Directives Based Programming of GPU Accelerated Systems

Saif Mulla

August 19, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

Graphics Processing Units (GPUs) are commodity chips primarily used as co-processors for processing high definition graphics on a computer system. It possess faster processing power and efficiency in handling accurate single and double floating point numbers with less power consumption compared to CPUs. Realising its potential in general purpose computing manufacturers of these chips have added programmability feature so that enormous power of these chips can be utilized for general purpose computing.

However, programming these chips has emerged new area of research and persuaded many companies into determining ways of flexibility and ease of programming. Presently there are few programming models which have their own advantages and disadvantages but most matured and widely used of its kind is CUDA developed by NVidia however it is architecture dependent on NVidia architecture. Another directive based programming model OpenMP for Accelerator is under development by OpenMP forum which is architecture independent.

Evaluations of both these models are elaborated by porting serial Molecular Dynamics code on each of these models. It appears that directive based approach is easier and less time consuming however it is slower in performance compared to CUDA. Moreover when these models are compared with Host OpenMP, they proved widely scalable with additional advantages of economics over maintainability and hardware cost.
Contents

Chapter 1 Introduction........................................................................................................2

Chapter 2 Background.........................................................................................................5
  2.1 Architecture of GPU......................................................................................................5
  2.2 Programming models for GPUs...................................................................................7
    2.2.1 CUDA................................................................................................................7
    2.2.2 OpenMP Accelerators.......................................................................................9
    2.2.3 Other Directive based GPU programming languages.......................................16
  2.3 Molecular Dynamics.................................................................................................16
  2.4 Related Work...............................................................................................................18

Chapter 3 Molecular Dynamics Code Analysis.................................................................21
  3.1 Pseudo code ...............................................................................................................21
  3.2 Code description.........................................................................................................22
    3.2.1 Fcc function.....................................................................................................23
    3.2.2 Maxwell function.............................................................................................23
    3.2.3 Dfill function...................................................................................................23
    3.2.4 Domove function .............................................................................................24
    3.2.5 Forces function .................................................................................................24
    3.2.6 Mkekin function: ............................................................................................26
    3.2.7 Velavg function .................................................................................................26
    3.2.8 Dscal function..................................................................................................26
  3.3 Code performance and execution timings .................................................................27
  3.4 GPU implementation.................................................................................................28
Chapter 4 OpenMP Accelerators

4.1 Linear Algebra

4.1.1 Vector Addition

4.1.2 Vector Multiplication

4.1.3 Matrix vector Multiplication

4.1.4 Matrix Multiplication

4.2 MD Code implementation

4.2.1 Use of acc_data directive and present clause

4.3 Kernel porting

4.3.1 Forces kernel

4.3.2 Domove Kernel

4.3.3 Mkekin Kernel

4.3.4 Velavg Kernel

4.4 Optimized MD code

Chapter 5 CUDA

5.1 Kernel Porting

5.1.1 Forces Kernel

5.1.2 Domove Kernel

5.1.3 Mkekin Kernel

5.1.4 Velavg kernel

5.2 Optimized MD code

Chapter 6 Comparing programming models

6.1 Comparing CUDA and OpenMP Accelerators

6.1.1 Comparing execution timings

6.1.2 Comparing Memory copies

6.2 Host OpenMP, Accelerators and CUDA

Chapter 7

7.1 Further Work
List of Figures

Figure 1: An overview of Fermi GPU architecture.........................................................6
Figure 2: Components of Streaming Multiprocessor and Streaming Processor (CUDA Core). .................................................................7
Figure 3: Serial Execution timings for 4000, 6912 and 13500 particles.........................27
Figure 4: Percentage of execution timings for each function on 4000 and 13500 particles........................................................................................................28
Figure 5: (A) Atom interactions for optimized code (B) Atom interaction for un-optimized loops for GPU ..................................................................................28
Figure 6: execution timing comparisons between optimized code and code modified for GPU........................................................................................................29
Figure 7: vector addition CPU and GPU execution timings (A) 10024 and 100024 particles. (B) 1000024 and 10000024 particles.......................................................31
Figure 8: vector multiplication CPU and GPU execution timings for (A) 10024 and 100024 particles. (B) 1000024 and 10000024 particles.............................................32
Figure 9: Matrix vector multiplication for matrix and vector of equal dimensions (A)10x10 and 100x100 (B) 1000x1000 and 10000 x 1000..................................................33
Figure 10: Matrix multiplication execution timings for equal matrix sizes (A) 10 and 100 (B) 100 and 1000. .........................................................................................34
Figure 11: Time spent performing memory copies on GPU to CPU after porting each kernel using acc_copy clause...............................................................35
Figure 12: Time spent performing memory copies on GPU to CPU after porting each kernel using present clause. .................................................................37
Figure 13: Results for Forces kernel execution timing with GPU to CPU ratio for different thread block sizes and ratio of kernel only time to CPU time for 13500 particles executed on 128 threads per block........................................38
Figure 14: Domove and forces kernel timings combined together for ratio of GPU to CPU time and execution timing plotted only for kernel execution timing excluding memory copies for 13500 particles on 128 Threads per block .............................................40

Figure 15: Domove, Forces and Mkekin kernel combined execution ratio of GPU to CPU time and execution timings excluding memory copy timings for 13500 particles on 128 threads per block plotted for “Kernel only” trend .................................................................43

Figure 16: Domove, Forces, Mkekin and Velavg kernel combined execution time with ratio GPU to CPU time and memory kernel execution time excluding memory copy time for 13500 particles on 128 threads per block ........................................................................43

Figure 17: ratio of GPU to CPU execution timings for optimized MD code. ................45

Figure 18: Memory copy timings for optimized MD code excluding kernel execution timings ........................................................................................................46

Figure 19: Comparative graph showing CUDA transfer timings after porting each kernel on Device. .................................................................48

Figure 20: Kernel forces execution timings for CUDA and kernel execution timing excluding memory copy timing for 13500 particles on 128 threads per block. ....50

Figure 21: Domove and Forces kernel combined execution timings ratio GPU to CPU and kernel only execution timing excluding memory copy timings for 13500 particles on 128 threads per block .................................................................52

Figure 22: Domove, Force and Mkekin combined execution timings ratio of GPU to CPU time and kernel only execution timing excluding memory transfer time on 13500 particles on 128 threads per block .................................................................53

Figure 23: Domove, Forces, Mkekin and Velavg combined execution timing ratio GPU to CPU and kernel only execution time for 13500 particles on 128 threads per block. 54

Figure 24: Ratio of execution timings for optimized MD code on CUDA.......................56

Figure 25: Memory copy timings for optimized MD code excluding kernel execution timings ........................................................................................................57

Figure 26: Ratio of execution timings for CUDA and OpenMP Accelerator for block size of 128 .................................................................................................................................60

Figure 27: Comparison of CUDA and OpenMP Accelerators memory transfer timings for optimized MD code. ...............................................................................................61

Figure 28: Parallel to CPU ratio of execution timings for Host OpenMP, CUDA and Accelerators .................................................................................................62
Acknowledgements

I am very grateful to my supervisors Dr Alan Gray and Dr Alistair Hart for their immense support, motivation and belief in me and also the amount of knowledge which they shared will be surely useful for me in years to come.

I would like to thank all the teaching and non-teaching staff of MSc for providing support and consideration. I would also like to thank Cray Inc. for granting me access to Puffin machine and also thanks to all puffin users for providing support. Personal thanks to all my fellow classmates for their kindness and consideration throughout the course.
Chapter 1

Introduction

Graphic Processing Units (GPU) have certainly emerged as solutions for personal computing such as game consoles, mobile Devices, hand held media Devices, personal computers, workstations, etc. there is also very significant usage in the field of scientific computing. GPUs since their evolutions are known for dedicated Graphical processing in a computer system, providing co-processing power significantly for gaming, professional graphical processing applications, Computer Aided Design (CAD). They are co-processors which are used along with Central Processing Units (CPU) although GPUs have enormous amount of faster processing capabilities and better efficiency in processing floating point numbers. GPUs consist of few control units (CU) and large number of Arithmetic Logic Units (ALU) and have their own dedicated on-chip memory with high bandwidth capabilities. Generally a GPU is given the most compute intensive task by a CPU which is offloaded to GPU consequently which can perform several other tasks in parallel while the GPU is processing the task.

Scientific computing requires complex applications. Problems demanding processing of mathematical models require efficient floating point calculations. Due to complex calculations it requires faster computing power to get the solution as fast as possible therefore solutions are mostly run on computer clusters or powerful workstations so that the processing can be performed in parallel and results can be obtained in much less time. These clusters of computers are composed of several computers with single-core or multi-core CPUs each, which work in parallel to provide faster solutions.

GPUs on the others hand are composed of many cores which works in parallel, therefore there is significant scope of usage in the area of scientific computing solutions for using many core capabilities of GPU rather than multiple CPU parallel computers. Although this does not means that they can replace CPU based parallel solutions but their capabilities can certainly provide high throughput solutions on cheap and efficient in-house desktop systems for small scale usage in scientific computing solutions.

GPUs are gradually but certainly influencing their usage in the scientific computing community as more and more parallel computer manufacturing companies
and research centres are attracted towards its processing capabilities. One can notice their presence as amongst the top ten fast super computers in top 500 fastest computers in the world there are now over three computers using GPUs. The former world fastest computer from China Tianhe-1A till (April 2011), now the second fastest in the world after Japan’s K computer, is composed of CPUs and GPUs which can in unison perform 2.57 quadrillion calculations per second. The 4th and 5th fastest computer “Nebulae” and TSUBAME from China and Japan respectively, are composed of GPUs as well [1].

The most significant factor which brought GPUs into High Performance Computing (HPC) is improved programmability. Since the introduction of Compute Unified Device Architecture (CUDA) programming language which facilitates the flexibility to program GPUs, There are several solutions in scientific computing which have been created using GPUs such as Mandelbrot set, ray tracing, molecular dynamics, computational biology, fluid dynamics, parallel video processing, computational chemistry, graphics processing etc. Nowadays several supercomputing companies are coming up with solutions exploiting GPUs. There are several research institutes around the world who have implemented their existing application code to GPU versions as well, such as AMBER and DLPOLY who have implemented their Molecular Dynamics codes version in CUDA.

Cray Inc. one of the largest super computers manufacturing company in the world also has shown their interest in deploying GPUs as coprocessors on their latest systems. Cray has built the former world’s fastest super computer which currently stands 3rd in the list of top 10 super computers in the world. “Jaguar” is built for Oak Ridge national laboratory in USA which is primarily used for scientific computing and data analysis [1]. Although two more computers in the top 10 list “Ceilo” and “Hopper” standing 6th and 8th in the ranking respectively are implemented by Cray [10]. The company is already experienced with shipping FPGA chips on their previous super computer systems named Cray XD1 and further more improved version on Cray XT4 [10].

Cray has already shipped GPUs on their latest systems on their MPP and workstation machines. The latest Cray MPP machine will be shipped with AMD Opteron “INTERLAGOS” 16 core CPUs and NVIDIA tesla x2090 GPUs as coprocessors. The use of GPUs on xk6 [12] has successfully allowed Cray to move towards Exascale computing power. It has further plans to update their existing systems XE6 and XE6m shipped with GPUs [11]. Cray also manufactures several medium sized parallel computers and workstations. Cray cx1000 series of SMPs (Symmetric Multiprocessors) have NVIDIA Tesla GPUs used as coprocessors on them with one GPU for every 2 CPU [11]. These small scale SMP machines from Cray are customized solutions for small and medium sized organisation which require speed not like a huge MPP but providing enough power and which can be installed in house in their premises with reasonable cost. Cray SMPs with GPU have succeeded in satisfying demands of their small and medium sized customer machines with the help of CPU and GPU combinations providing enormous amount of power inside a single work station. Cray SMP workstations are used in oil and natural gas industries to model
oil exploration and financial industries to run their derivative calculations, study market analysis etc.

The enormous potential of GPUs in parallel computing and supercomputing community is realised and also that power can be unleashed by programming API such as CUDA. However programming GPUs is another barrier which has risen, as CUDA is the programming language developed by NVidia and CUDA can only be used to program NVidia specific architecture GPUs and therefore it is not architecture independent. Although there are several other programming languages to program GPUs such as OpenCL which is an open language development by Khronos group to program any GPU irrespective of architecture but the problem with OpenCL is it lack of maturity in programming as it is still under development but currently cannot be relied on to get peak performance through programming. Another two directive based programming approach are hiCUDA (directive based version of CUDA) and PGI Directives developed by Portland group, these approaches provide simplicity in programming GPUs at higher level and complexity of optimization is handled by the compiler at base level by providing implicit parallelism. Indeed CUDA is most powerful language to program NVIDIA GPU available to date but it is quite low level and programming complexity is comparatively high. A programmer needs to know every detail to port a serial program into CUDA and this requires strong expertise from the programmer.

Therefore there is strong demand for better programming approaches towards programming GPUs which will be easier and requires fewer efforts from the programmer so that the programmer can concentrate on development of the algorithm and programming does not take more efforts from him. OpenMP forum has proposed a new directive based programming language for GPUs, it is a proposed extension to OpenMP and promises to contain similar characteristics of OpenMP shared memory programming API which is currently the best choice to program shared memory machines. The OpenMP forum Architecture Review Board committee is also co-chaired by Cray which has taken an active initiative to develop directive based programming language for GPUs, called OpenMP for Accelerators.

The aim of this project is to program GPUs using CUDA and OpenMP Accelerators using Cray compiler, which has inbuilt support for OpenMP Accelerators. The compiler is installed on Cray’s own machine “Puffin”. The codes need to compiled and executed on the same machine and therefore in the beginning serial vector and matrix multiplication codes will be developed and later implemented into OpenMP and finally ported into CUDA and OpenMP Accelerator Directives. In addition molecular dynamics (MD) serial code is implemented with OpenMP, CUDA and Accelerator directives.
Chapter 2

Background

GPUs were introduced by NVidia in 1999 for dedicated graphical processing primarily for computer games and professional computer graphics. A GPU is a coprocessor which can run along with the CPU but has most of silicon dedicated for double precision floating point calculations. The GPU handles most of the most compute intensive tasks by letting the CPU offload them and performing floating-point calculations more accurately at a very fast pace thereby increasing overall throughput of application. GPUs are much faster and more accurate in floating point arithmetic calculations with much faster memory bandwidth with 10:1 proportion [31]. GPU have more ALU and less CU compared to CPU which has more CU and less ALUs.

Programming GPUs in the past wasn't easy; there was lots of complexity involved in programming them because they were not actually developed for general purpose; rather they were dedicated for graphical processing on vertex and pixel shading for complex rendering pipelines. There were programming languages to perform the functions of vertex and pixel shading such as CG (C for Graphics), OpenGL (Open Graphics Language) DirectX3D developed by NVidia, Khronos group and Microsoft respectively, but programming GPUs for general purpose computing such as HPC remained a challenge because to utilize the power for GPGPU they have to trick the programming API available by passing the data through graphical functions to be processed on GPU and bring back the processed data through those functions. However this was awkward and cumbersome for naïve programmers. Responding to this demand NVidia realised the requirement and introduced CUDA enabled architecture GPUs and a programming language dedicated for them, which can facilitate for GPGPU.

