UNIVERSITÉ DE MONTRÉAL

NEAR DETERMINISTIC SIGNAL PROCESSING USING GPU, DPDK, AND MKL

ROYA ALIZADEH
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
JUIN 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

NEAR DETERMINISTIC SIGNAL PROCESSING USING GPU, DPDK, AND MKL

présenté par : ALIZADEH Roya
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Prof. Yvon Savaria for his unwavering support and guidance. I appreciate his patience and positive attitude which has helped me overcome difficult challenges during my studies. I thank him for providing me with excellent conditions to do my master studies at Ecole Polytechnique of Montreal.

I must thank Dr. Normand Bélanger for his professional advice and feedback during my research work. Our productive discussions helped me elevate the quality of my research work and improve my writing skills. I should also thank him for helping me with French translations in this thesis.

I owe a great debt of gratitude to Prof. Jean-François Frigon for his valuable comments for the completion of this thesis. I would like to thank Prof. Guchuan Zhu for having productive discussions during my research work. I must thank Prof. Michael Corinthios because of his offering of graduate courses which helped me strengthen my academic skills.

I would like to thank all of my great friends in Montreal for accompanying me and supporting me during my studies.

Last but not the least, my greatest and deepest gratitude goes to my family who have made many sacrifices in their lives for me.

RÉSUMÉ

En radio définie par logiciel, le traitement numérique du signal impose le traitement en temps réel des donnés et des signaux. En outre, dans le développement de systèmes de communication sans fil basés sur la norme dite Long Term Evolution (LTE), le temps réel et une faible latence des processus de calcul sont essentiels pour obtenir une bonne expérience utilisateur. De plus, la latence des calculs est une clé essentielle dans le traitement LTE, nous voulons explorer si des unités de traitement graphique (GPU) peuvent être utilisées pour accélérer le traitement LTE. Dans ce but, nous explorons la technologie GPU de NVIDIA en utilisant le modèle de programmation Compute Unified Device Architecture (CUDA) pour réduire le temps de calcul associé au traitement LTE. Nous présentons brièvement l'architecture CUDA et le traitement parallèle avec GPU sous Matlab, puis nous comparons les temps de calculs avec Matlab et CUDA. Nous concluons que CUDA et Matlab accélèrent le temps de calcul des fonctions qui sont basées sur des algorithmes de traitement en parallèle et qui ont le même type de données, mais que cette accélération est fortement variable en fonction de l'algorithme implanté.

Intel a proposé une boite à outil pour le développement de plan de données (DPDK) pour faciliter le développement des logiciels de haute performance pour le traitement des fonctionnalités de télécommunication. Dans ce projet, nous explorons son utilisation ainsi que celle de l'isolation du système d'exploitation pour réduire la variabilité des temps de calcul des processus de LTE. Plus précisément, nous utilisons DPDK avec la Math Kernel Library (MKL) pour calculer la transformée de Fourier rapide (FFT) associée avec le processus LTE et nous mesurons leur temps de calcul. Nous évaluons quatre cas : 1) code FFT dans le cœur esclave sans isolation du CPU, 2) code FFT dans le cœur esclave avec l'isolation du CPU, 3) code FFT utilisant MKL sans DPDK et 4) code FFT de base. Nous combinons DPDK et MKL pour les cas 1 et 2 et évaluons quel cas est plus déterministe et réduit le plus la latence des processus LTE. Nous montrons que le temps de calcul moyen pour la FFT de base est environ 100 fois plus grand alors que l'écart-type est environ 20 fois plus élevé. On constate que MKL offre d'excellentes performances, mais comme il n'est pas extensible par lui-même dans le domaine infonuagique, le combiner avec DPDK est une alternative très prometteuse. DPDK permet d'améliorer la performance, la gestion de la mémoire et rend MKL évolutif.

ABSTRACT

In software defined radio, digital signal processing requires strict real time processing of data and signals. Specifically, in the development of the Long Term Evolution (LTE) standard, real time and low latency of computation processes are essential to obtain good user experience. As low latency computation is critical in real time processing of LTE, we explore the possibility of using Graphics Processing Units (GPUs) to accelerate its functions. As the first contribution of this thesis, we adopt NVIDIA GPU technology using the Compute Unified Device Architecture (CUDA) programming model in order to reduce the computation times of LTE. Furthermore, we investigate the efficiency of using MATLAB for parallel computing on GPUs. This allows us to evaluate MATLAB and CUDA programming paradigms and provide a comprehensive comparison between them for parallel computing of LTE processes on GPUs. We conclude that CUDA and Matlab accelerate processing of structured basic algorithms but that acceleration is variable and depends which algorithm is involved.

Intel has proposed its Data Plane Development Kit (DPDK) as a tool to develop high performance software for processing of telecommunication data. As the second contribution of this thesis, we explore the possibility of using DPDK and isolation of operating system to reduce the variability of the computation times of LTE processes. Specifically, we use DPDK along with the Math Kernel Library (MKL) provided by Intel to calculate Fast Fourier Transforms (FFT) associated with LTE processes and measure their computation times. We study the computation times in different scenarios where FFT calculation is done with and without the isolation of processing units along the use of DPDK. Our experimental analysis shows that when DPDK and MKL are simultaneously used and the processing units are isolated, the resulting processing times of FFT calculation are reduced and have a near-deterministic characteristic. Explicitly, using DPDK and MKL along with the isolation of processing units reduces the mean and standard deviation of processing times for FFT calculation by 100 times and 20 times, respectively. Moreover, we conclude that although MKL reduces the computation time of FFTs, it does not offer a scalable solution but combining it with DPDK is a promising avenue.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

3GPP	3rd Generation Partnership Project
AM	Acknowledged Mode
AS	Access Stratum
DL	Downlink
DPDK	Data Plane Development Kit
FE	Full Expansion
FEC	Forward Error Correction
FPFSD	Fixed-complexity sphere decoder
GPU	Graphics processing unit
LLRs	Log-likelihood ratios
LTE	Long Term Evolution
MAC	Media Access Control
MIMO	Multiple Input Multiple Output
MKL	Math Kernel Library
MPI	Message Passing Interface
SDR	Software Defined Radio
NMM	Network-Integrated Multimedia Middleware
PDCP	Packet Data Convergence Protocol
PDU	Protocol Data Unit
PHY	Physical layer
RLC	Radio Link Control
RRC	Radio Resource Control
SDU	Service Data Unit
SE	Single-path search
SNR	Signal to Noise Ratio
SSFE	Selective spanning with fast enumeration
TM	Transparent Mode
UE	User Equipment
UM	Unacknowledged Mode
UP	Uplink

CHAPTER 1

INTRODUCTION

Recent advances in information and communication technology have drawn attention from the telecommunication industry to more efficient implementation of wireless standards. Specifically, a great deal of attention is dedicated to develop a form of Software Defined Radio (SDR) that performs different signal processing tasks over the telecommunication cloud [2]. However, performing the needed complex operations involved in the modern cellular technologies and meeting the real time constrains impose needs for extra computational resources, which may not be available in current telecommunication infrastructure.

The most recently deployed cellular technology, the Long-Term Evolution (LTE) standard, is an example of such complex systems. Real time and low latency computations are critical aspects in cloud-based implementation of LTE, which involves many challenges in practice. Specifically, the design and implementation of real-time and low-latency wireless systems aims at achieving two goals : first reducing the latency by recognizing its time consuming parts ; second proposing solutions to reduce the variability (randomness) in the processing times. In this thesis, we address the following objectives :

1. identifying computational bottlenecks in the LTE ;
2. reducing the computational latency to increase performance ;
3. implementing (near) real-time processing algorithms by reducing variability of computation times ;
4. studying the computational performance of different modules of LTE when implemented on graphics processing units ;
5. studying the computational performance of LTE modules for implementation on central processing units using different implementation tools.

Academia and industry have shown interest in using Graphics Processing Units (GPUs) for accelerated implementation of different applications. Essentially, GPUs are widely used for accelerating computation times because they are offering large number of processing cores and high memory bandwidth. For instance, Geforce GTX 660 Ti offers 7 multi-processors and 192 independent cores for each multi-processor. That is very promising and suggests possible accelerations of 1000 times using GPU implementation. In spite of such large number of available processing elements, we could never get acceleration larger than 10 and in many case we got no acceleration at all. That is why we looked at other acceleration methods to verify the efficiency. Moreover, Intel has also provided a Data Plane Development Kit (DPDK) [3] and a Math Kernel Library (MKL) [4] for low latency processing of telecommunication tasks over more conventional central processing units.

1.1 Contributions

In this thesis, we study different parts of the LTE standard and identify the time consuming portion which may act as computational bottlenecks in an implementation. Then, we propose different solutions for parallel computing of those computational bottlenecks and discuss their performance. Specifically, we study the implementation of different matrix and vector operations over graphical processing units and central processing units. For each case, we propose different implementation approaches and evaluate their efficiency by comparing their computation times. We use the fast Fourier transform (FFT) as a benchmark for many of our analysis as it is found to be one of the largest computational burdens for LTE. As it was discussed earlier, our primary goal is to achieve *near-deterministic* and *low latency* processing times for LTE operations.

This thesis focus on studying different means of performing some complex parts of LTE (specifically the FFT) and its main contributions can be summarized as follows :

– implementation of FFT, convolution, matrix multiplication and inversion using CUDA on GPUs,
– implementation of FFT, matrix multiplication and inversion using MATLAB on GPUs,
– analysis of related results and suitability of CPUs for supporting LTE tasks,
– implementation of FFT using Intel MKL on CPUs,
– implementation of FFT using DPDK and MKL on CPUs with and without the aid of isolation of processing units,
– comprehensive analysis of FFT implementation on CPUs in different scenarios.

1.2 Organization

This thesis is organized as follows. In Chapter 2, we present a comprehensive review of the literature on parallel processing. Further, we review the related literature about DPDK as a solution for parallel processing potentially useful in this thesis. In Chapter 3, we provide a review of the LTE standard, discussing its different layers and its main functions. Specifically, we describe the structure of orthogonal frequency division multiplexing (OFDM) using FFT and inverse FFT operations. Turbo decoding and MIMO (multiple input and multiple output) detection mechanisms are also described in Chapter 3. Further, we describe the GPU programming model for parallel programming. Chapter 3 concludes by introducing MKL and DPDK.

In Chapter 4, we discuss different algorithms for implementation of FFT (including Cooley-Tukey and Stockham algorithms), matrix inversion, convolution and cross-correlation. These algorithms are used in the following chapters for implementation on the hardware devices.

In Chapter 5, we describe parallel implementations of different operations on GPUs. We use MATLAB and CUDA for implementation of dense matrix multiplication, FFT, matrix inversion, convolution and addition. Our experimental results for implementation of these operations on GPUs are also presented in this chapter.

Although GPUs include large number of cores and computation elements, we could not get high percentage of acceleration using them. For that reason, in Chapter 6, we study the feasibility of using DPDK and MKL to achieve near deterministic computation for LTE processes. Our experimental results show different performance in terms of the mean and variance of the computation times when DPDK and MKL libraries are used for isolation of the central processing unit. Our concluding remarks and future directions are presented in Chapter 7.

CHAPTER 2

LITERATURE REVIEW

LTE supports and takes advantage of a new modulation technology based on Orthogonal Frequency Division Multiplexing (OFDM) and Multiple Input Multiple Output (MIMO) data transmission. The benefits of LTE come from increased data rates, improved spectrum efficiency obtained by spatially multiplexing multiple data streams [5], improved coverage, and reduced latency, which makes it efficient for current wireless telecommunication systems. Since MIMO systems increase the complexity of the receiver module, high-throughput practical implementations that are also scalable with the system size are necessary. To reach these goals, we explore two solutions to overcome LTE computation latency, which are parallel processing using GPU and DPDK.

2.1 Parallel processing to reduce computation time of LTE

GPUs have been recently used to develop reconfigurable software-defined-radio platforms [6, 7], high-throughput MIMO detectors [8, 9], and fast low-density parity-check decoders [10]. Although multicore central processing unit (CPU) implementations could also replace traditional use of digital signal processors and field-programmable gate arrays (FPGAs), this option would interfere with the execution of the tasks assigned to the CPU of the computer, possibly causing speed decrease. Since GPUs are more rarely used than CPUs in conventional applications, their use as coprocessors in signal-processing systems needs to be explored. Therefore, systems formed by a multicore computer with one or more GPUs are interesting in this context. In [11], the authors implement signal processing algorithms suitable with parallel processing properties of GPUs to decrease computation time of multiple-input–multiple-output (MIMO) systems. They develop a novel channel matrix preprocessing stage for MIMO systems which is efficiently matched with the multicore architecture.

Work is needed to decrease the run time for those configurations not attaining real-time performance. The use of either more powerful GPUs or more than one GPU in heterogeneous systems may be promising solutions for this purpose. Another interesting topic for research is to analyze the amount of energy consumed by the proposed GPU implementations. According to [11], channel Matrix preprocessing is well matched with GPU architectures and it reduces the order of computational complexity. According to [12], distributed Multimedia Middleware transparently combines processing components using CPUs, as well as local and remote GPUs for distributed processing. This middleware uses zero-copy to choose the best possible mechanism for exchanging data.

In [13], an effective and flexible N-way MIMO detector is implemented on GPU, two important techniques are used (instead of Maximum likelihood detection) : depth-first algorithms such as depth-first sphere detection and breadth-first algorithms such as the K-best. In depth-first sphere detection algorithm, the number of tree nodes visited vary with the signal to noise ratio (SNR). The K-best detection algorithm has a fixed throughput because it searches a fixed number of tree nodes independent of SNR. Compared to ASIC and FPGA, this implementation is attractive since it can support a wide range of parameters such as modulation order and MIMO configuration. In the second technique, selective spanning with fast enumeration (SSFE) and different permuted antenna-detection order are used in order to be well suited for GPUs. Moreover, modified Gram-Schmidt Orthogonalization to perform QR decomposition is implemented. This implementation performs MIMO detection on many subcarriers in parallel using hundreds of independent thread-blocks to achieve high performance by dividing the available bandwidth into many orthogonal independent subcarriers.

In [14], a 2×2 MIMO system using a GPU cluster as the modem processor of an SDR system is implemented for the purpose of exploiting additional parallel processing capabilities over a single GPU system. Moreover, a 3-node GPU cluster using MPI-based distributed signal processing is applied to some modules that need relatively large amounts of computational capacity such as the frame synchronization module, the MIMO decoder, and the Forward Error Correction (FEC) decoder (Viterbi algorithm). It is only applied to WiMAX data. However, the clustered MPI-based signal processing efficiency achieved in the WiMAX system would be applicable to LTE systems. In [14], a Software Defined Radio Base Station (SDR BS) is implemented using two-level parallelism. One level of parallelism is obtained by the distributed processing provided by MPI and the other level of parallelism is the parallel processing performed at each node using a GPU.

Based on [15], it can be concluded that matrix inversion is a bottleneck if MMSE-based MIMO detectors were to be implemented on GPUs. In [15], each matrix of size $N \times N$ is mapped to N threads. In this approach, each thread reads N elements in a single matrix from the shared memory, and N threads process a matrix inversion in parallel. In addition, multiple matrices can be inverted simultaneously in a block. Finally in a grid which is composed of several blocks, many matrices are processed, thus speeding up the algorithm. In this paper, two kinds of data transfers (synchronous and asynchronous) are considered. In the synchronous model, the kernel is executed after the data has been transferred completely. This paper mentions that frame synchronization, MIMO decoding, and Forward Error Correction (FEC) decoding are the most time consuming tasks of LTE. Another time consuming part of frame synchronization is cross-correlation. Frame synchronization on GPU has not been implemented yet. This work applied clustered MPI-based signal processing to WiMAX not to LTE.

The authors in [15] present an MMSE-based MIMO detector on GPU. Optimization strategies have been proposed to compute channel matrix inversion, which is the heaviest computational load in the MMSE detector. A Gaussian elimination approach with complete pivoting is employed to compute the matrix inversion. In [16] the authors mention that to improve coverage and increase data rate, LTE requires a very short latency of less than 1 millisecond across the backhaul network.

The authors in [17] explain that the high data rates of LTE enable interactive multimedia applications over networks. They investigated the performance of 2D and 3D video transmission over LTE by combining bandwidth scalability and admission control strategies in LTE networks.

In [18], the authors argue that minimizing the system and User Equipment (UE) complexities are the main challenges of LTE. It allows flexible spectrum deployment in LTE frequency spectrum as well as enabling co-existence with other 3GPP Radio Access Technologies (RATs). Also, they mention that load imbalance reduces LTE network performance because of non-uniform user deployment distribution. Load balancing techniques are proposed in this paper to improve network performance.

The authors in [8] develop a 3GPP LTE compliant turbo decoder accelerator on GPU. The challenge of implementing a turbo decoder is finding an efficient mapping of the decoder algorithm on GPU, e.g. finding a good way to parallelize workload across cores that allocates and uses fast on-die memory to improve throughput. This paper increases throughput through 1) distributing the decoding workload for a codeword across multiple cores, 2) decoding multiple codewords simultaneously to increase concurrency and 3) employing memory optimization techniques to reduce memory bandwidth requirements. In

addition, it also analyzes how different MAP algorithm approximations affect both throughput and bit error rate (BER) performance of decoders.
For simulation, the host computer first generates random 3GPP LTE Turbo codewords. After BPSK modulation, input symbols are passed through the channel with additive white Gaussian noise (AWGN), the host generates LLR values based on the received symbols which are fed into the Turbo decoder kernel running on a GPU.

2.1.1 Summary

As this study shows, the LTE challenges are : 1) a latency requirements of less than 1ms [16], 2) minimizing the system and User Equipment (UE) complexities, 3) allowing flexible spectrum deployment, 4) increasing capacity, 5) improving QoS, 6) enabling co-existence with other 3GPP Radio Access Technologies (RATs) [18], 7) scalability [17], and load balancing [18]. Since, implementation of LTE functions in data centers requires computing resources in wireless networks, it leads to more advanced algorithms and signal processing as well as load balancing and multi-threading.

Centralized radio access network needs to leverage massive parallel computing in order to increase data rate and decrease latency. Table 2.1 summarizes reported analysis and means of dealing with LTE time consuming tasks, which include FFT/IFFT in OFDM [19], matrix inversion in MIMO detection [15], convolution and cross correlation in channel model [19, 15].

Table 2.1 LTE time consuming tasks.

LTE time consuming tasks	Matrix Computation	Reference
OFDM	FFT, IFFT	[19]
MIMO detection	Matrix Inversion	[15]
Channel model, FEC and Turbo Decoding, Frame Synchronization	Convolution, Cross Correlation	[19, 15]

2.2 Reducing computation time of LTE using DPDK

All literature on DPDK that was found, relates to packet forwarding in layers two and three (L2 and L3). There is nothing related to computation and mathematical functions. In [20], the authors apply a combination of programmable hardware, general purpose processors, and Digital Signal Processors (DSPs) into a single die to improve the cost/performance trade off. Authors in [21] suggest to use Cloud infrastructure Radio Access Network (C-RAN) in two kinds of centralization (full and partial) to provide energy efficient wireless networks. The major disadvantage of this architecture is the high bandwidth and low latency requirements between the data center and the remote radio heads. Authors in [2] and [22] mention that SDN requires specific levels of programmability in the data plane and the Intel DPDK is a promising approach to improve performance in cloud computing applications. DPDK is proposed to enhance operating systems running on General Purpose Processors (GPPs) that already have some real-time capability.

Based on [2] DPDK proposes high performance packet processing. The authors in this paper propose Open flow 1.3 to implement the data plane. The authors also mention that the packet I/O overhead, buffering to DRAM, interrupt handling, memory copy and the overhead of kernel structures cause extra costs and delays, while using DPDK overcomes these bottlenecks. DPDK allows efficient transfer of packets between the I/O card and the code running in the user space. Transferring packets directly to L3 cache prevents to use high latency DRAMs. Thus DPDK increases performance.

2.3 Summary on literature review

The literature has shown ways to accelerate LTE with GPUs. It was reported that FFT/IFFT is the main time consuming function of OFDM and matrix inversion is a time consuming task associated with MIMO detection which can be possibly accelerated by GPU parallel processing. In fact the size of matrices for matrix inversion in LTE is small. By contrast, GPUs are more efficient for processing large number of data elements organized in a regular structure. Moreover, DPDK has a high performance packet processing capability. It helps implementing demanding applications on general purpose operating systems which have real-time capabilities.

CHAPTER 3

OVERVIEW OF LTE, GPU, DPDK

In this chapter we explain what is LTE. We describe layers of LTE and their main functionalities supported by the physical-layer processing blocks. As the main blocks of LTE in physical layer, we present OFDM and its implementation using FFT and IFFT. Also, we describe the Turbo decoder and MIMO detection. Since, GPUs are presented as a solution to reduce LTE latency, we describe the GPU programming model, and Geforce GTX 660 Ti specifications. MKL and DPDK features and usage are described as another solution to decrease latency.

3.1 LTE overview

Long term evolution (LTE) is based on the 3GPP standard that provides a downlink speed of up to 100 Mbps and an uplink speed of up to 50 Mbps. LTE brings many technical benefits to cellular networks. Bandwidth is scalable from 1.25 MHz to 20 MHz. This can suit the needs of different network operators that have different bandwidth allocations, and also allows operators to provide different services based on spectrum. LTE is also expected to improve spectral efficiency in 3G networks, allowing carriers to provide more data and voice services over a given bandwidth [1].

The LTE system architecture is based on the classical open system interconnect layer decomposition as shown in Fig. 3.3 (taken from [24]). Fig. 3.1 (taken from [23]) shows a high-level view of the LTE architecture. E-UTRAN (Evolved Universal Terrestrial Radio Access Network) and EPC (Evolved Packet Core) are two main components of LTE systems [23]. E-UTRAN is responsible for management of radio access and provides user and control plane support to the User Equipments (UEs). The user plane refers to a group of protocols used to support user data transmission, while control plane refers to a group of protocols to control user data transmission and managing the connection between the UE and networks such as handover, service establishment, resource control, etc. The E-UTRAN consists of only eNodeBs (eNBs) which provide user plane (PDCP/RLC/MAC/PHY) and control plane (RRC) protocol terminations toward the user equipment (UE). The eNBs are interconnected with each other by means of the X2 interface. The eNBs are also connected by means of the S1 interface to the Evolved Packet Core (EPC), more specifically to the Mobility Management Entity (MME) by means of the S1-MME interface and to the Serving Gateway (SGW) by means of the S1-U interface.

