ANALYSIS OF COMMUNICATION AND COMPUTATION OVERLAP IN ACCELERATED PROGRAMS

BY

SUSHMA YELLAPRAGADA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Adviser:

Professor Emeritus Marc Snir

ABSTRACT

Accelerated computing has revolutionized a broad range of industries by applying Graphics Processing Units (GPU(s)) to optimize workloads for efficient performance. Any application that is designed to run on such specialized devices has a typical flow that begins and ends with data movement between Central Processing Unit (CPU) and GPU. Previous research has shown that this data movement is indeed a major bottleneck with even most optimized applications hitting only 30% of the peak performance [1]. While most recent efforts have been to manually tune communication to improve performance, in this work, we present three different general optimization techniques to introduce overlap between communication and computation. The optimized model thus developed on a stencil application is evaluated against the baseline model using various performance metrics including time and throughput. Sets of experiments performed on two different GPUs reported that a certain combination of these optimizations result in a maximum speedup of 6.5 and a 160% increase in bandwidth utilization when compared to the baseline model in Compute Unified Device Architecture (CUDA).

To everyone on their journey of finding happiness.

ACKNOWLEDGMENTS

Throughout the writing of this thesis, I have received a great deal of support and assistance. I would first like to extend my sincere gratitude to my adviser, Professor Marc Snir, whose supervision and expertise have been invaluable in formulating the research methodologies in this work. I want to thank you for your support, patience and for all of the opportunities I was given to further my research throughout my degree program. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would like to thank Swann Perarnau, Argonne National Laboratory, for providing initial direction to this work and offering access to compute resources crucial for the research. I would also like to thank Prof. Steve Lumetta, for his GPU class sparked interest in me to pursue this subject. I sincerely appreciate my colleague, Hsin-Yu Huang, for helping setup the project environment and necessary tools to successfully complete my thesis.

I would like to acknowledge all of my lab mates for their passionate work that constantly motivated me, particularly Chen Wang, for collaborating on interesting research. In addition, I would like to thank my parents, grandparents and uncle for their wise counsel and sympathetic ear. You are always there for me.

Finally, I could not have completed this work without the support of my adorable sister, Sowmya, and all of my wonderful friends, who provided stimulating discussions as well as happy distractions to rest my mind outside of academics.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The parallel computing capacity of graphics processing unit (GPU) has surpassed that of the CPU, allowing GPU to improve the efficiency of compute-intensive applications. High performance GPU computing becomes an inevitable trend due to the ever-increasing demand on computation capability in emerging domains such as deep learning, big data and planet-scale simulations. Many popular applications today rely heavily on GPUs, including machine learning model training, self-driving vehicle computer vision, and atmospheric model simulations.

The GPU is a separate device from the CPU and CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to run on general purpose GPUs [2]. Before the CPU assigns the task to the GPU for execution, the data related to the task must be copied from the CPU's main memory to the GPU's global memory. In addition, when the GPU completes its tasks, it is usually necessary to copy the execution results back to the CPU from GPU for subsequent application use. Therefore, understanding how to effectively transfer data between the two - also referred to as host(CPU) and the device(GPU) becomes critical.

The Peripheral Component Interconnect Express (PCIe) is the hardware interface found in most GPU nodes that connects the host to device. This bus, depending on its type, has a certain bandwidth through which the host and devices can transfer data. The data transmission speed will directly affect the total execution time of the entire task. The peak theoretical bandwidth between device memory and the device processor is significantly higher than the peak theoretical bandwidth between the host memory and device memory. Therefore, in order to get best performance of an application, it is important to minimize these host-device data transfers.

(Original) PCI Bus Specification: Fig 1.1 shows PC architecture. While Northbridge connects components that must communicate at high speed, Southbridge serves as a concentrator for slower I/O devices. PCI is connected to the Southbridge. Original specifications were - 33 MHz, 32-bit wide, 132 MB/second peak transfer rate, however more recently it is 66 MHz, 64-bit, with 528 MB/second peak. Upstream bandwidth remain slow for device (256 MB/s peak). It holds a shared bus with arbitration (Winner of arbitration becomes bus master and can connect to CPU or DRAM through the Southbridge and Northbridge).

PCIe - PCI Express: PCIe forms the interconnect backbone; Northbridge and Southbridge are both PCIe switches. Some Southbridge designs have built-in PCI-PCIe bridge to allow old PCI cards while some PCIe I/O cards are PCI cards with a PCI-PCIe bridge.

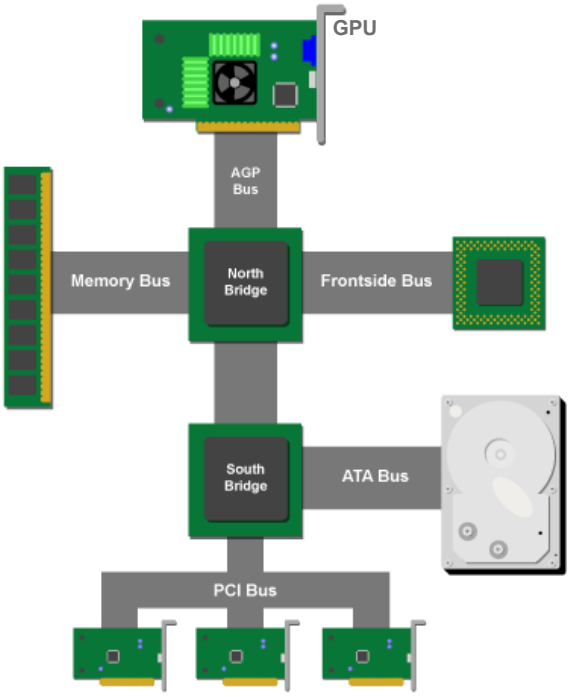

Figure 1.1: PC architecture

PCI-Express protocol is becoming the bottleneck [3]. Perhaps because of technologies as NVM-Express[4] for linking flash and soon other non-volatile persistent storage like 3D XPoint[5], more directly to the CPU-main memory and the attachment of GPU and sometimes FPGA accelerators for offloading massively parallel or specialized functions, the PCI-Express bus is getting overloaded. The bandwidth between key components ultimately dictates system performance and it is especially true for massively parallel systems processing massive amounts of data. Tricks like buffering, reordering and caching can temporarily defy the rules in some cases but ultimately, the performance falls back to what the speeds dictate.

Existing research addressing this issue focuses on either manual tuning of data communication, arrangement of data layouts, or using different levels of CUDA memory hierarchy, for example, unified memory with adaptable page size migrations[6] etc for efficient data transfers. The management of data communication between CPU and GPU continues to evolve. Initially, the programmer controls the data transfer between CPU and GPU explicitly. To simplify programming and enable system-wide atomic memory operations, GPU vendors have developed a programming model that provides a single virtual address space. The page migration engine in this model migrates pages between CPU and GPU on demand automatically. While these techniques might result in performance improvements, they are

often trained in an application specific manner or are related to physical aspects of the systems like interconnects and memory hierarchy and hence are not reproducible across memory intensive applications.

Despite the increasing investment in integrated GPUs and next-generation interconnect research, discrete GPUs connected by PCI Express still account for the dominant position of the market. Many programmers are unaware of the high overhead associated with data transfers and by intelligently reducing or eliminating them, very large gains in performance can be observed. In this study, we explore three general optimizations in order to better understand the behaviour of data transfers. The core of the work revolves around devising programming models for achieving concurrent data transfers and kernel executions. We apply CUDA streams to structure each stream such that memory copy from host to device, kernel executions, and memory copy from device to host all occur concurrently. The baseline model is a stencil application developed in CUDA that performs memcopy and kernel launches in a serial manner. We introduce an optimized model that utilizes CUDA streams and achieves significant speedup and bandwidth utilization over the baseline implementation.

The rest of this work is organized as follows. Chapter 2 gives introduction to accelerated programs, typical flow of a CUDA program, the data transfer bottlenecks and existing research addressing this issue. We then describe the three optimization approaches - Pinned memory, CUDA streams and host/device computation overlap in Chapter 3 followed by the implementation of optimized model on a stencil application with custom parameters in Chapter 4. Chapter 5 evaluates and analyses the performance of optimized model over baseline model using three performance metrics - the total execution time, the `memcpy` throughput and hardware-level performance. Finally, we conclude in Chapter 6.

