Parallel and Distributed Programming for Data-Computation Intensive Applications

Bilal Jan

Tutore
Dr. Bartolomeo Montrucchio (DAUIN)
Prof. Carlo Stefano Ragusa (DENERG)

Coordinatore del corso di dottorato
Prof. Matteo Sonza Reorda

February 2015
Dedicated to my beloved parents and family.
Acknowledgements

It is a great pleasure to offer my gratitude to the people who helped me in What I am now.

I am very thankful to my supervisors Dr. Bartolomeo Montrucchio and Prof. Carlo Stefano Ragusa for allowing me to work under their supervision and their consistent support throughout my PhD study. Their mesmerizing guidance has always been a source for my scientific and professional growth. Dr. Bartolomeo introduced me to parallel programming on GPUs and his discussions were very productive. Prof. Ragusa was always inspirational in elaborating intricate details of Micromagnetics. I am thankful for the scientific ideas and helpful suggestions from my research group members Dr. Omar Khan, Dr. Fiaz Khan and Dr. Arbab Rahim.

I am grateful to the administration of DAUIN and DENERG departments for their support in giving me necessary equipments and place to carry out this research work.

Lastly, I am very thankful to the Higher Education Commission (HEC) Pakistan for financing my studies abroad.
Summary

Scientific computing requires high computation power where large volumes of data are processed quickly usually in gigaFLOPS and teraFLOPS. Supercomputers, grid or cluster based systems are always the preferred choice for running such massively parallel scientific computing jobs. The Graphic Processing Units, GPUs, that are used in todays supercomputers (e.g Titan with 10 petaFLOPS performance) is one of the scientist dream came true for running complex simulations at enormous speed and at reduced cost. Due to its high performance and low cost (price-performance ratio) GPUs are the preferred choice in High Performance Computing.

The GPUs though originally were designed for rendering graphics in high resolution games, are now a days extensively used for computation extensive general purpose applications by the name GPGPU (General Purpose Graphic Processing Unit). A GPU which in simplest terms is an assembly of hundreds or even thousands of processors (e.g Nvidia Kepler K20 having 2496 cores) is attached to the computer mother board via PCIe or shared by a network and works in junction to the CPU. The main vendors are the NVIDIA, AMD and Intel. Various programming tools and APIs have been developed for GPU computing with greater attention received by CUDA, OpenCL and OpenGL. Compute Unified Device Architecture, CUDA, is proprietary of NVIDIA and although very rich in functionality and various programming language support (C, C++, C#, Fortran) but runs only on NVIDIA hardware with no cross platform support for other GPU vendor types. On the other hand Open computing language , OpenCL, targets heterogeneous platforms where different types of compute devices interact. OpenCL provides cross-platform portability of code where code written for an application can be run on any vendor type GPU (AMD, NVIDIA, Intel) and also on other processor types like CPU, FPGA and even mobile devices for the purpose to support cross-platform code portability. This thesis work is aimed at parallel programming of computation intensive tasks
such as sorting large data sets, design and implementation of parallel Fast Fourier transform library and the FFT based fast magnetostatic field computation in the area of Micromagnetics.

Sorting algorithms arrange a given sequence of input data into a certain order (monotonic increase or decrease) and are categorized by their computational complexity for best, average and worst case analysis. The time complexity is not the only deciding parameter, since several sorting techniques are of the order $O(n \log n)$, but other factors like stability, robustness, scalability, input distribution, memory storage and access patterns decide the applicability of a sorting algorithm for a certain application domain. The portion of the thesis work is devoted to the design and implementation of new parallel sorting techniques well suited for multi-processor architectures like GPUs and other multi-core systems. The novel sorting techniques, min-max Butterfly and Full Butterfly Network Sort exploit high parallelism in its design and thus achieve considerable speedup against state-of-the-art sorting techniques.

To dwell much further in the discussion of data parallelism and programming the GPUs, a more challenging problem of implementing a General-purpose Fast Fourier Transforms (FFT) library (named ToPeFFT) is addressed. ToPe-FFT is based on the well-known Cooley-Tukey algorithm with auto-tuning for multiple GPUs. The open source ToPeFFT implements several base radices (radices 2-10) along side the support for mixed-radices makes it an almost arbitrary length FFT library. The library takes Complex-to-Complex (C2C) input type with dimension sizes up-to 3D. The design and interface of ToPeFFT is similar to cuFFT (Nvidia GPU Library) and FFTW. The supported features of arbitrary input length, better accuracy in high dimension transforms, load balancing on multiple GPUs and above all significant speedup against cuFFT and FFTW makes ToPeFFT promising in delivering maximum performance. An optimized version is tested in Micromagnetic simulations for performance improvement as discussed below.

In Micromagnetic simulations the computation of magnetostatic field is the most time consuming part of the overall simulation time. In the case of a ferromagnetic region discretized into $n$ number of elementary cells, the computation of magnetostatic field $h_m$ at a particular location has a functional relationship with the magnetization $M$ at all other elements in the whole region. This long range elementary dipole interactions increase complexity since for the computation of $h_m$ at a particular cell,
magnetization $\mathbf{M}$ for all other cells need to be considered. This has a complexity of $O(n^2)$ for computing $\mathbf{h}_m$ across the whole region. In the FFT based magnetostatic field computation, the given model is treated as discrete convolution problem with a reduced complexity of $O(n \log n)$. We have used an optimized version of our ToPeFFT for accelerating magnetostatic field computation. Our GPU based optimized field solver has significant speedup against OOMMF magnetostatic field computation time.

In Micromagnetics to observe the magnetization dynamics, the Landau-Lifshitz equation is numerically integrated using semi-discretization techniques (space-time discretization). Among the standard time-stepping techniques, the mid-point rule is widely adopted as it is unconditionally stable and preserves magnetization dynamics independent of the time steps. However the mid-point rule is computation intensive requiring a large system of non-linear equations to be solved. A new Runge-Kutta like scheme is proposed with fewer computations in terms of linear equations, which preserves the magnetization dynamics at each time-space interval. The proposed scheme is tested against $\mu$-mag Standard Problem No 4 and accurate results are recorded for time-steps up to 1.25 $\text{ps}$. 

VI
# Table of contents

Acknowledgements .......................................................... III

Summary .............................................................................. IV

1 Motivation and Thesis Structure ........................................ 1
   1.1 Contribution ................................................................ 1
   1.2 Thesis Structure ...................................................... 2
   1.3 Publication List ....................................................... 3

2 Graphic Processing Units for Parallel Computing .................. 4
   2.1 The Graphic Processing Unit: GPU ............................... 4
   2.2 Parallel Computing and the GPUs ............................... 5
   2.3 GPGPU Computing .................................................... 7
   2.4 The Open Computing Language: OpenCL ..................... 10
      2.4.1 OpenCL Platform Model ....................................... 10
      2.4.2 OpenCL Execution Model ..................................... 10
      2.4.3 GPU Memory Hierarchy ......................................... 11
      2.4.4 Mapping Execution to Platform Model ..................... 13
   2.5 GPU Memory Bandwidth ............................................ 16
      2.5.1 Theoretical vs Effective GPU Memory Bandwidth .......... 16
      2.5.2 Spatial and Temporal Memory Locality ...................... 16
      2.5.3 Memory Access Patterns ....................................... 17
   2.6 PCIe Bottleneck Effect on Performance .......................... 19
      2.6.1 Performance Formula ........................................... 19
3 Sorting Large Datasets on Graphic Processing Units 21
  3.1 Sorting on GPUs ............................................. 21
  3.2 Butterfly Network Sorting .................................. 24
    3.2.1 Min-Max Butterfly ...................................... 24
  3.3 Results and Comparisons .................................... 31
    3.3.1 Architecture Details .................................... 31
    3.3.2 Experimental Setup ..................................... 33
    3.3.3 Results .................................................. 33
  3.4 Conclusion .................................................. 34

4 GPU Implementation of Fast Fourier Transforms: ToPe-FFT 36
  4.1 Discrete Fourier Transform: DFT .............................. 36
  4.2 The Fast Fourier Transform: Fast methods for DFT computation ... 39
    4.2.1 Cooley-Tukey FFT ...................................... 39
    4.2.2 Higher Radices for CT FFT .............................. 46
    4.2.3 Mixed Radix FFT ....................................... 46
  4.3 GPU FFT Implementation: ToPe FFT Library .................. 50
    4.3.1 ToPe-FFT Design ....................................... 51
    4.3.2 Cooley-Tukey Radices as OpenCL Kernels ............... 53
    4.3.3 Load balancing on multiple GPUs ....................... 58
    4.3.4 Usage and Interfacing .................................. 58
  4.4 1D FFT Results and Comparisons .............................. 58
    4.4.1 SpeedUp against FFTW and cuFFT ....................... 61
  4.5 Higher Dimension FFTs ..................................... 62

5 Micromagnetics: GPU Acceleration of Magnetostatic Field Com-
  putation .......................... 68
  5.1 Introduction ................................................ 68
  5.2 Magnetostatic Field Computation Model ...................... 70
  5.3 Magnetostatic Field Solver on GPU ........................... 74
  5.4 Results and Comparisons .................................... 75
    5.4.1 Experimental Setup ..................................... 75
    5.4.2 Validation and Performance ............................. 77
List of figures

2.1 Integrated and Discrete GPUs ........................................... 5
2.2 Parallel Computing paradigm ........................................... 8
2.3 SIMD Architecture ..................................................... 9
2.4 Heterogeneous Computing CPU-GPU Interaction .................... 11
2.5 Data Flow between Host and Device .................................. 12
2.6 2-dimensional NDRange index spacing. ............................... 15
2.7 Array Copy on CPU and GPU. ........................................... 18
2.8 Rnnign time of array copy on CPU and GPU. ........................ 18
3.1 A 2x2 Butterfly comparator diagram ................................. 25
3.2 Min-Max butterfly for input of size 8 ............................... 26
3.3 Full Butterfly sorting for input size 8 ............................... 29
3.4 Profiler Analysis ....................................................... 29
3.5 Thread Divergence ....................................................... 31
3.6 Global Memory Loads and Stores ..................................... 32
3.7 Sorting Time and Rate Comparison I .................................. 34
3.8 Sorting Time and Rate Comparison II ................................. 34
3.9 Speedup Results ......................................................... 35
4.1 Radix-2 Cooley-Tukey Butterfly ...................................... 42
4.2 Bit reversal and CT splitting .......................................... 43
4.3 DIT Radix-2 Cooley-Tukey strategy of splitting into sub-DFTs ... 43
4.4 An exampler radix-2 DIT CT for N=8 ................................ 47
4.5 CT radix-4 butterfly diagram .......................................... 48
4.6 Mixed-radix formulation of radix 9 using 3 × 3 factorization. ... 50
4.7 An example ToPeFFT framework ...................................... 52
4.8 ToPe-FFT Plan and Exec .............................................. 54
4.9 Data Type Comparison .................................................. 61
4.10 Results radices 5 and 6 ........................................ 62
4.11 Results radices 7 and 8 ........................................ 62
4.12 Results radix 10 and multi GPUs ............................. 63
4.13 Speedup against FFTW and cuFFT (I) ....................... 63
4.14 Speedup against FFTW and cuFFT (II) ..................... 64
4.15 Speedup against FFTW and cuFFT (III) ................. 64
4.16 Speedup against FFTW and cuFFT (IV) .................. 65
4.17 2D FFT by the Row-and-Column Approach .............. 66
4.18 2D FFT Running times ....................................... 66
4.19 2D FFT Speedup ............................................. 67
5.1 Demagnetizing Tensor representation ........................ 71
5.2 Zero padding of the given magnetized body across each dimension .... 71
5.3 Symmetry in $\mathbf{m}$ due to zero-padding technique ........ 73
5.4 GPU Magnetostatic Field Solver ............................... 76
5.5 Initial magnetization ......................................... 77
5.6 Computation time of magnetostatic field .................... 78
5.7 Computation time of magnetostatic field .................... 79
5.8 Memory Transfer to/from GPU ............................... 80
5.9 Speedup against OOMMF ..................................... 81
6.1 LLG damping and precession ................................ 87
6.2 Comparison with OOMMF .................................... 91
6.3 Plot showing conservation of magnetization ............... 92
6.4 Plot showing constant free energy of the system .......... 93
6.5 Relative error ................................................. 94
6.6 Plot of averaged relative error with damping ............. 95
Chapter 1

Motivation and Thesis Structure

1.1 Contribution

General Purpose Graphic Processing (GP-GPU) is where the GPU multi-processor architecture is used for boosting general purpose scientific applications. The introduction of high level programming tools and APIs like CUDA and OpenCL and others made it possible that GPUs could be used for general purpose simulations beyond their sole purpose of high end-gaming only. In this thesis work GPU architectures have been tested for running different applications. New parallel sorting techniques have been designed and implemented using OpenCL on the GPU architecture. Both the parallel as well as the serial version of the new sorting techniques have significant speedup against state of the art sorting algorithms. A highly optimized OpenCL based GPU FFT library have been implemented. The library named ToPeFFT is the first OpenCL based arbitrary input length complex-to-complex FFT library with relatively better accuracy for higher dimensions 2D and 3D transform types. An optimized FFT library specialized for magnetostatic field computation in Micromagnetics is designed. It is shown that by using the discrete convolution theorem and the inherent symmetries in the given structure, the magnetostatic field is accelerated where our GPU code achieves considerable speedup against state-of-the-art magnetostatic field computation simulators like OOMMF. In the area of Micromagnetics, a novel time stepping scheme for the integration of Landau-Lifshitz equation for the study of magnetization dynamics in ferromagnetic is discussed. The new scheme is efficient and very accurate even for relatively larger
time step interval. The new scheme is high parallel in design with less complexity in terms of number of equations solved per elementary cell. The new time stepping technique is tested against $\mu$-mag standard problem no.4 with relatively.

1.2 Thesis Structure

The thesis structure is organized as follows. In chapter 2, introduction to GPU computing is discussed which uncovers the basics of GPU architectures, their placement in high performance computing (hpc) systems and the programming environments in particular the Open Computing Language (OpenCL). To support the argument of GPU outperforming CPU, some sample results are compared on both devices. To dwell much further in the discussion of data parallelism and programming the GPUs, in chapter 3 a new highly parallel sorting algorithm Butterfly Network Sort is implemented and compared with state-of-the-art sorting algorithms. Being equipped with basic theory and implementation on a GPU architecture, a more challenging problem of implementing Fast Fourier Transforms library (ToPe-FFT) is discussed in chapter 4. FFT is collectively named to the fast algorithms for carrying out Fourier transformation and is the backbone in a plethora of scientific applications. We have adopted the Cooley-Tukey FFT algorithm as our implementation of the fourier transform because the Cooley-Tukey decomposition is very parallel and has inherent symmetries. The chapter briefly discusses the library implementation in a structure very similar to NVIDIA CUDA cuFFT. We adopted the same structure for in order to be easily adaptable and for better comparison. The chapter provides details of single versus multiple GPU implementation with several radices support making the library functional almost for arbitrary input size. The chapter is wound up with results showing better performance than CUDA cuFFT and FFTW. ToPe-FFT is tested in the area of Micromagnetics for accelerating magnetostatic field ($h$) computation as briefly discussed in chapter 5. In chapter 5, some theory of Micromagnetics is given as a foundation. The different energy contributions along with the LL (Landau Lifshitz) equation of motion for investigating magnetization dynamics is highlighted. FFT based computation of magnetostatic field is discussed. Using the discrete convolution theorem complexity in computing $h$ is reduced from $O(n)$ to $O(n \log n)$. Performance comparison and validation of magnetostatic field computation by our optimized FFT library is
compared with that of OOMMF micromagnetic simulator. A novel time stepping method for the solution of LL equation is introduced in chapter 6. The new time stepping technique based on mid-point and Runge-Kutta schemes is very efficient and fast in the sense that only $3 \times 3$ sets of equations are solved per computation cell. The chapter shows some results on better performance, accuracy and efficiency of the technique when tested on $\mu$mag standard problem no.4 with OOMMF.

1.3 Publication List

This thesis work is based on the following list of publications.

Journals


Conference proceedings


Chapter 2

Graphic Processing Units for Parallel Computing

2.1 The Graphic Processing Unit: GPU

A Graphic Processing Unit (GPU) in simplest terms is an assembly of processors soldered to the device main-board directly or as separate card attached to the device via an interconnect. The GPU processor circuitry combined with memory, I/O interfaces and other specialized components (BIOS, RAMDAC, etc) compose the modern video card. The GPUs were originally aimed at fast rendering (generation of image from a model-scene file) of high resolution graphic frames in high-end games. The introduction of generic processing units to the GPU and the development of parallel programming tools have enabled GPUs for general purpose computation i.e. General Purpose Graphic Processing Unit (GPGPU). This has driven the GPU usage beyond the graphics rendering to more diverse areas like Bioinformatics [5], Weather forecasting [47], pattern recognition [42], Micromagnetics [46, 71], Database [6, 26], Economics [32] and in every such field where similar operations on data are carried out simultaneously i.e the application is suitable for massively parallel processing (MPP). As mentioned in the start, a GPU can be either soldered on to the main-board of the device or attached through an interface. The former is called integrated GPU whereas the later as discrete GPU or in some contexts dedicated GPU. Integrated GPU uses a portion of the system RAM for video processing where
as Discrete GPU comes with its own RAM (arranged in a hierarchy of memory sub-types). The selection choice however depends on the functionality one requires from the graphic processor. For small and relatively less graphic operations, devices like cell phones and laptops, integrated GPUs are normally preferred. Discrete GPUs are used in environments where GPU computing power is intended for high performance. This include high-end devices like desktop computers, super computers, DSPs, gaming PCs etc. In the case of desktop machines the connection is usually made via PCIe. This however limits memory bandwidth to and from the GPU. This topic is more discussed in later sections. The following figure 2.1 shows how GPUs are typically homed in the computer motherboard. GPU market share is mainly captured by NVIDIA, ATI/AMD, Intel and ARM Mali.

