OpenCL API Extensions to achieve Multi-level Parallelism for Efficient Implementation of Strassen's Matrix Multiplication on GPUs

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Technology

IN

Computational Science

by

Sivaramakrishnan Kasiviswanathan

Supercomputer Education and Research Centre

Indian Institute of Science

BANGALORE – 560 012

JULY 2013

Acknowledgements

I would like to thank Dr. J. Lakshmi for sharing her valuable knowledge. The life experiences she shares are always inspirational. I am greatly thankful to Prof. S. K. Nandy, for his kind attention to all those lengthy lab presentations and for giving inputs to lead my project into right direction. I want to thank Dr. Ranjani Narayan, Mrs. Mitsu Mehta and Mr. Ganesh Garga for listening to my work, observing even the smallest things that need to be corrected and for giving ideas to shape my work. I would like to extend my sincere thanks to the Supercomputer Education and Research Centre for all the facilities provided and also I would like to thank Dr. Uday Kumar Reddy for his kind help in providing exclusive access to the machines in his lab when I was in need.

I would like to thank my batchmates Vivek, Shila, Ankit, Mohit, Hari, Sanjeevmachan, Sayee, Nilesh, Anurag, Calvin, JP, Kalyan, Abbey and all the friends in SERC for their support and empathy. I would like to thank my friends in Instruementation and Applied Physics for their company. I would like to thank my friends Tamizhanban, Tamilarasan Sabapathy, Tamilarasan Subbramani and Sivakumar Dhandapani for their intense support in research. I would like to thank Gouman J Archer, Balamuralidharan, Tamilselvan Velusamy, Immanuel Raja and Brij Kishore for their frequent visits to my room to make sure that I go to bed late and wake up early. I would also like to thank Rashmi, Sowmiya, Leo, Sriram, Punya, Sreejith, Rakesh, Sanith and Pooja for the everlasting memories. I am very thankful to my friends Hero, Nithiyaraj, Krishnaraj, Syam, Ashokanna, Saami, Sanil and Mani for their critisism, birthday bumps and kicks. I would like to thank Anilanna, Rajeshanna, Javed, Sharath, Arun, Srini, Venki and Phani for the joyful moments. I thank everyone who has spent their precious time with me in these two years. Without your empathy and support this would have not been possible.

Abstract

Strassen's matrix multiplication algorithm is an efficient and widely used practical algorithm for matrix multiplication. In its basic form, the algorithm is a series of recursive steps to decompose the matrices, multiply intermediate matrices and another set of recursive steps to recompose the product matrix. Implementing the algorithm on a GPU requires it to be converted into an iterative algorithm and choosing the right mode of execution to achieve maximum parallelism. The iterative algorithm has aspects of both task and data parallelism. Modern GPUs and OpenCL constructs help to program and execute algorithms in either data parallel mode or task parallel mode. Exploiting hybrid parallelism that handles both data as well as task parallelism will be beneficial. However, the onus is on the programmer to understand the underlying device architecture and APIs to extract maximum parallelism in either data parallel mode or task parallel mode. We present some results of such platform aware implementations of Strassen's algorithm. We also present results of implementation of hybrid parallelism by programmatic tweaking of the features of a device that supports simultaneous execution of multiple independent kernels. In our implementation each kernel is a light-weight kernel that does a single-task and within each kernel data parallelism is achieved. We have evaluated the scope for efficient implementation of the iterative Strassen's algorithm in different programming modes and based on the results we propose a hybrid task-and-data-parallel OpenCL API extension that reduces the burden on the programmer as well as the run-time synchronization overhead to execute multiple kernels. This proposed API extension will provide native support for hybrid parallelism which is the best among all programming modes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Matrix multiplication is an elementary operation in a large class of computational problems. There are a number of algorithms for matrix-matrix multiplication that try to improve the running time from the naíve value of $O(n^3)$. Strassen's matrix multiplication algorithm is one such algorithm that uses the divide and conquer approach. In its basic form, it has three operations. First, a number of recursive steps to decompose the input matrices, the second to multiply intermediate matrices and a third operation again with a number of recursive steps to recompose the product matrix.

Inherently the algorithm shows parallelism in all the three steps. The amount and type of parallelism that can be exploited vary based on the level of recursion. A significant improvement in performance can be obtained by parallel execution of Strassen's algorithm on multi-core or many-core architectures. GPUs are the most common many-core devices in use today. Many-core architectures have cores that are designed for multithread support unlike single-thread cores in multi-core architectures. Numerically intensive algorithms optimized to harness the processing power of GPUs and designed to exploit the parallelism available show significantly better performance as compared to naíve implementations on CPUs. Optimizing code for CPUs typically requires a greater effort and even with heavily optimized code, the performance cannot match that of a GPU [19].

1.1 Execution environments for parallel programming on GPUs

The highly parallel structure of modern GPUs makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data can be done in parallel. It is becoming increasingly common to use a general purpose Graphics Processing Units (GPUs) for such algorithms.
There are two main vendors in the GPU market, namely, NVIDIA and AMD. Their GPU cards support parallel programming using parallel programming libraries such as CUDA (”Compute Unified Device Architecture”) and OpenCL (Open Computing Language). CUDA is specifically for NVIDIA GPUs whilst OpenCL is designed to work across a multitude of architectures including GPU, CPU and DSP (using vendor specific SDKs). These technologies allow specified functions (kernels) from a normal C program to run on the GPU's Functional Units (FUs). This makes C programs capable of taking advantage of a GPU's ability to operate on large matrices in parallel.

1.1.1 Data-parallel execution on modern GPUs

GPUs are generally suited to high-throughput type computations that exhibit data-parallelism to exploit the wide vector width SIMD architecture of the GPU. The following sections describe the architecture, global scheduling, SIMD execution model and memory hierarchy in GPUs and how these allow for data parallelism on the GPU.

GPU Architecture

In the CUDA environment, processing cores are located in a unified processor pool and grouped into clusters called Streaming Multiprocessors (SMs). Each SM contains an instruction cache, an instruction queue, a warp-scheduler, registers and FUs as shown in Figure. 1.1 [9].
AMD GPUs have a similar architecture where they have SIMD engines instead of SMs, processing elements (PEs) instead of FUs, sequencers instead of warp schedulers, wavefronts instead of warps etc [1]. Even though the layout of these elements is different for AMD;

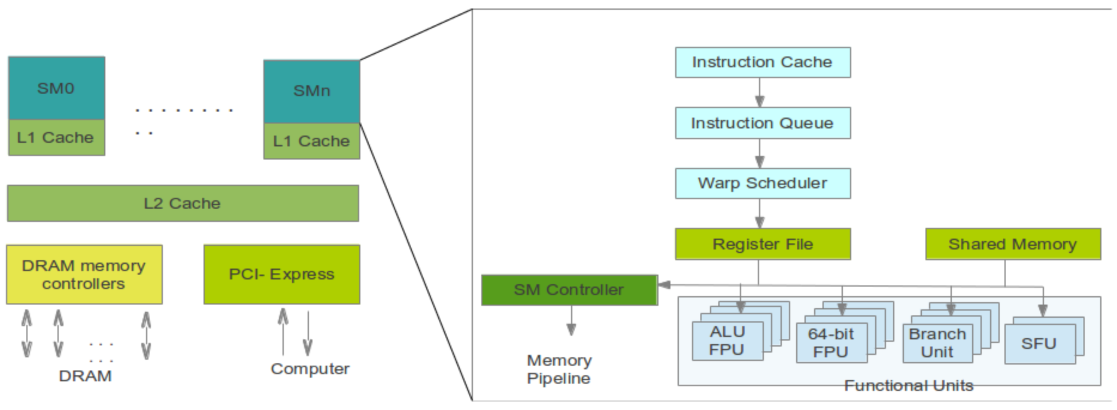

Figure 1.1: Hardware Architecture of a GPU.

the overall execution model is the same. In the rest of the paper, we have elected to use the NVIDIA architecture and OpenCL terminology for explanation.

Global Scheduling

A parallel program executing on a GPU is called a kernel. To exploit the processing power of the GPU, the kernel is spawned as several thousand parallel threads called work-items. Work-items are grouped into predefined size blocks called work-groups. The global scheduler of the GPU issues the work-groups to SMs to be executed. The global scheduler then launches one work-group on a single SM. A work-group cannot be split across multiple SMs. The global scheduler has to take into account hardware limitations such as the maximum number of work-groups that can be accommodated by the device and the maximum number of work-items that can be accommodated on a single SM.

Execution Model

The SMs in GPUs follow the Single Instruction Multiple Data (SIMD) model of execution. A predefined number of threads or work-items form an SIMD unit and all the work-items in the same unit are executed together on a single SM. The work-items are homogeneous in the sense that they all execute the same instruction or set of instructions. A work-group can be split into a number of such SIMD units (warps). A scheduler present in each SM called the warp scheduler chooses one among the available warps to be executed.

Memory

NVIDIA GPUs typically have a three tier memory hierarchy. Global memory is in the form of off chip DRAM modules and is shared across SMs. Within an SM, there are two kinds of memory a register file and shared memory. The registers in the register file are divided among all the work-items scheduled on an SM. The shared memory area facilitates communication among the work-items in a work-group.

1.1.2 Data parallelism and task parallelism implementations in OpenCL

To code a parallel application in OpenCL, the programmer is allowed to choose between implementations that exploit either data-parallelism or task-parallelism.

OpenCL Terminology

Parallelizable code to be executed on GPU devices is called a kernel. A kernel can be called by a multitude of data-parallel execution instances called work-items.

Data parallelism and task parallelism

Kernels are typically designed to be either data parallel or task parallel. In OpenCL terminology, the difference between the two is whether the same kernel or different kernels are executed in parallel.

A data parallel programming model defines a computation as a sequence of instructions applied to multiple data elements. Each work-item executes the same set of instructions but on different data elements. In the strictly data parallel model, there is a one-to-one mapping between work-items and data elements during kernel execution.

In the task-parallel mode, we can only launch a kernel that spawns a single work-group with a single work-item. For maximum parallelism, multiple such task-parallel kernels should be launched for execution on the device at the same time. However, at present, though most GPUs contain multiple FUs called Stream Processors (SPs) per SM, hardware such as instruction fetch and program counters are shared across them. So a single SM can execute only a work-group at a time. In this case each work-group contains a single work-item. For this reason, GPUs are inefficient in running different tasks in parallel. The task parallel mode is emulated by a relaxed data parallel model within a single kernel where different work-items are allowed to choose different execution paths. Since this is achieved by branching on the basis of the work-item id, it is essentially a serialized execution and hence inefficient.