EPC is a mobile core network and its main responsibilities include mobility management, policy management and security. The EPC consists of the Mobility Management Entity (MME), the Serving Gateway (S-GW), and the Packet Data Network Gateway (P-GW). The MME is the control node for the LTE access network. It is responsible for user authentication and idle mode User Equipment (UE) paging and tagging procedure including retransmissions. The functions of the S-GW is to establish bearers based on the directives of the MME. The PGW provides Packet Data Network connectivity to E-UTRAN capable UEs using E-UTRAN only over the S5 interface. Both E-UTRAN and EPC are responsible for the quality-of-service (QoS) control in LTE. The x2 interface provides communication among eNBs including handover information, measurement and interface coordination reports, load measurements, eNB configuration setups and forwarding of user data. S1 interface connects the eNBs to the EPC. The interface between eNB and S-GW is called S1-U and is used to transfer user data. The interface between eNB and MME is called S1-MME and is used to transfer control-plane information including mobility support, paging data service management, location services and network management. Home Subscriber Server (HSS) and the Policy Control and Charging Rules Functions (PCRF) are considered to be parts of the LTE core network [23].

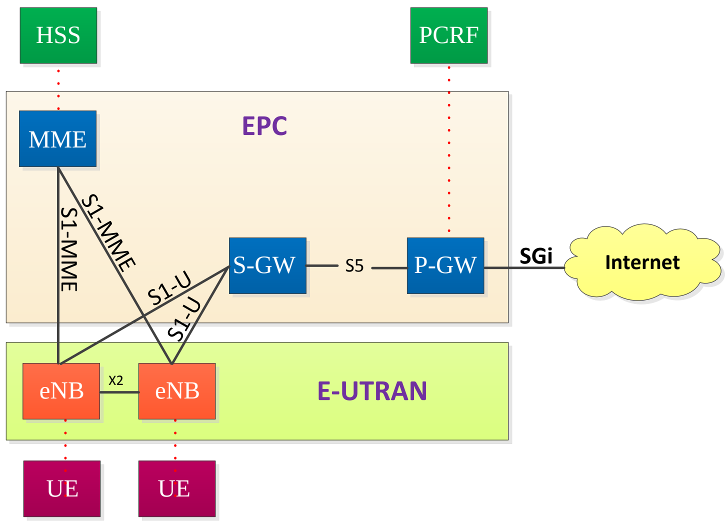


Figure 3.1 LTE system architecture.

Fig. 3.2 (taken from [24]) shows a diagram of the E-UTRAN Protocol Stack. Physical Layer carries all information from the MAC transport channels over the air interface. It takes care of link adaptation (AMC), power control, cell search (for initial synchronization and handover purposes) and other measurements (inside the LTE system and between systems) for the RRC layer. The Media Access Control (MAC) layer is responsible for mapping logical channels to transport channels. Also, it is resposible of Multiplexing the MAC SDUs from one or different logical channels onto transport blocks (TBs) to be delivered to the physical layer on transport channels. Demultiplexing of MAC SDUs from one or different logical channels from transport blocks (TBs) delivered from the physical layer on transport channels is another tasks performed by the MAC. Moreover, it schedules information reporting, corrects error through HARQ, handles priority between UEs by means of dynamic scheduling and between logical channels of one UE. The Radio Link Control (RLC) layer operates in 3 modes of operation : Transparent Mode (TM), Unacknowledged Mode (UM)[1], and Acknowledged Mode (AM)[2]. It is responsible for transfer of upper layer PDUs[3], error correction through ARQ (only for AM data transfer), concatenation, segmentation and reassembly of RLC SDUs[4] (only for UM and AM data transfer). RLC is also responsible for re-segmentation of RLC data PDUs (only for AM data transfer), reordering of RLC data PDUs (only for UM and AM data transfer), duplicate detection (only for UM and AM data transfer), RLC SDU discard (only for UM and AM data transfer), RLC re-establishment, and protocol error detection (only for AM data transfer). The main services and functions of the Radio Resource Control (RRC) sublayer include broadcast of System Information related to the non-access stratum (NAS)[5], broadcast of System Information related to the access stratum (AS)[6], paging[7], establishment, maintenance and release of an RRC connection between the UE and E-UTRAN, Security functions including key management, establishment, configuration, maintenance and release of point to point Radio Bearers. NAS protocols support the mobility of the UE and the session management procedures to establish and maintain IP connectivity between the UE and a PDN GW [24].

1. It does not require any reception response from the other party.
2. It requires ACK/NACK from the other party.
3. Protocol Data Unit (PDU) is information that is delivered as a unit among peer entities of a network and that may contain control information, such as address information, or user data.
4. Packets received by a layer are called Service Data Unit (SDU) while the packet output of a layer is referred to by Protocol Data Unit (PDU).
5. NAS is a functional layer in LTE stacks between the core network and user equipment. This layer is used to manage the establishment of communication sessions and for maintaining continuous communications with the user equipment as it moves.
6. AS is a functional layer in LTE protocol stacks between radio network and user equipment. It is responsible for transporting data over the wireless connection and managing radio resources.
7. The LTE network uses paging to notify UE in idle mode of an incoming connection requests.

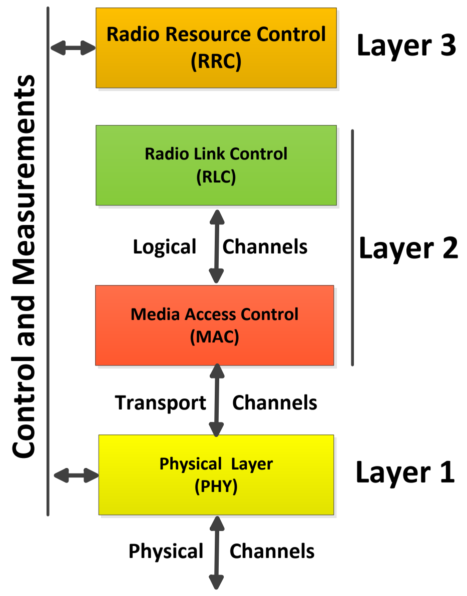


Figure 3.2 Dynamic nature of the LTE Radio.

3.1.1 Layers of LTE and their main functionalities

Fig. 3.3 (taken from [24]) illustrates how the decomposition was done. The authors in [1] describe LTE layers in more details.

Packet Data Convergence Protocol (PDCP) performs IP header compression to minimize the number of bits to send over the radio channel. This compression is based on Robust Header Compression (ROHC). PDCP is also responsible for ciphering and for the control plane, integrity protection of the transmitted data, as well as in-sequence delivery and duplicate removal for handover. At the receiver side, PDCP performs deciphering and decompression operations.

Radio Link Control (RLC) performs segmentation/concatenation, retransmission handling, duplicate detection, and in-sequence delivery to higher layers. The RLC provides services to the PDCP in the form of radio bearers.

Media Access Control (MAC) is responsible for multiplexing of logical channels, hybrid-ARQ retransmission, and uplink and downlink scheduling. The scheduling functionality is located in the eNodeB for both uplink and downlink. The hybrid-ARQ protocol is applied to both transmitting and receiving ends of the MAC protocol. The MAC provides services to the RLC in the form of logical channels.

Physical Layer (PHY) is responsible for coding/decoding, modulation/demodulation,

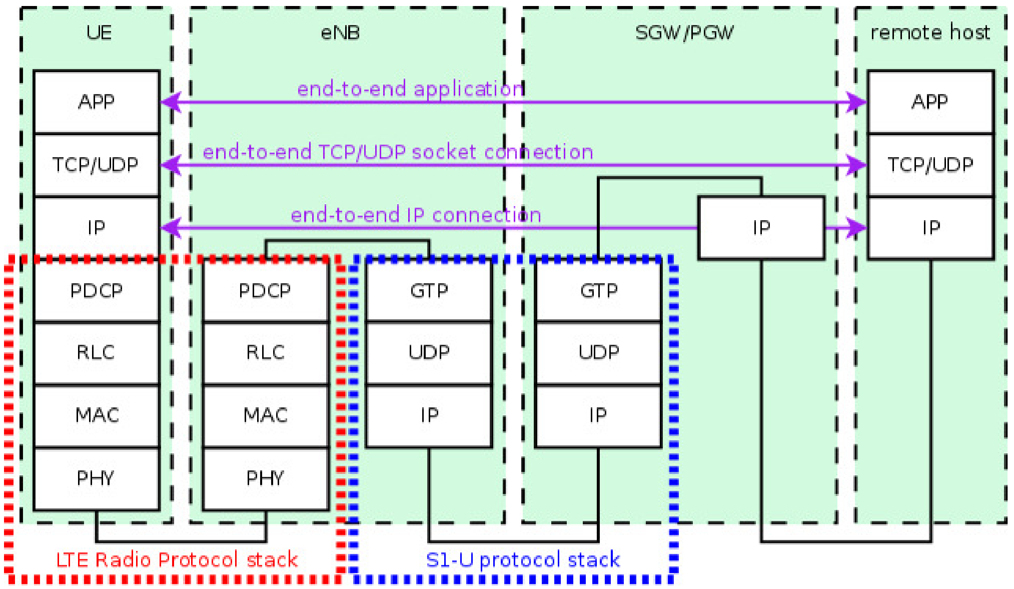


Figure 3.3 LTE-EPC data plane protocol stack.

multi-antenna mapping, and other typical physical-layer functions. The physical layer offers services to the MAC layer in the form of transport channels. Fig. 3.4 (taken from [1]) and Fig. 3.5 illustrate these functionalities graphically. In this figure, in an antenna and resource mapping block related to the physical layer, the antenna mapping module maps the output of the DFT precoder to antenna ports for subsequent mapping to the physical resource (the OFDM time-frequency module). Each resource block pair includes 14 OFDM symbols (one subframe) in time which follows in OFDM in LTE section.

3.1.2 OFDM in LTE

OFDM transmission is a kind of multi-carrier modulation. The basic characteristics of OFDM are : 1) the use of a very large number of narrowband subcarriers, 2) simple rectangular pulse shaping, and 3) tight frequency domain packing of the subcarriers with a subcarrier spacing $\triangle f = 1/T_u$. Where T_u is the per-subcarrier modulation symbol time. The subcarrier spacing is thus equal to the per-subcarrier modulation rate $1/T_u$. Fig. 3.6 (taken from [1]) illustrates a basic OFDM modulator. It consists of a bank of N_c complex modulators which are transmitted in parallel, and each modulator corresponds to one OFDM subcarrier.

Figure 3.7 (taken from [1]) illustrates the physical layer frame structure in a frequency-time grid. Time domain is divided into slots with duration of 0.5ms. Each sub-frame includes

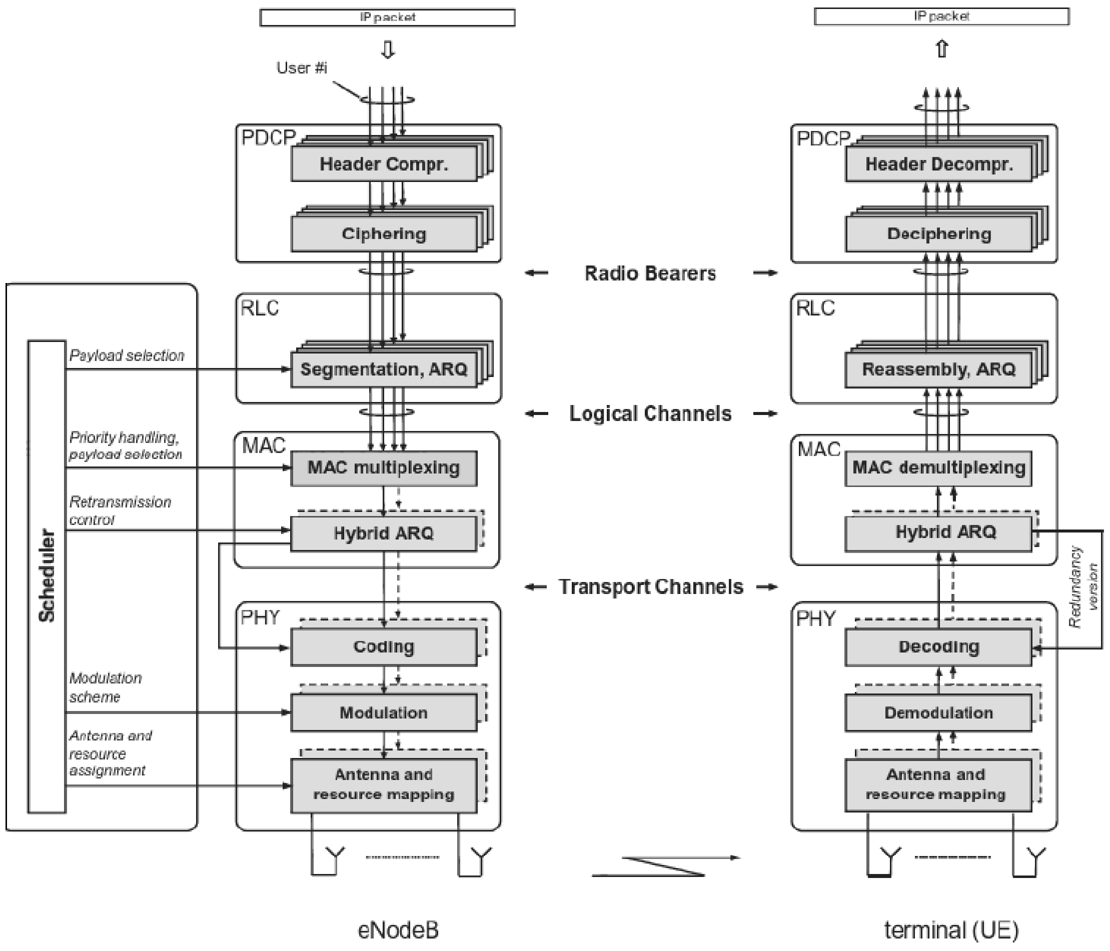


Figure 3.4 LTE protocol architecture (downlink) [1].

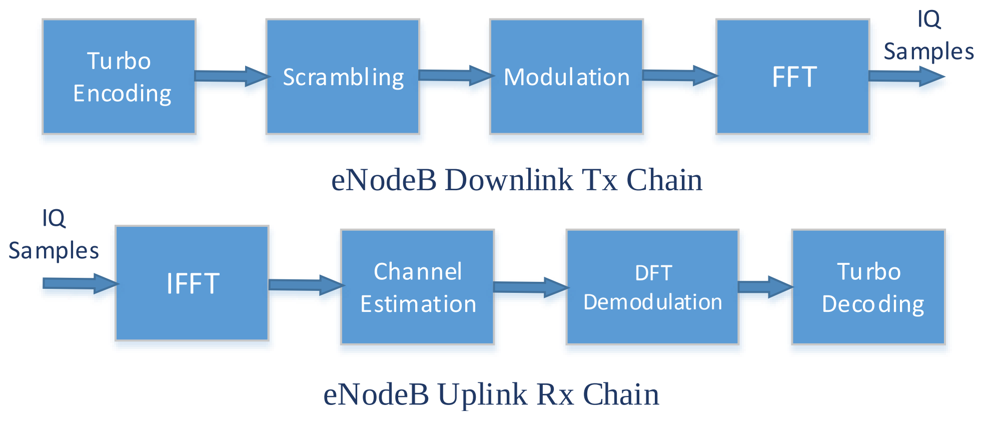


Figure 3.5 LTE physical layer blocks.

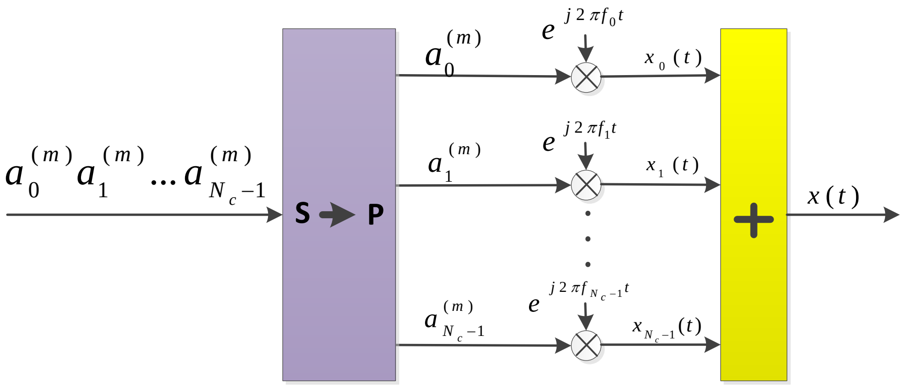


Figure 3.6 LTE OFDM modulation.

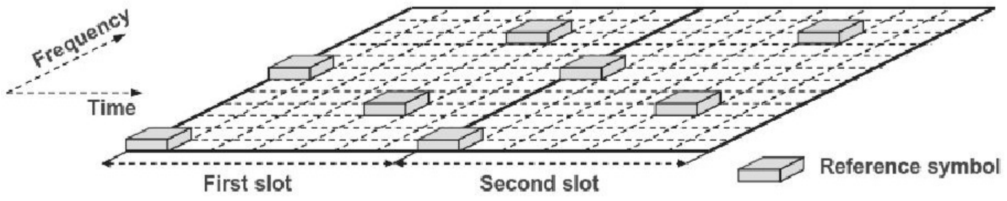


Figure 3.7 Structure of cell-specific reference signal within a pair of resource blocks.

2 time slots. In fact, there are 2×7 symbols in each sub-frame and there are 10 sub-frames in each frame. Frequency domain consists of sub-carriers. Each sub-carrier spans 15 KHz. There are 12 sub-carriers in each sub-band. Thus, 12 sub-carriers are transmitted in each time slot.

An OFDM signal x(t) during the time interval $mT_u \leq t < (m+1)T_u$ can be expressed as :

$$x(t) = \sum_{k=0}^{N_c-1} x_k(t) = \sum_{k=0}^{N_c-1} a_k^{(m)} e^{j2\pi k \triangle f t}, \tag{3.1}$$

where $x_k(t)$ is the k^{th} modulated subcarrier with frequency $f_k = k \triangle f$ and $a_k^{(m)}$ is the complex modulation symbol applied to the k^{th} subcarrier during the m^{th} OFDM symbol interval [1].

According to [1] the OFDM symbol consists of two major components : the CP and an FFT. As table 3.1 shows, LTE bandwidth varies from 1.25 MHz up to 20 MHz. In the case of 1.25 MHz transmission bandwidth, the FFT size is 128 and it is 2048 for 20MHz bandwidth.

Table 3.1 Transmission bandwidth vs. FFT size

Transmission bandwidth	1.25 MHz	2.5 MHz	5 MHz	10 MHz	15 MHz	20 MHz
FFT size	128	256	512	1024	1536	2048

3.1.3 OFDM implementation using FFT and IFFT

Fig. 3.6 illustrates, the basic principles of OFDM modulation. Choosing subcarrier spacing $\triangle f$ equal to the per-subcarrier symbol rate $1/T_u$, allows to implement an efficient FFT processing. We assume sampling rate f_s multiple of the subcarrier spacing $\triangle f$, $f_s = 1/T_s = N\triangle f$, the parameter N should be chosen to fulfill the sampling theorem [1]. The discrete-time OFDM signal can be expressed as :

$$x_n = x(nT_s) = \sum_{k=0}^{N-1} a_k e^{j2\pi nk/N} \tag{3.2}$$

where

$$a_k = \begin{cases} a_k^{'} & 0 \leq k < N_c \\ 0 & N_c \leq k < N \end{cases}, \tag{3.3}$$

Thus, the sequence x_n is Inverse Discrete Fourier Transform (IDFT) of modulation symbols $a_0, a_1, ..., a_{N_c-1}$ extended with zeros to length N to have a fixed length. As a result, OFDM modulation can be implemented by an IDFT of size N followed by digital to analog conversion which is shown in Fig. 3.8 [1].

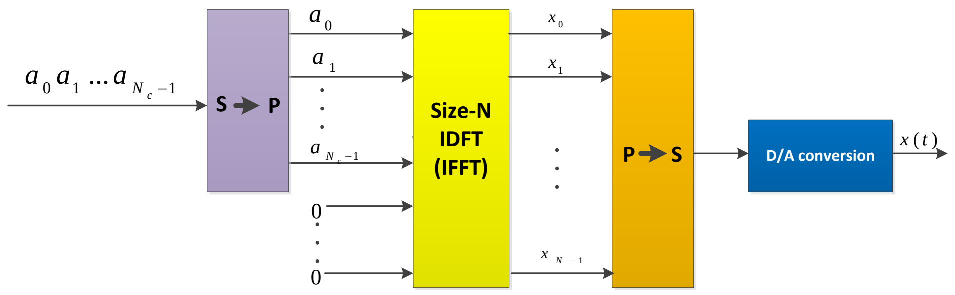


Figure 3.8 OFDM modulation by IFFT processing.

3.1.4 Turbo decoder

Turbo decoding as a forward error correction iterative algorithm achieves error performance near to the channel capacity. A Turbo decoder consists of two component decoders and two interleavers, which is shown in Fig. 3.9 (taken from [8]). It includes multiple passes through the two component decoders. One iteration includes one pass through both decoders. Despite the fact that both decoders perform the same sequence of computations, the decoders produce different log-likelihood ratios (LLRs). The de-interleaved LLRs of second decoder is the inputs of the first decoder and the interleaved LLRs of first decoder and channel are inputs of the second decoder. Each decoder operates a forward trellis traversal to decode a codeword with N information bits. Forward trellis traversal is used to compute N sets of forward state metrics, one α set per trellis stage. It is pursued by a backward trellis traversal which computes N sets of backward state metrics and one β set per trellis stage. Finally, the forward and the backward metrics are combined to compute the output LLRs [8]. Thus, turbo decoders, because of iterative decoding process and bidirectional recursive computing, requires optimal parallelism implementation to achieve high-data rates in telecommunication applications.

3.1.5 MIMO detection

Multiple-input multiple-output (MIMO) detection is the most time consuming task of LTE [15]. Authors in this paper proposed a novel strategy to implement the minimum mean square error for MIMO detection using OFDM. The key is using a massively parallel implementation of the scalable matrix inversion on graphics processing units (GPUs). A MIMO-OFDM system with M transmit antennae and N receive antennae can be expressed as

$$y = Hx + w \tag{3.4}$$

where $y = [y_0,\ y_1,\ y_2,\ ...,\ y_{N-1}]^T$ is the $N \times 1$ received data vector, H is the $N \times M$ MIMO channel matrix, x is the $M \times 1$ transmitted data vector and w is an $M \times 1$ white Gaussian noise vector. MIMO detector estimate the transmitted data vector $\hat{x}$ from the received noisy data y.

$$\hat{x} = G_{MMSE}\ \ y \tag{3.5}$$

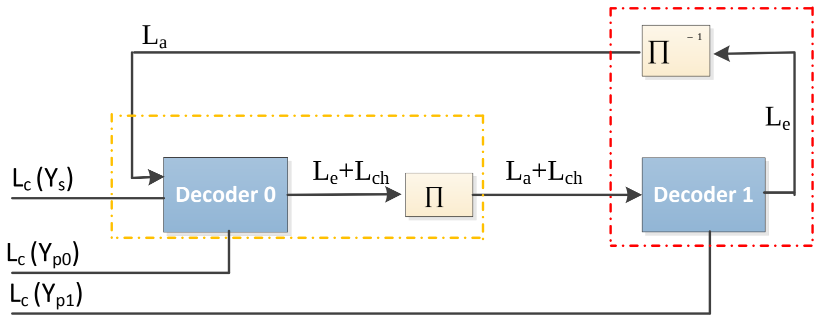


Figure 3.9 Overview of Turbo decoding.

