Exploiting Concurrency Patterns with Heterogeneous Task and Data Parallelism

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Technology

IN

Computational Science

by

A Geetha Venkatesh

Supercomputer Education and Research Centre

Indian Institute of Science

BANGALORE – 560 012

JULY 2014

Dedicated

To

My Dear Parents, Friends

and all my well-wishers...

Acknowledgements

It is a great pleasure for me to thank those who made this work possible. First of all, I thank SERC, IISc for giving me an opportunity to be part of this wonderful campus and ambiance to work in.

I would like to express my gratitude to my guides Prof. S.K.Nandy and Dr. J.Lakshmi for constant source of encouragement and guidance to me. A special thanks to Lakshmi Mam, for motivation, friendliness and support which is overwhelming and cannot be expressed in words. I thank Prof. Nandy for his ever-encouraging discussions and providing me the opportunity to work in CAD Lab. I am forever indebted and thankful to both of them for their efforts.

I am thankful to Dr. Ranjini Narayan, Mrs. Mitsu Mehta and Sivarama Krishnan for their discussions and insightful comments through out the project work. An extended sincere thanks to Prof Nandy, Lakshmi Mam, Mitsu Mam and Nitisha for bearing with my naive style of writing and teaching me how to express my thoughts clearly.

I am very happy to be a part of CAD Lab and thanks to all CAD Lab family members especially Kavitha and Madhav for the weekly interactions. Special thanks to Cloud Systems lab mates Nitisha, Aakriti, Pavan AK for their immense support and for making research more enjoyable in the lab.

I would like to thank my SERC friends Pavan Badarla, Jagvir, Aashish, Aditya, Aniruddha, Yogi, MK Reddy, Jayasimha, Jaganath, Prakash, Vasu for their encouragement and making my stay at IISc enjoyable. Thanks to my seniors Mohit, Ankit, Nilesh, Vivek and others for their valuable suggestions.

I thank the lab interns Monisha, Sindhuja and Anusha for the joyful interactions. The farewell given by our junior batch to us and the interactions with them throughout the last year *would be unforgettable, special thanks to all of them.*

I will be always thankful to my BITS friends for their constant support and encouragement. A special thanks for all IISc friends, my interaction with them has taught me many principles of life.

Finally, I would like to thank to my parents and sisters for their enormous support and love. I am just speechless in front of them with gratitude. They have been with me throughout my career and without them, I would not have been where I am. I would like to express my sincere gratitude to all those who have directly or indirectly helped in making this happen.

Abstract

Parallel programming of an application requires not only domain knowledge of the application, but also programming environment support and in-depth awareness of the target architecture. Often, all concurrency features of the architecture are not exposed to the programming environment. The challenge lies in efficient utilization of these unexposed features to write effective parallel programs. In our work, we explore different modes of OpenCL programming and focus on a specific application to reach its best performance. We have chosen iterative Strassen's matrix multiplication as our test application, as it exhibits variable amount of parallelism in each step and iteration. We explore a few parallel manifestations of the application based on fixed memory hierarchy and accounting for environmental constraints. These manifestations exploit various types of parallelism exhibited by the application such as data parallelism, task parallelism or a combination of both.

Concurrent Collections (CnC) is an architecture agnostic programming model in favor of the application developer. The dynamic execution model of CnC results in the best possible performance of the application. In CnC, parallelism of an application is expressed at the computational step level. Exploiting the parallelism within a coarse-grained computational step is a challenging task for CnC auto tuners. Ease of programming using CnC is adversely affected when we need to express fine-grained parallelism. OpenCL allows fine-grained programming through its hierarchical kernel structure. Through this work, we propose a close realization of the CnC programming model using OpenCL for achieving fine-grained parallelism and dynamic execution nature. We evaluate our work on two different architectures: nVidia Fermi C2070 GPGPU and Intel Core i3-350M 64-bit processor.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In general, parallel programming of a given application depends on the given target architecture and programming models supported by the application development environment. For any given application, there are some control and data dependency steps. A step can be a single instruction or a group of instructions. The amount of parallelism that can be achieved for a given application is restricted by dependencies among the steps in the application, specifically, data and control dependencies. Data dependency is when a step refers to the data produced by some preceding step(s). Control dependency is a situation in which the outcome of an earlier step determines the execution of later steps. The application gets executed in stages. At each stage, all concurrent steps, whose data and control dependencies are satisfied, can get executed. The aim is to get the best performance by executing as many concurrent steps as possible at each stage. However, the parallelism would then be restricted by the number of hardware resources available in the target platform. When there are more concurrent steps to execute than hardware resources, the steps are folded and are executed in sequence to satisfy resource constraints. When there are sufficient hardware resources to execute all of the concurrent steps at that stage, the best possible performance is achieved.

With recent developments in parallel architectures, the number of operations that can be executed per clock cycle has been increasing significantly. When there are no dependencies between steps ready to execute, theoretically, as many resources as possible could be used to maximize the performance. But usually there are dependencies between the steps which must be satisfied. We try to adapt the number of concurrent steps that can be executed at a stage to the resource constraints of the target architecture. Accounting for these hardware constraints, we execute different modes of the example application. A mode is a programming methodology which is followed throughout a particular implementation of the application. We have used the OpenCL framework as our programming platform. OpenCL (Open Computing Language) is a multi-vendor open standard programming language for general-purpose, cross-platform, parallel programming of heterogeneous systems that include CPUs, GPUs and other processors [16]. In the OpenCL programming model, functions that need to be executed on the computational device are named as kernels.

The 13 dwarfs from Berkeley [1] give different computation and communication patterns that are generally possible in different types of high performance applications. Iterative Strassen's Matrix multiplication exhibits some of these patterns like divide-and-conquer and map-reduce. Matrix multiplication has proven to be the most basic and important computation in numerous applications. This work, therefore, proposes a framework for optimizing Strassen's iterative matrix multiplication algorithm in the presence of resource constraints. We explore two different architectures i.e. GPGPUs supporting SIMT model of execution and CPUs having general purpose fast processing cores. In the implementation details of iterative Strassen's algorithm we also specify how we overcome the programming environment constraints which were noted in the process of implementing the programming modes. By adopting the closest equivalent model, these constraints can be overcome and the required functionality can be achieved. Through this process, we have explored different types of parallelism exhibited by the application such as data parallelism, task parallelism or a combination of both.

1.1 Motivation

Often, while implementing an application, in addition to domain constraints some of constraints due to programming environment might also get added [9]. Domain constraints are the actual scientific tasks to be performed and they remain valid regardless of the target architecture. Programming models like Concurrent Collections (CnC) [3] try to avoid adding the extra programming constraints during programming and specify only the domain constraints. In CnC, once the control and the data dependencies of the computational step are satisfied, the step is enabled for execution. Immutable (non-rewritable) intermediate data avoids false data dependencies between computational steps. CnC follows a dynamic model of execution of computational steps. Enabled computation steps get executed based on the availability of resources. The performance of the application written in CnC is highly dependent on the granularity of computational steps. Tuning of a coarse-grained computational step to the target device for exploring parallelism within the step is challenging for present CnC auto-tuners. The number of tags and tag functions required for writing CnC fine-grained computational steps can be huge for some applications. This increases the complexity of programming. On the other hand, in OpenCL exploiting fine-grained parallelism is easy by using the hierarchy of the kernel programming structure, work-groups and work-items. We observe that a close realization of the CnC dynamic execution model using the OpenCL programming framework gives a better application performance.

1.2 Related Work

Parallel programming model requires target architecture awareness [14]. One of the top ten issues in parallel computing is scheduling tasks at the right granularity onto the processors of the parallel machine [11]. [17] discuss about task and data parallelism on a multi-computer and discuss ideas on how to expose these trade-offs to the compiler. [18] try to achieve nested task and data-parallelism in High Performance FORTRAN (HPF). They proposed extension implementations using directives which was approved in HPF 2.0. OpenCL is supported across different architectures having various warp or wavefront sizes. This necessitates platform dependent optimizations for peak performance [16]. Fine-grained parallelism being explored by OpenCL is covered in [[19]. The paper [6] gives the time-formula for the overhead of recursion for the Strassen's algorithm. They have implemented a memory access pattern for the matrices to decrease the memory access time. The paper [20] use the Mentat Programming Language (MPL) to integrate task and data-parallelism. It shows that the invocation of both task-and-data-parallel elements at the same time would be beneficial. The memory access patterns and iterative Strassen's matrix multiplication algorithm are covered in [7].

Efficient computation of sum-products on GPU is descibed in [13]. The architectural constraints of GPU and the unexposed concurrency features in GPGPU is covered in [12]. The general computation and communication patterns are given by [1]. The effective implementaion of strassen's matrix multiplication on GPUs is given in [10]. Using OpenCL the programming of the 13 Berkeley dwarfs implementation is done in [8]. Using CUDA programming the programming patterns are explored in [15]. CnC is an architecture agnostic programming model [9]. The execution model of CnC is dynamic in nature which results in the peak performance on the target device [3]. Memory management aspects of CnC are discussed in [4]. The habanero multicore software research project tries to explore heterogeneous exascale multicore machines [2]. CnC performance comparison with other programming libraries and results of excellence is shown in [5].

1.3 Contributions of thesis

The key contributions of this work is as follows.

