Research on Modern Computer Architecture Optimization Techniques: Implementation and Measurements for Big Data Processing

by

Yan He
B.Eng., Anhui University, 2016

A Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Research on Modern Computer Architecture Optimization Techniques:
Implementation and Measurements for Big Data Processing

by

Yan He
B.Eng., Anhui University, 2016

Supervisory Committee

Dr. Jens Weber, Supervisor
(Department of Computer Science)

Dr. Sean Chester, Departmental Member
(Department of Computer Science)

ABSTRACT

With the rapid development of big data computing, our programmers need to improve the efficiency of big data processing. In our daily development process, we normally focus on bug-free deployment and often overlook another aspect of software engineering, performance optimization. Actually, a great deal of programming effort is required to achieve good performance. Many techniques are having the potential to significantly improve software performance. To achieve efficiency, I firstly conclude some optimization techniques dealing with memory-bound issues then moving to parallel programming to deal with compute-bound problems. As an example, we apply as many techniques that I mention in the report to a popular algorithm implementation, PageRank based on C++17. By providing constant feedback from performance measurement and profiling, we can see a drastic speed up after implementing all these optimization techniques. I also experiment with a real-world graph dataset provided by IMDb which can rank the top movies, and also evaluate performance before and after optimization. This report hopefully can provide potential future direction towards applying different optimization techniques to various big data processing applications.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to:

Dr. Jens Weber, my supervisor, for supporting and encouraging me in the writing of this project.

Dr. Sean Chester, for leading me into this interesting academic area.

Zhen Maihu, my fellow student, for helping me with some helpful discussions.

DEDICATION

I dedicate this project to my family and my friends, who have been a constant source of support and encouragement during the challenges of graduate school and life.

Chapter 1

Introduction

Performance plays a very important role in software engineering. How to use limited hardware resources to support more computations and tasks is one of the responsibilities of software engineers. Though good algorithm design and good implementations of these algorithms are the most crucial components for a well-performing application, in this report, we mainly focus on the general techniques that can deal with most of the memory-bound and compute-bound in our code.

When we talk about performance improvement, we normally come up with parallelism. Actually, before we get into the multi-threading optimization, normally there is a basis that the hardware resources are not fully utilized. A good approach to decide what kind of optimization techniques to apply is profiling. Profiling our application can give us a direction whether problems are mainly memory-bound or compute-bound. The memory bound refers to the problem that the time to complete a given computational problem is decided primarily by the amount of memory required to hold the working data[1]. That is to say, if we apply multi-threading or GPU computing methods to these memory-bound problems, we will not see an obvious speed up or even make code slower. This is because the time for computing instructions does not compensate for the efforts of reading or writing the memory. Compute bound refers to the problem that the number of instructions is high enough compared to the number of memory reads or writes. To achieve parallel-friendly, we need first to make our application compute-intensive. This report, it is mostly introduced Intel Top-down Microarchitecture Analysis Method (TMAM)[2] for identifying performance bottlenecks. This report will talk about what optimization techniques should be applied in different situations according to the different types of bottlenecks.

After the introduction of some important optimization techniques, the perfor-

mance of the PageRank algorithm was examined on two modern processors. The PageRank algorithm was chosen for the case study because it is widely applied to any sort of graph or network in many domains, such as chemical structure in chemistry, network data in biology, social networks, sports ranking, book ranking in literature and so on[3]. Besides, PageRank measures the importance of each node within the graph, based on the number of incoming relationships and the importance of the corresponding source nodes[4]. The graph being a pointer-based structure is another reason why we choose PageRank which the address of the vertex to be accessed next in the graph is only available through a pointer at the vertex that is currently being accessed. It has a significant portion of irregular memory accesses, which means we usually have to deal with the memory-bound first. Due to the special of this structure, the application implemented based on this structure is often heavily under-utilized by modern hardware systems. In this project, C++ is chosen as the experiment's programming language and I rent two large systems as the experiment device. Some of the techniques that are introduced in the report are applied and a significant performance improvement is gained in the end.

1.1 Motivation

Performance optimization refers to making the system run faster, take less time to complete specific functions without affecting the correctness of system operation. In the past years, we benefited a lot from hardware manufacturers and architectures to achieve performance gains. However, the free lunch is over. The major processor manufacturers and architectures have run out of room with most of their traditional approaches to boosting CPU performance[5]. So that it requires some significant redesign for our programmers to make hardware be fully utilized. Especially nowadays, we deal with more and more large-scale big data processing work, like deep learning and data mining etc. With the increasing amount of data to be stored and processed in the system, some performance problems are becoming increasingly prominent, resulting in the decline of the system processing speed and the failure to return the processing results in time. In severe cases, the system may not even be able to provide services. It is very necessary to optimize big data systems from many aspects according to the architecture level. It is quite complex work. In this report, I give an introduction to some general optimization techniques. PageRank algorithm can be applied across a wide range of domains and it is also be used on big data. So that it

is decided to design efficient and fully distributed algorithms to calculate PageRank.

1.2 Contribution of the Report

The main and new contribution of this report different from the previous course project we did is listed as followings:

- A summary of universal optimization techniques and a discussion of related research of PageRank optimization.
- I conduct extensive experiments with two different hardware environments (one physical server and one cloud server) for performance analyses.
- Redesign benchmarking code, matrix implementation and the test for verifying the correctness of GPU computing.
- Some extensive experiments are conducted on a real-world IMDb dataset to evaluate the performance before and after the optimization.
- Applying data visualization to comparing performance before and after optimization.

1.3 Structure of the Report

The rest of the report focuses on discussing the various techniques to improve the performance and apply them to implement an efficient PageRank algorithm. The following content is divided into four parts as follows:

Chapter 2 provides the background and related work of this project, including literature reviews and introduction to some profiling tools and methods.

Chapter 3 introduces various of optimization techniques.

Chapter 4 presents our experiments, including the details on implementation, optimization and evaluation of PageRank algorithm.

Chapter 5 concludes the project and potential future work.

Chapter 2

Background and Related Work

This chapter discusses the background and related work of optimization on modern computer architectures, research on PageRank optimization also some introduction for performance analysis.

2.1 Background

As we mentioned in Chapter 1, the free lunch is over[5]. We can no longer achieve many performance gains depending on the hardware improvement, like improving clock speed, making execution optimization or adding cache. However, the demand for big data systems is unlikely to abate, and even more so the demand to handle vastly growing quantities of application data is unlikely to stop boosting. Looking past the standard, off-the-shelf techniques and understand the nature of the hardware resources is crucial for us to create a better-performing system[6].

There are some principles that I summarize from some previous research and the experiments I did. The first one is that it's not efficient for us to blindly optimize our system without any profiling. When we suspect that there is a performance problem, it should be analyzed through profiling to find the bottlenecks, rather than relying on feelings and luck. In one program, normally most of the time is spent in a small number of code snippets which is as known as a hotspot. The only reliable way to find these codes is to profile. So that be familiar with these profile tools is the first step in performance optimization. I will talk more about that in Section 2.3. The second principle is that it is usually not worth spending a lot of time micro-optimizing code before it's obvious where the performance bottlenecks are. A well-known software

engineer Donald Knuth has a famous saying "premature optimization is the root of all evil." However, this does not mean that we should not concern about application performance during the early stages of an application's development[7]. That is to say, we should not disturb by some insignificant performance issue at the beginning of the development. A good software developer will automatically feel where performance issues will cause problems and code carefully from the early stage to the end. The last point is that we should not over-optimizing because some optimization techniques have side effects on code readability and maintainability.

Performance optimization is often a trade-off between "time" and "space". Optimization is the art of achieving the best possible result under given hardware resources. The general techniques I discussed later are very helpful if we use them properly and I will show you a whole process of optimizing the implementation of the PageRank algorithm.

2.2 Related Work

Many efforts have been made to improve the efficiency of PageRank calculating. Some methods target to improve part of the algorithm calculating, for example, Linear Extrapolation method[8], Efficient Numerical methods which reduce the PageRank problem to the convex optimization problem[9] or fast random walk-based distributed algorithms[10]. Another aspect of improving the performance of PageRank calculating is to fully use advanced computing hardware like GPU[11], ASCI[12] or FPGA[13]. Besides using these expensive distributed hardware devices, some research work has been done to optimize the usage of CPU memory and caches, for example, Yunming Zhang et al. present a cache optimized in-memory graph framework[14]. Different from existing works, this report combines the different optimization technologies working on CPU and GPU processors, including single-threaded optimization, multi-core optimization and GPU computing optimization.

2.3 Diagnostic Methods

In this section, a brief introduction of some benchmarking concepts, some common profiling tools, like VTune Profiler and CUDA nvprof and Intel Top-down Microarchitecture Analysis Method (TMAM) will be given.

2.3.1 Benchmarking

Sometimes we simply want to get a raw run time which is a general average time consuming by a code snippet. A benchmark nests an idea inside a loop that runs many times (to balance randomness) and uses timing libraries to calculate how long the loop takes. There are two primary things we should pay attention to when we benchmark. The first is that be careful when selecting the test data. The test data should be both representative and feasible, which means the test data can not be too small, too large or non-representative. The second point is that ensuring that the benchmark times what we are trying to time. We need to control the overhead is small enough so that it will not influence the real-time that cost by the target function. The overhead can not be too small either to avoid the compiler optimize away any of the function calls

2.3.2 Intel VTune Profiler

Intel VTune Profiler is the ultimate performance profiler and analyzer for processors. We can use VTune Profiler to analyze our choice of algorithm. Identify potential benefits for our application with available hardware resources[15]. It has key features like Hotspot analysis, Threading analysis, Microarchitecture Exploration and so on. It can also analyze GPU computing and OpenMP applications. Using this together with Intel TMAM can easily locate the bottleneck of the program.

2.3.3 TMAM

Most modern CPUs have a Performance Monitoring Unit (PMU), which is used to count specific hardware events that occur in the system, such as Cache Miss or Branch Misprediction. At the same time, several events can be combined to calculate some advanced metrics, such as the number of cycles per instruction (CPI), cache hit ratio, and so on. A specific microarchitecture can provide hundreds of events through a PMU. It's difficult to pick out them from the hundreds of events that are really useful for solving specific performance problems. Top-down Microarchitecture Analysis Method (TMAM) for identifying performance bottlenecks in out-of-order cores. The general hierarchical framework and the spirit of the hierarchical technique can apply to many out-of-order microarchitectures[2]. Figure 2.1[2] shows the hierarchical approach to classify performance bottlenecks correlating to major functional blocks of

modern microarchitectures.

