally-intensive tasks. However, for modern wireless communication and mobile computing systems, GPGPU is not always the best candidate to accelerate the complicated applications, due to limitations such as low algorithm parallelism, high memory bandwidth requirement and so on. Partitioning the tasks onto multiple accelerators in heterogeneous architecture is the key to efficiently accelerating parallel algorithms to meet the system requirements of performance, efficiency, and flexibility.

1.2.2 Thesis Contributions

In this thesis, we will study the architectural characteristics, performance bottleneck, design techniques, strategies and methodologies used when building the accelerators for computationally-intensive DSP algorithms used in wireless communication and mobile computing applications. We focus on the use of GPU as accelerator and study whether GPGPU is a good platform allowing us to achieve the performance goal for the four use cases studied in this thesis.

The thesis work has generated 17 technical papers, 1 book chapter, 1 U.S. patent, and 2 U.S. patent applications [33–56]. The contributions of this thesis are summarized as follows.

1. Algorithm analysis and architectural mapping onto GPGPU: We study the characteristics of the most computationally-intensive algorithm in four representative use cases. We explore the design space including algorithm parallelism,
arithmetic complexity, memory bandwidth requirement, memory access patterns and so on. We propose a methodology to determine whether an algorithm could be successfully accelerated by a GPU by comparing the algorithm analysis results to the system specifications. From the algorithm study and mapping, we found that some algorithms are very suitable for GPU’s architecture, and we can easily achieve 10X or even 100X speedup by adapting the algorithms to the parallel architecture of a GPU. For some other algorithms, initially it is difficult to achieve the required performance by direct algorithm mapping. However, with certain algorithm transformation and optimization, we are able to accelerate these algorithms and meet the performance requirement. Finally, for some other algorithms, even after we heavily optimize the algorithms and adapt them to the parallel architecture, we still cannot achieve the performance requirement due to the inherent limitations in the algorithm. For these algorithm, we propose to map the algorithms onto other accelerators, such as ASIC or CPU + GPU heterogeneous architecture.

2. High throughput parallel interleaver for multi-standard turbo decoder [33–40, 55]:

We show that the GPU-based turbo decoder implementation cannot meet the performance requirements. The very high throughput requirement forces us to implement the algorithm using ASIC technology. With ASIC implementation, there are some challenges preventing the implementation from achieving very high parallelism and high throughput. One of the most difficult challenges is the memory conflict problem, which is crucial to the overall system performance and complexity. We first propose a balanced scheduling scheme for turbo decoding and effectively eliminate the memory reading conflict problem. We create a cycle-accurate simulator of parallel turbo decoding and parallel interleaver architecture. With this simulator, we study the statistical properties of the parallel turbo decoder, especially the memory access
patterns with different interleaver algorithms. Based on the statistical results, we propose a double-buffer contention-free (DBCF) architecture to efficiently solve the memory writing conflict problem with very low hardware complexity. Furthermore, we propose a unified interleaver/deinterleaver architecture which enables hardware sharing between interleaver mode and deinterleaver mode and leads to significant reduction in hardware complexity. The proposed parallel interleaver computes the interleaving/deinterleaving addresses on the fly, therefore, significantly reducing the memory requirements compared to traditional look-up table-based solutions.

The proposed parallel interleaver architecture can support very high throughput turbo decoding with very low hardware overhead. The proposed architecture provides great flexibility and configurability, which supports different codeword block sizes, different parallel turbo decoding architectures, different interleaver algorithms specified by wireless communication standards, and different parallelism degrees to support divergent performance requirements.

Utilizing the proposed parallel interleaver architecture, we implemented a multi-standard turbo decoder supporting HSPA+/LTE/LTE-A standards. To the best of the authors’ knowledge, the proposed turbo decoder is the first to support over 672 Mbps data rate for the future extension of the HSPA+ standard. The implementation results show that the proposed architecture shows the best hardware efficiency compared to the related work.

3. Massively parallel LDPC decoding using GPGPU [41–45]: Traditionally, LDPC decoding is implemented using ASIC due to the very high throughput requirement. However, once designed, an AISC implementation lacks flexibility to support an SDR system. We investigate the capability of GPU to be used as a platform for accelerators in software-defined radio (SDR) systems, in which the radio signal processing and transmission is controlled by software stacks on top of config-
urable hardware such as FPGA, general-purpose processor and so on. We study the LDPC decoding algorithms and properties of the GPU architecture. We first choose appropriate LDPC decoding algorithms which are suitable for a massively parallel implementation on GPU many-core architecture. Algorithmic optimization techniques are proposed to fully utilize GPU computing resources and improve the decoding throughput. We propose to use two-min min-sum algorithm for LDPC decoding, and loosely coupled LDPC decoding algorithm to reduce the computation requirement and memory access bandwidth requirement. The LDPC decoding algorithms are partitioned into two stages and mapped onto two GPU kernels, which guarantee the synchronization of data and the unified behavior of threads inside the same thread blocks, and therefore provide very high throughput. To fully utilize the streaming multiprocessors and memory bandwidth of the GPUs, we propose a multi-codeword packing technique, which not only provides a large enough and scalable workload to GPU, but also allows all threads in the same thread block to follow the same execution path and reduce the execution latency.

To reduce the memory access overhead, we use efficient data structures to store parity-check matrix \( H \), check-to-variable messages, and variable-to-check messages. We propose fully coalesced memory access to further reduce the memory access latency by utilizing the shared memory as a software-manageable data cache.

In addition, to further improve the decoding throughput and reduce decoding latency, asynchronized memory data transfer between host (CPU) and device (GPU) are employed to hide the latency of data movement before and after new decoding iterations for new data blocks. We also propose to use multi-streaming programming to partition the workload into small pieces and assign them to each stream. Although the overall workload remains large enough to fully occupy the GPU’s computation resources, the decoding latency is significantly reduced due to the appropriate schedul-
ing of multiple streams. Experimental results show that the proposed GPU-based LDPC decoder provides very high throughput, and at the same time shows very small decoding latency.

4. Mobile computer vision accelerators based on OpenCL GPGPU co-processing [46–49]: Mobile computer vision applications have become more and more popular on mobile devices. However, the limited hardware resources on mobile devices still prevent many mobile computer vision applications from being practically used.

We study two representative algorithms for image processing and computer vision: the object removal algorithm and SIFT algorithm. We offload computationally-intensive modules from the mobile CPU to mobile GPU to reduce the processing time and energy consumption by using the OpenCL programming mode. This makes the computationally-intensive algorithms feasible for computer vision applications on mobile systems.

We first partition the algorithms into several key kernels. We benchmark the object removal algorithm and profile the processing time for each key block. Based on the experimental results, algorithm bottlenecks are identified. Mobile GPUs cannot provide enough processing power to meet the performance goal due to these algorithm bottlenecks. We explore the design space and propose algorithmic optimization strategies to improve, for example, the performance of the findBestPatch kernel: increasing patch size and reducing the search area. We also propose memory access optimization by utilizing the fast on-chip local memory of the mobile GPU. Design trade-offs and the impact of algorithm parameters are explored by experiments. By optimizing the algorithm, the final GPU implementation meets the performance goal and shows good energy efficiency.

The SIFT algorithm is well known for its exceptional performance and very high
computational complexity. To deploy the SIFT algorithm onto mobile platforms, algorithmic and architectural optimizations are necessary. We study the characteristics of the SIFT algorithm and partition the workload into several kernels. We compare several partition methods and choose the optimal partition method based on the total processing time which consists of execution time and memory data transfer time. We propose the recursive Gaussian blur algorithm to reduce the complexity of the algorithm based on the properties of the mobile GPU. Moreover, several optimization techniques including data packing and dynamic kernel generation are employed to boost the performance. The final CPU+GPU heterogeneous implementation demonstrates shorter processing time and lower energy consumption, both of which show significant improvement compared to the CPU-only implementation.

5. Design space exploration of mobile accelerators:

Although many previous studies have explored architectural differences and performance comparisons among numerous architectures, most of them focus on high performance computing on desktop or workstation environments. Few of them focused on emerging architectures on embedded platforms and mobile computing environments. The rapid development of heterogeneous mobile computing systems provide more possibility to take advantage of diverse low-level architectures to accelerate different types of algorithms. We explore the design space and study the characteristics of different accelerators for mobile systems, based on which we compare different architectures including custom logic, multi-core CPU and GPGPU on mobile devices in terms of computational complexity, memory architecture, flexibility, cost and performance.
1.2.3 Thesis Outline

In Chapter 2, we will introduce the architecture of mobile computing systems. We give an overview of the architecture of GPGPU and related programming models. We also describe the methodology and analysis steps for algorithm study. In Chapter 3, we will look at an algorithm suitable for GPU acceleration. We present a GPU-based high throughput low latency LDPC decoder for SDR systems. In Chapter 4, we study the turbo decoding algorithm and present the fact that even with optimizations, the GPU cannot provide enough resources to make the GPU implementation meet the performance goal. Instead, we implement a high throughput multi-standard turbo decoder supporting HSPA+/LTE/LTE-A using ASIC technology. We also present a parallel interleaver architecture for a multi-standard turbo decoder to make the ASIC implementation feasible. In Chapter 5, we will present algorithmic and architectural optimization strategies for both the object removal algorithm and SIFT algorithm using OpenCL targeting mobile computing platforms. We will show that algorithm transformation and CPU+GPU heterogeneous computing can make the originally infeasible algorithm benefit from the GPGPU. In Chapter 6, we describe a design space exploration strategy and compare the customized VLSI architecture and GPU from different aspects including performance, cost, power, memory architecture, flexibility and so on. Finally, Chapter 7 summaries the thesis.

1.3 List of Symbols and Abbreviations

Here, we provide a summary of the abbreviations and symbols used in this thesis:

- **ALU**: Arithmetic logic unit.
- **APP**: A *Posteriori* probability.
- **ASIC**: Application-specific integrated circuit.
**AWGN**: Additive white Gaussian noise.

**BER**: Bit error rate.

**BPSK**: Binary phase shift keying.

**CMOS**: Complementary metal-oxide-semiconductor silicon technology.

**CNP**: Check node processing.

**CTV**: Check node to variable node.

**CUDA**: Compute Unified Device Architecture.

**FEC**: Forward error correction.

**FER**: Frame error rate.

**FPGA**: Field-programmable gate array.

**GPGPU**: General-purpose computing on graphics processing units.

**GPU**: Graphics processing unit.

**HDL**: Hardware description language.

**HSDPA**: High-speed downlink packet access.

**HSPA+**: Evolved High-Speed Packet Access.

**OpenCL**: Open Computing Language.

**LDPC**: Low-density parity-check.

**LLR**: Log-likelihood ratio.

**LTE**: Long-Term Evolution.

**LTE-A**: Long-Term Evolution Advanced.

**MIMO**: Multiple-input and multiple-output

**MAP**: Maximum *A Posteriori*.

**NB-LDPC**: Non-binary low-density parity-check.

**TSMC**: Taiwan semiconductor manufacturing company.

**PCCC**: parallel concatenated convolutional code.

**QC-LDPC**: Quasi-cyclic low-density parity-check.
**QPP**: Quadratic permutation polynomial.

**RTL**: Register transfer level.

**SDR**: Software-defined radio.

**SIFT**: Scale-invariant feature transform.

**SISO**: Soft-input soft-output.

**SNR**: Signal-to-noise ratio.

**SoC**: System-on-chip.

**SURF**: Speeded Up Robust Features.

**VLSI**: Very-large-scale integration.

**UMTS**: Universal mobile telecommunications system.

**VNP**: Variable node processing.

**VTC**: Variable node to check node.

**WCDMA**: Wideband code division multiple access.

**WiMAX**: Worldwide interoperability for microwave access.

**WLAN**: WLAN: Wireless local area network.
Heterogeneous computing has become the trend of mobile architecture to incorporate capabilities to perform a wide range of tasks and enable different applications. Especially, during the past decade, GPU compute has become a very attractive approach to accelerate complicated algorithms. Among different applications, high performance computing (HPC) area has been changed significantly since the advent of GPU compute. Nowadays, GPU compute has become an essential tools for HPC. As the GPU designed for personal computer keeps growing and gaining higher computation power, GPU has also been used in more general areas in addition to HPC, including but not limited to image processing, video compression, cryptography, packet processing for network systems, machine learning, computer vision and so on.

However, due to the quite divergent characteristics of algorithms and corresponding hardware/software architecture, it is still challenging to partition workloads, accelerate algorithms on GPUs, and fully utilize GPUs resources. Therefore, algorithm transformation and architectural mapping are the major challenges to designing and implementing high performance accelerators based on GPGPU. For a lot applications, even with deep optimization, it is still hard to achieve required performance on GPU systems. It is crucial to study the characteristics of algorithms and compare
the achievable performance to guide the algorithm and system design. In this thesis, we will show both the capability and also limitations of GPU for accelerate applications in wireless communication and mobile computing algorithms through several case studies.

2.1 Related Work

With the advent of reconfigurable computing and GPU computing, many previous studies performed comparison among different hardware architectures such as general-purpose processors (GPP), field-programmable gate array (FPGA), and graphics processing units (GPU) on different computationally-intensive applications. Most of the previous work concluded that GPGPU outperforms other accelerators with its huge computation power and very high parallelism, especially in the high-performance computing field.

An early study performed by Che et al. [57] evaluated Gaussian elimination, data encryption standard, and Needleman-Wunsch sequence alignment, and concluded that FPGA outperformed the GPU in terms of performance, but with higher development costs. Howes et al. [58] presented the atomated compilation of source code for GPU, FPGA and Playstation 2 using the ASC (A Stream Compiler) compiler. In that study, GPU has shown better performance for Mente Carlo computations. Gac et al. [59] compared performance of 3D tomography computation on CPU, GPU and FPGA, and showed that GPU obtained the best absolute performance, while the FPGA has smaller clock cycle requirements.

A matched filter algorithm was evaluated by Baker et al. [60] on Cell processor, FPGA and GPU. The authors compared device cost, power consumption and performance. They concluded that the GPU provides the best speedup per unit of price and the FPGA is superior in performance per unit of energy.
processor, GPU and FPGA in a high performance computing system was discussed by Kelmelis et al. [61]. However, they did not provide comparisons among three architecture. Similar discussion was done by Mueller et al. [62] with computed Tomography applications, concluding that the GPU is better for giga-flop performance applications.

A survey of GPGPU was presented by Owens et al. [63]. The authors concluded that memory access reduction operations and the lack of memory scatter capability are weaknesses of GPGPU.

Research work done by Weber et al. [64] showed that GPU is better for high performance computing, and algorithms needs to be fine tuned to achieve high performance, in Quantum Monte-Carlo study.

Another comparison performed by Williams et al. [65] analyzed the characteristics of numerous devices, claiming that the GPU provides better computational performance for computationally-intensive applications; while, the FPGA is superior in applications with intense bit-level operations and integer operations.

In the field of image processing, research conducted by Asano et al. [66] found that FPGA has superior performance for complex applications, e.g. stereo imaging based on local window-shifting and k-means clustering. They also found that GPU is better at simple computations such as 2D convolution.

Bodily et al. [67] studied optical flow and MIMO digital wireless communication accelerators with GPP, FPGA and GPU, concluding that FPGA provides better flexibility for custom I/O with computations and GPU provides better aggregate performance by buffer large amounts of input data to utilize GPU’s resources. Therefore, FPGA provides better pipelined implementation with flexible data interface; while, the GPU shows longer processing latency, but still providing better throughput performance.
Future potential microprocessor architecture is evaluated by Chung et al. [68]. The authors concluded that the future microprocessor should include custom logic, FPGAs, and GPGPUs, because the combination can provide significantly performance gains. They also stated that even when bandwidth is not a concern, flexible GPUs and FPGAs are still competitive with custom logic if parallelism is moderate to high, although they are very efficient.

Cope et al. [69] presented a systematic approach to compare FPGA and GPU for image processing algorithms. They found that the GPU performs better than a fixed-point FPGA implementation, and for floating-point implementation, the GPU is superior by a further 12 times. The FPGA is superior for algorithms with large degrees of data dependencies and number of memory accesses. They also identified the key architectural features for GPU implementations to achieve impressive algorithm accelerations as memory systems, degree of multi-threading, and iteration parallelism.

The work done by Pauwels et al. [70] extended work research from Cope et al. [69] by comparing different architectures using medium to highly complex computer vision algorithms that can stretch the FPGA to its limits. They stated that high bandwidth provided by newer GPU architecture enables GPU to outperform the FPGA in terms of absolute performance, when complex models are implemented. As opposed to [69], which states that the FPGA’s embedded memory bandwidth make it superior than GPU for local and deterministic processing, the authors in [70] observed that external and random memory accesses are more suitable for the newer GPU architectures and their texture units. However, due to the size of GPUs, they are still not suitable for many embedded applications.

A recent work by Fowers et al. [71, 72] explored 1D convolution implementations, with analysis of different devices, input sizes, optimizations, and algorithms. They concluded that for small input data sizes, the FPGA implementation shows the best
performance, with GPU achieving comparable performance. For large kernel and data sizes, FPGA and GPU provide similar performance results, and GPU is slightly better than FPGA. Based on their experimental results, they suggested emerging architectures integrate GPUs and CPUs on a single chip to improve the energy and memory data transfer efficiency. They also suggested that the CPU-GPU processors should be evaluated with larger FPGAs or integrated logic to achieve better performance and efficiency.

Concluding the previous works, we find that most of the previous works claim GPGPU can show great advantages in terms of the computing power, flexibility and efficiency, for many high-performance computing algorithms and some of the image processing applications. In this thesis, we want to extend the application field to the wireless communication algorithms and mobile computing area. We want to see whether GPGPU are still a potential good choice for these applications. Although many previous studies have explored architectural differences and performance comparisons among numerous architectures, most of them focus on high performance computing on desktop or workstation environments. Few of them focused on emerging architectures on embedded platforms and mobile computing environments. The rapid development of heterogeneous mobile computing systems provide both challenges and opportunities for system-level design and optimization. Therefore, in this thesis, extending some of the methodologies employed in some of the previous work, we study the characteristics of different accelerators for mobile devices and compare different architectures in terms of computational complexity, memory architecture, flexibility, cost and performance.
2.2 Heterogeneous Accelerator Architecture for Wireless and Mobile Computing Systems

As we have described above, the modern chipsets designed for wireless and mobile computing systems provide multiple accelerators for different types of applications, including GPGPU, custom hardware logic, DSP, SIMD processors, and so on. In this thesis, we mainly focus on the performance and capability analysis of GPGPU as an accelerator for wireless and mobile computing systems.

To better understand the algorithm mapping and system design considerations, we first briefly study the hardware architecture, performance capability, programming model for GPGPU. We also briefly introduce other accelerators such as custom hardware logic, SIMD processors, and DSP.

2.2.1 General-Purpose Computing on Graphics Processing Units (GPGPU)

As an emerging many-core platform for high performance computing, GPU (graphics processing units) has been employed in a wide range of applications due to their immense computing power compared to traditional CPUs [73, 74]. Although designed originally for high quality graphics rendering, GPUs were found to have very regular structure and are suitable for massively parallel computing tasks. Especially during the past few years, new parallel programming models such as CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) have significantly reduced the effort required to program a GPU [75, 76]. Therefore, GPGPU has become an essential component in heterogeneous computing platforms, and has attracted a great deal of attention from almost every field in scientific and engineering computing. Other than image analysis and processing [77, 78], GPGPU have
shown to provide performance improvements for a large range of computationally-intensive applications, such as ocean simulation [79], storage systems [80], network packet processing [81, 82], cryptography [83], signal processing for wireless communication [84, 85], financial simulation [86] and so on.

Recently, as the mobile computing technology evolves, mobile GPUs also start to gain the general purpose computing capability via programming models such as CUDA and OpenCL. Since it becomes possible, GPU compute has been explored and utilized for numerous mobile applications, such as camera pipeline acceleration, image post-processing, and mobile computer vision. These applications have provided the SoC and mobile device vendor a way to show differentiation between competitors in high-end product lines. However, even the low-end products have started to get benefits from GPU compute. For example, in some low-end mobile SoC chipsets, hardware accelerators for high-definition video codec and JPEG image codec have been removed to cut the cost. Instead, these functions are achieved through GPU compute. Therefore, not only the powerful flagship mobile devices can be beneficial from GPGPU, but also these low-end products can achieve make possible some advanced features through with the help of GPGPU.

**Programming Models of GPU**

Regarding to GPU programming, CUDA and OpenCL are two most popular programming models. Since CUDA and OpenCL have very similar architecture, without loss of generality we use CUDA terminology to describe the GPU programming model in this section, but please note that most of the concepts described here can be applied to OpenCL. Therefore, in this section, we only describe the programming model for CUDA, more details about OpenCL can be found from [76]. A brief comparison of terminologies in CUDA and OpenCL is shown in Table 2.1.
Table 2.1: Terminology comparison between CUDA and OpenCL.

<table>
<thead>
<tr>
<th>Category</th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread model</td>
<td>thread</td>
<td>work item</td>
</tr>
<tr>
<td></td>
<td>thread block</td>
<td>work group</td>
</tr>
<tr>
<td></td>
<td>grid</td>
<td>NDRange</td>
</tr>
<tr>
<td>Memory model</td>
<td>device memory</td>
<td>global memory</td>
</tr>
<tr>
<td></td>
<td>constant memory</td>
<td>constant memory</td>
</tr>
<tr>
<td></td>
<td>shared memory</td>
<td>local memory</td>
</tr>
<tr>
<td></td>
<td>local memory</td>
<td>private memory</td>
</tr>
<tr>
<td>Synchronization</td>
<td>__syncthreads()</td>
<td>barrier()</td>
</tr>
</tbody>
</table>

Figure 2.1: Diagram of a modern GPU architecture.
Heterogeneous Architecture

The architecture of a modern GPU is shown in Fig. 2.1. A CUDA application is partitioned into two parts: host code and kernel code. The host code typically runs on CPU, and it creates computing context, creates and initialize memory buffer on the GPU device, compiles the kernel code, launches GPU kernel execution, and also executes the serial part of algorithms which are not suitable for GPU acceleration. The kernel code is written in C-like language (C with CUDA-specific extensions), which defines internal variables (such as thread id, dimension of thread block and so on) and built-in functions (such as vector functions, atomic operations and so on). The host code and kernel code for an example of vector addition are shown in List 2.1 and List 2.2, respectively. All the threads in the same work group execute the same kernel code in a SIMT manner.

Listing 2.1: An example CUDA host code for vector addition.

```c
void main()
{
    int N = 1024;
    float * h_A = (float *)malloc(N * sizeof(float));
    float * h_B = (float *)malloc(N * sizeof(float));
    // Initialize input vector h_A and h_B.
    ...
    float *d_A, *d_B, *d_C;
    cudaMemcpy(&d_A, N * sizeof(float));
    cudaMemcpy(&d_B, N * sizeof(float));
    cudaMemcpy(&d_C, N * sizeof(float));
    cudaMemcpy(d_A, h_A, N * sizeof(float),
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(float),
               cudaMemcpyHostToDevice);
    int threadsPerBlock = 256;
    int blockPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    vecAdd<<<blockPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cudaMemcpy(h_C, d_C, N * sizeof(float),
               cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
```
Execution Model

Modern GPUs contain many streaming multiprocessors (SMs), each of which consists of multiple stream processors (SPs). Each SM works as an independent SIMT (single instruction, multiple threads) processor. The GPUs are massively parallel architecture with thousands of threads running in parallel on SMs. The basic execution unit is called a warp, a group of 32 threads, sharing the same program counter. The threads in a warp execute in a SIMD way, and they follow the same execution path. If there is a divergent branch, then threads in the warp will follow divergent branches one by another, leading to poor performance due to insufficient usage of GPU cores and longer execution paths.

Block and Grid Structures

To easily manage massive numbers of threads, CUDA utilizes a hierarchical structure to group all the threads. According to the characteristic of algorithms, threads can be grouped to one-dimension (1D), two-dimension (2D) or three-dimension (3D) thread blocks. Furthermore, thread blocks can be further grouped into 1D, 2D or 3D grids.

Thread blocks can be executed independently and can be scheduled in any order based on the availability of the SMs. Threads within a block can cooperate by sharing
data via shared memory. However, the relax memory coherence in GPU programming
does not guarantee all threads see the same data in a shared memory. Developer can
force all thread in a block to synchronize by calling synchronization intrinsic function,
e.g. \texttt{__syncthreads()} in CUDA or \texttt{barrier()} in OpenCL.

\textbf{Memory Model}

CUDA defines a hierarchical memory space, which includes global memory, shared
memory, private memory and constant memory. Each thread has its own private
memory. The global memory is a large off-chip memory which provides typical several
gigabates capacity and can be accessed by all threads. However, global memory suffers
from very long access latency (several hundred clock cycles). Shared memory is visible
to all threads in a thread block and can be used to shared data across all threads
in the same block. Shared memory is fast on-chip memory, but with small capacity
(typically 32 48KB). The shared memory can be used as a software-managed cache,
allowing for data prefecth or cache to store frequently used data. However, due to
the limited size of shared memory, the algorithms should be carefully designed and
optimized to take advantage of the fast on-chip shared memory. The constant memory
is read-only and can be accessed very fast if all the threads in a block access the same
address in the constant memory in a broadcast way.

\textbf{Insights of GPU Programming}

It is not straightforward to achieve peak performance for a GPU-based heteroge-
neous implementation. The following aspects should be carefully considered when
mapping a algorithm onto GPU architecture.

1. We should exploit the parallelism in the algorithm. Some times the algorithms
   need to be modified to adapt to GPU’s architecture.
2. To fully utilize all the computation resources of GPUs, data-level parallelism or task-level parallelism should be exploited.

3. Data transfer between host and device takes significantly time and may become the bottleneck. We should optimize algorithms to minimize the data transfer between host and device. If possible, asynchronized memory operations can be utilized to hide the data transfer time.

4. We should carefully design the data structure suitable for GPU architecture and minimize the memory transaction. Coalesced memory access can be achieved by carefully design the data layout and alignment. Memory coalescing can significantly reduce the number of memory transactions, and therefore lead to improved performance.

5. Accessing to global memory has long latency which can lead to performance degradation. Shared memory can be used to store frequently-used data. However, the data coherence should be considered and necessary synchronization primitives should be added.

2.2.2 Other Accelerators

Customized Hardware Logic Modern mobile SoCs typically include one or more customized hardware logic modules, which perform very specific tasks and meet very high performance requirements.

Customized hardware logic can appear in many different forms, such as ASIC (application-specific integrated-circuit), ASIP (application-specific instruction-set processor), FPGA (field-programmable gate array), vector processing unit, and so on. These customized hardware logic are implemented targeting very specific algorithms or applications, focusing to solve different problems. The goal, however, is the same,
which is to provide extremely high performance while achieving the best power/energy efficiency.

As the processors and other more general accelerators evolve, some algorithms which were required to implement using custom hardware can achieve the required performance on CPUs or GPUs. However, there are still quite many complex algorithms, which seems to be too complicated for a CPU or GPU to handle. For example, the communication modems are always implemented using ASICs. These communication modems are the heart of the connectivity functionality of modern mobile devices. They are required to run all the time with very low power consumption. ASIC design is almost the only choice for these kinds of algorithms. Another good example is the ISP (image signal processing), which provides rich functions to handle image signals coming from the camera pipeline in the system, or coming from the user-level applications. The huge amount computation needed and sometimes the real-time processing requirements both make the ASIC the best choice to deploy an ISP.

However, as we are forced to use custom hardware logic, we should notice that the developing cost for the custom hardware logic is the biggest challenging for such design. Moreover, the lack of flexibility shortens the life-cycle of a design, again pushing the design cost even higher. For a custom hardware logic design, once the design is done, it is very difficult to add or change the functionality of a chip. Therefore, for a modern mobile system, we prefer to explore other alternative programmable accelerators to achieve the same (or similar) performance as the custom hardware logic could possibly provide. Although we may lose some performance and power efficiency, we still gain easy-to-program, fast prototyping ability, flexibility, ability to update after deployed, and possibly low cost. In addition, the programmable accelerators will continue to evolve to provide even more computation power to enable many
algorithms that are impossible for these platforms.

**SIMD Accelerator of General-Purpose Mobile CPU** A general purpose processor (such as CPU) typically provides flexible program flow and performance optimization for serial programs. Multi-core programming can be achieved by using programming models such as PThread or OpenMP. However, when performing operations on 32-bit or 64-bit microprocessors, parts of the computing power are underutilized. Modern processors usually use SIMD (single instruction, multiple data) technology to allow the processors to perform multiple operations in parallel on multiple data elements of the same type and size. For example, through vectorization, a processing hardware can perform four parallel additions on 8-bit data, instead of adding two 32-bit numbers, if the precision of 8-bit can meet the requirements. In this simple example, we simply get about 4 times speedup.

On mobile devices, ARM processors are typically integrated into the mobile SoCs. Since its ARM v7 architecture, a SIMD extension called NEON was introduced to provide vectorization capability to ARM processors [87]. It defines the SIMD concepts via groups of instructions operating on vectors stored in 64-bit double word registers. The NEON instruction sets extend the original ARM ISA (instruction set architecture) by defining assembler directives, labels and comment indicators.

For some applications such as image filtering, in which the precision requirement is not high, NEON is proved to improve the processing speed by a few times. However, there are some fundamental problems with NEON as well as the CPU itself, so in many cases, CPUs are not optimal accelerators for algorithms. Firstly, NEON programming requires the developer to write assembly code by hand, which leads to long learn curve and very long developing/debugging period. Secondly, the CPU provides very limited parallelism (even if we can tolerate the 8-bit precision, most of the ARM processor can only provide a parallelism of 16 to 32), which limits the achievable arithmetic
throughput. Thirdly, CPUs usually run operating system on top of it as well as some managing tasks. Therefore, the computing tasks must not fully occupy the CPU’s hardware, otherwise the system will freeze or even fail under some cases. Lastly, CPU runs at much higher clock frequency (several GHz) than other accelerators (several hundred MHz) in the SoC, so the power/energy consumption of CPUs is higher compared to other accelerators.

Based on the above analysis, to achieve better overall performance and system balance, our design goal will be mapping the computational-intensive modules onto GPUs and other accelerators if possible. For tasks with many branching operations and tasks with very low parallelisms, we will keep them running on CPU. The final solutions for each application/algorithm will demonstrate that heterogeneous computing is essential on mobile SoCs to achieve better performance and power balancing.

**Mobile Digital Signal Processor (DSP)**  To efficiently accelerate multimedia and communication signal processing, some SoC chips integrate a light-weight mobile DSP. These mobile DSPs are designed to enable device manufacturers and independent software providers to optimize the features and performance of multimedia software. These optimizations help enable audio, imaging, embedded vision and heterogeneous computing acceleration on mobile SoCs.

The Hexagon DSP is one of the famous mobile DSPs which can be found inside a Snapdragon mobile SoC chipset. The Hexagon architecture is designed to deliver performance with low power consumption. It has features such as hardware assisted multithreading, privilege levels, VLIW, SIMD, and instructions geared toward efficient signal processing. The CPU is capable of in-order dispatching up to 4 instructions (the packet) to 4 Execution Units every clock. Hardware multithreading is implemented as barrel temporal multithreading - threads are switched in round-robin fashion each cycle.
The Snapdragon SoC chipsets include several Hexagon DSP cores, including two Hexagon cores for modem signal processing, and one core for multimedia processing. The mobile DSP typically utilize C as programming model, however, supporting intrinsic instructions or even inline assembly code to achieve the optimal performance.

The mobile DSPs (such as Hexagon) have good support for branching operations and conditional operations, therefore, mobile DSPs can handle complicated control-orientated programs, or even an operating system. Although the mobile DSPs can achieve very low power consumption (due to very low clock frequency, typically 200MHz/thread), there are some limitations one need to consider when porting an algorithm onto the mobile DSP platforms. Firstly, the mobile DSPs are good at fixed point processing, and it only contains very limited floating-point units. Therefore, compared to mobile CPU and GPU, the mobile DSP can provide only very limited floating-point computation power. Secondly, the mobile DSPs contains 2-3 hardware thread per core, therefore, considering the VLIW and SIMD capability, the mobile DSP can provide the parallelisms between 32 64, which is higher than the mobile CPU, but still much lower than a mobile GPU. Thirdly, to achieve better performance, assembly-level programming is required, which makes DSP-based application difficult to develop. Usage of assembly programming also reduces the portability of a program across different generations of hardware.

Due to the above limitations, although the mobile DSP sounds promising in terms of the power efficiency, compared to mobile CPU and GPU, it has not yet shown significant advantages over the other two. Instead, the difficulties to develop highly-optimized code makes it hard to be used in real-world applications. In fact, in the current mobile systems, mobile DSPs are mainly used for packets control and processing for modem signals. We have not yet seen many mobile DSP-accelerated computationally-intensive applications.
Therefore, in this thesis, we will not analyze and measure the performance of algorithms on mobile DSP, instead, we try to map the algorithm either onto mobile CPU or GPU based on the characteristics of algorithms. However, in future work, it will be interesting to see the performance of the real-world use cases on mobile DSP. Although the mobile DSP may not provide the same performance as the mobile GPU can provide, and it cannot beat the mobile CPU for flexibility and control orientated applications. We can identify the bottleneck of the mobile DSP, and provide design guidelines to make the mobile DSP more useful. But for this work, we will keep our scope among customized hardware logic, mobile CPU and mobile GPU.

2.3 Methodology

In this thesis, we take a few representative use cases from wireless communication and mobile computing as examples, analyze the algorithm characteristics, and try to identify the performance bottleneck. Specifically, we will study algorithm transformation and optimization, and check if these effort can make the GPGPUs suitable accelerators for these kinds of algorithms.

2.3.1 Algorithm Mapping Guidelines

The mobile platforms provide heterogeneous architecture, which give the developers opportunities to choose different platforms to implement an algorithm and achieve the performance requirements. However, algorithm and system designers prefer to avoid the customized logic, due to the high cost, strict area constraint, and poor flexibility. Therefore, we prefer to map the algorithms onto programmable mobile processors, such as mobile CPU or the mobile GPU.

For a given workload (in this thesis, we present two representative use cases for
Figure 2.2: Methodology of algorithm analysis and mapping to heterogeneous architecture.
Table 2.2: Summary of compute capability of accelerators.

<table>
<thead>
<tr>
<th>Accelerator</th>
<th>Computing performance</th>
<th>Memory bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customized logic</td>
<td>Very high</td>
<td>Very high</td>
</tr>
<tr>
<td>GPGPU</td>
<td>2~4 TFLOPS</td>
<td>~200 GB/s</td>
</tr>
<tr>
<td>Mobile GPU</td>
<td>100~130 GFLOPS</td>
<td>~15 GB/s</td>
</tr>
<tr>
<td>Mobile CPU</td>
<td>~10 GFLOPS</td>
<td>~15 GB/s</td>
</tr>
</tbody>
</table>

wireless communication systems, and two very representative computer vision use cases as workloads), we first analyze the algorithm properties including parallelism (high or low), arithmetic complexity (high or low), memory access pattern (regular or irregular), and memory bandwidth requirement (high or low). Based on the algorithm analysis and the specific system specifications (such as the compute units available on the platform, the clock frequency, the theoretical peak performance, and so on), we will try to map the algorithms on to the processors and custom logic on the platform. At this time, we can identify the bottleneck for such algorithms when running on certain accelerators. We can then further transform the algorithms to utilize the hardware/software features of the accelerators to remove some bottleneck. But it should be noticed that in some cases, even highly optimized design cannot achieve expected performance due to some limitations (or mismatching) in the hardware and algorithms. If this is the case, we have to explore other opportunities for possible performance acceleration.

2.3.2 Capability of Different Accelerators

The computing capability of the major accelerators can be summarized in Table 2.2.

As we can see from the table, the customized logic is good for high complexity high memory bandwidth algorithms with irregular memory accesses. However, the disadvantages of the customized logic include high cost, long developing cycle, increase of chip area, poor flexibility and so on. The GPU is good for highly parallel algo-
Figure 2.3: Steps to identify the algorithm bottleneck based on architecture spec of an accelerator.

algorithms with regular memory access patterns. While, the CPU is good for algorithms with many branches and flow-control operations, but it provides lower performance compared to customized logic and the GPU.

2.3.3 Algorithm Analysis Steps

Figure 2.3 demonstrates the method to analyze the algorithm, determine the performance bottleneck, estimate the achievable performance. As mentioned above, we analyze four aspects of a give algorithm: parallelism, arithmetic, memory bandwidth, and memory access pattern. By analyzing parallelism of the algorithm, we can know if the algorithm can fully utilize the processor. If not, then the algorithm will not
be able to achieve the peak performance of the accelerator. Arithmetic complexity analysis allows us to check if the accelerator can provide enough computing power (raw GFLOPS of ALUs) to meet the requirements of the algorithm. We then compare the memory bandwidth requirement of an algorithm with the available memory bandwidth of the accelerator. Memory access pattern has huge impact on the memory load/store performance on many modern computer architectures. Some elements such as Memory alignment, randomness of the accesses, locality of the accesses and so on can vary significantly from algorithm to algorithm. By knowing the memory access patterns, it is possible for us to optimize the memory accesses by techniques such as data reorganization, data alignment, memory access coalescing, data packing and so on.

Based on the system specification of accelerator architecture, we can estimate the achievable performance of an algorithm. We can finally check if the achievable performance can meet the system requirements.

### 2.4 Summary and Challenges

GPGPU provides highly parallel architecture with complete software development toolchains as well as various programming models. Therefore, during the past several years, GPGPU has become a very hot topic in almost every fields where large computing power is needed. However, for heterogeneous architecture, especially on mobile systems, how to efficiently partition the workloads and whether GPGPU is still a potentially good accelerator, are still open questions.

In this thesis, we will focus on wireless communication signal processing and computer vision applications. To achieve peak performance, it is worth studying the characteristics of heterogeneous architecture of mobile SoCs, so that accordingly we can benefit from exploiting the parallelism in the computationally-intensive applica-
tions from several levels including algorithmic level, task level, data level and instruction level, and take advantage of properties of different accelerator architectures. For some use cases, the algorithm itself exhibits good parallelism, so that GPGPU could potentially improve the performance significantly.