—Expose the generic programming modes designed for the OpenCL programming model. This is the set of possible modes of programming the kernel. We explore different types of parallelism on different data types at multiple levels of the kernel hierarchy.
—Methods to overcome programming environment constraints. These constraints can be due to the programming model or may be due to the hardware resources of the target architecture. We propose programming tweaks and modified approach of executing the concurrent steps at a given stage.
—A design framework to identify the best subset of programming modes for specific computational patterns of an application on a particular target architecture
—Our results show that realization of the CnC dynamic execution model using fine-grained

OpenCL programming improves application performance to a significant extent.

1.4 Thesis Organization

In this thesis we explored few computational patterns using different OpenCL programming modes. We proposed a dynamic OpenCL programming mode with the motivation of Concurrent Collections programming model. The proposed dynamic mode of programming gives better performance when compared to the generic programming modes of OpenCL. The rest of the thesis is organized as follows:

Chapter 2 provides an introduction to the OpenCL programming model. This chapter gives the OpenCL programming terminology and kernel hierarchical structure. Chapter 3 provides a brief introduction to iterative Strassen's algorithm. The different computational patterns of the algorithm are shown with visual representation. In chapter 4, we describe the proposed general programming modes of OpenCL. The kernel hierarchical matching with memory hierarchy and exploration of parallelism is explained in these sections. Chapter 5 provides implementation details of appropriate OpenCL programming modes for Strassen's algorithm. In Chapter 6, we explain the architecture agnostic concurrent collections programming model and the dynamic OpenCL programming mode. We report our experimental results obtained for the different modes of execution on both Fermi GPU and i3 CPU. Followed by the conclusion in chapter 7.

1.5 Summary

In this chapter an introduction of the background of the project problem is explained. The motivation of the Concurrent Collections programming model for better execution of application is explained. The related work carried on in these fields is described. The contributions of the thesis work is also mentioned in this chapter. The following chapters explain in detail the different programming modes of OpenCL programming and the ways in which we explored different computational patterns.

Chapter 2

OpenCL Programming

OpenCL is a simple programming framework which supports heterogeneous architectures like GPGPUs, multicore CPUs and FPGAs. To explore the best possible parallelism of the application that can be executed on the device, some awareness of the target device architecture is required. GPGPUs have light weight processors that allow them to support significant data parallelism. On the other hand, general purpose CPU cores are fast general purpose cores which can support data parallel or task parallel threads. The onus is on the programmer to exploit different architectural features in appropriate ways to get the best performance. When the device driver supports OpenCL API extensions to the C programming language, a normal C program can invoke special functions called kernels to be executed concurrently on the device. A kernel can be called by a multitude of data-parallel execution instances called work-items. The computational device is composed of compute units and compute units, in turn, contain processing elements [19]. Work-items are grouped into work-groups. In the OpenCL execution environment there is a one-to-one correspondence between kernel and computational device, work-groups and compute units, and work-items and processing elements. The index space in which a kernel is launched is called 'NDRange' (N-Dimensional Range). This index space can be of one, two or three dimensions. The execution instance of a kernel in an NDRange space is called an NDRangeKernel. All work-items in an NDRangeKernel execute the same set of instructions. Work-items are differentiated by their work-item-ids. We operate these work-items on different data based on their work-item-id, so that they can process a large number of data-elements at the same time. All these work-items are orchestrated as warps or wavefronts. The SIMT mode of execution in GPGPU is actually at this warp level and threads within the same warp execute a single instruction at a time.

2.1 Data, Task and Heterogeneous Parallelism

Data parallelism in OpenCL context is achieved when different kernel hierarchy levels (kernels, work-groups or work-items) execute in parallel with the same set of instructions on multiple data items. On the other hand, task parallelism is the concurrent execution of kernel hierarchy levels doing different tasks on the same data or different data. We define heterogeneous parallelism as exploiting both data and task parallelism by the same type of kernel hierarchy levels. In the execution model of work-items, running all the code of the kernel is essentially a data parallel mode of execution. This makes achieving task-parallelism within the NDRangeKernel a tedious job. To achieve task parallelism, work-items need to be heterogeneous. For this, there needs to be a branch in the NDRangeKernel code, based on the work-item-id or work-group-id. In case of GPGPUs, the execution model is SIMT mode where a warp/wavefront of threads/work-items are executed concurrently. When there is branching within the wavefront based on work-item-id, the execution of work-items gets serialized. This serialization effect can be minimized if we try not to break work-items within a wavefront.

To achieve parallelism, we need to explore concurrent execution at different kernel hierarchy levels. The amount of concurrent execution at each level of hierarchy in the programming mode gets constrained by some hardware resources. For example, in the case of GPGPUs, the maximum number of concurrent work-items that can get executed is limited by the number of SPs (Streaming processors) within an SM (Streaming Multiprocessor). The maximum number of concurrent warps that can be executed is limited by the number of warp schedulers within an SM. The maximum number of concurrent work-groups that can get executed is limited by the number of SMs within the GPGPU device. Hence the various hierarchical levels of OpenCL kernels have to be coded bearing these constraints in mind.

2.2 Summary

This chapter gave a brief overview of OpenCL programming model. The data, task and heterogeneous parallelism in OpenCL terminology are explained. In the next chapter the iterative Strassen's matrix multiplication is explained in detail. Followed by the different programming modes of OpenCL.

Chapter 3

Iterative Strassen's Matrix Multiplication Algorithm

Strassen's matrix multiplication algorithm is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices. Strassen's iterative algorithm exhibits various computational patterns such as the divide-and-conquer pattern in the decomposition step and the map-reduce pattern in the merge step. In standard matrix-matrix multiplication, the resultant matrix to be calculated by equation 3.1 is done through the transformations of equation 3.2

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \tag{3.1}$$

$$\begin{aligned} r &= ae + bf \,, s = ag + bh \\ t &= ce + df \,, u = cg + dh \end{aligned} \tag{3.2}$$

For a matrix of size N, standard matrix multiplication involves $O(N^3)$ multiplication operations. Strassen's algorithm asymptotically reduces the running time of matrix multiplication to $O(N^{2.807})$ by doing only seven multiplication operations instead of eight. Strassen's original algorithm is recursive in nature. The recursive nature of the algorithm imposes run time overheads [6]. We have therefore chosen iterative Strassen's algorithm in our experiments. Iterative Strassen's algorithm exposes variable amount of task and data parallelism at each iteration, which would help in exploiting different programming modes and to work around hardware resource constraints. The resultant matrix is calculated in three steps given in table 3.1.

Table 3.1: Iterative Strassen's Matrix Multiplication Algorithm

Decomposition		**Multiplication**	**Re – composition**
$A_1 = a$	$B_1 = g - h$	$P_1 = A_1.B_1$	$r = P_5 + P_4 - P_2 + P_6$
$A_2 = a + b$	$B_2 = h$	$P_2 = A_2.B_2$	$s = P_1 + P_2$
$A_3 = c + d$	$B_3 = e$	$P_3 = A_3.B_3$	$t = P_3 + P_4$
$A_4 = d$	$B_4 = f - e$	$P_4 = A_4.B_4$	$u = P_5 + P_1 - P_3 - P_7$
$A_5 = a + d$	$B_5 = e + h$	$P_5 = A_5.B_5$	
$A_6 = b - d$	$B_6 = f + h$	$P_6 = A_6.B_6$	
$A_7 = a - c$	$B_7 = e + g$	$P_7 = A_7.B_7$	

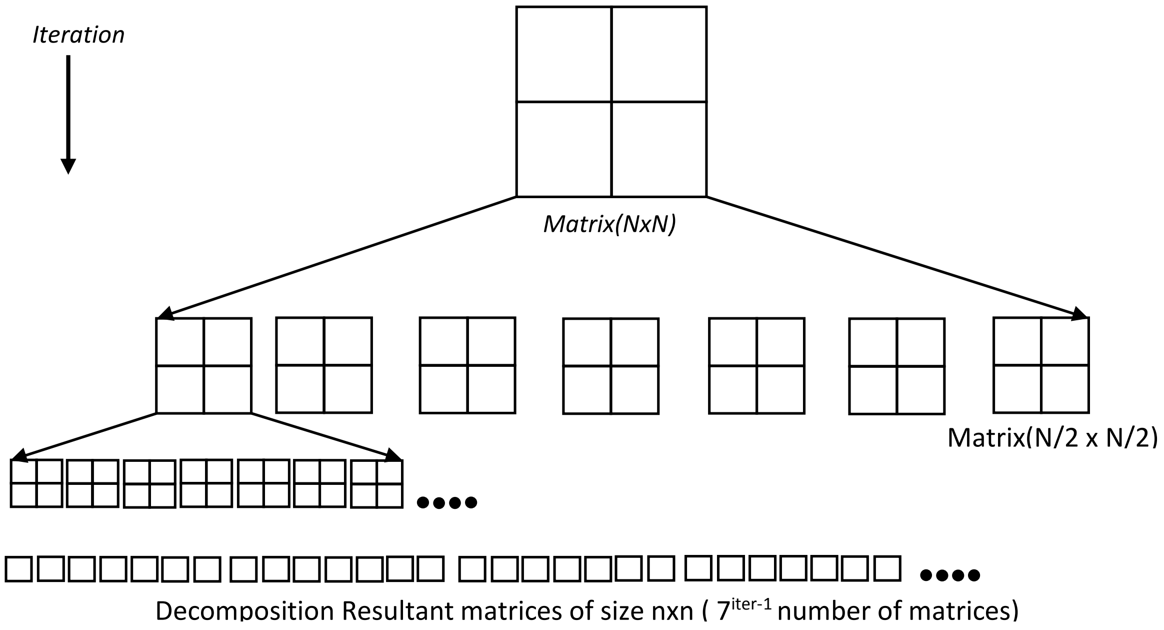
Iteration
Matrix(NxN)
Matrix(N/2 x N/2)
Decomposition Resultant matrices of size nxn (7^iter-1 number of matrices)

Figure 3.1: Decomposition Step of Iterative Strassen's Algorithm

i) Decomposition step - In this step, the input matrix of size $N \times N$ is subdivided into four sub-matrices of size $\frac{N}{2} \times \frac{N}{2}$. As given by the Strassen's iterative decomposition formulas given in table 3.1, seven basic operations (add, sub, copy) are done between these four sub-matrices to produce seven intermediate matrices. This is done iteratively until we obtain intermediate blocks of size $n \times n$. The threshold-block-size 'n' is arbitrary and can be chosen by the programmer appropriately. It also specifies the base condition for stopping the iterations. After the decomposition step, we get both the A derived set and B derived set of the intermediate matrices of size $n \times n$. Figure 3.1 gives a visual representation of the decomposition step of the algorithm.