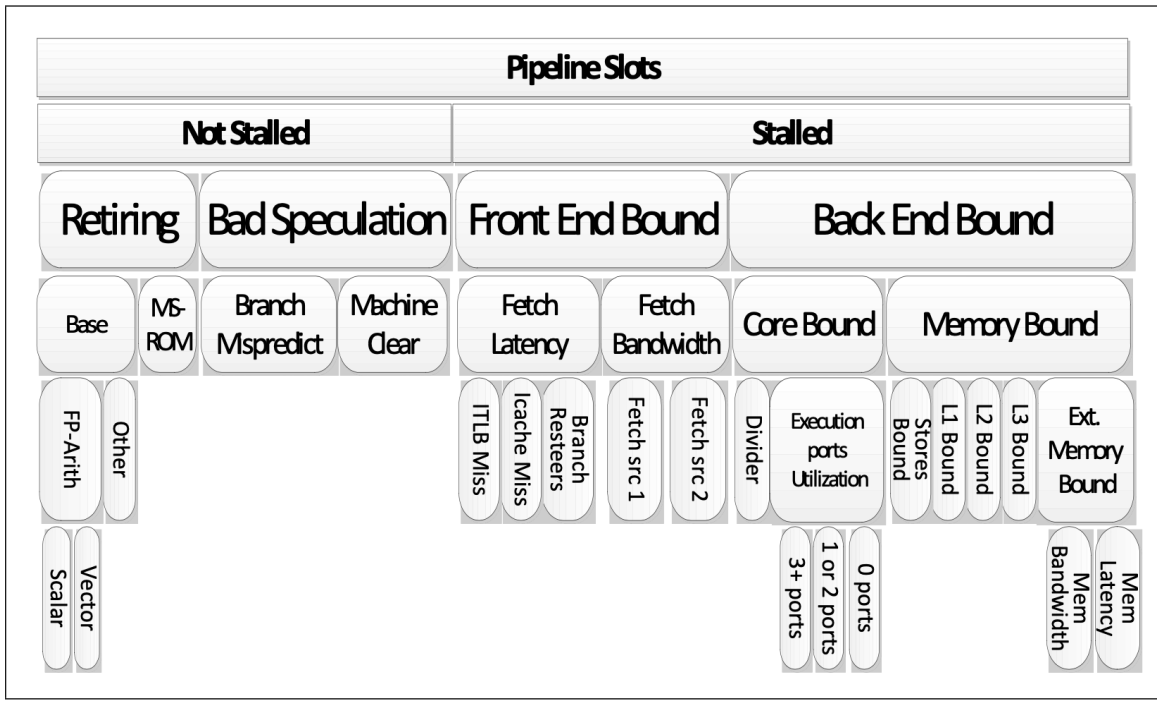

Figure 2.1: General TMAM Hierarchy for Out-of-Order Microarchitectures

The advantage of this top-down analysis framework is that it is a structured approach that selectively explores possible performance bottleneck areas. Hierarchical nodes with weights allow us to focus our analysis on issues that are really important while ignoring those that are not. TMAM will be used a lot in the later experiment.

2.3.4 CUDA nvprof

CUDA nvprof is the lightweight profiler that enables you to collect and view profiling data from the command line[16]. By running the target application with *nvprof ./myApp*, we can quickly see a summary of all the kernels and memory copies that it used. In its default summary mode, nvprof presents an overview of the GPU kernels and memory copies in our application. The summary groups all calls to the same kernel together, presenting the total time and percentage of the total application time for each kernel. It also provides many other modes like GPU-Trace and API-Trace modes, Event/metric Summary Mode and Event/metric Trace Mode.

Chapter 3

Optimization Techniques

Basically, we can classify performance problems as memory-bound or compute-bound. The first one refers to the low RAM bandwidth problems where the number of instructions that are done for each data element is low. The second one refers to the low CPU bandwidth problems where the number of instructions is high compared to the number of memory reads or writes. Normally we will first deal with memory-bound problems and make our program compute-incentive. This chapter discusses the optimization techniques including how to get more out of single-threaded execution and how to exploit parallelism at many different levels.

3.1 Single-threaded Optimization Techniques

Some large-scale applications heavily under-utilize modern hardware systems. In this section, I will introduce you to some single-threaded optimization techniques that can potentially optimize your program. For single-threaded optimization, four aspects are mainly focused which are how to help processors make branch prediction, how to efficiently use caches, implement instruction-level parallelism and multiple-data parallelism.

3.1.1 Branch Prediction

In our code, it is inevitable having branches, i.e., what code should be run depends on a condition. For the processor, if it meets a branch, it can't decide how to continue and what to do. It can only pause the operation and wait for the operation of the previous instruction to end. Then it can continue down the right path. Modern

processors are complicated and have long pipelines which means slow down and warm boot takes a lot of time. So that leads to branch prediction. If we successfully predict the branch, the processor does not need to pause and continues to execute. However, if we guess wrong, the processor will flush out the pipelines, roll back to the previous branch, then restart and select another path, which is called Bad Speculation. Bad Speculation can be tremendously expensive and is one of the top-level concerns in the Intel TMAM. The result of bad speculation could be a loss of spatial and/or temporal locality, which in turn creates cache misses.

What we programmers can do to improve the branch prediction to avoid pipeline stalled is listed below:

- Making full use of compiler static branch prediction and compile at full optimization (-O3) for example.
- If possible, presort the input data.
- Coding branch-free code which means convert the if condition statements to let condition directly multiply the statements inside the if condition.

In general, a branch misprediction ratio over a few percent is quite bad and indicates a probable bottleneck to optimize. Applying the above methods might solve the problem.

3.1.2 Take Full Advantage of Cache

As the saying goes, there are no performance problems that can not be solved by the cache. If so, add another level of cache. However, in reality, the cache resource is always limited. That is why we need some coding strategies to achieve the maximum utilization of caches.

3.1.2.1 Hot/Cold Fields Splitting

This strategy is to split heavily accessed hot portions of data structure from rarely accessed cold portions. In that case, we can divide a structure into several ones where hot fields getting together and some less used fields together into other structures. This may bring some overhead, but if we use it properly we can optimize data layout, improve temporal locality to reduce cache misses.

3.1.2.2 AoS or SoA

AoS refers to Array of Struct and SoA refers to Struct of Array. They all refer to contrasting ways to arrange a sequence of records in memory[17]. AoS stores data as a single structure consisting of multiple arrays and SoA stores data as multiple structures, each of which has a full set of fields, themselves stored in an array of all such structures. AoS is relatively easy to implement in our code and single records are easy to access and work with and it performs well when multiple fields need to be accessed at once. However, when working with just some of the data from a large number of records, the entire set of structures needs to be loaded into memory including the irrelevant data which is not cache-friendly. SoA is suitable if we need to loop over our data looking at only one or two fields from most structs, and only accessing the rest in rare cases and it also benefits a lot from SIMD (Single-instruction-multiple-data).

3.1.2.3 Field-reordering

This strategy also needs to modify data structures to optimize cache utilization. If the structure size is equal to or smaller than the cache line size, we will not consider this method. Only if the structure size is much larger than the cache line size, field reordering may improve cache line utilization[18]. In C and C++, the compiler is not allowed to reorder data members of structs, so if we're not careful with how we order them, we end up wasting valuable space. In java, there is no such issue. The compiler deals with variable declaration, and the programmer has no control over where to allocate them in memory.

3.1.2.4 Loop Tiling/Unrolling

Loop optimization in compilers has two important goals: one is to improve locality and the other is to improve parallelism. Loop tiling is one of the most important optimizations to improve data locality. For loop logic, the reuse distance of a memory element is reduced by disassembling a large loop iteration into several smaller loop iteration lines. In other words, it is ensured that after the memory element is loaded into the cache, it is retained in the cache as much as possible until it is accessed again, to reduce the overhead of expensive memory access. To put it simply, we rearrange the loop so that a cache line can be used again before it is evicted. The essence of loop tiling is to divide the data into tiles when the total amount of data cannot fit in the cache so that the data of each tile will fit in the cache. Therefore, tiling generally

starts from the innermost loop (if we start from the outer loop of a tile, a tile will include the entire inner loop which is too large).

3.1.3 Instruction-Level Parallelism

In 1978, Intel 8086 processors could only execute single instructions at one time. Intel began to use instruction-level parallelism(ILP) in the 486 chip for the first time. The principle is that when there is no correlation between instructions, they can be overlapped and executed in parallel in the pipeline. We can apply some of ILP in our code actually. When our code has instructions that don't depend on each other, for example, don't use the same variables, we can apply ILP to gain some performance.

3.1.4 Multiple-Data Parallelism

The main technology of multiple-data parallelism for single-threaded optimization is SIMD (single instruction, multiple data). SIMD can be part of the hardware design and also be applied to our code to achieve fine-grain parallelism. It is a parallel paradigm in which we have an array of data on which we want to concurrently apply the same instruction. One possible way to use SIMD instructions is using the *intrinsics* provided by the compiler, that is, the compiler implements some built-in functions and types. When using them, the corresponding operations will be translated into SIMD instructions.

We believe there are many other single-threaded optimization techniques that we did not mention above, however, these are some general and efficient methods that we use most frequently.

3.2 Multi-core Optimization Techniques

Multi-core data processing refers to the use of multiple processing cores to cooperatively solve a data processing task. Many techniques can help to achieve multi-core parallel, a popular one which is OpenMP (Open Multi-Processing) will be talked.

OpenMP is mainly used for the parallel computing platform of shared memory and it's relatively easy to program. We only need to add a group of compiler directives, library routines, or environment variables into the source program[19]. The compiler directives in OpenMP consists of a list of directive and clauses. The format is as

follows:
$\sharp$ *pragma omp* $<$ *directive* $>$ [*clause* [*clause*] ...]
OpenMP has a lot of directives, there are some most frequently used shown below:

- **parallel**, used before a code segment, indicates that the code will be executed by multiple threads in parallel.
- **for**, used before a for loop, to allocate loops to multiple threads for parallel execution, it is necessary to ensure that there is no correlation between each loop.
- **parallel for**, the combination of *parallel* and *for* statements which are also used before a for loop to indicate that the code of the for loop will be executed by multiple threads in parallel.
- **barrier**, it is used for thread synchronization of code in the parallel area. When all threads execute to the *barrier*, they will stop until all threads execute to the *barrier*.

