Optimization of CUDA Programs using Streams

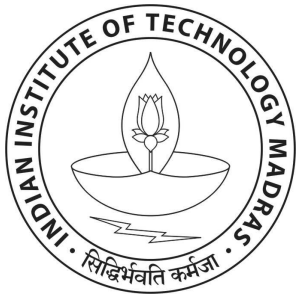
A Project Report

submitted by

MARYADA SIDDARTHA REDDY

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2014

THESIS CERTIFICATE

This is to certify that the thesis titled **Optimization of CUDA programs using Streams**, submitted by **Maryada Siddartha Reddy**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr.Shankar Balachandran
Project Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: May 2014

ACKNOWLEDGEMENTS

I am thankful to **Dr.Shankar Balachandran** for his guidance throughout the project. His valuable insights helped me a lot for the progress of my project. I also thank my friends who helped me throughout the project by discussing various aspects necessary for the project. I thank RISE Lab, IIT Madras for providing the infrastructure required for my project. I thank my parents for giving me the motivation and support all the way through.

ABSTRACT

KEYWORDS: streams;task overlap;task scheduling;CUDA;GPU;parallelism

In this project a mechanism to schedule tasks using streams is developed by taking advantage of advances in latest CUDA GPUs for simultaneous execution of computation , data transfer from CPU to GPU and GPU to CPU.Computations on GPU in general referred as kernels and GPU is referred as device and CPU as host.To enable simulteneous execution we divided application into smaller sub-tasks and allowed kernel and host to device and device to host data transmission of different sub tasks to overlap there by effectively reducing the run time of the program.To decide on optimal sub task size ,we have developed two models ,Compute bound model for computing intensive applications and Data bound model for data intensive applications. On the basis of these models we implemented scheduling algorithms to understand performance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

H2D	Host to Device
D2H	Device To Host
GPU	Graphics Processing Unit
CPU	Central Processing Unit

Host	CPU
Device	GPU
Kernel	Function that runs on device

CHAPTER 1

INTRODUCTION

Stream processing is a relatively new computing paradigm that enables parallel processing of a defined series of operations on multiple computing resources with extreme levels of efficiency and performance. Streaming has evolved with graphics and Stream processing has been widely used in applications such as audio and video processing[Owens *et al.* (2002)] .In these applications work is decomposed into several stages of computation and the data of input and output is organized as chunks of independent data to go through several stages of computation like a pipeline. Stream processing can exploit the inherent parallelism of the pipeline while the different stream elements also can be processed simultaneously to achieve data parallelism and task parallelism.

Graphics processing unit (GPU) is one of the most successful stream architectures in recent years which is originally designed for acceleration of graphics applications. Now, it is widely used as General-purpose computing on graphics processing units (GPGPU) to accelerate many scientific applications It is becoming increasingly common to use a Graphics Processing Unit(GPU) unit as a modified form of stream processor. This concept turns the massive computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power, as opposed to being hard wired solely to do graphical operations. Now, it is widely used as General-purpose graphics processing unit (GPGPU) to accelerate many scientific applications.There are many parallel programming languages that help programmers to write parallel applications with GPUs such as OpenCL[KhronosGroup (2008)] , Brook[Ian Buck (2004)], CUDA[NVIDIA (2007)] .Among the parallel programming languages CUDA is one of the most popular computing platform for GPU. It provides a C-like language for programmers to implement their GPU programs without any knowledge of graphics processing.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing

units (GPUs) that they produce. CUDA gives program developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

CUDA programs consists of host code (running on CPU) and device code (running on GPU).Host code calls the device code which implements the GPU computation i.e kernel operation . During computation on a GPU input data has to reside on GPU and output data will be generated in GPU memory .Since GPU (device) memory is separated from CPU(host) memory we need to explicitly transfer data to and receive data from GPU.Modern GPUs are enabled with simultaneous bi-directional data transfer and also parallel execution of kernel along with them.In order to achieve overlap between data transfers and kernel executions requires the use of CUDA streams.

A stream in CUDA is a sequence of operations such are kernel computation ,H2D,D2H that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently.

CUDA provides two copy engines for bi-directional data transfer since version 4.0 while enables to execute data send, receive and kernel simultaneously. Therefore, we can use three streams respectively for data send, receive and kernel to implement stream applications and launch kernel, data send and receive at the same time

1.1 Literature Review

Current methods follow batch scheduling[Hou *et al.* (2008),Nakagawa *et al.* (2010)] to take advantage of advances in modern GPUs thus ensuring data transfer overlap with kernel computation.There were also works on dividing individual tasks into sub tasks but they didn't consider several overhead parameters.

1.2 Contributions

In this work , we propose a task partition model taking various kernel launch ,memory copy and synchronization overhead parameters into consideration and using streams we schedule the sub-tasks depending on the type of application to improve performance of standalone application by overlapping data send send , data receive and kernel part of different sub tasks. With two copy engines, we support simultaneous data send, data receive and kernel execution. We classify applications into data independent and data dependent applications. Data independent applications mean that there are no shared input data elements between output data elements such as matrix addition.Data dependent applications mean that there are shared input data elements between output data elements such as matrix multiplication. For the two types, we provide two kinds of stream scheduling algorithms: static stream scheduling and application customized stream scheduling.We also design two application partition models: compute bound model and data transfer bound model to calculate the optimal subtask numbers. With the optimal subtask numbers from the task partition model model, the scheduling algorithms can schedule the bi-directional data transfer and kernel execution of each subtask with multiple streams and overlap them to improve the performance. We assume that the time cost of kernel execution part in each subtask can be predicted like in Luo and Suda (2011) and Hong and Kim (2009) programmers can calculate the data size to be sent for each subtask within a reasonable complexity.

1.3 Organization of Thesis

This thesis is organized into five chapters.

Chapter 1 Introduces the emergence of streaming applications and GPU hardware and summarizes the contributions of this thesis.

Chapter 2 explains the necessary background on CUDA required to gain a clear understanding of the thesis.

Chapter 3 Proposes the scheduling algorithms for the applications classfied ondifferent features.

Chapter 4 Presents the various overheads involved in task partitioning and developes a model for finding the optimal sub task size of the application.

Chapter 5 describes how to set up experiments to verify the proposed method and presents the results.

CHAPTER 2

CUDA

CUDA (Compute Unified Device Architecture) is the name of NVIDIA's parallel computing architecture in our GPUs. NVIDIA provides a toolkit for programming the CUDA architecture that includes the compiler, debugger, profiler, libraries and other information developers need to deliver products that use the CUDA architecture. The CUDA architecture also supports standard languages such as C and Fortran, and APIs for GPU Computing, such as OpenCL and DirectCompute. CUDA gives program developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

Using CUDA, the GPUs can be used for general purpose processing (i.e., not exclusively graphics); this approach is known as GPGPU. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.