2.1 Architecture of GPU

NVidia has recently introduced advanced architecture GPU with Code-Named “Fermi” [2]; it has several improvements compared to the other former version and is dedicated purely for scientific computing and data mining problems. It has several advancements such as error correcting memory, L1 and L2 cache, faster atomic operations and faster
context switching. Fermi architecture is used in NVidia Tesla GPUs, which is much more powerful and advanced, compared to the former ones; Figure 1 shows architecture of Fermi GPU. A Fermi GPU is composed of 16 Streaming Multiprocessors (SM) and each SM has 32 Streaming Processors (SP) also called CUDA cores in all the whole Fermi GPU is composed of 512 CUDA cores, a graphical view of SM and SP components is shown in Figure 2. Each SM performs task of Multiple Instruction stream Multiple Data stream (MIMD) parallelism and can execute different functions independently consequently they do not possess synchronization between them but share common global, constant and textured memories. SP inside each SM resembles Single Instruction stream Multiple Data stream (SIMD) and this fashion of parallelism is termed as Single Instruction Multiple Thread (SIMT) however all the SPs inside SMs can synchronize execution and communicate with each other over shared memory dedicated to each SM. Each core or SP is composed of dedicated Floating point and Integer unit and each SM has one Load/Store unit which makes sum of 16. there are four dedicated Special Function Units (SPU) 4 for each SM which can be used for sine, cosine, square root instructions executed by SMs. Each SM has a 64 KB configurable shared memory and L1 cache which is shared between each thread inside the SM. The shared memory and L1 cache can be configured and partitioned proportionately using CUDA as per the requirement of the application. L2 cache is shared among all SM and also performs all the load/store operations and Multiply Add operations. The on chip DRAM memory also called Device memory is 6GB till date and all the memory are secured by Error Correcting Code and memory bandwidth transfer speed is 100 mbps.

![Fermi GPU Architecture Diagram](image)

*Figure 1: An overview of Fermi GPU architecture.*
2.2 Programming models for GPUs

2.2.1 CUDA

CUDA stands for Compute Unified Device Architecture, is a parallel computing architecture introduced by NVidia to program NVidia architecture GPUs [3]. It is presently the most advanced and preferred programming language to program NVidia GPUs. CUDA GPU architecture consists of several hundred CUDA cores, which are composed of several thousand threads. A custom CUDA function executing on GPU (Device) is called a Kernel, it executes in parallel on the GPU and asynchronously from CPU, while a CUDA Kernel is executing on the Device the CPU can simultaneously perform its own operations as both the Device and Host (CPU) run simultaneously and independently. At one time there can be more than one Kernel executing simultaneously on the Device to make optimum utilization of the GPU. However, on a Tesla GPU 16 different Kernels can be launched for parallel execution. A Kernel is launched using execution configuration parameter inside a CUDA program; the execution configuration requires two arguments inside angular brackets “<<<arg1, arg2>>>”, first is the number of blocks and second is number of threads per block. For example if we want to launch a Kernel with following code “Kernelname<<<16,2>>>(”", it notifies the NVCC compiler there should be 16 blocks created or invoked with each containing two threads, precisely there will be 16 instances of Kernel function performing similar tasks and two threads executing similar
instructions. The parameter can also be passed as structure of dim3 type when the Kernel requires x and y dimensional blocks and x, y and z (3 dimensional) threads execution.

When a Kernel is launched it is actually instantiated on a single grid of parallel threads also called a Grid. A Grid is single dimensional and is an array of thread blocks, to access dimensions of a grid that is number of blocks inside grid a predefined CUDA variable “gridDim.x” is used, it obtains x axis of grid although grids only have 1 dimensions. The grid is composed of thread blocks which can be 1 dimension or 2 dimensional blocks are actually SM on a CUDA GPU and a separate instance of CUDA Kernel is executed on each block in parallel. There is no synchronization mechanism between blocks on a GPU but they share a common L2 cache, each block has its own shared memory and L1 cache which is accessible to thread inside the block and is not visible outside its respective block and apparently the entire block has access to Device memory. The dimensions of blocks can be determined by inbuilt variable “blockDim.x” and “blockDim.y” for x and y dimensions respectively and for determining the unique id of the block inside the grid is “blockIdx.x” and “blockIdx.y” for x and y dimensions respectively.

Each block is composed of several hundred light weight CUDA threads; each thread has its private memory in the form of registers and all threads inside a block share common L1 cache and shared memory. All the threads inside each block can perform synchronisation, threads inside a block can be 1D, 2D or 3D which corresponds to x, y and z coordinate. Each thread inside a block is identified uniquely through a thread id and could be obtained using “threadIdx.x, threadIdx.y and threadIdx.z” for x, y and z coordinates respectively.

When a CUDA kernel is launched each kernel is executed independently on each block in parallel, subsequently executing threads in parallel as well. However all threads for each block are not executed simultaneously at single instance but they are executed in form of warp of threads. A warp is composed of 32 threads each and there can be 1 or 2 warp instance launched for each block. Each block can simultaneously execute 2 warps at a time if there are single precision numbers and 1 warp if there are double precision numbers. For instance if there are 10 blocks launched by program there could be 10 or 20 warps of threads executing in parallel. The success behind speed of execution of a program depends upon larger number of warps executing in parallel therefore if thread block sizes are smaller there would be larger number of blocks initialized by the runtime consequently there could be more number of warp instances launched because each block has its own resident warp.

CUDA facilitates shared, constant, local and global memory, all the private variables specific to threads reside on the local memory, scalar variables are actually stored on registers and local arrays in local memory, Generally register access is much faster than access to local memory, they are just declared normally as C syntax. Shared arrays and shared scalar variables are stored on shared memory which is specific to each thread block, shared memory and L1 cache is 64KB in total but can be configured
proportionately depending upon the requirement of the program it can be declared using __shared__ keyword. All global read only scalar variable inside a CUDA program can be defined on constant memory as constants are cached off chip and memory access is very fast. It can be declared using __constant__ keyword inside a CUDA program. Global memory is memory which all threads have access to. It is the main on-chip Device memory and is the slowest memory to access from threads. Presently Fermi GPUs have 6 GB global memory.

2.2.2 OpenMP Accelerators

OpenMP API is the most preferred programming language for shared memory parallel systems. It is a directive based programming approach, the directives are treated like comments by non OpenMP compliant compilers and compilers supporting OpenMP directives handle the parallelism implicitly. Programmer just needs to define a parallel region which is to be executed in parallel and has to define the data access between processors. It supports features such as loop level parallelism, synchronization, atomic variables and nested parallelism. There are several compilers supporting OpenMP features such as Cray CCE, PGI, Visual C, GCC, IBM, Intel, Path Scale etc.

There is large amount of research under way to determine tools and techniques to make programming Accelerators easier and more efficient. A lot of work has been attempted and implemented towards using OpenMP as a programming model on hardware Accelerators such as Cell and GPUs, but for couple of years the interest has mainly been towards using OpenMP as programming model for GPUs. OpenMP is used as a programming model for IBM Cell broadband engine to make Cell programming easier. OpenMP directives are used in runtime library which is used over IBM XL compiler; the compiler translates OpenMP directive constructs to generate appropriate code [16]. The ease of OpenMP directive based programming API has generated lot of interest to develop framework for programming GPGPU. A program can be built by simply using OpenMP directives and later at compile time the program will be translated into CUDA and further to appropriate architecture code. The framework actually performs source-to-source translation of OpenMP to CUDA but actually increases the scope of ease of programming GPUs [17].

Considering the work and interest of programmers to use OpenMP for programming Accelerators or to provide similar simplicity and portability for heterogeneous systems, OpenMP ARB has considered that future supercomputers might not necessarily be homogenous CPU oriented parallel systems rather they could be system with diverse heterogeneous architectures consisting of CPU, GPUs, and FPGAs [18]. These systems with diverse architectures might be multicore or multi-sockets and therefore will demand diverse programming approaches for these architectures which could be nightmare for the programmers to integrate such different languages or perhaps require rewriting the code for the architecture, in order to ultimately generate a solution [19]. Therefore in order to provide similar kind of programming simplicity and portability and almost all the characteristics of OpenMP language, the OpenMP forum has
proposed to extend the OpenMP language for support for programming Accelerators, it is co-chaired by Cray and part of their initiative towards Exascale computing research.

OpenMP Accelerators are a proposed extension of OpenMP API and have similar characteristics like OpenMP. Moreover OpenMP Accelerator is used to program GPUs and provides directives based approach towards programming GPU and promises to remove the pain from programmers who are mostly involved in extracting performance from GPUs by using low level programming approaches. OpenMP Accelerators are designed to hide the intricacies from the programmer for configuring GPUs in terms of distribution of threads and load balancing the number of blocks. Most of the configuration and data transfer is done implicitly to great extent and also provides the programmer the flexibility to make those configurations explicitly. Generally to execute a compute intensive task of CPU on GPU the data structure has to be transferred from the Host to Device and the processed data has to be brought back to Host from Device. This programming task is handled explicitly by the Accelerators however the programmer can explicitly handle those tasks if required for specific application requirements. Due to its characteristics similar to OpenMP the advantage over other GPU programming approaches is its simplicity in programming and language flexibility as it can be programmed in three main programming languages C, C++ and FORTRAN [23]. Therefore this provides the flexibility of programming in three different languages and does not requires the programmer to learn a new language or to rewrite significant part of code again to port on Accelerator. Most of the times programmers are not convinced about the merits of using GPUs are due to lack of flexibility in language and to rework the code or rewrite it.

2.2.2.1.1 Execution model

Accelerator is termed used to define a GPU more technically; this is also called Device in CUDA. Accelerator memory aka Device memory or the global memory is the on-chip memory present on the Accelerator. Accelerator region aka Kernel is the program region which is to be executed on the Accelerator. The master thread or the main program starts execution of the program on single thread, however when the master thread encounter a Accelerator region it allocated that part of the program to the Accelerator and thus that program is executed on the Accelerator asynchronously. The data structure required by the program is transferred implicitly to the Accelerator and later the processed data structures are copied back from the Accelerator to the Host. Moreover if there are multiple Accelerators available on the system the runtime will implicitly determine which Accelerator to initialize for that particular Accelerator region, generally the first Accelerator is preferred if it is ideal and then other are allocated in sequence. However till now the OpenMP Accelerator directive support only one Accelerator but proposals are drawn for multiple Accelerators support in future. The Accelerator model has similar private and shared data clauses due to which the runtime determines the local and shared source of the data structure and implicitly handles synchronization wherever possible during the execution cycle. All the synchronization and barrier construct are determined at compile time by the Cray compiler. The loop immediately after the Accelerator directive is parallelised between
threads and thread blocks thus the nested loop inside is just executed inside each thread [23].

2.2.2.1.2 Memory model

An OpenMP Accelerators memory model is derived from OpenMP memory model and the functionality is similar to the former. OpenMP Accelerators have two type of access to variable shared and private, a shared variable is shared across all the threads inside a thread block and private access is the local scalar or the local array created for each thread. This private access variable is accessible only to the thread for which it is created, scalar variables are stored on the register allocated for the threads and private array is stored locally or the space allocated for the each thread resource.

Generally all the data copying is handled implicitly by compiler itself, the “acc_copyin” clause can be used to explicitly move data to the Accelerator, but when the data is copied from Host to the Accelerator it determines the scope of the data access depending upon the clauses defined in the directives or if they are not specified then the compiler itself determines the type of access. However when the data is modified on the Accelerator it is not directly reflected to the data copy on the Host, the modified data has to be copied back from the Accelerator to the Host and this is done by implicitly or explicitly using “acc_copyout” or “acc_copy” clause only then the modified data is actually available on the Host.

2.2.2.1.3 OpenMP Accelerator directives

OpenMP Accelerator directives are very powerful and implicit way of processing data from the Host to Accelerator although some of the most directives are still under development and some of the Internal Control Variables (ICV) is not accessible yet such as getting the Accelerator count, some predefined global variables.

2.2.2.1.3.1 Accelerator region directive

Accelerator region directive (acc_region) notifies the runtime that the code contained inside this region is to be executed on the Accelerator. When the runtime encounters this directive the code inside this region is copied and placed inside a function which will be executed on the Accelerator however this is completely done by the compiler and therefore no Device function is required to be explicitly written by the programmer. Generally the programmer needs to identify the part of the code to be ported on the Accelerator and include that code region inside Accelerator region directive.
```c
#pragma omp acc_region acc_copy(arr[0:N]) num_pes(2:128)
{
#pragma omp acc_loop
      for (i=0;i<N;i++)
         arr[i] = arr[i] * 2.04;
}
//acc_region ends

//printing the first and last element of the array.
Printf("arr[0] %lf, arr[N-1] %lf",arr[0],arr[N-1]);
```

**Code Block 2-1: Accelerator region directive.**

In *Code Block 2-1* shows code for OpenMP Accelerator `acc_region` directive and clauses which can be used. During compilation of the code when the compiler detects `acc_region` directive it ports the code on to Accelerator which should by default asynchronously execute on Accelerator. Generally data transfers are determined implicitly by compiler itself; however data movements can be defined explicitly by the programmer for performance optimization in terms of memory transfers. In the above code the array “arr” of size N will be copied from Host to Accelerator and later modified array should be copied back from Accelerator to Host after end of Accelerator region.

**Clauses**

- **num_pes**: this clause is used to define number of blocks and number of threads inside each block. It has 2 kind of depth for declaration depth 1 for number of block this is rarely used as when the number of thread per block is mentioned the runtime determines number of blocks based on threads per block. Depth 2 is used to declare number of threads per block the default for number of threads per block is 128, although at present it supports thread per block sizes for multiple of two from 64, 128, 256, 512, 1024. For instance number of threads per block can be mentioned as `num_pes (2:256)` this will configure 256 threads per block therefore if the size of N is 512 then N will be divided by 256 internally to determine number of blocks. This clause can also be used with `acc_loop` directive and `acc_loop_region` directive.

- **Acc_copy**: this clause is used to perform data copies from Host to Accelerator and from Accelerator to Host, it perform two functions by declaring one clause, it performs the task of `acc_copyin` and `acc_copyout` clause which are used to copy data to Accelerator and copy data back from Accelerator respectively. `Acc_copy` is generally used when the data is to be processed on the Accelerator and again the updated data is required by the Host therefore it needs to be copied back from the Accelerator.

  In the above code fragment `acc_copy` clause is used to copy “arr” array of size N from Host to Accelerator and later after processing the parallel loop and at the end of the `acc_region` the processed array is copied back from Accelerator to Host.