1.2 Related Work

The paper by Cohen et. al. [4] gives time-formula for the overhead of recursion for the Strassens algorithm. It describes a memory access pattern implementation for the matrices that reduces the memory access time. In the paper by West et. al. [20], the Mentat Programming Language (MPL) has been used 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 paper by Bikshandi et. al. [3] implements hierarchically tiled arrays in Object Oriented programming languages for the development of high-performance parallel code. In that work a significant amount of data-parallelism and only a limited amount of task-parallelism is achieved. The paper by Subhlok et. al. [17] discusses about the task and data parallelism on a multi-computer and it discusses ideas on how to expose these tradeoffs to the compiler. Another paper by them [18] tries to achieve nested task and data-parallelism in High Performance FORTRAN (HPF). It also proposes extension implementations using directives which has later been approved in HPF 2.0.

A paper by BenHassen et. al. [2] proposes a task-and-data-parallel programming language called Orca that uses a concept of shared objects which are stored in one or multiple of the processors. Task-parallelism is achieved using remote process forking and having them to communicate through these shared objects. Data-parallelism is achieved by making partitioned objects work in parallel. The work by Dhagat et. al. [5] achieves as much as 75% of efficiency with the combined task-and-data-parallel execution as compared to the pure data-parallel execution in UC language - a parallel programming extension to the C language. Combined task-and-data parallelism is achieved by asynchronous send and receive calls. The programmer needs to be aware of deadlocks that might occur in this implementation.

Kambadur et. al. [10] introduce PFunc a library for task-parallelism to efficiently parallelize HPC applications using SPMD model of parallel programming. A paper by Chao-Tung et. al. [21] achieves hybrid parallel programming on Multi-GPU clusters by using a combination of Message Passing Interface (MPI) and CUDA where task parallel work is distributed among different GPU cards and within each GPU card fine grained data-parallelism is achieved. Since they share the work among them, the GPU utilization is low. Vejarano et. al. [19] derive an analytical cost-model for mapping an application to different computing resources in CPU-GPU based co-design environment. And a paper by Elangovan et. al. [6] facilitates ease of programming in OpenCL using OmpSs - a language extending the OpenMP language inspired from the StarSs programming model and Nanos run-time system but their focus is only on data-parallel programming execution.

1.3 Need for extending the OpenCL API to support multi-level parallelism

The work-items and work-groups can be easily depicted using an index space called as an ’NDRange’ (N-Dimensional Range). This index space can be of 1, 2 or 3 dimensions. The execution instance of a kernel in an NDRange space is called an NDRangeKernel.

All the work-items in an NDRangeKernel are grouped into fixed size work-groups and execute the same set of instructions. These work-items can be indexed using three indices namely

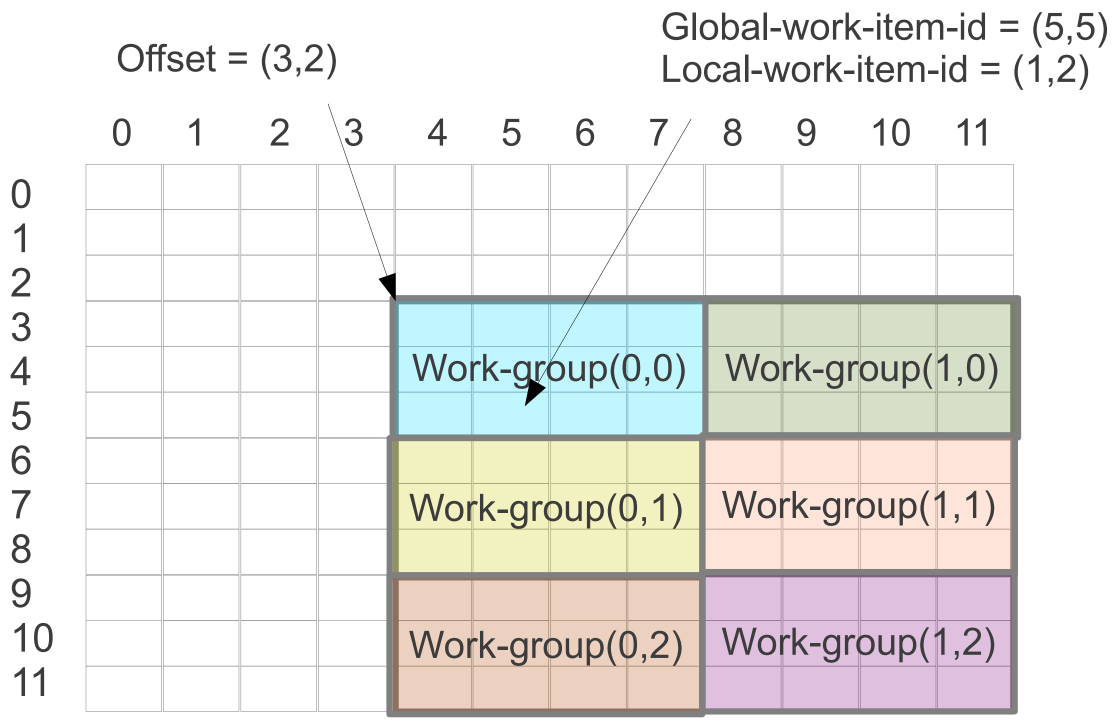

Figure 1.2: NDRange index space

work-group-id, global-work-item-id and local-work-item-id as shown in Figure. 1.2. They operate on different data based on these indices 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 and are scheduled to run on an SM concurrently [12] [15]. This is essentially a data parallel mode of execution.

To achieve task parallelism, the work-items need to be heterogeneous. For this, there needs to be a branch in the NDRangeKernel code, based on the work-item-id inside a warp. But in this case, the work-items are essentially serialized due to the inherent SIMD nature of GPUs. A smart programmer can put these task-parallel work-items in different warps to achieve a particular amount of parallelism. But experiments done in section 3.3.2 suggest that this is also limited by the number of warp-schedulers available. Therefore achieving task-parallelism in an NDRangeKernel is a tedious job.

Due to the SIMD execution model and the number of warp-schedulers being far less than the Load-Store units, GPUs more easily lend themselves to achieve data-parallelism. However there needs to be a support in OpenCL to facilitate execution of hybrid kernels that are both task-and-data-parallel. Only two levels of hierarchical grouping of the work-items are supported by NDRangeKernel for now. Whereas, in principle, it should support any number of levels or dimensions of hierarchical grouping. In this work, we introduce a third level of hierarchical grouping at the kernel level, that facilitates hybrid parallel execution of task-and-data-parallel modes at the same time.

1.4 Organization of the thesis

On devices like GPUs, Strassen's recursive algorithm is rather inefficient if not impossible. There are iterative implementations of Strassen's algorithm that are proven to be more efficient than the recursive one [4] . Efficiently implementing the iterative version of this algorithm involves making a choice between the type and the amount of parallelism that can be exploited in each step of the algorithm. In order to make appropriate choices, the programmer needs to be aware of details of the underlying architecture and run time system as well as the limitations of the programming language used to program the device.
OpenCL 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]. We propose API extensions in OpenCL which will automatically make implementation mode choices and make it easy for the programmer to specify the parallelism. This thesis is divided into following chapters. Chapter 2 is about the formulation and types of implementation of the Strassen's algorithm. Chapter 3 explains the classification of different parallel modes of execution. Chapter 4 derives a theoretical cost model for calculating the execution time based on the device parameters and the execution mode. Chapter 5 describes how knowledge of underlying device architecture can be leveraged to extract maximum parallelism and presents an API extension to achieve the same. Followed by the conclusion in Chapter 6.

Chapter 2

Strassen's Matrix Multiplication Algorithm

Strassen's matrix multiplication algorithm is one of the widely used algorithms used in practice which has broken the execution-time barrier of $O(n^3)$ for the first time. Most of the modern and more efficient algorithms for matrix multiplication are also based on the principle of Strassen's algorithm. The variable amount of parallelism that can be exploited in each stage of the algorithm makes it the right candidate for exploring hybrid-parallelism.

2.1 Implementation of Strassen's algorithm on GPUs - recursive to iterative for maximum parallelization

There are two implementations of Strassen's algorithm used in practice. The recursive algorithm appeals to programmers because it encapsulates neatly the application of the algorithm in reducing data sets. However, to implement recursion on machines there is a practical limitation in terms of how much memory of the process can be used in stack. This limitation can sometimes be prohibitive and does not allow the algorithm to scale beyond certain data sizes. This is where iterative algorithms gain. They are scalable and render themselves easily to any modes of task, data or hybrid parallelism. We explore both of them to find the one which is better suited to be implemented on parallel architectures.

2.1.1 Recursive Strassen's Algorithm

In the normal matrix-matrix multiplication, the resultant matrix is calculated using eqn. 2.1:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \tag{2.1}$$

This involves 8 multiplications and 4 addition operations as in eqn. 2.2:

$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned} \tag{2.2}$$

For input matrices of bigger sizes, each of the elements (a-h) above can be a matrix block and to calculate the overall product these elements can be further subdivided and solved in the same manner. For a matrix of size N the multiplication involves $O(N^3)$ multiplication operations. Strassen's algorithm asymptotically reduces the running time to $O(N^{2.807})$ by doing only 7 multiplication operations instead of 8. The steps involved in Recursive Strassen's Algorithm can be explained with the equations that follow [8].

Strassen's algorithm consists of three steps.

i) Decompose step

In this step, the input matrix of size $N \times N$ is subdivided into 4 sub-matrices of size $\frac{N}{2} \times \frac{N}{2}$. As given by the Strassen's formulae 7 basic operations (add, sub & copy) are done between these four sub-matrices to produce 7 intermediate matrices. This is done in recursion until we obtain intermediate blocks of size $n \times n$. The threshold-block-size n is chosen by the programmer based on some of the system parameters like cache sizes, maximum allocatable global memory size, etc. It also specifies the base condition for stopping the recursion.

$$\begin{aligned} A_1 &= a \\ A_2 &= a+b \\ A_3 &= c+d \\ A_4 &= d \\ A_5 &= a+d \\ A_6 &= b-d \\ A_7 &= a-c \end{aligned} \tag{2.3}$$

$$\begin{aligned} B_1 &= g-h \\ B_2 &= h \\ B_3 &= e \\ B_4 &= f-e \\ B_5 &= e+h \\ B_6 &= f+h \\ B_7 &= e+g \end{aligned} \tag{2.4}$$

ii) Multiplication step

The intermediate matrices obtained in equations 2.3 and 2.4 are multiplied using normal matrix multiplication to obtain intermediate product matrices $P_1, P_2, ..., P_7$ given by eqn. 2.5 .

$$\begin{aligned} P_1 &= A_1.B_1 \\ P_2 &= A_2.B_2 \\ P_3 &= A_3.B_3 \\ P_4 &= A_4.B_4 \\ P_5 &= A_5.B_5 \\ P_6 &= A_6.B_6 \\ P_7 &= A_7.B_7 \end{aligned} \tag{2.5}$$

iii) Recompose step

This step involves merging the intermediate product matrices $P_1, P_2, ..., P_7$ to calculate the final resultant matrix P.

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned} \tag{2.6}$$

The steps in equations 2.3, 2.4 and 2.6 should be recursively done. In each recursion, the problem size differs from the previous recursion.