NVIDIA has defined a general computing instruction set (PTX) and small set of C language extensions that allow developers to take advantage of the massively parallel processing capabilities in our GPUs. Other groups provide support for Fortran.Java, Python, .NET, and other languages on the cuda atchitecture
The term CUDA C to describe the language and the small set of extensions developers use to specify which functions will be executed on the GPU, how GPU memory will be used, and how the parallel processing capabilities of the GPU will be used by an application.

2.1 Streaming Multiprocessors(SMs)

The Streaming Multiprocessors (SMs) are the part of the GPU that runs our CUDA kernels.
Each SM contains:

- Thousands of registers that can be partitioned among threads of execution.
- Several caches
  1. shared memory for fast data interchange between threads,
  2. constant cache for fast broadcast of reads from constant memory,
  3. texture cache to aggregate bandwidth from texture memory,
  4. L1 cache to reduce latency to local or global memory.
- Warp schedulers that can quickly switch contexts between threads and issue instructions to warps that are ready to execute.
- Execution cores for integer and floating-point operations

The SMs are general-purpose processors, but they are designed very differently than the execution cores in CPUs: they target much lower clock rates; they support instruction-level parallelism, but not branch prediction or speculative execution; and they have less cache, if they have any cache at all. For suitable workloads, the sheer computing horsepower in a GPU more than makes up for these disadvantages.
Following diagram 2.1 clearly demonstrates the processing flow of computation in a CUDA GPU(GeForce 8800).

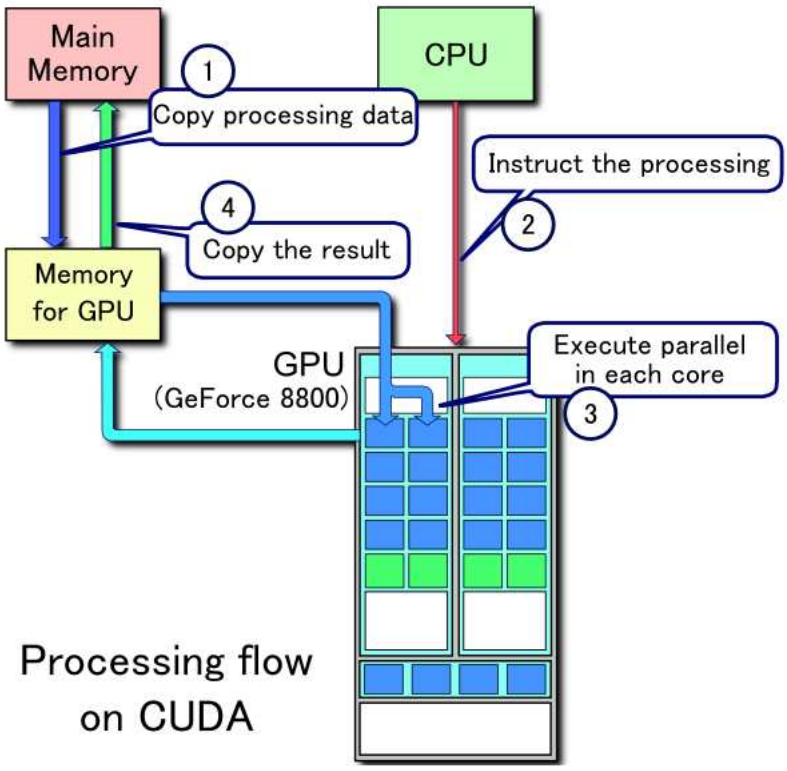


Figure 2.1: CUDA Processing Flow

1. Copy data from main memory to GPU memory

2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU memory to main memory

2.2 Programming model

2.2.1 Cuda Kernels and Threads

- Parallel portions of the applications are executed on device as kernels.Many threads are executed in each kernel.
- Difference between CUDA threads and CPU threads is that CUDA threads are extremely light weight.They have very little creation overhead and fast switching.
- A CUDA kernel is executed by an array of threads.All threads run the same code.Each thread has an ID that it uses to compute memory addresses and make control decisions
- Threads may need to cooperate on memory accesses.This may be helpful in bandwidth reduction and avoiding redundant computation.
- But cooperation between a monolithic array of threads does not scale well,cooperation within smaller batches of threads is scalable.

2.2.2 Thread Batching

kernel launches a grid of thread blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by 2.2. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

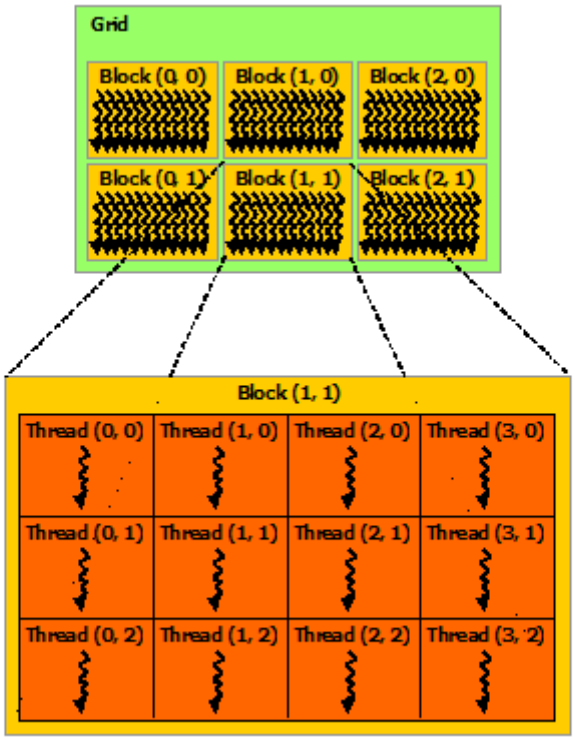


Figure 2.2: Grid of thread blocks

Threads with in a block cooperate via shared memory.Threads in different blocks cannot cooperate.

2.3 Memory Model

There are several different kind of memory available on a GPU.CUDA threads may access data from multiple memory spaces during their execution as illustrated by 2.3. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

Registers

- Per thread memory
- Data life time = thread life time

Local memory

- Per thread off-chip memory(physically on DRAM)
- Data life time = thread life time