However, efficient algorithm mapping and optimization are quite challenging due to the fact that few of the above-mentioned applications are inherently parallelizable and well-structured problems. Multiple limitations and constrains posed on the mobile architecture make it difficult for algorithms to take advantage of parallel architectures. The main impediments are irregular memory accesses, heavy data dependencies, highly complex inter-connection network, large memory unit requirement and so on. Due to the above reason, it is not always possible to achieve satisfactory speed-up on GPGPUs. For some use cases, due to the inherently low parallelism and very irregular memory access pattern, even with good amount of effort applied, the algorithm cannot benefit from GPGPU to meet the requirements. These algorithms should be implemented using custom hardware logic.

There are some other use cases, in which the original algorithms are not suitable for GPU processing. However, after algorithm transformation and optimization, we could take advantage of the heterogeneous architecture, especially the GPGPU, to achieve very satisfying results.

In summary, from next chapter, we present a few representative workloads as case studies including turbo decoding, LDPC decoding, object removal algorithm and SIFT algorithm by exploring several accelerator architectures. The methodologies of mapping algorithms to parallel architecture and performance optimization techniques are discussed in details. Specifically, we will look at what is the bottleneck for these applications, and whether GPU can help us accelerate the computational-intensive algorithms. We will demonstrate that for some cases, GPGPU can significantly im-
prove the processing speed with algorithm and architecture optimization. However, for some other use cases, the GPU implementation can hardly provide the required performance goal even with optimization techniques applied, due to some inherent limitations of the algorithms.
Low-Density Parity-Check (LDPC) codes are a class of error-correction codes which have been widely adopted by emerging standards for wireless communication and storage applications, thanks to their near-capacity error-correcting performance. Traditionally, LDPC decoding has been implemented using VLSI architecture, due to their high complexity and performance requirements. In this chapter, we will show that the modern GPUs can also be used to accelerate the LDPC decoding algorithm and the GPU-based implementation show comparable performance compared to the VLSI implementations.

During the past several years, researchers have been exploring GPUs’ parallel architecture and used GPUs as accelerators to speed up the LDPC decoding [41, 43, 85, 88–93].

Falcão first introduced GPU-based LDPC decoding using NVIDIA’s Compute Unified Device Architecture (CUDA) [75], and studied algorithm mapping onto GPU, data packing methods, and memory coalescing techniques [85, 88]. In [89], compact $H$ matrix representations and optimized memory access are studied for Quasi-Cyclic
LDPC codes. The forward-backward algorithm (FBA), optimized memory access and tag-based parallel early termination algorithm are discussed in our previous work [41]. Later, researchers studied the methodology to partition the workload based on availability of GPU’s resources, so that scalable LDPC decoding can be achieved on different GPU architectures [43, 90]. Kang proposed LDPC decoding based on unbalanced memory coalescing [91]. Recently, Falcão presented a portable LDPC decoding implementation using OpenCL [92].

Depending on the LDPC code structures and decoding algorithms, the current GPU-based LDPC decoding can normally achieve 50~150 Mbps peak throughput by packing a large number of codewords. As a side effect, the decoding latency becomes very high due to the data aggregation. Attracted by the highly parallel architecture and easy-to-use parallel programming environment provided by modern GPUs, researchers are attempting to build GPU-based software-defined radio (SDR) systems. In this scenario, reducing decoding latency is as important as increasing throughput.

In this chapter, we present a new GPU-based implementation of LDPC decoder targeting at future GPU-based SDR systems. Our goal is to achieve both high throughput and low latency. We will also compare our GPU implementation with the state-of-the-art VLSI implementation.

3.1 Introduction to Low-Density Parity-Check (LDPC) Codes

Low-density parity-check (LDPC) codes are a class of linear block codes providing near Shannon capacity limit performance. LDPC codes were originally introduced by Gallager in the early 1960s [22], and were ignored for a long time due to their high
computational complexity. Since their rediscovery by MacKay and Neal in 1990s [94],
LDPC codes have been one of the most attractive FEC codes for wireless communication systems, and have been adopted by numerous communication standards, such as
IEEE 802.11n/802.11ac WiFi, IEEE 802.16e WiMAX, China’s Digital Television Terrestrial Broadcasting standards (DTTB), Digital Broadcasting-satellite-Second Generation (DVB-S2) and so on.

3.1.0.1 Definition of LDPC Codes

A binary LDPC code is a linear block code defined by a sparse $M \times N$ parity-check matrix $H$. Each entry in the parity-check matrix is either a zero or a one, which nonzero entries are typically sparsely placed at random. When encoding a block of $K$ information bits, $N - K$ redundant bits are added to form a $N$-bit codeword (called a $(N, K)$ LDPC code), which has a code rate of $R = K/N$. Let $c$ denote a codeword, and $x$ denote a information bit block. The encoding process can be represented by

$$c = xG,$$  \hspace{1cm} (3.1)

where $G$ is a $K \times N$ generator matrix whose rows $g_0, g_1, \ldots, g_{K-1}$ form the span space of the $(N, K)$ LDPC codes. The generator matrix fulfills the following requirement

$$GH^T = 0.$$  \hspace{1cm} (3.2)

Thus, every valid LDPC codeword should satisfy the parity-check equation

$$H \cdot c^T = 0,$$  \hspace{1cm} (3.3)

which is normally used to check early-termination condition during a decoding process.
Parity-check matrix $\mathbf{H}$ can be represented by a Tanner graph containing $M$ check nodes (CNs) and $N$ variable nodes (VNs). Fig. 3.1 shows an example of parity-check matrix $\mathbf{H}$ and its corresponding tanner graph.

A check node represents a row in parity-check matrix, and a variable node represents a column in a parity-check matrix. The nonzero element in the parity-check matrix can be represented by a connection between a check node and a variable node, which is called an edge. Number of nonzero entries in a row (or column) of $\mathbf{H}$ is called row (or column) weight, denoted as $\omega_r$ (or $\omega_c$). An LDPC code is called regular if the weight of any check node and the weight of any variable node are constant. Otherwise, the code is called irregular. Generally speaking, irregular LDPC codes can slightly outperform regular LDPC codes in performance.
3.1.0.2 Quasi-Cyclic LDPC (QC-LDPC) Codes

Quasi-Cyclic LDPC (QC-LDPC) codes are a class of well-structured codes, whose matrix $H$ consists of an array of shifted identity matrices with size $Z$. QC-LDPC codes have been adopted in many standards such as IEEE 802.16e WiMAX and 802.11n WiFi, due to their good error-correction performance and efficient hardware implementation. Fig. 3.2 shows a typical $H$ of QC-LDPC codes, which contains $m_b \times n_b$ shifted identity matrices with different shift values $Z$. The WiMAX (2304, 1152) code and WiFi (1944, 972) code have similar structures, in which $m_b = 12$ and $n_b = 24$. $Z = 96$ and $Z = 81$ are defined in WiMAX (2304, 1152) code and WiFi (1944, 972) code, respectively.

3.1.0.3 LDPC Decoding Algorithms

The standard belief propagation algorithm (BPA), also known as sum-product algorithm (SPA) in the literature, was first proposed by Gallager [22]. In the SPA, belief messages are passed and processed between check nodes and variable nodes.

**Figure 3.2**: Parity-check matrix $H$ for a block length 1944 bits, code rate 1/2, IEEE 802.11n (1944, 972) QC-LDPC code. $H$ consists of $M_{sub} \times N_{sub}$ sub-matrices ($M_{sub} = 12, N_{sub} = 24$ in this example).
Let $c_n$ denote the $n$-th bit of a codeword, and let $x_n$ denote the $n$-th bit of a decoded codeword. The a posteriori probability (APP) log-likelihood ratio (LLR) of each bit $n$ can be defined as

$$L_n = \log \frac{\text{Prob}(c_n = 0)}{\text{Prob}(c_n = 1)}.$$  \hfill (3.4)

Let $Q_{mn}$ and $R_{mn}$ denote the messages from variable node (VN) $n$ to check node (CN) $m$ and the message from CN $m$ to VN $n$, respectively. The log-domain SPA iterative decoding algorithm is summarized below. Assume binary phase shift key (BPSK) modulation is used.

1) **Initialization:**

$L_n$ and VN-to-CN (VTC) message $Q_{mn}$ are initialized to channel input LLRs. The CN-to-VN (CTV) message $R_{mn}$ is initialized to 0.

2) **Check node processing (CNP):**

For each row $m$, the new check node message $R_{mn}^{\text{new}}$ corresponding to all variable nodes $j$ that connected to this check node are computed using the following equation

$$R_{mn}^{\text{new}} = \prod_{j \in N_m \setminus m} \text{sign}(Q_{mj}^{\text{old}}) \cdot \Psi \left( \sum_{j \in N_m \setminus n} \Psi(Q_{mj}^{\text{old}}) \right),$$  \hfill (3.5)

where “old” and “new” represent the previous and the current iterations, respectively. $N_m \setminus n$ denotes the set of all VNs connected with CN $m$ except VN $n$.

The non-linear function $\Phi(\cdot)$ is defined as

$$\Psi(x) = -\log \left[ \tanh \left( \frac{|x|}{2} \right) \right].$$  \hfill (3.6)

To reduce the computation complexity of (3.5), a simplified min-sum algorithm (MSA) is widely used to approximate the non-linear function $\Phi(\cdot)$ [95, 96]. The scaled min-sum algorithm improves the performance of min-sum algorithm by applying a
scaling factor. The scaling factor $\alpha$ compensates for performance loss in the MSA (typical value is $\alpha = 0.75$). The kernel of the scaled min-sum algorithm with a scaling factor $\alpha$ can be defined as

$$R_{mn}^{new} \approx \alpha \cdot \prod_{j \in N_m \backslash n} \text{sign}(Q_{mj}^{old}) \cdot \min_{j \in N_m \backslash n} |Q_{mj}^{old}|.$$ (3.7)

3) Variable node processing (VNP):

$$L_{n}^{new} = L_{n}^{old} + \sum_{j \in M_n} (R_{mn}^{new} - R_{mn}^{old}),$$ (3.8)

where $M_n$ is the set of CNs that connect to VN $n$. The VTC message is computed as:

$$Q_{mn}^{new} = L_{n}^{new} - R_{mn}^{new}.$$ (3.9)

4) Tentative decoding:

The decoder makes a hard decision to get the decoded bit $x_n$ by checking the APP value $L_n$, that is, if $L_n < 0$ then $x_n = 1$, otherwise $x_n = 0$. The decoding process terminates when a pre-set number of iterations is reached, or the decoded bits satisfy the check equations if early termination is allowed. Otherwise, go back to step 2 and start a new iteration.

### 3.2 Algorithmic Optimization to LDPC Decoding

In this section, we describe parallel LDPC decoding algorithms and optimization techniques to improve throughput.
3.2.1 Loosely Coupled LDPC Decoding Algorithm

It can be noted that the log-SPA or min-sum algorithm could be simplified by re-viewing the equations from (3.9) to (3.8). Instead of saving all the $Q$ values and $R$ values in the device memory, only $R$ values are stored [97]. To compute the new CTV message $R'$, we first recover the $Q$ value based on $R$ and APP value from the previous calculation. Based on this idea, Equation (3.9) can be changed to:

\[ Q_{mn} = L_n - R_{mn}. \]  

(3.10)

Due to the limited size of the on-chip shared memory, $R_{mn}$ values can only fit in the device memory. The loosely coupled LDPC decoding algorithm significantly reduces device memory bandwidth by reducing the amount of device memory used. For example, with an $M \times N$ parity-check matrix $H$ with the row weight of $\omega_r$ (the number of 1’s in a row of $H$), we save up to $M \omega_r$ memory slots. Also, the number of memory accesses to the device memory is reduced by at least $2M \omega_r$ since each $Q_{mn}$ is read and written to at least once per iteration.

3.2.2 Two-Pass Recursion CTV Message Update

Based on (3.5), we need to traverse one row of the $H$ matrix multiple times to update all the CTV messages $R_{mn}$. Each traversal pass requires $(\omega_r - 2)$ compare-select operations to find the minimum value from $(\omega_r - 1)$ data. Moreover, $(\omega_r - 1)$ XOR operations are also needed to compute the sign of $R_{mn}$. In addition, the traversal introduces some branch instructions which cause an unbalanced workload and reduce the throughput.

The traversal process could be optimized by using a forward and backward two-pass recursion scheme [95]. In the forward recursion, some intermediate values are
calculated and stored. During the backward recursion, the CTV message for each node is computed based on the intermediate values from the forward recursion. By roughly calculating, the number of operations can be reduced from $M\omega_r(\omega_r - 2)$ to $M(3\omega_r - 2)$ in one iteration of LDPC decoding. For the 802.11n (1944, 972) code, this two-pass recursion scheme saves us about 50% of the operations. This significantly reduces the decoding latency. The number of memory accesses and the number of required registers are also greatly reduced.

**Algorithm 1** Two pass traverse to compute CTV message $R_{mn}$ values for $i$-th row in $H$ matrix

1: Forward traverse:
2: Init: $s = 0$, $alpha[0] = C$; \{C is an initial value\}
3: for all $BN_n$ (columns in $H$) do \{comment: left to right\}
4: \hspace{1em} if $H_{in} == 1$ then
5: \hspace{2em} Read $R_{in}$ from the memory;
6: \hspace{2em} $Q[s] = APP - R_{in}$;
7: \hspace{2em} $alpha[s + 1] = \text{min\_sum\_func}(alpha[s], Q[s])$;
8: \hspace{2em} $addr[s] = n$;
9: \hspace{2em} $s + +$;
10: end if
11: end for
12:
13: Backward traverse:
14: Init: $t = s - 1$, $beta[t] = C$;
15: for all $i$ from $(s - 1)$ to 0 do \{comment: right to left\}
16: \hspace{1em} $beta[t - 1] = \text{min\_sum\_func}(beta[t], Q[t])$;
17: \hspace{1em} $R'_{in} = S \cdot \text{min\_sum\_func}(beta[t], alpha[t])$;
18: \hspace{1em} $\Delta_{in} = R'_{in} - R_{in}$;
19: \hspace{1em} write $R'_{in}$ and $\Delta_{in}$ to memory using index $addr[t]$;
20: \hspace{1em} $t - -$;
21: end for
22: \{So far, all the $R_{in}$ and $\Delta_{in}$ in $i$-th row have been updated.\}

3.2.3 Two-min Min-Sum LDPC Decoding Algorithm

The message values are represented by 32bit floating-point data type. Similar to [41], CNP and VNP are mapped onto two separate parallel kernel functions. Matrix $H$
Algorithm 2 TMA for check node processing.

1: \(\text{sign}_\text{prod} = 1;\) /* sign product; 1: positive, -1: negative */
2: \(\text{sign}_\text{bm} = 0;\) /* bitmap of \(Q\) sign; 0: positive, 1: negative */
3: for \(i = 0\) to \(\omega_r - 1\) do
4: \(\text{Load } L_n\) and \(R\) from device memory;
5: \(Q = L_n - R;\)
6: \(sq = Q < 0;\) /* sign of \(Q\); 0: positive, 1: negative */
7: \(\text{sign}_\text{prod} * = (1 - sq * 2);\)
8: \(\text{sign}_\text{bm} | = sq << i;\)
9: if \(|Q| < \text{min}_1\) then
10: update \(\text{min}_1, idx\) and \(\text{min}_2;\)
11: else if \(|Q| < \text{min}_2\) then
12: update \(\text{min}_2;\)
13: end if
14: end for
15: for \(i = 0\) to \(\omega_r - 1\) do
16: \(sq = 1 - 2 * ((\text{sign}_\text{bm} >> i)\&0x01);\)
17: \(R_{\text{new}} = 0.75 \cdot \text{sign}_\text{prod} \cdot sq \cdot (i \neq idx \? \text{min}_1 : \text{min}_2);\)
18: \(dR = R_{\text{new}} - R;\)
19: Store \(dR\) and \(R_{\text{new}}\) into device memory;
20: end for

is represented using compact formats, which are stored in GPU’s constant memory to allow fast data broadcasting. To fully utilize the stream multi-processors of GPU, we use multi-codeword decoding algorithm. \(N_{\text{MCW}}\) macro-codewords (MCWs) are defined, each of which contains \(N_{\text{CW}}\) codewords, so the total number of codewords decoded in parallel is \(N_{\text{codeword}} = N_{\text{CW}} \times N_{\text{MCW}}\) (typically \(N_{\text{CW}} \in [1, 4]\), and \(N_{\text{MCW}} \in [1, 100]\)). To launch the CNP kernel, the grid dimension is set to \((m_b, N_{\text{MCW}}, 1)\) and the thread block dimension is set to \((Z, N_{\text{CW}}, 1)\). For the VNP kernel, the grid dimension and the thread block dimension are \((n_b, N_{\text{MCW}}, 1)\) and \((Z, N_{\text{CW}}, 1)\), respectively. By adjusting \(N_{\text{MCW}}\) and \(N_{\text{CW}}\), we can easily change the scalable workload for each kernel. For data storage, since we can use \(R_{mn}\) and \(L_n\) to recover \(Q_{mn}\) according to (3), we only store \(R_{mn}\) and \(L_n\) in the device memory and compute \(Q_{mn}\) on the fly in the beginning of CNP. Please refer to [41] for the above implementation details.
To support both the SPA and the MSA algorithms, a forward-backward algorithm (FBA) is used to implement the CNP kernel in [41]. Here, we employ the two-min algorithm (TMA) to further reduce the CNP complexity [92, 98]. It is worth mentioning that FBA and TMA provide the same error-correcting performance when implementing the MSA. According to (1), we can use four terms to recover all $R_{mn}$ values for a check node: the minimum of $|Q_{mn}|$ (denoted as $min_1$), the second minimum of $|Q_{mn}|$ (denoted as $min_2$), the index of $min_1$ (denoted as $idx$), and product of all signs of $Q_{mn}$ (denoted as $sign_{prod}$). $R_{mn}$ can be determined by $R_{mn} = sign_{prod} \cdot sign(Q_{mn}) \cdot ((n \neq idx)?min1 : min2)$. The TMA is described in Algorithm 2. Since we do not store $Q_{mn}$ values, the sign array of $Q_{mn}$ needs to be kept for the second recursion. To save storage space, we use a char type $sign_{bm}$ to store the bitmap of the sign array. Bitwise shift and logic operations are needed to update this bitmap or extract a sign out of the bitmap. The $sign_{prod}$ can be updated by using either bitwise logic operations or floating-point (FP) multiplication. However, since the instruction throughput for FP multiplication is higher than bitwise logic operations (192 versus 160 operations per clock cycle per multiprocessor) [75], FP multiplication is chosen to update $sign_{prod}$ value efficiently.

Table 3.1 compares the complexity of a naive implementation of (1), the FBA and the TMA. Since compare-select (CS) is the core operation in the Min-Sum algorithm, we use the number of CS operations to indicate algorithmic complexity. Table 3.1 indicates that the TMA has lower complexity compared to the other two algorithms. It is worth mentioning that Algorithm 2 is targeted at decoding more challenging irregular LDPC codes ($\omega_c$ is not constant). If we decode regular LDPC codes, the loops in Algorithm 2 can be fully unrolled to avoid branching operations to further increase the throughput.
Table 3.1: Complexity comparison for CNP using a “native” implementation, the FBA and the TMA.

<table>
<thead>
<tr>
<th></th>
<th>“Naive”</th>
<th>FBA</th>
<th>TMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS operations</td>
<td>( M\omega_r(\omega_r - 1) )</td>
<td>( M(3\omega_r - 2) )</td>
<td>( M(\omega_r - 1) )</td>
</tr>
<tr>
<td>Memory accesses</td>
<td>( M\omega_r^2 )</td>
<td>( M(3\omega_r - 2) )</td>
<td>( 2M\omega_r )</td>
</tr>
</tbody>
</table>

### 3.3 CUDA Architecture

Computer Unified Device Architecture (CUDA) [75] is widely used to program massively parallel computing applications. The NVIDIA Fermi GPU architecture consists of multiple stream multiprocessors (SM). Each SM consists of 32 pipelined cores and two instruction dispatch units. During execution, each dispatch unit can issue a 32 wide single instruction multiple data (SIMD) instruction which is executed on a group of 16 cores. Although CUDA provides the possibility to unleash GPU’s computational power, several restrictions prevent programmers from achieving peak performance. The programmer should pay attention to the following aspects to achieve near-peak performance.

In the CUDA model, a frequently executed task can be mapped into a kernel and executed in parallel on many threads in the GPU. CUDA divides threads within a thread block into blocks of 32 threads, which are executed as a group using the same common instruction (a *warp* instruction). As instructions are issued in-order, a stall can occur when an instruction fetches data from device memory or when there is data dependency. Stalls can be minimized by coalescing memory access, using fast on-die memory resources and through hardware support for fast thread switching.

A CUDA device has a large amount (>1GB) of off-chip device memory with high latency (400~800 cycles) [75]. To improve efficiency, device memory accesses can be coalesced. For CUDA devices, if all threads access 1, 2, 4, 8, or 16 byte words and threads within half of a *warp* access sequential words in device memory, the memory requests of half of a *warp* can be coalesced into a single memory request.
Coalescing memory access improves performance by reducing the number of device memory requests.

Fast on-chip resources such as registers, shared memory, and constant memory can reduce memory access time. Access to shared memory usually takes one load or store operation. However, random accesses to shared memory with bank conflicts should be avoided since they will be serialized and cause performance degradation. On the other hand, memory access to constant memory is cached although constant memory resides in device memory. Therefore, it takes one cached read if all threads access the same location. However, other constant memory access patterns will be serialized and need device memory access, therefore, should be avoided.

Furthermore, to hide the processor stalls and achieve peak performance, one should map sufficient concurrent threads onto a SM. However, both shared memory and registers are shared among concurrent threads on a SM. The limited amounts of shared memory and registers require the programmers to effectively partition the workloads such that at least a sufficient number of blocks can be mapped onto the same processor to hide stalls.

3.4 Mapping LDPC Decoding Algorithm onto GPU

In this work, we map both the MSA and the log-SPA onto GPU architecture. In this section, we present modified decoding algorithms to reduce the decoding latency and the bandwidth requirement of device memory. The implementations of the LDPC decoding kernels are also described.
3.4.1 Implementation of the LDPC Decoder Kernel on GPU

According to Equations (3.5), (3.9), (3.8) and (3.10), the decoding process can be split into two stages: the horizontal processing stage and the APP update stage. So we can create one computational kernel for each stage. The relationship between the host (CPU) code and the device (GPU) kernel is shown in Fig 3.3.

(1) CUDA Kernel 1: Horizontal Processing

During the horizontal processing stage, since all the CTV messages are calculated independently, we could use many parallel threads to process these CTV messages. For an $M \times N \mathbf{H}$ matrix, $M$ threads are spawned, and each thread processes a row. Since all non-zero entries in a sub-matrix of $\mathbf{H}$ have the same shift value (one square box in Fig 3.2), threads processing the same layer have almost exactly the same operations when calculating the CTV messages. In addition, there is no data dependency among different layers. Therefore, each layer is processed by a thread block. $M_{sub}$ thread blocks are used and each consists of $Z$ threads. The CUDA kernel 1 is described in Algorithm 3.
the 802.11n (1944, 972) LDPC code as an example, 12 thread blocks are generated, and each contains 81 threads, so there are a total of 972 threads used to calculate the CTV messages.

Algorithm 3 CUDA Kernel 1: Horizontal processing

1: \(iLayer = blockIdx.x;\) /*the index of a layer in \(H^*\)*/
2: \(iSubRow = threadIdx.x;\) /*the row index in the layer*/
3: Calculate new CTV message using two-pass recursion;
4: Write CTV messages \((R_{mn}, \Delta_{mn})\) back into device memory;

(2) CUDA Kernel 2: APP value update During the APP update stage, there are \(N\) APP values to be updated. Similarly, the APP value update is independent among different variable nodes. So, \(N_{sub}\) thread blocks are used to update the APP values, with \(Z\) threads in each thread block. In the APP update stage, there are 1944 threads which are grouped into 24 thread blocks working concurrently for the 802.11n (1944, 972) LDPC code. The algorithm of CUDA kernel 2 is described in Algorithm 4.

Algorithm 4 CUDA Kernel 2: Update APP value

1: \(iBlkCol = blockIdx.x;\) /*the column index of a sub-matrix in \(H^*\)*/
2: \(iSubCol = threadIdx.x;\) /*the column index within the sub-matrix*/
3: Calculate the device memory address of APP value;
4: Read the old APP value;
5: for all All the sub-matrices in \(iBlkCol\) column of \(H\) do
6: Read the corresponding \(\Delta_{mn}\) value;
7: Update the APP value using (3.8);
8: end for
9: Write the updated APP value into the device memory;
10: Make a hard decision for the current bit.

3.4.2 Multi-codeword Parallel Decoding

Since the number of threads and thread blocks are limited by the dimensions of the \(H\) matrix, it is hard to keep all the cores fully occupied by decoding a single codeword.
Multi-codeword decoding is needed to further increase the parallelism of the workload. A two-level multi-codeword scheme is designed. $N_{CW}$ codewords are first packed into one macro-codeword (MCW). Each MCW is decoded by a thread block and $N_{MCW}$ MCWs are decoded by a group of thread blocks. The multi-codeword parallel decoding algorithm is described in Fig. 3.4.

Since multiple codewords in one MCW are decoded by the threads within the same thread block, all the threads follow the same execution path during the decoding process. So the workload is well balanced for the codewords in one MCW which is helpful to increase the throughput. Another benefit is that the latency of read-after-write dependencies and memory bank conflicts can be completely hidden by a sufficient number of active threads. To implement this multi-codeword parallel decoding scheme, $N_{MCW} \times N_{CW}$ codewords are written into the device memory in a specified order before kernel launch.
3.4.3 Implementation of Early Termination Scheme

The early termination (ET) algorithm is used to avoid unnecessary computations when the decoder already converges to the correct codeword. For software implementation and simulation, ET algorithm can significantly boost the throughput, especially for the high SNR (signal-to-noise ratio) scenario.

The parity check equations $H \cdot x^T = 0$ can be used to verify the correctness of the decoded codeword. Since the parity check equations are inherently parallel, a new CUDA kernel with many threads is launched to perform the ET check. We create $M$ threads, and each thread calculates one parity check equation independently. Since the decoded codeword $x$, compact $H$ matrix and parity check results are used by all the threads, on-chip shared memory is used to enable the high speed memory access. After the concurrent threads finish computing the parity check equations, we reuse these threads to perform a reduction operation on all the parity check results to generate the final ET check result, which indicates the correctness of the concurrent codeword.

This parallel ET algorithm is straightforward for a single codeword. However, for multi-codeword parallel decoding, the ET can help increase the throughput only if all the codewords meet the ET condition, which has a very low chance to happen. To overcome this problem, we propose a tag-based ET algorithm for multi-codeword parallel decoding. All the $N_{MCW} \times N_{CW}$ codewords are checked by multiple thread blocks. We assign one tag per codeword and mark the tag once the corresponding parity check equation is satisfied. Afterwards, the decoding process of this particular codeword is stopped in the following iterations. Once the tags for all the codewords are marked, the iterative decoding process is terminated. The experimental results in Section 3.7 show that the proposed parallel ET algorithm significantly increases the throughput.
3.5 Memory Access Optimization

3.5.1 Data Structure and Storage Optimization

The quasi-cyclic characteristic of the QC-LDPC code allows us to efficiently store the sparse H matrix. Two compact representations of H (H_{kernel1} and H_{kernel2}) are designed. Here, we regard the cyclic H matrix in Fig. 3.2 as a 12 × 24 matrix \( \bar{H} \) whose entry is the shift value for each sub-matrix of H. After horizontally compressing \( \bar{H} \) to the left, we get H_{kernel1}. Similarly, vertically compressing \( \bar{H} \) to the top gives us H_{kernel2}. The structure of H_{kernel1} and H_{kernel2} are shown in Fig 3.5. The H matrix is the same as in Fig. 3.2. After the horizontal compression and vertical compression, we get H_{kernel1} and H_{kernel2}, respectively. Each entry of the compressed H matrix is a customized structure shown in this figure. Each entry of the compressed H matrix contains 4 8-bit data indicating the row and column index of the element in the original H matrix, the shift value and a valid flag which shows whether the current entry is empty or not. Therefore, each entry could be expressed by a 32-bit data value which occupies one memory slot.

There are two advantages of the proposed compact representation of H. First,
since the compressed format reduces the memory usage, the time used to read the \( \mathbf{H} \) matrix from device memory (with long latency) is reduced. Second, the branch instructions cause throughput degradation. The compressed matrix shows the position of all the non-empty entries in \( \mathbf{H} \). Therefore, during the two-pass recursion there is no need to check whether one entry of \( \mathbf{H} \) is empty which avoids the use of branch instructions. Taking the 802.11n (1944, 972) \( \mathbf{H} \) matrix as an example, 40% of memory access and branch instructions are reduced by using the compressed \( \mathbf{H}_{\text{kernel1}} \) and \( \mathbf{H}_{\text{kernel2}} \).

The constant memory on the GPU device could be utilized to optimize the throughput. Based on GPU’s architecture, reading from the constant memory is as fast as reading from a register as long as all the threads within a half-warp read the same address. Since all the \( Z \) threads in one thread block access the same entry of the \( \mathbf{H} \) matrix simultaneously, we can store the \( \mathbf{H} \) matrix in the constant memory and take advantage of the broadcasting mode of the constant memory. Simulation shows that constant memory increases the throughput by about 8%.

### 3.5.2 Fully Coalesced Memory Access For CNP and VNP

Accesses to global memory incur long latency of several hundred clock cycles, therefore, memory access optimization is critical for throughput performance. In our implementation, to minimize the data transfer on the PCIe bus, we only transfer the initial LLR values from host to device memory and the final hard decision values from device to host memory. All the other variables such as \( R_{mn} \) and \( dR_{mn} \) (storing \( (R_{mn}^{\text{new}} - R_{mn}^{\text{old}}) \) values needed by (2) in VNP) are only accessed by the kernel functions without being transferred between host and device. To speed up data transfers between host and device, the host memories are allocated as page-locked (or pinned) memories. The page-locked memory enables a direct memory access (DMA) on the
Figure 3.6: Optimized coalesced memory access. A shifted identity matrix from WiMAX code \((Z = 96)\) with shift value 43 is shown. Combining CNP from (a) and VNP from (c), we achieve fully coalesced memory accesses.

GPU to request transfers to and from the host memory without the involvement of the CPU, providing higher memory bandwidth compared to the pageable host memory [75]. Profiling results indicate that throughput improves about 15% by using page-locked memory.

GPUs are able to coalesce global memory requests from threads within a warp into one single memory transaction, if all threads access 128-byte aligned memory segment [75]. Falcão proposed to coalesce memory reading via translation arrays, but writing to memory is still uncoalesced [85]. In [91], reading/writing memory coalescing is used in VTC messages, but CTV message accesses are still not coalesced. In this
section, we describe a fully coalesced memory access scheme which coalesces memory accesses for both reading and writing in both CNP and VNP kernels.

In our implementation, accesses to $R_{mn}$ (and $dR_{mn}$) in CNP kernels and memory accesses to APP values $L_n$ are naturally coalesced, as is shown in Fig. 3.6-(a). However, due to the random shift values, memory accesses to $L_n$ in CNP and memory accesses to $R_{mn}$ (and $dR_{mn}$) in VNP are misaligned. For instance, in Fig. 3.6-(b), three warps access misaligned $R_{mn}$ data, and warp 2 even accesses nonconsecutive data, so multiple memory transactions are generated per data request. As is shown in Fig. 3.6-(c), we use fast shared memory as cache to help coalesce memory accesses (size of shared memory: $\omega_r \cdot N_{CW} \cdot Z \cdot \text{sizeof(float)}$). We first load data into shared memory in a coalesced way using parallel threads. After a barrier synchronization is performed, the kernels can access data from the shared memory with very low latency. Finally, the kernels write cached data back to device memory in a coalesced way. Profiling results from NVIDIA development tools indicate the proposed method effectively eliminates uncoalesced memory accesses. Since all the device memory accesses become coalesced which leads to a reduction in the number of global memory transactions, the decoding throughput is increased.

### 3.5.3 Data and Thread Alignment for Irregular Block Size

Data alignment is required for coalesced memory access, so it has a big impact on the memory access performance. For the WiMAX $(2304, 1152)$ code, the shifted identity matrix has a size of $Z = 96$, which is a multiple of warp size (32). Therefore, the data alignment can be easily achieved. However, since $Z = 81$ is defined in the WiFi $(1944, 972)$ code, with straightforward data storing order and thread block assignment, few data are aligned to 128-byte addresses. Therefore, we optimize LDPC decoding for irregular block sizes (such as WiFi codes) by packing dummy threads, which means
that the thread block dimension becomes \(((Z + 31)/32 \times 32, N_{CW}, 1)\). Similarly, for data storage, dummy spaces are reserved to make sure all memory accesses are 128-byte aligned. Although we waste some thread resources and a few memory slots, the aligned thread and data enable efficient memory accesses, and therefore, improves the throughput by approximately 20%.

### 3.6 Reducing Decoding Latency

All the aforementioned optimization strategies applied to the decoding kernels will not only improve the throughput, but also help reduce the decoding latency. In this section, we present optimization techniques to reduce the LDPC decoding latency.

#### 3.6.1 Asynchronous Memory Transfer

The current generation NVIDIA GPU contains two memory copy engines and one compute engine. Therefore, we are able to hide most of the time required to transfer data between the host and device by overlapping kernel execution with asynchronous memory copy (Algorithm 5). Fig. 3.7 shows how the memory transfers overlap with CNP/VNP kernels. According to our experiments, this technique improves performance by 17% for a typical kernel configuration \((N_{CW} = 2, N_{MCW} = 40)\).

**Algorithm 5** Asynchronous data transfer.

1: \textbf{for} \(i = 0 \) to \(N_{\text{codeword}} - 1\) \textbf{do}
2: \hspace{1em} \text{memcpyAsync(codeword } i\text{, host} \rightarrow \text{device});
3: \textbf{for} \(j = 0 \) to \(N_{\text{iter}} - 1\) \textbf{do}
4: \hspace{2em} CNP\_kernel(codeword } i\text{, iter } j\text{)};
5: \hspace{2em} VNP\_kernel(codeword } i\text{, iter } j\text{)};
6: \hspace{1em} \textbf{end for}
7: \hspace{1em} \text{memcpyAsync(codeword } i\text{, device} \rightarrow \text{host});
8: \textbf{end for}
3.6.2 Multi-stream Scheduling for Concurrent Kernels

Computation kernels and memory operations in multiple streams can execute concurrently if there is no dependency between streams. Since the Kepler GK110 architecture, NVIDIA GPUs support up to 32 concurrent streams. In addition, a new feature called Hyper-Q is provided to remove false dependencies between multiple streams to fully allow concurrent kernel overlapping [75]. We take advantage of these new features and further reduce the LDPC decoding latency.

**Algorithm 6** Breadth-first multi-stream scheduling.

```plaintext
1: for $i = 0$ to $N_{Stream} - 1$ do
2:     memcpyAsync(streams[i], host→device);
3: end for
4: for $j = 0$ to $N_{iter} - 1$ do
5:     for $i = 0$ to $N_{Stream} - 1$ do
6:         CNP_kernel(streams[i], iter $j$);
7:         VNP_kernel(streams[i], iter $j$);
8:     end for
9: end for
10: for $i = 0$ to $N_{Stream} - 1$ do
11:     memcpyAsync(streams[i], device→host);
12: end for
13: for $i = 0$ to $N_{Stream} - 1$ do
14:     streamSynchronize(streams[i]);
15: end for
```

In the literature, high throughput is usually achieved via multi-codeword decoding in order to increase the occupancy ratio of parallel cores [41, 90–93]. One drawback
Algorithm 7 Depth-first multi-stream scheduling.

1: for $i = 0$ to $N_{Stream} - 1$ do
2:  memcpyAsync(streams[$i$], host→device);
3:  for $j = 0$ to $N_{iter} - 1$ do
4:      CNP_kernel(streams[$i$], iter $j$);
5:      VNP_kernel(streams[$i$], iter $j$);
6:   end for
7:  memcpyAsync(streams[$i$], device→host);
8: end for
9: for $i = 0$ to $N_{Stream} - 1$ do
10:  streamSynchronize(streams[$i$]);
11: end for

Figure 3.8: Multi-stream LDPC decoding.

of multi-codeword decoding is long latency. To overcome this drawback, we partition codewords into independent workloads and distribute them across multiple streams, so that each stream only decodes a small number of codewords. Multi-stream decoding not only keeps high occupancy thanks to concurrent kernel execution, but also reduces decoding latency. Breadth-first (Algorithm 6) and depth-first (Algorithm 7) GPU command issuing orders are two typical ways to schedule multiple streams. Our experimental results indicate that both issuing orders result in similar decoding throughput, but the depth-first scheduling listed in Algorithm 7 leads to much lower latency. Therefore, we choose the depth-first scheduling algorithm.

Fig. 3.8 demonstrates a timeline for the multi-stream LDPC decoding. The degree of kernel overlapping depends on the kernel configurations (such as parameters $N_{CW}$ and $N_{MCW}$). In a practical SDR system, we can use multiple CPU threads with each managing one GPU stream, so that all the GPU streams can run independently. The
Table 3.2: Achievable throughput. \(N_S = 16, N_{CW} = 2, N_{MCW} = 40\).

<table>
<thead>
<tr>
<th>Code</th>
<th># of iterations</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WiMAX (2304, 1152)</td>
<td>5</td>
<td>621.38</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>316.07</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>204.88</td>
</tr>
<tr>
<td>WiFi (1944, 972)</td>
<td>5</td>
<td>490.01</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>236.70</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>154.30</td>
</tr>
</tbody>
</table>

decoding latency is determined by the latency of each stream.

### 3.7 Experimental Results

The experimental platform consists of an Intel i7-3930K six-core 3.2GHz CPU and four NVIDIA GTX TITAN graphics cards. The GTX TITAN has a Kepler GPU containing 2688 CUDA cores running at 837MHz, and 6GB GDDR5 memory. Graphics cards are connected to the system via PCIe x16 interfaces. CUDA toolkit v5.5 Linux 64bit version is used. NSight v3.5 is used for profiling. In the experiments, two typical codes from the 802.16e WiMAX and 802.11n WiFi standards are employed. The processing time is measured using the CPU timer, so the kernel processing time plus the overhead including CUDA runtime management and memory copy time are counted.

