ACCELERATED SMPP DECODER IMPLEMENTATION BASED ON GPU

Prabath Sanjeewa Abeygunawardana Weerasinghe
Registration Number : 148244C

Degree of Master of Science

Department of Computer Science and Engineering
University of Moratuwa
Sri Lanka

December 2017

Declaration

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

Candidate

....

Prabath Sanjeewa Abeygunawardana Weerasinghe **Date**

I certify that the declaration above by the candidate is true to the best of my knowledge and he has carried out research for the Masters Dissertation under my supervision.

....

Prof. Sanath Jayasena **Date**

Acknowledgment

This project would not have been possible without the support of many people.

A special thank goes to my supervisor, Prof. Sanath Jayasena for his valuable guidance and immense support. Many thanks to our MSc research project coordinators, Dr. Shehan Perera and Dr. Malaka Walpola, for their dedication and support. Thanks to all the lecturers at the Faculty of Computer Science and Engineering, University of Moratuwa, for their valuable advices.

Abstract

Graphic processing unit (GPU) provides a low-cost but powerful hardware platform for implementing massively parallel high performance systems. The capabilities of GPUs have been used to provide fast and low cost solutions in areas such as machine learning, complex simulations such as global warming and genetic engineering and network traffic processing.

Our research is focused on using GPUs to accelerate the decoding process of the Short Message Peer to Peer (SMPP) protocol. SMPP protocol is used to exchange *Short Messages* between Short Message Service Centers (SMSCs) and TCP/IP based applications. From the point of view of a user, a SMS taking few seconds is acceptable and therefore a SMSC is mostly focused on achieving a higher throughput than a low per packet latency.

We have developed a SMPP decoder library in C with GPU support. It supports both CPU based and GPU based decoding. The library also includes two primary APIs. The first API is for general usage by any C based application and the second API could be used by Java Native Interface (JNI) based application.

We have evaluated the performance of the library in both CPU and GPU modes and compared it with a SMPP server based on Cloudhopper, a Java implementation of SMPP protocol. The evaluation shows around five times throughput gain in GPU mode over both Java and C based CPU modes.

Contents

List of Figures

List of Abbreviations

GPU	Graphics Processing Unit
SMPP	Short Message Peer to Peer
CPU	Central Processing Unit
SMS	Short Message System
SMSC	Short Message Service Center
ESME	External Short Message Entity
PDU	Protocol Data Unit
TLV	Tag Length Value
CUDA	Compute Unified Device Architecture
SM	Streaming Multiprocessor
SP	Streaming Processor
UMA	Unified Memory Access

1 Introduction

1.1 Background

Short Message Service (SMS) [1] is a popular value added service provided by telecommunication service providers and uses existing signaling networks for communication. Telecommunication infrastructures for signaling networks are expensive and setting up a test environment is rather costly.

The Short Message Peer to Peer (SMPP) [2], a TCP/IP based protocol, was created initially as a low cost solution for testing SMSC (Short Message Service Center) functionalities. Later SMS based value added services such as personalized applications (for example: thought of the day, jokes, and match making applications) gained popularity and SMPP was revamped as the standard communication protocol in the IP network for SMS. Well-known SMPP protocol implementations (Cloudhopper [3], OpenSMPP [4]) perform protocol data unit decoding and encoding using only the CPU.

Graphics Processing Units (GPU) [5] opened up a new paradigm in parallel computing. GPUs with their processing cores (streaming cores), are specifically engineered to run data independent (data parallelizable) operations efficiently. This generalized use of GPUs for operations other than graphics processing is known as *General Purpose GPU* (GPGPU) programming. Overtime GPU vendors have come up with much simpler programming models and APIs for utilizing GPGPUs.

In this research we have focused on developing a SMPP decoder library which offloads the decoding process to a GPU.

1.2 Problem Statement

There exists SMPP protocol implementations such as Cloudhopper [3] and OpenSMPP [4] etc. Cloudhopper [3] from Twitter is the most popular SMPP framework and is written in Java. It uses the Java I/O library Netty [6] for low level network traffic management. Cloudhopper [3] supports CPU based multi-threading through Java threads. But it is more focused on parallelizing SMPP sessions rather than packet encoding and decoding and that leaves room for improvement.

In this research, we wish to investigate whether the SMPP decoding throughput could be improved by utilizing parallel computing specifically with regard to GPUs. SMSCs usually

act as SMPP servers which act on received SMPP messages. Therefore our research has focused on improving SMPP decoding performance only.

1.3 Objectives

The objectives of our research are as follows.

1. Implementing a SMPP decoder based on a multi-core CPU.
2. Implementing a SMPP decoder which utilizes a GPU.
3. Evaluating the performance of both implementations against an existing implementation.

1.4 Overview of the Thesis

Chapter 2 contains the literature survey. The literature survey starts with an introduction to SMPP protocol and the SMPP decoding algorithm. Next we introduce Cloudhopper as a Java based SMPP framework and provide a brief description of the internal logic and architecture. Next we start looking into heterogeneous computing and GPU programming models. Then under related works we discuss network packet processing in relation with GPUs. The discussion is focused on four main aspects of network packet processing.

Chapter 3 discusses implementation details. It starts with a discussion of the architecture of our SMPP decoding library and provides brief introductions of two main APIs in the library.

Chapter 4 encapsulates evaluation details including the setup of the experiment and the analysis of the results. Chapter 5 contains the conclusion and chapter 6 presents potential future improvements for the system.

2 Literature Survey

2.1 Introduction to SMPP Protocol

The SMPP protocol, though the name says peer to peer, actually uses a client-server model. The client side components are commonly known as External Short Message Entity (ESME). An ESME is basically a SMPP client that receives and sends short messages. The server side is a SMSC [2].

SMPP protocol defines mainly 12 types of Protocol Date Units (PDUs) and all but one (ALERT_SM) have response PDUs as well. The most frequently used PDUs are the SUBMIT_SM and the DELIVER_SM. Both types have their restrictions on the direction. Figure 2-1 depicts the basic message configuration for SMPP protocol.

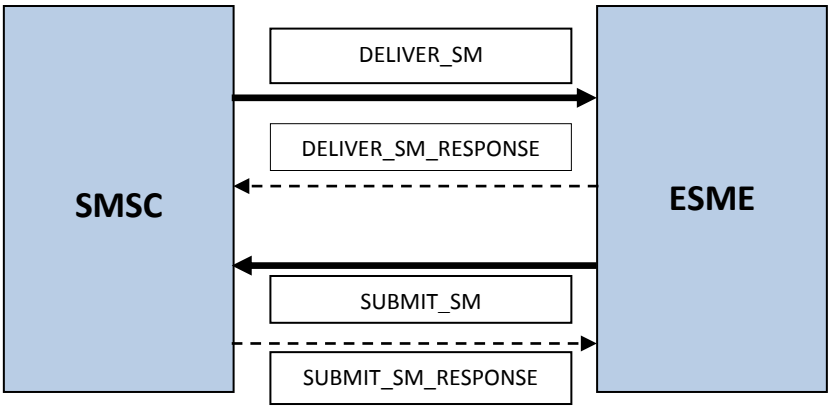

Figure 2-1 : SMPP frequently used message flow

SMPP PDU Structure

A SMPP packet consists of three main sections as shown in the Figure 2-2. The first section is the header. The header contains information needed to identify the type of the packet and the status of the request or the response (success or failure reason). The mandatory and optional parameter sections are part of the PDU body.

Mandatory parameters are well defined for their length and value types. The type and number of mandatory parameters in a PDU vary with the type of the PDU.

Optional parameters are defined as Tag Length Value (TLV) fields. The type and number of optional parameters are dependent on the PDU type as well.
For our work we mainly focus on DELIVER_SM and SUBMIT_SM and their responses as they are the mostly used PDUs in SMPP communication.

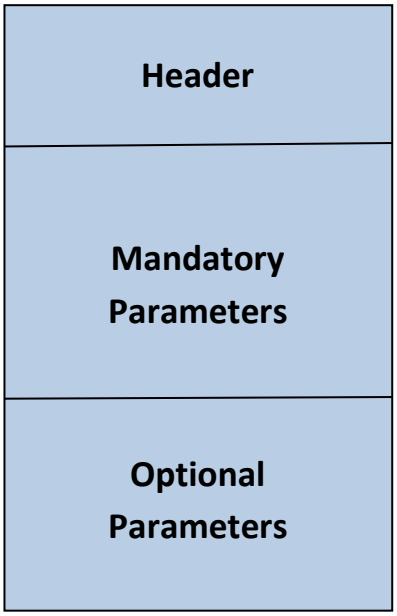

Figure 2-2 : SMPP PDU structure

2.2 Cloudhopper SMPP Library

Cloudhopper is a SMPP protocol framework written in Java by Twitter. In addition to decoding and encoding SMPP PDUs, Cloudhopper provides a set of convenient interfaces to manage network connections in terms of creating TCP connections, managing request and response timeouts, managing connection failures and managing SMPP sessions.

Cloudhopper implementation depends heavily on the popular Java NIO library Netty [7] [8]. It offers two basic operation modes, server and client. An implementation can create an SMPP server that accepts and process SMPP requests or an SMPP client which may initiate an SMPP session. The following subsections will look into the implementation of each mode.

2.2.1 SMPP Decoding Process

Cloudhopper uses one thread per PDU for decoding. SMPP uses TCP as the transport layer protocol. Therefore the data for a particular PDU might not be in a single TCP segment and Cloudhopper waits for the complete SMPP PDU.

After receiving all the data for the PDU, the framework first reads the PDU header and figures out the PDU type and total length. Then based on the PDU type, it first decodes the mandatory parameters. Lastly the framework decodes optional parameters. Decoding optional

parameters is not straight forward as decoding mandatory parameters. Optional parameters are encoded in *TLV* format as we have mentioned before. Therefore decoding optional parameters need to be done sequentially.

2.2.2 Netty Library Architecture

Netty uses the term *channel* to identify an entity which wraps components capable of doing I/O such as network sockets. When it comes to network I/O, a channel may use a TCP, UDP or SCTP socket.

A channel always has a pipeline associated with it. Pipeline acts as the carrier for sending (outbound) and receiving (inbound) data. A pipeline can have a set of handlers for processing the aforementioned outbound and inbound data. Handles carry the logic for processing inbound and outbound data.

For an example, a Netty based HTTP library implementation may have several handlers. One of the handlers has the logic for encoding and decoding data. Another handler would manage connections and so on. In a nutshell, handlers encapsulate the domain specific logic in a Netty based application. Figure 2-3 shows the relationship between *channels*, *pipelines* and *handlers*.

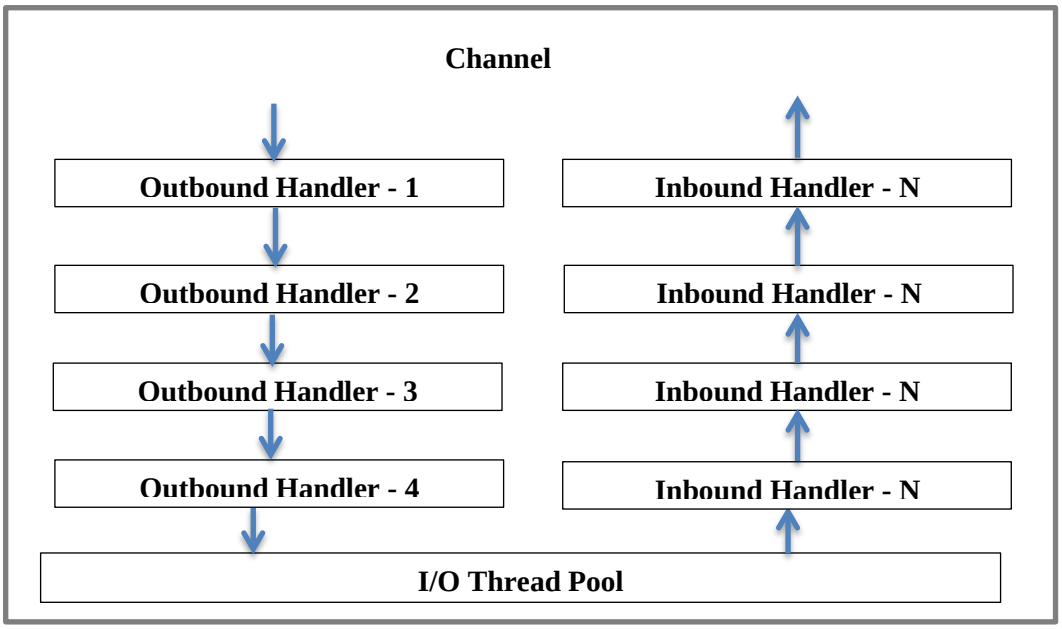

Figure 2-3 : Netty Inbound and Outbound Message Handling Architecture [6]

2.3 Graphic Processing Units (GPUs)

In 1999, Nvidia introduced GPU to the world of personal computers. At the time, a GPU is essentially a collection of graphic related processing units. The technical definition of a GPU coined by Nvidia is as follows. "*A GPU is a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second*" [9] .

One of the unique traits of graphic processing is that most of the operations are data independent. In other words in general, graphic processing operations are data parallelizable. GPUs have been designed specifically to optimize the data parallelizable operations and that opened up a new platform for developing and executing low cost and high performance applications. The term General Purpose GPU (GPGPU) is used to identify the use of GPUs for computations normally done by the Central Processing Unit (CPU).

In early days, writing applications for GPUs was cumbersome. One of the main reasons was the complexity associated with the APIs. Programmers had to master the inner working of a GPU before developing an application. As GPGPUs gained popularity, manufactures introduced APIs with less coupling to the underlying hardware layer. The two most popular such APIs are Open Computing Language (OpenCL) and Compute Unified Device Architecture (CUDA). OpenCL framework is more of a generic heterogeneous programming standard and apart from programming GPUs it can be used to program a wide array of devices including Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs). CUDA was developed not as a generic framework but as a platform and a framework to work with Nvidia GPUs. CUDA provides a much simpler API implementation than OpenCL and also has a rich echo system of libraries and tool sets.

For the project we focus on CUDA [10] parallel computing platform implemented in C/C++.

2.3.1 Heterogeneous Computing

Heterogeneous computing model uses different types of computing cores. It is different from homogeneous computing model where the latter uses a single type of computing core. For an example the original Cloudhopper framework follows a homogenous computing model where it uses only the CPU. In contrast, the SMPP decoder framework that we wish to develop uses the CPU as well as the GPU. In general GPGPU applications follow the heterogeneous model in which a part of the application tasks gets offloaded to the GPU.

2.3.2 Nvidia GPU Hardware Architecture

Usually a single GPU has a large number of cores compared to a modern multi-core CPU. For an example, a GTX 750 Ti GPU has around 600 cores. How a CPU core is different from a GPU core is fundamental to understand the way a GPU works as GPU and CPU cores are optimized for different purposes. Nvidia uses the term streaming Processor (SP) for a GPU core.

GPUs focus on data parallelism while CPUs are more generic and support both data and task parallelism. To achieve a high data parallelism, GPUs relies on a simple but a large number of cores. In order to reduce the cost, the power consumption and the size, a single GPU core has a limited set of functionalities as well as limited in computational power. Figure 2-4 shows an Nvidia Fermi architecture GPU core block diagram.

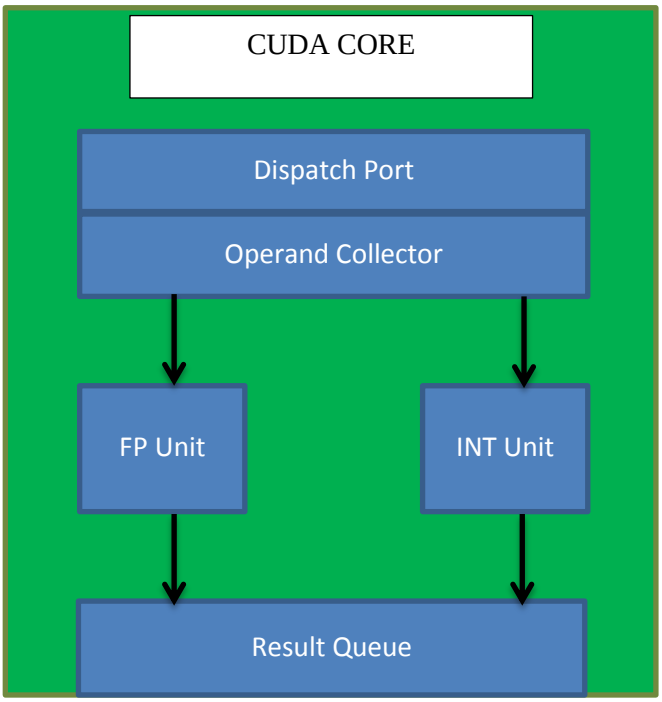

Figure 2-4 : GPU streaming Processor Core Block Diagram [11]

Figure 2-5 shows an AMD processor block diagram.