Some frequently used library routines are listed below:

- **omp_get_num_procs**, returns the number of processors of the multiprocessor running this thread.
- **omp_get_num_threads**, returns the number of active threads in the current parallel region.
- **omp_get_thread_num**, returns thread number.
- **omp_init_lock**, initializes an OpenMP lock without a hint.
- **omp_set_lock**, locking operation.
- **omp_unset_lock**, unlock operation which should be used with *omp_set_lock* function in pairs.

There are also several often used clauses presented below:

- **private**, specifies that each thread has its own private copy of its variables.
- **reduce**, used to specify that one or more variables are private, and these variables will perform some specified operation after parallel processing.

- **schedule**, specifies how for loop iterations are scheduled.
- **default**, used to specify the data-sharing attributes of all variables in a parallel region. The default is shared.

3.3 GPGPU Computing

GPU is based on SIMD, which is a single instruction multi-data flow model. The structure of GPU and CPU is quite different due to the different design goals of each. The CPU needs to process various data types, and at the same time, it needs to make logical decisions, which will introduce a large number of branch jumps and interrupt processing. GPU is faced with large-scale data with highly unified types and no interdependence, and a pure computing environment that does not need to be interrupted. Figure 3.1[20] below shows an example distribution of chip resources for a CPU versus a GPU. To make full use of the excellent computing power of GPU, introduced GPGPU (General-Purpose Graphics Processing Unit) which is general computing based on GPU. To deal with some obstacles when programming on GPGPU, NVIDIA company released CUDA in 2006. Using this computing platform, GPU can better carry out complex computing and also reduce the difficulty of programmers programming on GPU. Developers can use C language, C + + and FORTRAN to write code for GPU, which improves the programmability of GPU.

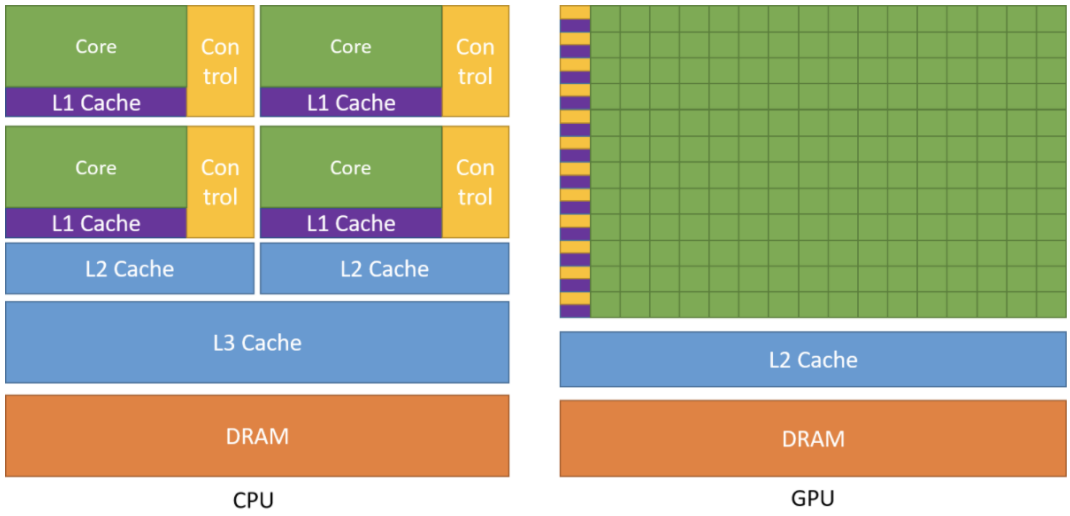

Figure 3.1: Example Distribution of Chip Resources for a CPU versus a GPU

The structure of a CUDA program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer. Each CUDA file consists of both host and device code[21]. The code for the host is just a normal program and the code running on the device(s) is called "kernel". So that is to say any traditional C program is a CUDA program that contains only host code. Once we compile the code, the compiler which can recognize and understand CUDA code will generate code for host and device separately. As for the memory allocation, the host and devices separate (DRAM) memories which means both of them having their own memory. CPU is in charge of the memory allocation and movement of memory between host and device. When GPU starts running, a host code lunches a kernel and this kernel function is to be executed by all threads during a parallel phase. Threads are grouped into thread blocks and blocks grouped into grids. Mapping this runtime environment to hardware:

- **CUDA thread** maps to CUDA core aka. streaming processor (SP)
- **CUDA thread block** maps to CUDA streaming multiprocessor (SM)
- **CUDA thread grid** maps to CUDA-capable GPU

A typical CUDA programming process is shown below:

1. CUDA allocates GPU memory (cudaMalloc).

2. CPU copies data to GPU (cudaMemcpy).
3. CPU launches kernel on GPU; Parallelism expressed here.
4. CPU copies result from GPU (cudaMemcpy).

As we know that the kernel function is executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. The big ideas of writing a kernel are listed below:

- Kernel looks like a serial program, that is to say, we no need to worry about writing any parallelism code.
- Write as if run on **one** thread, which will actually run on many threads.
- Each thread knows its thread index which is accessible within the kernel through built-in variables. So that we can control each thread to do different parts of the computation.

However, we should bear in mind that though CUDA programming is powerful, we still have some computation/communication trade-offs in it.

Chapter 4

Experiments and Evaluation

This chapter discusses how do I apply the above techniques to our PageRank implementation and how the desired results have been achieved. I profile and analyze using Intel oneAPI VTune Profiler before making the optimization. Then compare the optimization results with the original performance.

4.1 PageRank Algorithm Description

PageRank algorithm is initially developed to rank web pages by the Google search engine [22]. In the past decade, PageRank has worked as a very effective measure of a reputation for both web graphs and social networks, ever since it was first demonstrated that it can effectively determine the importance of nodes in large graphs [23]. It is implemented by counting the access number and quality of links to each web page to calculate a rough estimation of how important the website is. The underlying assumption is that the more important websites are, the more likely they are to receive more links from other websites [22]. As we enter the era of big data, the frequency of people using the Internet and the number of social networks are dramatically increasing, with the web growing rapidly as well. Analyzing and calculating through PageRank to find users' preferences and customization becomes more and more important. It is worthwhile for us to pay more attention to designing efficient PageRank implementation.

The initial Pagerank implementation is based on the original algorithm, which is also the best-known PageRank algorithm [22]. The algorithm can be split up into three steps: initialization, calculation, and finalization. The initialization of the ranks

is done by giving the same PageRank value for all pages. These initialization values do not play a decisive role since the algorithm converges after enough iterations. The next step is the calculation which iterates over all nodes in the network and calculates its rank value divided by the number of its neighbours, then the node communicates this rank to its neighbours. In this process, after every iteration, it is required to calculate the difference between the current PageRank array and the previous array to determine if the array has converged. Then coming to the last step, finalization, finalize when each node's PageRank from the previous iteration differs from the current iteration by less than or equal to the margin of error or once we reach the maximum iteration times.

The formula of PageRank that implemented in our experiment is shown below

$$PR\left(p_i\right) = \frac{1-d}{N} + d\left(\sum_{p_j \epsilon M(p_i)} \frac{PR\left(p_j\right)}{L\left(p_j\right)} + \sum_{p_j \epsilon dangling} \frac{PR\left(p_j\right)}{N}\right) \tag{4.1}$$

$PR\left(p_i\right)$ is the PageRank value of node p_i. N is the total number of nodes. $M\left(p_i\right)$ is the set of nodes that link to node p_i. d is the damping factor. PageRank theory assumes that a random click on a link by an imaginary viewer will eventually stop clicking. The probability that the person will continue at any step is the damping factor d, which is generally assumed to be set around 0.85. *dangling* is the nodes that do not point to any nodes or pages. And once we get to that node, the PageRank will sink, which has no contribution to the result when following the web graph in the next iteration, because it can not go anywhere else.

This origin algorithm is not hard to understand that takes the directed graph as input which consists of the number of pages and their outbound relationships and takes PageRank values and the iteration times as output. The pseudocode of the algorithm is shown below.

Algorithm 1 PageRank

```
1: procedure PageRank(graph, iteration)
2:     for all p in the graph do
3:         pre_pr [p] ← 1/N
4:     end for
5:     while iteration < maximum_iter_times do
6:         for all p in the graph do
7:             pre_pr [p] ← pr [p] ; pr [p] ← 0.0;
8:         end for
9:         for all p that has no out-links do
10:            pr_dangling ← pr_dangling + d ∗ pr [p]
11:        end for
12:        for all p in the graph do
13:            for all inward_edges of p do
14:                pr [p] ← pr [p] + d∗pre_pr[inward_edge] / total_links[inward_edge]
15:            end for
16:            pr [p] ← pr [p] + 1 − d/N + pr_dangling/N
17:        end for
18:        iteration ← iteration + 1
19:        if difference (pr, pr_pre) < tolerance then
20:            break
21:        end if
22:    end while
23: end procedure
```

4.1.1 Detailed Implementation

First construct a class Vertex with a list of nodes, each node containing the PageRank value, previous PageRank values, number of inward edges and number of outward edges and a list of its neighbours. Then a vector is used to store those vertices in the graph, which is the Array of Structure (AoS).

As for the input data set, I provide two types of data for our experiment. One is the customized data set that can be easily changed the size and the other one is from the real world. To obtain a sufficient data set, a mechanism is created to generate nodes and links randomly. By taking how people access the Web Pages in

reality into consideration, first, eliminate the node with links that point to itself and the repeated node pairs with the same source and destination. In addition to that, to avoid our generated links being too discrete which is inconsistent with reality, some links are generated with normal distribution and other links conform to the uniform distribution. The data generation is done in a separate application to profile the experiments as a stand-alone application, without measuring the overhead of the experiment set up. Another type of data set that I run the PageRank algorithm on is a data set of movie references from IMDb [24]. The data is preprocessed into the form that can be fed into our graph generator. The reason that I introduce a data set from the real world is that I want to prove that our optimization can deal with the real issues and also give us a good opportunity to verify the different performance between applying on our generated graph and the graph in reality.