**Accelerator region also supports several clauses such as:**

- **Private clause**: private clause can be used with `acc_region` directive, when this clause is mentioned a unique separate copy is given to each SM on the Accelerator,
ideally it is a shared resource among all the thread inside a block of threads which has scope of block on the Accelerator and there are such unique copies of the objects on each block which is equivalent to the number of block created by the execution environment. Variables which are automatically private in OpenMP (e.g. loop variables) are also private in Accelerator regions.

**Acc_Copypin clause:** this clause is used to explicitly mention the data to be copied from Host to Accelerator. Ideally when the data is only for read only purpose and not necessarily required back on Host for further processing then this clause is used to make data copying to Accelerator. Mostly function arguments scalar variable with values or input arrays are used to transfer using acc_copypin directive.

```c
#pragma omp acc_region acc_copypin(A[0:N],B[0:N]) acc_copyout(C[0:N])
#pragma omp acc_loop
  For(i=0;i<N;i++){
    C[i] = A[i] + B[i];
  }
//acc_region ends here
// print array C values
```

**Code Block 2-2: acc_copypin clause**

In Code Block 2-2 of vector addition two input vectors A and B are required to perform addition and generate resultant vector C therefore A and B vectors need to be copied to the Accelerator but only vector C is required to be copied back to the Host. Therefore by using acc_copypin for the read only arrays and acc_copyout for resultant vector C eliminated unnecessary data transfers between Host and Accelerator. Acc_copypin is similar to “CUDAMemcpyHostToDevice” keyword used with “CUDAMemcpy” function in CUDA.

**Acc_Copyout clause:** this clause is used to explicitly mention the data transfer required from Accelerator to Host. Ideally this clause is used when the processed or the updated data on the Accelerator is required by the Host. In the code fragment above vector “C” is processed in to resultant vector and copied back from the Accelerator to Host, it is similar to mentioning “CUDAMemcpyDeviceToHost” keyword with “CUDAMemcpy” function in CUDA.

**2.2.2.1.3.2 Acc_data region directive**

Acc_data region or acc_data directive is used to specify a region or the scope of program that will be executed on the Accelerator; it is different from acc_region as there can be many acc_region directives in a program having similar scope or sharing similar data to be processed. If there are more than one acc_region in program there can be that much number of data transfers required to and from the Accelerator which increases the execution time of the program. Therefore if there are more than one acc_regions that can be executed sequentially on the Accelerator sharing similar data and scope these all regions can be included less than one whole acc_data region directive. The significant advantage of using multiple acc_region under one single acc_data region directive is to data transfers and data access. For instance if there are three acc_regions in a program and all of them require similar input array to process
their tasks, then every time acc_region is encountered the array will be copied from Host to Accelerator even if one is using acc_copyin clause, certainly time will be consumed copying data from Host to Accelerator for each call to acc_region.

**Acc_shared clause:** this clause is used when the copied data is too utilized for multiple processing on the Accelerator. When an array is copied on the Accelerator using acc_shared and modified on the Accelerator and when another Accelerator region is called which also requires the modified data of that array. Therefore it is not required to copy the modified array back to the Host and again from Host to Accelerator using “copyout” and “copyin” respectively instead if acc_shared clause is used the modified version is still on the Accelerator and later other acc_regions can use for further processing during the execution of the program. Therefore this eliminates unnecessary copying to and from Accelerator and encourages data reuse.

```c
#pragma omp acc_data acc_copyin(A[0:N]) acc_copy(B[0:N], C[0:N])

#pragma omp acc_region

#pragma omp acc_loop
For(i=0;i<N;i++)

//first acc_region ends here
#pragma omp acc_region

#pragma omp acc_loop
For(i=0;i<N;i++)
    B[i] = (A[i] * A[i]) / 0.7963;

//second acc_region ends here
#pragma omp acc_region

#pragma omp acc_loop
For(i=0;i<N;i++)
    C[i] = sqrt(A[i] * B[i]) / 0.2;

//third acc_region ends here
//acc_data region ends here
//print value of vector C on host
```

**Code Block 2-3: acc_data region directive**

In **Code Block 2-3** there are three acc_region placed inside acc_data region, vector A is required as an input array and further processing on the Host is not required therefore it is copied to Accelerator (acc_copyin) and not copied back from the Accelerator (acc_copyout). Further vectors B and C are need to be processed on the Accelerator and later copied back to the Host therefore they are copied using acc_copy clause which perform both the acc_copyin and acc_copyout function. Generally acc_copy clause is used in collaboration with acc_data region where the data is copied once on Accelerator and further after all the processing steps copied back from the Accelerator and thus avoid unnecessary data copying to and forth.

There are three acc_region in the above code fragment and each region requires all the three vectors therefore if the acc_data region wasn’t used there were in total nine
data transfer steps for all the vectors which add data transfer time to execution time. Therefore using acc_data region becomes handy when there are more than one acc_regions in a program.

**Present clause:** In order to utilize the data present on the Accelerator the execution environment should be notified about it. In the above code fragment the data is copied from the Host to Device in the beginning of the acc_region and further in the program the acc_regions inside the program should use the data which is already present on the Accelerator to avoid unnecessary data transfers. Therefore to inform the execution environment to utilize the data present on the Device, present clause should be used which takes the data present on the Accelerator Device.

### 2.2.2.1.3.3 Acc_loop directive

This directive is similar to parallel loop in OpenMP, it distributes the iterations of the loop across all the threads on the Accelerator and performs execution of the loop in parallel. Each thread in the execution environment performs single iteration of loop and all threads are executed concurrently on Accelerator. It supports several clauses such as num_pes and reduction.

```c
#pragma omp acc_region acc_copyin(A[0:N]) acc_copy(B[0:N]) acc_copyout(sum)
{
    #pragma omp acc_loop num_pes(2:256) reduction(+:sum)
    for (i=0; i<N; i++){
        B[i] = A[i] * 0.48907;
        sum += B[i];
    }
}
```

*Code Block 2-4 acc_loop directive*

In Code Block 2-4 acc_loop directive is used to execute for loop in parallel across N threads on the Accelerator. Vector A is provided as input array and vector B is processed on the Accelerator and copied back to the Host, also execution environment will have 256 threads per block as mentioned explicitly in num_pes clause. Further variable sum is passed to Device and copied back to Device by performing parallel reduction sum of each element of updated vector B.

### 2.2.2.1.3.4 Reduction clause

Reduction clause is similar to OpenMP parallel reduction clause which performs a parallel reduction summation or subtraction between each thread on the Accelerator. Reduction is generally passed with the result acc_loop to perform a parallel sum prefix on the specified array.

### 2.2.2.1.3.5 Acc_region_loop directive

Acc_region_loop directive is a combination of acc_loop directive and acc_region directive. Generally while porting a code fragment on Accelerator, the code fragment is included inside acc_region directive construct so that compiler knows that following code needs to be ported on Accelerator. Moreover along with
the acc_region if there is for-loop which has to be executed in parallel on Accelerator, it also had to be included under acc_loop directive. However acc_loop_region combines these functionalities into one single directive supporting all clauses, which are supported by acc_region and acc_loop. This directive is generally used when an acc_region only has one single loop inside it; therefore instead of using two lines of directives for small code fragment this combination directive becomes handy.

2.2.3 Other Directive based GPU programming languages

2.2.3.1 PGI Directives

PGI provides directives based approach for programming GPUs, it provides a programming model where the programmer just needs to specify region to be executed on the Accelerator. This eliminates the need to select the Accelerator explicitly and to distribute work load across processors and threads by utilizing the MIMD and SIMD architecture of the Device. Moreover all the complexities from initializing Accelerator, memory transfers and distribution of data are handled by the compiler itself, this is implicit parallelism functionality provided by the compiler. The programmer just needs to define the regions to be ported on the Accelerator by specifying appropriate directives. Although PGI directives syntax looks different from OpenMP Accelerator both have more or less similar functions [32].

2.2.3.2 HiCUDA

hiCUDA is a high level approach toward programming GPGPUs. hiCUDA stands for high level CUDA which provides programming APIs which are much simpler to use than CUDA. It uses the #pragma (pre-processor) directives approach for programming, like OpenMP Accelerators and PGI directives. This benefits the programming model by implementing the Accelerator code directly on serial code and no extra files need to be created with different syntax. It provides similar approaches like the other directive based languages for Accelerators it has two models: computational model and data model. In the computational model the programmer decides the most compute intensive task to be ported on to the Accelerator and data model facilitates programmer to specify memory allocation, de-allocation and transfers. It provides functionality similar to CUDA but with much higher level interfacing compared to CUDA API. Instead of writing CUDA API which is bit different from the serial code the hiCUDA API provide simple statements which are actually translated to CUDA low level API functions.

2.3 Molecular Dynamics

Molecular Dynamics (MD) is a computer simulation technique used to analyse and study of atoms and molecule models in the area of Physics, Biology, and Chemistry using Mathematical equations. It is related with calculating the interaction of atoms in motion. MD simulation is a process in which atoms and molecules are simulated using Newtonian physics where scientist can learn how an atom or molecule
shapes changes in interaction. MD is used in diverse area of research where the study of atom motion is required by the application. It works by calculating the trajectory of the atoms interacting between each other and calculating the forces for each atom interaction for every time step and updates the velocity by integrating the forces, and further sending the updated forces and velocity to the next time step. Although all the atom interactions for each time step occurs independently with each without having any relation between time steps.

Molecular Dynamics simulation is used in fields of science such as biophysics, chemistry, bio-molecular simulation, Protein behaviour simulation, material sciences, semi-conductors simulation, and microscopic atom structures. Generally it is a tool to study any kind of particle simulation which involves pair-wise atoms and molecules interaction in any such problem. In biophysics it used to study the physical properties of substances, such as Nano-technological Devices and in biochemistry it is used to observe the molecule interaction to study protein structure. MD simulation is a very compute intensive solution and requires enormous amounts of processing for calculation of forces and velocity, as the problems can be for several million atoms with perhaps thousands of time steps.

Generally all the forces and velocity calculation occurs independently for each atom therefore it is embarrassing parallelism and thus has good potential to implement in parallel to solve the problem faster. The architecture of GPUs facilitate processing floating point calculations and 3D rendering in parallel therefore they are best suited to run simulation when there is embarrassing parallelism while processing application. MD simulation requires independent floating point calculation for tracking the trajectory and requires graphical rendering to display the atom interactions. Therefore MD simulations match appropriately for implementation on GPUs thus the potential is realised by several institutes and companies who have already implemented MD simulation on GPUs using CUDA.

A collaborative work between Nan yang technological institute and university of New South Wales regarding implementation of MD of codes on CUDA was the first of its kind. They have implemented a suitable algorithm for GPU in C++ and CUDA; although the algorithm was tested on NVIDIA GeForce 8800 GTX it gave 15 x speeds. The GPU used was the available GPU on that time however compared to presently available GPUs such as Tesla the algorithm is expected to provide much better speed up [8].

Simulation of Glassy Dynamics on GPU, these category of simulations are very large, time consuming and requires enormous computing power, the problem sizes can range from 1 million to $10^8$ particles. Simulations are implemented for glassy dynamics of a super cooled binary mixture; this simulation requires stability of architecture in terms of longer duration of running programmes and accuracy in double floating point numbers. When simulations are run for glass forming cooling and compression of liquids, minute changes in temperatures are required to be observed because these changes makes drastic impact on dynamics. The simulation is implemented on GPU to
exploit it double floating point accuracy, low power consumption and minimal hardware expenditure [29].

A comparative simulation is implemented on GPU for polymer brushes, these brushes are formed by grafting on a planar surface by using flexible linear polymers or ring polymers. There two distinct lattice models created one of the model is implemented using MD on GPUs for faster implementation and MD implementation is implemented using HOOMD-blue MD library [30].

AMBER MD: Assisted Model Building with Energy Refinement is one of the leading MD simulation parallel libraries in the world. It also consists of GPU version library which supports single and multi-GPU codes, the latest AMBER 11 package that supports multiple GPU have achieved 52 nanoseconds per day speed for 25000 atoms problem [4].

LAMMPS: it is a Molecular Dynamics library distributed under GPL by Sandia National Laboratories. LAMMPS is an abbreviation for Large Scale Atomic/Molecular Massively Parallel Simulator [5]. It can be used to perform microscopic and macroscopic particle simulation for biomolecules, polymer, metals and semiconductors. The code libraries are available for single-core and parallel simulation using MPI (Message Passing Interfaces); a GPU implementation is available as well which can be used on multi GPU cluster system using MPI and CUDA. It is developed in C++ and easier implementation for future extensions.

HALMD: it is CUDA based molecular dynamics package developed for GPU architecture; since it is executed on GPU it provides high precision numerical calculations. It is actually developed for simple and large MD simulations for liquids, it distributed free under GPL licence [6].

HOOMD-blue: It is an abbreviation for Highly Optimized Object Oriented Many-particle Dynamics Blue Edition. It can perform general purpose MD simulation on GPU. Due to its versatility characteristics several MD simulation can be implemented such as Tethered Nano rods on Rigid Body dynamics, Tethered Nano spheres on Brownian Dynamics, super cooled liquids, surfactant coated surfaces on DPD with constraints, polymer Nano composites on coarse-grained MD, super cooled liquids on 2D MD [28]. The library is developed in C++ using Templates which makes it easier for anyone extending functionality to the package which is freely distributed. HOOMD-blue is an extension project to the original HOOMD project at the Ames Lab, University of Michigan [7].

2.4 Related Work

A lot of work has been implemented using GPUs. This has caused many supercomputing companies and server manufacturing companies to seize on its potential in the future to implement much powerful systems with low cost. Therefore
they have plans or have already implemented co-processors (GPUs) on their existing
server hardware.

IBM has also come up with GPU installed on their blade servers; IBM systems
such as IBM “Blade centre” GPU expansion blade [14], IBM iDataplex dx 360 M3 and
IBM intelligent cluster have support for GPUs on them. Other server manufacturers
such as Dell and HP are also integrating GPUs on their blade servers by providing 2
GPU for 1 many core processors. A company called super micro [25] has come up with
server systems dedicated for supercomputing, which are composed of GPU blades.
They provide several different configurations of blade servers with at least 2 GPUs on
each blade. Several other general purpose computing solutions such as graphic
processing, companies using GPUs by providing enormous performance with cost-
effective solutions develop video encoding, etc.

Elemental technologies, a company providing massively parallel video
processing solutions for graphics professional, video editing, video processing and
video encoding. It has developed plugins considering and dedicated to NVidia Quadro
GPU that if available can provide more sophisticated graphic processing to adobe
creative suite applications. It also has a video encoding plug-in which facilitated video
encoding to various formats using GPU power processing a GPU based server, which
can process several video-encoding tasks simultaneously [22].

Digital negative is a company providing visual effects solution for the animated
film industry. They have developed a fluid simulation system in CUDA, which is run
inside a GPU farm consisting of NVidia Quadro and Tesla GPUs. The fluid simulation
is used to simulate various kinds of visual effect by different graphic artists however
with the GPU farm these simulations run faster and provides quicker results to the
graphic designers [27].