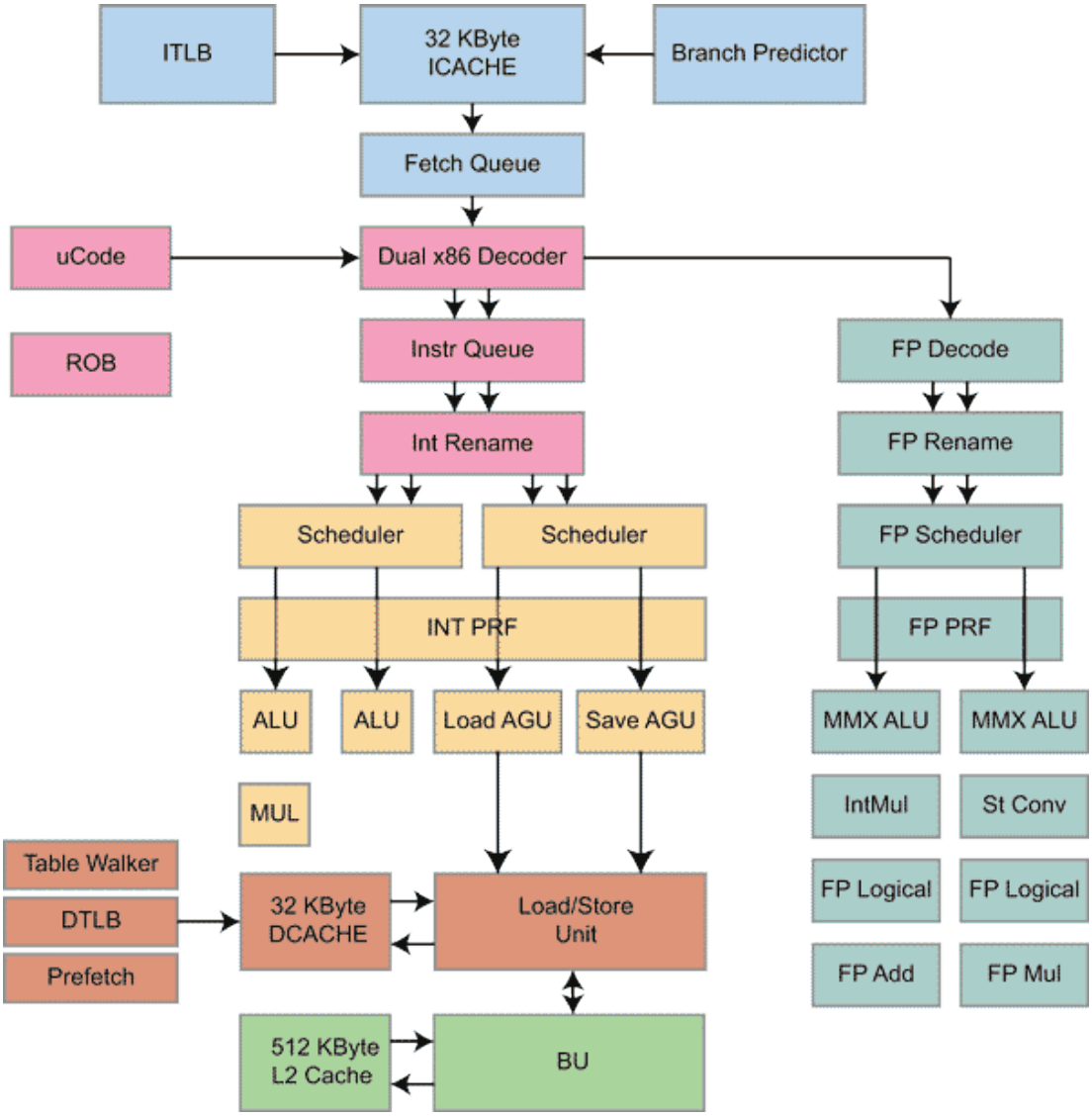

Figure 2-5 : AMD CPU Processor Core Block Diagram [12]

Here we have listed the differences between a CPU Core and a GPU Core.

1. A GPU core cannot fetch, decode or dispatch instructions. It relies on instruction schedulers of streaming Multiprocessors (SM) to do so. For now we may consider a SM as a controller of a set of GPU cores and a GPU would have multiple SMs. A normal CPU core has its own program counter and fetch, decode and execute units. Also a CPU core has a much larger number of pipeline stages, sometimes in the range of 20 parallel stages.

2. A CPU core supports branch predictions. GPU cores do not support branch prediction and it is highly advised to not use extensive number of branching instructions in a GPU program.

3. A GPU core does not have individual caches and registers and a CPU core has much large caches and registers. GPU cores use a set of shared registers and caches. A GPU core gets allocated a set of registers by the relevant SM.

4. A GPU core cannot handle interruptions, cannot handle I/O read/write where as a normal CPU core is well capable of handling I/O and interruptions.

As a summary, a CPU core is a more self-sustaining unit designed to execute a single instruction as fast as possible with a low latency. To achieve the low latency in instruction execution, a CPU core has more additional units of functionalities and that increase the amount of transistors used and the power consumption.

A GPU core on the other hand is not a self-sustaining computational unit. It relies on a common controller to fetch decode and schedule instructions. Apart from the aforementioned shortcomings, a GPU core has well designed floating point and integer ALU units. These floating point and integer ALUs make up most of the GPU core. In essence a GPU core is really a container of floating point and integer ALUs. Some GPU architectures have one floating point and one integer ALU per core as with the Fermi architecture and another set of GPUs have more than one ALU per core.

2.3.3 Streaming Multiprocessor

Nvidia GPUs have been evolving with several generations of microarchitectures such as Tesla [12], Fermi [11], Kepler [13], Maxwell [14] and Pascal [15]. For our discussion we have focused on Fermi microarchitecture. The reason for considering Fermi is that it has all the important elements of successor microarchitectures such as Kepler and only lacks in the number of *streaming processors* and *streaming multiprocessors*. Older microarchitectures such as Tesla lack multiple warp schedulers per *streaming multiprocessor* and store and load units.

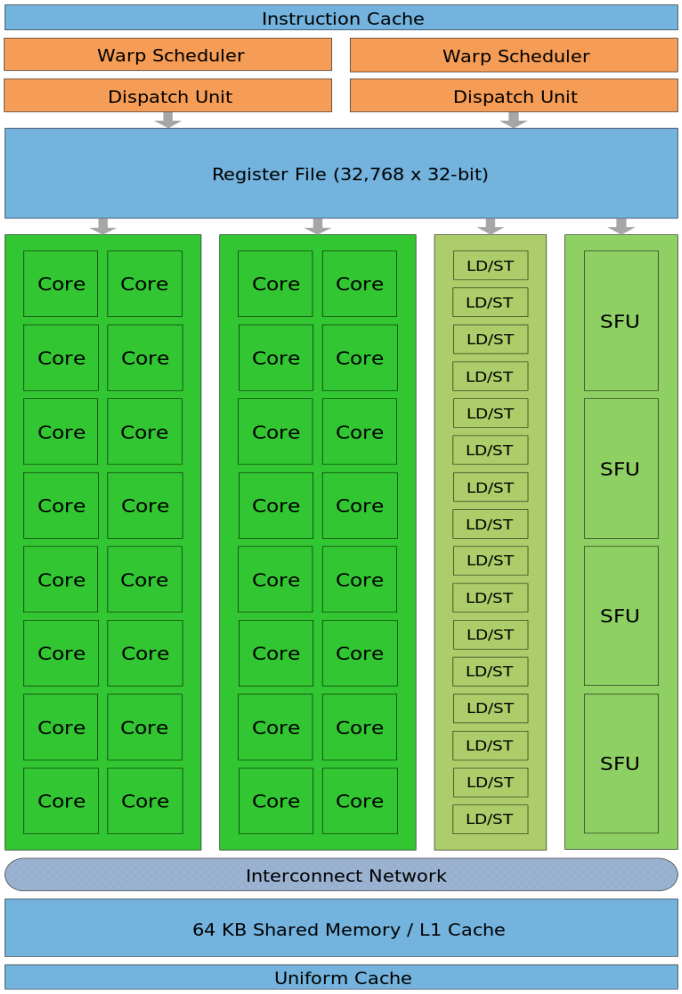

Figure 2-6 : Nvidia Fermi Architecture - streaming Multiprocessor Block Diagram [11]

Figure 2-6 shows an Nvidia Fermi architecture *streaming multiprocessor* micro-architecture block diagram [14]. Streaming multiprocessor is a term used by NVIDIA to identify a unit that controls a group of streaming Processors (SP). As shown in the figure 2-6, a SM has several key components.

1. GPU Cores (streaming processors)

   Fermi architecture puts 32 cores in a single SM. As explained previously, we may think of a single core as a container for a Floating Point and an Integer ALU and each of these cores can be utilized concurrently.

2. Warp Scheduler

   GPUs confirming to Fermi architecture have two warp schedulers as seen in the figure 2-6. Before understanding warp schedulers, it is important to understand what a warp is. A SM schedules N number (The number depends on the GPU architecture) of parallel threads at once to carry out an instruction. A group of such scheduled

threads are called a warp. Please note that warps are not a set of streaming processors but a set of threads and these threads can be scheduled on a given set of streaming processors. Usually the warp size is 32 threads for Fermi architecture. It means that when an SM has a set of instructions and needs to run them on N number of threads, the SM schedules the instructions in batches of 32 threads. But we have mentioned previously that in Fermi architecture, a SM has only 32 streaming processors. So why does a Fermi SM have two warp schedulers?

The answer is with the clock speed at which the warp schedulers work and the clock speed at which the streaming processors work. A SP works at about twice the rate of a warp scheduler. It is called the Hot Clock. So this allows each warp scheduler to schedule selected warps on just 16 SPs. As CUDA cores run at twice the scheduler's clock rate, by the time of the next instruction, the set of 16 cores would run 32 cores worth of operations. This allows the GPU to attain a higher level of hardware utilization. Figure 2-7 is a sample of how warp scheduling is managed with two warp schedulers.

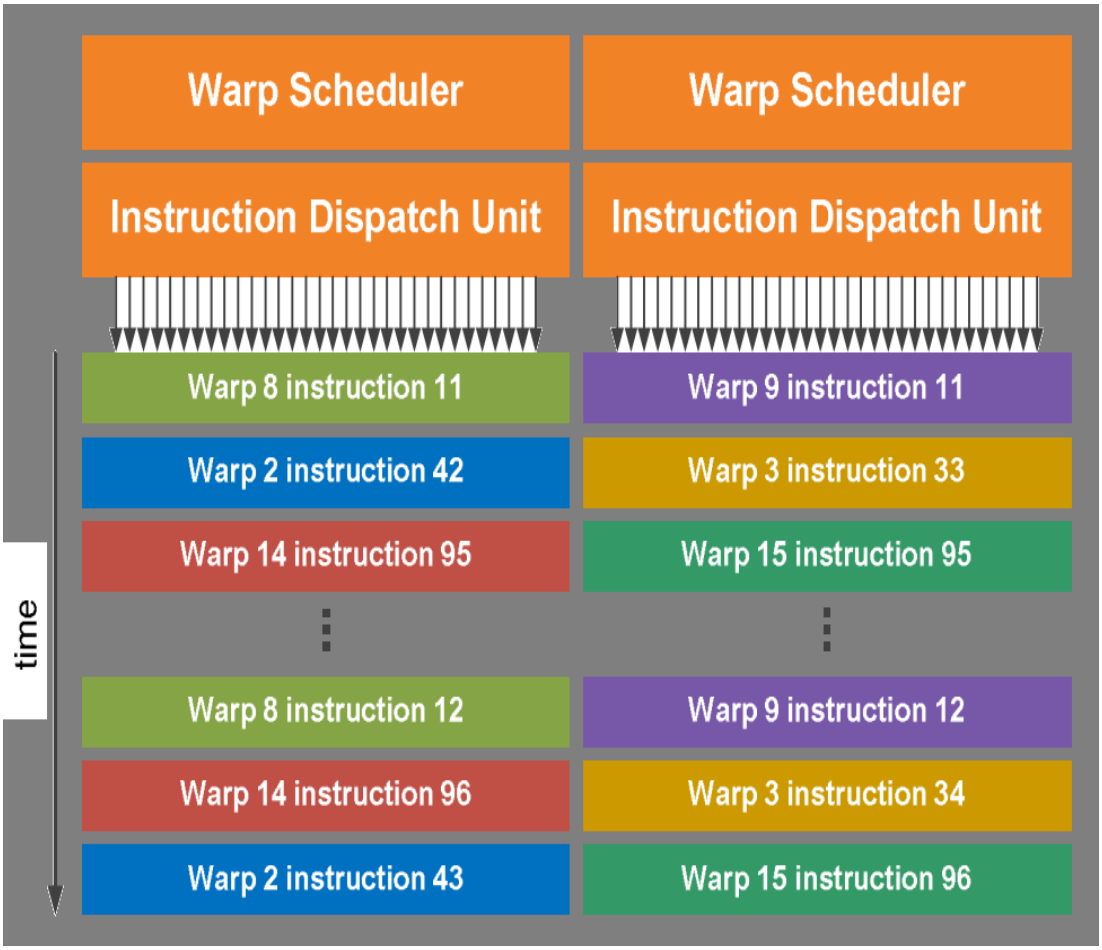

Figure 2-7 : Dual Warp Scheduler in Fermi Architecture [11]

3. Store and Load Units

   Warp schedulers use Store and Load units to store/load data to/from the global memory, shared memory and cache. Fermi architecture has 16 Load and Store units and they operate at the warp scheduler's clock speed. Therefore only a maximum of 16 threads can be served per clock cycle.

4. Special Function Units

   Special function units are for carrying out instructions such as sin, cosine, reciprocal or square root. Fermi architecture has four special function units. With that a complete warp takes about 8 cycles to execute a special function. Special function units have a decoupled pipeline from the instruction dispatcher of a scheduler. What it means is that the dispatchers can issue instructions to other units concurrently.

5. Registers, Cache and Shared Memory

   Cache and registers are shared within a thread and the shared memory is used for sharing and coordinating among threads in a block. We will discuss more about blocks and shared memory in later sections.

2.3.4 CUDA Programming Model

In previous sections we discussed the main components of a GPU and the basics of the Nvidia GPU architecture. We discussed the architecture mainly in reference with the Fermi architecture. As the architecture evolves the key components evolve too such as the number of streaming Processors per SM, the number of warp schedulers per SM, warp size, clock speed, Store/Load and Special Function Units etc.

For the idea of using GPUs for general operations to be viable, it is essential to decouple development API from the underlying GPU architectures. Therefore Nvidia CUDA framework wraps the underlying device specific architectural details and provides an abstract API for the application developers. Doing so allows application developers to focus on the business logic instead of every underlying hardware and driver detail. In following we will look into the CUDA programming model and the API. First we will discuss few key concepts in CUDA programming model.

1. Device and Host and kernel

   Device and *host* are two of the main abstractions in CUDA. The term device means the GPU and host means the CPU. Every CUDA application has a part that runs on the CPU and another part that runs on the GPU. The GPU bound execution units are called *kernels*. Though usually a kernel has to be launched by the host, there are advance ways to launch kernels by another kernel.

2. Blocks and Grids

   Blocks and *grids* are CUDA abstractions that for configuring threads. When scheduling a kernel, the application needs to provide the number of threads the kernel to be executed on. Instead of just configuring the number of threads for a kernel, block and grid provides a more flexible model of defining threads.

   A block is a collection of threads. The number of threads for a block can be given as a single or two or three dimensional array. For an example if the block size is N threads, it could be represented as <N> or <N/2, N/2> or <N/3, N/3, N/3>.

   A grid is a collection of blocks. Same as a block, the number of blocks for a grid can be given as a single or two or three dimensional array. Following example shows the thread configuration for a kernel that runs on 120 threads.
   It has 10 blocks per grid and 12 threads per block.
   kernel_1<<<10, 12>>>()
   CUDA API provides C structures for defining grid and block dimensions more conveniently. Blocks and grids have few unique execution behaviors too.

   First is that the threads of a block always get scheduled on the same SM. For an example, if the number of threads for a block is 1024 and the number of streaming processors in a SM is 32, then all 1024 threads would be scheduled on those 32 SPs regardless of the availability of other SMs. Therefore developers may need to pay some attention to hardware configurations.

   Second is utilizing shared memory. Threads in a block share the SM's shared memory. In fact the developer may define the size of the shared memory dynamically for a block too. But keep in mind that memory access is not volatile in the sense that changes made to the shared memory by threads in a block might not be visible to other threads in the same block immediately.

Third is barrier support. The developer may use thread barriers for threads of a block. It is useful in certain scenarios such as loading data to the shared memory from the global memory. Figure 2-8 shows the relationship between grids and blocks.

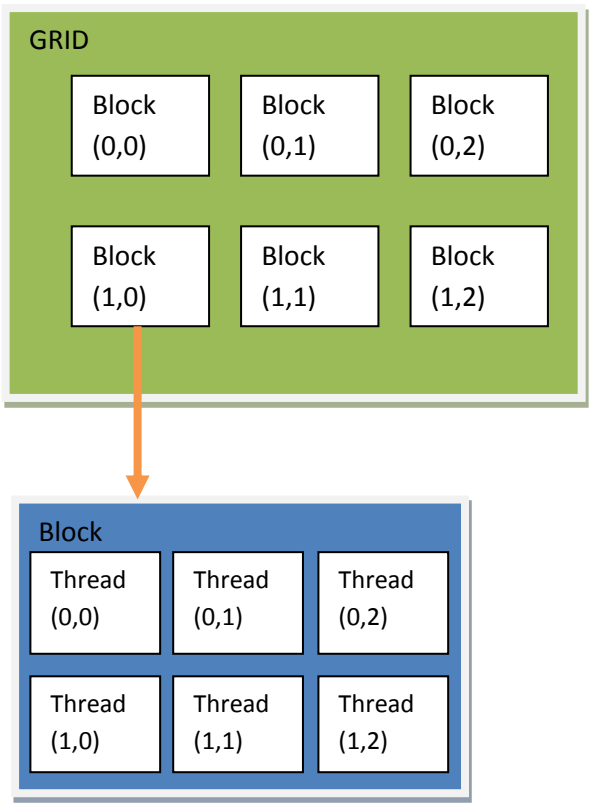

Figure 2-8 : CUDA Grid and Block Structure

Executing a kernel usually involves the following steps.