Shared memory

- Per thread block on-chip memory
- Data lifetime = block lifetime

Global memory

- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation

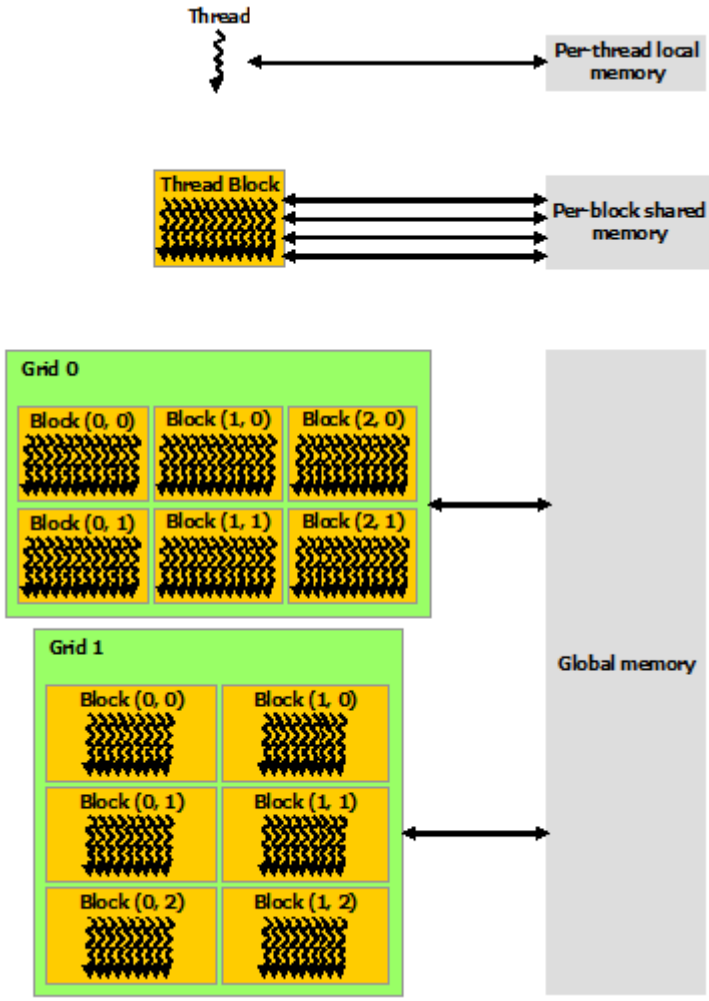


Figure 2.3: Memory Hierarchy

2.4 Execution Model

The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. Support for dynamic parallelism in CUDA extends the ability to configure, launch, and synchronize upon new grids to threads that are running on the device.

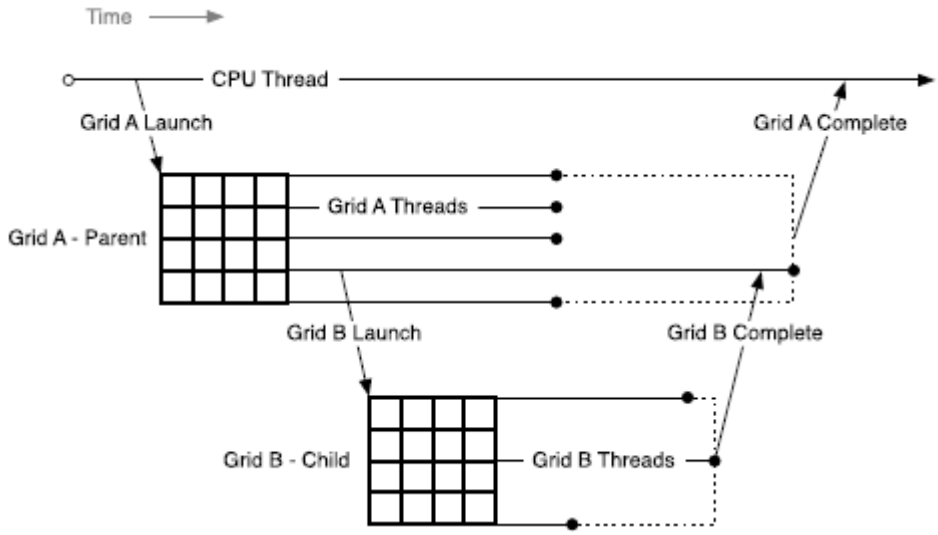


Figure 2.4: Cuda Kernel Launch

Kernels are launched in grids.A thread block execute on one multiprocessor , does not migrate.Several thread blocks reside on one multiprocessor .Number of concurrent blocks is limited by multiprocessor resources.Registers is divided among all resident threads.Shared memory is partitioned among all resident thread blocks.

2.5 CUDA Asynchronous Concurrent Execution

2.5.1 Concurrent Execution between Host and Device:

CUDA Run-time supports asynchronous function calls which facilitate concurrent execution between host and device, some function calls. In execution of these function calls, the control is returned to the host thread before the device has completed the requested task. Some of these are:

- Kernel launches
- Memory copies between two addresses to the same device memory
- Memory copies from host to device of a memory block of 64 KB or less
- Memory copies performed by functions that are suffixed with Async
- Memory set function calls.

2.5.2 Overlap of Data Transfer and Kernel Execution

Some devices of compute capability 1.1 and higher can perform copies between page-locked host memory and device memory concurrently with kernel execution. Applications may query this capability by checking the asyncEngineCount device property, which is greater than zero for devices that support it. For devices of compute capability 1.x, this capability is only supported for memory copies that do not involve CUDA arrays or 2D arrays allocated through cudaMallocPitch().

2.5.3 Concurrent Kernel Execution

Some devices of compute capability 2.x can execute multiple kernels concurrently. Applications may query this capability by checking the concurrent Kernels device property, which is equal to 1 for devices that support it. The maximum number of kernel launches that a device can execute concurrently is sixteen. A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context. Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels.

2.5.4 Concurrent Data Transfers

Some devices of compute capability 2.x can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page-locked host memory. Applications may query this capability by checking the asyncEngineCount device property, which is equal to 2 for devices that support it.

2.6 CUDA Streams

Applications manage concurrency through streams. A stream is a sequence of commands (possibly issued by different host threads) that execute in order.Several operations can be included into a stream and the order in which operations are added to the stream specifies the order in which they will be executed. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (e.g., inter-kernel communication is undefined).Some of Stream features are:

- Creation and Destruction
- Default Stream
- Explicit Synchronization
- Implicit Synchronization
- Overlapping Behavior
- Default Stream

2.7 Using a Single Stream Multiple CUDA Streams

Application employ single or multiple CUDA Streams. When we use single stream, at the beginning of application, the computations are divided into chunks and each chunked computation and the overlap of memory copies with kernel execution.
On multiple streams, different streams will perform CUDA operations as per application requirements. For example stream 1 will do copy input buffers to the GPU, Stream 0 will execute its kernel while stream 0 copies its results to the host.

CHAPTER 3

Scheduling Mechanism

Our aim is to partition the CUDA application into subtasks and overlap the data send, data receive and kernel execution part of these subtasks. Therefore, the main parts of mechanism are application partition models and sub-task scheduling schemes with multiple CUDA streams

3.1 Classification of Applications

To design task partition model and algorithm we need to classify the applications. We classified applications primarily on the basic of

- Relation between input and output.
- Computation-to-Communication ratio.

3.1.1 Relation between input and output

From the relationship between input data and output data aspect, we classify the applications into:

- data independent applications
- data dependent applications.

With **data independent applications**, the sets of data input required to compute different outputs are disjoint. For example, matrix addition is an typical data independent applications. For C = A + B , we only need A[i][j] and B[i][j] to calculate C[i][j] while data A[i][j] and B[i][j] can only be used to calculate C[i][j].