By reading the input data set, the graph consisting of a bunch of vertices and their relationships can be constructed. After constructing the graph, I implement the main PageRank algorithm according to the steps mentioned above.

4.2 IMDb Dataset Preprocessing

This section discusses the details about the IMDb movie reference dataset, how is this dataset preprocessed and what distribution does it follow.

4.2.1 Introduction to IMDb Movie References Dataset

References of movies could be music, a film poster visible in a scene, or one of the characters uttering a well-known line from another film[25]. According to IMDb's official website, "references" belongs to the "simple" connection which means it always occurs in pairs and is limited to the title pair only[26]. So that, if we look at every movie as vertex and the references between them as edges, we can apply the PageRank algorithm to obtain the importance of each movie.

The original dataset is from the old version of the IMDb dataset(2016)[27]. The sampler of the dataset is as follows in Figure 4.1.

```
CRC: 0x9F107E8C  File: movie-links.list  Date: Fri Apr  8 00:00:00 2016

Copyright 1991-2016 The Internet Movie Database Ltd. All rights reserved.

http://www.imdb.com

movie-links.list

2016-04-08

------------------------------------------------------------------------------

MOVIE LINKS LIST
================

"#1 Single" (2006)
  (referenced in "Howard Stern on Demand" (2005) {Lisa Loeb & Sister})

"#LawstinWoods" (2013) {The Arrival (#1.1)}
  (references "Lost" (2004))
  (references Kenny Rogers and Dolly Parton: Together (1985) (TV))
  (references The Grudge (2004))
  (references The Ring (2002))

"#MonologueWars" (2014)
  (references Trainspotting (1996))

"#PréTVB" (2014)
  (spin off from "The Voice Brasil" (2012))

"#lovemilla" (2013)
  (followed by Lovemilla (2015))

"$#*! My Dad Says" (2010)
  (featured in "The Wright Stuff" (2000) {(#15.40)})
  (referenced in "Community" (2009) {Anthropology 101 (#2.1)})
  (referenced in "Conan" (2010) {Love Gets Liposuctioned (#1.49)})
  (referenced in "Geeks Who Drink" (2015) {Jonathan Sadowski vs. Lenny Jacobson (#1.7)})
```

Figure 4.1: The Sampler of Original IMDb Movie References Dataset

4.3 Data Preprocessing

Since the implementation of PageRank supports a specific format of data which is a list of pairs of numbers, it is required some preprocessing on the original dataset. The dataset is processed using Python3.9 in the virtual environment.

First, the dataset is organized in paragraphs and each one is a movie with its references information. I split the paragraphs and extract each film's name with its references into a new file. The sampler shows in Figure 4.2. The movie on the right side of " " is the movie that is referenced in the movie on the left side.

```
1   Howard Stern on Demand ~ #1 Single
2   #LawstinWoods ~ Lost
3   #LawstinWoods ~ Kenny Rogers and Dolly Parton: Together
4   #LawstinWoods ~ The Grudge
5   #LawstinWoods ~ The Ring
6   #MonologueWars ~ Trainspotting
7   Community ~ $#*! My Dad Says
8   Conan ~ $#*! My Dad Says
9   Geeks Who Drink ~ $#*! My Dad Says
10  Late Show with David Letterman ~ $#*! My Dad Says
11  Robot Chicken ~ $#*! My Dad Says
12  RuPaul's Drag Race ~ $#*! My Dad Says
```

Figure 4.2: The Sampler of First Version of Processed IMDb Movie References Dataset

Then by using pandas.factorize API, convert the processed dataset to the final version of the dataset which is shown below in Figure 4.3.

```
1   0 1
2   2 3
3   2 4
4   2 5
5   2 6
6   7 8
7   9 10
8   11 10
9   12 10
10  13 10
11  14 10
12  15 10
```

Figure 4.3: The Sampler of Final Version of Processed IMDb Movie References Dataset

4.4 Distribution of the Dataset

As I mentioned before, the customized dataset has the randomly generated sources nodes that follow the uniform distribution and 80% of the destination nodes follow the uniform distribution with 20% of them follow the normal distribution.

To figure out what distributions do the movies and their references follow, I applied scipy.stats._continuous_distns module[28] which contains a total of 104 types of probability distribution to the movies(sources) and references(destinations) separately. The following Figure 4.4 shows the best fit distribution of source nodes fitting with Sum of Square Error.

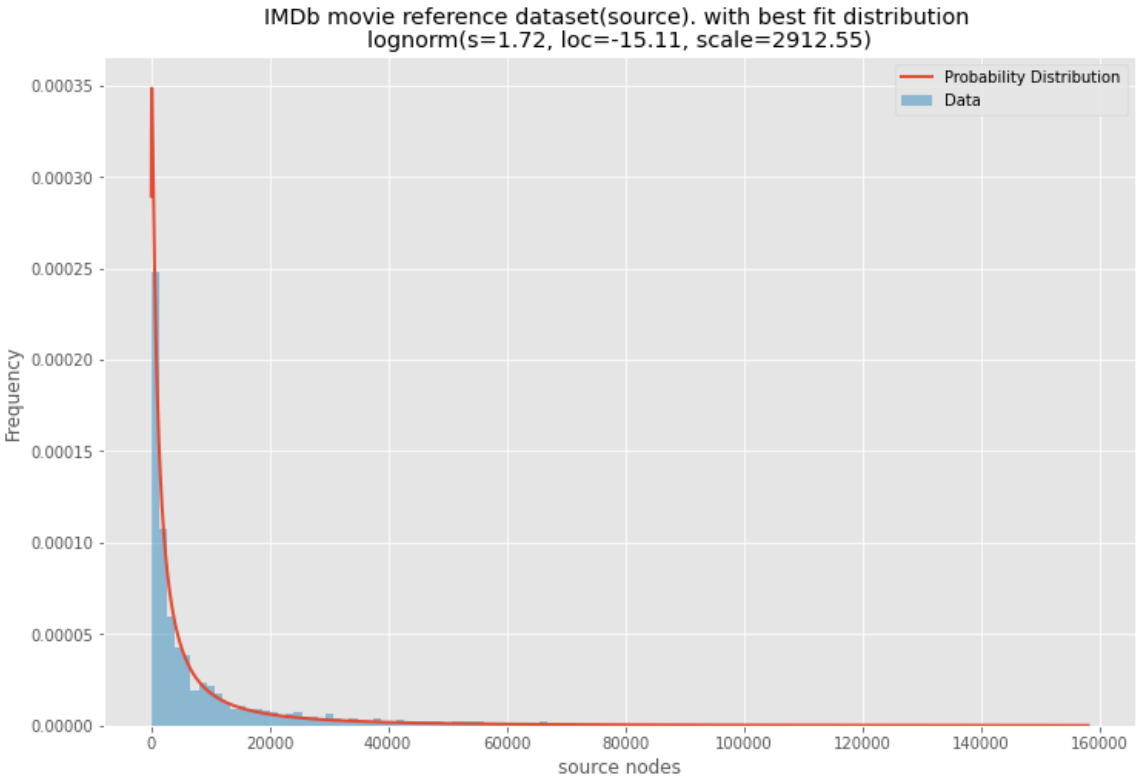

Figure 4.4: IMDb movie reference dataset(source). with best fit distribution

The following Figure 4.5 shows the best fit distribution of destination nodes fitting with Sum of Square Error.

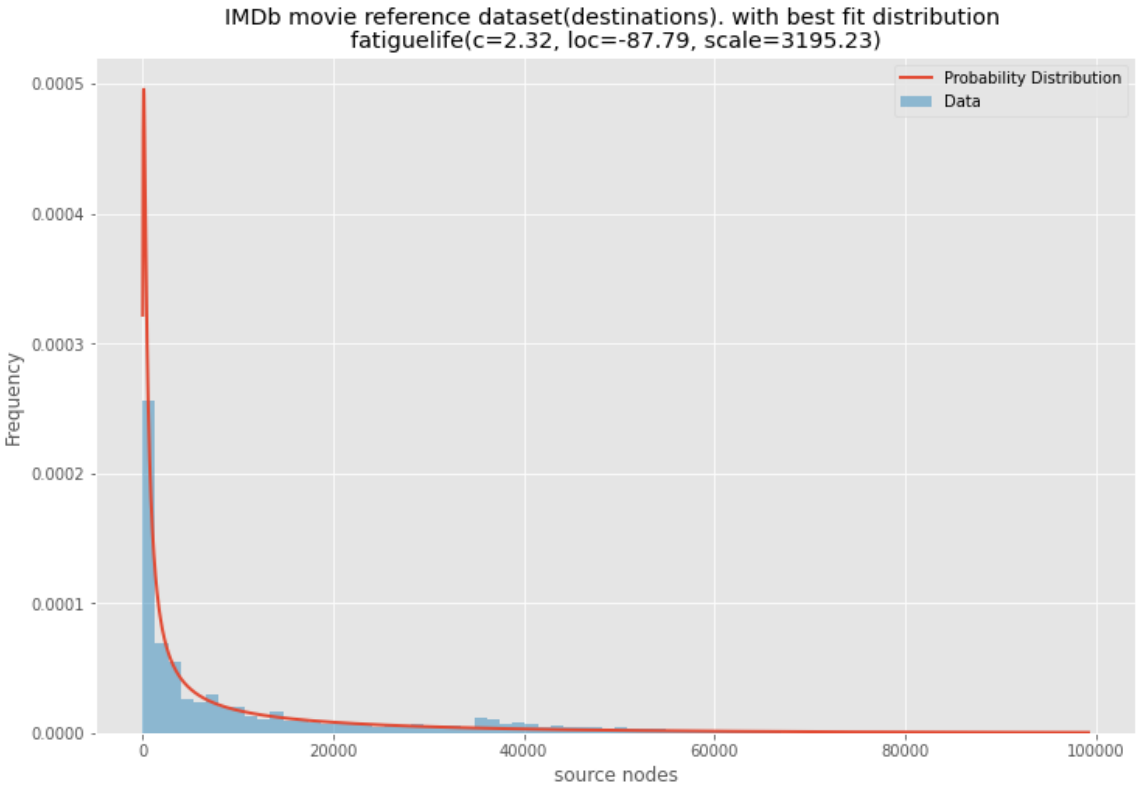

Figure 4.5: IMDb movie reference dataset(destination). with best fit distribution

As we can see in the above figures, movies as source nodes follow the Log-normal distribution and the corresponding reference nodes follow the Birnbaum–Saunders(aka. fatigue life) distribution.

