UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Memory-oriented optimization techniques in General Purpose GPU programming

---

Author:
Nazar PASTERNAK

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

APPLIED
SCIENCES
FACULTY

Lviv 2022

Declaration of Authorship

I, Nazar PASTERNAK, declare that this thesis titled, “Memory-oriented optimization techniques in General Purpose GPU programming” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“Nvidia, Fuck You!”

Linus Torvalds

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Memory-oriented optimization techniques in General Purpose GPU programming

by Nazar PASTERNAK

Abstract

Effective utilization of the GPU parallel execution potential for GPGPU solutions requires an extensive understanding of the internal GPU memory model. GPU's memory latency hiding techniques differ from those of the CPU, due to substantial design differences. In this thesis, we review the specifics of the GPU memory hierarchy, identify potential memory bottlenecks of GPGPU programs and address them using the CUDA programming model. We provide examples of such solutions along with corresponding performance measurements. As a demonstration of the proposed optimizations, we provide the implementation of the Parallel Failureless Aho-Corasick algorithm for pattern-matching and measure the performance speedup each optimization resulted in. Optimizations discussed in this paper result in almost 2x performance speedup of highly memory-dependant PFAC algorithm.

Acknowledgements

I am very grateful to my supervisor Oleg Farenyuk, who helped me convert my interests into the research topic, and always provided encouragement and guidance throughout my study at the university.

Thanks to the Applied Sciences Faculty of the Ukrainian Catholic University for building such a strong community, always staying up-to-date, and providing students with great opportunities.

I want to thank my family and friends, who always provided lots of support and motivation.

And last but not least, I want to express my gratitude to the Ukrainian Armed Forces for defending our nation and sovereignty.

Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Problem . . . 1
- 1.3 Goal . . . 2

2 Background 3
- 2.1 Parallelism . . . 3
- 2.2 CPU vs GPU . . . 3
- 2.3 GPGPU . . . 3
- 2.4 CUDA . . . 4
  - 2.4.1 Overview . . . 4
  - 2.4.2 GPU architecture and CUDA . . . 4
- 2.5 GPU Memory Model . . . 5
  - 2.5.1 Overview . . . 5
  - 2.5.2 Global memory . . . 6
  - 2.5.3 Registers . . . 6
  - 2.5.4 Local memory . . . 6
  - 2.5.5 Shared memory . . . 6
  - 2.5.6 Constant memory . . . 7
  - 2.5.7 Texture memory . . . 7
- 2.6 Performance metrics . . . 7
  - 2.6.1 Bandwidth . . . 8
  - 2.6.2 Profiling . . . 8
- 2.7 Aho-Corasick for Pattern-Matching . . . 9

3 Related work 10
- 3.1 CUDA Memory layout optimization . . . 10
- 3.2 RegDem: Increasing GPU Performance via Shared Memory Register Spilling . . . 10
- 3.3 Aho-Corasick GPGPU implementations . . . 11

4 Optimization techniques 12
- 4.1 Data transfers . . . 12
  - 4.1.1 Page-locked memory . . . 12
  - 4.1.2 Batching . . . 13
  - 4.1.3 Overlapping data transfers with computation . . . 13
  - 4.1.4 Zero copy and Unified Memory . . . 14

4.2 Memory utilization . . . 15
4.2.1 Coalescence . . . 15
Aligned access pattern . . . 15
Misaligned access pattern . . . 16
Strided access pattern . . . 17
4.2.2 Bank conflicts . . . 17
4.2.3 Local memory and register spilling . . . 19
4.2.4 Registers and shuffle intrinsics . . . 19
SHFL instruction . . . 19
Register cache . . . 20

5 Experimental results **21**
5.1 Pageable vs Pinned data transfers . . . 21
5.2 Overlapping data transfers with computation . . . 21
5.3 Coalescence . . . 22
5.4 Bank conflicts . . . 23

6 Solution: Parallel Failureless Aho-Corasick **25**
6.1 Overview . . . 25
6.2 Initial implementation . . . 26
6.2.1 Profiling . . . 26
6.3 Optimizing data transfers . . . 27
6.3.1 Data batching . . . 27
6.3.2 Staged copy-execute . . . 27
6.3.3 Explicit copies and pinned memory . . . 27
6.4 Optimizing memory utilization . . . 27
6.4.1 Shared memory utilization . . . 28
6.4.2 Batching global memory reads . . . 28
6.5 Results . . . 28

7 Conclusion and Future work **29**
7.1 Conclusion . . . 29
7.2 Future work . . . 29

Bibliography **30**

List of Figures

1.1 Performance constraints [10]. 1

2.1 GPU Memory Model [3]. 4
2.2 GPU Streaming Multiprocessor architecture [21]. 5
2.3 Aho-Corasick trie [11]. 9

4.1 Data transfer from Host to Device [5]. 12
4.2 Sequential vs Staged copy and execute [4] 13
4.3 CUDA Unified Memory [19] 14
4.4 Aligned access 15
4.5 Misaligned access 16
4.6 Aligned vs misaligned access bandwidth. 16
4.7 Bandwidth affected by strided access pattern. 17
4.8 Parallel reduction with bank conflicts 18
4.9 Conflict-free parallel reduction 18

5.1 Sequential copy-execute 22
5.2 Asynchronous copy-execute. 22
5.3 Kernel Performance Limiter tab Nvidia Visual Profiler. 22
5.4 Kernel Memory tab Nvidia Visual Profiler. 23
5.5 Kernel Performance Limiter tab Nvidia Visual Profiler. 23
5.6 Shared Memory Access Pattern tab Nvidia Visual Profiler. 24

6.1 Kernel Profile: initial. 26
6.2 Kernel Profile: optimized. 28

List of Abbreviations

MB	**M**ega**b**yte
API	**A**pplication **P**rogramming **I**nterface
IC	**I**ntegrated **C**ircuit
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
GPGPU	**G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
CC	**C**ompute **C**apability
AC	**A**ho-**C**orasick
PFAC	**P**arallel **F**ailureless **A**ho-**C**orasick
SIMT	**S**ingle **I**nstruction **M**ultiple **T**hreads
SM	**S**treaming **M**ultiprocessor
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
UM	**U**nified **M**emory

Dedicated to people affected by the war

Chapter 1

Introduction

1.1 Context

Starting from the 1990s, the time when CPU clock speed outperformed main memory speed – the problem of memory latency, or the so-called 'von Neumann bottleneck', arose. Since then this "memory wall" has always been a huge aspect of performance in computer systems. Along with increasing the number of transistors in a dense IC, layers of different memory types were added and their sizes increased in an attempt to hide the impact of slow memory on the overall system performance. GPUs, in the contrast to CPUs, can hide memory latency due to their massively parallel architecture. However, the memory problem persists on GPUs too, though in a slightly different form.

1.2 Problem

During the development of a high-performance application, understanding the memory model of the underlying hardware is crucial and often a key to achieving anticipated results. CPU memory hierarchy differs significantly from GPU's, due to substantially distinct designs. Not knowing the specifics of GPUs' memory architecture during development can often result in orders of magnitude slow-down. Moreover, being a lot more specialized than CPUs, GPUs' architectures keep evolving each year by expanding possibilities for performance boost in applications.

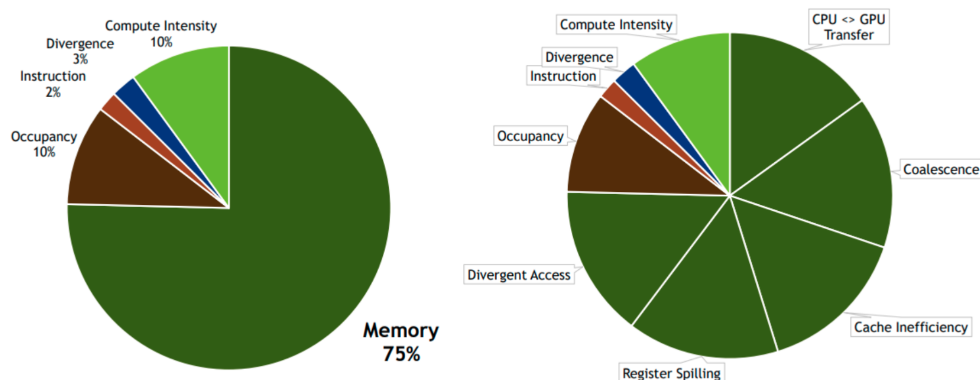

FIGURE 1.1: Performance constraints [10].

In his presentation on CUDA Optimization Tips and Tricks [10], Stephen Jones provides a chart (1.1) depicting the percentage relationship between different potential performance constraints during GPGPU development. Memory bottlenecks take up to 75% of all performance issues. The right half of the figure shows a granular view of potential memory issues.