CHAPTER 2: BACKGROUND

2.1 ACCELERATED PROGRAMS

Accelerated Computing is a computing model used in scientific and engineering applications whereby calculations are carried out on specialized processors (known as accelerators) in tandem with traditional CPUs to achieve faster real-world execution times. Accelerators are highly specialized microprocessors designed with data parallelism in mind. In the most practical terms, execution times are reduced by offloading parallelizable computationally-intensive portions of an application to the accelerators while the remainder of the code continues to run on the CPU.

Typical flow of an accelerated program consists of:

1. Copy data from CPU to GPU memory
2. CPU instructs the GPU (kernel configuration and launching)
3. Data processed by many cores in parallel
4. Copy result back from GPU to CPU's memory

Once the CPU launches the kernel, each thread in the GPU executes the same kernel with different input values based on its *id*(thread, block and grid). Steps 1 and 4 in Figure 2.1 is the focus of this study. Steps 2 and 3 are taken care by CUDA.

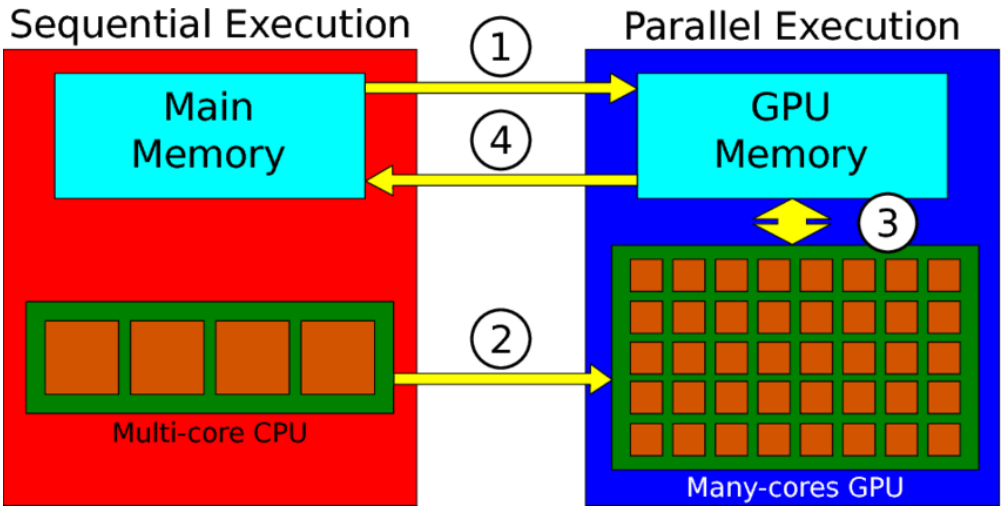

Figure 2.1: Flow of an accelerated program

CUDA (or Compute Unified Device Architecture): CUDA is a parallel computing platform and programming model developed by NVIDIA for improving computing performance by harnessing their graphical processing units (GPUs). The CUDA thread model is an abstraction that allows the programmer or compiler to more easily utilize the various levels of thread cooperation that are available during kernel execution. The CUDA thread model is closely coupled to the GPU architecture. When work is issued to the GPU, referred to as the *kernel* that is to be executed N times in parallel by N CUDA threads. CUDA threads are logically divided into 1,2, or 3 dimensional groups referred to as thread blocks. Threads within a block can cooperate through access to low latency shared memory as well as synchronization capabilities. All threads in a block must reside on the same streaming multiprocessor (SM) and share the limited resources.

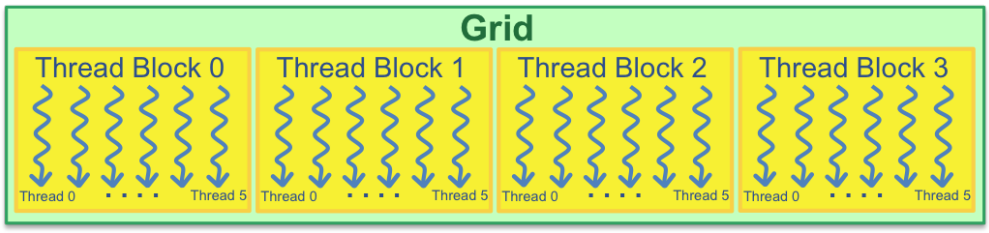

Figure 2.2: CUDA Thread Model - A 1D grid of 1D blocks

The thread blocks of a given kernel are partitioned into a 1,2, or 3 dimensional logical grouping referred to as a grid(Fig: 2.2). The grid encompasses all threads required to execute a given kernel. There is no cooperation between blocks in a grid, as such blocks must be able to be executed independently. When a kernel is launched the number of threads per thread block and the number of thread blocks is specified, which in turn defines the total number of CUDA threads launched.

Each thread has access to its integer position within its own block as well as the integer position of the thread's enclosing block within the grid. In general the thread uses this position information to read/write to/from device global memory. In this fashion, although each thread is executing the same kernel code, each thread has its own data to operate on.

2.2 BOTTLENECK

The lack of deep understanding on how modern GPUs are connected and the real impact of state-of-the-art interconnect technology in an accelerated application's performance is a hurdle. An accelerator(GPU) is a separate device from the host CPU and is attached with

some form of bus, like PCIe or CXL [7]. Some types of modern GPU interconnects are *PCIe*, *NVLink-V1*, *NVLink-V2*, *NV-SLI*, and *NVSwitch*.

The Peripheral Component Interconnect Express (PCIe), however, connects most current accelerators to the host system. This bus, depending on its type, has a certain bandwidth through which the host and devices can transfer data. An accelerator needs some data from host to do computation, and overall performance of the system is dependent on how quickly this transfer can happen. The PCIe interface is required to transfer data from the Host to the Device. However, because the PCIe interface is slower than the host and device memory IO speeds, it becomes a major bottleneck of the transferring process [8], [9]. Many applications including optimized ones are hitting only 30% of peak Flops[1]. It is not possible to simply increase bandwidth in PCIe since it basically implies accommodating the cost, power, size, and latency tradeoffs. With the current available hardware specifications of a PCIe, physics says the bus is a fundamental limiter of performance.

2.3 RELATED WORK

Current research work in this area mostly focuses on manual tuning of data communication and inputs. A case study on optimizing transformers [1] reports that data movement is the key bottleneck when training. Due to Amdahl's Law and massive improvements in compute performance, training has now become memory-bound. Their approach constructs a dataflow graph for the training process, which identifies operator dependency patterns and data volume. This representation identifies opportunities for data movement reduction to guide optimization. More modifications include data reuse through fusion and selecting data layouts for tensor operations. Work on efficient 3D stencil computations[10] introduces a new method of reading the data from global memory to the shared memory of thread blocks which maybe further optimized using different tiling strategies. This method avoids conditional statements and requires only two coalesced instructions to load the tile data with the halo.

Evaluation of modern GPU interconnects [11] indicate that, for an application running in a multi-GPU node, choosing the right GPU combination can impose considerable impact on GPU communication efficiency, as well as the application's overall performance. The management of data communication between CPU and GPU continues to evolve from programmer controlled data transfers between CPU and GPU, to a single virtual address space like the Unified memory model for simplifying programming and enabling system-wide atomic memory operations. The page migration engine in such models migrates pages between CPU and GPU on demand automatically. To meet the needs of high-performance

workloads, the page size tends to be larger. Limited by low bandwidth and high latency interconnects, larger page migration has longer delay, which may reduce the overlap of computation and transmission and cause serious performance decline. Partial-page migration [6] proposes that only the requested part of a page is migrated in order to shorten the migration latency and avoid performance degradation of the whole-page migration when the page becomes larger.

Increasing investment in being made by the industry in integrated GPUs and next-generation interconnect research. Different studies surrounding data transfer efficiency and related bottlenecks devised solutions resulting in improved performances, however, they looked at various different aspects such as leveraging CUDA memory hierarchy model, or manual tuning of data layout and movements; hardware/physical aspects like a combination of certain interconnects were also proposed. Efforts have been made to generate configurations that produce an optimized end-to-end implementation for applications, i.e., optimizations are application specific and might not directly carry over across diverse accelerated applications that are memory intensive.