Where MMSE minimize the mean square error of $E\{(\hat{x}-x)^H(\hat{x}-x)\}$ and $E\{.\}$ is the expectation of random variable. G_{MMSE} is

$$G_{MMSE} = (H^H H + I_M/\rho)^{-1} H^H = JH^H \tag{3.6}$$

where ρ is the signal to noise ratio [15]. As a result, matrix inversion is the bottleneck of this algorithm.

3.2 A glance at literature review

In this section, we describe more about the contents of literature review. The goal is to demonstrate graphically what the authors did in the literature review and to provide some more descriptions.

3.2.1 MIMO detection

In the literature review chapter of 2, the authors report on how to implement MIMO on GPUs to decrease computation time by developing a novel channel matrix preprocessing stage that enables parallel processing. As Fig. 3.10 (taken from [11]) illustrates, MIMO with bit-interleaved coded modulation (BICM) is applied to implement a fully parallel soft-output fixed-complexity sphere decoder (FSD).

Fig. 3.10 and Fig. 3.11 (taken from [11]) depict how to compute parallel tree search and soft information about the code bits in terms of log-likelihood ratios (LLRs). They

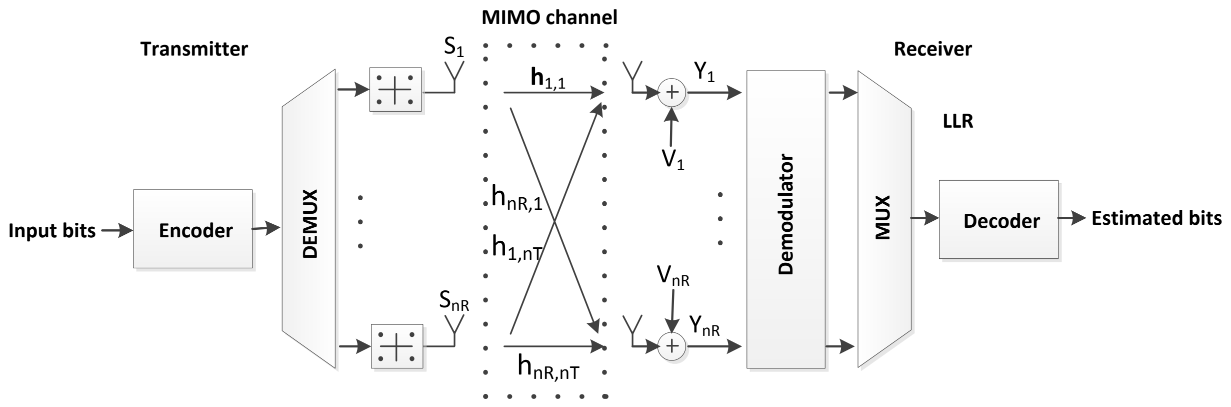


Figure 3.10 Block diagram of a MIMO-BICM system.

also illustrate how a fully parallel fixed-complexity sphere decoder (FPFSD) method can be implemented. The norms of the columns of the channel matrix are obtained (requiring n_T products, $n_T - 1$ sums, and one squared root operation each) and sorted in ascending order (n_T^2 floating point operations in the worst case). Thus, the complexity of this proposed ordering is $O(n_T^2)$. This can be computed considerably faster if the norms are processed in parallel. Generally, this ordering leads to more reliable decisions than random ordering, since symbols with the highest signal-to-noise ratio are detected before those with the lowest, thus reducing error propagation.

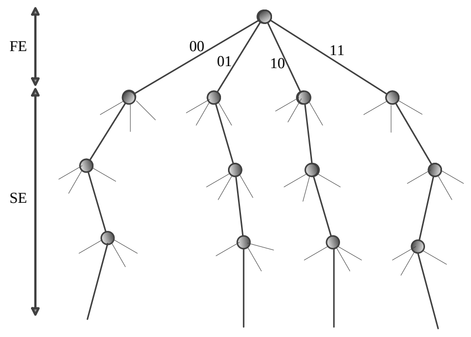


Figure 3.11 Decoding tree of the FSD algorithm for a 4×4 MIMO system with QPSK symbols.

3.2.2 Turbo decoder

We discussed in literature review that authors in [25] applied turbo decoder on GPU. Their challenges were how to parallelize workload across cores. Fig. 3.12 (taken from [25]) shows how threads are partitioned to handle the workload for N codewords.

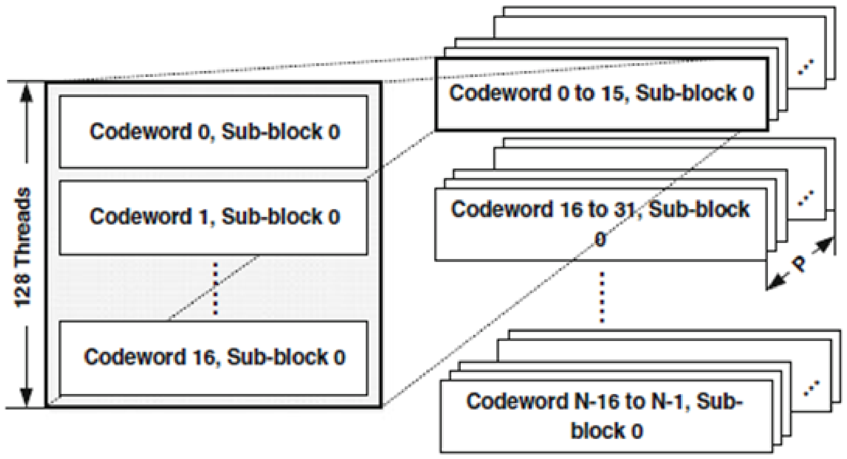

128 Threads
Codeword 0, Sub-block 0
Codeword 1, Sub-block 0
Codeword 16, Sub-block 0
Codeword 0 to 15, Sub-block 0
Codeword 16 to 31, Sub-block 0
P
Codeword N-16 to N-1, Sub-block 0

Figure 3.12 Handle the workload for N codewords by partitioning of threads.

The authors implemented a parallel Turbo decoder on GPU. As Fig. 3.12 depicts, instead of creating one thread-block per codeword to perform decoding, a codeword is split into P sub-blocks and decoded in parallel using multiple thread blocks. In this section, each thread-block has 128 threads and handles 16 codeword sub-blocks [25].

3.3 GPU

Graphics Processing Units (GPUs) are a computing platform that has recently evolved toward more general purpose computations. This section describes the GPU programming model as viewed by the Nvidia company, which is based on the CUDA (Compute Unified Device Architecture) programming language and SIMD (Single Instruction, Multiple Data) architecture. the Geforce GTX 660 Ti will be used as an example as it was used in experiments.

3.3.1 GPU programming model

GPUs include a massive parallel architecture. They work best when supporting a stream programming model. Stream processing is the programming model used by standard graphics APIs. Stream processing is basically on-the-fly processing, i.e. data is processed as soon as it arrives. The results are sent as soon as they are ready. Thus, data and results are not stored in global (slow) memory to save memory bandwidth. We keep temporary data and results in local memory and registers. A stream is a set of data that require similar computations. Those similar computations execute as kernels in the programming model for GPUs. Since, GPUs can only process independent data elements, kernels are performed completely independently on the data elements of input streams to produce an output stream. Because GPUs are stream processors, processors can operate in parallel by running one kernel on many records in a stream at once. In CUDA, threads are assembled in blocks. Multiple thread-blocks are called a grid. A grid is organized as a 2D array of blocks, while each block is organized as 3D array of threads. At runtime, each grid is distributed over multiprocessors and executed independently [26].

Threads within a thread-block execute in blocks of 32 threads. When 32 threads share the same set of operations, they are assembled in what is called a warp and are processed in parallel in a SIMD fashion. If threads do not share the same instruction, the threads are executed serially [26]. Multiprocessor has control unit and it starts and stops threads on compute engines. Control unit can select which threads run as a warp. Control unit can schedule blocks on the compute engines. The number of threads is higher than the number of compute engines to allow multitasking to improve performance. When we do not have enough threads, we do not have a good occupancy. The occupancy is the time which takes to pass data through the slowest component in the communication path. If we choose proper number of threads per block, we balance processing time with the memory bandwidth.

Fig. 3.13 (taken from [27]) illustrates CUDA memory model. As it shows, it consists of registers, local memory, shared memory, global memory, constant memory, and texture memory with the following descriptions.

Scalar variables that are declared in the scope of a kernel function and are not decorated with any attribute are stored in register memory by default. Access of register memory is very fast, but the number of registers that are available per block is limited. Any variable that can't fit into the register space allowed for the kernel will spill-over into local memory. Shared memory increases computational throughput by keeping data on-chip. It must be declared

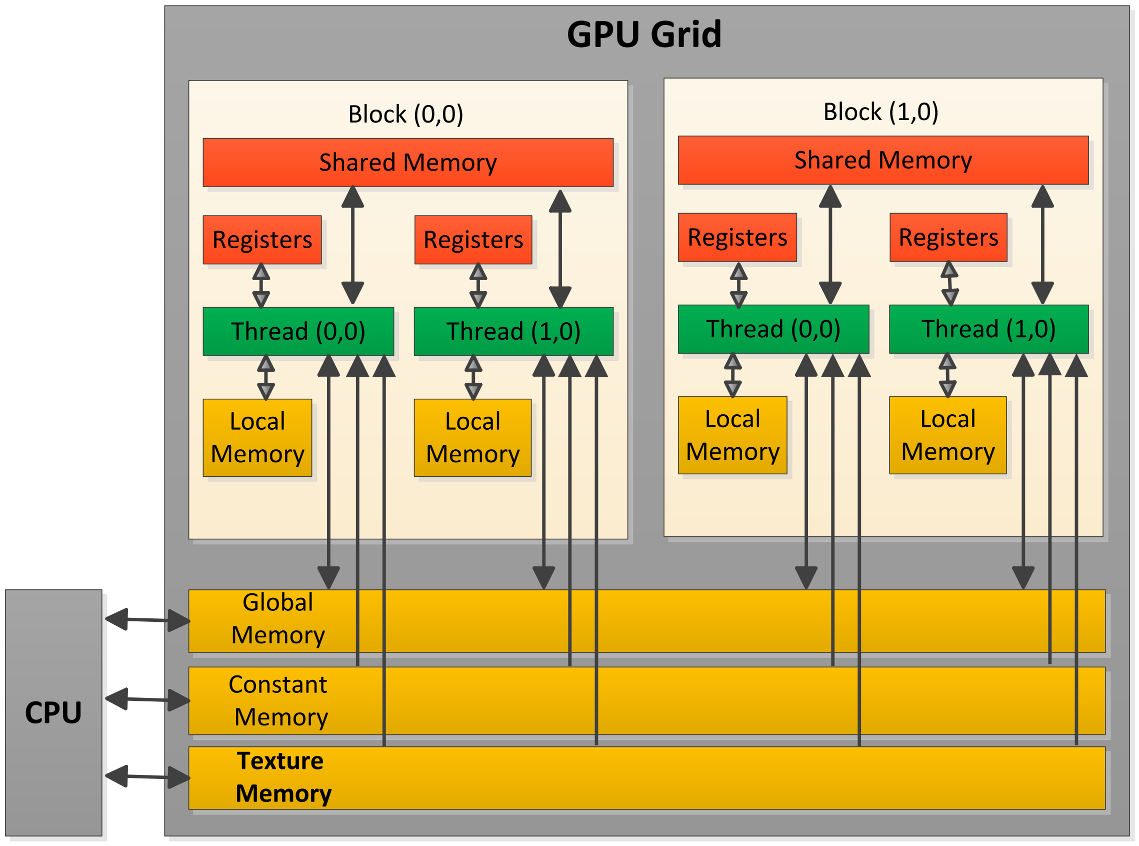


Figure 3.13 CUDA Memory Model.

within the scope of the kernel function. When execution of kernel is finished, the shared memory in the kernel cannot be accessed. Global memory is declared outside of the scope of the kernel function. The access latency to global memory is very high (100 times slower than shared memory) but there is much more global memory than shared memory. Constant memory is used for data that will not change over the course of a kernel execution and it is cached on chip. Texture memory is read only has an L1 cache optimized for 2D spatial access pattern. In some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM. It is designed for graphics applications where memory access patterns exhibit a great deal of spatial locality [26].

3.3.2 Geforce GTX 660 Ti specifications

Geforce GTX 660 Ti GPU includes 7 multiprocessors, 192 CUDA cores per multiprocessor (compute capability), SIMDWidth (threads per warp) equals to 32, 2G bytes global memory, 1024 Max threads per block, $(1024 \times 1024 \times 64)$ Max thread dimensions, $(2G \times 65536 \times 65536)$ Max Grid dimensions and Clock rate is about 1GHz. The number of shared memory per multiprocessor is 49152 while the number of registers per multiprocessor is 65536 [27].

3.4 Intel Math Kernel Library (MKL)

Intel Math Kernel Library (MKL) is a highly optimized Math library. It uses for applications that require maximum performance. Intel MKL can be called from applications written in either C/C++, or in any other language that can reference a C interface. It includes 1) BLAS and LAPACK linear algebra libraries for vector, vector-matrix, and matrix-matrix operations, 2) ScaLAPACK distributed processing linear algebra libraries for Linux and Windows operating systems, as well as the Basic Linear Algebra Communications Subprograms (BLACS) and the Parallel Basic Linear Algebra Subprograms (PBLAS), 3) the PARDISO direct sparse solver, 4) FFT functions in 1D, 2D or 3D, 5) Vector Math Library (VML) routines for optimized mathematical operations on vectors, 6) Vector Statistical Library (VSL) routines, which offer high-performance vectorized random number generators (RNG) for several probability distributions, convolution and correlation routines, and summary statistics functions, 7) Data Fitting Library, which provides capabilities for spline-based approximation of functions, derivatives and integrals of functions, and search, and 8) Extended Eigen solver and a shared memory programming (SMP) version of an eigen solver. For details see the Intel MKL Reference Manual. In this thesis, MKL is used to compute FFT. The results are shown in Chapter 6. Algorithm 3.1 describes the implementation of FFT by MKL.

In this algorithm, lines 6 and 7 allocates the descriptor data structure and initializes it with default configuration values. Line 8 performs all initialization for the actual FFT computation. The DftiComputeForward function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, this function computes the forward FFT. Line 10 frees the memory allocated for a descriptor [28]. Note that we must add 3 libraries (mkl_intel_ilp64, mkl_core and mkl_sequential) for compilation.

Algorithm 3.1 Float Complex FFT using MKL[28]

```
1: #include "mkl_dfti.h"
2: float _Complex x[N];
3: DFTI_DESCRIPTOR_HANDLE my_desc1_handle;
4: MKL_LONG status;
5: //...put input data into x[0], ..., x[N − 1];
6: status = DftiCreateDescriptor
7:     (&my_desc1_handle, DFTI_SINGLE, DFTI_COMPLEX, 1, N);
8: status = DftiCommitDescriptor(my_desc1_handle);
9: status = DftiComputeForward(my_desc1_handle, x);
10: status = DftiFreeDescriptor(&my_desc1_handle);
11: /* result is x[0], ..., x[N] */
```

3.5 DPDK

DPDK is an optimized data plane software solution developed by Intel for its multi-core processors. It includes high performance packet processing software that combines application processing, control processing, data plane processing, and signal processing tasks onto a single platform. DPDK has a low level layer to improve performance. It has memory management functions, network interface support and libraries for packet classifications.

3.5.1 DPDK features

DPDK is a core application that includes optimized software libraries and Network Interface Card (NIC) drivers to improve packet processing performance by up to ten times on x86 platforms [3]. On the Hardware side, DPDK has the capabilities to support high speed pipelining, low latency transmission, exceptional QoS, determinism, Real time I/O switching. Intel Xeon series with an integrated DDR3 memory controller and an integrated PCI Express controller lead to lower memory latency.

DPDK features are : 1) It does some of the management tasks that are normally done by the operating system (it does those tasks with low overhead), 2) DPDK uses a run-to-completion model and a parallel computation model which runs one lcore followed by another lcore for next processing step, 3) DPDK uses Poll mode (i.e. it does not support interrupts) which is simpler than interrupts, thus it has lower overhead, 4) DPDK allocates memory from kernel at startup. 5) DPDK is pthread based but abstracts the pthread create, join, and provides a wrapper for the worker threads, and 6) DPDK supports Linux Multi-process.

As Fig. 3.14 (taken from [29]) illustrates, Dual Channel DDR memory uses two funnels (and thus two pipes) to feed data to the processor. Thus, it delivers twice the data of the single funnel. To prevent the funnel from being over-filled with data or to reverse the flow of data through the funnel, there is a traffic controller or memory controller that handles all data transfers involving the memory modules and the processor [29].

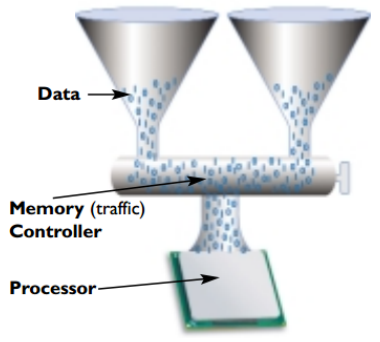


Figure 3.14 Dual Channel DDR Memory.

Thread is a procedure that runs independently from its main program. Pthread comes from IEEE POSIX 1003.1c standard. In fact it is Posix thread. It is one kind of thread. It is a software which a core executes. Pthread is light weight thread. Managing threads requires fewer system resources than managing processes. The most important functions of Pthread library are pthread_create and pthread_join. Pthread_create is a function to create a new thread. Since, we need to manually terminate all threads before the main thread ends, pthread_join does this task. When a parent thread (main thread) creates a child thread, it meets pthread_join waits until the child thread's execution finishes, and safely terminates the child thread.

Moreover, standard Linux operating system can also help to reduce overhead. In particular, using core affinity, disabling interrupts generated by packet I/O, using cache alignment, implementing huge pages to reduce translation look aside buffer (TLB) misses, prefetching, and new instructions save time.

On the computing side, DPDK includes dynamic resource sharing, workload migration, security, OpenAPIs, a developer community, virtualization and power management. Moreover, DPDK uses threads to perform zero-copy packet processing in parallel in order to reach high efficiency. Additionally, in its Buffer Manager, each core is provided a dedicated buffer cache to the memory pools which provides a fast and efficient method for quick access and release of buffers without lock. Fig. 3.15 illustrates Intel's DPDK architecture. A

Buffer/Memory pool manager in DPDK provides NUMA (Non Uniform Memory Access) pools of objects in memory. Each pool utilizes the huge page table support of modern processors to decrease Translation Lookaside Buffer (TLB) misses and uses a ring (a circular buffer) to store free objects. The memory manager also guarantees that accesses to the objects are distributed across all memory channels. So, DPDK memory management includes NUMA awareness, alignment, and huge page table support which means that the CPU allocates RAM by large chunks. The chunks are pages. Less pages you have, less time it takes to find where the memory is mapped.

DPDK allows user applications to run without interrupts that would prevent deterministic processing. The queue manager uses lockless queues in order to allow different modules to process packets with no waiting times. Flow classification leverages the Intel Streaming SIMD Extensions (SSE) in order to improve efficiency. Also, Intel DPDK includes NIC Poll Mode drivers and libraries for 1 GbE and 10 GbE Ethernet controllers to work with no interrupts, which provides guaranteed performance in pipeline packet processing. DPDK's Environment Abstraction Layer (EAL) contains the run-time libraries that support DPDK threads [30].

3.5.2 How to use DPDK

To install DPDK, it is important to have kernel version >= 2.6.33. The kernel version can be checked using the command of "$uname - r$". There is an installation guide for DPDK in [31]. To start using DPDK, in example directory of DPDK, there are sample codes such as helloworld and codes for packet forwarding L2 and L3. These codes are explained in [32]. Thus, it is important to know how to run helloworld file of DPDK. As it is in the DPDK documents in [31], to compile the application, you should go to your directory and configure two environmental variables of RTE_SDK and RTE_TARGET before compiling the application. The following commands show how those two variables can be set :

```
cd examples/helloworld
```

Set the path :

```
export RTE_SDK=$HOME/DPDK
```

Set the target :

```
export RTE_TARGET=x86_64-default-linuxapp-gcc
```

Build the application by :

```
make
```

To run the application in linux application environment, run the following command [32] :

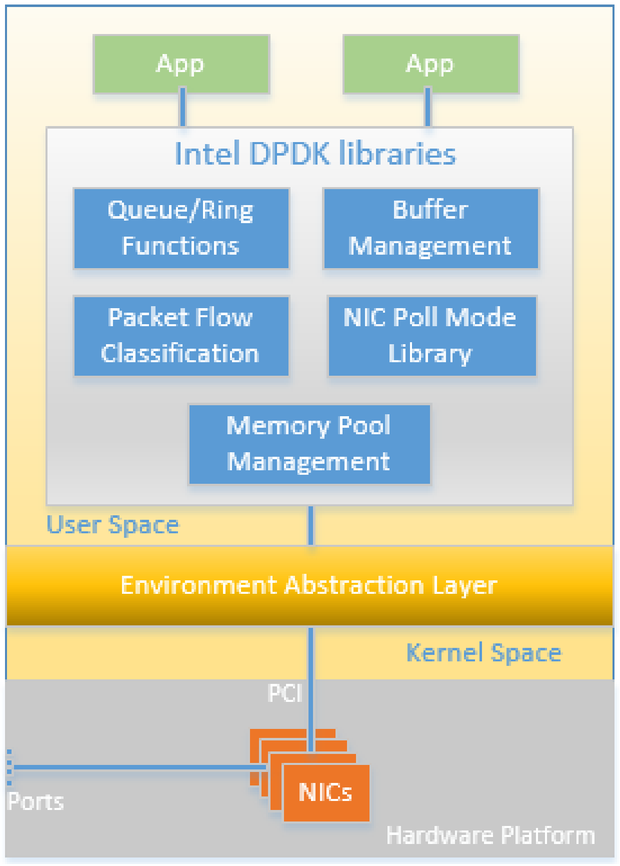


Figure 3.15 Intel Data plane development kit (DPDK) architecture.