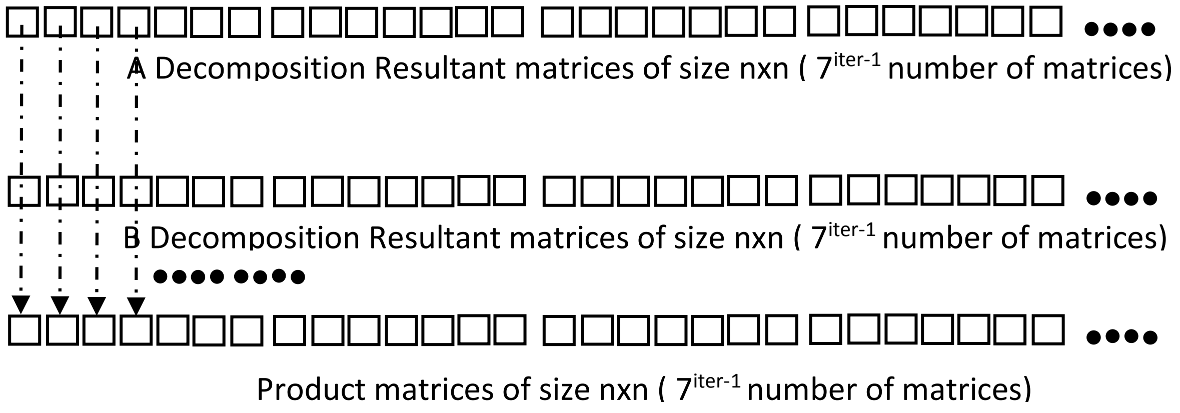
A Decomposition Resultant matrices of size nxn (7^iter-1 number of matrices)
B Decomposition Resultant matrices of size nxn (7^iter-1 number of matrices)
Product matrices of size nxn (7^iter-1 number of matrices)

Figure 3.2: Multiplication Step of Iterative Strassen's Algorithm

ii) Multiplication step - The intermediate matrices obtained from the decomposition equations given in table 3.1 are multiplied using standard matrix multiplication to obtain intermediate product matrices $P_1, P_2, ..., P_7$ given by the multiplication equations given in table 3.1. All the intermediate matrices in set A are multiplied by the corresponding intermediate matrices in set Bto get intermediate product matrices. Figure 3.2 gives a visual representation of the block matrix multiplication step of the algorithm.

iii) Merge step - This step is also known as the re-composition step. In this step, the intermediate product matrices are merged iteratively to form the final result matrix of the size $N \times N$. The merging of the intermediate product matrices $P_1, P_2, ..., P_7$ to calculate the final result matrix P is done by using the re-composition equations given in table 3.1. Figure 3.3

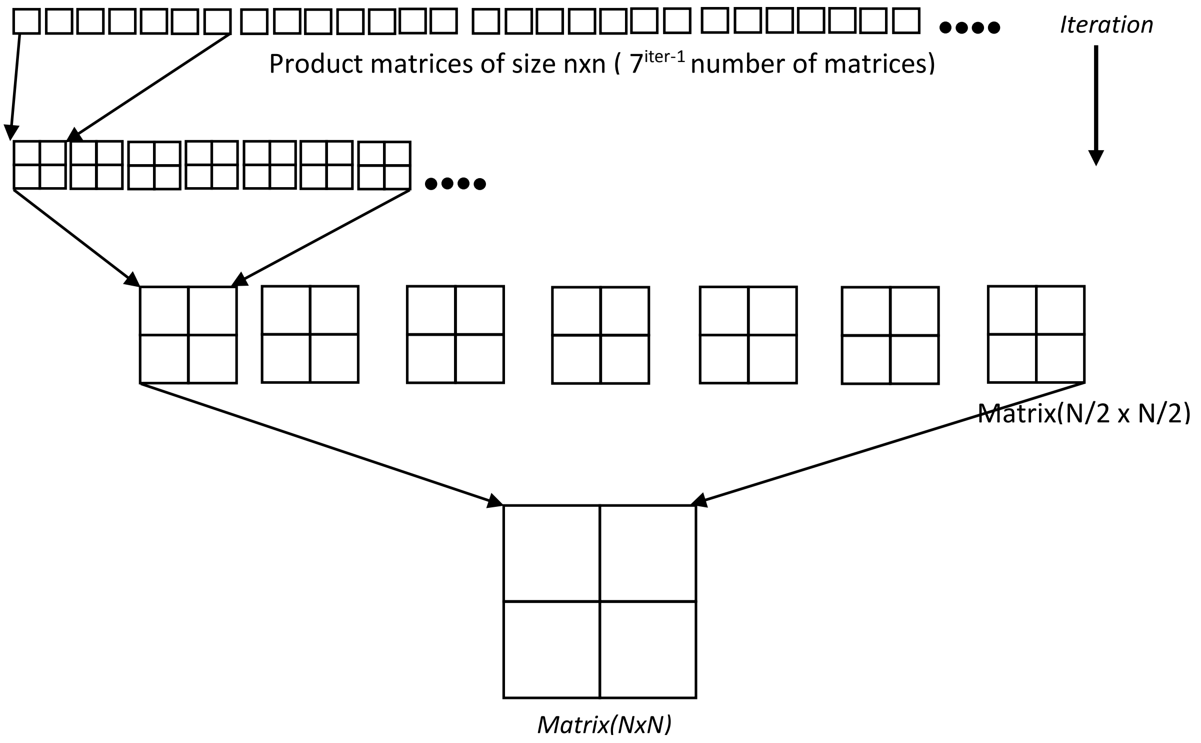

Figure 3.3: Re-composition/Merge Step of Iterative Strassen's Algorithm

gives a visual representation of the merge step of the algorithm.

3.1 Summary

This chapter explained the iterative Strassen's algorithm and the various steps involved. The different computational patterns at each step of the algorithm is explored by different programming modes in the following chapters. This exploration of the parallelism is extended by the proposed dynamic programming model.

Chapter 4

Generic OpenCL Programming Modes

In this section, we present our proposal for generic OpenCL programming modes. In the OpenCL framework, to execute concurrent steps on the target device, we need to launch them as kernels onto the device. The different programming levels of the kernel are work-item, wavefront, work-group and kernel which we denote as kernel hierarchy levels. The data which these hierarchy levels can act on, can be of one dimensional (1D), one dimensional vector (1Dv), two dimensional (2D), two dimensional vector (2Dv), three dimensional (3D), or three dimensional vector (3Dv) and so on. Additionally, it is possible to explore data, task or heterogeneous parallelism at various hierarchy levels. Table 4.1 shows the matrix for choosing the dimension of data to be handled vis-a-vis the type of parallelism at each kernel hierarchy level.

Table 4.1: Choices of parallelism and data types for kernel hierarchy

Kernel/Work-Group/Work-Item	**1D Data**	**1D Vector**	**2D Data**	**2D Vector**	**3D Data**
Data Parallelism (D)					
Task Parallelism (T)					
Heterogeneous Parallelism (H)					

When writing the code for a kernel, we chose the dimension of the data handled by each level in the hierarchy and also the type of parallelism to be exploited at each level. For example, a kernel can be formed with 2D vector data handling and data parallelism at the kernel level, 2D data handling and task parallelism at the work-group level, and 1D data handling and data level parallelism at the work-item level. We denote this using a 3-tuple naming convention as 2Dv Data - 2D Task - 1D Data (Kernel - Workgroup - Workitem tuple) or simply as 2vD-2T-1D. The first number in each part of the tuple indicates the dimension of the data being handled by the kernel hierarchy level. If 'v' is added, it denotes that the level works with vector type data. In the second position, D stands for Data parallelism, T for Task parallelism and H for Heterogeneous parallelism. This naming convention can be used for any generalized representation of the kernel. Some of these modes are not practical; for example, 1D-2vT-3H.

4.1 Strassen's Algorithm Programming Modes

In iterative Strassen's algorithm, the first step is the decomposition of matrices A and B where after a chosen number of iterations, we have sets of intermediate matrices obtained from matrices A and B. It is not possible to explore all the above mentioned generic OpenCL programming modes in the context of Strassen's iterative algorithm as the memory requirements for some of the modes are impractical. We therefore chose a reasonable mapping of memory hierarchy to the kernel hierarchy levels for pruning the programming modes. In our chosen memory hierarchy, kernels handle 2D vector data (sets of matrices), work-groups handle 2D data (matrices) and work-items handle 1D data (data items). Once this memory hierarchy is fixed, we are left with the following set of modes for iterative Strassen's algorithm as shown in figure 4.1.