![Figure 2.1: Integrated GPU AMD 780G (left) and (right) main board with dual Discrete Nvidia GPUs.](image)

### 2.2 Parallel Computing and the GPUs

Parallel computing aims at simultaneous execution of the parts of a problem on multiple processing elements for performance gains. Thus in terms of processor execution time only, $p$ processors working simultaneously on a problem would be $p$ times faster than the single processor executing the problem. But in reality the memory access times and the interprocess communication limits this factor to be less than $p$ times the single processor’s time. A typical parallel computing problem decomposition and execution flow is represented in figure 2.2. Here the given problem is decomposed into several sub tasks $T_1, T_2$ and so on. Each sub task is
executed sequentially one after another. During the execution of a sub task say it $T_M$, at each time instant $t_l$ same instruction is carried out simultaneously on all the processors from the available processor pool. It is assumed, not shown in the figure, that each instruction at any particular time instant operates on different data stream a technique known as single instruction multiple data (SIMD) from Flynn Taxonomy. Flynn Taxonomy categorizes multi processor architecture based on number of instructions carried out on memory units (data streams) in clock cycle (or portion of a cycle). The different types are the following.

- **SISD**: Single Instruction Single Data. A single instruction is carried out on single data stream. Let the accumulative time of instruction decoding, memory fetch and execution time is one clock cycle. Thus a single instruction carried out on 100 memory units would require 100 clock cycles on a single processor.

- **SIMD**: Single Instruction Multiple Data. In the SIMD architecture single instruction is carried out on multiple streams of data simultaneously. This requires however multiple processing units (PUs). Thus a device with 10 PUs each executing one instruction per clock cycle (1 IPC) would require 10 cycles in executing the instruction on 100 data streams. In other terms this means concurrent manipulation of different data items for same instruction during a clock cycle.

- **MIMD**: Multiple Instructions Multiple Data. At any clock cycle different instructions are carried on different data streams on different processing units. This is the most desired solution for any HPC system and is widely adopted in the top500 super computers (www.top500.org). Thus in the case of a MIMD architecture a device with 10 PUs, each executing one instruction per clock cycle (IPC of 1), would require 10 clock cycles in executing at most 10 possibly different instructions on 100 streams of data. Thus in a single clock-cycle concurrency in both data streams and instructions performed on the data. In the figure 2.2 executing sub-tasks ($T_1$, $T_2$ etc) simultaneously on different pools (groups of processors where processors in each group perform similar task) of processors is a typical MIMD architecture in parallel computing environments. As highlighted in [8] most MIMD systems use SIMD processing on sub-units of the problem domain for performance reasons and due to this fact GPUs placement in HPC systems is of significant importance in terms of
cost-to-performance ratio. The GPU SIMD architecture is discussed in more detail in later sections.

- MISD: Multiple Instruction single Data. In this case multiple instructions are carried out on single data stream. A simple example case is the addition and subtraction instruction followed simultaneously on same data. Very few application areas exist for this type (MISD is usually missed!). For the case of our hypothetical example of 10 PUs, each PU would execute 10 different instructions on single data stream during a clock cycle and hence concurrency in instructions only.

In the figure 2.2, consider any single sub-task as a standalone problem, the total fetch and execute cycles shows the case of SIMD architecture for a particular problem at hand as highlighted in subsequent figure 2.3.

2.3 GPGPU Computing

As noted at the start, GPUs were originally designed for fast manipulation of graphic functionality but soon it was realized that GPU computing power could be utilized for other applications as well and thus programmable shaders with support for floating point numbers were added to GPUs. By the development of some open and proprietary solutions for GPU programming the terminology of GPGPU, General-Purpose computing on GPU, caught up many minds for the GPU to be used for highly parallel computation intensive simulations. In the GPU (generalized from GPGPU) computing environment both CPU and GPU are used in together. The sequential portions of the program are executed on CPU whereas parallel modules are run on the GPU. This poses a great challenge for the developer for a modular design of the algorithm, identify sequential and parallel modules, code and run sequential and parallel modules on CPU and GPU respectively and get considerable speedup as reward. The design principle of transporting heavy but parallel code portions from CPU to GPU side is known as off-loading the CPU. Based on this principle several programming tools, APIs and platforms have been developed in the last decade (see 1 for a brief list) with major attention received by CUDA and

1http://www.clustermonkey.net/Applications/gpu-programming-for-the-rest-of-us.html
• CUDA: Compute Unified Device Architecture short for CUDA is a platform and programming model for parallel processing on NVIDIA GPUs. CUDA provides a platform for high level languages (C, C++, Python see\(^2\) for details) to access and issue commands to the GPU hardware for executing code in parallel on the multi processors. The parallel modules are coded, known as kernel, using CUDA C-like programming language. CUDA provides a rich API with other supporting libraries (CUBLAS, CUFFT, MAGMA etc) and

\(^2\)http://www.nvidia.com/object/cuda_home_new.html
language extensions but no code portability and hence a CUDA program will run only on NVIDIA GPUs.

- OpenCL: Open Computing Language is a royalty free open standard by the Khronos Group. OpenCL provides a framework for parallel programming on multiprocessor architectures ranging from handled devices like cell phones to more sophisticated devices (GPUs, CPUs, FPGAs, DSP processors etc). At the heart of the OpenCL framework are two things.

  - A cross platform C-like language based on C99 (ISO/IEC 9899:1999)
  - An API which coordinates parallel computation on heterogeneous environments (control, access and execution on different processor types).

Portability of code on cross platforms has made OpenCL a viable option for parallel programming in the heterogeneous systems. In the following OpenCL constructs for parallel programming is discussed in more detail. In this case CPU and GPU heterogeneous system is considered where the OpenCL language (based on OpenCL 1.2 specification) is used to write parallel code intended to run on GPU multi-processors invoked from CPU by the OpenCL API function calls. These abstract statements are explained in the following.

---

3 https://www.khronos.org/opencl/
2.4 The Open Computing Language: OpenCL

Based on the underlying device architecture a vendor implementation (as a separate software development kit-sdk or embedded inside the device driver) of the OpenCL uses different naming conventions for the OpenCL constructs. An OpenCL vendor implementation (AMD, Intel, NVIDIA etc) must conform to the specification [52]. Here we will stick to our terminology to the specialization and mention the vendor specific naming conventions where ever necessary. In the heterogeneous environment of CPU-GPU, the CPU accesses the GPU via the OpenCL API through a piece of code called host code. The parallel code intended to be run on the GPU is coded in OpenCL-C and is called kernel code. The OpenCL program starts from the host where using the 5 basic data structures provided by the OpenCL API, program control and execution flow is switched between host and device respectively. A typical program control and execution flow is shown in the following.

2.4.1 OpenCL Platform Model

The OpenCL platform model is shown in figure 2.4. An OpenCL enabled device (GPU, CPU, DSP Processor etc) sees the hardware architecture as a set of one or more compute units. Each compute unit consists of one or more scalar processors called processing elements (PE). A GPU device contains more than one compute units and several PEs inside a single compute unit. The compute units frame the SIMD execution flow where at particular time instant single instruction is carried out simultaneously on multiple PEs on streams of data.

2.4.2 OpenCL Execution Model

It is where the actual parallel programming and its execution takes place. The execution model defines set of OpenCL programming constructs and rules to access the device hardware, code the kernels, feed the kernels with proper memory objects, allocate and group execution units (work-items discussed in the following), run concurrents execution of the kernel instances on device (or devices if any) and finally fetch the results from device memory to host memory. The execution model provides the 6 basic data structures (see [61] for details) for OpenCL to get job done. These are discussed in their order of occurrence in an OpenCL program.
2.4.3 GPU Memory Hierarchy

A GPU device contains a hierarchy of memory spaces as outlined in the following.

- **Global Memory**: A large chunk of memory accessed by all compute units. This size is typically in GBs. The kernel code access this data by the variables declared with `__global` qualifier. Global Memory has highest latency than other GPU memory types and typically memory access takes around 400-800 cycles.

- **Constant Memory**: The constant memory has the same scope as of global memory but remains constant during the kernel execution. The constant memory is allocated from the global memory region and accessed by variables declared with `__constant` qualifier. The access latency is same as global memory, but some OpenCL implementations provide constant memory to be limited but cached always.
• Local Memory: Local memory is small memory per compute unit. This memory region is relatively fast compared to global/constant memories and is shared by all the kernel instants (called work-items) executing on the particular compute unit. The local memory type is accessed via \_\_local keyword. In NVIDIA terminology local memory is named as shared memory. Local memory is faster than global memory but limited in size and typically in KBs.

• Private Memory: Private Memory also called Register space has scope limited only to the PE i.e. the work item executing the kernel instance and is the fastest of all memory types. The register space per PE is limited and data is poured from global-constant or local memory into registers before operations on data are performed, as shown in the figure 2.5 highlighting data flow between CPU and GPU.

![Data Flow between Host and Device](image)

Figure 2.5: Data Flow between Host and Device

2.4.3.1 Basic OpenCL Data Structures

• Platform: The OpenCL platform, accessed via a call to clGetPlatformIDs, implements features enabling the host to access OpenCL device list, device(s) information and maintain context. OpenCL provides the ability ot work around with multiple platforms in systems having more than one OpenCL implementations (e.g NVIDIA GPU with AMD CPU and user want to leverage parallelism on both devices) but in most cases is a single OpenCL implementation, so one platform per host side.
• Device: The OpenCL device list (in cases more than one devices) is accessed by a call to clGetDeviceIDs which fetches all the available devices in the platform. The device where the kernel code will be executed eventually is defined by in the call to clGetDeviceIDs. The device capabilities can be viewed by a call to API function clGetDeviceInfo.

• Context: The context data structure (accessed by clCreateContext API call) provides a way for the host to interact with the device. The context acts as a gateway between the host and device for their interaction. The context associates the command queue(s) to a device (or devices if multiple queues) for different commands and manages memory-program and kernel objects.

• Command Queue: A command queue acts as a bridge between the host and device where different actions (program execution commands etc) and memory transfers take place. A Command-queue is created per device and has the job of enqueuing kernels for execution, host-to-device and device-to-host memory transfers. The enqueued commands can be executed both in-order and out-of-order by the command-queue and is set during the command-queue creation (clCreateCommandQueue).

• Program: The program is points to the kernel code (with dot.cl extension) or its binary. In the former case the program object (of type clProgram) is created with clCreateProgramWithSource API call or clCreateProgramWithBinary for the later case respectively. The kernel code is the parallel part written in the OpenCL CL language extension and intended to be run the device i.e. on multi-processor GPU architecture. One program object is created for more than one kernels all packed into one file (e.g. myKernels.cl). The program is compiled and linked by the OpenCL call to clBuildProgram.

2.4.4 Mapping Execution to Platform Model: The OpenCL Data Parallel Programming

How an OpenCL execution model is mapped onto the underlying platform model (device hardware architecture)? This defines how different executions are carried out on different processing elements of the device. The OpenCL API defines two
logical but important constructs for carrying out parallel kernel executions on the device. These are

- Work-Item
- Work-Group

A work-item, called a thread in NVIDIA-CUDA, is an instance of the kernel. For performance and synchronization point of view, work-items are grouped into work-groups. This logical grouping is done explicitly by the programmer. In the implicit case the programmer only defines the total number of work-items i.e. the total kernel executions and the work-group arrangement is done by the OpenCL run time. In either case of placing work-items into work-groups, each group is executed on a compute unit in its entirety. Thus work-items of the same work-group can not diverge from one compute unit to another. The total number of work-items \( \text{get\_global\_size(dim)} \) is program dependent. For example in simple array copy-kernel if only a single location is copied from input to output array during a call to the kernel (work-item), then in this case the total number of work-items (denoted here by \( \text{globalSize} \) variable) is the size of the input array. The work-group size, accessed by built-in function \( \text{get\_local\_size(dim)} \) and denoted by \( \text{localSize} \), is such that the \( \text{globalSize} \) is evenly divisible by \( \text{localSize} \) i.e. total number of work-groups is an integer value. This highlights an important aspect that all work-groups executing same kernel must be of the same sizes. Unequal work-group division is not possible in OpenCL. There is also a limit on maximum number of work-items in a work-group \( \text{localSize} \) and is device defined accessed by \( \text{clDeviceInfo} \) with \( \text{CL\_DEVICE\_MAX\_WORK\_GROUP\_SIZE} \) set in the \( \text{cl\_device\_info} \) field. OpenCL allows execution of the work-items in multiple dimensions \( \text{dim} = 1,2,3 \) for 1D,2D and 3D respectively). Each dimension has maximum limit on the total number of work-items \( \text{globalSize[dim]} \) and is device dependent accessed by a call to \( \text{clDeviceInfo} \) with \( \text{CL\_DEVICE\_MAX\_WORK\_ITEM\_SIZES} \) in each dimension. A typical 2D-index space of work-items with work-groups is shown in figure (2.6). Due to the underlying hardware limitations all work-items in a work-group could not be executed all at once, but rather in bunches by the hardware scheduling unit. The bunch of work-items is given the name of warp (usually 32-threads) by NVIDIA and wave-front (64-ATI Radeon HD 5800 series) by ATI/AMD.
Figure 2.6: 2-dimensional \textit{NDRange} index spacing.
2.5 GPU Memory Bandwidth

GPU memory bandwidth plays a major role in obtaining peak performance in terms of FLOPs. In this section it is discussed that how different factors influence performance. An overview of theoretical and effective GPU memory bandwidth is discussed.

2.5.1 Theoretical vs Effective GPU Memory Bandwidth

Theoretical bandwidth is given by the following formula.

\[ BW_{th} = C \times W \times \frac{1}{8} \times R \]

Here \( C \) is the clock-speed or frequency expressed typically in \( MHz \), \( W \) is the width of memory interface in bits and \( R \) is the rate of number memory operations (read/write) per clock tick. For NVIDIA-GTX 260, the clock speed is 999MHz, \( W \) is 448-bits and \( R \) is 2, as in the DDR RAM two memory operations are performed per clock-cycle. Thus the theoretical bandwidth for GTX-260 is 111.888 GB/s. An example explanation is given in [40]. Effective bandwidth \( BW_{eff} = \frac{B_{r} + B_{w}}{T_{t}} \) is the ratio of total bytes of memory used (read from memory and written) to the total time taken in executing the kernel. The effective bandwidth (expressed in Bytes/sec) is practically less than theoretical bandwidth. The reasons are the data load and store patterns, PCI bottleneck, hardware overhead, spatial locality, cache misses etc.

2.5.2 Spatial and Temporal Memory Locality

The locality of reference determines how frequently a particular location and the nearby locations in its vicinity, are accessed. The two important types of locality of reference are the spatial and temporal locality used as a design principle in cache based systems. If a memory location \( m_{i} \) is accessed at time \( t_{i} \), then the spatial locality means that it is highly likely that closed location(s) \( m_{i+1} \) will be accessed at the next time interval \( t_{i+1} \) and is thus cached. Temporal locality refers to the reuse of memory locations in future by the same or other processes. Both locality principles are aimed at minimizing latency i.e. the time between by service request and service grant for a memory access. The later sections discuss in more detail how
the memory access patterns alter performance of an algorithm from peak to worse. If the memory access patterns are not coherent with the spatial/temporal locality based caches results in most cache misses and therefore loads and stores from/to memory (GPU global memory in particular) are made in greater chunks of bytes (mostly 128 Bytes) penalizing performance.

2.5.3 Memory Access Patterns

GPU memory hierarchy differs from vendor to vendor. Here we use global memory, that follows nearly same architecture and is a large memory and slower than other memory types (private, local, shared etc). The table 2.1 shows how different OpenCL implementations of a copy-kernel for copying array elements from one array into another produce different performance results. The linear-copy kernel threads access data elements corresponding to their ids (get_global_id(dim)) in ascending order just like traversing linear array indices. In the permuted-copy kernel, data elements are accessed in a permuted order as shown in figure, where each thread accesses data-element from a shifted location. The following table shows running time and memory bandwidth achieved for a copy-kernel for different memory access patterns. See Chap.3 [40] for different memory access patterns. A coalesced example of array copy for GPU and CPU is shown in the following 2.7. It is shown how a single thread in CPU copy (copy-function) progresses for accessing N items of an array sequentially with corresponding concurrent access (copy-kernel) by N threads (work-items) in the GPU case. The result of CPU-vs-GPU copy is shown in figure 2.8.