./build/hellowworld -c f -n 4
I used the following command to run my application (I put MKL_FFT function, which I explained it MKL section, in helloworld file) :
./build/helloworld -c 5 -n 1 –no-huge
Where -c is COREMASK. It is an hexadecimal bit mask of the cores to run on. Core numbering can change between platforms and should be determined beforehand. You can monitor your PC cores by system monitor in linux.
-n NUM is number of memory channels per processor socket.
and –no-huge means to use no huge page.
Based on DPDK documents, there are the list of options that can be given to the EAL :
./rte-app -c COREMASK -n NUM [-b <domain :bus :devid.func>] [–socket-mem=MB,...] [-m MB] [-r NUM] [-v] [–file-prefix] [–proc-type <primary|secondary|auto>] [–xen-dom0].
As Fig. 3.16 (taken from [33]) depicts, the first step to write a code using DPDK is to initialize the Environment Abstraction Layer (EAL). It creates worker threads and launch commands to main. This function is rte_eal_init() in the master lcore. Indeed, Master lcore runs the main function. This function is illustrated in algorithm 3.2. This algorithm finishes

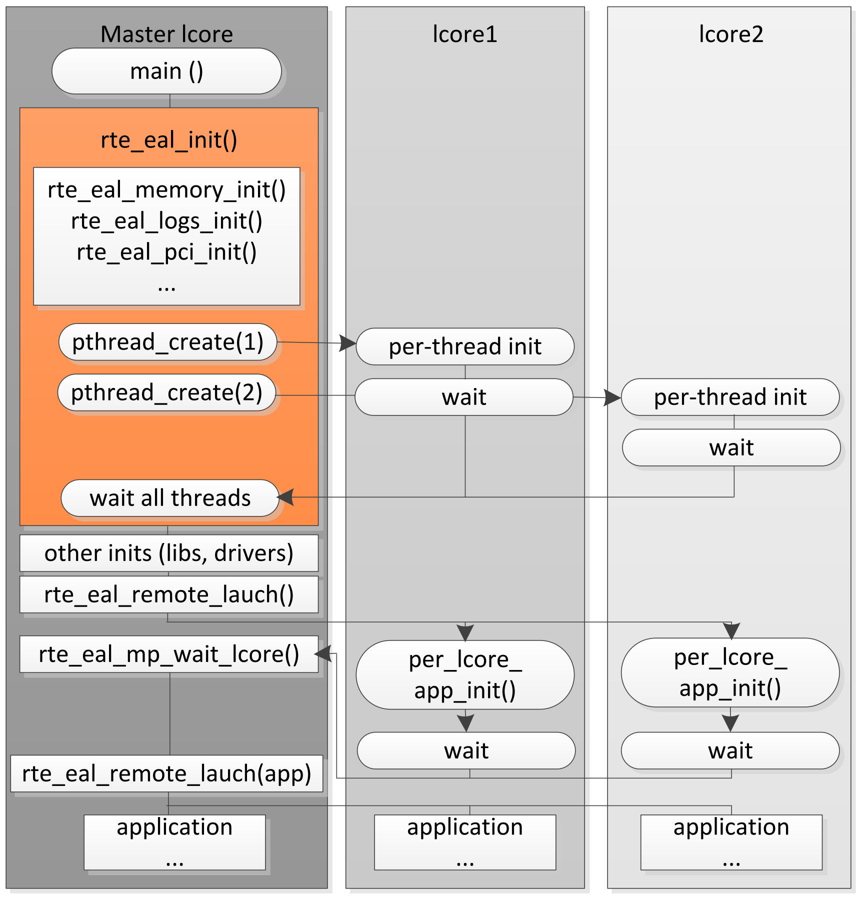


Figure 3.16 EAL initialization in a Linux application environment.

the initialization steps. Then, it is time to launch a function on an lcore.
An lcore is an abstract view of a core. It corresponds to either the full hardware core when hyper threading is not implemented, or it is hardware thread of a core that has hyper threading. In algorithm 3.3 and 3.4 lcore_hello() is called on every available lcore and the code that launches the function on each lcore is demonstrated, respectively.

Algorithm 3.2 EAL Initialization [32]

```
1: $int\ \ main(intargc,\ char**\ \ argv)$
2: {
3: $ret = rte\_eal\_init(argc,\ \ argv);$
4: **if** $ret\ < \ 0$ **then**
5:     $rte\_panic("Can\ not\ init\ EAL");$
6: **end if**
```

Algorithm 3.3 Definition of function to call lcore

```
$static\ int\ lcore\_hello(\_attribute\_((unused))\ void\ *arg)$
{
$unsigned\ lcore\_id;$
$lcore\_id = rte\_lcore\_id();$
$printf("hello\ from\ core\ \%u",\ lcore\_id);$
$return\ 0;$
}
```

Algorithm 3.4 Launch the function on each lcore

```
$/*\ call\ lcore\_hello()\ on\ every\ slave\ lcore */$
$RTE\_LCORE\_FOREACH\_SLAVE(lcore\_id)$
{
$rte\_eal\_remote\_launch(lcore\_hello,\ NULL, lcore\_id);$
}
$/*\ call\ it\ on\ master\ lcore\ too */$
$lcore\_hello(NULL);$
```

Those algorithms exist in Helloworld example. To explain it clearly, I divided that code to several parts. Then I put my FFT_MKL function (in MKL section) in that code. Replace your function with lcore_hello function. The function of lcore_hello just write hello at the output. A function of rte_eal_remote_launch() sends a message to each thread telling what function to run. Rte_eal_mp_wait_lcore() waits for thread functions to complete and rte_lcore_id() returns core id. Pthread_create is a function to create a new thread. Wait all treads means to wait all threads finish their tasks. More explanations are in [32]. In chapter 6, we explain the usage of MKL and further discuss its advantages. To do isolcpus, modify grub file such as below (it depends on the operating system).

GRUB_CMDLINE_LINUX="isolcpus = 4,5,6,7" in /etc/default/grub file.

Perform grub-mkconfig -o /boot/grub/grub.cfg.

Reboot your system and see system monitor to see the core isolation.

CHAPTER 4

FFT, MATRIX INVERSION AND CONVOLUTION ALGORITHMS

Since Fast Fourier Transform (FFT), matrix inversion and convolution are our benchmark in this thesis, the aim of this chapter is to describe in more details their algorithms in order to know if they have a potential for parallel implementation. We explain FFT and Discrete Fourier Transform (DFT) Radix-4. Then we perform radix-4 FFT using Matlab. Also, we present FFT Cooley-Tukey and Stockham algorithms to be familiar with different FFT algorithms. Finally, we show Gaussian Elimination algorithm for matrix inversion as well as introducing convolution and cross-correlation. We describe computational complexity order of all these functions. In the next chapter we will see how to accelerate the computation of these algorithms.

4.1 Fast Fourier Transform

This section describes DFT, that is, a Fourier transform as applied to a discrete complex valued series. For a continuous function of one variable $x(t)$, the Fourier Transform X(f) is defined as :

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \tag{4.1}$$

and the inverse transform as

$$x(t) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}df \tag{4.2}$$

where j is the square root of -1. Consider a complex series $x(k)$ with N samples $x_0, x_1, x_2, \cdots, x_{N-1}$. Where x is a complex number. Further, assume that the series outside the range 0, $N-1$ is extended N-periodic. So that $x_k = x_{k+N}$ for all k. The following equation represents the DFT in matrix form with input vector x with dimension of N and output vector $X[k]$ (see[34]).

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi nk/N} \tag{4.3}$$

where $n \in [0; N-1]$ and $k \in [0; N-1]$. The sequence $x[n]$ is referred to as the time domain and $X[k]$ as the frequency domain. The DFT can be written in the matrix form as $X = F_N x$ where F_N is an $N \times N$ matrix given by

$$[F_N]_{rs} = w^{rs} \tag{4.4}$$

where

$$w = e^{-j2\pi/N}. \tag{4.5}$$

The inverse equation is given by :

$$x = \frac{1}{N} F_N^* X. \tag{4.6}$$

Where $*$ means complex conjugate. Figure 4.1 (taken from [34]) illustrates FFT computation for $N = 8$ graphically. The output of this figure is the vector X in reverse bit order. As this figure shows, there is a potential to perform parallelism in this algorithm. In the next chapter, we show how to leverage parallelism to reduce computational time of FFT. For a value $N = 2^n$, the FFT factorization includes $log_2 N = n$ iterations, each containing $N/2$ operations for a total of $(N/2)log_2 N$ operations. This factorization is a base-2 factorization applicable if $N = 2^n$.

4.1.1 Discrete Fourier Transform Radix-4

In this section we show that how to compute Discrete Fourier Transform Radix-4 and how the computational load of radix-4 is lower. The s^{th} sample of time series calculated by sampling of $f(t)$ in a duration T. DFT of N samples is derived by [35]

$$F_r = \frac{1}{N} \sum_{s=0}^{N-1} e^{j2\pi rs/N} f_s \tag{4.7}$$

where F_r is the r^{th} Fourier coefficient and $j = \sqrt{-1}$. We define T_N as :

$$\begin{aligned} (T_N)_{rs} = &\ e^{j2\pi rs/N} && (4.8)\\ = &\ w^{rs} && (4.9) \end{aligned}$$

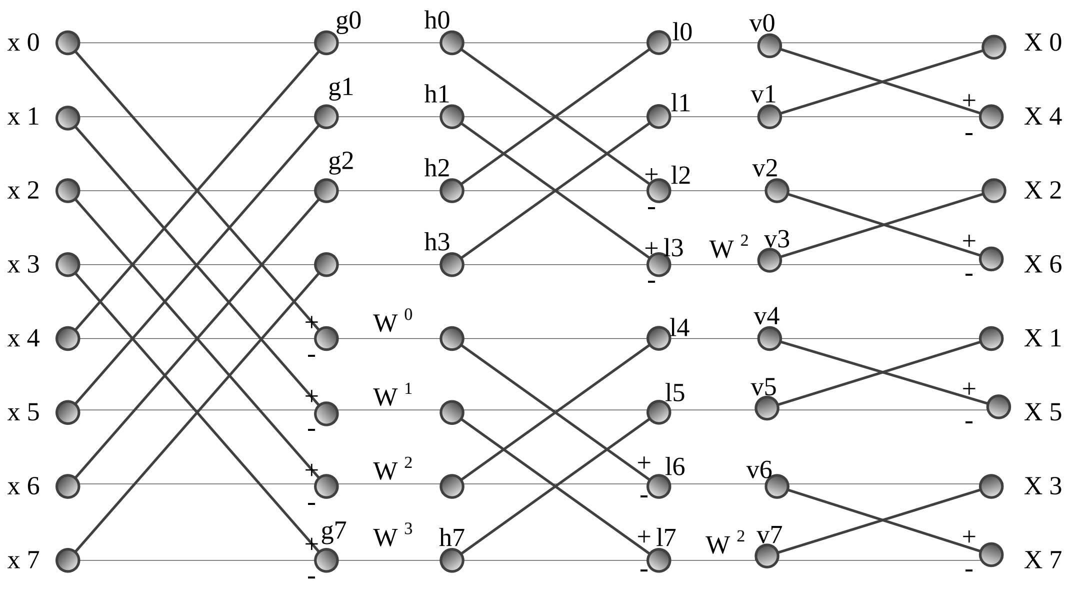


Figure 4.1 FFT factorization of DFT for N = 8.

where

$$w = e^{j2\pi/N} \tag{4.10}$$

Equation (4.7) can be written in the form of

$$F = (1/N)T_N f. \tag{4.11}$$

If $N = r^n$, n is integer and r is basis, we can write

$$T_N = P_N^{(r)} T_N^{'} \tag{4.12}$$

$$T_N^{'} = P_N^{'(r)} T_N \tag{4.13}$$

and

$$P_N^{'(r)} = \{P_N^{(r)}\}^{-1} \tag{4.14}$$

Where $P_N^{(r)}$ is a permutation matrix specific to the basis r. Thus Equation (4.12) expressed that we can show matrix T_N based on its transpose matrix $T_N^{'}$ using permutation matrix $P_N^{(r)}$. Matrix $T_N^{'}$ is partitioned into $r \times r$ square sub-matrices with dimension of $N/r \times N/r$. $T_{N/r}$ is expressed in terms of T_{N/r^2}. This process is iteratively applied. The symbol $\times$ is Kronecker product of matrices. The i^{th} iteration is represented as :

$$T_{N/k} = P_{N/k}^{r}(T_{N/rk} \times I_r) D_{N/k}^{(r)} (I_{N/rk} \times T_r) \tag{4.15}$$

where

$$D_{N/k}^{(r)} = quasi - diag(I_{N/rk}, K_k, K_{2k}, K_{3k}, ..., K_{(r-1)k}) \tag{4.16}$$

Where quasi-diag means to have values of K instead of zeros in Identity matrix of I.

$$K_m = diag 0, m, 2m, 3m, ..., (N/rk - 1)m \tag{4.17}$$

For any m as an integer we have

$$T_r = \begin{pmatrix} 0 & 0 & 0 & 0 & ... & 0 \\ 0 & N/r & 2N/r & 3N/r & ... & (r-1)N/r \\ 0 & 2N/r & 4N/r & 6N/r & ... & 2(r-1)N/r \\ . & . & . & . & ... & . \\ 0 & (r-1)N/r & . & . & ... & (r-1)^2 N/r \end{pmatrix} \tag{4.18}$$

For simplicity, we express k instead of w^k. I_k is the unit matrix of dimension k. By partitioning the matrix $T_{N/k}$ and using the following equation

$$(ABC...) \times I = (A \times I)(B \times I)(C \times I)... \tag{4.19}$$

where $A, B, C, ..., I$ are square matrices with the same size, we derive radix-r fast Fourier transform.

$$\begin{aligned} T_N &= P_N^r (P_{N/r}^r \times I_r)...(P_{N/k}^r \times I_k)...(P_{r^2}^r \times I_{N/r^2}) \\ &\quad . (T_r \times I_{N/r})(D_{r^2} \times (I_{N/r^2})(I_r \times T_r \times (I_{N/r^2})... \\ &\quad . (D_{N/k}^r \times I_k)(I_{N/rk} \times T_r \times I_k)... \\ &\quad . (D_{N/r}^r \times I_r)(I_{N/r^2} \times T_r \times I_r) D_N^r (I_{N/r} \times T_r). \end{aligned} \tag{4.20}$$

Considering

$$S^{(r)} = (I_{N/r} \times T_r) \tag{4.21}$$

and applying the property of the powers of shuffle operators,

$$\{P_N^{(r)}\}^{-i} \, S_N^{(r)} \, \{P_N^{(r)}\}^i = I_{N/r^{i+1}} \times T_r \times I_{r^i} \tag{4.22}$$

we obtain

$$T_N = \prod_{m=1}^{n} (P_m^{(r)} \mu_m^{(r)} S^{(r)}) \tag{4.23}$$

where

$$\mu_i^{(r)} = I_{r^{n-i}} \times D_{r^i}^{(r)} \tag{4.24}$$

and $P^{(r)}$ represents the permutation matrix $P_N^{(r)}$. For $N = 256$ and base 4 ($r = 4$)

$$T_{N=256} = p_1\mu_1 s p_2\mu_2 s p_3\mu_3 s p_4\mu_4 s \tag{4.25}$$

s is obtained from equation (4.21) where

$$T_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \tag{4.26}$$

$\mu_m^{(r)}$ is obtained from equation (4.24)

$$p_1 = \mu_1 = I_N \tag{4.27}$$

and

$$p_i^{(r)} = I_{r^{n-i}} \times P_{r^i}^{(r)} \tag{4.28}$$

$$\begin{aligned} P_K^{(r)} \quad &. \quad col(x_0, x_1, x_2, ..., x_{K-1}) \\ &= \quad col(x_0, x_p, x_{2p}, x_{3p}..., x_1, x_{p+1}, x_{2p+1}, x_{3p+1}, ..., x_2, x_{p+2}, x_{2p+2}, ..., x_{K-1}). \end{aligned} \tag{4.29}$$

where $K/r = p$. In Chapter 5 we implement this algorithm on CPU to see its speed of computation.

4.1.2 Cooley-Tukey and Stockham formulation of the FFT algorithm

Fig. 4.2 (taken from [36]) illustrates Cooley-Tukey and Stockham algorithms. These two algorithms merge pairs of smaller FFTs into larger ones. Each box depicts the FFT of the listed sequence elements. Based on these algorithms, Cooley-Tukey algorithm requires an initial bit-reversal step, while Stockham algorithm does not. The initial bit-reversal permutation in Cooley-Tukey algorithm can cause delay. Because the memory accesses are unstructured. While radix-2 Stockham FFT algorithm eliminates the bit-reversal necessity

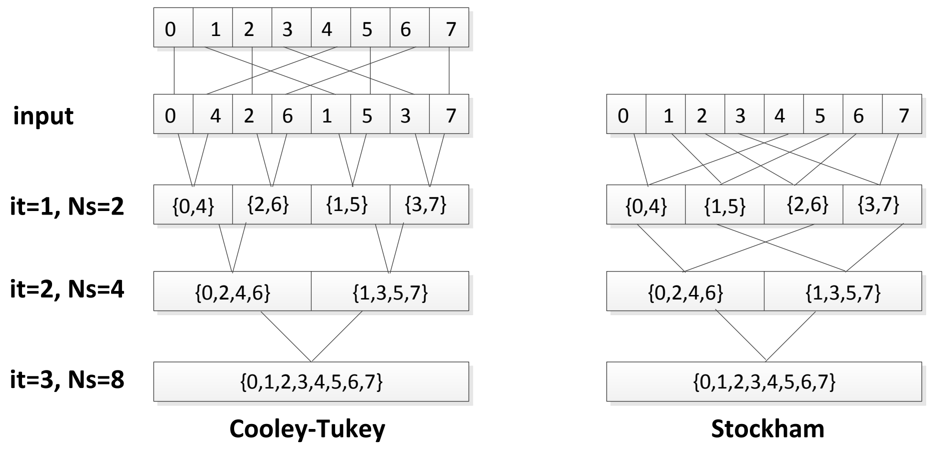


Figure 4.2 Dataflow for two DFT algorithms (Cooley-Tukey and Stockham).

by reordering the dataflow [37].

4.1.3 Summary on Fast Fourier Transform

As Fig. 4.1 illustrates and equation (4.23) represents, DFT needs parallelism to speed up computation. If the number of points N satisfies $N = r^n$ where r, called the radix or base, is an integer, then the FFT reduces the number of complex multiplications needed to evaluate the DFT from N^2 to $(N/r)log_rN$. For smaller N, the number of computation is $NlogN$. FFT Stockham algorithm compared to Cooley-Tukey algorithm requires twice more memory because it does not perform the FFT in-place and there is no simultaneous read and write access [37]. To reduce more the computation time of Radix-4, based on equation (4.23) which is multiplication of three matrices, we can do these matrix multiplication in parallel. In the next chapter we describe how to do matrix multiplication in parallel.

4.2 Matrix Inversion

As we saw in the literature review, channel matrix inversion is proposed for MIMO detection. For this purpose, we introduce Gaussian Elimination algorithm which is a numerical method. This method is implemented in parallel to accelerate computation of matrix inversion. In fact, it solves linear system $AX = B$ where A is an square $n \times n$ matrix, X and B are both $n \times 1$ vectors. Gaussian Elimination algorithm reduces $AX = B$

system to an upper triangular system $UX = Y$ which is solved through backward substitution. In the numerical program, vector B is stored as the $(n+1)^{th}$ column of matrix A. In this algorithm, we consider loop k controls the elimination step, loop i controls to access the i^{th} row and loop j controls access of j^{th} column. The following pseudo code (algorithms 4.1 and 4.2) describe Gaussian Elimination. In backward substitution, x_i is stored in the space of $a_{i,n+1}$. To perform Parallel Gaussian Elimination in forward elimination section, the following task can be parallelized for $k = 0$ to $k = n - 1$.

```
$a_{ik} = a_{ik}/a_{kk};$
$for\ j = k + 1\ \ to\ \ n - 1$
$a_{ij} = a_{ij} - a_{ik} \cdot a_{kj};$
$end\ for$
```

In backward substitution, the following part is performed in parallel :

```
$for\ j = i + 1\ \ to\ \ n - 1$
$x_i = x_i - a_{ij} \cdot a_{kj};$
$end\ for$
$x_i = x_i/a_{ii}$
```

Algorithm 4.1 Forward Elimination [38]

```
1: for $k = 0\ \ to\ \ n - 1$ do
2:    for $i = k + 1\ \ to\ \ n - 1$ do
3:       $a_{ik} = a_{ik}/a_{kk};$     /* divided by pivot element */
4:       for $j = k + 1\ \ to\ \ n - 1$ do /* for all rows below the pivot row */
5:          $a_{ij} = a_{ij} - a_{ik} \cdot a_{kj};$
6:       end for
7:    end for
8: end for
```

Algorithm 4.2 Backward substitution [38]

```
1: for $i = n\ \ to\ \ 1$ do
2:    for $j = i + 1\ \ to\ \ n - 1$ do
3:       $x_i = x_i - a_{ij} \cdot a_{kj};$
4:    end for
5:    $x_i = x_i/a_{ii}$
6: end for
```

4.2.1 Complexity of Gaussian Elimination algorithm

Complexity of Gaussian Elimination algorithm means that in the worst case how many steps it requires to compute this algorithm. Addition and multiplication are the main

functions of Gaussian Elimination. Based on the experimental result in the next chapter which is hardware acceleration using GPU, the computation time for multiplication is much longer than the computation time for addition. If we consider to compute the inversion of matrix $A_{m\times m}$, we have $m+1$ columns. There are $m+1$ multiplications for row i ($\mathcal{R}_i \to c\mathcal{R}_i$). In the next step, each $m+1$ elements in row j needs multiplication and then addition to the relevant element in row i. Therefore, in this step, there are $m+1$ multiplications and $m+1$ additions ($\mathcal{R}_i \to \mathcal{R}_i + c\mathcal{R}_i$). With a rough calculation we can say that Gaussian Elimination has order n^3 ($\mathcal{O}(n^3)$). These steps are illustrated by the following example.
$.143x_1 + .357x_2 + 2.01x_3 = -5.17$
$-1.31x_1 + .911x_2 + 1.99x_3 = -5.46$
$11.2x_1 - 4.30x_2 - .605x_3 = 4.42$

$$\begin{pmatrix} .143 & .357 & 2.01 & -5.17 \\ -1.31 & .911 & 1.99 & -5.46 \\ 11.2 & -4.30 & -.605 & 4.42 \end{pmatrix}$$

$$\begin{pmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ -1.31 & .911 & 1.99 & -5.46 \\ 11.2 & -4.30 & -.605 & 4.42 \end{pmatrix} \Longleftarrow$$
Dividing the first row in parallel by 0.143 ($A_{0,0}$) to produce a new first row (divide by Pivot element) $A_{k,j} = A_{k,j}/A_{k,k}$;

$$\begin{pmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 4.19 & 20.5 & -52.9 \\ 11.2 & -4.30 & -.605 & 4.42 \end{pmatrix} \Longleftarrow$$
Adding 1.31 $A_{1,0}$ times the first row to the second row in parallel to produce a new second row $A_{i,j} = A_{i,j} - A_{i,k} \times A_{k,j}$;

$$\begin{pmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 4.19 & 20.5 & -52.9 \\ 0.00 & -32.3 & -159 & 409 \end{pmatrix} \Longleftarrow$$
Adding -11.2 $A_{2,0}$ times the first row to the third row in parallel to produce a new third row;

$$\begin{pmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 1.00 & 4.89 & -12.6 \\ 0.00 & -32.3 & -159 & 409 \end{pmatrix} \Longleftarrow$$
Dividing the second row by 4.19 $A_{2,1}$ in parallel to produce a new second row;

$$\begin{pmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 1.00 & 4.89 & -12.6 \\ 0.00 & 0.00 & -1.00 & 2.00 \end{pmatrix} \Longleftarrow$$
Adding 32.3 $A_{3,1}$ times the second row to the third row in parallel to produce a new third row;

$$\begin{pmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 1.00 & 4.89 & -12.6 \\ 0.00 & 0.00 & 1.00 & -2.00 \end{pmatrix} \Longleftarrow$$ Multiplying the third row by -1 in parallel to produce a new third row.