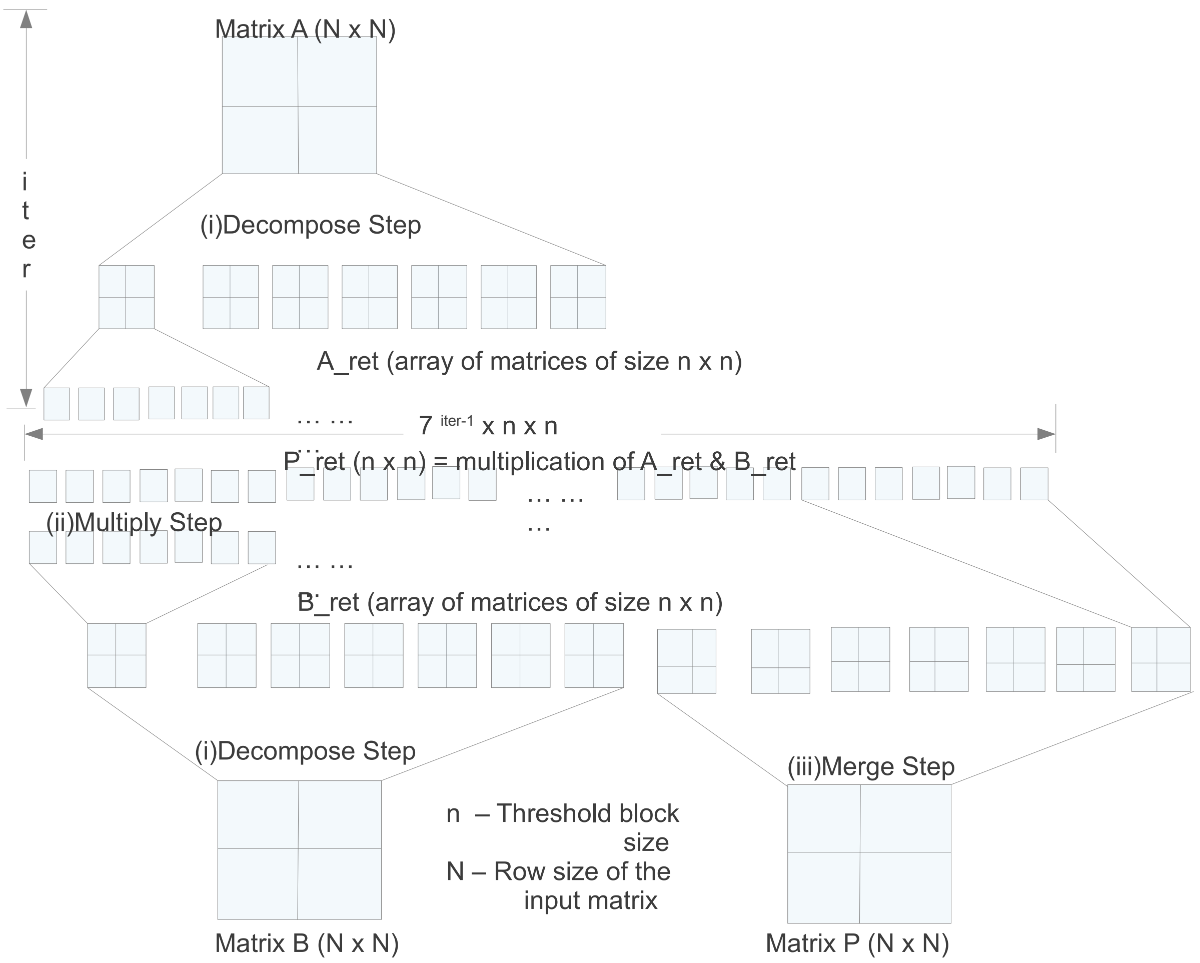

Figure 2.1: Strassen's Matrix Multiplication Algorithm

2.1.2 Iterative Strassen's Algorithm

Strassen's algorithm is originally a recursive one. There is a certain amount of overhead on running time imposed by the recursion [J. Cohen 1976]. Converting it into an iterative one will help in performance by improving the memory access patterns. Especially for GPUs recursion overhead and memory access time are crucial.

The iterative algorithm starts with an input matrix A of size $N \times N$ to decompose. In the first iteration, this matrix is decomposed into 7 matrices of size $\frac{N}{2} \times \frac{N}{2}$ namely $A_1, A_2, ..., A_7$. In the next iteration, each of these matrices is further subdivided into 49 matrices of size $\frac{N}{4} \times \frac{N}{4}$ namely $A_{11}, A_{12}, ..., A_{17}$, $A_{21}, A_{22}, ..., A_{27}$, ..., $A_{71}, A_{72}, ..., A_{77}$. This continues till the size $n \times n$ where we have intermediate matrices $A_{11}, A_{12}, ..., A_{17}$, $A_{21}, A_{22}, ..., A_{27}$, ..., $A_{x1}, A_{x2}, ...$, A_{x7}. ($x = 7^{Iter-1}$, Iter is the number of iterations needed to reach $n \times n$) Similar calculations are done for the other input matrix B in parallel to obtain $B_{11}, B_{12}, ..., B_{17}$, $B_{21}, B_{22}, ..., B_{27}$, ..., $B_{x1}, B_{x2}, ..., B_{x7}$ respectively.

In the multiplication step these 7^{Iter} intermediate matrices of A are multiplied with the 7^{Iter} intermediate matrices of B to get 7^{Iter} intermediate product matrices $P_{11}, P_{12}, ..., P_{17}$, $P_{21}, P_{22}, ..., P_{27}$, ..., $P_{x1}, P_{x2}, ..., P_{x7}$. Finally these intermediate product matrices are merged in the recompose step to give the final resultant product P.

In first iteration of the recompose step, P_{11}, P_{12} , ..., P_{17} , $P_{21}, P_{22}, ..., P_{27}$, ..., $P_{x1}, P_{x2}, ..., P_{x7}$ are merged to give $P_{11}, P_{12}, ..., P_{17}$, $P_{21}, P_{22}, ..., P_{27}$, ..., $P_{\frac{x}{7}1}, P_{\frac{x}{7}2}, ..., P_{\frac{x}{7}7}$. The formulae for recomposing are given by equation 2.6 [7]. This algorithm is pictorially explained in Figure. 2.1.

2.2 Summary

This chapter explained the Strassen's algorithm, the various steps involved and the different variants in the implementation of the algorithm. It also gives an insight of how much is the type and amount of parallelism available in each step. The following chapters will explain the different modes of execution of iterative implementation of the algorithm and techniques to exploit maximum parallelism.

Chapter 3

Classification of Parallel Modes of Execution

OpenCL offers two modes of parallelism namely, task parallelism and data parallelism [11]. The programmer must choose the appropriate mode of parallelism for the application. In chapter 2 both modes of parallelism for Strassen's algorithm have been explored as the algorithm is task-parallel in nature and GPUs inherently support data-parallelism. Another mode of parallelism that allows some amount of hybrid parallelism to be achieved is the relaxed data-parallel mode where, task-parallel work-items take different execution paths. The following subsections describe these various parallel modes of execution and performance results of the implementations on NVIDIA Kepler K20 GPU. Also they describe the theoretical model for optimal performance in each mode and compare the practical results with the results expected using the theoretical model.

3.1 Classification of parallel implementation modes using The K-G-I model

This thesis uses a 3-tuple Kernels, work-Groups, work-Items (K-G-I) for describing the parallel implementation modes. The classification is based on how many kernels can be executed at a time. It can be S = single kernel (as described in section 3) or M = multiple kernels (as described in section 4). Work-groups and work-items can be hoMogeneous (M) or heTerogeneous (T) depending on whether they take the same execution path or different execution paths.

The programmer can make use of the relaxed data-parallel model to achieve some amount of task-parallelism using the NDRangeKernel itself. The programming mode choice needs to be made from one among the following modes. The classification is based on how many kernels can be executed at a time (single or multiple) and how the kernel code has been written. The kernel code can be written in such a way that the work-groups take the same (homogeneous group) or different execution paths (heterogeneous groups) and the work-items inside a work-group can take same (homogeneous items) or different (heterogeneous items) execution paths. All these programming mode choices are explained for the second iteration in the decompose step of the Strassen's algorithm for the multiplication of 1024 x 1024 matrices in which we decompose 49 blocks of size 256 x 256 to produce 343 blocks of size 128 x 128. In the following diagrams, work-items or work-groups are colored based on the operation they do. If two of the boxes are of same color and then it means they follow the same execution path. If an output matrix is colored, then only one of the work-groups or work-items with the same color have created it.

3.1.1 Single-kernel, Homogeneous Groups, Heterogeneous Items (S-M-T) mode

This relaxed data parallel mode of execution has heterogeneous work-items inside a work-group and all the work-groups execute same set of instructions.

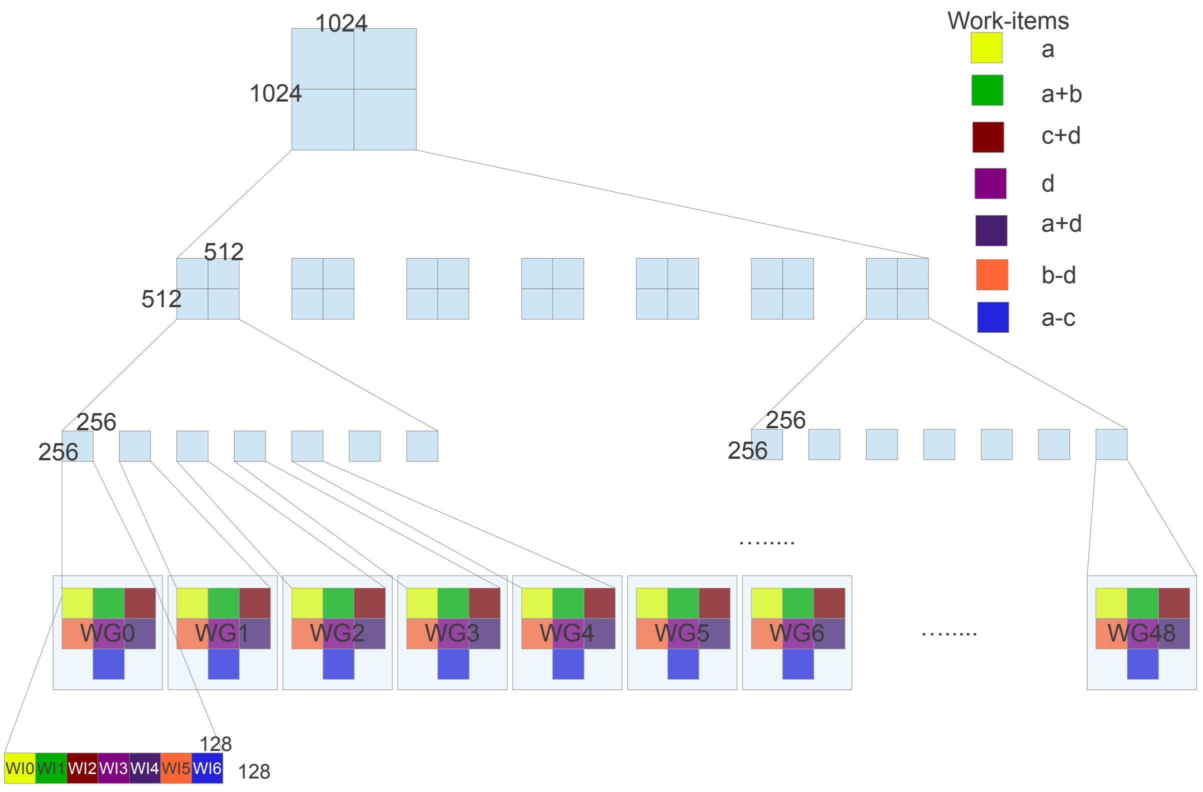

Figure 3.1: S-M-T mode of execution

Each work-group contains 7 work-items each doing one of the operations in eqn. 2.3 or 2.4. The number of work-groups increases with the iteration number. In the example shown in Figure. 3.1, in the last iteration of the decompose step, each work-item does 128 x 128 calculations. Work-items inside a work-group take the branch corresponding to their work-item-id. This is explained by the Pseudocode 28. So inside an SM, each work-item is mapped to an FU/SP. To achieve maximum parallelism in this mode, the active work-items have been kept in different warps along with dummy work-items. But still the parallelism is limited by the number of Warp-schedulers available.