<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Time(sec)</th>
<th>B/W GBps</th>
<th>Theoretical B/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coalesced</td>
<td>0.000103</td>
<td>81.058754</td>
<td>111.9</td>
</tr>
<tr>
<td>Permutated</td>
<td>0.000117</td>
<td>71.761292</td>
<td>111.9</td>
</tr>
<tr>
<td>Misaligned</td>
<td>0.000220</td>
<td>38.063599</td>
<td>111.9</td>
</tr>
<tr>
<td>Strided</td>
<td>0.001687</td>
<td>4.972948</td>
<td>111.9</td>
</tr>
</tbody>
</table>

Table 2.1: Effective bandwidth achieved for copying 4MB data ($2^{20}$ float) for different global memory access patterns on GTX-260, W-Group Size=512
Figure 2.7: Array Copy on CPU and GPU.

Figure 2.8: Running time of array copy on CPU and GPU.
2.6 PCIe Bottleneck Effect on Performance

In this section we address the PCIe affecting overall performance of a GPU device which might answer the typical question, "Why an n-processors GPU performance is not n-times of a single core CPU performance?" from people having little understanding of parallel-computing. Well, a lot of factors to be discussed in the answer to this question ranging from distribution of work across parallel processors, memory latency (scatter and gather locality issues), whether task is data/compute intensive, problem size, problem execution paths (mostly serial or parallel or random) and the hardware capabilities of memory load and store, instruction fetch-execute per lock step etc. GPUs need data to operate on, which must be brought into device memory registers before any fetch-execute cycle succeeds. In case of typical discrete GPU computation, firstly data is copied from CPU (host) to GPU (device) memory via PCIe, a point-to-point serial bus of lanes (x4,x8,x16), necessary operations are carried out on portion of data by work-items in a SIMD fashion and output is written to device memory and subsequently transferred to host memory in the last step. In both cases memory transfer from CPU-to-GPU (mtHD) or vice-versa (mtDH) data is transferred via PCIe. Data transfer over the PCI incurs very high latency and very low bandwidth. Thus a slower PCI card adversely affects overall performance. Data transfers both to/from GPU is the most unwanted and unavoidable (in most cases) overhead affecting performance adversely. Generally mtDH takes lesser time than mtHD as mentioned in [9].

2.6.1 Performance Formula

\[
T = M_{HD} + K + M_{DH}
\]

\[
M_{ij} = \sum \frac{d\text{Size}}{B_{ij}}
\]

\[
K = \frac{dR}{B_R} + \frac{dW}{B_W} + C
\]

where \(T\) is the total computation time, \(i,j\) refer to either D (device) or H (host), \(M_{ij}\) is memory transfer time from \(i\) to \(j\), \(d\text{Size}\) data size being transferred, \(B_{ij}\) is bandwidth from \(i\) to \(j\), \(K\) is kernel execution time, \(dR\) amount of data read from
global memory, \( dW \) amount of data written to global memory \( B_{R/W} \) Bandwidth for corresponding read or write to global memory and \( C \) Real operation time of the kernel.

The rest of this thesis briefly explains GPU computational power towards high performance when used in multiple application domains like Sorting large data sets 3, Fourier Transformation in general 4 and accelerating Micromagnetic simulations 5, 6 using GPUs.
Chapter 3

Sorting Large Datasets on Graphic Processing Units

This chapter discusses implementation of sorting algorithms on GPUs. The state-of-the-art sorting algorithms in literature are implemented on GPU architectures. A new sorting algorithm is discussed and implemented on GPU hardware with relatively better performance than state-of-the-art sorting schemes.

3.1 Sorting on GPUs

The sorting problem of arranging a given sequence of input data into a certain order (monotonic increase or decrease) has been widely studied in literature. Several sorting techniques exist in literature (for further details see [18, 22, 59, 54]) Many computational areas involve operations on the sorted data or output is produced in sorted order and thus time complexity of the underlying sorting algorithm is significant in overall simulation time. Sorting algorithms are categorized by their time complexity for best, average and worst case analysis. The time complexity is not the only deciding parameter but other factors like stability, robustness, scalability, input distribution, memory storage and access patterns decide the applicability of a sorting algorithm in the area in focus. In this chapter only parallel sorting techniques are discussed for the purpose to utilize GPU computation power. As mentioned in the following, sorting technique having inherent data parallelism in the design

performs better when implemented on GPUs. The following algorithms have been implemented and tested against their CPU serial implementations.

- Rank Sort
- Odd-even Sort
- Bitonic Sort
- Min-Max Butterfly Network Sort
- Full Butterfly Network Sort

The list above is in increasing order of sorting algorithms in terms of parallelism involved in the design of respective algorithms. As the mentioned techniques are well discussed in literature, an algorithm is provided for each with detailed design and implementation for the last two which are new (see [33, 34] for more details). The algorithms for Rank, Odd-even and Butterfly sorting is given in the following.

\begin{verbatim}
input : data_{random}, size=N
output: data_{sorted}
begin
    for k_{out} ← 1 to \frac{N}{2} do
        do parallel
            if \ i > i+1 \forall \ i \% 2 \neq 0 then
                swap (i, i+1)
            end
        end
        do parallel
            if \ i > i+1 \forall \ i \% 2 == 0 then
                swap (i, i+1)
            end
        end
    end

Algorithm 1: Odd Even Sort
\end{verbatim}

In these algorithms the parallel part is coded as OpenCL kernel which is run on the GPU hardware.
input : data\textsubscript{random}, size = N
output: data\textsubscript{sorted}

begin
for \( k \leftarrow 1 \) to \( N \) do
\textbf{do parallel}
\hspace{1em} Q[n] = 0
\hspace{1em} for \( i \leftarrow 1 \) to \( N \) do
\hspace{2em} for \( j \leftarrow 1 \) to \( N \) do
\hspace{3em} if \( Q[j] < Q[i] \) then
\hspace{4em} \( r[i]++ \)
\hspace{3em} else
\hspace{4em} \( r[j]++ \)
\hspace{2em} end
\hspace{1em} end
\textbf{end}
\textbf{do parallel}
\hspace{1em} for \( k \leftarrow 1 \) to \( N \) do
\hspace{2em} U[r[k]] = a[k]
\hspace{1em} end
\textbf{end}
\textbf{do parallel}
\hspace{1em} for \( k \leftarrow 1 \) to \( N \) do
\hspace{2em} a[k] = u[k]
\hspace{1em} end
\textbf{end}
end

Algorithm 2: Rank Sort
3.2 Butterfly Network Sorting

A sorting algorithm is either in-place or out-place in terms of memory storage. In the first case, also called internal, no extra memory other than consumed by input sequence is required. In the out-of place, a.k.a external, an extra disk storage is required for storing intermediate results. Another classification is based on whether sorting is based on elements of the data or their binary representation. In the comparison based technique, the former case, two or more values are compared and swapped based on the desired output scheme. The non-comparison techniques (e.g. radix-sort) operates on the binary representation of the input values or the distribution of the input data. The comparison based techniques [14] (odd-even, rank, bitonic etc) has complexity proportional to $\Theta(n \log n)$. Radix-sort has complexity of $\Theta(n)$. The Butterfly sorting is in an in-place highly parallel comparison based sorting technique discussed in the following. a minimal version of the algorithm scans the input array of inputs for extreme values in relatively small steps with relatively small memory accesses than a full sorting algorithm. The minimal version is given the name of Min-Max Butterfly sorting. The complete sorting technique is simply named as Butterfly sorting or interchangeably Full Butterfly sorting wherever a distinction is made from the Min-Max Butterfly.

3.2.1 Min-Max Butterfly

The butterfly is a $2 \times 2$ comparator that compares the input values and swaps in ascending or descending order. A sample $2 \times 2$ butterfly diagram is shown in fig 4.1 which compares two values $x_0$ and $x_1$ and places output at the right wings of the butterfly. In this case it is assumed that $x_0 < x_1$ for a descending order. The min-max butterfly searched for the minimum and maximum in the input data in relatively less time. The min-max algorithm for input data of size $N$ has $\log_2 N$ stages. Inside each particular stage $s_i$ total number of comparators (the butterflies) remains the same i.e. $N/2$. Each comparator operates on two input values only with a stride between the input values changing from stage to stage. The stride (gap between input values to the butterfly) has a regular structure depending upon the stage number which varies from $s_1 \cdot \log_2 N$. Thus the stage number value ($1...\log_2 N$) is used for the stride calculation between the inputs to a butterfly. The complexity
of min-max algorithm in terms of butterfly comparators is \((N/2)\log_2 N\). A min-
max butterfly method is shown in fig 3.2. Here the input sequence can be of any
distribution (uniform, random, exponential etc). The parallel implementation is
such that at each stage \(N/2\) butterflies are executed in parallel. In OpenCL context
this means \(N/2\) work-items are launched at a particular stage. Each work-item
operates on different two data values which form the corresponding inputs and
outputs of the butterfly comparator. In the algorithm (3) \(Pos_{start}\) and \(Pos_{end}\) points
to the two values of the butterfly comparator. As the input to any particular stage,
excluding the first one, is dependent on the output from previous stage, thus stages
can not be executed in parallel. An example min-max butterfly method is shown in
fig 3.2. After completion of all stages, minimum and maximum are placed at first
and last location respectively.

\textbf{input} : data, size=N, base=2

\textbf{output} : data

\begin{verbatim}
begin
  for stage ← 1 to \log_2(size) do
    \(P = base^{stage};\)
    do parallel \(T\)
      \(yIndex = t/(P/radix);\)
      \(kIndex = t\%(P/radix);\)
      \(Pos_{\text{start}} = kIndex + yIndex \times P;\)
      \(Pos_{\text{end}} = kIndex + yIndex \times P + P/radix;\)
      if \(data[Pos_{\text{start}}] > data[Pos_{\text{end}}]\) then
        swap(data[Pos_{\text{start}}],data[Pos_{\text{end}}])
      end
    end
  end
end
\end{verbatim}

\textbf{Algorithm 3}: Min/Max Butterfly Sorting Network

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{min_max_butterfly.png}
\caption{Min-Max butterfly for input of size 8. Minimum and maximum values are placed at the most upper and bottom location respectively.}
\end{figure}
3.2.1.1 Full Butterfly Sort

A sorting network is a column wise arrangement of the comparison operators. Multiple such columns are interconnected in such a way that the output of any intermediate column is input to the subsequent column. The Full Butterfly sort is a sorting network where input values are compared and exchanged through a butterfly circuit (shown in figure 4.1). The number of inputs-outputs to/from the comparator are determined by the radix (or base) for the butterfly sort. For a given input sequence of length $N$ where $N = r^n$, $r$ is the radix for the butterfly sort. The computational complexity for radix-2 Butterfly sort is given in the following.

$$
Stages = \log_2(N) + \sum_{i=1}^{\log_2(N)-1} i \quad (3.1)
$$

$$
Comparators_{2\times2} = \frac{N}{2} \times Stages \quad (3.2)
$$

In equation 3.1 $\log_2(N)$ are the stages determined by the first $do-parallel$ block of the algorithm 4. The $\sum$ term in equation 3.1 gives the stages based on the second $do-parallel$ block. In the OpenCL implementation At each stage the total number of comparators remain the same i.e. $N/2$. An example of butterfly sorting is shown
Algorithm 4: Full Butterfly Sorting

```
in figure 3.3.

\begin{algorithm}
\caption{Full Butterfly Sorting}
\begin{algorithmic}
\State \textbf{input} : \text{data}_{\text{random}}, \text{size}=N
\State \textbf{output}: \text{data}_{\text{sorted}}
\begin{algorithmic}
\State \text{begin}
\For{$x_{\text{out}} \leftarrow 1$ to $\log_2(\text{size})$}
\For{$x_{\text{in}} \leftarrow x$ to 1}
\State $\text{PowerX} = \text{radix}^{x_{\text{out}}}$;
\State \text{do parallel} $T$
\State \quad $\text{yIndex} = t / (\text{PowerX}/\text{radix});$
\State \quad $\text{kIndex} = t \% (\text{PowerX}/\text{radix});$
\State \quad $\text{Pos}_{\text{start}} = \text{kIndex} + \text{yIndex} \times \text{PowerX};$
\State \quad $\text{Pos}_{\text{end}} = \text{PowerX} - \text{kIndex} - 1 + \text{yIndex} \times \text{PowerX};$
\If{\text{Pos}_{\text{start}} > \text{Pos}_{\text{end}}}
\State swap($\text{Pos}_{\text{start}}, \text{Pos}_{\text{end}}$)
\EndIf
\EndFor
\If{$x > 1$}
\For{$x_{\text{in}} \leftarrow x$ to 1}
\State $\text{PowerX} = \text{radix}^{x_{\text{in}}}$;
\State \text{do parallel} $T$
\State \quad $\text{yIndex} = t / (\text{PowerX}/\text{radix});$
\State \quad $\text{kIndex} = t \% (\text{PowerX}/\text{radix});$
\State \quad $\text{Pos}_{\text{start}} = \text{kIndex} + \text{yIndex} \times \text{PowerX};$
\State \quad $\text{Pos}_{\text{end}} = \text{kIndex} + \text{yIndex} \times \text{PowerX} + \text{PowerX}/\text{radix};$
\If{\text{Pos}_{\text{start}} > \text{Pos}_{\text{end}}}
\State swap($\text{Pos}_{\text{start}}, \text{Pos}_{\text{end}}$)
\EndIf
\EndFor
\EndIf
\State \text{end}
\EndFor
\State \text{end}
\EndAlgorithm
\end{algorithmic}
\end{algorithm}
```
Figure 3.3: Full Butterfly Sorting for $N=8$ with $2 \times 2$ butterfly comparators. *kernel-out* and *kernel-in* points to upper and lower parallel blocks in the algorithm 4 as highlighted by dark and light color stages respectively.

Figure 3.4: Time breakdown for memory transfer and GPU execution of each stage/sub-stage for input data of size 16

### 3.2.1.2 OpenCL Implementation

The host side of the butterfly sort need all necessary *OpenCL* data structures explained in 2.4.3.1. Two kernels (*bFlySort1DRadix2in* and *bFlySort1DRadix2in*) are created for the parallel blocks in 4. The total number of work-items (*globalSize*) in the *NDRange* is set as $N/r$ ($r = 2 \forall N = 2^n$) for each kernel execution. The work group size (*localSize*) is optimized based on calculation from 2. Number of kernel invocations for each kernel type (kernel-out or kernel-in) are based equation 3.1.

---

3.2.1.3 Profiler Analysis

OpenCL vendor implementations (tool kit or sdk) are often equipped with profiler analysis and debugging tools. These profiling tools present several useful information (both console and GUI based) about the API, kernel execution and the underlying hardware architecture. The information from such profilers are important for optimizing an OpenCL program. The example butterfly sort technique is tested on various NVIDIA GPUs so profiling information from NVIDIA Visual Profiler are discussed here. NVIDIA compute visual profiler presents useful information in a graphical way, generated after compiling the code. Such information include GPU-CPU times, memory load-store analysis, occupancy etc. One such profiler output of the GPU Time Summary plot is shown in Fig. 3.4. The corresponding bars in the figure, are sorted in decreasing order of the GPU time spent by each method. All profiler analysis are taken on NVIDIA GeForce GTX 260 with compute capability 1.3. Number of work-groups and their sizes plays significant role in over performance of the application. Throughput is also affected by the global memory access type that is typically accessed in 32, 64 or 128-byte chunks. Figure 3.5 shows number of diverging threads in the case of one stage of the full butterfly sort for input sizes 128 and 512 using float data type that is 4-byte long. For each butterfly two memory accesses; $Pos_{Start}$ and $Pos_{End}$, are carried out for each memory load and store operation in 32, 64 or 128 byte chunks. After every access, the chunk size may be changed depending on device compute capability rules.

Throughput is mainly affected by global-local work-group sizes and the memory accesses by work-items as shown in Fig. 3.6. Memory load ($\rho_l$) and store ($\rho_s$) throughput for all stages ($st$) from visual profiler are analyzed as percentage average values for data of various sizes (n) by using following equations:

$$\rho_l = \sum_{st} \left[ gld_{32} \times 32 + gld_{64} \times 64 + gld_{128} \times 128 \right]$$

$$\rho_{lavg} = \frac{\rho_l}{total_{st}} \times 100$$

---

3\textsuperscript{.3} Results and Comparisons

3.3.1 Architecture Details

The architecture details for GPU and CPU devices used for testing the min-max and full butterfly sorting are presented in table 3.1. OpenCL 1.2 and the GNU C
Figure 3.6: Memory load ($\rho_l$)-store ($\rho_s$) counts and thread Divergence counts for a single stage of full butterfly sort. The position of peaks observable in (a) where $localSize$ comes from equation 3 and (b) $localSize = 256$ for $dataSize \geq 512$ conform to the length of the work-group sizes and are inevitable because the work-group sizes cannot be exceeded beyond 512. The thread divergence grows exponentially as the size increases. Figure (c) showing the log scales reports a step again conforming to the length of the work-group size. This step is explained in Figure-3.5.