1.3 Goal

The goal of this thesis is to use the CUDA programming model to review the GPU memory architecture in-depth, identify potential memory bottlenecks of GPGPU programs and address them; provide a comparison of different GPGPU programming approaches, including the newest features of the language accounting for latest architecture changes. We also provide an implementation of the Parallel Failureless Aho-Corasick algorithm, an extension of Aho-Corasick, a powerful pattern-matching algorithm, using the suggested optimizations, and measure the performance speedup each optimization resulted in.

Chapter 2

Background

2.1 Parallelism

Moore's law [24] states that the number of transistors in a dense IC doubles every two years. That law worked great until 2002 when it hit the Power Wall due to physical limitations. The continuation of Moore's law nowadays means not increased performance of an individual processor, but rather more increased number of processors. Thus, application performance depends on how much of the potential parallel processing power is used. GPUs, having colossal parallel nature, act as an extreme example of such development, which, when utilized correctly, yields great performance.

2.2 CPU vs GPU

CPUs are versatile, general-purpose processing units with a few high clock speed, heavyweight cores. CPUs are great at executing sequential, diverse tasks. CPU architectures are latency-optimized and support quick context switching to imitate parallelism. However, due to the relatively small number of cores, they aren't great at performing tasks that require a high degree of parallelism.

GPUs, on the other hand, are specialized to do just that. They were initially designed for the sole purpose of receiving a stream of binary data from the CPU and performing graphics-related operations used to render images. These calculations needed to be executed as quickly as possible multiple times on different data. GPU's instruction sets are optimized for floating-point calculation and matrix arithmetic, resulting in an extreme performance boost in highly parallel simple tasks. With thousands of lightweight cores, divided into groups of multiple Streaming Multiprocessors each containing a large number of registers, GPUs achieve true massive parallelism with latency penalty-free context switching.

2.3 GPGPU

In the early 2000s, the rapid evolution of image quality in 3D graphics pushed GPU capabilities further, such that it was becoming more and more suitable for a variety of applications unrelated to graphics, thus creating a new utilization for GPUs. However, at that time only graphic-specific APIs existed (e.g. OpenGL or DirectX). Using GPU for general purpose applications was hard since each operation needed to be mapped to a graphic equivalent one. This led to the creation of the universal GPGPU programming models (e.g. CUDA/ OpenCL), which solved these challenges [23].

2.4 CUDA

2.4.1 Overview

As was mentioned in the previous section, the rapid evolution of General Purpose GPU programming gave rise to the creation of GPGPU programming models. These GPU APIs generalized the graphics card programming by abstracting away its initial emphasis on exclusively graphics-related computing.

Nvidia was one of the first and by far one of the most successful ones, that attempted to create such a programming model. The first version of Nvidia CUDA was released in 2006 and came out as a convenient software environment, allowing developers to use C++ as a high-level language. Being an integrated, heterogeneous parallel programming system, CUDA programming model allows programmers to write conventional C/C++ code, with a few minor changes concerning device code, since the host code of a CUDA application will be compiled using any existing C/C++ compiler while the device code will be compiled separately using Nvidia CUDA compiler [6][7][21].

2.4.2 GPU architecture and CUDA

FIGURE 2.1: GPU Memory Model [3].

GPUs consist of multiple parallel Streaming Multiprocessors, each containing a number of Streaming Processors (or CUDA cores) and other units, specialized for specific purposes. SMs represent the highly parallel nature of GPU following the SIMT architecture: at any clock cycle, a number of streaming processors execute the same instruction on different data. The number of cores performing the same instruction during one cycle is fixed - 32 cores and is called a warp. Many warps can execute simultaneously on a single SM and no context switches are performed before running the next warp, due to the big register file.

Figure 2.1 demonstrates the internal architecture of a single SM of Pascal GP100 architecture. Each SM has four on-chip memory types: register file, shared memory block, texture, and constant caches, which will be covered in detail in further sections.

As was mentioned above, a CUDA program consists of two parts - host code and device code. Functions, that are run on GPU are called kernels. They usually exhibit large amount of data parallelism. Kernels are executed by GPU threads, that form the CUDA thread hierarchy. Threads form thread-blocks, that have up to 3 dimensions, for easier mapping of the problem onto the kernel execution. Thread-blocks form grids, which can be 1, 2 or 3 dimensional. Each thread-block is guaranteed to be executed on a single SM, so that the access to the same shared memory block is guaranteed. Upon kernel call, programmer specifies the number of threads in a thread-block and thread-blocks in a grid.

A more detailed overview of the GPU architecture in terms of CUDA is provided in official Nvidia documentation [6][7].

2.5 GPU Memory Model

Just as CPU, GPU has multiple memory layers differing in speed and size. However, due to the highly specialized nature of GPUs, these memories reflect distinct from CPU memory model paradigms.

In this section, we explore the GPU memory hierarchy in-depth from the developer's perspective in terms of the CUDA Programming Model [7].

2.5.1 Overview

During kernel execution, GPU threads have access to different types of device memory. Some of these memories are on-chip, others off-chip. Memories vary depending on their speed, size, and purpose. Figure 2.2 shows the memory hierarchy in a thread memory access context.

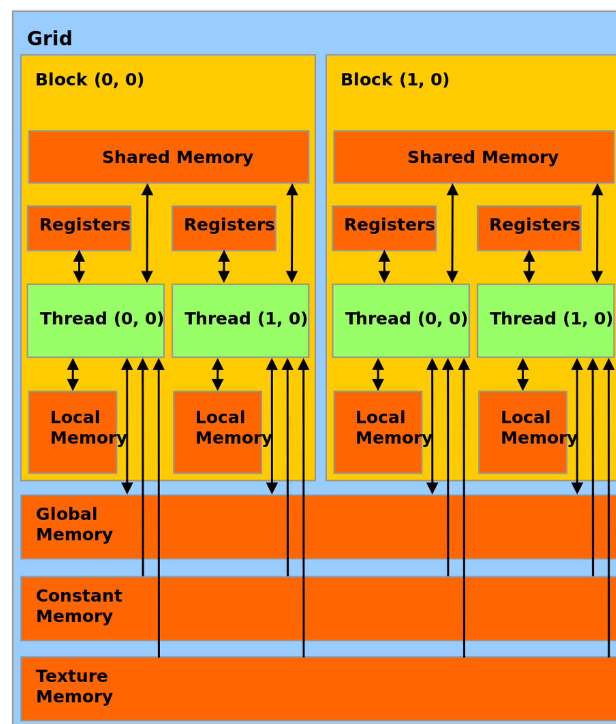

FIGURE 2.2: GPU Streaming Multiprocessor architecture [21].

As we discussed in a previous section, all threads of a thread block are guaranteed to be executed on a single SM. Each SM can execute a number of concurrent thread blocks, depending on the resources required by a block.

The problem of defining the correct amount of thread blocks and threads in each block is called the occupancy problem and usually, about 10% of all performance bottlenecks are related to this issue (1.1). Although occupancy is out of the scope of this thesis, the problem of managing memory resources required by each thread is tightly coupled with occupancy constraint.

2.5.2 Global memory

Main memory, DRAM, a high-latency, high-capacity device frame buffer. Mostly used to store data copied from the host. It is shared by all threads, and thus serves the purpose of the main memory of the device. Accesses to a global memory are usually minimized and coalesced, when possible. One big advantage of such memory – is that there are no restrictions regarding who can access it, so it can be easily served as a means of communication between threads of any thread block.

2.5.3 Registers

Each thread has access to a certain number of registers, the fastest and the most plentiful memory on the SM. SMs contains thousands of registers. For example, Pascal GP100 with CC 6.0 has 65536 registers totaling 256KB. Developers cannot explicitly control which variables are put into registers. It is possible, however, to set the maximum number of registers per kernel, which affects the number of threads and, accordingly, thread blocks that fit in SM. The number of registers used by the kernel can be viewed by verbalizing the ptxas, using the compilation command-line option `--ptxas-options --verbose`.

Registers can be used for very fast inter-warp communication using shuffle intrinsics. We will review them in later sections on optimizations.

2.5.4 Local memory

Each thread has its private local memory, which is a memory chunk physically residing in slow global memory. Local memory is used for some automatic variables. It is a reserve memory in case of register spillage, when the compiler cannot resolve the indexing of variables, or when the size of the local array or structure requires too much register space.

Local memory reflects the same latency characteristics as global memory. Local memory organization however implies that all accesses coalesce, if all threads in warp access the same relative address.

Since Local Memory is usually coupled with Registers, it is useful to understand where the data will end up. In the future sections, we will discuss local memory in-depth and propose potential optimizations.

2.5.5 Shared memory

Unique for GPU type of memory is Shared Memory. It is similar to the L1 cache on the CPU since it physically resides on the chip however, developers can explicitly access it, as well as convert its portions to an automatic L1 cache. Because it is on-chip, it provides much higher bandwidth and lower latency than global memory.

It is allocated per thread block, so threads within a single thread block are able to access it and communicate.

Shared memory, same as registers, is a scarce resource, usage of which is accounted for when deciding the number of thread blocks being able to run on a single SM simultaneously.