3.1.2 Single-kernel, Heterogeneous Group, Homogeneous Items (S-T-M) mode

This relaxed data parallel mode of execution has homogeneous work-items in a work-group. And the work-groups are heterogeneous. There are always 7 heterogeneous work-groups each

Algorithm 1 Kernel code for S-M-T mode

```
Input:matrix A of size $2N \times 2N$
Output: Array of matrices Aret of size $7 \times N \times N$
switch (itemid)
    case (0) :
        for {i = 0 to N }
            for {i = 0 to N }
                $Aret_{group\_id,0}[i \times N + j]$ = $A_{group\_id}[i \times 2N + j]$ //a
            endfor
        endfor
    endcase
    case (1) :
        for {i = 0 to N }
            for {i = 0 to N }
                $Aret_{group\_id,1}[i \times N + j]$ = $A_{group\_id}[i \times 2N + j]$ + $A_{group\_id}[i \times 2N + j + N]$
//a + b
            endfor
        endfor
    endcase
            .
            .
            .
    case (6) :
        for {i = 0 to N }
            for {i = 0 to N }
                $Aret_{group\_id,6}[i \times N + j]$ = $A_{group\_id}[i \times 2N + j]$ - $A_{group\_id}[2N \times N + i \times 2N + j]$
// a - c
            endfor
        endfor
    endcase
end switch
```

doing a single operation on the given data as shown in the Pseudocode 28. Each SM has its own instruction cache and program counters, so different SMs can follow totally different execution paths. The number of work-items increases with the iteration number. In this case, the number of work-items that can be scheduled at a time per SM becomes a limitation. After 3 iterations the number of work-items per work-group is increased to 2401 which cannot be accommodated on an SM. In the example step shown in Figure. 3.2, there are 7 work-groups; each work-group contains 49 work-items. Each work-item does 128 x 128 calculations.

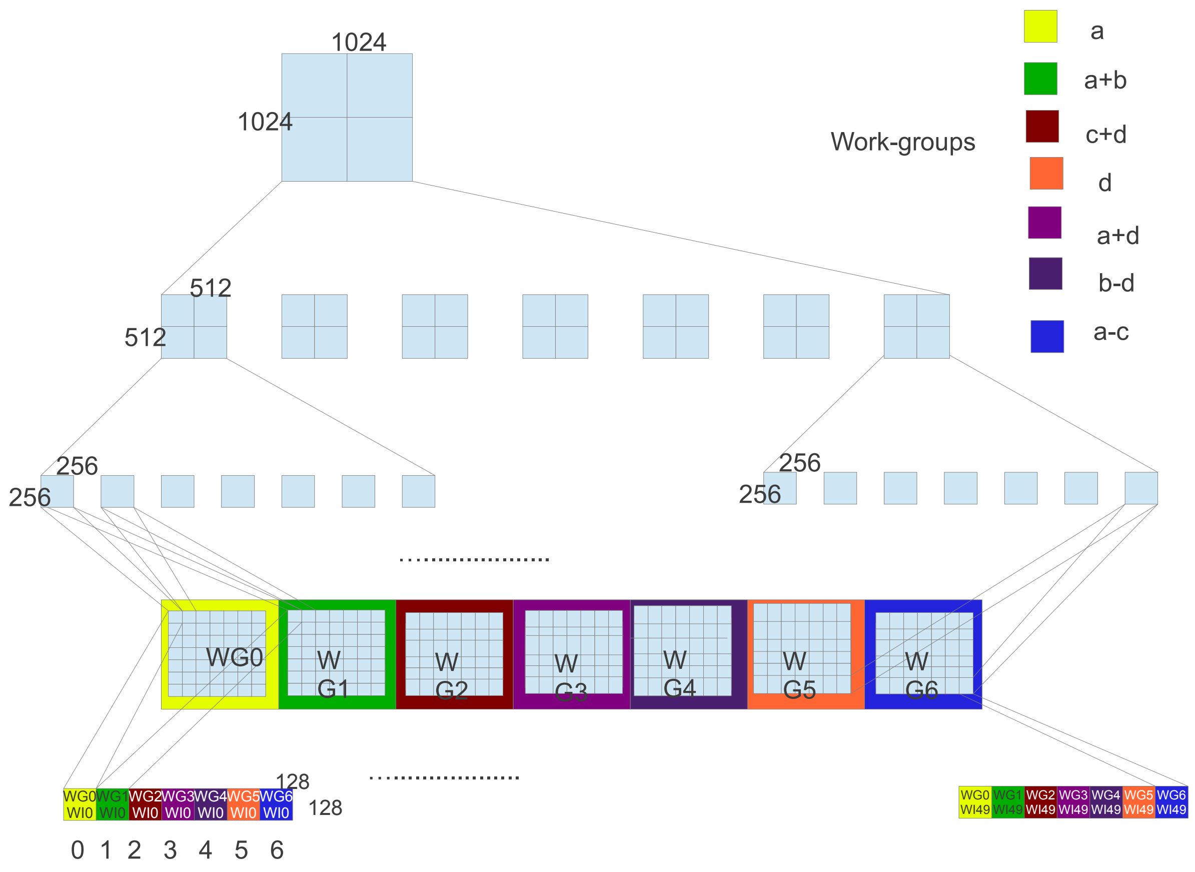
1024
1024
Work-groups
a
a+b
c+d
d
a+d
b-d
a-c
512
512
256
256
256
256
WG0
WG1
WG2
WG3
WG4
WG5
WG6
128
128
WG0 WG1 WG2 WG3 WG4 WG5 WG6
WI0 WI0 WI0 WI0 WI0 WI0 WI0
0 1 2 3 4 5 6
WG0 WG1 WG2 WG3 WG4 WG5 WG6
WI49 WI49 WI49 WI49 WI49 WI49 WI49

Figure 3.2: S-T-M mode of execution

3.1.3 Single-kernel, Homogeneous Groups, Homogeneous Items (S-M-M) mode

This is a pure data parallel mode and all the work-items have the same instructions. But based on their work-item id, the data they access differs. This mode is expected to perform better than the other two modes due to the inherently data-parallel nature of the GPUs. But this mode of execution does not exploit any task-parallelism. In the example step implemented in this mode, ideally one needs to launch 49 work-groups of 128x128 work-items each. These work-items may each work on 4 input data-elements and do 7 operations to calculate the 7 output data-elements as shown in the Figure. 3.3. But the limitation on the number of work-items

Algorithm 2 Kernel code for S-T-M mode

```
Input : matrix A of size $2N \times 2N$
Output : Array of matrices Aret of size $7 \times N \times N$
switch (groupid)
    case (0) :
        for {i = 0 to N }
            for {i = 0 to N }
                $Aret_{item\_id,0}[i \times N + j]$ = $A_{item\_id}[i \times 2N + j]$ //a
            endfor
        endfor
    endcase
    case (1) :
        for {i = 0 to N }
            for {i = 0 to N }
                $Aret_{item\_id,1}[i \times N + j]$ = $A_{item\_id}[i \times 2N + j]$ + $A_{item\_id}[i \times 2N + j + N]$ //a + b
            endfor
        endfor
    endcase
            .
            .
            .
    case (6) :
        for {i = 0 to N }
            for {i = 0 to N }
                $Aret_{item\_id,6}[i \times N + j]$ = $A_{item\_id}[i \times 2N + j]$ - $A_{item\_id}[2N \times N + i \times 2N + j]$
// a - c
            endfor
        endfor
    endcase
end switch
```

that can be launched per SM allows one to only to launch 32 x 32 work-items. In this example step shown, there are 49 work-groups and each having 32×32 work-items. Each work-item does 7 operations on 4 x 4 data elements as in Figure. 3.5.

These steps are explained in Pseudocode 13.

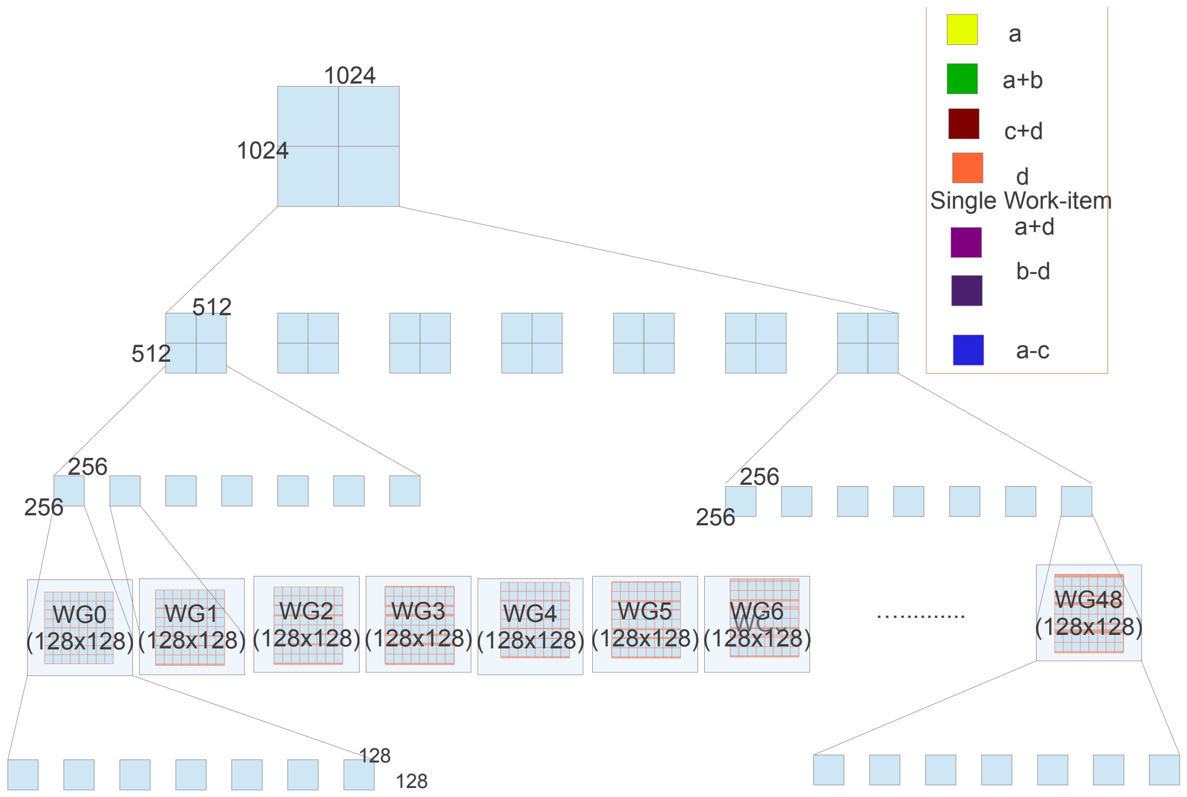

Figure 3.3: S-M-M mode of execution when there is no limitation on the number of work-items launched