The boxes indicate the type of parallelism expressed at each kernel hierarchy level. We limit ourselves to the seven tasks in the decomposition step, one in the multiplication step and four in the merging step. We don't consider any division of these tasks into sub-tasks. This implies that once heterogeneous or task parallelism is chosen at any kernel hierarchy level, the next level in the kernel hierarchy cannot opt to use heterogeneous or task parallelism. The implementation details of these OpenCL programming modes for Strassen's iterative algorithm are briefly explained in the next section.

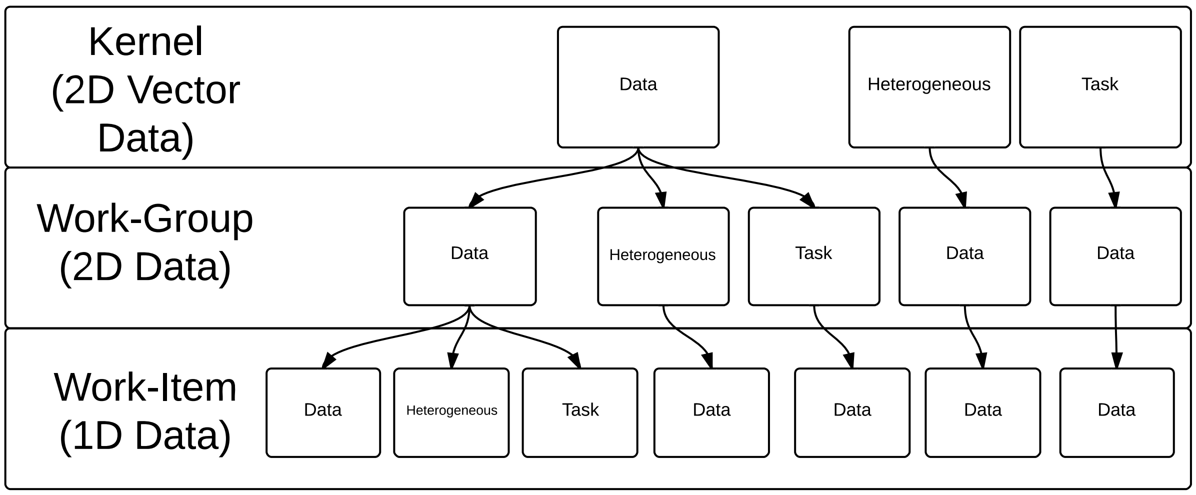

Figure 4.1: Hierarchical tree showing the different modes possible for iterative Strassen's algorithm

4.2 Summary

This chapter gives the generic programming modes of OpenCL programming. Based on the kernel hierarchy, type of parallelism and memory hierarchy the modes are defined. In the following chapters the implementation details of the generic modes for Strassen's algorithm is explained.

Chapter 5

Programming Modes for Strassen's Algorithm

In the present work, a programming mode denotes the programming methodology adopted throughout a particular parallel programming manifestation. We now discuss the practical implementation details of a few chosen modes for the iterative Strassen's algorithm. These modes are derived by choosing an appropriate memory hierarchy for the kernel levels. This set of modes is not exhaustive but contains those candidate modes which are appropriate for exploiting concurrency features of the selected target architectures. The processing elements in GPGPUs are light weight. Executing single matrix element computations on these processing elements is beneficial. Work-items are associated with processing elements during kernel execution. Therefore we implement work-items that act on 1D data elements. Work-groups are a group of work-items that act on 2D data. Work-groups acting on matrix data type (2D data) provide greater flexibility in exploring parallelism over matrices. At the kernel level, sets of matrices need to be handled and so, kernels are associated with 2D vector data type (sets of matrices).

On nVidia Fermi GPGPU due to the limitations of the supported OpenCL driver, the execution of concurrent kernels launched into different command queues gets serialized. Therefore, we also used multicore i3 CPU as an alternative to explore concurrent kernel execution. The following sections describe the implementation details of kernel programming modes. In all the modes, we strive to ensure that warps or wavefronts are not broken. Work-items acting on data items are treated as wavefronts acting on 1D vector data.

5.1 2vD-2D-1D

In this mode, the kernel handles two dimensional vector data and exhibits data parallelism, work-group handles two dimensional data and exhibits data parallelism and work-items handle one dimensional data and exhibit data parallelism. In this mode of execution, pure data parallelism is chosen at all the kernel hierarchy levels. Each work-item does all the required tasks on single matrix elements. Tasks are executed serially by the work-items. These tasks can also be handled serially at kernel or work-group level. In cases where the limit of the maximum number of work-items per work-group is crossed, work-items are enhanced to handle extra data-items. This increase in the workload of work-items changes this mode to 2vD-2D-1vD. This switching of modes happens when the number of matrix elements exceeds the maximum number of work-items allowed per work-group on the target device. Another way to handle this situation is to increase the number of work-groups that handle operations on a single matrix. A visual representation for the decomposition step using 2vD-2D-1D mode is shown in the figure 5.1.

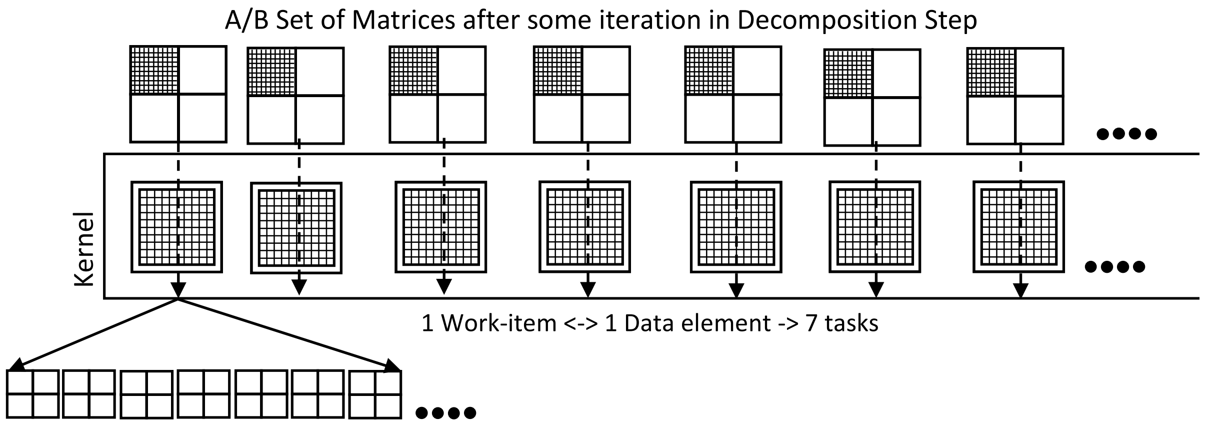
A/B Set of Matrices after some iteration in Decomposition Step
Kernel
1 Work-item <-> 1 Data element -> 7 tasks

Figure 5.1: 2vD-2D-1D mode for matrix decomposition

In this implementation, all the seven tasks are done by each of the work-items and one work-group is associated with one matrix. Matrices A and B have separate kernels for decomposition step, whereas for the multiplication and merging steps, a single kernel acts on the intermediate matrices. In the multiplication step, we follow a similar methodology of one work-item acting on one matrix element and a work-group acting on a single matrix. In the merging step, a work-group acts on seven matrices to merge them into a single matrix and the four tasks are serialized within the work-items.

5.2 2vD-2D-1H

In this mode, the kernel handles two dimensional vector data and exhibits data parallelism, work-group handles two dimensional data and exhibits data parallelism, and work-items handle one dimensional data and exhibit heterogeneous parallelism. In this mode of execution, both task and data parallelism are explored at the work-item level without breaking the wavefronts. In the decomposition step, each data-item requires seven work-items, i.e. one work-item per task. Similar to the 2vD-2D-1D mode, this mode is affected by the resource constraint of maximum number of work-items allowed per work-group. When we reach the limit of the maximum number of work-items per work-group, we enhance the work-items to handle extra data-items. This increase in workload of work-items modifies this mode to 2vD-2D-1vH. This switching of modes happens when the number of matrix elements is more than the maximum number of work-items per work-group. A visual representation for the decomposition step using 2vD-2D-1H mode is shown in figure 5.2.

In this case, the seven tasks of the decomposition step are handled by seven work-items per data element and one work-group is associated with one matrix. Similar to 2vD-2D-1D mode, matrices A and B have separate kernels for decomposition step. For the multiplication and merging steps, a single kernel acts on the intermediate matrices. In the multiplication step, we follow the methodology of one work-item per data-item and one work-group per matrix. In the merging step, four work-items act on one of the data-items of the seven matrices and a work-group acts on seven matrices to merge into a single result matrix.