Compiler (gcc 4.5.3) were used in testing aforementioned sorting techniques.

| Architecture Details | NVIDIA | | | Intel |
|---|---|---|---|
| | GT320M | GTX 260 | Quadro 6000 | Core2Quad Q8400 |
| Micro-processors |
| Cores per Processor |
| Total-Cores |
| Clock-Rate in MHz |
| FLOPS |
| Memory-Bandwidth (GB/s) |
| | 3 | 27 | 14 | 2 |
| | 8 | 8 | 32 | 2 |
| | 24 | 216 | 448 | 4 |
| | 1100 | 1242 | 1147 | 2660 |
| | 158 | 874.8 | 1030.4 | 42.56 |
| | 9.95 | 91.36 | 144 | - |

Table 3.1: Specifications of CPU and GPUs
3.3.2 Experimental Setup

All simulations are carried out in OpenCL 1.2 and standard C compiler for different queue sizes in the power of 2. Input data of type float, is taken from a random number generator with size in the range of $2^{10}$ to $2^{25}$. Variable declaration/initializations, random variate generators and other memory reads/writes to/from queues are mainly limited to CPU in host program. Actual sorting, butterfly computation, is carried out on GPU in kernel code. Hardware architectures used for simulations are Nvidia-Quadro6000, GeForce GTX260 and GeForce GT320M for parallel implementation and Intel Core2Quad CPU Q8400 for serial implementation. Visual profiler analysis are carried out on NVIDIA-GeForce GTX 260 of compute capability 1.3\(^5\).

3.3.3 Results

In the following sorting time, sorting rate and speedup comparisons for min-max and full butterfly sorting algorithms are shown for different GPU architectures specified in 3.1. The sorting time is computed here as the total time taken by the OpenCL kernels for sorting a given input size data. The kernel execution times are computed using OpenCL profiling command (clGetEventProfilingInfo) with event objects associated with each kernel invocation (see Chap.5 of [60] for measuring kernel execution times). The sorting time does not take into account any memory transfer times between CPU and GPU device. The sorting rate which is defined by the ratio of the input size and sorting time, is shown for both min-max and full butterfly sorting. Due to highly parallel structure of the min-max and full butterfly algorithms, the GPU implementation achieves considerable speedup against corresponding GPU implementations of bitonic,odd-even and rank-sort techniques. For comparison purposes the results of bitonic, odd-even and rank-sort are taken from [38]. A serial implementation of each sorting type is implemented and tested on CPU as highlighted in the figures by "Core 2 Quad CPU" legend type. As shown in figure 3.9 serial implementation of our butterfly sorting has a speedup of 2x against serial implementation of bitonic sort.

\(^5\)Nvidia Compute Capability.

[www.nvidia.co.uk/object/cuda_gpus_uk.html](www.nvidia.co.uk/object/cuda_gpus_uk.html)
Figure 3.7: Sorting time (left) and rate (right) of min-max butterfly for different power-2 sizes

Figure 3.8: Sorting time (left) and rate (right) of full butterfly sorting for different power-2 sizes.

3.4 Conclusion

Sorting algorithms are broadly used in variety of different application types. Comparison based sorting algorithms are dealt for their CPU and GPU implementations. For the GPU implementation, it is highlighted that the more parallel a sorting technique is, the lesser computation time it takes in sorting a given input sequence. To this fact a speedup comparison highly parallel butterfly sorting versus less parallel
Figure 3.9: Speedup of full butterfly against bitonic (left) and odd-even (right) sorting algorithms.

rank-sort is shown in figure 3.9.
Chapter 4

GPU Implementation of Fast Fourier Transforms: ToPe-FFT

4.1 Discrete Fourier Transform: DFT

Discrete Fourier Transforms named after the French mathematician J.B.J Fourier, is the decomposition of a discrete signal in terms of complex sinusoids for analysis. It is more useful to decompose the signal in terms of sin-cosine waves than any other decomposition type because such decomposition does not change frequency of the system which remains the same both at input and output i.e. the sinusoidal fidelity of linear systems (see chap 5 of [65]). The input to a DFT can be real or complex. Throughout this discussion we refer DFT to the real Fourier transformation applied to a finite length discrete signals sampled usually at regular time intervals. A Real DFT, converts a signal of $N$ samples into corresponding sine and cosine amplitude values. The frequency of each sinusoid (both sine and cosine) varies from one component to another. The input signal is sampled at regular time intervals and is called time-domain signal where as the output signal are $N + 2$ amplitude values in total (each of $N/2 + 1$) and is termed as frequency domain signal. A DFT in other words is the Fourier transformation of time-domain signal into frequency domain. The input signal usually is the time domain signal of $N$ samples indexed by variable $n$. The output of the DFT engine are sets of cosine and sine amplitude signals. Each set is composed of $N/2 + 1$ samples indexed by variable $K$. Each $K^{th}$ instance of the set is composed of $N$ points with $K$ representing the frequency ($K$ complete
cycles in \( N \) sample points) of the corresponding sinusoid. The \( X[K] \) represents sum of the amplitudes at that instant. To put the description in mathematical form, we obtain the following DFT formula for a forward transform.

\[
X[K] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi nk}{N}}
\]

(4.1)

\[
= \sum_{n=0}^{N-1} x[n] W_N^{nk}
\]

(4.2)

The inverse DFT (IDFT) converts back the frequency domain signal into time domain in the similar way but this time the amplitude values are multiplied with the basis functions and then summed up for each time domain sample. The IDFT is computed using the same DFT formula with conjugated twiddles and the input values are scaled at \( 1/N \). Mathematically expressed in the following synthesis equation 4.4.

\[
x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[K] e^{j\frac{2\pi km}{N}}
\]

(4.3)

\[
= \frac{1}{N} \sum_{k=0}^{N-1} X[K] W_{N}^{-kn}
\]

(4.4)

In equations 4.2 and 4.4, \( W_N^{nk} \) is the called the twiddle factor which is the complex number representation of the sinusoids. The matrix representation of all the twiddle factors involved in the DFT computation is called Fourier matrix or coefficient matrix. This representation leads to a more simplified definition of a DFT to be a matrix-vector product of coefficient matrix and the time domain input vector as given in the following.

\[
\begin{bmatrix}
X_0 \\
X_1 \\
X_2 \\
\vdots \\
X_{N-1}
\end{bmatrix} =
\begin{bmatrix}
W^0 & W^0 & W^0 & \cdots & W^0 \\
W^0 & W^1 & W^2 & \cdots & W^{N-1} \\
W^0 & W^2 & W^4 & \cdots & W^{2(N-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
W^0 & W^{(N-1)} & W^{2(N-1)} & \cdots & W^{(N-1)(N-1)}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_{N-1}
\end{bmatrix}
\]

(4.5)
The complex multiplication of two numbers require 4 real multiplications and 2 real additions. In direct matrix-vector product for DFT of a complex input vector \( x(n) \) of size \( N \), the computation of each \( k^{th} \) value in the frequency domain signal \( X[k] \) require \( N \) complex multiplications and \( N - 1 \) complex additions. Thus computation complexity for the DFT of \( N \) size output is given in the following.

\[
\text{Complexity} = N \cdot N + N(N - 1) \\
= 2N^2 - N \quad (4.6) \\
\approx \mathcal{O}(N^2) \quad (4.7)
\]

The complex multiplication of two numbers typically require 4 real multiplications and 2 real additions. The complex addition however require 2 real additions. If each real multiplication or addition consumes one FLOP (Floating Point Operation), thus total number of flops for the DFT is given as.

\[
\text{FLOPs} = N \cdot N + N(N - 1) \\
= N^2(4 + 2) + (N^2 - N)(2) \quad (4.8) \\
= 8N^2 - 2N \quad (4.9)
\]

The high computation time of DFT via the direct matrix-vector product for larger input sizes \( N \) overwhelm its benefits to be used in practical applications. It was until the introduction of fast algorithm by J.W Cooley and J.W Tukey in 1965 [17] with a reduction in computational complexity by hundreds than the traditional approach. The complexity was reduced to \( \mathcal{O}(N \log N) \) under the collective name of \textit{fast Fourier transform (FFT)} methods. In the following table a comparison is shown between complexity of matrix-vector product approach and that of the FFT. Several FFT techniques where complexity is reduced roughly to the order of \( 5N \log N \) are discussed in the rest of this chapter.
4.2 The Fast Fourier Transform: *Fast methods for DFT computation*

Fast Fourier Transform (FFT) is the set of fast algorithms for computing Discrete Fourier Transform (DFT). The FFT method was first proposed by J.W. Cooley and J.W. Tukey in their landmark paper \[17\] but the roots of discovery goes back to Gauss as briefly highlighted in here \[29\]. The Cooley-Tukey algorithm exploits the properties of the twiddle matrix (a.k.a Fourier matrix, coefficient matrix) in equation (4.11) where the DFT is computed recursively by divide-and-conquer approach. The FFT techniques exploit the following twiddle factor properties.

- Symmetry \(W_N^{k+N} = -W_N^k\)
- Periodicity \(W_N^{k+N} = W_N^k\)
- Recursion \(W_N^2 = -W_{N/2}\)

The Fourier coefficient for \(N = 4\) (i.e. \(F_4\)) based on the above properties is shown in the following.

\[
F_4 = \begin{bmatrix}
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
\end{bmatrix} = \begin{bmatrix}
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
W_4^0 & W_4^0 & W_4^0 & W_4^0 \\
\end{bmatrix}
\]  
(4.10)

\[
F_4 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
i & -1 & -i & i \\
1 & -1 & 1 & -1 \\
i & -1 & -i & i \\
\end{bmatrix}
\]  
(4.11)

4.2.1 Cooley-Tukey FFT

The Cooley-Tukey (CT) algorithm splits a given composite input size \(N = r^n\) into smaller sub sequences recursively until the block (or sequence) size reaches \(r\). The DFTs of \(r - \text{point sequences}\) are computed and combined in sub transformed signals. This computation-combining process is repeated until the transform length reaches \(N\). The process of splitting, computing the DFT of each sub length and
combining is called divide and conquer approach. The divide and conquer approach of the DFT for composite \( N \) saves many computations. Consider the basic case where \( N \) is a power of 2 (\( N = 2^n \)). The Cooley-Tukey algorithm in its basic form of \( N = 2^n \) splits the given input time domain signal (hence decimation-in-time:DIT) into odd and even sequence. In other words the radix-2 DIT CT algorithm divides the given \( N \) into two \( N/2 \) sequences (odd and even sequence). Each \( N/2 \) length sequence is further divided into corresponding even and odd signals each one of length \( N/4 \). The process of splitting is repeated for each sub-sequence until the sub-sequence size \( (N') \) reduces to 2 (in general case to that of radix size i.e. \( N' = r \)).

The general case of radix \( r \) DIT CT of input size \( N = r^n \), (1) split the input signal of size \( N \) into \( N/r \) sub signals each of length say \( N' \) (2) repeat the splitting in 1 until \( N' = r \) called elementary size (3) perform \( r \)-point DFT of each sub signal (radix-\( r \) butterfly) (4) combine the sub DFTs to make larger signals of size say \( N'' \) (5) repeat 3 and 4 until transform size reaches \( N \) that of input size.

The expression for radix-2 DIT CT is given in the following.

\[
X[K] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi kn}{N}} \quad \forall k = 0, 1 \ldots N - 1
\]

\[
= \sum_{n=0}^{N/2-1} x[2n]W_N^{kn} + \sum_{n=0}^{N/2-1} x[2n+1]W_N^{k(2n+1)}
\]

\[
= \sum_{n=0}^{N/2-1} x[2n]W_{N/2}^{kn} + \sum_{n=0}^{N/2-1} x[2n+1]W_{N/2}^{kn}W_N^k
\]

\[
= \sum_{n=0}^{N/2-1} x[2n]W_{N/2}^{kn} + W_N^k \sum_{n=0}^{N/2-1} x[2n+1]W_{N/2}^{kn}
\]

\[
= A[k] + W_N^k \cdot B[k]
\]

Equation (4.13) is obtained using recursion property of the twiddle factors discussed above. In equation (4.15) \( A[k] \) is the \( N/2 \)-point DFT for the even terms whereas \( B[k] \) is the \( N/2 \)-point DFT for odd terms. The two \( N/2 \)-point DFTs are combined by \( k \) twiddle factors multiplied with the DFT of odd terms to give the final \( N \)-point DFT.
i.e. \( X[k] \). In equation (4.15) each DFT complexity is \((N/2)^2\) and the cross twiddle multiplication complexity is \(N\). The total complexity \(\frac{N^2}{2} + N \approx \frac{N^2}{2}\) is reduced by an order of 2 to the total complexity of equation (4.2). The complexity can be reduced even more by exploiting periodicity and symmetry properties. In equation (4.15) both \(A\) and \(B\) are periodic with period \(N/2\). The periodicity property leads to the following.

\[
X[k] = A[k] + W_N^k \cdot B[k]
\]

\[
X[k + N/2] = A[k + N/2] + W_N^{k+N/2} \cdot B[k + N/2] \quad \forall k = 0,1 \ldots \frac{N}{2} - 1
\]

(4.16)

Using the periodicity property it comes out that \(A[k] = A[k + N/2]\) and similarly \(B[k] = B[k + N/2]\). The symmetry property of twiddle factors state that \(W_N^{k+N/2} = -W_N^k\). Putting the results of periodicity and symmetry the above equation takes the form as:

\[
X[k] = A[k] + W_N^k \cdot B[k]
\]

\[
X[k + N/2] = A[k] - W_N^k \cdot B[k] \quad \forall k = 0,1 \ldots \frac{N}{2} - 1
\]

(4.17)

The complexity is reduced further by a factor \(N/2\) in cross twiddle multiplication and thus the overall complexity becomes \(\frac{N^2}{2} + N/2\). In matrix notation the radix-2 DIT Cooley-Tukey is written as:

\[
\begin{bmatrix}
X[k] \\
X[k + N/2]
\end{bmatrix} = \begin{bmatrix}
I & W_N^k \\
I & -W_N^k
\end{bmatrix} \times \begin{bmatrix}
A[k] \\
B[k]
\end{bmatrix}
\]

(4.18)

Here \(I\) is the identity matrix of the order same as \(A\) or \(B\). This expression is represented by the following radix-2 CT butterfly diagram.

The radix-2 CT butterfly in the above diagram requires one complex multiplication and two complex additions. The decimation of original sequence by a factor of \(r\) (\(r\) is radix) is repeated recursively for each subsequence until the sequence
containing only one element \((N \text{ reduces to } 1)\). For radix-2 it takes \(\log_2 N\) steps to reduce \(N\) size sequence to unit size. Since each division stage has complexity \(N/2\) in terms of complex multiplications and \(N\) in terms of complex additions. Thus overall complexity for all stages turns out to be approximately \(N/2 \log_2 N\) (dominating term is the complexity of complex multiplications consuming higher number of flops than complex additions). In summary the radix-2 DIT Cooley-Tukey algorithm splits the given input sequence into even and odd sequences and this splitting is continued for the resulting sub sequences and so on until single point sequences occur. The single point sequences are directly obtained by applying bit-reversal algorithm to input sequence i.e. re-arranging given input sequence corresponding to its binary representation of the location index. In bit reversal algorithm for example the second location value \(x[l_1 = (001)_{\text{binary}}]\) is moved to new location (fifth position) \(x[l'_4 = (100)_{\text{binary}}]\) as shown in the figure 4.2.

The Cooley-Tukey divide and conquer strategy of sub-DFTs is shown in the following figure.

