ANALYSIS OF SCRATCH-PAD MEMORY-BASED PROCESSOR ARCHITECTURE FOR GRAPH APPLICATIONS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Muhammad Aamir Saeed

September 2021

Analysis Of Scratch-pad Memory-based Processor Architecture For Graph Applications
By Muhammad Aamir Saeed
September 2021

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

M. Mustafa Özdal(Advisor)

Özcan Öztürk(Co-Advisor)

Naveed ul Mustafa(Co-Advisor)

Uğur Güdükbay

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

ANALYSIS OF SCRATCH-PAD MEMORY-BASED PROCESSOR ARCHITECTURE FOR GRAPH APPLICATIONS

Muhammad Aamir Saeed
M.S. in Computer Engineering
Advisor: M. Mustafa Özdal
Co-Advisor: Özcan Öztürk
Co-Advisor: Naveed ul Mustafa
September 2021

In graph analytic applications, main memory accesses prove to be a bottleneck as graphs have a poor spatial and temporal locality usage in the caches and higher memory hierarchy. Although this bottleneck is slightly mitigated with the use of miss status handling registers (MSHRs) in caches, the problem becomes more significant in the case of large graphs. The MSHR, which relies on an out-of-order processor's reorder buffer, becomes quickly saturated as the memory requests keep on piling up because of the limited instruction window size.

To tackle the memory bottleneck for graph applications, the use of a Scratch-pad Memory (SPM) together with custom instructions is proposed. This model is implemented and tested on a custom in-order processor using the x86 architecture to accommodate the related custom instructions. The custom instructions provide non-blocking access to data from the main memory while overlapping with other non-blocking instructions in the CPU pipeline.

This design is evaluated on an industry-level simulator, GEM5, and uses the graph kernels from the GAP Benchmark to test the proposed system. The system shows a speedup of up to 7x for PageRank while averaging a speedup of 1.5x for the other graph kernels such as Single-Source shortest path, Connected Components, and Triangle Counting.

Keywords: Domain-specific processor, custom x86 instructions, iterative graph applications, Scratch-pad memory, GEM5.

ÖZET

ÇIZGE UYGULAMALARI İÇİN KARALAMA DEFTERİ BELLEĞİ TABANLI ÖZELLEŞTİRİLMİŞ İŞLEMCİ MİMARİSİ ANALİZİ

Muhammad Aamir Saeed
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: M. Mustafa Özdal
İkinci Tez Danışmanı: Özcan Öztürk
Üçüncü Tez Danışmanı: Naveed ul Mustafa
Eylül 2021

Çizge analiz uygulamalarında, çizgelerin önbellekte ve daha üst bellek sıradüzeninde zayıf uzamsal ve geçici yerellik kullanımı sebebiyle, ana bellek erişimlerinin bir darboğaz oluşturduğu bilinmektedir. Bu darboğaz önbelleklerde MSHR kullanımı ile ötelense de, problem devasa çizgelerle daha önemli hale gelmektedir. Hizmet dışı olan bir işlemcinin yeniden sıralama arabelleğine ihtiyaç duyan MSHR, sınırlı komut pencere uzunluğu sebebiyle bellek isteklerinin birikmesi sonucu kolayca doygunluğa ulaşmaktadır.

Çizge uygulamalarında bellek darboğazı problemini gidermek için özel komutlar ile beraber Karalama Defteri Belleği (Scratchpad Memory (SPM)) önerilmiştir. Model, özel komutları yerleştirmek için x86 mimarili özel bir sıralı işlemci kullanılarak uygulanmış ve test edilmiştir. Özel komutlar ana bellekteki veriye erişim sağlarken birbirleriyle örtüşür ve merkez işlemci birimi (CPU) boru hattındaki diğer komutları engellemez.

Bu tasarım endüstri seviyesinde bir simülatör olan GEM5 ile değerlendirilmiş ve önerilen sistemin testlerinde GAP uyguluma (benchmark) bulunan çizge çekirdekleri kullanılmıştır. Sistem sayfa sıralama (PageRank) algoritması için 7 kata kadar hız artışı gösterirken, tek kaynaklı en kısa yol (single-source shortest path), bağlı bileşenler (connected components) ve üçgen sayma (triangle counting) gibi diğer grafik çekirdeklerinde ortalama 1.5 kat hız artışı sağlamaktadır.

Anahtar sözcükler: Alana özgü işlemci, Özel x86 komutları, Yinelemeli çizge uygulamaları, Scratch-pad memory, GEM5.

Acknowledgement

I would like to thank my supervisor Dr. Mustafa Özdal and co-supervisors Dr. Özcan Öztürk and Dr. Naveed ul Mustafa, who gave me the opportunity to work on this project. Their support and contribution to this thesis is invaluable as it would not have been possible without them.

I would also like to thank Bilkent University and its Department of Computer Engineering for allowing me to pursue my master's thesis.

Lastly, I would like to thank my family and friends who provided me with immense support throughout my time at Bilkent University. I will forever be grateful to them.

This work has been supported in part by a grant from Technological Research Council of Turkey (TUBITAK) 1001 program through the EEEAG 119E559 project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the widespread use of big data applications, graphs continuously increase and gain popularity as their application in popular domains such as machine learning, artificial intelligence, robotics, and biology ever grows[1] [2]. Moreover, representing data in graphs has many advantages over the traditional data structures, such as better visualization and systematically representing a large amount of data. Hence, it has become a popular choice of data structure. To process data in graphs, several algorithms have been designed for particular domains.

For example, PageRank (PR), a popular way to rank pages on the web and find sentence similarity, is a significant application in the graph world. Similarly, Single-Source Shortest Path (SSSP) algorithm is used to find the shortest path between two places, i.e., between two nodes in a graph, and is widely used in cognitive systems. Betweenness Centrality (BC) determines how much influence a node has over the flow of information in a graph and is used to analyze social networks. While these algorithms are used for different purposes, they all execute iteratively and converge after a certain number of iterations.

Although graphs are continuously becoming a popular choice of a data structure, they lack hardware support for quick and efficient processing. Processing large graphs of millions of vertices and billions of edges can sometimes take days

if not weeks, even with increased processor speeds [3]. Finding a practical implementation of a graph has many challenges due to the complex pattern and large size. Large graphs tend to require a large amount of memory which is not readily available on most systems that process graph data of such large sizes. This calls for developers to focus on application-specific hardware in specific domains such as machine learning and signal processing. There are many efforts to improve the efficiency of such systems. Although these systems do mitigate the problem, they still cannot process graphs efficiently because most graph applications are proven to have poor memory locality and data reuse[3], [4].

Due to the poor locality of the graphs in the graph application, the system tends to make many irregular random memory accesses which further slows down the system. Accessing the main memory has become quite expensive for a processor due to the memory wall since processor speeds continue to increase exponentially while memory speeds have not improved over the years at the same rate as the processors.

Researchers have developed various methods to overcome the memory wall. Some solutions are hiding latencies by the usage of non-blocking caches, prefetching data, and speculative load. Other solutions include introducing newer technologies such as on-chip memories rather than off-chip memories, using instruction-level parallelism, and introducing virtual memory. Of the solutions mentioned, the most widely used in practice today is the usage of non-blocking caches. Non-blocking caches employ the Miss Status Handling Register (MSHR) to overcome the memory wall. The MSHR is a special purpose register that holds cache miss information while allowing the processor to continue processing other instructions. However, this approach calls for introducing more complex out-of-order processors, increasing hardware threads, or using multiple cores. The same goes for the earlier solutions, which have drawbacks, such as synchronization overhead cost, more complex hardware, and load imbalance. [5], [6].

1.1 Research Problem

Many of the graph applications are known to have memory access as the main bottleneck in their performance. The memory bottleneck arises from the following issues: i) memory latency being slow compared to the processor, and ii) low memory level parallelism, which in most cases does not fully benefit from the DRAM bandwidth.

Many out-of-order processors employ caches with miss status holding registers (MSHR) to solve this problem, allowing the processors to continue processing while a miss has been recorded. Using MSHRs allows for memory level parallelism to a certain extent, but it cannot be fully achieved as most processors have a limited number of MSHRs because they need to be searched associatively, limiting their number to a few. According to the authors in [7] simply increasing the number of MSHRs does not solve the problem as many graph applications are not fully utilizing the MSHRs. Also, the instruction windows are not large enough to allow multiple reordering of instructions, thus limiting the MSHR functionality.

While having multiple cores enables better utilization of memory bandwidth since more requests get sent to the main memory, it also increases the cache misses, synchronization overheads, and sometimes load imbalance penalties. Furthermore, multiple cores demand more energy thus are not energy efficient [6].

For the memory bottleneck problem mentioned above, assume a system with ten MSHRs and a graph application running on this system has made more than ten random data memory requests from the main memory. The MSHRs will only register the first ten in the cache, stall the CPU pipeline, and wait for the main memory to respond then process the eleventh memory request. Assuming that each access to the main memory costs one hundred cycles in a system with ten MSHRs and it takes one cycle to process it, after sending ten memory requests to the main memory, the eleventh memory request will have to wait for at least 92 cycles before proceeding onwards 1.1. As it is seen from the Table 1.1, the first long instruction would leave the pipeline on the 102^{nd}, and the 11^{th} long

Table 1.1: Long latency instructions getting blocked due unavailability of MSHRs

Instruction	Issue Cycle	Leaving Pipeline	Retire cycle
LLI 1	1	102	102
LLI 2	2	103	103
LLI 3	3	104	104
LLI 4	4	105	105
....	...	...	...
...	...	...	...
LLI 10...	10	111	111
LLI 11	Blocked for 92 cycles due to non-availibity of MSHRs		

latency instruction will then enter on the 103^{rd} after waiting for 92 cycles. The CPU stall, as shown in Figure 1.1 , can quickly become expensive when we have numerous accesses to the memory. For graph applications, the memory accesses tend to be random, thus making it more challenging.

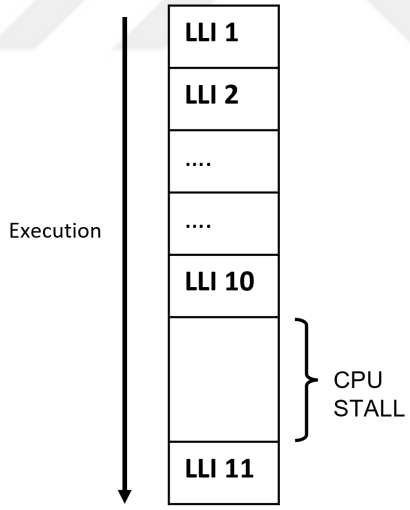

Figure 1.1: A CPU stall that is being caused due to the unavailability of an MSHR

1.2 Objective of the Thesis

This thesis endeavors to provide an analysis of a graph domain-specific processor that improves the performance of graph applications using minimum hardware

resources. We eliminate the CPU stall caused by the long latency instructions by sending them to the memory in batches and overlapping these long latency instructions. We introduce a Scratch-pad Memory (SPM) to store graph data with the assistance of custom instructions. We overlap memory accesses as much as possible while at the same time avoiding the stalls in the processor. The SPM is software controlled, thus allowing the user with more freedom on what to store inside it, unlike a conventional cache system and the MSHRs. The main contributions of this thesis can be summarized as follows:

- We analyze a custom graph processor tuned for graph applications for better performance and efficiency.
- An SPM is added to our system to store and retrieve graph data according to the processor's request at the shortest time possible.
- We extend the x86 architecture instruction set with custom instructions to control and communicate with SPM, CPU, and the main memory.
- We introduce custom instructions that allow for multiple memory accesses simultaneously and overlap with one another.
- We provide extensive results on how the graph applications utilize our custom instructions and compare the results with the GAP benchmark under different conditions.
- We also provide a mechanism on how the graph applications can use our custom graph processor and increase their performance output.

1.3 Organization of the Thesis

After the introduction chapter, the thesis moves on to the next chapter, which covers previous works, current frameworks, and the problems faced by the conventional systems. Afterward, in Chapter 3, background information is given,

while Chapter 4 explains our system's hardware and the software architecture in detail.

Chapter 5 presents how the various graph algorithms are designed to benefit from the graph processor. Chapter 6 starts with the experimental setup of our system that we use to collect results. We then present the performance results of the various graph applications that we run on our graph processor. Specifically, we present the results for PageRank, Single-Source Shortest Path, Connected Components, and Triangle Counting. We also do a sensitivity analysis of our system and provide the results in this chapter.

Chapter 7 discusses our observations and conclusions from the experimental results. Finally, we conclude the thesis with Chapter 8, giving a brief overview of our work, accomplishments, and future remarks.

Chapter 2

Related work

This chapter gives a brief overview of all the related work that has been done on graph applications to increase their performance. There have been various recent and old studies that look at graph applications from different angles. Broadly, these studies can be divided based on the work done on the software and the hardware.