CHAPTER 3: DESIGN

Now that the previous chapter laid the groundwork for understanding behaviour of accelerated programs and the associated bottlenecks, this chapter introduces three general optimizations studied w.r.t to achieving overlap between host and device memory copy operations and kernel executions.

3.1 PINNED MEMORY

In computer operating systems, memory paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory/RAM [12]. With paged memory, each program has its own logical memory, which can be broken into consecutive pages. These pages are then mapped to the physical memory via page table. When a program tries to access a page that is not stored in RAM, the processor treats this action as a page fault. When this occurs, the operating system must perform a sequence of expensive operations to page in and page out.

Conversely, the specific memory, which is not allowed to be paged in or paged out, is called page-locked memory or pinned memory. CUDA provides special system API function calls such as `CudaMallocHost()` for memory to be allocated inside this pinned memory while `CudaMalloc()` is a memory allocation function that enables allocation in pageable memory. As shown in Fig: 3.1, when copying data from the host (CPU) to the device (GPU), CUDA checks to see if the data is in pinned memory. If not, CUDA will copy the data to pinned memory before the data transfer can begin. Moving data from pageable memory to pinned memory could be slow, because some pages in pageable memory may be paged out and additional time is required to collect the missing pages. Even when missing pages is not the case, there would still be an extra copy operation from pageable to pinned memory. Since the data transfer between host and CUDA device have to use pinned memory anyway, to optimize the data transfer, instead of allocating pageable memory for storing data, we could allocate pinned memory directly for efficient data transfer performance.

3.2 CUDA STREAMS

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code [13]. The serial flow of an accelerated program as described in section 2.1 is executed in default CUDA stream 0. A default stream is the

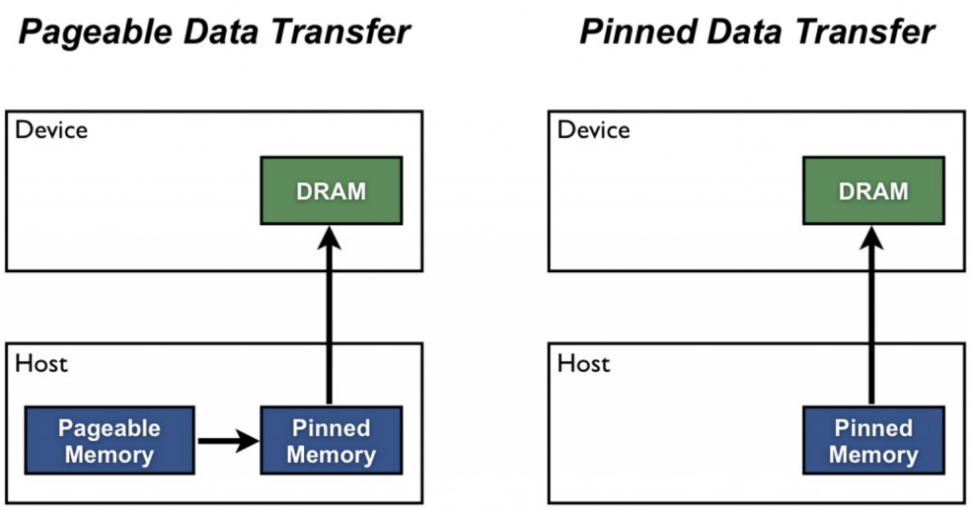

Figure 3.1: Data Transfer from Host to Device through Pinned Memory

stream used when no stream is specified. It is completely synchronous w.r.t. host and device. Parallelism in default accelerated programs is leveraged once the kernel is launched and all threads execute operations inside the kernel simultaneously. In practice there's two steps in addition to kernel executions that has the potential to be executed concurrently with kernel computation:

- Memory copy from Host to Device
- Memory copy from Device to Host

Therefore to make the memory copy more efficient, we may further improve the performance by doing memory copy from host to device, kernel executions, and memory copy from device to host all concurrently. This idea is implemented in CUDA using what is called *"streams"*. While all tasks launched into the same stream are executed in order, different streams, on the other hand, may execute their tasks out of order with respect to one another or concurrently (depending on available resources: compute and memory copy engines). Hence, to overlap different tasks, they should just be launched in different streams. To issue a data transfer to a non-default stream we use the `cudaMemcpyAsync()` function, which is similar to the `cudaMemcpy()` function, but takes a stream identifier as a fifth argument[1]. For optimal performance of CUDA streams, it is required that the data is allocated in pinned memory and is transferred using non-blocking call like `cudaMemcpyAsync()`. And, to issue a kernel to a non-default stream we specify the stream identifier as a fourth execution configuration parameter[2].

[1] cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0)