The high bandwidth of the shared memory is achieved by arranging it as equally-sized memory banks, that can be accessed simultaneously. Successive Shared memory banks are assigned 32-bit successive words, thus the per-cycle bank bandwidth is 32 bits. This means, that parallel accesses to different banks yield n times higher bandwidth than the bandwidth of a single moduled memory, where n is the number of different banks accessed. These specifics result in roughly two orders of magnitude lower latency compared to Global memory.

If multiple simultaneous accesses that request different memory locations fall into the same bank, a bank conflict occurs and all requests are processed sequentially, which degrades the performance. If accesses that fall into the same bank request the same memory location, broadcasting occurs. Many broadcasts can coalesce into a single multicast.

Since shared memory is managed programmatically, it is up to the developer to set correct access patterns to avoid bank conflicts. In further sections, we will address the addressing issue in detail since memory bank conflicts is a huge performance bottleneck.

2.5.6 Constant memory

Constant memory is a specialized read-only memory residing in the DRAM. It is backed up by a read-only constant cache and is visible by all threads.

Constant memory access for a warp is first split into two requests, one for each half-warp, and then split into a number of separate requests, depending on the number of different memory addresses. Thus, it is useful for broadcasting the result of read requests to many threads. The fact that Constant memory and the corresponding cache are read-only simplifies the hardware cache management and allows for additional optimizations.

2.5.7 Texture memory

Texture memory has a similar purpose to Constant memory. It resides in the device memory, backed up by a read-only texture cache, and visible to all device threads. The texture memory is specialized for 2D spacial locality cases, hence the name.

Since the texture cache is read-only, a thread can read the correct texture memory location only if it has not been updated during the same kernel call. In other words, texture cache is flushed upon each kernel call.

2.6 Performance metrics

Any application development process should be accompanied by persistent measurements and profiling. Any optimization attempt can yield unexpected results, so remeasurement and comparison are necessary. In this subsection, we review performance metrics used during the measurement stage.

2.6.1 Bandwidth

Due to the nature of having massively parallel architectures, GPUs are extremely throughput-oriented. Thus, memory bandwidth is one of the most important performance measures in GPGPU programming. Each GPU has its peak theoretical memory bandwidth. For example, NVIDIA MX150 has 48.06 GB/s main memory bandwidth.

However, as described in the GPU Memory Model subsection, the GPU memory hierarchy consists of several components, which have orders of magnitude bandwidth differences.

A thorough understanding of the memory layout can result in drastic bandwidth gain. Using theoretical bandwidth as a metric for different implementations comparison is not enough and effective bandwidth should be calculated.

Effective bandwidth is calculated by knowing the time it takes for the measured program to complete and data access details [4].

$$EB = \frac{R_b + W_b}{10^9} \div T$$

where R_b - total bytes read, W_b - total bytes written and T - total execution time.

2.6.2 Profiling

To measure the kernel execution time both CPU and GPU timers can be used. Due to the kernel calls being asynchronous, GPU timers provided by CUDA event API should be used to avoid potential synchronization overhead.

Starting with CUDA 5, CUDA Toolkit provides a command-line profiler nvprof, which simplifies the time tracking of each data transfer. It provides the maximum, minimum, and average time for each data transfer executed. nvprof output example is provided below.

```
==12415== Profiling application: ./cuda_memory_optimizations
==12415== Profiling result:
Time(%)      Time   Calls       Avg       Min       Max  Name
 35.29%  184.59ms      17  10.858ms  5.5907ms  85.395ms  [CUDA memcpy HtoD]
 33.84%  177.00ms      17  10.412ms  5.0945ms  81.902ms  [CUDA memcpy DtoH]
 30.87%  161.48ms      17  9.4989ms  4.8167ms  82.054ms  kernel(float*, int)
 44.37%  249.41ms       2  124.70ms  85.429ms  163.98ms  cudaMemcpy
 28.26%  158.83ms       1  158.83ms  158.83ms  158.83ms  cudaMallocHost
 19.42%  109.17ms       2  54.586ms  4.1390us  109.17ms  cudaEventSynchronize
  7.66%  43.046ms       1  43.046ms  43.046ms  43.046ms  cudaFreeHost
  0.07%  411.05us       1  411.05us  411.05us  411.05us  cudaFree
  0.05%  275.54us       1  275.54us  275.54us  275.54us  cudaMalloc
  0.04%  221.90us     101  2.1960us     139ns  154.41us  cuDeviceGetAttribute
  0.02%  121.95us      17  7.1730us  3.0330us  32.621us  cudaLaunchKernel
  0.02%  98.969us      16  6.1850us  1.3330us  65.050us  cudaStreamCreate
  0.02%  89.005us      32  2.7810us  1.8960us  13.925us  cudaMemcpyAsync
```

Another powerful profiling tool included in a CUDA Toolkit is Nvidia Visual Profiler. It traces and analyzes both device code and CUDA API calls and results in an overall performance picture of the application. The Visual Profiler outputs a program timeline, which shows what API calls were made, how long each call took and the kernel execution time. Profiler also includes a detailed guided analysis, which suggests possible optimizations to perform.

2.7 Aho-Corasick for Pattern-Matching

The Aho-Corasick algorithm is a pattern-matching algorithm invented by Alfred V. Aho and Margaret J. Corasick [22]. It effectively is an extension of the Knuth-Morris-Pratt algorithm [8].

It consists of two stages – construction of pattern matching state machine and input string occurrences search. The state machine is a prefix tree, where each node represents a letter of the pattern. Each node also contains a failure pointer – a pointer to another node in the trie (prefix tree), which contains the longest proper suffix of the current state and is a proper prefix of some pattern. The time complexity of the algorithm is O(n + m + z), where n is the length of the input text, m is the sum of all lengths of patterns and z is the total number of pattern occurrences in the input text. The space complexity is equal to the sum of the lengths of all patterns, however usually patterns share a lot of prefixes, resulting in tree overlaps, thus saving memory space.

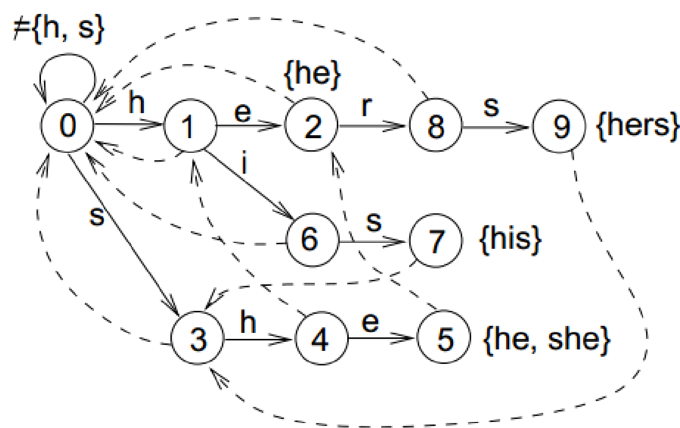

FIGURE 2.3: Aho-Corasick trie [11].

Figure 2.3 shows a visualization of the trie. Failure links are dotted lines. For example, if we were to process the input string "shers", on the state 'e' we would fail to node number two and eventually find word hers as well, without backtracking.

Aho-Corasick is widely used in the medical biotechnology domain for effectively searching for particular markers in genomes. Oftentimes such projects already utilize GPUs for efficient model training. Providing a GPU-optimized version of Aho-Corasick would be relevant and helpful for further advances in the field.

Chapter 3

Related work

Papers regarding GPGPU have been written long before specialized GPGPU programming models, such as CUDA, appeared. As well as works regarding memory bottleneck, or von Neumann bottleneck, have been conducted for a long time now. Thus, there is a significant amount of research in the area of memory optimizations in GPGPU programming.

In this chapter, we will choose a few studies, that are the closest to ours content-wise, compare them and point out their pros and cons.

As a target algorithm for proposed optimizations, we use the Aho-Corasick algorithm. Hence, we will review papers proposing GPGPU Aho-Corasick implementations.

3.1 CUDA Memory layout optimization

The memory layout optimization study presented by Siegel et al. [20]; targets the acceleration of existing applications performed on the CPU by extracting computational extensive kernels to the GPU. The main optimization focus of this paper is on the improvement of the application performance by changing the user-defined data structures, accounting for all specifics of the GPU memory model. The authors propose different approaches to memory utilization, provide benchmarks of each optimization and optimize the Gravit application.

We follow a similar structure, by reviewing different optimization techniques, measuring their effectiveness, and then applying them to optimize the Aho-Corasick algorithm. The main goal of our study is to cover a broad amount of memory-related optimizations, while the paper by Jakob Siegel and others mainly focuses on the memory layout of the data structures and loop unrolling.

3.2 RegDem: Increasing GPU Performance via Shared Memory Register Spilling

The study by Sakdhnagool et al. [18] aims to optimize occupancy, a common GPGPU application's performance limiter, using memory-related optimizations. The Paper proposes a GPU assembly translation technique, which spills excessive registers to the underutilized shared memory, rather than local memory, improving occupancy, thus performance. The study confronts common memory-related issues, such as Register pressure and Shared Memory Bank Conflicts, addressed in our research. Authors claim to have achieved up to 1.18x speedup in 7 out of 9 benchmarks over the alternative approaches.