4.5 Single-Threaded Baseline Optimization

4.5.1 Experiment Setup

For this single-threaded optimization, a system with an Intel Xeon processor and 65GB of physical memory (Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz) is used. The processor runs at 2.1GHz and has an 8-way associative 32KB L1d cache, an 8-way associative 32KB L1i cache, an 8-way associative 256KB L2 cache and a 20-way associative 15360KB L3 cache. Setup and configure the Intel VTune environment both on the macOS host and the Linux(*CentOS*) target server. Set CPU sampling interval to 1ms for Microarchitecture Exploration analysis.

4.5.2 Profile and Determine Limiter

First, test with 50,000 vertices and 500,000 links generated by the custom-made mechanism to find the bottleneck of the implementation by using the Intel VTune application. Since the total running time includes the graph constructing phase and the PageRank calculating phase, the customized benchmark method is written which can benchmark each part separately. The test is looped 10 times and calculates the average baseline execution time which is 52947 us for graph generating, 597168 us for PageRank calculating and 785183 us for the total process. Then the Intel VTune Hotspots analysis is run on the whole implementation, I find that microarchitecture usage inefficiency was highlighted as the top issue and the PageRank calculation block takes the most running time. So I perform Microarchitecture Exploration analysis to understand how efficiently your code is passing through the core pipeline [29]. The overview result is shown below in Figure 4.6.

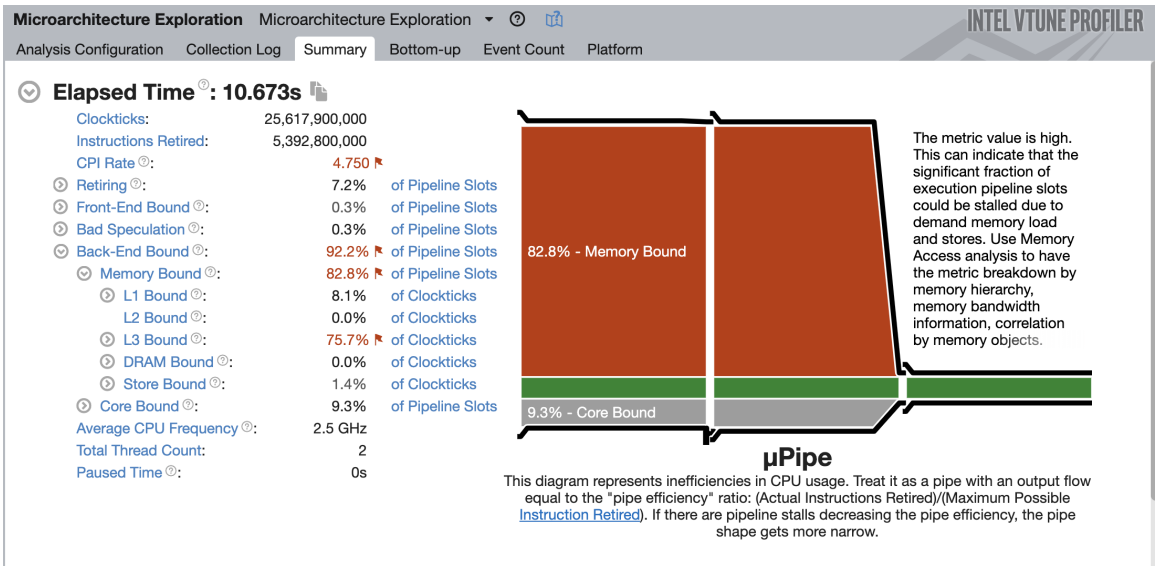

Figure 4.6: Initial Implement Profiling

As we can see, the µPipe diagram shows inefficiencies in hardware usage. Especially, the Memory Bound metric is extremely high, only a small fraction(7.15% of pipeline slots) of pipeline slots are being retired. In the hierarchy of event-based metrics, two potential bottlenecks are mainly noticed which are the high CPI(Clockticks per Instructions Retired) rate(4.750 of pipeline slots) and the high memory bound(82.8% of pipeline slots) under high back-end bound(92.2% of pipeline

slots). The processor I used is capable of executing 4 instructions per cycle, with an optimum CPI (clock cycles per instruction) = 0.25. Here, we see a CPI value that is greater than 4, a near 16-fold slowdown from the optimum, due to poor cache utilization. Furthermore, we can see that L3 latency is the main factor that causes the back-end bound. According to the Intel Top-Down method, this is mainly related to the data access patterns and whether the cache line is appropriately allocated when accessing. Thus, the data structure and how we access data to obtain a more appropriate data access pattern should be improved when calculating the PageRank so that the cache misses can be reduced and the access speed can be improved to gain more instructions per cycle. As for the core-bound(9.3% of pipeline slots) which only takes a little responsibility for the bad performance for now. I plan to focus on dealing with memory bound to make compute-bound before going to multi-core optimization.

4.5.3 Single-threaded Optimisation

Several single-threaded optimization techniques are applied to the PageRank implementation to reduce the high cost of memory access.

4.5.3.1 Hot/Cold Data Splitting

By splitting hot and cold data, the working set size can be compressed. A "Cold-Edge" structure is generated and is used to store source and destination nodes, a "HotData" structure to store the PageRank values and the number of outwards links that are frequently used. Then the adjacency list is applied to construct the whole graph rather than a list of "Vertices". The most frequently used data are sorted in the "HotData" structure. This significantly degrades cache line utilization. By using the same input data as I used for initial implement profiling, I re-run the Microarchitecture Exploration analysis on the optimized code, as shown below in Figure 4.7.

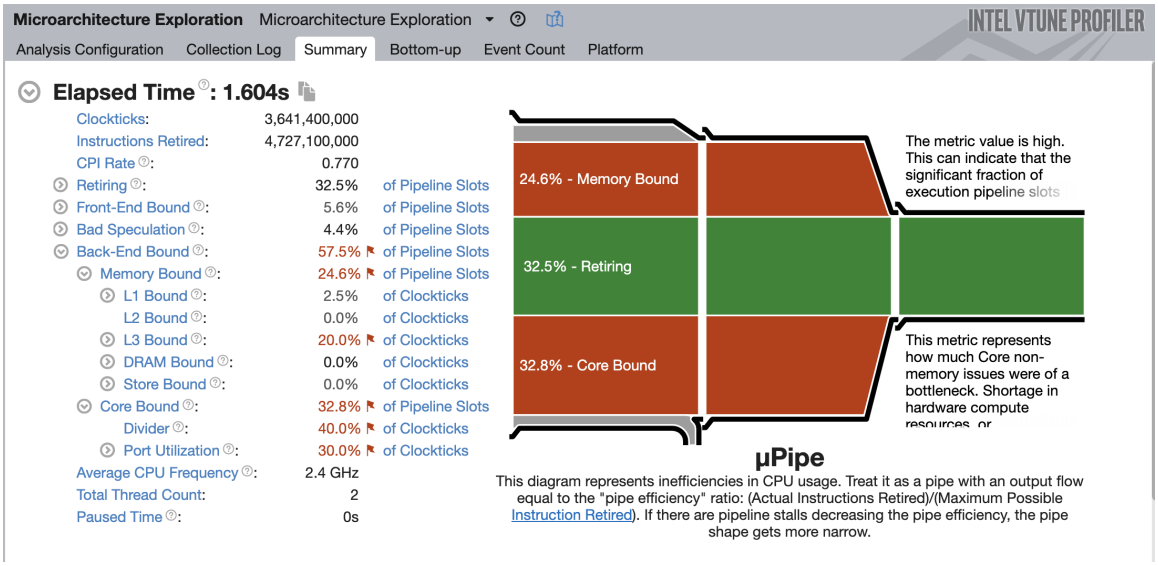

Figure 4.7: Hot/Cold Data Splitting Optimization Profiling

As it shows, the μPipe diagram shows a significant increase of the Retiring metric value, from 7.15% to 32.5%. The total elapsed time decrease obviously from 10.673s to 1.604s. CPI rate decreases from 4.750 to 0.770 which is not a bottleneck anymore. Back-end bound also shows a substantial decrease from 92.2% to 57.5%, though it is still one of the performance bottlenecks. In addition, we can observe that core bound comes into play which accounts for 32.8% of pipeline slots.

4.5.3.2 Structure of Arrays(SoA) Optimization

Based on the hot/cold data splitting optimization, the graph data structure is changed from an array of structure(AoS) to a structure of array(SoA). It makes the data layout consecutive when accessing to improve the data reading and accessing speed. By using the same input data as used for initial implement profiling, I re-run the Microarchitecture Exploration analysis on the optimized code. However, it shows almost nothing improvement. The main reason for that is that we have already allocated sequentially access patterns for arrays of “ColdEdge” and “HotData” in the previous graph implementation.

4.5.3.3 Branch Removing

Based on the previous optimization, a branch is removed in the for loop which is in the hot spot - PageRank calculation part. Since this condition depends on the number of outgoing edges of each node, it is hard to predict each node status of the randomly generated graph. To make this segment of code branch-free, I multiply the condition to the inner statement instead of the if condition. The profiling result shows that it only improves a little of the Retiring metric value.

4.5.3.4 Matrix

Instead of an adjacency list, the whole data structure is implemented into a 0-1 adjacency matrix where 0 represents no edges between the two vertices and 1 represents there is an edge between two nodes. So that a fixed size inner loop can be achieved in the PageRank algorithm. In that case, we can apply the tilling method to two for loops of the PageRank method. However, it is found that the overhead of constructing an $O(n^2)$ adjacency matrix-based algorithm is much higher than the gain using tilling. Running the matrix-based algorithm with tilling even decreases the speed. In addition, the matrix constructs excessive memory consumption, it is not able to run on massive data sets. The data set I used which is 50,000 vertices and 500,000 links is already too large for it, which takes over 5 minutes without completing.

4.5.3.5 Overall Comparison

All in all, I compared the final single-threaded optimization with the initial implementation. Figure 4.8 below shows the performance improvement by using VTune.