SciComp Inc. specializes in computational finance solutions and the energy
industry. They provide parallel solutions for financial derivatives for credit, bonds,
equity calculations, and these parallel solutions are implemented in OpenMP and
CUDA or CUDA C. They have made applications which can read the keywords from
programmers and generate CUDA code which does not require any programming
expertise in CUDA from the programmer. The applications are built targeting NVidia
GPU architecture and the company claims 46% [21] increased performance with single
GPU rather than using multiple CPU work stations or clusters.

GPUs are used for accelerating SQL database operation by offloading the CPU
task and running the queries in parallel on the GPU to generate faster database search
results [13]. This concept provide a huge potential for general purpose computing
database driven applications such as web applications where maximum amount of
latency in the applications is taken for accessing and running SQL queries. Therefore
such kind of acceleration can provide higher throughput to applications and reduce
latency times.
Application framework for GPUs which consists of map-reduce functionality called DisMaRC is developed using CUDA programming language and provides functionality for executing map-reduce functionality on network of GPUs on a massively parallel computer or on distributed computers [15].

GPUs are used to accelerate computer vision applications, which can harness the computational power and floating point precision calculation to perform faster processing for video surveillance systems, motion tracking and analysis and for image processing with better clear projection [20].

Spin model simulation on GPU [26]; the institute of physics at university of Mainz in Germany has developed computer simulation for spin models algorithm on GPU. Spin models are generally used in physics to observe magnetism. They have implemented a double check board metropolis spin model simulation on GPU, the architecture of check board is similar to the design architecture of GPU and therefore they have developed an algorithm which will distribute the checks across all the blocks of the GPU and further will be subdivided among the threads in each block.
Chapter 3

Molecular Dynamics Code Analysis

The serial version of MD code chosen for this project is implemented in C programming language it is actually not been developed from scratch in fact it has been developed by Dr Mark Bull for EPCC. Although this code have been provided as practical exercise for implementing on Host OpenMP.

3.1 Pseudo code

The execution steps of the code are briefly explained in Pseudo code, it show serial execution sequence.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Generate FCC Lattice for atoms inside box.</td>
</tr>
<tr>
<td>2.</td>
<td>initialize velocities and forces</td>
</tr>
<tr>
<td>3.</td>
<td>initialize forces array elements to double precision scalar values</td>
</tr>
<tr>
<td>4.</td>
<td>FOR move = 1 to 20</td>
</tr>
<tr>
<td></td>
<td>i. CALL domove with velocity, forces and FCC Lattice</td>
</tr>
<tr>
<td></td>
<td>ii. CALL forces with FCC Lattice and forces</td>
</tr>
<tr>
<td></td>
<td>iii. CALL mkekin with forces and velocity RETURNING ekin</td>
</tr>
<tr>
<td></td>
<td>iv. CALL velavg with velocity RETURNING avgvel</td>
</tr>
<tr>
<td></td>
<td>v. IF move value LESSTHAN 20 AND EQ 10 THEN</td>
</tr>
<tr>
<td></td>
<td>1. Calculate scales by SQROOT of (ekin times tscale by tref)</td>
</tr>
<tr>
<td></td>
<td>2. CALL dscal with velocity and scales</td>
</tr>
<tr>
<td></td>
<td>3. Calculate ekin</td>
</tr>
<tr>
<td></td>
<td>vi. END IF</td>
</tr>
<tr>
<td></td>
<td>vii. PRINT values of potential energy, viral, kinetic energy and velocity average.</td>
</tr>
<tr>
<td>5.</td>
<td>END FOR</td>
</tr>
</tbody>
</table>
3.2 Code description

The current serial implementation of code consists of MD functions implemented in separate files for better maintainability of the code. The main function invokes each of these functions on execution of the program. The functions implemented in the code are fcc, mxwell, dfill, domove, forces, mkekin, velavg, dscal and prnout. Detailed description of each function is described further in the chapter. Fcc, mxwell and dfill are initialization function which are used to initialize data inside forces, fcc and velocity arrays. These array are vector arrays with size equal to number of particles mentioned in the problem. However array sizes are actually defined as number of particles by 3 which is used to 3 dimensionally define the atoms in x, y and z axis.

Domove, Forces, Mkekin, velavg and dscal functions are invoked iteratively for each time step except dscal which is invoked only once at half time step to scale velocities based on generated scale value. The main initially invokes fcc, mxwell and dfill functions to initialize arrays and later runs time step loop which is used to invoke each function and generate output of the program as shown in Code Block 3-1. Time step considered for program is 20, therefore loop iterates 20 times and invokes each function 20 times to eventually generate values for potential energy, virial, kinetic energy, average velocities and counter.
for (move=1; move<=movemx; move++) {
    /* Move the particles and partially update velocities */
    domove(3*npart, x, vh, f, side);
    /* Compute forces in the new positions and accumulate the virial */
    /* and potential energy. */
    forces(npart, x, f, side, rcoff);
    /* Scale forces, complete update of velocities and compute k.e. */
    ekin=mkekin(npart, f, vh, hsq2, hsq);
    /* Average the velocity and temperature scale if desired */
    vel=velavg(npart, vh, vaver, h);
    if (move<istop && fmod(move, irep)==0) {
        sc=sqrt(tref/(tscale*ekin));
        dscal(3*npart, sc, vh, 1);
        ekin=tref/tscale;
    }
    /* Sum to get full potential energy and virial */
    } // print generated values after 20 time step
    prnout(move-1, ekin, epot, tscale, vir, vel, count, npart, den);

Code Block 3-1: serial implementation of time step loop

3.2.1 Fcc function

This function is responsible for generating Face Centered Cubic lattice atoms inside a box. It populates data array of size number of particles by 3 for x, y and z axis of atom positions, it is invoked once during lifetime of program.

3.2.2 Maxwell function

This function consists of Maxwell – Boltzmann distribution formula implementation, it generates velocity array for temperture samples for array size equal to number of particles by 3 for x, y and z axis atoms. This function is an initialization function and invoked only once during execution of program for populating initial temperture data in velocity array.

3.2.3 Dfill function

This function does simple task of initializing forces array with double precision values, the values assigned to each element of forces array are zero. Forces array is assigned double precision values to avoid errors when double precision values will be assigned while updating forces for each time step. Forces array is a vector of size equal to number of particles by 3 for x, y and z axis.
3.2.4 Domove function

domove function is used to move FCC Lattice particles, partially update velocity and initialize forces to zero for next time step. This function is invoked first in the loop as shown in Code Block 3-1, it is responsible to updates fcc lattice array based on each corresponding velocity and forces elements, further observing at Code Block 3-2 it checks boundary condition for new data generated for each element and according increments or decrements elements of fcc latices array. Next each element of velocity array are updated with each corresponding element of forces array and finally each element of forces arrays is initialized to zero.

```c
void domove(int n3, double x[], double vh[], double f[], double side){
  int i;
  for (i=0; i<n3; i++) {
    x[i] += vh[i]+f[i];
    /*
     * Periodic boundary conditions
     */
    if (x[i] < 0.0) x[i] += side;
    if (x[i] > side) x[i] -= side;
    /*
     * Partial velocity updates
     */
    vh[i] += f[i];
    /*
     * Initialise forces for the next iteration
     */
    f[i] = 0.0;
  }
}
```

Code Block 3-2: domove function.

3.2.5 Forces function

This function is implemented using Newtonian force calculation formula, for pair wise calculation of forces for each atoms interaction by observing trajectory between each pair of atom interaction. It takes arguments as forces, FCC Lattice arrays, boundary condition and coefficient and further calculates forces for each atom based on values of FCC Lattice and finally calculate potential energy and virial. The function consists of two nested loops which iterates through N number of particles with each increment of 3 iterations although the increments of 3 are used for each atom values of x, y and z axis. The data structure is generated in single dimension vector arrays therefore values of x, y and z axis are stored in sequence on every 3 elements on respective vector arrays.

As show in Code Block 3-3 First for loop iterates through each atoms x, y and z coordinates values for FCC Lattice arrays such x[i], x[i+1] and x[i+2] which corresponds to x, y and z axis respectively, ideally each iteration step in array is one particle. The second for loop is nested inside first loop, therefore each iteration of I loop will execute N iteration of J loop. However j loop initial execution is based on value of I loop, this step is done for performance optimization as trajectory observation is not
done for atoms which are positioned behind existing atom. Moreover these interactions can be avoided; ideally the first loop starts with first atom in FCC Lattice array and retrieves x, y and z position of each atom by iterating the loop. Second nested loop having N iterations are executes starting from i+3 for every increment x[j], x [j+1] and x [j+2] values are obtained for x, y and z coordinates of each atom respectively. The atom coordinates obtained for each atom in I loop is interacted with each atom of j loop by iterating through each J+3 iterations of J loop. The interaction values for x, y and z axis are stored in temporary variable for x, y and z axis respectively and further each axis is checked with boundary condition and if value is less than boundary value that axis is incremented by side value or if it is greater than boundary that axis is decremented by side value.

```c
void forces(int npart, double x[], double f[], double side, double rcoff)
{
    for (i=0; i<npart*3; i+=3) {
        xi  = x[i];
        yi  = x[i+1];
        zi  = x[i+2];
        fxi = 0.0;
        fyi = 0.0;
        fzi = 0.0;

        for (j=0; j<npart*3; j+=3) {
            xx = xi-x[j];
            yy = yi-x[j+1];
            zz = zi-x[j+2];
            if (xx<-sideh) xx += side;
            if (xx> sideh) xx -= side;
            if (yy<-sideh) yy += side;
            if (yy> sideh) yy -= side;
            if (zz<-sideh) zz += side;
            if (zz> sideh) zz -= side;
            rd = xx*xx+yy*yy+zz*zz;

            if (rd<=rcoffs) {
                epot    += (rrd6-rrd3);
                vir     -= rd*r148;
                forcex = xx*r148;
                fxi     += forcex;
                forcey = yy*r148;
                fyi     += forcey;
                forcez = zz*r148;
                fzi     += forcez;
                f[i]    += fxi;
                f[i+1]   += fyi;
                f[i+2]   += fzi;
            }
        }
    }
}
```

*Code Block 3-3: forces function.*
The summation of squares of all axes for each atom is calculated and compared with coefficient value if it is less than coefficient value, potential energy variable is incremented and virial variable is decremented. All the axes of each atom of forces arrays are decremented based on interaction values of forces axes. Finally x, y and z axis interaction values generated are incremented by \( f[i] \), \( f[i+1] \) and \( f[i+2] \) respectively to \( I^\text{th} \) element of forces array.

### 3.2.6 Mkekin function:
This function performs operation of scaling forces, updating velocity and calculating value of kinetic energy from updated velocities. It takes forces and velocity arrays as input iterates through each element of forces and velocity arrays, further it scales each element of forces followed by incrementing each element of velocity by updated force values and consequently performs summation of squares of each velocity element. The final summation value is divided by mass to eventually obtain value of kinetic energy and return to calling function, *Code Block 3-4* shows loop for mkekin function.

```c
for (i=0; i<3*npart; i++) {
    f[i]=hsq2;
    vh[i]+=f[i];
    sum+=vh[i]*vh[i];
}
kin=sum/hsq;
```

*Code Block 3-4: loop for scaling forces, updating velocity and calculating kinetic energy.*

### 3.2.7 Velavg function
This function performs calculation of average velocity and generates a count of velocity squares greater than relative velocity. It takes velocity array and iterates through but increments of 3 considering axes of each atom particle, *Code Block 3-5* displays loop which calculates average velocity and counter.

```c
/*
 *  Compute average velocity
 */
for (i=0; i<npart*3; i+=3) {
    sq=sqrt(vh[i]*vh[i]+vh[i+1]*vh[i+1]+vh[i+2]*vh[i+2]);
    if (sq>vaverh) count++;
    vel+=sq;
}
vel/=h;
```

*Code Block 3-5: average velocity calculation.*

### 3.2.8 Dscal function
This function is used to scale each element of velocity based on the scaled value generated using kinetic energy.
3.3 Code performance and execution timings

MD code is already optimized for CPU, therefore most of the loop execution has their own local variables to avoid frequent main memory access and attempts have been made to optimistically utilize cache memories. For instance in forces function there are several use of temporary variables inside loop so that variable are stored on cache memories and time taken to access them will be faster. The code is executed on particle size of 4000, 6912 and 13500 for 20 time steps on single CPU. Further analysis of the code using “Cray Pat”, it advocates that force calculation function takes approximately 99% of execution time for 20 time steps on13500 particles. Cray Pat is a profiling tool used on Cray CCE compiler it generates percentage analysis each function in a program.

![Serial Time Graph]

Figure 3: Serial Execution timings for 4000, 6912 and 13500 particles.

Accordingly after observing Cray pat results, further manual timings were obtained to analyse time taken by each function in the program. Therefore program is executed for particle sizes of 4000, 6912 and 13500, Figure 3 shows execution timings for each function however due to very low execution they are not apparent on the graph due to forces function, which show dominating 100% execution time.
3.4 GPU implementation

As mentioned earlier the code is optimized for GPU and timing achieved in graph Figure 3 is the result of optimization. While calculating forces pair wise interaction between each atom and their trajectory is observed for atoms axes which are above, bottom and forward direction of atom although the atoms which are behind current atom are ignored therefore J loop starts with (i+3). This optimism helps to avoid extra iterations which can be ignored irrespective of result. In fact iterations reduces as position of atom moves forward, that is there could be less atoms to interact with each other until atom position converges towards final atom position. Probably in the beginning interactions are N but as position is moved forward to next atom there will N – I interactions which actually reduces the number of floating point calculations to be performed on CPU. Figure 5 (A) shows view of optimized iteration structure as the position of atoms increments.

Figure 4: Percentage of execution timings for each function on 4000 and 13500 particles.

Figure 5: (A) Atom interactions for optimized code (B) Atom interaction for un-optimized loops for GPU
Generally optimizations made for CPU work wonderfully, but it becomes difficult to run similar optimizations on GPU because of the difference in hardware architecture. Therefore to make the code run well on the GPU some modifications are made inside forces calculation function. As described earlier there are N-I iterations occurring for pair wise calculation of atoms, therefore in order to run appropriately on GPU, instead of starting the second iteration with I it starts with zero therefore the iterations are equal for every atom position that is N iterations. A diagrammatic representation of the iterations for all the atoms is presented in Figure 5 (B).

The modifications are applied to J loop which was supposed to start with (i+3) now starts with 0, consequently now interaction occurs between each atom selected in I loop against each atom in J loop and no interaction is ignored. However to ensure that interaction does not occurs for particles behind atoms even though J starts with zero, a conditional check is done which checks that values of I and J should be unequal while performing pair-wise interaction. Further modifications are made to for computation of forces axes and value of potential energy and virial. Moreover inside J loop temporary axes of forces are calculated and decremented from forces array \( f[j], f[j+1], f[j+2] \) that is x, y and z axes respectively. These operations are eliminated and later after all iterations of J loop are finished incremented value of axes are added to \( f[i], f[i+1], f[i+3] \) for every iteration that is atom of I loop. Eventually due to double calculation of potential energy and virial values inside J loop, final values of potential energy and virial are divided by 2 to get exact result.