[2] kernel<<< Dg, Db, Ns, S >>> where S is of type cudaStream_t and specifies the associated stream

```
1   cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
2   kernel<<<1,N>>>(d_a)
3   cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Listing 3.1: A sample CUDA code performing memory copy and kernel launch operations

Fig 3.2 best explains the idea of CUDA streams, in Serial Model, there is only one (default) CUDA stream. All the commands were executed in order. However, in Concurrent Model, there are N (non-default) CUDA streams where all the commands in the same stream were executed in order but different streams have overlap. Note that the kernel executions on different CUDA streams may look exclusive in the figure, but depending on the GPU, the kernel executions on different CUDA streams could also overlap. We wanted to utilize this ability to achieve overlapping communication and computation so the optimal computation resources could be used and the lowest latency could be achieved. Implementation is described in Chapter 4.

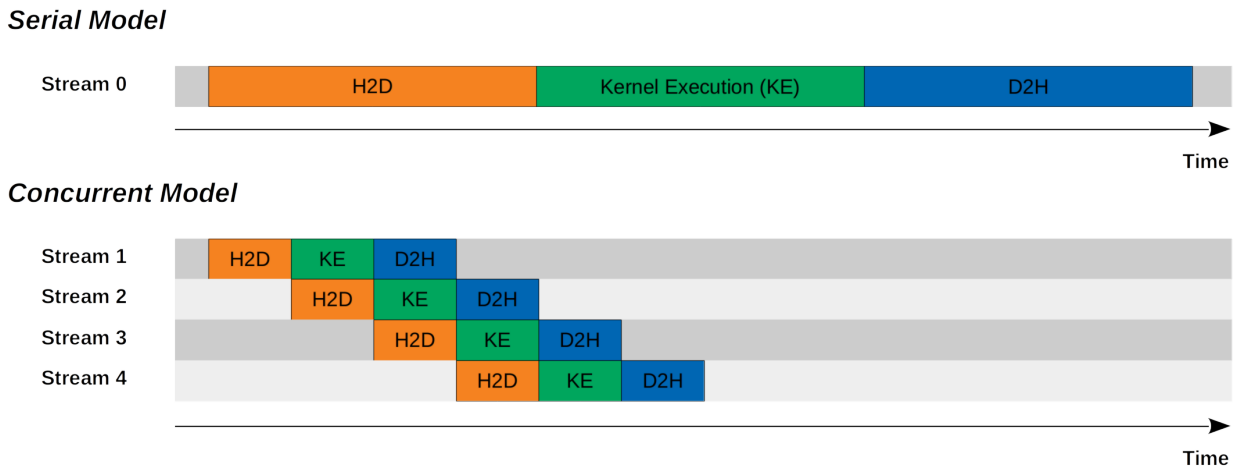

Figure 3.2: Serial Model vs Concurrent Model

3.3 DEVICE AND HOST COMPUTATION OVERLAP

So far the approaches discussed focused on efficient data transfer and overlaps between kernels and memory copy operations. In this approach to optimize an accelerated program, we look at the progress of operations from the perspective of the host as well as the device and study possible computation overlap between them.

For the lines of code in Listing 3.1, from the perspective of the device, all three operations are issued to the same (default) stream and will execute in the order that they were issued. However, from the perspective of the host, the implicit data transfers are synchronous or

blocking transfers, i.e., the CPU thread will not reach the kernel call on the second line until the host-to-device transfer in the first line is complete. However, the kernel launch from the host is asynchronous or non-blocking, meaning, once the kernel is issued, the CPU thread moves to the third line, but the transfer on that line cannot begin due to the device-side order of execution. This means that the host essentially enters an idle state after launching the kernel to the device as it waits for the device to complete its task. Utilization of this idle time on the host for computation is expected to enhance resource utilization efficiency and reduce computation time. The asynchronous behavior of kernel launches from the host's perspective makes overlapping device and host computation very simple. The above code could be modified to include an independent CPU function thereby offloading a portion of the total kernel computations to CPU. Once the host and device process their respective parts of the computations concurrently, partial results are aggregated. However, since the scope of the study is limited to overlapping data transfer and kernel executions on the GPU, this approach is not included in the final implementation.

CHAPTER 4: IMPLEMENTATION

This chapter describes implementation that achieved communication and computation overlap in detail by providing particulars on the application, algorithm, and custom parameters developed to implement and evaluate the optimizations described in previous sections.

4.1 STENCIL APPLICATION

Iterative Stencil Loops are a class of numerical data processing solution which update array elements according to some fixed pattern, called a stencil [14]. Stencil computations (arising from Jacobi iterative method) are most commonly found in computational science problems, widely applied in solving partial differential equations, in computational fluid dynamics and chemical simulations, etc. During each iteration in the application:

1. each grid point is updated
2. with a weighted linear combination
3. of a subset of neighboring values and itself.

This study is based on a representative case of such stencil application wherein the data points in the grid to be computed are represented in the form a matrix of size N x N x N (N is the number of elements on each of the x, y and z dimensions). Although real Partial Differential Equation solvers use bigger orders, for simplicity, we use a 7-point stencil computation implying that each output value in the resultant matrix is a combination of six other data points /neighbors, as shown in Fig 4.1, four neighbors - North, South, East, West on its plane and two neighbors on either side in z dimension, i.e., `A_in(i,j,k-1)` and `A_in(i,j,k+1)`. We set the weighted linear combination[1] for determining output value as

$$\begin{aligned} A\_out(i,j,k) = (constant) * A\_in(i,j,k) + A\_in(i,j,k-1) + A\_in(i,j,k+1) \\ +A\_in(i-1,j,k) + A\_in(i+1,j,k) + A\_in(i,j-1,k) + A\_in(i,j+1,k); \end{aligned} \tag{4.1}$$

Some of the reasons behind selecting this application are listed below:

- The symmetric and well-defined structure of stencil grids made it less complicated to apply customization, i.e., moving different sized chunks of data back and forth for computations and experiments.

[1] Although it maintains the same dependencies as real-world stencil computations, it may not to be treated as a representation of one

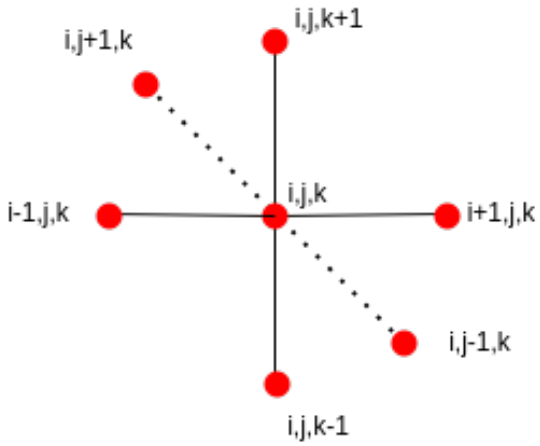

Figure 4.1: A 7-point stencil computation pattern

- Even though computation seems trivial for stencil applications, memory bottleneck becomes a big problem since the computation is tightly coupled to the memory. Most stencil codes are categorized as memory-bound [15]; therefore, more potential for efficient parallel implementation, particularly,in achieving the communication and computation overlap.
- Application contains significant amount of computation but without involving complex logic/formulas, hence, convenient to code.
- Lastly, a good example to study and analyse both software(kernel launches, execution time) and hardware performance metrics(arising from cache locality), etc.

4.2 BASELINE MODEL

CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing unit (GPU) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU) [2]. CUDA code compiles for the execution of compute kernels on GPU hardware. We use C/C++ language extensions provided by CUDA to develop our implementation.

The algorithm for baseline model follows the structure described in section 2.1. The data points generated on host are first copied to the device, the kernel is launched and finally the results are transferred to the host. Upon kernel launch, each thread in a block is responsible for computing all data points in its corresponding z dimension, i.e., each thread computes a vertical pencil (Figure 4.2) in the resulting grid by progressing at the pace of one plane.