1. Allocating global memory in the GPU (device).
2. Copying data to be processed to the allocated memory.
3. Scheduling the kernel and thus processing the data.
4. Copying the result back from the GPU (device) to CPU (host).

Depending on the application, some of the steps can be skipped like copying data back to host memory. But at the end, these four steps or several of the steps would be followed by every CUDA based application. Figure 2-9 depicts three of the aforementioned four steps (allocating memory in GPU is not shown).

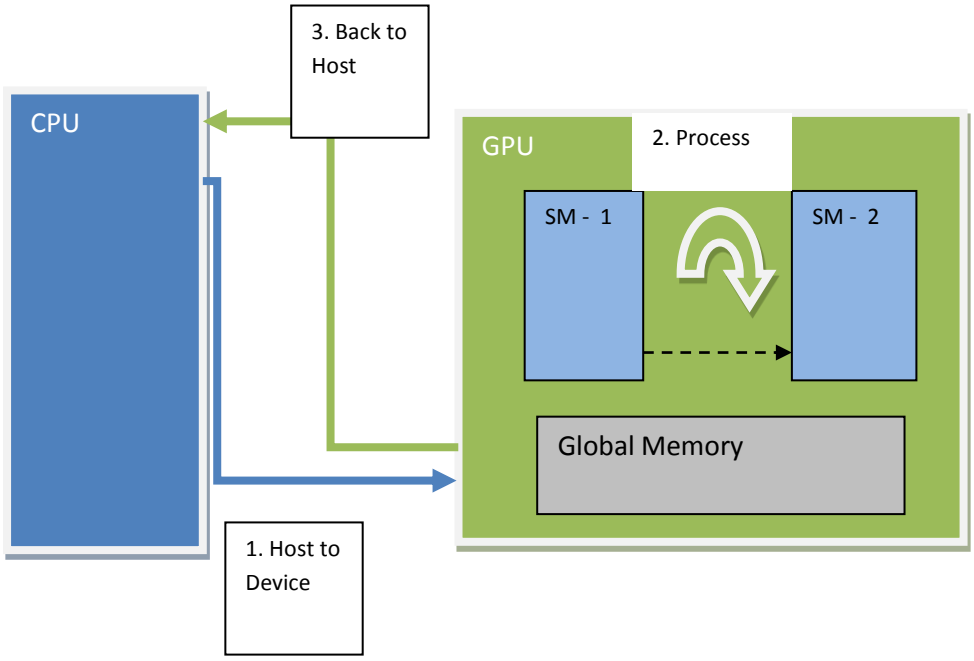

Figure 2-9 : Basic CUDA data flow

2.4 GPU Based Network Packet Processing

Research has been done on various types of GPU based network packet processing systems such as Intrusion Detection Systems [15] [16], Routers [17] [18] [19] [20] and TPC/IP packet decoders [21] [22]. Though these systems have different functionalities and algorithms, generally they all use a common set of activities as shown below.

1. Packet capturing.
2. Packet classification and pre-processing.
3. Transferring packets between the GPU and the CPU.
4. Processing packets in the GPU.

Let us take a more comprehensive look into each activity. Please note that the aforementioned activities are not listed in any particular execution order. For an example, packet classification and pre-processing could be done either in the CPU or the GPU. Therefore packets might need to be transferred to the GPU before carrying out classification or pre-processing.

2.4.1 Packet Capturing

A network packet processor, irrespective of the type, operates on a specific layer of the OSI network stack. For an example, a GPU based router might need only to decode the OSI Layer 3 data and a GPU based switch might need to decode the OSI Layer 2 data. An implementation like the SMPP decoder handles a specific application layer protocol such as SMPP.

Most of the network packet processors run on general purpose operating systems (Windows, Ubuntu, Fedora, FreeBSD). These operating systems may not handle network traffic in an optimized way. For an example, Linux network driver uses two types of buffers, data and metadata. The driver allocates and de-allocates these buffers for each packet. The allocation and de-allocation overhead could be considerable at a higher receiving and transmission rate. Because of that some GPU based packet applications have changed the network driver to avert the aforementioned overhead.

A solution for this has been proposed in PacketShader [17] in which the operating system allocates a large data and metadata buffers initially to minimize the allocation and de-allocation overhead.

2.4.2 Packet Classification and Pre-Processing

Packet classification is a key aspect of a network packet processor. Classification extracts useful information from the packet prior to the decoding stage. The specific nature of the information depends on the application. For an example, an application might need to ensure the availability of the complete TCP based packet prior to start the decoding process. Thus the classification stage might need to check the length of the available data based on the protocol specification.

Another example would be to identify the type of the packet. In order to do that, the classification module might use pattern matching on a byte stream. When it comes to GPU based network packet processors, one might choose to do the classification in the GPU or in the CPU [21]. For the GPU to be utilized properly, the classification mechanism should be complex and parallelizable enough to justify the time taken for transferring data from the CPU to the GPU.

For an example, a simple TCP segment check, to make sure the availability of the complete packet might not be efficient enough to be done in the GPU. Even if the logic is complex and

parallelizable enough to run on the GPU, the application needs to decide on the size of the batch size and block and grid sizes etc. based on performance metrics such as the throughput and the latency.

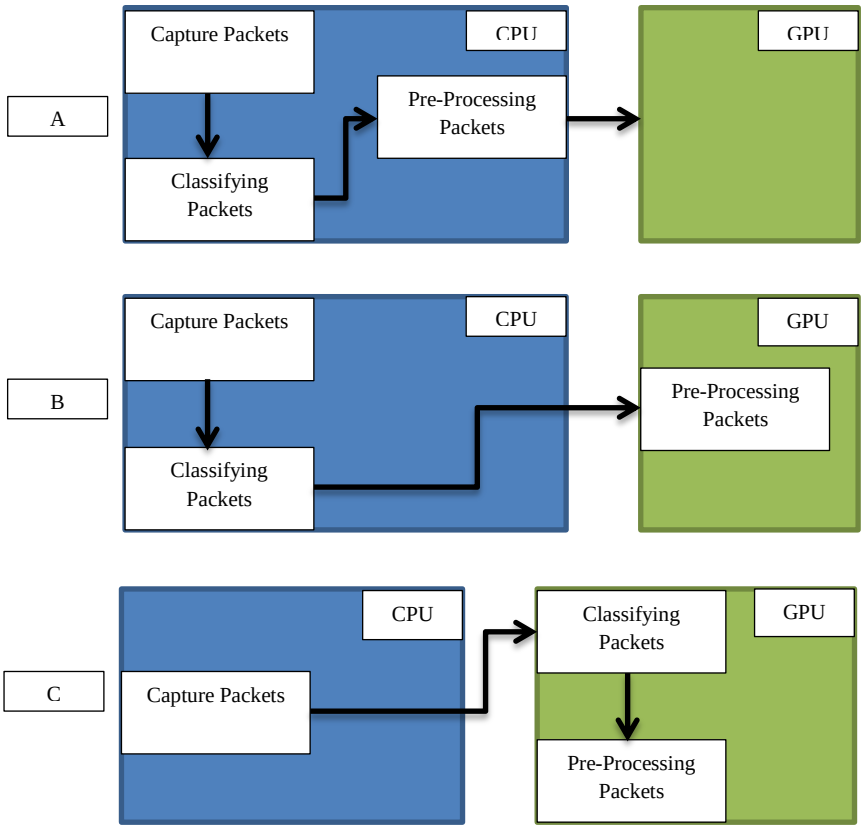

Figure 2-10 : Three options for pre-processing and classifying packets. Option A shows classification and pre-processing in the CPU, option B shows classification in the CPU and pre-processing in the GPU, option C shows both classification and pre-processing in the GPU.

Packet pre-processing changes the original packet by adding some meta-data or altering the original content. For an example, pre-processing stage might update a packet with the local timestamp or a session identifier. Like classification, pre-processing can also be done either in the CPU or the GPU. For an example, CPU or GPU pre-processing could be used for marking packet boundaries or updating packets with metadata. Figure 2-10 shows the three options available for executing packet classification and pre-processing. Please note that packet classification and pre-processing might be done in several stages. For an example an initial classification may identify packet types and a later classification which runs in the GPU, might extract useful information such as packet boundaries.

2.4.3 Transferring Packets Between GPU and CPU

The fundamental question when it comes to transferring data between the CPU and the GPU is the size of the data set and the frequency of transferring data. Throughout this section, we will discuss the available options for finding the correct data set size and the frequency of data transfer.

The first option is transferring data as they arrive [21] without buffering and the second option is transferring data in batches [23]. Both options have their own pros and cons. Transferring data to the GPU as they arrive without buffering has the following concerns.

1. Identifying whether all the data for a packet is available
   As stated in section 2.4.2, an application or a framework may decide to let the GPU does the packet boundary identification or have it done in the CPU. If the identification of a specific PDU boundary is a responsibility of the GPU, then we may transfer each packet to the GPU. If the packet is found to be incomplete, then the GPU kernel would notify the CPU to wait for the complete packet. But transferring each packet to the GPU incurs a considerable CPU to GPU data transfer overhead.

   If the boundary identification is done in the CPU, then we may transfer the complete packet to the GPU. Though this has a lower transfer overhead than transferring incomplete packets, it still incurs in a considerable transferring overhead.

2. The complexity of the decoding algorithm
   Let us assume that the algorithm supports data parallelism. With that if the per packet decoding time is roughly the same as the time taken to transfer a packet from the CPU to the GPU, one may try to justify the transferring of packets as they arrive. But the argument can only be made in terms of per packet decoding latency. Decoding one packet at a time would underutilize the GPU and the throughput would be degraded severely.

Second option is batch transferring. In batch transferring, packets are put into a buffer of fixed size as they arrive. Then either after the buffer is full or a timeout period, the complete batch gets transferred to the GPU to be decoded.

The most trivial advantage of batch transferring is the high throughput. Batch transferring achieves a high throughput by several means.

1. Increased GPU utilization by using more GPU cores for decoding than the option of transferring packets to the GPU as they arrive.
2. When having multiple packets decoded concurrently, it enables the GPU to manage memory more efficiently with techniques like memory coalescing.
3. Reduces the number of GPU kernel launches. kernel launches have a scheduling overhead.

As it has been mentioned before, the data transfer from the CPU to the GPU could also be triggered by a timeout. The reason for that is the inconsistent nature of the network traffic. If the application only transfers data upon reaching the maximum batch size, then at a low traffic time, it might take a long time to fill up the buffer. Thus the timeout is a safeguard to assure the quality of service of the decoder in terms of the throughput.

Another important aspect of transferring data is the use of pinned memory. To understand the importance of the pinned memory, it is essential to understand the mechanics behind the data transfer. When a CPU to GPU memory copy is issued on a non-pinned or pagable memory block, the CPU first copies the data from the pageable memory to a pinned memory and the DMA (Direct memory access) then takes over and copies the data to the GPU from the pinned memory. The CPU copies data from pageable to pinned memory to avoid paginations during the actual data transfer process. If a pinned memory is used in the first place, then the extra memory copying by the CPU from non-pinned to pinned memory could be avoided.

On the same note, allocation and de-allocation cost of pinned memory is high, so it is advisable to allocate a large pinned memory buffer beforehand and reuse it. That way we can avoid frequent pinned memory allocations and de-allocations. If the size of the buffer found to be not enough, then we may increase the size of it. Figure 2-11 shows the normal process of using non-pinned or pageable memory.

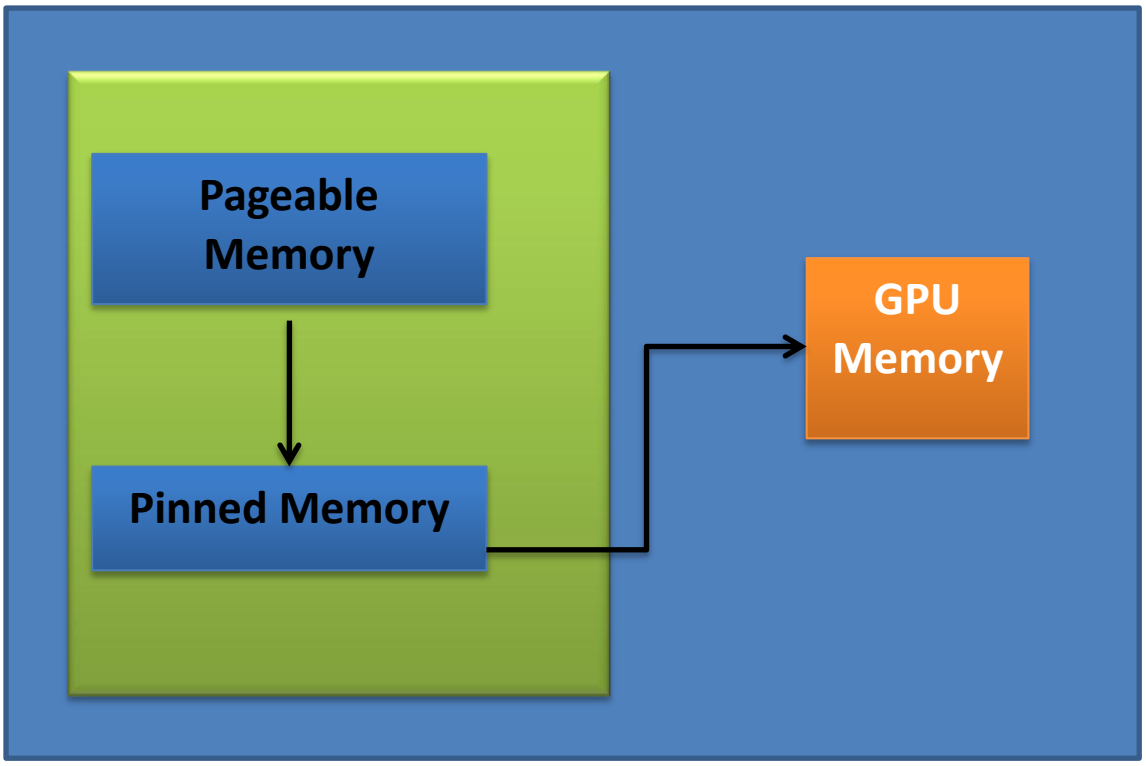

Figure 2-11 : Normal data flow from CPU to GPU

Figure 2-12 shows the use of pinned memory directly.

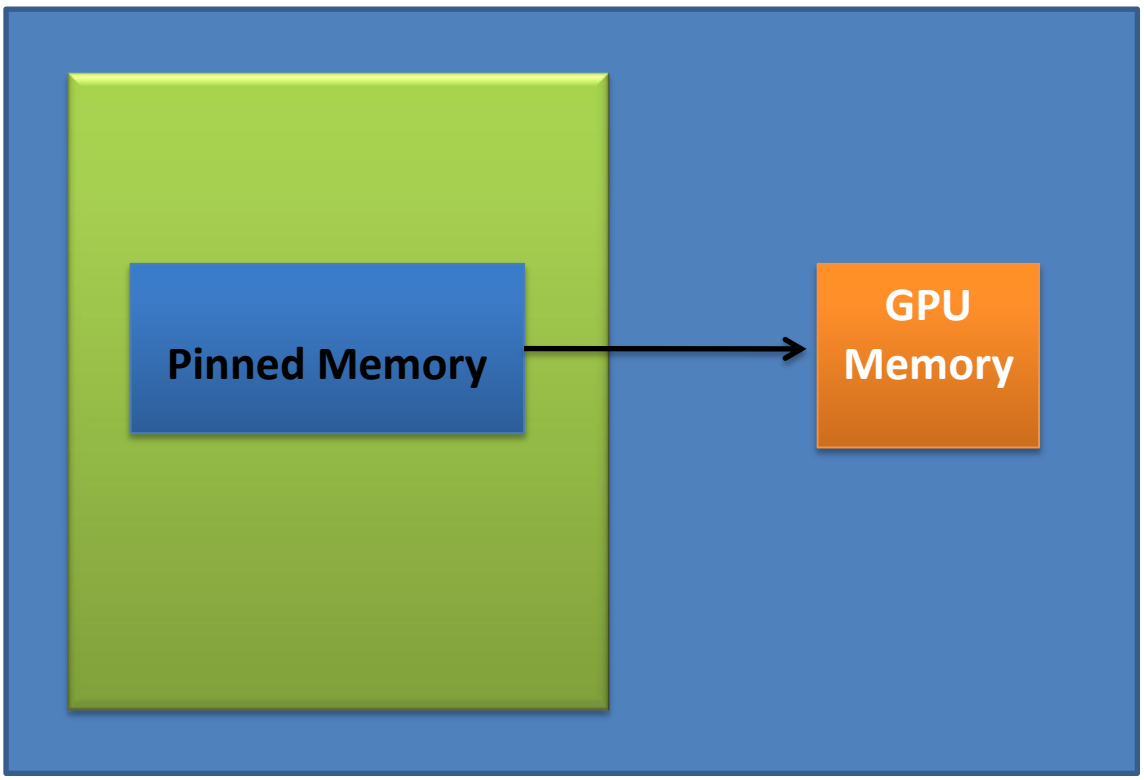

Figure 2-12 : Data flow from the CPU to the GPU when data stored in pinned memory

Use of Asynchronous Memory Transfer

A GPU based application has three main steps as discussed in section 2.3. Copying data from the CPU to the GPU (H2D or Host to Device), executing the kernel (K) in the GPU and lastly copying the results back to the CPU from the GPU (D2H or Device to Host). Some steps are optional though, like copying data from and to the CPU/GPU.

Though it might not be evident in a simple GPU based application, every aforementioned operation belongs to something called a *stream*. Please note that these streams are different from byte or data streams. A stream could be thought of as a sequence of operations that executes in the issued order.

Within a single stream, there could be multiple H2D, D2H and kernel executions and one of the main characteristics of a stream is that all operations issued within a stream are carried out in the insertion order. In other words operations inside a stream are synchronized.