![Figure 6: execution timing comparisons between optimized code and code modified for GPU](image)

Eventually modifications made to code have certainly increased number of double precision operations on CPU and consequently has increased time taken to execute the program. Figure 6 shows time taken by modified code on CPU for different number of particles and also compares previous optimized code timings.
Chapter 4

OpenMP Accelerators

This chapter discusses implementation of OpenMP acceleration for linear algebra functions such as vector addition, multiplication, matrix vector multiplication and matrix by matrix multiplication with performance result and comparisons. Further in chapter porting of MD code on GPU using OpenMP Accelerators is discussed in detail with performance results and comparisons. The code is ported stepwise taking each function into consideration and comparing CPU-GPU ratio for each function. Moreover after every progressive step the execution result of each function is merged with previously ported function and comparisons are made with similar number of serial function. Eventually after porting each function further optimizations are applied to the code to generate ultimate outcome of implementation. In this context the term “Accelerator” will be used for GPU as OpenMP Accelerator use this term in their programming context.

4.1 Linear Algebra

The best way to learn any programming language is to implement some codes using the language of choice this provides a better understanding to understand the language syntax and language rules. Linear algebra is very common data structure in scientific computing and therefore to get a basic understanding of OpenMP Accelerator directives linear algebra codes are implemented. The serial codes are implemented in C language and results of ported OpenMP Accelerator codes are executed on thread block size of 128, 256 and 512, further by comparing results against serial codes.
4.1.1 Vector Addition

The first code implemented was vector addition; vectors are initialized on Host followed by addition of vectors on Host and on Accelerator. Moreover for processing on Accelerator three input vectors are copied from Host to Accelerator and resultant vector that is only one vector is copied back from Accelerator to Device. The array sizes considered for the vectors are 10024, 100024, 1000024 and 10000024, Code Block 4-1 (A) shows serial and (B) OpenMP Accelerator code samples for vector addition. The Accelerator code begins with acc_region directive which is used to specify the region on the code which is executed on Accelerator and num_pes clause is used to specify the compiler about the size of each block in this case it is 128 threads per block. For further details of acc_region and num_pes please refer to 2.2.2..1.3.1 section.

```c
/*addition of vector a and b
* and assigning corresponding values to c
*/
for(i=0;i<N;i++)
    c[i] = a[i]*b[i];

(A)
```

```c
#pragma omp acc region num_pes(2:128)\n    acc_copy(cc[0:N]) acc_copyin(a[0:N],b[0:N])
#pragma omp acc_loop
for (i=0;i<N;i++)
    cc[i] = a[i]*b[i];
//end of acc_loop call
//end of acc_region

(B)
```

**Code Block 4-1:** (A) Serial implementation of vector Addition (B) OpenMP Accelerator implementation of vector Addition with acc_region and acc_loop directives.
Figure 7 A and B shows trends for different vector sizes and execution timings on Accelerator based on number of threads allocated for each block. Apparently CPU execution timings are substantially better compared to Accelerator. However this case is true with smaller vector sizes where substantial amount of time is spend between data transfers to and from Accelerator but as vector sizes increases timings and become comparable in size. The primary reason for Accelerator code struggling to perform is the domination of memory copy timings. While vectors are already on Host therefore there is no need for any copies however in case of Accelerator all the vectors are required to be copied to Accelerator process them and copy back to Host.

4.1.2 Vector Multiplication

![Graphs showing vector multiplication CPU and GPU execution timings]

Figure 8: vector multiplication CPU and GPU execution timings for (A) 10024 and 100024 particles. (B) 1000024 and 10000024 particles.

Unlike vector addition vector multiplication performs arithmetic operation of multiplication between two vectors and generates a resultant vector. The noticeable part in the program is the difference of multiplication operation between Host and Accelerator. The problem sizes considered for the vectors are 10024, 100024, 1000024, and 10000024 compared with Accelerator block sizes of 128, 256 and 512 threads. The performance of this code is almost similar to vector addition, the execution timings are better for CPU and primary cause is same. The additional task of memory copies to and from Accelerator so large that it almost matches equally to CPU execution timings. Although GPU performs better than CPU over double precision multiplication but due smaller sizes of vectors this distinction cannot be determined.
4.1.3 Matrix vector Multiplication

Matrix vector multiplication performs arithmetic multiplication between elements of matrix and a vector. A matrix of size MxN is multiplied to vector of size N, each element of row matrix is multiplied by each corresponding element of vector and the sum of multiplication is assigned to corresponding index of vector size M. The M and N sizes for the program are kept equal and the N size of vector is equal to N of the matrix. So the problem sizes considered for matrix are 10*10, 100*100, 1000*1000 and 10000*10000 and the vector size are 10, 100, 1000 and 10000.

Figure 9 shows execution timings for matrix-vector multiplication for different element sizes. The execution timings from 10 to 100 are mostly similar for Accelerator irrespective of number threads per block and CPU shows better timing. Although for problem size of 10000 execution timing on CPU have risen but still better compared to timings on Accelerator. The distinction cannot be made due to dominating amount of memory which has to be copied to the Accelerator from the Host, which results in better speed up for Host against Accelerator. Code Block 4-2 (A) and (B) show Serial and Accelerator implementation of code respectively.

**Figure 9:** Matrix vector multiplication for matrix and vector of equal dimensions (A)10x10 and 100x100 (B) 1000x1000 and 10000 x 10000

```
for(i=0;i<M;i++)
{
    sum = 0.0;
    for(j=0;j<N;j++)
    {
        sum = sum +(mat[i][j] * vec[j]);
    }
    v[i] = sum;
}
```

(A)

```
#pragma omp acc_region num_pes(2:128)
{
    #pragma omp acc_loop
    for(i=0;i<M;i++)
    {
        sum = 0.0;
        for(j=0;j<N;j++)
        {
            sum = sum +(mat[i][j] * vec[j]);
        }
        vv[i] = sum;
    }
}
```

(B)

**Code Block 4-2:** Matrix vector Multiplication code snippet for (A) Serial code (B) Accelerator code
4.1.4 Matrix Multiplication

Matrix multiplication performs arithmetic operation of multiplying each element of row matrix A with corresponding column of second matrix B, the sum of this operation is assigned to similar element of matrix C. Matrix size of MxN considered are 10*10, 100*100 and 1000*1000 and timings generated for Accelerator for different block size of 128, 256 and 512 as show in Figure 10 A and B. The execution timing for matrix sizes of 10*10 and 100*100 are not impressive compared with CPU, however the block size of 512 performs much slower compared to other block sizes on the Accelerator. However, most impressive part to observe for matrix size of 1000*1000 as shown in Figure 10 (B), which is significantly impressive for Accelerator compared with CPU, even though there are memory copies to Accelerator. CPU timings are considerably slower compared with Accelerator timings which are significantly better irrespective of block sizes. This indeed demonstrates the speed GPUs can achieve for double precision calculations compared with CPU which is lot slower. Code Block 4-3 (A) and (B) shows code snippet for serial and Accelerator implementation respectively.

```c
for(i=0;i<M;i++)
    for(j=0;j<N;j++)
        sum = 0.0;
        for(k=0;k<N;k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;

#pragma omp acc_region num_pes(2:512)
    pragma omp acc_loop reduction(+:sum)
    for(i=0;i<M;i++)
        for(j=0;j<N;j++)
            sum = 0.0;
            for(k=0;k<N;k++)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;

(A)  
```

```c
#pragma omp acc_region num_pes(2:512)
    pragma omp acc_loop reduction(+:sum)
    for(i=0;i<M;i++)
        for(j=0;j<N;j++)
            sum = 0.0;
            for(k=0;k<N;k++)
                sum += a[i][k] * b[k][j];
            c[i][j] = sum;

(B)
```

Figure 10: Matrix multiplication execution timings for equal matrix sizes (A) 10 and 100 (B) 100 and 1000.

Code Block 4-3: Matrix multiplication code for (A) Serial (B) Accelerator implementation.
4.2 MD Code implementation

Generally, most compute intensive serial code is best candidate to be ported on Accelerator and after observing at code performance in section 3.3 it is clear that forces function is the most compute intensive function. Consequently forces kernel will be first to be ported on Accelerator followed by Domove, Mkekin, velavg. After porting each function the performance timings of each Accelerator function will be compared with execution timings of serial function and comparisons are performed using ratio of GPU to CPU. Further after each function is ported on Accelerator the summation of timings of number of function ported on Accelerator will be compared with equal and similar number of serial function timings. Each kernel is ported step by step and execution timings and time occupied by memory copies are discussed in detail further in the chapter.

4.2.1 Use of acc_data directive and present clause

Initially when each of the kernels is ported on Accelerator they are invoked inside acc_data region directive and using acc_copy clause for copying data. Acc_copy combines two functions in one it performs memory copies to Accelerator similar to acc_copyin clause and copies memory back to Host and similarly for acc_copyout clause as well, a detailed description of acc_copy clause can be found in section 2.2.2..1.3.1. However use of acc_data region have not proved beneficial to the implementation because data copies to and from Accelerator are occurring after each function call. Figure 11 shows memory copies to and from Accelerator after porting each kernel on Accelerator. It shows time spent on memory copies by GPU and overall time including GPU and CPU processing; The memory copy timings and kernel execution timings are generated using CUDA Textual Profiler and all the values from it is used to calculate execution timing taken only by kernel.

![Figure 11: Time spent performing memory copies on GPU to CPU after porting each kernel using acc_copy clause.](image-url)
Moreover after porting each kernel numbers of memory copies have substantially increased, the primary reason is after every invocation to kernel the arguments passed in kernel are copied from Host to Accelerator and after kernel has finished execution those data are copied back to Host. For instance forces kernel requires forces and FCC Lattice arrays, consequently they are copied to Device, when execution of forces kernel is finished they are copied back to Host along with potential energy and virial results and this happens for 20 times step in the program. Moreover when another kernel such as domove is ported on Device it requires all three arrays forces, velocity and FCC Lattice. Consequently all three arrays are copied to Accelerator and subsequently copied back at the end of kernel call and similarly forces and FCC Lattice arrays are copied back and forth to Accelerator on invocation of forces kernel. Therefore memory copies have increased incrementally as numbers of kernels are implemented on Accelerator and thus advantage of using acc_data region is never fulfilled. The actual benefit of acc_data region is there can be number of acc_region within and data copied to Accelerator can be shared among those Accelerator regions, further details on acc_data region please refer to section 2.2.2.1.3.2.

To reduce memory copies another clause that can be used on each acc_region instead of acc_copy is present clause. The advantage of this clause is it searches for data on Accelerator; this data is specified as argument in present clause and assumes that data is already present on Accelerator. Certainly it helps to reduce memory copies and once data is copied from Host to Accelerator on invocation of acc_data region that data is resident on Accelerator until the end of acc_data region and is persistent across all kernels. Figure 12 show changes in trends of memory copies by using present clause with acc_data region. Apparently it has not benefited forces kernel because due to one single kernel there is no scope of data being shared with other kernels. However as number of kernel: Domove, Mkekin and Velavg are ported, it has certainly benefited in reducing memory copies after each kernel call. The data is persistently shared across other kernels requiring data and copied back after end of acc_data region. The significance of present clause can be advocated as when more kernels are executed on Accelerator and memory copy timings have reduced considerably for GPU time.
4.3 Kernel porting

4.3.1 Forces kernel

This is the first function considered for porting on Accelerator also this is the most compute intensive kernel of all, although for a task to be run on Accelerator all data structures required for processing needs to be explicitly copied to the global memory of GPU and all processed data needs to be explicitly copied from Accelerator to Host for further processing. In case of force calculation kernel, it computes forces and calculates potential energy and viral for forces in each time step. For calculation it requires Face-Centric cubic (FCC) lattice for atoms data array and updated forces array. The code is implemented and run for 4000, 6912 and 13500 particles, which mean that data should be distributed among similar number of threads in the program. However running these numbers of threads simultaneously requires distributing them across blocks that is SMs; they are actually blocks of threads running in parallel across Accelerator. Therefore it require number of threads executing in each block, in this program execution block size of 128, 256 and 512 threads per block are considered.
Figure 13 shows execution results for forces kernel, it appears that almost 63x of speedup is achieved on Accelerator without considering memory copies. Compared to serial implementation of forces kernel as in Code Block 3-3 for Accelerator the code is just included inside acc_region directive to notify the compiler that this part of code has to be executed on Accelerator. In Code Block 4-4 it shows OpenMP Accelerator implementation of forces kernel, we can see there is not enough modification required on actual code, only few lines of preprocessor directives are included to notify the compiler. The acc_region directive notifies compiler that this code is to be executed on Accelerator and the data has to be distributed across threads of size N with thread block size mentioned in num_pes clause.

The iterations of first loop iterations are distributed across each thread equal to size of N. Each thread executes N iterations to interact with all N particles in the problem. Each thread creates its own local variables which is intended so that the registers on the GPU should be utilized. Each thread locally calculates potential energy and viral and later all the local values of threads are added to perform global sum by using reduction clause on potential energy and viral variables.

The particles are distributed across the threads and threads spread in blocks of threads, each thread block has a chunk of particles data, for instance in case of block size of 128 threads, the data specific to 128 particles will be assigned to the block. If the block is the first block which will have block index of 0 should have particles ranging from 0 to 127 assigned to threads of similar thread index. Threads in each block have their own thread index specific to each block therefore to identify the thread index across entire threads in the grid a formula is applied to uniquely identify each thread.
void forces(int npart, double x[], double f[], double side, double rcoffs)
    //initialize variable

/* acc_region starts here */
#pragma omp acc_region present(f[0:n3], x[0:n3], side, epot, vir)
    acc_copyin(rcoffs, npart) num_pes(2:128)
{
#pragma omp acc_loop reduction(-:vir) reduction(+:epot)
    for (i=0; i<npart*3; i+=3) {
        xi = x[i];
        yi = x[i+1];
        zi = x[i+2];

        for (j=0; j<npart*3; j+=3) {
            //check for value of j and i
            if (j!=i) {
                //code for forces calculation
            }
        }
        //end of time step check condition
    } //end of loop
    for (i=0; i<npart*3; i+=3) {
        f[i] += fxi;
        f[i+1] += fyi;
        f[i+2] += fzi;
    } //for loop ends here
} //acc_region
epot /= 2;
vir /= 2;
}

Code Block 4-4: kernel forces code implementation.

Generally number of threads per block variances also causes variances in execution timing of the code on Accelerator. The execution timing varies marginally for block sizes of 128, 256 and 512 as shown in Figure 13. The primary reason is the number of threads running in parallel, in CUDA when a block of threads is allocated to SM the threads are divided into chunks of Warps of 32 threads each. These Warps of 32 threads per SMs are formed into grid and executed in parallel. When threads block sizes are increased there are less number of blocks created thus less number of SMs allocated per grid for the Kernel to execute. When less SMs are allocated per grid then there will be less number of threads executing in parallel, as per SM can execute 1 warp that is 32 threads and further dispatch to other waiting threads.