In contrast to our approach, this study proposes an optimization to a specific part of memory-related performance issues, while we propose optimization approaches, which address many potential performance bottlenecks.

3.3 Aho-Corasick GPGPU implementations

Studies by Lin et al. [14] and Kouzinopoulos et al. [13] provide the Aho-Corasick algorithm for string matching implementations on the GPU. Lin et al. [14] propose the Parallel Failureless AC algorithm, which we use in our demonstration.

Kouzinopoulos et al. [13] review different string matching algorithm implementations and optimization techniques on GPU using CUDA. They implement PFAC described by Lin et al. [14] with various optimizations. In contrast to our study, they emphasize on PFAC optimization, while we provide various memory-oriented GPGPU optimizations with an optimized PFAC as an example.

Chapter 4

Optimization techniques

The main goal of all memory optimizations on a GPU is to maximize the effective bandwidth. That is – maximizing utilization of fast memory and minimizing slow memory usage as much as possible. We provided a detailed overview of different types of memories resident on a GPU in the Background section. In this section, we review all potential memory-related bottlenecks that have a major effect on the performance and propose solutions with example implementations.

4.1 Data transfers

When measuring the execution time of a running kernel on GPU we should always consider the time it takes for data to migrate from Host to Device memory and backward over the PCI-e bus. It is especially important when deciding whether to use CPU or GPU implementations of a solution since sometimes the transfer overhead might compensate for the speed gain on GPU and result in an even worse performance outcome. The reason for that is comparably low PCI-e bandwidth. For example – the theoretical main memory bandwidth of the GPU we use, NVIDIA MX150, is 48.06 GB/s compared to the 3.94 GB/s theoretical bandwidth on the PCI-e 3.0 x4 bus, used on MX150. Hence, all Host-Device communications should be minimized. In the following subsections, we review possible optimizations regarding data transfers.

4.1.1 Page-locked memory

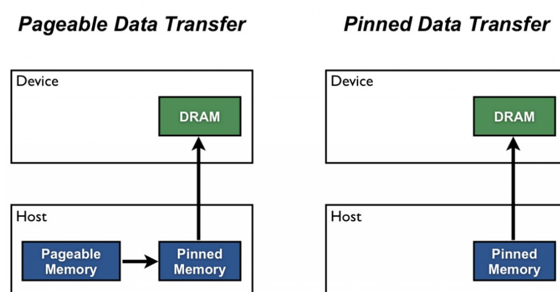

FIGURE 4.1: Data transfer from Host to Device [5].

During the data transfer between Host and Device GPU can access Host memory directly only from pinned, or page-locked, memory. Page-locked memory is one, which cannot be paged out to the secondary memory. Hence, if the data being copied is not pinned, the CUDA driver has to copy that data from pageable to temporary page-locked memory on the host, from which the GPU can DMA the

data (Figure 4.1). Excessive data copying on the host can be prevented by allocating page-locked memory directly. This way, during the data transfer, the GPU can DMA the Host memory right away. Although pinned transfers are faster than non-pinned transfers, the decision on how much memory should be allocated this way has to be reasonable. Allocating too much page-locked memory can degrade the overall system performance, due to the potential lack of free memory and the inability to page out.

4.1.2 Batching

As was already mentioned — data transfers are slow and should be minimized. One approach to do so is to batch many smaller transfers into a single big transfer. This includes flattening node-based data structures, combining unrelated data into one array and unpacking on the device, etc. Before issuing a transfer, memory pinning of the resulting data should be considered. CUDA provides an API for transfering 2D and 3D arrays - `cudaMemcpy2D(...)` and `cudaMemcpy3D(...)`.

4.1.3 Overlapping data transfers with computation

Most data transfer API calls result in a blocking manner, meaning the control to the host thread is returned only upon the completion of the call. However, sometimes data can be split up into chunks and processed in multiple stages by multiple kernel calls. This behavior can be achieved with non-blocking data transfer functions, such as `cudaMemcpyAsync()`. Even though we add a slight data transfer overhead due to making multiple API calls instead of one, given a time-consuming kernel this loss is compensated.

```
int size = N * sizeof(float) / numStreams;

for (i=0; i < numStreams; ++i) {
    offset = i * N/numStreams;
    cudaMemcpyAsync(a_d + offset, a_h + offset,
                                      size, dir, stream[i]);
    kernel<<<N/(numThreads * numStreams), numThreads, 0,
                stream[i]>>>(a_d + offset);
}
```

Piece of code above is a demonstration of the staged concurrent copy and execute technique. The data is divided into nStreams chunks and asynchronously copied to the device. Both copy and kernel calls are assigned a unique stream id since all operations inside a single stream are executed sequentially so that each kernel call will wait until the corresponding copy is completed. Figure 4.2 demonstrates the timeline comparison between sequential and staged copy and execute techniques.

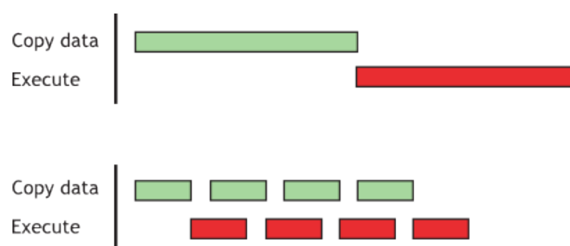

FIGURE 4.2: Sequential vs Staged copy and execute [4]

4.1.4 Zero copy and Unified Memory

In previous subsections on data transfers, we reviewed the optimal ways to copy data between host and device. CUDA Toolkit also provides features, which allow starting kernel without needed data resident in the device memory: Zero Copy and Unified Memory.

Zero Copy is a feature, which allows GPU threads directly access the host main memory through a device pointer. This feature only works with page-locked memory. It mostly is advantageous on the integrated GPUs, where GPU and CPU memory are physically the same. Zero Copy can also be useful in specific cases, such as data not fitting in the device main memory or as a means of communication between host and device.

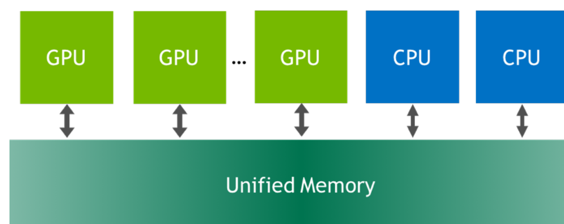

FIGURE 4.3: CUDA Unified Memory [19]

A similar but a lot more powerful feature is Unified Memory space supported on devices with CC 2.0 and later. With UM both host and all resident devices share one virtual address space, meaning the allocated Unified Memory block can be accessed both from the CPU and GPU using the same pointers. Pascal architecture pushes this feature further with its Page Migration Engine, which implements support for virtual memory page-faulting and migration on a hardware level.

In short, since pre-Pascal GPUs did not support page faulting, all necessary pages were migrated to the GPU memory just before the kernel launch. Thus, potential migration overhead.

Pascal architecture introduces the possibility of starting the kernel without all necessary pages being resident in the device memory and faulting on non-resident pages during the kernel execution. This is especially useful when it is unknown which part of data will be used during the kernel execution since pre-loading whole chunks of data is avoided.

This feature will not necessarily yield a performance boost, since a well-written program, which utilizes async copy-execute will mostly outperform UM. But it is indeed a useful tool, which simplifies CUDA application development. One such example is the elimination of deep copies. Before UM was introduced, transferring a node-based data structure was a mess. Developers either had to use the above-mentioned Zero copy, which would be PCIe link bounded, or flatten these structures, which is not always optimal.

Knap et al. [12] review the cases when UM utilization outperforms manual copying.

4.2 Memory utilization

In the Background section, we reviewed different types of memories accessible during kernel execution. These memories reflect distinct usages. Specifics of each type of memory should be acknowledged during the GPGPU application development.

In this section, we will review the main performance issues regarding access to each memory type and propose optimizations to be applied in order to boost the application bandwidth.

4.2.1 Coalescence

GPU's global memory, DRAM, is too slow for sequential access. Thus, when a location is accessed, many consecutive locations, including the one requested, are fetched. Exploiting this access technique will lead to a great application performance boost.

All global memory accesses during kernel execution should be optimized for maximal coalescence. All threads within a warp execute the same instruction. When an instruction is a load from global memory, the hardware detects whether the memory accesses of threads of a warp are consecutive and then coalesces the access into as few transactions as possible. On devices with CC 6.0 or above, the hardware coalescence algorithm will coalesce the warp's memory accesses into as few 32-byte transactions as possible.

Aligned access pattern

The simplest and most favorable access pattern is when all threads of a warp access consecutive memory locations in a 32-byte aligned array.