Algorithm 3 Kernel code for S-M-M mode

```
**Input :** matrix A of size $2N \times 2N$
**Output :** Array of matrices Aret of size $7 \times N \times N$
$workAreaSize = \frac{N \times N}{MAX\_WORK\_ITEM\_SIZE}$ ; $offset = item\_id \times workAreaSize$

    for {i = 0 to workAreaSize }
        $Aret_{group\_id,0}[offset + i] = A_{group\_id}[4 \times (offset + i)]$ //a

        $Aret_{group\_id,1}[offset + i] = A_{group\_id}[4 \times (offset + i)] + A_{group\_id}[4 \times (offset + i) + N]$ //a + b

        .
        .
        .
        $Aret_{group\_id,6}[offset + i] = A_{group\_id}[4 \times (offset + i)] - A_{group\_id}[4 \times (offset + i) + 2N \times N]$ // a - c
```

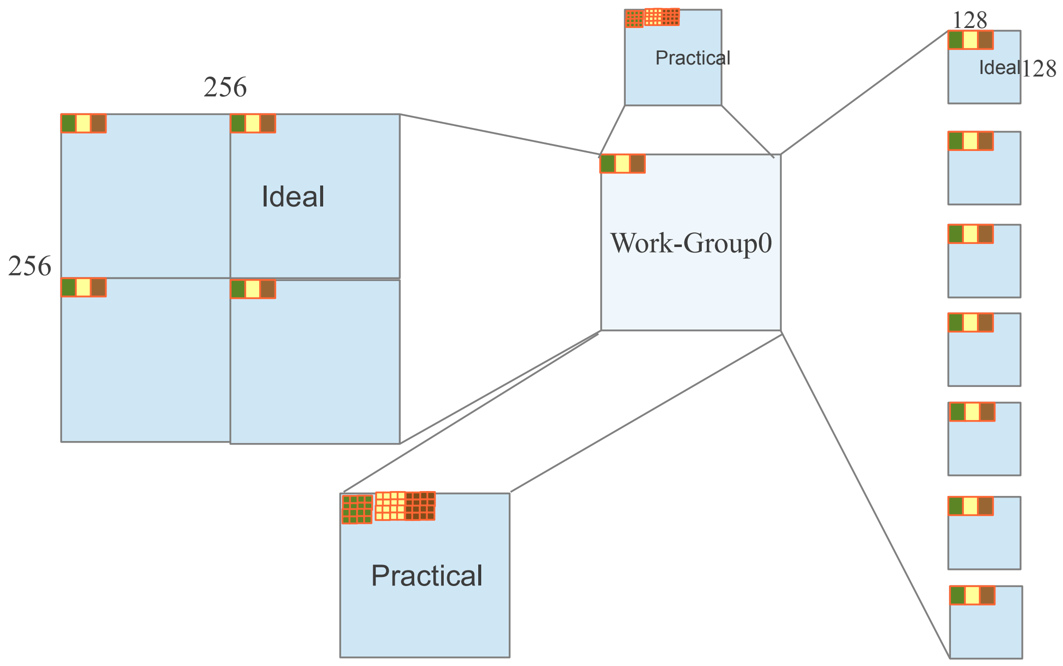

Figure 3.4: Work distribution in Ideal case

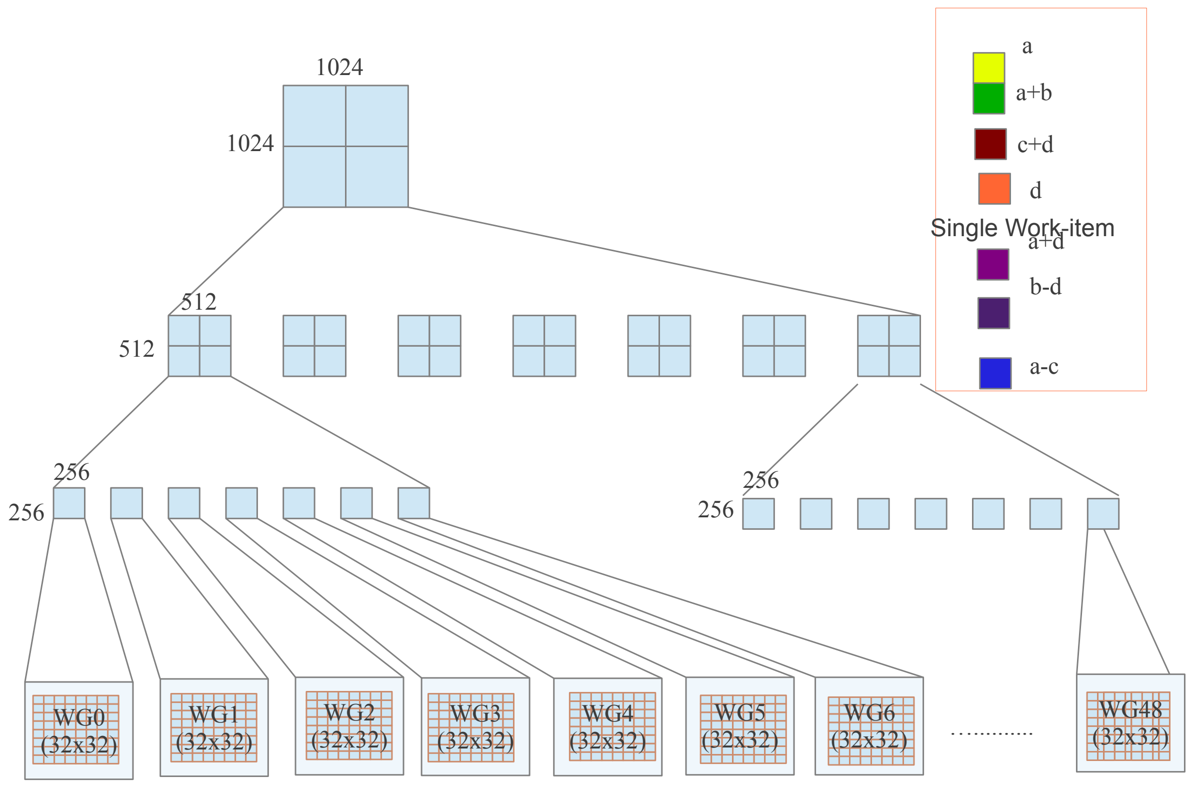

Figure 3.5: S-M-M mode of execution with practically possible number of work-items launched

3.2 Summary

This chapter introduced a classification of parallel execution modes based on the 3-tuple K-G-I model and explained the difference between each of them and the limitations of each imposed by the hardware. The following chapters will present the formulation for the execution time based on these modes and hardware parameters. They try to find the hardware parameters using pilot programs and then introduce another mode of hybrid execution that is only supported on the recent hardware.

Chapter 4

Theoretical Cost Model

A cost model has been devised to obtain theoretical calculations for each operation in the iterative algorithm by exploring the underlying architectural details of the NVIDIA Fermi and Kepler K20 GPUs. The device properties of both these GPU architectures are given in table 4.1. Pilot programs have also been devised and executed on both these GPUs to arrive at the practical numbers to compare with the theoretical calculations for each operation in the algorithm and the results are discussed in this chapter.

Table 4.1: Architecture Parameters

Parameter	Kepler K20 (Tesla K20c)	Fermi (Tesla c2070)
Global Memory (MB)	4799	5375
Max. Allocatable Memory (MB)	1199	1343
Local Memory (kB)	48	48
Max. Clock (MHz)	705	1147
Warp Size	32	32
Max. No. of work-items per work-group	1024	1024
Max. work-item sizes in each dimension	(1024, 1024, 64)	(1024, 1024, 64)
No. of SMXs/SMs	13	14
No. of SPs per SMX/SM	192	32

4.1 Cost model for different modes in K-G-I classification

The following cost model formulates the time taken for execution of the G work-groups having I work-items in each of them is given by eqn. 4.1.

$$T = L_g \times \sum G_{steps} + L_i \times \sum (G_{steps} \times I_{steps}) + B \times \sum (S \times C \times G_{steps} \times I_{steps}) \quad (4.1)$$

Where, G_{steps} is the number of steps taken to G number of work-groups. It is given by eqn. 4.2.

$$G_{steps} = \left\lceil \frac{G}{G_{prac}} \right\rceil \quad (4.2)$$

And, I_{steps} is the number of steps taken to run all I work-items in a work-group. It is given by eqn. 4.3.

$$I_{steps} = \left\lceil \frac{I}{I_{prac}} \right\rceil \quad (4.3)$$

The parameters in these equations are of two types.

i) Parameters that are dependent on the programming mode

G_{prac} is the maximum number of work-groups that can be scheduled at a time on the device ($\leq$ number of SMs),
I_{prac} is the maximum number of work-items that can be scheduled at a time on the device ($\leq$ maximum work-group size supported by the device).
S is the size of the data elements need to be processed, C is the number of calculations done by a work-item.
ii) Parameters due to the programmatic limitation and are dependent on the architecture
G_{launch} is the maximum number of work-groups that be launched by a kernel execution call,
I_{launch} is the maximum number of work-items that can be launched by a kernel execution call,
L_g is the Latency for launching a work-group on an SM,
L_i is the Latency for launching set of work-items that are executed in parallel,
B is the Time taken for doing a single computation.

4.2 Pilot Programs for finding the behavior of the runtime system (finding out value for Iprac)

Even though most of the parameters can be substituted from the parameters from the white-paper, the practical parameter I_{prac} are not explicitly stated and varies based on a number of other parameters as well as the device's compute capability. So it has to be properly found out by running pilot programs. The value of I_{prac} is also dependent on how the work-items take branches. For homogeneous work-items and for heterogeneous work-items, separate pilot programs have been run to find out the system behavior in scheduling.