Ideally as number of particles in program are increased the differences in execution timings for different block sizes changes significantly to almost similar timings. When number of particles are increased number of threads to process them increases as well. As number of threads are increased, number of thread blocks increases, therefore number of Warps to be executed increases which apparently increases possibility of more number of threads to be executed in parallel. Although current Fermi Architecture supports 2 Warps of size 32 threads can be executed
simultaneously but when data structure is double precision then only one Warp per sm can be executed.

When the forces Kernel is executed on Accelerator it requires data to be copied from Host to Accelerator and later processed data is copied from Accelerator to Host. Moreover forces Kernel is called 20 times in program to execute 20 time steps therefore forces and FCC Lattice arrays needs to copied to and from Accelerator 20 times. Due to data copying, significant amount of execution time is accumulated overall. Therefore comparatively even if the kernel executes faster but there is also time spent in memory copies to and from Accelerator. Figure 13 shows ratio of execution time taken only by kernel excluding memory copy timings on “kernel only” trend. The plotting is implemented for 13500 particles on 128 threads per block. We can clearly observe the trend of execution timing for “kernel only” is substantially less compared with overall execution timing which includes memory copies as well.

4.3.2 Domove Kernel

Domove is second Kernel ported on Accelerator, this Kernel moves particles for the next time step and partially updates velocity array. Although it does not have a significant impact on the execution time of the program but data arrays updated in this function are required to copied to Accelerator for the forces Kernel for each time step. This should assist in reducing number of memory copies to and from Accelerator. There are now two Kernels executing on Accelerator both Kernel calls are included inside acc_data region directive. The implementation and advantages of using acc_data region directive are discussed in sections 2.2.2.1.3.2 and 4.2.1 respectively. Consequently this should facilitate to share data structures already present on Accelerator between domove and forces kernel and updates to memory done by one acc_region code is visible to another acc_region code.

Figure 14: Domove and forces kernel timings combined together for ratio of GPU to CPU time and execution timing plotted only for kernel execution timing excluding memory copies for 13500 particles on 128 Threads per block
When acc_data region is invoked, three main arrays are copied to Accelerator from the Host, domove Kernel requires forces, velocity and FCC arrays for execution. Due to increase in another array to be copied compared to forces Kernel which required 2 arrays there is increase in memory copy activity. For each time step these copies will occur to and from Accelerator and therefore have increased memory copy time as well. In Figure 12 we can compare forces and domove Kernel memory copies and also individual memory copy distribution is shown in Figure 14 for domove Kernel. Due to additional memory copies we can observe that the execution time ratio for “kernel only” has increased even though memory copy timings are excluded. Infact the timing is equal to overall execution time, the change in trend has occurred because currently there are two kernel executing on Accelerator and memory copies are occurring before and end of both these kernel execution. This shows that the Accelerator is occupied mostly for kernel execution and therefore memory copies and kernel execution timings are proportionally balanced.

Due to memory copies using acc_data region forces Kernel which requires two arrays, however it does not require to explicitly copy two arrays again from Host. Due to presence of arrays already on Accelerator, they are accessed using present clause of acc_region directive. Present clause uses same data copies which are already present on Accelerator unlike acc_copy which requires data to be copied from Host to Accelerator every time it is encountered by runtime. Therefore the use of acc_data and present clause has not made enough difference to the execution time as compared to the execution timings of forces Kernel, but certainly it has added a bit of optimization to memory copies. Eventhough there are two Kernels operating on the Accelerator the data copies required are only once before Accelerator starts executing the Kernels.

The modification applied to domove kernel using OpenMP Accelerator directives is shown in Code Block 4-5 it also show, main function implementation for calling each kernel inside acc_data region directive.
/* domove kernel */
#pragma omp acc_region present(f[0:n3]) present(x[0:n3])
present(vh[0:n3]) present(side) num_pes(2:128)
{
#pragma omp acc_loop
  for (i=0; i<n3; i++) {
    //code comes here
  }
  //acc_loop and loop ends here
//acc_region ends here
\\\\\\\\\\
/* main function implementation of acc_data region
* for calling domove and forces kernel
*/

for(move=1;move<=movemx;move++){
#pragma omp acc_data acc_copyin(f[0:npart*3],x[0:npart*3],vh[0:npart*3])
acc_copyin(side,tref,tscale,hsq2,hsq,h,vaver)
{
    //call to kernels comes here
    domove(3*npart, x, vh, f, side);
    forces(npart, x, f, side, rcoff);
}
}

Code Block 4-5: Domove kernel implementation using OpenMP Accelerators Directives.

4.3.3 Mkekin Kernel

Mkekin Kernel is the third Kernel to be ported on Accelerator, it requires forces and velocity data arrays which have already been copied consequently the data is present on Accelerator. It is called after forces Kernel and included inside acc_data region, although it does not requires any other data to be copied on Accelerator but after Kernel execution updated data needs to be copied back to the Host for each time step. Figure 15 shows overall combined execution timing ratio of GPU to CPU for all three Kernels, there is no change in the trend compared to the previous two kernel porting. However by observing Figure 12, which shows comparisons between all Kernel memory copies, “Mkekin” Kernel takes lowest GPU transfer time compared to overall time and the primary reason is due to presence of data on Accelerator. Precisely it never required data to be copied back and forth for its execution, Figure 15 shows trend for “kernel only 128 threads per block” which considers time spent for kernel execution excluding memory copy timings. Apparently the sole kernel execution timings has risen substantially this proves it has significantly reduced memory copies to Accelerator because the data structure required by the kernel is already on Accelerator.
4.3.4 Velavg Kernel

The fourth and final Kernel ported on the Accelerator is “velavg”, this Kernel is also included inside acc_data region on Accelerator and called after “mkekin” Kernel in sequence. It calculates average velocity for updated velocities for each time step, therefore it requires updated velocities array. Moreover velocities are updated by domove and mkekin Kernels before giving to this kernel, therefore velocity array is already present on Accelerator. The velocities array is not required to be copied from Host to Accelerator for each time step but calculated average velocity values has to be copied back to Host along with other arrays after acc_data region ends.

Figure 15: Domove, Forces and Mkekin kernel combined execution ratio of GPU to CPU time and execution timings excluding memory copy timings for 13500 particles on 128 threads per block plotted for “Kernel only” trend.

Figure 16: Domove, Forces, Mkekin and Velavg kernel combined execution time with ratio GPU to CPU time and memory kernel execution time excluding memory copy time for 13500 particles on 128 threads per block.
Figure 16 shows trend for combined execution timing after porting velavg Kernel on accelerator, execution timing for 128 threads per block for all particles input is similar to the other Kernel timings. However there is slight change in the timings for the block sizes of 256 and 512 threads, this could be the case as the number of memory copies for the Kernel has been reduced and this reduction in memory copies have exposed the Warp waiting times as the number of blocks reduces there are less Warps and consequently threads running in parallel.

Figure 12 shows trends for velavg Kernel comparing other Kernels, apparently after porting velavg Kernel GPU transfer has reduces significantly due to presence of entire data and all Kernels performing execution on Accelerator followed by data copies back to Host. Moreover overall transfer time shows significant increase and takes maximum share of the overall memory transfer timing. All Kernels performs execution on the Device and then data is copied back to Host. Due to presence of required data on Accelerator there have been less data copies compared to previous Kernels. Figure 16 showing trend for kernel execution time which excludes memory copy timings, as we can observe due to reduced overhead in memory copies to Accelerator the kernel execution timings ratio has increased substantially.

4.4 Optimized MD code

Finally all vital kernels of the code are ported on Accelerator and we have seen trends of execution timings and memory copies after step wise porting each kernel on Accelerator. Although, even after applying optimization memory copy timings are still struggling and consuming maximum amount of execution time, consequently yet there are 20 memory copies occurring for 20 time steps in the program. The reason for 20 memory copies is that acc_data directive still lies inside time step loop therefore there are memory copies for every time step. Each subsequent kernel requires data updated by previous kernel for further processing. Also Dscal function is still executing on Host and it requires updated velocities array for scaling therefore if acc_data region is moved outside time step loop then scaling of velocities done during 10th step by Dscal function cannot be done which will affect ultimate results.
In previous section we observed timings after porting each kernel on Accelerator and comparing with summation of timings of similar serial functions. The final step of optimization is to include entire time step loop inside acc_data region directive. The benefit of this technique is the data required for processing should be copied to Accelerator only once before the loop is invoked and eventually when processing is completed data not required by Host should not be copied back; in fact what is required is the ultimate resultant data of MD calculation. As mentioned earlier MD code requires FCC lattice, forces and velocity arrays to perform calculation of potential energy, viral, kinetic energy and average velocity. Finally when these values are generated then input arrays are not required further by Host, however calculated values are obtained and utilized for displaying results of calculation.

Apparently, when acc_data region is invoked, all 3 arrays would be copied to Accelerator however due to presence of time step loop inside acc_data region it would be invoked once during program execution and data would be copied once to Accelerator. Although when all iterations of loop are finished, final values of potential energy, viral, kinetic energy and average velocity should be copied after completion of 20 time steps. In Figure 17, it does not show any significant change in execution timing but certainly there are changes in trends when number of particles increases. When numbers of particles are increased from 4000 to further 13500 there is enhancement in performance for thread blocks of larger sizes 256 and 512, they converge towards similar timing to block size of 128. Due to large number of particles there are more number of thread blocks created which facilitates more Warps executing in parallel and correspondingly more threads executing in parallel. Since data is already present on Accelerator consequently no data copying time is consumed and more threads executing in parallel which results in enhanced scalability and further no large amount of memory copies from Accelerator to Host are required which actually have impacted in performance gains to large thread block sizes.
Figure 18: Memory copy timings for optimized MD code excluding kernel execution timings.

Figure 18 shows time taken by the Accelerator for copying memory to and from the CPU, GPU memory copies has reduced significantly because of larger data arrays are copied only once to Accelerator during first invocation to acc_data region directive and later there are small memory copies back to Host at the end of acc_data region. The data residing on Accelerator are copied to Host but not those input arrays which were copied to Accelerator in the beginning. The overall time is higher due to memory copies to and from Accelerator, the possibility of overhead on overall time could be because at the end of every kernel and acc_data region the Accelerator copies data back to Host and due to slower memory bandwidth of the CPU this copies occur slowly unlike GPU which has faster memory bandwidth.
Chapter 5

CUDA

This chapter describes use of CUDA programming API MD code, serial implementation of Molecular Dynamics code mentioned in section Chapter 3 is considered for CUDA porting. The code was run with particles of size 4000, 6912 and 13500 with CUDA thread block size of 128, 256 and 512. Details of implementations and further applied optimizations are discussed further in the chapter.

CUDA programming API requires explicit declaration of thread blocks in conjunction with size of threads per block on data size. Thread id which should be unique among entire threads across blocks of threads have to be retrieved explicitly and formula to obtain unique thread from entire thread blocks is “threadIdx + blockIdx * blockDim”. This can be done for threads of x, y and z axis and blocks of x and y axis based on the implementation of the program. For this program, serial data structure was created in vectors of single dimension therefore x axis is used for data distribution among each thread on Device presumably threads and blocks are distributed in single dimension.

Each thread in MD program represents one particle in size of number of particles and therefore if number of particles is 13500 then there should be 13500 threads distributed across blocks of threads of equal sizes. The loops in the program run in parallel in CUDA and each iteration in loop is done by each thread, for this unique thread is assigned to begin loop in each SM for every thread. Further increments of loop for each thread is done by size of block by total number of blocks inside grid (blockDim * gridDim).

CUDA should be stored in file with “cu” extension therefore kernel implementation of each MD function is saved in separate “.cu” files. All CUDA files are compiled using NVCC under CUDA compute capability of 2.0 and highest compiler optimization flag, consequently NVIDIA compiler switch --arch=sm_20 is used for compute capability and -O3 switch is used for compiler optimization. The other C files are compiled using Cray CC compiler and NVCC linker is used to link CUDA and C object files.
A wrapper function is created to invoke a C function implemented to perform memory copies and call CUDA kernel calls inside CUDA file. This wrapper is created to avoid compiler errors while compilation because it will be called from the main function which is “C” file and compiled using “C” compiler which does not understand CUDA syntax. The wrapper function is declared extern with “C” to notify the compiler that it is “C” implementation of the function.

The compute intensive function of the code is forces kernel as described in section 3.3; consequently forces function is first function to be implemented followed by other functions in sequence. The implementation of each kernel will provide evidence of scalability of the CUDA code and data copies between Host and Device. Further in section each kernel is implemented on GPU and summation of execution timings of number of kernels on GPU is compared with similar number serial function timings, although comparison of timings are made on ratio of GPU to CPU time. In CUDA context the term ‘Device’ is used for GPU therefore we will use term ‘Device’ while mentioning about GPU for the rest of this section.

![Figure 19: Comparative graph showing CUDA transfer timings after porting each kernel on Device.](image)

### 5.1 Kernel Porting

#### 5.1.1 Forces Kernel

Forces function is the most compute intensive function among MD code and as described in section 3.3 it takes 99% of execution time. It is ported in CUDA in “forcescu.cu” file and named as “kernelForces”. In Code Block 5.1 sample implementation of CUDA forces code is shown, moreover comparing it with serial implementation of forces kernel in Code Block 3.3 the difference lies in declaration of function and loop implementation. CUDA kernels have to be implemented using __global__ keyword to notify NVCC runtime that this kernel can be called from Host and further implementation of loop is different in terms of values assignment and increments. The loop initialization is done by assigning each unique thread id; this
shows that each thread will execute single iteration of loop that is one particle in the program.

```c
__global__ void kernelForces(int npart, double *x, double *f, double side, double rcoff, double *epotArr, double *virArr)
{

//new local variables for epot and vir...

//retrieving thread id
id = threadIdx.x + blockIdx.x * blockDim.x;
//retrieving x dimension of grid by block dimension of x
numT = gridDim.x * blockDim.x;
for (i=tid*3; i<npart*3; i+=numT*3) {
    //obtaining axes for atom[i]
    xi = x[i];
    yi = x[i+1];
    zi = x[i+2];
    for (j=0; j<npart*3; j+=3) {
        //forces calculation code goes here...
    }
}
```

**Code Block 5-1: sample CUDA implementation of forces kernel.**

Forces kernel performs forces calculation by calculating the trajectory between each atom in x, y and z axis. Consequently the result of calculation is each thread generating potential energy and virial locally for its own atom position (as mentioned earlier each thread is one atom particle in the problem). When each thread calculates potential energy and virial, subsequently each thread assigns its local potential energy and virial values to global array of potential energy and virial. Consequently each thread id assigns values to each element of respective potential energy and virial arrays. The sum of each value from potential energy and virial arrays need to be calculated to obtain value of potential energy and virial.