Table 3.2 shows the achievable throughput when using one GPU. \(N_S\) denotes the number of concurrent streams. 16 concurrent streams are used, and experiments show that using 32 streams provides similar throughput performance. We achieve the peak throughput of 316.07 Mbps (@10 iters) when decoding the WiMAX code. We also notice that there is still a gap in throughput results between WiMAX codes and WiFi codes, although specific optimizations have been performed for WiFi LDPC codes as discussed in Section 3.5.3. The reason is two fold. Firstly, by aligning the size of a thread block to a multiple of the warp size, 15.6% threads (15 out of 96) are idle;
Table 3.3: Lowest achievable latency for different throughput goals ($N_{iter} = 10$). WiMAX (2304, 1152) code. (T: throughput)

<table>
<thead>
<tr>
<th>$T_{goal}$ (Mbps)</th>
<th>$N_S$</th>
<th>$N_{CW}$</th>
<th>$N_{MCW}$</th>
<th>Latency (ms)</th>
<th>$T$ (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.207</td>
<td>62.50</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>0.236</td>
<td>110.25</td>
</tr>
<tr>
<td>150</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>0.273</td>
<td>155.43</td>
</tr>
<tr>
<td>200</td>
<td>16</td>
<td>2</td>
<td>7</td>
<td>0.335</td>
<td>201.39</td>
</tr>
<tr>
<td>250</td>
<td>16</td>
<td>2</td>
<td>10</td>
<td>0.426</td>
<td>253.36</td>
</tr>
<tr>
<td>300</td>
<td>32</td>
<td>2</td>
<td>25</td>
<td>1.266</td>
<td>304.16</td>
</tr>
</tbody>
</table>

while for the WiMAX codes, all threads perform useful computations. Secondly, the $H$ matrix of the WiFi LDPC code has 13.16% more edges than the WiMAX codes, which requires more computations.

Table 3.3 shows the minimum workload per stream (so as to get the lowest latency) needed to achieve different throughput goals. The workload can be configured by changing parameters ($N_S, N_{CW}, N_{MCW}$) to meet different latency/throughput requirements. We sweep through all combinations of ($N_S, N_{CW}, N_{MCW}$) for $N_S \in [1, 32]$, $N_{CW} \in [1, 5]$ and $N_{MCW} \in [1, 150]$. We searched the whole design space and found the configurations that meet the $T_{goal}$ Mbps performance with the lowest latency, which are reported in Table 3.3. For example, to achieve throughput higher than 50 Mbps, one stream ($N_S = 1$) with $N_{CW} = 2$ and $N_{MCW} = 3$ is configured. With this configuration, we can actually achieve 62.5 Mbps throughput while the latency is only 0.207 ms. As is shown in Table 3.4, this work achieves much lower decoding latency than other GPU-based LDPC decoders.

In this section, we focus on improving the raw performance of the computation kernels. Please note that we can still apply the tag-based parallel early termination algorithm and achieve the corresponding speedup as we reported in [41].

The above experiments are performed on a single GPU. We have successfully further pushed the throughput limit by using all four GPUs in our test platform. In order to distribute the decoding workload evenly across four GPUs, we create four
Table 3.4: Decoding latency comparison with other works. ($N_C$: number of code-words; $T$: throughput; $L$: latency)

<table>
<thead>
<tr>
<th>LDPC code</th>
<th>GPU</th>
<th>$N_{iter}$</th>
<th>$N_C$</th>
<th>$T$ (Mbps)</th>
<th>$L$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[85]</td>
<td>8800GTX</td>
<td>10</td>
<td>16</td>
<td>14.6</td>
<td>1.12</td>
</tr>
<tr>
<td>[89]</td>
<td>GTX280</td>
<td>10</td>
<td>1</td>
<td>1.28</td>
<td>1.8</td>
</tr>
<tr>
<td>[41, 43]</td>
<td>GTX470</td>
<td>10</td>
<td>300</td>
<td>52.15</td>
<td>13.25</td>
</tr>
<tr>
<td>[90]</td>
<td>9800GTX</td>
<td>5</td>
<td>256</td>
<td>160</td>
<td>3.69</td>
</tr>
<tr>
<td>[91]</td>
<td>GTX480</td>
<td>10</td>
<td>N/A</td>
<td>24</td>
<td>N/A</td>
</tr>
<tr>
<td>[92]</td>
<td>HD5870</td>
<td>10</td>
<td>500</td>
<td>209</td>
<td>19.13</td>
</tr>
<tr>
<td>[93]</td>
<td>M2050</td>
<td>17.42</td>
<td>16</td>
<td>55</td>
<td>18.85</td>
</tr>
</tbody>
</table>

This work

<table>
<thead>
<tr>
<th>LDPC code</th>
<th>GPU</th>
<th>$N_{iter}$</th>
<th>$N_C$</th>
<th>$T$ (Mbps)</th>
<th>$L$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2304, 1152)</td>
<td>GTX TITAN</td>
<td>10</td>
<td>16</td>
<td>62.50</td>
<td>0.207</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>110.25</td>
<td>0.236</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>201.39</td>
<td>0.335</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td>304.16</td>
<td>1.266</td>
</tr>
</tbody>
</table>

Figure 3.9: Multi-GPU LDPC decoding managed by multiple CPU threads.

independent CPU threads using OpenMP APIs, with each CPU thread managing a GPU, as shown in Fig. 3.9. As a result, an aggregate peak throughput of 1.25 Gbps (at 10 iterations) is achieved for decoding the WiMAX (2304, 1152) LDPC code. The workload configuration for each CPU thread is $N_S = 16$, $N_{CW} = 2$, and $N_{MCW} = 40$.

3.8 VLSI Implementation of LDPC Decoding

In this section, we briefly introduce a double-layer LDPC decoding architecture and the corresponding ASIC implementation.
Figure 3.10: Memory access conflict problem for layered LDPC decoding.

3.8.1 Algorithm and Architecture

To support layer-level parallelism, we propose a multi-layer (M-layer) parallel decoding algorithm, where the maximum row parallelism is increased to $KZ$. When using the conventional layered algorithm to process multiple layers at the same time, data conflicts may occur when updating the LLRs because there can be more than one check node connected to a variable node. Fig. 3.10 shows an example of the data conflicts when updating LLRs for two consecutive layers, where check node (or row) $m_0$ and check node $m_1$ are both connected to variable node (or column) $n$. To resolve the data conflicts, we use the following LLR update rule for a $K$-layer parallel decoding algorithm. For a variable node $n$, let $m_k$ represents the $k$-th check node that is connected to variable node $n$. Then the LLR value for variable node $n$ is updated as:

$$L_n^{new} = L_n^{old} + \sum_{k=0}^{K-1} (P_{m_kn}^{new} - R_{m_kn}^{old}),$$  \hspace{1cm} (3.11)

Compared to the original LLR update rule, the new LLR update rule combines all the check node messages and adds them to the old LLR value. We can define a macro-layer as a group of $K$ layers of the parity check matrix. The multi-layer parallel decoding algorithm is summarized as follows. For each layer $k$ in each macro-layer $l$, do the following:

$$Q_{m_kn} = L_n - R_{m_kn},$$ \hspace{1cm} (3.12)
\[ R_{m_k n}^{new} = \prod_{j \in N_{m_k} \setminus m} \text{sign}(Q_{m_k j}^{old}) \cdot \Psi \left( \sum_{j \in N_{m_k} \setminus m} \Psi(Q_{m_k j}^{old}) \right), \] (3.13)

\[ L_n^{new} = L_n^{old} + \sum_{k=0}^{K-1} (R_{m_k n}^{new} - R_{m_k n}^{old}), \] (3.14)

In the above calculation, the LLR values \( L_n \) are updated macro-layer after macro-layer. Within each macro-layer, all the check rows can be processed in parallel, which therefore leads to a \( K \) times larger parallelism than the conventional layered algorithm. For example, we can use \( KZ \) number of check node processors to process \( KZ \) rows in parallel.

### 3.8.2 VLSI Implementation

A flexible double-layer parallel decoder which fully supports IEEE 802.11n LDPC codes was designed in Verilog HDL. The fixedpoint design parameters are as follows.
Table 3.5: Throughput performance of the proposed LDPC decoder.

<table>
<thead>
<tr>
<th>Block length</th>
<th>Rate 1/2</th>
<th>Rate 2/3</th>
<th>Rate 3/4</th>
<th>Rate 5/6</th>
</tr>
</thead>
<tbody>
<tr>
<td>648 bits</td>
<td>380 Mbps</td>
<td>520 Mbps</td>
<td>760 Mbps</td>
<td>1.0 Gbps</td>
</tr>
<tr>
<td>1296 bits</td>
<td>750 Mbps</td>
<td>1.1 Gbps</td>
<td>1.3 Gbps</td>
<td>2.0 Gbps</td>
</tr>
<tr>
<td>1944 bits</td>
<td>1.1 Gbps</td>
<td>1.7 Gbps</td>
<td>2.2 Gbps</td>
<td>3.0 Gbps</td>
</tr>
</tbody>
</table>

The channel input LLR is represented with 6-bit signed numbers with 2 fractional bits. The word lengths of the extrinsic R values and the APP LLR values are 6 bits and 7 bits, respectively. According to the computer simulation, this fixed-point implementation introduces a performance loss of 0.05 dB compared to the floating-point implementation at $10^{-4}$ FER for rate $1/2$ code. Figure 3.12 shows the architecture of the double-layer LDPC decoder.

We have synthesized the decoder for a TSMC 45nm CMOS technology. The maximum clock frequency is 815 MHz and the area is 0.81 mm$^2$ based on the Synopsys Design Compiler synthesis result. Table 3.5 summarizes the throughput performance of this double-layer parallel decoder for the decoding of IEEE 802.11n LDPC codes at 15 iterations, where the throughput is calculated by counting the number of the clock cycles for each iteration. The throughput increases with the block size and the code rate.

### 3.9 Design Space Exploration

#### Algorithmic Complexity

Assume the parity-check matrix $H$ is an $M \times N$ matrix. Each shifted-identity matrix has the size of $Z \times Z$. The matrix $H$ contains $m_b \times n_b$ submatrices with size $Z \times Z$. Denote the row weight and column weight of LDPC codes as $\omega_r$ and $\omega_c$, respectively. Assume the decoding throughput is $T$.

The major computation complexity lies in the CNP (check-node processing) units,
whose complexity can be estimated as $M(\omega_r - 1)$ min-finder operation plus $2M\omega_r$ adders. To find the 1st minimum value and the 2nd minimum value, the min-finder can be implemented using two adders and two multiplexers. Therefore, the total computation complexity of the CNP units is $M(6\omega_r - 4)$ operations per iteration. We can calculate the operations per decoded bit ($N_{OPPDB}$) as follows:

$$
N_{OPPDB} = \frac{M(6\omega_r - 4)N_{iter}}{M} = (6\omega_r - 4)N_{it}.
$$

(3.15)

Therefore, the complexity measured in GOPS can be calculated as:

$$
Complexity = N_{OPPDB} \cdot T = (6\omega_r - 4)N_{iter}T.
$$

(3.16)

For example, for a (1152, 2304) WiMAX code, $\omega_r = 7$, and the typical number of iterations is 15. To achieve 1Gbps data rate, 570 GOPS compute capability is required.

**Throughput Performance Model**

We firstly define some notations used in the performance model in Table 3.6.

The throughput for the LDPC decoding can be expressed as:

$$
T = \frac{N \cdot N_{\text{codeword}}}{N_{iter} \cdot (t_{CNP} + t_{VNP} + t_{ET})}.
$$

(3.17)

If we consider the memory data transfer time between the host and the accelerator,
Table 3.6: Notations for parallel LDPC decoding.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Codeword size</td>
</tr>
<tr>
<td>$f_{\text{clk}}$</td>
<td>Clock frequency</td>
</tr>
<tr>
<td>$N_{\text{iter}}$</td>
<td>Number of iterations</td>
</tr>
<tr>
<td>$N_{\text{codeword}}$</td>
<td>Number of codeword decoded in parallel</td>
</tr>
<tr>
<td>$t_{\text{CNP}}$</td>
<td>Time for check node processing</td>
</tr>
<tr>
<td>$t_{\text{VNP}}$</td>
<td>Time for variable node processing</td>
</tr>
<tr>
<td>$t_{\text{ET}}$</td>
<td>Time for early termination calculation</td>
</tr>
<tr>
<td>$t_{\text{H2D}}$</td>
<td>Host to device data transfer time</td>
</tr>
<tr>
<td>$t_{\text{D2H}}$</td>
<td>Device to host data transfer time</td>
</tr>
</tbody>
</table>

Table 3.7: Memory accesses for each iteration in LDPC decoding.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Memory read</th>
<th>Memory write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNP</td>
<td>$2M_\omega r$</td>
<td>$2M_\omega r$</td>
<td>$4M_\omega r$</td>
</tr>
<tr>
<td>VNP</td>
<td>$N(\omega_c + 1)$</td>
<td>$N$</td>
<td>$N(\omega_c + 2)$</td>
</tr>
<tr>
<td>ET</td>
<td>$N$</td>
<td>$0$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

The above equation can be modified as:

$$T = \frac{N \cdot N_{\text{codeword}}}{N_{\text{iter}} \cdot (t_{\text{CNP}} + t_{\text{VNP}} + t_{\text{ET}}) + t_{\text{H2D}} + t_{\text{D2H}}}.$$ (3.18)

**Memory Access Pattern**

Assume we use the two-min algorithm (TMA) to decode LDPC codes as shown in Algorithm 2. Based on the algorithm, the memory accesses are summarized in Table 3.7.

For most of VLSI architecture, the VNP can be merged into the CNP. That said, the *a posteriori* probability (APP) information can be directly updated in the end of the CNP of each node, therefore, the VNP doesn’t require extra memory accesses. However, for more general cases, based on Table 3.7, the total numbers of memory...
reads \(N_{\text{mem\_read/iter}}\) and memory write \(N_{\text{mem\_write/iter}}\) can be represented as:

\[
N_{\text{mem\_read/iter}} = 2M\omega_r + N(\omega_c + 1),
\]

and

\[
N_{\text{mem\_write/iter}} = 2M\omega_r + N.
\]

To estimate the required memory bandwidth, we can utilize the throughput requirements to compute the effective memory accesses per each decoded bit. We define this metric as \(N_{MPDB}\), which can be computed as:

\[
N_{MPDB} = M \cdot 2\omega_r N_{it}/M = 2\omega_r N_{iter}
\]

The memory bandwidth requirement can be calculated as:

\[
BW_{mem} = T \cdot 2\omega_r N_{it} = 2T\omega_r N_{iter}
\]

As examples, if we consider the 802.16e WiMAX (1152, 2304) code, in which \(M = 1152, N = 2304, \omega_r = 7, \omega_c = 6\). We list the memory bandwidth requirements for typical throughput and iterations numbers in Table 3.8. To achieve 1 Gbps data rate at 15 iterations, the memory read (or write) bandwidth requirement is 210 GSamples/s. For an VLSI implementation, the extrinsic message can be represented.

\footnote{Please note that, different hardware architecture and memory architecture may result in different memory bandwidth requirement. But their memory bandwidth requirements should have the same order of magnitude as presented in the table.}
using 6bit, so the memory bandwidth requirement becomes 1260 Gb/s. If floating-point representation is used, the memory bandwidth requirement becomes 6720 Gb/s (or 840 GB/s).

Table 3.8: Memory bandwidth for different throughput requirements for LDPC decoding.

<table>
<thead>
<tr>
<th>Throughput</th>
<th>300 Mbps</th>
<th>600 Mbps</th>
<th>1 Gbps</th>
<th>1.5 Gbps</th>
<th>3 Gbps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>$N_{it} = 10$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>42</td>
<td>84</td>
<td>140</td>
<td>210</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>$N_{it} = 15$</td>
<td>63</td>
<td>126</td>
<td>210</td>
<td>315</td>
</tr>
<tr>
<td>[GSamples/s]</td>
<td>$N_{it} = 20$</td>
<td>84</td>
<td>168</td>
<td>280</td>
<td>420</td>
</tr>
</tbody>
</table>

**Flexibility and Configurability**

Similar to turbo decoding, it is also desirable to have flexible LDPC decoding implementations, so that the design cost can be reduced. The reconfigurable LDPC decoders can support different standards, block sizes, decoding algorithms, code rates and so on, as is summarized in Table 3.9.
Table 3.9: Flexibility requirements for a reconfigurable LDPC decoder.

<table>
<thead>
<tr>
<th>Item</th>
<th>Flexibility requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standards</td>
<td>WiFi, WiMAX, DVB-S2 etc.</td>
</tr>
<tr>
<td>Block sizes</td>
<td>A wide range of valid block sizes (e.g. 576–2304 in WiMAX)</td>
</tr>
<tr>
<td>Decoding algorithm</td>
<td>log-SPA, min-sum, offset min-sum</td>
</tr>
<tr>
<td></td>
<td>scaled min-sum, ...</td>
</tr>
<tr>
<td>Code rates</td>
<td>1/2, 2/3, 3/4, 5/6</td>
</tr>
<tr>
<td>Decoding architecture</td>
<td>Fully parallel, partial parallel</td>
</tr>
<tr>
<td></td>
<td>layered, flooding, etc.</td>
</tr>
<tr>
<td>Interconnection network</td>
<td>Shuffle network, network-on-chip</td>
</tr>
<tr>
<td></td>
<td>barrier shifter</td>
</tr>
</tbody>
</table>

**Performance Comparison between ASIC and GPU Implementations**

In Table 3.10, we compare the architecture and implementation results of ASIC [50] and GPU [45] implementations.

The ASIC implementation provides higher performance than the GPU implementation, at the extra cost of design effort and price. In terms of the configurability, the GPU implementation is much more flexible and can be easily modified to support new extensions of the algorithms. We can expect the GPU-based LDPC implementation to be a good candidate used in SDR communication systems. However, at the same time, we should notice that the GPU is huge in area and also consumes much more power than the ASIC implementation. These constraint factors prevent the GPU from being used in the mobile devices. However, as the mobile GPUs keeps evolving to gain more computation power, we can expect to see mobile-GPU based channel decoding modules in mobile SDR systems.
Table 3.10: Comparison between ASIC and GPU implementations of LDPC decoding.

<table>
<thead>
<tr>
<th></th>
<th>ASIC implementation [50]</th>
<th>GPU implementation [45]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Customized logic</td>
<td>NVIDIA GTX Titan</td>
</tr>
<tr>
<td>Technology</td>
<td>TSMC 45nm</td>
<td>TSMC 28nm</td>
</tr>
<tr>
<td>Clock freq</td>
<td>815 MHz</td>
<td>837 MHz</td>
</tr>
<tr>
<td># of cores</td>
<td>N/A</td>
<td>2688</td>
</tr>
<tr>
<td>Memory</td>
<td>SRAM</td>
<td>GDDR5 memory on-chip</td>
</tr>
<tr>
<td>Language</td>
<td>Verilog HDL</td>
<td>C++, CUDA C</td>
</tr>
<tr>
<td>Mode/standard</td>
<td>WiFi, WiMAX</td>
<td>WiFi, WiMAX</td>
</tr>
<tr>
<td></td>
<td>(can be easily extended)</td>
<td></td>
</tr>
<tr>
<td>Decoding algorithm</td>
<td>min-sum</td>
<td>log-SPA, min-sum (configurable)</td>
</tr>
<tr>
<td>Decoding schedule</td>
<td>Layered decoding</td>
<td>Fully parallel</td>
</tr>
<tr>
<td>Block size</td>
<td>Configurable</td>
<td>Configurable</td>
</tr>
<tr>
<td>Parallelism</td>
<td>$2 \times Z$</td>
<td>Configurable</td>
</tr>
<tr>
<td>Data type</td>
<td>Fixed-point</td>
<td>Float point</td>
</tr>
<tr>
<td>$N_{iter}$</td>
<td>fixed 15</td>
<td>Configurable</td>
</tr>
<tr>
<td>Decoding method</td>
<td>Streaming, one codeword at a time</td>
<td>Multi-codeword decoding.</td>
</tr>
<tr>
<td>Design effort</td>
<td>Several months</td>
<td>3–4 weeks</td>
</tr>
<tr>
<td>Throughput</td>
<td>1.1 Gbps @ 15iter</td>
<td>204 Mbps @15iter, on single GPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>844 Mbps @15 iter, on four GPUs</td>
</tr>
</tbody>
</table>

3.9.1 Analysis and Discussion

The LDPC decoding has very high parallelism in the CNP processing stage. It is suitable to spawn sufficient threads to handle the CNP computation for each check node. The irregular memory access patterns are determined by the parity-check matrix $H$ and the data requested by the CNP are quite random. In addition, due to the high parallelism of message passing, the memory bandwidth requirement is quite high (840 GB/s for 1 Gbps throughput goal), which has far exceeded GPU’s memory bandwidth. Apparently, the bottleneck of LDPC decoding’s throughput performance is the system memory bandwidth. Therefore, we can roughly estimate the achievable throughput by using the GPU’s peak memory bandwidth data.
Figure 3.12: Analysis of performance bottleneck of LDPC decoding algorithm.

\[
T_{\text{achievable}} = \frac{BW_{\text{mem, peak}}}{BW_{\text{mem}}} \cdot T.
\]

\[
= \frac{288.4}{840} \times 1 \text{Gbps}
\]

\[
= 343 \text{Mbps},
\]

in which \(BW_{\text{mem}}\) and \(BW_{\text{mem, peak}}\) represent the memory bandwidth requirement of LDPC decoding and the memory bandwidth provided by the GTX-Titan GPU.

Using our methodology described in Chapter 2, we can analyze the performance of the LDPC decoding algorithm, as shown in Figure 3.12. As we can see, as a memory-bound application, the predicted throughput performance is 343 Mbps, which is very close to the experimental result 319 Mbps. The LDPC decoding shows high paral-
lelism degree, which enables high throughput GPU implementation. Although the GPU implementation still cannot achieve 1 Gbps data rate, the achieved >300 Mbps data rate can meet requirement for many applications. Especially, when we consider its flexibility, reconfigurability, and low development cost, the high performance GPU-implementation shows its value. Moreover, the irregular memory access problem can be overcome to some extent by utilizing the on-chip shared memory to reorganize the data layout. For future optimization of LDPC decoding on GPU, memory optimization is still the key to achieving higher performance. Schemes that can reduce the memory traffic to the main memory such as data packing should be considered.

Although the current generation of desktop GPUs still have some limitations for which they cannot be used in a mobile system. But as the mobile GPU rapidly evolve, we will foresee the gap between the mobile GPU and desktop GPU is becoming smaller. Therefore, the optimization strategies proposed for the desktop GPU should also apply to the mobile GPU, as the mobile GPU gains more computation power and higher memory bandwidth.

Of course, if 1 Gbps is the hard requirement, we still need customized hardware logic to accelerate the decoding. The high clock frequency and fast on-chip memory accesses will guarantee that we achieve the required data rate.

3.10 Summary

As we can see from the experimental results and comparison, LDPC decoding algorithm provides inherently high parallelism, which fits into GPU’s parallel architecture very well. LDPC decoding algorithm shows very high parallelism among processing nodes, and especially, after optimization, the memory accesses become regular and are able to utilize GPU’s fast shared memory as data cache. These algorithm properties make GPU a good platform for algorithm acceleration. Therefore, we see GPU imple-
mentation provide comparable performance as the ASIC design, however, providing much better configurability and flexiblity.
Case Study 2: Parallel Multi-mode Turbo decoding

Wireless connectivity is one of the most important features of mobile devices. Modern mobile devices contain a communication modem chipset to enable advanced signal processing with high performance. These advanced signal processing algorithms are very computationally-intensive, therefore, accelerators using the application-specific integrated-circuits (ASIC) technology are usually incorporated into the mobile SoC chipset to achieve a certain performance goal. As one of the most computationally-intensive modules in the modern wireless communication modem, high throughput turbo decoding accelerator is challenging to implement, especially under strict area and power constraints. Therefore, we first study some key issues in parallel architecture design and implementation of high throughput turbo decoders.

During the past few years, modern 3G/4G wireless communication systems such as 3GPP (3rd Generation Partnership Project) UMTS/HSPA+ (Universal Mobile Telecommunications System/High-Speed Packet Access Evolution) [12], 3GPP LTE (Long Term Evolution) and LTE-Advanced [13] have been deployed to meet the ever-growing demand for higher data rates and better quality of service. High throughput
is one of the most important requirements for emerging wireless communication standards. For instance, the 3GPP UMTS standard Release 11 extends HSPA+ with several key enhancements including increased bandwidth and number of antennas. These enhancements lead to 336 Mbps peak data rate with $2 \times 2$ MIMO (multiple-input multiple-output) and 40 MHz bandwidth (or $4 \times 4$ MIMO and 20 MHz bandwidth) [12]. Recently, up to 672 Mbps data rate has been proposed for the future release of 3GPP standards [14, 15]. As a 4G candidate, the 3GPP LTE-Advanced promises up to 1 Gbps data rate as its long term goal.

Turbo codes are specified in many wireless communication standards such as the HSPA+ and LTE/LTE-Advanced as forward error-correction codes to ensure reliable communications via wireless channels, due to their outstanding error-correcting performance [21]. A turbo decoder contains two key components: soft-input soft-output (SISO) decoders and interleavers. During the decoding process, log-likelihood ratio (LLR) soft values are exchanged between component SISO decoders in an iterative way. The interleaver is a critical component for the turbo decoder to achieve good error-correcting performance, by permuting the LLRs randomly between iterations and maximizing the effective free distance of turbo codes. Since the introduction of turbo codes, numerous VLSI architectures and implementations have been proposed [99–102]. To achieve high throughput, parallel turbo decoding architectures are usually employed, in which several SISO decoders operate in parallel with each working on a segment of the received codeword [34, 101–115]. The parallel turbo decoders suffer from severe memory conflict problems due to the randomness of the interleaver, which becomes a major challenge for high throughput designs and implementations [34, 103, 107, 111, 112, 116–127].

Although LTE and LTE-Advanced can provide higher data rates than HSPA+, the high cost of LTE infrastructure prevents the rapid deployment of LTE systems;
in contrast, existing infrastructure can be upgraded with low cost to support the new features defined in HSPA+. Therefore, HSPA+ and LTE are expected to co-exist for a long term, especially when HSPA+ still keeps evolving rapidly [14, 15]. It is of great interest for mobile devices to support multiple standards. As an essential building block, multi-standard turbo decoders have been studied in the literature [34, 102, 105, 106, 109, 111, 114, 125, 128–130]. However, since the challenging memory conflict problem caused by the HSPA+ interleaving algorithm limits the parallelism degree of turbo decoders, none of them can meet the high throughput requirements of the recent HSPA+ extensions (336 Mbps for 3GPP Release 11 and 672 Mbps proposed for 3GPP Release 12 and up).

In this chapter, we propose a parallel interleaving architecture to solve the memory conflict problem for highly parallel multi-standard turbo decoding. The remainder of the paper is organized as follows. Section 4.2 introduces the memory conflict problem in parallel turbo decoding and related work focusing on the solutions to solve the memory conflict problem. In Section 4.4, we describe a balanced turbo decoding scheduling scheme to eliminate memory reading conflicts. In Section 4.5, we present a double-buffer contention-free (DBCF) buffer architecture to solve the memory writing conflict problem. In Section 4.6, we adapt a unified and on-the-fly interleaver/deinterleaver address generation architecture to the multi-standard turbo decoding architecture. In Section 4.7 we present a high throughput implementation of HSPA+/LTE multi-standard turbo decoder to demonstrate the effectiveness of the proposed architecture. Finally, we conclude this paper in Section 4.10.

### 4.1 Error-Correction Codes

Forward error correction (FEC) codes are designed to correct errors caused by noise and interference in a digital communication systems. In a practical system with
limited power, the energy per bit to single sided noise density ratio ($E_b/N_0$) is desired to be as low as possible. FEC adds redundancy using specially-designed codes by an encoder in the transmitter. At the receiver side, a decoder will take advantage of this redundancy information to correct as many errors as possible. Fig. 4.1 shows the channel coding scheme in a typical wireless communication systems.

**Table 4.1:** FEC codes used by major wireless communication standards.

<table>
<thead>
<tr>
<th>Generation</th>
<th>Technology</th>
<th>FEC codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2G</td>
<td>GSM</td>
<td>Convolutional codes</td>
</tr>
<tr>
<td>3G</td>
<td>W-CDMA, CDMA2000, 1xEV-DO, WiMAX (802.16e)</td>
<td>Convolutional codes, Turbo codes</td>
</tr>
<tr>
<td>4G</td>
<td>IMT-Advanced, LTE-Advanced, HSPA+ Advanced, WiMAX (802.16m)</td>
<td>Turbo codes, LDPC codes</td>
</tr>
</tbody>
</table>

Turbo codes and low-density parity-check (LDPC) codes are two classes of the most important FEC codes. Since they were introduced, they have been widely adopted by many wireless communication standards. They have become indispensable modules in modern wireless communication systems. Especially, when the wireless telecommunication systems evolve from 3G to 4G, the error-correction codes have migrated from convolutional codes to turbo codes and LDPC codes due to their superior error-correcting performance. Table 4.1 summarizes the channel coding schemes used by major wireless communication standards.
Table 1. Some applications of Turbo codes

<table>
<thead>
<tr>
<th>Application</th>
<th>Code structure</th>
<th>Polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDMA, WCDMA, UMTS, LTE</td>
<td>8-state binary</td>
<td>13, 15, 17</td>
</tr>
<tr>
<td>WiMax, DVB-RCS</td>
<td>8-state double-binary</td>
<td>15, 13</td>
</tr>
</tbody>
</table>

Binary Turbo Code in 3GPP LTE Standard

Turbo coding scheme in 3GPP LTE standard (3GPP TS 36.212, 2008) is a parallel concatenated convolutional code (PCCC) with two 8-state constituent encoders and one quadratic permutation polynomial (QPP) interleaver. The coding rate of the Turbo code is 1/3. The structure of the Turbo encoder is shown in Figure 1.

![Figure 1. Structure of rate 1/3 Turbo encoder in 3GPP LTE.](image)

4.1.1 Turbo Codes

Since they were first introduced by Berrou in 1993 [21], the turbo codes have been attracting a great deal of interest in the field of error-correction codes. Due to their outstanding near the Shannon capacity limit performance, many wireless communication standards have adopt turbo codes as the FEC codes, such as CDMA-2000, UMTS W-CDMA, DVB-RCS, HSDPA, IEEE 802.16e WiMAX, 3GPP LTE and 3GPP LTE-Advanced.

4.1.1.1 Turbo Encoder

Turbo encoder scheme in 3GPP UMTS/HSPA+ and 3GPP LTE standards is a parallel concatenated convolutional code (PCCC) with two 8-state constituent encoders and one internal interleaver [12, 13]. The structure of the 3GPP LTE turbo code with coding rate of 1/3 is shown in Fig. 4.2.
A turbo encoder consists of two binary convolutional encoders and an internal interleaver. The transfer function of the 8-state constituent code for PCCC is:

\[ G(D) = \begin{bmatrix} 1 & g_1(D) \\ g_0(D) & 1 \end{bmatrix}, \]  

(4.1)

in which

\[ g_0(D) = 1 + D^2 + D^3, \]  

(4.2)

\[ g_1(D) = 1 + D + D^3. \]  

(4.3)

The initial value of the shift registers in the 8-state constituent encoders are set to all zeros when we start the encoding process. After all information bits are encoded, trellis termination is performed by taking the tail bits from the shift register feedback.

As is shown on the left hand side of Fig.4.3, when a block of information bits \( x_k \) \((k = 0, 1, ..., K - 1)\) is streamed into an encoder, a sequence of systematic bits \( x_k^S \) \((x_k^S = x_k)\), a sequence of parity-check bits \( x_k^{P1} \) and a second sequence of parity-check bits \( x_k^{P2} \) are generated and transmitted over the wireless channel. The interleaver
permutes the input bits $x_k$ to $x_{\pi(k)}$ based on the interleaving law $\pi$ defined in standards.

### 4.1.1.2 Turbo Decoder

A turbo decoder receives soft reliability information for the transmitted sequences $x_k^S$, $x_k^{P_1}$ and $x_k^{P_2}$, in the form of log-likelihood ratios (LLRs), denoted as $L_k^S$, $L_k^{P_1}$ and $L_k^{P_2}$, respectively. The LLR indicates the probability of a received bit being a binary 0 or 1. The LLR can be defined as:

$$L(x_k) = \log \frac{\text{Prob}(x_k = +1)}{\text{Prob}(x_k = -1)}. \quad (4.4)$$

Shown in the right-hand side of Fig. 4.3, a turbo decoder consists of two constituent soft-input soft-output (SISO) decoders and an interleaver. The idea of the turbo decoding algorithm is to iteratively update the probability information between two SISO decoders, separated by an interleaver and a deinterleaver. In one iteration, two SISO decoders process reliability information and exchange so-called extrinsic information. The computation of each SISO decoder is called one half-iteration. In the first half-iteration, SISO decoder 1 computes extrinsic information $L_{k}^{E_1}$ using the LLRs of the received bits $L_k^S$, parity-check bits $L_k^{P_1}$ and a priori information $L_k^{A_1}$. $L_k^{A_1}$ is generated by deinterleaving the extrinsic information $L_{k}^{E_2}$ from the other SISO decoder ($L_k^{A_1} = L_{\pi^{-1}(k)}^{E_2}$). Similarly, during the second half-iteration, SISO decoder 2 generates extrinsic information $L_{k}^{E_2}$ based on $L_k^S$, $L_k^{P_2}$ and $L_k^{A_2}$ ($L_k^{A_2} = L_{\pi(k)}^{E_1}$). This iterative process continues until a preset maximum number of iterations is reached or a stopping criterion is met.

The maximum a posteriori (MAP) algorithm is normally used to implement SISO decoders [21]. The MAP decoding process can be represented as a trellis traversal. The trellis nodes represent the states of convolutional codes, and the branches rep-
Figure 4.4: Forward and backward recursions for trellis step \( k \) in the MAP decoding algorithm.

resent state transitions. Branch metric \( \gamma_k(s, t) \) denotes a transition from state \( s \) to state \( t \) at step \( k \).

\[
\gamma_k(s, t) = \frac{1}{2}(L_k^s + L_k^{Ai})u_k + \frac{1}{2}L_k^{Pi}c_k,
\]

(4.5)
in which, \( u_k \) and \( c_k \) are the input/output pairs for the branch connected state \( s \) to state \( t \) at step \( k \); \( i \ (i \in \{1, 2\}) \) indicates the index of a half iteration.

As shown in Fig. 4.4, the trellis traversal is performed in both forward and backward directions to compute the state metrics \( \alpha_k(s_k) \) and \( \beta_k(s_k) \) for all eight states:

\[
\alpha_k(s_k) = \max \{ \alpha_{k-1}(s'_{k-1}) + \gamma_k(s'_{k-1}, s_k), \alpha_{k-1}(s''_{k-1}) + \gamma_k(s''_{k-1}, s_k) \},
\]

(4.6)

\[
\beta_k(s_k) = \max \{ \beta_{k+1}(s'_{k+1}) + \gamma_{k+1}(s_k, s'_{k+1}), \beta_{k+1}(s''_{k+1}) + \gamma_{k+1}(s_k, s''_{k+1}) \},
\]

(4.7)

where the \( \max \) operation is typically implemented using the max-log approximation
as follows:

\[
\max^*(x, y) = \max(x, y) + \log(1 + e^{-|x-y|}) \approx \max(x, y). \tag{4.8}
\]

Once the forward metrics and backward metrics are computed, the LLRs of the \textit{a posteriori} probabilities (APPs) for information bits \(x_k\) are computed as follows:

\[
L(\hat{x}_k) = \max_{(s_{k-1}, s_k); x_k=0} \left\{ \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \right\} \\
- \max_{(s_{k-1}, s_k); x_k=1} \left\{ \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \right\}. \tag{4.9}
\]

For each MAP decoder \(i (i \in \{1, 2\})\), the extrinsic LLR information is computed by

\[L_{k}^{Ei} = L(\hat{x}_k) - L_{k}^{S} - L_{k}^{Ai}.\]

### 4.1.1.3 Internal Interleaver for Turbo Codes

Interleaver is an essential component for turbo codes to achieve outstanding error-correction performance. The interleaver is designed to mix the bits in a frame at the input of the second constituent encoder. The concept is to combine the low weight codewords generated from one encoder with the high weight ones generated from the other, so that a reduced multiplicity of the codewords of low global weight can be achieved. Mathematically, an interleaver can be described by a permutation function

\[\pi : I \rightarrow I \tag{4.10}\]

where \(I = \{0, 1, 2, ..., K - 1\}\) is the set of indices corresponding to the input and output bits. According to the interleaving law defined in the turbo encoding algorithm, the bit stream will be scrambled before feeding into the second constituent
encoder. To achieve better error-correcting performance, interleavers typically utilize pseudo-random algorithms. Due to randomness of interleaving algorithms, the memory accesses in turbo decoders become quite random, which as a result cause serve memory conflict problem when parallel decoding architecture is employed.

During the past decades, numerous interleaver algorithm have been proposed [120, 131–133]. 3GPP UMTS/HSPA+ standards use a pseudo-random interleaver based on matrix permutation [12]. The 3GPP LTE and 802.16e WiMAX standards use quadratic polynomial permutation (QPP) [131] and almost regular permutation (ARP) [120] interleavers, respectively.

Here, we provide a brief introduction to interleaving algorithms defined in 3GPP UMTS/HSPA and LTE standards.

