HARDWARE/SOFTWARE CO-DESIGN OF DOMAIN-SPECIFIC RISC-V PROCESSOR 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
Mehmetali Semi Yenimol
May 2022

HARDWARE/SOFTWARE CO-DESIGN OF DOMAIN-SPECIFIC RISC-V PROCESSOR FOR GRAPH APPLICATIONS

By Mehmetali Semi Yenimol

May 2022

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.

Özcan Öztürk(Advisor)

Süleyman Tosun

Uğur Güdükbay

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

HARDWARE/SOFTWARE CO-DESIGN OF DOMAIN-SPECIFIC RISC-V PROCESSOR FOR GRAPH APPLICATIONS

Mehmetali Semi Yenimol
M.S. in Computer Engineering
Advisor: Özcan Öztürk
May 2022

Graph applications are employed in many fields but show poor performance on general-purpose computing systems due to heavy, irregular, and data-driven memory access patterns. The diverse topology of real-life graphs also affects the performance. Even though many hardware accelerators have been proposed to mitigate performance issues and provide energy efficiency, programmability and flexibility have not been sufficiently addressed. This thesis presents a domain-specific processor design based on extending the RISC-V Instruction Set Architecture (ISA). The proposed approach uses new instructions supported by the compiler and software library. Micro-architectural design for executing the new instructions is based on a scratchpad-memory (SPM), prefetcher, and a non-blocking cache system. The custom processor is implemented using System Verilog HDL and simulated with Xilinx's Vivado Design Suite. LLVM Compiler Framework is used for compiler support and optimization. The software library for utilizing the custom instructions uses Gather-Apply-Scatter (GAS) paradigm. The system is evaluated on well-known graph benchmarks, while sensitivity analysis is done on various parameters for achieving the best performance with minimal cost. Performance is measured on both native benchmarks and the software library. In addition, compiler support is evaluated for its effect on performance. Cost-efficient performance evaluations show average speedups between 10% and 49% for different benchmarks, while the single-core architecture can achieve up to 73%.

Keywords: Domain-Specific Processor, Graph Applications, RISC-V, Hardware/-Software Co-Design.

ÖZET

ÇİZGE UYGULAMALARI İÇİN ALANA ÖZGÜ RISC-V İŞLEMCİSİNİN DONANIM/YAZILIM ORTAK TASARIMI

Mehmetali Semi Yenimol
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Mayıs 2022

Çizge uygulamaları birçok alanda kullanılmaktadır ancak ağır, düzensiz ve veriye dayalı bellek erişim kalıpları nedeniyle genel amaçlı bilgi işlem sistemlerinde düşük performans göstermektedir. Gerçek hayat çizgelerinin farklı topolojik yapıları da performansı etkiler. Performans sorunlarını azaltmak ve enerji verimliliği sağlamak için birçok donanım hızlandırıcı önerilmiş olsa da, programlanabilirlik ve esneklik yeterince ele alınmamıştır. Bu tez, RISC-V Komut Kümesi Mimarisi'nin (KKM) genişletilmesine dayalı alana özgü bir işlemci tasarımı sunar. Önerilen yaklaşımda, derleyici ve bir yazılım kütüphanesi tarafından desteklenen yeni komutlar kullanılır. Yeni komutların yürütülmesi için mikro mimari tasarım, bir hızlı işlem belleği (HİB), önyükleyici ve tıkanmasız bir önbellek sistemine dayanmaktadır. Özel işlemci, System Verilog HDL kullanılarak gerçekleştirildi ve Xilinx'in Vivado Design Suite kullanılarak simüle edildi. LLVM Derleyici Çatısı, derleyici desteği ve optimizasyonu için kullanıldı. Öte yandan, özel komutları kullanmak için yazılım kütüphanesi, Topla-Uygula-Dağıt (TUD) paradigmasını kullanır. Sistem, iyi bilinen grafik denektaşlarında değerlendirilirken, minimum maliyetle en iyi performansı elde etmek için çeşitli parametreler üzerinde duyarlılık analizi yapıldı. Performans, hem yerli denektaşlarında hem de yazılım kütüphanesinde ölçüldü. Bunun yanında, derleyici desteği performans üzerindeki etkisi açısından değerlendirildi. Düşük maliyetli performans ölçümleri farklı denektaşları için ortalama %10 ile %49 arasında hızlanma gösterirken, tek çekirdekli mimari %73'e kadar ulaşabilir.

Anahtar sözcükler: Alana Özgü İşlemci, Çizge Uygulamaları, RISC-V, Donanım/Yazılım Ortak Tasarımı.

Acknowledgement

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

I would like to express my gratitude to my advisor, Prof. Özcan Öztürk, who gave me opportunity to work on this project and guided me patiently throughout my studies.

I would like to thank jury members, Prof. Süleyman Tosun and Prof. Uğur Güdükbay, for taking the time to read this thesis and providing valuable feedback.

Last but not least, I would like to thank my father Sabahattin, my mother Şengül, my sister Elif Bahar, my family and my friends for their great support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Applications centered around graph data structure are employed in many fields. These fields include but are not limited to data science, computational science, databases, social networks, genomics, healthcare, traffic control, telecommunication, security, and supply chain optimization. Execution time and power usage of graph applications increase when data size and application complexity increase.

The main challenges in improving the performance and power usage of graph applications are [1, 2, 3]:

- graph applications have high data access to computation ratio,
- the computations are mostly data-driven and unstructured,
- memory access patterns are irregular, which causes poor locality on modern cache systems, and
- the topology of real-life graphs is diverse.

Gui et al. show that hardware acceleration for the graph domain is needed rather than general-purpose processors (i.e., software solutions on CPUs and GPUs) since they are not good at load balancing, memory divergence, and superfluous

memory access [2]. Furthermore, graph applications have high energy consumption. This work evaluates 37 such hardware solutions regarding design techniques, hardware platform, performance, and energy efficiency. They point out programmability as a significant challenge in existing accelerators, in addition to increasing graph sizes, dynamic graph processing, graphs with more complex vertex/edge attributes, and practical issues in adapting the new technologies.

Application-Specific Instruction Set Processor (ASIP) methodologies have emerged as an alternative in developing specialized hardware to mitigate design and manufacturing costs, and to provide flexible design by programmability [4, 5]. Due to their programmability advantage, a custom architecture based on RISC-V instruction set architecture (ISA) was proposed for the iterative graph applications domain in a previous study [6]. The study shows that blocking memory accesses are reduced by up to 70%, yet actual speedup is not shown and programmability is only provided through inline assembly calls for custom instructions. We try to tackle these issues in this work by proposing software support and a novel micro-architectural design based on the previous study. Our main motivation follows the ASIP methodology to handle programmability issues on high-performance graph analytics architectures, especially considering the increased complexity in graph algorithms. Our main contributions are:

- proposing a novel micro-architectural design that uses extensions to the instruction set,
- providing software support through compiler extensions, compiler optimizations and software library, and
- evaluation of architectural parameters for achieving the best performance with minimal cost.

We evaluate our system with many well-known graph benchmarks. We observe average speedups between 10% and 49% for different benchmarks, and the single-core architecture is capable of having speedups up to 73%.

The rest of the thesis is organized as follows: In Chapter 2, we give some background and describe the general behavior of graph applications, then show the main challenges that are needed to be addressed. We present related work in ASIP designs and architectures for high-performance graph analytics in Chapter 3. Chapter 4 elaborates on the previous design, modifications to it, and the custom instruction set architecture. In Chapter 5, we explain the micro-architectural design of our system. Software support is explained in Chapter 6. Evaluation of our architecture is given in Chapter 7, with a discussion in Chapter 8. The thesis is concluded in Chapter 9.

Chapter 2

Graph Applications

A graph is a data structure to show relations between objects. The objects are often called *vertices*, and relations between the vertices are referred to as *edges*. If the set of vertices is V and the set of edges is E, then we define a graph G as $G = (V, E)$. Algorithms running on a graph need to use the proper representation of the graph structure. Vertices of a graph are usually indexed and identified by an integer i within the $[0, |V|)$ interval. The simplest way to represent a graph is using an edge-list format that holds a list of vertex pairs (not necessarily ordered), where each pair represents an edge $e \in E$. The edge-list format is useful for edge-centric computation models that iteratively process the stream of edges. Even though it provides a good cache locality for edge accesses, vertex accesses become random and limits scheduling potential for parallel systems [2].

Another common way to represent graphs is using an adjacency list, where an array of size $|V|$ is used for the graph. Each element of this array corresponds to a vertex $v \in V$ and holds a pointer to an adjacency list, where each adjacency list of v holds all the vertices u such that $(v, u) \in E$. An example graph is shown in Figure 2.1 with its edge-list and adjacency list representations given in Figure 2.2.

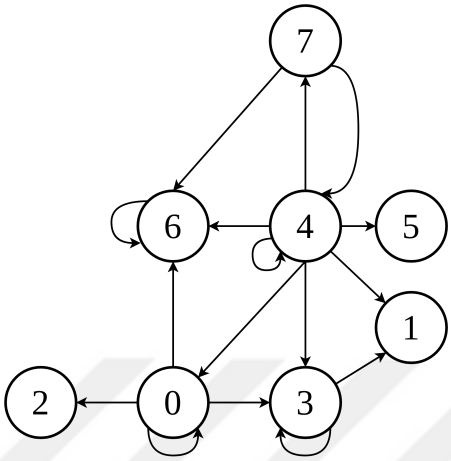
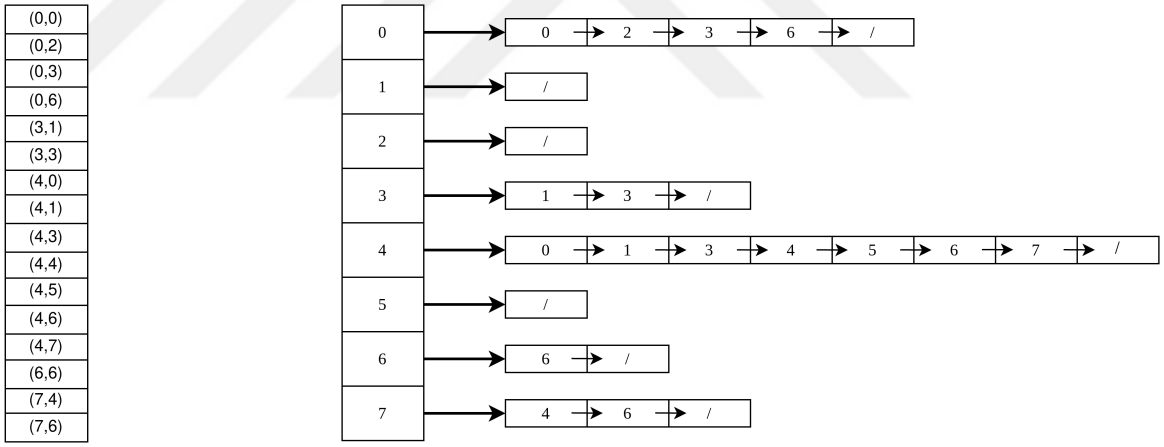


Figure 2.1: Sample graph with 8 vertices and 16 edges.



(a) Edge-list representation.

(b) Adjacency-list representation.

Figure 2.2: Representations of the sample graph given in Figure 2.1.

In practice, adjacency-list representation is often used in compressed structures. The most used compact representation is Compressed Sparse Row (CSR), introduced in [7] as a sparse matrix representation which is suitable for graphs due to matrix-graph duality. Accordingly, we can represent a graph as an adjacency matrix A of size $|V| \times |V|$. An element at row i and column j of matrix A, $A_{i,j}$, gives a non-zero value if edge $(i, j) \in E$ and 0 otherwise. In CSR representation, there are two arrays: *rows* for storing rows (vertices) and *columns* for

storing non-zero column entries (edges) of a sparse matrix. An index i in *rows* array point to the location of its non-zero column entries in adjacency matrix A until the next pointer in the *rows* array. As an example, the CSR representation of the graph in Figure 2.1 is given in Figure 2.3.

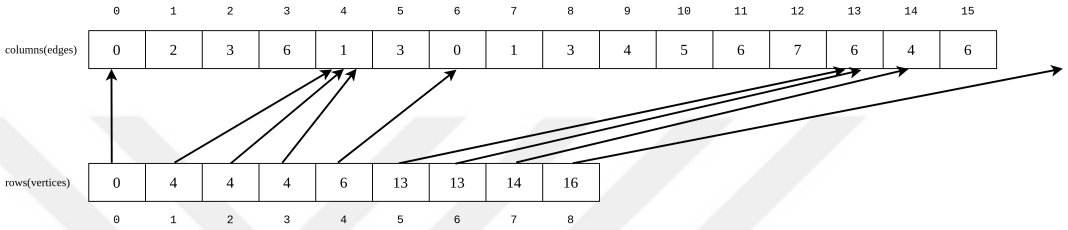


Figure 2.3: CSR Representation of the sample graph given in Figure 2.1

Let $off1 = rows[i]$ and $off2 = rows[i+1]$ for index i that represent vertex i. The values in the range of $columns[off1]$ and $columns[off2]$ are the out-neighbors for the vertex i. For example, vertex 0 has $off1 = 0$ and $off2 = 4$ such that elements from $columns[0]$ to $columns[4]$ gives the out-neighbors of vertex 0, which are vertices 0, 2, 3, and 6. Note that if $off1 = off2$, as in the case for vertex 1, it means that there is no outgoing edge from vertex 1. Also, note that, although there are only eight vertices (indexed from 0 to 7), an extra index in the *rows* array is used to point to the one plus last index of the *columns* array. The symmetric of CSR is called Compressed Sparse Column (CSC), which is used when concerned with incoming edges of vertices. For practical reasons, we choose CSR to represent graphs in this work. A third array (*values*) similar to *columns* is mainly used for non-zero values of matrix A. The non-zero values are often referred to as weights or edge attributes in the graph context. Moreover, a fourth array (*data*) similar to *rows* is also often used for vertex attributes in the graph.

Even though CSR representation itself is cache-friendly, random accesses that result in poor cache locality still occur due to graph applications' irregular and data-driven nature. The algorithms used in graph applications can be roughly described in a vertex-centric manner as it is also expedient for CSR representation.