A GPU reduction function is implemented which perform partial sum of values locally for each block consisting of array indexes equal to number of thread inside the block. Subsequently partial sum for each block is assigns to global arrays of potential energy and virial by each block id comprising to each element of arrays, moreover the arrays sizes are equal to number of blocks configured to execution configuration during compilation. Correspondingly when block size arrays are generated they are assigned to another summation kernel which perform sum of arrays on thread id zero serially by running a loop having iterations equal to size of arrays. All the three kernels such as forces, reduction and serial summation are done on Device sequentially; finally when the values for virial and potential energy are generated they are copied back to the Host for further processing. Invocation of forces kernel, reduction kernel and final summation from C wrapper function is shown in Code Block 5-2, another function `CUDAThreadSynchronize()` is invoked after every kernel call. It is used to sequentially execute all kernels one after other or after one kernel finishes execution then another starts executing.
Code Block 5-2: complete kernel invocation for executing forces calculation function.

```
kerneldomove<<<BLOCKS,TB>>>(npart*3,d_x,d_vh_d_f,side);
cudadthreadsynchronize();
kernelforces<<<BLOCKS,TB>>>(npart,d_x,d_f,side,rcodf_d_epot_d_vir);
cudadthreadsynchronize();
reduction1<<<BLOCKS,TB>>>(d_vir,d_epot_d_be,d_bv,npart*3);
cudadthreadsynchronize();
summation<<<1,1>>>(d_be,d_bv,d_se_d_sv,BLOCKS);
```

Figure 20: Kernel forces execution timings for CUDA and kernel execution timing excluding memory copy timing for 13500 particles on 128 threads per block.

Figure 20 shows execution timings for 4000, 6912 and 13500 particles with threads per block size of 128, 256 and 512. The significant difference between CPU and GPU timing seems very obvious, due to substantial occupancy of execution time by forces kernel, porting it on Device has substantially decreased execution timings for all the particles sizes. The execution timings for 6912 and 13500 particles are almost similar irrespective of thread block sizes; precisely block size of 256 is marginally higher compared to other two block sizes. However performance diminishes for particle size of 4000 and significantly for larger thread blocks. On 4000 particles block size of 128 threads performs better than other higher thread blocks due to possibility of larger number of threads executing concurrently. When blocks are configured on Device, 32 threads from each block can execute on the Device in parallel for double precision variables. These 32 threads form Warp on each block, therefore higher number of blocks results in higher number of threads executing in parallel in multiples of 32 consequently lower number of blocks and higher number of threads in each block will result in lower number of threads executing in parallel. In case of 128 threads there could be 31 blocks generated on the Device which will result in simultaneous execution of 31 warps by 32 threads. Although for 256 and 512 the number of blocks generated could be approximately 15 and 7 respectively which might result in less number of
warsps and subsequently less number of threads executing concurrently which ultimately results increase in execution time.

Moreover for calculating values of potential energy and virial, forces kernel uses FCC lattice and moved forces values. The data structures containing these values are present on Host and to utilize those values during execution these data structures are required to be copied from Host to Device. In addition after performing calculation updated data structures are required to be copied back to Host from Device so that data can be further utilized by other function for further processing. We are calculating forces for 20 time step therefore forces kernel will be called 20 times and consequently memory copies to and from Device will also be carried out 20 times. We can observe the trend in Figure 20 showing kernel only execution time excluding memory transfer timings for 13500 particles executed on block size of 128 threads for forces kernel. The trend shows the ratio of execution timing is higher than overall kernel execution timing even though there are memory copies occurring. The forces kernel takes 99% of execution timing as described in section 3.3 so even there are memory copies the kernel processing timing is dominating amongst them. Figure 19 shows memory copy timings for GPU and overall time after porting forces kernel on device.

5.1.2 Domove Kernel

In fact Domove function is first function to be called in the program but generally the most compute intensive part of code is targeted to be ported on Device and therefore forces kernel was considered first to be ported. The functionality of Domove function is to move particles for next time step and partially update velocities, moreover for execution it requires forces, FCC lattice and velocity arrays. Although this function does not takes substantial amount of execution timing and also does not makes crucial impact in increasing performance of code in terms of execution timings. In Figure 21 we can observe there is no substantial change in the trends in scalability as the trends looks similar to Figure 20. But the noticeable part is a marginal decrease in performance for 4000 particles on thread block size of 512 moreover, rest of the trends are similar compared to previous forces kernel results.

As mentioned earlier Domove kernel requires forces, FCC and velocity data structures for calculation these memory copies are required to be achieved before Domove kernel is invoked. However in previous forces kernel implementation, two arrays were copied to Device and brought back, but for Domove an additional velocity array is required. In Figure 19 we can observe increase in GPU transfer time due to increase of another memory copy to and from Device as compared to previous forces kernel implementation. The noticeable part of this kernel implementation is, even though there are two kernels ported on Device but amount of memory copies haven’t increased substantially. The primary reason is before execution of Domove kernel all memory copies are done to Device and after execution of Domove kernel, forces kernel requires those arrays but this time it does not need to copy explicitly from Host because the arrays are already present, consequently this has reduced one memory copy. Therefore in Figure 21 even though memory copy has increases the kernel only
execution ratio has increased as well, precisely there are two kernels executed on Accelerator: domove and forces. The memory copies to device are occurring before invocation of these kernels and copied back to Host after execution of these kernels is done. However there are no memories copy between them and the data present on device is utilized therefore execution timings has over shadowed memory copy timings in overall execution timing.

![Figure 21: Domove and Forces kernel combined execution timings ratio GPU to CPU and kernel only execution timing excluding memory copy timings for 13500 particles on 128 threads per block.](image)

5.1.3 Mkekin Kernel

Mkekin kernel is 3rd to be ported; this kernel scales forces, updates velocities and calculates kinetic energy for every time step. The calculated kinetic energy is required to be copied back to Host for every time step. This kernel first scales forces, updates velocities and further calculates the squares of velocities. The square of velocities is accumulated in a global square array which takes square of velocities for each thread index. Further another kernel is invoked which performs GPU reduction for squares of velocities. Moreover reduction sum of each block is assigned to each index of global array of size equal to number of blocks. One noticeable fact observed after implementing reductions for square arrays is that, sometimes the code behaves abnormally and results are not accurate, perhaps the Device is not able to allocate enough number of threads to perform execution1. Moreover further summation is performed by invoking a summation kernel with 1 block and 1 thread size which perform a serial sum of elements of the array on one single thread.

---

1 GPU Reduction of array for Mkekin kernel was not consistent, the results were sometimes wrong or junk but most of the times the results were generated correctly and plot timings were generated considering only correct values of output.
In Figure 22 we can see combined execution timings after implementing Mkekin kernel along with Domove and forces kernel, it shows significant decrease in performance for particle size of 4000 compared to previous porting and it is almost similar across all thread block sizes. The primary reason is due to less number of threads, there are fewer blocks created, consequently there are fewer warps running simultaneously and additionally the input arrays and resultant values are required to be copied back to Host. As we have observed in 4.1 of linear algebra that data copy impact is higher when data is smaller and which should be copied to and from Device. However in Figure 22 we can see as number of particles is increased performance starts improving and it give much better performance on 13500 particles which shows significant improvement compared to previous porting and this performance improvement is consistent across different block sizes.

Moreover in terms of memory copies Mkekin kernel requires velocity and forces arrays for calculating kinetic energy, however due to presence of both arrays on Device it facilitates in eliminating memory copies from Host-To-Device for velocity array. The array is already been copied on the Device before invocation of Domove kernel this help to avoid unnecessary memory copies for each time step. In Figure 19 we can see benefit in terms of memory copies for Device, GPU transfer timing have increased marginally compared to previous two porting attempts. Although, overall transfer timing has increased substantially but minimum contribution is from GPU transfer time. Moreover in Figure 22, the trends for kernel only ratio is constant and similar to previous trend. The reason for constant behaviour of kernel only trend is due to the fact although there is no memory copies to device due to presence of data but also Mkekin kernel does not requires significant execution time which results in no significant changes in processing time.

![Figure 22: Domove, Force and Mkekin combined execution timings ratio of GPU to CPU time and kernel only execution timing excluding memory transfer time on 13500 particles on 128 threads per block.](image-url)
5.1.4 Velavg kernel

This is the fourth kernel to be ported on the Device after Mkekin kernel; it calculates average velocity from the updated velocities for each time step. For performing this function it requires updated velocities for each time step, since velocities are already updated by Domove and Mkekin kernels therefore velocity array is already present on the Device and thus does not requires memory copies for each time step.

Average velocity is calculated by calculating summation of square roots of x, y and z axis and assigns the square root to global square root array index for each unique thread (atom) index. It also assigns value 1 or 0 to each thread index of global counter array by comparing square root value for each thread index. The technique of global square root array and counter array is used to calculate overall sum of entire square root and counter values generated for each thread index. Further two kernels are launched simultaneously which performs GPU reduction for square root and counter arrays, moreover simultaneous execution is used because both kernels require two different data arrays present on Device and this will help to enhance more Device utilization. After successful completion of both reduction kernels another summation kernel is invoked this kernel sums all values of counter and square root arrays and calculates average velocity on square root sum. However this kernel does not spans array elements across threads instead one single thread runs a single instance of serial loop to perform summation so execution configuration parameters use for invoking this kernel was 1 block and 1 thread.

![Figure 23: Domove, Forces, Mkekin and Velavg combined execution timing ratio GPU to CPU and kernel only execution time for 13500 particles on 128 threads per block.](image)

Figure 23 shows GPU to CPU ratio of combined execution timing of Domove, forces, Mkekin and velavg kernel. There is not any significant influence on performance because this kernel does very little and less time consuming functionality of program. Although the idea behind porting this kernel is to perform all vital
operations on Device to avoid frequent data copies and subsequently perform memory copies back to Host. *Figure 19* shows trends in memory copy timings after porting Velavg kernel along with previous ported kernels although kernel execution timings are not considered, the graphs clearly shows significant increase in overall memory copies compared to all the previous kernel porting, consequently it has impacted overall and GPU transfer times. The primary cause of increase in Device memory copies is from Device to Host as value of average velocity and counter has to be copied to Host for every time step. Although due to memory copies CPU transfer timings have certainly increased but comparatively they are not substantial, this proves the data sizes copied from Device to Host are not large to impact CPU transfer timings which indeed affect overall time.

In *Figure 23* it shows kernel only execution timings which exclude memory copy timings for combination of Domove, force, Mkekin and velavg kernels. We know that there are no further memory copies from Host to Device because required arrays are already on Device. The kernel only execution timing is constant what it was in *Figure 22* because as mentioned about this kernel does not performs any time consuming task therefore it does not influences kernel only timing to increase.

### 5.2 Optimized MD code

Eventually all four kernels are ported on Device and due to that several memory copies for each time steps are eliminated. However there is still one more kernel remaining to be ported on the Device but this kernel is called only once in 20 time step it performs operation of scaling velocities based on value of kinetic energy calculated on Device. Moreover even after porting almost all kernels on Device still there is substantial amount of memory copies occurring to and from Device which has to be eliminated.

By observing output of program, ultimately the values required are potential energy, virial, kinetic energy, average velocity, counter and scaling for scaling velocity after half time steps. In fact input array for forces, FCC lattice, and velocity are updated for each time step and provided as input to next time step to calculated values for updated values. However if these input arrays are already present on Device they can be updated there and also can be provided as input to next time step and eventually benefit in avoiding memory copies back to Host because these arrays are just input arrays and are not further required once values are generated for all time steps.

The optimizations applied to code are:

1. The time step loop present in main function which invoked the “forcesGPU” wrapper function has been removed from main function and placed inside “forcesGPU” wrapper function.

2. Forces, FCC lattice and Velocity arrays are copied to Device before invoking time step loop.
3. Another function for scaling velocities after half time step is ported on Device; eventually there is no operation of MD done on Host except for initializing input arrays.

![Figure 24: Ratio of execution timings for optimized MD code on CUDA.](image)

In spite of fact that optimization 1 seems almost similar to previous implementation such as all kernels are called for 20 time steps and only visible difference is instead of forcesGPU invoked for 20 time steps the loop itself is invoked inside forcesGPU. The advantage of this technique is that input arrays are copied to Device before time step loop is invoked. Due to presence of all three input arrays on Device they would be updated for each time step and consequently updated arrays will be utilized by all kernels for each time step. Although after completion of all time steps, resultant values are copied back to Host but input arrays are not require because eventually our requirement is calculation of values therefore arrays are not required to be copied back to Host and thus memory allocated for those arrays on Device can be freed by invoking “cudaFree” routine on respective arrays.

Even though kernelDscal is called only once in the lifetime of the program but the benefit of porting this kernel on GPU is due to the fact that it requires velocity array and kinetic energy value for scaling velocities. If the kernel would not have ported then memory copy from and to Device for velocity array would be required. Another noticeable implementation observed while implementing kernelDscal was, implementation of CUDA memory copy “Device-To-Device”. KernelDscal requires value of kinetic energy calculated for 10th time step and the values is present inside Device array. Therefore to pass this value as argument to kernelDscal another new CUDA variable is created and the value of kinetic energy present on the Device is assigned to new variable by using memory copy from Device-To-Device. In Figure 24 we can observe optimization has impacted marginally to the scalability of program however it has certainly impacted memory copies between Host and Device. In Figure 25 we can observe that overall transfer timings have decreased substantially and due to
single memory copy to Device, GPU transfer time takes approximately 25% of time compared to overall time which contributes 75% of transfer time composed of memory copies for resultant values from Device to Host.

*Figure 25: Memory copy timings for optimized MD code excluding kernel execution timings.*

Eventually above porting and final optimization provides several evidences such as CPU memory bandwidth impacts the performance while making memory copies from Device to Host unlike GPU memory bandwidth which is 10 times faster than CPU. However when memory copies are reduced the performance of application should certainly increase in terms of program with large sized data structures. Hence another advantage realised by porting all kernels on Device is double precision values calculated were more accurate compared to CPU and GPU are much faster in performing double precision calculations which one of the advantage and characteristic of GPU over CPU.
Chapter 6 Comparing programming models

In previous sections we have seen two different ways to program GPUs using OpenMP Accelerator directive and CUDA further we have witnessed that by porting a serial code on GPU, provides enormous amount of performance improvement. Moreover several other facts we witnessed is that the impact of memory copies are mostly similar to execution of both programming methodologies and optimizing these memory copies has significantly facilitated transfer times. Well GPUs are known to be possible replacement for applications requiring several numbers of multi core CPUs however we are aware about downside about memory copies required from CPU to GPU for running applications on GPUs. Moreover we should also be able to determine which porting is faster in terms of programming flexibility, portability and time spent on implementation. Host OpenMP is another programming model which is a shared memory programming model used to program multi core CPUs. Further in chapter we are going to compare results of OpenMP Accelerators, CUDA and Host OpenMP to determine conclusions about each of them.