There have been numerous ways in which researchers have addressed the inefficiencies of graph application. Of the many such ways, is through the use of Processing-in-Memory (PIM) to overcome the memory wall [8], [9], [10], [11], [12] is the most prominent way. Such architectures feature ingenious memory-related optimizations, such as using non-blocking message passing that covers communication latencies. Initially, it was not successful due to design complexity and fabrication limitations. However, recently PIM has regained attention due to advances in 3D stacking technologies. GraphPIM [13] uses PIM and can achieve a speedup of up to 2.4X, and by processing in memory, it helps the processors avoid irregular data accesses. Another application that uses PIM is GraphH [14], which achieves 5.12X speedup when compared to an earlier work done on PIM [15]. Other applications are Gemini[16] and Graphicionado [17].

Large-scale graph processing systems have also been developed for graph applications. These systems execute on different platforms, including heterogeneous systems, single machine systems, and distributed systems. Heterogeneous systems use both a general-purpose system and specialized processing units like GPUs and FPGAs.

Some of the heterogeneous systems utilize GPUs. GPUs are power efficient and can carry out high memory bandwidth processing, which is essential for processing graph applications. Furthermore, they can exploit parallelism in computationally demanding applications and have the potential to accelerate sparse matrix algebra due to their memory-bound nature. A variety of optimizations have been performed in the literature to improve the performance of sparse matrix-vector multiplication (SpMV) [18]. Although GPUs offer many advantages, programming them to attain high memory bandwidth can be a challenge. Also, the performance can sometimes be limited by the interface from the CPU to the GPU, thus creating an additional unwanted bottleneck.

Xing et al. have presented HPGraph[19], which is a GPU graph framework, which maps vertex programming to an optimized matrix back end. Their results, when compared to the advanced CPU library, show significant improvement. Khorasani et al. [20] also use GPUs where they use wrap segmentation to improve SIMD efficiency and achieve a speedup of up to 1.29 to 2.8X. Merrill et al. [21] also present a theory that GPUs are well-suited for sparse graph traversal and can achieve very high levels of performance on a broad range of graph applications.

On the FPGA level, many researchers have offloaded the processing of graph applications to the FPGA environment, which allows the main CPU to handle other rudimentary tasks. The initial attempts to introduce application-specific solutions to graph processing were made by specifically customizing the datapath of the processors for improved performance. For example, graph search [22], and reduction [23] have both been improved by expanding the datapaths of general-purpose processors in these architectures with additional units such as a re-configurable stack, an ALU, and address registers.

In more recent efforts, more sophisticated systems have emerged, allowing for more flexibility. For example, ForeGraph [24] is a multi-FPGA architecture that works by partitioning the graphs on the FPGA BRAMs. By doing this, they achieve a 5.89X speed up and increase their throughput by up to 2.03X. Another framework is GraphGen [25], where an update function is defined in the specification as a collection of custom graph instructions that are mapped to the graph processor of an FPGA. It targets vertex-centric graphs and uses off-chip DRAMs to manage the graph data. They achieve up to 14X speedup when compared to typical CPU implementations. Cygraph[26] is another implementation of an FPGA framework that introduces an application-specific graph representation based on the compressed sparse row (CSR) format to attain a speedup of almost five times.

Single machine systems have similar implementations and can be further divided into two categories: single machine shared-memory systems and single machine out-of-core systems.

Single machine shared-memory systems have large memories which can fit large-sized graphs of billions of edges. The advantage of such a system is that it eliminates an unnecessarily large amount of communication overhead. Also, storing data in the memory saves disk access. Multiple systems have been proposed, and out of all of them, Ligra [27] stands out as state of the art. Ligra provides routines for mapping over edges and vertices and is highly suitable for traversal-based graphs. GRACE [28] is another platform that uses message passing to process graphs and offers a synchronous iterative graph programming model.

Single machine out-of-core systems represent a disk-based single machine graph processing system where only a tiny portion of a large graph is kept in the main memory while the rest is kept in the disk. Although this poses a significant challenge since accessing the disk is costly and slows the performance radically, it can be executed on an off-the-shelf general-purpose processor. Several novel methods have been introduced that mitigate this problem by just using a simple machine. One example is GraphChi [29] which uses a parallel sliding window for processing large graphs that are on disks. It implements an asynchronous

model of computation, therefore, requiring very few non-sequential accesses from the disk. Another significant work in this category is Pearce [30] which also proposed an asynchronous system for graph traversals. The proposed solution uses CSR graph format, and the computation work is scheduled using concurrent work queues.

Additionally, there are distributed systems made of multiple processing nodes where each node has its memory. Compared to single machine shared memory systems, distributed systems are less limited by the hardware in terms of scalability. However, implementing distributed systems has its challenges, such as finding the appropriate data partitioning strategy, which significantly affects the system's performance. Also, data that has been partitioned over the processing nodes need to communicate over the partitions, which further increases the memory bottleneck. Nevertheless, improvement is noticeable in distributed systems.

One of the important works on distributed systems is Google's Pregel [31] framework, which leverages bulk synchronous parallel model and provides a synchronous vertex-centric framework. It aims to give users a natural application programming interface (API) for programming graph algorithms while managing communication specifics in the back-end, such as messaging and fault tolerance.

Many other distributed systems have been designed on top of the Pregel framework, such as Trinity [32], which takes the memory of multiple machines and organizes them into a globally addressable memory to allow for addressable memory cloud. Another platform that is built on top of Pregel is GraphLab [5]. It is an asynchronous distributed shared memory abstraction that is designed for machine learning and data mining algorithms on graphs. GraphX [33] is another example, which unifies advances in graph processing systems with advances in data-flow systems. It can perform general-purpose data-flow operations as well as graph computations quickly. CGMgraph [34] has a similar concept for distributed systems by providing several parallel graph algorithms using the Coarse-Grained Multicomputer model based on MPI.

Another novel way of getting rid of the memory wall is by introducing custom

instructions to a system. The instruction-set extension with new custom instructions can broadly be divided into instruction generation and custom instruction selection. The process of generating custom instructions encompasses grouping some basic operations into larger and more complex operations, where subgraphs of the application graph identify these complex operations. The generation and selection of the subgraphs are made through a guide function and cost function.

An example is the work by Cheung et al. [35] which generates custom-instruction templates based on an exhaustive search. In contrast, GraphGen [25] describes an update function as a composition of custom graph instructions, which on compilation generates a sequence of custom instructions that use the scope and temporary data variables to perform an update on a given vertex. The custom instruction generation introduced in [36] and [37] rely on incremental clustering of related data-flow graph nodes using heuristic approaches to identify recurring and reusable custom instruction patterns.

In the custom instruction selection, certain parts of the instructions are chosen to execute on a custom function unit, e.g., FPGA, while the rest are run on the base processor. This mechanism allows for data-level parallelism. Sebastian et al.[38] introduce automatic rules which select instructions for architecture-specific based on the mapping generated by the Partitioned Boolean Quadratic Problem (PBQP). Pan Yu et al. [39] use a heuristic approach and present an efficient algorithm for exact enumeration of all possible instructions for a given data flow graph.

Another example of extending the instruction set architecture is where Nachiket et al.[40] uses custom instructions on FPGA-based soft processors to customize the execute stage of the pipeline depending on the graph application via ucoded datapaths. The methods mentioned above of introducing custom instructions all rely on the usage of FPGAs together with a base processer. Like the GPUs, they sometimes face a bottleneck when the CPU and the FPGA want to communicate.

All of the frameworks above try to solve the memory bottleneck problem by

employing various methods and optimizations. Some of them require extra resources, which tend to be expensive, while others are not scalable to other platforms. In this thesis, we try to introduce minimum resource requirements. Apart from the custom instructions and required logic to manage these instructions, the only additional hardware required in this work is a Scratch-pad Memory, which tends to be very small. At the same time, custom instructions improve the overall performance of the processor as well.

Chapter 3

Background

This chapter gives a brief theory of graph applications, instruction set architecture and Scratch-pad Memory as they form the background of this thesis.

3.1 Graph Applications

Graphs are made of vertices which are connected to each other via edges, formally they are described as a pair of sets (V, E), where V is an arbitrary non-empty finite set, whose elements are called vertices or nodes, and E is a set of pairs of elements of V, called edges. Broadly they can be divided into two categories, i.e., directed and undirected, where the former has ordered pairs while the latter has unordered pairs. Graphs can be of different types, representing data in various ways to allow algorithms to process them efficiently.

In this thesis, we consider iterative and vertex-centric graphs algorithms due to their specific characteristics. These graph algorithms process graphs in multiple iterations until a specific criterion is fulfilled. Most criteria are marked by a point of convergence or by finding a particular vertex in the graph. In every iteration of the algorithm, an active vertex scans its edges looking for relevant information and then locally applies a pre-determined function to itself. This method is commonly

known as Gather-Apply-Scatter (GAS), wherein the gather and scatter stages refer to the vertex's communications with the edges and neighboring vertices, and the apply stage refers to the vertex's local computation. One such example is the PageRank algorithm. The vertices gather the ranks of their in-neighbors in each iteration of PageRank, perform local computations on the gathered data, and scatter their resulting ranks to their out-neighbors as shown in Figure 3.1.

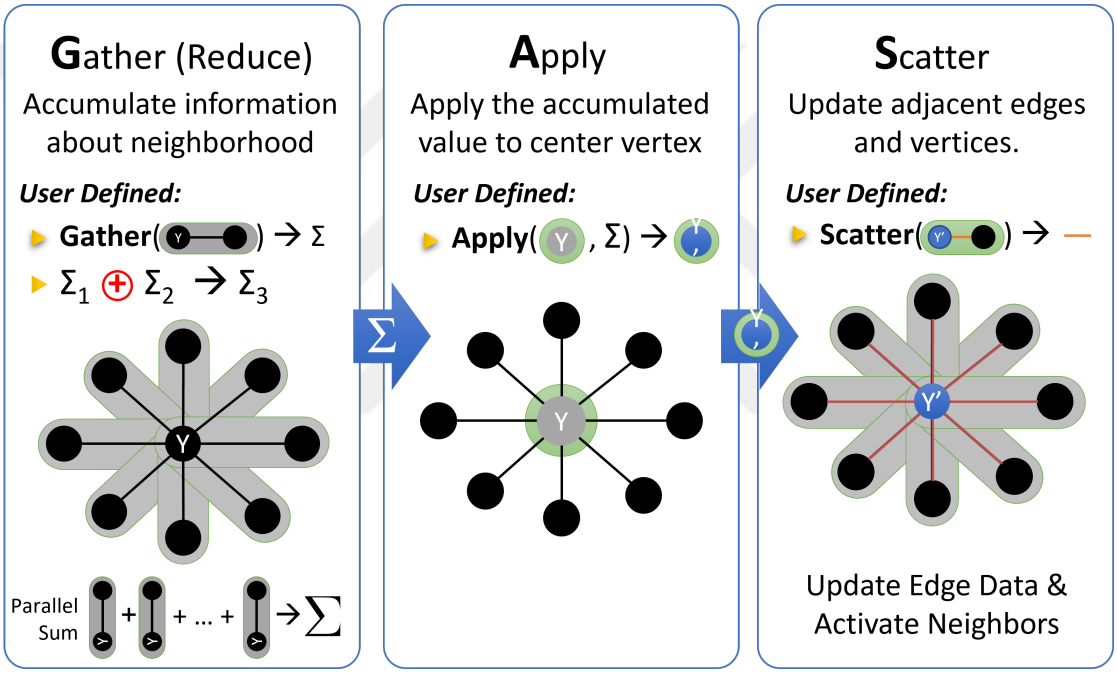

Figure 3.1: The operational steps of Gather, Apply, and Scatter

For these algorithms to find the required criteria, the vertices collect information from their neighbors, which are usually scattered across the graph. Information gathering from the neighbors has to be done in every iteration until the criterion is fulfilled. Accessing the information from the scattered neighboring vertices translates to accessing random memory locations in the memory. Therefore, the algorithms have very poor locality in the lower levels of the memory. Furthermore, such graph algorithms are known to have communication-centric workloads that cause memory bottlenecks because the arithmetic work performed at each iteration is very small compared to the number of communications that must be performed to reach the neighboring vertex.

3.2 x86 Instruction set architecture

The Instruction Set Architecture (ISA) is the part of the processor visible to the programmer or compiler writer. The ISA serves as the boundary between software and hardware. Various ISAs define their own rules on how to program hardware via the software interface. The most common ones are x86, ARM, RISC-V, SPARC, and MIPS. Apart from RISC-V, which is fully open-source, all the rest are proprietary, where users do not fully control the hardware programmability. x86 and ARM are the most common ones deployed on almost every processor ranging from embedded systems to high-end supercomputers, even though they are proprietary.

To evaluate and extend the ISA with our custom instructions, we choose the x86 ISA. The x86 ISA is a Complex Instruction Set Computer (CISC) architecture developed by Intel and AMD. CISC architecture is defined as where single instructions can perform multiple low-level operations, such as storing data in memory, loading from memory, or performing an arithmetic operation. They are capable of multi-step operations or addressing modes within a single instruction.