Accordingly, we process vertices in an order given by a working list, which can also be dynamic. For each vertex v being processed, we may access the neighboring vertices of v and run a function on v, its neighboring vertex, and edge attributes. We may also apply another function directly on vertex v. A rough description of vertex processing is given in Algorithm 1. The *work-list* can be a simple list of all vertices as in some iterative algorithms such as PageRank. Similarly, it can be a queue for a breadth-first search, a stack for a depth-first search, a priority queue for Dijkstra's shortest path algorithm, etc. On the other hand, function f represents the operation on the vertex and/or edge. It depends on the algorithm and can interact with the *work-list*. Even though such abstraction might not always cover all graph algorithms, most of the graph-based applications show similar behavior.

Algorithm 1: General description of vertex-centric graph algorithms, using arbitrary *work-list* and function *f*.

```
Input: G=(V,E)
Initialize work-list
for each vertex v ∈ V in work-list do
    for each edge (v,u) ∈ E or (u,v) ∈ E do
        f(G,v,u)
    end
end
```

Suppose we re-write the algorithm given in Algorithm 1 for graphs represented in CSR format. In that case, we get Algorithm 2, where we limit ourselves to process only the outgoing edges in the inner loop. It is noteworthy to mention that an additional CSC representation of graph can be used for incoming edges for efficient access. Furthermore, function f can access *data* and *values* arrays for graph attributes using i and j indices as well as *rows* and *columns* arrays for topological information.

Algorithm 2: General description of vertex-centric graph algorithms with graph G in CSR representation.

Input: G=(*rows,columns,data,values*)

```
1 Initialize work-list
2 Initialize data if necessary
3 Initialize values if necessary
4 for each vertex index i in work-list do
5 |   for j ← columns[rows[i]] to columns[rows[i + 1]] do
6 |   |   f(G,i,j)
7 |   end
8 end
```

CSR representation is cache-friendly for especially iterative scanning of all vertices in the graph as well as scanning the edges of a particular vertex. However, the representation does not entirely obviate the random access patterns in graph applications. The for loop at line 5 in Algorithm 2 to access edges gives a sequential memory access pattern but accessing the vertex data with the index j usually results in random memory accesses. Moreover, *work-list* of vertices does not necessarily give a sequential pattern for accessing vertex data. The random accesses are illustrated in Figure 2.4, for the same graph given in Figure 2.1. Figure 2.4 **(a)** shows that; processing vertices in the work-list might not always give a sequential pattern, thus a random access pattern is likely to occur. For each vertex index from the work-list we may access the *rows* and *data* arrays. In the figure, we might first process the vertex 0, but then 7, 1, etc. Figure 2.4 **(b)** shows that scanning the edges of a particular vertex has a sequential memory access pattern, as we get the indices of out-neighboring vertices of a particular vertex. On the other hand, if we want to access the data of neighboring vertices, as in Figure 2.4 **(c)**, memory access patterns depend on the topological structure. In the figure, neighboring vertices of vertex 0 are 0, 2, 3, and 6. Once we access the data of vertex 0 (vertex 0 is neighbor to itself), we might have a temporal locality, but accessing vertices 2 and 6 will not likely utilize temporal or spatial locality. On the contrary, vertex 3 might benefit from the spatial locality due to possible early access to vertex 2.

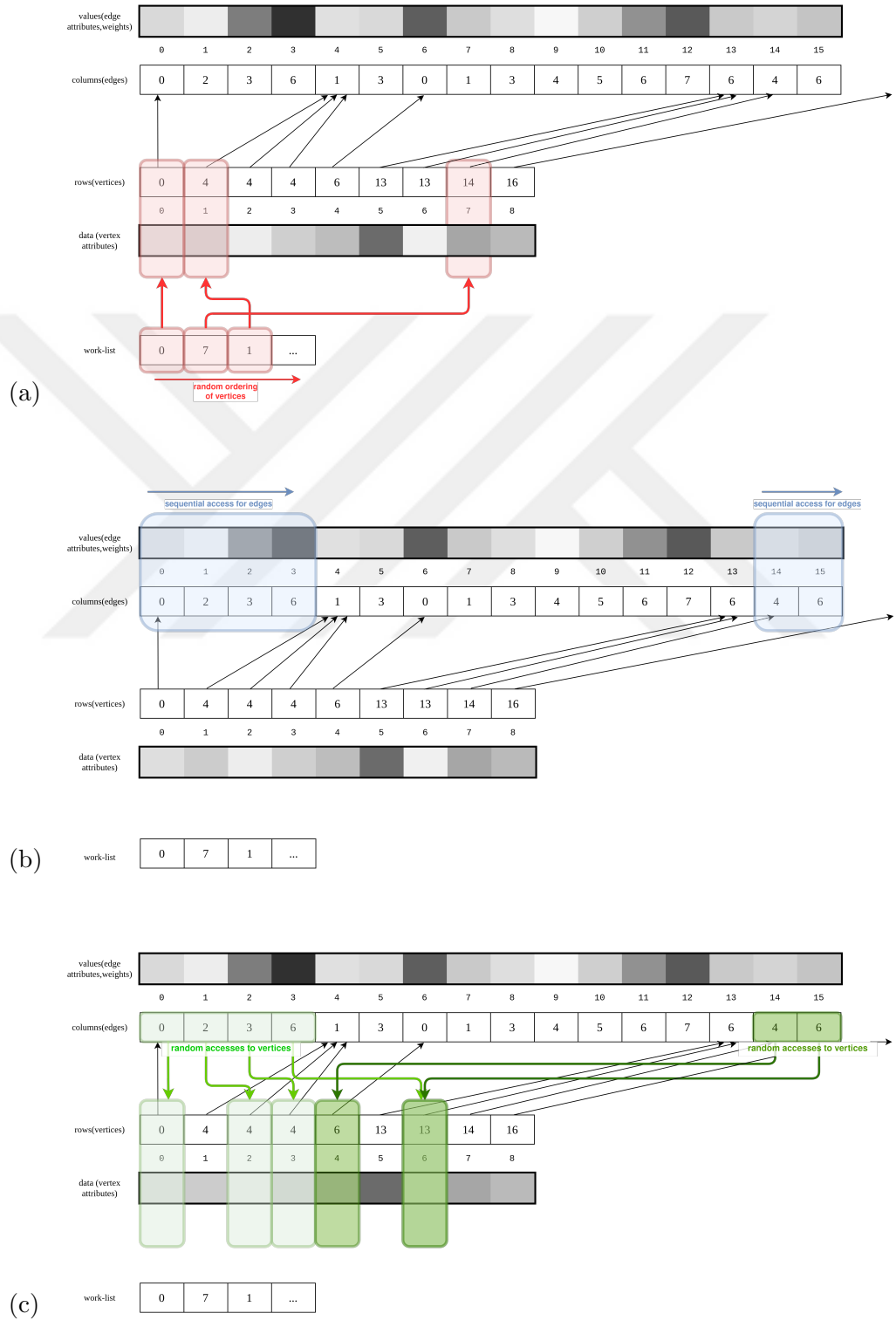


Figure 2.4: Illustrating the access patterns in graph algorithms: (a) possible random access depending on *work-list*, (b) sequential access on getting the neighbors of a vertex, (c) random access pattern on neighboring vertex data.

The random accesses cause the under-utilization of modern cache systems due to poor locality, as their benefits rely on the sequential memory access patterns and temporal locality.

These observations are considered in designing our custom architecture as we also depend on the CSR representation of graphs. In particular, we aim at reducing random memory accesses caused by:

1. irregular order of vertices in the working list, and
2. data-driven access pattern for getting the data of neighboring vertices of a given vertex.

Our architecture addresses the nature of graph applications, and its design is described in the following sections.

Chapter 3

Related Work

3.1 Specialized Hardware

Throughout the history of computing, the hardware design focus has not been only on the development of general-purpose computers but also on special-purpose machines. In fact, the first known digital electronic computer, known as Atanasoff–Berry Computer (ABC) [8], is a special-purpose machine to solve systems of linear equations with its limited instruction set [9]. Modern specialized hardware development mainly consists of accelerators for specific functional computations, domain-specific processors, and application-specific processors.

3.1.1 Accelerators

Hardware acceleration is needed for various reasons, one of which is to provide parallelism on both data and task levels. The accelerators are built as separate hardware units in addition to general-purpose processors to support specialized tasks. The recent accelerator development focuses on various application domains including computer graphics, digital signal processing, cryptography, computer networking, artificial intelligence, scientific computing, data compression, and

bioinformatics. Hardware accelerators are mostly built as dedicated application-specific integrated circuits (ASIC) or field-programmable gate arrays (FPGA). The former lacks flexibility (as it has a fixed design for a specific application) while providing high performance and energy efficiency. The latter is reconfigurable usually with hardware description languages (HDL) or with higher-level programming languages with High-Level Synthesis (HLS).

Even though ASIC manufacturing is a traditional way of customizing hardware, the dynamic nature of applications requires the flexibility of these specialized circuits. The application-specific instruction set processor (ASIP) paradigm is an emerging alternative to ASIC. Nohl et al. remark it as a solid solution that provides flexibility without drawbacks on performance and energy efficiency [10]. Keutzer et al. address the trend from ASIC design to ASIP design, stating the motivations as increasing design cost due to deep-submicron effects related to scaling in physical design, increased complexity with the increasing number of transistors, heterogeneous integration of functionalities on the same die, shrinking time-to-market as a commercial challenge, and increasing manufacturing cost [4].

3.1.2 ASIP Design

Specialized processors can be divided into application-specific and domain-specific processors, where the former has more efficiency but less flexibility. On the other hand, the latter is closer to general-purpose processors as it supports a set of applications in a domain. Since both processor types are programmable and flexible compared to ASIC technologies, they are considered to be part of the ASIP methodology. These specialized processors are also named as customizable processors [11], application domain-specific processors (ADSP) [12], or domain-specific architectures (DSA) [13] from an architectural point of view.

Lenne and Leupers [11] define three design approaches to design customizable processors as (i) parameterizable processors where customization is done

by activating/deactivating functional units and scaling of design parts on a single architectural skeleton such as ARC EM Processor Family[1] and Tensilica[2], (ii) extensible processors where customization is provided with extensions on the instruction set and done with the related functional hardware, register files, memory interfaces, and (iii) custom processor development tools to design from scratch with instruction set generation, compiler design, and hardware design steps. In all of these approaches, programmability is usually provided with the existence of an instruction set. Even though these custom instruction sets are designed for a specific application or domain, customizable processors are still capable of general-purpose computing tasks.

3.2 Application-specific Processors

Application-specific processors provide flexibility in the generations of an application or a small set of similar applications by their design specialization. These processors still use a programmable interface, mostly with compiler support for their instruction sets. We divide them into two: ISA-based and custom hardware designs.

3.2.1 ISA-based Processors

ISA-based processors are designed with ISA (typically on RISC architectures) extensions, also called ISEs (Instruction-Set Extensions), implemented inside a CFU (Custom Functional Unit) along with base-processor core [14] or with parameterizable processors. They use a base ISA specification for basic operations and customize it by adding new instructions for efficiency and performance. Oolong [15] is an extended processor on RISC-V ISA for baseband processing applications. They extend RISC-V by adding instructions for arithmetic operations on complex numbers without changing addressing mode, adding a four-lane 8-bit

[1]https://www.synopsys.com/designware-ip/processor-solutions/arc-em-family.html
[2]https://ip.cadence.com/ipportfolio/tensilica-ip

ALU for data-level parallelism with corresponding instructions, and special functional instructions such as an instruction for Galois Field multiplication. The architecture is flexible for software-defined radio (SDR) applications to support different protocols and evolving algorithms.

EFFEX processor is another processor example for application-specific ones specializing in feature extraction algorithms mainly for computer vision applications on mobile systems proposed by Clemons et al. [16]. They evaluate the processor with FAST, HoG, and SIFT feature extraction algorithms, yet the design is flexible for other algorithms in the domain. The design is heterogeneous as it consists of a single complex superscalar core coupled with a variable number of simple single-issue cores. Complex one is used for control operations, whereas simple cores are designed for repetitive independent data processing. The design is modeled as ISA-based such that complex core uses a custom ARM A8 model and simple cores use ARM A5 (with floating point) model, where both are RISC architectures. They extend the instruction set for three functional units: (i) one–to-many compare (OTM) unit for comparison between a pixel location and its neighbors, (ii) convolution multiply-accumulate (CMAC) unit for inner product operation of two floating-point vectors, and (iii) gradient Unit for gradient operation on single-precision floating-point data for the two dimensions in image data. They also suggest a patch memory architecture to reduce memory latency and throughput by putting 2D regions of the image in a single DRAM row and adding special patch memory instructions to convert a pixel address quickly into its corresponding patch memory address.

3.2.2 Custom Hardware Processors

Custom hardware processors require the design of instruction sets, micro-architecture from scratch, and compiler support. Muller et al. [17] design a multiprocessor platform, where processing elements consist of ASIP for Soft Input Soft Output (SISO) decoding operations used in turbo decoding algorithms. The processor's instruction set contains 30 instructions of eight-bit words. Since

turbo decoding algorithms are based on recursion, SIMD architecture along with special control instruction for butterfly scheme loops and operative instructions for its basic "maximum a posteriori (MAP)" function.

Saponara et al. designed ASIP for Retinex-like processing algorithm related to its image/video filter, consumer electronics, video surveillance, driving assistance, and medical imaging applications along with their improved design on the algorithm. Their design includes pipelining on the pixel level, special SRAMs for parallel processing, single-instruction for nonlinear transformation with adapted pipeline stages, an address generation unit (AGU) for automatic address calculation, and a zero-overhead loop with an instruction set consisting of 42 instructions. The work provides flexibility for external control of filter parameters, the processing of different color spaces, and various input types [18].

3.3 Domain-specific Processors

Domain-specific processors are closer to general-purpose processors in terms of the number of applications they aim to support than application-specific processors. Even though the target domain has many applications, the custom design usually complicates the implementation of applications outside of the domain. Nevertheless, a domain-specific processor is capable of general-purpose tasks. GPU is an example of a domain-specific processor for the graphics domain. Even though it is a processor designed for accelerating applications regarding computer graphics, it can be used as a general-purpose processor with software tools like CUDA and OpenCL. It is more judicious to use GPU for applications involving vectors as it is preferred in high-performance computing tasks for parallelization rather than daily applications despite its capability.