All the corner rows/columns are halo elements and hence the CUDA kernel is iterating from plane 1 to *nz - 2* as it computes outputs on each plane (line 11 in Listing 4.1).

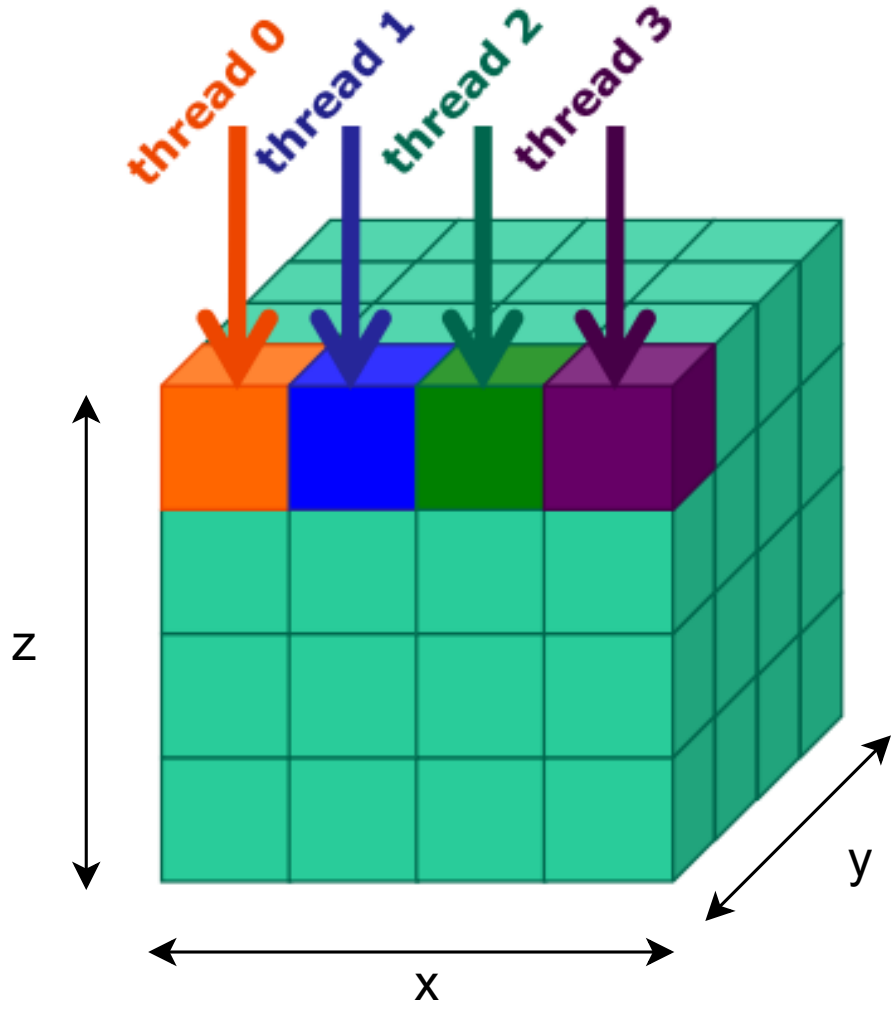

Figure 4.2: Representation of threads computing an individual pencil in the grid

Thread Coarsening and Register Tiling: Thread coarsening and register tiling are two important, closely related techniques for trading reduced parallelism in return for increased memory and compute efficiency. These are especially important when an application is memory-bound or compute-bound. Listing 4.1 is the baseline algorithm which includes these optimizations pertaining to its execution on the GPU. With thread coarsening, computation from merged threads can be shared through registers. In thread-level coarsening, each thread block performs the same amount of work but with fewer threads. The total number of thread blocks therefore remains unchanged, while the number of threads per block and the number of threads in total are reduced. Below listed are some properties of GPU registers

that enables re-use of result, avoiding redundant work when multiple threads are merged into one:

- extremely fast (short latency), and
- extremely high throughput: register file allows access to multiple registers per thread per cycle,
- but private to each thread: cannot use to share computation or loaded memory data, and
- named directly: any "arrays" must be fully expanded to constant indices (and loops unrolled).

The algorithm begins with each thread loading three values in its registers as indicated in lines 7-9: a `current` data point corresponding to each thread's coordinates, i.e., a thread with `threadIdx.x = 1` and `threadIdx.y = 1` will compute the element at (1,1) in the matrix; a neighbor from the plane directly above it; and a neighbor from the plane directly below it. The remaining neighbors are loaded from GPU's global memory when required (line 13). Finally, we take benefit of register tiling as applied in the `z` dimension in the algorithm (see lines 19-21); when moving from `z`→`z+1`, `current` input becomes `plane_previous`, `plane_next` input becomes `current`, and new `plane_next` input is loaded from memory. A total of `nz-2` planes are computed and stored in resulting `A_out` matrix which is transferred back to the host.

4.3 OPTIMIZED MODEL

This section will report optimizations w.r.t to achieving communication and computation overlap when compared to the base implementation. All CUDA compatible GPUs shipped in about the last 3 years support simultaneous host-device data transfers with kernel execution. As explained in section 3.2, CUDA streams are used in such scenarios to implement concurrent memory copy from host to device, kernel executions, and memory copy from device to host. While all tasks launched into the same stream are executed in order, different streams, on the other hand, may execute their tasks out of order with respect to one another or concurrently.

As an optimization to the Baseline model (Section 4.2) where the flow of the data transfers and kernels is serial, we use CUDA streams to split the data copying between nz (size of the input matrix in z dimension) number of streams such that each stream performs one

plane of data copying between host/device and executes the kernel for one plane. Listing 4.2 shows the structure of each stream. Each stream z here essentially performs:

1. Initiate transfer of data points in plane *z+1*
2. Compute kernel for data points in plane z
3. Copy the computed kernel for plane z back to the host.

A caveat, however, is to make sure that all the data required to compute a plane arrives in the GPU's global memory before it starts executing the kernel. There are two types of stream synchronization in CUDA in this regard. A programmer can place the synchronization barrier explicitly, to synchronize tasks such as memory operations. Some functions are implicitly synchronized, which means one or all streams must complete before proceeding to the next section. CUDA provides stream management functions to ensure synchronization among streams. `cudaEventRecord()`[2], captures in *event* the contents of *stream* at the time of this call. The *event* and *stream* must be on the same CUDA context. Calls such as `cudaStreamWaitEvent()`[3] will then examine or wait for completion of the work that was captured. Uses of stream after this call do not modify event. We include explicit CUDA synchronization calls specific to each stream(lines 23-24, Listing 4.2) that will guarantee the correctness of the program. It is sufficient that a stream waits only for the host→device `cudaMemcpyAsync()` from the previous stream to account for the data dependence between planes.

Fig: 4.3 is a snapshot obtained from Nsight Systems Visual Profiler[16] when run on the optimized model with matrix dimensions as 256 on a Tesla V100 GPU. NVprof[17] is a command-line light-weight GUI-less profiler available on Linux. This tool enables collection and viewing profiling data of CUDA-related activities on both CPU and GPU, including kernel execution, memory transfers, etc. Using NVprof to profile executable generates a `.qdrep` report that can be visualised in Nsight Systems Profiler. The figure demonstrates overlapping of the above three steps - memory operations and kernel executions across streams specifically between streams 82 through 94 in the figure. The memcopy operations from both directions (host and device) as well as kernel executions (blue blocks) can be seen occurring concurrently throughout the execution.

We see that each plane in the optimized algorithm, Listing 4.2, makes one sync call which is expensive because the thread remains in wait state till the previous stream's data copy step is complete. Therefore, as a further optimization, in order to reduce the total number

[2] cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)

[3] cudaStreamWaitEvent (cudaStream_t stream, cudaEvent_t event, unsigned int flags = 0)

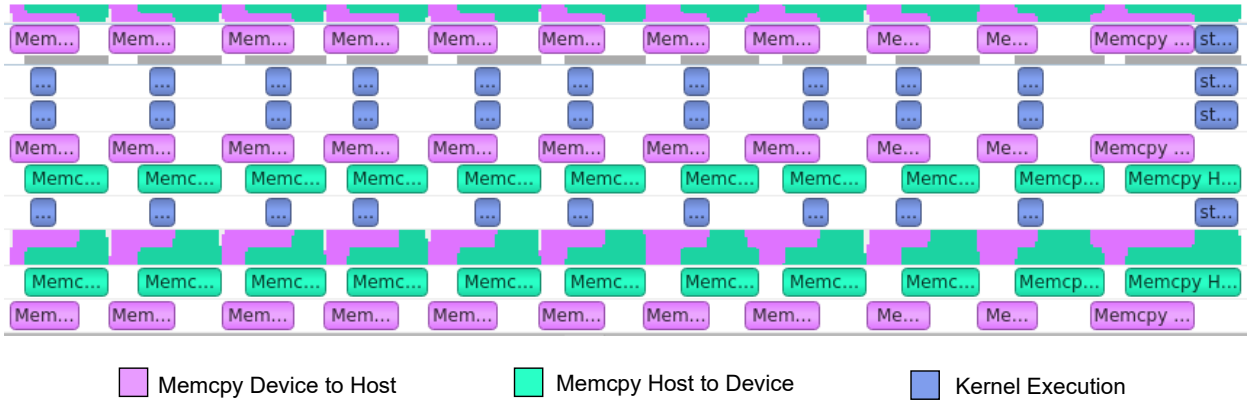

Figure 4.3: A timeline view of events from Nsight Visual Profiler

of sync calls made by the application, we split the number of planes `nz` into chunks of size `c`. `c` here is the number of consecutive planes in each stream and is responsible for computing the corresponding `c` planes in the resultant matrix. Related performance and analysis is made in Chapter 5.