For **Data Dependent Applications** there is share set of input elements for different output elements. For example, matrix multiplication is a data dependent application. For C = A B , we need ith row elelments in matrix A and jth row elements in matrix B to calculate C[i][j] .Each element in ith row of A and jth column of B used to calculate C[i][j] are also used to calculate other elements in matrix

3.1.2 Computation to Communication ratio

From the computation-to-communication ratio , we classify applications into

- Data Bound Applications
- Compute Bound Applications

Data Bound applications are the applications where bi-directional data transfer cost is greater than the computation cost

Compute Bound Applications have longer kernel execution time cost than the bi-directional data transfer time cost

To measure the bi- directional data transfer time cost, we launch the data send and the data receive simultaneously with two streams and synchronize the two streams. Then we can obtain the time cost of the bi-directional data transfer by calculating the time taken for transfer of data

The feature of computation-to-communication ratio decides the optimal subtask partition while the feature of relationship between input and output data mainly affects the stream scheduling.

3.2 Scheduling Scheme

Scheduling scheme of subtasks can be broadly divided into two types for two kinds of applications namely Static Scheduling for data independent applications and Application customized stream scheduling for data dependent application

3.2.1 Static Scheduling

Static stream scheduling will calculate a scheduling plan first as we know all the parameters of the runtime of program and then schedule the streams following the plan. The advantage of static stream scheduling is that it is simple and also efficient to data independent applications. For the data independent applications, we design two static stream scheduling algorithms basing on computation- to-communication ratio:

- Data bound static scheduling algorithm
- Compute bound static scheduling algorithm

Data bound static Scheduling

When the time cost of bi-directional data transfer is longer than the time cost of kernel execution, the time cost of each step is decided by the data transfer part. The lower bound of the time cost of the application with stream scheduling will be equal to the time cost of the bi-directional data transfer part. Therefore, it is better to hide the kernel execution part by overlapping the data transfer and kernel execution. As shown in 3.1, we use three streams: send, kernel, receive. Send stream is used for data send part of subtask, kernel stream is used for kernel execution part of subtask and receive stream is used for data receive part of subtask. As each subtasks in the CUDA stream are executed in order and there are data dependency between the data send, kernel execution and data receive from the same subtask, we launch the data send, kernel execution and data receive from different subtasks to the corresponding streams. Then we synchronize the three streams before the next step. Due to the long time cost of data transfer parts in each step, we set the subtask in each step to the same size. This can make the algorithm simple and efficient. The optimal subtask size will be decided by task partition model which will be introduced in the next chapter.

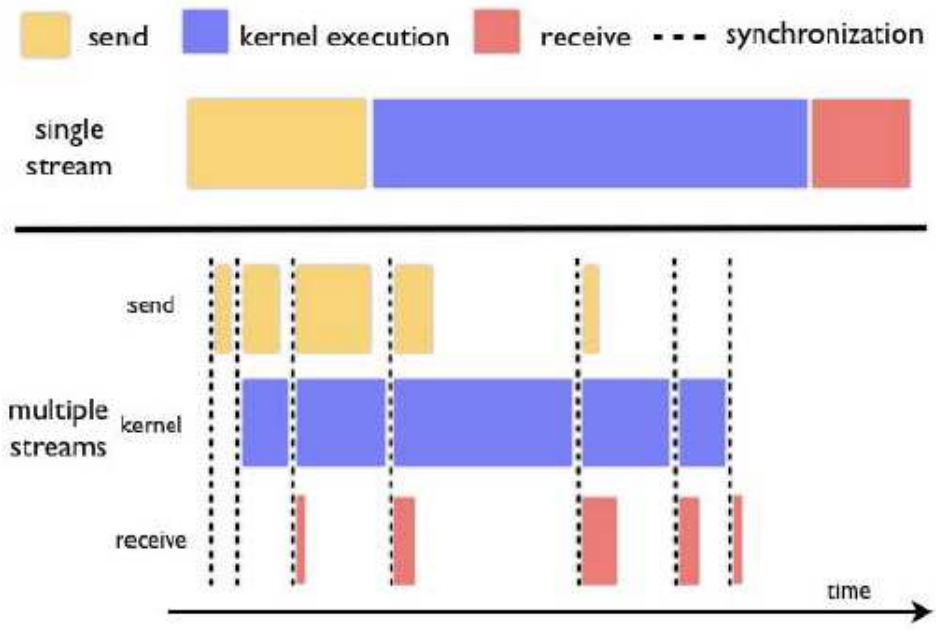


Figure 3.1: Data Bound Stream Scheduling

Compute bound static Scheduling

When the time cost of kernel execution is longer than the time cost of bi-directional data transfer, the time cost of each step is decided by the kernel execution part. In this case, the total time cost of the application is equal to the sum of the send time cost in the first step, the total kernel execution time cost and the receive time cost in the last step. Therefore, it is better to make the first send time cost and the last receive time cost as short as possible. As shown in 3.2, to make the first send as short as possible, we set the first sub-task to a small size which only sends a small size of data for kernel execution. Then we exponentially increase the sub-task size in the first half of the application. To make the last receive as short as possible, we exponentially decrease the sub-task size in the latter half of the application. Here the size of the first sub-task size and the last sub-task size will be decided by the task partition model as well.

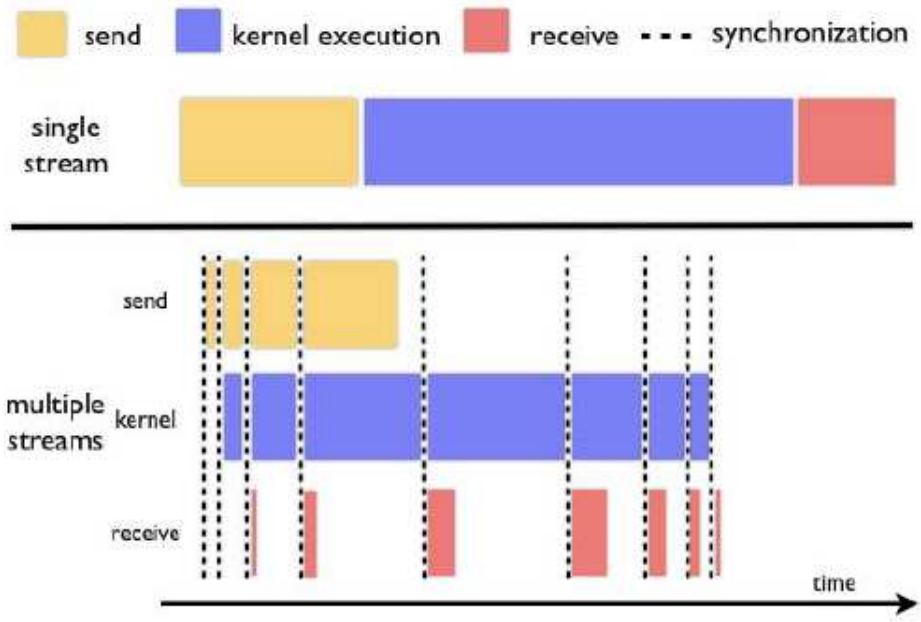


Figure 3.2: Compute Bound Stream Scheduling

3.2.2 Application customized stream scheduling

A major chunk of applications we deal with are data dependent applications.This section is aimed at giving pointers regarding the way an application can be customized depending on its properties to make it efficient.The example we are dealing with is Matrix-Multiplication which is a very common application in the field of computing

In the matrix multiplication of C = A x B, naive GPU implementation will transfer data of two matrices A and B after the allocation of memory for matrices A,B,C in GPU and then compute matrix C , after the computation we can send back the matrix C on to CPU.Here we are losing an opportunity to do simultaneous data transfer and kernel execution. If we take a closer look at properties of a matrix multiplication, we can divide each matrix into 4 sub matrices then do a partial matrix multiplication between the sub matrices and add them later.This division of computation across sub matrices enables us to do computation between two sub matrices while coping data of other sub matrices

So in this case we have divided the matrix into sub-matrices , we can enable this by memory reordering of matrix from row major memory order to Block major memory order i.e memory organised as blocks of sub-matrices in memory.