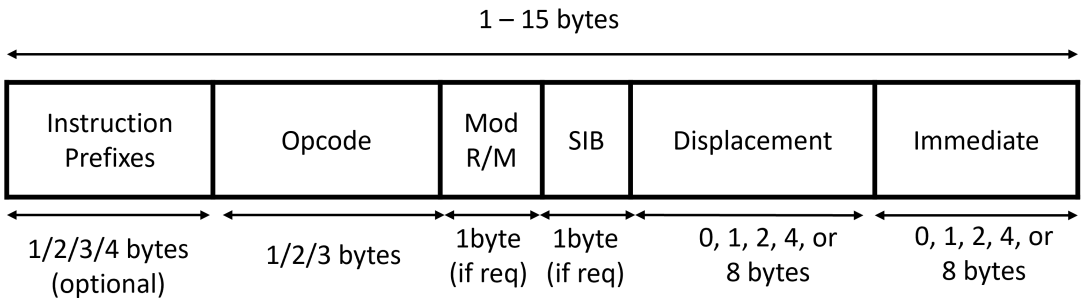

Figure 3.2: The breakdown of the x86 instruction encoding format

Instructions can range in length from 1 to 15 bytes, as shown in Figure 3.2. In the x86 instruction format, the instruction must include an opcode, displacement, and an immediate byte where opcodes are 1 to 3 bytes long and represent the operation to execute. Assembly mnemonics, such as ADD, MOV, or RET, are commonly used by programmers and can be represented by various opcodes in

the x86-64 architecture (e.g., at least 26 different opcodes correspond to the ubiquitous MOV instruction).

Each opcode can also be prefixed with several other prefixes to alter its semantics. For example, in our thesis, the data transfer command, MOV, appears with 61 different prefixes. Prefixes such as 0x66 and 0x67 toggle between 16-bit and 32-bit operands, REX specifies 64-bit operands, and VEX denotes vector operands, overriding the default operand and address sizes. A prefix can also signal atomicity (LOCK), repetition (REPNE/REPE), overriding segment registers (e.g., 0x64 to override the FS segment) for an instruction. Prefixes are usually optional; however, they may be required for some instructions. Typically, a MOV instruction does not include a repetition prefix, but we introduce one in this thesis and set the custom MOV instruction to perform the necessary operations as required as per the research's objective.

The ModR/M byte specifies the registers that the instruction operates on and the addressing mode, and whether or not the SIB byte is present. If the memory operand is addressed using the indexed or scaled addressing modes, the SIB byte contains the scale, index, and base to use. The instruction's final two sections are used to encode displacement or immediate values if needed. Because of different operands and addressing modes, instructions with identical prefixes and opcodes might be of varying lengths: for example, the most frequent encoding of the MOV mnemonic (0x48 8b) comes in 5 different widths — 3, 4, 5, 7, and 8 bytes.

3.3 Scratch-pad Memory

Unlike the cache, which is hardware controlled, the Scratch-pad Memory (SPM) is a fast on-chip memory that is software controlled. The developer is responsible for reading and writing data from the SPM, therefore providing more control over its usage. By having control, one can decide what to write and when to write in the SPM, hence increasing the efficiency and performance of the desired algorithm. This behavior is desirable as caches sometimes bring in additional

data from the main memory, which is not used, thus wasting resources, especially when the overhead is significant and the data being fetched from memory is not nearby.

In our case, the SPM will be responsible for holding the vertices and edges of a graph, and the graph application will be modified to be intelligent enough to decide what to write in the SPM. Considering the graph applications' random and unique access patterns, our modified graph applications and custom graph processor can avoid the memory bottleneck and increase performance by utilizing the SPM most efficiently.

Chapter 4

Architecture

In this chapter, the architecture of the graph processor is described in detail. The graph processor architecture involves modification at the hardware level and the software level. At the hardware level, we modify the CPU, integrate a Scratch-pad Memory, and introduce new interfaces to handle the new hardware. At the software level, we introduce new custom instructions that will efficiently utilize the introduced hardware.

4.1 Hardware implementation

The hardware implementation resembles the hardware model found in the GEM5[41] simulator because we will be simulating the proposed system on it. The hardware model found in the GEM5 simulator is a near replica of an actual hardware, therefore allowing visualization of a system before actual implementation on the hardware.

The architecture of our proposed system, as shown in Figure 4.1 includes the introduction of a single Scratch-pad Memory. The Scratch-pad Memory is at the same level as a level 1 cache. i.e., the latency and all other data read and write parameters are the same as the level 1 cache.

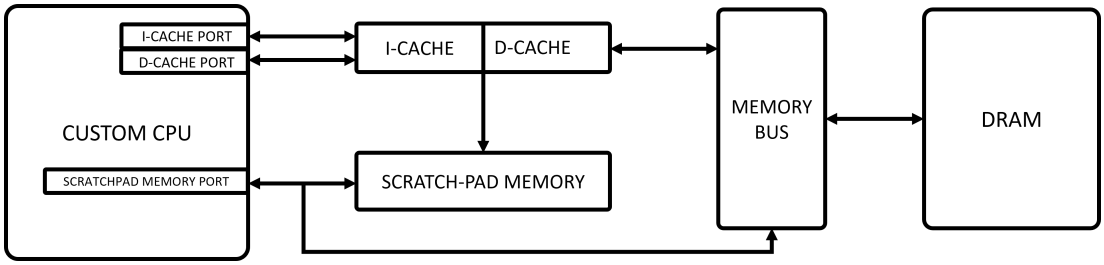

Figure 4.1: Overall block diagram of the system

The introduced Scratch-pad Memory has interfaces with the CPU, level 1 cache, and the main memory bus. An interface between the SPM and the level 1 cache exists because the GEM5 only has a write-back policy for caches and sometimes data required to be stored in the SPM can exist in the level 1 cache thus making this interface necessary.

The Scratch-pad Memory interface with the main memory is done through a memory bus just as it is with the level 1 cache. The memory bus of our system can handle multiple requests at the same time from both the cache and the SPM.

When writing data into the SPM, our graph CPU performs a complex check where it first checks if the data is available in the cache or not. If it is not available in the cache, the request is then forwarded to the memory to be fetched. If the required data is available in the cache, then the data is copied from the cache to the SPM, saving a trip to the memory. Hence, there are three possible ways of writing in the SPM: when we have to write from the cache to the SPM, second when writing the SPM from the main memory, and lastly, the CPU writing to the SPM. All three can write to the SPM without going through the other.

Data from the main memory that is required to be written in the SPM is sent directly to the SPM without going through the cache. By doing so, we avoid data duplication and making the cache available for other resources.

An additional port is added to the in-order CPU in order to utilize the SPM efficiently. The CPU uses this port to handle the reads and writes to the SPM. A visual representation is shown in Figure 4.1 which also shows the direction of data flow of the system. All the ports allow data flow in both directions, apart

from the interface between the cache and the Scratch-pad Memory, which is a one-way direction from the cache to the Scratch-pad Memory.

To handle the custom instructions which will be used, the internal working of the processor is modified as well. The new custom graph processor is built on top of the already available Minor-CPU in the GEM5. The Minor-CPU is an in-order processor model with a fixed pipeline but configurable data structures and execute behavior. It has a fixed 4 stage CPU pipeline, namely Fetch1, Fetch2, Decode, and Execute stage. To meet our design requirements, we modify the Decode and Execute stage. The execute stage is modified the most with respect to the other stages to accommodate the custom instructions.

The Execute stage provides all the instruction execution and memory access mechanisms. The memory access mechanism is split into two: one for regular memory instructions and the other for long-latency memory instructions for the SPM. We shall refer to the instructions bound to the SPM as SPM instructions from now onwards.

The SPM long latency instructions are issued just like any regular instructions, but as they exit the functional unit after calculating their memory address, they are tagged as SPM instructions to make it easier for them to be identified in the following stages and for debugging.

As load instructions are only committed once their memory response is received, they tend to stay in the execution stage's instruction window, which causes a pipeline stall. Therefore, per our design, all SPM instructions should be popped out from the instruction window after their ALU operations have been performed and kept in a separate window or queue. Keeping the long latency instructions in a separate window allows other instructions to continue executing and committing without the processor facing a pipeline stall from this long latency instructions.

The Minor-CPU model has a load and store queue (LSQ) responsible for handling memory requests and responses. We introduce another queue, the SPM

Buffer queue, which will be responsible for handling our SPM instructions as they are sent to the memory for data fetching. The SPM Buffer queue is introduced to avoid a pipeline stall in the CPU which are caused by the SPM instructions.

The SPM Buffer is further broken down into an SPM request and SPM transfer queue. The SPM request queue will hold the SPM instructions that have been sent to the main memory to fetch data. And once the data has arrived back, the SPM instruction is popped from the SPM request queue and transferred to the SPM transfer queue, where it will wait for the barrier instruction.

The SPM request queue allows for multiple overlapping memory accesses to ensure maximum performance. Once the memory response is back, a flag is set for that particular SPM load instruction indicating that its response is back. Also, it is popped from the SPM request queue and pushed into the SPM response queue. When all responses are back for all the SPM load instructions (we issue memory accesses in batches so that they can overlap), an overall flag is set. Later on, the barrier instruction checks the overall flag, whose responsibility is to ensure all the responses have arrived in SPM transfer queue. After the barrier instruction has checked the validity of the data, it directs the data to the SPM and signals the SPM response queue to be cleared. Also, at this point, correct data can be read from the SPM.

The barrier instruction is always issued before we try to read from the SPM and its sole purpose is to check the flags of the SPM queue, which in return checks the flags if the memory response is back or not. The barrier instruction will stall the pipeline if that overall batches flag is not set.

4.2 Simulator implementation

To fully utilize our hardware implementation, we need software support to interface with our hardware. We introduce custom instructions namely **SPMNBL**

and **SPMBARRIER** instruction. These two instructions working together can take full benefit of the graph processor and the Scratch-pad Memory. These custom instructions are built from extending the x86 instruction set architecture (ISA).

x86 instructions typically do a lot of work. While one can implement the functionality of each instruction individually, this is not the case as many instructions do tasks that are quite similar to each other. Each instruction is therefore implemented as a combination of several smaller parts. The entire instruction is typically referred to as a macro-op, while the smaller parts are referred to as micro-ops. To implement an instruction, we first provide the ISA decoder with the information on the macro-op, then we provide an implementation of the macro-op in terms of micro-ops. Finally, we implement the micro-ops that are not already implemented.

Also, in the x86 architecture for each macro-op, we have to provide three different implementations: one that only uses registers, one that reads one of the operands from memory using the address provided in the instruction, and the last implementation uses the address of the instruction pointer to read the operand.

To implement our instructions in the x86 architecture, we extend some already similarly existing instructions. One way to extend the x86 architecture is to use legacy prefixes. Though some legacy prefix instructions already exist in the x86 architecture that already extends some instructions, we add a few more custom instructions in this category. Looking at the AMD64 Architecture Programmer's Manual Volume 3 [42], it is noted that they are five legacy prefixes, out of which 2 of them can be explicitly used to modify an already existing instruction.

These are the LOCK and REPEAT legacy prefixes. For our case and simplicity of introducing a new instruction in the system, we shall focus on the REPEAT legacy prefix, as it is easier to utilize. By default, the REPEAT legacy prefix is used with string instructions that are regularly repeated, though it can be repurposed for other use.

The REPEAT legacy prefix can further be divided into REPE (0xF3), REPNE (0xF2) and REP (0xF3). We shall be using the REPNE legacy prefix to extend our instructions even though we can use REPE or REP equivalently.

To ensure that we do not need to modify our GNU C compiler, we shall try to find an already existing instruction that mirrors our new instruction as it will be easier to take parameters from our C application without errors as we expect to have similar input, output, and instruction execution.

Our first custom instruction: **SPMNBL** is responsible for loading and storing data in the SPM. This instruction will have the same parameters and functionality as that of a MOV instruction. We therefore, extend the MOV instruction with the legacy prefix REPNE while the opcode and the operand parameters remain the same as shown in table 4.1. The SPMNBL instruction is a replica of the MOV instruction but sets it apart as it is the only instruction that can write and read from the SPM, and its micro-ops are the only ones that can put the SPM load instructions inside the SPM queue. The SPMNBL, just like the MOV instruction, can take the following parameters:

- SPMNBL(Source-Address, SPM-Address) – Moving data from the main memory to the Scratch-pad Memory.
- SPMNBL(SPM-Address, Destination-Address) – Moving data from Scratch-pad Memory to the main memory.
- SPMNBL(Immediate Value, SPM-Address) – Moving an immediate value to the Scratch-pad Memory.

As mentioned earlier, an instruction in the x86 architecture is a macro-op, which is made of micro-ops. We therefore also have to introduce our micro-ops. Macro-ops will be called in our application while the CPU will execute them in micro-ops.

Breaking the instruction further as shown in Figure 4.2, we have our custom micro-ops that are part of our SPMNBL instruction. The micro-ops in the

Table 4.1: x86 instructions extended to add new instructions

Legacy Prefix	One-byte opcode	Operand 1	Operand 2
0xF2	88	Eb	Gb
0xF2	89	Ev	Gv
0xF2	8B	Gb,Gv	Eb,Ev
0xF2	B0 + r	Bb	Ib
0xF2	B0 + r	Bv	Iv

SPMNBL instruction are: **spmld**, **spmaddress** and **spmst**. Their functionalities are as follows:

- **spmld:** This micro-op is responsible for carrying the data from the main memory to a temporary CPU register.
- **spmaddress:** This micro-op is responsible for holding the address of the SPM at which we will store the data of the spmld instruction
- **spmst:** This micro-op is responsible for transferring the data that is in the CPU register to the SPM using the address provided by the spmaddress instruction.