If, for example, each thread accesses 4-byte words in a manner described above, that warp's memory accesses will be coalesced into 4 32-byte transactions, satisfying all threads' requests. The same four transactions will be performed, if the threads' memory accesses were in any way distributed across four 32-byte segments.

An example of such access is provided on Figure 4.4 taken from Cuda Toolkit Documentation [7]. All threads' memory requests of a warp are satisfied with four fetches from global memory, or, if that memory was cached, one fetch from the cache.

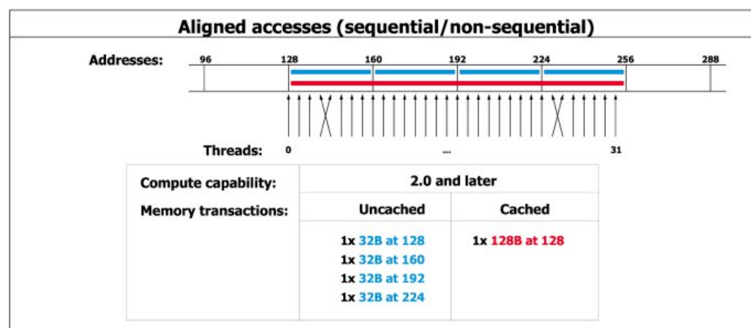

FIGURE 4.4: Aligned access

Misaligned access pattern

A misaligned access pattern usually adds redundant memory access, reducing overall bandwidth. Figure 4.5 from Cuda Toolkit Documentation [7] demonstrates such a pattern: all threads of a warp accessing 4-byte words in a segment, which is not 32-byte aligned results in 5 global memory loads or 2 cache fetches.

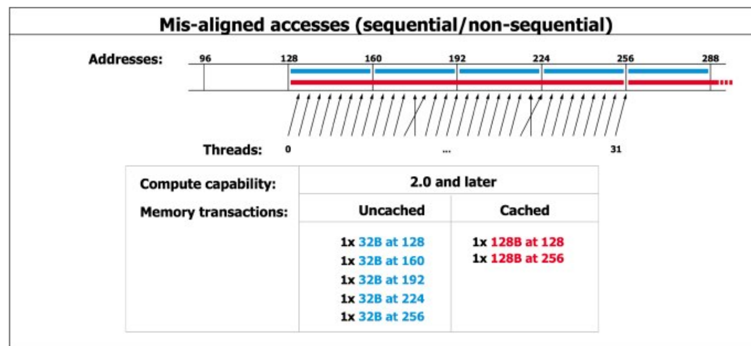

FIGURE 4.5: Misaligned access

```
__global__
void misalignedPattern(float *out, float* in,
                               int alignmentOffset) {
    int i = blockIdx.x * blockDim.x + threadIdx.x
            + alignmentOffset;
    out[i] = in[i];
}
```

CUDA Toolkit Documentation [4] provides Figure 4.6, which demonstrates how a misaligned access pattern provided in a code chunk above decreases bandwidth on Telsa V100 GPU. The bandwidth of the misaligned access pattern drops from approximately 790 GB/s to around 700 Gb/s. However, this demo code contains a high degree of memory reuse of adjacent warps. In cases, when cached data cannot be reused by other warps, the bandwidth drop would be around 20%.

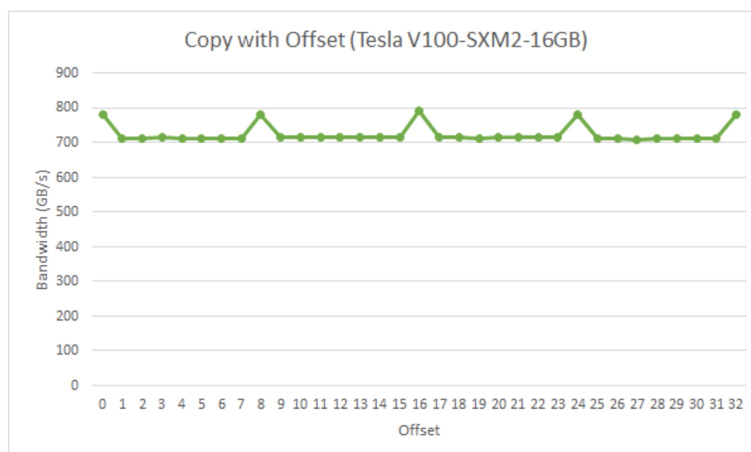

FIGURE 4.6: Aligned vs misaligned access bandwidth.

Alignment should always be considered, especially, when declaring new types. Some CUDA built-ins account for the alignment issue by explicitly aligning memory. For example, the type `double2` is defined with `__align__(16)` attribute, which aligns

variables of this type in memory and allows for coalesced access. Another example is the `cudaMalloc()` memory allocation API, which allocates memory chunks at least 256-bytes aligned. The number of threads in a block should be also chosen with alignment and coalescence in mind and generally should be multiple of the warp size.

Strided access pattern

Another common access pattern, which occurs frequently when working with multidimensional data, is strided pattern. When threads within a warp access data with some stride, global memory transactions are still fetching consecutive memory locations, resulting in large chunks of memory not being used. For example, when threads access data with a stride of 2, bandwidth drops by 50%. Figure 4.7 from CUDA Toolkit Documentation [4] shows a bandwidths drop as access stride increases.

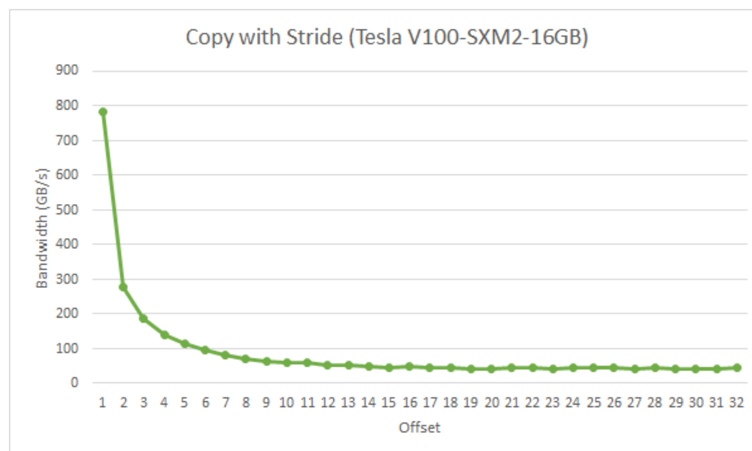

FIGURE 4.7: Bandwidth affected by strided access pattern.

This issue can be resolved using Shared memory, which we discuss in the following sections.

4.2.2 Bank conflicts

As we mentioned in the Background section, on-chip shared memory achieves high bandwidth due to its organization of equally-sized banks, which can be accessed simultaneously. However, to exploit its speed, thread access patterns to shared memory should be optimized.

Generally, shared memory utilization is performed by reading necessary global memory locations and writing them into the shared memory. This way global accesses number is reduced since the next memory fetches will be performed from shared memory.

Shared memory can also serve as a means of communication between threads in one block. However, in this case, synchronization should be taken into account.

Mark Harris [9] in his presentation on Optimizing Parallel Reduction in CUDA, demonstrates bank conflict removal as one of the possible optimizations.

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
     if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

In the piece of code above each thread accesses two locations of the shared memory with a stride, depending on a loop iteration, sums them, and stores the result at the leftmost location. For example, if we were summing 64 integers, the first iteration would resolve in a 2-way bank conflict, since the first thread would access index 0, which resides in bank 0 and the 16th thread would access index 32, which also resides in the 0th bank. Figure 4.8 [9] demonstrates the algorithm on each iteration.

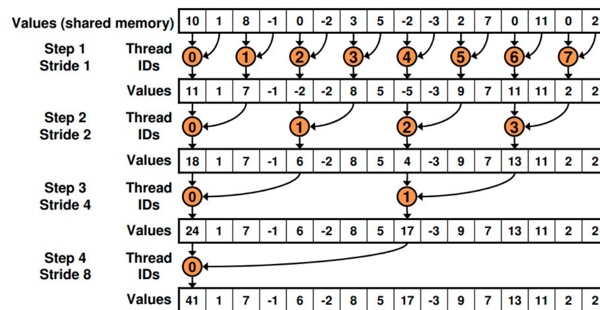

FIGURE 4.8: Parallel reduction with bank conflicts

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

The solution to this problem is sequential addressing. Threads access sequential memory locations, achieving a conflict-free access pattern. Figure 4.9 [9] demonstrates the conflict-free version of the algorithm on each iteration.

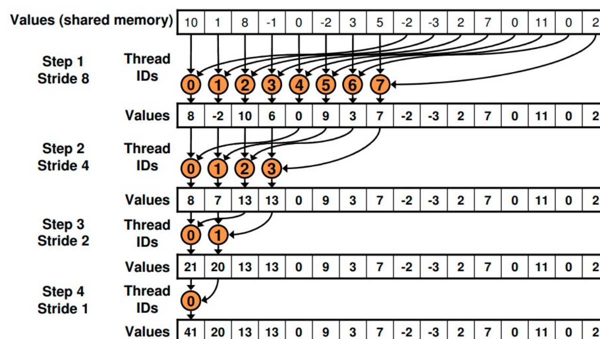

FIGURE 4.9: Conflict-free parallel reduction

4.2.3 Local memory and register spilling

Optimizations related to local memory are often about how to avoid local memory usage. This is because local memory in fact resides in global memory, thus possessing an expensive access time.