So,depending on the application , we need to analyse its computation properties(i.e data dependencies across elements) and decide on the way to divide application into sub tasks , then schedule it across streams.

CHAPTER 4

Task Partition Model

The task partitioning can affect the performance greatly. Small sub-task partitioning method does not necessarily mean high performance. In the contrary, it might introduce significant overhead which leads to poor performance. We will analyze the possible factors which decide overhead. Then we provide two task partition models to partition the application which can minimize the total time cost of the application considering different overheads.

We have done our experiments on Tesla K20C GPU ,its configuration is mentioned in table 4

4.1 Overhead Factors

In our experiments we found mainly four sources of overheads in various scales depending on the state of computation,namely

- Synchronization overhead
- Bi-directional data transfer bandwidth
- kernel launch overhead
- Memory transfer overhead

4.1.1 Synchronization Overhead

The stream synchronization overhead is the overhead incurred by synchronization across streams . As sub-task size becomes smaller we need more synchronizations which means more synchronization overhead.This at some point outweighs advantage we gain by task overlapping.So need to choose optimal sub-task size. We have done experiments on synchronization overhead when synchronizing different number of streams which rums a dummy kernel.

Name	Tesla K20C
CUDA Run Time Version/Major.Minor version	5.5/3.5
Total amount of global memory	4096 MB
L2 Cache Size	1310720 bytes
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024,1024,1024)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)
Multiprocessors	13
CUDA Cores/MP	192
CUDA Cores(Cores/MP* Cores)	2496
Clock Speed	706MHz
Run time limit on kernels	No
Integrated GPU sharing Host Memory	No
Support host page-locked memory mapping	Yes
Concurrent copy and kernel execution	Yes with 2 copy engine(s)

Table 4.1: Specifications of Tesla K20C GPU

We have taken 32768(2 pow(15)) samples of overhead and calculated mean of them.We have calculated overhead while synchronizing 1-6 kernels.

$$T \quad = n \times \ (t_{sync} + t_{koh} + t_{dk}\ ) \tag{4.1}$$

$$t_{dk} << t_{koh} \tag{4.2}$$

$$t_{sync} = \ (T - n \times t_{koh}\ )/n \tag{4.3}$$

T :total time cost

t_{dk} : time taken by dummy kernel

t_{koh} : time taken for kernel launch

t_{sync} : synchronization overhead

Below is the graph shows the variation of synchronization overhead across different no. of streams.

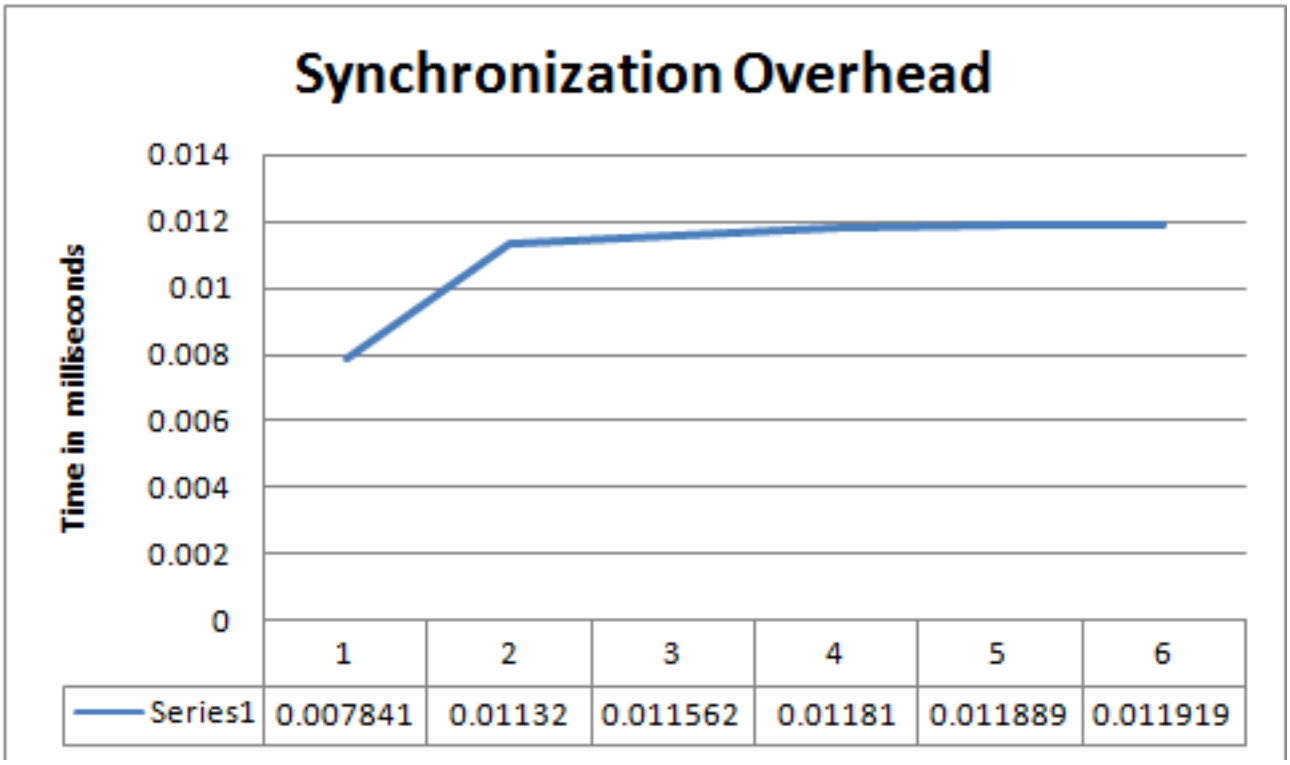


Figure 4.1: Synchronization overhead

We have observed that synchronization overhead remains almost constant across multiple streams for static synchronization.

4.1.2 Bi-directional data transfer bandwidth

Although current CUDA (from 4.0) provides two copy engines in some latest GPUs to support bi-directional data transfer, we found that the time cost of sending and receiving with two streams simultaneously is much longer than the time cost of only sending or receiving with single stream from our experiments. Without going into details of the hardware design, we can build an empirical model to predict the time cost of sending and receiving simultaneously with two streams.