. The second instruction is the **SPMBARRIER** instruction. The SPMBARRIER is a lock instruction that is executed to confirm if all the data directed to the SPM has received a response from memory. Once the responses are confirmed, it signals the CPU to transfer all data from the SPM transfer queue to the SPM. It can stall the CPU pipeline if any response from a batch of SPMNBL instruction is not yet back from the main memory and will wait for the response to come back before allowing the CPU to move on to the next instruction. The SPMBARRIER instruction takes no parameters.

The SPMBARRIER is relatively easy to implement as it takes no parameters. The NOP instruction, which already exists in the x86 architecture, is used and is extended using the REPNE legacy prefix, similar to the SPMNBL instruction. The SPMBARRIER is by itself a micro-op, and therefore it is not broken down any further.

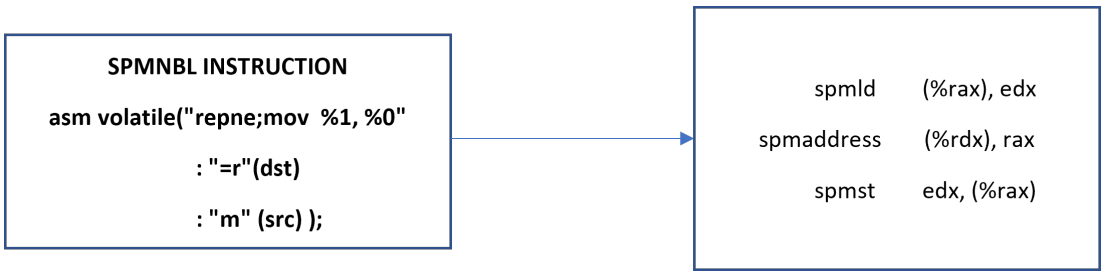

Figure 4.2: Break down of a macro-op into micro-op

Chapter 5

Graph Kernels

In this chapter, we describe the graph kernels used to measure our new system's performance. All the graph kernels are derived from the GAP benchmark as the GAP benchmark is our base for comparison.

Out of the six graph kernels available in the GAP benchmark, we chose four which are PageRank, Single-Source Shortest Path (SSSP), Connected Components (CC), and Triangle Counting (TC). The aforementioned are chosen because they are vertex-centric graphs with a communication-centric workload. Their access patterns are random therefore having a poor locality. The algorithms provided in this chapter are both the native and the modified version that utilizes the SPM and custom instructions.

5.1 PageRank

The PageRank algorithm is a well-known graph application that Google originally used to rank pages on their search engine. It became widely popular in many frameworks as it displays the unique properties of a graph. It captures the irregular memory access, work scheduling, and load imbalance characteristics of many graph algorithms.

The goal of the PageRank algorithm is to compute the PageRank score of each vertex in the graph iteratively until a convergence point is reached. A vertex score is calculated as follows, where d is a damping factor.

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N^-(v)} \frac{PR(u)}{\left|N^+(u)\right|} \tag{5.1}$$

The algorithm that is implemented in the GAP benchmark is shown in Algorithm 1.

Algorithm 1: PageRank Algorithm

```
input  : G(V, E), epsilon, max_iteration, damping factor(d)
output: score
score[:] ← 1 / no_of_nodes;
base_score ← (1 − d) / no_of_nodes;
for i ← 0 to max_iteration do
    error ← 0;
    for u ∈ V do
        outgoing_contrib[u] ← scores[u] / outdegree(u);
    end
    for u ∈ V do
        incoming_total ← 0;
        for v ∈ in_neigh(u) do
            incoming_total += outgoing_contrib(v);
        end
        old_score[u] ← score[u];
        score[u] ← base_score + (d * incoming_total);
        error ← error + |score[u] − old_score[u]| ;
    end
    if error < epsilon then
        break;
    end
end
```

For the PageRank algorithm to utilize the SPM and our system, it has been modified such that the basic principle of the algorithm is that there is a read and write pointer in the SPM, where the write pointer is always a spmbuffersize ahead of the read pointer. The algorithm is described in algorithm 2 and 3.

Algorithm 2: PageRank Algorithm SPM Aware

```
input  : G(V, E), epsilon, max_iteration, damping factor(d)
output: score
 /* graph is stored in CSR format it is taken and utilised as
it is  */
score[:] ← 1 / no_of_nodes;
base_score ← (1 - d) / no_of_nodes;
spm_outgoing_contrib ← firstSPMaddress;
for x ← 0 to max_iteration do
    error ← 0;
    for u ∈ V do
        outgoing_contrib[u] = scores[u] / outdegree(u);
    end
    read_node ← 0;
    read_edges ← 0;
    incoming_total ← 0;

     // preprocess the beginning nodes that have no edges
    increment_node(read_node);
    for i ← 0 to spmbuffer_size do
        spmnbl(&outgoing_contrib[col_index[i]],&spm_outgoing_contrib(i));
    end

    for i ← spmbuffer_size to total_number_of_edges do
        pos ← i % spmbuffer_size;
        if pos = 0 then
            spmbarrier();
        end
        spmnbl(&outgoing_contrib[col_index[i]],&spm_outgoing_contrib(i));
        incoming_total += spm_outgoing_contrib(pos);
        read_edges++;

         // now done calculating for the current node
        if read_edges = row_index[read_node] then
            old_score ← score[read_node];
            score[read_node] ← base_score + (d * incoming_total);
            error ← error + |score[read_node] − old_score| ;
            incoming_total ← 0;

             // for the next node that does not have any edges
            increment_node(read_node);
            read_node++;
        end
    end
```

```
    // for the remaining unread edges
    last_read_edges ← total_number_of_edges % spmbuffer_size;
    spmbarrier();
    for i ← read_edges to total_number_of_edges do
        incoming_total ← incoming_total +
          spm_outgoing_contrib(last_read_edges % spmbuffersize);
        read_edges++;
        last_read_edges++;
        if read_edges = row_index[read_node] then
            old_score ← score[read_node];
            score[read_node] ← base_score + (d * incoming_total);
            error ← error + |score[read_node] − old_score| ;
            incoming_total ← 0;

            // for the next node that does not have any edges
            increment_node(read_node);
            read_node++;
        end
    end
    if error < epsilon then
        break;
    end
end
```

Algorithm 3: increment_node function

```
input : read_node
while row_index[read_node] = row_index[read_node + 1] do
    old_score ← score[read_node];
    score[read_node] ← base_score;
    error ← error + |score[read_node] − old_score| ;
    read_node++;
end
```

5.2 Single-Source Shortest Path (SSSP)

The SSSP algorithm will give a path with the minimum cost from a vertex to a specific destination vertex. The distance between the two vertices will be the minimum sum of edge weights along the path. A solution might exist where more than one path may have an equal minimum sum. In this case, the path that is discovered first is chosen to be the optimal path. The straightforward way of implementing SSSP is via Dijkstra's algorithm, but in GAP Benchmark, it is implemented via Δ-stepping algorithm [43] with some optimizations from Madduri et al. [44].

The Δ -stepping algorithm is an *approximate bucket implementation of Dijkstra's algorithm.* It keeps an array of buckets B, where $B[i]$ contains the collection of vertices $\{u \in V : u \text{ is queued and } d(u) \in [i\Delta,(i+1)\Delta)\}$. Δ is a positive integer that represents the "bucket width". All vertices in the current bucket are removed, added to the set S, and light edges ($\ell(e) \leq \Delta$, $e \in E$,where ℓ is the length function) next to these vertices are relaxed in each phase of the algorithm. New vertices may be added to the current bucket, which will be eliminated in the next phase. It is also possible that vertices that were previously removed from the current bucket will be reintroduced if their tentative distance improves. In the algorithm, heavy edges ($\ell(e) > \Delta$, $e \in E$) are not relaxed since they produce tentative values outside of the current bucket. All heavy edges out of the vertices in S are relaxed at the same time if the current bucket remains empty following relaxations. The algorithm continues until all the buckets are empty.

The edge relaxations in each phase can be done in parallel, and in our case, this phase can be moved to the SPM for faster processing, as long as individual tentative distance values are updated atomically. We modify the relaxedge stage of the algorithm as it is responsible for most memory accesses, which tends to cause a memory bottleneck.

The naïve and SPM aware relaxedge are described in Algorithms 5 and 6 respectively, whereas algorithm 4 presents the overall Δ- stepping Algorithm:

Algorithm 4: SSSP overall algorithm implementing Δ algorithm

```
input  : G(V, E), source, delta, kMaxBin, kDistInf, kBinSizeThreshold
output: the weight of the shortest path from the source
for u ∈ V do
    dist[u] ← kDistInf;
end
dist[source] ← 0;
for i to num_of_edges do
    frontier[i] ← 0;
end
shared_indexes[2] ← {0, kMaxBin};
frontier_tails[2] ← {1, 0};
frontier[0] ← source;
local_bins[ ][ ] ← Ø;
iter ← 0;
while (shared_indexes[iter&1] ! = kMaxBin) do
    &curr_bin_index ← shared_indexes[iter&1];
    &next_bin_index ← shared_indexes[(iter+1)&1];
    &curr_frontier_tail ← frontier_tails[iter&1];
    &next_frontier_tail ← frontier_tails[(iter+1)&1];
    for i ← 0 to curr_frontier_tail do
        u ← frontier[i];
        if dist[u] ≥ delta ∗ curr_bin_index then
            RelaxEdges(u, delta, dist, local_bins);
        end
    end
    x ← (!local_bins[curr_bin_index].empty()) &&
     (local_bins[curr_bin_index].size() < kBinSizeThreshold);
    while (curr_bin_index < local_bins.size() && x) do
        curr_bin_copy ← local_bins[curr_bin_index];
        local_bins[curr_bin_index].resize(0);
        for u ∈ curr_bin_copy do
            RelaxEdges(u, delta, dist, local_bins);
        end
    end
    for i ← curr_bin_index to local_bins.size() do
        if !local_bins[i].empty() then
            next_bin_index ← min(next_bin_index, i);
            break;
        end
    end
    curr_bin_index = kMaxBin;
    curr_frontier_tail = 0;
```

```
    if next_bin_index < local_bins.size() then
        copy_start ← fetch_and_add(next_frontier_tail,
          local_bins[next_bin_index].size());
          copy(local_bins[next_bin_index].begin(),
          local_bins[next_bin_index].end(), frontier.data() + copy_start);
          local_bins[next_bin_index].resize(0);
    end
    iter++;
end
```

Algorithm 5: SSSP RelaxEdges

```
input  : u, delta, dist, local_bins
output: resized local_bins with respect to the distance
for wn ∈ out_neigh(u) do
    old_dist ← dist[wn.v] ;                                  // v is node id
    new_dist ← dist[u] + wn.w ;                  // w is weight of node id
     while new_dist < old_dist do
        if (compare_swap(dist[wn.v], old_dist, new_dist) then
            dest_bin ← ⌊new_dist/delta⌋;
            if dest_bin >= local_bins.size() then
                local_bins.resize(dest_bin+1);
            end
            local_bins[dest_bin]← wn.v;
            break;
        end
        old_dist ← dist[wn.v];
    end
end
```

Algorithm 6: SSSP RelaxEdges with SPM Aware

```
input  : u, delta, dist, local_bins
output: resized local_bins with respect to the distance
spm_wn ← first SPM address;
k ← 0;
for wn ∈ out_neigh(u) do
    spmnbl(&wn,&spm_wn(k++));
    if k == spmbuffer_size then
        break;
    end
end
spmbarrier();
i ← 0;
for wn ∈ out_neigh(u + spmbuffer_size) do
    spmnbl(&wn,&spm_wn(k++));
    if k % spmbuffer_size == 0 then
        spmbarrier();
    end
    old_dist ← dist[spm_wn[i].v] ;                    // v is node id
    new_dist ← dist[u] + spm_wn[i].w ;     // w is weight of node id
    while new_dist < old_dist do
        if (compare_swap(dist[spm_wn[i].v], old_dist, new_dist) then
            dest_bin ← ⌊new_dist/delta⌋;
            if dest_bin >= local_bins.size() then
                local_bins.resize(dest_bin+1);
            end
            local_bins[dest_bin]← (spm_wn.v);
            break;
        end
        old_dist ← dist[spm_wn[i].v];
    end
    i++;
end
```

```
spmbarrier();
k=i;
for i ← k to out_degree(u) do
    old_dist ← dist[spm_wn[i].v] ;                    // v is node id
    new_dist ← dist[u] + spm_wn[i].w ;     // w is weight of node id

    while new_dist < old_dist do
        if (compare_swap(dist[spm_wn[i].v], old_dist, new_dist) then
            dest_bin ← ⌊new_dist/delta⌋;
            if dest_bin >= local_bins.size() then
                local_bins.resize(dest_bin+1);
            end
            local_bins[dest_bin]← (spm_wn.v);
            break;
        end
        old_dist ← dist[spm_wn[i].v];
    end
end
```

5.3 Connected Components(CC)

The connected component of an undirected graph is a subgraph in which each pair of nodes relates to each other via a path. The connected component algorithm thus finds these subgraphs. In the GAP benchmark, this algorithm is implemented via the Shiloach-Vishkin (SV) algorithm [45] which has significant speedup compared to the naïve method of implementation.