If an operation does not have a stream assigned, then CUDA places it within the default stream. The default stream is a unique stream as it synchronizes other streams in the device. The default stream waits until all the operations issued before within other streams complete. All the CPU side operations issued after the default stream have to wait till default stream finishes execution.

Usually to get a performance gain through multiple streams, the framework needs to comply with at least one of the two scenarios below.

1. The most apparent one is the support for asynchronous batch execution. What it means is that the application should not block CPU threads till previous kernel cycles finish. Let us take an example. Think of a system with a kernel that processes PDUs in batches of 1000 elements. If the application does not support non-blocking kernels, then successive batch execution requests have to wait for previous executions to finish.
   Figure 2-13 shows the timeline diagram for the aforementioned scenario.

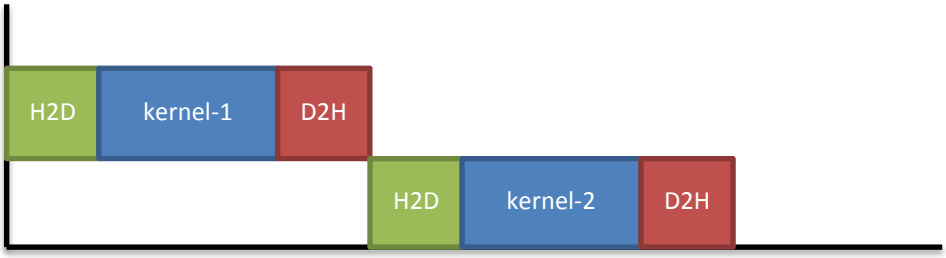

Figure 2-13 : Timeline diagram of synchronized decoding cycles

As we can see in the figure 2-13, each Host to Device data transfer and kernel cycle has to wait till the previous cycle finishes. This would be acceptable if each of the cycle utilizes all GPU resources. For an example if the kernel logic consumes all the cores for a batch of 1000 records, then executing kernel cycles sequentially would be acceptable. But most of the time, the kernel does not utilize much of the GPU resources and therefore decreases the efficiency. By using multiple streams, the application can issue kernel execution requests concurrently. The number of streams could be decided based on the resource availability.

2. The second scenario does not require non-blocking kernel execution. But it looks at the transfer delays of raw data to the GPU and processed data from the GPU to the CPU. If we assume a near linear or better relationship between the number of packets and the overall processing time, then by using two streams we could significantly reduce the overall processing time. The figure 2-14 shows how this works.

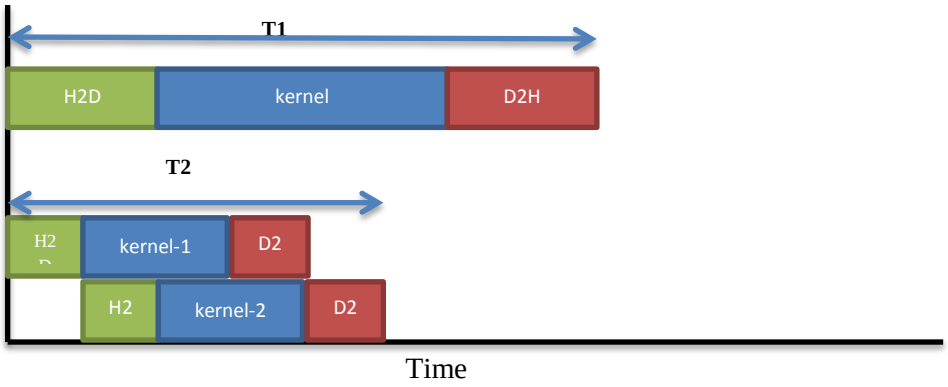

Figure 2-14 : Timeline diagram for using non-blocking execution cycles with multiple streams

2.4.4 Processing Packets in the GPU

2.4.4.1 GPU Concurrent Kernel Execution Patterns

In this section we will discuss several ways of using asynchronous memory transferring and streams and their effectiveness [24] .

A typical GPU contains two types of functional units, the execution and the copy engines. Each GPU contains one or more of each of the engine types. The most important aspect of

knowing the number of engines in a GPU is that the information is critical in deciding the execution pattern of streams.

Most of the GPUs today have either one of following engine configurations. One execution engine and two copy engines or one execution engine and one copy engine. Let us first discuss general execution patterns for each type. The first pattern is obviously to execute all sequentially.

Pattern – A

Pattern A uses only one stream and first it copies the data to the GPU from the CPU. Then it issues the kernel execution command and lastly copies the processed data back to the CPU from the CPU.

Pattern - B

```
For i <= number_of_streams :
    Copy_to_device (stream[i]);
    Issue_kernel_execution(stream[i]);
    Copy_from_device(steam[i]);
```

The execution pattern B issues all the operations for a given stream sequentially one after another.

Now the third execution pattern and we may call it the execution pattern C.

Pattern - C

```
For i <= number_of_streams :
    Copy_to_device (stream[i]);
For i <= number_of_streams :
    Issue_kernel_execution(stream[i]);
For i <= number_of_streams :
    Copy_from_device(steam[i]);
```

Pattern C issues Host to Device copy instructions for each stream first and then issues kernel executions for each stream and lastly issues Device to Host copy instructions for each stream. Both patterns B and C have pros and cons based on the number of copy and execution engines available. First let us see how a GPU with *one execution engine* and *one copy engine* works with the three patterns. For the example we may think of a *two stream* configuration.

Pattern comparison with single copy engine and single execution engine

Pattern A

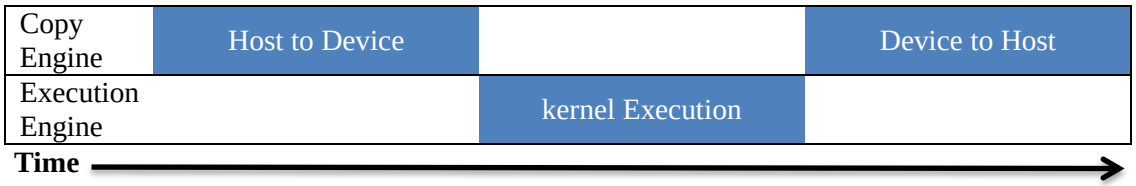

Figure 2-15 : Sequential execution pattern with one copy engine and one execution engine

Figure 2-15 illustrates the sequential execution of copying raw data to the GPU (host to device) and the execution of the kernel and lastly copying the processed data back to the CPU from the GPU (device to host). Following figure 2-16 and figure 2-17 illustrate the effect of using two streams on the overall processing time. The raw data has been divided equally between the two streams.

Pattern B

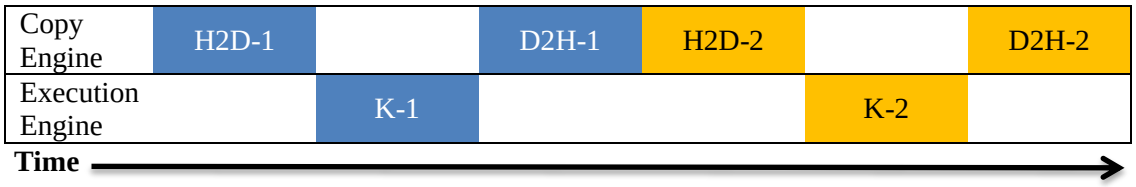

Figure 2-16 : Execution pattern B with one copy engine and one execution engine. The number of streams is two.

Figure 2-16 shows the effect on the overall execution time when we use two streams and follow the pattern B. As we may clearly see in the figure 2-16, the overall execution time has not improved compared to the pattern A (sequential execution).

Pattern C

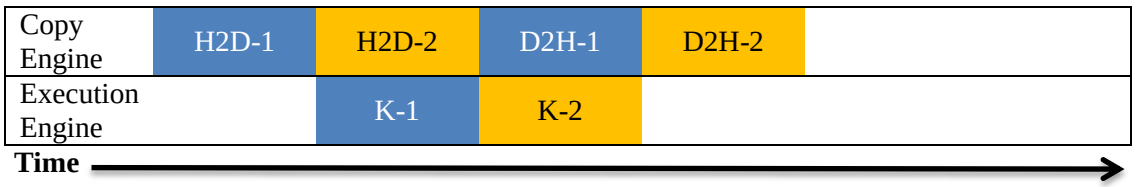

Figure 2-17 : Execution pattern C with one copy engine and one execution engine. The number of streams is two.

Figure 2-17 shows the effect on the overall execution time when we use two streams and follow the pattern B. Looking at the figure 2-17 and then figures 2-15 and 2-16, it is evident that for a GPU with only one copy engine and one execution engine the best option is to use the pattern C execution model.

Now we have seen how pattern A, B and C performs in a GPU with only *one copy engine* and *one execution engine*, next we will check how a GPU with *two copy engines* and *one execution engine* performs with each pattern. The two copy engines would be responsible for host to device and device to host data transferring respectively.

Pattern comparison with two (H2D and D2H) copy engines and one execution engine

Pattern A

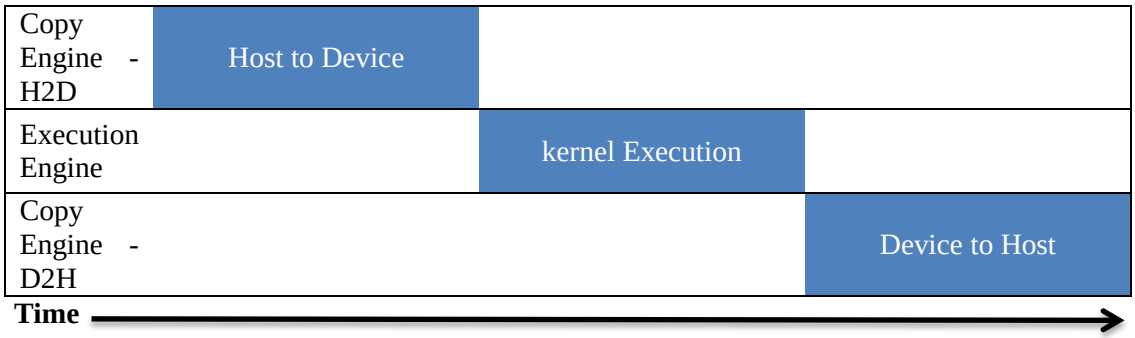

Figure 2-18 : Sequential execution pattern with two copy engines and one execution engine. The number of streams is two.

Figure 2-18 illustrates the sequential execution of copying raw data to the GPU (host to device) and the execution of the kernel and lastly copying the processed data back to the CPU from the GPU (device to host).

Pattern B

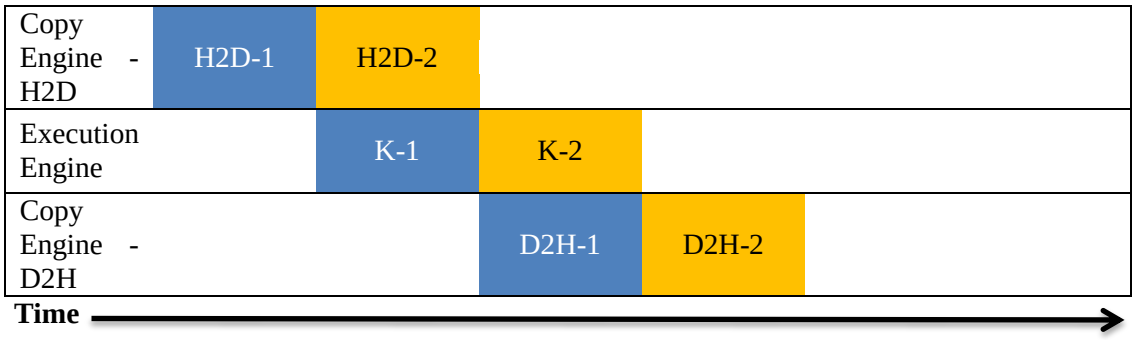

Figure 2-19 : Execution pattern B with two copy engines and one execution engine. The number of streams is two.

Figure 2-19 shows the effect of dividing data into two streams while utilizing two copy engines and one execution engine on the overall execution time. As it is shown in the figure 2-19, pattern B is more efficient than the pattern A in terms of the total execution time and the reason for that is the overlapping kernel executions and data transfers between the two streams.

Pattern C

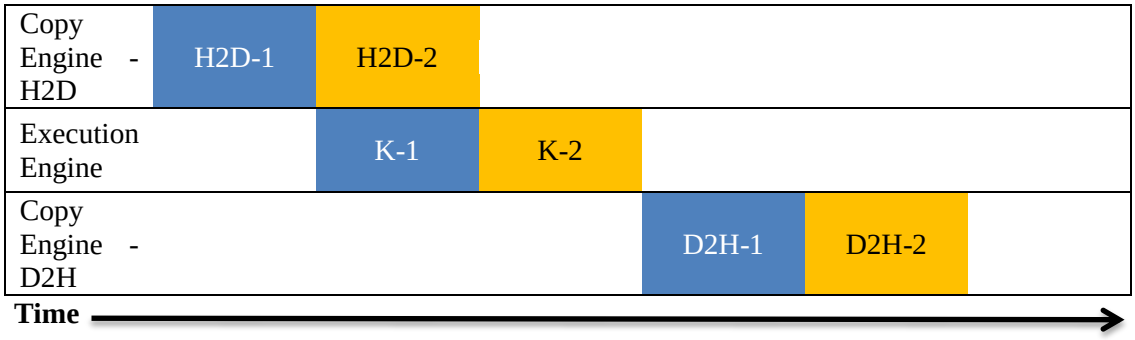

Figure 2-20 : Execution pattern C with two copy engines and one execution engine. The number of streams is two.

Looking at figures 2-18, 2-19 and 2-20, a GPU with two copy engines and one execution engine is better off using the pattern B. In the figure 2-20, it might look a bit strange that the D2H copy engines waits till the end of all kernel executions to start D2H transferring in pattern C. The reason for the delay is due to the behavior of the execution engine queue. We will discuss the reason after looking into the behavior of queues in copy and execution engines.

Now that we have discussed the behavior of three GPU execution patterns used in a multi-stream CUDA application, next we will discuss the behavior of *execution and copy engines queues*. Every copy and execution engine has a queue in which it stores copy and execution requests and dispatches them in the sequence they were issued. Thus the order of copy and execution requests matters.

Throughout the section we will discuss the impact of the copy and execution request order on the overall system performance. For the discussion we may consider a *two stream* configuration. Throughout the rest of the chapter we may use the following notation.

1. H2D{identifier}-{stream number} – Identifies a host to device data transfer.
2. D2H{identifier}-{stream number} – Identifies a device to host data transfer.
3. K{identifier}-{stream number} – Identifies a kernel execution.

The first stream configuration is as follows.

Stream-1 : H2Da-1, H2Db-1, K-1, D2H-1

Stream-2 : D2H-2

Figure 2-21 shows the execution order of each stream against a timeline. The order of the execution requests issued is as follows.

H2Da-1, H2Db-1, K-1, D2H-1, D2H-2

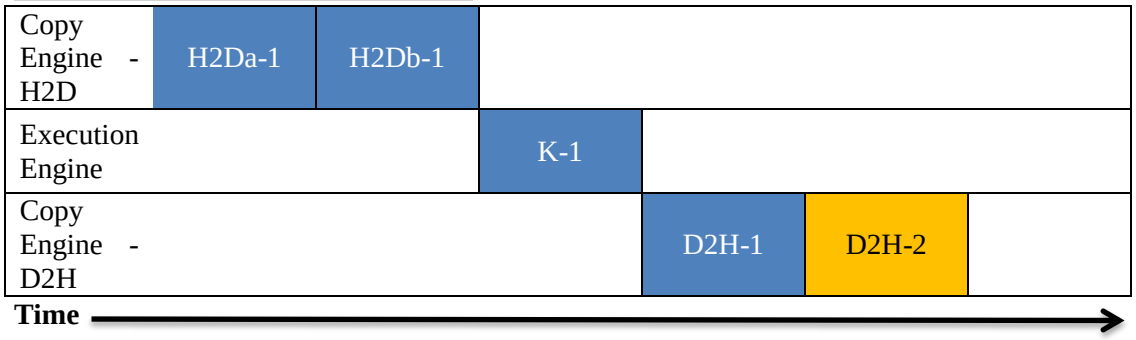

Figure 2-21 : Timeline diagram for the execution order for H2Da-1, H2Db-1, K-1, D2H-1, D2H-2 for two streams.

As it is shown in the figure 2-21, the Device to Host transfer of the second stream has to wait till the device to host transfer of the first stream finishes. The reason is that the device to host copy engine has its own queue and it only dispatches copy request in the insertion order.

But if we issue the device to host request of the second stream before start issuing requests of the first stream, then the second streams device to host request would be dispatched first. The resulted execution pattern is shown in the figure 2-22.

D2H-2, H2Da-1, H2Db-1, K-1, D2H-1

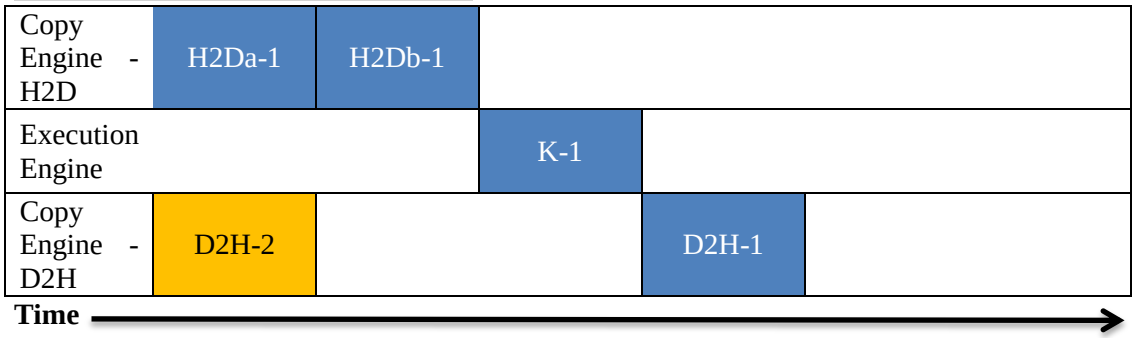

Figure 2-22 : Timeline diagram for the execution order for D2H-2, H2Da-1, H2Db-1, K-1, D2H-1 for two streams.