In Deep Neural Network (DNN) research, there is a massive investment in hardware to improve the performance of applications in this domain and reduce the computation cost. Google's Tensor Processing Unit (TPU) is one of the well-known examples of such efforts. Even though TPU is officially stated as an ASIC,

it should not be confused with traditional, inflexible ASIC implementations. It is designed to ease deployment as a coprocessor on the PCIe I/O bus. Its architecture fits more into an ASIP design as it has its own CISC architecture set. It includes special computational units such as Matrix Multiply Unit for 8-bit words and has programmability support through Google's TensorFlow framework[3] providing an interface to TPUs. Its evaluation results show better performance and energy efficiency than CPUs and GPUs for applications in its domain [19].

Myriad 2 is a special system on chip (SoC) designed for the computer vision domain, steered with heat dissipation and energy density constraints. Especially for personal digital assistants (PDA), the system design aimed to dissipate less than 1 W while providing high-performance. Its design has programmable hardware accelerators for specific kernel functions of the domain, a RISC processor for control tasks such as runtime scheduling, vector processors for algorithms in its domain, and a software development kit [20].

Google's Pixel Visual Core is an architecture designed for image processing and computer vision as a replacement of image signal processors (ISPs) on mobile devices for flexibility. Its core processor design includes two dimensional SIMD architecture on 16-bit data words. When performing simultaneous stencil computations between processing elements, the data movement is software controlled. The core also includes a special load-store unit Sheet Generator (SHG) to access blocks of 1x1 to 256x256 pixels and a scalar processor lane (SCL) for control instructions such as branches and interrupts. It uses Halide programming language and a two-step compilation process as in GPUs. Thus, it has two instruction sets; (i) vISA (virtual instruction set architecture) which is similar to RISC-V but it has an image-specific memory model and extensions, and (ii) pISA (physical instruction set architecture) that is designed as VLIW [13].

For the media stream processing, Imagine [21] is proposed driven by real-time constraints of media applications. It exploits the media stream domain's three main characteristics: little data reuse, high data parallelism, and high

[3]https://www.tensorflow.org/

computation-to-memory access ratio. Each streaming data goes through a sequence of functions called kernel operations, and these operations are mapped to parallel arithmetic cluster units by a processor. It has a hierarchical memory system such that streaming data is kept in a stream register file (SRF), and data resulting from inter-kernel operations are stored in a local register file (LRF). The programming model provides the mapping of these kernel operations and parallelization.

3.4 Specialization in Graph Domain

3.4.1 Graph Processing on Acceleretors

Gui et al. discuss graph theory usage in many applications and give a survey of accelerators for the graph domain that target both energy efficiency and performance [2].

Graphicionado [22] is a state-of-the-art accelerator for graph processing, where a pipeline is combined with some fixed parts for common operations such as performing random/sequential memory accesses, keeping active vertices, and custom parts in pipeline stages. They rely on the graph processing model provided by GraphMat [23] to program the applications with the custom hardware modules. They suggest that the custom parts can be implemented as FPGAs, programmable ALUs, or fully custom implementation on a chip for a set of algorithms. Even though the accelerator is not a completely programmable solution for graph processing, it is a high-performance and energy-efficient solution with a flexible design to support a large domain of graph applications. Graphicionado's performance relies on effectively using an on-chip Scratch-Pad Memory (SPM) with access times similar to an L1 cache. Yet, the data in the SPM are explicitly controllable, unlike hardware caches. They are commonly used in graph accelerators to mitigate random memory accesses.

AccuGraph [24] is another state-of-the-art hardware accelerator that provides

a solution for atomic operations on real-world graph data. The same vertex may be read/written by many vertices simultaneously in parallel implementations, where concurrency is a key issue. Unlike the other solutions for race conditions such as graph partitioning and dynamic scheduling structures, they propose a parallel accumulator structure optimized for both low-degree and high-degree vertices. Even though it provides performance efficiency, the design depends on fixed algorithms, thus not fully programmable.

The template-based design methodology is proposed in [25] as a flexible solution to generate hardware accelerators for graph processing in a short amount of time. The design follows Gather-Apply-Scatter (GAS) programming paradigm written in high-level C language to produce the final RTL in SystemC to be deployed in ASIC and FPGAs. The generated accelerators have architectures similar to [26], suited for energy efficiency, and rely on SPMs to handle random memory accesses. GraphGen [27] framework is an accelerator generator for the graph applications domain. Unlike [25], programmability is provided by custom RTL implementations in a vertex-centric manner according to the framework's specifications. An architecture based on Soft-processors on FPGA is proposed in [28]. The custom instruction set and its pipelined datapath are designed to execute graph applications in soft-processor cores. A parallel architecture with Bulk Synchronous Parallel (BSP) [29] model is designed using those soft cores. Even though [25], [27], and [28] provide limited programmability with re-configurable design blocks, the final generated design is still fixed and needs to be deployed on a device. GraphOps [30] supports a larger domain of graph applications by providing building blocks for hardware programmers. It generates a dataflow architecture accelerator to maximize the memory bandwidth for performance. It alleviates the random accesses by using locality-optimized graph representation such that the *data* array in CSR representation is scattered to an array parallel to the *columns* array. Even though it provides locality for accessing the neighbor data (which is better for streaming applications), the system should maintain this locality-optimized array during the computations.

GraphDynS [31] is a more recent accelerator that uses run-time information of the graph computation to have a balanced workload scheduling and run an exact

prefetching scheme for averting memory irregularities and random accesses. The specific computations of the graph algorithms are mapped to specialized processing cores based on their vertex-centric programming model. An accelerator for Graph neural networks (GNN) is proposed in [32], which includes characteristics of the aforementioned graph accelerators as well as elements for the acceleration of Deep Neural Networks (DNN). Unlike the other graph accelerators, it targets only GNN applications rather than the domain of graph applications.

An explicit prefetcher for graph traversals is proposed in [33]. Requiring only custom configuration instructions in ISA, prefetching provides speedup for parallel and single-core implementations due to high cache utilization. As the prefetcher can be installed on any L1 cache with little intervention on the CPU, it is almost fully programmable as long as graph data is represented in CSR format and an explicit work-list so that configuration can be done.

3.4.2 Graph Processing on GPUs and Vector Machines

GPUs are also considered for parallel graph processing. Frequent data transfers between CPU and GPU, ineffective use of GPU caches, load imbalance, limited GPU memory, and control-divergences are considered bottlenecks for GPU's graph processing [2, 34, 35]. In addition to this, Gui et al. [2] point out that GPU solutions consume more energy compared to accelerator solutions. Preprocessing graph data for performance improvements on GPUs is also considered costly [31]. Recent works such as Grus [36] solve the high-performance problem for graph processing on GPUs with Unified Memory (UM) for large graph data. Most GPU solutions either adapt existing software frameworks for programmability support or build their own.

VGL [37] is proposed as a high-performance solution for vector architectures similar to graph processing on GPUs. Its performance relies on the preprocessing step that causes better cache utilization and its specific programming abstraction for vector machines.

Prefetching schemes are also proposed for irregular computations on GPUs. In [38], a hardware-prefetching scheme and optional compiler support are proposed for both direct and indirect memory accesses by exploiting the spare registers in GPUs.

3.4.3 Emerging Technologies

Tesseract [39] is proposed as a Processing-In-Memory (PIM) architecture by using 3D-stacked memory technologies. The architecture maximizes the available memory bandwidth in the parallel system with a high-level API for programming. The API is also utilized for a prefetching scheme embedded in the architecture. GraphiDe [40] is another PIM architecture based on circuit-level schemes and capable of doing bulk bitwise operations inside DRAM. It provides a special ISA that maps the bitwise operations on DRAM to instructions for software support.

GraphR [41] is a graph accelerator based on another emerging technology, ReRAM. It embraces Sparse-Matrix Vector Multiplication (SpMV) for graph computations due to adaptability to ReRAM crossbar architecture and the streaming model for large graphs. Large graph processing on Spin-Orbit Torque Magnetic Random Access Memory (SOT-MRAM) model is studied in GraphS [42] as another promising technology.

3.5 Software for Graph Analytics

Besides the hardware optimizations for graph analytics, software-wise optimizations are also studied. Many frameworks are proposed in this context, targeting distributed environments, GPUs, or disk-based systems. They are adapted on custom parallel architectures as well. High-level abstractions for the graph algorithms are necessary for most software frameworks so that the underlying backend engine can map the algorithm to parallel platforms. These programming abstractions are evaluated in [43]. They commonly follow an iterative paradigm

in which particular functions are iteratively applied to graph data. These abstractions can be divided into vertex-centric, edge-centric, and hybrid programming models [2].

The vertex-centric model is mainly summarized as "think-like-a-vertex," where we define functions for vertices as described in Chapter 2 that follow a similar semantics to Algorithm 1. Pregel [44] uses a vertex-centric approach based on BSP [29] model for distributed systems. Accordingly, graph applications are defined as a single `Compute` function as if for running a single vertex in a graph. Along with functionalities for message passing communication with neighbor vertices, the `Compute` function is called at each super step in the BSP model. The function processes the messages coming from the previous super step. The result of computations can be sent to other neighbors to be used in the next super step until convergence. GraphMat [23] uses a vertex-centric programming paradigm as an interface to map graph algorithms to iterative SpMV operations on distributed systems. Namely, it requires four abstract functions to be defined by a programmer, and these functions are mapped to matrix-vector multiplication operations with the following analogies: `Send_Message` corresponding to vector initialization, `Process_Message` corresponding to scalar multiplication, `Reduce` corresponding to summation, and `Apply` as a final operation on resulting vector of each iterative SpMV step. PowerGraph [45] also follows a vertex-centric approach by Gather-Apply-Scatter (GAS) abstraction for distributed environments. In this abstraction, three main functions are defined by a programmer, namely, `gather`, `apply`, and `scatter`. Those functions are called iteratively until convergence, where `gather` function on the edges of an active vertex is invoked to accumulate a value for each active vertex. Then, the `apply` function on the accumulated value for each active vertex is invoked. After the `apply` function, the `scatter` function is invoked for the edges of the active vertices to produce new edge values for the next iteration.

As opposed to vertex-centric models, X-stream [46] is designed as an edge-centric framework for in-memory computing (graph data fits in memory) and out-of-core computation. It employs the scatter-gather paradigm in the edge-centric manner, which is also an iterative paradigm. Each iterative step consists

of a `scatter` function that runs over all edges in the graph, followed by a `gather` function that is applied to the updates from the `scatter` function. Edge accesses and updates computed by the `scatter` function are sequentially streamed, and random accesses to vertices are alleviated with partitioning.

ComboBLAS [47] is designed by exploiting the matrix-graph duality as a sparse matrix-graph processing framework. It provides algebraic primitives to which graph algorithms can be mapped with their programming interface.

Chapter 4

Custom Architecture and Instruction Set Extension

4.1 Previous Work

A previous study already defines the custom architecture and its instruction set [6]. Even though the core concepts are also employed in this work, some modifications are needed for better results. The previous work is based on a design that allows managing scratch-pad memories (SPM). The design uses an SPM, referred to as Edge Scratch-Pad (ESP), to keep read-only edge data (*columns* array in the CSR representation). The design also divides the cache memory into two parts. One part functions as a general-purpose cache called Global Scratch-Pad (GSP). Vertex Scratch-Pad (VSP) is the other part for storing vertex-related data. The division is based on the load of memory traffic in the graph applications. Accordingly, a small address range of vertex-related data accesses dominates the total memory requests in graph applications. Moreover, the memory accesses in this range are responsible for most of the cache misses due to the nature of graph applications.

ESP is mainly managed by software through custom instructions. On the

other hand, management of VSP and GSP is hardwired in the architecture, and only an initial configuration is needed through custom instruction calls. The configuration gives an address range for VSP such that when CPU requests an address, it is compared with this range. If the address is within the given range, data is accessed from VSP. Otherwise, it is accessed from GSP. The internal workings of VSP and GSP are like a modern-cache system.

The architecture's instruction set is defined by extending the RISC-V ISA [48]. The custom instructions are previously defined in [6] and includes four new instructions to RISC-V instruction set: *spmcon, memspm, spmreg, and delspm*. The *spmcon* instruction is used for configuring the SPMs in the architecture. On the other hand, *memspm, spmreg*, and *delspm* instructions are defined to manage ESP.

4.2 Modifications to Instructions

The same instructions from the previous study are also used in this work. However, minor functionality changes are done on both ISA and micro-architectural levels. The most notable change in micro-architecture is using a single non-blocking cache with a prefetch buffer instead of GSP and VSP structures. The prefetch buffer is employed for the same reason VSP is employed. A small prefetch buffer is suitable rather than dividing the cache into two parts, thus providing more space for the cache. The micro-architectural design is explained in Chapter 5.

At the ISA level, functionalities of custom instructions are modified to reflect the changes in micro-architecture. Graph data is represented in CSR format, and elements of *rows* and *columns* are assumed to be 4 byte integer values. The descriptions of these instructions, their functionality, and the difference from the previous study are listed below:

spmcon

- **Description**: Used for configuration of ESP memory at the start of program execution.
- **Function**: Called three times at the start of a program. The first call to *spmcon* marks the start address of the edge-related data array (*columns* or *values* array at CSR representation) given as an argument. The second call marks the start address of the vertex-related data array given as an argument. Depending on the application, it could be *rows* or *data* arrays in CSR representation or any other array. The last call to *spmcon* gives the size of a single data element of the vertex-related data array in bytes whose address was given in the second *spmcon* call.
- **Difference from the Previous Study**: The previous study uses the first and second calls of *spmcon* to mark the start address of the edge-related and the vertex-related data arrays. But the third call marks the end address of the vertex-related data array and the architecture in previous works does not need to know the size of a single element in this array. Since we developed a prefetching scheme over this array, we need the element size information as well.

memspm

- **Description**: Used to move edge array data from external memory to ESP memory.
- **Function**: The address of the element of *rows* array in CSR representation is given as an argument to the instruction to indicate the edges of the given vertex index. The edge data is to be placed on ESP. To fetch the edge data from the memory, the vertex index is fetched first from the memory as given in *memspm* argument. The given index is multiplied by 4 and added to the start address of the edge array previously marked by *spmcon* to get the starting location (offset) of the given vertex index. The chunk of edges is retrieved from the memory accordingly, irrespective of the number of edges that the vertex has.