Local memory only stores automatic variables. A variable is decided to be held in local memory if register pressure is too high or if a variable is a dynamically indexed array.

Local memory accesses are cached in the L1 cache and the addressing is managed by the compiler, resulting in mostly coalesced accesses.

To find out whether the kernel uses Local Memory, nvcc compiler option `-Xptxas -v,-abi-no` can be used, which will print the local memory bytes used by each kernel.

Avoiding local memory usage is often hard. If possible, indexing should be made explicit for the compiler to deduce statically. Since local memory uses an L1 cache, increasing its size can help reduce expensive memory accesses; profiler counters can be used to analyze the L1 cache hit rate.

Generally, L2 cache accesses should be avoided, due to them having relatively high latency. Nvidia Visual Profiler allows to query of L2 cache accesses, caused by local memory access (and L1 cache miss); using this feature L1 cache to shared memory ratio can be tuned for the best L1 cache hit rate.

Caching for global memory loads can be turned off using compiler option `-Xptxas -dlcm = cg`, which will leave more room for local memory.

Another option is increasing the register count or lowering the thread count, however, this will likely reduce occupancy, which in turn reduces parallelism. But eventually can lead to better performance, if memory latency is the most problematic.

Paulius Micikevicius, in his presentation on Local Memory and Register Spilling [15], reviews the problem of register spilling and Local memory utilization with profiler analysis examples.

4.2.4 Registers and shuffle intrinsics

We have already touched on registers in some previous optimization strategies. Register count plays a role in occupancy. Compiler option `-maxrregcount` sets the maximum number of registers used per thread. It should be tweaked to achieve the best occupancy and memory bandwidth.

SHFL instruction

Nvidia Kepler architecture introduced new means of inter-thread communication within warp using registers – SHFL instruction. Shuffle allows threads within a single warp to exchange variables. This optimization targets shared memory usage, which is very often a bottleneck since the register file in modern GPUs is much bigger than the shared memory size.

Apart from potentially increasing the occupancy, shuffling will also be faster, since it requires one instruction versus three when using shared memory – write, sync, and read. Shuffle usage also eliminates unnecessary thread synchronization within a block – `__syncthreads`, which is required when using shared memory.

Shuffle intrinsics allow threads to access other threads' variables using lane id. A lane is a coordinate of a thread within a warp: `threadIdx.x % 32`. Different intrinsics are provided differing in lane id patterns. For example, the simplest version

is `__shfl_sync()`, which takes in a lane id of the thread, whose variable we want to read. `__shfl_up_sync` and `__shfl_down_sync` will read the variable of the thread with lane id upper/lower than the calling thread's lane id by the parameter `delta`. `__shfl_xor_sync` will perform an XOR operation of the calling thread's lane id with the parameter `laneMask`. Each intrinsic takes in the `mask` parameter, which represents which threads of a warp participate in a shuffle, where each set bit of `mask` indicates the activeness of the corresponding lane. The `width` parameter of the shuffle intrinsics is used to calculate the lane id of the source thread and if its value is less than the size of a warp, intrinsic behavior is changed correspondingly.

Register cache

Shuffle instruction usage allows for efficient register cache implementation. Register cache is basically a warp-level cache, which utilizes shuffle instructions. Register cache can be used to replace shared memory cache usage. The idea of the implementation is that each thread holds and manages its cache portion in an array, which is stored in register memory – the same idea as for the common shared memory usage. Ben-Sasson et al. [2] demonstrate the usage of intra-warp register cache for polynomial multiplication with a comparison with corresponding implementation utilizing shared memory. They show that the register cache version is 50% faster than the shared memory one.

Chapter 5

Experimental results

In this chapter, we provide measurements of different optimization techniques we have discussed. We chose to include only those approaches, which demonstrate their efficiency on small examples and complement their description in the Optimization Techniques chapter.

To measure the results we use metrics reviewed in the Performance metrics section of the Background chapter: effective bandwidth, CUDA event API, nvprof, and Nvidia Visual Profiler. The code of each measurement can be found on our GitHub repository [16].

All measurements were run on Nvidia MX150 GPU of CC 6.1 using nvcc 11.6 release version.

5.1 Pageable vs Pinned data transfers

```
Pageable vs Pinned memory transfer comparison
  Transfer size in MB: 16
Pageable transfer:
  Host to Device Bandwidth in GB/s: 2.89555
  Device to Host Bandwidth in GB/s: 3.01612
Pinned transfer:
  Host to Device Bandwidth in GB/s: 3.13566
  Device to Host Bandwidth in GB/s: 3.28345
```

The result of increased bandwidth is noticeable and quite significant for memory-bound applications.

The pageable/pinned bandwidth difference will increase on newer Graphics Cards; in this data transfer, the bandwidth is limited by the PCI-e interconnect, not the host->host transfer, which we avoid, when using pinned memory. However, MX150 uses PCIe3, which is twice as slow as newer PCIe4, used in modern GPUs, while host->host transfers obviously will not become twice as fast. So using PCIe4 the pageable/pinned difference will become even more noticeable.

5.2 Overlapping data transfers with computation

```
Sequential vs Asynchronous copy-execute:
Sequential copy-execute completed in 30.7331 ms
Asynchronous copy-execute completed in 16.9258 ms
```

This optimization resulted in a great performance boost – asynchronous copy and execution, using 4 streams, finished almost 2x faster than the sequential version.

To better visualize the optimization technique, Figures 5.1 and 5.2 from the Nvidia Visual Profiler are provided.

Figure 5.2 demonstrates how the kernel execution and data transfers are executed concurrently in different streams.

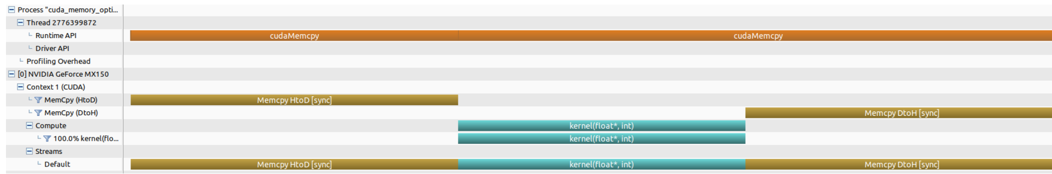

FIGURE 5.1: Sequential copy-execute

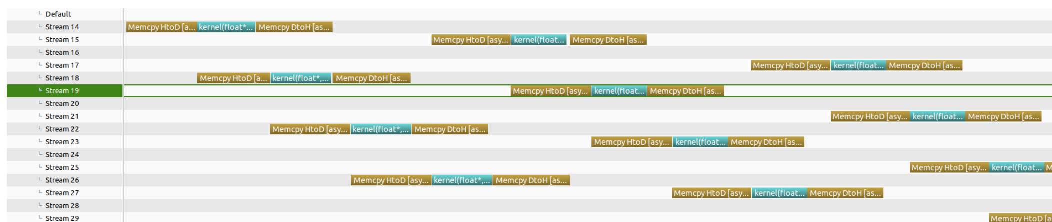

FIGURE 5.2: Asynchronous copy-execute.

The number of streams is dependent on the problem set and should be tweaked. Having increased the input and number of streams from 4 to 16 we now achieve a 2.3x performance boost.

```
Sequential copy-execute completed in 243.468 ms
Asynchronous copy-execute completed in 105.967 ms
```

5.3 Coalescence

To reproduce a problem of uncoalesced accesses, we chose a simple problem of matrix multiplication ($C = AA^T$), since a simple non-optimized solution suffers from a large global memory bandwidth bottleneck. This is due to the fact, that we access matrix entries in a column sequence of the matrix's transpose, which results in strided accesses.

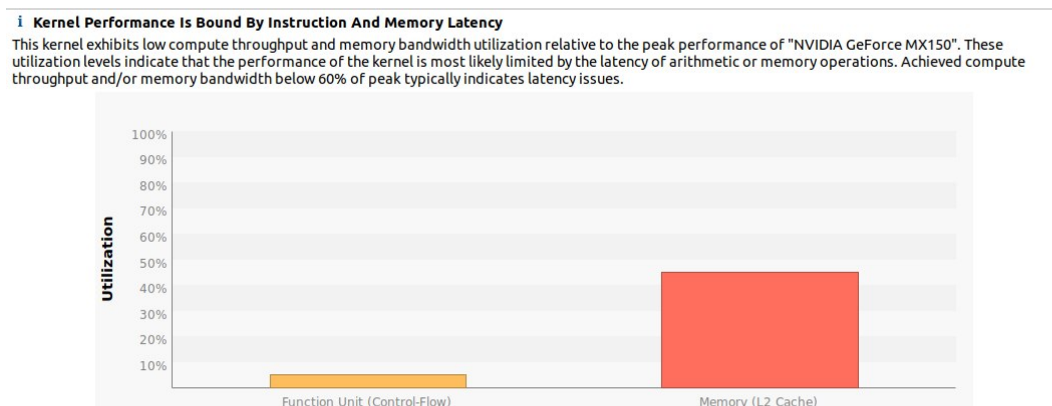

FIGURE 5.3: Kernel Performance Limiter tab Nvidia Visual Profiler.