### 4.2.1.1 Kronecker product explanation of CT-FFT

The FFT techniques are explained with the help of kronecker product \([62]\) with a matrix decomposition explanation is provided along side. The kronecker (a.k.a tensor product) of two matrices say \(A\) and \(B\) is defined as:

\[
A \otimes B = \begin{bmatrix} a_{k,l}B \end{bmatrix}
\]
\[ A \otimes B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} \\ \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \end{bmatrix} \]
The CT algorithm uses bit reversal either at input or output. Three strategies are usually used to digit-reverse an index space. These are bit-reversal, matrix transposition (in case of multi-dimensional transforms) and multiply input vector by a stride permutation matrix $L_{mn}$. These methods are interchangeable. The stride permutation method rearranges the indices of input vector of length $mn$ ($N = m \times n$) as a mapping of indices from $in + j \mapsto jm + i$ where $0 \leq i < m$, $0 \leq j < n$. For example the stride permutation matrix for size $N = 6$ is given in the following. An example for radix-2 is given in [24]. Matrix permutation $L_{mn}$ results like matrix transposition if the input vector $x$ is reshaped into $n \times m$ row-major matrix.

\[
L_3^6x = L_3^6 \times x \\
= \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \times \begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5
\end{bmatrix} = \begin{bmatrix}
x_0 \\
x_3 \\
x_1 \\
x_4 \\
x_2 \\
x_5
\end{bmatrix}
\]
Similarly

\[ L_2^6 x = L_2^6 \times x \]

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_0 \\
x_2 \\
x_4 \\
x_1 \\
x_3 \\
x_5
\end{bmatrix}
\]

The kronecker product and stride permutation for radix-r DIT CT for input of a composite size \( N (N = rm) \) is given by the following relation.

\[ DFT_N = (DFT_r \otimes I_m) T_m^{\text{rm}} (I_r \otimes F_m)L_r^{\text{rm}} \]

In equation (4.24) \( I \) is the identity matrix, \( T \) is the diagonal twiddle matrix and \( L \) represents a permutation matrix. An example of \( N = 4 \) based on equation (4.24) is shown in the following.
\[ F_N = F_4 = (F_2 \otimes I_2) \quad T_2^4 \quad (I_2 \otimes F_2)L_2^4 \]
\[ F_{2m} = (F_2 \otimes I_m) \quad T_2^{2m} \quad (I_2 \otimes F_m)L_2^{2m} \]

\[
F_{2m} = \begin{bmatrix} I_m & I_m \\ I_m & -I_m \end{bmatrix} \begin{bmatrix} I_m & 0 \\ 0 & -W_m \end{bmatrix} \begin{bmatrix} F_m & 0 \\ 0 & F_m \end{bmatrix} L_2^{2m}
\]

(4.25)

\[
F_{2/2} = \begin{bmatrix} I_2 & I_2 \\ I_2 & -I_2 \end{bmatrix} \begin{bmatrix} I_2 & 0 \\ 0 & -W_2 \end{bmatrix} \begin{bmatrix} F_2 & 0 \\ 0 & F_2 \end{bmatrix} L_2^{2/2}
\]

\[
F_{2/2} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

In equation (4.25) \( W_m = \text{diag}(W_{N/2}^{0,\cdots,N-1}) \) is the diagonal matrix of twiddle factors.

A radix-2 DIT CT for input size \( N = 8 \) is shown in figure 4.4.

### 4.2.2 Higher Radices for CT FFT

Higher radices improves performance by reducing number of computations [36] involved in combining phase of the CT algorithm. The case of radix-4 DIT CT butterfly diagram is shown in 4.5. ToPe-FFT therefore extends FFT kernels beyond radix-2 to several base radices like radix-3,4,5,6,7,8,10 and also mixing these base radices as discussed in the following. A radix-4 butterfly is shown in the following figure.

### 4.2.3 Mixed Radix FFT

In cases where input size \( N \) is not any power of the base radices but rather a composite number of several factors each one as a power of the base radices i.e. \( N = N_1 \times N_2 \cdots \times N_n \). Thus in mixed radix approach to \( N \), each factor say \( N_i \) is dealt as \( N_i - \text{point} \) DFT which is computed using a radix-\( r \) (\( N_i = r^n \)) CT FFT technique as explained above. The factors of \( N \) can be computed in any
order but will be explained later (during implementation of FFT) that how their order improves the performance when implementing them. The mixed radix FFT is expressed mathematically in the following.
Figure 4.5: CT radix-4 butterfly diagram

\[
N = N_1 \times N_2 \cdots \times N_n
\]

\[
X[K] = R_1[K_1] \times R_2[K_2] \cdots \times R_n[K_n]
\]

\[
= \prod_{i=0}^{F-1} R_i[i]
\]  

48
Thus, the twiddle factors can be re-written as:

\[ W_{nk}^N = W_{N_x N_y}^{(n_x N_x + n_y)(k_x N_y + k_y)} \]

After simplification, the DFT using the new set of factors can be re-written as:

\[ X(k_x, k_y) = \sum_{n_x=0}^{N_x-1} \sum_{n_y=0}^{N_y-1} x(n_x, n_y) W_{N_y}^{n_y k_y} W_{N_x N_y}^{n_x k_y} W_{N_x}^{n_x k_x} \]  

(4.27)

This new scheme can be realized as a 2D FFT using the row-column approach [13], with an intermediate step involving the point-wise multiplication of the twiddles \( W_{N_x N_y}^{n_x k_y} \). Preserving the order of these steps is crucial to a properly computed mix-radix algorithm. An example based on this method is shown in Fig. 4.6, showing the 3 × 3 factorization matrices of the twiddle factors for a radix-9 FFT. The inner-most matrix representing a toeplitz structure is used for performing column-based FFT’s. The central diagonal matrix performs a point-wise complex multiplication and the outer most block-diagonal matrix is used for performing the row-based FFT’s. A hierarchical factorization structure is used for problems involving a higher number of factorizations. Considering a transform of size 1024, the factorization can be given as:

\[ F_{1024} = A_1 A_2 A_3 \cdots A_{10} P_{1024} \]

where \( F_N \) is the transform size, \( A_n \) is the smallest factorization of size 2, and \( P_n \) is the permutations of the \( A \) matrices. The FFT solves the two smallest factorization units and recurses into the remaining factors. This introduces two more components in the factorization process:

\[ F_{1024} = A_1 A_2 A_3 \cdots A_8 P_{256} B_1 B_2 Q_4 \]

where \( B_n \) are the transformed factors of size 2 and \( Q_n \) is the permutations that have
been performed. The recursive process then proceeds as:

\[
F_{1024} = A_1 A_2 A_3 \cdots A_7 P_{128} B_1 \cdots B_3 Q_8 \\
F_{1024} = A_1 A_2 A_3 \cdots A_6 P_{64} B_1 B_2 \cdots B_4 Q_{16} \\
\vdots \\
F_{1024} = A_0 P_0 B_1 B_2 B_3 \cdots B_{10} Q_{1024}
\]

An FFT factorization engine using this approach can significantly improve performance [44]. Our code is able to handle up-to 4 factors using this engine.

![Figure 4.6: Mixed-radix formulation of radix 9 using 3 x 3 factorization.](image)

Mixed radix-9 formulation is shown in figure 4.6 using a 3 \times 3 factorization. The inner most Toeplitz matrix performs the FFT using the column positions, the central diagonal matrix performs a point-wise multiplication of the twiddle factors \(W_{n_k n_y}\), and the outer-most block diagonal matrix performs the FFT using the row positions.

### 4.3 GPU FFT Implementation: ToPe FFT Library

The ToPe-FFT library provides implementation of FFT based on the Cooley-Tukey method for nearly arbitrary input size. The dimension size can be up-to 3D. At the core of the library are the OpenCL kernels for base radices. ToPe-FFT provides implementation for various even and odd radices (2-10). Moreover the library supports mixing radices up-to two factors. The interface with base radices implementation is discussed in the following. The library design is kept consistent with state of the art FFT libraries: FFTW and CUFFT library. The library features are summarized in the following.

- Almost arbitrary length transform size (radices 2-10)
Mixed Radices support

- Complex-to-Complex (C2C) Transform type, can handle (R2C)
- Single and Double precision support (C2C,Z2Z)
- 1D, 2D and 3D transformations
- Load balancing on multiple GPUs
- Batch execution of multiple same length transforms
- In-place or out-of-place transformation depends on FFT algorithm being selected ([58, 35])
- Cross platform (tested version Linux only)
- Open source GPU FFT library using OpenCL [57]

In the following the design, implementation, usage and performance results of ToPe-FFT are presented.

4.3.1 ToPe-FFT Design

ToPe-FFT library is based on the modular design principle for easy debugging, maintenance, robustness and scalability for cross-platforms. ToPe-FFT implements the following code-blocks as separate modules for carrying out FFT transformation of a specific type. As mentioned in chapter 1, heterogeneous programming using OpenCL is roughly divided into host and kernel code-portions which run on host (typically CPU) and device (GPU) respectively. The following code-blocks constitute the host which internally call specific kernel codes of a certain type.

4.3.1.1 Framework

The framework structure queries the GPU device and initializes the OpenCL platform model basic data structures (discussed in 2.4.3.1) based on the device capabilities. This includes the OpenCL version supported, list of available GPU devices (in cases of multi-GPU devices for load balancing 4.3.3), a list of command queues one per device, program and the creation of OpenCL context. The figure 4.7 shows the multi-GPU framework.
4.3.1.2 Plan

The plan data structure includes all the required constructs for an FFT of a given type. It is where the FFT theory meets with its implementation. The configuration of plan includes almost all the prerequisites of an OpenCL program for performing an FFT of specific type. The plan constructs based on their occurrence are highlighted in the following.

- **Transform Properties**
  At the start of the plan preparation, the transform properties are set. These include the transform type (Complex-to-Complex, Real-to-Complex etc), direction (forward DFT, inverse DFT), dimension size (1D, 2D or 3D) and transform precision (single precision or double precision e.g. C2C or Z2Z for complex-to-complex transforms respectively).

- **Decide Algorithm**
  Factorize the input size $N$ in order to decide the algorithm type to be invoked for the FFT i.e. a base radix Cooley-Tukey (radix-2, 3, 4 etc) or mixed radix approach to FFT.

- **Data Reshape**
  Some higher dimension transformation based on the input data hierarchy may
require initial reshaping of the input prior to any transformation being applied for improving performance. For example in batch execution or 2D transformation the initial data is reshaped (e.g. matrix transpose) for better memory access (e.g. memory coalescing).

- **Kernel setup and Memory allocations**
  
  Based on the transform properties and algorithm type, appropriate kernels are created and loaded with required arguments. The kernels include *bit reversal*, *swap*, *transpose*, *radix* and other auxiliary kernels required during the transformation.

  The reason why *plan* is separated from the actual FFT execution for the purpose to be created once and reused again and again. This reduces time in cases where an FFT of same properties and same data is required multiple times and hence an already baked *plan* is simply re-used (in a loop or as desired) for multiple executions. The *plan* setup is shown in the following figure.

4.3.1.3 **Execute**

The *Exec* module based on transform dimension (1D,2D or 3D) takes a created *framework* and baked *plan* for actual execution of the FFT. This is where the FFT is performed on the GPU device. The *Exec* module maps the FFT model (created and configured during the *plan*) to its execution. All the *kernels* in the baked *plan* are executed in here. After successful execution the transformed data (result/output) are read from the output buffers into in-place or out-of-place memory arrays. The *Exec* module is shown in the following figure.

4.3.1.4 **Destroy**

The *Destroy* routine frees all the resources being utilized in the FFT computation. This include the data structures in framework, memory input-output buffers, kernel objects etc, in the *plan* and *exec* code blocks.

4.3.2 **Cooley-Tukey Radices as OpenCL Kernels**

The Cooley-Tukey basic radices are implemented as *OpenCL kernels*. For each radix type a separate kernel code has been written. *ToPe-FFT* supports basic radices
Figure 4.8: Design overview of ToPeFFT. The first phase involves preparing the multiple-GPU framework (Fig. 4.7), followed by the creation of a plan. This plan chooses the algorithms for performing the transforms in accordance with the input data. The transforms are performed by calls to Exec functions. For 1D the dotted lines follow the path for a mixed radix with 2 factors. Whereas the solid lines is the execution path for base radices. Performing 3D transforms need an additional step to the 2D transforms as shown by the dotted lines. Finally a Destroy function frees all the allocated objects created in the previous steps.

2,3,4,5,6,7,8,10 and mixed radices for a composite given size $N$ of two factors given that each factor is a power of any of the mentioned basic radices. An example radix-2 and radix-5 kernel code is presented in the following. The same is followed for coding other radices (see source code freely available at [57]).

### 4.3.2.1 Radix-2 Implementation

The kernel code for the radix-2 decimation-in-time FFT discussed in is shown in Algo. 5. Here, the input parameters to the kernel are the data to be transformed, residing in the global memory pointer $X$ and the stage $s$, where $\{s\in \mathbb{N}|1 \leq s \leq \log_2 N\}$. The transformation process is completed by calling the kernel for each of the $s$ stages. All kernel executions are mutually exclusive. Since the basic execution block is a butterfly operation, the total number of work-items is equivalent to the total butterflies in a given stage, i.e., $N/2$. Inside the kernel, each butterfly based work-item identifies it’s position amongst other butterflies using an indexing scheme which we identify as a butterfly clip. This indexing is an extension to that obtained
through the OpenCL built-in calls such as `get_global_id(0)` or `get_global_size(0)` and is calculated in the preamble in lines 4-8. This is followed by fetching the data points $A(n)$ and $B(n)$ to private registers. The rest of the lines perform the actual computation of the butterfly as given in (4.17). The results from the calculations are placed back into global memory, which is identified by the global address space specifier. All operations are vectorized. Since the kernel assumes a scrambled input signal, the execution is preceded by a separate call to a bit-reversal kernel.

```c
typedef double2 double;
__kernel void radix2(__global double *X, int s)
{
    // Preamble (Identifying Butterfly)
    int id = get_global_id(0), k;
    k = id % pow(2,s-1);
    int2 clip;
    clip.x = k+2*id;
    clip.y = clip.x + pow(2,s-1);

    // Loading Registers
    double A = X[clip.x]; // A(n)
    double B = X[clip.y]; // B(n)
    int N = get_global_size(0)*2;
    double W = getTwiddle(k,N); // $W_k^N$
    double T = complex_mul(B,W);

    // Radix 2 Butterfly
    X[clip.x] = complex_add(A,T);
    X[clip.y] = complex_sub(A,T);
}
```

**Algorithm 5:** Example implementation of a Radix-2 based OpenCL kernel using Global Memory, taking as input the scrambled complex vector data points $X$ and the stage $s$, where $s = 1 \ldots \log_2 N$. The transform will be completed after calling the routine $\log_2 N$ times.
4.3.2.2 Radix-5 Implementation

For higher radices, the DFT can be decomposed in a similar way to that of radix-2. For example, a radix-5 decomposition would be given as:

\[
X(k) = \sum_{n=0}^{N-1} \sum_{l=0}^{4} x(5n + l) W_N^{(5n+l)k} \tag{4.28}
\]

Here, the inner summation would represent the butterfly operation that would be assigned to a work-item. This can also be re-written as follows, for convenience:

\[
X(k) = A(k) + W_5^k B(k) + W_5^{2k} C(k) + \cdots + W_5^{4k} E(k)
\]

The resulting butterfly will consist of five inputs and outputs, which for a periodicity of \(N/5\), would be identified as \(k, k+N/5, \ldots, k+4N/5\). The extra-coefficients for each period can thus be represented in a matrix form as:

\[
W^* = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
1 & W_5^1 & W_5^2 & W_5^{-2} & W_5^{-1} \\
1 & W_5^2 & W_5^4 & W_5^{-4} & W_5^{-2} \\
1 & W_5^{-2} & W_5^{-4} & W_5^4 & W_5^2 \\
1 & W_5^{-1} & W_5^{-2} & W_5^2 & W_5^1
\end{bmatrix} \tag{4.29}
\]

These extra twiddle co-efficients are precomputed for all radix algorithms and retrieved during compilation time. Thus the radix-5 butterfly in matrix form will be given as:

\[
\begin{bmatrix}
X(k) \\
X(k+N/5) \\
X(k+2N/5) \\
X(k+3N/5) \\
X(k+4N/5)
\end{bmatrix} = W^* \begin{bmatrix}
A(k) \\
B(k) \\
C(k) \\
D(k) \\
E(k)
\end{bmatrix} \tag{4.30}
\]

An example OpenCL implementation for the radix-5 is given in Algo. 6. Here, as in Algo. 5, the kernel takes as input the data residing in global memory and the stage number \(s\) as input parameters. In this case, \(\{s \in \mathbb{N} | 1 \leq s < \log_5 N\}\). The first
part of the kernel is concerned with identifying the positions of the radix-5 butterfly. This is followed by loading the registers from the identified positions. The radix-5 butterfly based on Eq. 4.30 is performed in the end.

```c
typedef double2 double;

__kernel void radix5(__global double *X, int s)
{
    // Preamble (Identifying Butterfly)
    int id = get_global_id(0), k;
    k = id % pow(5,s-1);
    int origin = k+5*id;
    int4 clip = (int4)(1,2,3,4);
    clip = origin + clip*pow(5,s-1);

    // Loading Registers
    int N = pow(5,s);
    double W; // Given as W
    double A = X[origin];
    W = getTwiddle(k,N); // W\k_N
    double B = complex_mul(X[clip.x],W);
    W = getTwiddle(2*k,N); // W^{2k}_N
    double C = complex_mul(X[clip.y],W);
    W = getTwiddle(3*k,N); // W^{3k}_N
    double D = complex_mul(X[clip.z],W);
    W = getTwiddle(4*k,N); // W^{4k}_N
    double E = complex_mul(X[clip.w],W);

    // Radix 5 Butterfly
    X[origin]=//A + B + C + D + E;
    X[clip.x]=//A + BW + CW^2 + DW^2 + EW^1
    X[clip.y]=//A + BW^1 + CW + DW^4 + EW^4
    X[clip.z]=//A + BW^2 + CW^4 + DW^4 + EW^2
    X[clip.w]=//A + BW^4 + CW^2 + DW^2 + EW^1
}
```

**Algorithm 6:** Example implementation of a Radix-5 based OpenCL kernel using Global Memory, taking as input the scrambled complex vector data points X and the stage s, where s = 1…log_5 N. The transform will be completed after calling the routine log_5 N times.
4.3.3 Load balancing on multiple GPUs

In multiple GPU simulations the given input load can be divided either statically or dynamically in real time systems [43, 15, 27]. In ToPe-FFT multi-GPU implementation, as the input transform length is known in advance therefore the load is statically distributed among the available devices (fetched during the framework phase) in the system. When the supported GPU devices are of same capabilities then the load is evenly distributed as shown in algorithm 7. The load balancing results for different transform sizes with four GPUs arranged on two physical cards, each one NVIDIA GTX-295, are shown in figure 4.16.