- **Difference from Previous Study**: Calling the *memspm* is similar to previous work. However, internal workings are modified to be more straightforward. In the previous work, a controller is responsible for getting the offset of the vertex index and the offset of the next vertex in *rows* array. It only loads the edge data if the two offsets are not equal; that is to say, the given vertex should have non-zero edges. In this modified version, the offset of the next vertex is not sought, and the edges are fetched even if the vertex has no edges. This provides pre-fetching edge data, especially for iterative algorithms, since the fetched edges will be the edges of the next vertex in the *rows* array. Another subtle difference is that if a chunk of edge data is already in ESP marked by an offset value calculated earlier, the controller may decide not to retrieve them.[1]

spmreg

- **Description**: Used to read edge array data values from ESP to registers in the CPU core.
- **Function**: The usage is the same as a load instruction, that we give the address of data we request. The address is expected to be an address in the edge array. The address is searched on ESP, and the data is sent to the register file if found in ESP. If not found, a memory request is issued as an ordinary load instruction.
- **Difference from Previous Study**: The previous study does not issue a memory request if the given address is not in ESP but instead gives an indicator to the software for the miss. Software is then responsible for loading the data. The usage of *spmreg* is also considered as an indicator for prefetching scheme discussed in Chapter 5. The *spmreg* might also issue memory requests for chunks of edge array data as *memspm* does.

[1]Consider using *memspm* for vertex 1 that has no edges in Figure 2.3. Using *memspm* later on vertices 2 and 3 will be ineffective since we have already marked the ESP with the same offset value. When we want to scan the edges of vertex 3 later, those edges may already be in ESP due to the early *memspm* request of vertex 1.

delspm

- **Description**: Used to remove edge array data from ESP memory.
- **Function**: Address of an offset at edge-related data array is given as an argument to *delspm* instruction, which is then searched in the chunks of ESP. Suppose the offset of a chunk of edge data previously fetched from memory by a *memspm* instruction matches the given argument of *delspm*. In that case, the chunk can become free to be replaced by another chunk of edge data.
- **Difference from the Previous Study**: The semantic and internal workings of *delspm* are almost identical to previous work. Only a slight change to keep free locations in ESP is done at the micro-architectural level.

A summary of these custom instructions over the CSR representation of the sample graph is given in Figure 4.1. The *spmcon* marks the start address of *columns* and *data* arrays. It also marks the size of a single element of *data* array. Vertices 1 and 7 are given as subjects for *memspm* instruction so that their edge data are placed on ESP. Note that vertex 1 has no edges, but the chunk of edges pointed by its *rows*[1] value are still placed on ESP. The *delspm* marks the chunks of edge data that are in ESP memory to be free such that they can be replaced by another *memspm* request. For example, after we call *delspm* on *columns*[*rows*[3] (edges of the vertex 3), a *memspm* request can purge the edge data associated with this free chunk and place its related edge data. The *spmreg* is used to get the edge data of a vertex. Usage of it also gives clues on prefetching-scheme. In the figure, if we want to get the first edge of vertex 7 by giving *columns*[*rows*[7] + 0]] as an argument to *spmreg*, we get 4 as the index of outgoing neighbor. While getting this information, we can also make a prefetching request for the subsequent outgoing neighbor data, which is 6 according to the Figure. Thus, *data*[6] can be prefetched from external memory that is likely to be used in the future execution of the program. The prefetching scheme is explained in Chapter 5 in detail.

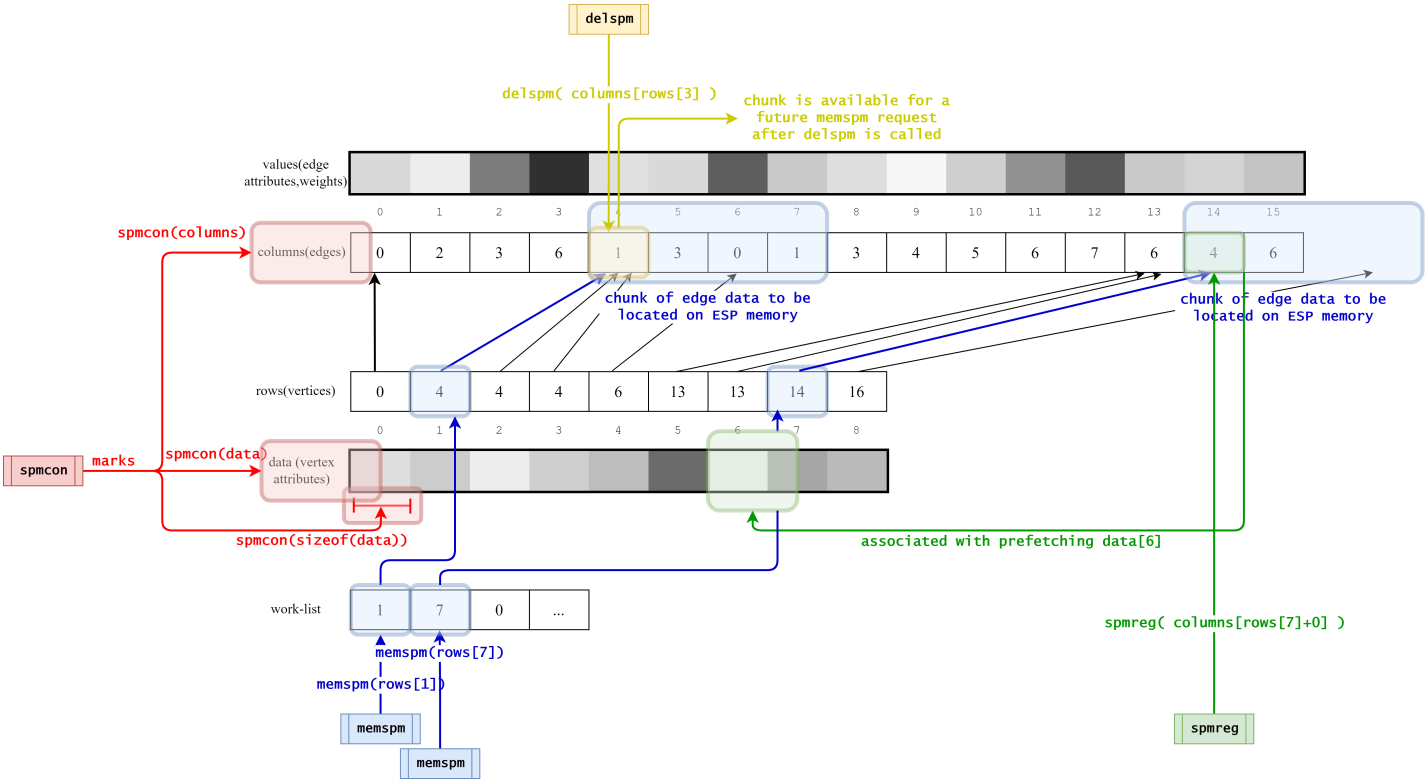


Figure 4.1: Summary of custom instructions used in graph processing with CSR representation. The instructions are given for the sample graph in Figure 2.1.

Chapter 5

Micro-architectural Design

The micro-architecture of the system is explained in this chapter. The main challenge in developing the architecture is to have a design that allows making multiple memory requests without blocking the execution of the CPU. Two major components are designed to address this challenge, *ESP Controller* and *Non-blocking Cache*. ESP Controller is a hardware unit that runs asynchronously with the CPU and manages ESP. CPU communicates with the ESP Controller when custom instructions are to be executed. ESP Controller can issue memory requests along with the CPU. Since both CPU and ESP Controller can issue memory requests simultaneously, a non-blocking cache structure is designed. This way, multiple outstanding memory requests can be handled. We first describe the overall system, then elaborate on the major components in subsections.

5.1 Overall System

The system consists of 6 hardware components: *CPU* with its *Instruction Cache*, ESP Controller, *Request Arbiter*, Non-blocking Cache, *Miss Status Holding Registers (MSHR) Controller*, and *RAM Memory*. The interactions between these

components are illustrated in Figure 5.1. Core CPU interacts with ESP Controller when executing the custom instructions defined earlier in Chapter 4. The *spmcon*, *memspm*, and *delspm* instructions do not cause a stall in the CPU and are executed immediately by directing the proper signals to ESP Controller. ESP Controller uses a queue (*Read Buffer*) to keep the signals from the CPU and executes these instructions. On the other hand, the *spmreg* instruction causes a stall on the CPU as if it is a *load* instruction. CPU waits until ESP Controller responds with the requested data for the *spmreg* instruction. The waiting time depends on the availability of data in ESP (we also refer to ESP as *SPM*, as we only have a single scratch-pad memory structure in the design). Both CPU and ESP Controller can send multiple data requests to memory. Request Arbiter stands as an interface to the memory for both CPU and ESP Controller. It keeps the order of requests for CPU and ESP separately in queues (*CPU Result Queue* and *ESP Result Queue*). It directs the memory requests to the Non-blocking Cache, and the Non-blocking cache responds with a miss or a hit, depending on the availability. The results of requests are kept in the Request Arbiter queues. When the head of queues holds "ready" data (cache hit or a future memory response on cache misses), the Request Arbiter directs the result to the CPU or ESP Controller. The non-blocking cache has its tag and data memories that keep the local data. When data is available for a memory request coming from Request Arbiter, it returns the result as a hit. When the data is not available, it allocates a space and issues memory request(s) through Miss Status Holding Registers (MSHR) Controller. Since it uses a write-back policy for dirty lines, it can issue more than one memory request for a data request coming from Request Arbiter. The non-blocking cache also holds a *Prefetch Buffer* that keeps the result of predictable future memory requests. MSHR Controller issues the actual memory requests coming from the Non-blocking Cache. It utilizes the MSHR structure proposed by Kroft [49] to handle multiple outstanding misses in the cache. When a memory response is received, MSHR Controller will direct the response to Non-blocking Cache and Request Arbiter.

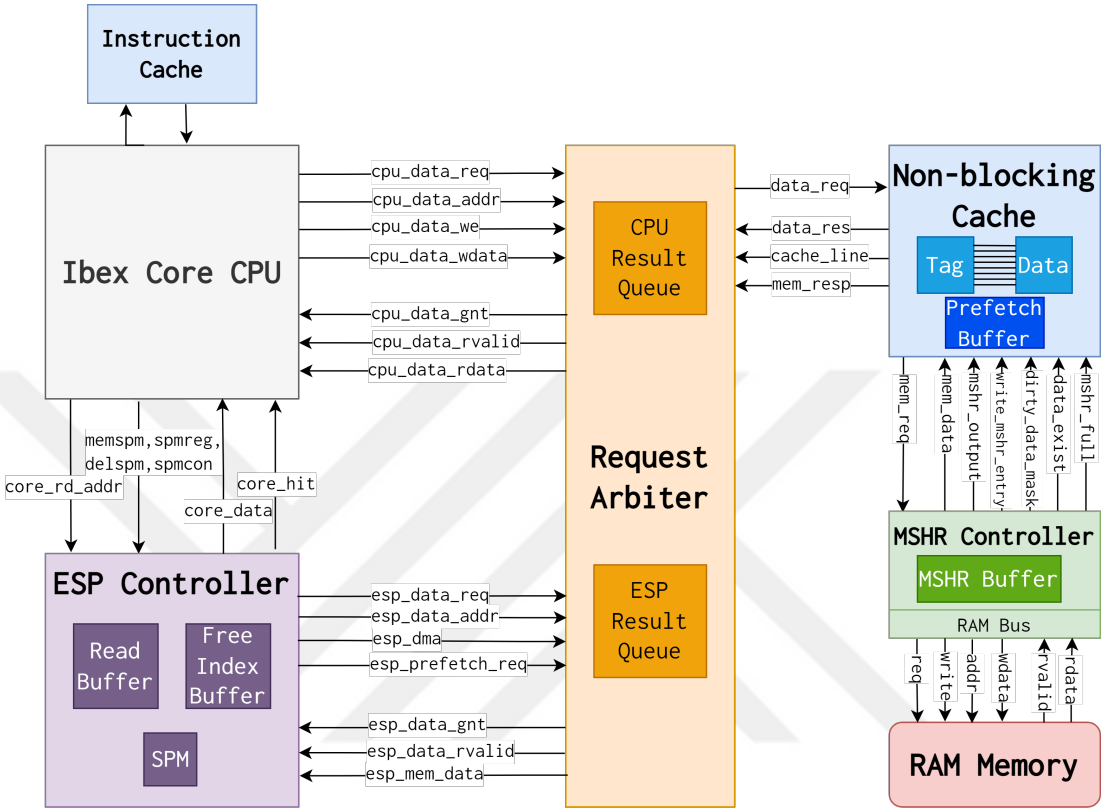


Figure 5.1: Overall System with the interactions of its 6 six components: CPU with its Instruction Cache, ESP Controller, Request Arbiter, Non-blocking Cache, Miss Status Holding Registers (MSHR) Controller, and RAM Memory.

The components in the design keep tables for storing data during the execution. These tables are mostly implemented as buffers that work in a First In First Out (FIFO) manner, and their fields are illustrated in Figure 5.2. We will refer to this figure in the following subsections while describing the components.

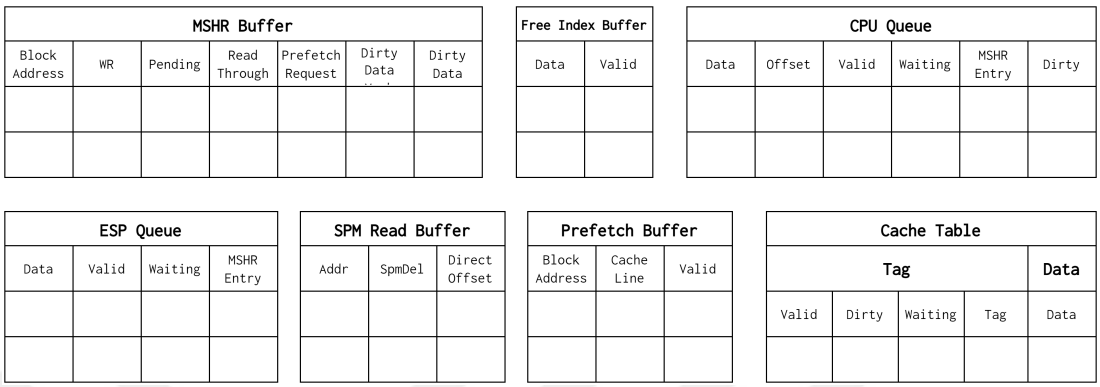


Figure 5.2: Summary of tables (buffers) used in the custom processor.