We used Nvidia Visual Profiler to analyze the simple implementation. Figure 5.3 demonstrates the Kernel Performance Limiter tab of the Profiler output, which notices latency issues of the kernel.

If we take a look at the Kernel Memory section (Figure 5.4), we can see that the Device Memory utilization is below Low; the bandwidth is only 624 MB/s when the peak theoretical bandwidth is 48.06 GB/s. This is not a surprise, since in our matrix multiplication implementation we perform lots of strided accesses. In the Strided access pattern subsection of the Optimization techniques chapter, we provide a graph, which demonstrated the effect of strided access pattern on bandwidth.

Device Memory

Reads	9115	23.084 MB/s	
Writes	237405	601.227 MB/s	
Total	246520	624.31 MB/s	Idle Low Medium High Max

FIGURE 5.4: Kernel Memory tab Nvidia Visual Profiler.

To resolve this issue we employ shared memory to "cache" matrix tiles. The performance boost of such optimization is around 8x.

```
Non-optimized matrix multiplication completed in 12.7904 ms
Optimized matrix multiplication completed in 1.58725 ms
```

5.4 Bank conflicts

By utilizing shared memory to avoid uncoalesced accesses in the simple matrix multiplication implementation we introduced a problem of bank conflicts. Bank conflicts occur when we copy tiles from the global memory into shared memory. This is because one of the shared memory arrays represents a tile of a transposed matrix; because of that, entries should be written to shared memory in columns, which results in each thread of the warp hitting the same bank (size of tile = 32).

When we profile the application with Nvidia Visual Profiler, the Kernel Performance Limiter (Figure 5.5) indicates problems with Shared memory bandwidth, which is exactly what we anticipated.

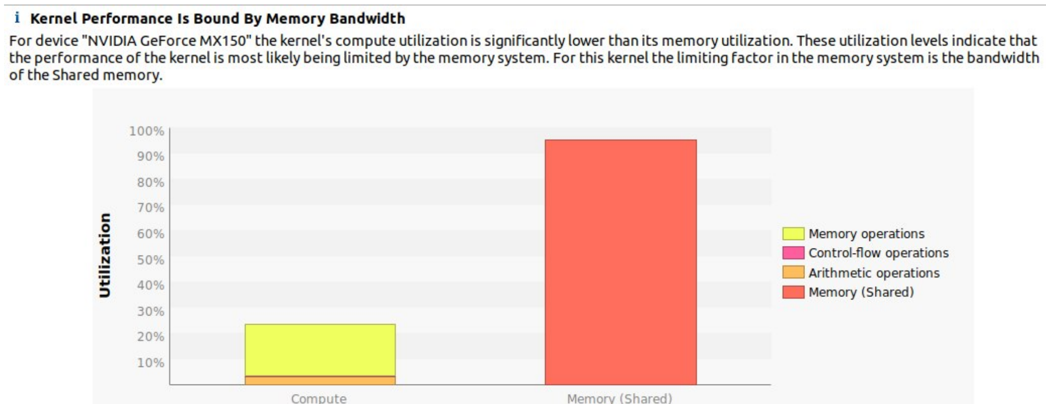

FIGURE 5.5: Kernel Performance Limiter tab Nvidia Visual Profiler.

Shared Memory Access Pattern tab states, that kernel utilizes inefficient access pattern to Shared Memory (Figure 5.6).

Shared Memory Alignment and Access Pattern
Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.
Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access. More...

Line / File	NA
NA	Shared Store Transactions/Access = 32, Ideal Transactions/Access = 1 [2377728 transactions for 74304 total executions]

FIGURE 5.6: Shared Memory Access Pattern tab Nvidia Visual Profiler.

To resolve this issue, we can simply pad our problematic shared memory array, such that it has an extra column (32x33). This way all warp accesses are shifted by one, hence no bank conflicts occur.

After this optimization, the performance increased by another 25%.

```
Non-optimized matrix multiplication completed in 12.7904 ms
Optimized (gmem) matrix multiplication completed in 1.58725 ms
Optimized (smem) matrix multiplication completed in 1.17835 ms
```

Chapter 6

Solution: Parallel Failureless Aho-Corasick

We chose to use Aho-Corasick as a demonstration of discussed optimizations since it is one of the leading algorithms for pattern-matching. It offers the linear time complexity for any input and usually requires relatively low additional space. The trie used during the matching stage can also be precomputed and reused many times, saving execution time. However, Aho-Corasick is highly memory-dependent, which provides the opportunity to optimize its memory utilization.

To demonstrate proposed optimizations we use the PFAC [14] extension of the AC algorithm; the main difference between PFAC and ordinary Aho-Corasick is the absence of failure connections. These connections can be left out since in PFAC each input string character is taken by a separate thread as a starting point for trie traversal.

As demonstrated by Figure 1.1, GPGPU application performance constraints include other than memory-related issues, such as occupancy, thread divergence, etc. Since in this study we emphasize only memory-oriented techniques, we chose the PFAC algorithm. PFAC does not suit the ideal GPU execution model well, since it will always contain a high amount of divergence and latency issues. However, most of the memory utilization optimizations we have reviewed can still be applied to PFAC.

The following sections describe step-by-step optimization application along with description and profiling. All code is available on the GitHub repository [17]; each optimization was added as a separate commit.

6.1 Overview

All measurements are conducted on Nvidia MX150 GPU of CC 6.1 using nvcc 11.6 release version.

The input data was taken from the www.gutenberg.org website. Ten files are parsed and a trie is built from the words of all files. The same input of ten files goes as an input string to the matching algorithm.

To demonstrate proposed optimizations only the matching algorithm was used, however, parallel trie construction on the GPU is considered (see Future Work).

To compare different implementations, we measure the time taken to copy/prefetch all necessary data to the device, run the kernel, and copy the results back to the host.

To validate the results we used python implementation of the Aho-Corasick algorithm provided in the *ahocorapy* package [1] to precompute the correct results.

6.2 Initial implementation

The initial implementation used the following *trie* struct, representing a single node of a trie. Each instance has an array of 26 pointers to child-nodes and an id. Total size of the struct is 216 bytes for 64-bit system.

```
struct trie {
    struct {
        trie *ptr = nullptr;
    } next[26];
    int id = -1;
};
```

The initial kernel had the following signature:

```
__global__ void matchWords(const char *inputStr,
                           int *matched,
                           trie *root,
                           int size);
```

$constchar * inputStr$ - a global memory pointer to the input string;
$int * matched$ - a global memory pointer to the resulting output array
$trie * root$ - a global memory pointer to the root of the trie
$intsize$ - size of the input string

Initial implementation used Unified Memory allocations for input string and the trie.

6.2.1 Profiling

The time it took for the initial implementation to complete was around 29 ms. We used the Nvidia Visual Profiler to pinpoint major issues.

FIGURE 6.1: Kernel Profile: initial.

Due to warps constantly accessing global memory - the majority of stalls were due to memory dependency.

6.3 Optimizing data transfers

PFAC algorithm is highly dependent on global device memory; the whole trie must be visible to each thread, each byte of the input string is accessed once, and results are written to global memory using atomic operations. Thus, data transfers should be optimized as much as possible to reduce the memory bandwidth limitation.

6.3.1 Data batching

To begin with data transfer optimization we firstly optimized our trie data structure. Usage of Unified Memory allowed us to use pointers to child nodes, which was more convenient on the first iteration. However, the size of the pointer being 8 bytes resulted in a trie struct having a size of 216 bytes; hence, our example trie took up around 17 MB of device memory space. Lowering the size of the struct would enhance data transfer speed and increase cache efficiency. Instead of using pointers to child nodes, we converted our code to use indices.

```
struct trieFlattened {
    int next[26]{};
    int id = -1;
};
```

This decreased the trie size down to 8.5 MB resulting in a 1.3x speedup; execution time approx. 22ms

6.3.2 Staged copy-execute

The next optimization we performed was Staged Copy-Execute for Memcpy/Kernel overlap.

Since at any point of execution a thread may need any node of the trie, only the input string could be split up and transferred in chunks. Each chunk was asynchronously copied to the device memory in a separate stream and a kernel call was issued in the same stream.

This optimization resulted in another 1.1x speedup; execution time approx. 20ms

6.3.3 Explicit copies and pinned memory

As we mentioned previously, Unified Memory was initially used for convenience. However, even with explicit prefetching, Unified Memory has a bit of overhead compared to explicit copies. Moreover, we have previously removed the usage of pointers in the trie structure, so the shallow copy would do just fine.