The SV algorithm stores the connectivity information in a forest of rooted trees, with each vertex v having a field that points to either itself or another vertex in the same connected component. All vertices in a tree belong to the same component, and at the end of the procedure, all vertices in a connected component belong to the same tree. Each tree has a designated root (a vertex having a self-loop) that serves as the representative vertex for the corresponding component.

As the Connected Components is made up of various functions, we target the

Link stage as it is responsible for the memory bottleneck in the algorithm as it causes multiple out-of-order memory requests. The native and SPM aware link stage are described in Algorithms 7 and 8, while Algorithm 9 presents the overall CC algorithm that utilizes the link stage.

Algorithm 7: CC Link stage Algorithm

```
input  : source u, destination v, comp
output: updated comp
p1 ← comp[u];
p2 ← comp[v];
while p1 != p2 do
    high ← p1 > p2 ? p1 : p2;
    low ← p1 + (p2 - high);
    p_high ← comp[high];
    if ((p_high == low) || (p_high == high &&
     compare_&_swap(comp[high], high, low))) then
        break;
    end
    p1 = comp[comp[high]];
    p2 = comp[low];
end
```

Algorithm 8: CC Link stage Algorithm with SPM Aware

```
input  : source u, destination v, comp
output: updated comp
SPM_p1 ← SPM_Address_1;
SPM_p2 ← SPM_Address_2;
spmnbl(&comp[u], &SPM_p1);
spmnbl(&comp[v], &SPM_p2);
spmbarrier();
while SPM_p1 != SPM_p2 do
    high ← SPM_p1 > SPM_p2 ? SPM_p1 : SPM_p2;
    low ← SPM_p1 + (SPM_p2 - high);
    p_high ← comp[high];
    if ((p_high == low) || (p_high == high &&
     compare_&_swap(comp[high], high, low))) then
        break;
    end
    spmnbl(&comp[comp[high]], &SPM_p1);
    spmnbl(&comp[low], &SPM_p2);
    spmbarrier();
end
```

Algorithm 9: CC overall algorithm utilizing the Link algorithm regularly

```
input  : G(V,E), neighbour_rounds
output: connected components
for n ∈ V do
    comp[n] ← n;
end
for r ← 0 to neighbour_rounds do
    for u ∈ V do
        for v ∈ out_neigh(u, r) do
            // Link at most one time if neighbor available at offset r
            Link(u, v, comp);
            break;
        end
    end
    Compress(G, comp);
end
 /* Sample 'comp' to find the most frequent element -- due to
prior compression, this value represents the largest
intermediate component */
c ← SampleFrequentElement(comp);
for u ∈ V do
    if comp[u] == c then
        continue;
    end
    for v ∈ out_neigh(u, neighbour_rounds) do
        Link(u, v, comp);
    end
    for v ∈ in_neigh(u) do
        Link(u, v, comp);
    end
end
Compress(G, comp);
```

5.4 Triangle Counting (TC)

As the name suggests, Triangle Counting is used to compute the total number of triangles available in a graph where a triangle is defined to be three vertices directly connected to each other. Because a triangle is invariant to permutation, the same three vertices should be counted as one triangle regardless of their order therefore, To count triangles, we sum the sizes of the overlap between a vertex's neighbor list and its neighbors' neighbor lists. Also, in triangle counting, the direction of the edges is ignored therefore requiring an undirected graph as input.

The algorithm is optimized in GAP benchmark as it is compute-intensive. Therefore, the authors relabel the graph using a heuristic to decide when to label the graph. Labeling the graph makes it less compute-intensive, leaving it only communication-intensive, which we solve using our system.

The naïve TC Algorithm is described in Algorithm 10 whereas the SPM aware algorithm is described in Algorithms 11.

Algorithm 10: TC Algorithm

```
input  : G(V, E)
output: total triangles
total ← 0;
for u ∈ V do
    for v ∈ out_neigh(u) do
        if v > u then
            break;
        end
        /* first_w is a pointer pointing to the first neighbour
        of node v */
        first_w ← out_neigh(v).first_neighbour;
        for w ∈ out_neigh(v) do
            if w > v then
                break;
            end
            while *first_w < w do
                first_w++;
            end
            if w == *first_w then
                total++;
            end
        end
    end
end
```

Algorithm 11: TC Algorithm with SPM aware

```
input  : G(V, E)
output: total triangles
  /* graph is stored in CSR format it is taken and utilised as
 it is */
total ← 0;
spmnodes ← first SPM address;
for u ← 0 to number_of_nodes do
    for v ← row_index[u] to row_index[u + 1] do
        if col_index[v] > u then
            break;
        end
        first_w ← &col_index[row_index[u]];
        k ← 0;
        for w ← row_index[col_index[v]] to
         row_index[col_index[v]] + spmbuffer_size do
            spmnbl(&col_index[w], spmnodes(k++%spmsize));
        end
        read_edge ← 0;
        spmbarrier();
        for w ← row_index[col_index[v]] + spmbuffer_size to
         row_index[col_index[v] + 1] do
            if (spmnodes[read_edge%spmsize] > col_index[v]) then
                break;
            end
            spmnbl(&col_index[w],spmnodes(k++%spmsize));
            while (*first_w < spmnodes[read_edge%spmsize]) do
                first_w++;
            end
            if (spmnodes[read_edge%spmsize] ==*first_w) then
                total++;
            end
            read_edge++;
            if k%spmbuffer_size == 0 then
                spmbarrier();
            end
        end
```

```
        spmbarrier();
        for w ← row_index[col_index[v]] + read_edge to
          row_index[col_index[v] + 1] do
            if (spmnodes[read_edge%spmsize] > col_index[v]) then
              break;
            end
            while (*first_w < spmnodes[read_edge%spmsize]) do
              first_w++;
            end
            if (spmnodes[read_edge%spmsize] == *first_w) then
              total++;
            end
            read_edge++;
        end
    end
end
```

Chapter 6

Experiments

6.1 Experimental tools

In this thesis, we use GEM5 [41] as the simulator of choice, x86 as the instruction set architecture, Scratch-pad Memory as additional hardware, and GAP Benchmark [46] as the benchmark for comparing the performance of our system.

We need to compare the results against an already set standard for a fair comparison to evaluate vertex-centric graph algorithms. One choice is the GAP benchmark which contains all such graph algorithms and graphs.

The GAP benchmark is an open-source benchmark designed to ease the evaluation of graph processing systems and provides a baseline for the developer community. The graph kernels given by the GAP benchmark are the most popular in the related research literature. Furthermore, this benchmark is designed in conjunction with not only work-load characterization but with computational demands as well. Also, the graphs provided with the benchmark are from both real-world data and synthetic generators from Kron and Urand.

Therefore, the GAP benchmark characteristics make it the most suitable and appropriate benchmark to compare our results with.

For our simulator, we choose GEM5. GEM5 is a cycle-accurate simulator with an event-driven simulation framework that has different levels of abstractions. This simulator combines two older simulators M5 and GEMS, which were later combined to become one powerful simulator. GEM5 offers two different modes of simulation: system emulation (SE) and full system (FS). The SE mode emulates the most basic operating system level execution, while FS mode performs a complete system simulation ranging from the OS to the device drivers and the thread scheduler. GEM5 also has multiple CPU models starting with a simple CPU to a very complex out-of-order CPU. In addition to all this, GEM5 provides support for simulation on various architectures, including x86, ARM, MIPS, and RISCV.

GEM5 has been chosen as the simulator for this research work because it is a cycle-accurate simulator, and we want to measure the exact performance gain from the custom graph processor. GEM5 also offers high flexibility and modularity. The memory system of GEM5, in particular, is of interest since it can help create a new memory system, i.e., an SPM, without having to start from the building blocks.

Other reasons for choosing GEM5 is that it supports the x86 architecture and its ability to simulate in System emulation (SE) mode. Simulating in SE mode saves simulation time by a significant factor as it only simulates the necessary instructions needed for an application to run. GEM5 also provides easy debugging after minor modifications and allows its builds to be attached to an IDE, which further simplifies the development of the custom graph processor. Furthermore, GEM5 has a very active mailing list where the authors and other contributors provide support in developing this simulator, hence providing quick technical support, unlike other simulators that lack this kind of active online support.

6.2 Experimental Setup

The graph applications are executed in GEM5 while using the custom graph processor and the default processor. As mentioned before, the custom graph

Table 6.1: Graphs used for the benchmarks

Graph	No. of vertices	No. of edges
G1	1,024	12,041
G2	2,048	25,525
G3	4,096	53,404
G4	8,192	110,869
G5	16,384	228,223
G6	32,678	467,807
G7	65,535	955,403

processor is built on top of the Minor CPU provided by GEM5. The custom graph processor can interact with the Scratch-pad Memory directly using a specialized port that has been introduced to allow such interaction. We then compare the results of the custom-made graph processor against the general-purpose Minor CPU. The parameters of the two CPUs are kept similar to that of a Nehalem core to maintain fairness. Any parameter that is different is taken into account, and a sensitivity analysis is taken into consideration.

The benchmarks that have been used to compare the system are taken from the GAP benchmark. PageRank, Single-Source Shortest Path, Connected Components, and Triangle Counting are used in testing the system.

The input graphs are synthetically generated from a Kronecker function that the GAP Benchmark provides. The number of vertices and edges of each generated graph are shown in Table 6.1. As the graphs are randomly generated, some graphs exhibit more random properties than others. Real graphs provided by the GAP benchmark are not taken into testing due to their inherently large size, which leads to the simulation run time taking multiple days to complete on the cycle-accurate simulator. Also, all the generated graphs are directed graphs, apart from the graphs used to test the Triangle Counting benchmark, which requires the input graphs to be undirected.

The graphs are stored in the CSR format to minimize memory utilization and also the CSR format is the default graph storage used by the GAP benchmark. The Scratch-pad Memory size is kept constant at 512 entries, except for the

sensitivity analysis case where the SPM size varies.

As our focus is on the performance, therefore, while comparing the results, we only compare the time taken by the actual benchmark to finish execution. File reading and other unnecessary pre-execution and post-execution processing are not taken into account in the performance calculation.

The GEM5 simulator calculates cycles in the number of ticks, which are then converted into seconds to make it easier to comprehend the results. Since custom instructions are being used in C++ applications, there tend to be some unnecessary instructions that would otherwise not be present if such a system had compiler support. The unnecessary instructions exist due to a lack of support from the GCC compiler to interpret the custom instructions. In the experiments, such artificial costs incurred by these instructions are removed in a precisely calculated manner.

The experiments performed on both of the systems, include only one level of cache. The instruction cache and data cache are separate as per default, and each with a size of 2Kb. The reason for choosing a small cache size is to keep the simulation runtime as minimum as possible. A large cache would mean using larger graphs to mimic enough cache misses, which is the objective of this thesis. All other parameters are kept with their default values as per Nehalem core specifications [47].

The main memory latency is set to that of a Nehalem core as described in [47], which is 194 cycles. We round it up to 200 cycles in the simulation environment as GEM5 sometimes throws errors when it is not a multiple of 10. The default memory latency in GEM5 is 4 cycles which is very little compared to real systems.

The results from using the custom graph processor are compared with a generic processor that does not have a Scratch-pad Memory and has no access to the custom memory instructions. As mentioned earlier, both the custom graph processor and the generic processor have been set to have the same properties, with the only difference being the SPM and the ability to execute custom instruction on

the custom graph processor.

6.3 Experimental Results

6.3.1 PageRank

The results are firstly obtained with the PageRank benchmark. The PageRank algorithm is highly random, thus providing a good baseline to compare with. We have two versions of the PageRank algorithm, which are both compared against the baseline provided by the GAP Benchmark.

The first version of the PageRank implementation employs the reuse of data in the Scratch-pad Memory, i.e., graph data that has been previously fetched from memory is not fetched again, as it is already available in the Scratch-pad Memory. This version reduces the number of main memory accesses as redundant graph data is fetched only once. The second version of the PageRank algorithm employs no-reuse of data that is stored in the Scratch-pad Memory. Figure 6.1 shows the results when there is data reuse in the SPM.