```
__global__ void kernel(int *A_in, int *A_out, int nx, int ny, int nz) {

    // Compute the data point coordinates corresponding to each thread
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    int plane_previous = A_in(i, j, 0);
    int current =  A_in(i, j, 1);
    int plane_next = A_in(i, j, 2);

    for (int k = 1 to k < nz - 1) {

        A_out(i,j,k) = -6 * current + plane_previous + plane_next + A_in(i-1, j, k) +
        ↪  A_in(i+1, j, k) + A_in(i, j-1, k) + A_in(i, j+1, k);

        if (k == nz - 2) {
          plane_next = plane_next;
        }
        else{
          plane_previous = current;
          current = plane_next;
          plane_next = A_in(i, j, k+2);
        }
    }
}

int main() {

    // nx, ny, nz = dimensions of input matrix
    long int bytes = nx * ny * nz * sizeof(int);


    // Allocate memory(pageable) on host and device
    hostA_in = (int*)malloc(bytes);
    hostA_out = (int*)malloc(bytes);

    cudaMalloc(&deviceA_in, bytes);
    cudaMalloc(&deviceA_out, bytes);

    // Copy Input data from Host to Device
    cudaMemcpy(deviceA_in, hostA_in, nx * ny * nz * sizeof(int),
    ↪  cudaMemcpyHosttoDevice);

    // Launch the kernel with required configurations
    launchStencil(deviceA_in, deviceA_out, nx, ny, nz);

    // Copy the computed data from Device to Host
    cudaMemcpy(hostA_out, deviceA_out, nx * ny * nz * sizeof(int),
    ↪  cudaMemcpyDevicetoHost);
}
```

Listing 4.1: CUDA kernel for computing one iteration of the stencil in Baseline Model

```
int main() {

    // Allocating memory (pinned) on the host
    cudaMallocHost(&hostA_in, bytes);
    cudaMallocHost(&hostA_out, bytes);

    // Creating CUDA streams
    cudaStream_t stream[nStreams];
    const int streamSize = (nx*ny*nz) / nStreams;
    const int streamBytes = streamSize * sizeof(int);
    for (int i = 0; i < nStreams; ++i)
        cudaStreamCreate(&stream[i]);

    // Each stream launching memcpy and kernel corresponding to a plane
    for (int i = 1; i < nStreams-1; ++i) {
        int offset = i * streamSize;
        int send_offset = (i+1) * streamSize;

        // Asynchronous memcpy operation from Host to Device
        cudaMemcpyAsync(&device_in[send_offset], &hostA_in[send_offset],
        ↪  streamBytes, cudaMemcpyHostToDevice, stream[i]);

        // Synchronization
        cudaEventRecord(event, stream[i]);
        cudaStreamWaitEvent(stream[i+1],event,0);