To avoid unnecessary CPU-CPU copies and allow GPU directly access host data, we used pinned memory to allocate host data.

This optimization resulted in an execution time speedup of another 1.1x; approx. 18ms

6.4 Optimizing memory utilization

Currently, the kernel only uses global memory pointers. Accessing data in such a manner results in high memory latency, thus memory dependency stalls. Moreover, current global memory access pattern is far from being optimized, since we only read one byte per access, when requesting the character of the input string. These problems can be resolved by utilizing Shared Memory.

6.4.1 Shared memory utilization

Firstly, we want to decrease memory dependency stalls during kernel execution. We can do so by using Shared Memory to copy chunks of the input string to the shared buffer. Each block allocates a shared buffer and copies a chunk of the input string from global memory to shared memory. Then each warp proceeds with its normal computation, however using a shared array buffer to access data now. Upon finishing matching, the block refills the shared array buffer with new data.

Since a block may access global data with an offset of grid size * shared memory per block, staged copy-execute becomes inefficient – memory needed by a block resides in chunks of the size of shared memory per block with a step of size * shared memory per block; each kernel would have to wait for all required chunks to be copied.

This optimization resulted in another performance boost of 1.1x.

6.4.2 Batching global memory reads

In all previous implementations, global memory reads were inefficient due to one-byte requests per memory access. This now can be solved when filling the shared memory buffer.

We cast the input pointer to *uint*4 type, which allows the compiler to generate a correct vector load instruction and fetch more data with fewer requests, thus utilizing memory bandwidth more efficiently.

We then fill up the corresponding shared memory locations using *uint*4 words; This optimization reduced the execution time to around 15ms.

6.5 Results

The cumulative speedup amounts to about 1.93x, which is a decent result, accounting for the fact, that algorithm still uses highly divergent branches, lacks occupancy optimization, and is overall highly memory-dependent, thus limited by the memory bandwidth.

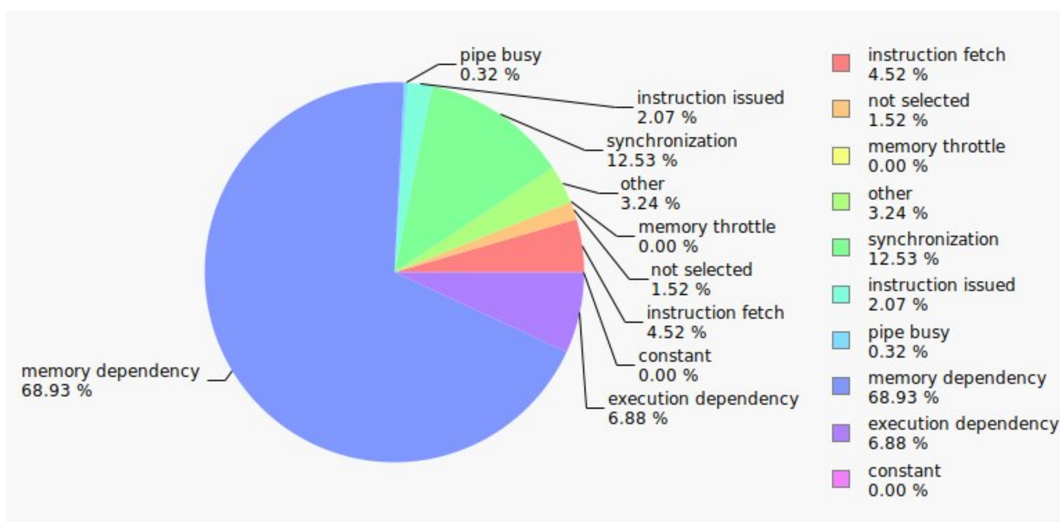

FIGURE 6.2: Kernel Profile: optimized.

The Nvidia Visual Profiler reports 69% of all stalls being due to memory dependency, compared to 87% before applying optimizations.

Chapter 7

Conclusion and Future work

7.1 Conclusion

GPGPU programming models along with GPU hardware have had a great expansion over the past decade and will continue to grow. Both hardware manufacturers and software engineers are aware of the memory bottleneck and strive to optimize it, however, so far no hardware or compiler optimizations are able to ideally utilize the memory resource.

In this work, we reviewed the GPU memory hierarchy, pinpointed potential memory-related performance bottlenecks, and provided optimization approaches to resolve them along with measurements and sample implementations.

For our experiments, we used one of the latest CUDA versions and Pascal architecture GPU, which perform a decent amount of optimizations. Nonetheless, program-level optimizations often resulted in a crucial performance boost.

To demonstrate the effectiveness of the proposed optimizations, we implemented the leading pattern-matching algorithm Aho-Corasick to execute on the GPU device. Then, along with constant profiling, applied the optimization approaches we discussed. As a result, we managed to get an almost 2x performance boost compared to the non-optimized version, despite Aho-Corasick being a highly data-dependent algorithm.

7.2 Future work

For the future work, we consider investigating memory-related issues on the newest Nvidia GPU architectures:

1. perform the same benchmarks on newer hardware;
2. study the architectural changes of newer GPUs and adjust optimization approaches;
3. move trie construction to the device (possible on Turing and newer architectures; development in *device-trie-construction branch* of our GitHub repo [17]);

Bibliography

[1] abusix. *ahocorapy - Fast Many-Keyword Search in Pure Python*. URL: https://pypi.org/project/ahocorapy/.

[2] Eli Ben-Sasson et al. "Fast multiplication in binary fields on gpus via register cache". In: *Proceedings of the 2016 International Conference on Supercomputing*. 2016, pp. 1–12.

[3] btarunr. *NVIDIA "Pascal" GP100 Silicon Detailed*. https://www.techpowerup.com/221641/nvidia-pascal-gp100-silicon-detailed?cp=2.

[4] NVIDIA Corporation. *CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[5] NVIDIA Corporation. *How to Optimize Data Transfers in CUDA C/C++*. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[6] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[7] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction.

[8] V. Pratt D. Knuth J. Morris. *FAST PATTERN MATCHING IN STRINGS*. http://static.cs.brown.edu/courses/csci1810/resources/ch2_readings/kmp_strings.pdf. 1977.

[9] Mark Harris et al. "Optimizing parallel reduction in CUDA". In: *Nvidia developer technology* 2.4 (2007). https://cuvilib.com/Reduction.pdf.

[10] Stephen Jones. *CUDA Optimization Tips and Tricks*. https://on-demand.gputechconf.com/gtc/2017/presentation/s7122-stephen-jones-cuda-optimization-tips-tricks-and-techniques.pdf.

[11] mohsen kamrani. *Aho-Corasick state transition table*. https://stackoverflow.com/questions/22398190/state-transition-table-for-aho-corasick-algorithm.

[12] Marcin Knap and Pawel Czarnul. "Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs". In: *The Journal of Supercomputing* 75 (Nov. 2019). DOI: 10.1007/s11227-019-02966-8.

[13] Charalampos Kouzinopoulos, Panagiotis Michailidis, and Konstantinos G. Margaritis. "Multiple string matching on a GPU using CUDA". In: *Scalable Computing: Practice and Experience* 16 (June 2015). DOI: 10.12694/scpe.v16i2.1085.

[14] Cheng-Hung Lin et al. "Accelerating String Matching Using Multi-Threaded Algorithm on GPU". In: *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. 2010, pp. 1–5. DOI: 10.1109/GLOCOM.2010.5683320.

[15] Paulius Micikevicius. "Local memory and register spilling". In: *NVIDIA Corporation* (2011). https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.

[16] Nazar Pasternak. *CUDA Memory Optimizations*. https://github.com/heeveG/cuda-memory-optimizations. 2022.

[17] Nazar Pasternak. *PFAC CUDA*. https://github.com/heeveG/PFAC-CUDA. 2022.

[18] Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann. *RegDem: Increasing GPU Performance via Shared Memory Register Spilling*. 2019. arXiv: 1907.02894.

[19] Nikolay Sakharnykh. *Maximizing Unified Memory Performance in CUDA*. https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/.

[20] Jakob Siegel, Juergen Ributzka, and Xiaoming Li. "CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator". In: *2009 International Conference on Parallel Processing Workshops*. 2009, pp. 174–181. DOI: 10.1109/ICPPW.2009.78.

[21] W. Tang. "Graphics Processing Units". In: *The Geographic Information Science & Technology Body of Knowledge (2nd Quarter 2017 Edition), John P. Wilson (ed.)* https://gistbok.ucgis.org/bok-topics/graphics-processing-units-gpus. 2017. DOI: 10.22224/gistbok/2017.2.8.

[22] Wikipedia. *Aho–Corasick algorithm — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm. 2022.

[23] Wikipedia. *General-purpose computing on graphics processing units — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units. 2022.

[24] Wikipedia. *Moore's law — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Moore's_law. 2022.