4.3.4 Usage and Interfacing

ToPe-FFT library routines can be integrated with C based code using Algo. 8 as an example. The interface follows a four-part approach. The first part is used to prepare a framework for enabling CPU-GPU interaction. It follows the OpenCL programming model outlined in Sec. 2.4.3.1 and is illustrated in Fig. 4.7. The second part uses an an initializing routine (plan) for the FFT computation. This involves selection of an optimum FFT algorithm, running the factorization engine, calculation of bit permutation indices, and pre-computation of the twiddle factors, if required. Finally, the actual calls to the computation of the FFT butterflies can be made using one of the Exec set of routines. This multi-part structure is necessary for cases where the same DFT is performed a number of times, as it avoid repeated reconstruction of the framework and plan. The final step involves a destroy routine which essentially frees up system resources. The overall working structure of these routines is shown in Fig. 4.8.

4.4 1D FFT Results and Comparisons

The time comparison of an FFT implementation is generally broken down to memory transfer times, pre-kernel times (a.k.a plan-time such as bit-reversal, transposition and data swap) and the actual running time of the FFT kernel for which the data is already transfered to GPU global memory (excluding any memory transfer times to/from the GPU). The reported times include memory transfer times to/from GPU,
static_LB(input_size, radix, dev_available){
    dev_required = dev_available;
    int g = input_size/radix;
    if(input_size == radix)
        dev_required = 1;
    else{
        while(g % dev_required !=0 )
            dev_required--;
    }
    global_size = input_size/(radix*dev_required);
    if global_size < opt_local
        local_size = global_size;
    else
        local_size = opt_local;
}

Algorithm 7: Multi-GPU Load Balancing. The inputs to the algorithm are the transform length, CT radix and the number of GPU devices present in the system. Lines 3-11 find the number of GPU devices for even distribution of the load. Lines 13-17 find global and local sizes for the respective butterfly NDRange kernel executions. A separate command_queue is associated for each GPU in the list as represented dev_required variable. As an example for input size of 262144 and radix 8 utilizing 4 GPUs, the global_size is 8192 and optimized local_size is 64. The opt_local in the algorithm depends on the input data type (double or float), compute capability of the device, radix and available shared and global memory in the device as the common optimization approach [56].
```c
#include <topefft.h>

int main() {
    struct topeFFT framework;
    topeFFTInit(&framework);

    struct topePlan1D plan;
    tope1DPlanInit(&framework, &plan, N, C2C, data);

    tope1DExec( &framework, &plan, data, FORWARD);

    tope1DDestroy( &framework, &plan);
}
```

Algorithm 8: Integration interface with C code, showing the three part structure. The first part enables a framework for the CPU-GPU interaction. The second part constructs a plan for performing the FFT. This may include selection of an FFT algorithm, preparation of bit-reversal indices, etc. These are finally followed by the actual calls to the FFT butterfly. The second part constructs a plan for performing the FFT. This may include selection of an FFT algorithm, preparation of bit-reversal indices, etc. This is followed by the actual calls to the FFT butterflies. The last step involves freeing up system resources using a destroy routine.
plan-time and the FFT kernel execution times denoted by case2. All times are measured using standard OpenCL profiling commands shown in 9.

```c
cl_event event;
cl_ulong timeStart, timeEnd;
// command queue with profiling enabled flag set
clCreateCommandQueue(..., CL_QUEUE_PROFILING_ENABLE,NULL);
clEnqueueNDRangeKernel(..., &event);
cFinish(command_queue[dev_no]);
// wait for all events to finish
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
    sizeof(cl_ulong),&timeStart,NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong), &timeEnd,NULL);
cl_ulong timeNanoSeconds = timeStart-timeEnd;
```

**Algorithm 9:** OpenCL standard method of measuring kernel execution time in nano-seconds. See chapter 5 of [60] for more details.

![Figure 4.9: Time comparison of radix-3 for float and double data types](image)

### 4.4.1 SpeedUp against FFTW and cuFFT

In the following (figures 4.13, 4.14, 4.15) the speedup of ToPe-FFT is shown against FFTW and cuFFT for different complex-to-complex input sizes evaluated by the set of base radices to mixed-radix FFT algorithms discussed above.
4 – GPU Implementation of Fast Fourier Transforms: ToPe-FFT

Figure 4.10: Time comparison of radices 5 and 6 for different sizes of float data type

Figure 4.11: Time comparison of radices 7 and 8 for different sizes of float data type

4.5 Higher Dimension FFTs

Higher dimension transformations like 2D and 3D use the same design principle as mentioned in figure 4.8 where the corresponding FFT routines are performed with the same execution flow as mentioned in interface 8. Higher dimension input perform the same 1D FFT routines multiple times. In this context the 2D explanation applies to 3D FFT with the z-axis added as third dimension. As an example the two dimensional FFT of the input data, arranged in a matrix form, is performed by...
Figure 4.12: Time comparison of radix 10 (left) on GTX-260 and all basic radices (right) for different sizes on different GPU devices. As shown in the right figure, ToPe performance improves as the device capabilities (number of cores, caches, read-write access time etc) enhance. The running time (both kernel and memory transfer times) on both GTX-260 and GTX-295 for small sizes is in close agreement. For higher sizes GTX-295 (contains 4 GPUs, two on each board) performance improves as for higher sizes multiple GPUs are used where FFT computation is carried out on multiple devices (discussed later in the chapter).

Figure 4.13: Speedup of ToPe-FFT vs FFTW and cuFFT by considering kernel time, memory transfer time and/or plan times of the corresponding FFT library.

taking the 1D FFT along columns followed by 1D FFT along rows as highlighted
Figure 4.14: Speedup of ToPe-FFT vs FFTW and cuFFT by considering kernel time, memory transfer time and/or plan times of the corresponding FFT library.

Figure 4.15: Speedup of ToPe-FFT vs FFTW and cuFFT on Kepler k20 (left) and Quadro 6000 (right) for case-2 (Kernel + Memory Transfer Time) for different input sizes in float.

in figure 4.17. The matrix based row-column approach suffers from long-strides memory access while performing the FFT along the columns, and is very hard to have coalesced access in this case. The FFT along columns suffers from spatial locality which increases with the number of columns of the matrix. For coalescing the given two dimensional data is reshaped for the purpose to avoid transposition and the long stride in the column FFT. As an example the 2D FFT of $512 \times 1024$
4 – GPU Implementation of Fast Fourier Transforms: ToPe-FFT

Figure 4.16: Speedup of ToPe-FFT vs FFTW and cuFFT for multiple GPUs where load on each GPU is evenly distributed.

size matrix is performed by 1D FFT of size \( N_y = 1024 \) followed by 1D FFT along \( N_x = 512 \). The time break down of ToPe-FFT for 2D transformation is shown in the following table.

<table>
<thead>
<tr>
<th>Size</th>
<th>Pre-Kernel Y</th>
<th>Pre-Kernel X</th>
<th>FFT-Y</th>
<th>FFT-X</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 × 1024</td>
<td>0.00034</td>
<td>0.000485</td>
<td>0.006605</td>
<td>0.005307</td>
<td>0.012737</td>
</tr>
<tr>
<td>8192 × 512</td>
<td>0.002602</td>
<td>0.003594</td>
<td>0.076934</td>
<td>0.080518</td>
<td>0.163648</td>
</tr>
<tr>
<td>8192 × 1024</td>
<td>0.005297</td>
<td>0.007214</td>
<td>0.162497</td>
<td>0.160215</td>
<td>0.335223</td>
</tr>
<tr>
<td>1000 × 100</td>
<td>0.000143</td>
<td>0.000111</td>
<td>0.001266</td>
<td>0.002237</td>
<td>0.003756</td>
</tr>
<tr>
<td>10000 × 512</td>
<td>0.003072</td>
<td>0.004078</td>
<td>0.076036</td>
<td>0.066333</td>
<td>0.149519</td>
</tr>
</tbody>
</table>

Table 4.1: ToPe-FFT timings on GTX-260 with complex-to-complex transform data represented in double precision. Size is the transform length in 2D. \( Y \) represent number of columns and \( X \) represents number of rows. Pre-Kernel is the time taken in reshaping data across each axis. The reshape includes time taken by generating bit sequences (bit-reversal kernel) and swapping of data based on generated bit sequences (swap-kernel). \( FFT - Y \) and \( FFT - X \) are the FFT butterfly kernel times. Each butterfly kernel (along \( X \) or \( Y \) dimension) is invoked \( p \) (in general) times where \( p_x = \log(N_x)/\log(r_x) \) or \( p_y = \log(N_y)/\log(r_y) \) given that \( N_x = r_x^{p_x} \) and similarly for \( N_y = r_y^{p_y} \). Note that \( r_x \) and \( r_y \) are the respective base radices for \( X \) and \( Y \) length sequences respectively. All times are measured using OpenCL standard method for measuring kernel execution times and reported here in seconds.
Input A → FFT along Columns → FFT along Rows → Output A

Figure 4.17: 2D FFT by the Row-and-Column Approach

Figure 4.18: 2D FFT running time of ToPe-FFT, FFTW and cuFFT are shown for varying input sizes $N = Nx \times Ny$. In the (left) $Nx = 512$ and (right) $Nx = 2048$ are fixed with varying $Ny$ (for even radices). The cuFFT 2D FFT gives correct results only when both dimensions are equal i.e. $Nx = Ny$ as highlighted by the filled marker on the dotted line representing cuFFT times.
Figure 4.19: Speedup of ToPe-FFT against FFTW for 2D FFT is shown for varying input sizes $N = N_x \times N_y$. In this case $N_x = 1000$ is fixed with varying $N_y$ for even (left) and odd (right) radices.
Chapter 5

Micromagnetics: GPU Acceleration of Magnetostatic Field Computation

5.1 Introduction

In this chapter an OpenCL based Fast Fourier Transform library optimized for exploiting symmetries in computation of the magnetostatic field, is introduced. Recent developments in the Micromagnetic simulator landscape suggests that code developers are increasingly writing specialized code for acceleration hardware, such as Graphic Processing Units, using proprietary tools. As such, this nature of code development limits the simulators to run on specific hardware only. Our implementation follows the OpenCL standard which permits compliant code to run on different types of hardware platforms. The library provides support for transforms of near arbitrary input lengths in multiple-dimensions. Furthermore, it supports distribution of the transform across multiple GPU’s. Based on our benchmark results, we report a speed-up of $3\times$ against the equivalent cuFFT library for GPU’s, and a speed-up of $48\times$ against the CPU based FFTW library. Micromagnetic simulations play a key role in understanding magnetization dynamics of magnetic materials. In some newly emerged areas like Magnonics; involving mesoscopic sized devices that utilize spin dynamics integrated with microwave circuits [41], the usage of simulations to understand underlying behaviour is imperative. This is also the case in the study of magnetization reversal processes in permanent magnets [31], or vortex core switching behaviour in high frequency magnetic fields [66]. In both cases,
the simulations address large problem sizes involving computations across millions of cells due to the small scales under consideration (both time and spatial). Running simulations on such a large scale is possible with the right combination, and availability of, processing speeds, massive parallelism, memory storage, and memory bandwidth. Computation of magnetostatic fields in Micromagnetic simulations is also a well known source of computational bottleneck, primarily due to its nature of long-range interactions. With these challenges, developing 3D large scale Micromagnetic simulation software is a very difficult process for many developers. Over the past few years, some new open-source and commercial simulator packages have been introduced which cater for the intense performance requirements [46, 70]. These requirements are predominantly met by the off-the-shelf availability of Graphic Cards, or Graphic Processing Units (GPU’s), which provide support for massive parallelism. Graphic Cards are throughput oriented hardware, that until recently were intended for rendering of computer graphics only, but are now increasingly used in many areas of scientific and general purpose computing. The most recent card, for example, the NVIDIA Tesla K40 with 2,880 computing cores offers $1.43 \times 10^{12}$ double-precision floating point operations per second, compared to $0.1 \times 10^{12}$ on a 4 core Intel Core-i7-965 XE, i.e., almost $13\times$ more throughput\(^1\). To take advantage of this extra throughput, developers code their programs specifically using programming languages and libraries specified by the hardware vendor. This restriction implies that simulators would be restricted to devices supplied by a single vendor only. Our focus is to remove this restriction by choosing an open standard called OpenCL.

This chapter discusses the integration of our multi-dimensional FFT library [57], including optimizations for accelerating magnetostatic field computations. This library, distributed under an open-source license, is the main contribution of this paper. The novelty is it’s implementation under the OpenCL specification, which to the best of our knowledge, has not been implemented before. Our benchmark and profiling results report our library to be $48\times$ faster than a single CPU based FFTW code and $3\times$ faster than NVIDIA’s cuFFT library[1]. The results are averaged for various input sizes involving complex-to-complex transforms.

\(^1\)NVIDIA Tesla K40 Specification
5.2 Magnetostatic Field Computation Model

The general approach to computing the magnetostatic field involves discretization of continuous magnetization into a finite number of computational cells. As such, the magnetization can be represented as a discrete distribution $m(r')$, where $r'$ is the position vector of each cell. The magnetostatic field at a given position is:

$$h(r) = -\sum_{r'}^n N(r - r') \cdot m(r'), \quad (5.1)$$

where $r'$ is the position vector of the source, $r$ is the position vector of the field, $n$ is the total number of cells, and $N$ is the demagnetizing tensor, defined as a function depending upon the relative position between the source and field cells $R = r - r'$ [39]. The figure 5.1 shows the decomposition of a magnetized body into cubic cells with relative distance values denoting the corresponding demagnetization factors as the components of the position vector $r = r_x \hat{i} + r_y \hat{j} + r_z \hat{k}$ in 3-dimension Cartesian coordinate system. The position difference values are combined into a matrix notation, called tensor matrix, as shown in the following.

$$N = \begin{bmatrix}
N_{xx'} & N_{xy'} & N_{xz'} \\
N_{yx'} & N_{yy'} & N_{yz'} \\
N_{zx'} & N_{zy'} & N_{zz'}
\end{bmatrix} \quad (5.2)$$

The elements forming this tensor matrix has various odd-even symmetries. For example the diagonal elements, $N_{xx'}$, $N_{yy'}$ and $N_{zz'}$ have pure even symmetry where as the non diagonal elements, $N_{xy'}$, $N_{xz'}$ and $N_{yz'}$ contain mixed odd/even symmetries. Most Micromagnetic simulators compute (5.1) by employing the Finite Difference method [49] for performance reasons. Here, it is assumed that magnetization within a cell is uniform, and both the cell and magnetization volume are regular shaped hexahedrons.

The algorithmic complexity for computing (5.1) directly is of order $O(n^2)$. As a result, computing a problem size with a large $n$ becomes computationally prohibitive. However, since $N$ only depends on the differences of integer indices, (5.1) can be treated as a discrete convolution. Many solvers exploit this property using the Convolution Theorem [11, 2, 69, 45], using which, the operation can be reduced
Figure 5.1: Demagnetizing Tensor representation (adopted from [39]) of a magnetized body discretized into cuboid cells. Here $S$ and $F$ represents the source and field cells (a.k.a points or elements) respectively.

Figure 5.2: Ferromagnetic body discretized by the FD scheme into $n$ number of total cells where $n = N_x \times N_y \times N_z$. Given dimensions are doubled and zero-padded along each axis as given in [10]. This not only removes aliasing effects but also produces a periodic signal in Fourier space. The white spaces represent the padded regions in this case.

To a point-wise multiplication in Fourier space. This theorem can be applied if the discrete $m$ is periodically repeated in an infinite space. In cases, where this is not so, $m$ is reconstructed using a zero-padding technique such that the overall length of $m$ is doubled along each Cartesian axis, resulting in an eight-fold increase in $n$ (See Fig. 5.2. The overall complexity is still reducible to order $O(n \log n)$. The major contribution to this complexity is the Discrete Fourier Transform (DFT), which converts a finite sequence of samples from time or spatial domain to Frequency domain and vice-versa. Thus, the application of the DFT to (5.1) gives:

$$\tilde{H} = -\tilde{N} \cdot \tilde{M},$$  \hspace{1cm} (5.3)
where the overhead tilde now represents the component in Fourier space. Considering a 2D case, $\tilde{M}$ would be:

$$\tilde{M}(k'_x, k'_y) = \sum_{r'_x=0}^{2N_x-1} \sum_{r'_y=0}^{2N_y-1} \mathbf{m}(r'_x, r'_y) \exp\left[\frac{-2\pi j r'_x k'_x}{2N_x}\right] \exp\left[\frac{-2\pi j r'_y k'_y}{2N_y}\right], \quad (5.4)$$

where the exponential part is known as the \textit{twiddle factor}, $r'$ is the spatial position vector, $k'$ is the frequency space position vector, and $N_x, N_y$ are the total number of cells, extended to twice their size due to zero-padding. The case for computing $\tilde{N}$ is slightly different, where although the size is also doubled in each Cartesian-axis, it is not filled with zeros, rather odd-even symmetries of the physical data are used instead [39].