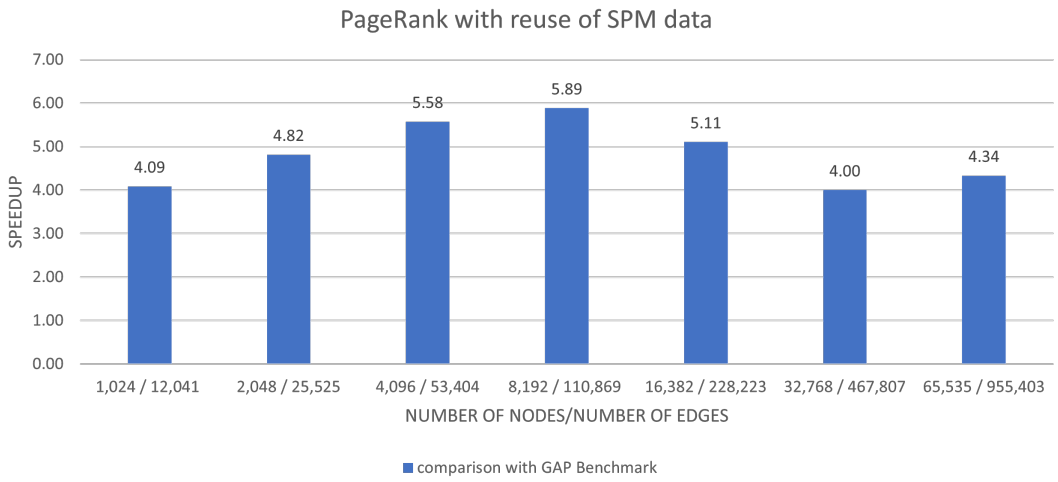

Figure 6.1: PageRank with reuse of SPM data

The drop in performance for the graph with 65,535 vertices is attributed to the fact that the graph is not as random as the previous ones. The performance is related to how random the graph is. If there is less random data access (i.e., data is being sequentially fetched from memory), the system tends to slow down as there is an additional overhead in the implementation which usually does not exist in a typical system.

The second version of the PageRank implementation does not employ the reuse of data in the Scratch-pad Memory. The advantage of this implementation is that we can use a Scratch-pad Memory of a size similar to that of the SPMBuffer, requiring a minimal SPM size. Figure 6.2 shows the speedup for the version without data reuse.

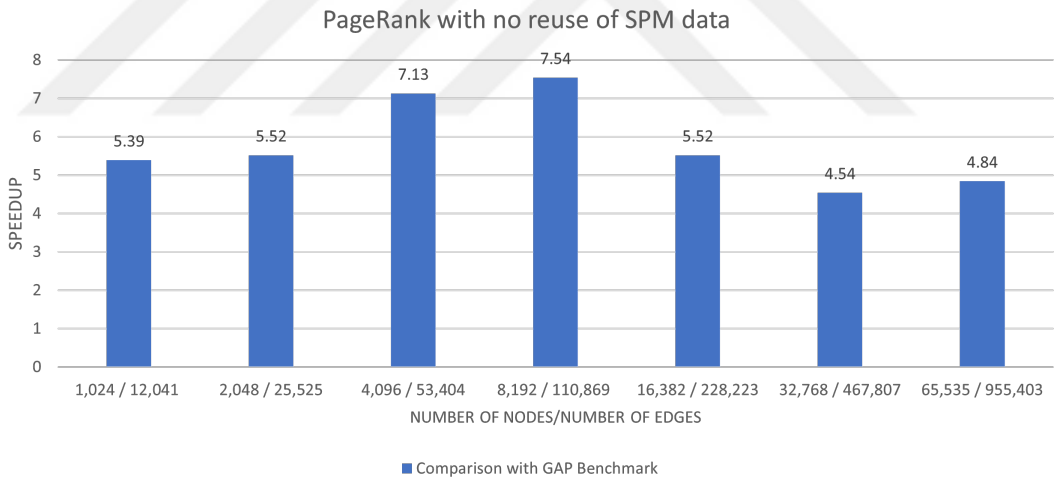

Figure 6.2: PageRank with no reuse of SPM data

When comparing the version with no SPM data reuse against SPM data reuse, the version with no data reuse outperforms the other. This is because the SPM data reuse version has additional overhead encountered while checking if the data is previously available in the SPM. Also, since the data being accessed has a high chance of not being accessed again until the next iteration, therefore this check is costly and slows down the system.

From the experiments carried out with the PageRank application, the system

with the version of data reuse shows an improvement of 5X and up to 7X in speedup when there is no reuse of data from the SPM.

The two presented versions of the PageRank algorithm experiment on how the data reuse in SPM can affect performance. It can be noted that without the reuse of data, we have a better speedup because there is less overhead in the software. Whereas the overhead in the reuse data version results from keeping unique data in the SPM, which tends to be costly as we have to keep checking which data to keep in the SPM.

6.3.2 Single-Source Shortest Path

The second benchmark that the system is tested on is the SSSP. For SSSP, we again compare the results against the baseline provided by the GAP Benchmark. The methodology in which SSSP has been implemented leads to having only one version, i.e., without data reuse in the SPM. To configure the SPM so that the data can be reused adds a large amount of unnecessary overhead, which causes the performance to drop considerably. Therefore, SSSP was implemented without data reuse in the experiments.

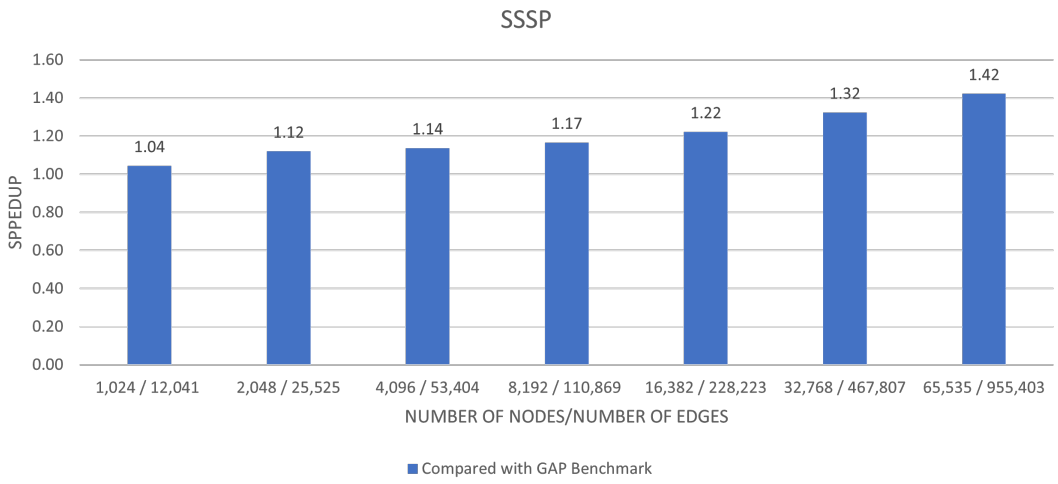

Figure 6.3: Speedup of various graphs while using the SSSP Benchmark

While comparing the results to GAP Benchmark, the SSSP benchmark does

not indicate a significant increase in performance with small graphs. However, as we increase the size of the graph, the performance increases because there are more random memory accesses to the main memory. Also, due to the overlapping instruction capabilities of the custom graph processor, an increase in performance is evident.

From the experiments, as shown in Figure 6.3 an increase in performance up to 1.4X for large graphs is observed. It can be assumed that for graphs larger than those used in the experiments, we can have more speedup as they will have more random memory accesses.

6.3.3 Triangle Counting

For the third benchmark, we use the triangle counting benchmark. For this benchmark, the results vary depending on the shape of the graph. As this algorithm tries to find triangles in a graph, sometimes a triangle can be found in the first instance, while sometimes it cannot be found at any instance. As the system loads data from the main memory in batches, sometimes unnecessary data is loaded, which leads to a loss in performance, which is not the case in a regular system.

Furthermore, in triangle counting, we do not reuse the data in the SPM, as reuse of data is expensive and affects the performance negatively. Triangle counting requires the graphs to be undirected as well. The graphs used in Triangle counting are different from the other graphs, as all other benchmarks use directed graphs.

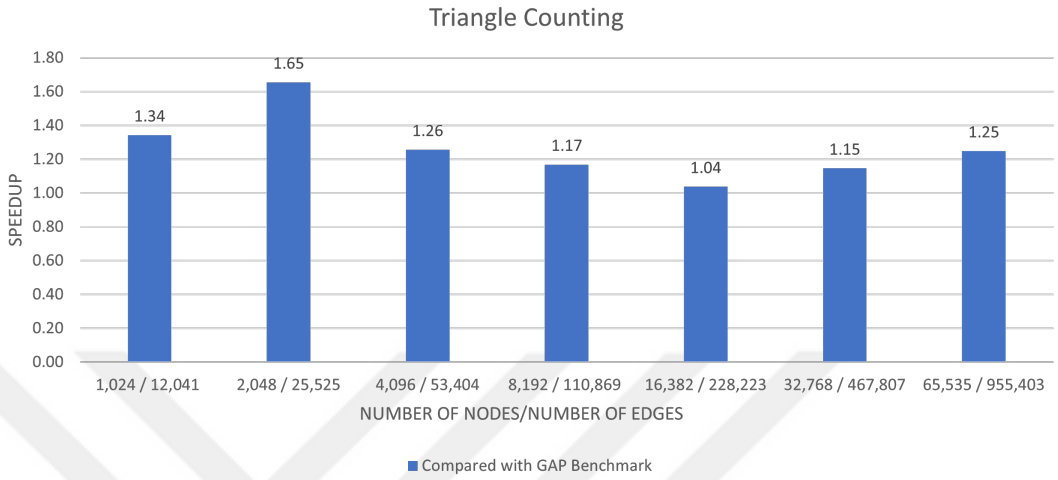

Figure 6.4: Speedup of various graphs while using the Triangle Counting Benchmark

From the experiments that have been conducted, a significant speedup is not observed because of how the Triangle Counting benchmark executes. An average speedup of 1.2X is observed as shown in Figure 6.4.

6.3.4 Connected Components

The last benchmark with which we test the custom graph processor is the connected components. The connected components benchmark uses directed graphs. From the experiments that have been conducted, it is observed that the Connected Component benchmark increases in performance as the graph size increases, as shown in Figure 6.5. This behavior is similar to the SSSP Benchmark, i.e., the larger the graph, the more random memory accesses, which leads to more speedup.

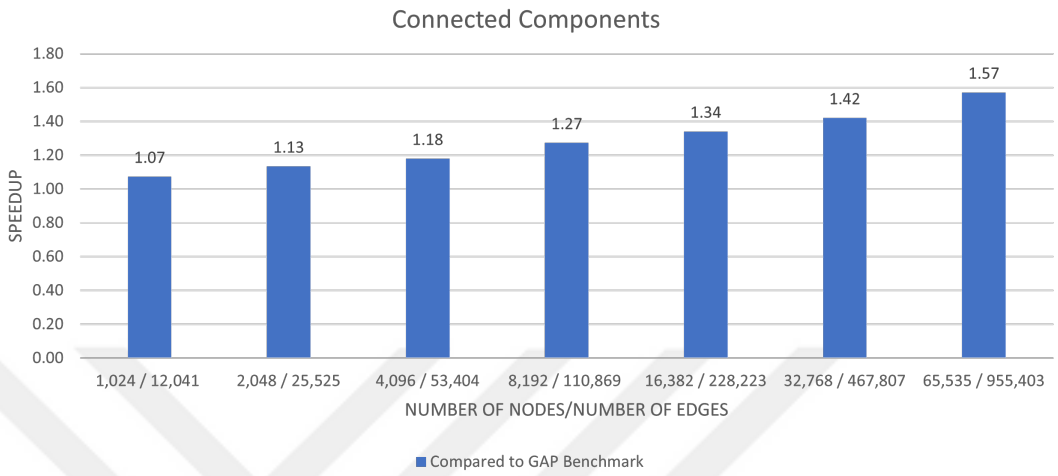

Figure 6.5: Speedup of various graphs while using the Connected Components Benchmark

6.4 Sensitivity Analysis

In this section, we analyze how the SPM size, the SPM Buffer size, the main memory latency, and how the cache size affects the performance of the proposed system. The PageRank algorithm is used to carry out all the sensitivity analyses.

6.4.1 Main Memory latency

This section analyzes how the main memory latency affects the performance of the custom graph processor. The main memory latency is a significant factor in the performance, as it determines how quickly the data is available to the CPU. A large latency will cause a drop in performance in a typical system, while the latency of a few cycles would have high performance.

In the custom graph processor, it was noticed that the higher the latency, the higher the performance of the system when compared to a typical system with the same latency. This performance increase is possible because the long latency

memory instructions can process in the background, allowing other instructions to continue their execution.

The experiments are again performed with different graph sizes ranging from small to large. The main memory latency is also varied from 4 cycles to 200 cycles, and the results with these different configurations are shown in Figure 6.6.