4.2.2 Summary on Matrix Inversion

Matrix Inversion is an important element of matrix computation. Gaussian Elimination algorithm makes compute matrix inversion in parallel which includes addition and multiplication in a parallel way. It has the complexity order of n^3 ($\mathcal{O}(n^3)$).

4.3 Convolution and Cross-Correlation

As wee saw in literature review, convolution and cross-correlation is required for Turbo decoding. In this section we give the equations to compute convolution and cross-correlation considering the fact that multiplication in time domain is convolution in frequency domain and vice versa. The discrete convolution of two sequences $v[n]$ and $x[n]$ is obtained by $y[n]$

$$y[n] = \sum_{m=-\infty}^{\infty} v[m]x[n-m]. \tag{4.30}$$

The discrete cross-correlation $r_{vx}[n]$ for two sequences $v[n]$ and $x[n]$ is

$$r_{vx}[n] = \sum_{m=-\infty}^{\infty} v[n+m]x[m], \quad n = 0,\ \pm 1,\ \pm 2,\ ... \tag{4.31}$$

where n is integer between $-\infty$ and ∞. The convolution has the complexity order of $\mathcal{O}(n \times m)$. The auto-correlation $r_{xx}[n]$ has the same expression such as cross-correlation $r_{vx}[n]$ with v replaced by x. And cross-correlation is written as a convolution by the following equation :

$$r_{vx}[n] = v[n] * x[-n] \tag{4.32}$$

Convolution is one of the main functions in signal processing. The importance of convolution is such as the importance of multiplication. Convolution in time domain is the multiplication in frequency domain and visa versa. It has complexity order of $\mathcal{O}(n \times m)$.

CHAPTER 5

HARDWARE ACCELERATION USING GPU

In chapter 4, we discussed parallel implementation of different operations, including FFT, matrix inversion and convolution. In this chapter, we discuss parallel implementation of those operations using CUDA and Matlab on GPUs. We will evaluate the performance of parallel processing of the operations on GPUs. This allows us to explain the advantages of GPU computing using Matlab.

5.1 Implementation on GPU using CUDA

We implement algorithms on GPU using CUDA, the Compute Unified Device Architecture. CUDA is a C/C++ based platform for development of parallel computing modules, invented by NVIDIA. It enables dramatic increase in computing performance by harnessing the power of the GPU. CUDA programs include two pieces : a host code on the CPU which interfaces to the GPU and a kernel code which runs on the GPU. The host level code is in charge of memory allocation on the graphic card and data transfer to and from the device memory. In this section, FFT, matrix inversion, matrix multiplication, and convolution are used for demonstration and benchmarking.

5.1.1 Matrix Multiplication

In this section, we describe the implementation of matrix multiplication as the fundamental block in many algebraic operations. Specifically, we consider the matrix multiplication of the form,

$$A_{N\times N} \cdot B_{N\times 1} = C_{N\times 1}, \tag{5.1}$$

implying :

$$\{C\}_{i,j} = \sum_k \{A\}_{i,k}\{B\}_{k,j}. \tag{5.2}$$

We will assign the calculation of each element of C, $\{C\}_{i,j}$, to one independent thread of GPU.

As we discussed in Chapter 3, we should compute the number of blocks per grid to match our data and simultaneously maximize occupancy, that is, how many threads are active at one time. Thread block size should always be a multiple of 32 ; *i.e.* $32, 64, 128, \ldots$. Because kernels issue instructions in warps (32 threads). For example, if there is a block size of 50 threads, the GPU will still issue commands to 64 threads and some threads are wasted. Moreover, better performance is expected when blocks are dimensioned based on the maximum numbers of threads and blocks supported in agreement with the compute capability of the card. For example, if there are N data elements, only N threads are needed in order to perform our computation. So in this case, the number of threads should be set to the smallest value that is a multiple of the the number of threads per block and that is greater than or equal to N. Therefore, the total number of blocks is

$$\frac{\text{total number of threads} + \text{threads per block-1}}{\text{threads per block}}, \tag{5.3}$$

where the total number of threads is equal to the number of rows of the matrix (in the range between 10 and 20000).

The *occupancy* is the time required for the passing of data through the slowest component in the communication path [38]. The occupancy limits the speed (frequency) of initializing the communication operations. Specifically, each data transfer has to wait until the critical resource is no longer occupied by the previous procedure. The theoretical occupancy is 25% when there are 32 individual threads per block with no shared memory. Since thread instructions are executed sequentially, executing other warps is the only way to hide latency and keep the hardware busy.

Kernel Implementation

Our implementation of the kernel for matrix multiplication on GPU is presented in Algorithm 5.1. A kernel code is executed on the GPU and requires to specify the grid and block dimensions, discussed in chapter 3. The kernel functions are specified by declaring them as __global__ in the code.

In Algorithm 5.1, the first line includes the _global_ void mutrixmul() which shows that it is a function for running on the GPU device. The integer variable *tmp* accumulates the product of row and column entries. The next line helps the thread to discover its row and column within the matrix. The *if* statement prevents the thread from falling outside the

Algorithm 5.1 Kernel code for Matrix Multiplication.

```
_global_ void mutrixmul(int *a, int *b, int *c)
{
_shared_ int b_s [N];
int tmp = 0;
int x = blockIdx.x · blockDim.x + threadIdx.x;
b_s[x] = b[x];
if $x < N$ then
    for int i = 0; i < N; i++ do
        tmp += a[N · x + i] · b_s[i];
    end for
    c[x] = tmp;
    tmp = 0;
end if
}
```

bounds of the matrix. Finally, the *for* loop computes the product of row and column of the matrix and the sum of these products are stored in *tmp*.

Experimental Results

Fig. 5.1 shows experimental results for the matrix multiplication with different (square) matrix sizes (ranging between 10 and 19×10^3) with non-zero entries. Our experiments are done on GPU Geforce GTX 660 Ti and multi-core devices. Further, we compare the results in case of using GPU with shared memory and global memory (without shared memory) and using multiple-cores in a CPU.

In this figure, the horizontal axis represents the size (number of rows) of the square matrix and the vertical axis is an execution computation time in micro seconds. This figure represents the three cases of computation time of matrix multiplication 1) using shared memory of GPU, 2) using global memory of GPU (without shared memory) and 3) the case of using multiple-cores for calculations.

In summary, using shared memory makes the computation times around 1.5 times faster for array size of 1000 or more. On the contrary, using multiple-cores (8 threads) makes the computations up to 2 times and 30 times slower for small and large matrices, respectively. Indeed, for square matrix size of 100 or less, the GPU is up to two times faster than the multiple-core CPU. For square matrix size between 100 and 1000, the GPU is just a little bit faster. For the largest considered matrices, the GPU is up to 30 times faster. The bigger the matrix size, the more speed up is obtained with the GPU. As shown in this figure, the

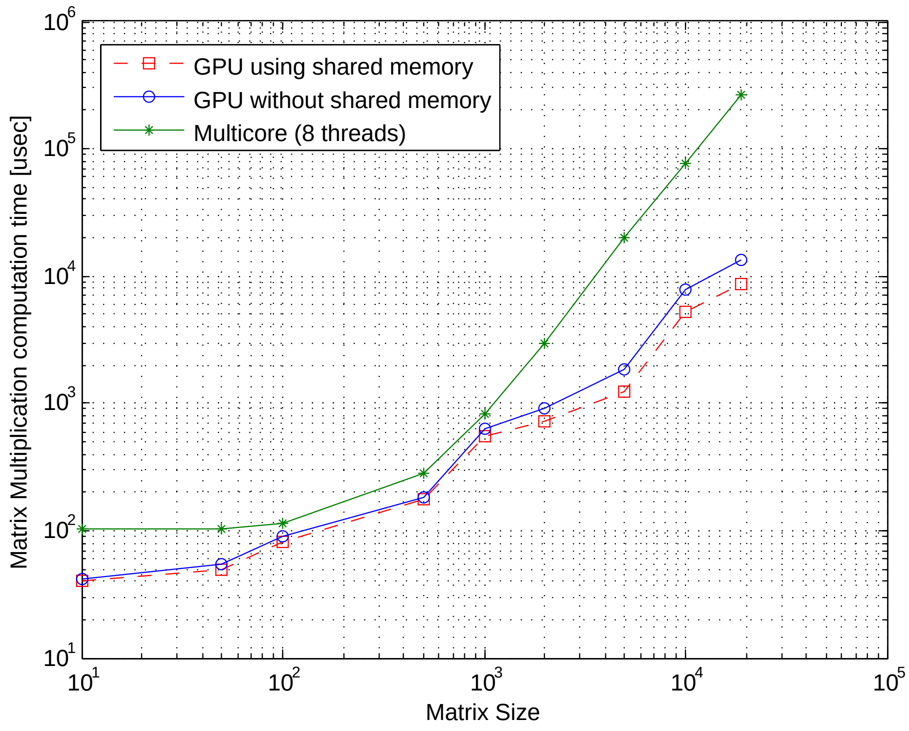


Figure 5.1 Computation time of matrix multiplication vs the matrix size for GPU (Geforce GTX 660 Ti)(with and without shared memory) and multi-core (8-core CPU x86_64). GPU Clock rate and CPU Clock rate are about 1 GHz and 3 GHz, respectively.

computation times decrease when we use the GPU with shared memory. The advantage of shared memory is to reuse the data and have an efficient computation process. All of these results show that GPU is a suitable option for the computations of matrix multiplication as it scales much less than the complexity order of vector-matrix multiplications, $\mathcal{O}(N^2)$ (there are N times N multiplications and $(N-1)$ sums).

5.1.2 Fast Fourier Transform

Fast Fourier Transform (FFT) is used in a vast variety of signal and image processing applications. This makes its fast and efficient implementation a vital need for many applications. In this section, we describe our implementation of FFT using CUDA.

NVIDIA's CUDA provides an interface, called CuFFT, for fast computing of FFT on NVIDIA GPUs. An NVIDIA GPU has hundreds of processor cores which can accelerate FFT computations up to 10 times [39]. CUDA helps in efficient computation of discrete Fourier transforms of complex and real-valued data sets. CuFFT provides a floating point performance for GPUs.

Furthermore, CuFFT uses a combination of GPUs and CPUs to carry out parallel computations. Specifically, CPUs are used to handle irregular and serial operations, resulting in a faster mathematical functionality. Although some other libraries implement radix-2 FFT, CuFFT has the following features which makes it a better alternative than other libraries :

- 1D, 2D, and 3D transforms of complex and real valued data,
- batch execution for doing multiple transforms of any dimension in parallel,
- 2D and 3D FFT transform with sizes in the range of between 2 and 16384,
- 1D transform sizes of up to 8 million elements,
- in-place and out-of-place transforms for real and complex data,
- double precision transforms on compatible hardware (*e.g.* GT200 and later GPUs),
- support for streamed execution, which enables simultaneous computation along with data movement.

Implementation and Experimental Results

In Algorithm 5.2, we present an example code for performing forward and inverse FFT computations using CuFFT library [39]. In Algorithm 5.2, the function in line 5 allocates size bytes of linear memory on the device and returns in devPtr, a pointer to the allocated memory. The function in line 7 creates a 1D FFT plan configuration for a specified signal size and data type. The batch input parameter specifies how many one-dimensional transforms CuFFT needs to configure. Finally, the function in line 9 and 11 execute a CuFFT single precision complex to complex transform plan as specified by direction. The CuFFT implementation uses the GPU memory, pointed to by data parameter, as its input [39].

Table 5.1 Computation time of FFT for different input sizes.

Input Vector Size	Computation Time of FFT [msec]
2560	173.2
25600	326.6
256000	997.7

In Table 5.1, we depict the FFT computation times using CuFFT library versus the size of input vectors. The input vector sizes for FFT are 2560, 25600 and 256000. The entries of the input vectors are randomly and uniformly generated complex float numbers. Moreover, Algorithm 5.2 1D Complex-to-Complex FFT/IFFT Transforms using CUDA[39].

```
$\#define\ NX\ N$
$\#define\ BATCH\ M$
$cufftHandle\ plan;$
$cufftComplex\ *data;$
$cudaMalloc((void **)\&data, sizeof(cufftComplex) * NX * BATCH);$
$/*\ Create\ a\ 1D\ FFT\ plan. */$
$cufftPlan1d(\&plan,\ NX,\ CuFFT\_C2C,\ BATCH);$
$/*\ Use\ the\ CuFFT\ plan\ to\ transform\ the\ signal\ in\ place. */$
$cufftExecC2C(plan,\ data,\ data,\ CuFFT\_FORWARD);$
$/*\ Inverse\ transform\ the\ signal\ in\ place. */$
$cufftExecC2C(plan,\ data,\ data,\ CuFFT\_INVERSE);$
$/* Note:$
$(1)\ Divide\ by\ number\ of\ elements\ in\ data\ set\ to\ get\ back\ original\ data$
$(2)\ Identical\ pointers\ to\ input\ and\ output\ arrays\ implies\ in-place\ transformation$
$/*\ Destroy\ the\ CuFFT\ plan. */$
$cufftDestroy(plan);$
$cudaFree(data);$
```

our implementation is done on Geforce GTX 660 Ti GPUs. As it is shown in this table, the computation time of FFT increases as the size of input vector is increased. The computational complexity of FFT operation (with no parallel computing) is in the order of $Nlog(N)$ where N is vector size. Our experimental results have shown that the computation time of calculating FFT using CuFFT has a smaller order than $Nlog(N)$. For instance, if the input size N scales up 10 times, the computation time using CuFFT does not scale up 10 time because of increasing in resource utilization. This shows that CuFFT is an appropriate choice for implementation of FFT on GPUs.

Matlab implementation of Radix-4 FFT implementation on CPU

In chapter 4, we described the architecture of radix-4 FFT, as characterized by the equation (4.25). Specifically, one has to implement p, μ and s matrices. In our implementations, we consider $N = 256$ and choose $f(n)$ as a specific input such that :

$$f(n) = sin(\beta n + \pi/4)R_N(n), \tag{5.4}$$

where

$$\begin{aligned} \beta &= 7.5\frac{2\pi}{N}, && (5.5)\\ R_N &= u(n) - u(n-N), && (5.6) \end{aligned}$$

and $u(n)$ is the discrete-variable step function :

$$u(n) = \begin{cases} 0 & n < 0 \\ 1 & n \geqslant 0 \end{cases} . \tag{5.7}$$

The pseudocode for implementation of radix-4 FFT of the signal $f(n)$ is presented in Algorithm 5.3. In this code, $\mathrm{kron(a, b)}$ is the Kronecker product of matrix a and b. Further, $eye(n)$ is an identity matrix of size $n \times n$. Based on equation (4.25), we need to calculate matrices p, μ and s. Meanwhile, calculating μ requires the calculation of matrix D which is done in the first inner loop (lines 9-13) from the equation (4.16). The second inner loop (lines 18-32) calculates matrix p from the equations (4.28) and (4.29). This will allow us to calculate μ (line 30) and s (line 40) using the (4.24) and (4.21), respectively. Finally, the FFT values are obtained by using equation (4.25) (line 46).

Using tic/toc instruction in Matlab, we measure the elapsed times for calculation of FFT. The elapsed time for calculating radix-4 FFT of the signal $f(n)$ with $N = 256$ on CPU is 104 milliseconds. In the case where CuFFT was used for calculating FFT on GPU, the computation time was 173.2 milliseconds for $N = 2560$. The complexity order of FFT is $Nlog(N)$. Thus, for $N = 256$ we expect that to have computation time of about 17 milliseconds. In radix-4 FFT implementation on CPU, the computation time is about 6 times bigger ($17ms \times 6 = 102ms$), while in CUDA small vector size is not efficient to be implement by GPU. Because the computation time for small vector size is mostly initialization time and communication time. Further, we know that Matlab has its own overhead. As a result, Radix-4 Fast Fourier Transform will have much less computation time in case of being implemented on GPU.

5.1.3 Matrix Inversion

We used Gaussian elimination algorithm to compute matrix inversion in parallel. As it was described in chapter 4, Gaussian elimination is a method for solving linear equations of the form

$$A_{N\times N} \cdot X_{N\times 1} = B_{N\times 1}.$$

In each iteration of this method, a pivot column is used to reduce the rows where the process of row reduction is divided in two steps. The first step is forward elimination which reduces a matrix to row echelon form. In the second step is back substitution which is applied to find

Algorithm 5.3 Radix-4 Fast Fourier Transform

```
$w = exp(-2\pi\ sqrt(-1)/N);$
for $i1 = 1\ \ to\ \ r$ do
    $N1 = r^{i1};$
    $b = 0 : r^{r-i1} : ((N1/r) - 1)r^{r-i1}$
    $K1 = zeros(N1/r, N1/r, (r-1));$
    $D_{Nk} = zeros(N1, N1);$
    $D_{Nk}(1 : N1/r, 1 : N1/r) = eye(N1/r);$
    $/ * Constructing\ D * /$
    for $i1 = 1\ \ to\ \ (r-1)$ do
        $w_p = b. * i;$
        $K1(:, :, i) = diag(w.^{(w_p.\times(-1))});$
        $D_{Nk}((i \times N1/r) + 1 : (i+1) \times N1/r, (i \times N1/r) + 1 : (i+1) \times N1/r) = K1(:, :, i);$
    end for
    $/ * Constructing\ P * /$
    $P1 = zeros(N1);$
    $col = 1;$
    $m2 = 1;$
    for $i2 = 1\ \ to\ \ N1$ do
        if $col \leqslant N1$ then
            $P1(i2, col) = 1;$
            $col = col + N1/r;$
        else
            $m2 = m2 + 1;$
            $col = m2;$
            $P1(i2, col) = 1;$
            $col = col + N1/r;$
        end if
    end for
    $/ *\ \ \mu\ \ / *$
    $u(:, :, r + 1 - i1) = kron(D_{Nk}, eye(r^{(r-i1)}));$
    $/ *\ \ p\ \ / *$
    $P(:, :, r + 1 - i1) = kron(P1, eye(r^{(r-i1)}));$
end for
for $i1 = 1\ \ to\ \ r$ do
    for $j1 = 1\ \ to\ \ r$ do
        $Tr(i1, j1) = w^{(-N\times(i1-1)\times(j1-1)/r)};$
    end for
end for
$/ * Constructing\ S * /$
$S = kron(Tr, eye(N/r));$
$/ *\ Computing\ FFT\ radix - 4 * /$
$/ * n = 4\ \ r = 4\ \ N = 256 * /$
$/ * TN = P1\ u1\ S\ P2\ u2\ S\ P3\ u3\ S\ P4\ u4\ S * /$
$TN = S\ P(:, :, 3)\ u(:, :, 3)\ S\ P(:, :, 2)\ u(:, :, 2)\ S\ P(:, :, 1)\ u(:, :, 1)\ S;$
$/ *\ F : FFT;\ f : input\ * /$
$F = (1/N)\ TN\ f;$
$/ * Computing\ \ IFFT\ radix - 4 * /$
$f = inv(TN)\ F;$
```

the solution. In fact, the unknown coefficients in the equation are represented as matrix A. Explicitly, this matrix is converted to an upper triangular matrix and then back substitution is applied on the result. The pivot element is the diagonal element for a specific iteration of the k loop, and its row is known as the pivot row.

Experimental Results

The computation time of matrix inversion using CUDA implementation is presented in Table 5.2 for different matrix sizes. As it is expected, the computation time increases when the size of matrix is increased. However, the computation times do not increase proportionally with the size of the matrix. As we discussed in Chapter 4, matrix inversion has a complexity order of N^3. The results in Table 5.2 show that the computation time of matrix inversion using CUDA scales with N^3, when using sufficiently large matrices, 500 and 1000 in this context. However, this does not hold for smaller matrices, for example, matrix size 500 is faster than matrix size 200. This is an odd behavior. Since it is running a function from a library, maybe it is because of wasting threads. While, it should be fully utilized warp which include 32 threads. The result shows that case of 200 wastes threads more than case of 500. This shows that CUDA is a good choice for implementing matrix inversion on GPUs for large matrices.

Table 5.2 Computation times of matrix inversion for different matrix sizes.

Matrix Size	Computation Time [msec]
50×50	445
200×200	4541
500×500	1500
1000×1000	11000

5.1.4 Convolution and Cross-Correlation

As it was discussed in chapter 4, convolution in time domain can be implemented by using multiplication in the frequency domain. In this section, we introduce different approaches for calculating convolution and discuss their computational times. Although the procedure looks very simple, its efficient implementation is challenging and needs careful design and allocation of hardware resources. In the following, we describe our proposed implementation of the convolution by using zero padding and shared memory in order to leverage memory and take advantage of memory management.

Kernel Implementation

As an example, consider the convolution of vectors a and b, $c = a * b$, where

$$a = [1, 2, 3, 4, 5], \tag{5.8}$$

$$b = [6, 7]. \tag{5.9}$$

In Table 5.3, we illustrate the procedure for calculating this convolution. As it is shown in this table, vector a is right-shifted in this case. Then, it is multiplied by reversed vector b. Finally, these multiplications are added in each instant.

Algorithm 5.4 shows part of kernel code to compute convolution in this naive approach. To compute the convolution, the elements of two (shifted and reversed) vectors are multiplied together and then the results are added to obtain the value of convolution at a point (a specific shift). As it is shown in Algorithm 5.4, it is necessary to verify the size of vectors with the if statement (line 6 of the algorithm).

Conventional (Naive) Approach

Using global memory is the most trivial and naive approach for sending data to device and memory in the process of calculating convolution. To obtain the lower performance limit on the computation of convolution, we implement it using global memory in this subsection. Explicitly, we do not use any of the techniques use in the next sections (zero padding, shared memory) for reducing the computation times. For this basic approach, we do not use any thread of GPU which results in an implementation only on the CPU. Further, we do not use any special technique for improving the computational efficiency of verification of boundary conditions (i.e. if statement in the kernel in Algorithm 5.4) in the kernel.