Equation (5.4) is computed as a multi-dimensional transform, where the transform is first carried over the inner summation representing rows (i.e., the $x$-axis), followed by the outer summation representing the columns (i.e., the $y$-axis) [13]. The transformed matrix $\tilde{M}$ is then multiplied point-wise with the tensor coefficients $\tilde{N}$ in (5.3). The obtained field $\tilde{H}$ is still in the Fourier space and is then transformed back using an inverse Fourier transform. It should be noted, however, (5.4) is in itself a multiplication of an input and twiddle-factor matrices giving a computational complexity of $O(n^2)$. A faster approach devised by Cooley and Tukey (CT) reduces this complexity to $O(n \log n)$ in a landmark paper in [17]. Their approach, known as the \textit{Fast Fourier Transform} essentially decomposes an $N$-point DFT into $N/r$ smaller DFT’s, each of length $r$. Popular FFT Libraries like FFTW [25] for CPU’s, and cuFFT [1] for GPU’s are based on the CT Model. Our DFT implementation also follows similar convention.

$$\tilde{M}(k'_x, k'_y, k'_z) = \sum_{r'_z=0}^{2N_z-1} \sum_{r'_y=0}^{2N_y-1} \sum_{r'_x=0}^{2N_x-1} \mathbf{m}(r'_x, r'_y, r'_z) \exp\left[\frac{-2\pi j r'_x k'_x}{2N_x}\right] \exp\left[\frac{-2\pi j r'_y k'_y}{2N_y}\right] \exp\left[\frac{-2\pi j r'_z k'_z}{2N_z}\right], \quad (5.5)$$

For a 3D case, (5.4) can be rewritten to introduce a summation over components $r'_z$ of $\mathbf{m}$, followed by a 3rd set of twiddle factors as shown in (5.5). The magnetostatic field in component form for the 3-dimension discretized body is given in the following (5.6).
\[
\begin{align*}
\tilde{H}_x &= \tilde{N}_{xx} \tilde{M}_x + \tilde{N}_{xy} \tilde{M}_y + \tilde{N}_{xz} \tilde{M}_z \\
\tilde{H}_y &= \tilde{N}_{yx} \tilde{M}_x + \tilde{N}_{yy} \tilde{M}_y + \tilde{N}_{yz} \tilde{M}_z \\
\tilde{H}_z &= \tilde{N}_{zx} \tilde{M}_x + \tilde{N}_{zy} \tilde{M}_y + \tilde{N}_{zz} \tilde{M}_z
\end{align*}
\] (5.6)

The computation cost of the entire transform for the 3D magnetostatic field, would be given as \(8N_xN_yN_z \log(8N_xN_yN_z)\). From figure. 5.2, it is clear that a significant proportion of computations are performed over zero-filled values. Since the FFT of zero input is zero, all such FFT’s can be stripped, thus reducing the computational cost to \([2N_yN_z \log (2N_x)] + [(N_x + 1) 2N_zN_y \log (2N_y)] + [(N_x + 1) 4N_yN_z \log (2N_z)]\) (See Fig. 5.3). Here, the first set corresponds to \(N_yN_z\) transforms (on physical size) of length \(2N_x\), the second set corresponds to \(N_z\ (N_x + 1)\) transforms (also on physical size) of length \(2N_y\), and so on. The +1 for \(N_x\) is contributed due to the conjugate-symmetry property of transforms due to real-valued input. Finally, once the FFT on all cartexian-axes are completed, we obtain a partially filled \(\tilde{M}\). The remaining part is simply the conjugate symmetrical values which can be obtained using a simple copy: \(\tilde{M}(-k_x, -k_y, -k_z) = \tilde{M}^*(k_x, k_y, k_z)\) Evaluating the ratio of the total transform, inclusive of the FFT, and the reduced transform gives a maximum saving of 66% in the computational cost. A similar approach can be taken in computing the FFT of the demagnetizing tensor \(\tilde{N}\). However, recall that the tensor is not zero-padded but contains odd/even symmetries. These can be exploited in the same mannerism.

![Figure 5.3: Symmetry in m due to zero-padding technique](image_url)
5.3 Magnetostatic Field Solver on GPU

The Micromagnetic simulation for the computation of magnetostatic field is carried out in the following main steps. The execution flow of the Field solver is shown in fig 5.4.

- At the start the geometry dependent tensor components and the magnetization field values are loaded from respective files stored on the secondary storage. For a given geometry and mesh size, the corresponding tensor values are either obtained from OOMMF `demag.cc` class based on [53] or from simplified formulation as listed in [39].

- **Init**: In the initialization phase the type of OpenCL device (CPU/GPU) or devices (in case of multiple GPUs) are selected for the simulation. The required variables and data structures are initialized both for the host and device portions of the code.

- **Plan**: The necessary Plan is prepared for the parallel portion of the code. This includes twiddle factor pre-computation, data reshape (bit reversal) for the FFT algorithm. This step is performed in parallel on GPU both for the tensor and magnetization data as both are later transformed in Fourier domain.

- **Transformation in Fourier Space**: The Fourier transform treatment for the convolution equation [eq:discretized magnetostatic field] require both tensor and magnetization to be transformed in Fourier space. Exploiting the symmetries in the tensor matrix, only the non-symmetric coefficients are transformed in to Fourier space using our own optimized FFT library in 3D. Similarly the three components of the magnetization vector field are transformed in Fourier space. The order of transformations ($\tilde{N}$) and ($\tilde{M}$) is not important.

- **Pointwise Multiplication**: After transforming both tensor and magnetization in Fourier space, they are then pointwise multiplied as a required step for the convolution theorem. The point wise multiplication itself is highly parallel and GPU would have been a better choice. But we instead opt for performing this step on the CPU. The reason that for larger and finer mesh size geometry (hence larger number of cells, $n$) the memory transfer time between GPU and
CPU for all the six-components of tensor matrix and the magnetization components dominates the computation time. Another constraint is that available memory on GPU is very limited (only few GBs). The scarce GPU memory thus can not accommodate larger tensor and magnetization component arrays as the number of cells ($n$) increase. The output of this stage is the required magnetostatic field in Fourier space.

- **Inverse FFT transformation**: The Fourier domain magnetostatic field values (of all the three components) are converted back to time domain by inverse FFT. This step concludes the four step procedure of the convolution theorem.

### 5.4 Results and Comparisons

FFT based magnetostatic field computation is widely adopted in many Micromagnetic solvers like MuMax [70], GPMagnet [46], TetraMag [37], OOMMF [21] etc. These simulation tools use generic FFT libraries like cuFFT [1] on GPU or the widely used FFTW library on CPU. These generic and close source FFT libraries are not very useful in utilizing special features of the FFT based magnetostatic field computation, like avoiding the FFTs of the zero-filled regions and other tensor symmetries. A specialized FFT library was thus necessary for accelerating over all magnetostatic field computation. In the following, performance comparison of our specialized FFT based magnetostatic field computation is shown with OOMMF which is state-of-the-art Micromagnetic simulator using FFTW for Fourier transformation.

#### 5.4.1 Experimental Setup

A Permalloy body of 512 nm $\times$ 125 nm $\times$ 3 nm discretized into number of cells with dimensions of 0.9765625 nm $\times$ 0.9765625 nm $\times$ 1.5 nm. This gives mesh size of $n = 512 \times 128 \times 2$, number of cells on x, y and z-axis respectively. An initial $S$-state magnetization (see figure 5.5) configuration is considered for the $n$ cells with saturation magnetization $M_s = 8.0 \times 10^5$ A/m. These magnetization and the tensor values are pre-computed by simulating $\mu$-mag standard problem no.4 in OOMMF.
Figure 5.4: Flow chart of our GPU Micromagnetic field solver. The doubled solid rectangle components show the blocks of code performed in parallel on the GPU. The dash line double rectangle represent the point wise multiplication performed by multiple threads on CPU. This could also be performed on GPU side but for larger $n$ the total memory transfer times (for both $\mathbf{N}$ and $\mathbf{M}$ respective components) over the slow PCI dominates the total computation time.

By simulating the same geometry and cell size with other parameters kept constant, we report a speed-up factor of approximately 4x against OOMMF for single precision values. The time information about our code is computed using OpenCL.
standard profiling functions as highlighted in chapter 3 section 9. The timing information of various FFT routines in the OOMMF simulator are calculated using timing-constructs in the OOMMF `demag.cc` class file.

### 5.4.2 Validation and Performance

In this section, we verify the accuracy of our method by comparing the computed magnetostatic field with equivalent results from OOMMF using the \( \mu \)-mag standard problem 4.

In the following table (tab 5.1), we report the mean absolute \( \epsilon_{\text{mean}} \), the maximum absolute \( \epsilon_{\text{max}} \), and the mean relative error for different cell sizes.

<table>
<thead>
<tr>
<th>Computation Cells</th>
<th>( \epsilon_{\text{mean}} )</th>
<th>( \epsilon_{\text{max}} )</th>
<th>( \eta_{\text{mean}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Million</td>
<td>( 7.59e - 11 )</td>
<td>( 3.52e - 10 )</td>
<td>( 1.01e - 14 )</td>
</tr>
<tr>
<td>2 Million</td>
<td>( 4.04e - 11 )</td>
<td>( 1.68e - 10 )</td>
<td>( 5.33e - 15 )</td>
</tr>
<tr>
<td>4 Million</td>
<td>( 7.56e - 11 )</td>
<td>( 3.65e - 10 )</td>
<td>( 9.96e - 15 )</td>
</tr>
<tr>
<td>8 Million</td>
<td>( 7.91e - 11 )</td>
<td>( 3.38e - 10 )</td>
<td>( 1.04e - 14 )</td>
</tr>
</tbody>
</table>

Table 5.1: Reported differences for magnetostatic field \( \mathbf{h} \) for various computational cells. The simulation is carried out using the \( \mu \)-mag standard problem 4 with an \( S \)-state as initial magnetization configuration. The differences are based on results from OOMMF \( \mathbf{h}' \) and our computed test simulation \( \mathbf{h} \). Here, \( \epsilon = |\mathbf{h} - \mathbf{h}'|_2 \) and \( \eta = \epsilon / |\mathbf{h}'| \).
Figure 5.6: Demagnetizing field computation time on GTX 260 for different time steps and various cells size. Reported time takes into account the time of FFT kernels (bit-reversal+FFT kernel time+memory transfer time to/from GPU) for all the components of $\mathbf{N}$ and $\mathbf{M}$ and also the IFFT of $\tilde{\mathbf{H}}$. It also accounts for the point-wise-multiplication for the components of $\tilde{\mathbf{N}}$ and $\tilde{\mathbf{M}}$ on CPU as highlighted in 5.4.
Figure 5.7: Demagnetizing field computation time on GTX 260 for different time steps and various cells size. Reported time takes into account total FFT computation time for all the components of $\mathbf{N}$ and $\mathbf{M}$ and also the IFFT of $\tilde{\mathbf{H}}$. It does not account for any time spent on CPU for point-wise multiplication.
Figure 5.8: Difference of Memory transfer time to/from GPU. It is depicted that memory transfer time over the slow PCIe card cuts the over all performance. Moreover synchronous memory transfer from GPU-to-CPU consumes many clock-cycles than CPU-to-GPU transfer time. This limits speedup of our GPU code over CPU based parallel OOMMF.
Figure 5.9: The speedup against parallel OOMMF (in this case utilizing 4-cores) is reported. The total time of our code takes into account all the FFT/IFFT times + Memory transfer times to/from GPU + point-wise-multiplication time on CPU. Considerable speedup (around 4x) is obtained for very finer mesh sizes (or larger geometry dimensions). The scarce GPU memory limits the total number of computation elements (Cells). In cases where GPU memory falls insufficient to accommodate total cells, the individual cell size should be adjusted with a penalty on accuracy of final magnetostatic field obtained.
Chapter 6

Micromagnetics: Time Stepping Scheme for the Integration of Landau-Lifshitz Equation

Micromagnetics deals with the study of magnetization behavior of certain magnetic materials at micro-meter length scale. The materials under this consideration, known as ferromagnetic materials, possess spontaneous magnetization without the effect of any external field. The property of spontaneous magnetization (net magnetic moment in the absence of an applied field) and the bistate characteristic of the ferromagnetic materials (under the influence of an applied magnetic field) make them ideal choice for magnetic recording media e.g. hard drive, floppy disks, magnetic stripe cards etc. A typical magnetic recording system, under required conditions are satisfied as briefly discussed in [51], stores around 100 Gbit/in$^2$. The concept of universal memory [4] is under development where magnetization alignment of elementary elements (also called cells) is used for storing information. The devices made up of such thin ferromagnetic layers, as a grid of elementary cells, are collectively named as magneto-resistive random access memory (MRAMs) devices. In both traditional hard disk type storage devices and the newly MRAMs, the magnetization-state of the underlying ferromagnetic thin film is connected to the bit-sequence of the information being stored.

In Micromagnetics the magnetization phenomenon is studied at various positions inside the ferromagnetic specimen at different time intervals. The Micromagnetics
theory combines the set of Maxwell equations with the time dependent equation of motion (see 6.2) for the investigation of spatial distribution of magnetization inside a ferromagnetic body. Consequently the elementary magnetic moments (typically represented by \( \mu \) or \( m \)) as continuous function of position and time. The different energy terms that govern magnetization dynamics inside a magnetized body are discussed in section 6.1. In section 6.2 the Landau-Lifshitz equation of motion is discussed. The state-of-the-art discretization schemes for the LL equation are discussed in 6.3. In section 6.4, a new and efficient time stepping technique is discussed. The accuracy, stability and efficiency of the new scheme is tested using \( \mu \)-mag standard problem no-4. The results of the proposed scheme are compared with OOMMF and are presented in section 6.5.

6.1 Energy Contributions Towards Magnetization Dynamics

Several experimental techniques could be used for observing distribution of magnetic moment distribution on the surface of the material. However for understanding of domain [72] formation, domain wall thickness, and for related energy and anisotropy influence on magnetization distribution[48] the Micromagnetics calculations are required. Let \( \mathbf{J}(\mathbf{r}) \) be the magnetic polarization, \( \mathbf{m} \) as the magnetic moment and \( V \) as the volume of the magnetic body, then the following basic equation holds.

\[
\mathbf{J}(\mathbf{r}) = \frac{\mathbf{m}}{V} \tag{6.1}
\]

\[
= \mu_0 \mathbf{M}(\mathbf{r}) \tag{6.2}
\]

Where \( \mu_0 \) is the magnetic permeability of the vacuum and \( \mathbf{M}(\mathbf{r}) \) is the vector magnetization field as a function of position across the whole body. Under uniform magnetization (\( \mathbf{M} = M_s \mathbf{m} \), were \( M_s \) is the saturation magnetization and \( \mathbf{m} \) is the unit vector describing direction of \( \mathbf{M} \)). In the metastable equilibrium as discussed in [12], magnitude of the magnetic polarization with respect to direction cosines (\( \beta(\mathbf{r}) \)), takes the form as:
\[ |J| = J_s(T) \quad (6.3) \]
\[ \mathbf{J} = \beta J_s \quad (6.4) \]

In ferromagnetic materials net magnetization is a function of position and time, \( \mathbf{M}(r,t) \), which varies in space and time based on various short and long range interactions [50]. Each interacting force component is assigned an energy term. The summation over all these energy terms for all elementary cells gives the Gibbs free energy \((E_g)\). The total energy (Gibbs free energy) is the sum of the following energies, which must be minimized to bring the system in equilibrium state magnetization. The different energy terms influencing net magnetization are:

- **Exchange Energy**: The exchange energy takes into account short range interactions among the spins (moments) of the cells in close vicinity (neighborhood) identified by Heisenberg [30]. This energy term favors parallel alignment of neighboring spins. Let \( S_i \) and \( S_j \) represents atomic spins at at positions \( r_i \) and \( r_j \) respectively. The exchange interaction of these atomic spins with angle \( \theta \) and \( r' \) separation distance, is given by the following Heisenberg hamiltonian form.

\[
H' = -2 \sum_{i,j} J(r_{ij}) S_i \cdot S_j, \quad \forall i \neq j \quad (6.5)
\]

When the spins, \( S \), are represented by magnetic moments \( \mathbf{M} = S(\frac{\mu_{Bohr}}{g}) \) the above quantum mechanical expression takes the form as:

\[
E_{ex} = -\frac{2}{V} \left( \frac{VM_s}{g\mu_{Bohr}} \right)^2 J_0 \sum_{i,j} m_i \cdot m_j, \quad \forall i \neq j \quad (6.6)
\]

Here \( g \) is the Lande’ factor, \( M_s \) is the saturation magnetization, \( \mu_{Bohr} \) represents Bohar magneton (magnetic moment of an electron), and \( m_k \) are the magnetic moment in the \( k \) cell. The integrand \( J \) usually considers 6 neighbors in the neighbor set \( nn \). Here the interaction of a cell with itself is ignored. The exchange field, after simplification (see [67]), is given as:
\[ h_{\text{ex}} = l_{\text{ex}}^2 \cdot \nabla^2 m \]  

(6.7)

Here \( l_{\text{ex}} \) is the exchange length, given by

\[ l_{\text{ex}} = \sqrt{\frac{2A}{\mu_0 M_s^2}} \]  

(6.8)

where \( A \) is exchange constant and \( \mu_0 \) permeability of free space. Minimization

this energy term is responsible for different domain regions inside the body. (check)

- **Anisotropy Energy**: The anisotropy energy takes into account the interactions of the spins with the crystal lattice. This energy, also called magneto crystalline energy, is responsible for direction of net magnetization along a certain axis, called easy axis, of the lattice (or body). Based on the the structure of the material, the anisotropy energy may be uni-axial, triaxial or cubic.