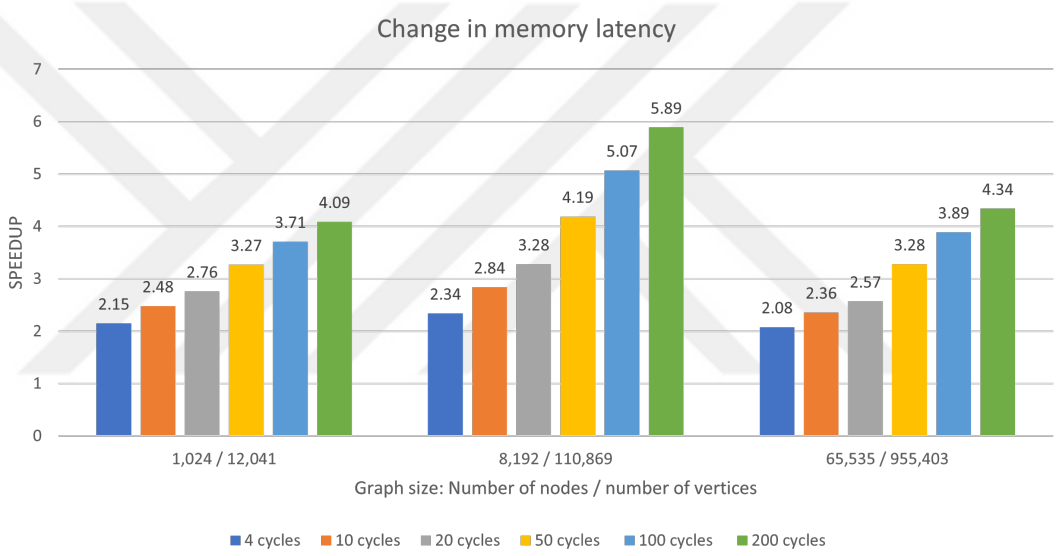

Figure 6.6: Speedup of various graphs when comparing different main memory latencies against graph G3, G5, and G6

From the experimental results, the custom graph processor performs very well when the memory latency is high. A speedup of almost 6X is observed for a graph size of 8192 nodes and a speedup of up to 4X for larger graphs. An increase in speedup is also observed in a smaller graph of 1024 nodes.

The speedup is also directly related to how the memory is accessed, the more random accesses the graph has to the memory, the more the speedup. This is evident from the results as the larger graphs that are used for testing have less random accesses compared to some of the smaller ones.

This fact is verified by using Intel Vtune[48] application where we run the

application on a native PC and check for memory bottlenecks. It is observed that some of the larger graphs have fewer percentage of memory bottlenecks compared to the others. For example, it is noted that the graph with 65535 nodes has 10.5% memory bound instructions that are causing a bottleneck while the graph with 8192 nodes has 13.6% memory bound instructions that are causing a memory bottleneck. Therefore, confirming that the speedup is also directly related to the number of random memory accesses a graph application makes.

6.4.2 Cache size

The cache size of a system also affects the performance of the system. Using the SPM to store graph data frees up the cache, leaving several empty cache lines, which can be utilized for other purposes. As the graph data is primarily random, and data is fetched from the main memory in sizes of the cache line, several cache lines can be filled with graph data due to the principle of spatial locality. The additional data brought up by spatial locality may not be used as we have random accesses, thus causing a waste of cache resources. By storing the data in the SPM, we have several cache lines available for other processes, which can take the benefits of cache locality.

The results are compared to the default system with a similar cache size. For example, for the cache size of 1Kb, the custom graph processor and the default processor will both have 1Kb of cache size.

Although there is an increase in speedup as the size of the caches is increased, but it is very minimal, as shown in Figure 6.7. Therefore, it can be argued that the cache size does not have much impact on the performance of the custom graph processor system.

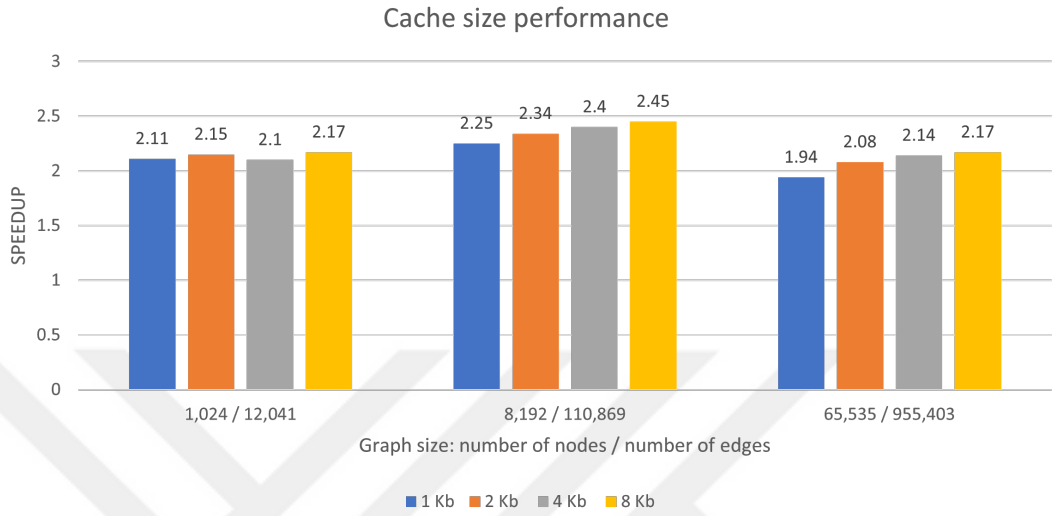

Figure 6.7: Speedup of various graphs when comparing different cache sizes against graph G3, G5, and G6.

6.4.3 SPM size

The size of the SPM is another factor that affects the performance of the custom graph processor for the cases where we employ data reuse policy. The less the size of the SPM, the lesser the resources required. However, this results in a performance trade-off, as we cannot store much data in a smaller SPM. In the experiments, the size of the SPM is varied from 8Kb to 256Kb, and the system is tested against graphs G3, G5, and G6. The results are measured in ticks which is the default measuring unit in GEM5, and the results are shown in Figure 6.8 for graph G3, Figure 6.9 for graph G5, and Figure 6.10 for graph G6.

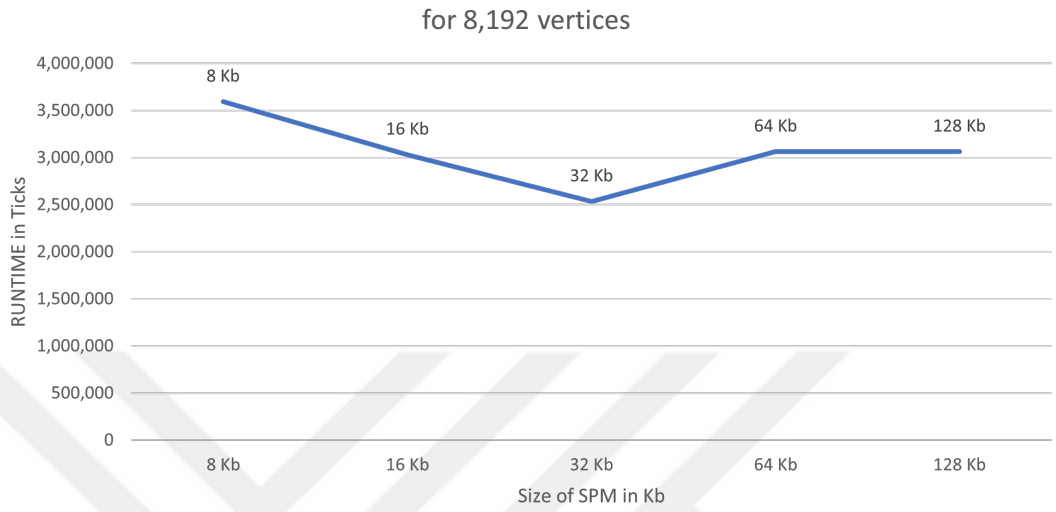

Figure 6.8: Runtime for the graph with 8,192 vertices for different SPM sizes

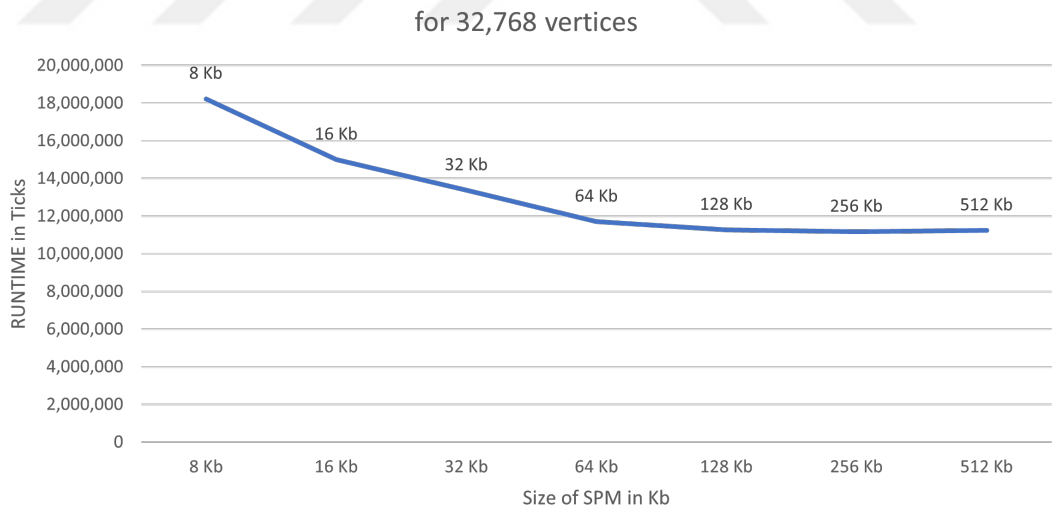

Figure 6.9: Runtime for the graph with 32,768 vertices for different SPM sizes

An increase in the SPM size leads to an increase in performance. However, there is not much increase in performance for smaller graphs once the SPM size is greater than 64Kb. For larger graphs, a size greater than 128Kb does not yield any additional significant performance increase. In the experiment, we noticed that an SPM size equivalent to three times the size of the SPMBuffer is more than

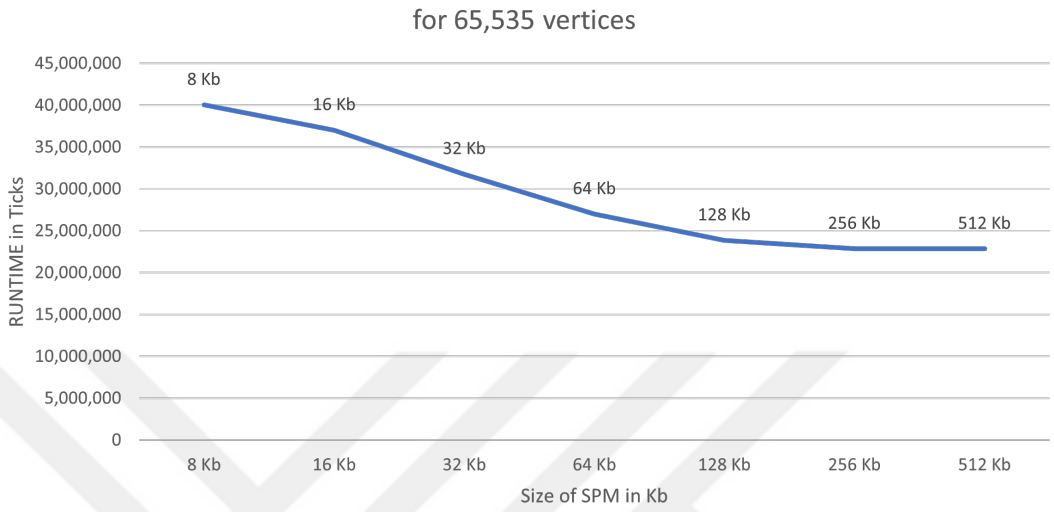

Figure 6.10: Runtime for the graph with 65,535 vertices for different SPM sizes

enough to produce significant speedup in the system because we hardly reuse any data in the SPM due to random memory access patterns of the applications.

6.4.4 SPMBuffer Size

The SPMBuffer is a small buffer that holds the data request information while the data is being fetched from the main memory. Data stays inside this buffer until the execution of the SPMBarrier instruction. It is then that the SPMBuffer is emptied, and the data is stored inside the SPM.

Three graphs of 8,192, 32,768, and 65,535 vertices are considered, and the PageRank algorithm is run on them to investigate how a different size of the SPMBuffer affects the performance.

We investigate various sizes of the SPMBuffer, ranging from $2^3, 2^4, ..., 2^8$. Again, the results are measured in ticks which is the default measuring unit in GEM5.