Figure 2-22 shows the effect of blocked queues. In the example, when issuing stream 1 requests, the D2H engine queue gets blocked until the H2D and kernel execution operations of the stream 1 finish. The reason is that though engines can dispatch requests concurrently, requests of a single stream get dispatched sequentially. Thus it is imperative to check the order of the dispatch requests.

In multi-stream GPU applications there will be situations in which we may have to use multiple kernels instead of using a single kernel. For an example a network packet processor may first run the classification kernel and then run the decoding kernel. Provided that multiple

kernels are used by an application, then the kernel dispatching order has a significant impact on the performance as well. For the next example we will consider *two streams and two kernels*. Each kernel would take up only half of the GPU resources thus enabling them to execute concurrently. The first dispatch order is as follows.

Stream-1 : Kernel-1 (Ka-1), Kernel-2 (Kb-1)
Stream-2 : Kernel-1 (Ka-2), Kernel-2 (Kb-2)

Kernel Dispatch Order : Ka-1, Kb-1, Ka-2, Kb-2

Note that the execution order of a given stream is sequential regardless of the resource availability.

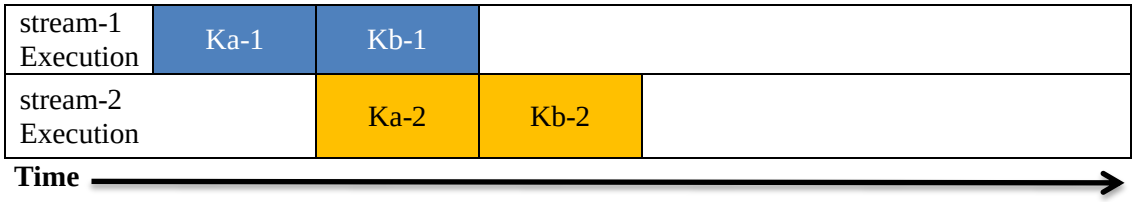

Figure 2-23 : Timeline diagram for the kernel execution order Ka-1, Kb-1, Ka-2, Kb-2 with two streams.

When looking at the figure 2-23, it is evident that during the Ka-1 execution the GPU has not been fully utilized. We may increase the GPU utilization by changing the order of kernel dispatch requests as shown in the figure 2-24.

Kernel Dispatch Order: Ka-1, Ka-2, Kb-1, Kb-2

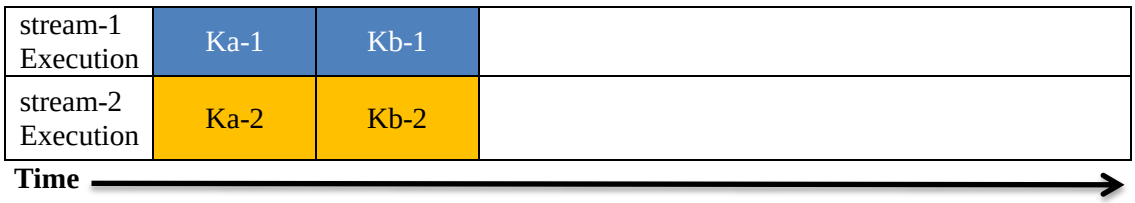

Figure 2-24 : Timeline diagram for the kernel execution order Ka-1, Ka-2, Kb-1, Kb-2 with two streams.

As shown in the figure 2-24, just by changing the order of execution we could cut the execution time by 1/3 and utilize the GPU 100% all the time.

But what if kernels are different in sizes in terms of the execution time? Let us consider 4 kernels of different sizes. Kernel Ka and Kc have the same execution time and kernel Kb and Kd have twice the execution time of Ka or Kc. Let us discuss four kernel execution patterns and their overall execution time. For our discussion we assume that the four kernels can run

independently of one another and every kernel irrespective of the execution time takes half of the GPU resources.

Kernel Dispatch Order: Kb-1, Ka-1, Kc-2, Kd-2

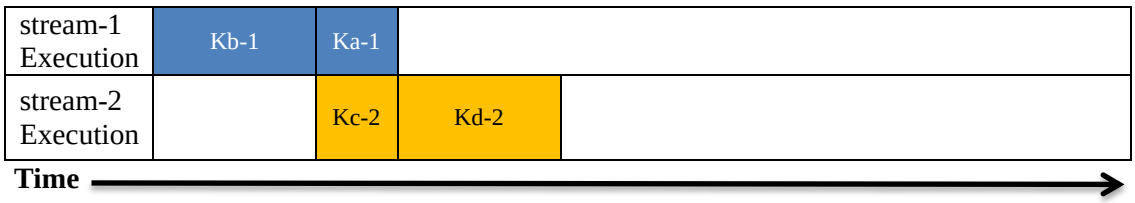

Figure 2-25 : Timeline diagram of the kernel execution order Kb-1, Ka-1, Kc-2, Kd-2. Kernel Kb and Kd have twice the execution time of Ka or Kc.

Figure 2-25 shows that though the GPU has enough resources to run Kc of the second stream, the execution engines waits till it dispatches both kernels of the first stream. The reason is that the execution engine queue always honors the insertion order and therefore Ka-1 has to be dispatched before dispatching the Kc-2. The reason for not running both Ka and Kb concurrently is that within the same stream, operations are carried out sequentially.
The overall execution time for this pattern is about 5 time units.

Kernel Dispatch Order: Kb-1, Kc-2, Ka-1, Kd-2

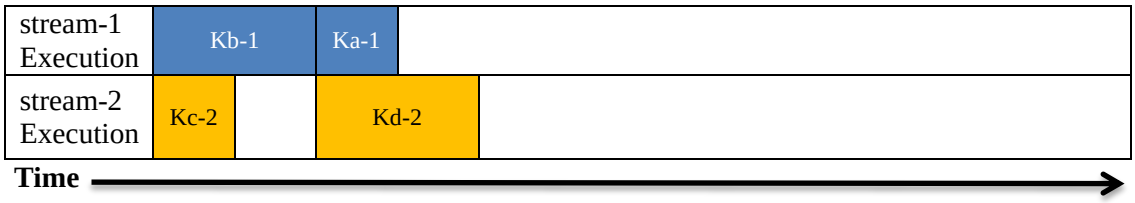

Figure 2-26 : Timeline diagram of the kernel execution order Kb-1, Kc-2, Ka-1, Kd-2. Kernel Kb and Kd have twice the execution time of Ka or Kc.

The execution pattern shown in the 2-26 shows the effect of blocking the execution queue by a stream on other streams. By the end of the execution of Kc, the GPU has enough resources to schedule Kd. But Ka issued on stream one is blocked from been dispatched till Kb ends running and thus blocks Kd from been executed. The overall execution time for this pattern is about 4 time units.

Kernel Dispatch Order: Kb-1, Kc-2, Kd-2, Ka-1

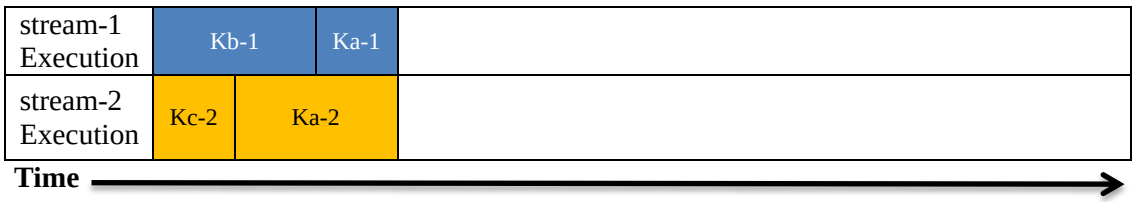

Figure 2-27 : Timeline diagram of the kernel execution order Kb-1, Kc-2, Kd-2, Ka-1. Kernel Kb and Kd have twice the execution time of Ka or Kc.

The pattern shown in figure 2-27 has significantly improved the execution time from 5 time units in figure 2-25 to 3 time units. The reason for the increased performance is that none of the streams are blocking potential kernel executions of other streams.

Figure 2-25, 2-26 and 2-27 illustrate the importance of the kernel dispatching order on overall performance when each kernel has different execution times. Next we will see how the dispatching order of kernels with different resource consumptions affects the overall execution time. For that the following kernel configuration would be used. Kernel Ka, Kb, Kc and Kd have the same execution time. Ka and Kc use 1/3 of the GPU resources and Kb and Kd use 2/3 of the resources. We will start with the pattern shown in the figure 2-27.

Kernel Dispatch Order: Kb-1, Kc-2, Kd-2, Ka-1

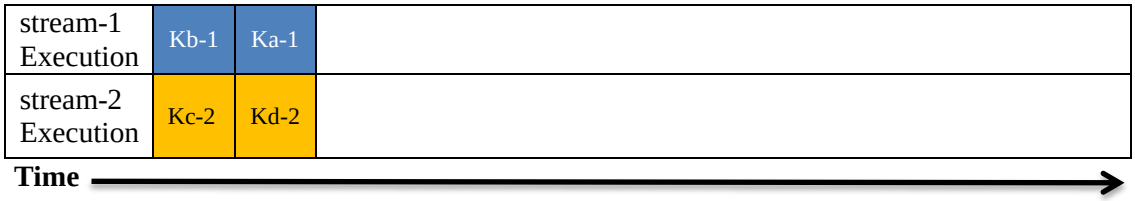

Figure 2-28 : Timeline diagram of the kernel execution order Kb-1, Kc-2, Kd-2, Ka-1. Kernel Kb and Kd consumes 2/3 of the GPU resources while kernel Ka and Kc consumes 1/3 of the GPU resources.

Looking at the figure 2-28 it is evident that the order of kernel dispatch has no major effect on the overall execution time provided that low resource consuming operations are interleaved with high resource consuming operations.

Next we will go a step further and see the performance impact of kernels with different execution times and resource consumptions. The execution time and resource consumption of each kernel would be as follows. Kernel Ka and Kc have the same execution time and kernel Kb and Kd have twice the execution time of Ka or Kc. In addition to that Ka and Kc consume 1/3 of the GPU resources and Kb and Kd consume 2/3 of the GPU resources. Let us start with the kernel dispatching order shown in the figure 2-27.

Kernel Dispatch Order: Kb-1, Kc-2, Kd-2, Ka-1

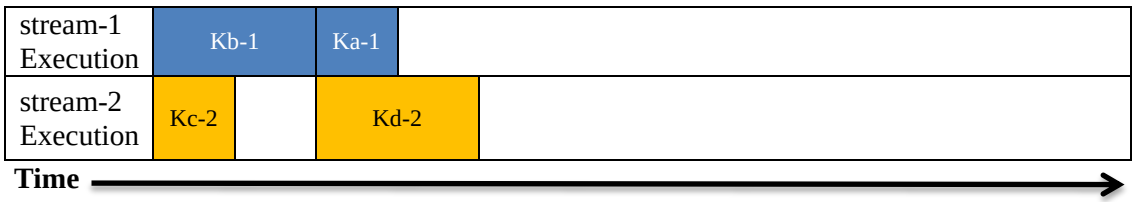

Figure 2-29 : Timeline diagram of the kernel execution order Ka-1, Kb-2, Ka-2, Kb-1. Kernel Kb and Kd consumes 2/3 of the GPU resources while kernel Ka and Kc consumes 1/3 of the GPU resources.

The figure 2-29 shows that the kernel dispatch sequence which yielded the best performance of 3 time units shown in the figure 2-27, now takes 4 time units to run. The reason is that Kb takes 2/3 of the GPU resources and thus even though stream one is not blocking stream two, Kd cannot start until Kb finishes.

Each execution pattern from the figure 2-23 to figure 2-29 illustrate the effect of kernel dispatching order on the overall performance. The patterns we discussed can be put into three categories.

First category is kernels having the same execution time and same resource consumption. Second category is kernels having the same resource consumption but different execution times. Third category is kernels having different resource consumptions as well as different execution times. Throughout all three categories, to gain the best performance, we have make sure that streams do not block each other from dispatching kernels and kernel dispatches from different streams are interleaved.

Now that we have discussed several important aspects of copy and execution engine queues, we may clarify the behavior seen with the execution pattern C with multiple streams. The behavior was that even though the kernel has finished execution for the stream-1, the GPU waits till all kernels are dispatched to start the D2H transfer for the stream-1. We may explain it in two steps. The GPU inserts a signal into the queues after each dispatch request to dispatch the next operation in the same stream. But when the execution engine is sequentially loaded with kernel dispatching requests, the signal gets delayed till the end of dispatching all kernels. This delay of issuing the signal causes D2H copy engine queue to block.

2.4.4.2 Thread Allocation in GPU

In sections 2.4.3 and 2.4.4.1 we discussed different kernel launch and memory transferring patterns and compared their efficiencies. The next question is how to organize and allocate threads in a GPU in an efficient manner. For the discussion we will consider a network packet processing application. We have two options to allocate threads. First is using *one thread per packet* and the second is using *multiple threads per packet*.

When using the first option which is one thread per packet, all threads in a block first load respective packets from the global memory to the shared memory of the block and then proceed with the actual data processing. The main concern with this method is the per packet processing latency. GPU cores are not designed to efficiently handle complex instructions

such as branching and looping. Therefore a fairly complex kernel logic would take a longer time to process a single packet than a CPU core and thus increases the per packet latency [25]. The advantage of this option is that if the application processes a large number of packets in parallel, the throughput will be higher. But for achieving the high throughput, we need to make sure to utilize GPU cores at a higher percentage.

When using the second option which is multiple threads per packet, the content of a packet would be separated into different sections and each section would be processed by a separate thread in a block. For an example, one thread would be processing from the 0^{th} byte to 100^{th} byte and another thread would be processing from 100^{th} byte to 200^{th} byte. This option is viable only if the packet itself is data parallelizable. Another concern is that the number of packets get processed by a block is lower compared to the one thread per packet option. Therefore it lowers the throughput but because of using multiple threads to process a single packet, it also lowers the per packet latency. As a summarization, the first option is better for applications aiming for high throughput and the second option is better for low latency focused applications.

Another point to remember is that with GPUs, it is best advised to not use complex data structures. The reason is that complex data structures are not localized in memory and result in un-coalesced memory access patterns. In GPUs, localization of data is important to gain performance.

2.4.4.3 Dynamic Parallelism

The data processing flow of an application can be broken down into one or more logical steps. For an example, the packet processing flow of a network packet processing application can be broken into two logical steps. First step is finding and marking boundaries of each packet and the second step is processing the packets based on marked boundaries. The Logic is shown in the figure 2-30.

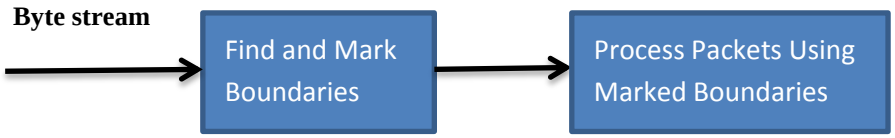

Figure 2-30 : The SMPP PDU decoding process flow diagram

These steps can be either fully executed in the CPU or can be splitted between the GPU and the CPU. When breaking down and allocating steps to either the CPU or the GPU, we need to take two points into consideration.

1. Steps running on the GPU need to be data parallelizable to take advantage of the GPU architecture.
2. The execution order of steps and their inter dependencies.

The first point is obvious with what we have discussed so far in chapter 2.4. Without been data parallelizable, it is hard to achieve any performance gain by using the GPU. To explain the second point, let us consider a data processor with three steps. Throughout the section 2.4.4.3 we will discuss two data flow patterns.

The first data flow that we will discuss is as follows. *Step-1 and step-3 are massively data parallelizable and step-2 is not data parallelizable*.

Step-1(Parallelizable) -> Step-2(Not Parallelizable) -> Step-3(Parallelizable)

Now let us see how we may efficiently allocate each one of these steps to either the CPU or the GPU. The obvious solution is offloading step-1 and step-3 to a GPU as they are data parallelizable. But doing so does not always yield a performance gain and we have to take few factors into consideration. First factor is the data transferring delays associated with crossing the CPU/GPU boundary. We have to cross the CPU/GPU boundary *four times* (copying to the GPU from the CPU and copying from the GPU to the CPU) if we choose to use the GPU for step-1 and step-3. Therefore when we are presented with data flows in which parallelizable and non-parallelizable steps are interleaved, first a performance evaluation should be done for each step. Based on the performance evaluation we may then decide on which step to be offloaded to the GPU.
The second data flow pattern is as follows. *Step 1 is non-parallelizable and step-2 and step-3 are parallelizable.*

Step-1(Not Parallelizable) -> Step-2(Parallelizable) -> Step-3(Parallelizable)

With the second data flow pattern, the second and third steps both are massively parallelizable. Therefore if we run them both in the GPU and follow the conventional GPU execution model, CPU and GPU boundary would be crossed four times. First two times, the boundary would be crossed to transfer data for the step-2 and last two times to transfer data for the step-3. As both second and third step are executed in the GPU, what if we can schedule the third step from the GPU without involving the CPU.

With *dynamic parallelism* [26] [27] [28] we may reduce the number of CPU/GPU boundary crossings from four to two. By using dynamic parallelism a kernel launched by the CPU can launch child kernels from the GPU. The concept of *dynamic parallelism* is as follows.