Let \( \theta \) be the angle between magnetization vector \( M \) and the easy axis of the lattice, then in the case of only one easy axis (uni-axial) the anisotropy energy is given by the following relation.

\[ E_{\text{anis}} = KV \sin^2 \theta \]  

(6.9)

Where \( K \) is the anisotropy constant (energy per unit volume) and \( V \) is the volume of the body. This energy is minimized when the both magnetization is aligned in the direction of easy axis (\( \theta = 0^\circ \) or \( 180^\circ \)). Triaxial and cubic anisotropy energy terms are realized by a similar equation with different direction cosines.

- **Zeeman Energy**: The interaction of the strong external applied field (\( h_a \)) with the magnetic moments give rise to Zeeman energy, \( E_{\text{zeeman}} \). The Zeeman energy term tends to rotate magnetization vector field \( M \) parallel to the applied field \( h_a \). Mathematically expressed as:

\[ \phi_a(\mathbf{r}) = -\mu_0 M_s m(\mathbf{r}) \cdot H_a \]  

(6.10)

Here \( \phi_a \) is the Zeeman energy density and \( \mu_0 \) permeability of vacuum.
• **Magnetostatic Energy**: Magnetostatic energy, also called demagnetization energy, arises from the small magnetic field produced by elementary moment that opposes the influential applied field. The magnetostatic energy $E_{demag}$ and hence the demagnetizing field $H_d$ favors anti alignment of the spins and is results into a long range (dipole-dipole) interaction by all cells of the ferromagnetic body. The demagnetization energy is expressed in the following.

$$E_{demag} = -\frac{1}{2} M_s^2 \int M(r)H_d(r)$$  \hspace{1cm} (6.11)

Here the $M_s$ is the saturation magnetization and both $M$ and $H_d$ representing magnetization and demagnetizing fields both as functions of positions. The following sets of equations based on Maxwell equations derives expression for $H_d$.

$$H_d = -\frac{M}{4\pi} \int dS \nabla dS' \frac{n}{|r - r'|}$$  \hspace{1cm} (6.12)

$$H_d = -\hat{N}M$$  \hspace{1cm} (6.13)

Here $\hat{N}$ called the demagnetization tensor matrix is the geometry dependent position values of the cell relative distances from each other.

### 6.2 Equation of Motion

Under the influence of an applied magnetic field, $h_a$, a torque is experienced by elementary magnetic moment ($\mu$). This torque is responsible for the direction alignment of the moment to that of the applied field which results in a precessional motion as highlighted in figure 6.1. The equation of moment under the influence of an applied is given by:

$$\frac{d\mu}{dt} = -\gamma \mu \times H$$  \hspace{1cm} (6.14)

$$\frac{d\vec{M}}{dt} = -\gamma \vec{M} \times \vec{H} - \frac{\gamma \alpha}{M_s} \vec{M} \times (\vec{M} \times \vec{H})$$  \hspace{1cm} (6.15)

The time evaluation of the magnetization behavior is studied using the well known Landau-Lifshitz (LL) equation.
6.3 Numerical Solution of the LL Equation

The integro-differential LLG equation of motion is numerically solved given the fact that full analytical solution of the LLG equation is very hard to obtain. Thus nonlinear LLG equation is numerically integrated for an approximate solution using a suitable time stepping technique which at the same time should be stable and must preserve geometrical properties of the LL dynamics as briefly discussed in [23]. Several numerical techniques have been used for the solution of LLG equation [7, 16, 28, 64].

A common approach in use, is the mid-point rule as a time stepping scheme for numerically solving the LLG equation. Apart from its simplicity in applying, the mid-point rule is unconditionally stable and holds the fundamental property of the LLG dynamics: the magnitude of magnetization must be preserved at each time step and spatial location. Another important property is the conservation of free energy of the system i.e. with no damping ($\alpha = 0$) the free energy of the system for all time steps should be the same (remain constant). The problem in using the mid-point rule as time integration scheme is its complexity in terms of large set of equations with high dependency in them. For better accuracy and hence for very small time steps, solving such a large inter coupled sets of nonlinear equations
require high computation time. The authors in [20] use an iterative quasi-Newton. In the semi-discretization techniques of numerically solving LL equation, the given equation is first discretized spatially either using finite difference or finite element method and then solved using time-stepping technique. A predictor/corrector (p/c) semi-analytical time stepping scheme is discussed in [68] where the $h_{\text{eff}}$ is computed twice per time step at each finite difference cell. Their time stepping schemes, when FFT based magnetostatic field computations are carried out, consumes $68N$ floats of memory, where $N$ is the total number of discretized FD cells.

### 6.4 Proposed Scheme

Consider the following normalized LL equation:

$$\frac{\partial \mathbf{m}}{\partial t} = -\mathbf{m} \times (h_{\text{eff}} + \alpha (\mathbf{m} \times h_{\text{eff}}))$$  \hspace{1cm} (6.16)

In the above equation, $\mathbf{m}$ is the magnetization vector field normalized by the saturation magnetization $M_s$, $h_{\text{eff}}$ is the effective field and $\alpha$ is the damping factor. As explained earlier, $h_{\text{eff}}$ is the sum of applied field $h_a$, exchange field $h_{\text{ex}}$, anisotropy field $h_{\text{an}}$ and the computation intensive magnetostatic field $h_m$ [3]. In these constituent forces, the most time consuming is the long range interactive $h_m$ with computational complexity of $O(n^2)$ with $n$ as total number of discretized cells. As discussed briefly in chapter 5, discrete convolution theorem is applied with a reduced complexity in the order of $O(n \log n)$. In the first phase of the semi-discretization, the $h_m$ value per cell for the whole magnetized region $\Omega$ is computed. This is followed by a time integration scheme to investigate magnetization dynamics in equation 6.16. Several time-integration schemes (Runge-Kutta, Adams-Bashforth, Crank-Nicholson etc see [19, 63] for more details) have been proposed in literature. We propose here a mixed midpoint Runge-Kutta like scheme for the integration of LLG equation. At the surface of the boundary $\mathbf{m}$ obeys $\frac{\partial \mathbf{m}}{\partial n} = 0$ where $n$ is the outward normal to the surface. The structure of the proposed time stepping technique is explained in the following five steps.

- **Step 1:** In the first step, the LL equation in 6.16 is discretized in using midpoint rule. Consider a finite cell $i$ of the spatial mesh, let $\mathbf{m}^k(i)$ is the magnetization at cell $i$ at instant of time $t^k$. The mid-point discretization of equation
6.16 yields the following equation.

\[ m^{k+1}(i) - m^k(i) = \tau/2 \left[ m^{k+1}(i) + m^k(i) \right] \times \mathcal{H} \left[ i, \frac{m^{k+1} + m^k}{2}, t^k + \tau/2 \right] \]  

(6.17)

Here \( m^k(i) \) is the magnetization at finite element or cell \( i \), \( \tau \) is the time-step interval and \( \mathcal{H} \) is a functional of cell, magnetization and time as given in the following.

\[ \mathcal{H}(i,m,t) = h_{\text{eff}}(i,m,t) + \alpha m(i,t) \times h_{\text{eff}}(i,m,t) \]

Given the known magnetization \( m^k \) at cell \( i \), the unknown magnetization \( m^{k+1} \) at the particular cell can be found by solving equation 6.17 iteratively. The iterative solution starts from an initial guess value for the magnetization. Traditionally in semi discretization techniques like in [20], equation 6.17 is solved using \( 3N \times 3N \) set of equations where \( N \) is the total number of discretized cell with distinct \( m, h_a \) and \( h_{\text{eff}} \) values per cell. This requires huge computation time for large \( N \). In our proposed scheme, we solve equation 6.17 by combining mid-point rule to the Runge-Kutta method where the at each location (cell) merely \( 3 \times 3 \) instead of \( 3N \times 3N \) equations are solved. In the first step equation (6.17) is solved for \( \mathcal{H} \) at pint \( m = m^k \) and at time instant \( t = t^k \). This leads the following equation:

\[ m_1^{k+1}(i) - m^k(i) = \tau/2 \left( m_1^{k+1}(i) + m^k(i) \right) \times \mathcal{H} \left[ i, m^k, t^k \right] \]  

(6.18)

- Step 2: In the second step the \( \mathcal{H} \) is computed at mid-point magnetization field \( \frac{m^k + m^{k+1}}{2} \) at time instant \( t^k + \tau/2 \). The following equation is thus obtained.

\[ m_2^{k+1}(i) - m^k(i) = \tau/2 \left( m_2^{k+1}(i) + m^k(i) \right) \times \mathcal{H} \left[ i, \frac{m_1^{k+1} + m^k}{2}, t^k + \tau/2 \right] \]  

(6.19)

- Step 3: Repeating the same procedure as done in step-2 leads the following

\[ m_3^{k+1}(i) - m^k(i) = \tau/2 \left( m_3^{k+1}(i) + m^k(i) \right) \times \mathcal{H} \left[ i, \frac{m_2^{k+1} + m^k}{2}, t^k + \tau/2 \right] \]  

(6.20)
• Step 4: At the fourth step, $H$ is computed at time $t^{k+1}$ using the previous approximation of magnetization $m^{k+1}_3$. Thus, the following equation is obtained:

$$m^{k+1}_4(i) - m^k(i) = \tau/2 \left( m^{k+1}_4(i) + m^k(i) \right) \times \mathcal{H}[i, m^{k+1}_3, t^{k+1}]$$ \hfill (6.21)

• Step 5: The weighted sum of the unknowns in above steps yield the best approximated magnetization value for $m^{k+1}_5$, as given in the following.

$$m^{k+1}_5 = m^{k+1}_1/6 + m^{k+1}_2/3 + m^{k+1}_3/3 + m^{k+1}_4/6$$

Putting all together, equation 6.18 is written as:

$$m^{k+1}(i) - m^k(i) = \tau/2 \left( m^{k+1}(i) + m^k(i) \right) \times \mathcal{H}[i, (m^{k+1}_5 + m^k)/2, t^k + \tau/2]$$ \hfill (6.22)

Which gives the approximated value for the unknown $m^{k+1}(i)$.

### 6.4.1 Efficiency of the proposed scheme

The proposed scheme is efficient against state-of-the-art discretization schemes given the facts that (1) at each cell of the discretized mesh only $3 \times 3$ linear equations are dealt unlike other schemes where computations are carried out on $3N \times 3N$ sets of equations and hence a significant reduction in overall computation time for the whole mesh, (2) at each mesh cell $i$ it preserves magnitude of the magnetization at the cell: a requirement of the LL equation that must be addressed, (3) it has high accuracy even for larger step sizes, (4) unlike other schemes where the sets of equations to be solved are closely coupled, the proposed scheme is highly parallel where the steps could be followed in parallel on all the mesh cells simultaneously. In future these computations would be performed in parallel on GPU architectures to speedup the process for problems with much finer cell sizes (larger $N$) and magnetized samples of large dimension sizes.
6.5 Results and Comparisons

The scheme has been tested for mu-mag standard problem no-4 with the following assumptions considered during the tests. A permalloy thin film discretized into \( N \) number of cells with uniform magnetization is considered. For short range interactions, 6 neighbors are considered for the exchange field. The initial equilibrium "s-state" with saturation \( M_s = 8 \cdot 5 A/m \), damping of \( \alpha = 0 \) and zero anisotropy value for the \( K_1 \) are considered. Applied field \( h_a \) with x and y component values \(-35.5mT\) and \(-6.3mT\) respectively, is considered perpendicular to x-axis.

Two geometries of cell sizes \([2.5e-9, 2.5e-9, 3e-9]\) with \( N = 200 \times 50 \times 1 \) and \([5e-9, 5e-9, 3e-9]\) with \( N = 2500 \) are considered in 3 dimensions. A comparison of the computed magnetization with the one obtained from OOMMF Micromagnetic simulator is shown in figure 6.2. As can be seen all the three components of the magnetization \( (m_x, m_y, m_z) \) are closely related to the ones obtained by OOMMF. The plot in figure 6.3 shows conservation of magnetization at a cell for every time

![Figure 6.2: Comparison plot of magnetization components with OOMMF for constant \( h_a \) at time step of 1 ps.](image-url)
step. The averaged value of the normalized magnetization is almost equal to 1 $(1 - m_{av} \approx 10e^{-16})$. With no damping ($\alpha = 0$) the law that free energy should remain constant is obeyed even for relatively larger step intervals as shown in figure 6.4. The relative error as a function of time is computed using the following formula.

$$e_g = \frac{g(m^k; h_a) - g(m^0; h_a)}{g(m^0; h_a)}$$  \hspace{1cm} (6.23)

Finally relative error and its averaged value results for 5nm mesh size are reported in figures 6.5 and 6.6.
Figure 6.4: Relative error $e_g$ values with no damping $\alpha = 0$ for various time-steps.
Figure 6.5: Relative error plot $e_\alpha = (\alpha' - \alpha)/\alpha$ for various time-steps. Here cell size is $5e-9m$. 

94
Figure 6.6: Averaged relative error plot \( < e_\alpha > \) for various time-steps and cell size is \( 5e-9m \).
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In terms of best price-performance ratio, GPUs are viable solutions for massively parallel applications in achieving maximum performance. This research work focused on the suitability and usefulness of parallel-processor architecture, in the case of single or multiple GPU, to exploit inherent parallelism in the application at hand. Out of various tools and frameworks for programming GPUs, the OpenCL parallel programming framework was selected because of its scalability, heterogeneity and being an open standard with support for a diverse set of processor types. New parallel sorting algorithms were designed and tested against popular sorting techniques. It was shown that the proposed techniques are highly parallel and thus well suited for multi-processor architectures. The achieved speed-up against well-known techniques uncovers the aim of having a highly parallel sorting technique best suited on GPUs. The thesis portion devoted to implementation of an FFT library provide at one hand the intricate theoretical details of the Fourier transformation in general and on the other hand the nitty-gritty aspects of GPU architecture for general purpose programming. It was shown how the different memory types and the data transfer time between CPU-GPU affects performance. Performance comparison for different transform sizes on different GPU types shows that our library achieves significant speedup against state-of-the-art FFTW and a speedup of the order 8x against NVIDIA cuFFT library. Along that our library supports auto-tuning and load balancing in systems with more than one GPU devices. An optimized version
of the mentioned library is tested for accelerating magnetostatic field computation in the area of Micromagnetics. The GPU based magnetostatic field solver has 4x speedup against OOMMF Micromagnetic simulator for large number of cells (around 8-16 millions). By computing magnetostatic field per cell by the FFT method the Landau-Lifshitz equation of motion is discretized in space and integrated in time for investigation of magnetization dynamics in the cells. A new time integrating scheme, based on mid-point and Runge-Kutta method, has been introduced. The novel time stepping technique has lower complexity in a since that only a limited number of equations are solved per elementary cell unlike other time integrators for the LLG equation discussed in literature. The new scheme has been tested and validated against $\mu$-mag problem no.4 for preservation of basic Micromagnetic properties and accuracy comparison with OOMMF. The new scheme is highly efficient, more accurate even for larger time steps and is parallel in a sense that computations on multiple elementary cells could be carried out simultaneously.

### 7.2 Future Work

The generic FFT library would be tested over cluster of distributed GPU systems using Message Passing Interface (MPI) in order to support larger input sizes (beyond $2^{24}$ limit). A strategy for dynamic load balancing would be defined in order to balance the load based on the GPU capabilities (like Memory size, cache limits etc). In future it is aimed that the specialized version of the FFT library would be integrated to OOMMF Micromagnetic simulator. Some more optimizations for example carrying out the zero-padding of both the tensor matrix and the magnetization vector components would be handled inside the corresponding kernel code. This would save us unnecessary transferring of zero padded memory space from CPU to GPU. Moreover the time integration method would be implemented entirely on GPU architecture where simulation could be carried out on multiple cells simultaneously.
Bibliography


[39] Omar Khan, Carlo Ragusa, Fiaz Khan, and Bartolomeo Montrucchio. A mutual


[67] Ben Van de Wiele. Numerical study of magnetic processes: extending the