**Pseudo-random interleaver in 3GPP UMTS/HSPA+**

The interleaving algorithm in the 3GPP UMTS/HSPA+ standards is based on matrix permutation. We first write the bit sequence $x_k$ ($k = 0, 1, \ldots, K - 1$) into a rectangular matrix with $R$ rows and $C$ columns. $R$ is defined as

$$R = \begin{cases} 5, & \text{if}(40 \leq K \leq 159) \\ 10, & \text{if}(160 \leq K \leq 200) \text{ or } (481 \leq K \leq 530). \\ 20 & \text{if } f(K = \text{any other value}) \end{cases} \quad (4.11)$$

We then determine the prime number used in the intra-permutation $p$ and $C$. If $K$ satisfies $481 \leq K \leq 530$, we have $p = 53$ and $C = p$; otherwise, $p$ is defined as the minimum prime number from a prime number table defined in the standard such
that $K \leq R \times (p + 1)$, and $C$ can be defined as

$$C = \begin{cases} 
  p - 1, & \text{if } K \leq R \times (p - 1) \\
  p, & \text{if } R \times (p - 1) < K \leq R \times p. \\
  p + 1 & \text{if } R \times p < K
\end{cases} \quad (4.12)$$

Once the bits stream in a block is written the $R \times C$ rectangular matrix, the intra-row permutations and inter-row permutations are performed. The intra-row permutation $s(j)$ can be defined as:

$$s(j) = (v \times s(j - 1)) \mod p, j = 1, 2, \ldots, (p - 2), \text{and } s(0) = 1. \quad (4.13)$$

Use inter-row permutation pattern $T(i)$ to permute a sequence $q_i$ to get a sequence $r_i$, such that $r_{T(i)} = q_i, i = 0, 1, \ldots, R - 1$. Then, we can perform the $i$-th intra-row permutation as

$$U_i(j) = s((j \times r_i) \mod (p - 1)), \, j = 0, 1, \ldots, (p - 2), \text{ and } U_i(p - 1) = 0. \quad (4.14)$$

where $U_i(j)$ is the original bit position of $j$-th permuted bit of $i$-th row. Finally, inter-row permutation is performed based on the pattern $T(i) (i = \{0, 1, \ldots, R - 1\})$, in which $T(i)$ is the original bit position of $j$-th permuted bit of $i$-th row.

**QPP interleaver in 3GPP LTE**

Given an block length $K$, the $x$-th interleaved output position is given by

$$\pi(x) = f_2 x^2 + f_1 x \mod K. \quad (4.15)$$

where $f_1$ and $f_2$ are integers defined in the standard and depend on the block size
Figure 4.5: (a) Parallel turbo decoding architecture with $P$ SISO decoders and $M$ extrinsic memory modules; (b) A 2-way memory conflict happens when SISO decoder 1 and SISO decoder $P$ access (read from or write to) memory 2 at the same time.

$K$ ($0 \leq x, f_1, f_2 < K$). The block size $K$ defined in the 3GPP LTE standard ranges from 40 to 6144.

The interleaver is usually implemented using a look-up table. However, if the block is very long or configurable decoder is required, it may not be feasible to store all possible interleaver patterns in a look-up table. In those cases, the interleaver pattern need to be generated on the fly.

## 4.2 Parallel Turbo Decoding Algorithm and Memory Conflict Problem

### 4.2.1 Parallel Turbo Decoding Architecture

To achieve high throughput, numerous parallel turbo decoding architectures have been extensively investigated and several levels of parallelisms have been explored [34, 35, 102–104, 106–108, 110–112, 114].

Most parallel turbo decoding algorithms exploit parallelism in SISO-decoder level, where a codeword (with block size $K$) is partitioned into $P$ sub-blocks with size of each
being $K/P$. Multiple SISO decoders work in parallel with each operating on one of the sub-blocks as shown in Fig. 4.5a. The Radix-4 SISO decoding algorithm is another way to increase the throughput by exploring the trellis-level parallelism. It applies a one-step look-ahead concept to the forward/backward trellis recursions, in which two trellis steps are computed simultaneously [107, 108, 110, 111, 129]. In addition to the above algorithms, many variants of SISO decoder architectures explore parallelism from other aspects such as recursion-unit level parallelism, sliding window schemes and so on [102, 106–108, 110, 111]. Therefore, it is quite challenging to design an interleaver architecture which is able to handle the aforementioned different variants of parallel turbo decoding algorithms. Our goal is to design a flexible and scalable interleaver architecture which can fulfill this requirement.

In our implemented parallel turbo decoder, we choose a Radix-4 cross-MAP (namely XMAP) decoding architecture to implement the SISO decoders due to its high throughput performance [110]. In an XMAP decoding architecture as is shown in Fig. 4.6, $\alpha$ and $\beta$ recursion units work in parallel in a crossed manner, towards the forward and backward directions, respectively. In each half iteration, before the cross point, the $\alpha$ and $\beta$ recursion units compute $\alpha$ and $\beta$ state metrics. After the cross point, each Radix-4 XMAP decoder starts to produce four extrinsic LLR values per clock cycle. When $P$ such SISO decoders working in parallel to decode a codeword with block size $K$, each SISO decoder performs computation on a segment (or sub-block) of the codeword with size $K/P$. A method call Next Iteration Initialization (NII) is employed to avoid acquisitions for the $\alpha$ and $\beta$ state metrics [100, 110]. Only the stake information in the end of the recursion in a sub-block is saved and propagated between SISO decoders.
Figure 4.6: Diagram of the cross-MAP (XMAP) decoding architecture with NII.
4.2.2 Design Challenges

A parallel turbo decoder which generates $P_{LLR}$ extrinsic LLRs in each clock cycle requires a memory bandwidth of $P_{LLR}$ memory accesses per clock cycle. Memory used to store the extrinsic LLR values are partitioned into $M$ ($M \geq P_{LLR}$) memory modules to provide higher memory bandwidth.

However, due to the randomness of the interleaver/deinterleaver, multiple SISO decoders may attempt to access the same memory bank concurrently, which causes a memory conflict, as is depicted in Fig. 4.5b. A significant reduction in effective memory bandwidth caused by frequent memory conflicts decreases the decoding throughput. Therefore, the memory conflict problem has become a major bottleneck in the design of a high throughput parallel turbo decoder [34, 35, 103, 107, 111, 112, 116–126].

A traditional way of implementing the turbo code interleaver is static memory mapping when the parallelism of turbo decoders is low. The interleaving patterns are precomputed and stored in a look-up table (LUT). For every decoding algorithm and parallel architecture, a unique LUT is required since significantly different memory access patterns are generated. If multiple standards and variable block sizes need to be supported, memory usage for LUTs will grow drastically. This apparently is not efficient for an embedded SoC implementation. Therefore, on-the-fly interleaver address generation (IAG) is preferred in highly parallel turbo decoders since it requires less memory and can be easily parametrized to support different configurations and multiple standards.
Figure 4.7: (a) Memory writing conflict, in which 3 SISO decoders try to write to the same memory; (b) Memory reading conflict, in which 3 SISO decoders try to read data from the same memory. $L_1$, $L_2$ and $L_3$ represent LLR data; $D$ represents a clock cycle’s delay.
4.3 Related Work

Many interleaver architectures have been proposed to solve the memory conflict problem in parallel turbo decoding systems, and they can be classified into three categories: design-time solutions, compilation-stage solutions and runtime solutions [102, 124].

(1) Design-time solutions usually employ algorithm-architecture co-design methods in the early design stage [107, 124]. Contention-free algorithms are developed by applying specific algebraic properties to the interleaver algorithms. As examples, the ARP (almost regular permutation) interleaver adopted by the WiMax standard and the QPP (quadratic permutation polynomial) interleaver used by LTE/LTE-Advanced standards fall into this category [120, 122]. However, interleavers designed by these design-time methods can be contention-free only under some constraints. For instance, the QPP interleaver is contention-free when the decoding parallelism is a factor of the block size $K$ [110, 122]. In addition, these solutions lack flexibility to support existing interleaving algorithms, such as the 3GPP HSPA+ interleaver. Therefore, in any of the above cases, we should seek other solutions.

(2) Compilation-stage solutions employ certain memory mapping rules to generate data access patterns to avoid memory conflicts [104, 112, 116, 119, 126]. It has been shown by Tarable et al. that there is always a memory mapping scheme which allows contention-free memory accessing for any interleaving algorithm in a parallel turbo decoder [119]. Recently, Chavet et al. proposed memory mapping methods based on graph coloring algorithms, in which a memory mapping pattern can be calculated in polynomial time [112, 126]. The limitation of the compilation-stage solutions is that they require complex off-line computations to generate memory mappings and a large amount of memory resources to store memory mappings. To alleviate this problem, Inseher et al. proposed to compress permutation vectors to reduce the memory requirement. However, the permutation vectors still lead to large memory
area, even when a hybrid compression method is used [102].

(3) Runtime conflict solutions use buffers or on-chip networks to dynamically reschedule data access orders to avoid memory conflicts [34, 103, 111, 117, 118, 121, 123, 125, 127]. As discussed above, both the design-time solution and compilation-stage solution require the memory access patterns to be precomputed and stored in memory. As a comparison, a runtime conflict solution does not need large memories. It can provide a maximum degree of flexibility, enabling highly configurable multi-standard turbo decoder implementation. Therefore, we prefer runtime conflict resolution. In the rest of this section, some existing runtime solutions are briefly described.

Thul et al. proposed a tree-based concurrent interleaving architecture (namely TIBB) and an improved solution based on local buffer cells interconnected via a ring network (namely RIBB), respectively [103, 117]. These solutions have a high connectivity of LLR distributors, which limit the max clock frequency of the circuits and the parallelism of a turbo decoder. In [118], Speziali et al. extended Thul’s work and proposed an improved architecture for the interconnection network. A stalling mechanism is introduced to the interleaver architecture. However, the stalling mechanism leads to a high hardware complexity and results in unacceptable delay penalty for highly parallel SISO decoder architectures such as Radix-4 SISO decoders. Recently, Network-on-chip (NoC) approaches have been proposed to solve the memory conflict problem. For example, in [121, 123, 127], the authors proposed packet-switched NoC approaches such as Butterfly NoC and Benes-based NoC architectures. However, these NoC methods suffer from a large delay penalty limiting the maximum throughput. Furthermore, to avoid network contention, complex scheduling methods and control logic are required to schedule the network packets. In [111], Asghar et al. presented misalignment among memory access paths using delay line buffers. In fact,
these schemes can only alleviate the memory conflict problem when the parallelism of turbo decoder is low. The delay line buffers also cause long delay and increase hardware cost.

To overcome the limitations of the aforementioned solutions, we propose a flexible runtime interleaver architecture to solve the memory conflict problem efficiently, so as to enable high throughput multi-standard turbo decoding.

### 4.4 Eliminating Memory Reading Conflicts via Balanced Turbo Decoding Scheduling

In a traditional turbo decoding architecture, due to the existence of data dependencies during trellis traversal in a turbo decoder, SISO decoders should stall until the currently requested datum is ready. Frequent stalling of SISO decoders will significantly decrease the decoding throughput. In order to achieve high throughput, we should keep all the SISO decoders running at full speed without stalling. Fig. 4.7 demonstrates two cases showing traditionally how the memory conflict problem was solved using memory conflict solvers, when three SISO decoders are trying to access (write or read) the same memory module. We will analyze the disadvantage of these solutions, and then discuss our solution in Section 4.4.2.

#### 4.4.1 Properties of Memory Reading and Writing Conflicts

For a memory writing conflict (Fig. 4.7a), a buffer structure can be used as a “write conflict solver” to cache the data accesses and resolve the memory conflicts. All the SISO decoders can execute at full speed so that the overall high throughput can be guaranteed. In contrast, if three SISO decoders request data from memory in the same clock cycle (Fig. 4.7b), only SISO decoder-1 can get the requested LLR
value $L_1$ immediately. The SISO decoder-2 must wait for a clock cycle to get $L_2$ and SISO decoder-3 must wait for two clock cycles to get $L_3$. Although buffering and data prefetching may help reduce the number of memory reading conflicts, it is difficult to completely eliminate memory reading conflicts in a practical hardware implementation. In addition, the reading conflict solver and the writing conflict solver are so different that implementing both of them utilizes more hardware resources and requires extra multiplexing logic.

### 4.4.2 Unbalanced and Balanced Scheduling Schemes

In this Section, we describe the advantages of a balanced scheduling scheme to deal with the memory conflict problem. There are two scheduling schemes when implementing a turbo decoder, namely an unbalanced scheme and a balanced scheme. As shown in Fig. 4.8a, in an unbalanced scheduling scheme, the SISO decoder reads
(writes) contiguous data from (to) the memory in a natural order in the 1st half iteration. During the 2nd half iteration, the SISO decoder first reads data from memory in an interleaved way; once computation is done, the SISO decoder writes the output data into memory in a deinterleaved way. Due to the randomness of the interleaving/deinterleaving algorithms, both memory reading and writing in the 2nd half iteration suffer from memory contentions in a parallel turbo decoder.

Fig. 4.8b shows the balanced scheduling scheme in which SISO decoders write output data in the interleaved/deinterleaved order in both half iterations, so that in the next half iteration they can read the data in a naturally continuous order. Since all the memory reading operations are in-order, there is no memory conflict at all [35]. The balanced scheduling scheme can lead to more efficient designs than the unbalanced scheduling scheme because of the following reasons. First of all, the memory writing conflict is easier to solve than the memory reading conflict as is discussed in Fig. 4.7. Secondly, without memory reading conflicts, we can remove the reading conflict solver hardware from the interleaver and reduce the hardware cost. Thirdly, the balanced scheduling method generates a more balanced workload between two half iterations. Since both the read and write conflict solvers introduce extra penalty clock cycles, the memory conflicts occurring in both memory reading and writing in a unbalanced scheduling scheme lead to more penalty clock cycles in the 2nd half iteration. Furthermore, during the 2nd half iteration in unbalanced scheme, the control logic need to keep track of the execution status of SISO decoders, since the SISO decoders may run/stall independently. The bookkeeping and frequent stalling/restarting operations on the SISO decoders require more complex control logic in the unbalanced scheduling scheme. In the balanced scheduling scheme, the symmetric structure between two half iterations results in a more balanced workload, which simplifies the control logic.
Due to the above-mentioned benefits, we choose the balanced scheduling scheme in our multi-standard turbo decoder implementation.

4.4.3 Further Discussions

Although the balanced scheduling scheme requires both interleaver and deinterleaver hardware modules, the fact that the interleaver and deinterleaver algorithms typically have symmetric structures implies that hardware sharing is possible to save computational resources and storage resources. Therefore, the overhead of a two-mode interleaver/deinterleaver would be small compared to a single-mode interleaver, if hardware sharing is explored. In addition, as an enabler for highly parallel turbo decoders supporting standards that do not employ contention-free interleaver algorithms (such as the HSPA+ standard), a small hardware overhead can be acceptable. This hardware sharing method will be further discussed in Section 4.6 and Section 4.7.1.

4.5 Double-buffer Contention-free Architecture for Interconnection Network

By using the balanced turbo decoding scheduling scheme, we eliminate the memory reading conflicts, but the memory writing conflicts still exist. In this section, we describe a double-buffer contention-free (DBCF) architecture to solve the memory writing conflict problem with low hardware complexity [34]. We first introduce the methodology and system model to analyze properties of memory conflicts and then describe the DBCF architecture in detail.
4.5.1 Methodology and System Model

To study the statistical property of memory conflicts under an interleaving law with a special parallel architecture, a cycle-accurate simulator is designed using a methodology shown in Fig. 4.9. The simulator is written in the C language, and can simulate the behaviors of SISO decoders’ I/O ports, the buffer system and the memory system in each clock cycle. Interleaving address patterns are calculated and stored in a look-up table in the beginning of a simulation. The simulator can be easily extended to support new interleaving laws. This cycle-accurate simulator can also be used to verify or simulate a memory conflict solver by modifying the buffer system simulator. This function is used to determine some architectural parameters of the DBCF architecture in Section 4.5.3.

Depending on the parallel decoding strategies and memory organizations, the
memory access patterns can vary from case to case. Our implemented parallel turbo decoder consists of $P$ parallel Radix-4 XMAP decoders as described in Section 4.2.1, and $M$ memory modules storing the extrinsic LLR values. In such a system, a Radix-4 XMAP decoder produces 4 new LLR data in each clock cycle. The total number of LLR data to be written in the extrinsic memory per clock cycle is defined as $P_{LLR} = 4P$, which can be considered as the effective parallelism of a parallel turbo decoder.

4.5.2 Statistical Analysis of Memory Conflicts

The HSPA+ mode is the most challenging part in a high throughput multi-standard turbo decoder due to the frequent and severe memory conflicts [111, 125]. Therefore, in order to design an efficient VLSI architecture, it is important to study the properties of memory conflicts in HSPA+ mode. We first analyze how the memory conflict ratio changes as the block sizes and the effective parallelism $P_{LLR}$ change by performing
Figure 4.11: The percentage of $n$-way memory conflicts (for $P_{LLR} = 16$ and $P_{LLR} = 32$). Interleaving algorithm from 3GPP HSPA+ standard is employed. Turbo code block size is $K = 5114$.

cycle-accurate simulations. Memory conflict ratio (MCR) is defined as a ratio of the number of clock cycles with memory conflicts to the total number of clock cycles for a half iteration: $MCR = N_{conflict\_cycle}/N_{total\_cycle}$. The MCR can indicate how severe the memory conflict problem is for certain decoding configurations. Fig. 4.10 shows that for a fixed parallelism the MCR does not change significantly across different block size configurations. However, when parallelism $P_{LLR}$ goes higher, the MCR increases dramatically. For parallelism $P_{LLR}$ higher than 8, memory conflicts occur in almost every clock cycle, which indicates very severe memory conflict problems. Therefore, as the parallelism goes higher, the difficulty to resolve the memory conflict problem increases drastically.

Simulation results show that during the whole decoding process, the average number of memory accesses for each memory module per clock cycle is very close to 1. This implies the possibility to use buffers to smooth the bursty data requests to a memory module, and finally to achieve near 1 data/cycle/memory throughput. We further analyze the statistical distribution of memory conflicts. If $n$ LLR data in the same
memory module are accessed by multiple SISO decoders in one clock cycle, we call it an $n$-way memory conflict ($n \in [2, P_{LLR}]$). The percentage of $n$-way memory conflicts ($Percent_{n\_way}$) is calculated as the ratio of the number of $n$-way conflicts ($N_{n\_way}$) to the total number of memory conflicts ($N_{Conflict}$): $Percent_{n\_way} = N_{n\_way}/N_{Conflict}$.

The percentage of $n$-way memory conflicts for parallelism of 16 and 32 in 3GPP HSPA+ interleaver are shown in Fig. 4.11. We can notice that most of the memory conflicts are 2-, 3- and 4-way conflicts. These three types of conflicts cover 93.35% and 82.97% of the memory conflicts for $P_{LLR} = 16$ and $P_{LLR} = 32$, respectively. Based on the above observations, we propose a double-buffer contention-free architecture (DBCF) to efficiently solve the memory conflict problem.

### 4.5.3 Double-buffer Contention-free (DBCF) Architecture

The DBCF buffer architecture is built around the interleaver (with interleaving address generation units) between the SISO decoders and memory modules. Fig. 4.12 shows the diagram of the DBCF architecture. Major components of the DBCF architecture include FIFOs associated with SISO decoders, circular buffer associated with memory modules, buffer routers and bypass units. The name of DBCF architecture comes from the fact that two sets of buffers (FIFOs and circular buffers) are employed, which gives the proposed architecture a large advantage over the traditional single-buffer-based architectures [103, 118].

#### 4.5.3.1 Micro-architecture of DBCF

The main idea of the DBCF architecture is to use circular buffers to cache the concurrent data writing operations. The circular buffers should be implemented using registers in order to support concurrent data writing. A write pointer and a read pointer are provided to control the buffer access. If full inter-connection is fulfilled
between the interleaver and the circular buffers, the exponentially growing hardware cost as the parallelism goes higher makes the implementation very inefficient [118]. According to the statistical analysis of memory conflicts, it is rare to have an \( n \)-way memory conflict for \( n > 4 \), therefore, a selection-based data routing scheme is implemented by the buffer router. We define a selection parameter \( S (S \in [1, N_{LLR}]) \), which controls the number of data that are allowed to be routed to the circular buffers in a clock cycle. At any time, up to \( S \) LLR data are allowed to access the circular buffer, the other \( \max(0, n - S) \) LLR data are rejected for a \( n \)-way memory conflict case. For example, if there are 5 LLR data trying to access a certain memory module \( (n = 5) \). The conflict detector detects a memory conflict. Assume we have set \( S = 3 \), 3 out of 5 LLR data will be stored into the circular buffer associated with the memory module. At the same time, the remaining 2 LLR data \( (\max(0, n - S) = \max(0, 5 - 3) = 2) \) are pushed into the FIFOs connected to the producer SISO decoders. This selection-based
data routing method reduces complexity of the inter-connection network between the interleaver and the circular buffers from $P_{LLR} \times P_{LLR}$ to $P_{LLR} \times S$.

The conflict detector can detect the memory conflicts at runtime using a compare-select cluster. It is worth mentioning that in the proposed architecture memory conflicts are detected and solved at runtime, without any pre-stored data sets from the initial design phase. Then, with the help of pseudo-random number generator (PRNG) circuits, priority selector circuits determine which $S$ data are selected with equal fairness. The buffer control units maintain the writing and reading operations to the circular buffers. The bypass units directly route the data into the memory module when the circular buffer is empty to avoid unnecessary latency caused by circular buffers.

The FIFOs associated with SISO decoders is one of the important reasons why the DBCF architecture can efficiently solve the memory writing conflict problem. If a traditional single-buffer architecture is used, we have to stall the corresponding SISO decoders if their LLR outputs are rejected by the buffer router circuits; otherwise, the rejected data will be lost. Thanks to the FIFOs introduced by the DBCF architecture, a rejected LLR datum is pushed into a FIFO associated with the SISO decoder that produced the rejected LLR datum.

4.5.3.2 Further Analysis of DBCF Architecture

The DBCF architecture can significantly reduce the memory usage for the data buffers. A model in Fig. 4.13 can help further understand the principle behind the DBCF architecture. The two main components of the DBCF architecture, FIFOs and circular buffers, work collaboratively like two sets of springs which distribute data pressure evenly across both side of the inter-connection network. Think about a possible case in which multiple SISO decoders send data concurrently to memory-1
Figure 4.13: Dynamic balance of data pressure on inter-connection network via “spring-like” structure of the DBCF architecture.

(MEM 1) in several consecutive clock cycles. This burst behavior generates high data pressure on the circular buffer associated with MEM 1. Therefore, if without FIFOs, a large circular buffer is required to handle the occasional burst data accesses, which leads to inefficient implementations. As a comparison, in DBCF architecture, we use small circular buffers. Even when the circular buffer is fully occupied by the bursty data, the incoming data are pushed into the FIFOs, so that the circular buffers will not overflow and we do not lose any data. That being said, the FIFOs and circular buffers together maintain a dynamic balance of the data pressure on the interconnection network. As a result, the size of the circular buffers can be significantly reduced. Moreover, since the rejected data are stored in several FIFOs associated with SISO decoders, with each FIFO accepting only one datum. The data back pressure is gently distributed across FIFOs, so that small FIFO size is sufficient.

In addition to the function of pressure balancing, the insertion of FIFOs also decouples the SISO decoders from the interleaver. This is important because of the following reasons: (1) with DBCF architecture, SISO decoders can run at full speed
Table 4.2: Simulation results for parameter determination.
HSPA+ interleaver, $K = 5114$, $P_{LLR} = 16$.

<table>
<thead>
<tr>
<th>Simulation parameters</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{LLR}$</td>
<td>$M$</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 4.3: Simulation results for parameter determination.
HSPA+ interleaver, $K = 5114$, $P_{LLR} = 32$.

<table>
<thead>
<tr>
<th>Simulation parameters</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{LLR}$</td>
<td>$M$</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

without stalling, no matter how severe the memory conflicts occur on the right hand side of interleaver. This is a key to achieving high throughput; (2) any SISO decoder architecture is supported to achieve high throughput. For instance, a Radix-4 XMAP decoder generates 4 LLR data per clock cycle, among which some may be successfully written into a memory module, while others may be pushed back to FIFOs. But the Radix-4 XMAP decoder keeps producing new outputs all the time. Without the help of the proposed double-buffer architecture, a Radix-4 XMAP decoder has to stall even if only 1 LLR value (out of 4) is rejected by a buffer router; (3) The decoupling between SISO decoders and interleaver also reduces the complexity of the SISO decoders since no feedback control from interleaver (or memory modules) to the SISO decoders is needed.

4.5.3.3 Determining Architectural Parameters

To determine a parameter configuration (selection parameter $S$, FIFO depth $D_{FIFO}$, circular buffer depth $D_{buf}$) for a parallel turbo decoder with parallelism of $P_{LLR}$ and $M$ memory modules, the cycle-accurate simulation is performed by adding the DBCF
Figure 4.14: Architecture of unified parallel interleaving address generation (IAG) supporting both interleaving and deinterleaving. Note that we need $P_{LLR}$ copies of each duplicated block in this figure.

architecture to the buffer system simulator. By changing different combinations of ($S$, $D_{FIFO}$, $D_{buf}$), we can tune the design to meet different performance/complexity goals in terms of memory usage, decoding latency, and complexity of the interconnection network. In Table 4.2 and Table 4.3, parameter configurations for $P_{LLR} = 16$ and $P_{LLR} = 32$ to achieve high throughput goal as well as the corresponding simulation results are shown. Here, we show results for the largest block size in HSPA+ standard ($K = 5114$), which typically causes more serious memory conflict problem and longer latency. In these tables, $M$ denotes number of banks in the extrinsic memories; $C_0$ denotes the number of clock cycles to decode a codeword in an ideal case without any memory conflict; $C_1$ denotes the actually clock cycles including the clock cycles used to solve memory conflicts; $\Delta C$ denotes extra clock cycles actually used compared to the ideal case ($\Delta C = C_1 - C_0$). Although all SISO decoders run at full speed without stalling in the proposed architecture, the FIFOs and circular buffers still need a few clock cycles to unload data, which causes extra clock cycles $\Delta C$. A
well-tuned parameter configuration can lead to small $\Delta C$. Taking Table 4.3 as an example, the first row provides a baseline case in which only a large circular buffer (without priority selection and FIFOs) is used to cache the bursty data. In this baseline case, with buffer size set to 128 ($D_{buf} = 128$), 108 extra clock cycles are needed to finish decoding. The second row shows the usage of priority selection with $S = 3$, $D_{FIFO} = 8$ and $D_{buf} = 12$. In this case, 10 extra cycles are needed. The third row shows the case in which sub-banking techniques (here, we set $M = 2 \times P_{LLR}$ for sub-banking) are used to further reduce $D_{FIFO}$ (reduced to 4) and $D_{buf}$ (reduced to 7), and only 4 extra clock cycles are required to finish the decoding. Table 4.2 and Table 4.3 have also demonstrated the effectiveness and efficiency of the proposed DBCF architecture in terms of the reduced buffer sizes and decreased extra clock cycles.

4.6 Efficient Unified Interleaver/Deinterleaver Architecture

To improve the flexibility and configurability of a turbo decoder, it is of great interest but also quite challenging to design and implement an on-the-fly interleaving address generator (IAG) supporting both interleaving and deinterleaving modes for multiple standards. In this section, we propose a unified parallel IAG architecture to support both interleaver and deinterleaver algorithms with low hardware overhead.

We notice that many interleaving laws in standards have similar computation kernels for both interleaving and deinterleaving. For instance, a pseudo-random column-row permutation algorithm is employed in 3GPP UMTS/HSPA+ standards [12]. The core operation of the HSPA+ interleaver can be summarized as computing a column index function $U_i(j)$ from the original column $j$ after intra-row permutation $j \rightarrow U_i(j)$
for the $i$-th row, using the formula $U_i(j) = s((j \times r_i) \mod (P - 1))$. Similarly, it has been proved that the core operation in the UMTS/HSPA+ deinterleaver is to compute an intra-row permutation $j' \rightarrow U_i^{-1}(j')$ using formula $U_i^{-1}(j') = (s^{-1}(j') \times m_i) \mod (P - 1)$ [36]. The core computation units in the interleaver and deinterleaver can be summarized as the same computation kernel $(a \times b) \mod c$. Furthermore, most of the intermediate values precomputed by the pre-processing units can be reused, except for only a few intermediate values used exclusively by the deinterleaver mode.

Similar observations can be made by investigating the LTE QPP (Quadratic Permutation Polynomial) interleaving algorithm [13]. The QPP interleaver implements a permutation based on the quadratic polynomial $f(x) = (f_1 x + f_2 x^2) \mod K$. Efficient implementations by recursively computing $f(x)$ have been proposed [108, 110].

Researchers have proved that the inverse permutation in the form of quadratic polynomials exist for most of the $K$ (153 out of 188), and for other $K$ inverse permutation in the forms of cubic (31/188) or quartic (4/188) polynomials can be found [134, 135]. Similarly, the deinterleaving addresses can also be computed recursively using efficient hardware. Therefore, the QPP interleaver and deinterleaver can also share the kernel computation unit $((a + bx) \mod c)$.

Based on the above observations, we propose a unified interleaver/deinterleaver architecture which shares a large portion of the hardware between the interleaver and deinterleaver modes, leading to only a small overhead compared to a single-mode interleaver. Fig. 4.14 shows the proposed unified parallel IAG architecture, which consists of the following key blocks: (1) preset parameter storage, (2) preprocessing unit, (3) runtime parameter storage, and (4) runtime IAG block. Here, let us take the column-row random interleaving algorithm in the HSPA+ standard as an example to explain how these blocks work. The algorithm details and notation definitions can be found from 3GPP UMTS/HSPA+ standards [12] or references [36, 113]. The preset
parameter storage implemented using ROMs contains static parameters such as inter-row permutation pattern $T_i$ ($128 \times 5$ bit tROM) and permutated sequence $r_i$ ($1440 \times 7$ bit rROM) used by both modes, and the modular multiplicative inverse sequence $m_i$ ($1200 \times 7$ bit mROM) used exclusively by the deinterleaver [36]. The preprocessing unit computes the parameters used in the runtime IAG block. The preprocessing unit first generates several runtime parameters based on the block size $K$, such as the number of rows/columns ($R/C$) of the permutation matrix, and then stores them in registers. It also produces permutation patterns such as the base sequence for intra-row permutations $s_i$ ($256 \times 8$ bit SRAM) and the corresponding inverse base sequence used in the deinterleaver mode $s_{inv,i}$ ($256 \times 8$ bit SRAM). Finally, the runtime IAG block, which consists of computation and control logic, converts input addresses to the interleaved/deinterleaved addresses. Based on the input block size $K$ and interleaver/deinterleaver mode selection, the runtime IAG can be configured to work in the corresponding mode.

As is shown in Fig. 4.14, we need to duplicate some memory and computation units for $P_{LLR}$ times to generate $P_{LLR}$ parallel addresses in one clock cycle. It is worth noting that we only need one single copy of the preprocessing hardware. The balanced scheduling scheme proposed in Section 4.4 makes it possible that the parallel interleaver/deinterleaver can function in a time-division multiplexing manner (different half iterations, to be specific). Therefore, a significant amount of hardware resources for two modes can be shared between two half iterations, which include the preprocessing unit, kernel computation units inside the runtime IAG, as well as ROMs and RAMs storing necessary parameter arrays. The hardware sharing feature has been integrated in our turbo decoder implementation presented in Section 4.7, which can demonstrate the efficiency of the proposed unified IAG architecture.
Figure 4.15: Overall architecture of the proposed multi-standard (HSPA+/LTE/LTE-Advanced) turbo decoder.

4.7 VLSI Implementation

To show the effectiveness of the proposed architecture, we implement a high throughput HSPA+/LTE multi-standard turbo decoder as is shown in Fig. 4.15. The design goal is to achieve more than 1 Gbps throughput for LTE mode and 672 Mbps throughput for HSPA+ mode.

The Radix-4 XMAP decoding architecture described in Section 4.2.1 is used to implement the SISO decoders. The max-log-MAP algorithm is employed with a scaling factor 0.75 applied to improve the decoding performance [136]. In the SISO decoder, fixed-point representation is used for state metrics and LLR data as follows:
Figure 4.16: Bit error rate (BER) performance with fixed-point simulation for turbo decoder (code rate 1/3) with the proposed parallel architecture, using BPSK modulation over an additive white Gaussian noise (AWGN) channel. The parallelism degrees for HSPA+ and LTE mode are 8 and 16, respectively. The number of iterations is 5.5.

5-bit channel LLR values, 6-bit extrinsic LLR values; 9-bit branch metrics ($\gamma$), and 10-bit state metrics ($\alpha$ and $\beta$). Fig. 4.16 shows the fixed-point simulation results for both HSPA+ and LTE turbo decoder using the proposed architecture and quantization. The simulation results show that the fixed-point performance degradation is less than 0.1 dB.

To achieve the throughput goal, the turbo decoder consists of $P = 16$ Radix-4 XMAP decoders, which results in an effective parallelism of 64 ($P_{LLR} = P \times 4 = 64$). Thus, the extrinsic LLR memory is partitioned into 64 modules. Each extrinsic LLR memory module is implemented using a $(K/P_{LLR} \times B_{ext})$ two-port
SRAM ($B_{ext}$ is the word length of the extrinsic LLR). Moreover, 16 channel LLR memory modules are used to store the input channel LLR values. Each channel LLR memory module is a $(3K/P \times 2B_{ch})$ single-port SRAM ($B_{ch}$ is the word length of the channel LLR). Every channel LLR memory module is further partitioned into several banks to provide parallel accesses to the systematic and parity check values. To support the high throughput of LTE mode, all 16 SISO decoders work actively during decoding, with the corresponding 64 extrinsic LLR memory modules, 16 channel LLR memory modules, and 64 QPP interleavers. The HSPA+ interleaver and DBCF buffer architecture will be bypassed under LTE mode. On the other hand, when configured to work under the HSPA+ mode, the turbo decoder uses 8 SISO decoders leading to an effective parallelism of $P_{LLR} = 32$. Accordingly, 32 extrinsic LLR memory modules, 8 channel LLR memory modules, and 32 HSPA+ interleavers are employed. One DBCF buffer structure per extrinsic LLR memory module is implemented to handle the memory conflicts. The remaining 8 SISO decoders along with memory modules are disabled using HSPA+ mode.

The multi-standard HSPA+/LTE turbo decoder, which is described using Verilog HDL, has been synthesized, placed and routed with the TSMC 45nm CMOS technology. In this section, we will describe the implementation results in detail and comparisons with related work.

4.7.1 Implementation Results for the Contention-free HSPA+ Interleaver
Table 4.4: Comparison with related work on UMTS/HSPA+ interleaver. (Note: the chip area in this table includes area for IAG and interconnection-network, except for the first three publications.)

<table>
<thead>
<tr>
<th>Publication</th>
<th>Supported parallelism</th>
<th>Block size configurable</th>
<th>Interleaving mode</th>
<th>Technology</th>
<th>Result type</th>
<th>Area ($Area_{Norm}{^a}$) $[mm^2]$</th>
<th>Frequency [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[103]</td>
<td>16</td>
<td>N/A^b</td>
<td>N/A^b</td>
<td>180 nm</td>
<td>Synthesis</td>
<td>8.91 (0.254)</td>
<td>166</td>
</tr>
<tr>
<td>[118]</td>
<td>32</td>
<td>N/A^b</td>
<td>N/A^b</td>
<td>130 nm</td>
<td>Synthesis</td>
<td>2.784 (0.087)</td>
<td>200</td>
</tr>
<tr>
<td>[121]</td>
<td>16</td>
<td>N/A^b</td>
<td>N/A^b</td>
<td>180 nm</td>
<td>Synthesis</td>
<td>1.2 (0.034)</td>
<td>200</td>
</tr>
<tr>
<td>[102]</td>
<td>16</td>
<td>No</td>
<td>Interleaver/deinterleaver</td>
<td>40 nm</td>
<td>Synthesis</td>
<td>0.15 (0.086)</td>
<td>350</td>
</tr>
<tr>
<td>[111]</td>
<td>2</td>
<td>Yes</td>
<td>Interleaver</td>
<td>65 nm</td>
<td>P&amp;R^c</td>
<td>0.014 (0.027)</td>
<td>150</td>
</tr>
<tr>
<td>[113]</td>
<td>1</td>
<td>Yes</td>
<td>Interleaver</td>
<td>180 nm</td>
<td>Synthesis</td>
<td>0.24 (0.110)</td>
<td>130</td>
</tr>
<tr>
<td>[128]</td>
<td>1</td>
<td>Yes</td>
<td>Interleaver</td>
<td>250 nm</td>
<td>P&amp;R^c</td>
<td>2.69 (0.489)</td>
<td>N/A</td>
</tr>
<tr>
<td>[101]</td>
<td>1</td>
<td>Yes</td>
<td>Interleaver</td>
<td>130 nm</td>
<td>P&amp;R^c</td>
<td>0.4 (0.4)</td>
<td>246</td>
</tr>
<tr>
<td>[137]</td>
<td>1</td>
<td>Yes</td>
<td>Interleaver/deinterleaver</td>
<td>FPGA (90 nm)</td>
<td>Synthesis</td>
<td>0.23^a (0.229)</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>This work</strong></td>
<td>32 or higher</td>
<td>Yes</td>
<td>Interleaver/deinterleaver</td>
<td>45 nm</td>
<td>P&amp;R^c</td>
<td>0.43 (0.121)</td>
<td>600</td>
</tr>
</tbody>
</table>

^a Area is scaled to 130nm assuming $Area \sim 1/s^2$ ($s$ is feature size of technology nodes) and is normalized to equivalent area per LLR value.

^b These papers only present the interconnection-network to solve memory conflicts. The IAG is not included/reported in the paper.

^c “P&R” indicates that the reported results are after place and route.

^d The chip area was not reported in the paper. The chip area is estimated using the gate count reported in the paper.
The HSPA+ interleaver is the most critical part in the proposed multi-standard turbo decoder design. To achieve up to 672 Mbps data rate under HSPA+ mode, the turbo decoder employs an effective parallelism of $P_{LLR} = 32$, which can result in severe memory conflicts as indicated by Fig. 4.10. Therefore, we first present implementation results for the HSPA+ interleaver.