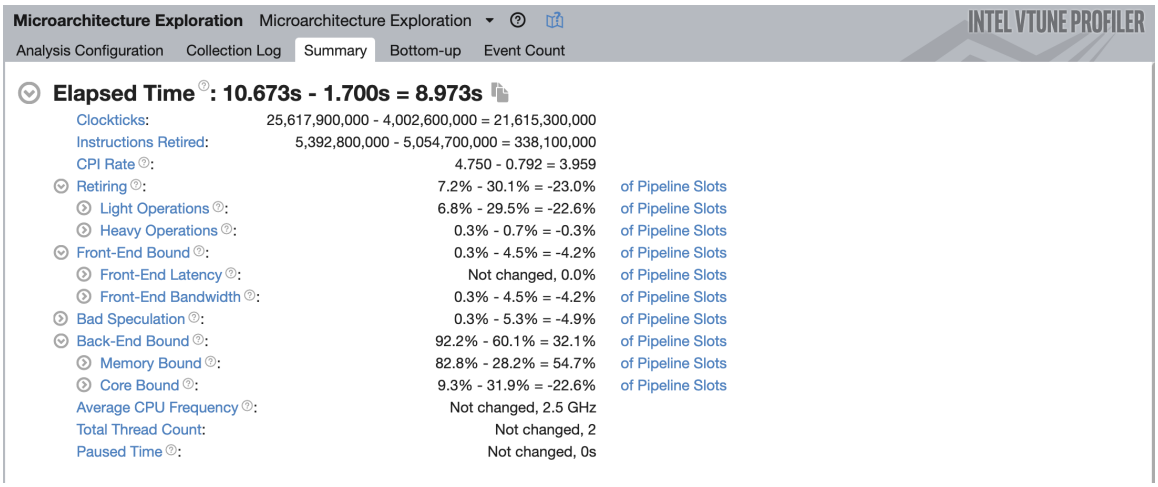

Figure 4.8: Performance Gain by Applying Single-threaded Optimization Techniques

In the figure, we can see that the Elapsed Time shows 8.973 seconds of optimization for the whole execution and an improvement from 7.2% of instructions retired to 30.1% of instructions retired. The Back-End Bound metric improved by 32.1%.

I perform the profiling with 3 sets of data. Two of them are generated ourselves, which consist of $66,892$ vertices with $686,829$ links(as same size as IMDb data set) and $100,000$ vertices with $1000,000$ links. The other data set is from IMDb which contains $66,892$ vertices with $686,829$ links. I loop each test 10 times and get the average execution time of the graph generating part, PageRank calculation part and total execution separately as mentioned above. I calculate the speed-up times by dividing initial implementation time by optimized time. The speed-up result is shown in the following chat in Figure 4.9.

From this figure, we can see that a totally of 6.25x speed up is achieved by experiencing on 67k(66,892) vertex with 687k(686,829) links, 7.59x speed up by using the same size of data set provided by IMDb and 11.33x speed up by using data set of 100k vertex with 1000k links. With the increase in the size of the data set, we can see a more significant performance improvement. Moreover, an interesting fact I observed is that it seems like the dataset from IMDb gets better performance than that work on the generated data set. I assume that this is because the data set from the real world provides a better cache spatial locality and not completely random, which in turn lowers the cache misses.

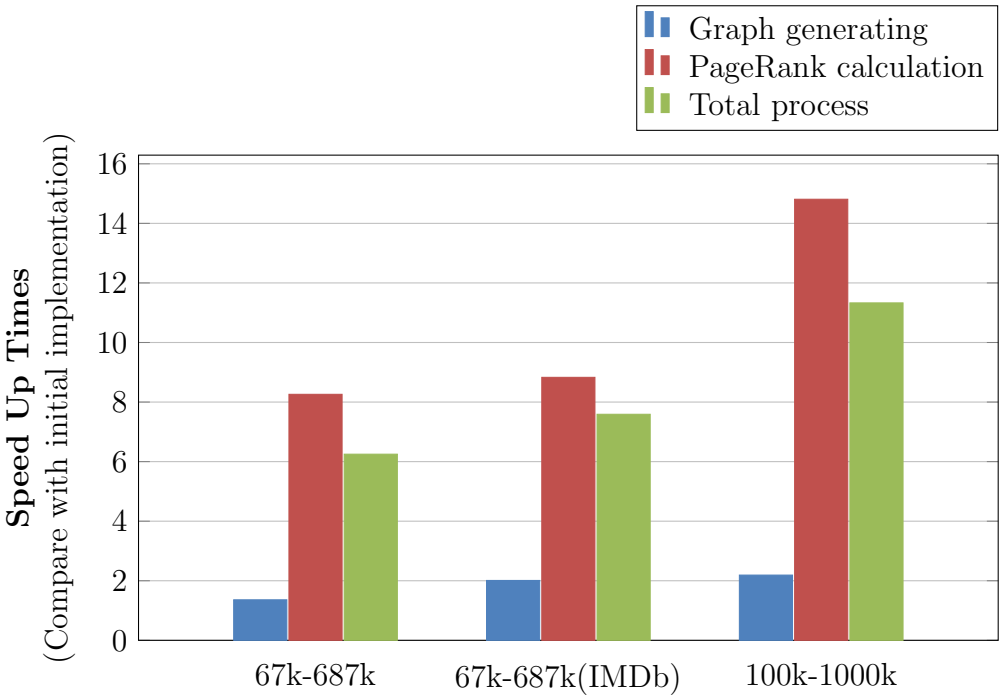

Figure 4.9: Comparison Single-threaded Opt. Performance on 66,892 Nodes with 686,829 Links, 66,892 Nodes with 686,829 Links(from IMDb) and 100,000 Nodes with 1000,000 Links

4.6 Parallelism Optimization

There is a rich source of parallelism available in the server and we can make good use of them to improve the performance to the next level.

4.6.1 Experiment Setup

For this multi-threaded optimization, I still use the same system as before with the VTune profiling tool. The "hardware_concurrency()" in the thread[30] library is used to return the number of concurrent threads supported by the implementation which is 24. One more *argv* is added in the *main*() method which is responsible for setting the active core numbers. So that it is easy to control the number of threads condition each experiment to make a good comparison of the performance.

4.6.2 Parallel Algorithm Description

Basically, three parallel methods are applied to the program:

1. Change the algorithm implementation that parallelly computes multiple nodes' PageRank at the same time.

2. Apply 0-1 adjacency matrix-based version to do parallel computing.

3. Apply the openMP.

The first method can help us to reduce a vector to store outgoing edges for each source index when computing. However, it is found that this method is not suitable for parallel execution using OpenMP since we have to add '#pragma omp atomic' when writing data to random locations in PageRank values calculation, which really hurts parallel scalability and reduces the speed. The second parallel algorithm is to apply OpenMP in the matrix as we mentioned before. Though it can achieve better parallel outcomes than the parallel implementation of the optimized single-threaded code, the overhead brought from the total time used is much higher than the time saved from parallel. Finally, openMP is simply applied to the optimized single-threaded version and it achieves a good performance improvement. First, the '$\#pragma\ omp\ parallel\ for$' is applied to update the pre_pagerank and PageRank values parallelly. The loop for calculating PageRank values and dangling node values are combined into one loop to reduce the number of instructions. As we know that the reduction clause in OpenMP allows us to accumulate the shared variable without the atomic clause that is less efficient, reduction clause '$\#pragma\ omp\ parallel\ for\ reduction(+ : dangling\_pr\_sum)$' is added when calculating the sum of dangling node values. The last omp code segment is applied to the nested for loops which consume most of the calculation time. The '$\#pragma\ omp\ parallel\ for$' is used to achieve parallel computing for the outer loop.

4.6.3 Parallel Scalability

The number of cores is taken as a parameter to test the parallel performance from 1 core to 24 cores on test data of 66,892 nodes with 686,829 links(generated) and 100,000 nodes with 1000,000 links. I compare the experimental results as follows in Table 4.1. To record the detailed scalability clearly with the vary of the number of cores used, the number of cores is taken as a parameter to test the parallel performance from 1 core to 24 cores on test data of 66,892 nodes with 686,829 links and 100,000 nodes with 1000,000 links.

Version	#Cores	Total Time /us (67k-687k)	Speedup Times (67k-687k)	Total Time /us (100k-1000k)	Speedup Times (100k-1000k)
Single threaded Optim. (baseline)	1	760217	1	1194700	1
	1	487440	1.56	804644	1.48
	2	325178	2.34	510195	2.34
	3	271895	2.80	426720	2.80
	4	242584	3.13	328799	3.63
	5	206994	3.67	321074	3.72
	6	199435	3.81	282216	4.23
	7	181788	4.18	279962	4.27
	8	173526	4.38	260619	4.58
	9	164598	4.62	250393	4.77
	10	150424	5.05	228947	5.22
	11	150009	5.07	215972	5.53
Parallel Optim.	12	139338	5.46	211027	5.66
	13	140019	5.43	225887	5.29
	14	148924	5.10	228826	5.22
	15	145548	5.22	231060	5.17
	16	141917	5.36	221065	5.40
	17	140567	5.41	221507	5.39
	18	137092	5.55	210542	5.67
	19	134691	5.64	209677	5.70
	20	134171	5.67	207691	5.75
	21	130200	5.84	215084	5.55
	22	121149	6.28	212043	5.63
	23	123680	6.15	199353	5.99
	24	138693	5.48	210979	5.66

Table 4.1: Parallel performance on two different data sets

To clearly see the trend of the speedup by increasing the threads, we create a line chart as follows in Figure4.10.