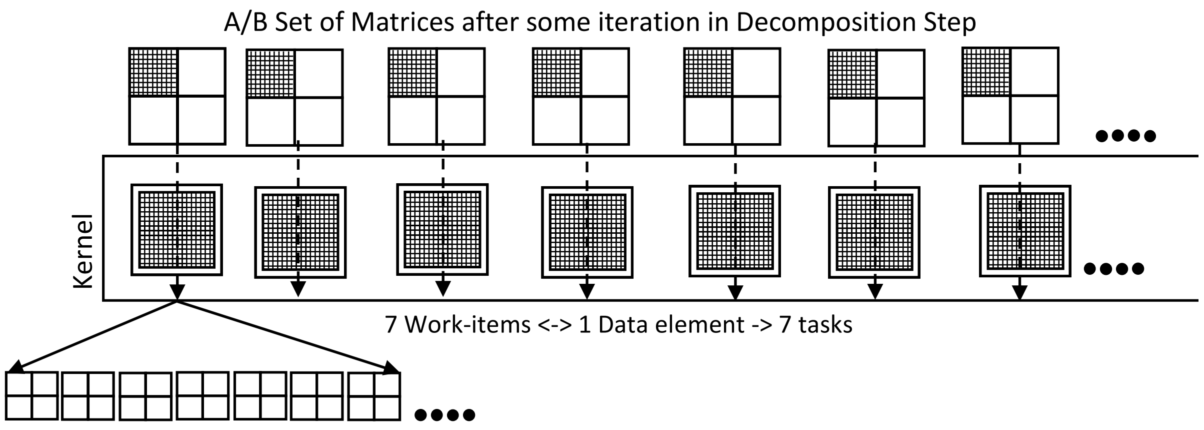
A/B Set of Matrices after some iteration in Decomposition Step
Kernel
7 Work-items <-> 1 Data element -> 7 tasks

Figure 5.2: 2vD-2D-1H mode for matrix decomposition

5.3 2vD-2D-1T

In this mode, kernel handles two dimensional vector data and exhibits data parallelism, work-group handles two dimensional data and exhibits data parallelism, and work-items handle one dimensional data and exhibit task parallelism. In this mode of execution, task parallelism is explored at the work-items level without breaking the wavefronts. Here each work-group has wavefronts whose number is equal to the number of tasks. In the decomposition step, each work-group have seven wavefronts. Each of the wavefront perform one task on the whole matrix vector-by-vector level. A visual representation for the decomposition step using 2vD-2D-1T mode is shown in the figure 5.3.

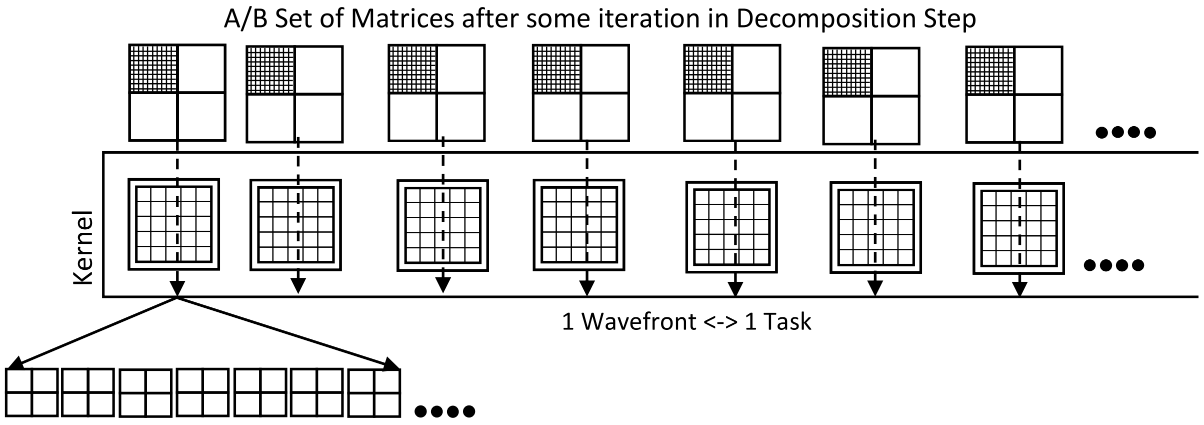
A/B Set of Matrices after some iteration in Decomposition Step
Kernel
1 Wavefront <-> 1 Task

Figure 5.3: 2vD-2D-1T mode for matrix decomposition

In this case, the seven tasks of the decomposition step are handled by seven wavefronts and one work-group is associated with one matrix. The seven wavefronts operate on the entire matrix to complete the seven tasks. Similar to above modes, matrices A and B have separate kernels for decomposition step. For the multiplication and merging steps, a single kernel acts on the intermediate matrices. In the multiplication step, we follow the methodology of one wavefront per work-group handles a matrix. In merging step, four wavefronts per work-group acts on seven matrices to merge into a single matrix.

5.4 2vD-2H-1D

In this mode, the kernel handles two dimensional vector data and exhibits data parallelism, work-group handles two dimensional data and exhibits heterogeneous parallelism and work-items handle one dimensional data and exhibit data parallelism. Here both task and data parallelism are explored at work-groups level over matrices. In the decomposition step, each matrix requires seven work-groups i.e. one work-group per one matrix task. Similar to the 2vD-2D-1D/H modes, this mode is affected by the resource constraint of maximum number of work-items allowed per work-group. When we reach the limit of the maximum number of work-items per work-group, we enhance the work-items to handle extra data-items. The increase in workload of the work-items modifies this mode to 2vD-2H-1vD. This switching of modes happens when the matrix elements within a matrix are more than the maximum number of work-items per work-group. A visual representation for the decomposition step using 2vD-2H-1D mode is shown in the figure 5.4.

In this case, the seven tasks of the decomposition step are handled by seven work-groups per matrix and one work-item is associated with one data-item. Matrices A and B have separate kernels for decomposition step. For the multiplication and merging steps, a single kernel acts on the intermediate matrices. In the multiplication step, we follow the methodology of one work-item per data-item and one work-group per matrix. In the merging step, four work-groups act on seven matrices to merge into a single matrix and one work-item acts on one of the seven matrices elements.

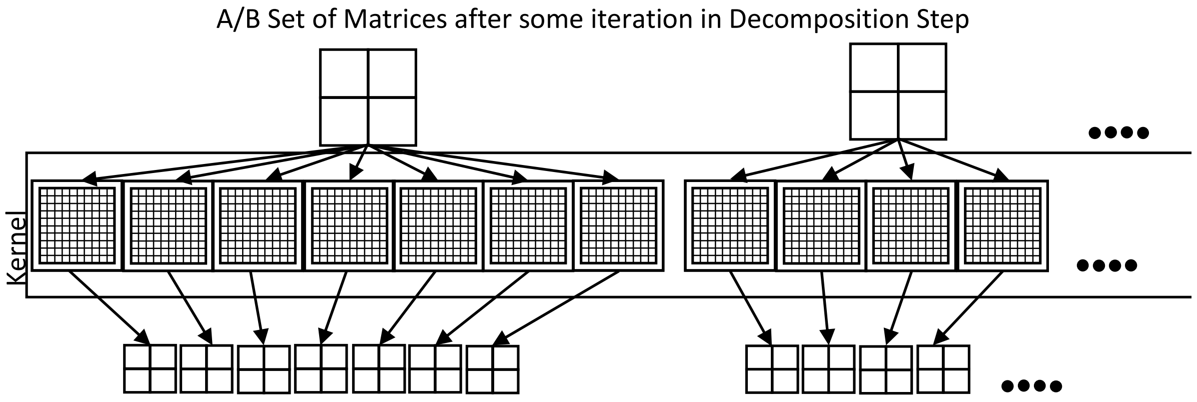
A/B Set of Matrices after some iteration in Decomposition Step
Kernel
●●●●
●●●●
●●●●

Figure 5.4: 2vD-2H-1D mode for matrix decomposition

5.5 2vD-2T-1D

In this mode, the kernel handles two dimensional vector data and exhibits data parallelism, work-group handles two dimensional data and exhibits task parallelism and work-items handle one dimensional data and exhibit data parallelism. Here task parallelism is explored at work-groups level over matrices. In this mode, the number of work-groups is equal to the number of tasks. In the decomposition step, kernel have seven work-groups. Each of the work-group performs one task on a matrix. This mode is affected by the resource constraint of maximum number of work-items allowed per work-group. When we reach the limit of the maximum number of work-items per work-group, we enhance the work-items to handle extra data-items. The increase in workload of the work-items modifies this mode to 2vD-2T-1vD. This switching of modes happens when the matrix elements within a matrix are more than the maximum number of work-items per work-group. A visual representation for the decomposition step using 2vD-2T-1D mode is shown in the figure 5.5.

In this case, the seven tasks of the decomposition step are handled by seven work-groups per matrix and one work-item is associated with one data-item. The seven tasks of all the matrices is executed by these seven work groups. Matrices A and B have separate kernels for decomposition step. For the multiplication and merging steps, a single kernel acts on the intermediate matrices. In the multiplication step, we follow the methodology of one work-item per data-item and one work-group per matrix. In the merging step, four work-groups act on seven matrices to merge into a single matrix and one work-item acts on one of the seven

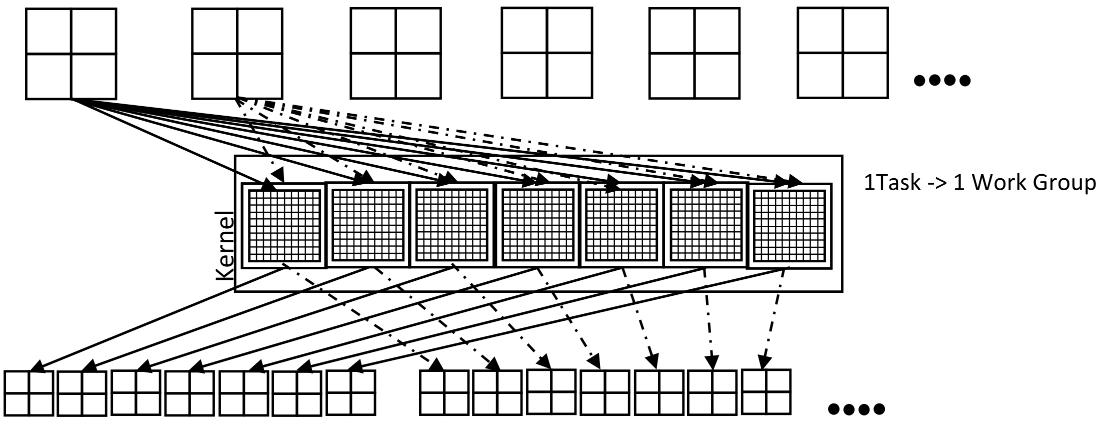

Figure 5.5: 2vD-2T-1D mode for matrix decomposition

matrices elements.