The HSPA+ interleaver is implemented based on the balanced scheduling scheme, the DBCF buffer architecture and the unified parallel interleaver/deinterleaver architecture described in the previous sections. The block size $K$ can be configured at runtime. When a new block size $K$ is set, the preprocessing unit re-calculates the runtime parameters. The interleaver can work at both interleaver and deinterleaver modes based on a mode selection signal.

As a reference design, we first implemented a single-function IAG which only supports the interleaving algorithm. The chip area is 0.249 $mm^2$ when synthesized with the TSMC 45nm CMOS technology and targeting 600 MHz clock frequency. Then we implemented a dual-function IAG supporting both the interleaving and deinterleaving algorithms of the HSPA+ standard. The area for the unified IAG is 0.278 $mm^2$. Thus, compared to the single-function IAG, the chip area of the dual-function IAG is only increased by 0.029 $mm^2$, which indicates a 11.6% complexity overhead compared to a single-function IAG. This result has demonstrated that the proposed unified parallel IAG architecture enables great hardware sharing, leading to an efficient hardware implementation. Inside the unified parallel IAG, the preprocessing logic utilizes 806 $\mu m^2$ chip area, and the runtime computing logic utilizes 0.097 $mm^2$ to generate 32 parallel interleaved (or deinterleaved) addresses in one clock cycle. The unified HSPA+ IAG uses 65 Kb memory as runtime parameter storage to support parallelism degree of 32 and different block sizes.
Table 4.5: VLSI implementation comparison with existing high speed UMTS/HSPA+ and multi-mode turbo decoders.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SISO decoder</td>
<td>Radix-4 XMAP</td>
<td>Radix-1 MAP</td>
<td>Radix-2 MAP</td>
<td>Radix-2 MAP</td>
<td>Radix-4 MAP</td>
<td>Radix-2 MAP</td>
<td>Radix-4 MAP</td>
</tr>
<tr>
<td>$N_{SISO}$</td>
<td>16/64</td>
<td>1/1</td>
<td>4/4</td>
<td>1/1</td>
<td>4/8</td>
<td>1/16</td>
<td>5/10</td>
</tr>
<tr>
<td>Block size K [bits]</td>
<td>5114/6144</td>
<td>5114</td>
<td>5114/4800</td>
<td>5114</td>
<td>5114/6144</td>
<td>5114/6144</td>
<td>5114/6144/4800</td>
</tr>
<tr>
<td>Technology</td>
<td>45 nm</td>
<td>180 nm</td>
<td>130 nm</td>
<td>130 nm</td>
<td>65 nm</td>
<td>40 nm</td>
<td>130 nm</td>
</tr>
<tr>
<td>Supply Voltage $V_{DD}$</td>
<td>0.81 V</td>
<td>1.8 V</td>
<td>N/A</td>
<td>1.2 V</td>
<td>1.1 V</td>
<td>1.1 V</td>
<td>1.2 V</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>600 MHz</td>
<td>111 MHz</td>
<td>200 MHz</td>
<td>246 MHz</td>
<td>285 MHz</td>
<td>350 MHz</td>
<td>125 MHz</td>
</tr>
<tr>
<td>Core area</td>
<td>2.43 mm$^2$</td>
<td>9.0 mm$^2$</td>
<td>N/A</td>
<td>1.2 mm$^2$</td>
<td>0.65 mm$^2$</td>
<td>1.46 mm$^2$</td>
<td>6.4 mm$^2$</td>
</tr>
<tr>
<td>Gate count</td>
<td>1470 K</td>
<td>34.4 K</td>
<td>204 K</td>
<td>44.1 K</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory</td>
<td>550 Kb</td>
<td>201 Kb</td>
<td>148.6 Kb</td>
<td>120 Kb</td>
<td>214 Kb</td>
<td>160 Kb</td>
<td></td>
</tr>
<tr>
<td>Max N iteration</td>
<td>5.5</td>
<td>6</td>
<td>8</td>
<td>5.5</td>
<td>6</td>
<td>6.5</td>
<td>5</td>
</tr>
<tr>
<td>Throughput</td>
<td>826 Mbps (HSPA+)</td>
<td>4.1 Mbps (W-CDMA)</td>
<td>12 Mbps (UMTS)</td>
<td>20.2 Mbps (HSDPA)</td>
<td>49.5 Mbps (HSPA+)</td>
<td>300 Mbps (HSDPA/LTE)</td>
<td>105.6 Mbps (LTE)</td>
</tr>
<tr>
<td>Power consumption</td>
<td>870 mW</td>
<td>292 mW</td>
<td>N/A</td>
<td>61.5 mW</td>
<td>570 mW</td>
<td>452 mW</td>
<td>374.75 mW</td>
</tr>
</tbody>
</table>

- a The W-CDMA, HSDPA and HSPA+ are terms referring to different generations of technologies used in the 3GPP UMTS standard. They all use the same turbo code structure and interleaving algorithms.

- b $N_{LLR}$ denotes the number of parallel LLR values produced per clock cycle, which represents the effective parallelism of a turbo decoder.

- c Technology is scaled to 130nm CMOS assuming: $Area \sim 1/s^2$, and $t_{pd}$ (propagation delay) $\sim 1/s$. ($s$: feature size of a technology node). Throughput is linearly scaled to 5.5 iterations.

- d Normalized area efficiency = $Area_{Norm}/Throughput_{Norm}$ [108–110, 138].

- e Architecture efficiency = $Throughput \times N_{iter}/Area_{Norm}/f_{clk}$ [114].
Table 4.4 summarizes the key features of the implemented contention-free HSPA+ interleaver, as well as the comparison with related work. All the work shown in Table 4.4 focus on the design of high performance and efficient interleavers for the 3GPP UMTS/HSPA+ turbo decoder. Among them, references [102, 103, 111, 118, 121] concentrate on the interconnection-network solving the memory conflict problem, so they did not report the area for the IAG modules. While the others present efficient solutions for IAG modules for the UMTS/HSPA+. To make a fair comparison, we show the normalized chip area for each implementation; furthermore, for those supporting parallel interleaving address generation, the chip area is further normalized by the supported parallelism to determine an equivalent chip area for each generated interleaving address. The results in [101, 113, 128, 137] support on-the-fly interleaving address generation for HSPA+, but they do not support parallel decoding due to the lack of a memory conflict solver. The interleavers in [102, 103] are not reconfigurable because they support only one block size. Only [102, 137] and our work support both the interleaver and deinterleaver modes. The architecture in [111] can support a parallel turbo decoder with different block sizes, but the architecture is limited to turbo decoders with low parallelism. Our proposed architecture shows the best tradeoff among flexibility, configurability, and hardware complexity.

4.7.2 Implementation Results of Multi-Standard Turbo Decoder

The key characteristics of the implemented multi-standard turbo decoder are summarized in Table 4.5. The design is synthesized using the TSMC 45nm standard cell library. The chip core area of the implemented turbo decoder is 2.43 $mm^2$. A top-level chip layout view after place and route is shown in Fig. 4.17. The maximum number of iterations of this turbo decoder is set to 5.5. The block size $K$ can
Figure 4.17: VLSI chip layout for an HSPA+/LTE multi-standard turbo decoder with the architecture shown in Fig. 4.15. The TSMC 45nm standard cell library is used. This turbo decoder contains 16 Radix-4 XMAP decoders, 64 LTE QPP interleavers, and 32 HSPA+ interleavers.
be configured at runtime ($K = 40 \sim 5114$ for HSPA+; $K = 40 \sim 6144$ for LTE). When clocked at 600 MHz, a maximum throughput of 1.67 Gbps can be achieved under the LTE mode with all 16 Radix-4 XMAP decoders running. While in the HSPA+ mode, a maximum throughput of 826 Mbps can be achieved with 8 Radix-4 XMAP decoders active. Compared to a single-mode LTE turbo decoder, the area overhead of the HSPA+/LTE multi-standard turbo decoder is approximately 17% in our implementation ($0.426 mm^2 / 2.43 mm^2$).

To the best of our knowledge, our multi-standard turbo decoder implementation is the first to support the peak data rate of 336 Mbps in Release 11 of the 3GPP UMTS/HSPA+ standard [12]. Moreover, our implementation also exceeds the 672 Mbps peak data rate requirement of the potential future extension of the HSPA+ standard and the 1 Gbps peak data rate requirement for the LTE-Advanced standard [13]. A comparison with related work can be found in Table 4.5. These related publications are either high speed UMTS/HSPA+ turbo decoder designs or multi-standard turbo decoders supporting the UMTS/HSPA+ standard. Implementation details such as clock frequency, chip core area, throughput, and power consumption are included. We adopt three normalized metrics to evaluate implementation efficiency: area efficiency, energy efficiency and architecture efficiency [108–110, 114, 138]. The normalization is done by scaling the reported throughput and area numbers to 130nm CMOS technology. From Table 4.5 and Fig. 4.18, we can see that our implementation shows the best normalized energy efficiency among the publications under both HSPA+ and LTE mode. In HSPA+ mode, our implementation shows the best area (or architecture) efficiency. While for LTE mode, our implementation has better normalized area (or architecture) efficiency than most of papers listed in Table 4.5. Note that only [111] shows better area (or architecture) efficiency. However, it has lower energy efficiency than the proposed implementation. Moreover, the parallel inter-
Figure 4.18: Hardware efficiency (energy/area) comparison with related work.
leaver architecture proposed by [111] only supports low parallelism for the HSPA+ mode, which limits its achievable throughput under HSPA+ mode.

4.8 GPU Implementation of Parallel Turbo Decoding

We implemented a parallel Turbo decoder on GPU. Instead of spawning one thread-block per codeword to perform decoding, a codeword is split into $P$ sub-blocks and decoded in parallel using multiple thread-blocks. The algorithm described in Section 3 maps very efficiently onto an SM since the algorithm is very data parallel. As the number of trellis states is 8 for the 3GPP compliant Turbo code, the data parallelism of this algorithm is 8. However, the minimum number of threads within a warp instruction is 32. Therefore, to reduce horizontal waste, we allow each thread-block to process a total of 16 sub-blocks from 16 codewords simultaneously. Therefore the number of threads per thread-block is 128, which enables four fully occupied warp instructions. As $P$ sub-blocks may not be enough to keep all the SMs busy, we also decode $N$ codewords simultaneously to minimize the amount of horizontal waste due to idling cores. We spawn a total of $\frac{NP}{16}$ thread-blocks to handle the decoding workload for $N$ codewords. Figure 4.19 shows how threads are partitioned to handle the workload for $N$ codewords.

In our implementation, the inputs of the decoder, LLRs from the channel, are copied from the host memory to device memory. At runtime, each group of 8 threads within a thread-block generates output LLRs given the input for a codeword sub-block. Each iteration consists of a pass through the two MAP decoders. Since each half iteration of the MAP decoding algorithm performs the same sequence of computations, both halves of an iteration can be handled by a single MAP decoder kernel.
Figure 4.19: To decode $N$ codewords, we divide each codeword into $P$ sub-blocks. Each thread-block has 128 threads and handles 16 codeword sub-blocks.

After a half decoding iteration, thread-blocks are synchronized by writing extrinsic LLRs to device memory and terminating the kernel. In the device memory, we allocate memory for both extrinsic LLRs from the first half iteration and extrinsic LLRs from the second half iteration. For example, the first half iteration reads a priori LLRs and writes extrinsic LLRs interleaved. The second half iteration reads a priori LLRs and writes extrinsic LLRs deinterleaved. Since interleaving and deinterleaving permute the input memory addresses, device memory access becomes random. In our implementation, we prefer sequential reads and random writes over random reads and sequential writes as device memory writes are non-blocking. This increases efficiency as the kernel does not need to wait for device memory writes to complete to proceed. One single kernel can handle input and output reconfiguration easily with a couple of simple conditional reads and writes at the beginning and the end of the kernel.

In our kernel, we need to recover performance loss due to edge effects as the decoding workload is partitioned across multiple thread-blocks. Although a sliding window algorithm with training sequence can be used to improve the BER performance of the decoder, it is not implemented. The next iteration initialization technique im-
proves the error correction performance with much smaller overhead. In this method, the $\alpha$ and $\beta$ values between neighboring thread-blocks are exchanged through device memory between iterations.

The CUDA architecture can be viewed as a specific realization of a multi-core SIMD processor. As a result, although the implementation is optimized specifically for Nvidia GPUs, the general strategy can be adapted for other many-core architectures with vector extensions. However, many other vector extensions such as SSE and Altivec do not support transcendental functions which lead to greater throughput difference between max-log-MAP and full-log-MAP implementations.

The implementation details of the reconfigurable MAP kernel are described in the following subsections.

### 4.8.1 Shared Memory Allocation

If we partition a codeword with $K$ information bits into $P$ partitions, we need to compute $\frac{K}{P}$ stages of $\alpha$ before we can compute $\beta$. If we attempt to cache the immediate values in shared memory, per partition, we will need to store $\frac{8K}{P}$ floats in shared memory. As we need to minimize vertical waste by decoding multiple codewords per thread-block, the amount of shared memory is quadrupled to pack 4 codewords into a thread-block to match the width of a *warp* instruction. Since we only have 48KB of shared memory which is divided among concurrent thread-blocks on an SM, we will not be able to have many concurrent threads if $P$ is small. For example, if $K = 6144$ and $P = 32$, the amount of shared memory required by $\alpha$ is 24KB. The number of concurrent threads is only 64, leading to vertical waste as we cannot hide the pipeline latency with concurrent running blocks. We can reduce the amount of shared memory used by decreasing $P$. This, however, can reduce error correction performance. Therefore, we need a better strategy for managing shared memory.
instead of relying on increasing \( P \).

Instead of storing all \( \alpha \) values in shared memory, we can spill \( \alpha \) into device memory each time we compute a new \( \alpha \). We only store one stage of \( \alpha \) during the forward traversal. For example, suppose \( \alpha_{k-1} \) is in shared memory. After calculating \( \alpha_{k} \) using \( \alpha_{k-1} \), we store \( \alpha_{k} \) in device memory and replace \( \alpha_{k-1} \) with \( \alpha_{k} \). During LLR computation, when we need \( \alpha \) to compute \( \Lambda_{k}(s_{k}|u_{b} = 0) \) and \( \Lambda_{k}(s_{k}|u_{b} = 1) \), we fetch \( \alpha \) directly into registers. Similarly, we store one stage of \( \beta_{k} \) values during the backward traversal. Therefore, we do not need to store \( \beta \) into device memory. In order to increase thread utilization during extrinsic LLR computation, we save up to 8 stages of \( \Lambda_{k}(s_{k}|u_{b} = 0) \) and 8 stages of \( \Lambda_{k}(s_{k}|u_{b} = 1) \). We can reuse shared memory used for LLR computation, \( \alpha \), and \( \beta \). Therefore, the total amount of shared memory per thread-block, packing 16 codewords per thread-block, is 2048 floats or 8KB. This allows us to have 768 threads running concurrently on an SM while providing fast memory access most of the time.

### 4.8.2 Forward Traversal

During the forward traversal, eight cooperating threads decode one codeword sub-block. The eight cooperating threads traverse through the trellis in locked-step to compute \( \alpha \). There is one thread per trellis level, where the \( j^{th} \) thread evaluates two incoming paths and updates \( \alpha_{k}(s_{j}) \) for the current trellis stage using \( \alpha_{k-1} \), the forward metrics from the previous trellis stage \( k - 1 \). The computation of \( \alpha_{k}(s_{j}) \), however, depends on the path taken \( (s_{k-1}, s_{k}) \). The two incoming paths are known \text{a priori} since the connections are defined by the trellis structure as we shown before. Table 4.6 summarizes the operands needed for \( \alpha \) computation.

The indices of the \( \alpha_{k} \) are stored as a constant. Each thread loads the indices and the values \( p_{k}|u_{b} = 0 \) and \( p_{k}|u_{b} = 1 \) at the start of the kernel. The pseudo-code for one
Table 4.6: Operands for $\alpha_k$ computation.

<table>
<thead>
<tr>
<th>Thread id ($i$)</th>
<th>$u_b = 0$</th>
<th>$u_b = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$s_{k-1}$</td>
<td>$p_k$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

iteration of $\alpha_k$ computation is shown in Algorithm 8. The memory access pattern

**Algorithm 8** thread $i$ computes $\alpha_k(i)$

\[
a_0 \leftarrow \alpha_k(s_{k-1}|u_b = 0) + L_c(y_k^s) \ast (p_k|u_b = 0) \\
a_1 \leftarrow \alpha_k(s_{k-1}|u_b = 1) + (L_c(y_k^s) + L_a(k)) \\
\quad + L_c(p_k^s)(p_k|u_b = 1) \\
\alpha_k(i) = \max^*(a_0, a_1) \\
\text{write } \alpha_k(i) \text{ to device memory} \\
\text{SYNC}
\]

is very regular for the forward traversal. Threads access values of $\alpha_k$ in different memory banks. There are no shared memory conflicts in either case, that is memory reads and writes are handled efficiently by shared memory.

### 4.8.3 Backward Traversal and LLR Computation

After the forward traversal, each thread-block traverses through the trellis backward to compute $\beta$. We assign one thread to each trellis level to compute $\beta$, followed by computing $\Lambda_0$ and $\Lambda_1$ as shown in Algorithm 9. The indices of $\beta_k$ and values of $p_k$ are summarized in Table 4.7. Similar to the forward traversal, there are no shared memory bank conflicts since each thread accesses an element of $\alpha$ or $\beta$ in a different bank.
Table 4.7: Operands for $\beta_k$ computation

<table>
<thead>
<tr>
<th>Thread id ($i$)</th>
<th>$s_{k+1}$</th>
<th>$p_k$</th>
<th>$s_{k+1}$</th>
<th>$p_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

**Algorithm 9** thread $i$ computes $\beta_k(i)$ and $\Lambda_0(i)$ and $\Lambda_1(i)$

Fetch $\alpha_k(i)$ from device memory

$b_0 \leftarrow \beta_{k+1}(s_{k+1}|u_b = 0) + L_c(y^*_k) \cdot (p_k|u_b = 0)$

$b_1 \leftarrow \beta_{k+1}(s_{k+1}|u_b = 1) + (L_c(y^*_k) + L_a(k)) + L_c(p^*_k)(p_k|u_b = 1)$

$\beta_k(i) = \max^*(b_0, b_1)$

SYNC

$\Lambda_0(i) = \alpha_k(i) + L_p(i)p_k + \beta_{k+1}(i)$

$\Lambda_1(i) = \alpha_k(i) + (L_c(k) + L_a(k)) + L_p(s_k)p_k + \beta_k(i)$
After computing $\Lambda_0$ and $\Lambda_1$ for stage $k$, we can compute the extrinsic LLR for stage $k$. However, there are 8 threads available to compute the single LLR, which introduces parallelism overhead. Instead of computing one extrinsic LLR for stage $k$ as soon as the decoder computes $\beta_k$, we allow the threads to traverse through the trellis and save 8 stages of $\Lambda_0$ and $\Lambda_1$ before performing extrinsic LLR computations. By saving eight stages of $\Lambda_0$ and $\Lambda_1$, we allow all 8 threads to compute LLRs in parallel efficiently. Each thread handles one stage of $\Lambda_0$ and $\Lambda_1$ to compute an LLR. Although this increases thread utilization, threads need to avoid accessing the same bank when computing an extrinsic LLR. For example, 8 elements of $\Lambda_0$ for each stage are stored in 8 consecutive addresses. Since there are 16 memory banks, elements of even stages $\Lambda_0$ or $\Lambda_1$ with the same index would share the same memory bank. Likewise, this is true for even stages of $\Lambda_0$. Hence, sequential accesses to $\Lambda_0$ or $\Lambda_1$ to compute an extrinsic LLR result in four-way memory bank conflicts. To alleviate this problem, we permute the access pattern based on thread ID as shown in Algorithm 10.

\begin{algorithm}
\caption{thread $i$ computes $L_e(i)$}
\begin{algorithmic}
\State $\lambda_0 = \Lambda_0(i)$
\State $\lambda_1 = \Lambda_1(i)$
\For{$j = 1$ to $7$}
\State $index = (i + j) \& 7$
\State $\lambda_0 = \max^*(\lambda_0, \Lambda_0(index))$
\State $\lambda_1 = \max^*(\lambda_1, \Lambda_1(index))$
\State $L_e = \lambda_1 - \lambda_0$
\State Compute write address
\State Write $L_e$ to device memory
\EndFor
\end{algorithmic}
\end{algorithm}

### 4.8.4 Early Termination Scheme

Depending on SNR, a Turbo decoder requires a variable number of iterations to achieve satisfactory BER performance. In the mid and high SNR regime, the Turbo
decoding algorithm usually converges to the correct codeword with a small numbers of decoding iterations. Therefore, fixing the number of decoding iterations is inefficient. Early termination schemes are widely used to accelerate the decoding process while maintaining a given BER performance. As the average number of decoding iterations is reduced by using an early termination scheme, early termination can also reduce power consumption. As such, early termination schemes are widely used in low power high performance Turbo decoder designs.

There are several major approaches to implement early termination: hard-decision rules, soft-decision rules, CRC-based rules and other hybrid rules. Hard-decision rules and soft-decision rules are the most popular early termination schemes due to low complexity. Compared to hard-decision rules, soft-decision rules provide better error correcting performance than other low complexity early termination algorithms. Therefore, we implement two soft-decision early termination schemes: minimum LLR threshold scheme and average LLR threshold scheme.

The stop condition of the minimum LLR scheme can be expressed by:

$$\min_{1 \leq i \leq N} |LLR_i| \geq T, \quad (4.16)$$

in which we compare the minimum LLR value with a pre-set threshold $T$ at the end of each iteration. If the minimum LLR value is greater than the threshold, then the iterative decoding process is terminated.

The stop condition of the average LLR scheme can be represented by:

$$\frac{1}{N} \sum_{1 \leq i \leq N} |LLR_i| \geq T. \quad (4.17)$$

where $N$ is the block length of the codeword and $T$ is the pre-set threshold.

The simulation results show that for multi-codeword parallel Turbo decoding, the
variation among minimum LLR values for different codewords is very large. Since each
thread-block decodes 16 sub-blocks from 16 codewords simultaneously, we can only
terminate the thread-block if all 16 codewords meet the early termination criteria.
Therefore, the minimum LLR values are not accurate metrics for early termination
for our implementation. The average LLR value is more stable so that the average
LLR scheme is implemented for this parallel Turbo decoder. As mentioned in the
previous sub-section, during the backward traversal and LLR computation process,
eight stages of \( \Lambda_0 \) and \( \Lambda_1 \) are saved in the memory. After \( \Lambda_0 \) and \( \Lambda_1 \) are known,
the eight threads are able to compute LLRs using these saved \( \Lambda_0 \) and \( \Lambda_1 \) values in
parallel. Therefore, to compute the average LLR value in one codeword, each thread
can track the sum of the LLRs when going through the whole trellis. In the end of the
backward traversal, we combine all 8 sums of LLRs and compute the average LLR
value of the codeword. Finally, this average LLR value is compared with a pre-set
threshold to determine whether the early termination condition is met. The detailed
algorithm is described in Algorithm 11.

Algorithm 11 Early termination scheme for thread \( i \)

\begin{algorithm}
\begin{algorithmic}
\State Compute the codeword ID \( C_{id} \)
\If{\text{tag}[C_{id}] == 1}
\State Terminate the thread \( i \)
\EndIf
\EndAlgorithm
\end{algorithm}
\end{algorithm}

Another challenge is that it is difficult to wait for hundreds of codewords to con-
verge simultaneously and terminate the decoding process for all codewords at the
same time. Therefore, a tag-based scheme is employed. Once a codeword meets the
early termination condition, the corresponding tag is marked and this codeword will 
not be further processed in the later iterations. After all the tags are marked, we 
stop the iterative decoding process for all the codewords. By using a tag-based early 
termination scheme, the decoding throughput can be significantly increased in the 
mid and the high SNR regime.

4.8.5 Interleaver

An interleaver is used between the two half decoding iterations. Given an input 
address, the interleaver provides an interleaved address. This interleaves and dein-
terleaves memory writes. In our implementation, a quadratic permutation polyno-
mial (QPP) interleaver, which is proposed in the 3GPP LTE standards was used. 
The QPP interleaver guarantees bank free memory accesses, where each sub-block 
accesses a different memory bank. Although this is useful in an ASIC design, the 
QPP interleaver is very memory I/O intensive for a GPU as the memory write access 
pattern is still random. As inputs are stored in device memory, random accesses 
result in non-coalesced memory writes. With a sufficient number of threads running 
concurrently on an SM, we can amortize the performance loss due to device memory 
accesses through fast thread switching. The QPP interleaver is defined as:

\[ \Pi(x) = f_1 x + f_2 x^2 \pmod{N}. \]  

(4.18)

The interleaver address, \( \Pi(x) \), can be computed on-the-fly using Equation (4.18). 
However, direct computation can cause overflow. For example, \( 6143^2 \) can not be 
represented as a 32-bit integer. Therefore, the following equation is used to compute 
\( \Pi(x) \) instead:

\[ \Pi(x) = (f_1 + f_2 x \pmod{N}) \cdot x \pmod{N} \]  

(4.19)
Another way of computing $\Pi(x)$ is recursively, which requires $\Pi(x)$ to be computed before we can compute $\Pi(x + 1)$. This is not efficient for our design as we need to compute several interleaved addresses in parallel. For example, during the second half of the iteration to store extrinsic LLR values, 8 threads need to compute 8 interleaved addresses in parallel. Equation (4.19) allows efficient address computation in parallel.

Although our decoder is configured for the 3GPP LTE standard, one can replace the current interleaver function with another function to support other standards. Furthermore, we can define multiple interleavers and switch between them on-the-fly since the interleaver is defined in software in our GPU implementation.

4.8.6 max* Function

We support the Full-log-MAP algorithm as well as the Max-log-MAP algorithm. Full-log-MAP is defined as:

$$\max^*(a, b) = \max(a, b) + \ln(1 + e^{-|b-a|}).$$  \hspace{1cm} (4.20)

The complexity of the computation can be reduced by assuming that the second term is small. Max-log-MAP is defined as:

$$\max^*(a, b) = \max(a, b).$$  \hspace{1cm} (4.21)

As was the case with the interleaver, we can compute $max^*(a, b)$ directly. We support Full-log-MAP as both natural logarithm and natural exponential are supported on CUDA. However, logarithm and natural exponentials take longer to execute on the GPU compared to common floating operations, e.g. multiply and add. Therefore we expect throughput loss compared to Max-log-Map.
Table 4.8: Peak throughput for the $K = 6144$, rate-1/3 LTE turbo code.

<table>
<thead>
<tr>
<th>$I$</th>
<th>max-log-MAP</th>
<th>linear log-MAP</th>
<th>max-log-MAP</th>
<th>log-MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>122.6</td>
<td>62.3</td>
<td>54.2</td>
<td>55.2</td>
</tr>
<tr>
<td>6</td>
<td>76.2</td>
<td>40.0</td>
<td>37.0</td>
<td>35.5</td>
</tr>
<tr>
<td>8</td>
<td>55.6</td>
<td>30.8</td>
<td>30.0</td>
<td>27.8</td>
</tr>
<tr>
<td>10</td>
<td>46.9</td>
<td>25.1</td>
<td>24.9</td>
<td>23.7</td>
</tr>
</tbody>
</table>

4.8.7 Experimental Results

To evaluate the throughput of our Turbo decoder, we tested our Turbo decoder on a Windows 7 platform an Intel Core i7-3770K, a quad core Ivy Bridge processor, and a Nvidia GTX 680, a Kepler GK104 GPU. In our experiments, we used the Intel C++ Compiler 14.0 and the CUDA 5.5 toolkit. In the following, we compare the throughput of our optimized turbo decoder implementations.

Table 4.8 shows the peak throughput of our CPU and GPU implementations. The results for HSPA+ are similar; for log-MAP algorithm and 6 decoding iterations, we achieved 41.6 Mbps and 36.7 Mbps on CPU and GPU respectively.

As expected, the throughput for both implementation is inversely proportional to the number of decoding iterations $I$. The CPU implementation appears to be instruction bounded as additional instructions increase the runtime proportionally. The number of instructions required to compute a forward state metric is 7 using the max-log approximation, whereas the use of the linear log-MAP approximation requires 6 additional instructions. As a result, the number of instructions required for linear log-MAP is $2 \times$ of max-log-MAP. For 6 decoding iterations, the throughput of the linear log-MAP approximation is 40 Mbps. The throughput of the linear log-MAP approximation is approximately $2 \times$ lower than the throughput of max-log-MAP implementation, which is 76.2 Mbps.

For the GPU implementation, we found the instructions per cycle (IPC) is low, typically $\sim 1$. Using the Nvidia profiler, we found that instructions operands are often
unavailable (due to memory latency), leading to a low execution unit utilization. As a result, additional instructions (that do not require additional memory access) can execute on the idling execution units and thus do not significantly degrade the throughput. Hence, the throughput of log-MAP is not significantly slower than that of the max-log-MAP algorithm on the GPU.

For the CPU implementation, a workload of 8 parallel codewords is sufficient to reach the peak throughput. For the GPU implementation, significantly more codewords need to be processed in parallel. Given a codeword, increasing the number of windows, $P$, does not increase peak throughput as the number of computation and memory transactions required to process the codeword stays the same. Increasing $P$, however, is an effective way of reducing the number of codewords required to reach peak throughput. On the Kepler architecture, we require 512 codewords for $P = 1$, 16 codewords for $P = 32$ and above, to reach the peak performance.

In summary, our GPU implementation is up to $2\times$ slower than that of our CPU implementation for the max-log approximation, and only $1.2\times$ slower for the optimal log-MAP algorithm. For the Nvidia Kepler architecture, the maximum IPC is 6-to-7, while the achieved IPC is $\sim 1$. The achieved IPC is low as operands of the instructions are often not ready due to memory access latency, leading to low execution unit utilization and low throughput. Coupled by the fact that CPU is clocked much faster than the GPU, the CPU implementation was able to outperform the GPU implementation.

We emphasize that it is possible to further improve the decoding throughput on the GPU. For example, we can reduce the number of memory accesses via data compression, which reduces the time for which execution units wait for operands. Nevertheless, the CPU implementation seems to be better suited for SDR applications since we achieve higher throughput with $2\times$ fewer number of parallel codewords,
which significantly reduces the latency of the entire system.

4.9 Design Space Exploration of High Performance Turbo decoder Design

Performance Requirements

The performance requirements for turbo decoding for two major wireless communication standards are summarized in Table 4.9. It is quite challenging to design turbo decoders to meet these performance requirements, but also provide very flexible and configurable implementations to enable high speed wireless connectivity under different cases.

<table>
<thead>
<tr>
<th></th>
<th>3GPP UMTS/HSPA+</th>
<th>3GPP LTE/LTE-Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>40–5114</td>
<td>40–6144</td>
</tr>
<tr>
<td>Interleaver</td>
<td>Pseudo-random</td>
<td>QPP interleaver</td>
</tr>
<tr>
<td>Code rates</td>
<td>1/3</td>
<td>1/3</td>
</tr>
<tr>
<td>Performance requirements</td>
<td>336Mbps (Rel-11)</td>
<td>Up to 1 Gbps</td>
</tr>
<tr>
<td></td>
<td>672Mbps (expected in Rel-12)</td>
<td></td>
</tr>
</tbody>
</table>

Algorithmic Complexity

The main computational complexity lies in the MAP decoder, which contains 2 $\gamma$ branch-metric compute units (BMCs), $\alpha$ state-metric compute units (SMCs), $\beta$ state-metric compute units (SMCs) and 2 LLR compute units (LLRCs).

Table 4.10 lists the complexity for each BMC, SMC and LLRC. Please note that we use radix-4 MAP decoding algorithm, therefore, all compute units here implement radix-4 computations. In each $\alpha$ or $\beta$ SMC, 8 add-compare-select units (ACSUs) are
included. In each LLRC unit, 32 adders are used to add $\alpha$ and $\beta$ state metrics, followed by 4 Max8 units (selecting a maximum from 8 input values), 8 compare-select-add units (CSAUs), and 2 three-input adders to compute the final extrinsic LLRs.

Table 4.10: Computation complexity of components in MAP decoder per clock cycle.

<table>
<thead>
<tr>
<th>Compute unit</th>
<th>Number</th>
<th>Complexity/unit</th>
<th># of operations/unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma$ BMC</td>
<td>2</td>
<td>64 2-input adders</td>
<td>$64 \times 2 = 128$</td>
</tr>
<tr>
<td>$\alpha$ SMC</td>
<td>1</td>
<td>8 ACSUs</td>
<td>80</td>
</tr>
<tr>
<td>$\beta$ SMC</td>
<td>1</td>
<td>8 ACSUs</td>
<td>80</td>
</tr>
<tr>
<td>LLRC</td>
<td>2</td>
<td>32 2-input adders, 4 Max8 units, 8 CSAUs, 2 3-input adders</td>
<td>$86 \times 2 = 172$</td>
</tr>
<tr>
<td>Total OPs</td>
<td>–</td>
<td>–</td>
<td>460 operations</td>
</tr>
</tbody>
</table>

According to Table 4.10, for each computation stage, there are totally 460 operations ($N_{OP/MAP/CLK} = 460$) per MAP decoder. We can calculate the computation complexity measured in GOPS\(^1\) as follows:

$$\text{Complexity} = N_{OP/MAP/CLK} \cdot f_{clk} \cdot P. \quad (4.22)$$

For instance, with 8 MAP decoders active in parallel turbo decoders for HSPA+ mode, to meet 672 Mbps throughput requirement, computation capability of 2122.4 GOPS is required. While, to achieve 1 Gbps throughput for LTE, a turbo decoder containing 16 active MAP decoders needs computation capability of 3114 GOPS. Therefore, turbo decoding is always considered to be one of the most computationally-intensive modules in the wireless communication modem chipset.

\(^1\)GOPS: Giga operations per second.
Throughput Performance Model

We first define some notations for parallel turbo decoder in Table 4.11.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>Block size</td>
</tr>
<tr>
<td>$P$</td>
<td>Parallelism. Number of parallel MAP decoders.</td>
</tr>
<tr>
<td>$f_{clk}$</td>
<td>Clock frequency</td>
</tr>
<tr>
<td>$N_{iter}$</td>
<td>Number of decoding iterations</td>
</tr>
<tr>
<td>$L_{MAP}$</td>
<td>MAP decoder latency</td>
</tr>
<tr>
<td>$L_{interleaver}$</td>
<td>Latency of interleaver</td>
</tr>
</tbody>
</table>

The overall decoding latency can be expressed as:

$$L_{total} = \left( \frac{K}{2P} + L_{MAP} + L_{interleaver} \right) \times 2N_{iter} \quad (4.23)$$

The decoding throughput can be modeled as:

$$T = \frac{K \cdot f_{clk}}{2N_{iter} \cdot \left( \frac{K}{2P} + L_{MAP} + L_{interleaver} \right)} \quad (4.24)$$

As we can see from (4.24), as parallelism $P$ goes higher the throughput increases accordingly. However, in a practical implementation, there is certain limitation to $P$. As a block code, turbo decoding requires training process as the decoder traverses the trellis, so when parallelism $P$ becomes bigger, the size of sub-block becomes smaller, leading to a significantly degradation in error-correction performance.

Memory Access Pattern

Due to the existence of interleaver, half of memory accesses to the extrinsic memory are random and irregular, which may become the bottleneck of the whole turbo
decoder. In each clock cycle, one data fetching and one data writing to the extrinsic memory will be performed. Assume the parallelism is $P$, the peak memory bandwidth requirement is $4P$ memory reads and $4P$ memory writes per clock cycles for a turbo decoder with architecture described in Chapter 4. The required memory read/write bandwidth requirement can be computed as:

$$BW_{mem} = f_{clk} \cdot N_{mem\_access/cycle}$$

$$= \frac{T \cdot 2N_{iter} \cdot \left( \frac{K}{2P} + L_{MAP} + L_{interleaver} \right) \cdot N_{mem\_access/cycle}}{K},$$

(4.25)

in which, $N_{mem\_access/cycle}$ represents the number of memory accesses per cycle per MAP decoder. $N_{mem\_access/cycle} = 4$ in our proposed architecture for both memory reading and writing.

We have summarized the memory bandwidth requirements for HSPA+ and LTE-Advanced modes for typical data rates in Table 4.12. For example, to achieve the required more than 1 Gbps data rate for LTE standards, based on the performance model in (4.24), the minimum clock frequency should be above 423 MHz, if we use a turbo decoder with $P = 16$ and $N_{iter} = 6.5$. The extrinsic LLR message is represented using 6bit fixed-point representation, therefore the read (or write) bandwidth requirements for the extrinsic memory system can be estimated as 162.5 Gb/s. If all data are represented using floating-point representations, the memory bandwidth requirement becomes 866.7 Gb/s (or 108.3 GB/s).
Table 4.12: Memory bandwidth for different throughput requirements for turbo decoding with different degrees of parallelism. Assume 6-bit fix-point data representation is used.

<table>
<thead>
<tr>
<th>Standards</th>
<th>HSPA+</th>
<th>LTE-Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>337 Mbps</td>
<td>1 Gbps</td>
</tr>
<tr>
<td></td>
<td>672 Mbps</td>
<td>1.5 Gbps</td>
</tr>
<tr>
<td></td>
<td>1 Gbps</td>
<td>3 Gbps</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P = 8$</td>
<td>9.3</td>
<td>26.5</td>
</tr>
<tr>
<td></td>
<td>18.5</td>
<td>39.8</td>
</tr>
<tr>
<td></td>
<td>26.5</td>
<td>79.6</td>
</tr>
<tr>
<td>$P = 16$</td>
<td>9.7</td>
<td>27.1</td>
</tr>
<tr>
<td></td>
<td>19.4</td>
<td>40.6</td>
</tr>
<tr>
<td></td>
<td>27.1</td>
<td>81.3</td>
</tr>
<tr>
<td>$P = 32$</td>
<td>10.7</td>
<td>28.2</td>
</tr>
<tr>
<td></td>
<td>21.4</td>
<td>42.3</td>
</tr>
<tr>
<td></td>
<td>28.2</td>
<td>84.5</td>
</tr>
</tbody>
</table>

We use $B_{ch}$, $B_{sm}$ and $B_{ext}$ to represent the bit width of channel input LLR, state metric, and extrinsic LLR messages. If sliding window algorithm is employed, we define the window size as $W$.