A CUDA kernel can launch other kernels from the GPU. Once a secondary (*child*) kernel is launched, the CUDA runtime implicitly synchronizes the parent kernel execution with the secondary kernel execution. Simply putting it, the CUDA runtime ensures that the parent kernel waits till the secondary kernel finishes executing. Figure 2-31 illustrates the relationship between parent and child kernels.

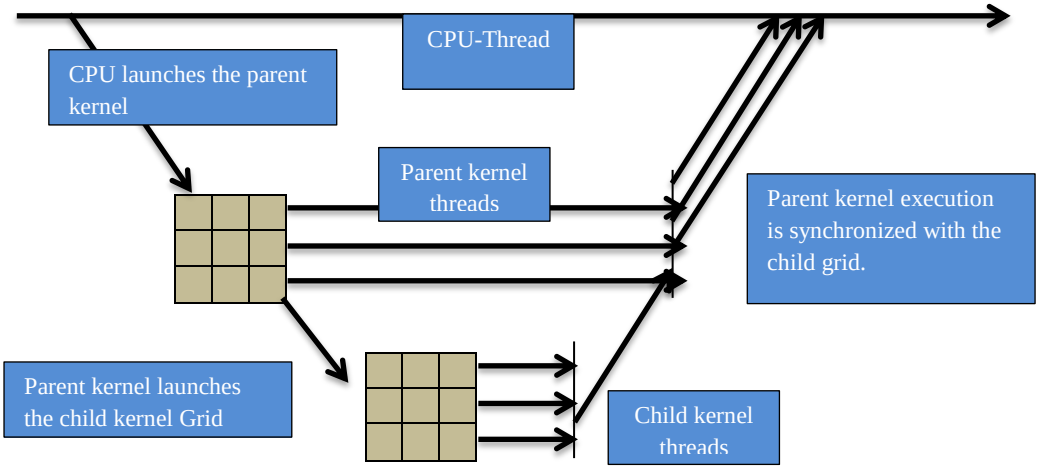

Figure 2-31 : Dynamic Parallelism parent kernel and child kernel execution model

Apart from reducing the overhead of having to transfer data back and forth between the CPU and the GPU, dynamic parallelism has few other advantages over normal execution patterns.

1. The device algorithm (Algorithm that runs on the GPU) can be written without having to fit it into a flat single level structure. Dynamic parallelism enables recursions and simplifies the designing process.

2. Enables managing GPU resources at a more granular level. For an example, a single level processing of an image enhancement would allocate the same level of resources throughout the image grid. Whereas with dynamic parallelism, resources could be allocated based on the intensity of different parts of the image.

3 Implementation

3.1 Introduction

The implementation consists of four primary components. First component is the SMPP decoder library written in C language. Second component is the integration with the Java based SMPP framework Cloudhopper. Third component is the SMPP traffic simulator written in Java. The fourth component is a performance tuner written in C. Following figure 3-1 illustrates how these components are related.

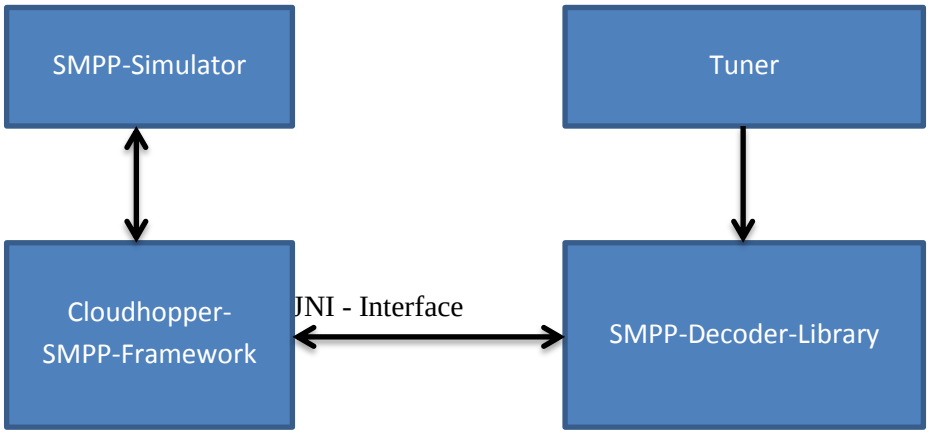

Figure 3-1 : Primary components in the SMPP decoder library

3.2 Cloudhopper Changes

We have used the Cloudhopper SMPP framework for managing TCP connections and SMPP sessions. The framework does not support GPU based decoding and therefore we did few design modifications. Figure 3-5 shows the original design for decoding SMPP packets.

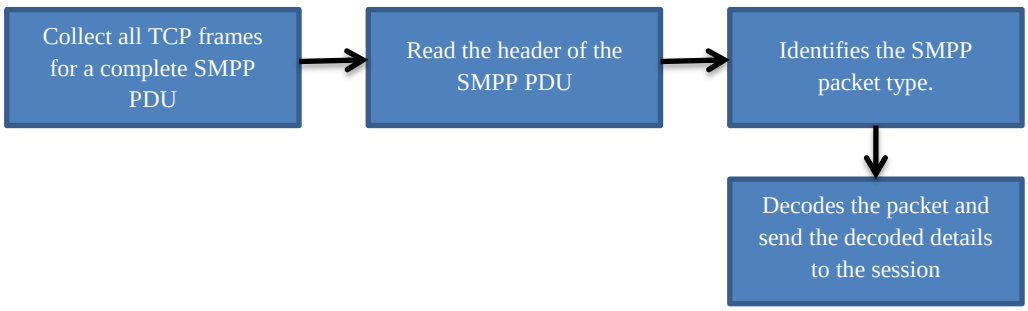

Figure 3-2 : Cloudhopper SMPP PDU original decoding flow

Figure 3-6 shows the altered Cloudhopper decoding flow integrated with the SMPP decoder library.

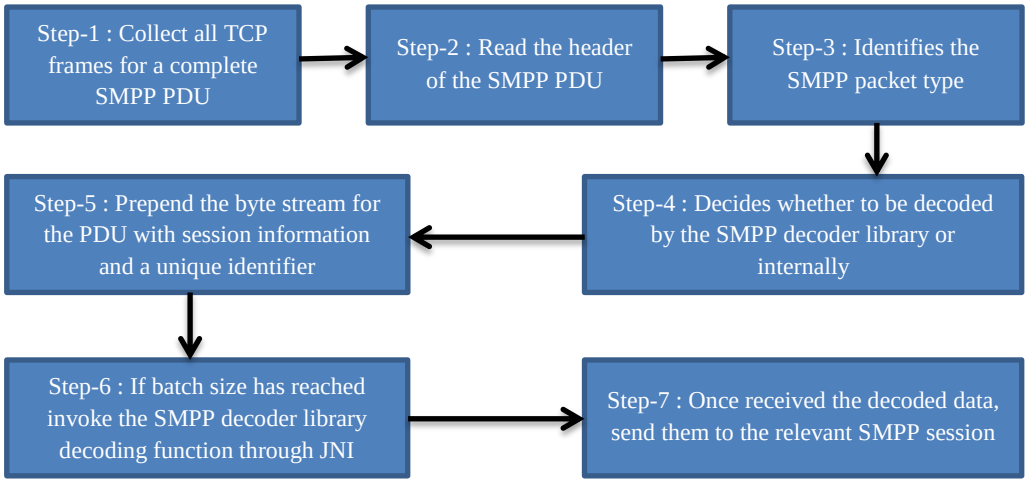

Figure 3-3 : The altered design with SMPP decoder library integration

Figure 3-6 illustrates the changes made to the Cloudhopper original decoding flow. Step-1, step-2, step-3 and step-4 are packet classification stages (section 2.4.2) whereas the step-5 represents a packet pre-processing stage (section 2.4.2). The main alterations are the introduction of a batch of packets and updating each packet with session information and boundary markers and lastly invoking the SMPP decoder library through JNI.

3.3 SMPP Decoder Library

The SMPP Decoder library provides following key features through an API.

1. GPU based SMPP decoding using *dynamic parallelism*.
2. GPU based SMPP decoding without using dynamic parallelism.
3. CPU based SMPP decoding.
4. Performance tuner.

The library implementation can be separated into five components.

1. A JNI interface to communicate with the Cloudhopper framework.
2. An interface with methods to either tune or perform normal SMPP decoding. The interface supports both GPU or CPU based decoding.
3. The GPU based decoder implementation in CUDA.
4. The CPU based decoder implementation.
5. A set of utility methods to manipulate a stream of bytes.

Throughout the sub-sections we would discuss each step in detail.

3.3.1 JNI Interface

As it has been mentioned before, the SMPP decoder library is written in C language
And the Cloudhopper SMPP framework is written in Java. The Java based Cloudhopper runs on a JVM (Java Virtual Machine) and the data structures and language semantics are different from C. Thus, for the Cloudhopper services to access the SMPP decoder functionalities, we need an adapter between the two platforms. Java comes with a standard implementation to communicate with native implementations called JNI (Java Native Interface).

JNI offers a set of primitive data types to match Java primitive data types. Apart from the primitive data types, JNI defines jobject, jclass, jstring, jthrowable, jarray data structures synonymous with Java counterparts. A native function accepts data as arguments of aforementioned data types and also returns results in one of the aforementioned data types.

When a Java application needs to invoke native methods such as SMPP decoder functionalities, it first has to generate a header file with required methods with arguments and return types. Here follows the three functions generated for the SMPP decoder JNI interface and their descriptions.

```
JNIEXPORT void JNICALL _initialize (
        JNIEnv *env, jobject thisObj, jstring configurationFilePath)
```

The first function is to initialize the library. Calling native functions through JNI is a heavy task. Creating new objects or calling a method of a Java object or accessing attributes of a Java object requires accessing the JVM from the native code. For an example to call a method of a Java object, the native code should first acquire the equivalent JNI structure for the method which is called *jmethod*. It is same for Java classes and attributes as well. This process takes time and it is advised to cache those frequently used JNI data structures. The initialize method caches *jmethods*, *jclasses* and *jfields*. Apart from caching, the method configures the core decoder implementations as well.

```
JNIEXPORT jobject JNICALL _decodePDUDirect (
        JNIEnv *env, jobject thisObj, jobject pduContainerBuffer, jint size, jint correlationIdLength)
```

The second function is the most important one. It acts as an adapter for the SMPP decoder core API functions. The most important arguments are the *pduContainerBuffer*, *size* and the *correlationIdLength*.
The *pduContainerBuffer* is an off heap byte buffer containing a stream of PDUs. At earlier stages of the SMPP decoder development, the off heap byte buffer was not used. Instead a

List of byte buffers was passed as a parameter. This approach was time consuming for two reasons.

First is that when passing a Java object allocated in the heap through JNI, the JVM has to first copy it into a non-heap memory location and then copy it to the relevant location. By using off heap byte buffers we can reduce the number of copy operations to one.

The second reason is that, to access each PDU data element in a list, the native implementation has to use Java method. Calling Java methods per element is time consuming as each call has to cross the JNI boundary.

By using an off heap byte buffer (The name used in Java is DirectByteBuffer) we can eliminate aforementioned concerns. All we have to do is to allocate an off heap byte buffer and then write to it and pass the pointer for that buffer as a method argument.

The *correlationIdLength* is the length of the unique id used to identify each individual packet. The SMPP decoder library allows attaching a unique id for each packet. The id would be sent back with the decoded packet which can then be used to identify the SMPP session to which the packet belongs. The *size* parameter represents the length of the byte buffer.

```
JNIEXPORT void JNICALL _startTuner(
        JNIEnv *env, jobject thisObj, jobject pduContainerBuffer, jint size, jint correlationIdLength)
```

The SMPP decoder performance depends on several factors. To name a few, the type of the GPU, number of CPU cores and number of GPU cores, block and grid configurations, the batch size etc. Therefore the library provides a build-in performance tuner. The *startTuner* method could be used to run the tuner. The parameters are the same as the *decodePduDirect* method.

Another important feature is that we can set a threshold value for the batch size through a configuration. If it is set to a greater than zero value, then the GPU would be used for decoding only for batches larger than the threshold. If the threshold value is zero, then a compilation time flag would decide between CPU and GPU decoder implementations.

3.3.2 SMPP Decoder Library Interface

This is the primary API provided by the SMPP decoder library and following are the available methods.

1. *CudaDecodedContext *decodeGpu(DecoderMetadata decoderMetadata)*
2. *CudaDecodedContext *decodeGpuDynamic(DecoderMetadata decoderMetadata)*
3. *DecodedContext *decodePthread(DecoderMetadata decoderMetadata)*
4. *void startPerfTuner(DecoderMetadata decoderMetadata)*
5. *void startPerfTunerPthread(DecoderMetadata decoderMetadata)*

Before start discussing aforementioned methods figure 3-2 depicts the format that the SMPP decoder expects each PDU to confirm to.

Correlation-Id	PDU-Content	PDU-End-Indicator

Figure 3-4 : Structure of the decodable element accepted by the API

First part is the Correlation-Id and the second part is the actual PDU content and the last part contains a byte pattern to mark the end of the PDU. Every method uses one input parameter of the type DecoderMetadata. DecoderMetadata is a C structure and contains several parameters including the byte buffer for the batch of PDUs, the length of the byte buffer, the correlation id length and the batch size.

*CudaDecodedContext *decodeGpu(DecoderMetadata decoderMetadata)*

The method decodes SMPP PDUs using the GPU without utilizing Dynamic Parallelism. The flow of the execution is as follows.

1. Creates a C structure (ByteBufferContext) to hold the pointer to the PDU byte buffer as well as the complete length of the buffer and the read index of the buffer. The structure would be used throughout the decoding process.

2. Next by using the received byte buffer, the method creates an array of C structures (CudaPduContext) to hold start and end positions and the correlation id of each PDU.

3. Next we create another array of C structures (CudaDecodedContext) to hold the decoded PDUs.

4. Lastly we create another C structure (CudaMetaData) which holds the preferred block and grid dimensions, the byte buffer of PDUs and pre-processed PDU details and the array of structures for holding the decoded PDUs. This structure would then be sent to the GPU to get decoded.

*CudaDecodedContext *decodeGpuDynamic(DecoderMetadata decoderMetadata)*

The method decodes PDUs using the GPU and utilizes Dynamic Parallelism provided with CUDA. The steps of decoding are different from the normal GPU based decoding process.

1. Creates an array of C structures (CudaDecodedContext) to hold decoded PDUs.

2. Creates a CudaMetaData C structure which contains the initial block and grid dimensions, the byte buffer of PDUs. The main thing to notice here is that the pre-processing of the provided byte buffers happens in the GPU as opposed to the normal GPU based decoding option.

3. Next invokes the Dynamic Parallelism based GPU decoding process.

*DecodedContext *decodePthread(DecoderMetadata decoderMetadata)*

The method decodes PDUs only using the CPU. The steps are almost same as the normal GPU based decoding process.

1. Creates a C structure (ByteBufferContext) to hold the pointer to the PDU byte buffer, the complete length of the buffer and the read index of the buffer. The structure would be used throughout the decoding process.

2. Next by using the received byte buffer creates an array of C structures (DirectPduContext) with the start and end positions of PDUs and a pointer to the byte buffer.

3. Creates a predefined number of Pthreads and allocates each thread a part of the PDU list to decode. The only difference with the normal GPU based decoding method is that the Pthread version does the decoding in the CPU instead of the GPU.

startPerfTuner(DecoderMetadata decoderMetadata)

The method is used to find the best GPU grid and block configurations for a given batch size. We may configure a set of block and grid sizes and batch sizes through a configuration file. The method then calculates the average throughput and the latency for each configuration and logs the result to a file.

startPerfTunerPthread(DecoderMetadata decoderMetadata)

Same as the startPerfTuner, the method can be used to calibrate the CPU thread count for different batch sizes. Just like the startPerfTuner method, startPerfTunerPthread generates a result file with average throughput and latency for each CPU thread configuration with different batch sizes. These measurements can be used to find the best CPU thread configuration for a given batch size for different machines.

3.3.3 The Decoder Implementation

The SMPP decoding process can be split into two steps. We will referrer the first step as pre-decoding in which boundaries and lengths of each packet would be identified and extracted into a set of C structures. Final step is executing the actual decoding logic based on the information from the first step. The library offers three different options for performing the pre-decoding and the decoding steps. Please note that the pre-decoding is a packet classification step (section 2.4.2) in which we extract useful information from packets without altering the content.

The first option executes both pre-decoding and decoding steps in the CPU. The second option performs pre-decoding in the CPU and the decoding is done in the GPU. The third option executes both pre-decoding and decoding steps in the GPU.

Running Pre-Decoding and Decoding Steps in the CPU

This option allows both the pre-decoding and decoding to be carried out in the CPU. The library accepts packets in batches and if the batch size is small enough, the throughput gain in using GPU might get overshadowed by the overheads associated with GPU processing.

Therefore for smaller batch sizes, CPU based decoding yields the best performance. The CPU based decoder has used *Pthread* in both pre-decoding and decoding steps. The number of CPU threads is configurable.

Running Pre-Decoding and Decoding Steps in the GPU

The common way of performing both pre-decoding and decoding in the GPU is as follows. First we copy the byte stream to the GPU. Then the pre-decoding kernel calculates metadata for each packet. Next the metadata would be copied back to the CPU. Finally the decoding kernel is executed.

With this approach, the application crosses the CPU/GPU boundary four times. Therefore to reduce the number of data movements between the CPU and the GPU, we have used CUDA *dynamic parallelism* (see section 2.4.4). With dynamic parallelism, once the pre-decoding kernel finishes executing, a child kernel with the decoding logic would be scheduled by the respective pre-decoding kernel. Following figure 3-3 illustrates the decoding flow.