Zero padding

In order to improve the computation time of convolution, we proposed to use zero padding. The if statement in the kernel code (see line 6 in Algorithm 5.4) decreases the efficiency in

Table 5.3 Conventional procedure for calculating convolution : Right shifted vector a is multiplied by reversed vector b. Then, these multiplications are added in each instant.

	shift by 1	shift by 2	...	shift by 6($size\ a + size\ b - 1$)
reverse of vector b	[7,6]	[7,6]	$\ldots$	[7,6]
vector a	[1,2,3,4,5]	[1,2,3,4,5]	$\ldots$	[1,2,3,4,5]
Partial result	7×5	$6 \times 5 + 7 \times 4$	$\ldots$	6×1
$c(\centerdot)$	35	58	$\ldots$	6

Algorithm 5.4 Part of kernel code for calculating convolution using the naive approach.

```
...
for $int\ i = 0;\ i < convolution\_length; i++$ do
    $k = i;$
    $tmp = 0;$
    for $int\ j = 0;\ j < B\_vector\_length; j++$ do
        if $k \geq 0$ && $k < A\_vector\_length$ then
            $tmp = tmp + (A[k] \cdot B[j]);$
            $k = k - 1;$
            $C[i] = tmp;$
        end if
    end for
end for
...
```

parallel computing. Specifically, in parallel computing, all parts should work similarly to achieve near-optimal efficiency. Checking some conditions (*i.e.* the if statement in our case) results in loosing unique parallel computing structure. Using zero padding allows us to remove the if statement and achieve a unique structure for our parallel computing which can also takes care of verifying the boundary condition.

Table 5.4 Convolution procedure using zero padding : After zero-padding, right-shifted vector a is multiplied by reversed vector b. These products then are added in each instant.

	shift by 1	shift by 2	...	shift by 6
reverse of vector b	[0,0,0,0,7,6,0,0,0,0]	[0,0,0,0,7,6,0,0,0,0]	$\ldots$	[0,0,0,0,7,6,0,0,0,0]
vector a	[1,2,3,4,5,0,0,0,0,0]	[0,1,2,3,4,5,0,0,0,0]	$\ldots$	[0,0,0,0,0,1,2,3,4,5]
Partial result	7×5	$6 \times 5 + 7 \times 4$	$\ldots$	6×1
$c(\centerdot)$	35	58	$\ldots$	6

The procedure of calculating convolution with zero padding is illustrated in Table 5.4. In this case, the right shift and rotation are both applied to vector a. In fact, after doing rotational right shift on vector a, both vector a and reversed vector b are multiplied with each other without checking the size of vectors.

Zero padding, Shared Memory

As a second contribution for computing convolution, we used zero padding along with shared memory to achieve a better computational efficiency. Using zero padding allowed us to remove the if statement and have a kernel code which results in a good parallel computing structure. On top of zero padding, we proposed to use the shared memory to be able to reuse data between different threads.

Shared memory is located on the chip (see Fig. 3.13) and therefore it is much faster than the local and global memories. Explicitly, in the case of using fully utilized threads and warps, shared memory latency is around 100 times less than the latency of un-cached global memory latency. As it is shown in Fig. 3.13, shared memory is assigned for each thread block and all of the threads in the block have access to the same shared memory. This capability results in high performance parallel algorithm.

Summation Reduction

As it is shown in Algorithm 5.4 (see line 7), summation is a main part of computations in calculating the convolution. In the following, we adopt summation reduction instead of simple addition (sum) as a further improvement on top of discussed techniques (*i.e.* on top of zero padding and shared memory). The adopted summation reduction technique has low arithmetic intensity and uses memory bandwidth in an efficient way.

Simple sum is done using only one thread which take time proportional with the length of the array. However, since we have hundreds of threads available for computing, we can design a new sum algorithm which takes advantage of parallel computing over multiple threads. In the following, we present two different approaches for calculating the sum in a parallel way.

The first approach is called *parallel reduction with sequential addressing*, and is illustrated in Fig. 5.2. As it is shown in this figure, each thread is in charge of adding two values and storing the result. This will combine two entries into one and *reduce* the number of additions in the next step. This reduction is repeated in the next step to the remaining entries. As shown in Fig. 5.2, at each step, the number of additions of two values is reduced by half.

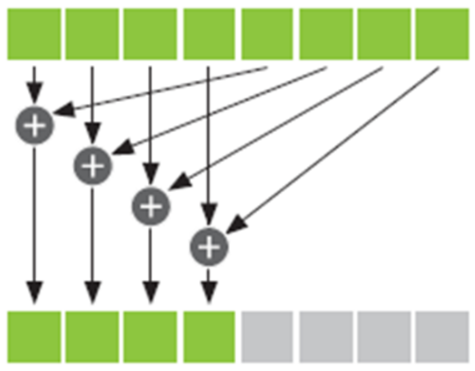


Figure 5.2 One step of a summation reduction based on the first approach : Assuming 8 entries in cache variable, the variable i is 4. In this case, 4 threads are required to calculate the sum of the entries at the left side with the corresponding ones at the right side.

Algorithm 5.5 Summation reduction using first approach [40].

```
// for summation reductions, threadsPerBlock must be a power of 2
int  $i = blockDim.x/2$;
**while** $i \neq 0$ **do**
  **if** $cacheIndex < i$ **then**
    $cache[cacheIndex] += cache[cacheIndex + i]$;
    $\_syncthreads()$
    $i /= 2$;
  **end if**
**end while**
```

Algorithm 5.5 demonstrates summation reduction using the first approach. The first step of algorithm starts with variable i as half of the number of threadsPerBlock. The threads with indices less than this value i are used for computing while the rest are left un-used. Specifically, the two entries of cache variable are added if the thread's index is less than i. This addition is protected with using an if statement : if(cacheIndex < i).

Each thread will take the entry at its index in cache variable and adds it to the corresponding entry in the other half. The result is then stored at the entry with the same index as the thread index. For example, assume that there are 8 entries in cache variable and, hence, variable i is 4. As shown in Fig. 5.2, in this case, 4 threads are required to calculate the sum of the entries at the left side with the corresponding ones at the right side.

In our second approach, we combine the arrays in a different way than in the first approach. Specifically, the entries are combined together based on a tree structure, as shown in Fig. 5.3.

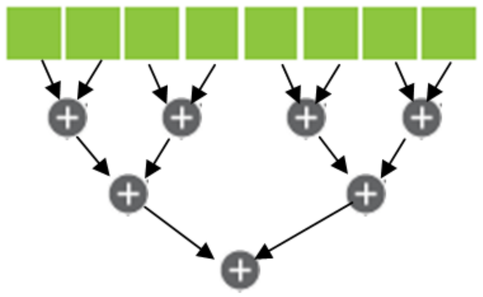


Figure 5.3 Tree-based summation reduction : Entries are combined together based on a tree structure.

The first approach is more suitable for parallel processing because of having consecutive indexing.

In the first approach, parallel reduction with sequential addressing, there are a total number of $\log(N)$ steps of calculating. In each step, k, there are $N/2^k$ independent operations which are done in parallel (using multiple threads). As a result, the total number of operations is in the order $\mathcal{O}\Big(log(N)\Big)$.

In the second tree-based approach, with $N = 2^K$, there is a total of

$$\sum_{k=1}^{K} 2^{K-k} = N - 1 \tag{5.10}$$

operations which has a higher order of computational complexity than the first approach. As a result, it is more efficient to use parallel reduction with sequential addressing than the tree-based reduction. Further, the advantage of reduction is in efficient use of memory bandwidth which makes the arithmetic intensity very low. In the following, we use parallel reduction with sequential addressing for computation of sum (referred to as summation reduction) in the convolution.

Experimental Results

We have run our experiments to calculate the computation time of convolution using different approaches. For each approach, we used different vector sizes and measured the corresponding computation times, as presented in Table 5.5. In the following, we discuss our results obtained for each approach :

Conventional (Naive) Approach : By measuring the initialization time for

Table 5.5 Computation times of convolution for three different scenarios : Naive approach, zero-padding and zero padding with shared memory.

A vector size	B vector size	Computation Times [sec]		
		Naive Approach	Zero Padding	Zero Padding, Shared Memory and Summation Reduction
5	2	8.6×10^{-5}	2.54×10^{-4}	2.2×10^{-4}
16	8	6.5×10^{-4}	2.6×10^{-3}	2.6×10^{-4}
32	16	2.5×10^{-3}	1.04×10^{-2}	2.89×10^{-4}
64	32	9.7×10^{-3}	4.18×10^{-2}	1×10^{-4}
128	64	3.8×10^{-2}	1.64×10^{-1}	1.3×10^{-5}
256	128	.15	.6	3.6×10^{-5}
512	256	.55	2.189	5×10^{-5}
1024	512	2.04	8.569	7.6×10^{-5}
2048	1024	7.95	34.348	8×10^{-5}

computation of different vector sizes, we conclude that the computation time is mostly due to the initialization time, especially for small vectors. Further, comparing the computation times for the last two entries of the table shows that the computation times of the convolution are from the order N^2. This was verified in Chapter 4 where we analyzed the complexity order of convolution. However, for smaller vector sizes (first few entries of the table), the computation times do not have a second order relation with the size of input vectors. This fact is resulting from the initialization time (of memory) which is not negligible (compared to other factors) in the cases with small vector sizes.

Zero padding : The computation time is mostly the initialization time for the first two cases. For the rest, as the sizes of a and b double, the computation time becomes 4 times. This can also be concluded from the complexity order, discussed in Chapter 4. Compared to naive approach, zero padding is slower. Because in this case we have more computations and we retrieve more data because of zero padding. Moreover, it does not have shared memory. Zero padding is only interesting when using shared memory. Vectors with smaller size need few processors to be computed while vectors with bigger size need more processors. Using shared memory allows us to read 1 data and send it everywhere, while in case of zero padding we can not do that.

Zero padding, Shared Memory, Summation Reduction : Our results for the last scenario in Table 5.5 show that for vector sizes smaller than 32 the computation times are greater than those for bigger vector sizes. Although this may seem to be invalid, it can be justifies by understanding the computation mechanism in GPUs (as discussed in Chapter 4). CUDA implementation combines every 32 threads as instructions are issued per warp. As a result, CUDA implementation reaches its maximum computation efficiency when 32 threads per warp are fully utilized. High computation times for vector sizes smaller than 32 is resulting from the overhead involved in the processing of small vector sizes.

The results in Table 5.5 show that using shared memory makes computation much faster. Further, the results imply that computation time does not scale with the size of input vectors (*i.e.* $N \times M$ where N and M are sizes of A and B vectors).

The computational complexity of convolution is proportional to the number of required memory accesses. However, data re-using with the aid of shared memory results in computation time of convolution not to scale with the size of input vectors $N \cdot M$ (where N and M are the sizes of A and B vectors). In fact, we are able to carry out the computation of convolution using M warps, each performing N computation, as opposed to $N \cdot M$ individual computations when shared memory is not used.

Each thread is considered a compute engine where every 32 threads compose one warp. Since there are 192 CUDA cores on our GPUs, we have a total of around 6000 compute engines. Therefore, performing 1000 or 2000 (our maximum vector size based on Table 5.5) multiplications will not saturate the bandwidth of memory. This makes GPU with shared memory implementation a good choice for performing convolution.

Our experimental results shows that memory management obtained by deleting the if statement in the kernel code improves the computational efficiency. Such a structure is sometimes referred to as Single Instruction and Multiple Data (SIMD structure) where multiple data are executed with the same operation at the same time. Moreover, we observed that the occupancy varies by the size of input vectors. Specifically, bigger vector sizes uses warps fully which increases the computational efficiency.

For the last entry in Table 5.5, there are a total of 1024×2048 multiplications and 1023×2048 additions. This results in a total of about $2^{22} \simeq 4 \times 10^6$ operations. Now, using a clock frequency of 1 GHz for compute engines, it takes

$$t = \frac{4 \times 10^6 \text{operation}}{10^9 \text{operations/second}} = 4 \times 10^{-3} (second)$$

to do the computations. As the computation time for that entry is 8×10^{-5} second, 50 compute engines have been used for the computation ($4 \times 10^{-3} / 8 \times 10^{-5} = 50$). Further, since every

32 compute engines form a warp, a total of 2 warps are used for the computation. Moreover, two warps use one shared memory for the computations (see Fig. 3.13). Therefore, two warps with one shared memory can be used for performing the summation reduction. This allows us not to need extra communications for performing the summation reductions. In the case with big input vectors (vector sizes of 2048 and 1024) only 2 warps are required while in other cases, only one warp is used. In the cases with small vector sizes, only a tiny portion of the warp is used for computations.

5.2 Implementation on GPU using Matlab

Matlab parallel computing toolbox provides embedded implementation of data-intensive signal processing algorithms for multi-core processors, GPUs and computer clusters. Matlab built-in parallel computing features have been shown to be efficiently implemented. However, one has to maintain the transfer of data between the GPU and CPU. In this section, we discuss the implementation of different signal processing operations using MATLAB and evaluate their computational efficiency.

Before discussing GPU implementation of different operations using MATLAB, we mention the main functions required for GPU parallel computing. Some of these functions are listed in below :

- GPUDevice shows GPU devices attached to the computer and lists their properties,
- GPUArray transfers an array from Matlab workspace [1] to the GPU device,
- gather retries data from the GPU to the Matlab workspace (computer memory). In fact, this function transfers the results from GPU memory to the computer memory (RAM).

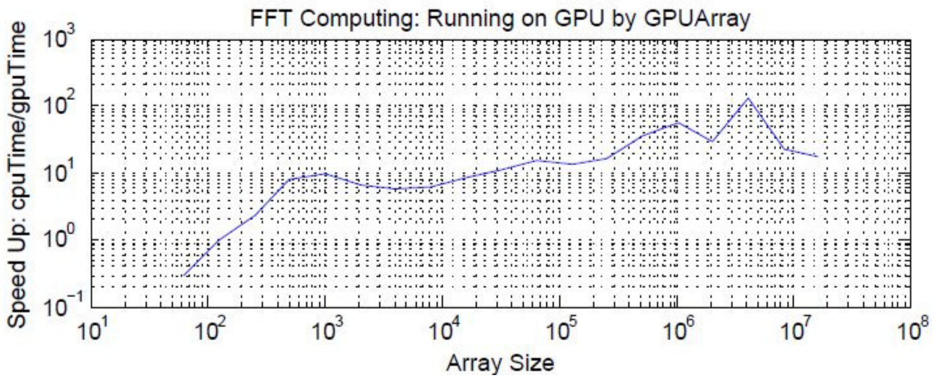
5.2.1 FFT

The process of calculating FFT using MATLAB implementation for GPU follows three consecutive steps. First, gpuArray() function is executed to transfer the data from Matlab workspace (computer memory ; *i.e.* RAM) to the memory of GPU device. Then, FFT operation is executed on the GPU device and the result is stored on the memory of the GPU. Finally, gather function transfers the results from the memory of GPU to the computer RAM (Matlab workspace). This process is shown in Algorithm 5.6.

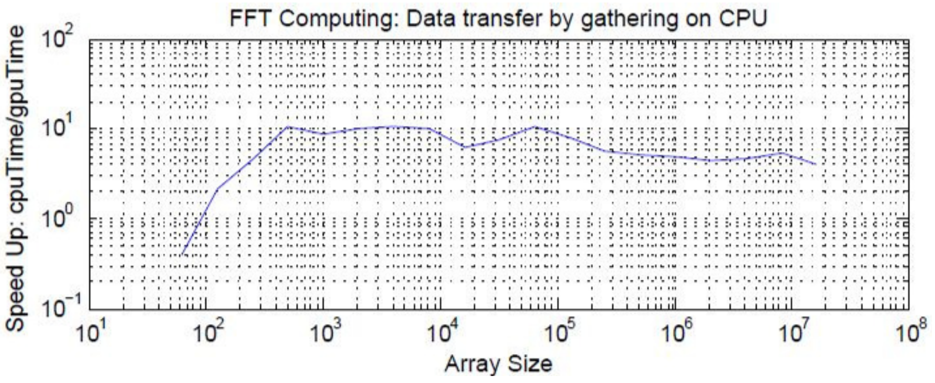
1. Matlab workspace is an environment to keep the numerical data and program.

Algorithm 5.6 FFT GPU Computing in MATLAB

```
1: $A = gpuArray(rand(M, 1));$
2: $B = fft(A);$
3: $C = gather(B);$
```



(a) GPUArray



(b) gather

Figure 5.4 CPU/GPU times for (a) GPUArray method (b) gather method in computation of FFT using Matlab implementation on GPU.

Fig. 5.4 shows the resulting cpu-time/gpu-time versus input array size. Specifically, the vertical axis represents the speedup as the ratio of the computation time on CPU to the computation time on GPU. The input vectors used for our experiments are floating point numbers with double precision. We vary the size of vectors from 60 to 10^6.

In Fig. 5.4(a), the computation time of GPU corresponds to the transfer of data from the computer CPU to the GPU memory and the calculation of FFT on GPU. Our goal is to show the speedup of FFT computation on GPU compared to FFT computation on CPU. As it is shown in this figure, computation using GPU takes less time (about 10 times faster) than computing on CPU, especially for array sizes greater than 100. For smaller array sizes, overhead takes more time compared to the computation time of FFT. This fact is because memory bandwidth on GPU is greater than memory bandwidth on CPU.

In Fig. 5.4(b), the computation time of GPU includes the transfer of data from computer memory to GPU, calculating the FFT on GPU, and transferring the result from GPU to the computer memory. This will allow us to compare the overall computation time using GPU compared to the case where they are done on the CPU. Our experimental results in Fig. 5.4(b) shows that using GPU is still a faster approach than using CPU. Explicitly, it achieves a speedup of around 10 and 100 for array sizes around 400 and 4×10^6.

5.2.2 Matrix Inversion

Similar to the procedure for calculating FFT, Matlab-based matrix inversion on GPU requires three consecutive steps. First, we use gpuArray(MatrixA) function to transfer data from the Matlab workspace (computer RAM) to the memory of GPU device. This will configure the next function to be executed on the GPU. Specifically, by executing inv function, Matlab implementation of matrix inversion on GPU is done. Similar to the FFT calculation steps, the results in this case are stored on the memory of GPU. Executing gather function will return the results back to the computer memory where we would have access to (via Matlab workspace). Algorithm 5.7 summarizes these steps in the calculation of matrix inversion on GPU.

Algorithm 5.7 Matrix inversion GPU Computing in Matlab

```
1: $A = gpuArray(MatrixA);$
2: $B = inv(A);$
3: $C = gather(B);$
```

In Fig. 5.5, the speedup of computing matrix inverse is depicted versus the size of input matrices. The vertical axis represent the speedup which is the ratio of the computing time on CPU to the computation time using GPU. The input matrices are square and their entries are floating point numbers with double precision. We change the size of input matrices between 60×60 and 4000×4000 and measure the computation times in each case.

The curve in Fig. 5.5(a) shows the speedup versus the input matrix size. For this figure, the computation time on GPU is the sum of times required to transfer input data from

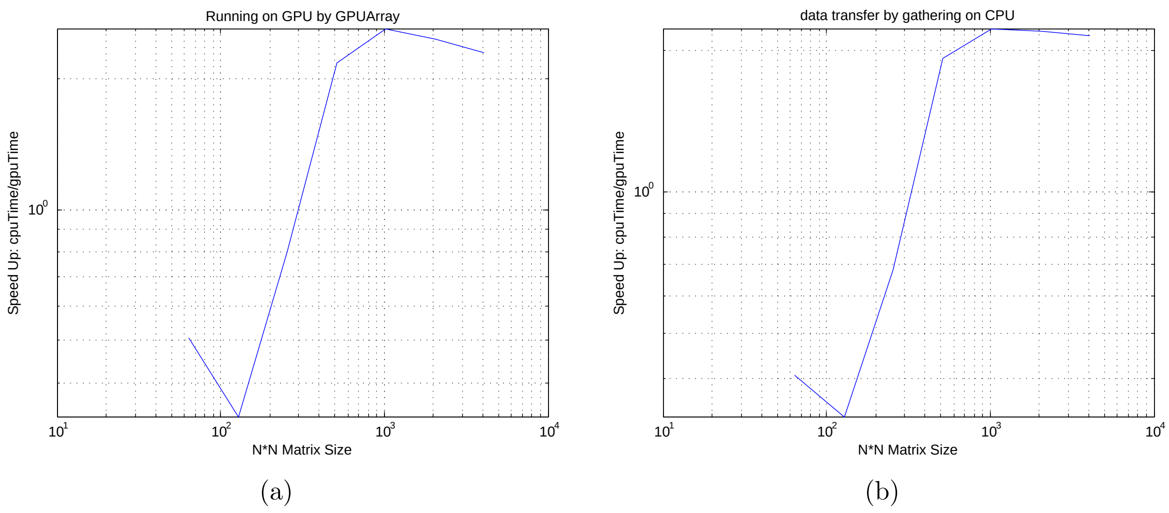


Figure 5.5 Speedup vs matrix size for Matlab based computation of matrix inversion on GPUs : (a) including the times of data transfer from RAM to GPU memory and calculation of matrix inverse on GPU (b) including the times of data transfer to and from GPU memory and calculation of matrix inverse on GPU.

computer memory to the GPU memory and calculate the matrix inverse [2]. We have taken into account all of the elapsed times (data transfer from computer memory to GPU memory, calculating the inverse on GPU, and transferring the result back to the computer memory) for computing the matrix inverse in the curve of Fig. 5.5(b). This curve will show us if there is any overall benefit by using GPU for computing when the data is originally on the computer memory and the result is needed on the computer memory.

In this figure, the maximum speedup is around 2 which happens for a matrix size of 1000×1000. Similar to the computation of FFT, the minimum speedup happens when the size of input matrix is small (130×130). The gain of using GPU for computing matrix inverse is significant when the size of input matrix is bigger than 300×300. However, as the maximum speedup (in Fig. 5.5) is around 2, using Matlab for computing matrix inverse may not be a good alternative for using CUDA (or even computation using CPU).

5.2.3 Matrix Addition

Similar to the previous cases, the procedure for adding two matrices on GPU using Matlab implementation needs transfer of input matrices to the memory of GPU, performing addition,

2. It does not include the time required for transferring the data back to the computer memory. This will allow us to understand the performance in case all of the other (remaining) computations are done on the GPU.

and then returning the result back to the computer memory. This is shown in Algorithm 5.8 where gpuArray(MatrixA) and gpuArray(MatrixB) transfers the input data to the memory of GPU and gather(C) returns the result to the Matlab workspace.

Algorithm 5.8 Summation GPU Computing in Matlab

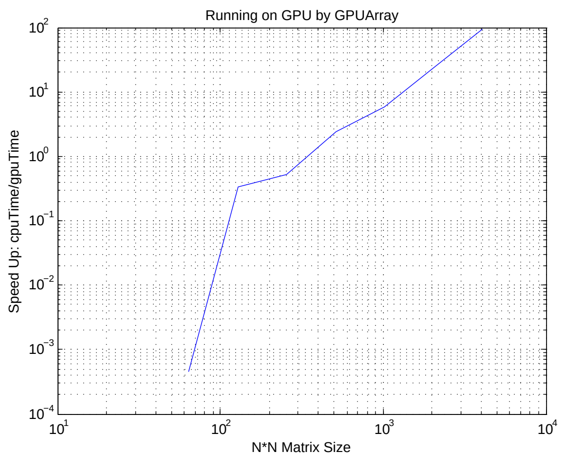
```
1: $A = gpuArray(MatrixA);$
2: $B = gpuArray(MatrixB);$
3: $C = A + B;$
4: $D = gather(C);$
```

The computational advantage of using a GPU instead of a CPU for computing matrix additions is shown in Fig. 5.6. Similar to the curves in Fig. 5.5, the vertical axis represents the speedup and the horizontal axis is the number of elements of input matrices. We have used square matrices of size $N \times N$ as our input matrices where their entries (elements) are floating point numbers with double precision.