Table 4.13: Memory usage for parallel turbo decoding.

<table>
<thead>
<tr>
<th>Message</th>
<th>Memory type</th>
<th>Memory usage (per MAP decoder)</th>
<th>Memory usage (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systematic channel LLR</td>
<td>single-port SRAM</td>
<td>$N/P \cdot 2B_{ch}$</td>
<td>$P \cdot 2B_{ch}$</td>
</tr>
<tr>
<td>Parity channel LLR 1</td>
<td>single-port SRAM</td>
<td>$N/P \cdot 2B_{ch}$</td>
<td>$P \cdot 2B_{ch}$</td>
</tr>
<tr>
<td>Parity channel LLR 2</td>
<td>single-port SRAM</td>
<td>$N/P \cdot 2B_{ch}$</td>
<td>$P \cdot 2B_{ch}$</td>
</tr>
<tr>
<td>State metric ($\alpha$, $\beta$, and $\gamma$)</td>
<td>single-port SRAM</td>
<td>$(W/4) \cdot (2 \times 8 \times B_{sm})$</td>
<td>$P \cdot (W/4) \cdot (2 \times 8 \times B_{sm})$</td>
</tr>
<tr>
<td>Extrinsic LLR</td>
<td>dual-port SRAM</td>
<td>$(N/P) \cdot B_{ext}$</td>
<td>$N \cdot B_{ext}$</td>
</tr>
</tbody>
</table>
Table 4.14: Flexibility requirements for a reconfigurable turbo decoder.

<table>
<thead>
<tr>
<th>Item</th>
<th>Flexibility requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standards</td>
<td>UMTS/HSPA+, LTE, LTE-Advanced, WiMAX etc.</td>
</tr>
<tr>
<td>Block sizes</td>
<td>A wide range of valid block sizes. (e.g. 40–6144 in LTE)</td>
</tr>
<tr>
<td>MAP algorithm</td>
<td>Radix-2, Radix-4 and higher.</td>
</tr>
<tr>
<td></td>
<td>Sliding window, non-sliding window</td>
</tr>
<tr>
<td></td>
<td>Double-flow, single-flow</td>
</tr>
<tr>
<td>Memory architecture</td>
<td>sub-banking, no sub-banking</td>
</tr>
<tr>
<td>Interleaver algorithms</td>
<td>pseudo-random, QPP, APR etc.</td>
</tr>
<tr>
<td>Interleaver architecture</td>
<td>on-the-fly address generation</td>
</tr>
<tr>
<td></td>
<td>Interleaver and deinterleaver algorithm.</td>
</tr>
</tbody>
</table>

**Flexibility and Configurability**

It is desirable to design flexible turbo decoders supporting different system configurations and parameter scenarios to reduce the design and production cost. Regarding to turbo decoder, this becomes especially important, because there are so many variations of the decoding algorithms, parallel decoding schemes, numerous parameter combinations, and even future extensions. Table 4.14 summarizes some of the flexibility requirements to the turbo decoder design.

**Performance Comparison between ASIC and GPU Implementations**

Table 4.15 compares the ASIC implementation of a multi-standard turbo decoder proposed in Chapter 4 and a GPU implementation from one of our previous work [44].

As we have expected, the ASIC implementation outperforms the GPU implementation significantly, given the very high computation complexity and very irregular memory accesses. However, there are some benefits from the GPU implementations. Firstly, the GPU implementation provides very flexible design. We could easily change...
Table 4.15: Comparison between ASIC and GPU implementations of turbo decoding.

<table>
<thead>
<tr>
<th></th>
<th>ASIC implementation [37]</th>
<th>GPU implementation [44]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Customized logic</td>
<td>NVIDIA GTX 690</td>
</tr>
<tr>
<td>Technology</td>
<td>TSMC 45nm</td>
<td>TSMC 28nm</td>
</tr>
<tr>
<td>Clock freq</td>
<td>600 MHz</td>
<td>915 MHz</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>N/A</td>
<td>2810.88</td>
</tr>
<tr>
<td>Memory</td>
<td>SRAM</td>
<td>2GB GDDR5 memory + On-chip memory</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>N/A</td>
<td>192.2 GB/s</td>
</tr>
<tr>
<td>Language</td>
<td>Verilog HDL</td>
<td>C++, CUDA C</td>
</tr>
<tr>
<td>Mode/standard</td>
<td>HSPA+/LTE</td>
<td>HSPA+/LTE (can be easily extended)</td>
</tr>
<tr>
<td>MAP algorithm</td>
<td>max-log-MAP</td>
<td>max-log-MAP, log-MAP (configurable)</td>
</tr>
<tr>
<td>Block size</td>
<td>Configurable 40–6144</td>
<td>Configurable 40–6144</td>
</tr>
<tr>
<td>Parallelism</td>
<td>8 or 16</td>
<td>Configurable</td>
</tr>
<tr>
<td>Interleaver</td>
<td>On-the-fly computation</td>
<td>On-the-fly computation + lookup table</td>
</tr>
<tr>
<td>Data type</td>
<td>Fixed-point</td>
<td>Floating-point</td>
</tr>
<tr>
<td>$N_{iter}$</td>
<td>fixed 6.5</td>
<td>Configurable</td>
</tr>
<tr>
<td>Decoding method</td>
<td>Streaming, one codeword at a time</td>
<td>Multi-codeword decoding.</td>
</tr>
<tr>
<td>Design effort</td>
<td>Several months</td>
<td>2–3 weeks</td>
</tr>
<tr>
<td>Throughput</td>
<td>826 Mbps @6.5iter (HSPA+)</td>
<td>37 Mbps @6.5iter (Both HSPA+ and LTE)</td>
</tr>
<tr>
<td></td>
<td>1672 Mbps @6.5iter (LTE)</td>
<td></td>
</tr>
</tbody>
</table>

decoding algorithms (max-log-MAP, log-MAP, or others) with only slight changes to the code. The support of different block sizes, number of iterations and parallelisms can be easily modified via parameters; while this is not the case in the ASIC design. The GPU implementation can easily plug in any new interleaving algorithms, while the customized hardware solution will suffer from implementing the new interleaving algorithms. Moreover, the floating-point implementation of turbo decoder can provide some performance gain compared to the fixed-point implementation in hardware. In addition, the design efforts are much more when comparing the hardware design to the software-based GPU implementation.

Regarding to the impact of memory access pattern to performance, it is heavily depends on the system architectures. In the half iterations when the MAP decoders access the extrinsic data in a natural order from the extrinsic memory, the memory access pattern is very regular and shows good spatial locality. However, during the
half iterations with interleaving/deinterleaving memory addresses, the memory access behavior becomes very irregular. First of all, the data requested by MAP decoders are distributed across the whole memory address range. Either for a customized hardware implementation, or for a multi-core processor-based software implementation, this random shuffling behavior makes it difficult to arrange the data in the memory to take advantage of spacial locality. Secondly, the randomness of the interleaving and deinterleaving causes severe memory bank conflicts, which requires serialized data accesses to the memory. Therefore, the performance is significantly reduced. For a customized hardware implementation, we can implement customized buffer systems to eliminate the memory writing conflicts, so that the memory system still provide enough memory bandwidth. As a comparison, for a software-based implementation targeting multi-core processors or GPGPUs, the existing memory/cache systems are designed to handle memory accesses with temporal or spacial locality. Manipulating and re-arranging data layout in such systems are quite expensive. Especially, in turbo decoding, in each half iterations, a same memory location is only read and written once, respectively. There is no benefit to move the data around, and the overhead will kill the performance improvement.

Although the GPU implementation has so many advantages over the hardware implementation, to meet the high throughput requirements from the 3G/4G wireless communication standards, we have no other choice but to use the customized hardware design. In addition, the huge power consumption of the GPUs also makes GPU-based turbo decoders impractical for a mobile system.

4.9.1 Analysis and Discussion

The turbo decoding algorithm has low parallelism. The feasible number of MAP decoders is limited by the minimum BER performance requirement. The parallelism
we can exploit is around several hundred. For ASIC implementations, this is a good parallelism degree. However, for GPU implementation, this parallelism is too low to fully occupy GPU’s computation resources even when we consider introducing another level of parallelism: multi-codeword parallel decoding. Assume the number of codewords decoded in parallel is $N_{\text{codeword}}$. Number of subblocks is denoted as $P$. Number of trellis statuses is denoted as $N_s$. The total parallelism (PL) can be computed as:

$$PL = N_{\text{codeword}} \cdot P \cdot N_s.$$  \hspace{1cm} (4.26)

Considering the parallel turbo decoding with 32 MAP decoders working in parallel and 8 parallel processing units working on 8-state trellis in parallel. As described in [44], in order to keep all cores busy, 16 codewords are decoded in parallel. Therefore, the resultant parallelism is 4096 ($P_L = 32 \times 8 \times 16 = 4096$).

In a GTX-690 GPU core, there are 1536 CUDA cores in hardware. Based on CUDA architecture, if there is read-after-write dependency for a register data, 24 clock cycles are needed until the data is available [75]. Therefore, to keep the pipeline busy, at least $1536 \times 24$ threads are required. In addition, the multiprocessor in GTX-690 contains 192 CUDA cores, which requires significantly more thread blocks to keep the cores active [75]. This means that we need at least 2–3 times more threads in practice. In summary, we need at least $1536 \times 24 = 37328$ threads to fully occupy the GPU’s computation resources, and achieve peak performance of the GPU device.

Given a GPU’s architecture specification and the properties of the turbo decoding algorithm, we should be able to roughly estimate the achievable throughput. So far, we have the parallelism of the GPU implementation ($P_L = 4096$), the theoretical parallelism to achieve peak performance ($P_{L_{\text{peak}}} = 73728$), the arithmetic complexity requirement ($C_A = 3114$ GFLOPS) to achieve targeted 1 Gbps data rate, and the peak
Figure 4.20: High bandwidth requirement of turbo decoding algorithm, due to the irregular memory access pattern.

performance of the GPU ($Pr_{peak} = 2810$ GFLOPS). We can predict the achievable data rate for the GPU-based turbo decoding:

\[
T_{achievable} = \frac{PL}{PL_{peak}} \frac{Pr_{peak}}{C_A \cdot T}.
\]

\[
= \frac{4096}{73728} \times 2810/3114 \times 1Gbps
\]

\[
= 50 \text{ Mbps}
\]  

(4.27)

As we can see, the predicted achievable throughput (50 Mbps) is close to the experimental result (37 Mbps). With limited parallelism, the GPU implementation of turbo decoding can hardly fully utilize GPU’s computational resources. In addition, turbo decoding also suffers from very irregular memory accesses, which is not friendly for a massively parallel architecture. As shown in Figure 4.20. Based on the memory bandwidth requirement, we can use our proposed methodology to analyze the algorithm bottleneck, as shown in Figure 4.21.

Therefore, turbo decoding is not suitable for GPU acceleration. As a comparison, with ASIC architecture, we can implement customized buffer systems to eliminate the memory conflict problem and relieve the irregular memory access pressure. Therefore,
Figure 4.21: Analysis of turbo decoding algorithm with the proposed methodology.

to achieve the targeted throughput goal such as 1 Gbps for LTE standard, the ASIC implementation should be employed.

4.10 Summary

In this chapter, we discuss the turbo decoding algorithms. Even with a lot of optimization techniques applied, the GPU-implementation still cannot meet the performance requirements (not even close). Therefore, we understand that GPU is not always the good candidate for an accelerator. For algorithms like turbo decoding, which lacks parallelism and shows very irregular memory access pattern which leads to extremely high memory bandwidth requirement, we cannot use GPU to implement the algorithm to achieve the desirable performance. In this case, the ASIC implementation are necessary to meet all requirements. we propose a VLSI architecture for highly parallel
turbo decoding aiming at multi-standard 3G/4G wireless communication systems. We first utilize the balanced scheduling scheme to avoid memory reading conflicts. To demonstrate the effectiveness of our design approaches, we synthesize, place and route an ASIC design for the HSPA+/LTE/LTE-Advanced multi-standard turbo decoder using the proposed VLSI architecture with a 45nm CMOS technology. The implementation results show that as an enabler, the proposed parallel interleaver architecture is able to lead to highly parallel but also hardware-efficient turbo decoding implementations.
Case Study 3: Computer Vision Accelerator using GPU on Mobile Devices

Mobile devices have evolved rapidly and become ubiquitous over the past decade, giving rise to new application demands through the convergence of mobile computing, wireless communication and digital imaging technologies. On one hand, as mobile processors have become more and more powerful and versatile during the past several years, we are witnessing a rapid growth in the demand for the computer vision applications running on mobile devices, such as image editing, augmented reality, object recognition and so on [26, 139–145]. On the other hand, with the recent advances in the fields of computer vision and augmented reality, the emerging algorithms have become more complex and computationally-intensive. Therefore, the long processing time due to the high computational complexity prevents these computer vision algorithms from being practically used in mobile applications.

To address this problem, researchers have been exploring general-purpose computing using graphics processing units (GPGPUs) as accelerators to speed up the image processing and computer vision algorithms thanks to the heterogeneous architecture of the modern mobile processors [28, 46, 146–152]. On desktop computers or super-
computers, numerous programming models have been extensively studied and utilized to facilitate the parallel GPGPU programming, such as the Compute Unified Device Architecture (CUDA) [153] and the Open Computing Language (OpenCL) [76, 154]. As a comparison, due to the lack of parallel programming models in the mobile domain, the OpenGL ES (Embedded System) programming model was commonly used to harness the computing power of the mobile GPU [155]. However, the inherent limitations of the OpenGL ES lead to poor flexibility and scalability, as well as limited parallel performance, due to the fact that the OpenGL ES was originally designed for 3D graphics rendering. Recently, emerging programming models such as the OpenCL embedded profile [154] and the RenderScript [156] have been supported by the state-of-the-art mobile processors, making the mobile GPGPU feasible for real-world mobile devices for the first time [47, 48, 145].
Figure 5.2: OpenCL programming model and hierarchical memory architecture.

5.1 GPGPU on Mobile Devices

As is shown in Fig. 5.1, a modern mobile SoC (system-on-chip) chipset typically consists of a multi-core mobile CPU, a mobile GPU with multiple programmable shaders, and an image signal processor (ISP) [157–159]. Unlike their desktop counterparts, the mobile CPU and GPU share the same system memory via a system bus. The mobile platforms also contain a variety of sensors and accelerators. Modern mobile platforms tend to employ heterogeneous architectures, which integrate several application-specific co-processors to enable the computationally intensive algorithms such as face detection and so on. However, the space limitation and the power constraints of the mobile devices limit the number of integrated hardware co-processors. It is preferable to seek the general-purpose computing power inside the mobile processor. The mobile GPUs are suitable candidates to accelerate computationally intensive tasks due to their highly parallel architecture.

The lack of good parallel programming models becomes an obstacle to perform general-purpose computing on the mobile GPUs. As a compromise, researchers have
been using the OpenGL ES programming model for GPGPU to achieve performance improvement and energy efficiency on mobile devices during the past decade. For instance, Singhal et al. implemented and optimized an image processing toolkit on handheld GPUs [147]. Nah et al. proposed an OpenGL ES-based implementation of ray tracing, called MobiRT, and studied the CPU-GPU hybrid architecture [160]. Ensor et al. presented GPU-based image analysis on mobile devices, in which the Canny edge detection algorithm was implemented using the OpenGL ES [148]. Researchers have also attempted to accelerate feature detection and extraction using the mobile GPUs [46, 143, 149, 152]. Recently, performance characterization and energy efficiency for mobile CPU-GPU heterogeneous computing have been studied [28, 151].

Thanks to the unified programmable shader architecture and the emerging parallel programming frameworks such as OpenCL, the new generation of mobile GPUs have gained real general-purpose parallel computing capability. The OpenCL is a programming framework designed for heterogeneous computing across various platforms [154]. Fig. 5.2 shows the programming and the hierarchical memory architecture of OpenCL. In OpenCL, a host processor (typically a CPU) manages the OpenCL context and is able to offload parallel tasks to several compute devices (for instance, GPUs). The parallel jobs can be divided into work groups, and each of them consists of many work items which are the basic processing units to execute a kernel in parallel. OpenCL defines a hierarchical memory model containing a large off-chip global memory but with long latency of several hundred clock cycles, and a small but fast on-chip local memory which can be shared by work items in the same work group. To efficiently and fully utilize the limited computation resources on a mobile processor to achieve high performance, partitioning the tasks between CPU and GPU, exploring the algorithmic parallelism, and optimizing the memory access need to be carefully considered. Few prior works have studied the methodology of using OpenCL to pro-
gram mobile GPUs and achieve speedup. Leskela et al. demonstrated a prototype of OpenCL Embedded Profile emulated by OpenGL ES on mobile devices and showed advantages in performance and energy efficiency [146]. In our previous work, we have explored the mobile GPGPU capability of mobile processors to accelerate computer vision algorithms such as an image editing algorithm (object removal), and SIFT-based feature extraction [47, 48]. Performance improvement and energy consumption reduction have been observed.

With the increasing popularity of digital technology and embedded devices such as digital cameras, smartphones, and tablets during the past several years, the demand for creating, editing, consuming and sharing multimedia contents such as pictures, audios and videos are growing drastically. New sensors such as image sensors, depth sensors, temperature sensors and so on with the help of more powerful processors are enabling many new applications such as stereo imaging, virtual reality, and augmented reality. The core of all those new applications are image processing and computer vision algorithms.

Among these advanced algorithms, image editing and local feature extraction are two important classes of algorithms, which form the basis of many other powerful algorithms. In this Section, we briefly introduce exemplar-based image inpainting for object removal algorithm, and the scale-invariant feature transform (SIFT) algorithms.

The object removal algorithm and SIFT algorithm require various types of workloads and computation kernels such as raw pixel manipulation, block-level image patch search, iterative image processing technique, sum of squared difference (SSD), histogram computation, and gradient orientation calculations. Therefore, they can represent a class of computer vision algorithms, such as image stitching, object recognition, motion estimation, texture analysis and synthesis, and so on. Therefore, by
Figure 5.3: An example of the object removal algorithm. The mask image indicates the object to be removed from the original image.

studying and evaluating the performance of these algorithms on mobile devices with CPU-GPU partitioning, the feasibility and advantages of using the mobile GPU as a co-processor can be demonstrated. Furthermore, the optimization techniques proposed in this paper can possibly be applied to other computer vision algorithms with similar operation patterns or algorithm workflows.

5.2 Parallel Implementation of Exemplar-based Object Removal Algorithm

5.2.1 Overview of Object Removal Algorithm

Object removal is one of the most important image editing functions. As is shown in Fig. 5.3, the key idea of object removal is to fill in the hole that is left behind after removing an unwanted object, to generate a visually plausible result image. The exemplar-based inpainting algorithm for object removal can preserve both structural and textural information by replicating patches in a best-first order, which can generate good image quality for object removal applications [161, 162]. In the meanwhile, this algorithm can achieve computational efficiency thanks to the block-based sampling processing, which is especially attractive for a parallel implementation.
The major steps of the exemplar-based inpainting algorithm for object removal proposed by Criminisi et al. is depicted in Fig. 5.4 [161]. Assume we have a source image $S$ with a target region $\Omega$ to be filled in after an object is removed (the empty region). The left image region is denoted as $\Phi$ ($\Phi = S - \Omega$). The border of the object region is denoted as $\delta \Omega$. The image patches are filled into the object region $\Omega$ one by one based on priority values $C(p)$. Given an image patch $\Psi_p$ centered at pixel $p$ for $p \in \delta \Omega$, the priority value $C(p)$ is defined as the product of two terms:

$$P(p) = R(p) \cdot D(p),$$  \hspace{1cm} (5.1)

in which $R(p)$ is the confidence term indicating the amount of reliable information surrounding the pixel $p$, and $D(p)$ is the data term representing the strength of texture and structural information along the edge of the object region $\delta \Omega$ in each iteration. $R(p)$ and $D(p)$ are defined as follows:
\[ C(p) = \frac{\sum_{q \in \Psi_p \cap \Omega} C(q)}{|\Psi_p|}, \]
\[ D(p) = \frac{\nabla I_p^\perp \cdot n_p}{\alpha}. \]  

(5.2)

where \(|\Psi_p|\) is the area of \(\Psi_p\), \(\alpha\) is a normalization factor (for a typical grey-level image, \(\alpha = 255\)), and \(n_p\) is a unit vector orthogonal to \(\delta \Omega\) in the point \(p\).

According to the priority values for all patches across the border \(\delta \Omega\) of the target region, we select a candidate patch with the maximum priority value. Then, we search the image region \(\Phi\) and find a patch \(\Psi_{\bar{q}}\) that best matches the patch \(\Psi_p\) (we call this step findBestPatch). The goal of findBestPatch is to find the best matching patch \(\Psi_{\bar{q}}\) from candidate patches \(\Psi_q\) in the source image region \(\Phi\), to match an object patch \(\Psi_p\) in the object region \(\Omega\) based on a certain distance metric. The sum of squared differences (SSD) is typically used as a distance metric to measure the similarity between the patches [161]. We denote the color value of a pixel \(x\) by \(I_x = (R_x, G_x, B_x)\). For an object patch \(\Psi_p\), the best patch \(\Psi_{\bar{q}}\) is chosen by computing:

\[ \Psi_{\bar{q}} = \arg \min_{q \in \Phi} d(\Psi_p, \Psi_q), \]  

(5.3)

in which \(d(\Psi_q, \Psi_p)\) is defined as follows:

\[ d(\Psi_q, \Psi_p) = \sum_{p \in \Psi_p \cap \Psi_q \cap \Phi} (I_p - I_q)^2. \]  

(5.4)

Assume that the size of the original image is \(M \times N\), and the size of the patch is \(P \times P\). The complexity of findBestPatch can be estimated as \(O(MNP^2)\). Based on our profiling (will be shown in Section 4), findBestPatch compute kernel occupies the most computations in the whole object removal algorithm.
Figure 5.5: Algorithm workflow of the exemplar-based object removal algorithm. Please note that one OpenCL block might be mapped to one or multiple OpenCL kernels depending on the computation and memory data access patterns.

Once the best matching patch is found, we copy the pixel values of $\Psi_q$ into $\Psi_p$. The aforementioned search and copy process is repeated until the whole target region $\Omega$ is filled up. More details of the algorithm can be found in reference [161].

5.2.2 Algorithm Workflow Analysis and CPU-GPU Partitioning

In this section, we analyze the workflow of the object removal algorithms and describe the algorithm partitioning between the CPU and GPU to fully utilize the resources of the mobile SoC chipset.

5.2.2.1 Experimental Setup

The profiling and experiments are performed on a development platform consisting of a Qualcomm Snapdragon 8974 chipset [157], which supports OpenCL Embedded Profile for both CPU and GPU. The details of the experimental setup are listed in
Table 5.1: Specification of the experimental setup.

<table>
<thead>
<tr>
<th>Mobile SoC</th>
<th>Snapdragon 8974</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Krait 400 Quad-core</td>
</tr>
<tr>
<td>Max clock frequency</td>
<td>2.15 GHz</td>
</tr>
<tr>
<td>Compute units</td>
<td>4</td>
</tr>
<tr>
<td>Local memory</td>
<td>32 KB/compute unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Adreno 330</td>
</tr>
<tr>
<td>Max clock frequency</td>
<td>400 MHz</td>
</tr>
<tr>
<td>Compute units</td>
<td>4</td>
</tr>
<tr>
<td>Local memory</td>
<td>8 KB/compute unit</td>
</tr>
<tr>
<td>GPU performance</td>
<td>130 GFLOPS</td>
</tr>
<tr>
<td>GPU memory bandwidth</td>
<td>12.8 GB/s</td>
</tr>
<tr>
<td>Operating system</td>
<td>Android 4.2.2</td>
</tr>
<tr>
<td>Development toolset</td>
<td>Android NDK r9</td>
</tr>
<tr>
<td>Instruction set</td>
<td>ARM-v7a</td>
</tr>
</tbody>
</table>

Table 5.1.

5.2.2.2 Algorithm Mapping

Fig. 5.5 shows a workflow diagram of the exemplar-based inpainting algorithm for object removal. The algorithm can be partitioned into three stages: initialization stage, iterative computation stage, and the finalization stage. The blocks with the slashed lines are core functions inside the iterative stage and represent most of the computational workload. We can map the core functions into OpenCL kernels to exploit the 2-dimensional pixel-level and block-level parallelisms in the algorithms. The CPU handles the OpenCL context initialization, memory objects management, and kernel launching. By analyzing the algorithm, we partition the core functions into eight OpenCL kernels based on the properties of the computations, as is shown in Table 5.2. In each OpenCL kernel, the fact that no dependency exists among image blocks allows us to naturally partition the tasks into work groups. To represent color pixel values in RGBA (red green blue alpha) format, we use efficient vector data structures such as `cl_uchar4` to take advantage of built-in vector functions of
Table 5.2: Breakdown of execution time for OpenCL kernel functions running only on CPU.

<table>
<thead>
<tr>
<th>Kernel functions</th>
<th>Exec time [s]</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert RGB image to gray-scale image</td>
<td>0.08</td>
<td>0.09%</td>
</tr>
<tr>
<td>Update border of the area to be filled</td>
<td>0.60</td>
<td>0.69%</td>
</tr>
<tr>
<td>Mark source pixels to be used</td>
<td>0.66</td>
<td>0.76%</td>
</tr>
<tr>
<td>Update pixel priorities in the filling area</td>
<td>0.45</td>
<td>0.52%</td>
</tr>
<tr>
<td>Update pixel confidence in the filling area</td>
<td>0.36</td>
<td>0.41%</td>
</tr>
<tr>
<td><strong>Find the best matching patch</strong></td>
<td><strong>84.4</strong></td>
<td><strong>97.0%</strong></td>
</tr>
<tr>
<td>Update RGB and grayscale image of the filling patch</td>
<td>0.46</td>
<td>0.53%</td>
</tr>
<tr>
<td><strong>Total Time</strong></td>
<td><strong>87.0</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

OpenCL.

To better optimize the OpenCL-based implementation, we first measure the timing performance of the OpenCL kernels. Table 5.2 shows a breakdown of processing time when running the program on a single core of the CPU on our test device. The OpenCL kernel function used to find the best matching patch with the current patch (denoted as findBestPatch) occupies most of the processing time (97%), so optimizing the findBestPatch kernel becomes the key to improving performance.

5.2.3 Algorithm Optimizations and Implementation Trade-offs

5.2.3.1 OpenCL Implementation of findBestPatch Kernel Function

The algorithm mapping of findBestPatch kernel used for the OpenCL implementation is shown in Fig. 5.6. To perform a full search for the best patch $\Psi_q$ to match the current filling patch $\Psi_p$ in the findBestPatch OpenCL kernel, we spawn $M \times N$ work items, with each computing an SSD value between two $P \times P$ patches. We partition these $M \times N$ work items into work groups according to the compute capability of the
Figure 5.6: Algorithm mapping of *findBestPatch* kernel function using OpenCL.

Table 5.3: Description of images in the test dataset. These images are selected to represent different types of image scenes and different shapes of objects to be removed.

<table>
<thead>
<tr>
<th>Image</th>
<th>Image type</th>
<th>Image size</th>
<th>Object type</th>
<th>Object size</th>
</tr>
</thead>
<tbody>
<tr>
<td>WalkingMan</td>
<td>Complex scene</td>
<td>512 × 384</td>
<td>Blob</td>
<td>78 × 126</td>
</tr>
<tr>
<td>River</td>
<td>Complex scene, texture</td>
<td>480 × 639</td>
<td>Small blob</td>
<td>59 × 93</td>
</tr>
<tr>
<td>DivingMan</td>
<td>Texture</td>
<td>576 × 384</td>
<td>Big/random shape</td>
<td>155 × 186</td>
</tr>
<tr>
<td>Hill</td>
<td>Complex scene</td>
<td>1024 × 550</td>
<td>Long strip</td>
<td>1024 × 10</td>
</tr>
</tbody>
</table>
Figure 5.7: The best patch mapping found by a full search and the result images. The 1st row: original images. The 2nd row: masks covering the unwanted objects. The 3rd row: best patch mapping; the small squares indicate the best patches found by the `findBestPatch()` function. The 4th row: result images.
Algorithm 12 Compute SSD values between all candidate image patches and the image patch to be filled using an OpenCL kernel function.

0: **Input:**
1: Position of object patch $\Psi_p$: $(p_x, p_y)$;
2: Patch size $P$;
3: **Output:** SSD array $ssdArray$;
4: **Begin OpenCL kernel:**
5: Get global ID for the current work item: $(i, j)$;
6: float $sum = 0.0$;
7: int $src_x, src_y$; // source pixel position
8: int $tgt_x, tgt_y$; // target pixel position
9: int $wsize = P/2$;
10: for (int $h = -wsize; h \leq wsize; h += )
11: for (int $w = -wsize; w \leq wsize; w += )
12: $src_x = i + w; src_y = j + h$;
13: $tgt_x = p_x + w; tgt_y = p_y + h$;
14: if (($src_x, src_y$) or $(tgt_x, tgt_y$) is out of image)
15: continue;
16: end if
17: if(pixel $(tgt_x, tgt_y$) is inside source region $\Phi$)
18: Read pixel $(tgt_x, tgt_y$) data into $tgtData$;
19: Read pixel $(src_x, src_y$) data into $srcData$;
20: $sum += (tgtData - srcData)^2$;
21: end if
22: end for
23: end for
24: Store $sum$ into $ssdArray$;
25: **End OpenCL kernel**

GPU. The size of 2-dimensional work groups can be expressed as

$$([M/localSize.x],[N/localSize.y]).$$ (5.5)

In our implementation, each work group contains $8 \times 8$ work items ($localSize.x=8$, $localSize.y=8$). Therefore, each work group of work items perform SSD computations for 64 patch candidates. The parallel implementation of the SSD computation in the findBestPatch kernel function is detailed in Algorithm 12.
5.2.3.2 Experimental Dataset

To demonstrate the best patch matching behavior of the object removal algorithm, we employ several different test images to cover different scenarios, which are shown in the first row of Fig. 5.7. The characteristics of these test images are summarized in Table 5.3. We chose images with different background scenes and textures, with variable image sizes and object sizes, and with different object shapes, so that by performing experiments on these images, we can better understand the implementation trade-offs related to the performance improvement.

5.2.3.3 Reducing Search Space

We have done experiments to verify the locations of the best patches found by a full search across the whole image area. For the test images shown in Fig. 5.7, most of the best patches are found in a region surrounding the object area. The reason is that adjacent areas usually have similar structures and textures in natural images. To reduce the searching time, we can utilize this spatial locality by limiting the search
space. To better model this optimizing strategy in a scalable manner, we define a search factor $\alpha$. Assume the width and height of the object area are $w$ and $h$ respectively. The new search area is formed by expanding the object area by $\alpha h$ to the up and down directions, and $\alpha w$ to the left and right directions, as is shown in Fig. 5.8. The search area factor $\alpha$ has a range of $0 \leq \alpha < \max (M/h, N/w)$. Assume the object region is centered at coordinate $(o_x, o_y)$. Fig. 5.8 shows a typical case in which the whole search area is inside the image area. For more general cases, the boundary of the new search area becomes:

$$
\begin{align*}
B_{left} &= \max(0, o_x - \frac{1}{2} + \alpha w), \\
B_{right} &= \min(N, o_x + \frac{1}{2} + \alpha w), \\
B_{top} &= \max(0, o_x - \frac{1}{2} + \alpha h), \\
B_{bottom} &= \min(M, o_x + \frac{1}{2} + \alpha h).
\end{align*}
$$

By defining the search factor $\alpha$ this way, we can easily adjust the search area. Moreover, this method allows the search area to grow along four directions with an equal chance, so as to increase the possibility of finding a better patch. Since there are no useful pixels in the object area for patch matching, only the candidate patches not in the object region will be compared with the object patch. So the actual size of the search area ($SA$) can be expressed as:

$$
SA = (2\alpha + 1)w \cdot (2\alpha + 1)h - wh = ((2\alpha + 1)^2 - 1)wh.
$$
The complexity of \textit{findBestPatch} can be estimated by $O(((2\alpha+1)^2 - 1)whP^2)$. Thus, when we reduce the search area (reducing $\alpha$), the complexity to search the best patch reduces significantly.

Fig. 5.9 demonstrates the effect of reducing the search factor $\alpha$. After reducing the search factor, the best matching patches are limited to a small region around the object region. Even when the search factor is reduced significantly to only $\alpha = 0.05$, we still get visually plausible results. Due to its importance, choosing a proper search factor $\alpha$ is critical for practical applications to achieve good performance and efficiency. Based on our experiments, for a regular object region, choosing parameter $\alpha$ in the range of $0.05 \leq \alpha < 0.5$ normally leads to good implementation trade-offs. If the algorithm is to be applied to a certain type of images, trainings on datasets can be performed to determine a proper search factor $\alpha$.

In addition to time reduction, reducing the search area can also reduce the possible false matching. As a comparison metric, SSD can roughly represent the similarity of two patches, but it cannot accurately reflect the structural and color information embedded in the patches. Therefore, the patches with the highest distance scores (SSD in this algorithm) may not be the best patches to fill in the hole and to generate visually plausible results due to the limitation of SSD metric, especially for complex scenes with different color information and structures. The algorithm sometimes can lead to false matching, in which the reported “best” patch with a high correlation score may have very distinctive textural or color information compared to the object patch. Under such circumstances, the artificial effects introduced by the false matching will degrade the quality of the result images significantly. Fortunately, spatial locality can be observed in most of the natural images, therefore, the visually plausible matching patches (in terms of color and texture similarity) tend to reside in the surrounding area of the candidate patch with high chances. By reducing the search area to a
Figure 5.9: The effect of reducing the search area. The search area factor $\alpha$ is defined as in Fig. 5.8.
certain degree, we can reduce the possibility of false matching and therefore generate visually plausible result images.

5.2.3.4 Optimizing Patch Size

The object removal algorithm is an iterative algorithm, in which one object patch is processed in each iteration. We need roughly $\frac{wh}{P^2}$ iterations to finish the whole process. That said, if we increase patch size $P$, fewer iterations are needed to fill the object area which may lead to shorter processing time. Meanwhile, the complexity of the SSD computation ($O(P^2)$) becomes higher for the patch matching, which tends to increase the processing time. Therefore, it is not straightforward to determine the impact of increasing patch size $P$ to the overall complexity and performance.

We use Fig. 5.10 to help us analyze the overall computation complexity. First of all, we assume the search factor $\alpha$ is defined as in the previous section. We also define the search area as $SA$ as in (5.7).
Figure 5.11: Impact of increased patch size $13 \times 13$ and $17 \times 17$. “WalkingMan” test image. The processing time for $13 \times 13$ and $17 \times 17$ patches is normalized by the time of the $9 \times 9$ patch. (The $9 \times 9$ patch size is suggested by the original algorithm proposed by Criminisi et al.).

Secondly, because the patch size is $P$, any candidate patch within $(P - 1)$ pixels range surrounding the object area $\Omega$ will partially overlap with the object area. We define this overlapping area as $OA$, whose area can be calculated as:

$$OA = (2(P - 1) + w) \cdot (2(P - 1) + h) - wh.$$  \hspace{1cm} (5.8)

If the candidate patch lies outside the overlapping area\(^1\), the complexity of the SSD computation can be estimated as $O(P^2)$. When the candidate patch and the object area overlaps\(^2\), we only use the pixels $I_q$ in the intersection of the candidate patch $\Psi_q$ and source image $\Phi$ ($I_q \in \Psi_q \cap \Phi$) to perform the computation. We can estimate the computation complexity as $O(kP^2)$, in which $k$ is a constant value, representing the average ratio of the overlapping area. For a typical rectangle search

\(^1\)In this case, the candidate patch is in the area of $(SA - OA)$.

\(^2\)In this case, the candidate patch is in the area of $OA$.  

Figure 5.12: The experimental results for increased patch size. ($\alpha = 0.05$ in this experiment.)
\[
Complexity_{overall} \approx O(wh/P^2) \cdot ((2\alpha + 1)^2 - 1)wh \cdot \left[ \text{Prob}(\Psi_q \in (SA - OA)) \cdot O(P^2) 
+ \text{Prob}(\Psi_q \in OA) \cdot O(kP^2) \right]
\]
\[
= O\left(\left(\frac{SA - OA}{SA}\right) + \frac{OA}{SA}\right)
\]
\[
= O\left(\frac{SA - OA}{SA} \cdot \frac{(2\alpha + 1)^2 wh - (2(P - 1) + w)(2(P - 1) + h)}{(2\alpha + 1)^2 wh - (2(P - 1) + w)(2(P - 1) + h)} - wh\right)
\]
\[
+ k \frac{2(P - 1) + w)(2(P - 1) + h) - wh}{((2\alpha + 1)^2 - 1)wh}
\]
\[
= O[wh \cdot ((2\alpha + 1)^2 wh - (2(P - 1) + w)(2(P - 1) + h) + k((2(P - 1) + w)(2(P - 1) + h) - wh))]
\]
\[
= O[wh \cdot ((2\alpha + 1)^2 wh - (1 - k)(2(P - 1) + w)(2(P - 1) + h))]
\]
\[
= O[wh^2 \cdot ((2\alpha + 1)^2 - (1 - k)(2(P - 1) + 1)(2(P - 1) + 1) - k)]. \quad (5.9)
\]

Experimental results show that the processing time can be reduced by increasing patch size while reducing the search area. From Fig. 5.10, an intuitive explanation is that when we reduce the search area, more candidate patches overlap with the object region. In this scenario, the bigger the patches are, the more overlap there will be. As a result, the amount of required computations becomes smaller. Thus, as we increase the patch size and reduce the search area, the processing time decreases. Equation (5.9) shows that for bigger search factor \( \alpha \), the term \((2\alpha + 1)^2\) dominates the complexity. In this case, the complexity change caused by the increased patch size is negligible. However, when \( \alpha \) becomes smaller, the search area decreases. When the search area \( SA \) becomes comparable to or even smaller than the object area \( \Omega \), the term \((2\alpha + 1)^2\) becomes less dominant. Therefore, for smaller search factor \( \alpha \), increasing the patch size \( P \) can reduce the computation complexity. Experimental results shown in Fig. 5.11 verify the above analysis. In Fig. 5.11, the processing time for patch sizes of \( 13 \times 13 \) and \( 17 \times 17 \) is normalized by the processing time of
Table 5.4: Local memory usage for “WalkingMan” image.