5.2 ESP Controller

ESP Controller is responsible for managing the ESP and executing custom instructions. The controller also existed in previous work [6], but many modifications are done to the interior architecture. CPU directs the single-bit signals for *memspm* and *delspm* instructions and a 32-bit address value to the ESP Controller so that ESP Controller keeps the required signals in a queue (Read Buffer) to execute without stalling the CPU. For the execution of *spmcon* instruction, the associated 32-bit signal (address or data size information) and a single bit signal for the instruction are provided by the CPU to ESP Controller. ESP Controller keeps the start address of edge-related data array at `off_base_addr` register, start address of vertex-related data array at `data_addr` register, and the size of a single data element of the vertex-related data array at `data_item_size` register within the same execution cycle with CPU. Only *spmreg* instruction will stall due to waiting for the data from ESP Controller to arrive. ESP Controller consists of 5 extended finite state machines (*Spmreg Controller, Memspm Controller, Memory (Request) Controller, Memspm Block Receiver Controller, Spmreg Receiver Controller*), a register file to store configuration data and intermediate signals, a scratchpad memory (SPM), two buffers (Read Buffer, *Free Index Buffer*) that act as queues for incoming CPU signals and keep available locations for SPM.

Interactions between these components are illustrated in Figure 5.3. On the left-hand side of the figure, *core_rd_addr* is the 32-bit signal coming from the CPU that gives an address value except for the third call of *spmcon* instruction. The signal is directed to Read Buffer, Register File, and Spmreg Controller. Single bit *memspm* and *delspm* signals are directed to Read Buffer as well. Single bit *spmcon* signal is directed to Register File that functions similar to a "write-enable" signal on a register file in CPUs. Single bit *spmreg* signal is directed to Spmreg Controller that handles the data request from the CPU. Once data is handled, single bit *core_hit* becomes active and data is ready at the 32-bit *core_data* signal. On the right-hand side of the figure, *esp_data_req, esp_data_addr, esp_dma*, and *esp_prefetch_req* are directed to the Request Arbiter for memory requests from the Memory Controller in the ESP Controller. *esp_data_req* is a single bit signal for requesting data from memory, *esp_data_addr* is a 32-bit signal for the address of data in memory. *esp_dma* is a single bit that indicates direct memory accesses. We also refer to it as read-through mode as the requested data is not stored in the cache but is directly sent to the ESP memory. *esp_prefetch_req* is a single bit signal to indicate the prefetch requests. The *esp_data_gnt, esp_data_rvalid, esp_mem_data* signals are for the results of memory requests, the first one is a single bit "grant" signal that indicates the memory request is acknowledged by the Request Arbiter, the second one is also a single bit signal that is active when 128-bit *esp_mem_data* signal is ready. We use a 128-bit signal for a block of data in a cache line; thus the ESP Controller also uses 128-bit data for memory request results. Depending on the order and current execution, the memory data is directed to either the Memspm Block Receiver Controller or the Spmreg Receiver Controller.

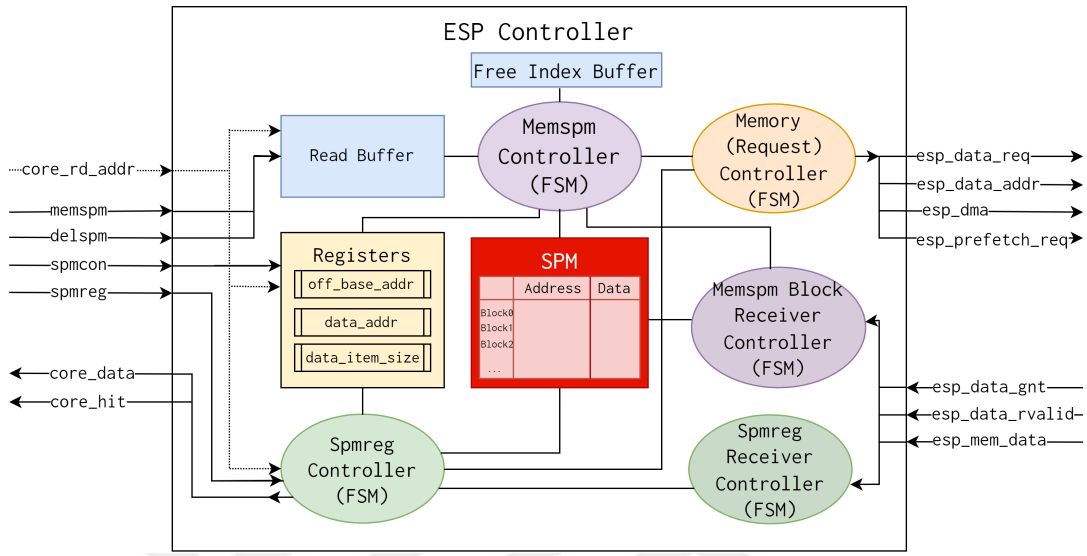


Figure 5.3: ESP Controller and its interactions with the other components.

Scratch-pad memory functions almost the same as in the previous work. Edge array data are moved in chunks into the SPM. A chunk consists of blocks of words whose size is considered to be an architectural parameter. The SPM holds more than one chunk, where each chunk points to a different edge array address. An example with four chunks, each consisting of sixteen 32-bit words is given in Figure 5.4. The SPM capacity (number of chunks) is set as 4 in this figure, where each chunk holds 16 words (4 memory blocks). Each chunk is identified by the address of the first memory block in the chunk. When edge data is requested, an associative check is done on the chunk identifiers to see if data exists in the SPM. For example, on the execution of *memspm* instruction, we check if the requested edge data is already in the SPM. When executing *delspm*, we check if the chunk number for the given address is in the SPM, so that we can set the chunk as available to be replaced by a future request. While executing the *spmreg* instruction, we check the address of the given argument against the addresses in the SPM.

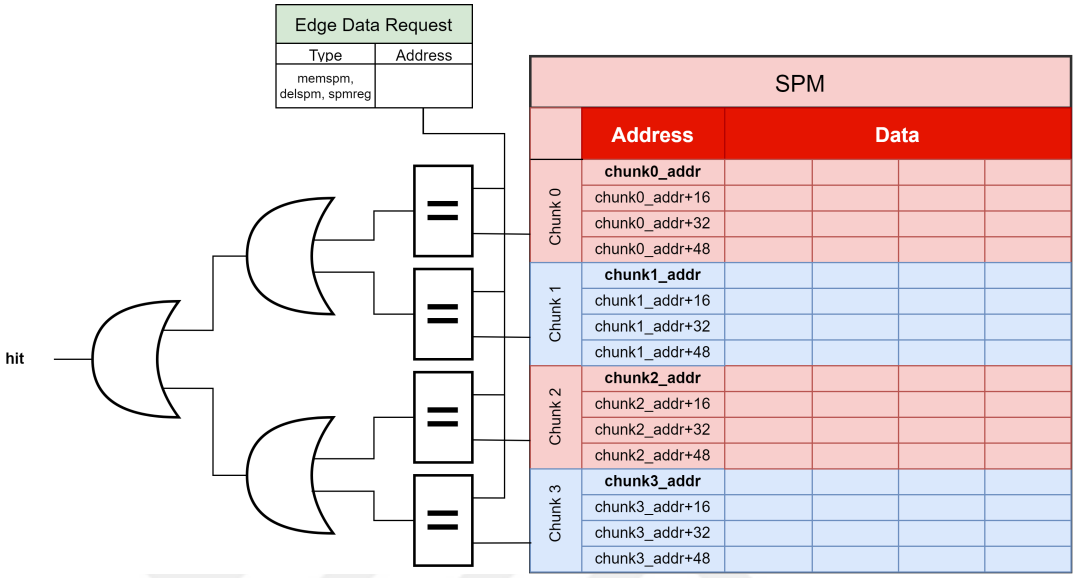


Figure 5.4: Scratchpad Memory (SPM) and associative check on edge data requests in ESP Controller.

Memspm Controller is responsible for the execution of *memspm* and *delspm* instructions. When the CPU sends a signal for the execution of these instructions, the values are stored in Read Buffer. The Read Buffer acts as a queue and stores a 32-bit address field, a 1-bit SpmDel field, and a 1-bit Direct Offset field (see Figure 5.2). Memspm Controller dequeues the Read Buffer when new requests are available, then updates the SPM and Free Index Buffer on execution. Diagram is given in Figure 5.5 to show how Memspm Controller works. It starts with an *idle* state. Whenever there are new execution requests in Read Buffer while in *idle*, it goes to *request-offset* state to initiate the processing. If the execution is for *delspm* instruction, it may update the Free Index Buffer. Otherwise, it is a *memspm* request. If there are free chunks available in SPM, we can request edge-data. There are two modes of *memspm* request. For both of these request modes, if all chunks are full in the SPM, we simply ignore the request and go back to the *idle* state from the *request-offset* state.

The first *memspm* request mode is the indirect mode from CPU execution. In this mode, the given address is the address at the *rows* array in CSR representation. Because this is an indirect address for edge data, we should first fetch

the index data at the *rows* array that points to the *columns* array. Thus, we make a memory request by issuing a signal to Memory Controller. If the memory controller grants Memspm Controller's request, Memspm Controller can transition to *read-offset* state. At this state, it simply waits for memory data to be available. As soon as the memory data is available and is meant for Memspm Controller, we can compute the actual edge address. In doing so, we compute it as ($off\_base\_addr + esp\_mem\_data \times 4$) and transition to *address-check* state. The second mode of *memspm* request is direct offset mode which comes from Spmreg Controller, explained later. In this mode, the address is already the actual edge-data address, so we directly transition to *address-check* state from the *request-offset* state if free chunks are available in SPM.

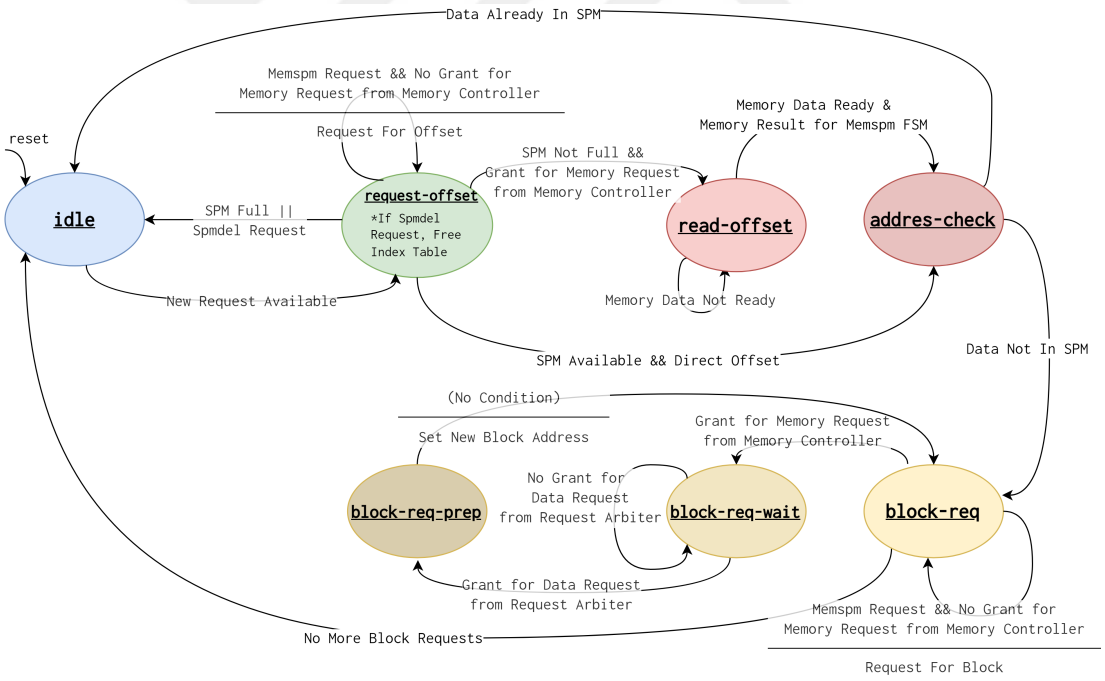


Figure 5.5: Memspm Controller FSM diagram.

For both modes, we reach to the *address-check* state, and we check if the address is already in SPM. If the data is already in SPM, we transition back to the *idle* state. Otherwise, we should request edge-data from memory by transitioning to the *block-req* state. When requesting memory blocks at Memspm Controller, we ask consecutive memory blocks without waiting for memory data to return.

Receiving the memory data for edge blocks is the responsibility of the Memspm Block Receiver Controller. At Memspm Controller, we initiate an edge-block request at *block-req* state. Here we send the request signal for edge data. As soon as Memory Controller gives a grant for the request of the Memspm Controller, it goes to the *block-req-wait* state. In this state, the Memspm Controller waits for the actual grant signal coming from the Request Arbiter. This is to ensure that we get the grant signal from the Request Arbiter. The Memory Controller in the ESP Controller might grant the memory request for the Memspm Controller, but the Request Arbiter might not due to some reasons because MSHR might not be available, CPU Request may come at the same time, etc. After the Request Arbiter responds to a grant signal for the edge data request, we transition to the *block-req-prep* state, where we compute the address of the next block. As the following block address is calculated at this state, we transition back to the *block-req* state. We iterate the block edge block requesting depending on the chunk size in SPM. For the example in Figure 5.4, we hold four memory blocks in a single chunk, so we request four memory blocks. When we request enough to fill a chunk in SPM, we transition from the *block-req* state to the *idle* state.

Spmreg Controller is responsible for the execution of *spmreg* instructions. Its state diagram is shown in Figure 5.6. As in the Memspm Controller, it starts with an *idle* state. Whenever an *spmreg* instruction is executed, the CPU sends the *spmreg* signal to Spmreg Controller through ESP Controller and Spmreg Controller goes into the *look-table* state. At the *look-table* state, SPM is checked for the address requested by the CPU. If data for the given address is on the SPM, it activates the *core_hit* signal and sends the edge data in SPM to the CPU. While sending the data, Spmreg Controller might issue a prefetch request to memory. This prefetch request is made to get vertex data of the next neighbor. In Chapter 2, it is illustrated that once we get a neighboring index of a vertex and access data of the neighbor, it usually results in random memory access (See Figure 2.4(c)). We use a prefetching technique to fetch the data as anticipated by the *spmreg* instruction to remedy this situation. When we use *spmreg* instruction, we get the neighboring vertex index from SPM quickly and indicate that we might need to use the data of the neighboring vertex. Moreover, when we access a

neighbor of a vertex, we are likely to access the next neighbor of the same vertex as described in Chapter 2. Since the edge-data, in other words, the neighbor indices, are in SPM, we can quickly get the index of the next neighbor. The algorithmic execution might later need to get the next neighbor and the data related to the next neighbor. Thus, we can prefetch the data of the following neighbor while execution continues for the current neighbor in the CPU. If we refer to Figure 4.1 and consider the spmreg access to get the first neighbor of vertex 7, we get that it is 5. If we assume that this data is in the SPM already, we most likely have information in the SPM that the next neighbor is 6 (*columns*[14] and *columns*[15] are in the same chunk). Now we can issue a memory request for *data*[6]. We could anticipate that we will access *data*[5] and ask why not prefetch it. But this access happens in a short time after getting the neighbor index 5 and prefetching *data*[5] would have minimal effect on reducing latency. Thus, instead of prefetching the *data*[5], we prefetch *data*[6].