We use the following equations to calculate the time cost of only sending data and only receiving data with single stream. We noticed that bandwidth of sending is little different than bandwidth of receiving.

In our experiments We found that when we overlap the sending and receiving, the time cost of simultaneous transfer part is longer than the time cost of only sending or receiving with single. Once either of them stop and the other one goes on, the time cost

of this part is equal to the time cost of only sending or receiving. Therefore, we can use the following equation to calculate T_p

$$T \quad = \quad min\{Ts, Tr\} \times f \quad + |T_s - T_r| \tag{4.4}$$

T :total time cost of simultaneous data send and receive

T_s :total time cost of data send from H2D

T_r :total time cost of receive data from D2H

f : bi-directional data transfer factor

The parameter f is a hardware related factor. For our GPU, f is 1.67.

4.1.3 Kernel Launch Overhead

We have observed that , there is some overhead attributed to the launch of kernels i.e compute engine needs some time time to allocate resources and launch the kernel.This is validated by calculating time taken for execution of dummy kernel across different number of streams.

Latest CUDA supports concurrent kernel execution (up-to 8 kernels),but when launched n dummy kernels across n streams we observed that time taken for individual kernel launch in each stream takes longer as we increase the number of streams(and hence no. of kernels), this is explained by the fact that compute engine needs time to allocate resources and schedule kernel.

We have launched a dummy kernel across streams .We took 2 pow(15) launches (iterations) in each stream . Calculated the time taken by them , which is largely contributed by kernel launch overhead.

$$T \quad = n \times \ (t_{dk} + t_{kohm}\ ) \tag{4.5}$$

$$t_{dk} << t_{kohm} \tag{4.6}$$

$$=> T \quad = \quad n \times t_{kohm} \tag{4.7}$$

T :total time cost

t_{dk} : time taken by dummy kernel

t_{kohm} : time taken for kernel launch when used m streams

Below is the graph of variation of overhead(in milliseconds) using different number of streams thus suggesting the model. In order to verify what we are calculating is kernel launch overhead , we can check the time line of the program using visual profiler.We have observed that kernel overhead increases almost linearly with the number of kernels launched

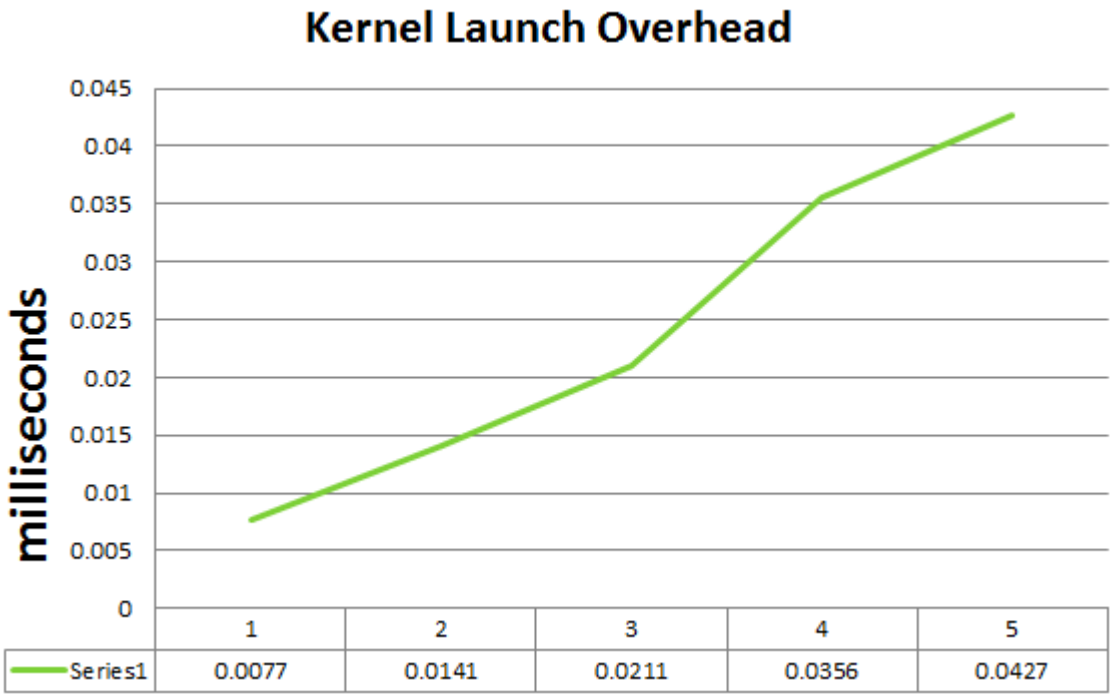


Figure 4.2: Kernel Launch Overhead

4.1.4 Memory Transfer overhead

With out going into hardware details of the GPU, we have observed that overhead of memory transaction is different for different types of data.i.e it is different for a float data type from character data type.From our experiment it is observed that this overhead is largely dependent on size of basic unit of data.Also overhead is different when there is a bi-directional transfer from uni-directional data transfer.

Below table shows the overhead of floating point data type at various conditions on

a TeslaK20C GPU.

Data Transfer	Overhead(in ms)
H2D	0.0047
D2H	0.0048
H2D,D2H	0.011

Table 4.2: Data Transfer Overhead

4.2 Task Partition Schemes

After taking all overheads into consideration,we can design the task partition models to minimize the total time cost of the applications

we have developed two models depending on compute-to-communication run time ratio of the application.

1. Data Bound Model
2. Compute Bound Model

4.2.1 Data Bound Model

The data bound model is used to calculate the optimal subtask number for data transfer bound application to minimize the total time cost of application. We assume the number of sub-task to be n. For data transfer application, we partition the application into sub-tasks with equal size. Therefore, we can use the following equation to calculate the time cost of send part in the sub-task and the time cost of receive part in the subtask

$$t_s = T_s/n \quad , \quad t_r = T_r/n \tag{4.8}$$

ts: time cost of single subtask data sent Ts: time cost of total data sent tr: time cost of single subtask data received Tr: time cost of total data received

As shown in figure 3.1, the total time cost is the sum of the first send time cost, n 1 times of the simultaneous send and receive time cost, the last receive time cost and overhead time cost. We can use the following equation to calculate the total time cost.

$$T_{total} = t_s+(n-1)\times t_p+t_r+(n+2)\times t_{sync}+n\times t_{koh}+(n-2)\times t_{boh}+2\times(t_{soh}+t_{roh}) \tag{4.9}$$

From above two equations we have the following equation to calculate the total time cost which is a function of n.

$$T_{total} = (n-2)\times(f\times min\{T_s/n, T_r/n\}+|T_s-T_r|/n)+(T_s+T_r)/n+n\times(t_{syn}+t_{koh}+t_{boh}+2\times(t_{syn}+t_{so} \tag{4.10}$$