<table>
<thead>
<tr>
<th>Patch size</th>
<th>Data</th>
<th>Local memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 × 9</td>
<td>Source data</td>
<td>1024 bytes</td>
</tr>
<tr>
<td></td>
<td>Patch data</td>
<td>324 bytes</td>
</tr>
<tr>
<td></td>
<td>Patch pixel label</td>
<td>324 bytes</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>1672 bytes</strong></td>
</tr>
<tr>
<td>13 × 13</td>
<td>Source data</td>
<td>1600 bytes</td>
</tr>
<tr>
<td></td>
<td>Patch data</td>
<td>676 bytes</td>
</tr>
<tr>
<td></td>
<td>Patch pixel label</td>
<td>676 bytes</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>2952 bytes</strong></td>
</tr>
<tr>
<td>17 × 17</td>
<td>Source data</td>
<td>2304 bytes</td>
</tr>
<tr>
<td></td>
<td>Patch data</td>
<td>1156 bytes</td>
</tr>
<tr>
<td></td>
<td>Patch pixel label</td>
<td>1156 bytes</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>4616 bytes</strong></td>
</tr>
</tbody>
</table>

the 9 × 9 patch to show the performance speedup achieved by increasing the patch size. Experimental results show that, for bigger search areas (α ≥ 1), patch size does not affect the performance. However, as the search area keeps decreasing (α < 1), bigger patch size leads to more significant time reduction. The experimental results in Fig. 5.11 also indicate that increasing the patch size works for the experiments with (or without) the memory optimization discussed in Section 5.2.3.5. In addition, Fig. 5.12 shows that by increasing the patch size, we can generate visually plausible result images without degrading the image quality.

5.2.3.5 Memory Optimization

Similar to desktop GPUs, mobile GPUs also suffer from long latency of the off-chip global memory. The local memory on the mobile GPU provides fast memory accesses and can be shared by work items in the same work group. As mentioned before, a work group contains 8 × 8 work items, each of which computes an SSD value between an object patch and a candidate patch. As shown in Fig. 5.13, the n-th work item works on the n-th patch. Most of the pixels in adjacent candidate patches overlap. For instance, patch 1 and patch 2 share most of the image pixels and only one column of
Figure 5.13: A detailed diagram showing how \( 8 \times 8 \) work items in a work group compute SSD values in parallel. Most of the pixels in the dashed box are accessed multiple times by parallel work items.
pixels in each patch are different. Similarly, all adjacent candidate patches processed by $8 \times 8$ work items have many overlapped pixels, each of which is accessed multiple times by several different work items. These unnecessary memory accesses to the global memory can lead to long latency and increase the processing time. We can also tell from Fig. 5.13 that for a $P \times P$ patch, $(P + 8 - 1) \times (P + 8 - 1)$ pixels are actually shared among work items. Thus, we can load these pixels into the on-chip local memory to allow data sharing and avoid unnecessary accesses to the global memory. In our OpenCL implementation, $(P + 8 - 1)^2 \cdot \text{sizeof}(\text{cl}_\text{uchar4})$ source image data, $P^2 \cdot \text{sizeof}(\text{cl}_\text{uchar4})$ patch image data and $P^2 \cdot \text{sizeof}(\text{cl}_\text{int})$ patch pixel label data can be loaded into the local memory.

Algorithm 13 Parallel data loading from global memory to local memory in findBestPatch OpenCL kernel function.

1: Get global ID of the current work item: $(\text{gid}.x, \text{gid}.y)$;
2: Get local ID of the current work item: $(\text{lid}.x, \text{lid}.y)$;
3: $\text{local\_id} = \text{lid}.y \ast \text{lsize}.x + \text{lid}.x$;
4: Get local work group size: $(\text{lsize}.x, \text{lsize}.y)$;
5: $\text{group\_size} = \text{lsize}.x \ast \text{lsize}.y$;
6: $\text{local\_mem\_size} = (P + 8 - 1) \ast (P + 8 - 1)$;
7: while($\text{local\_id} < \text{local\_mem\_size}$)
8: \hspace{1em} if($\text{local\_id} < P \ast P$)
9: \hspace{2em} Calculate global memory address for patchData and patchPixelLabel;
10: \hspace{2em} Load patchPixelLabel into local memory;
11: \hspace{2em} Load patchData into local memory;
12: \hspace{1em} end if
13: \hspace{1em} Calculate global memory address for srcData;
14: \hspace{1em} Load srcData into local memory;
15: \hspace{1em} $\text{local\_id} + = \text{group\_size}$;
16: \hspace{1em} end while
17: barrier(CLK\_LOCAL\_MEM\_FENCE);
18: Start SSD computation from here. (Similar to Algorithm 1, except for that the needed data are already inside local memory.)

Table 5.4 lists the local memory usage for different patch sizes for the “Walking-Man” test image. The required local memory for all the patch sizes listed in Table 5.4 can be fit into the 8KB local memory of the Adreno GPU. In addition, if we carefully
design the method to load data from the global memory to the local memory by
data striping, we can coalesce the global memory access to further reduce latency.
The parallel data loading from the global memory to the local memory is shown in
Algorithm 13, in which the coalesced global memory access is achieved.

Experimental results in Fig. 5.14 demonstrate the performance improvement by
utilizing the local memory to enable the data sharing between work items inside the
same work group. We observe on average a 30% reduction in processing time after
using the local memory.

5.2.4 Experimental Results

We implemented the exemplar-based inpainting algorithm for object removal on a test
platform based on the Snapdragon chipset using OpenCL and the Android NDK [154,
156]. We applied the proposed optimization techniques discussed in Section 5.2.3.
Figure 5.15: An interactive object removal demo on Android with the OpenCL acceleration. (a) original image; (b) a mask indicating the object area; (c) intermediate result; (d) final result image after iterative editing.
To demonstrate the efficiency and practicality of the proposed implementation, we developed an interactive OpenCL Android demo on the test platform. Fig. 5.15 shows screen-shots of the implemented Android demo application, in which an end user can draw a customized mask by touching the touchscreen to cover an unwanted object and then remove it by pressing a button. The demo allows iterative editing, so that the user can keep editing an image until a satisfying result is obtained.
<table>
<thead>
<tr>
<th>Search factor $\alpha$</th>
<th>Search area</th>
<th>CPU-only</th>
<th>Heterogeneous CPU+GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time w/o local memory (s)</td>
<td>Time w/ local memory (s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Patch size</td>
<td>Patch size</td>
</tr>
</tbody>
</table>
|                        |             | $9 \times 9$ | $13 \times 13$ | $17 \times 17$ | $9 \times 9$ | $13 \times 13$ | $17 \times 17$
| 2                      | $382 \times 384$ | 18.08 | 16.97 | 18.30 | 16.32 | 14.93 | 14.52 | 9.31 | 8.82 | 8.42 |
| 1                      | $234 \times 311$ | 13.37 | 9.91 | 9.74 | 8.17 | 7.80 | 7.61 | 5.34 | 4.69 | 4.94 |
| 0.5                    | $156 \times 248$ | 11.27 | 9.80 | 8.20 | 5.29 | 4.24 | 3.95 | 3.93 | 2.93 | 2.43 |
| 0.2                    | $109 \times 176$ | 7.13 | 5.20 | 3.79 | 2.95 | 2.25 | 1.90 | 2.26 | 1.59 | 1.33 |
| 0.05                   | $86 \times 139$ | 6.02 | 4.73 | 3.85 | 2.12 | 1.65 | 1.45 | 2.04 | 1.52 | 1.25 |

Table 5.6: Total processing time for “River” image. With OpenCL kernels running on the GPU.

<table>
<thead>
<tr>
<th>Search factor $\alpha$</th>
<th>Search area</th>
<th>CPU-only</th>
<th>Heterogeneous CPU+GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time w/o local memory (s)</td>
<td>Time w/ local memory (s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Patch size</td>
<td>Patch size</td>
</tr>
</tbody>
</table>
|                        |             | $9 \times 9$ | $13 \times 13$ | $17 \times 17$ | $9 \times 9$ | $13 \times 13$ | $17 \times 17$
| full search            | $480 \times 639$ | 25.94 | 25.01 | 27.41 | 18.28 | 16.19 | 16.01 | 14.77 | 13.80 | 13.35 |
| 2                      | $295 \times 465$ | 12.23 | 12.15 | 12.68 | 8.52 | 7.93 | 7.95 | 6.71 | 6.94 | 6.59 |
| 1                      | $177 \times 279$ | 7.76 | 6.98 | 5.34 | 5.09 | 3.13 | 3.11 | 4.15 | 2.74 | 2.60 |
| 0.5                    | $118 \times 186$ | 4.80 | 3.91 | 3.82 | 2.81 | 2.06 | 1.42 | 2.34 | 1.87 | 1.5 |
| 0.2                    | $83 \times 130$ | 3.21 | 1.66 | 1.57 | 1.93 | 1.18 | 1.05 | 1.84 | 1.34 | 1.06 |
| 0.05                   | $65 \times 102$ | 2.29 | 1.72 | 1.91 | 1.39 | 1.15 | 1.01 | 1.66 | 1.16 | 1.00 |
Table 5.7: Total processing time for “Dive” image. With OpenCL kernels running on the GPU.

<table>
<thead>
<tr>
<th>Search factor $\alpha$</th>
<th>Search area</th>
<th>CPU-only</th>
<th>Heterogeneous CPU+GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time w/o local memory (s)</td>
<td>Time w/ local memory (s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Patch size</td>
<td>Patch size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 × 9</td>
<td>13 × 13</td>
</tr>
<tr>
<td>full search</td>
<td>576 × 384</td>
<td>64.48</td>
<td>62.86</td>
</tr>
<tr>
<td>2</td>
<td>576 × 384</td>
<td>65.11</td>
<td>64.83</td>
</tr>
<tr>
<td>1</td>
<td>465 × 384</td>
<td>52.17</td>
<td>54.31</td>
</tr>
<tr>
<td>0.5</td>
<td>310 × 369</td>
<td>54.36</td>
<td>37.07</td>
</tr>
<tr>
<td>0.2</td>
<td>217 × 260</td>
<td>40.92</td>
<td>29.64</td>
</tr>
<tr>
<td>0.05</td>
<td>171 × 205</td>
<td>35.44</td>
<td>22.22</td>
</tr>
</tbody>
</table>

Table 5.8: Total processing time for “Hill” image. With OpenCL kernels running on the GPU.

<table>
<thead>
<tr>
<th>Search factor $\alpha$</th>
<th>Search area</th>
<th>CPU-only</th>
<th>Heterogeneous CPU+GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time w/o local memory (s)</td>
<td>Time w/ local memory (s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Patch size</td>
<td>Patch size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 × 9</td>
<td>13 × 13</td>
</tr>
<tr>
<td>full search</td>
<td>1024 × 550</td>
<td>153.36</td>
<td>217.15</td>
</tr>
<tr>
<td>2</td>
<td>1024 × 50</td>
<td>21.07</td>
<td>15.86</td>
</tr>
<tr>
<td>1</td>
<td>1024 × 30</td>
<td>14.46</td>
<td>14.00</td>
</tr>
<tr>
<td>0.5</td>
<td>1024 × 20</td>
<td>12.88</td>
<td>14.15</td>
</tr>
<tr>
<td>0.2</td>
<td>1024 × 14</td>
<td>13.33</td>
<td>14.00</td>
</tr>
<tr>
<td>0.05</td>
<td>1024 × 12</td>
<td>11.52</td>
<td>15.46</td>
</tr>
</tbody>
</table>
From Table 5.5 to Table 5.8, we show the processing time of the OpenCL-based implementations, including the CPU-only results (utilizing multi-core CPU on the chip) and CPU-GPU heterogeneous implementations. We can notice that the OpenCL implementations with proposed optimizations significantly improve the processing performance compared to the serial version implementation. The CPU-GPU heterogeneous implementations further improve the performance compared to the CPU-only implementations.

Table 5.5 shows experimental results for the “WalkingMan” image, in which the image size is $512 \times 384$, and the size of the object area $76 \times 128$. The mask is manually drawn to cover the walking person. With the default parameter configuration ($9 \times 9$ patch size, full search), the serial version of the implementation running on one CPU core uses 87 seconds to finish the processing (shown in Table 5.2), which is a long processing time for a practical mobile application. The fact that iterative editing is required under many circumstances makes the serial implementation far from being practical. Table 5.5 shows experimental results for OpenCL-based parallel solutions. With the default parameter configuration, the multi-core CPU-only version reduces the processing time to 23.26 seconds, and the heterogeneous CPU-GPU implementation further reduces the processing time to 20.37 seconds (76.6% time reduction compared to 87 seconds processing time for the serial implementation).

With all the proposed optimization techniques applied, we observe significant performance speedup. With search factor $\alpha = 0.05$, patch size $17 \times 17$, and local memory enabled, the processing time is reduced to only 1.25 seconds, which indicates a 93.9% reduction in processing time compared to 20.37 seconds for the default parameter configuration (full search, $9 \times 9$ patch, without using the local memory). The subjective quality of resultant images does not degrade according to experimental results shown in Fig. 5.9 and Fig. 5.12. According to the research conducted by Niida et al., users
Table 5.9: Speedup for OpenCL-based heterogeneous implementations with the proposed algorithmic and memory optimizations.

<table>
<thead>
<tr>
<th>Image</th>
<th>Processing time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w/o opt.</td>
<td>w/ opt.</td>
</tr>
<tr>
<td></td>
<td>Full search</td>
<td>α = 0.05</td>
</tr>
<tr>
<td>Patch size 9 × 9</td>
<td>Patch size 17 × 17</td>
<td></td>
</tr>
<tr>
<td>WalkingMan</td>
<td>20.37</td>
<td>1.25</td>
</tr>
<tr>
<td>River</td>
<td>18.28</td>
<td>1.00</td>
</tr>
<tr>
<td>DivingMan</td>
<td>44.83</td>
<td>5.31</td>
</tr>
<tr>
<td>Hill</td>
<td>248.55</td>
<td>8.79</td>
</tr>
</tbody>
</table>

of mobile applications can tolerate several seconds average processing time for mobile services before they start to feel frustrated [163]. By accelerating the object removal algorithm using heterogeneous CPU-GPU partitioning on mobile devices, we successfully reduce the run time, which makes these types of computer vision algorithms feasible in practical mobile applications.

We can draw similar conclusions from the timing results for other test images shown in Table 5.6, 5.7 and 5.8. To demonstrate the effectiveness of our proposed optimization schemes, the speedup gained from the proposed optimization strategies are concluded in Table 5.9. We observe speedups from 8.44X to 28.3X with all our proposed optimizations applied to the heterogeneous OpenCL implementations.

5.2.5 Design Space Exploration

The object removal algorithm is a typical computer vision algorithm, which contains different operations such as raw pixel manipulating, image patch matching, color space conversion, iterative processing and so on. In the object removal algorithm, the best patch matching (findBestPatch) process is the most computationally-intensive module, which occupies over 97% of the overall computation time.
Performance Requirement and Complexity

Assume the image size is $M \times N$, the patch size is $P \times P$, and the object size is $w \times h$. We denote the color value of a pixel $x$ by $I_x = (R_x, G_x, B_x)$. For an object patch $\Psi_p$, the best patch $\Psi_q$ is chosen by computing:

$$
\Psi_q = \arg \min_{q \in \Phi} d(\Psi_p, \Psi_q),
$$

(5.10)

in which $d(\Psi_q, \Psi_p)$ is defined as an SSD (sum of squared difference) computation kernel as follows:

$$
d(\Psi_q, \Psi_p) = \sum_{p \in \Psi_p \cap \Phi} \sum_{q \in \Psi_q \cap \Phi} (I_p - I_q)^2.
$$

(5.11)

Since we perform SSD on color pixels, the computation of $(I_p - I_q)^2$ for pixel $I_p$ and $I_q$ requires 5 floating-point additions and 3 multiplications. Therefore, each SSD operation contains 8 floating-point operations.

Assume that the size of the original image is $M \times N$, and the size of the patch is $P \times P$. The complexity of $\text{findBestPatch}$ can be estimated as $O(8MNP^2)$ float-point operations. While, the overall object removal processing is iterative. We fill in a patch in a step. By performing patch filling for about $mn/P^2$ steps, we successfully fill in the hole left by removing the object. Therefore, the overall computation complexity can be roughly estimated as:

$$
\text{Complexity} = N_{\text{step}} \cdot N_{\text{OPS/step}}
\approx wh/P^2 \cdot 8 \cdot MNP^2
\approx 8 \cdot MNwh,
$$

(5.12)
in which $N_{OPS/step}$ indicates the number of operations per step, and $N_{step}$ indicates the number of steps to finish the object removal.

Since the object removal performs very complicated operations, and it is typically an iterative process when people try to use it to edit their images, the performance requirement is usually several seconds per frame. Based on the research conducted by Niida et al., users of mobile applications can tolerate average processing time up to several seconds on mobile devices before they start to feel frustrated [163].

For example, if we assume a typical image size $800 \times 600$, object size $200 \times 100$, and we assume the processing requirement is 1 frame/second, we can calculate the computation requirement as follows:

\[
\text{Operations per seconds} = 8 \cdot MNwh \cdot 1 \text{(operations/frame)} \cdot 1 \text{(frame/second)} \\
= 8 \times 800 \times 600 \times 200 \times 100 \text{FLOPS} \\
= 76.8 \text{GFLOPS}. \tag{5.13}
\]

Ideally, the current generation of mobile GPUs can provide around 100 GFLOPS peak performance, which is possibly able to accelerate the object removal algorithm on the mobile platform to reach a reasonable frame rate.

**Memory Bandwidth**

To perform the patch matching in the `findBestPatch` kernel, we need to access the pixels in the object patch once, and pixels of the whole source image except for the object region. Therefore, the memory bandwidth can be estimated as follows:
\[ BW_{\text{mem}} = 3(MN + 1 - wh)P^2 \cdot \frac{wh}{P^2} \]
\[ = 3(MN - wh + 1)wh \]  
(5.14)

This translates to a memory bandwidth requirement of 110.4 GB/s for a 800 \times 600 image with a 200 \times 100 object. Such a high memory bandwidth cannot be provided by the current generation of mobile GPUs.

To reduce the memory requirement, we optimize the algorithm by using on-chip shared memory to reduce the redundant memory accesses to global memory, as is described in Section 5.2.3.5 in Chapter 5. For each group of 8 \times 8 work items, we load a block of data with size of \((P + 8 - 1) \times (P + 8 - 1)\) into fast on-chip local memory. Then we reuse these data, but without generating new global memory traffic. The new memory bandwidth requirement for the optimized algorithm becomes:

\[ BW_{\text{mem, opt}} = 3\left(\frac{MN}{8^2}\right) \times (P + 8 - 1)^2wh/P^2 \]
\[ = 3MNwh(P + 7)^2/(8P)^2 \text{ samples/s.} \]  
(5.15)

Using the same example (800 \times 600 image with a 200 \times 100 object, patch size 9 \times 9), the memory bandwidth requirement becomes 5.69 GB/s. This is a more realistic memory bandwidth for modern mobile GPUs, which usually provide over 10 GB/s memory bandwidth.

Considering the fact that the mobile GPU has cache for global memories, the real memory bandwidth should be smaller than 110.4 GB/s for the original algorithm without using local memory. However, as shown in the experimental results in Section 5.2.4, the use of local memory as software-managed cache can lead to almost
Figure 5.16: Analysis of object removal algorithm with the proposed methodology.

2X performance improvement, which demonstrates the effectiveness of the proposed memory bandwidth optimization.

5.2.6 Analysis of Algorithm and Discussion

The object removal algorithm maps every pixel onto a thread of mobile GPU, therefore it demonstrates very high parallelism. To achieve around 1 frame/second performance, the arithmetic complexity is 76.8 GFLOPS, which can be handled by the modern mobile GPU that can provide computing capability of around 100 GFLOPS. The original findBestPatch kernel requires very high memory bandwidth (110.4 GB/s). The modern mobile GPU typically has a memory bandwidth from 12.8 GB/s to 14.9 GB/s, therefore, 110.4 GB/s memory bandwidth cannot be fulfilled by mobile GPU’s memory. Fortunately, the memory optimization techniques proposed in Chap-
Chapter 5 reduces the memory bandwidth requirement to 5.69 GB/s, which can be handled by the modern mobile GPUs. As is shown in Figure 5.16, we use the proposed methodology to analyze the optimized object removal algorithm, and the mobile GPU can meet all requirements for object removal algorithm, so we can implement an efficient object removal application on GPU.

Our experimental results showed that the computer vision algorithms such as object removal can be offloaded to the mobile GPU efficiently and we observe performance improvements. High parallelism and regular memory access pattern allows the software implementation to take advantage of the parallel compute units and hardware threads, and also enables data reusing and coalesced global memory access via the memory cache and shared on-chip memory (or local memory).

The computation of \textit{findBestPatch} in object removal algorithm is well-structured, and has inherent parallelism which allows us to easily map the algorithm onto parallel architecture. The algorithm mapping can be done very efficiently to GPU’s many-core architecture, with each sub-unit mapped to a work item in GPGPU.

Moreover, the memory access shows very regular patterns. Neighboring work items access adjacent memory data, and work items can also share a large amount of the data blocks. Therefore, these characteristics allow us to utilize the coalesced memory transaction to accelerate the memory accesses, and also use the fast on-chip local memory as cache to avoid redundant data fetching. In addition, we can take advantage of mobile SoC’s unified memory architecture, enabling the co-processing between CPU and GPU without adding extra memory transfer overhead.
5.3 OpenCL-based Accelerator of SIFT Algorithm using Mobile GPU

Recent advances in the computational capabilities of mobile processors have made possible a wide range of computer vision applications on smartphone and tablet platforms, such as image stitching, object recognition, and augmented reality. Efficient feature detection and extraction are essential building blocks for all of these tasks. The scale-invariant feature transform (SIFT) algorithm can produce distinctive keypoints and feature descriptors [164], and has been considered one of the most robust local feature extraction algorithms [165]. Although many alternative or high speed approximation algorithms to SIFT have been proposed, such as SURF [166], the SIFT algorithm remains in wide use and is attracting more attention.

Many research efforts have been made in prior works to design and optimize high speed SIFT implementations. Most of them use customized hardware or high performance workstations due to the high complexity of the SIFT algorithm [167]. General-purpose computing on graphics processing units (GPGPU) has also been used to speed up the processing of SIFT [168, 169]. However, most of the design and optimization techniques in these prior works are not practical for mobile devices, due to their high computational complexity and memory usage. There are only a few works targeting SIFT implementations on mobile platforms [46, 170], in which only part of the SIFT algorithm is accelerated, or feature accuracy is traded for processing speed.

Throughout the past several years, the power of mobile processors for parallel computation has improved, by the integration of a GPU utilizing multiple programmable graphics pipelines. Except for in the fields of mobile gaming and 3D texture rendering, many computationally-intensive algorithms can be accelerated by modern
Figure 5.17: Diagram of SIFT keypoint detection algorithm showing one octave with 6 Gaussian image layers.

mobile GPUs by means of emerging parallel programming models such as the Open Computing Language (OpenCL) [47, 171]. Due to the special hardware architecture of mobile processors, many techniques previously applied to desktop GPUs are not suitable for mobile applications. To achieve high performance, we need to analyze the algorithms and workloads specifically on mobile platforms and find an efficient mapping to embedded hardware.

5.3.1 Overview of Scale-invariant Feature Transform (SIFT)

A local feature keypoint represents an image pattern which differs from its neighboring pixels. The properties of keypoints can be extracted and described using keypoint descriptors. Keypoint descriptors are highly distinctive, enabling image matching or object recognition with high probability in a large database of features [164, 165]. The SIFT algorithm consists of the following stages to detect keypoints and extract feature descriptors [164]. The major steps for keypoint detection are shown in Fig. 5.17

**Gaussian pyramid.** First, an input image is convolved by a series of Gaussian smoothing kernels $G(x, y, \sigma)$ with different $\sigma$ to get the first set of images, called the
first octave. We downsample the input image to form the base of a smaller set of images, called the next octave, which will in turn be smoothed to an even higher scale factor. Each octave includes a few layers, each corresponding to a different scale factor $\sigma$. In this work, we use typical algorithm parameters suggested by Lowe’s paper: 5 octaves and 6 layers in each octave [164].

**Difference of Gaussian (DoG) pyramid.** We subtract every two adjacent layers to get DoG pyramids. For each octave, 5 DoG images are generated.

**Keypoint detection and refinement.** We compare each pixel in a DoG image with its surrounding $3 \times 3$ neighbor pixels from the current layer, a lower layer and a higher layer. If the value of a pixel is a maxima or minima among the total 26 neighbors, it is identified as a keypoint candidate. Then we perform a quadratic interpolation in scale space around the candidate to refine the accuracy of keypoint location to subpixel level. Finally, keypoints with low contrast are considered non-distinctive, so they are rejected.

**Gradient orientation assignment.** For each keypoint, we compute the gradient magnitude and orientation angle of the Gaussian pyramid images for all pixels in a region around the keypoint. We build an orientation histogram, which partitions 360 degrees of angular orientation into 36 bins. Each pixel in the KP-region contributes to one of the 36 bins based on its gradient orientation angle. The contribution score is the gradient magnitude value weighted by a Gaussian function $G(x_0, y_0, \sigma)$, where $(x_0, y_0)$ is the pixel’s coordinate relative to the keypoint. We then choose the bins with highest scores as dominant orientations for the keypoints. Each keypoint can have multiple orientations, and each orientation is used to generate a unique feature descriptor.

**Descriptor generation.** A feature descriptor is a vector which represents the local properties of a keypoint. In this step, a keypoint region (KP-region) surrounding
a keypoint is extracted and rotated to account for the keypoint orientation. Then, the KP-region is partitioned into $4 \times 4$ subregions. In each subregion, we build an orientation histogram with 8 bins, or 45 degrees per bin. Then, in a process similar to the gradient orientation assignment step, pixels in each subregion contribute to the histogram by adding the Gaussian-weighted gradient magnitude to the corresponding bins. Repeating this for all $4 \times 4$ subregions results in a feature vector of 128 values, representing 8 bins for each of the 16 subregions. Finally, we truncate and normalize this vector to form a keypoint descriptor.

5.3.2 Algorithm Profiling and Workload Partitioning

To better understand the workload and performance of the SIFT algorithm on mobile devices, we implemented the SIFT algorithm using both serial C++ and parallel OpenCL computation kernels. The SIFT algorithms are roughly partitioned into computation functions based on the workflow described in Section 5.3.1. By running our SIFT implementations over a benchmark dataset on a mobile processor, we measure the processing time for each function for both the CPU and GPU implementations. The data transfer time between the host CPU and the device GPU is also counted for the GPU implementation. Carefully analyzing the profiling results finally leads to a heterogeneous implementation, which efficiently maps the SIFT algorithm onto the CPU and GPU in a mobile processor.

(1) Experimental Setup

We use 48 images from a benchmark dataset designed for local feature performance evaluation [165, 172]. The dataset covers transformations such as blurring, change of viewpoint, zoom, rotation, and compression. All our test images have widths of 320 pixels, and image height varies from 214 to 256. We use a mobile device with a Qualcomm Snapdragon S4 Pro APQ8064 chipset, which includes a 1.5GHz quad-
core CPU and an Adreno320 GPU, with four compute units containing an array of ALUs running at 325MHz. We compiled our optimized C++ code using the Android Native Development Kit (NDK) version r8c and OpenCL 1.1. In this section, profiling is performed on an image called “graf” in the test dataset, as shown later in Fig.5.20. The “graf” image has more feature keypoints than other images, revealing the worst case processing time for the dataset.

(2) Data Structure

We first did experiments to determine the necessary data precision. When using a 32-bit floating point format, 269 keypoints are extracted, while switching to an 8-bit fixed point format reduces the number of keypoints to 137. Specifically, we observed that most of the keypoints for larger scales disappeared with the loss in precision. In addition, we lose sub-pixel level keypoint accuracy. These facts could significantly degrade the image matching performance, so we choose to use the floating point format.

For our OpenCL implementation, we define the data buffers as the Image2D type, taking advantage of the GPU’s high-performance texturing hardware, which provides image interpolation, border pixel handling and texture caching. By doing this, access to image data is accelerated and branching instructions in the kernel are reduced, since we do not need to pay special attention to handling the border pixels.

Packing several grayscale data into an RGBA texel is a popular technique to reduce the memory bandwidth for GPGPU applications [46, 168]. In our case, four 32-bit floating point data in a $2 \times 2$ square are packed into a CL_RGBA|CL_FLOAT format data. Data packing is applied to the Gaussian pyramid, DoG pyramid, and gradient pyramid. We can benefit from reduced device memory accesses, as well as utilization of the GPU’s vector processing units. With this data packing approach, we observed a 28% reduction in execution time for Gaussian pyramid generation, and
about a 40% reduction for gradient pyramid generation.

(3) Implementation of OpenCL Kernel Functions

After analyzing the workflow described in Section 5.3.1, the following major steps were implemented using separate functions: Gaussian pyramid generation, DoG pyramid generation, local extrema detection, keypoint refinement, orientation assignment, and descriptor generation. It is worth mentioning that both orientation assignment and descriptor generation require the gradient information, or the gradient magnitude and orientation angle, of the pixels in the KP-region. The gradient calculation needs the computationally expensive dot-product, square root and \( \text{atan}2() \) operations. Although for each keypoint, we only compute the gradient for a small KP-region, experiments show that the gradient information of each pixel is requested an average of more than one time during the whole SIFT computation. Therefore, when both the SIFT detector and descriptor are computed for the same image, it is more efficient to pre-compute the gradient information for all pixels to avoid redundant computations. In fact, all keypoints are located on layers 1~3, so the orientation and descriptor are only computed on those three layers. Therefore, we only build the gradient pyramid for layers 1~3.

In order to take advantage of the parallel architecture of the mobile GPU, we use an image tiling technique. We divide an input image into tiles, where each tile is assigned to a work group consisting of multiple work items, each of which processes one or more pixels in a tile. Due to the properties of the hardware, the typical size of a work group is \( 8 \times 8 \). Image tiling is applied to the Gaussian pyramid generation, DoG pyramid generation, local extrema detection, and keypoint refinement kernels, to exploit the parallelism of the algorithms. However, for orientation assignment and descriptor generation, computations are only performed in the KP-regions for a list of keypoints. For these two kernels, we map all work groups to one KP-region, using
Table 5.10: profiling results for the major processing steps of SIFT, using the 320×256 “graf” image. 269 keypoints are detected.

<table>
<thead>
<tr>
<th>Step</th>
<th>Time results (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td>Gaussian pyramid</td>
<td>98.5</td>
</tr>
<tr>
<td>DoG</td>
<td>16.08</td>
</tr>
<tr>
<td>Gradient pyramid</td>
<td>80.73</td>
</tr>
<tr>
<td>Local extrema</td>
<td>14.13</td>
</tr>
<tr>
<td>Keypoint refinement</td>
<td>0.85</td>
</tr>
<tr>
<td>Orientation assignment</td>
<td>30.17</td>
</tr>
<tr>
<td>Descriptor generation</td>
<td>193.87</td>
</tr>
</tbody>
</table>

*a. Bold fonts indicate our choices for each step.

*b. Data transfer time is included in GPU time, due to small data sizes.

atomic instructions to compute the histograms.

(4) Profiling Results and Workload Partitioning

In Table 5.10, the processing times are shown for each of the major steps for both the C++ and OpenCL implementations. The readback time also includes the time for data unpacking from RGBA format to the normal image format, since this must happen after memory readback. In the steps of local extrema detection, keypoint refinement, orientation assignment and descriptor generation, the GPU readback time is negligible due to the very small data size.

Since the algorithms for Gaussian pyramid generation, gradient pyramid generation and descriptor generation provide a sufficient amount of parallelism, we can benefit from the GPU’s parallel architecture. The GPU processing time plus data transfer overhead is shorter than the CPU processing time, indicating that we can achieve better performance by offloading these functions to the GPU. On the other hand, the CPU version outperforms the GPU version for the other kernels. Obviously, keeping them on the CPU results in higher performance. Among them, orientation assignment has much longer processing time on the GPU than the CPU, since the main operation in this kernel is histogram computation, which is inherently serial and
Figure 5.18: Algorithm partitioning for heterogeneous implementation.
becomes a major bottleneck limiting the performance. For DoG pyramid generation, local extrema and keypoint refinement, the GPU implementation is slower since the relatively simple computations cannot fully utilize the GPU’s resources. In these cases, the higher clock frequency of the CPU leads to better performance. Based on the profiling results, we partition the SIFT algorithm to a CPU-GPU heterogeneous implementation to minimize the total processing and memory transfer time, as shown in Fig.5.18.

5.3.3 Optimization of Parallel Gaussian Pyramid Generation

Generation of the Gaussian pyramid is the most time consuming step in keypoint detection, so it is necessary to explore optimization techniques. For desktop GPUs, due to the large number of cores, we may compute Gaussian blur for all 6 layers in an octave (or even multiple octaves) in a single parallel kernel. In mobile devices, the limited number of compute units and onchip memory make the fully parallel approach infeasible. Following our previous work, in this implementation, we use a separable two-pass Gaussian filtering method. We also apply dynamic runtime code generation to produce branchless OpenCL code [46]. To be specific, we use standard I/O library functions of C++ to generate formatted OpenCL kernel code incorporating filter parameters at runtime, so that loops in OpenCL kernel functions are fully unrolled without branch operations.

In addition to the above approaches, we utilize a recursive Gaussian blur method which is a good fit for mobile GPUs. The recursive Gaussian blur is based on the idea that one pass of the Gaussian blur with \( \sigma_a \) is equivalent to applying two passes with \( \sigma_b \) and \( \sigma_c \) sequentially if \( \sigma_a^2 = \sigma_b^2 + \sigma_c^2 \). Therefore, a higher layer Gaussian blur image with \( \sigma_1 \) can be generated by applying an extra amount of blurring \( \sqrt{\sigma_1^2 - \sigma_0^2} \) to the previous layer with \( \sigma_0 \). One major benefit of the recursive Gaussian blur is that the
Figure 5.19: Efficient Gaussian pyramid generation. The procedure for octaves $0 \sim 2$ is shown. For octaves $3 \sim 4$, the process is similar to octave 2.

Subsequent convolution is done using small scale factor $\Delta \sigma = \sqrt{\sigma_1^2 - \sigma_0^2}$, therefore, the required Gaussian filter kernel can be shorter (which is typically $6\sigma + 1$). A shorter filter kernel means fewer memory accesses, which allows for an efficient mobile GPU implementation, since low memory bandwidth is one of the major bottlenecks for the GPU.

Furthermore, layers $0 \sim 2$ in each octave $n$ are directly generated simply by downsampling the layers $3 \sim 5$ in octave $(n - 1)$ by one half according to scale space theory, without doing any Gaussian filtering. Downsampling by one half can be implemented very efficiently by accessing $1/4$ of the pixels without the need for arithmetic operations. Our efficient Gaussian pyramid workflow is shown in Fig.5.19, in which for each octave other than the first, we only need 3 downsample operations and 3 Gaussian blur operations. The Gaussian blur and downsample blocks are implemented using OpenCL kernels with the image tiling technique.
Figure 5.20: Keypoint detection results for “graf”, “ubc”, and “boat” images. Each circle represents a keypoint, its radius indicating the scale of the feature. The lines in the circle represent orientation of the keypoints. Numbers of keypoints are 269, 133, and 192, respectively.

Table 5.11: Processing time on the whole dataset with 48 images.

<table>
<thead>
<tr>
<th>Time results (ms)</th>
<th>CPU only</th>
<th>CPU+GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARM-v5</td>
<td>ARM-v7a</td>
</tr>
<tr>
<td>Gaussian pyramid</td>
<td>710.96</td>
<td>81.78</td>
</tr>
<tr>
<td>Readback Gaussian</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>DoG pyramid</td>
<td>30.28</td>
<td>15.30</td>
</tr>
<tr>
<td>Gradient pyramid</td>
<td>232.37</td>
<td>78.81</td>
</tr>
<tr>
<td>Readback Gradient</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>42.94</td>
<td>21.40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local extrema</th>
<th>(Incl. refinement, orientation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keypoint detection total</td>
<td>1015.65</td>
</tr>
<tr>
<td>Descriptor generation total</td>
<td>261.43</td>
</tr>
<tr>
<td>Complete SIFT</td>
<td>1278.09</td>
</tr>
</tbody>
</table>

5.3.4 Experimental Results

We benchmark the complete SIFT implementation with keypoint detection and descriptor generation on 48 images using the whole dataset [172]. The keypoint detection results for three images in the dataset are shown in Fig.5.20. Thanks to the full precision we used, there is no performance loss in our heterogeneous implementation in terms of keypoint number and accuracy, compared to some popular SIFT implementations for the desktop CPU, such as Lowe’s implementation [164] and VLFeat [173].

Table 5.11 reports the average time per image. For the CPU-only implementa-
tion, we compile the NDK code with both the ARM-v5 and ARM-v7a instruction set architectures. The average number of keypoints detected per image is 95. First, we notice that ARM-v7a generates much more efficient code than ARM-v5. For the detection portion, we observe a 1.69X speedup with the heterogeneous implementation, compared to the CPU-only implementation. With the heterogeneous implementation, we can achieve 8.5 frames per second (FPS) for keypoint detection, and 19 FPS for descriptor generation. We note that a nonnegligible part of time for the heterogeneous solution comes from memory transfers (24.6ms, which is 21% of the total detection time). If the next generations of mobile GPUs can include newer memory technology and faster memory bandwidth between the CPU and GPU, we foresee greater speedup using heterogeneous computing techniques.

To measure energy efficiency, we compute SIFT on the same dataset for several minutes, measure the average system power, and then subtract the idle system power. We measured 1490mW and 1429mW power consumptions for the CPU-only implementation and the heterogeneous implementation, respectively. The average energy consumption per image is 413.0mJ for the CPU-only implementation, and 242.3mJ for the heterogeneous one. Therefore, a 41% reduction in energy consumption is achieved.