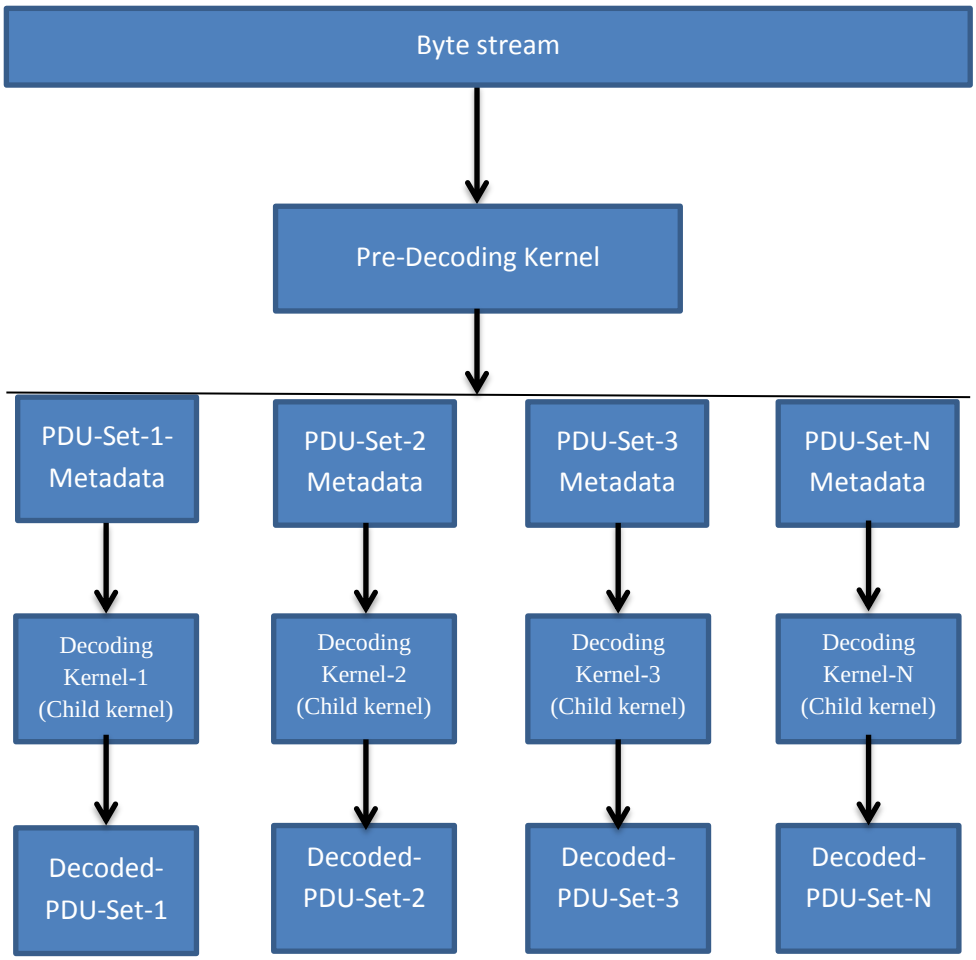

Figure 3-5 : Dynamic Parallelism based SMPP decoding flow

Each thread of the parent kernel (pre-decoding) iterates through a part of the byte stream looking for packet boundaries. Once a packet boundary is identified, the thread creates and populates a metadata structure with the start and end indexes. Once every thread in a block completes pre-decoding, the first thread of the block schedules a child kernel with the actual decoding logic. As discussed in section 2.4.4, the parent block waits till the child kernel finishes.

Running Pre-Decoding Step in the CPU and Decoding Step in the GPU

Pre-decoding at the CPU calculates metadata for each packet. Metadata contains the start and end indexes of a packet and a pointer to the byte stream. After calculating metadata for each packet, application dispatches the decoding kernel. The number of streams to be used is a configuration. In the section 2.4.4.1 we introduced three multi-stream kernel execution patterns. Pattern B is recommended for GPUs with only one copy engine and one execution engine. Pattern C is recommended for GPUs with multiple copy engines and one execution engine. *Therefore the library checks for the number of execution and copy engines first and then decides on which pattern to be used*. In terms of the development effort, pre-decoding in the CPU is much easier than the GPU based pre-decoding. Both pre-decoding and decoding logics are much simpler in the CPU version. Following figure 3-4 illustrates the data flow.

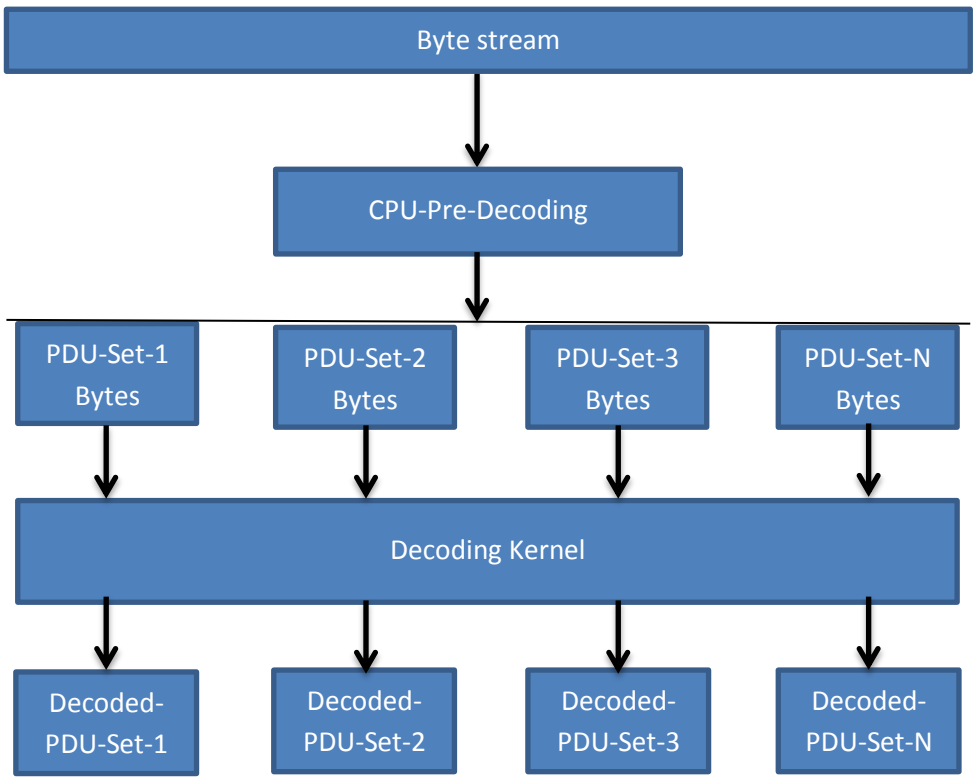

Figure 3-6 : SMPP decoding flow with CPU based pre-decoding and GPU based decoding

The SMPP decoder library supports all three pre-decoding and decoding methods (CPU based pre-decoding and decoding, GPU based pre-decoding and decoding and CPU based pre-decoding and GPU based decoding) and the user may decide on the specific method based on throughput requirements and network traffic patterns.

The GPU Thread Allocation

As discussed in section 2.4.4.2, we have two options for allocating thread in the GPU. First is to use one thread per packet and the second is to use multiple threads or a thread block for a single packet. The first options yields better throughput and the second option yields lower latency. The SMPP decoder is focused on the throughput and therefore we are using one thread per packet. The implementation distributes the batch of packets among the available threads and each thread processes one or more packets. But a single packet is always processed by a single thread.

3.3.4 Utility methods for reading a Byte stream

The SMPP decoder library implements methods for extracting following data types from the byte stream. *Unsigned 8 bit Integer*, *Unsigned 16 bit Integer*, *Unsigned 32 bit Integer*, *NULL terminated character array (String)*, *byte array of a given size*. Every method takes a *ByteBufferContext* C structure as one of the input parameter and increment the read index at the end of a read accordingly.

4 Evaluation

4.1 Workload Generation

Our implementation is focused on improving the throughput of SMPP decoding. Therefore the workload is not focused on traffic patterns such as variations in TPS over time but mostly on the structure and size of PDUs. One of the major issues we faced was that though there are open source SMPP frameworks, there were no standard SMPP simulators. Therefore we wrote our own simulator.

The simulator is capable of generating SMPP PDUs of different types and lengths. It supports mixing of PDUs of different lengths. For the experiment we have used PDUs of SUBMIT_SM type only. SUBMIT_SM and DELIVER_SM are structurally identical except for the COMMAND_ID header. On top of that those two are the most widely used PDU types in SMPP applications. Following are the compositions of different batch sizes that we have used.

1. For 1000 batch – 1000 SUBMIT_SMs. Each PDU had different destination and source addresses and different messages of 20 characters. Experiments were carried out with 5 optional (TLV) parameters.

2. For 10,000 batch –SUBMIT_SMs with different source and destination addresses and different message of 20 characters. Experiments were carried out with 5 optional (TLV) parameters. The simulator used two SMPP sessions with each sending 5000 PDUs. Each session fired at a TPS of 500. Thus it took 5 seconds to fill the buffer.

3. For 100,000 batch –SUBMIT_SMs with different source and destination addresses and different message of 20 characters. Experiments were carried out with 5 optional (TLV) parameters. The simulator used four SMPP sessions with each sending 25,000 PDUs. Each session fired at a TPS of 1000. Thus it took 25 seconds to fill the buffer.

4. For 1,000,000 batch –SUBMIT_SMs with different source and destination addresses and different message of 20 characters. Experiments were carried out with 5 optional (TLV) parameters. The simulator used ten SMPP sessions with each sending 100,000 PDUs. Each session fired at a TPS of 2000. Thus it took 50 seconds to fill the buffer.

4.2 Experiment Setup

Following are the GPU, CPU, Operating System and Memory configurations used.

1. GPU - GeForce GTX 750 Ti GPU
2. Intel Core i7 – 4790 CPU
3. Operating System - Ubuntu 16.04 LTE Release
4. Memory - 16GB RAM

The SMPP Simulator has been allocated 1024 MB of maximum heap size and 512 MB of initial heap size. The SMPP Server with the GPU accelerated decoder has been allocated 4096 MB of maximum heap size and 1024 MB of initial heap size.

4.3 Analysis

4.3.1 Maximum Throughput Analysis for Different Batch Sizes

Following figure 4-1 illustrates the maximum throughput for a set of batch sizes. For each batch size, the throughput has been given for Java decoding mode with 4 threads, Pthread decoding mode with 4 threads, GPU normal execution mode and GPU Dynamic Parallelism execution mode.

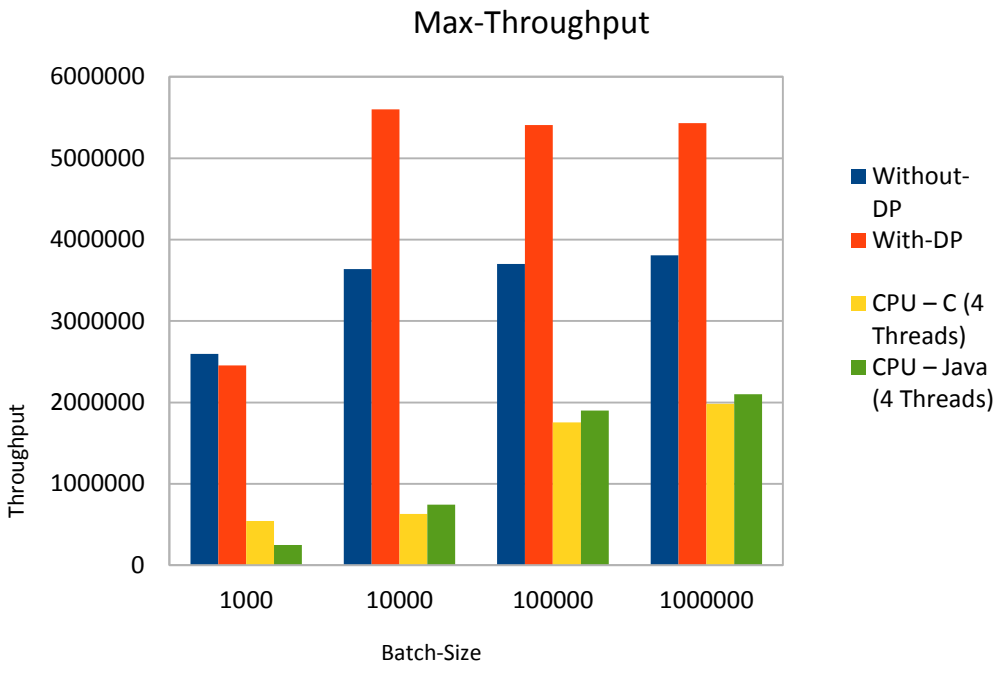

Figure 4-1: Max throughput analysis for four decoding modes and four batch sizes

Looking at the figure 4-1, it is evident that both normal and dynamic parallelism based GPU implementations have higher throughputs than the Java multi-thread or C multi-thread decoder implementations. The performance gain of GPU over CPU based implementations (Java or C) is nearly 5 times for batch sizes around 10,000. Apart from the obvious performance gain, we may make two key observations.

1. Dynamic parallelism based GPU implementation has better performance for large batch sizes.
2. Normal GPU based implementation is faster for small batch sizes like 1000.

Let us discuss why the system behaves like that. As it has been discussed before, the SMPP decoder logical flow can be divided into two parts. First part is pre-processing and the second part is the actual decoding.

With the dynamic parallelism based implementation, both stages (pre-processing and decoding) are carried out in the GPU. The parent kernel first pre-processes the byte stream and marks boundaries of each PDU. Finally each parent kernel block launches a child kernel to carry out the decoding logic. The grid and block sizes for the child kernel are based on the number of PDU boundaries marked by the parent kernel block.

With the normal multi stream implementation, first the pre-processing is performed in the CPU. Then the actual decoding logic is offloaded to the GPU. The performance difference that we observed is mostly due to the implementation of the pre-processing stage. With lower batch sizes such as 1000, the time taken by the CPU to mark the boundaries is less than the time taken by the GPU. It is mostly due to the overheads associated with launching parent and child kernels and also the logic at the GPU for marking boundaries is more complex than the CPU version. Thus the computational advantage of the GPU is not evident with low batch sizes.

When the batch size increases, the overhead associated with the GPU bound pre-processing becomes insignificant and provides a significant performance gain over a normal GPU based implementation.

4.3.2 Throughput Analysis of Dynamic Parallelism GPU Mode

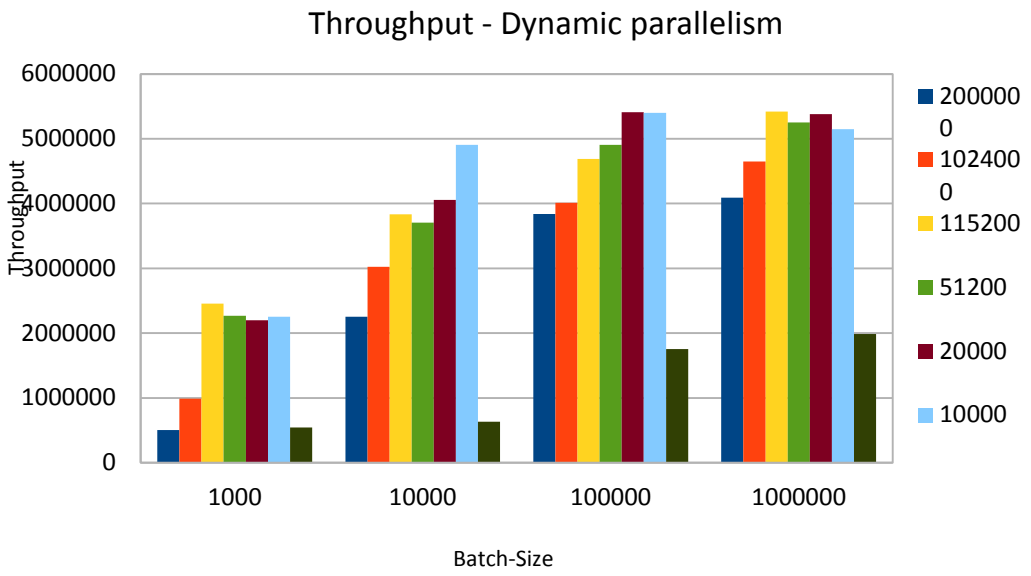

Figure 4-2 : Throughput analysis for different thread configurations - Dynamic parallelism mode

The figure 4-2 illustrates the throughput for different batch sizes with different thread configurations in dynamic parallelism mode. For the experiment we have used six different GPU thread configurations ranging from 10,000 to 2 million. With such a wide range of thread configurations we can make few interesting observations.

1. With a 1000 element batch, the CPU version has better performance than the GPU version of 2 million threads (Around 2000 blocks were used).

2. For medium and high batch sizes, low thread counts have better performance.

To understand aforementioned behaviors we need to recall how the pre-processing works in a GPU. During the pre-processing, each thread is allocated a range of bytes from the byte stream. The byte stream is distributed among threads equally. At the end of marking boundaries, the starting thread of each block launches a child kernel to carry out the decoding. For that child kernel to be scheduled, there should be enough resources available in the GPU. What happens with a large number of threads is that it exhausts the maximum allowed active warp count on each SM and many of these warps would not be doing any significant work either. Therefore child kernel launch requests would be delayed and queued up in the execution engine.

With a low thread count such as 10,000 or 20,000 the amount of work done by each thread is significant and most importantly, there would be enough resources left for child kernels as well. The bottom line is that in order to get the best performance out from the SMPP decoder, it is imperative not to over allocate threads to pre-processing stage.