6.1 Comparing CUDA and OpenMP Accelerators

In Chapter 4 and Chapter 5 we have discussed in details about implementation of OpenMP Accelerators and CUDA respectively, also we have generated several results on execution timings and memory copies for each ported kernel and further optimizations applied to entire execution of code. Both GPU programming models supports C programing like syntax in fact OpenMP Accelerators actually uses C pre-processor for directives thus they are directly used inside C files. Unlike CUDA which makes it mandatory to store files with CUDA code with “cu” extension.

In CUDA implementation, to parallelize loop each unique thread or block index has to be retrieved explicitly using CUDA API as we can see in Code Block 5-1 thread index and block index are explicitly retrieved. However in Accelerator implementation this functionality is achieved implicitly by compiler, the programmer just needs to specify parallel region using directives as shown in Code Block 4-4. However this functionality of obtaining each executable thread index and block index is not yet implemented in Accelerator but this functionality proves beneficial while debugging code and if one needs to know what values are distributed or processed by each thread therefore this functionality was useful while implementing CUDA code.
In terms of memory copies both programming models have homogenous functionality but methods are distinct. CUDA provides keywords to explicitly notify compiler which type of memory copies are required to be done such as Device-To-Host, Host-To-Device and Device-To-Device, similarly for Accelerators there are clauses \texttt{acc\_copyin}, \texttt{acc\_copyout} and \texttt{acc\_copy}. Actually Accelerators do memory copies implicitly as well but it is use full to perform memory copies explicitly to avoid irrelevant memory transfers. In CUDA prior to invocation of kernel all the data required should be copied to Device and during execution the kernel uses data which is already present on Device and there is no need to explicitly mention about it. However in case of Accelerator it has to be explicitly mentioned which data to use otherwise it just copies data from Host which results in unnecessary memory copies. In section 4.2.1 we have observed differences it makes if the compiler is not explicitly notified about using resident data on Accelerator.

CUDA facilitates memory copies within Device using Device-To-Device, this facilitates to utilize data which is updated on Device to be passed as argument to another kernel call without copying back to Host, moreover the benefit of using this memory copy is described in section 5.2 when the value generated on Device has to be passed as argument to another kernel call. However in Accelerators this benefit cannot be claimed because to pass a resultant value as argument to another kernel it is required to be copied to Host and then passed as argument.

Parallel reduction on GPUs is one of most vital part for summing large arrays across blocks of threads because summing these arrays on GPU in parallel contributes to increased performance and accuracy in double precision values. Reduction in OpenMP Accelerators is simple, programmer does not require to hand code reduction it is implemented using reduction clause on \texttt{acc\_loop} directive. The programmer simply need to specify type of reduction (subtraction or addition) followed by name of variable on which reduction values should be assigned and operation has to be performed. However in CUDA reduction is not built inside API, programmer is responsible for implementing reduction code although there are code samples provided inside NVidia SDK [9]. The programmer himself had to implement reduction code and for this he should have good level of understanding because each block of thread performs local reduction for threads inside it and local reduction of these blocks should be collected into global array of size equal to number of blocks inside grid. The global arrays should be further processed on Device to invoke another reduction but here execution configuration of kernel should divide block size array into number of block by size of thread per blocks. Further after invoking another reduction it would generate new array equal to size of number of blocks generated in new execution configuration. Finally smaller array generated can be summed using single thread on Device as shown in Code Block 5-2, therefore this is a three step process and requires effort from programmer to initially identify size of first array, secondly size of reduction array equal to number of blocks, thirdly size of second reduction array with changes in execution configuration parameter and finally invoking serial summation on GPU. It is a complex process and requires attentiveness; a minute mistake could prove disturbances in code unlike Accelerators where reduction is handles implicitly by compiler.
While implementing MD code on GPU, Accelerator code was quicker and easier to implement unlike CUDA which required enough understand ability and expertise. Moreover the amount of rework in terms of making code portable on GPU was more compared to Accelerators therefore it took more time implementing code in CUDA. These comparisons are apparent but each of these models certainly has several advantages and disadvantages. Such as CUDA supports NVidia architecture but its performance is much better due to its homogeneity with the architecture. OpenMP is architecture independent however it is still under development and bit slower but certainly there will be improvisations in future versions by the time it is released.

6.1.1 Comparing execution timings

Now we have two versions of MD code implementation on GPU one ported in CUDA and second ported in OpenMP Accelerators. The optimized codes are run on 128, 256 and 512 threads per block sizes on particle sizes of 4000, 6912 and 13500. The results of these tasks have already been shown in sections 4.4 and 5.2 however for this section we have chosen 128 threads per block execution timings for both code implementations. 128 threads per block execution timings are best performing timings for both the codes.

In Figure 26 it shows performance of CUDA code is significantly better compared with Accelerator, even though there is significant difference in execution timings for both the codes but both codes behaves similarly for particles size of 4000. Execution timings have risen because of less number of threads running in parallel; due to lower number of particles there blocks generated for 128 threads are less compared to other larger particle sizes. Therefore creation of few blocks there are less warps created consequently less number of threads running in parallel.

![Figure 26: Ratio of execution timings for CUDA and OpenMP Accelerator for block size of 128.](image)

This shows that even though warps concept is part of CUDA architecture, not only it impact CUDA performance but also it has similar impact in OpenMP
Accelerators. Moreover as number of particles are increased both codes starts performing better, on CUDA execution timings for 6912 and 13500 particles are mostly similar but there is marginal change of timings on Accelerator timings for 13500 particles compared to 6912 particles. Eventually after observing results for both models we should be able to conclude that CUDA performance is better than Accelerators and in spite of complexity in programming and low level interfaces of CUDA it provides better performance than Accelerators. One reason could be possible for lack of Accelerators performance is the language is still under development and there still lots of things to be done to bring it to stable state however till date CUDA performance has proved better for this implementation.

6.1.2 Comparing Memory copies

In Figure 27 memory copies to and from GPU are plotted for MD code implemented in CUDA and Accelerators, kernel execution timings are excluded and only memory copies are considered. Apparently there is substantial distinction of memory copies between these two models, even though code is similarly implemented CUDA memory copies are very less after optimistically applying memory copies by transferring required data to Device before invocation of time step loop, and later transferring only required results and consequently ignoring input arrays which are not required further. However similar kind of optimization is applied to Accelerators but still there are sufficient amount of memory copies occurring to add up to execution time although not significantly but marginally.

The extra memory copy overheads in Accelerator implementation is because every time a function finishes execution, resultant values generated are copied back to Host but these memory copies are smaller and not in terms of input arrays which are never copied back. The proof that memory copies from GPU were smaller can be observed by looking at GPU time for OpenMP Accelerators which is minor unlike Overall time which is substantial. GPU memory copies time is consumed by copying

![Figure 27: Comparison of CUDA and OpenMP Accelerators memory transfer timings for optimized MD code.](image)
Back resultant values from GPU to Host for each time step but comparatively amount of overall time is substantial because initially input arrays are copied from Host to GPU before invocation of time step loop therefore one part of memory copy time is consumed make large amount of memory copies from CPU to GPU. Secondly another part contributing to time consumption is when resultant values are copied back to Host from Device for each time step. Moreover all these contribution have dominated overall memory copy times because CPUs have lower memory bandwidth which takes time copying data to and from GPU unlike GPU which have higher memory bandwidth. Even though there is smaller memory copies made by GPU for every time step, but times spend on these copies are very less due to higher memory bandwidth. Therefore in this section we are able to make distinctions in CPU and GPU memory bandwidth which has impacted in MD code for OpenMP Accelerators.

6.2 Host OpenMP, Accelerators and CUDA

Host OpenMP is the first choice programming language for shared memory and Multi core CPU based systems. It is directive based approach and OpenMP Accelerators are GPU extensions to these directives. Each core is termed as thread and it spawns number of threads mentioned during compiled time, numbers of threads are mostly equivalent to number of cores on a CPU or on a shared memory machine. In this context we have implemented a Host OpenMP version of MD code, which is executed on 12 threads because Puffin machine has AMD Magny Cours processor which consists of 12 cores and we have attempted to utilize all cores because comparisons have to be made with GPU which has hundreds of core on it.

![Figure 28: Parallel to CPU ratio of execution timings for Host OpenMP, CUDA and Accelerators.](image)

CUDA and Accelerator timings are considered for 128 threads per block because this is the optimum performance gained from both of these models. In Figure 28 we can observe there is substantial difference between Host OpenMP and GPU programming models CUDA and Accelerators. We have already seen in section 6.1 on scalability of CUDA and Accelerators but both GPU programming models have
performed substantially better compared to CPU programming model. Although considering overheads of memory copies from Host to Device for GPU but still performance is impressive. Although Host OpenMP does not have any overhead of memory copies the data is resident on Host memory unlike GPUs which requires memory copies back and forth.

Certainly we have seen performance of GPUs is impressive in this context and CUDA code has provided optimum performance. Although in terms of economies of cost maintaining and using GPUs is cheaper compared to large amount of CPUs because to create a shared memory machine which could provide optimum performance there should be several numbers of commodity CPU used and configured into shared memory system. Moreover maintaining this system applies cost as well in terms of power and operational cost however maintaining a GPU based system is cheaper in this context because a single or dual GPU system can be easily configured on a single dual core or quad core commodity CPU depending upon requirement of application on a single system. Also in terms of power efficiency and operational cost it is cheaper because a single CPU does not require enough power and GPU by default requires less power compared to a CPU and operating such small system is reasonable.
Chapter 7

Conclusions

In this dissertation we have analysed two programming model for GPUs such as: CUDA and directive based programming language OpenMP for Accelerators. We have considered serial C Molecular Dynamics code to be ported on CUDA and OpenMP for Accelerators. The code is implemented and executed on Cray Puffin machine consisting of AMD Magny Cours (12 cores) processor with 16GB RAM and NVidia Tesla C2050 GPU with 3GB RAM. The codes are compiled on Cray CCE compiler and NVCC compiler for OpenMP Accelerators and CUDA respectively. The ultimate results of the code achieved 124X and 65X speed up for CUDA and OpenMP Accelerators respectively.

The initial analysis revealed that the code optimized for CPU was not suitable for GPU porting therefore several changes were made in the code and this observe substantial decrease in performance of the code which precisely increased approximately 120% of execution time on single CPU compared to previous version.

Initially Accelerators were implemented on simple linear algebra codes on vector and matrix operations however due to data transfers from CPU to GPU and GPU to CPU the performance of serial codes were far better compared to GPU. Moreover during this implementation the fact realized is the data transfer timing mostly overshadows execution timings on GPUs. Further MD code was implemented in step by step process, by porting one kernel and observing the execution timing and as the numbers of kernels are ported their execution timings are summed to compare with summed timings of similar serial functions. It was realised that maximum speed up have been achieved by porting forces calculation kernel alone because this was the most compute intensive function of the code however further kernel implementation did not significantly impacted substantial gains to performance. But the vital differences realised were in terms of memory copies as the number of kernel went on increasing on kernel there were reduction in memory copies to and from GPU as the data present on GPU was updated and utilized among kernels. considering these factors the scope for optimization have been revealed subsequently the optimizations were applied by performing memory copies before the kernels are invoked this benefited in copying input arrays to GPU once and all the processing are done on GPU and only the
resultant values were copied back which certainly reduced substantial memory copies and added benefits to overall execution of program.

Next the MD code was implemented in CUDA similarly by porting one kernel at one time on GPU and then combining the total timings of number of kernels on GPU compared with total timings of similar number of serial functions. Similar to Accelerators as the number of kernel are ported on device they did not significantly contributed to the execution timings but certainly they contributed in reducing memory copy timings because the data present on GPU was utilized among the kernels. moreover when realizing that after kernel have finished execution the data are required to copied back from GPU to CPU but if all the processing is done on GPU then only the resultant values are required on CPU and not the input arrays. Therefore optimization is applied similar to Accelerators by copying all the input data to device before invocation of kernels and performing all the operation on GPU and eventually copying the required resultant values back to CPU. This helped to increase approximately 5% of performance to overall execution timings.

Further when the performances of CUDA and Accelerators proved that CUDA code performed well compared to Accelerators code however also some features of CUDA proved beneficial which are not yet implemented on Accelerators as the language is still under development and it is expected to provide better performance when it will be released. However till now CUDA performance is better due to its maturity and familiarity with NVidia architectures because it is actually meant for the same. But compared to maintainability and ease of programming Accelerators are better because the changes can be made on same serial code and programming them are easier due implicit parallelism handled by compiler. Unlike CUDA which requires separate files to be maintained for its code and programming in CUDA requires better understanding and experience of the language.

Finally evaluations are made by comparing Host OpenMP language with both the GPU programming models and it is concluded that GPU models performs significantly better than Host OpenMP with addition benefits such as: cost of maintenance, energy consumption and simplicity of Architecture because GPU can be small blade server with comparative reasonable cost which consumes less power compared to a multi CPU shared memory machine which is more expensive moreover cost of maintenance and power consumption is higher as well.

Due to advancement in memory transfers in Fermi with latest version of CUDA where CPU and GPU memories are implemented as one global memory called Unified Virtual Address there is interest around to exploit them in General purpose computing.

7.1 Further Work

Further enhancements with several new features in OpenMP for Accelerators are certainly going to be released in future. Therefore it would be great to work with more flexibility in code implementation. NVidia has also made enhancement in memory
handling by implementing Unified Virtual Address this would certainly be benefit for several applications. There is lot of scope in data mining applications on GPUs and also Accelerated Video codecs processing for media industry therefore it would be a great idea to develop these applications using CUDA and OpenMP Accelerators and evaluating the best choice.
Appendix A

Cray Compiler

The Cray Puffin machine has its very own compiler for compiling C, C++ and FORTRAN programs. cc and ftn are wrapper scripts which on compilation eventually call “craycc” (for C), “crayCC” (for C++), “Crayftn” (for FORTRAN) with additional flags. It has default support for OpenMP directives however it can be disabled by using –h [no]omp switch. Further executing OpenMP program environment variable is used to set number of threads such as export OMP_NUM_THREADS=n, where n is number of threads. Moreover for mpich program mpiexec –n N (name of output file) can be used where N is the number of processors to be used.

Compiling C programs

cc –o (name of output file) myfile.c

No explicit switch is required to turn OpenMP option on as it is on by default.

Compiling openmp for accelerators

cc –o (name of output file) myfile.c

Cray compiler is optimized by default therefore no explicit switch tis required to be turned on for compilation. However the highest optimization option is –O3 and –h list=a switch can be used to check for applied optimization by the compiler while compilation.

Profiling

For profiling the code currently two profiling tools are used: CUDA Textual profiler and Cray Pat. The CUDA compute profiler generates a log file which has details about GPU and overall transfer timings between CPU and GPU. It requires environment variable to set to switch it on and off such as export COMPUTE_PROFILE=1 set the profiler on and 0 sets it off.

The Cray pat profiler is Cray profiler for profiling CPU codes.
References


[12] Cray XK6: Cray,


[24] Introduction to Cray Programming environment: Puffin, Cray Inc.,
(accessed April 2011)