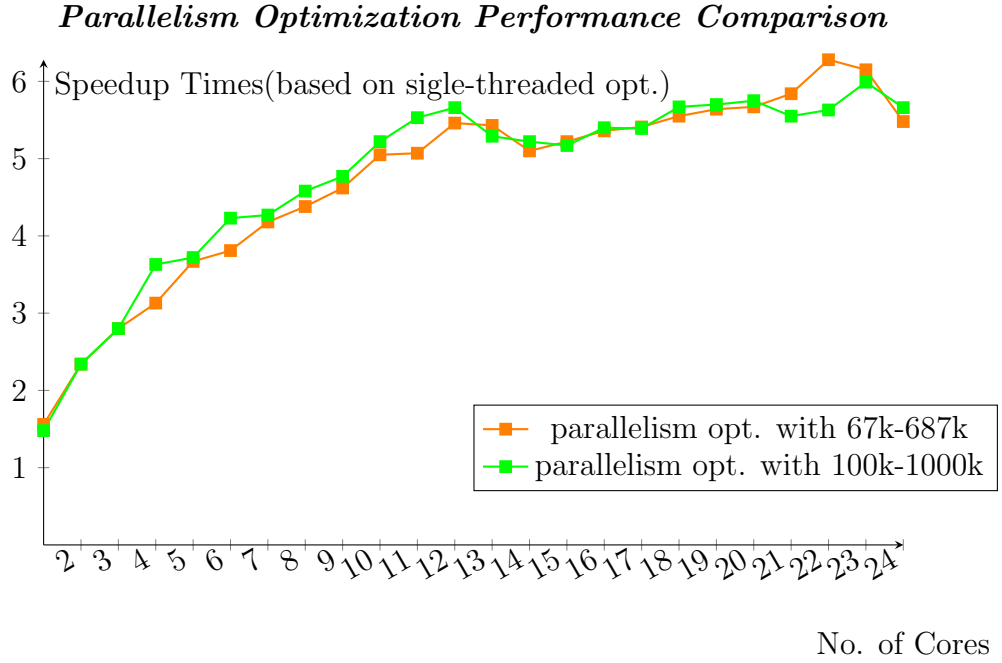

Figure 4.10: Comparison Between Single-threaded Baseline and Parallel Version Performance on 66,892 Nodes with 686,829 Links and 100,000 Nodes with 1000,000 Links

As shown in the chart, the speedup is relatively obvious at the beginning, while the growth rate decreased with the number of cores increased. The scalability and the performance tendency are as expected. Because of the physical limitations to clock speed such as the increasing communication times between threads, not all the cores are fully utilized and the switching delay between cores. In the experiment, a maximum of 6x speed-up can be achieved.

4.7 GPGPU Computation Optimization

This section describes our CUDA implementation of the PageRank algorithm to achieve a significant performance improvement.

4.7.1 Experiment Setup

For this GPU computing implementation, I use a system with an Intel Xeon processor and 132GB of physical memory (Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz). The

GPU on this system is NVIDIA TITAN X and the CUDA version is 11.2. Setup and configure NVIDIA Nsight Compute environment both on the macOS host and the Linux($Ubuntu18.04$) target server.

4.7.2 Modified PageRank Implementation

To make data parallelism to be able to utilize a massive number of arithmetic units on GPU, We need to modify a bit of PageRank implementation to feed the device kernel. As introduced in section 4.2, we already eliminated the branch prediction in the algorithm for reducing bad speculations and since all the PageRank values are written to the same address in the for loops, there are no data race conditions and the threads have no need to wait for others, which means we do not have branch divergence that will affect GPU computing. The previous data structure of the graph is using an adjacency list $std :: vector < std :: vector < int >> adjEdges$ to store all the nodes and their relationships. However, the inner vector size depends on the number of inward edges of each node which is not a fixed value. In that case, it is not suitable for GPU memory allocation and data transformation. We use an approach to transfer the 2D unfixed size array into some 1D fixed-size arrays group: store the number of inward edges of each node in a vector, store every node's begin index in a vector. After modifying the graph data structure, we also need to change part of the PageRank calculation division and the pseudo-code is shown below.

Algorithm 2 Kernel

```
1: procedure PageRank(gpu_graph)
2:     for all node i do
3:         for num_nward_edges j of i do
4:             in_e_index = adjE[begin_index + j]
5:             pr[p] ← pr[p] + d*pre_pr[in_e_index] / total_links[in_e_index]
6:         end for
7:         pr[p] ← pr[p] + 1 − d/N + pr_dangling/N
8:     end for
9: end procedure
```

We write all the contents in the while loop with 100 iterations into _global_ function, so that the data only be transferred once at the beginning and once at the end of the while loop after GPU computation. As the PageRank values of dangling

nodes and previous PageRank values should be updated at every iteration, we create and use 'thrust::reduce'[31] to implement the reduction of the PageRank values of dangling nodes at every iteration on the host. In addition, we use 'cudaMemcpyDeviceToDevice' to update the previous PageRank values with new PageRank values at every iteration.

4.7.3 Evidence of Correctness

Before we step into measuring the performance of the GPU implementation, it is very important to verify the correctness by comparing the final PageRank value of each node both running on single-threaded optimization(baseline) version and GPU computing version. I test with 500 nodes with 5,000 links and the comparing result is shown below in Figure 4.11.

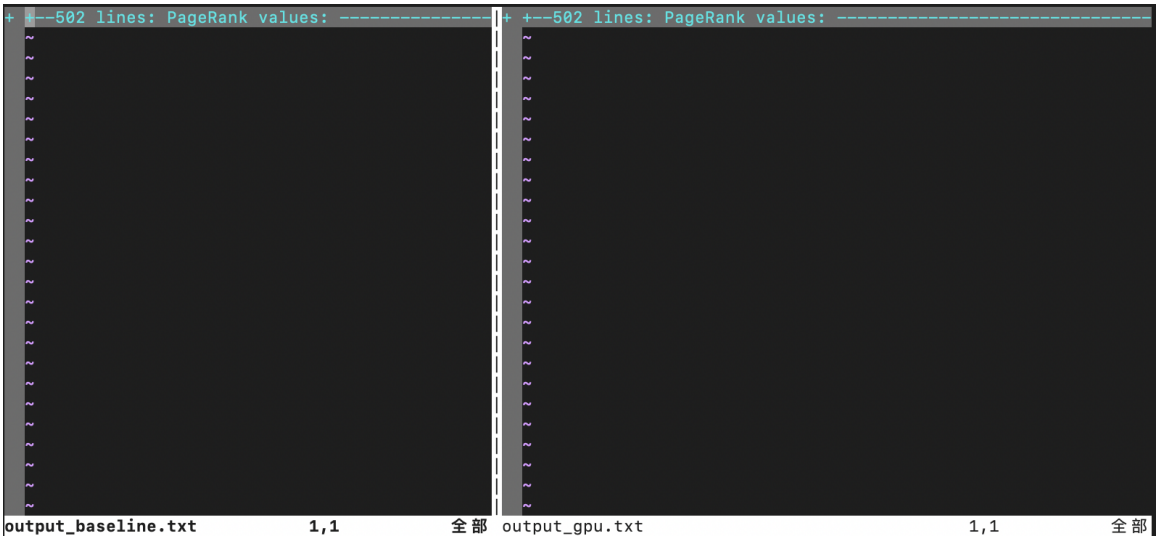

Figure 4.11: Vim difference comparison for baseline and GPU version on 500 nodes.

As we can see that there is no difference between the two results, the correctness is verified.

4.7.4 Raw Performance Evaluation

I still work on the same 3 data sets as before. Two of them are generated ourselves, which consist of 66,892 vertices with 686,829 links(as the same size as the IMDb

data set)and 100,000 vertices with 1000,000 links. The other data set is from IMDb which contains 66,892 vertices with 686,829 links. I run the program with these data sets on three versions, which are the single-threaded optimization version, multi-core optimization version and GPU computing version separately. The result is shown below in Table 4.2.

Data set (#vertex, #links)	Version	Total Time/us	Speedup Times
66892, 686829 (generated)	Single-threaded Optim.	720194	1
	Muti-core(4 cores) Optim.	215896	3.34
	GPU Computing	70956.9	10.15
100000, 1000000	Single-threaded Optim.	1087240	1
	Muti-core(4 cores) Optim.	319716	3.40
	GPU Computing	42545.1	25.55
66892, 686829 (from IMDb)	Single-threaded Optim	571071	1
	Multi-core(2 cores) Optim.	340946	1.67
	GPU Computing	454682	1.26

Table 4.2: Comparison of the performance of three versions on different data sets

According to the table, we can achieve a maximum of nearly 26x times speedup for the data set of 100,000 vertexes with 1000,000 links by using GPU computing. The experiments on our generated data set all have significant performance improvements comparing with single-threaded optimization and multi-core optimization. However, when I change the data set to IMDb movie reference data set, the GPU computing version performs even not as well as the multi-core optimization version. The above results are all from the bench-marking code that was generated by ourselves by using the $std :: chrono$[32] library. Next, I use a profiling tool to obtain more accurate and detailed results of performance.

4.7.5 Profile and Analysis

I use 'nvprof'[33] to obtain the Nvidia profile that gives a summary of all the kernels and memory copies that it used. The following figures are the screenshots of the experiments on three data sets as shown separately.

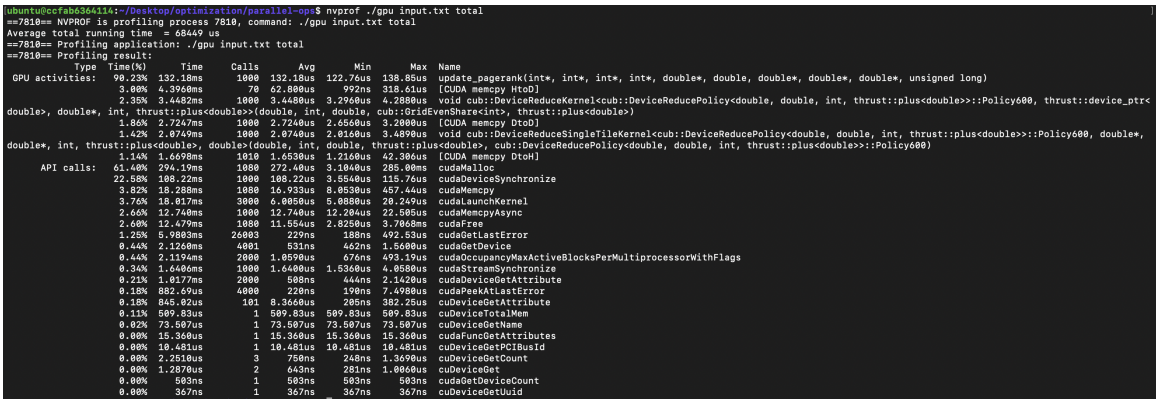

Figure 4.12: Nvidia profiler(summary mode) result of 66,892 nodes with 686,829 links(generated)