4.3.3 Throughput Analysis of Normal GPU Mode

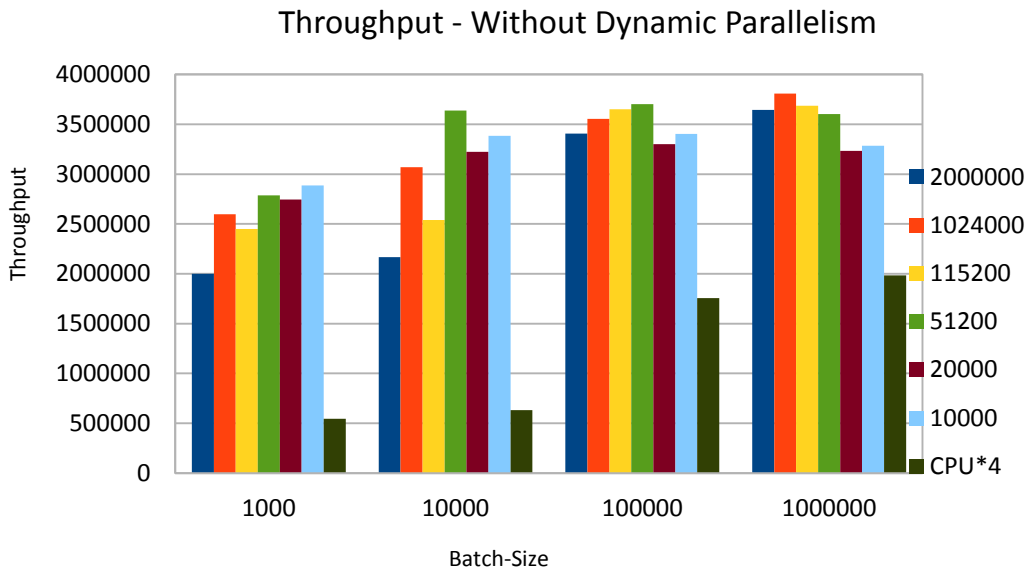

Figure 4-3: Throughput analysis for different thread configurations - Normal GPU mode

The test runs were done with 4 CUDA streams. When comparing figure 4-3 and figure 4-2, it is evident that for small batch sizes like 1000, normal GPU implementation has an advantage over using Dynamic Parallelism. The reason is that for small batch sizes, the time taken by the CPU is less than the time taken by any GPU configuration to mark PDU boundaries.

Next observation is that with a batch size of 1 million, the thread configuration for the best throughput was 1.2 million. When it comes to Dynamic Parallelism version, it was 20,000 threads.

The reason is that with normal GPU implementation, the GPU decoding throughput was only affected by the available cores. Whereas with Dynamic Parallelism based version, over allocation of threads to pre-processor stage might exhaust GPU resources and therefore reduces the throughput.

For an example, with a batch size of 1 million, the number of threads need to mark PDU boundaries is not that much higher than the number of threads need with a 100,000 batch size. It stays around the same value. What matters is the availability of resources to schedule child kernels to do the decoding.

Last observation is that the rate of the throughput difference for different thread configurations is low for large batch sizes. The reason again is the time taken for pre-processing. Normal GPU execution performs the pre-processing at the CPU. Thus the throughput is limited by the number of available CPU resources. In other words when the batch size increases, CPU based pre-processing becomes the performance bottleneck.

4.3.4 Effect of Block and Grid Sizes on Performance

Regardless the GPU mode, finding the best thread configuration is a key to achieve a high throughput. Even with the same number of threads, different Block and Grid configurations yield different performances. For an example, for a batch size of 10,000, the best performance is given by 10,000 threads (Dynamic parallelism mode). With that said, we may configure the 10,000 threads in different Grid and Block configurations. Following figure 4-4 illustrates performance for each configuration.

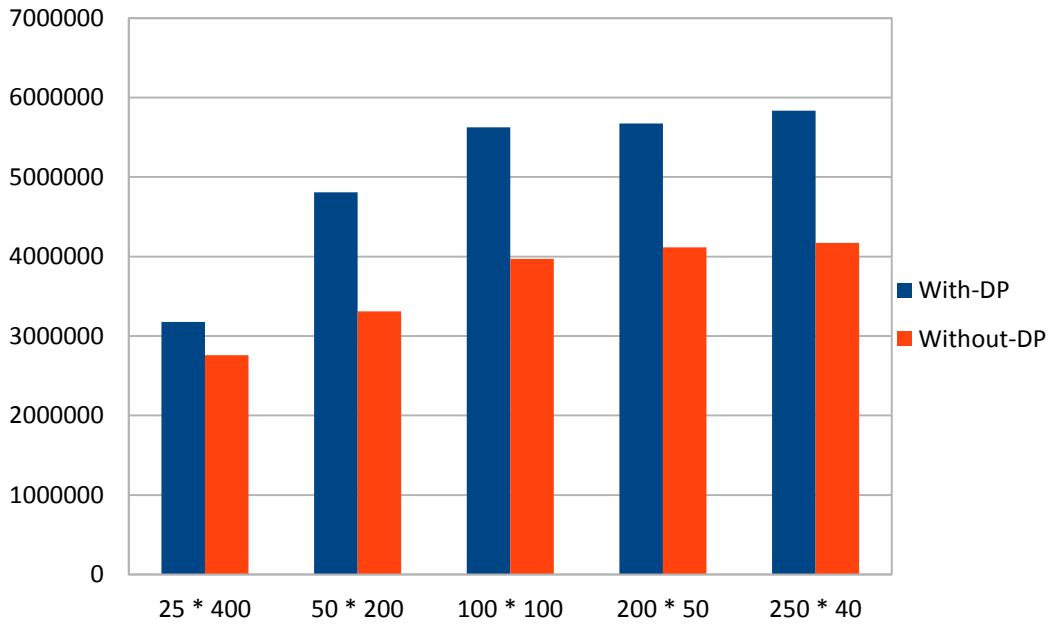

Figure 4-4 : Performance for different Block and Grid sizes. The batch is 10,000 elements. X axis format is (Block Size * Grid Size)

Looking at figure 4-4, two observations can be made. First is that irrespective of the GPU mode (Dynamic parallelism or normal mode), when the number of Blocks decreases, throughput increases. Scheduling Blocks is a costly operation compared to scheduling threads in a SM. To schedule a Block, the GPU need to assign Shared memory, check for the available SPs in the potential SM etc. On the contrary scheduling threads for a Block is more and done by their respective SMs. Also there is a bigger penalty for using higher number of blocks in dynamic parallelism mode. It exhausts the available SMs for child kernels. All the aforementioned factors contribute to the first observation.

Second observation is that irrespective of the GPU mode, for Block numbers below 100, the rate of performance gain is negligible. Every GPU based application has a sweet spot when it comes to Block and Grid configurations. In our case, it is close to 100 Blocks per Grid and 100 threads per Block. When we are at a sweet spot, any further improvement in thread allocations may not yield a significant performance gain. In the context of the example, around 100 Blocks the overhead of Block scheduling becomes insignificant compared to the workload.

5 Conclusion and Future Work

5.1 Conclusion

We started the discussion with an introduction to the SMPP protocol and then introduced the popular SMPP framework Cloudhopper (section 2.1 and 2.2). Next we discussed GPUs in general along with heterogeneous computing model and following that looked into Nvidia GPU architecture basics and CUDA programming model (section 2.3).

Next we started looking into GPU based network packet processing. Throughout the section 2.4, the discussion was planned out across four major steps of network packet processing. For each step, we discussed the available options and their pros and cons.

Then we stared discussing the SMPP decoder library implementation in chapter 3. We first introduced primary modules involved in the project and then started discussing specific implementation details of the library.

Next in chapter 4 we explained the evaluation process and the section 4.3 produces an analysis on the throughput of the system for various grid and block sizes and batch sizes. The analysis was done for three decoding modes namely CPU based decoding, normal multi-stream GPU based decoding and dynamic parallelism based GPU decoding.

The analysis in chapter 4 shows that the SMPP decoder library provides better performance in both dynamic parallelism based and normal multi-stream based GPU modes than the CPU based implementations (Implemented in both Java using Cloudhopper and C). The performance gain of GPU based decoding modes for most commonly used batch sizes like 10,000 is more than five times and in average the gain is around four times over CPU based decoding. Therefore in conclusion, we have achieved a significantly accelerated SMPP decoding through GPUs.

5.2 Future Work

1. One of the major drawbacks in the SMPP decoder is the poor JNI performance. When the JNI interface is used, the library adaptation layer has to convert each decoded C Structure into a Java object. For an example, for a batch size of 10,000 PDUs, the decoder has to create 10,000 Java objects and put them into a List and then return the List back to the Java application. Crossing the JNI boundary for 10,000 times is costly. One suggestion is to not create Java objects for every decoded PDU. Instead the decoder will create a new output byte stream with decoded bytes and delimiters for each parameter. For an example, when the decoder encounters the Destination

Address field of a PDU, it would first update the output byte stream with the start delimiter for the Destination Address field then update the output byte stream with the actual decoded Destination Address bytes and lastly inserts the Destination Address end delimiter to the output byte stream.

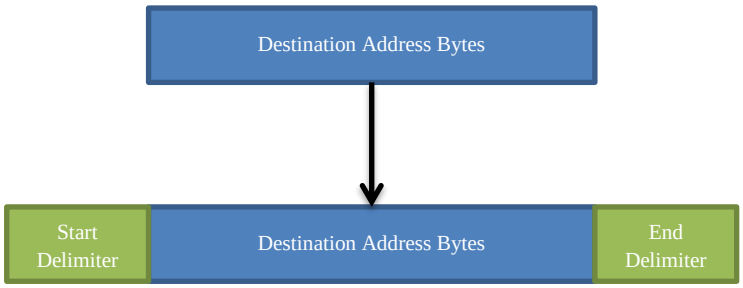

Figure 5-1 : Updating decoded fields with delimiters

The figure 5-1 shows the layout of the updated output byte stream for the Destination Address field. With that, the SMPP decoder does not have to create decoded Java objects for each PDU. Instead we could allocate an off heap memory for the new output stream and return the pointer. The Java application can now create Java objects just by reading the stream. The delimiters for each parameter are unique and would be stored in a table for fast lookup.

2. One of the most complex parts in CUDA application development is manipulating data in multiple memory spaces (CPU and GPU). For an example a pointer to a data structure allocated in CPU cannot be used in the GPU. Unified Memory Access (UMA) [29] [30] solves much of the programming complexities associated with working on multiple memory spaces. Unified Memory is a single memory address space that can be accessed by any processor (GPU or CPU) in the system. It enables the programmer to allocate and manipulate data in the CPU as well as in GPU without explicitly copying data from and to CPU/GPU.

3. The SMPP decoder library currently supports only a limited amount of TLVs in GPU modes. The number of TLVs supported cannot be changed dynamically at runtime and we need to update the library to decode any number of TLVs in a single packet.

4. The library uses a maximum SMPP PDU size in order to limit unnecessary memory allocations. We need to update the library to facilitate the PDU size based on the batch size so that smaller batches could have large SMPP packets.

6 References

[1] 3gpp-specification. (2013, Dec.) 3GPP Web Site - SMS Specification Archive. [Online]. http://www.3gpp.org/ftp//Specs/archive/23_series/23.040/

[2] SMPP Specification Version 3.4. [Online]. http://opensmpp.org/specs/smppv34_gsmumts_ig_v10.pdf

[3] Twitter. Git Hub - Cloudopper SMPP implementation source - Git-Tag - v5.0.9. [Online]. https://github.com/twitter/cloudhopper-smpp

[4] opensmpp. open smpp. [Online]. http://opensmpp.org/

[5] Peter Schwabe. (2013, Mar.) Graphics Processing Units- First Chapter. [Online]. https://cryptojedi.org/papers/gpus-20130310.pdf

[6] Netty IO. Netty IO. [Online]. https://netty.io/4.0/api/io/netty/channel/ChannelPipeline.html

[7] Netty-IO. (2017, Oct.) Netty Framework Main Website. [Online]. https://netty.io/

[8] Netty-IO. (2017, Sep.) Netty Source Code - GitHub - Tag - netty-4.1.16.Final. [Online]. 25

[9] Nvidia. Nvidia-GPU-Introduction. [Online]. http://www.nvidia.com/object/gpu.html

[10] NVIDIA. (2017) NVIDIA GPU Programming Guide - Version 2.5.0. [Online]. https://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf

[11] NVIDIA. (2010, April) Nvidia-Fermi-Architecture-Whitepaper. [Online]. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[12] Sergey Pakhomov. compress.ru - Microarchitecture AMD Bobcat vs. Intel Atom. [Online]. http://compress.ru/article.aspx?id=21785

[13] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym. NVIDIA TESLA:A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE. [Online]. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=795272016A91B31BF765F47CE0E3A758?doi=10.1.1.210.5016&rep=rep1&type=pdf

[14] Nvidia. NVIDIA's Next Generation CUDATM Compute Architecture- Kepler TM GK110. [Online]. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[15] Wikipedia. Maxwell-Microarchitecture. [Online]. https://en.wikipedia.org/wiki/Maxwell_(microarchitecture)

[16] Wikipedia. Pascal-Microarchitecture. [Online]. https://en.wikipedia.org/wiki/Pascal_(microarchitecture)

[17] Nvidia. (2010) Nvidia - Fermi GPU Architecutre Document. [Online]. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[18] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis, "Gnort: High Performance Network Intrusion Detection Using

Graphics Processors," in *RAID '08 Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, Cambridge, MA, USA, 2008, pp. 116-134.

[19] Nigel Jacob and Carla Brodley, "Offloading IDS Computation to the GPU," in *ACSAC '06 Proceedings of the 22nd Annual Computer Security Applications Conference*, Washington, DC, USA, 2006, pp. 371-380.

[20] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon, "PacketShader: A GPU-Accelerated Software Router," in *SIGCOMM '10 Proceedings of the ACM SIGCOMM 2010 conference*, NEW DELHI, INDIA, 2010, pp. 195-206.

[21] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, and Roger Boislaigue, "Caesar: a content router for high-speed forwarding on content names," in *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Marina del Rey, CA, USA, 2014, pp. 137-147.

[22] Jin Zhao, Xinya Zhang, Xin Wang, and Xiangyang Xue, "Achieving O(1) IP lookup on GPU-based software routers," in *SIGCOMM '10 Proceedings of the ACM SIGCOMM 2010 conference*, NEW DELHI, INDIA, 2010, pp. 429-430.

[23] Shuai Mu, Xinya Zhang, Nairen Zhang, and Jiaxin Lu, "IP routing processing with graphic processors," in *DATE '10 Proceedings of the Conference on Design, Automation and Test in Europe*, Leuven, Belgium, 2010, pp. 93-98.

[24] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis, "GASPP: A GPU-Accelerated Stateful Packet Processing Framework," in *USENIX ATC'14 Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, Philadelphia, PA, 2014, pp. 321-332.

[25] Chengkun Wu, Jianping Yin, Zhiping Cai, En Zhu, and Jieren Chen, "A Hybrid Parallel Signature Matching Model for Network Security Applications Using SIMD GPU," in *APPT '09 Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*, Rapperswil, Switzerland, 2009, pp. 191 - 204.

[26] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao, "Dynamic Load Balancing on Single- and Multi-GPU Systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, USA, 2010, pp. 1-12.

[27] Mohammed Sourouri, Tor Gillberg, and Scott B. Baden, "Effective multi-GPU communication using multiple CUDA streams and threads," in *2014 20th IEEE International Conference on Parallel and Distributed Systems*, Hsinchu, Taiwan, 2014, pp. 981 - 986.

[28] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. (2009) Semantic Scholar. [Online]. https://www.semanticscholar.org/paper/Enabling-Task-Parallelism-in-the-CUDA-Scheduler-Guevara-Gregg/95eeb5141693f807fba987a66fb63de40a5a0c27?tab=abstract

[29] Jin Wang and Sudhakar Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications," in *2014 IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, USA, 2014, pp. 51 - 60.

[30] Peter Zhang et al., "Dynamic parallelism for simple and efficient GPU graph algorithms," in *IA3 '15 Proceedings of the 5th Workshop on Irregular Applications:*

Architectures and Algorithms, Boston, 2015, pp. Article No:11.

[31] NVIDIA. Nvidia Dynamic Parallelism Tech Brief. [Online]. http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

[32] Mark Harris. (2017, June) devblogs.nvidia.com. [Online]. https://devblogs.nvidia.com/parallelforall/unified-memory-cuda-beginners/

[33] Raphael Landaverde, Tiansheng Zhang, and Ayse K. Coskun, "An investigation of Unified Memory Access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference*, Waltham, MA, USA, 2014, pp. 1-6.

[34] NVIDIA-CUDA Programming-Guide. [Online]. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#concurrent-kernel-execution

[35] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi, "Exploiting Concurrent Kernel Execution on Graphic Processing Units," in *2011 International Conference on High Performance Computing & Simulation*, Istanbul, Turkey, 2011, pp. 24 - 32.

[36] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron, "Fine-Grained Resource Sharing for Concurrent GPGPU," in *HotPar'12 Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, Berkeley, CA, 2012, pp. 10-10.

[37] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *ASPLOS '13 Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, Houston, Texas, 2013, pp. 407-418.

[38] Nicholas Wilt, *The CUDA Handbook*, 1st ed. Boston, Massachusetts: Addison-Wesley, 2013.

[39] NVIDIA. Nvidia CUDA Streams and Concurrency Webinar. [Online]. http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[40] Michael Bauer, Henry Cook, and Brucek Khailany, "CudaDMA: Optimizing GPU Memory Bandwidth via Warp," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Seatle, WA, USA, 2011, pp. 1-11.