4.2.1 Pilot program for homogeneous work-items

A pilot program has been run for just copying the data to the device memory; add it to itself and to write the result to another array on memory (no local/shared memory is used). Each work-item works on 128 data-elements. All the work-items are homogeneous. Various readings are

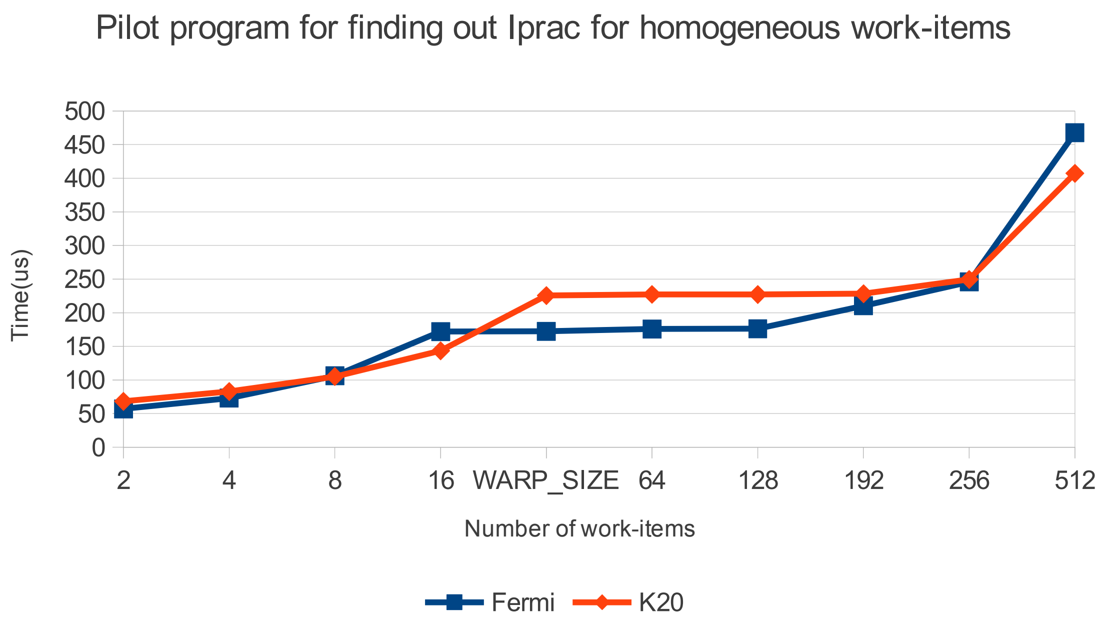

Figure 4.1: Pilot program for homogeneous work-items

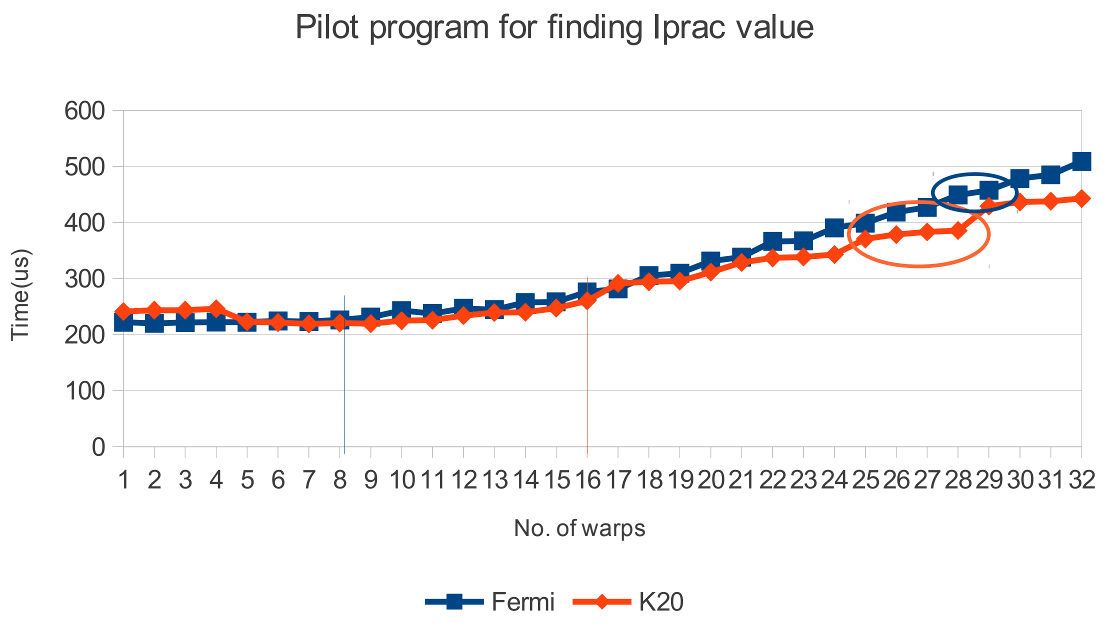

Figure 4.2: Pilot program for heterogeneous work-items

taken by increasing the number of work-items by a factor of two and the kernel execution time is measured. From these time values, one can derive what is the actual concurrency level inside an SM.

From the graph results in Figure. 4.1 it can be seen that the number of work-items running in parallel are limited to 128 in Fermi and 192 in Kepler K20 after which the work-items are serialized.

4.2.2 Pilot Program to find the number of heterogeneous work-items launched at a time

Another pilot program containing all heterogeneous work-items was also run and the time taken for execution was observed by increasing the number of warps.
One can see that there is no increase in execution time when increasing the warp-size till 8 warps in Fermi and till 16 warps in Kepler K20 after which there is a step increase in execution time is for every 2 warps in Fermi and every 4 warps in Kepler K20. The number of warp schedulers available in Fermi and Kepler K20 are 2 and 4 respectively. Fig. 4.2 suggests that there need to be at least 8 warps in Fermi and 16 warps in Kepler K20 to hide the pipeline latency. So the I_{prac} value can be approximated to 8 for Fermi and 16 for Kepler K20.
Experiments have also been conducted to find out the parameters namely, work-group launch latency (L_g), launch latency for work-items (L_i) and time taken for a single computation (B) for Kepler K20. These values are found out to be 13, 0 and 1 respectively.

4.3 The theoretical execution time calculated based on the analytical model for the algorithm

Presented in this subsection is the execution time calculations for the decompose step of the algorithm based on the formula for running time given in eqn. 4.1

The theoretical execution time calculations done for the Kepler K20 are presented in three modes of executions namely S-M-T, S-T-M and S-M-M.

4.3.1 Single-kernel, Homogeneous Group, Heterogeneous Items (S-M-T) mode

The S-M-T mode has an increasing number of work-groups and a constant number of work-items. Table 4.2 gives sample calculation of the parameters for finding out the theoretical T value for the decompose step using eqn. 4.4.
Number of iterations = 3, C = 1, $G_{prac} = 13$ (= Number of SMXs), $I_{prac} = 16$ (refer to section 3.3.2)

Table 4.2: Theoretical parameters for S-M-T mode's Decompose step on Kepler K20

Iter. no.	S	G (= 7^{i-1})		I(= 7)		G_{steps}		I_{steps}	
		G_A	G_B	I_A	I_B	G_{stepsA}	G_{stepsB}	I_{stepsA}	I_{stepsB}
1	512 × 512	1	1	7	7	1	1	1	1
1	256 × 256	7	7	7	7	1	1	1	1
1	128 × 128	49	49	7	7	4	4	1	1

$$\begin{aligned} T =& 12 \times L_g + 12 \times L_i + (512 \times 512 \times (1 \times 1 + 1 \times 1) + \\ & 256 \times 256 \times (1 \times 1 + 1 \times 1) + 128 \times 128 \times (4 \times 1 + 4 \times 1)) \times B. \end{aligned} \tag{4.4}$$

4.3.2 Single-kernel, Heterogeneous Group, Homogeneous Items (S-T-M) mode

The S-T-M mode has an increasing number of work-items and a constant number of work-groups. Table 4.3 gives sample calculation of the parameters for finding out the theoretical T value for the decompose step using eqn. 4.5.

Number of iterations = 3, C = 1, $G_{prac} = 13$ (= Number of SMXs), $I_{prac} = 192$ (refer to section 3.3.2)

Table 4.3: Theoretical parameters for S-T-M mode's Decompose step on Kepler K20

Iter. no.	S	G (= 7)		I(= 7^{i-1})		G_{steps}		I_{steps}	
		G_A	G_B	I_A	I_B	G_{stepsA}	G_{stepsB}	I_{stepsA}	I_{stepsB}
1	512 × 512	7	7	1	1	1	1	1	1
1	256 × 256	7	7	7	7	1	1	1	1
1	128 × 128	7	7	49	49	4	4	1	1

$$\begin{aligned} T =& 6 \times L_g + 6 \times L_i + (512 \times 512 \times (1 \times 1 + 1 \times 1) \\ & + 256 \times 256 \times (1 \times 1 + 1 \times 1) + 128 \times 128 \times (1 \times 1 + 1 \times 1)) \times B. \end{aligned} \tag{4.5}$$

4.3.3 Single-kernel, Homogeneous Group, Homogeneous Items (S-M-M) mode

The S-M-M mode has an increasing number of work-groups and a variable number of work-items. Table 4.4 gives sample calculation of the parameters for finding out the theoretical T value for the decompose step using eqn. 4.6.

Number of iterations = 3, C = 7, $G_{prac} = 13$ (= Number of SMXs), $I_{prac} \leq 192$ (refer to section 3.3.2), I_{steps} = $\left\lceil \frac{I_{launch}}{I_{prac}} \right\rceil$, $S(= \frac{I}{I_{launch}})$.

$$\begin{aligned} T =& 12 \times L_g + 72 \times L_i + (256 \times (1 \times 6 + 1 \times 6) + \\ & 64 \times (1 \times 6 + 1 \times 6) + 16 \times (4 \times 6 + 4 \times 6)) \times 7(ops) \times B. \end{aligned} \tag{4.6}$$

Table 4.4: Theoretical parameters for S-M-M mode's Decompose step on Kepler K20

Iter. No.	G_A	G_B	G_{lau} $_{nch}$	I_A	I_B	I_{lau} $_{nch}$	G_{ste} $_{psA}$	G_{ste} $_{psB}$	I_{ste} $_{psA}$	I_{ste} $_{psB}$	S
1	1	1	1	512 × 512	512 × 512	32 × 32	1	1	6	6	256
2	7	7	7	256 × 256	256 × 256	32 × 32	1	1	6	6	64
3	49	49	49	128 × 128	128 × 128	32 × 32	4	4	6	6	16

4.4 Summary

This chapter presented a theoretical cost model for calculating the runtime for each of the modes in K-G-I model. It has also tabularized the various parameters in the execution time formula for each iteration in the decompose step of the algorithm as executed on the Kepler K20 machine. The upcoming chapter gives the techniques to exploit maximum parallelism based on the knowledge of the device architecture and the runtime system. It also presents how to ease the parallelism specification in the program using an API extension.