5.6 2vH-2D-1D

In this mode, the kernel handles two dimensional vector data and exhibits heterogeneous parallelism, work-group handles two dimensional data and exhibits data parallelism and work-items handle one dimensional data and exhibit data parallelism. Here both task and data parallelism are explored at the kernel level over sets of matrices. In the decomposition step, both sets of matrices A and B are operated concurrently and the seven tasks of each are explored by different kernels. This mode is affected by the resource constraint of maximum number of work-items allowed per work-group. When we reach the limit of the maximum number of work-items per work-group, we enhance the work-items to handle extra data-items. The increase in workload of the work-items modifies this mode to 2vH-2D-1vD. This switching of modes happens when the matrix elements within a matrix are more than the maximum number of work-items per work-group. A visual representation for the decomposition step using 2vH-2D-1D mode is shown in the figure 5.6.

In this case, the seven tasks of the decomposition step are handled by seven kernels per set of matrices and one work-item is associated with one data-item. The total fourteen tasks of the matrices A and B are executed by fourteen kernels. Matrices A and B have separate kernels for decomposition step. For multiplication step, one kernel acts on all the intermediate matrices.

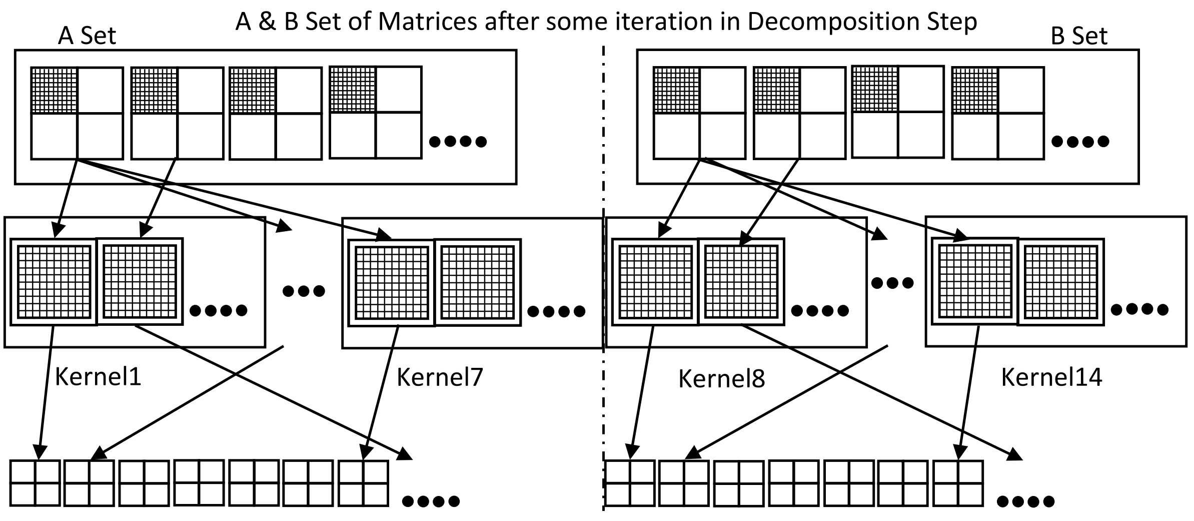
A & B Set of Matrices after some iteration in Decomposition Step
A Set
B Set
Kernel1
Kernel7
Kernel8
Kernel14

Figure 5.6: 2vH-2D-1D mode for matrix decomposition

In the multiplication step, we follow the methodology of one work-item per data-item and one work-group per matrix. In the merging step, four kernels act on seven matrices to merge into a single matrix and one work-item acts on one of the seven matrices elements.

5.7 2vT-2D-1D

In this mode, the kernel handles two dimensional vector data and exhibits task parallelism, work-group handles two dimensional data and exhibits data parallelism and work-items handle one dimensional data and exhibit data parallelism. Here task parallelism is explored at the kernel level over sets of matrices. In the decomposition step, both sets of matrices A and B are operated concurrently and the seven tasks of each are explored by different kernels. This mode is affected by the resource constraint of maximum number of work-items allowed per work-group. When we reach the limit of the maximum number of work-items per work-group, we enhance the work-items to handle extra data-items. The increase in workload of the work-items modifies this mode to 2vT-2D-1vD. This switching of modes happens when the matrix elements within a matrix are more than the maximum number of work-items per work-group. A visual representation for the decomposition step using 2vT-2D-1D mode is shown in the figure 5.7.

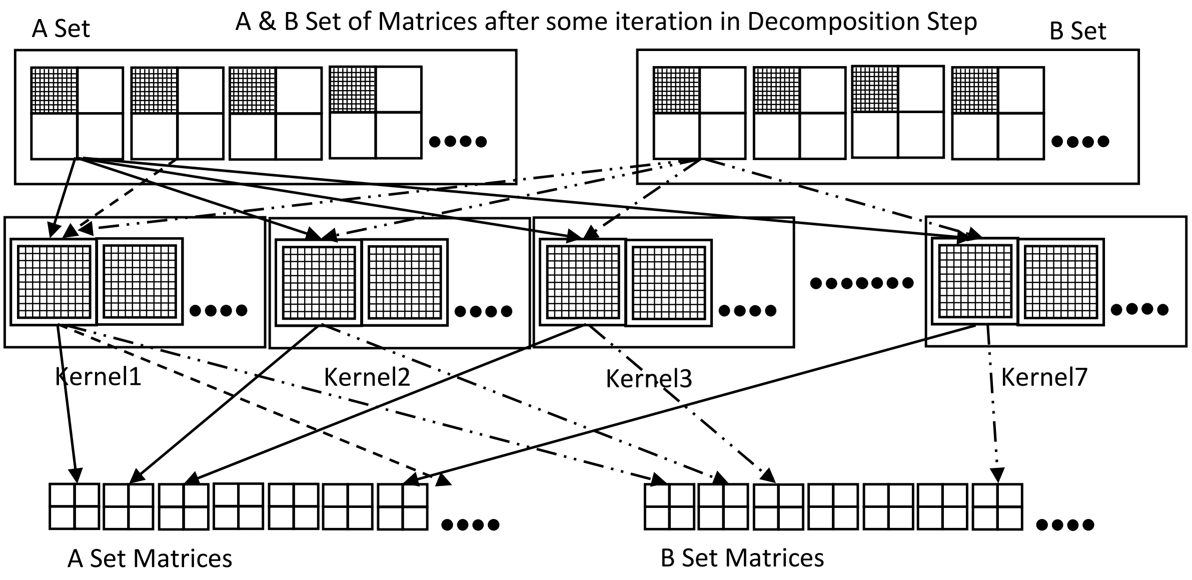

Figure 5.7: 2vT-2D-1D mode for matrix decomposition

In this case, the seven tasks of the decomposition step are handled by seven kernels per set of matrices and one work-item is associated with one data-item. The seven tasks of the matrices A and B are executed by seven kernels. Matrices A and B have separate kernels for decomposition step. For multiplication step, one kernel acts on all the intermediate matrices. In the multiplication step, we follow the methodology of one work-item per data-item and one work-group per matrix. In the merging step, four kernels act on seven matrices to merge into a single matrix and one work-item acts on one of the seven matrices elements.

5.8 Summary

In this chapter the implementation details of the generic OpenCL programming modes for iterative Strassen's matrix multiplication algorithm are explained. The kernel and memory hierarchy fixation limited the number of modes to these few possibilities. Overcoming of the environmental constraints during the implementation of the modes was explained in this chapter. In the following chapter a dynamic execution mode which improves the performance of the application is proposed.

Chapter 6

Dynamic Mode and Results

6.1 Concurrent Collections

The Concurrent Collections (CnC) programming model facilitates ease of programming by allowing the programmer to write the application in an architecture agnostic way. In this model, data is immutable (not over-written) and this prevents false data dependencies. CnC exhibits a dynamic execution nature. The execution of the application is expressed using computational steps. The computational step is initialized as and when the data for the step is available. At each execution stage, the computational steps whose data and control dependencies are satisfied are ready to execute. CnC breaks the close relationship of when a step gets executed from if a step gets executed. This makes the tuning of the application to the target architecture, dynamic in nature. Based on the availability of resources, the enabled steps get executed. Due to dynamic execution nature and limited architecture constraints, the performance results of CnC are more promising than other parallel programming models.

In CnC, the granularity of a computational step can be of any size. Exploring parallelism within a coarse-grained computational step becomes challenging for CnC. Efficient execution of the application is difficult to achieve for specific devices. The performance of the application might possibly get degraded. For implementing the application in fine-grained computational steps, the complexity of the implementation is very high. The number of tags and tag functions that need to be handled creates immense programming overheads in the case of many high performance applications. On the other hand, in OpenCL programming, the hierarchical structure of kernel, work-groups and work-items makes fine-grained parallelism exploration easy. OpenCL is a low level programming language but OpenCL programmers can choose to be architecture agnostic. We use command queues and events of the OpenCL programming model to mimic the dynamic execution nature of CnC.