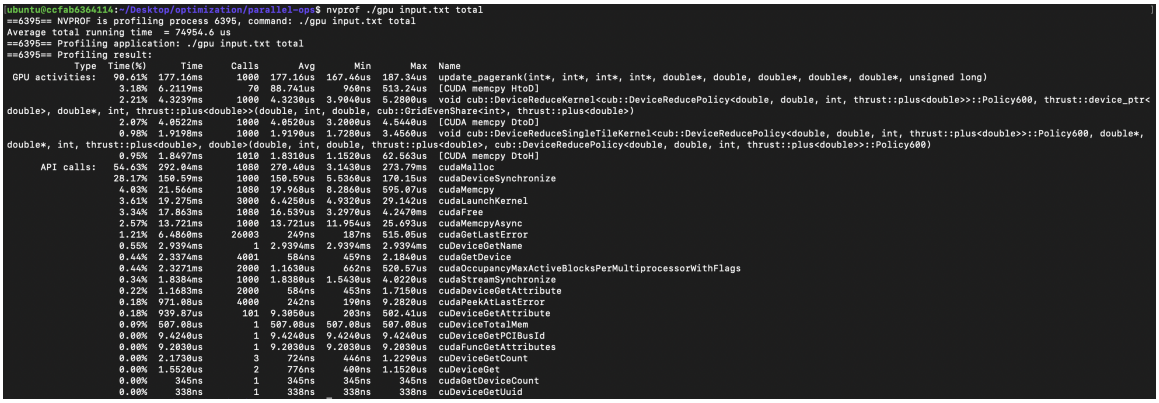

Figure 4.13: Nvidia profiler(summary mode) result of 100,000 nodes with 1000,000 links

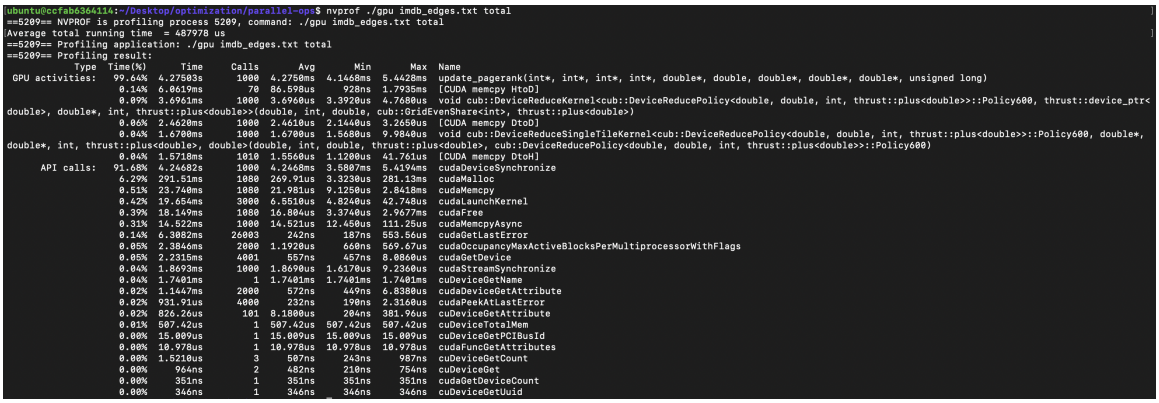

Figure 4.14: Nvidia profiler(summary mode) result of 66,892 nodes with 686,829 links(from IMDb)

As can be seen, these profiling results give a breakdown of the time spent in each function, both in raw time and percentage of overall run time on the host side and the device side. They also show the number of times that each function is called and some basic statistics (min, max, average) on time per call. Since I set the iteration times for the kernel to run 100 times and set 10 times loop to run the whole graph generation and PageRank calculation in the main method so that some parts of the code are called 1000 times, for example, the kernel. Observe that, on average, our CUDA kernel actually executes in just 132.18 microseconds with 66,892 nodes with 686,829 links(generated), 177.16 microseconds with 100,000 nodes with 1000,000 links and 4.275 milliseconds with 66,892 nodes with 686,829 links(from IMDb). So it is much faster than the single-threaded baseline version in computation since it spends more than 90% of the GPU in the computing process rather than data transferring after we move all the calculations in the while loop to the device side. However, we notice that the experiment on realistic data set(from IMDb) has a relatively worse performance using GPU computing. Since we can see that there are over 99% of time slots consuming by the CUDA kernel, I assume the worse performance is caused by the complexity of the data set. As the distributions of movies and references are analyzed in section 4.2, the source nodes follow Log-normal distribution and the destination nodes follow an even more complex distribution which is Birnbaum–Saunders distribution. They are very different from the customized datasets which only contain uniform and normal distribution data. So that the IMDb dataset requires more time on calculating PageRank values in the kernel.

Chapter 5

Conclusions and Future Work

In this report, some useful optimization techniques including single-threaded optimization, multi-core optimization and GPU computing are discussed firstly. Also, many of these techniques have been implemented into the PageRank algorithm implementation. The substantial performance improvements are observed out of the two systems show that the implementation of these techniques is highly effective for big data system optimization. In the process of optimizing the PageRank implementation, it is found that profile the program first to locate bottlenecks of performance is very crucial. In addition, through the experiment, we observe that before step into any parallelism, substantial performance improvement can still be achieved on only one thread by refining poor data locality. This work allows other big data system developers to achieve further potential performance improvement on their systems.

In the future, the current implementation of the PageRank algorithm can be tuned further. Other optimization techniques could be added to achieve better performance. A tool can also be generated to combine these optimization techniques which provides a way to optimize the unoptimized code.

Bibliography

[1] "Memory-bound function." `https://en.wikipedia.org/wiki/Memory-bound_function`. [Online; accessed 19-July-2021].

[2] Intel Corporation., *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[3] M. S. P. Boldi and S. Vigna, "Pagerank: Functional dependencies," *ACM Transactions on Information Systems (TOIS)*, vol. 27, no. 4, pp. 1–23, 2009.

[4] "Pagerank." `https://neo4j.com/docs/graph-data-science/current/algorithms/page-rank/`. [Online; accessed 19-July-2021].

[5] H. Sutter, "The free lunch is over - a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005.

[6] A. Jacobs, "The pathologies of big data," *Commun. ACM*, vol. 52, p. 36–44, Aug. 2009.

[7] R. Hyde, "The fallacy of premature optimization," *Ubiquity*, vol. 2009, Feb. 2009.

[8] T. Y.-j. WU Jia-qi, "Pagerank algorithm optimization and improvement," *CEA*, vol. 45, no. 16, p. 56, 2009.

[9] A. Anikin, A. Gasnikov, A. Gornov, D. Kamzolov, Y. Maximov, and Y. Nesterov, "Efficient numerical methods to solve sparse linear equations with application to pagerank," 2015.

[10] A. Das Sarma, A. R. Molla, G. Pandurangan, and E. Upfal, "Fast distributed pagerank computation," *Theoretical Computer Science*, vol. 561, pp. 113–121, 2015. Special Issue on Distributed Computing and Networking.

[11] S. Lai, B. Shao, Y. Xu, and X. Lin, "Parallel computations of local pagerank problem based on graphics processing unit," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, p. e4245, 2017. e4245 CPE-17-0114.R2.

[12] F. Sadi, J. Sweeney, S. McMillan, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Pagerank acceleration for large graphs with scalable hardware and two-step spmv," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, 2018.

[13] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing memory performance for fpga implementation of pagerank," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, 2015.

[14] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 293–302, 2017.

[15] "Intel vtune profiler user guide." `https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/introduction.html`. [Online; accessed 19-July-2021].

[16] "Cuda toolkit documentation - nvprof overview." `https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview`. [Online; accessed 20-July-2021].

[17] "Aos and soa." `https://en.wikipedia.org/wiki/AoS_and_SoA`. [Online; accessed 20-July-2021].

[18] T. Chilimbi, M. Hill, and J. Larus, "Making pointer-based data structures cache conscious," *Computer*, vol. 33, no. 12, pp. 67–74, 2000.

[19] "Openmp." `https://en.wikipedia.org/wiki/OpenMP`. [Online; accessed 20-July-2021].

[20] "Cuda toolkit documentation - cuda c++ programming guide." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. [Online; accessed 25-July-2021].

[21] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*. Saint Louis;San Diego;: Morgan Kaufmann [Imprint], 3rd ed., 2016;2012;.

[22] "Pagerank." `https://en.wikipedia.org/wiki/PageRank`. [Online; accessed 19-July-2021].

[23] G. P. Atish Das Sarma, Anisur Rahaman Molla and E. Upfal, "Fast distributed pagerank computation," *Theoretical computer science*, vol. 561, pp. 113–121, 2015.

[24] "Imdb datasets." `https://www.imdb.com/interfaces/`. [Online; accessed 05-July-2021].

[25] "Link analysis of imdb movie connections." `https://zegoggl.es/2010/12/link-analysis-of-imdb-movie-connections#disqus_thread`. [Online; accessed 30-August-2021].

[26] "Movie connections." `https://help.imdb.com/article/contribution/titles/movie-connections/GNUNL9W2FTZDGF4Y#`. [Online; accessed 30-August-2021].

[27] `https://github.com/goldshtn/spark-workshop/blob/master/data.zip`. [Online; accessed 29-June-2021].

[28] "Statistical functions (scipy.stats)." `https://docs.scipy.org/doc/scipy/reference/stats.html`. [Online; accessed 9-July-2021].

[29] "Run microarchitecture exploration analysis." `https://software.intel.com/content/www/us/en/develop/documentation/vtune-hotspots-tutorial-linux-c/top/run-microarchitecture-exploration-analysis.html`. [Online; accessed 19-July-2021].

[30] "hardware_concurrency." `https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency`. [Online; accessed 28-July-2021].

[31] "An introduction to the thrust parallel algorithms library." `https://www.sie.es/wp-content/uploads/2015/12/`

Intro-to-Thrust-Parallel-Algorithms-Library.pdf. [Online; accessed 30-July-2021].

[32] "Date and time utilities." https://en.cppreference.com/w/cpp/chrono. [Online; accessed 30-July-2021].

[33] "Cuda pro tip: nvprof is your handy universal gpu profiler." https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/. [Online; accessed 30-July-2021].