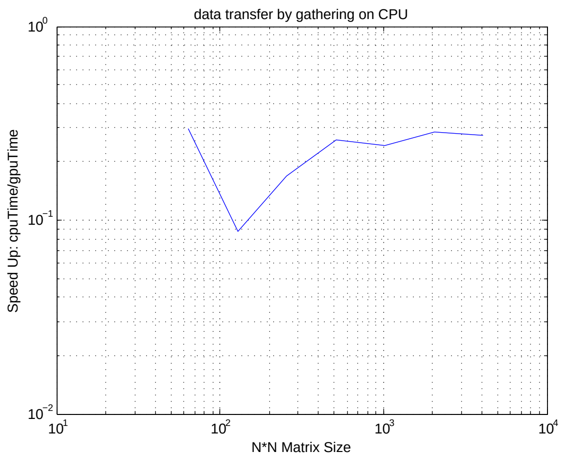
In Fig. 5.6(a), the GPU time includes the time required for transfer of data from computer memory to the GPU memory and computing the addition on the GPU. However, the GPU time for Fig. 5.6(b) also includes the time required for the transfer of result from the GPU memory to the computer memory.

As it is shown in Fig. 5.6(a), the speedup increases dramatically when the matrix size increases. Further, when the matrix size is bigger than 350×350, there is a steady increasing trend in the changes of speedup. This steady increase reaches a gain of 100 for using GPU when the size of matrices are such that : $N \cdot N = 4000$. However, for small matrices, the communication overhead results in a poor performance compared to the computation on CPU.

By studying the curve in Fig. 5.6(b) and comparing it with Fig. 5.6(a), one may notice that the overhead time for transferring data to computer memory (from GPU memory) is more than the time of adding two matrices on GPU. In other words, computing matrix addition on GPU is beneficial when the result is used for other operations on GPU and the communication overhead is negligible considering all of the operations done on the GPU.



(a)



(b)

Figure 5.6 Speedup vs matrix size for Matlab based computation of matrix addition on GPUs : (a) including the times of data transfer from RAM to GPU memory and calculation of matrix addition on GPU (b) including the times of data transfer to and from GPU memory and calculation of matrix addition on GPU.

5.2.4 Matrix Multiplication

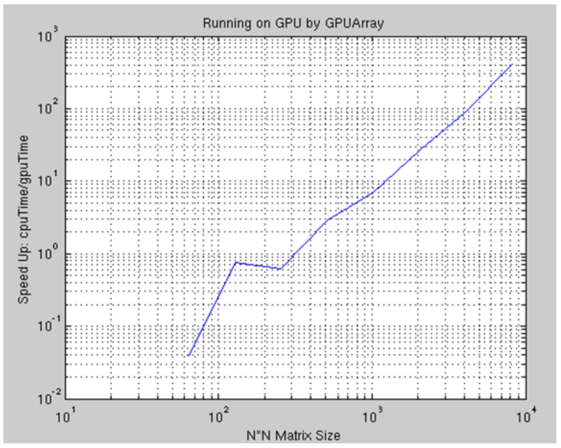
In this section, we study the efficiency of using Matlab implementation for computing matrix multiplication on GPUs. As it is presented in Algorithm 5.9, the computation on GPU is done by transferring the input matrices from the computer memory to the GPU memory. Then, matrix multiplication is done on GPU and the result is transferred back to the computer memory.

Algorithm 5.9 Matrix Multiplication GPU Computing in Matlab

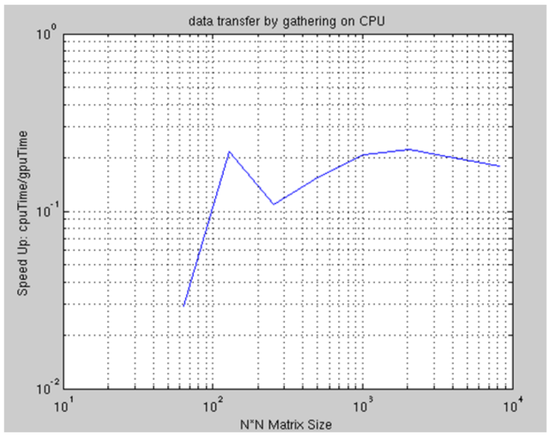
```
1: $A = gpuArray(VectorA);$
2: $B = gpuArray(MatrixB);$
3: $C = A \cdot B;$
4: $D = gather(C);$
```

In Fig. 5.7, we have shown the speedup of using GPU over using CPU for computing matrix multiplication. The input matrices and the times included in calculating the GPU computing time are similar to those for Fig. 5.6.

As it is shown in Fig. 5.7(b) and Fig. 5.7(a), the speedup of matrix multiplication increases as the size of input matrices increases. For small matrices, the overhead involved in the transfer of data between computer memory and GPU memory makes GPU not a suitable



(a)



(b)

Figure 5.7 Speedup vs matrix size for Matlab based computation of matrix multiplication ($Y = A \cdot X$) on GPUs : (a) including the times of data transfer from RAM to GPU memory and calculation of matrix multiplication on GPU (b) including the times of data transfer to and from GPU memory and calculation of matrix multiplication on GPU.

choice. However, using GPU for computing of multiplication is a good alternative for using CPU when the multiplication result is not needed to be sent to the computer memory. Essentially, GPU is a better alternative than CPU for computing when a number of different operations are performed on it such that the overhead of data transfers is negligible.

5.3 Summary on hardware acceleration using GPU

In this chapter, we discussed parallel computing of different matrix operations on GPU devices using CUDA and Matlab implementations. We have described our methods for increasing the computational efficiency of implementations by using shared memory in different situations. Our experimental results provided an understanding of the performance of each computational operation using different implementation strategies.

Our experimental results showed that parallel computing using CUDA improves the computational efficiency for the calculation of FFT, matrix inversion, matrix convolution and matrix multiplication, compared to the computation on CPU. Shared memory allows us to achieve higher bandwidth to accelerate processing [3] on top of the speedup obtained by using threads and a unified memory model.

Matlab has some built-in functions which are optimized for parallel computation of

3. via re-using data

matrices and vectors on GPU devices, attached to the computer. However, one needs to transfer data from the computer memory to the memory of GPU and then transfer the results back to the computer memory. In cases where this communication overhead is negligible compared to the total computation time of all operations, Matlab implementation on GPU is a good alternative for conventional (serial) computing on CPU. Our experimental results showed that Matlab based parallel computing for calculation of FFT, matrix multiplication and matrix addition on GPUs can outperform computing on CPUs. It was also shown that Matlab implementation for calculation of matrix inverse may not achieve a better computational efficiency than computing on CPU, due to the associated communication overheads.

CHAPTER 6

COMPUTING FFT USING DPDK AND MKL ON CPU

In chapter 5, we discussed parallel computing using GPUs as a solution for acceleration of some key operations in the process of telecommunication standards (especially LTE). In this chapter, we study the computational efficiency of implementing some of those operations on multiple central processing units (CPU). Specifically, we study the sources of randomness of the computation times in a data processing center. Moreover, we propose to use DPDK and MKL as two important tools for control and isolation of computational loads on multiple computing cores. As we will discuss in this chapter, DPDK and MKL will allow us to achieve a near real time computational performance for our operations which is desired in large scale systems.

6.1 Sources of non-determinism in data centers

Data centers are one of the building blocks of a cloud based computing system. Specifically, each data center provides a large amount of computing and storage devices. In current generation of systems, all of the computations needed for serving a request is handled at a data center. As a result, providing low latency and real time response is a vital requirement for each data center. In this section, we study and discuss some of the main sources (reasons) of randomness which may cause unacceptable computational performance in a data center.

A high level overview of the architecture of a data center is shown in Fig. 6.1. Each data center is composed of a number of racks of servers, also referred to as blade servers. A blade server is a computer board with high computational capabilities with an optimized design to reduce the maintenance costs. Usually, a server has a few processors and memory units (DRAM[1]). Each processor is composed of some cores and level-three (L3) cache where each processor consists of one CPU, one level-one (L1) cache and one level-two (L2) cache. Different modules of a server are connected together via Quick Path Interconnect (QPI) links in all levels of hierarchy. However, different blades and racks are connected together via a

1. Dynamic Random Access Memory

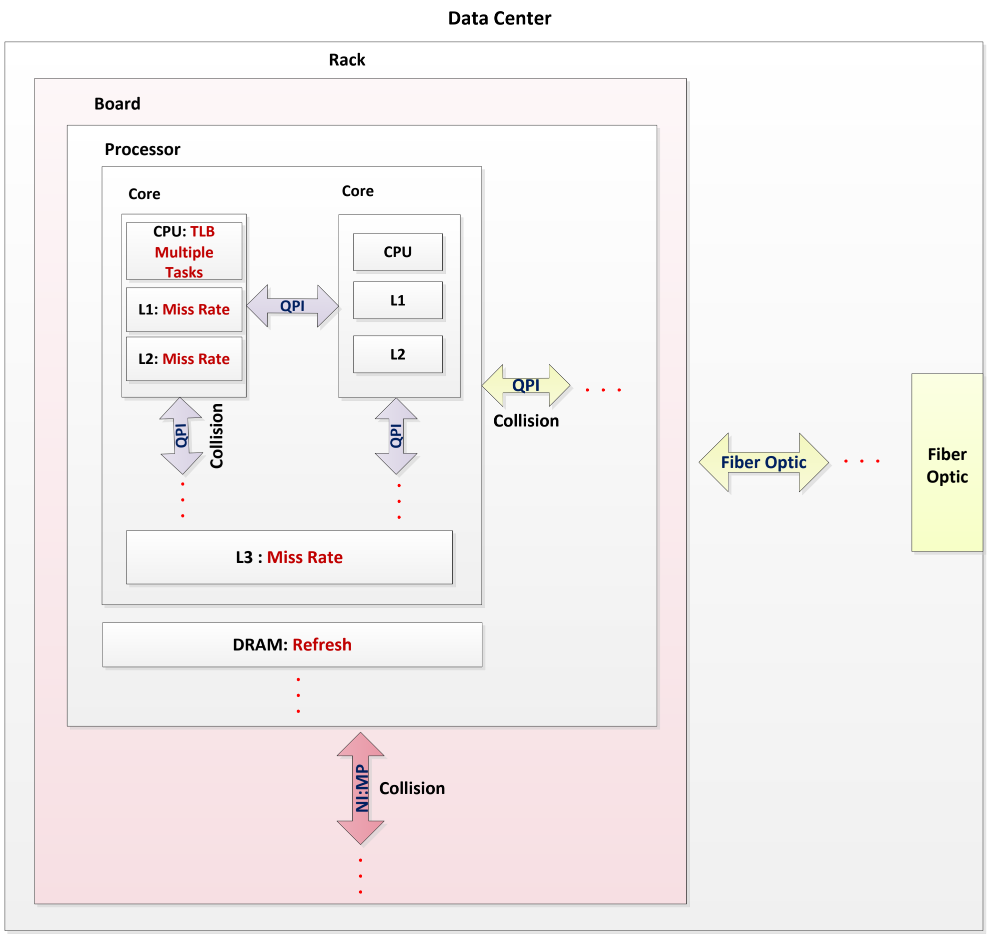


Figure 6.1 Architecture of a data center.

high speed fiber optic connection.

Generally, having multiple parallel running tasks where they are competing for computational and memory resources causes some complications. The random nature of these tasks (both in terms of submission time and the time needed for processing) creates a scenario where deterministic computation times are almost not achievable. TLB misses, concurrency issues and cache miss rates are some of the reasons for having random computation times.

In the following, we discuss some specific sources of non-determinism in the computation times :

– *TLB miss* is the translation lookaside buffer which is in charge of translating the virtual data addresses to the corresponding physical addresses. In fact, it is a cache of recent virtual to physical address mappings.

– Cache is a high speed memory which is intended to store the data from frequently used memory addresses. In some cases, CPU needs to recover a data from (or store to) the cache and it can not be found there. Therefore, the data has to be loaded from (or stored to) the memory (DRAM). This is referred to as *cache miss* and may be the cause of non-determinism in the handling of a request in some cases.
– *DRAM refresh* is done because of the dynamic nature of DRAMs which needs the memory be refreshed periodically in order to prevent the memory cells from loosing their contents.
– Since, there are several cores on a board, they compete to access QPI between boards which is used for fast communication between the processors on a board. This *Competition for QPI* is the source of uncertainty in the time required for handling a request and may cause high latency for some requests.
– The communication between different server blades is done through their network interfaces. As in other networking scenarios, *collision* and *congestion* are the main sources of randomness in responding to a request made through a network connection.
– The *interrupts* may cause a processor core to suspend its normal processing tasks to carry out a special request with higher priority. Depending on the architecture of the server and the operating system (or firmware), this may result in high computation times. Management of threads and isolation of cores lead to handle of the interrupts in a multi-core system.

In Table 6.1, we present a summary of different sources of variability in the computation times.

As it is mentioned in Table 6.1, cache miss, TLB miss and ECC (Error-correcting code) memory are the sources of variability in the fetch instruction and execute. ECC is a protocol that can detect and correct the most common kinds of internal data corruption. Cache locking techniques are proposed as a solution to overcome these issues. As another recent solution, DPDK provides huge page sizes to reduce the TLB misses. Using DPDK buffers may also help us in improving the cache misses.

Multi-core architecture of servers allow us to increase the computational capacity by parallel processing over multiple number of cores. However, task switching [2], task migration [3] and interrupts can potentially result in a temporary performance drop. Isolcpus is a boot parameter which allows us to isolate CPUs from scheduler algorithms and overcome the

2. Refers to operating systems or operating environments that enable you to switch from one program to another without losing your spot in the first program.

3. Task migration is the process of moving from the use of one operating environment to another operating environment

Table 6.1 Computer architectural features which cause variable delay.

Category	Source of variability	Possible solutions
Fetch Instruction and Execute	Cache miss TLB miss ECC	Cache locking
Multi-Tasking	Task switching Interrupts (task migration)	Isolcpus (Isolated CPUs) assign Interrupts to specific cores Interrupt routing (thread affinity)
Memory Sharing	Cache coherency serialization	Avoid sharing by using message passing or DPDK buffers
Communications (between threads)	Memory to memory copy data size distance (number of hops) collisions and types location in memory hierarchy	DPDK using thread placement and migration
System Management Features	DRAM refresh memory scrubbing hot swaps Clock frequency management	Do everything in cache - (cache locking) disable DRAM Scrubbing fast switching protocol
GP-Communications	–	DPDK

issues involved in the conventional task switching and task migration. Further, DPDK has the ability to control the interrupts and prevent task switching by assigning the interrupts to specific codes and reducing the computation time of processes (applications).

Interrupt routing determines how incoming interrupts are directed to the CPU interrupt request numbers. One issue of multi-tasking and multi-core environment is to assign processors and cores to specific tasks. For example, one processor or core handles the GUI, another handles the database and the others handle the real time functions. This is done through the magic of thread affinity, the ability to associate certain programs (or even threads within programs) with a particular processor or processors or cores. In fact, thread affinity allows software threads to be executed within the scope of specific processing resources.

DPDK can also provide the ability of memory sharing by using buffer management instead of message passing. This will prevent cache coherency and serialization. Cache coherency is the consistency of data stored in local caches of a shared resource. To reduce the serialization of processes caused by mutual exclusion (waiting to enter critical sections) or dependencies, we propose to use DPDK buffers.

Communication issues can also be handled by using thread placement and migration capabilities of DPDK. It will also provide management tools for general purpose communications.

Removing sources of randomness in the computation times is a very challenging task. In general, it is almost impossible to remove all sources of randomness and have a deterministic computation time on servers. However, it is possible to reduce the randomness by using the solutions that we discussed earlier in this section. Specifically, in the following sections, we discuss the computational performance of such servers when DPDK is used for improvement. As we will see, it provides a level of control over the hardware which was not available by using conventional operating systems and software.

6.2 DPDK

Exploding demand for network bandwidth has drawn attention to Intel DPDK as an enabling solution for high performance packet processing to accelerate signal processing in telecommunication systems. Essentially, DPDK accelerates the delivery of packets from the network interface card to the application layer. The set of optimized libraries and drivers, available in DPDK (described in Chapter 3) enable fast packet processing based on Intel architecture. It supports Intel processors in 32-bit or 64-bit mode from Intel Atom to Intel Xeon generation.

DPDK is compatible with Linux operating system and implements a run-to-completion model for packet processing (running as an execution units on logical processing cores). In this model, each task runs until it is finished (with no interrupt). Further, all of the computing and storage resources are allocated before calling data plane applications. This model does not support a scheduler, and all devices are accessed by polling, which reduces the overhead produced by the interrupt processing in high speed applications [33].

As it was shown in Fig. 3.15 in Chapter 3, Intel DPDK libraries are executed in userspace by creating the Environment Abstraction Layer (EAL). The EAL provides an interface for the interaction with the application. Queue functions, buffer and memory pool management functions are other capabilities of DPDK. Further, it has functions to classify packet flows and to pass the packets from the network interface card to the application. However, DPDK lacks a sophisticated mathematical library for complex signal processing applications. To address this issue, we proposed to use Intel MKL library within DPDK for such operations, as described in the following.

6.2.1 Combining DPDK and MKL

Intel has developed MKL as a mean to effectively leverage its processors for intensive signal processing and mathematical applications [41]. It is probably the fastest library for Intel processors as it provides support for threads and vectors using features that are not easily accessible via other software. Our initial experiments showed that they are not directly compatible and one has to follow specific steps in order to get them working together (as explained in Chapter 3). These steps include making changes to enable MKL in combination with DPDK.

In order to use MKL along with DPDK, initially one has to modify the DPDK makefile to add the following line into CFLAGS :

−I/opt/intel/composer_xe_2013_sp1.0.080/mkl/include.

Further, the following has to be added to section # *default path for libs* of file *rte.app.mk* :

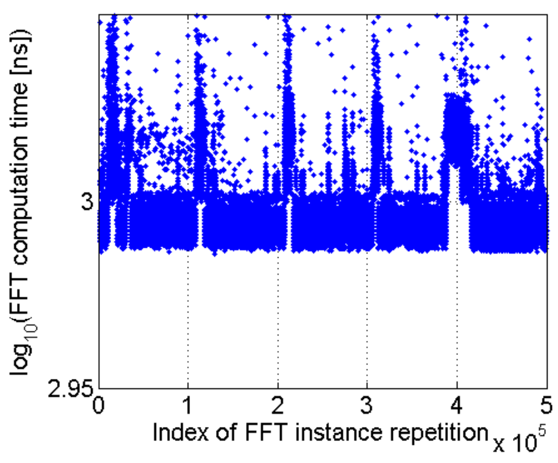
– LDLIBS+ = −L/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64
– LDLIBS+ = −lmkl_intel_ilp64
– LDLIBS+ = −lmkl_core
– LDLIBS+ = −lmkl_sequential

Then, it is important to export the MKL library path before compiling, as follows : export LD_LIBRARY_PATH = $LD_LIBRARY_PATH :/opt/intel/composer_xe_2013_sp1.0.080/mkl/lib/intel64.

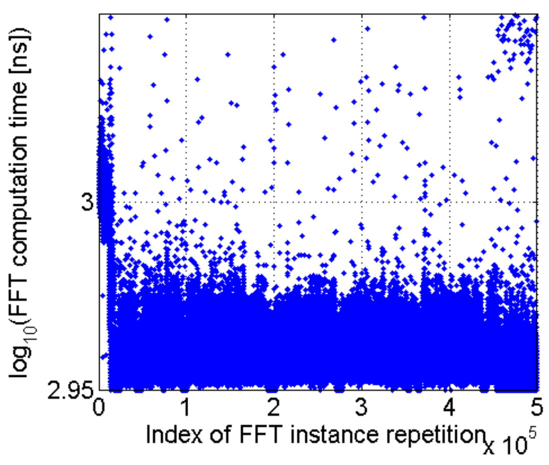
In [41] and [4], the authors illustrate an implementation of FFT using MKL. This is also presented in Algorithm 3.1 in Chapter 3.

6.3 Experimental Results

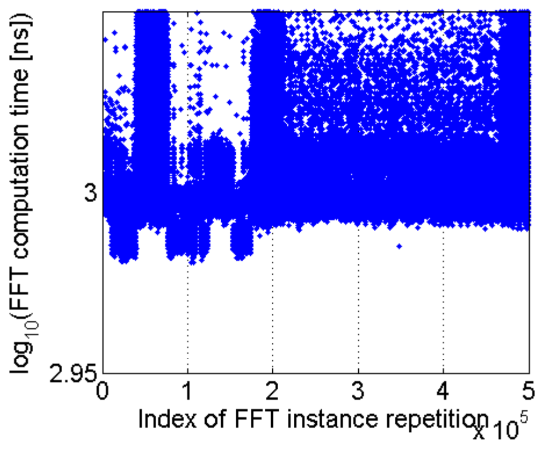
In order to estimate by how much the variability of processing time can be reduced, we ran FFT computations using MKL under DPDK performed on input vectors of complex



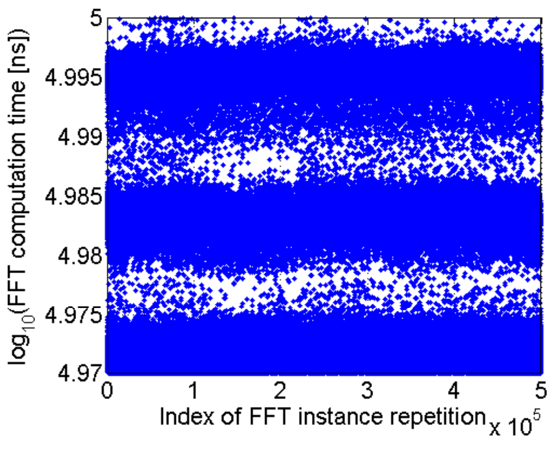
(a) Slave core without core isolation.



(b) Slave core with core isolation.



(c) MKL without DPDK.



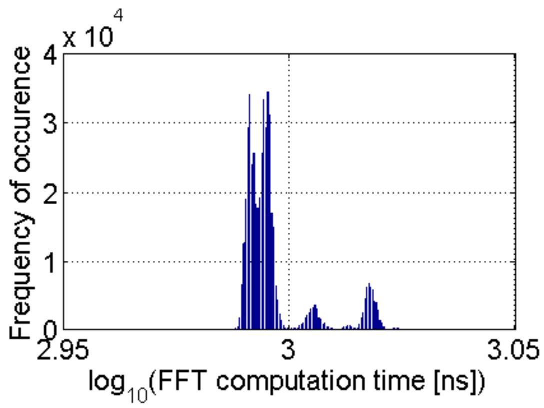
(d) Straight C implementation.