Chapter 5

API Extension

5.1 Extracting maximum parallelism by leveraging on the knowledge of device architecture

Changing trends in modern GPU architectures expose further opportunities for parallelism. In this section, experiments have been done to exploit the capability of NVIDIA's new Kepler K20 to execute multiple concurrent kernels. This feature allows one to move from the contrived task parallelism mode (in reality, a relaxed data-parallel mode) to actual task parallelism where multiple kernels can be launched concurrently.

5.1.1 Multiple-kernel, Homogeneous Group, Homogeneous Items (M-M-M) mode

This mode of execution is exclusively supported only by the newest NVIDIA devices with compute capability of 3.5 and above. These devices allow multiple kernels to execute at the same time on the device.

In this execution mode, each of the seven elementary operations in the decompose operation is specified as a separate kernel and these kernels have pure data-parallel elements; they all follow the same execution path. These are all light-weight kernels having work-items that do a single operation. In the example above, there are 7 kernels, each performing single operations (add, multiply or copy). In each of these kernels, there are 49 work-groups. Each work-group has 32 × 32 work-items. Each work-item does a single operation on 4×4 data as shown in Figure.5.1 and Figure.5.2.

In the current state-of-the-art Kepler K20 GPU from NVIDIA, this mode of execution can

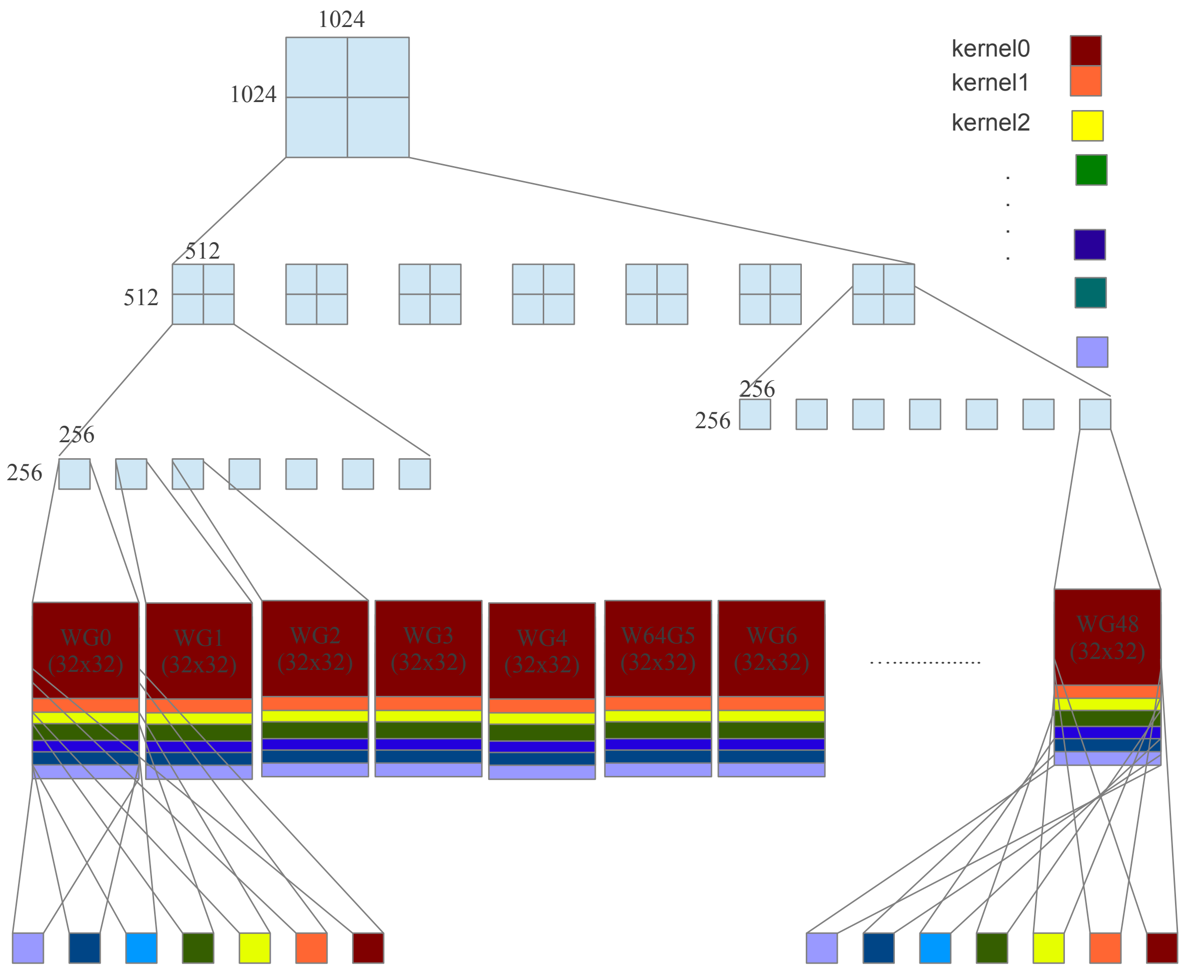

Figure 5.1: M-M-M mode of execution on Kepler K20

be supported by utilizing the HyperQ technology. HyperQ is a feature that enables multiple CPU cores to simultaneously utilize multiple CUDA cores on a single Kepler GPU. Kepler has 32 hardware queues from which the GPU can simultaneously take jobs and execute. It has 13 SMXs (SMX is the next generation SM that supports a number of new features including multiple kernel executions at the same time), so it can simultaneously execute 13 different jobs [14]. This cannot be directly implemented using the existing OpenCL APIs. In this work

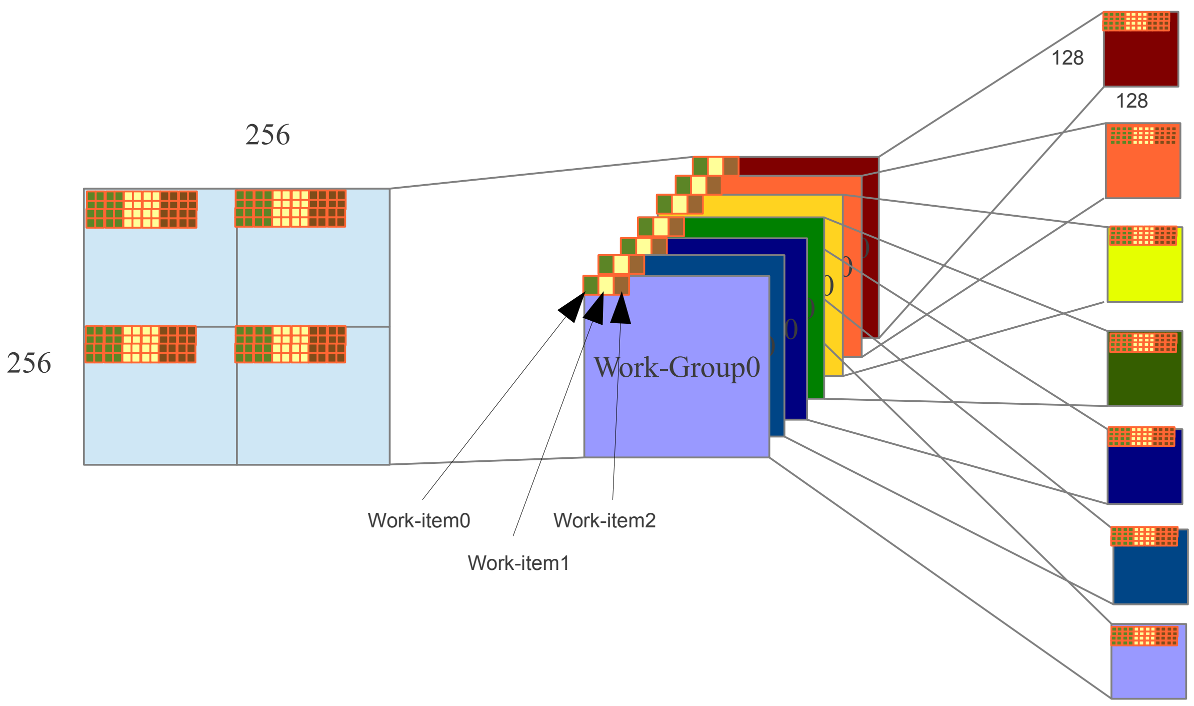

Figure 5.2: Work distribution for M-M-M mode on Kepler K20

this has been achieved by creating multiple command queues (for a single device), each of which was enqueued with a different kernel. These kernel executions overlap. But there needs to be a synchronization mechanism in every iteration to ensure correctness. Since OpenCL's native execution modes are i. Task Parallel and ii. Data-parallel, this hybrid mode of execution was not explicitly supported by the OpenCL API.

5.1.2 The theoretical execution time calculated based on the analytical model for the algorithm

For the M-M-M programming mode, the number of work-groups that are run in parallel are 7 times higher than the S-M-M model. In this model the kernels are made as light-weight kernels that does a single operation on the input data. The theoretical parameters for finding out the execution time are tabulated in table 5.1 and the theoretical execution time is given by eqn. 5.1 Number of iterations = 3, C = 1, $G_{prac} = 13$ (= Number of SMXs), $I_{prac} \leq 192$ (refer to section 3.3.2) , I_{steps} = $\left\lceil \frac{I_{launch}}{I_{prac}} \right\rceil$, $S = \frac{I}{I_{launch}}$

Table 5.1: Theoretical parameters for M-M-M mode's Decompose step on Kepler K20

Iter. No.	G_A	G_B	G_{launch}	I_A	I_B	I_{launch}	G_{stepsA}	G_{stepsB}	I_{stepsA}	I_{stepsB}	S
1	1×7	1×7	1×7	512×512	512×512	32×32	1	1	6	6	256
2	7×7	7×7	7×7	256×256	256×256	32×32	4	4	6	6	64
3	49×7	49×7	49×7	128×128	128×128	32×32	27	27	6	6	16

$$\begin{aligned} T =& 64 \times L_g + 384 \times L_i + (256 \times (1 \times 6 + 1 \times 6) + \\ & 64 \times (4 \times 6 + 4 \times 6) + 16 \times (27 \times 6 + 27 \times 6)) \times B. \end{aligned} \tag{5.1}$$

5.1.3 Comparison of the Theoretical and Experimental execution times

The graphs of theoretical and experimental execution times of the decompose step are shown in Figure. 5.3 and Figure. 5.4 respectively. The theoretical model doesn't take into account some of the system parameters like the L1 and L2 cache sizes, Memory bandwidth, etc. But still the trend is the same as compared to the experimental results. For a programmer to efficiently code an algorithm, one needs to find the behavior of the run-time system and a number of parametric values of the architecture. Also an analytical model cannot accurately predict the system behavior. So the burden is on the programmer to specify the parallelism for efficient implementation.