With the mapping function between the total time cost and the number of subtasks, we can obtain the optimal number of subtasks by calculating the derivative of the function. We have the following equation to calculate the optimal number of subtasks

$$n = \sqrt{\frac{T_s + T_r - f \times min\{T_s, T_r\} - |T_s - T_r|}{t_{syn} + t_{boh} + t_{koh}}} \tag{4.11}$$

4.2.2 Compute Bound Model

The compute bound model is used to calculate the optimal number of subtasks for compute bound application.Comparing to the equal size of all subtasks in data transfer bound model, the subtask size will linearly increase in the first part of the application and then linearly decrease in the latter part of the application.Assuming the time cost of kernel execution in the first subtask to be t and the number of subtasks to be n and n to even, then we have the following equation to present the mapping between the time cost of kernel part in the first subtask and the total kernel execution time cost of the application.

$$T_k = 2 \times t \times (1 + 2 + 3 + ... + n/2 + n/2 + ...1) \tag{4.12}$$

Then we can calculate the computation time cost of the first subtask with n and T_k as shown in the following equation.

$$t = (4 \times T_k)/(n \times (n+2)) \tag{4.13}$$

As shown in 3.1, we can sum up the send part in the first subtask, the total kernel execution of all subtasks, the receive part in the last subtask and overhead to calculate the total time cost of the application with the following equation which is also a function of n.

$$T_{total} = t_{s1} + T_k + t_{rn} + (n+2) \times t_{sync} + n \times (t_{koh} + t_{boh}) + c \tag{4.14}$$

Notice that the subtask size queue is symmetric, i.e the size of subtask 1 is equal to the size of subtask n and the size of subtask 2 is equal to the size of subtask n 1. Therefore, we can use the following equation to calculate the time cost of send part in the first subtask and the time cost of receive part in the last subtask.By using differential we can approximate the optimal no. of sub tasks to following value

$$n = \sqrt{\frac{8 \times (T_s + T_r)}{t_{syn} + t_{boh} + t_{koh}}} \tag{4.15}$$

CHAPTER 5

Results

We have applied our task partition and scheduling schemes on 3 applications with varied parameters and verified our model and analysis.

5.1 Sepia Filter

First of all we have applied our model on Sepia Filter which comes under Data Independent Application.We have applied filter over the data several times to vary the computation time while keeping data transfer time constant and observed how total time taken in varying compared with naive implementation.

Below graph shows the improvement in run time of optimized application for different number of times the filter is applied against the naive implementation(send->compute->receive).

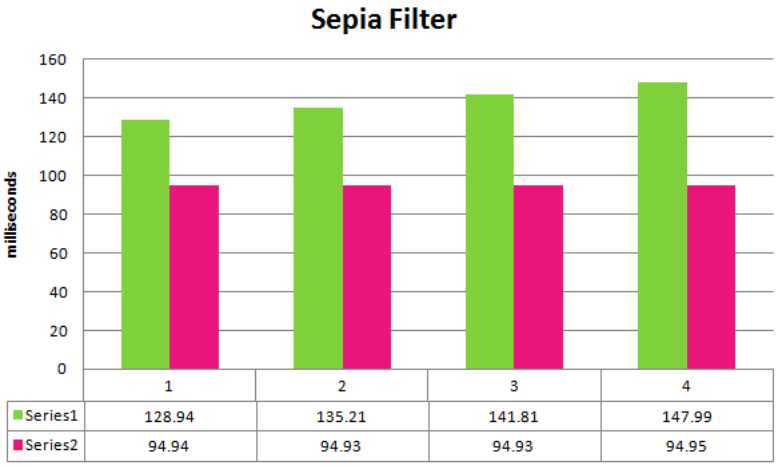


Figure 5.1: Sepia Filter

We have observed that effective run time of the application hasn't changed much despite the increase in computation time which is visible in the increase of run time for naive computation as the no. of times filter is applied increases.This can be explained

from the fact that the application is data intensive and hence all the computation of the application is overlapped by data transfer ,this leads to no increase on effective runtime.

Below is the image of timeline of the application captured using NVIDIA Visual Profiler validating the complete overlap of data transfer with computation.

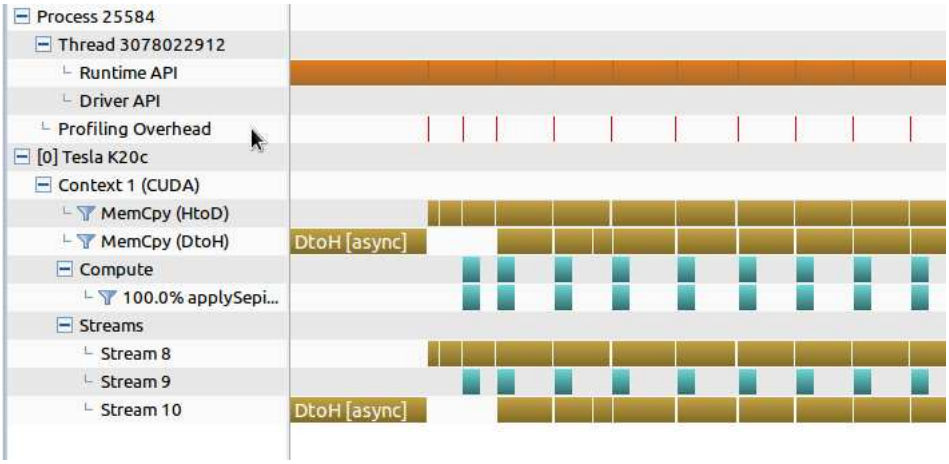


Figure 5.2: Sepia Timeline

5.2 Matrix Multiplication

Matrix multiplication is an important part of various applications , we have optimised matrix multiplication by dividing matrix as blocks and scheduling partial multiplication across different streams.

We have used cuda events to resolve dependencies that arise while doing a partial multiplication.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \quad (5.1)$$

Lets divide matrix is divided into 4 sub-matrices, then the following four equations represent individual values of $C_{11}, C_{12}, C_{21}, C_{22}$

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21} \tag{5.2}$$

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22} \tag{5.3}$$

$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21} \tag{5.4}$$

$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22} \tag{5.5}$$

From the above equation we see the data independency across partial multiplication of different sub matrices.So we schedule the entire multiplication in such a way that there is overlap of data transfer and computation.

We declare 6 streams , 1 for data send,1 for data receive,4 for computation of four submatrices. we load matrix A,matrix B in chunks of sub matrices and notify a CUDA event for each successful transaction.These are be used to resolve data dependency issues while scheduling.

Below is the time line of matrix multiplication.We can see the overlap of memory transfer and computation.Also overlap of computation kernels.