6.2 Strassen's Dynamic Execution Mode (DYN)

To get better performance for applications, we need dynamic execution of the application and the ability to exploit fine-grained parallelism. The OpenCL hierarchical kernel structure makes expressing fine-grained parallelism easier. For Strassen's algorithm, all the programming modes explored in the previous section exhibit fine-grained parallelism. Among these modes, some modes execute more effectively on a specific target architecture. A subset of these efficient modes are also better suited to a particular computational pattern in the application. We have chosen the best performing mode for each computational pattern to get the overall application performance improvement. In this dynamic execution mode, we ensure that once the required dependencies of a step are satisfied, the step is ready to get executed. For enabling the immutable nature of the data in OpenCL, we need memory management techniques. In our model we make a close realization of the data immutability property, but this can be improved to get better application performance.

6.3 Results

The implementations of the different modes of iterative Strassen's multiplication algorithm described in section 4 and the dynamic execution model (DYN) from the last section are executed on two different types of target architecture devices. On the Fermi GPGPU, modes adopting data parallelism showed good performance. Whereas in the CPU, both task and data parallel modes showed good performance. In both the CPU and GPU cases, the dynamic execution mode clearly performed better than the other modes. In the graphs, we have shortened the names of the modes to simply indicate the parallelism that is being explored. For example, 2vD-2H-1D is renamed to DHD mode, because the fixed memory hierarchy Kernels - 2D vector data, Work-groups - 2D data and for Work-items - 1D data is same for all the modes.

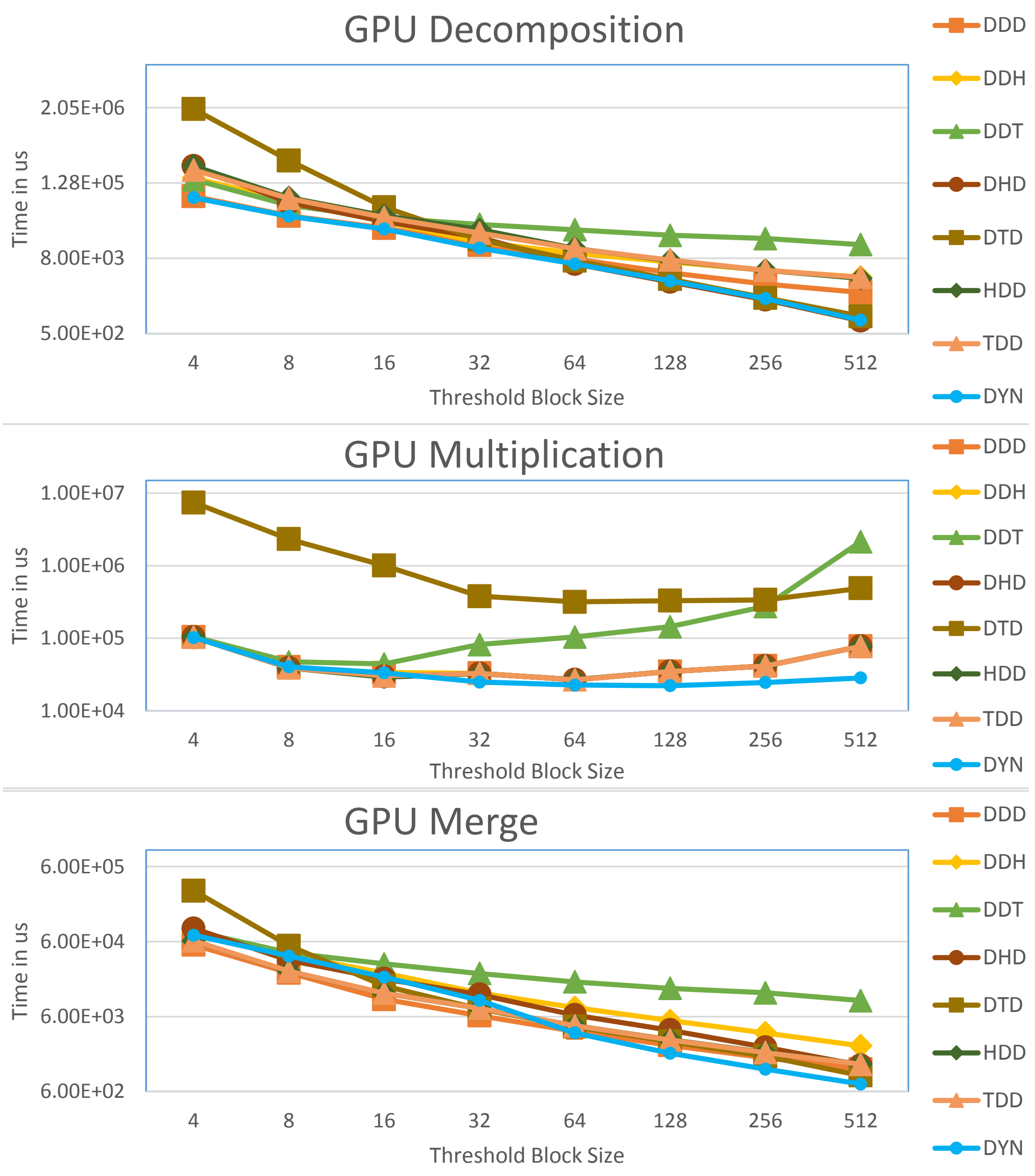

Figure 6.1: Strassen's Algorithm individual Steps results on Fermi GPU

From the individual step result graphs shown in figure 6.1 we can conclude that a few modes of programming always perform better than rest on a given architecture. On the GPU, when the modes are executing for a threshold block size of less-than '32'(the warp size), the

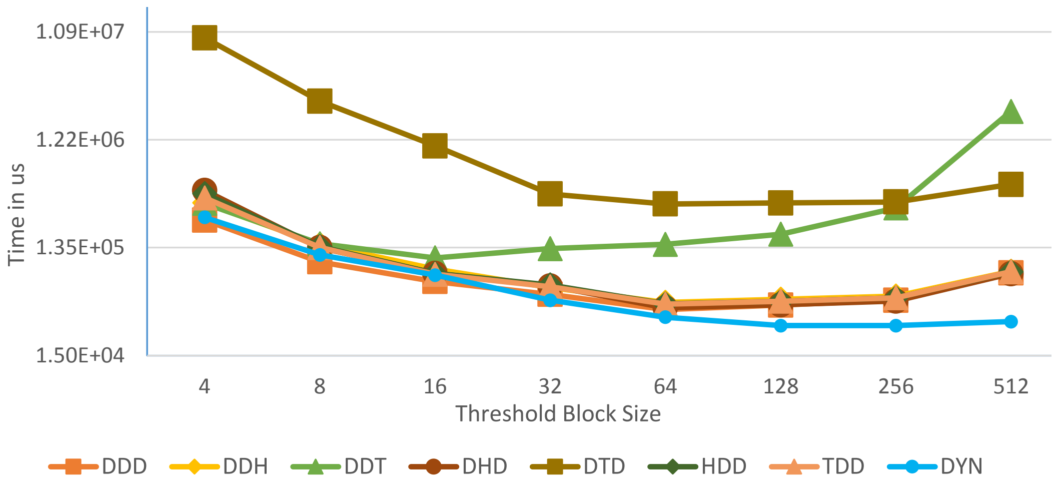

Figure 6.2: Total time taken for Strassen's Multiplication on Fermi GPU

performance of modes altered. This is due to the warp breaking during the memory access and executions. In these cases, the dynamic execution mode in the three steps of the algorithm gave at least as good a performance as other modes. We can observe the DDT and DTD modes performed badly in all the three steps for GPU. This could be attributed to the notion that GPUs are intended for data parallelism modes and these modes are mostly exploring task parallelism. For the dynamic mode in decomposition and merging steps we used an extended DHD mode of execution, and for multiplication we used extended DDD mode. To ensure that all the task and data parallel operations are enabled on available data during kernel launching, and not to serialize the parallel executions possible we launched one work-item per individual task on the available data. This resulted in improving the performance and the dynamic execution nature of the application. it was observed that the dynamic mode resulted in the best performance for the application. The graph in figure 6.2 shows the total time taken for all the 3 steps algorithm to get executed on GPU.

For the execution on CPU, similar to the GPU results, we observe that the dynamic mode of execution performed better than most other modes of execution, as shown by the graphs in figure 6.3. In the case of the CPU, the threads can handle high load, and we observe that only DTD performed badly. In DTD mode, the number of work-groups are fixed by the number of tasks. The work-groups are assigned to the compute units. In i3 processor we have four compute units. In the decomposition step, we have seven tasks and so the work-groups cannot be well distributed on the cores. This resulted in performance degrade as the resources were not fully utilized all the time. In case of multiplication we have only one work-group and so only one out of four compute units is being used. We can observe that in the merge step, the DTD mode performed comparably to the other modes as the resources were well utilized.