        // Launch the kernel with required configurations
        stream_kernel<<<DimGrid, DimBlock, 0, stream[i]>>>(deviceA_in,
        ↪  deviceA_out, nx, ny, nz, i);

        // Copy the computed data from Device to Host asynchronously
        cudaMemcpyAsync(&hostA_out[offset], &deviceA_out[offset],
        ↪  streamBytes, cudaMemcpyDeviceToHost, stream[i]);

    }

    // Destroy Streams
}
```

Listing 4.2: CUDA Streams to perform overlapping communication and computation in the Optimized Model

CHAPTER 5: EVALUATION

5.1 ENVIRONMENT SETUP

Computing Instance: We used a Chameleon [18] computing instance to conduct experiments, study performance and evaluate. Chameleon is an open production experimental environment with sites at the University of Chicago and the Texas Advanced Computing Center. Chameleon supports computer science systems research including new operating systems, virtualization methods, power management and artificial intelligence. Chameleon is a testbed system for computer science experimentation funded by the National Science Foundation(NSF). It has configurable hardware for researching systems, networking, distributed and cluster computing, and security. A computing instance on Chameleon with GPUs installed was created. The detailed instance specification is shown in Table 5.1. The CUDA Toolkit (Version 10.2) that provides a comprehensive development environment in C and C++ was used for building GPU-accelerated stencil application introduced in Section 4.1.

OS	Ubuntu 20.04.2 LTS
CPU	2 Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz * 24 cores
L1d cache	768 KiB
L1i cache	768 KiB
L2 cache	24 MiB
L3 cache	38.5 MiB
Memory	187 GiB
GPU 0	NVIDIA Tesla V100
GPU 1	NVIDIA Tesla K80

Table 5.1: Hardware Specifications of the compute instance

Profiling Tools: Profiling tools enable performance analysis by providing detailed information about how applications are using devices in a system. The following profiling tools were applied in this study:

Device Properties	Tesla V100	Tesla K80
Multiprocessors	80	13
CUDA Cores/MP	64 (Total - 5120)	192 (Total - 2496)
Global Memory	32510 MB	11441 MB
Concurrent copy and kernel execution:	Yes with 7 copy engine(s)	Yes with 2 copy engine(s)
Supports Cooperative Kernel Launch	Yes	No

Table 5.2: V100 vs K80 Device Properties

1. Nvprof[17] - Profiling tool to collect and view profiling data from the command-line. NVprof generates textual reports providing summary of GPU and CPU activity. NVprof can be used on headless node to collect this data and later used in visualising the timeline of events with Visual Profiler.

2. Nsight Systems[16][19] - Graphical profiling tool that displays a timeline of application's CPU and GPU activity. It offers low overhead capture for GPU compute; has faster GUI with detailed data on events and logs.

3. Linux perf[20] - For hardware-level performance counters. Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. They form a basis for profiling applications to trace dynamic control flow and identify hotspots. *Perf* can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing). It is capable of lightweight profiling. *perf* provides rich generalized abstractions over hardware specific capabilities. Among others, it provides per task, per CPU and per-workload counters, sampling on top of these and source code event annotation.

5.2 PERFORMANCE ANALYSIS

Experiments and observations consisting implementation of baseline and different optimizations described in Chapter 3 with varying input sizes are reported in this section. Results are aggregated and presented w.r.t. three performance metrics - The total execu-

tion time, throughput of memory operations, and hardware performance in the following subsections.

5.2.1 Total Execution Time

(a) Baseline Vs Optimized: Figure 5.1 is a comparison of total execution time as observed on Tesla V100 GPU of baseline model vs optimized model(chunk size = 1, i.e., achieving overlap by transferring and computing one plane in each stream). The input matrix dimensions are varied between 4 to 512 with 2x increments. The blue line in the graph is for implementation identical to optimized version with the only exception being the memory allocated in pageable memory instead of pinned.

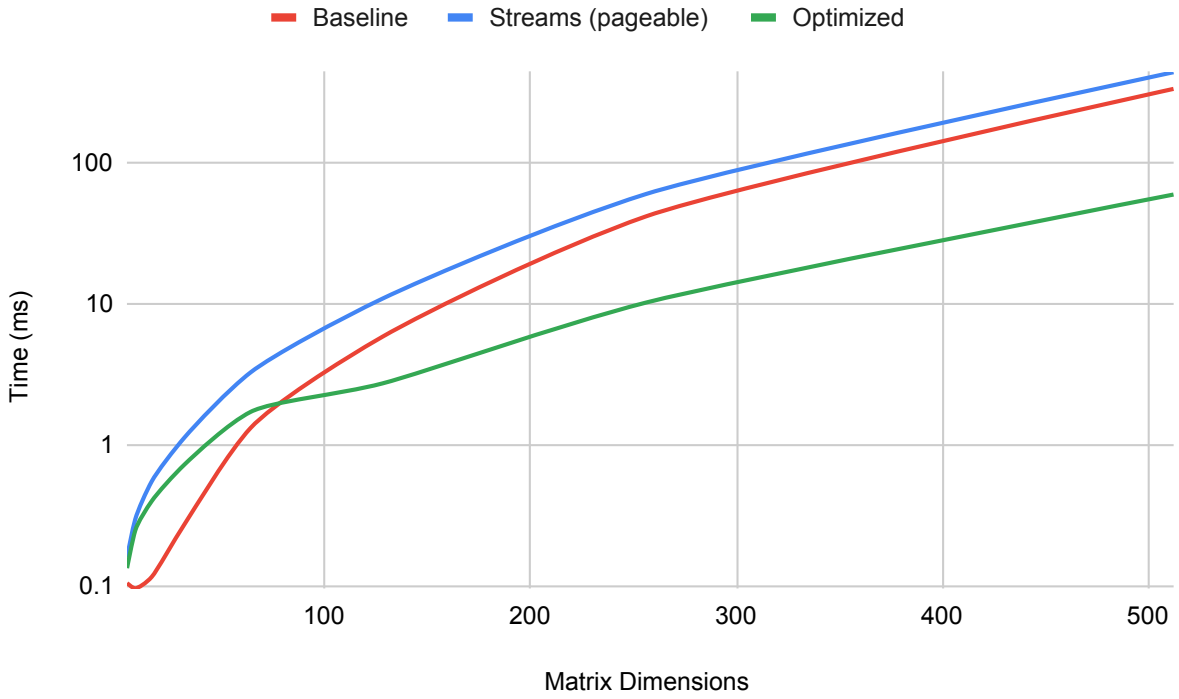

Figure 5.1: Total Execution Time Comparison

Observations:

(i) The optimized version is faster compared to the baseline across all sizes with a maximum speedup of 5.6 for size 512 (on V100) and 6.5 for size 1024 (on K80).

(ii) For asynchronous memory operations to take place, it is necessary for the memory to be allocated in pinned memory. The optimized version when performed using

pageable memory (in blue) performs the worst since no overlap ever took place, hence CUDA streams in the algorithm of this version only introduced additional overhead over base implementation.

(iii) Cost of CUDA kernel launches appear to be negligible since the total execution time is not effected in optimized version where each stream launches a kernel as opposed to the baseline where the kernel is launched only once. This could be explained by how a significant 70-80% of the time is spent in `memcpy` operations.

(b) Custom Chunk Size: In an attempt to reduce the number of sync calls, thereby reducing the total execution time, the optimized version was modified to experiment with custom chunk sizes of 4 and 8 as shown in Fig 5.2.

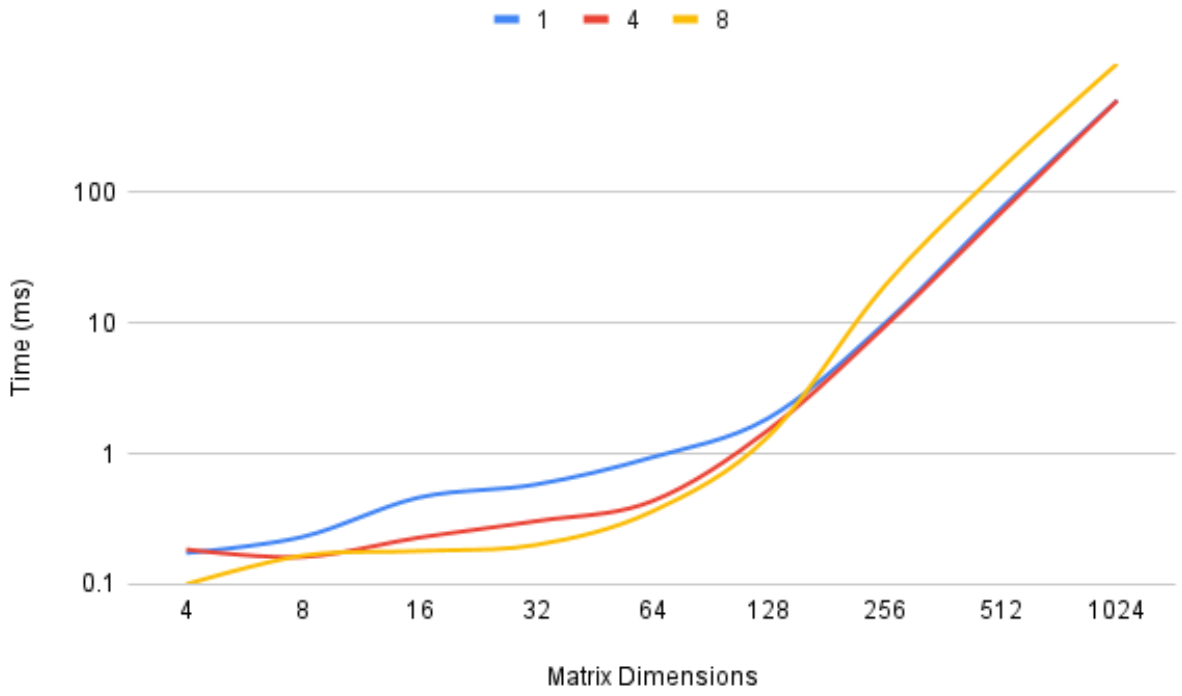

Figure 5.2: Total Execution Time (for chunk sizes - 1, 4, 8)

Observations:

(i) The modified chunk sizes of 4 and 8 report improved performance of at least 2x over chunk size = 1. This is expected since each stream is responsible for computing a chunk of planes without the need of synchronization calls in each plane.

(ii) However, there still exists the need for synchronization calls between each stream of chunk size c. It is logical to deduce that as c increases, so does the data dependence between streams. Therefore, the corresponding synchronization calls become expensive since the stream waits for larger amounts of data being transferred from previous stream. This is verified in the figure by matrix size 128 and higher where synchronization overhead starts to dominate the performance gained from reduced number of sync calls, resulting in overall diminished performance, more so for size 8 than for 4.

(c) K80 Vs V100: The following is an experiment to observe performance of the same two models across different GPU architectures(see Fig: 5.3). One being a Tesla K80, while the other one is a Tesla V100.

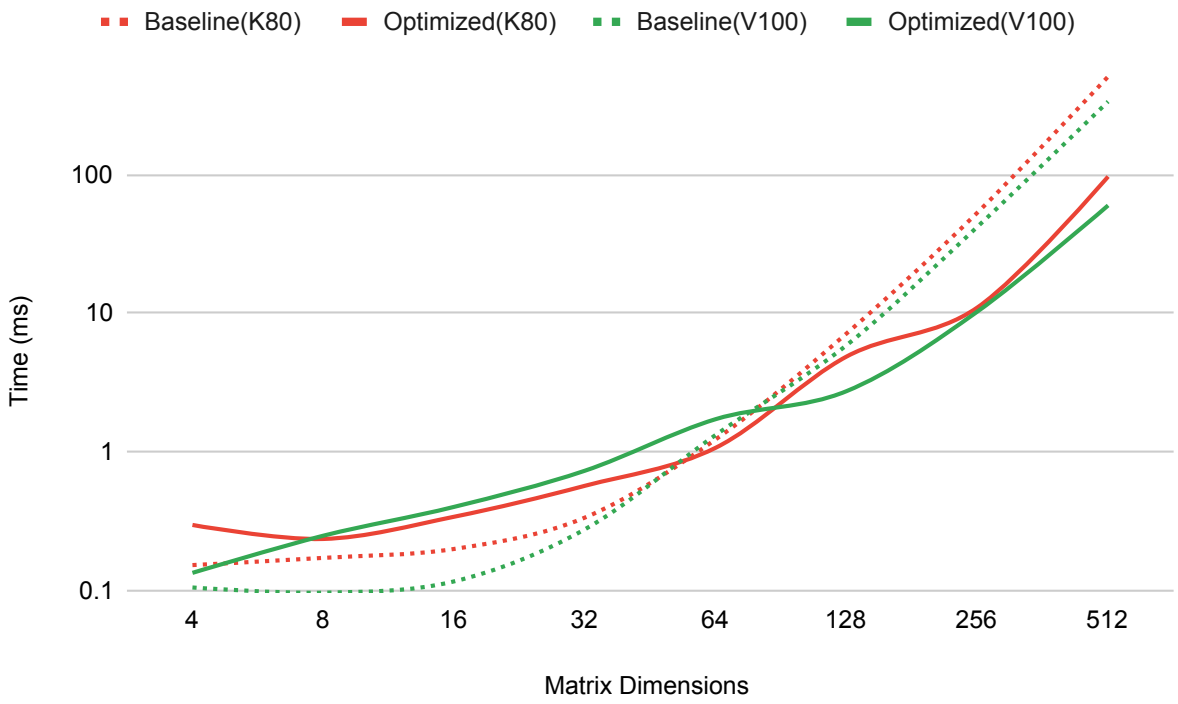

Figure 5.3: Total Execution Time on Tesla K80 and Tesla V100

Observations:

(i) As expected, both GPUs report an average of 5.5x speedup against their respective baselines.

(ii) Both the GPUs seem to follow similar trends w.r.t to total execution time vs the size. However, V100 is faster than K80 for bigger sizes. This can be reasoned

with the physical specification differences between the two GPU models. Both GPUs have a PCIe 3.0 x16 interconnect but V100 has more compute resources from more cores and faster flop performance (Refer Table 5.2).

5.2.2 Throughput

Figure 5.4 reports the peak memcopy throughput, i.e., the number of bytes communicated per second observed across PCI-e. Throughput in both directions; from host to device(H2D) and device to host(D2H) for varying sizes on baseline and optimized versions are included. The NVIDIA CUDA Example Bandwidth test is a utility for measuring the memory bandwidth that can be expected between the CPU and GPU for the given device. When tested on our instance(V100) for pinned memory transfers, it reported 11.5 GiB/s and 12.3 GiB/s from Host to Device and Device to Host respectively. On the other hand, Device to Device bandwidth was reported as 678.75 GiB/sec.

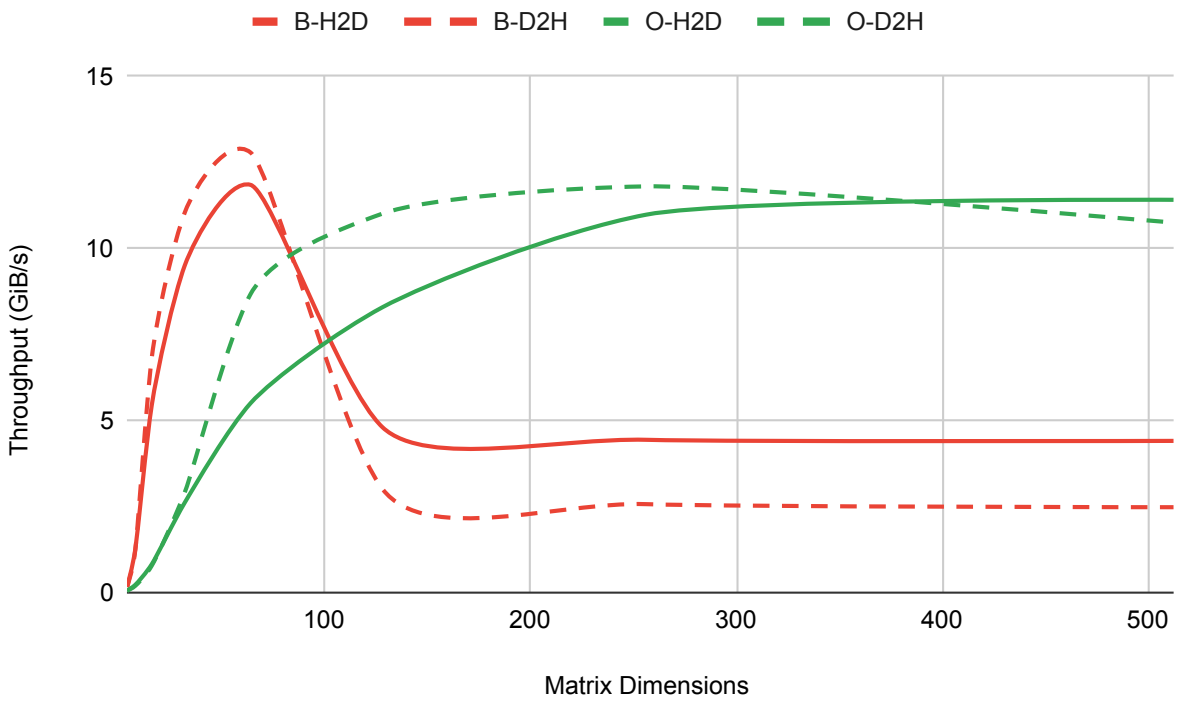

Figure 5.4: Memory Throughput Comparison

Observations:

(i) Baseline model reports higher throughput for smaller data sizes with a peak throughput of 12.7 GiB/s for size 64, however, optimized version implemented using CUDA streams shows 3x to 5x throughput improvement for sizes above 64. This is expected because PCIe is packetized transport, so throughout will differ based on transfer size, with smaller transfer sizes leading to lower throughput. In optimized model, since the data transfer occurs in sizes of one plane at a time, it reports lower bandwidth than baseline model which sends/receives all planes of the matrix per transfer. However with increased sizes, the optimized model continuously overlaps data transfers across streams, therefore resulting in efficient bandwidth utilization as opposed to baseline model where only two memcopy operations take place.

(ii) Throughput on both directions exhibit a similar trend, however, achieved values are not identical even for matching data sizes.

(iii) The total bandwidth found is dependent on the size of the transfer. Each transfer has a certain amount of overhead that will drag throughput down when attempted to perform small transfers instead of a few large ones. However, this effect largely disappears for transfers larger than 1048kB, though.

5.2.3 Hardware-level performance

Finally, Linux *perf* [20] tool was applied to measure hardware performance statistics, particularly page faults and load misses in order to compare Pageable and Pinned memory kinds presented as one of the optimizations in Chapter 3. Evaluation of few lower-level performance metrics when baseline implementation algorithm is executed with and without pinned memory for various data sizes is made in Tables 5.3 and 5.4 respectively.

Observations:

(i) Pinned version is observed to perform better only in a certain range of data size transfers depending on the system and architecture available. For example, in this compute instance only data transfers between 6K to 18K bytes resulted in improved performance over pageable memory. Otherwise, the data transfer using pageable memory is not much slower than the data transfer using pinned memory, presumably because of the powerful Intel CPU Xeon Gold 6126 made transfers from the pageable memory to the temporary pinned memory very fast.

data size (bytes)	total time (sec)	page faults	LLC load misses
1,296	0.057	8,473	163,814
4,624	0.051	8,479	176,483
17,424	0.067	8,484	174,053
67,600	0.106	8,515	173,617
266,256	0.307	8,614	165,692
1,056,784	1.05	9,003	177,428
4,210,704	3.997	10,542	186,217
16,810,000	14.221	16,690	227,845

Table 5.3: Performance Counters for Pageable Memory Version

data size (bytes)	total time (sec)	page faults	LLC load misses
1,296	0.069	8,483	179,269
4,624	0.067	8,483	177,692
17,424	0.061	8,484	173,931
67,600	0.067	8,481	177,687
266,256	0.094	8,483	166,427
1,056,784	0.235	8,481	173,676
4,210,704	0.749	8,484	168,810
16,810,000	2.788	8,485	171,791

Table 5.4: Performance Counters for Pinned Memory Version

(ii) Therefore, while pinned memory is encouraged, it should not be abused. Typically, if is known that the data will be transferred multiple times between host and device and, it might be a good idea to utilize pinned memory to avoid the unnecessary overhead.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

In this work, three different general optimization techniques to achieve overlap between communication and computation were studied. The optimized model implemented in a concurrent fashion using CUDA streams with pinned memory and asynchronous `memcpy` calls was evaluated against the baseline model with varying input sizes and a maximum speed between 5 - 6.5 times was reported. Similarly, a 160% peak increase in the bandwidth utilization from 4.4 GiB/s to 11.4 GiB/s was observed. These optimizations convey potential to be applied on problems of similar nature where multiple data transfers and kernel computations could be accommodated simultaneously.

6.2 FUTURE WORK

Although, research has been quite active in this area [8],[9],[1],[14], there are unique methodologies proposed in different studies including designing an automation data flow, on demand queue of architectures using FPGAs or techniques to efficiently schedule threads on the device by leveraging CUDA memory hierarchy. Our work's focus was particularly on the data transfer bottlenecks from CPU to GPU and proposed utilizing CUDA streams in a concurrent fashion to achieve performance improvements. We expect this work to be extended to include new methodologies or combined with other proven methodologies to optimize the proposed models further. Some potential areas to be included in the future work:

1. Optimizations to include different memory models such as the new Unified Memory mode. Since Unified Memory in CUDA provides a single memory space accessible by all GPUs and CPUs in a system, data can be allocated in unified memory using `cudaMallocManaged()`, which returns a pointer that can access from host (CPU) code or device (GPU) code.

2. The proposed models be performed across different CPU, GPU architectures and compilers for similar analyses to gain insights for addressing the issue.

3. It would also be interesting to study the bi-directional nature of PCIe interconnect and reason behind varying throughput in each direction.

4. One of the optimization approaches in this study described overlapping compute on CPU as well, which could be included in the optimized implementation.

5. The set of experiments presented in this work, when executed with custom parameters of tile/block/grid sizes, memcopy sizes, data layouts and custom chunking of the input data could prove as a benchmark suite that could be used to measure individual systems and architectures and thereby adapting optimizations accordingly.

REFERENCES

[1] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[2] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[3] T. P. Morgan, "The system bottleneck shifts to pci-express," July 2017. [Online]. Available: https://www.nextplatform.com/2017/07/14/system-bottleneck-shifts-pci-express/

[4] A. Huffman and S. P. Engineer, "Nvm express overview & ecosystem update," *Proceedings of Flash Memory Summit*, 2013.

[5] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[6] S. Zhang, Y. Yang, L. Shen, and Z. Wang, "Efficient data communication between cpu and gpu through transparent partial-page migration," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2018, pp. 618–625.

[7] "oneapi gpu optimization guide," 2022.

[8] S. Pabst, A. Koch, and W. Strasser, "Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces," in *Computer Graphics Forum*, vol. 29, no. 5. Wiley Online Library, 2010, p. 1605–1612.

[9] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 140–149.

[10] M. Krotkiewski and M. Dabrowski, "Efficient 3d stencil computations using cuda," *Parallel Computing*, vol. 39, no. 10, pp. 533–548, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781911300094X

[11] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, jan 2020. [Online]. Available: https://doi.org/10.1109%2Ftpds.2019.2928289

[12] A. Silberschatz, J. Peterson, and P. Galvin, "Operating systems," *John Willey*, 1991.

[13] M. Harris, "How to optimize data transfers in cuda c/c++," Dec. 2012. [Online]. Available: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-ccs

[14] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, "On how to accelerate iterative stencil loops: a scalable streaming-based approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–26, 2015.

[15] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "Opencl-based fpga-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1390–1402, 2016.

[16] NVIDIA Corporation, "Nsight system," 2021. [Online]. Available: https://developer.nvidia.com/nsight-systems

[17] T. Bradley, "Gpu performance analysis and optimisation," *NVIDIA Corporation*, 2012.

[18] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Mambretti, P. Rad, and P. Ruth, "Chameleon: a scalable production testbed for computer science research," in *Contemporary High Performance Computing*. CRC Press, 2019, pp. 123–148.

[19] NVIDIA Corporation, "Nvidia profiler user's guide," 2021. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[20] I. M. T. Gleixner, "Performance counters for linux," Dec. 2008. [Online]. Available: https://github.com/torvalds/linux/blob/master/tools/perf/design.txt