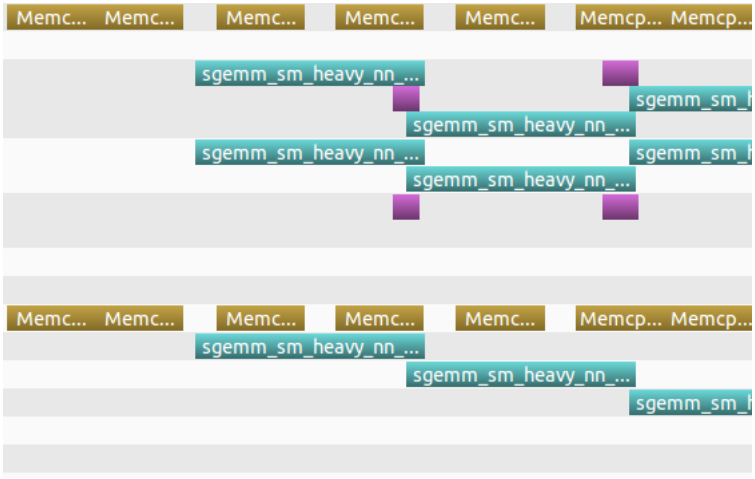


Figure 5.3: Matrix Multipliction

Complete timeline is given below

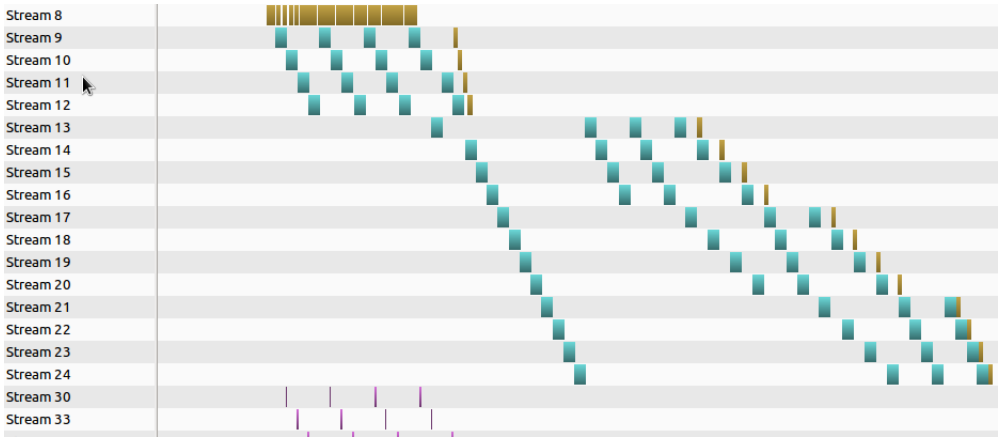


Figure 5.4: Matrix Multiplication Full Time Line

We automated the scheduling for different block sizes and ran it through different block sizes.We have observed that as the block size decreases i.e no. of blocks to schedule increases and hence the scheduling overhead but as the the block size increases no. of optimal no. of sub blocks it is divided into increases.

Below no. of 3D graph shows the variation of runtime when divided into different blocks.

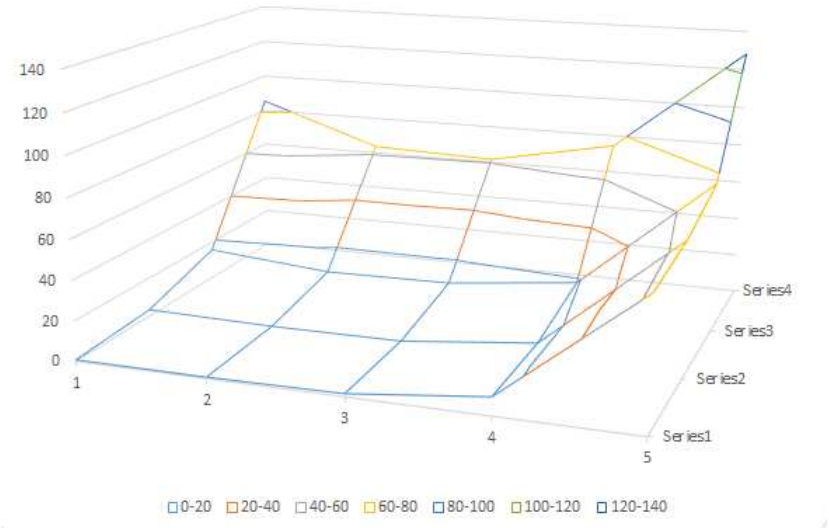


Figure 5.5: Matrix size vs Time Taken vs No.of blocks

Below Table has the run time of different sized matrices divided into different number of blocks

size	1*1	2*2	4*4	8*8	16*16
512	.749ms	0.639ms	1.488ms	9.065ms	65.011ms
1024	3.175ms	2.256ms	2.570ms	9.941ms	68.399ms
2048	15.788 ms	9.973ms	10.943 ms	18.425 ms	77.342ms
4096	85.821 ms	63.063ms	61.888 ms	74.315 ms	128.528 ms
8192	546.423 ms	452.392 ms	447.390 ms	474.103 ms	573.209 ms

Figure 5.6: Matrix size vs Time Taken vs No.of blocks

CHAPTER 6

Conclusion

In this work , we have presented ways to divide applications in order to maximize overlap between memory transfer and computation.We have also modeled various parameters involved in overhead generated while scheduling sub tasks of the application and thus deciding the optimal sub task size.From our experiments we successfully overlapped data transfer and kernel execution.We also found improvement in run time of the application depending on the extent to which we can overlap different sub tasks.For application with very little scope of memory transfer and computation overlap , there will be very little improvement in run time.In the future work , we can build more accurate models for overhead generation by taking into account the probabilistic variation involved delay in scheduling the sub tasks.

REFERENCES

1. **Hong, S.** and **H. Kim**, An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *In Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09. ACM, 2009. ISBN 978-1-60558-526-0. URL `http://doi.acm.org/10.1145/1555754.1555775`.

2. **Hou, Q.**, **K. Zhou**, and **B. Guo**, Bsgp: bulk-synchronous gpu programming. *In ACM Transactions on Graphics (TOG)*, volume 27. ACM, 2008.

3. **Ian Buck, D. H. J. S. P. H. M. H. K. F., Tim Foley** (2004). Brook stream program language. URL `http://graphics.stanford.edu/projects/brookgpu/`.

4. **KhronosGroup** (2008). Open computing language (opencl). URL `https://www.khronos.org/opencl/`.

5. **Luo, C.** and **R. Suda**, A performance and energy consumption analytical model for gpu. *In Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*. 2011.

6. **Luo, C.** and **R. Suda**, Mssm: An efficient scheduling mechanism for cuda basing on task partition. *In Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. 2012. ISSN 1521-9097.

7. **Nakagawa, S.**, **F. Ino**, and **K. Hagihara** (2010). A middleware for efficient stream processing in cuda. *Computer Science - RD*, 41–49.

8. **NVIDIA** (2007). The cuda computing platform. URL `http://www.nvidia.in/object/cuda_home_new.html`.

9. **Owens, J.**, **S. Rixner**, **U. Kapasi**, **P. Mattson**, **B. Towles**, **B. Serebrin**, and **W. Dally**, Media processing applications on the imagine stream processor. *In Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*. 2002. ISSN 1063-6404.