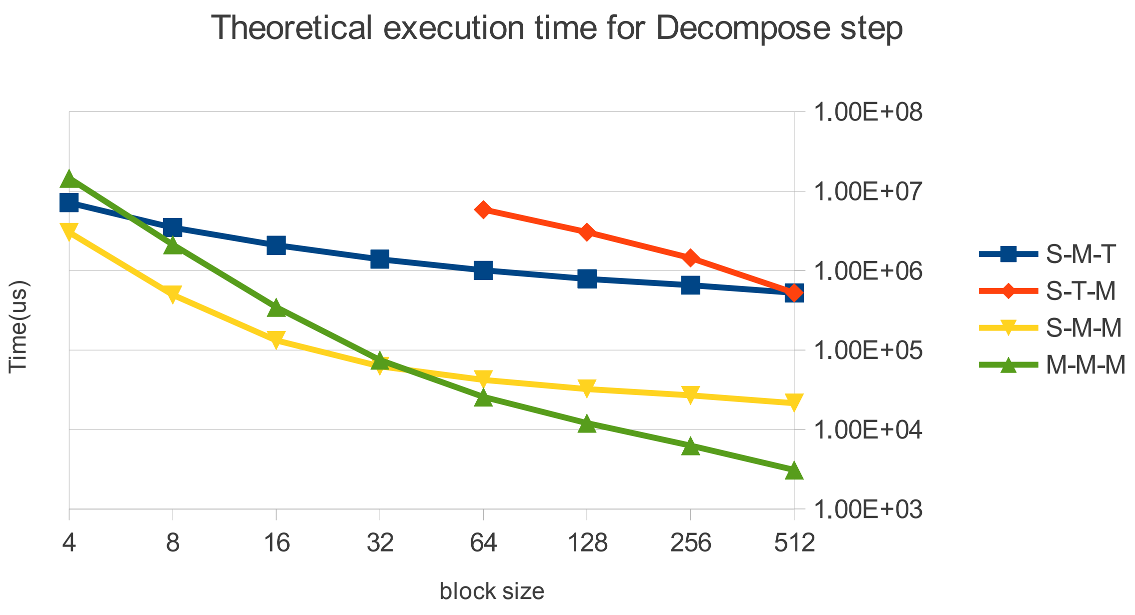

Figure 5.3: Theoretical results

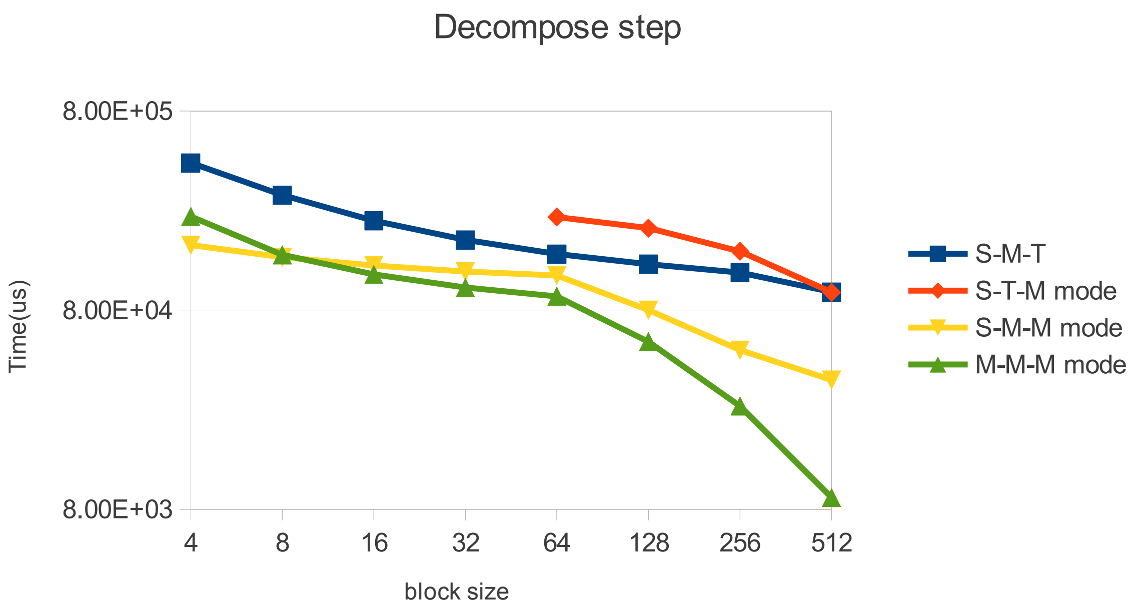

Figure 5.4: Practical results

5.2 API extension to support hybrid parallelism

OpenCL specifications allow extensions to be added to its core API subset. The kernel execution call API in OpenCL and its parameters are explained in the following subsection. An API extension in OpenCL has been proposed then that will support the hybrid mode parallel execution of kernels and makes it easy for the programmer to express parallelism.

5.2.1 OpenCL's current kernel execution API call

The kernels in OpenCL are executed on the device using the command clEnqueueNDRangeKernel which specifies the index space in which the kernel function should be executed. This API call and its parameters are explained below. $clEnqueueNDRangeKernel(command\_queue,$
$kernel,$
$work\_dim,$
$*\,global\_work\_offset,$
$*\,global\_work\_size,$
$*\,local\_work\_size,$
$num\_events\_in\_wait\_list,$
$*\,event\_wait\_list,$
$*\,event)$

command_queue	- The command queue that executes this NDRange call,
Kernel	- The kernel function to be executed,
Work_dim	- global_work_size & local_work_size dimensions (1 or 2 or 3),
global_work_offset	work-item_id offset in each of the dimension,
global_work_size	Total No. of work-items in each of the dimension,
local_work_size	No. of work-items in a work-group in each of the dimension,
num_events_in_wait_list	No. of events this NDRange call waits on,
event_wait_list	Ref. to the events this enqueue call waits on.

This API call allows users to specify only the size of work-groups and total number of work-items. All these work-items are considered to be data-parallel by most of the devices.

There is no explicit way to tell the task-parallelism using this API call.

5.2.2 Proposed kernel execution API

The current OpenCL API can support either data parallelism or task parallelism but it cannot support a hybrid mode where multiple data-parallel work-groups with different execution paths can be launched. In order to implement a hybrid mode, one should be able to group data-parallel work-items and work-groups into different task-parallel work-clusters that can be concurrently launched. The proposed API call clEnqueueNDRangeHyperKernel introduces a new parameter universal_work_size that gives the total number of work-items among all the work-clusters. The number of task-parallel work-clusters are given by universal_work_size/sizeof(global_work_size_array). In this proposed mode, all work-groups and work-items have the same sizes. A future enhancement would take into consideration different-sized work-groups and work-items.

$clEnqueueNDRangeHyperKernel(command\_queue,$

$work\_dim,$

$*\,universal\_work\_size,$

$*\,global\_work\_offset,$

$*\,global\_work\_size,$

$*\,local\_work\_size,$

$num\_events\_in\_wait\_list,$

$*\,event\_wait\_list,$

$*\,event)$

If this extension is supported by OpenCL, devices having MIMD capabilities can execute hybrid data-and-task-parallel applications efficiently. The programmer doesn't have to worry about the behavior of the runtime system and architectural parameters while coding for efficient parallelization. The runtime system can then take care of the mapping of the task-parallel clusters containing data-parallel groups.

5.3 Summary

This chapter described how the features of the device were utilized for its maximum to achieve maximum parallelism by programmatically choosing the decomposition of the work-items. It compared the graphs of the theoretical running time with the experimental running time for each of the modes explained in the previous chapter as well as for the hybrid execution mode that exploited maximum parallelism by utilizing the device features. It also introduced the API extension to make it easy for the programmer to specify hybrid parallelism.

Chapter 6

Conclusion and Future Work

In this thesis, efficient task and data parallel implementations of the iterative Strassen's algorithm have been proposed. It presented results of the execution of all these different programming models on NVIDIA Fermi and Kepler K20 GPUs [13] [14]; It devised cost models to derive execution times for the decompose step in the iterative algorithm for each of these implementations; It has also presented results of pilot programs that are run to arrive at the actual device parameters required for formulating the cost models and compared them with the theoretical calculations. The thesis has proposed OpenCL API extension to exploit multiple level parallelism and to leverage features in modern GPUs to achieve true task parallelism by launching multiple kernels. The proposed hybrid model of parallelism allows the programmer to choose multiple modes and degrees of parallelism thereby facilitating efficient implementations of task-and-data-parallel algorithms on inherently data parallel architectures such as GPUs.

There is a scope to design a run-time system that supports this API extension and efficiently maps the work-items to the resources available on the device. In future the API extension can also be improved to support launching of work-clusters that have data-parallel work-groups of different size.

Bibliography

[1] AMD Corporation. *AMD Radeon HD5870 GPU Architecture Details*, October 2012.

[2] Saniya Ben Hassen, Henri E. Bal, and Ceriel J. H. Jacobs. A task- and data-parallel programming language based on shared objects. *ACM Trans. Program. Lang. Syst.*, 20(6):1131–1170, November 1998.

[3] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.

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

[5] Maneesh Dhagat, Rajive Bagrodia, and Mani Chandy. Integrating task and data parallelism in uc, 1994.

[6] VinothKrishnan Elangovan, Rosa.M. Badia, and EduardAyguade Parra. Ompss-opencl programming model for heterogeneous systems. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, volume 7760 of *Lecture Notes in Computer Science*, pages 96–111. Springer Berlin Heidelberg, 2013.

[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] Brian Grayson, Ajay Pankaj Shah, and Robert A. Van De Geijn. A high performance parallel strassen implementation. In *Parallel Processing Letters, Vol 6*, pages 3–12, 1995.

[9] Mats Johansson and Academy Winter. General purpose computing on graphics processing units using opencl, 2010.

[10] Prabhanjan Kambadur, Anshul Gupta, Amol Ghoting, Haim Avron, and Andrew Lumsdaine. Pfunc: modern task parallelism for modern high performance computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 43:1–43:11, New York, NY, USA, 2009. ACM.

[11] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[12] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[13] NVIDIA Corporation. *NVIDIA Fermi Architecture Specification Sheet*, July 2010.

[14] NVIDIA Corporation. *NVIDIA Kepler K20 Architecture White Paper*, October 2012.

[15] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[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 *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 13–22, New York, NY, USA, 1993. ACM.

[18] Jaspal Subhlok and Bwolen Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '97, pages 1–12, New York, NY, USA, 1997. ACM.

[19] Rafael Alejandro Vejarano, Phuong Thi Yen, and Jeong-Gun Lee. Parallel acceleration on manycore systems and its performance analysis: Opencl case study. 2013.

[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.

[21] Chao-Tung Yang, Chih-Lin Huang, Cheng-Fang Lin, and Tzu-Chieh Chang. Hybrid parallel programming on gpu clusters. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pages 142–147, 2010.