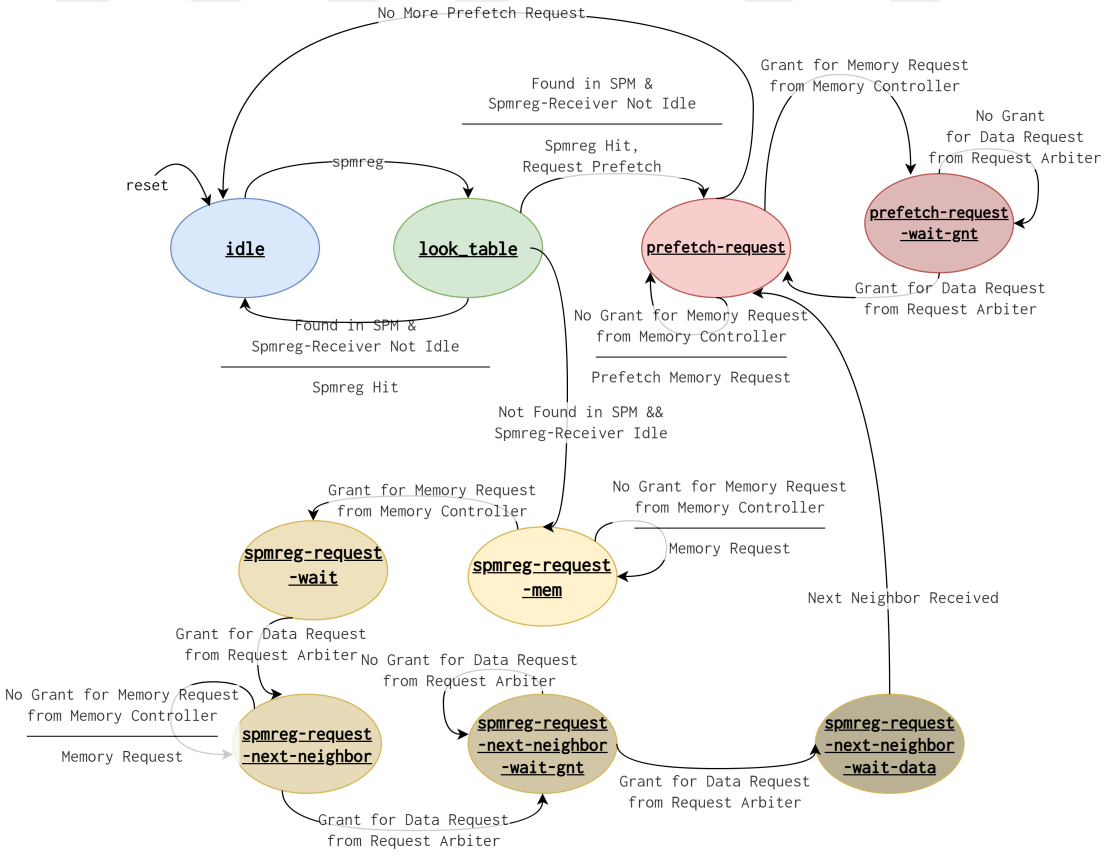


Figure 5.6: Spmreg Controller FSM diagram.

Spmreg Controller proceeds to make a prefetch request if the Spmreg Receiver Controller is idle. Suppose it decides to do so while in the *look-table* state, it goes to the *prefetch-request* state. Otherwise, it goes back to the *idle* state. At the *prefetch-request* state, the Spmreg Controller computes the address of the following neighbor as ($data\_addr + next\_neighbor\_index \times data\_item\_size$). The *next_neighbor_index* is found in the SPM as the next word of the current requested word in the *spmreg* call. Having computed the address, Spmreg Controller requests the data from Memory Controller at the ESP Controller. After grant is given to Spmreg Controller, it goes to the *prefetch-request-wait-gnt* state where it waits for the actual grant signal from Request Arbiter. The waiting for the actual grant signal from Request Arbiter is similar to making memory requests in Memspm Controller. When a grant signal is given from Request Arbiter, it goes back to the *prefetch-request* state. There could be more than one prefetching request related to data location, an architectural parameter. In other words, we can prefetch the next memory block of the previous prefetched request address. Thus, we may iterate the state transition from *prefetch-request* state to *prefetch-request-wait-gnt* state, similar to Memspm Controller behaves for requesting edge data. After no more prefetch request is required, Spmreg Controller transitions from the *prefetch-request* state to the *idle* state and becomes ready for new *spmreg* requests from the CPU.

It is said that edge data is checked at SPM while in the *look-table* state. If edge data is not found in SPM, Spmreg Controller goes to the *spmreg-request-mem* state. At this state, it requests for the edge data whose address is given in *spmreg* instruction. It first waits for a grant from the Memory Controller and then from the Request Arbiter, similar to the other memory request transitions. As soon as the grant is given from the Memory Controller while in the *spmreg-request-wait* state, it goes to the *spmreg-request-next-neighbor* state and requests for the next neighbor index. This is done to make a prefetch request for the next neighbor even if data is not in SPM yet. As Request Arbiter grants for the request of the next neighbor, Spmreg Controller transitions to the *spmreg-request-next-neighbor-wait-gnt* state. And from this state, it transitions to the *spmreg-request-next-neighbor-wait-data* state and waits here for the next

neighbor's index data so that it can make prefetch requests. As soon as the data comes from Request Arbiter, it transitions to the *prefetch-request* state and creates prefetch requests as explained before.

Note that, Spmreg Controller does not activate the *core_hit signal* if we have to make data request for the requested address. The Spmreg Receiver Controller is responsible for this, and as soon as the Memory Controller responds with the data for the current *spmreg* request, the *core_hit* signal to the CPU is activated. It means that the CPU is stalled by ESP Controller while waiting for the data from memory.

In making edge-data block requests, it is told that there are two modes for *memspm* requests. While the indirect mode is initiated from the CPU, the direct offset mode is initiated from Spmreg Controller. If data is not found while checking in the *look-table* state, we issue a *memspm* request for missing edge data. The *memspm* request is given to the Read Buffer, just like how CPU issues *memspm* requests. The only difference is that we already know the actual address of edge-data as *spmreg* instruction has it in its argument. Thus, we issue a direct offset mode *memspm* request from the Spmreg Controller.

For both the Memspm Controller and the Spmreg Controller, if they make a memory request, they first wait for a grant signal from the Memory Controller at the ESP Controller. The function of the Memory Controller is to ensure that only one component at the ESP Controller can request data and control the order of requests, so that received data from memory is delivered to the correct component. While the Memory Controller is responsible for making the requests and keeping the order, receiving the memory data is given to the Memspm Block Receiver Controller and Spmreg Receiver Controller.

As in describing the function of Memspm Controller, it requests edge data without waiting for the data to arrive from memory. Memspm Block Receiver Controller is initiated to listen to the upcoming edge data from the memory once requests for edge data are issued. A simple FSM diagram for it is given in Figure 5.7. On an edge-block request, it goes to the *receive-block* state from

the *idle* state and listens for the edge data. When edge data comes, it puts them onto SPM. It continues to listen for edge data according to the number of requests made by the Memspm Controller. After all edges are received for a current *memspm* request, it goes back to the *idle* state.

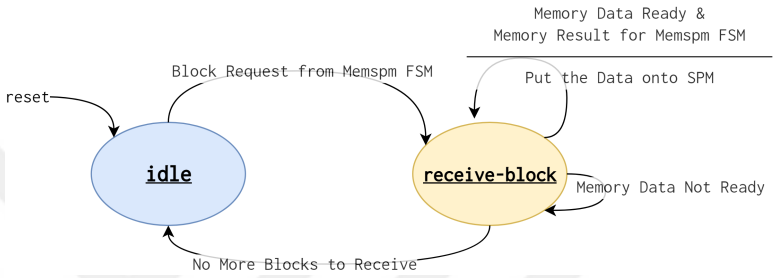


Figure 5.7: Memspm Block Receiver Controller FSM diagram.

ESP Controller can request the memory data in various ways. The first one is a normal data request interpreted the same as CPU's data requests. This request is made when Spmreg Controller requests data for edge data and the subsequent neighbor data when the request for *spmreg* is not found in SPM. The second way of the data request is direct memory access or read-through mode. In this request, the requesting component indicates that the requested data should not be stored in the cache, but data should come through the cache instead. This is especially useful for edge data requests as we are already using the SPM for storing edge data with fast access. If we do not use the read-through mode, then the same edge data will also occupy a space in the cache and spoil the purpose of using an SPM. The third way of requesting data is prefetch requests. This is done for prefetch requests made by Spmreg Controller. The size of this buffer is minimal compared to the size of the cache table, and it works in a FIFO manner. This means that prefetch requests do not directly occupy a space in the cache, but they are moved to the cache table only if a standard data request is made for the prefetched data.

Spmreg Block Receiver works similar to the Memspm Block Receiver, except it might wait for different kinds of data. An FSM diagram for it is given in Figure 5.8. While in *idle* state, it can go to either *receive-edge* state if the

current *spmreg* request is not found in SPM or goes to *receive-prefetch* state otherwise. At the *receive-edge* state, it waits for the edge-data for the current spmreg request. As soon as the requested data arrives, it activates the *core_hit* signal, and the CPU can continue to execute. Spmreg Controller goes to the *spmreg-receive-next-neighbor* state and waits for the address of the next neighbor. As soon as the next neighbor data comes, it now goes to the *receive-prefetch* state and waits for prefetch data requests to be completed. It waits similar to Memspm Block Receiver waits for the edge data, as there might be more than one iteration of prefetch data requests.

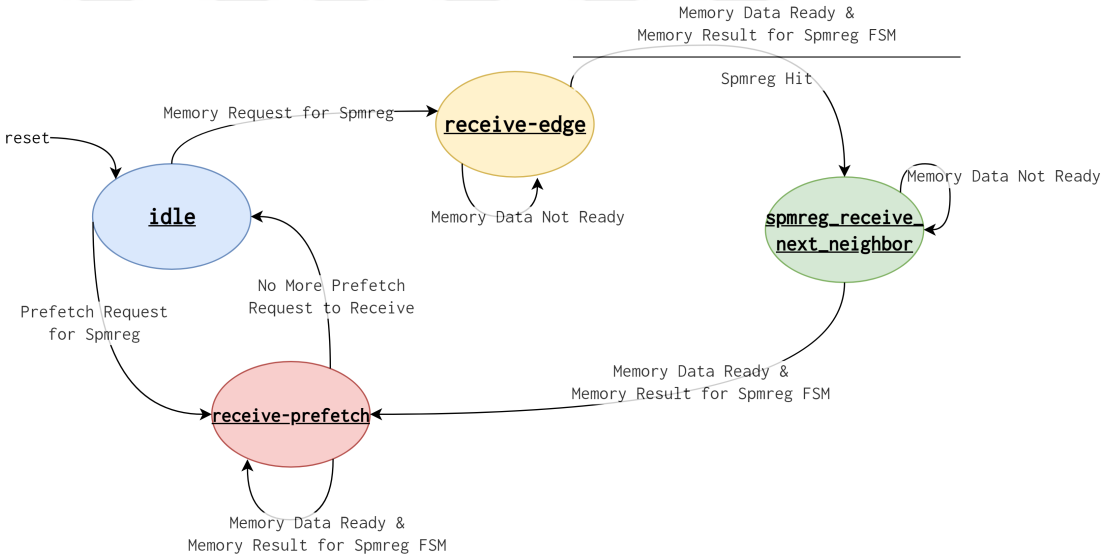


Figure 5.8: Spmreg Block Receiver Controller FSM diagram.

Free Index Buffer gives Memspm Controller a chunk index value to store the chunk of edges of the requested address. It works in a FIFO manner and holds an index value as data whose size depends on the size of SPM and a valid bit (see Figure 5.2). When the index value at the head of the queue is given to Memspm Controller, the head moves to the next free index element. If the head of the queue buffer points to an invalid index element, then no free index is available for *memspm* requests. *delspm* instruction adds new free index values to the tail of the queue. At executing the request for *delspm* instruction, the chunk index corresponding to the given address in *delspm* is found in SPM, and the index

value is appended to Free Index Buffer as a valid free index. For each *memspm* request on an address, there should be a corresponding *delspm* instruction in the application, or SPM become full and cannot process any future *memspm* instructions. Even though *spmreg* instruction call can also issue *memspm* requests, the free index given to Memspm Controller will be appended again to the Free Index Buffer. It works like calling a *delspm* just after the direct *memspm* requests so that the chunk is still subject to replacement. The programmer is not responsible for freeing the chunks of *memspm* requests made by *spmreg* instructions.

5.3 Request Arbiter

Request Arbiter is the memory interface for CPU and ESP Controller. It is an arbiter between the processing units and memory. When it receives a memory request from a processing unit, it transmits it to Non-blocking Cache and waits for a hit or a miss response. While waiting for the response from the cache, it cannot accept any other memory request. As soon as the cache responds, it records the response for the data request. The records are kept in separate buffers for the CPU and ESP Controller. A record keeps the data coming from cache, valid bit, waiting bit, and MSHR entry index for both buffers. If the requested data is in the cache, hit and valid bits are set to 1, whereas the waiting bit is set to 0. If the requested data is not in the cache, it is a miss, so the valid bit is 0 and waiting bit is 1. On a miss, the MSHR entry field of the record is also updated such that it waits for the data of associated entry in MSHR Buffer at MSHR Controller to be available. After a miss or a hit, the Request Arbiter can receive other requests. The processing units can request as many as the buffer size is capable, and they get the result data in the order they asked for. These buffers are implemented in a FIFO manner to keep the order of requests, thus named CPU Result Queue and ESP Result Queue (see Figure 5.2 for fields). Whenever the head of these queues holds a record with the valid bit on, it reads the record and transmits the result to the associated processing unit. While reading the record, valid and waiting bits are set to zero.