5.3.5 Design Space Exploration

The SIFT algorithm is a good representation of complex computer vision algorithms. The SIFT contains several computationally-intensive blocks, including Gaussian filtering, difference of Gaussian (DoG), gradient pyramid, non-maximal suppression (for local extrema detection), gradient histogram (for orientation assignment and descriptor generation), and linear equation solver (for keypoint refinement).
Algorithm Analysis and Workload Characteristics

Each algorithm has very different characteristics in terms of parallelism degree, arithmetic intensity and memory bandwidth requirement. Table 5.12 summarizes the characteristics for these major algorithm modules and their corresponding processor choice based on the profiling results on a real mobile device (from Table 5.10).

Table 5.12: SIFT algorithm breakdown and analysis.

<table>
<thead>
<tr>
<th>Algorithm module</th>
<th>Parallelism</th>
<th>Arithmetic complexity</th>
<th>Memory BW requirement</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian pyramid</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>GPU</td>
</tr>
<tr>
<td>DoG</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>CPU</td>
</tr>
<tr>
<td>Gradient pyramid</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
<td>GPU</td>
</tr>
<tr>
<td>Local extrema</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>CPU</td>
</tr>
<tr>
<td>Keypoint refinement</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>CPU</td>
</tr>
<tr>
<td>Orientation assignment</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>CPU</td>
</tr>
<tr>
<td>Descriptor generator</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
<td>GPU</td>
</tr>
</tbody>
</table>

From Table 5.12, we can notice that the algorithm modules with high parallelism degrees usually are suitable for off-loading to GPU to achieve performance gain. One exception is DoG, which shows very high and regular parallelism. The image pixels can be directly mapped onto GPU’s threads, and each thread performs exactly the same operations on the pixel values. However, the very low arithmetic complexity (only a subtraction) and medium memory bandwidth (reading two pixels for each output pixel). This high ratio of memory transfer to arithmetic operation makes it unsuitable for GPU’s architecture, since the latency hiding scheme of GPU for slow global memory accesses will fail in this case.

For other algorithm modules, such as local extrema detection, keypoint refinement
and orientation assignment, they expose very low parallelism and complicated flow control for divergent branches, these modules therefore shows better performance for a CPU implementation.

**Complexity Analysis and Memory Bandwidth**

Assume the image size is $M \times N$. Number of octaves and number of layers per octave are denoted as $N_O$ and $N_L$, respectively. We also use $N_{FPS}$ to indicate the required performance. The number of operations per layer is denoted as $N_{op}$. The memory accesses for each layer can be denoted as $N_{mem}$. We can model the overall complexity as follows:

$$
Complexity = MN \cdot N_{FPS} N_O N_L N_{op}.
$$

(5.16)

Similarly, the overall memory bandwidth can be modeled as:

$$
BW_{mem} = MN \cdot N_{FPS} N_O N_L N_{mem}.
$$

(5.17)

Since the number of keypoint depends the texture structure of images, and algorithm modules such as keypoint refinement and orientation assignment are operated on the detected keypoints. We can only get a rough estimation of the number of operations and memory accesses.

For a $320 \times 240$ image, if we set the SIFT parameters as follows: $N_O = 5$, $N_L = 6$. The $N_{op}$ can be roughly estimated as 150 operations per layer; while the memory accesses can be estimated as 100 memory accesses per layer. Based on the equations (5.16) and (5.17), to achieve 15 frame/second performance, we can compute the estimated arithmetic requirement as 5.2 GFLOPS and memory bandwidth requirement as 110 Gb/s (or 13.75 GB/s).
Comparison with Hardware Implementations and Further Discussion

Due to the divergent characteristics of algorithm modules in the SIFT algorithm, the heterogeneous architecture containing CPU-GPU is necessary to accelerate the processing speed and save computation energy on mobile devices. There are some VLSI implementations of SIFT algorithm. For example, Bonato et al. [174] implemented SIFT algorithm using FPGA and achieved about 30 frame/seconds performance for $320 \times 240$ images. Kim et al. [175] presented a reduced-memory requirement SIFT implementation on FPGA, which reduces 80% of the memory storage for the SIFT detection; however, no performance number is given. These implementations only consider the SIFT keypoint detection part, without implementing the feature descriptor. Moreover, they achieve relatively high frame rates at the cost of the simplified SIFT algorithm, therefore, reducing the SIFT keypoint accuracy and matching performance. Because they also use very specific memory architectures, which lead to poor flexibility and availability. Recently, Zhong et al. [176] proposed a hardware implementation of SIFT algorithm based on FPGA+DSP platform. Both SIFT detector and description are implemented. Their system can achieve around 10 ms SIFT keypoint detection and 80 us SIFT descriptor generation per keypoint for $320 \times 240$ images. Again, this FPGA+DSP system still utilizes very specific optimization techniques for hardware design and memory systems, causing it unfeasible for a general purpose mobile platform.

As a comparison, the proposed CPU-GPU heterogeneous implementation explores the computation capability of the existing hardware modules, without adding extra hardware overhead. With the help of the new parallel programming models, we accelerate the SIFT algorithm to achieve near real-time performance on mobile devices, at the same time reduce the energy consumption by 41%. Under most of the cases in which the SIFT is used as a preprocessing block, the near real-time perfor-
mance can satisfy most of the applications. With the exponential growth of mobile GPU’s processing capability (almost doubled compute units and doubled memory bandwidth [177]), the next generation mobile GPUs can possibly achieve real-time processing. Moreover, the proposed CPU-GPU-based implementation can be easily integrated into other computer vision algorithms, since they use standard system memory for input and output, rather than some customized I/O interfaces for VLSI implementations. For example, the proposed SIFT implementation can be integrated into the camera pipeline of the mobile SoC to enable some advanced computational imaging/photography applications, with the mobile processor and mobile GPU directly accessing data from memory of the camera pipeline. The proposed heterogeneous implementation also shows great flexibility, availability, and portability.

5.3.6 Analysis and Discussion

The SIFT algorithm includes several computation kernels with different arithmetic intensities and memory access patterns, so these computation kernels show very different behaviors on mobile CPU or mobile GPU. For some kernels like Gaussian pyramid, gradient pyramid and so on, the parallelism is high, so we can achieve speedup by offloading these kernels onto the mobile GPU.

Moreover, most of the operations are performed on the pixels in images, and operations for adjacent pixels usually request data with good spacial locality. By using GPU’s texture engines, we enable data sharing and reusing via the hardware-managed cache for the texture memory. For the other kernels such as DoG or orientation assignment, they either have very low ratio of arithmetic to memory access (leading to poor memory access latency hiding), or they have low parallelism (for example, orientation assignment is only applied to the detected several hundred keypoints). In addition, some kernels even have complicated divergent branches, such as the keypoint
Figure 5.21: Analysis of SIFT algorithm with the proposed methodology.
refinements. These kernels may not benefit from mobile GPU’s parallel architecture. Fortunately, the mobile CPU is becoming more powerful and can handle these kernels efficiently.

The overall arithmetic complexity (5.2 GFLOPS) and memory bandwidth requirement (13.75 GB/s) to achieve 15 frames/second can be met by the current generation mobile GPUs, as shown in Figure 5.21. Therefore, by utilizing the CPU-GPU heterogeneous architecture, we can achieve performance improvement and energy reduction compared to a CPU-only implementation.
Design Space Exploration and Comparison on Heterogeneous Architecture

Heterogeneous computing has become the trend of mobile architecture to incorporate capabilities to perform a wide range of tasks and enable different applications. However, due to the quite divergent characteristics of different hardware architectures and corresponding software stacks, it is still challenging to partition workloads and choose the right platform for the right task.

In previous chapters, we have explored design mythologies and optimization strategies to map computationally-intensive wireless communication and computer vision use cases onto GPU and VLSI architectures. We have explored design spaces for each specific platform from aspects including parallelism exploitation, algorithmic optimization, memory optimization and so on. In this chapter, we further explore design space in the perspective of which are the superior mobile accelerators.

A practical application of this design space exploration is to provide a heuristic by which one may choose among alternate accelerators for a specific application domain. On the other hand, we may form a theoretical basis for hardware-software co-design among customized hardware, many-core GPU and multi-core CPU in a heterogeneous
6.1 Comparison and Design Tradeoffs

To efficiently map algorithms onto a heterogeneous hardware architecture, we need to study the system specification and algorithm properties, especially the performance requirement of the algorithms to be implemented. Based on the system specification and algorithm analysis results, we can roughly predict the performance of the algorithm on certain architectures, therefore determining the appropriate accelerators for algorithms. We have already studied the characteristics of the four algorithms and analyzed the performance requirements of those algorithms in a few different aspects.

6.1.1 Performance Requirements

Based on the algorithm analysis and performance modeling, we first summarize the performance requirements for turbo decoding, LDPC decoding, object removal algorithm and the SIFT algorithm in Table 6.1.

It is worth mentioning that the performance requirements depend on the performance goals or input data sizes. For turbo decoding, we target at 1Gbps data throughput for LTE standard with rate of 1/3 and block size of 6144. For LDPC decoding, the requirements targeting the performance goal of 1Gbps data rate for the (1152, 2304) WiMAX code are shown. For object removal, the input data sets are 800×600 pixel resolution images with 200×100 pixel resolution object regions. While, for the SIFT, we assume that the SIFT detector/extractor can achieve 15 frames/sec performance for 320 × 240 pixel images. Please notice that, the arithmetic complexity and the memory bandwidth requirements heavily rely on the texture structure of the input images, therefore, we give an estimated arithmetic complexity and memory
bandwidth requirements.
Table 6.1: Comparisons of computation and memory access properties of different algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Computation kernels</th>
<th>Computation type</th>
<th>Parallelism degree</th>
<th>Arithmetic complexity</th>
<th>Memory bandwidth</th>
<th>Pattern</th>
<th>Predictable</th>
<th>Memory reuse potential</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turbo decoding</td>
<td>MAP decoder</td>
<td>Arithmetic intensive</td>
<td>Low: ~100</td>
<td>3114 GOPS</td>
<td>108.3 GB/s</td>
<td>Irregular</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LDPC decoding</td>
<td>CNP, VNP</td>
<td>Arithmetic intensive &amp; memory intensive</td>
<td>High: ~1000</td>
<td>570 GOPS</td>
<td>840 GB/s</td>
<td>Irregular</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Object removal</td>
<td>Patch matching</td>
<td>Arithmetic intensive</td>
<td>High: ~10^5</td>
<td>76.8 GFLOPS</td>
<td>5.69 GB/s</td>
<td>Regular</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIFT</td>
<td>Gaussian pyramid</td>
<td>Arithmetic intensive &amp; memory intensive</td>
<td>High: ~10^4</td>
<td>5.2 GFLOPS</td>
<td>13.75 GB/s</td>
<td>Regular</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>DoG</td>
<td>Memory intensive</td>
<td>High: ~10^4</td>
<td></td>
<td></td>
<td>Regular</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Gradient pyramid</td>
<td>Arithmetic intensive</td>
<td>High: ~10^4</td>
<td></td>
<td></td>
<td>Regular</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Local extrema</td>
<td>Memory intensive</td>
<td>High: ~10^4</td>
<td></td>
<td></td>
<td>Regular</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Keypoint refinement</td>
<td>Branch intensive</td>
<td>Low: ~100</td>
<td></td>
<td></td>
<td>Regular</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Orientation assignment</td>
<td>Branch intensive</td>
<td>Low: ~100</td>
<td></td>
<td></td>
<td>Regular</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
In Table 6.1, the parallelism degree indicates the maximum achievable parallelism for each algorithm. For example, in parallel turbo decoding, 2~32 MAP decoders can be utilized in parallel decoding. If more MAP decoders are employed, the short length for each sub-codeword will degrade the BER performance. At the same time, we can also utilize the trellis-level parallelism (typical 8 threads in parallel for an 8-state trellis). That being said, the total achievable parallelism can be ranging from 16~256, which is relatively low for massively parallel architectures. As a comparison, to parallelize an LDPC decoder, we can spawn a thread to process one check node messages, which easily leads to thousands of threads in total.

For the memory access, we analyze memory access patterns. The channel decoding algorithms involve message passing among processors or processing nodes. Between iterations, an interleaving or shuffling network routes the messages into very random memory addresses. The resultant irregular memory accesses introduce extra complexity for the memory sub-systems, and possibly reduce the decoding performance due to the memory bank conflict problem. In addition, in each iteration of the turbo decoding and LDPC decoding, we only read each message once and also write each message once. There is no data reuse, and this causes a very high memory bandwidth requirement. As a comparison, the computer vision algorithms operate on structured 2D arrays of image pixels. The typical operations in object removal and the SIFT algorithm show good temporal and spacial locality. Therefore, not only can we take advantage of the memory reading/writing routing logic, but we can also utilize a cache to enable the potential opportunity of data reuse, leading to higher memory performance.

An algorithm can be categorized as compute-bound or memory-bound, according to the property of operations in the algorithm. Relatively speaking, turbo decoding can be considered to be compute-bound. The intensive computation along with
the trellis traversal generates extremely high arithmetic requirements. Although the
turbo decoder’s memory bandwidth is not as high as in LDPC decoding, its irregular
memory access pattern caused by the interleaver significantly reduces the effective
memory bandwidth in a parallel architecture.

LDPC decoding demonstrates both high computation complexity and intensive
memory accesses. But its extremely intensive memory accesses are typically the
bottleneck for high throughput implementations. Many research works focused on
memory architecture optimizations, especially the shuffle network for the LDPC de-
coder. As we have discussed, turbo decoding and LDPC decoding have very poor
memory data reuse potential, therefore, we cannot take advantage of the caching
memory system.

The object removal algorithm is computationally complex and requires high mem-
ory bandwidth. Fortunately, this algorithm has very regular memory access pattern
to image pixels with good data reuse potential. Therefore, we can take advantage
of hierarchical memory systems to reduce the traffic to the main memory. With our
proposed memory optimization applied, the memory bandwidth requirement is sig-
nificantly reduced. The optimized version can be considered as a compute-bound
algorithm.

The SIFT algorithm contains multiple algorithm modules, which show very diverse
properties. Among them, DoG and local extrema kernels are memory-bound; others
can be considered as compute-bound (Gaussian pyramid generation can be considered
both). High parallelism degrees and regular memory accesses make the compute-
bound kernels suitable for GPU acceleration.

Given the high algorithm complexity and different memory bandwidth require-
ments, general-purpose processors cannot provide enough computing power to achieve
the performance goals. Accelerators are designed to meet different system perfor-
mance requirements such as high data throughput, lower latency, or lower power consumption. From the next section, we will briefly introduce the performance specifications of some major accelerators.

6.1.2 System Performance Specifications for Accelerators

The customized hardware logic can provide very high processing power for not only computationally-intensive signal processing applications but also control-intensive algorithms. The modern ASIC design or reconfigurable logic typically run at from several hundred Meta-hertz to several Giga-hertz clock frequency, using 45nm∼20nm CMOS technologies. The application-specific hardware logic can provide extremely high performance of more than several Tera-FLOPS thanks to the customized pipeline and fine-tuned scheduling. At the same time, they can also provide the best hardware efficiency in both chip area and power consumption. In mobile systems, some applications have very strict performance and efficiency requirements, therefore, customized hardware logic is widely used to accelerate algorithms in mobile devices, such as wireless communication modem, face recognition, video encoding/decoding and so on.

Although customized hardware designs can handle almost all applications, it is not feasible to implement every algorithm using customized hardware design due to the extremely high cost and long design cycles. Only those algorithms that have very strict performance and efficiency requirements need to be implemented using customized logic blocks. Moreover, to achieve the best efficiency, the customized logic are usually tailored to specific algorithms for specific systems, therefore, it is difficult to design and implement flexible or reconfigurable customized hardware. In many cases, beyond achieving high performance and efficiency, we also require the system to handle different situations and different configurations in a single hardware
design. The customized hardware accelerator is not the best candidate in such cases, where we prefer software-based solutions.

For example, the core modules in a wireless communication modem should handle very high data rates with small hardware overhead and power consumption, without any compromise. However, to extend the current hardware to support some new communication algorithms or new standards, the feasibility may be more important than performance. In some other examples such as image processing and computer vision systems, it is impossible to implement every algorithm using hardware. Under these circumstances, the scalability, flexibility and reconfigurability provided by the software-based accelerators are preferred to provide better hardware reuse capability. Therefore, we prefer to map the algorithms onto mobile SoC’s heterogeneous accelerators, if the performance requirements can be met.

The GPGPU usually contains a massively parallel architecture, which consists of thousands of processing elements. These processing elements are typically lightweight ALUs plus some control logic and memories. The GPGPUs are suitable for throughput-driven applications and can provide several Tera-FLOPS computing performance. However, they are not good for control-intensive algorithms due to their SIMT architecture. Currently, the large chip area and high power consumption of GPUs make them impractical for embedded systems. However, the design and optimization methodologies for the desktop GPUs and mobile GPUs are quite similar. During the past several years, the mobile GPUs are becoming more and more like desktop GPUs, in terms of architecture and performance capability. We can foresee many more algorithms accelerated by GPGPU. Moreover, the massively parallel architecture of GPUs provide us important hints to adapt the customized logic design, which may inspire GPU-like architectures in ASIC design, which incorporate both the GPU’s SIMD/SIMT architecture and specially-designed customized pipeline to
accelerate potential bottleneck operations.

The technical specifications of the NVIDIA GTX690 GPU and GTX Titan used in [44] and [45] are shown in Table 6.2. They can represent the mainstream GPUs for the current generation. We can see that the current GPU architecture can provide Tera-FLOPS computation power and around 200 GB/s memory bandwidth.

Table 6.2: Technical specification of GTX 690 and GTX Titan. GTX 690 contains two GK104 GPU cores. The table shows the numbers for one GK104 core.

<table>
<thead>
<tr>
<th></th>
<th>GTX 690</th>
<th>GTX Titan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>28 nm</td>
<td>28nm</td>
</tr>
<tr>
<td>CUDA cores</td>
<td>1536</td>
<td>2688</td>
</tr>
<tr>
<td>Clock freq</td>
<td>915 MHz</td>
<td>837 MHz</td>
</tr>
<tr>
<td>GFLOPS (FMA)</td>
<td>2810</td>
<td>4500</td>
</tr>
<tr>
<td>Memory</td>
<td>6.0 Gbps 2GB GDDR5</td>
<td>6.0 Gbps 6GB GDDR5</td>
</tr>
<tr>
<td>Memory interface width</td>
<td>256-bit</td>
<td>384-bit</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>192.2 GB/s</td>
<td>288.4 GB/s</td>
</tr>
</tbody>
</table>

The technical specifications of the state-of-the-art mobile GPUs are listed in Table 6.3. Most of the current generation of mobile GPUs have a very similar unified shader architecture except for the NVIDIA Tegra 4. These mobile GPUs typically contain 64–128 ALU (arithmetic logic unit) cores (the ALU core in a mobile GPU has similar architecture to a CUDA core in NVIDIA desktop GPUs), providing more or less 100 GFLOPS performance, and around 10–15 GB/s memory bandwidth. From the technical specification, we can see that the state-of-the-art mobile GPUs provide about 1/10 the processing power of desktop GPUs in terms of the ALU cores. However, during the past several years, the mobile GPUs have been moving forward faster and the gap between the mobile GPUs and the desktop GPUs have been decreasing.
We can foresee a great potential of mobile GPUs to handle more computationally-intensive tasks in the next few years.

Table 6.3: Technical specifications of the major mobile GPUs.

<table>
<thead>
<tr>
<th>Mobile GPU</th>
<th>Adreno 330</th>
<th>Tegra 4</th>
<th>Mali T-628</th>
<th>PowerVR G6430</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>28 nm</td>
<td>28 nm</td>
<td>28 nm</td>
<td>28 nm</td>
</tr>
<tr>
<td>Shader Arch.</td>
<td>Unified</td>
<td>PS+VS</td>
<td>Unified</td>
<td>Unified</td>
</tr>
<tr>
<td>Compute units</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Shader cores)</td>
<td>4</td>
<td>4 PSs</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>(ALUs/core)</td>
<td></td>
<td></td>
<td>12/PS</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>6 VSs</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>ALU cores</td>
<td>128</td>
<td>72</td>
<td>96</td>
<td>128</td>
</tr>
<tr>
<td>Clock freq</td>
<td>450 MHz</td>
<td>672 MHz</td>
<td>533 MHz</td>
<td>450 MHz</td>
</tr>
<tr>
<td>GFLOPS (FMA)</td>
<td>130</td>
<td>96.8</td>
<td>102</td>
<td>110.4</td>
</tr>
<tr>
<td>Memory</td>
<td>LPDDR3</td>
<td>LPDDR3</td>
<td>LPDDR3</td>
<td>LPDDR3</td>
</tr>
<tr>
<td>Memory frequency</td>
<td>800 MHz</td>
<td>933 MHz</td>
<td>933 MHz</td>
<td>800 MHz</td>
</tr>
<tr>
<td>Memory interface width</td>
<td>32-bit dual-channel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>12.8 GB/s</td>
<td>14.9 GB/s</td>
<td>14.9 GB/s</td>
<td>12.8 GB/s</td>
</tr>
</tbody>
</table>

6.1.3 Comparisons of Accelerator Architectures

Performance

Turbo decoding and LDPC decoding algorithms have been well studied in terms of their fixed-point implementation. These algorithms do not require the full precision of floating-point to achieve good error-correction performance. Typical bit-representations of the messages or extrinsic values are only 6–10 bits, which significantly simplifies the arithmetic operations, as well as the data storage requirements. In addition, turbo decoding and LDPC decoding only require addition and multiplexing logic thanks to the simplified algorithms, at a very small and tolerable BER performance loss. Therefore, turbo decoding and LDPC decoding can both be imple-
mented using efficient customized hardware logic.

On the other hand, both of the mobile computer vision algorithms we studied require certain complicated operations, even transcendental functions, during the computing processing. For example, the object removal algorithm requires multiplication to compute SSD (sum of squared differences). The SIFT algorithm requires multiplication, \( \exp \), \( \text{atan2} \) and also division. Although some of them may be implemented using look-up table, the performance and accuracy of the keypoints will be affected. Under such circumstances, it is hard to utilize the VLSI architecture to implement these computer vision algorithms. Although some of the transcendental functions can be implemented using a look-up table, which still depends on the specific input workloads, the flexibility of the implemented accelerators will be significantly affected. On the other hand, the mobile CPU and GPU usually provide libraries to support transcendental functions, some of which may also be provided as hardware-accelerated intrinsic functions. For example, a modern GPU provides intrinsic functions such as \texttt{fast_recip} (reciprocal), \texttt{fast_exp} (exponential), \texttt{fast_tan}, \texttt{fast_log} and so on. These intrinsic functions can either be implemented using dedicated SFU (special function unit), or emulated by using the basic instructions such as the \texttt{mad} (multiply-add) operation.

Memory architecture

A customized VLSI hardware accelerator is able to provide very high speed on-chip memory access, in which the on-chip memory is implemented using SRAM. Two major advantages of the memory architecture in customized hardware are high performance and adaptability. The SRAM for the customized hardware design can fulfill new data requests in every clock cycle. This feature fulfills the very high memory bandwidth requirement of high throughput turbo decoding. In addition, the size, layout and interface of the memory systems can be tailored to exactly fit the systems. Therefore,
we can reduce the chip area and power consumption in the memories. However, the memory controller logic must be carefully designed to avoid possible memory bank conflicts, otherwise, the performance and sustainable memory bandwidth will be significantly reduced. Therefore, very fine-grain data access patterns should be analyzed to improve the efficiency of the memory architecture for customized hardware.

The mobile CPU and GPU both have hierarchical memory architectures, including slow but large system memory, fast but small on-chip memory, memory cache and so on. To fully utilize the computation units, cache or on-chip memory should be exploited to enable data sharing and reuse, so that we can reduce the number of global memory transactions and at the same time try to use fast on-chip memory as much as possible. High level analysis of the memory access pattern is usually performed to take advantage of the hierarchical memory architecture.

**Flexibility**

The demand for high mobility in modern mobile computing systems always positions these systems in frequently and fast changing environments. This requires the mobile computing systems to be reconfigured with a set of configurations and parameters to adapt to the complicated environments. Moreover, with the flexible architecture provided, new features and new standards can be supported by software upgrade, reducing the overall cost and extending the value and the life time of the mobile devices. Therefore, the design of a highly configurable architecture with reasonable cost is preferable.

When it comes to flexibility, the software-based accelerators such as mobile GPU are obviously the winners. New standards, new algorithms, and new parameters can be adopted by the software implementation instantaneously. As discussed in previous chapters, our proposed GPU-based LDPC decoding can easily support multiple standards, different decoding algorithms, different block sizes, code rates and so on, with
very minimal design effort. The CPU-GPU heterogeneous architecture of computer vision accelerators allows us to fully explore design spaces with only a few parameter changes.

Similarly, as wearable smart devices become popular, more and more advanced computer vision algorithms will become essential in such systems. No single chip can support so many different algorithms. Programmable heterogeneous accelerators will be the key to enabling these computationally-intensive algorithms in a single chip.

To achieve a similar level of configurability, we need to spend much more effort on ASIC implementations. As we have shown in the parallel interleaver design in turbo decoding, we may provide a contention-free interleaver for certain turbo decoding architectures or certain interleaving algorithms. However, to support more general architectures and algorithms, it is non-trivial to design a contention-free buffer system enabling very high throughput, while only adding a small overhead. In addition, the supported functions are determined at the design stage. Therefore, once deployed, the systems cannot accept any upgrade to introduce new features to the existing system.

**Cost and Design productivity**

The very high cost and long design cycle of customized VLSI hardware makes the software-based accelerators more attractive. Instead of designing the hardware architecture for every algorithm, existing hardware plus programming language support allows us to quickly generate software-based solutions in a short time. To design a high performance VLSI architecture for turbo decoders and LDPC decoders, the entire design process (including floating-point simulation, fixed-point simulation, hardware modeling, simulation, verification, and so on) typically requires several months to finish. As a comparison, the software-based accelerator can be quickly designed, debugged and implemented in just a few weeks.

The short design time is especially promising for many advanced computer vision
use cases. Although we still cannot achieve real-time processing for some advanced algorithms with the current generation of hardware, the speedup we achieved by applying the heterogeneous CPU-GPU acceleration already makes these systems usable (or even satisfying) on mobile devices. In most of the cases, there is no need to spend much more effort for real-time processing, especially when the time-to-market factor becomes more and more important.

6.2 Summary

In this section, we summarize the design space exploration and performance comparison based on the analysis from the previous chapters and sections. We first summarize the use cases we studied in the previous chapter by comparing their algorithm characteristics and the design choices. Then we present the design guidelines of mapping parallel algorithms onto different parallel accelerators architectures.

Table 6.4 summarizes the key characteristics of the four use cases we presented in detail in the previous chapters. By carefully partitioning the algorithms and analyzing the algorithm properties, we can identify the performance bottleneck of an algorithm, and apply appropriate optimizations to eliminate the performance bottlenecks.

Based on the algorithm analysis and architecture comparison, we now can provide some general guidelines of how to map algorithms onto mobile platforms. For algorithms with very high parallelism degrees, GPGPU is a potentially good choice to accelerate the algorithm. If the memory access is regular, we can easily map the memory accesses by using SIMD instructions and reduce the number of memory transactions. If the algorithms show very irregular memory access behaviors, we can take advantage of the GPU on-chip memory to reorganize the memory data requests to improve the memory performance.

For the algorithms requiring very high arithmetic intensity but having low par-
allelism such as turbo decoding, the current software-based parallel solutions still cannot provide enough computation power to achieve performance goals. For these applications, customized logic is still the way to go. However, in use cases such as turbo decoding, the irregular memory accesses cause extra overhead for the hardware design and implementation. Extra effort should be made to eliminate memory conflict problems to achieve the expected performance goal. Since software-based parallel architectures such as multi-core CPU and GPU can easily handle irregular memory accesses with on-chip cache memory, as these processors are gaining more computation power, we expect to see high performance parallel turbo decoding on CPU or GPU.

The regular memory access patterns of computer vision algorithms make them suitable for CPU or GPU implementations. Among computation kernels, raw pixel manipulation and block-based image processing are especially good fits for the mobile GPU’s architecture, because they provide sufficient parallelism to fully occupy the parallel architecture. On the other hand, some computation kernels with many branching operations or changes in flow control show very low parallelism. Therefore, they show better performance when executing on the mobile CPU. Because most of the advanced computer vision algorithms are a mix of the above-mentioned computation kernels, the heterogeneous CPU-GPU accelerator can provide the best performance and energy efficiency.
Table 6.4: Summary of design space exploration and accelerator choice for the presented case studies.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parallelism degree</th>
<th>Arithmetic complexity requirement</th>
<th>Memory bandwidth requirement</th>
<th>Memory access pattern</th>
<th>Suitable for GPU acceleration?</th>
<th>Bottleneck analysis and solutions</th>
<th>Accelerator choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turbo decoding</td>
<td>Low: ~100</td>
<td>3114 GOPS</td>
<td>108.3 GB/s</td>
<td>Irregular</td>
<td>No</td>
<td>Low parallelism, Very irregular memory access, GPU cannot meet B/W requirements.</td>
<td>ASIC</td>
</tr>
<tr>
<td>LDPC decoding</td>
<td>High: ~1000</td>
<td>570 GOPS</td>
<td>840 GB/s</td>
<td>Irregular</td>
<td>Yes</td>
<td>Memory bandwidth was bottleneck. Memory access optimization significantly improves performance.</td>
<td>ASIC or GPU</td>
</tr>
<tr>
<td>Object removal</td>
<td>High: ~10^5</td>
<td>76.8 GFLOPS</td>
<td>5.69 GB/s</td>
<td>Regular</td>
<td>Yes</td>
<td>Before optimization, too complex for GPU. After optimization, suitable for GPU.</td>
<td>Mobile GPU</td>
</tr>
<tr>
<td>SIFT</td>
<td>High: ~10^4</td>
<td>5.2 GFLOPS</td>
<td>13.75 GB/s</td>
<td>Regular &amp; irregular</td>
<td>Yes for parts of the algorithm</td>
<td>High computational complexity. Offload computations from CPU to GPU. CPU+GPU heterogeneous implementation.</td>
<td>Mobile CPU+GPU</td>
</tr>
</tbody>
</table>
Modern mobile SoCs have become heterogeneous platforms consisting of customized hardware logic, mobile CPU, mobile GPU, and other accelerators. Given such a heterogeneous architecture, one has more choices to deploy applications and at the same time how to map algorithms onto different accelerators becomes important for overall system performance and hardware efficiency.

In the previous sections, we have analyzed algorithm properties and performance requirements in detail for several important workloads for wireless communication modem and computer vision algorithms on mobile devices, including turbo decoding, LDPC decoding, object removal and the SIFT algorithm. We also compared the architectural specifications for the major accelerator architectures on mobile devices. With algorithm analysis information and architecture specifications, let us revisit Fig. 1.2. Based on the discussion in the previous three chapters, we have proposed how to map the four algorithms (turbo decoding, LDPC decoding, object removal, and the SIFT) onto different accelerator architectures. Fig. 6.1 demonstrates the algorithm mapping results based on the parallelism and memory access pattern.

If the performance requirement is very high, algorithms such as turbo decoding and LDPC decoding are required to be implemented using customized hardware logic. The irregular memory accesses in turbo decoding and in LDPC decoding always complicate the hardware design significantly. As alternative low-cost and reconfigurable solutions, GPU architecture can be considered as accelerators for these decoders. However, the low parallelism in the turbo decoding algorithm makes it unsuitable for GPU acceleration. As a comparison, the LDPC decoding algorithm shows inherent parallelism, therefore it can take advantage of the GPU architecture and achieve good throughput performance, as well as good flexibility.

The object removal and SIFT algorithms contain very regular memory accesses to pixel data, which can be efficiently reused and prefetched. The object removal algo-
Figure 6.1: Algorithm properties and architecture offloading to mobile accelerators.

Algorithm and parts of the SIFT algorithm show high parallelism and reasonable memory bandwidth requirement, which are good features that make these algorithms suitable for GPU-based parallel implementations. Some modules in the SIFT algorithm contain mostly serial code and show very low parallelism, we choose to map them onto mobile CPU to take advantage of the CPU’s powerful flow control capability and to minimize data transfer effort between the mobile CPU and GPU.

For a general algorithm to be implemented on mobile SoC systems, we can predict the performance of algorithms implemented using different architectures, by generating algorithm complexity and memory bandwidth models, and comparing them with architecture specifications. The comparison between the performance requirements and the predicted performance will help us determine the appropriate architecture to map. In addition to this, there are also some general rules of mapping algorithms.
onto parallel architectures. Firstly, algorithms with high parallelism degrees may take advantage of the GPU massively parallel architecture and distribute the algorithms on to threads to collaboratively finish the computing tasks. Secondly, the memory access pattern makes a big impact on the memory reading/writing behaviors. The irregular memory accesses are usually performance bottlenecks for high speed systems, due to the poor data locality. The irregular memory accesses always cause severe memory conflicts and bandwidth waste, leading to a degraded overall performance. On the other hand, the regular memory accesses allow us to transfer data between the parallel producers and memory modules efficiently. Data reuse and data locality can be explored to increase the memory access performance.

Considering that the wireless communication and mobile computer vision systems in modern mobile computing platforms contain many computationally-intensive and very diverse algorithms, there is really no single accelerator that can handle all workloads with optimized performance. By carefully analyzing algorithms and identify the performance requirements as well as some architecture-related properties such as memory access patterns, we can distribute different algorithms across heterogeneous accelerators, and take advantage of the benefits of each accelerator’s architecture to improve better efficiency and high performance.
Chapter 7

Conclusions

7.1 Conclusion of the Current Results

In this thesis, we explore the design space and optimization strategies for parallel algorithms and architecture of wireless communication and mobile computing systems. We mainly focus on studying the capability of GPGPU as accelerators to speedup different algorithms. We showed that GPU is not always the suitable platform to accelerate the computationally-intensive algorithms, due to some limitations in the algorithm and possible mismatching between the algorithm and the GPU architecture.

First, we propose a methodology to analyze algorithms and determine the algorithm performance bottleneck with a compute device. Algorithm parallelism, arithmetic complexity, memory bandwidth requirements, and memory access patterns are the most important factors to affect the achievable performance of a certain algorithm. The proposed methodology can be used to predict the performance of an algorithm before we really implement the algorithms in a real hardware platform.

Second, we propose a GPU-based implementation of LDPC decoding. We focus on arithmetic optimization, memory system optimization, architectural optimization,
data structure optimization and so on. Experimental results show that the proposed LDPC decoder is able to provide high throughput but also leads to very low decoding latency, therefore, the GPU is a good candidate for future SDR systems.

Since the GPU cannot provide enough arithmetic resources and memory bandwidth, GPU implementations show very low decoding throughput for the turbo decoding algorithm. Although we proposed many optimization schemes to simplify the algorithm and adapt the algorithm to the GPU architecture, the performance is still limited by the inherent algorithm bottlenecks. To meet the very high performance goal, we propose a VLSI-based parallel interleaver design for high throughput multi-standard turbo decoding systems. We study the statistical properties of the memory conflict problem, and propose a contention-free memory architecture to solve the memory conflict problem efficiently for parallel turbo decoding. Thus, the proposed memory architecture can efficiently solve the memory conflict problem for highly parallel turbo decoders supporting multiple standards, and can provide great flexibility and configurability.

We then present mobile GPU-based computer vision accelerators. Two case studies are discussed: the object removal algorithm and the SIFT algorithm. According to mobile processors and mobile GPU architectures, we propose several optimizations to fully utilize the compute units and memory bandwidth. The proposed optimization techniques reduce the arithmetic complexity of the algorithm and reduce the memory bandwidth requirements, and they make the algorithm feasible on a current generation mobile GPU. Moreover, we also demonstrate that the computer vision algorithms can be significantly accelerated by using a heterogeneous mobile CPU-GPU architecture in the SIFT algorithm use case.

Finally, we study characteristics of the proposed parallel algorithms in depth and analyze the algorithm performance requirements. We show that based on the char-
acteristics of algorithms and the architecture of accelerators, different accelerators may be chosen. Therefore, to speed up computationally-intensive applications such as wireless communications and computer vision, we should take advantage of the heterogeneous architecture of modern mobile SoC processors.

7.2 Future Work

The following problems can be further investigated as future work:

1. The proposed methodology can be further extended to incorporate more factors such as cache systems, data dependency, memory access locality and so on, so that more general models can be achieved to predict the performance of more algorithms. More specifically, for mobile computing, power/energy consumption is very crucial. We may be able to improve our methodology by considering the power consumption as an additional factor.

2. Many signal processing algorithms targeting the same field have similar computation kernels and can be categorized into several classes. Similar computation kernels apparently share many properties and can be parallelized in a similar way. By optimizing these common computation kernels, we may be able to determine the acceleration speedup or overhead for different architectures. Since we can partition a signal processing algorithm by using these common computation kernels, we can derive a performance model and complexity requirements for more general signal processing algorithms. Then, by analyzing an algorithm and identifying its kernel computations, we can predict the performance and possible speedup by applying certain optimizations.

3. In a wireless communication modem, there are other computationally-intensive blocks such as MIMO (multiple-input multiple-output) detection and FFT (fast
Fourier transform). Similarly, there exist many other computer vision algorithms with computation kernels in other formats. A more complete study would include major wireless communication blocks and a set of representative computer vision algorithms. A more general conclusion may be drawn by analyzing these algorithms from a system perspective.

4. In LDPC decoding algorithm, the shuffle network defined by the parity-check matrix $H$ can also cause memory conflict problems in a hardware design. Similarly, the memory conflict problem makes the shuffle network complicated and may decrease the throughput performance. We can extend the proposed interleaver architecture to support contention-free memory access in an LDPC decoder. This will enable very high throughput LDPC decoding with low hardware complexity. Then, based on this, a contention-free memory architecture can be designed for the unified turbo/LDPC decoding, allowing more hardware sharing and higher efficiency in future 5G and beyond wireless communication systems.