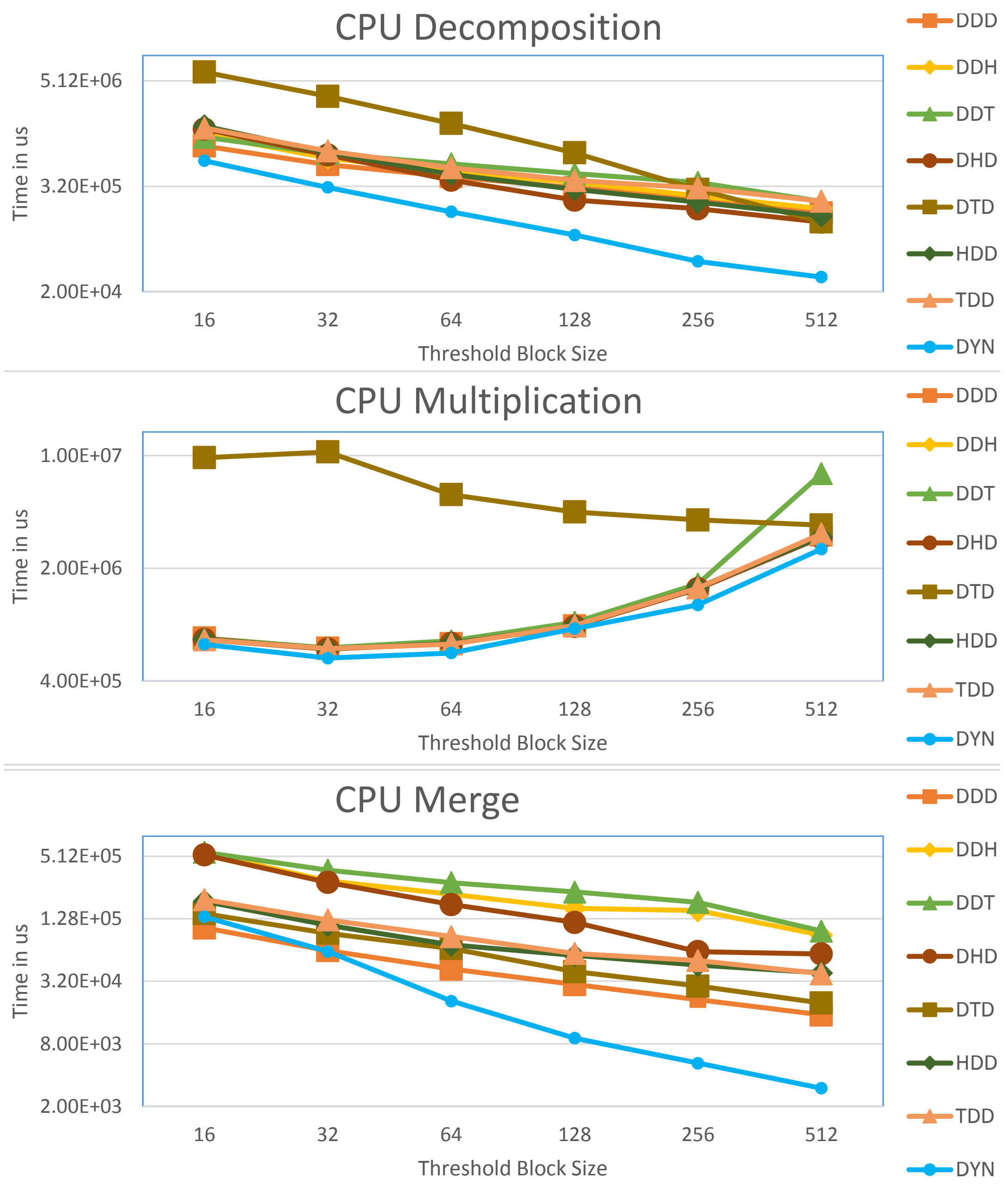

Figure 6.3: Strassen's Algorithm individual Steps results on muticore CPU

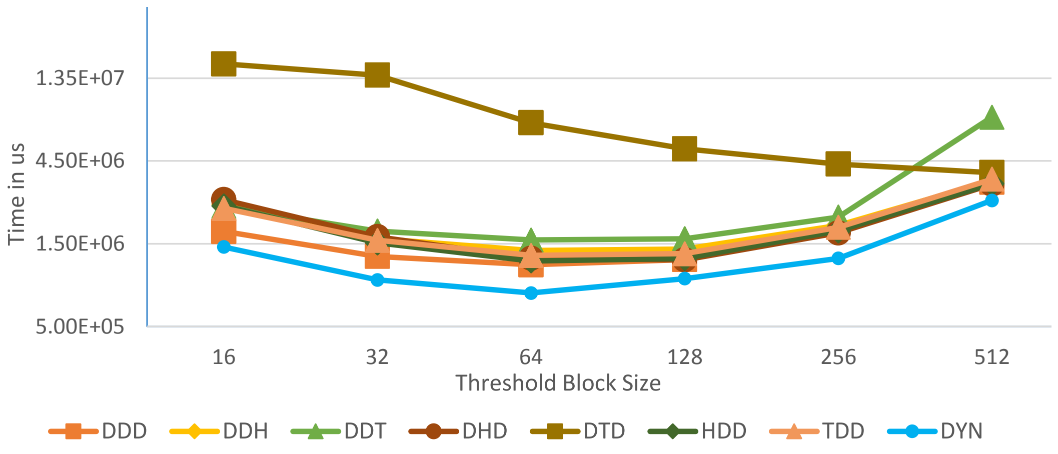

Figure 6.4: Total time taken for Strassen's Multiplication on multicore CPU

As a CPU core can support higher workload than the GPU processing cores, we implemented the complete data parallel mode DDD for all the steps in the dynamic mode. All the available data items to be executed are assigned with one work-item and the tasks are serialized. As observed from the results, the serialization of tasks did not degrade the performance, but instead improved the performance. This is due the cache hierarchy of CPU architecture. Here also, we can observe that the dynamic mode resulted in the best performance for the application. The graph in figure 6.4 shows the total time taken for all the 3 steps algorithm to get executed on CPU.

6.4 Summary

This chapter proposes the dynamic mode of programming in OpenCL, derived from the execution nature of Concurrent Collections. The results of the Strassen's algorithm implementation of all the modes on Fermi GPU and multi-core CPU where explained in this chapter. The comparison of the results to show that the dynamic mode performed better is done in this chapter. The degraded performance of few modes due to architectural constraints are also explained in these sections. The conclusions and the future work is mentioned in the next chapter.

Chapter 7

Conclusion and Future Work

The dynamic execution nature of the CnC programming environment allows execution of all possible concurrent steps at each stage of the application. This provides the basis for better performance of the application. Specific target devices are better suited for certain modes of execution. The architecture agnostic nature of CnC enables ease of programming but achieving fine-grained parallelism for the compute steps is a challenge. OpenCL provides a good platform for exploiting fine-grained parallelism. Using the iterative version of Strassen's matrix multiplication algorithm, we have proposed a framework to identify suitable modes of parallel programming in OpenCL for computational patterns such as divide-and-conquer and map-reduce. A similar analysis can be done for other computational patterns. Exploiting fine-grained parallelism on specific target devices requires a better understanding of the device architecture. It is challenging for auto-tuners to adapt an application to specific target devices. This work presents a basis for adapting the dynamic execution model and coarse-grained parallelism in compute steps expressed using the CnC paradigm by leveraging the suitability of OpenCL for fine-grained parallelism on specific target devices.

In future, our work can be extended for exploring the fine grained parallelism in auto-tuners. For different computational patterns, different modes of programming gives advantage on a given target architecture. This approach can be used to determine the complete set of the matching modes and patterns to the given architecture. These results can be used in auto-tuners of CnC for better exploration of parallelism in coarse grained computational steps.

Bibliography

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, et al. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736. ACM, 2009.

[3] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3):203–217, 2010.

[4] Zoran Budimlic, Aparna M Chandramowlishwaran, Kathleen Knobe, Geoff N Lowney, Vivek Sarkar, and Leo Treggiari. Declarative aspects of memory management in the concurrent collections parallel programming model. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 47–58. ACM, 2009.

[5] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In

Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1–12. IEEE, 2010.

[6] Jacques Cohen and Martin Roth. On the implementation of strassen's fast multiplication algorithm. *Acta Informatica*, 6(4):341–355, 1976.

[7] Hossam A. ElGindy and George Ferizis. On improving the memory access patterns during the execution of strassen's matrix multiplication algorithm. In *ACSC*, pages 109–115, 2004.

[8] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. Opencl and the 13 dwarfs: a work in progress. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, pages 291–294. ACM, 2012.

[9] Kathleen Knobe. Ease of use with concurrent collections (cnc). *Hot Topics in Parallelism*, 2009.

[10] Pai-Wei Lai, Humayun Arafat, Venmugil Elango, and P Sadayappan. Accelerating strassen-winograd's matrix multiplication algorithm on gpus. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 139–148. IEEE, 2013.

[11] Tim Mattson and Michael Wrinn. Parallel programming: can we please get it right this time? In *Proceedings of the 45th annual Design Automation Conference*, pages 7–11. ACM, 2008.

[12] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[13] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D Owens. Efficient computation of sum-products on gpus through software-managed cache. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318. ACM, 2008.

[14] David B Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.

[15] Paul Springer. BerkeleyŠs dwarfs on cuda. 2011.

[16] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, 2010.

[17] Jaspal Subhlok, James M Stichnoth, David R O'hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Notices*, volume 28, pages 13–22. ACM, 1993.

[18] Jaspal Subhlok and Bwolen Yang. A new model for integrated nested task and data parallel programming. In *ACM SIGPlan Notices*, volume 32, pages 1–12. ACM, 1997.

[19] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 2012.

[20] E. A. West and A. S. Grimshaw. Braid: integrating task and data parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, FRONTIERS '95, pages 211–, Washington, DC, USA, 1995. IEEE Computer Society.