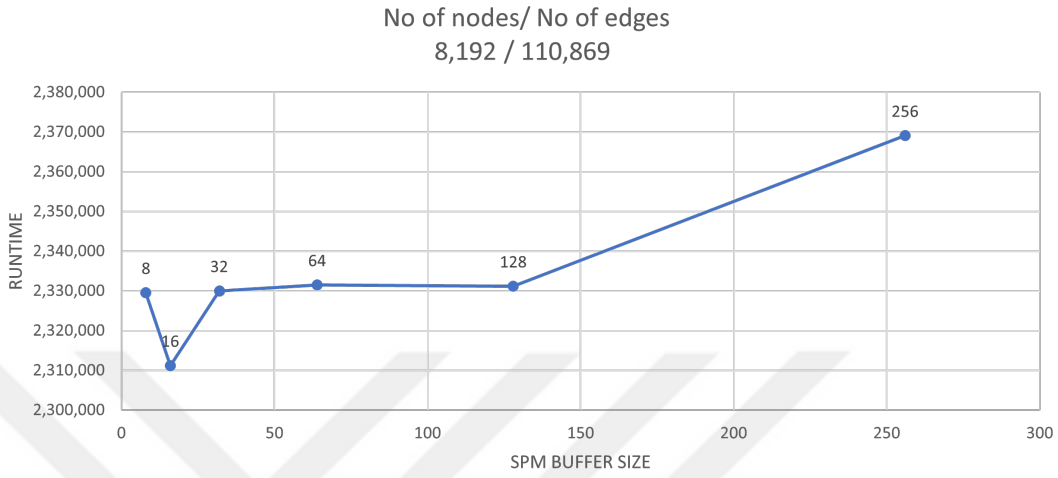

Figure 6.11: Runtime for the graph with 8,192 vertices for different SPMBuffer sizes

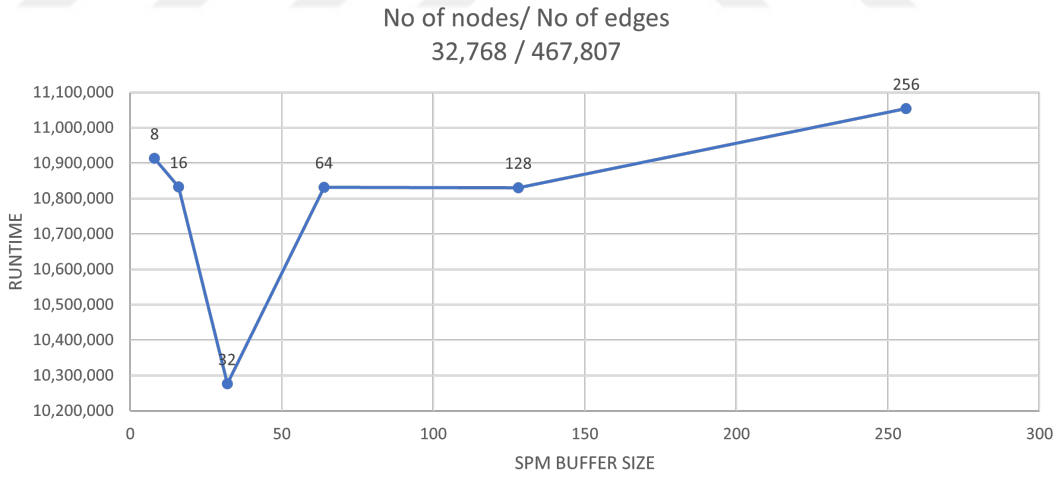

Figure 6.12: Runtime for the graph with 32,768 vertices for different SPMBuffer sizes

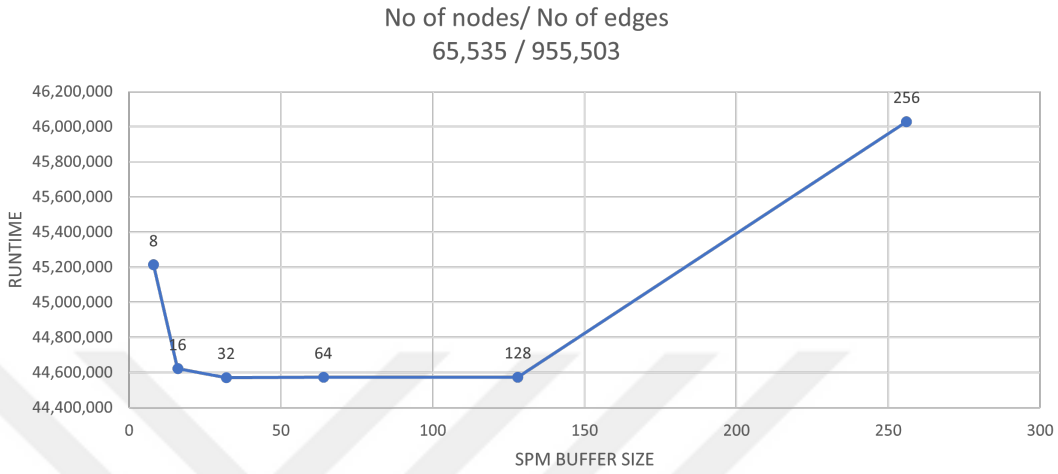

Figure 6.13: Runtime for the graph with 65,535 vertices for different SPMBuffer sizes

From the experimental results that have been collected and as shown in Figure 6.11, 6.12, and 6.13, the optimum size for the SPMBuffer is 32. Also, for a larger graph, smaller SPMBuffer size is seen to perform equally well. When the size of the SPMBuffer is quite large, we see a reduction in performance. The reduction in performance is because of unnecessary CPU stalls which are caused by the SPMBarrier instruction. As the SPMBarrier instruction has to wait for all instructions of a batch to return before executing the next batch, while in the meantime, the flow of control of the algorithm has returned to the point where the graph data is needed to continue execution. A CPU stall is therefore seen due to the SPMBarrier not releasing data to be used. Although the system's performance with a large SPMBuffer does out-perform the native system, it is still slower than that of a system with a smaller SPMBuffer.

Chapter 7

Discussion

This chapter gives remarks on the experimental results that have been collected. It also mentions the system's shortcomings and describes the possible future work that can be done in this research.

In the experimental results which have been obtained by testing on various graph applications from the GAP Benchmark, it can be seen that all of them show a positive speedup. PageRank has the highest increase of speedup of up to 7x, while all the other applications have an average speedup of 1.5x. It has also been deduced that the performance is directly related to the number of random accesses made by a graph in a certain application. This conclusion can be easily deduced because a large graph (which has fewer random memory accesses) does not show much improvement as compared to a small graph (which has numerous random memory accesses). Therefore, This system can be used by any graph application that will have random memory accesses.

In the sensitivity analysis, various noteworthy features were found. It was noticed that with an increase in memory latency, the system performed better due to the introduced overlapping of memory instructions. It was also found that compared to various cache sizes, this system performed much better as the cache size was increased, which is a trivial contradiction with the initial prediction in

the research. Also, it was noticed that the SPM does not need to be large. An SPM equivalent to three times the size of the SPMBuffer is more than enough to produce significant speedup. Although it would be recommended to keep it four times the size of the SPMBuffer in case a user wants to store temporary variables that are being used repeatedly. For the SPMBuffer, it is observed that a size greater than 128 elements is not required, thus not requiring many resources.

One can deduce that this system is more than efficient for graph applications, especially those with poor locality that tend to stall the processors due to the memory wall. The presented graph processor does not require much additional hardware apart from an SPM. Although being as efficient as possible, this system has a few drawbacks, the major one being that there is no compiler support for the custom instructions therefore, the user has to use inline ASM assembly to access the custom instructions. The use of inline ASM assembly calls for additional instructions to be called, which introduces unnecessary overhead. In the future work, it is intended to introduce compiler support for the custom instructions. Having compiler support would make it user-friendly and help avoid the additional overhead, which would help increase the speedup.

Our system has been tested on the GEM5 simulator, a cycle-accurate simulator, which is not an actual hardware. In the future, we tend to build the same system on an actual hardware and measure the potential speedup, which we believe would be slightly better than the simulated ones due to additional latencies that occur on actual hardware and are unaccounted for in the GEM5 simulator. We also intend to develop a similar system on an FPGA and compare its performance against our system and a general-purpose system.

Lastly, the last limitation of this system, which is also unavoidable, is that the graph applications have been modified to make use of the SPM. A native implementation of the graph application cannot benefit from the SPM and the custom instructions. The user has to modify the graph application to utilize the SPM. As these custom instructions work in batches to provide overlapping of the long latency memory instructions, writing code in this way might be challenging. As one of our future works, we intend to introduce a library that can ease the

usage of the SPM, making it more portable and easy to use.

Chapter 8

Conclusion

This thesis presents a novel idea on processors for graph applications with minimal additional resources. In the proposed system, an SPM is added, and to utilize the SPM properly, custom instructions are introduced as well. Apart from helping to utilize the SPM most effectively, the custom instructions are also responsible for providing instruction overlapping, especially for long latency instructions. The goal of this research work is to increase the performance of graph applications by solving the memory wall problem. With respect to the experimental results collected, we were able to gain significant speedup while using the new system.

The graph processor in the proposed model utilizes the x86 instruction set with a slightly modified pipeline for executing the SPM non-blocking instructions. The results of this approach have been simulated on the GEM5 simulator using the graph applications from the GAP Benchmark. The graph applications that have been used are PageRank, Single-Source Shortest Path, Connected Components, and Triangle Counting. We choose these applications because they randomly access the memory, exposing the memory bottleneck faced by graph applications. The graph applications have also been modified to utilize the SPM efficiently. When compared to a native system with a general-purpose processor, the results of this work show a significant speedup of up to 7x for PageRank application and 1.5x times for the other graph applications. We also carried out sensitivity

analysis, which showed that memory latency plays a significant role in the speedup compared to the other factors affecting the performance.

As part of the future work, we intend to implement the system on actual hardware and provide compiler support to achieve the best performance for graph applications. This graph processor design is a helpful tool for graph applications by providing significant speedup and reducing computational time for large graphs.

Bibliography

[1] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos, "Using graph theory to analyze biological networks," *BioData Mining*, vol. 4, no. 1, pp. 1–27, 2011.

[2] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 78–88, IEEE, 2011.

[3] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics," *Communications of the ACM*, vol. 59, no. 5, pp. 78–87, 2016.

[4] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *arXiv preprint arXiv:1204.6078*, 2012.

[6] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 166–177, 2016.

[7] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '15, pp. 56–65, IEEE, 2015.

[8] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[9] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, *et al.*, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, SC '99, pp. 57–57, IEEE, 1999.

[10] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 192–201, IEEE, 1999.

[11] P. M. Kogge, "EXECUBE-a new architecture for scaleable MPPs," in *Proceedings of the International Conference on Parallel Processing*, vol. 1, pp. 77–84, IEEE, 1994.

[12] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: A computation model for intelligent memory," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pp. 192–203, IEEE, 1998.

[13] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, HPCA '17, pp. 457–468, IEEE, 2017.

[14] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2018.

[15] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, HPCA '15, pp. 105–117, 2015.

[16] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, (OSDI '16), pp. 301–316, 2016.

[17] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '16, pp. 1–13, IEEE, 2016.

[18] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp. 781–792, IEEE, 2014.

[19] Y. Xing, Y. Feng, S. Yu, Z. Chen, F. Liu, and N. Xiao, "HPGraph: A high parallel graph processing system based on flash array," in *Proceedings of the IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 497–505, IEEE, 2016.

[20] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of the International Conference on Parallel Architecture and Compilation*, PACT '15, pp. 39–50, IEEE, 2015.

[21] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[22] S. Glinski, T. Lalumia, D. Cassiday, T. Koh, C. Gerveshi, G. Wilson, and J. Kumar, "A processor for graph search algorithms," in *Proceedings of the IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, vol. 30, pp. 162–163, IEEE, 1987.

[23] M. Scheevel, "NORMA: a graph reduction processor," in *Proceedings of the ACM Conference on LISP and Functional Programming*, pp. 212–219, 1986.

[24] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 217–226, 2017.

[25] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "GraphGen: An FPGA framework for vertex-centric graph computation," in *Proceedings of the IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28, IEEE, 2014.

[26] S. Noel, E. Harley, K. H. Tam, M. Limiero, and M. Share, "CyGraph: graph-based analytics and visualization for cybersecurity," in *Handbook of Statistics*, vol. 35, pp. 117–167, Elsevier, 2016.

[27] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pp. 135–146, 2013.

[28] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research*, vol. 13 of *CIDR '13*, pp. 3–6, 2013.

[29] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pp. 31–46, 2012.

[30] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, IEEE, 2010.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146, 2010.

[32] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 505–516, 2013.

[33] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pp. 599–613, 2014.

[34] A. Chan, F. Dehne, and R. Taylor, "CGMGRAPH/CGMLIB: Implementing and testing CGM graph algorithms on PC clusters and shared memory machines," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 81–97, 2005.

[35] N. Cheung, S. Parameswaran, and J. Henkel, "Inside: Instruction selection/identification & design exploration for extensible processors," in *Proceedings of the International Conference on Computer Aided Design*, ICCAD '03, pp. 291–297, IEEE, 2003.

[36] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Proceedings of the ninth international symposium on Hardware/software codesign*, pp. 61–66, 2001.

[37] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pp. 129–140, IEEE, 2003.

[38] S. Buchwald and A. Zwinkau, "Instruction selection by graph transformation," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, pp. 31–40, 2010.

[39] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '04, pp. 69–78, 2004.

[40] N. Kapre, "Custom FPGA-based soft-processors for sparse graph acceleration," in *Proceedings of the IEEE 26th International Conference on Application-specific Systems, Architectures and Processors*, ASAP '15, pp. 9–16, IEEE, 2015.

[41] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[42] "AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions." `https://www.amd.com/system/files/TechDocs/24594.pdf`, 2021. Accessed: 2021-09-08.

[43] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[44] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances," in *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments*, ALENEX '07, pp. 23–35, SIAM, 2007.

[45] Y. Shiloach and U. Vishkin, "An O (log n) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, pp. 57–67, 1982.

[46] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[47] R. S. Verdejo, K. Asifuzzaman, M. Radulovic, P. Radojković, E. Ayguadé, and B. Jacob, "Main memory latency simulation: the missing link," in *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, pp. 107–116, 2018.

[48] "Fix Performance Bottlenecks with Intel® VTune™ Profiler." `https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html`. Accessed: 2021-09-08.