Figure 6.2 Computation times of FFT when running on slave core (a) MKL with DPDK without core isolation (b) MKL with DPDK and core isolation (c) MKL without DPDK (d) straight C implementation.

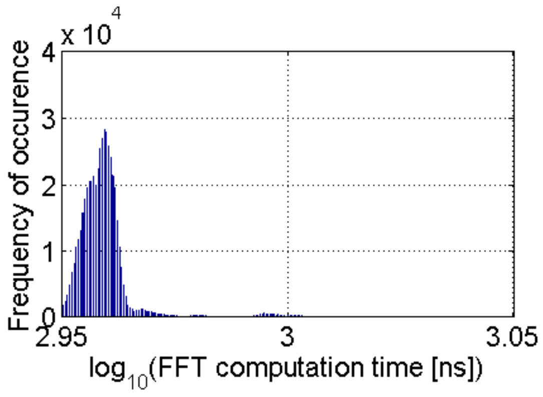
(float) numbers of size 1024. We changed DPDK parameters in order to see if and by how much they could help to reduce this variability.

In each experiment, 500000 FFTs are performed and time stamps are taken just before and after each FFT to compute time. In the first experiment, the computations are done on a DPDK slave core with MKL, but without isolating it from the OS. The second experiment is also run on a slave core but with isolation from the OS. The third experiment consists of running the computations with MKL without DPDK. Finally, as a basis for comparison, a straight C implementation of the FFT is also performed 500000 times.

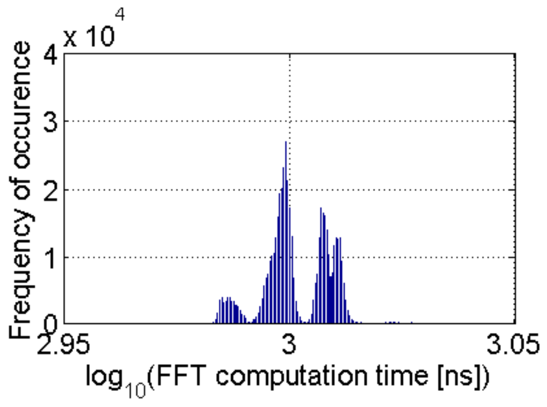
In each experiment, the first iteration takes much more time due to some initializations



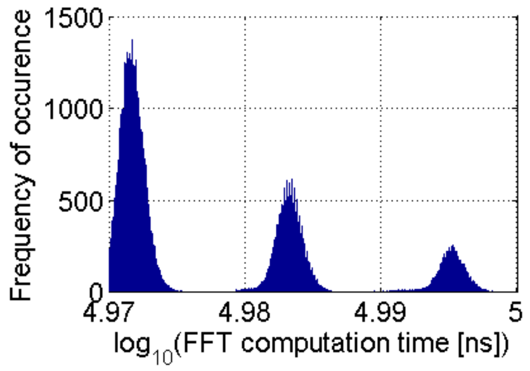
(a) Slave core without core isolation.



(b) Slave core with core isolation.



(c) MKL without DPDK.



(d) Straight C implementation.

Figure 6.3 Histograms of computation times running on slave core (a) MKL with DPDK without core isolation (b) MKL with DPDK and core isolation (c) MKL without DPDK (d) straight C implementation.

and this result is not included as it could be hidden in the real application and would thus be insignificant. That very large first run time would also mask the latency jitter that we wish to characterize.

Fig. 6.2 shows the FFT computation run time for each of the four experiments. In this case, X axis shows the FFT computation time. The FFT computation time in these figures is reported as the logarithm (base 10) of the FFT computation time in nano seconds. Thus, 3 in the Y-axis means 1000 nano secondes. As Fig. 6.2(a) depicts, in case of using slave core without core isolation, most FFT computation times are about 3 (1μ seconds) or a little bit less. They are between 2.98 and 3. Since Y axis has a logarithmic scale, it means they are between 955 and 1000 nano seconds. Fig. 6.2(b) is related to FFT computation run time in slave core with core isolation and most FFT computation times are between 2.95 and 2.975 (between 891 and 944 nano seconds). As a result, using core isolation in slave core is faster

Table 6.2 Statistics of the processing time observed in the four scenarios.

Scenario	Mean (ns)	Std Deviation
Slave Core	1.005×10^3	2.99×10^2
Isolated Slave Core	9.176×10^2	2.6×10^2
MKL without DPDK	1.019×10^3	2.62×10^2
Straight FFT	9.61×10^4	5.36×10^3

than using slave core without core isolation. Since the computation time in core isolation case is lower, this case has higher performance. Fig. 6.2(c) presents FFT computation in case of applying MKL without using DPDK. In this case, most FFT computation times are between 2.99 and 3.1 (between 977 and 1259 nano seconds). Therefore, it is not as fast as using slave core with core isolation. But it is more efficient compared to the first case. Fig. 6.2(d) illustrates FFT computation run time using straight C implementation of the FFT. Based on this figure, FFT computation time is between 4.97 and 5 (between 93325 nano seconds and 100 micro seconds) which means it is the slowest case.

Fig. 6.3 shows the histograms of the same experiments which is another way to analyze the same results. Fig. 6.3(a) depicts histogram of slave core without core isolation. Similar to Fig. 6.2(a) most FFT computation times are between 2.98 and 3 (between 955 and 1000 nano seconds). Fig. 6.3(b) is related to histogram of FFT computation run time in slave core with core isolation and like Fig. 6.2(b), most FFT computation times are between 2.95 and 2.975 (between 891 and 944 nano seconds) near to Y axis. Again, as a result shows, using core isolation in slave core is faster and more efficient than using slave core without core isolation. Fig. 6.3(c) presents histogram of FFT computation in case of applying MKL without using DPDK. In this case, most FFT computation times are between 2.99 and 3.1 (between 977 and 1259 nano seconds). It confirms the obtained conclusion from Fig. 6.2(c). Fig. 6.3(d) illustrates histogram of FFT computation run time using straight C implementation of the FFT. In this case FFT computation time is scattered between 4.97 and 5 (between 93.3 micro seconds and 100 micro seconds) which means it is the slowest case. It looks like to have the first, the second and the third harmonics. On the other hands, in this case, there are more bins and less histogram compared to the other cases.

Table 6.2 illustrates statistics of the processing time observed in the four scenarios. It depicts that after performing 500000 times FFT, the minimum mean of FFT computation is belong to slave core with isolation of CPU. While the maximum mean is related to straight FFT which is about 100 times more. The standard deviation for the isolated slave core has the smallest value. While the standard deviation for straight FFT is 20 times more.

Our results show that using MKL in DPDK, especially when isolated from the OS and other tasks, is the best solution to make the computation time as deterministic as possible. It eliminates the second and the third modes in the computation time distribution histograms that are apparent in Figs. 6.3(a), 6.3(c) and 6.3(d). Eliminating these two modes results in a histogram with only one major mode (shown in Fig. 6.3(b)), meaning less randomness in the computation times. Using DPDK along with MKL allows us to achieve lower latency with small variations (near-deterministic computation time). Small variations in the computation times is a key enabling feature to obtain scalable computational capacity for applications with real time constraints. This is an important aspect in the design of large scale cloud-based applications, since big variations make it difficult to predict the system behavior. For instance, signal processing for applications such as LTE baseband processing requires an implementation with near-deterministic computation times in order to be scalable and obey the guaranteed quality of service for the subscribers.

6.4 Summary on Computing FFT using DPDK and MKL

In this chapter, we discussed system features inducing non-determinism and some solutions. We described the parameters which cause delay and degrade performance of computation and communication processes. We conclude that it is not possible to overcome all those factors which cause delay and latency by conventional approach. While packages like DPDK and MKL were developed by Intel to control hardware and OS architecture and increase performance.

We combined DPDK and MKL to enable the abilities of Math computations in DPDK. We computed FFT on slave core as a benchmark in four cases of 1) MKL with DPDK without core isolation, 2) MKL with DPDK and core isolation, 3) MKL without DPDK and 4) straight C implementation of FFT.

Results show that DPDK can be developed as a part of the kernel for higher performance which needs kernel development. This shows that DPDK can help in increasing processing performance and reducing variability. On one hand, we have all sources of variability. On the other hand, when we use DPDK and MKL, we have much less variability compared to straight FFT. This means that DPDK effectively mitigates those sources of variability. So, maybe many of those solutions are implemented by DPDK. It was also observed that the mean computation time for the straight FFT is about 100 times longer while the standard deviation is about 20 times higher compared to the isolated slave core. Indeed, it allows us to achieve near-deterministic computation time and lower latency with small variations.

Although MKL offers excellent performance in terms of computation times, it is not scalable and therefore not suitable for large scale cloud-based applications. By contrast, DPDK allows us to improve the performance significantly while making MKL adaptable in a scalable way. Further, DPDK supports threads and memory management, which give us more flexibility in the design of large scale parallel computing architectures that can support cloud-based applications.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this thesis, we studied the computational complexity of LTE processes and identified FFT calculation as a major source of latency for LTE processing. We have proposed different means to exploit parallelism for reducing the calculation times of FFTs. In this chapter, we provide a summary of this thesis and discuss some of the main conclusions of our work.

7.1 Summary, Contributions and Lessons Learned

In Chapter 2, we discussed the main challenges in the calculations needed to implement the LTE standard. Specifically, in this thesis, we focused on the latency as one of the critical parameters for real time implementation of LTE. We explored the use of GPUs for parallel computation of LTE processes and reduction of its computation times. Our analysis showed that FFT/IFFT is the main component in OFDM and matrix inversion is the most time consuming task in MIMO detection. Further, since convolution in time has the same role as a multiplication in frequency in the LTE process, it needs to have less computation time as well. It was found that in spite of large number of elements on GPU, we could never get large acceleration in computation. Another technology of interest is DPDK, proposed to help implementing on data centers the data plane of complex applications subject to real time constraints. All existing literature found on DPDK in relation with wireless communication is about packet forwarding in layer two and three (L2 and L3). There is nothing related to computation and mathematical functions.

In Chapter 3, we presented a review of the LTE standard, GPU technology and DPDK interface. In Chapter 4, we described different algorithms for FFT, matrix inversion, convolution and cross-correlation operations and discussed their computational complexity.

In Chapter 5, we investigated the possibility of using GPUs for calculation of FFT, matrix multiplication, matrix inversion and convolution all needed by the LTE standard. Explicitly, we compared computation times of matrix multiplication (using CUDA) when shared memory, global memory and multi-core architectures are used. Our experimental results showed that matrix multiplication takes less time when we use a GPU with shared memory, compared to the other two cases (global memory and multi-core). Reusing of data in the shared memory architecture allows us to have an efficient computation process which reduces the computation time. Further, our experimental results shows that parallel computing achieves a significant improvement (decrease) on the computation times when dealing with large-size matrices and vectors. Specifically, GPUs are well suited for matrix multiplication as its computational time has a smaller order of complexity than matrix multiplication performed without using GPUs[1]. It is also observed that using GPU prevents the saturation of memory bandwidth for calculation of large scale matrices.

The implementation of FFT is also discussed in Chapter 5. Specifically, we have used CuFFT for implementation of FFT on NVIDIA GPUs expressed with the CUDA programming paradigm. Our experimental results showed that CuFFT is an appropriate solution (library) for parallel implementation of FFT[2]. Further, we used the Gaussian elimination algorithm for calculation of matrix inversion using the CUDA programming paradigm. Finally, our experimental results showed that the computational time has a complexity dominated by a term of order $\mathcal{O}(N^3)$ for big matrices. While for small size of matrix, the processing time is dominated by the initialization time.

As another contribution, we enhanced the computation time of calculating the convolution in Chapter 5. Explicitly, we used zero padding and right shift rotation to obtain an appropriate set of instructions which executes the kernel efficiently in parallel by having no boundary check for the size of the vectors. We used shared memory to reuse the data and save some of the memory bandwidth as well. Moreover, using summation reduction instead of simple sum operation reduces the computation times and prevents the memory bandwidth from being saturated. Hence, it was concluded that GPUs are a very practical choice for implementation of convolution[3].

In Chapter 5, we also explored the possibility of using MATLAB for GPU programming by implementing FFT as a benchmark. Our experimental results show that implementation of FFT on GPUs using MATLAB is around 10 times faster than its MATLAB implementation without parallelism (without using GPUs)[4]. The implementation of matrix inversion on GPU using MATLAB can reduce the computation time only by half (for matrix sizes of more than

1. The computation time of matrix multiplication is $\mathcal{O}(N^2)$ which is reduced to 25% when GPU is used.
2. Because the computation time is much less than the FFT complexity order which is $Nlog(N)$.
3. It was also deduced that parallel reduction with sequential addressing is more efficient than tree-based reduction.
4. Specifically for array sizes of more than 100.

300×300) which is not significant. Therefore, matrix inversion function of Matlab is not very effective[5]. Finally, MATLAB is very well suited for fast implementation of matrix addition and multiplication on GPUs when the result does not need to be sent to the computer memory.

In summary, GPUs have the following characteristics :

– they have a large number of computing engines,
– their shared memory has a small capacity but one read operation can feed several computing engines,
– their external memory is very slow and is a bottleneck when trying to implement fast operations on computation engines.

The ability to feed several computation engines makes GPUs a suitable choice for many of calculations in the LTE, as discussed earlier. Specifically, GPUs are well suited for the operations where :

– the size of data is small and the (small) shared memory of GPU is useful,
– there are many simultaneous reuse of data for calculations on different computation engines,
– there are few output variables (results) to keep as they can be stored in the registers and the bandwidth of shared memory would not be saturated[6].

Another contribution of this thesis is the study of the computation times of FFT calculations on multiple central processing units (CPU) using DPDK and MKL libraries. This was carried out in Chapter 6 by considering four different scenarios and comparing their performance metrics. Those four scenarios include 1) straight C implementation of FFTs, 2) the Intel MKL implementation, 3) combining DPDK and MKL without isolation of CPUs, and 4) combining DPDK and MKL with CPU isolation. Intel DPDK is an excellent technology for supporting highly scalable execution of applications such as LTE over a multiprocessor platform. It is designed for workload consolidation and load balancing, which can provide a near-deterministic environment to compute LTE processes by isolating CPUs from the kernel scheduler.

Our experimental results in Chapter 6 show that DPDK can help to increase processing performance and to reduce variability that cause delays in computations. Further, in Chapter 6, we discussed different sources of randomness (variability) in the scheduling of processes by the operating system. Using DPDK along with MKL helps us achieve less variability in implementation of FFT compared to the case where FFT is implemented

5. In fact the size of matrix operations in LTE is smaller than 300×300.
6. It is important to note that the bandwidth of memory is a very critical parameter in GPUs.

without using them. Explicitly, our experimental results show that the mean and the standard deviation of the computation time of FFT without using DPDK, MKL and isolation are around 100 times and 20 times more than those observed when DPDK, MKL and cpu isolation are used. It was also noted that although MKL offers an improvement of the performance, it is not scalable. DPDK allows us to take advantage of MKL while it provides means for achieving scalability[7].

7.2 Future work

Efficient and scalable implementation of the LTE standard requires the consideration of several complex functions and layers in its architecture. This fact makes it challenging to provide a unique solution for all of its practical issues and needs further research work to understand and address them. In the following, we present a list of different possible directions for future work and discuss them briefly.

7.2.1 GPU work

Studying other LTE functions to find an appropriate solution for their implementation on GPU is an important step for future works. Specifically, it is important to study the performance of using CUDA for implementation of the radix 4 FFT algorithm. This will provide a better and comprehensive overview of different choices for implementation of FFT as one of the biggest computational bottlenecks in the LTE.

7.2.2 Exploring other capabilities of DPDK

As it was discussed earlier, DPDK offers different benefits for implementation of LTE operations. However, full exploration of its advantages can only be done by studying all of its capabilities. For instance, it is essential to study the effect of using DPDK buffers on top of memory and thread management. This has potentially significant benefits especially for implementation of the turbo decoding algorithm. Multi-threading and software level isolation of processing units are other capabilities that should be investigated in the future. Moreover, further work remains to use multi-core processing environment in parallel to prove the deterministic processing time.

7. DPDK also provides support for the management of threads and memory.

Using other processing devices, including XEON-Phi processors, for DPDK-based implementation of LTE processes should also be studied. Specifically, XEON-Phi processors may offer a useful balance between the bandwidth of memory and processing power. In general, one has to look for the best choices of algorithm, software implementation and hardware for efficient and scalable implementation of a specific function in the LTE standard.

REFERENCES

[1] Erik Dahlman, Stefan Parkvall, and Johan Skold. *4G : LTE/LTE-Advanced for Mobile Broadband : LTE/LTE-Advanced for Mobile Broadband*. Academic Press, 2011.

[2] Gergely Pongrácz, Laszlo Molnar, and Zoltán Lajos Kis. Removing Roadblocks from SDN : OpenFlow Software Switch Performance on Intel DPDK. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 62–67. IEEE, 2013.

[3] Dominik Scholz. A look at Intel's Dataplane Development Kit. *Network*, 115, 2014.

[4] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[5] Arogyaswami J Paulraj, Dhananjay A Gore, Rohit U Nabar, and Helmut Bolcskei. An overview of MIMO communications-a key to gigabit wireless. *IEEE Proceedings*, 92(2) :198–218, 2004.

[6] Martin Palkovic, Praveen Raghavan, Min Li, Antoine Dejonghe, Liesbet Van der Perre, and Francky Catthoor. Future software-defined radio platforms and mapping flows. *IEEE Signal Processing Magazine*, 27(2) :22–33, 2010.

[7] June Kim, Seungheon Hyeon, and Seungwon Choi. Implementation of an SDR system using graphics processing unit. *IEEE Communications Magazine*, 48(3) :156–162, 2010.

[8] Michael Wu, Yang Sun, Siddharth Gupta, and Joseph R Cavallaro. Implementation of a high throughput soft MIMO detector on GPU. *Journal of Signal Processing Systems*, 64(1) :123–136, 2011.

[9] Teemu Nylanden, Janne Janhunen, Olli Silvén, and Markku Juntti. A GPU implementation for two MIMO-OFDM detectors. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 293–300. IEEE, 2010.

[10] Gabriel Falcão, Vitor Silva, and Leonel Sousa. How GPUs can outperform ASICs for fast LDPC decoding. In *Proceedings of the 23rd international conference on Supercomputing*, pages 390–399. ACM, 2009.

[11] Sandra Roger, Carla Ramiro, Alberto Gonzalez, Vicenc Almenar, and Antonio M Vidal. Fully parallel GPU implementation of a fixed-complexity soft-output MIMO detector. *Vehicular Technology, IEEE Transactions on*, 61(8) :3796–3800, 2012.

[12] Michael Repplinger and Philipp Slusallek. Stream processing on GPUs using distributed multimedia middleware. *Concurrency and Computation : Practice and Experience*, 23(7) :669–680, 2011.

[13] Michael Wu, Bei Yin, and Joseph R Cavallaro. Flexible N-way MIMO detector on GPU. In *2012 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 318–323. IEEE, 2012.

[14] Chiyoung Ahn, Saehee Bang, Hyohan Kim, Seunghak Lee, June Kim, Seungwon Choi, and John Glossner. Implementation of an SDR system using an MPI-based GPU cluster for WiMAX and LTE. *Analog Integrated Circuits and Signal Processing*, 73(2) :569–582, 2012.

[15] Dan Sui, Yunzhou Li, Jing Wang, Peng Wang, and Bin Zhou. High throughput MIMO-OFDM detection with graphics processing units. In *Computer Science and Automation Engineering (CSAE), 2012 IEEE International Conference on*, volume 2, pages 176–179. IEEE, 2012.

[16] Axel Klatt and Peter Stevens. Method for considering the subscriber type in mobility and radio resource management decisions in a radio access network, January 4 2008. US Patent App. 12/522,465.

[17] Moustafa M Nasralla, Ognen Ognenoski, and Maria G Martini. Bandwidth scalability and efficient 2d and 3d video transmission over lte networks. In *Communications Workshops (ICC), 2013 IEEE International Conference on*, pages 617–621. IEEE, 2013.

[18] Alatishe S Adeyemi and Dike U Ike. A review of load balancing techniques in 3gpp lte system. *Int. J. Comput. Sci. Eng*, 2(4) :112–116, 2013.

[19] Erik Dahlman, Stefan Parkvall, and Johan Skold. *4G : LTE/LTE-advanced for mobile broadband*. Academic Press, 2013.

[20] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M Voelker. SORA high-performance software radio using general-purpose multi-core processors. *Communications of the ACM*, 54(1) :99–107, 2011.

[21] China Mobile. C-RAN, the road towards Green RAN. October 2011.

[22] Ichiro FUKUDA and Tomonori FUJITA. Deployment of OpenFlow/SDN technologies to carrier services. *IEICE Transactions on Communications*, 96(12) :2946–2952, 2013.

[23] Amitabha Ghosh and Rapeepat Ratasuk. *Essentials of LTE and LTE-A*. Cambridge University Press, 2011.

[24] LTE Tutorial. http ://lte/lte_protocol_stack_layers.htm. 2014.

[25] Michael Wu, Yang Sun, and Joseph R Cavallaro. Implementation of a 3gpp lte turbo decoder accelerator on gpu. In *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, pages 192–197. IEEE, 2010.

[26] NVIDIA. http ://docs.nvidia.com/cuda/. 2014.

[27] Stefan Zerbst and Oliver Düvel. *3D game engine programming.* Thomson Course Technology, 2004.

[28] Intel. Intel Math Kernel Library Cookbook. August 2014.

[29] Intel. Intel dual-channel ddr memory architecture white paper. 2015.

[30] Intel. http ://www.intel.com/go/dpdk. 2014.

[31] Intel DPDK. Getting started guide. 2014.

[32] Intel DPDK Sample. Dpdk sample aplication user guide. 2014.

[33] Intel DPDK. Programmers guide. 2014.

[34] Michael Corinthios. *Signals, systems, transforms, and digital signal processing with MATLAB.* CRC Press, 2009.

[35] MICHAEL J Corinthios. The design of a class of fast fourier transform computers. *IEEE Transactions on Computers*, 20(6) :617–623, 1971.

[36] LD Brandon, Chas Boyd, and Naga Govindaraju. Fast computation of general fourier transforms on gpus. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 5–8. IEEE, 2008.

[37] Charles Van Loan. *Computational frameworks for the fast Fourier transform*, volume 10. Siam, 1992.

[38] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture : a hardware/software approach.* Gulf Professional Publishing, 1999.

[39] NVIDIA. http ://math/GPU/documents/CUFFT_Library_3.0.pdf. August 2014.

[40] Jason Sanders and Edward Kandrot. *CUDA by example : an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[41] Ramses van Zon. http ://wiki.MKLTechTalkMarch2012.pdf. August 2014.