A 1 waiting bit means that the record waits for the data to be available from memory. MSHR keeps the information for these outstanding misses in a table. A waiting record in the queues has the MSHR entry field as a pointer to the table. When data becomes available at an MSHR entry pointed by a record, the data at MSHR is transmitted to the record, the waiting bit is set to 0, and the valid bit is set to 1.

CPU Result Queue and ESP Result Queue are separated because they hold different kinds of information. The first difference is the data size. ESP Controller requests for memory blocks instead of single words. As in our system, a memory block is 128-bit, and the data size for the record is 128-bit in ESP Result Queue. On the other hand, CPU requests for single 32-bit words; thus, the data size in the CPU Result is 32-bit. It also has a 2-bit offset value to locate the word in a memory block. The offset is needed because an outstanding miss in the MSHR table holds a memory block address rather than a single word address. CPU Result Queue also has a dirty bit in each record. This is used for the cases where we first have a write-miss, then have a read-miss for the same data address. When data is available in MSHR, we should not read the data coming from MSHR but the dirty data that was previously written. The logic for dirty data is explained in subsequent sections. ESP Result Queue does not need the dirty data bit because it only works with read-only data.

5.4 Non-blocking Cache

The Non-blocking cache receives the data requests from Request Arbiter and responds by either a hit or a miss. It is a non-blocking cache because it can handle multiple misses without blocking the execution of neither CPU nor ESP Controller. It does so by cooperating with MSHR Controller that holds the MSHR table. The Non-blocking cache keeps a table of Tag and Data pairs as in direct-mapped cache implementations [50]. The Tag part of the table holds a valid bit, a dirty bit, a waiting bit, and a tag field whose size depends on the size of the cache. The data part of the cache table holds 128-bit memory blocks

of data, also referred to as the cache line (see Figure 5.2). The non-blocking cache also has a Prefetch Buffer, a controller, and an MSHR receiver component for upcoming memory results. A detailed diagram of the Non-blocking Cache is given in Figure 5.9, which shows the interactions between these sub-components.

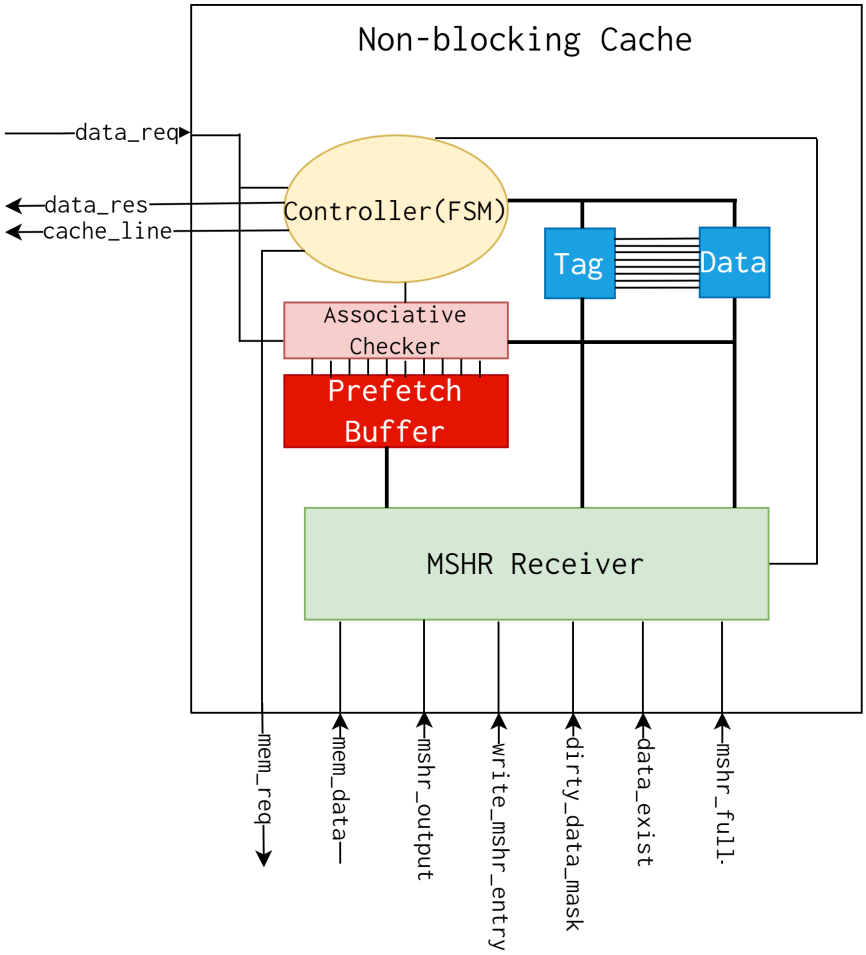


Figure 5.9: Non-blocking Cache diagram.

The Controller at the Non-blocking Cache is mainly responsible for the interactions for the data requests coming from the Request Arbiter and making the memory requests in the case of missing data or write-back requests. It is implemented as an extended finite state machine. Its state diagram is given in Figure 5.10. The controller FSM starts in an *idle* state and waits for upcoming data requests. On a valid data request, it transitions to the *compare* state and checks if the requested address' mapped location is in the cache by comparing

tags. If tags are equal and the cache holds valid data, it means the requested data is in the cache and a hit signal is generated for Request Arbiter to process the data. If it is a hit on a write request, the dirty bit is also set to 1. If data is not found in the cache, it means a miss unless it is in the Prefetch Buffer.

We can allocate the location for the requested data for a miss onto a clean block (the located index at the cache holds invalid data or read-only data whose dirty bit is 0). In doing so, a memory request is made for the missing data, where the valid bit is set to 0, the waiting bit is set to 1, and the tag is also replaced by the tag of the new address. The dirty bit might also be set to 1 if it is a miss for a write request. When a waiting bit is set to 1 on a tag location, data is not valid yet, but a memory request has been made. As soon as a response for the request is received (at MSHR Receiver), the tag will be updated, the valid bit will be set to 1, and the waiting bit will be set to 0. When the tag is set to the waiting position for the miss on a request with a clean block for replacement, the controller goes back to the *idle* state and outputs the miss signal for the Request Arbiter. If data is not found in cache but in Prefetch Buffer, and the block to be replaced is clean, the data request is not missed. Instead, we can move it from Prefetch Buffer to Cache Table and give a hit signal while transitioning back to the *idle* state. For read-through and prefetch requests, the data is also checked in the cache. If the data is found in the cache, no memory request is required, and a hit response is given to the Request Arbiter. If a read-through or prefetch data request is missed in the cache, the controller responds with a miss without updating any field in the tag or replacing any data in the cache table.

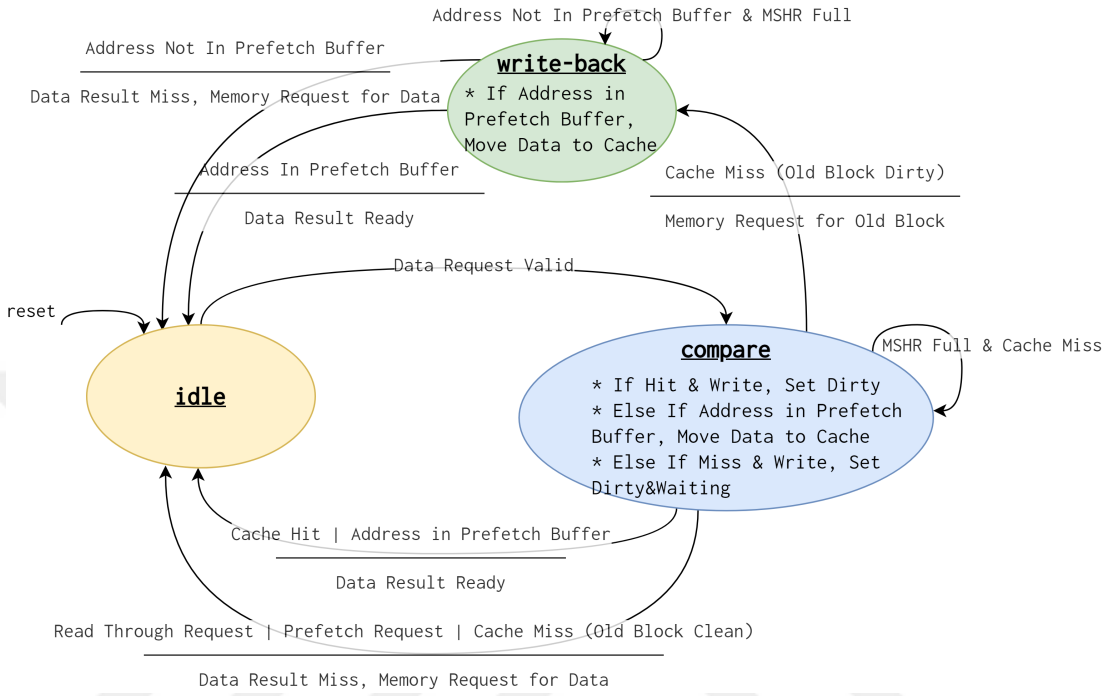


Figure 5.10: Non Blocking Cache Controller FSM State diagram.

We need to write the old block onto memory for a miss with a dirty old block before replacing the data with a new cache line. If that is the case, the controller issues a write memory request for the old block and transitions to the *write-back* state from the *compare* state. At the *write-back* state, the data is requested from the memory, and a miss signal is given to the Request Arbiter. Note that, the tag is already updated in the *compare* state. Also, note that, while in the *write-back* state, the requested address is checked again in Prefetch Buffer. This is done because the data might come at the same time the controller transitions to the *write-back* state. After making the memory request or finding the requested data in the Prefetch Buffer, the controller transitions back to the *idle* state.

Before making a data request, MSHR is also checked for capacity as we cannot make a memory request if the MSHR table cannot hold any more entries. The controller might stall the execution at *compare* and *write-back* states if MSHR Buffer is full.

MSHR Receiver is a sub-component responsible for listening to the memory

responses coming from MSHR Controller. It receives not only memory data but also information related to MSHR entries. Since memory responses are directed to result queues in the Request Arbiter through the non-blocking cache, the MSHR information related to the memory response is also directed. The *mshr_output* signal includes the MSHR entry, memory block address, 1-bit values for read-write, read-through, and prefetch request types, 128-bit dirty data, and 4-bit dirty data mask associated with the memory response. On the other hand, the *write_mshr_entry* is the MSHR entry for a request from the Request Arbiter. This value is wrapped in a *data_res* signal so that the Request Arbiter can store the MSHR entry value in its result queues. 4-bit *dirty_data_mask* and 1-bit *data_exists* signals are related to write misses and explained below. 1-bit *mshr_full* signal is to check if the MSHR table is full.

When memory responds as a valid 128-bit cache-line, the block address associated with this response is given in *mshr_output* signal by the MSHR Controller. This block address is used to locate the index at the cache table and check if the tag value is the same as the one on the cache table. We could have a miss for a request, where the to-be-replaced block's waiting bit is 1 and its dirty bit is 0. We can replace it without waiting for the memory response for the old block for such a case. Hence, when a memory response is received, if the tag value at the located index is still the same, we put the data in the cache table; otherwise, we ignore the result in the cache (memory response is directed to the Request Arbiter in any case). Memory data is not directly written onto the cache but an additional modification on data might be done if we had a write-miss for the received memory response. On a write-miss, the write data is written to a field (*dirty_data*) in the MSHR Table at the MSHR entry for the miss. Because the *dirty_data* is 128-bit long and write-data is 32-bit long, we should locate the position of write-data at the *dirty_data* field. For this, we use the *dirty_data_mask* field in the MSHR Table. It is a 4-bit field that indicates which positions of *dirty_data* are dirty and should be replaced in an upcoming memory response. For example, if the *dirty_data_mask* is set to "0110", the bits from 32 to 63 and the bits from 64 to 95 in the 128-bit memory data should be replaced by the same bits in the *dirty_data* but bits from 0 to 31 and bits from 96 to 127 in the memory

data are clean. This process is also done at result queues of Request Arbiter for dirty misses.

Note that there is a separate 4-bit *dirty_data_mask* signal coming from MSHR Controller to Non-blocking Cache along with 1-bit *data_exists* signal. Even though the non-blocking cache allows multiple write-misses to the same memory block, it does not allow if it is a write-miss to the same offset at the same memory address. To control this, Non-blocking Cache needs a *data_exists* signal that is 1 if there is an outstanding miss for the current memory address. If it is 1 and it is a write-miss, the *dirty_data_mask* associated with the corresponding MSHR entry of the missing data is controlled for the offset value of the current write-miss. If the bit value for the offset at the *dirty_data_mask* is 0, write-miss can be processed, and dirty data is written to MSHR Controller. Otherwise, the non-blocking cache cannot write onto the exact location before the memory response comes. Thus, the processor will stall in such a case.

The MSHR Receiver writes the memory responses for prefetch requests onto the Prefetch Buffer. The address and 128-bit memory block are written to the Prefetch Buffer in a circular FIFO manner (see Figure 5.2); the valid bit is also set to 1 when appending it. The MSHR Receiver overwrites the old values in the Prefetch Buffer on an upcoming prefetch response. When the controller checks for the data in the Prefetch Buffer, it makes an associative check on the address values found in the Prefetch Buffer. The valid bit is set to 0 if data is moved from the Prefetch Buffer to the cache table.

5.5 MSHR Controller

MSHR Controller is the actual component that makes the memory requests. It stores an MSHR Table that holds Miss Status Holding Registers as proposed in [49]. For each memory request issued, an entry at MSHR Table is stored.

These entries include block address, single bit fields to indicate write-read, pending (waiting for memory response), read-through, and prefetch requests (see Figure 5.2. Dirty Data Mask and Dirty Data are also fields in MSHR Table whose functionalities are explained in the previous sub-section. MSHR Table is implemented as a FIFO Buffer, as it expects the responses from the memory to arrive in the order that requests are made.

It receives the memory request from Non-blocking Cache. Once it gets a request, an associative check is done on the block addresses of the MSHR Table to see if there is already a pending memory request for the same address. It only issues the memory request if there is no pending entry already in the MSHR Table, and it appends the new entry to the MSHR Table. Once the memory returns the data, the entry at the head of the MSHR Table is returned to Non-blocking Cache and its pending bit is set to 0.

For a data request, if there is an already pending entry in the MSHR table and its read-through or prefetch bits are set to 1, these bits might be overwritten if the new request is a normal request. A read-through request made earlier might turn into a normal request if the CPU also requests for the same address while waiting for it. The same applies to the prefetch requests.

5.6 Implementation Details

The micro-architecture is implemented with RTL design principles. SystemVerilog code is used for the implementation and code is simulated with Xilinx's Vivado Design Suite. Ibex Core developed by lowRISC is used for the core CPU, which is a low-area, low-cost, in-order, and 2-stage pipeline RISC-V CPU implementation [51]. Ibex is also used in previous efforts for extending the RISC-V ISA with the new instructions [6]. The decoding logic of the custom instructions in the core stayed the same in this work. In addition, some of the signals between the components in the design have complex structure types, and are illustrated in Figure 5.11.

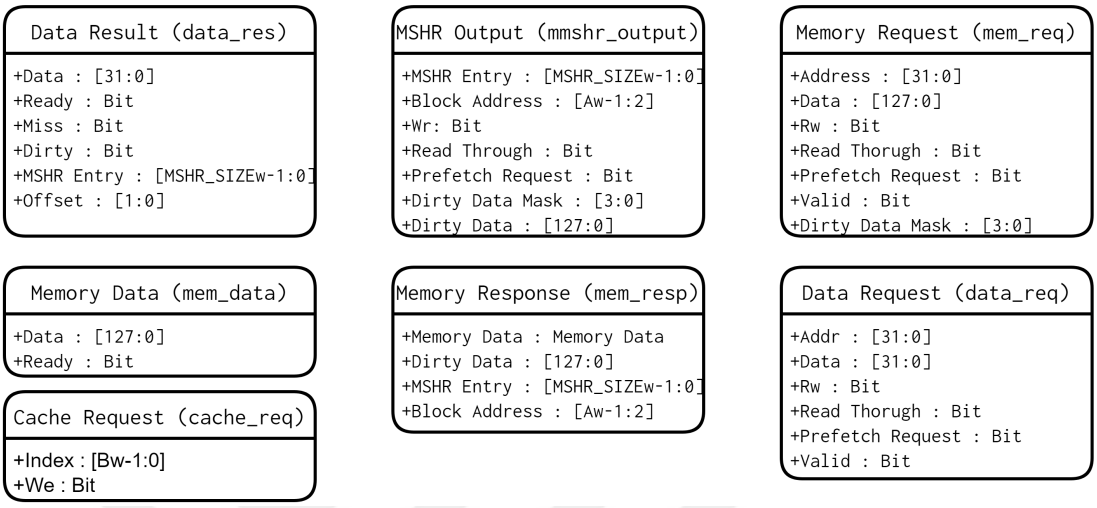


Figure 5.11: Complex data signals between the components in the architecture.

For the correctness of computations in the system, programs are first executed on a PC, and a portion of the memory is dumped in a file. The same program is also executed in the simulation at the RTL level and the memory is also dumped after program termination. Then the two memories are compared and verified to make sure there is no difference between the two memory portions. The summary of the verification is given in Figure 5.12. Because we could not directly use the existing benchmarks for graph applications in our work, we implemented many graph benchmarks in C and C++. We compared them against existing implementations for correctness.

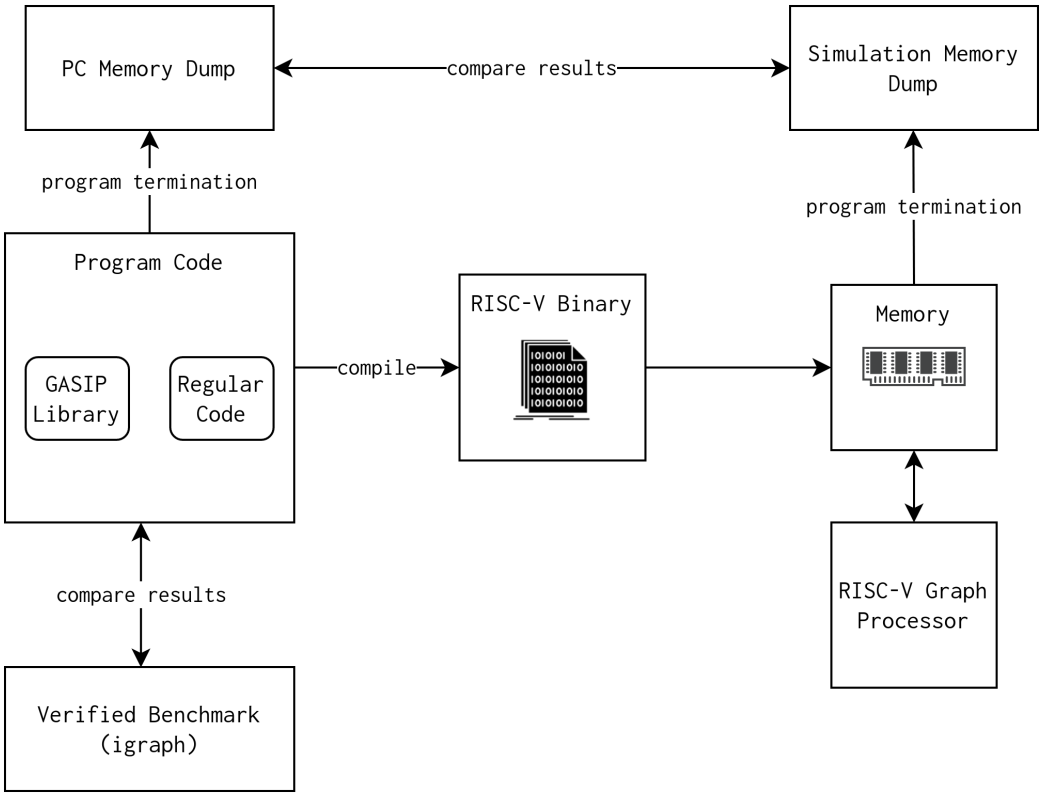


Figure 5.12: System verification summary.

Chapter 6

Software Support

As we extended the RISC-V ISA with custom instructions, utilizing these new instructions requires software support. In the previous work, inline assembly calls inside programs are employed to use these instructions [6]. In this work, software support for the custom architecture is provided through compiler and library in addition to inline assembly calls.

6.1 Compiler Support

For the compiler support, LLVM Compiler Framework [52] is used. At the most basic level, the new instructions are defined for the RISC-V target, and intrinsic (built-in) functions are defined for these new instructions. A programmer can employ the intrinsic calls to use the new instructions. Yet, a higher-level abstraction is also proposed for the new instructions. In this abstraction, the programmer is only responsible for marking arrays related to graph data and additional variable markings for *memspm* and *delspm* signals. After marking the special variables in a graph application, a compiler optimization pass is performed on the program. The optimization pass inserts the new instructions in the proper places. An example is shown for Breadth-First Search (BFS) given in Algorithm 3.

Algorithm 3: Breadth-First Search (BFS) algorithm with special markings before applying the compiler Pass.

Input: G=(*rows,columns,data,values*), sourceVertex

```
1  mark G
2  mark memspm_index, delspm_index
3  Initialize Q as a queue
4  Initialize data as an array of tuples (state, level, parent)
5  data[:] = (unvisited, ∞, −1)
6  data[sourceVertex] = (visited, 0, −1)
7  ENQUEUE(Q,sourceVertex)
8  while Q ≠ ∅ do
9      u = DEQUEUE(Q)
       // Put the neighbors of the next head in queue
10     memspm_index = HEAD(Q)
11     for j ← columns[rows[u]] to columns[rows[u + 1]] do
12         if data[j] is unvisited then
13             data[j] ← (visited, data[u].level + 1, u)
14             ENQUEUE(Q, j)
15         end
16     end
17     delspm_index = rows[u]   // Remove the neighbors of vertex u from SPM
18 end
```

Once the CSR arrays of the graph are marked, proper *spmcon* instructions are called. By default *columns* array and *data* arrays are chosen for the configuration. When we access an element of a *columns* array using the array subscript provided by the programming language, *spmreg* instruction is used instead of a load instruction. Different arrays can be marked depending on the frequency of usage in the algorithm. For example, *values* array (generally holding *weight* values) may also be used for configuration instead of *columns* array if we are to access weights of the edges more than the indices of edges.

In addition to marking the arrays associated with graph data, two integer

variables might also be marked for *memspm* and *delspm* instructions. For *memspm*, an index of the *rows* array is assigned in the program to indicate inserting *memspm* instructions. Similarly, an index of the *columns* array is assigned in the program to indicate inserting *delspm* instructions. We should note that markings for the *memspm* and *delspm* instructions are optional since *spmreg* by itself can make implicit *memspm* requests as described in Chapter 5. Also, further work on compiler pass might enable automated detection of these variables without programmer's intervention.

The compiler optimization pass runs on the LLVM's Intermediate Representation (IR). We run three optimization passes of LLVM before running our custom optimization pass to ease our implementation, which are *mem2reg*[1], *loop-rotate*[2], and *instcombine*[3]. After these optimization passes and our custom optimization pass, we finally apply LLVM's *-O3* level optimizations. Even though we also worked on loop analysis functionalities, they are not included in the final design. Any programming language that can be translated into LLVM IR can benefit from the custom optimization pass. We used C and C++ programming languages for our experiments. For marking the variables, the ***__attribute__*** keyword is used to add attributes to variables. Setting the *annotate* attribute for the variables is done as in the following syntax:

```
1 int main(int argc, char **argv){
2     ...
3
4     __attribute__((annotate("riscv_edges"))) uint32_t *columns =
    ↪ new uint32_t[EDGE_SIZE];
5     __attribute__((annotate("riscv_versp-data"))) graph_data_t *
    ↪ data = new graph_data_t[VERTEX_SIZE];
6
7     ...
8 }
```

The compiler pass adds a layer of abstraction to the low-level architecture

[1]https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register
[2]https://llvm.org/docs/Passes.html#loop-rotate-rotate-loops
[3]https://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions

that executes the program. However, the programmer still needs to give hints for it to utilize the architecture fully. In this sense, the new instructions are not added automatically to a program without any hints. Using only the markings for the graph data inserts only the *spmcon* and *spmreg* instructions, which still helps to have performance improvements on graph applications. Yet, a more advanced user needs to know about lower architecture for using the markings for the *memspm* and *delspm* indices. In the next section, we present another abstraction layer that a programmer does not need to know about the underlying architecture.

6.2 Library Support

In this section, we describe a library designed for graph applications. The library is implemented so that it utilizes our custom architecture without the user knowing anything. Like the CUDA library in programming Nvidia GPUs or the TensorFlow library in programming Google's TPUs, our library is used for programming our custom architecture. Our primary motivation in building the library is to abstract out the details of the underlying architecture, and have predictable and consistent application performance, easily map the graph applications into further parallel architectures based on our single-core architecture.

We adopt the Gather-Apply-Scatter (GAS) paradigm of PowerGraph [45] for our library. The semantics of the graph applications used in our library are given in Algorithm 4, slightly different from the PowerGraph. GATHER, APPLY, and SCATTER functions are defined by the user, and additional SUM and initialization functions are included for active-vertex-list and accumulated-vertex-list. Programmers can choose which edges should be processed on gather/scatter phases (incoming edges, outgoing edges, or both). They can also set if all vertices should be considered active or not until the computations are converged.

Algorithm 4: Gather-Apply-Scatter (GAS) semantics of graph algorithms.

Input: G=(V,E)

Initialize active-vertex-list
Initialize accumulated-vertex-list
while *not converged* **do**
    // Gather Phase
    for *each vertex* $v \in V$ *in active-vertex-list* **do**
        for *each edge* $e = (v,u) \in E$ *or* $(u,v) \in E$ **do**
            accumulated-vertex-list[v] $\leftarrow$ SUM(accumulated-vertex-list[v], GATHER(v,e))
        end
    end
    // Apply Phase
    for *each vertex* $v \in V$ *in active-vertex-list* **do**
        accumulated-vertex-list[v] $\leftarrow$ APPLY(v, accumulated-vertex-list[v])
    end
    // Scatter Phase
    for *each vertex* $v \in V$ **do**
        for *each edge* $e = (v,u) \in E$ *or* $(u,v) \in E$ **do**
            active-vertex-list[v] $\leftarrow$active-vertex-list[v] | SCATTER(v,e)
        end
    end
end

The library is implemented in object-oriented style, in C++. The user defines a vertex program by inheriting the abstract *VertexProgram* class. Furthermore, the programmer specifies the required `gather`, `apply`, and `scatter` methods, while the backend of the library inserts the custom instructions in proper places to improve the performance. UML class diagram of the library is given in Figure 6.1. VertexProgram is an abstract class that contains a *GASEngine* instance. GASEngine is the main class that runs the Gather-Apply-Scatter semantics of the graph algorithms. Graph data is set through GASEngine, uses CSR and CSC

representation. CSR representation is used to process outgoing edges of vertices, whereas CSC is used for incoming edges of vertices. In setting the graph data, GASEngine allocates an array of *Vertex* instances. On the other hand, the *Edge* class is used as an abstraction for VertexProgram's methods. Only one instance of Edge class is created at the *run* method of GASEngine with proper pointers to the inner CSR/CSC data arrays for saving memory space. Vertex instances from the *vertices* array and a single Edge instance are passed to the functions of VertexProgram during the execution.

We assumed that input for the program is given as pre-processed CSR and CSC arrays. Whenever we access the edge indices using *getOutNeighbor*, *getInNeighbor* methods of GASEngine, proper *spmreg* instructions are inserted instead of simple loads. Similarly, *weight* arrays of CSR and CSC representations are also put in ESP, hence *getWeight* method of Edge class also employs *spmreg* instruction. While executing the run method of GASEngine, the semantics given in Algorithm 4 is followed. Whenever, we process CSR array for outgoing edges, proper *spmcon* instructions are inserted. When the execution requires processing of incoming edges, *spmcon* instructions are inserted again for CSC array. *vertices* array is used as the *data* array of the CSR/CSC representations so that the elements are prefetched during the execution. *memspm* and *delspm* signals are also inserted by exploiting the active-vertex-list of the current iteration.

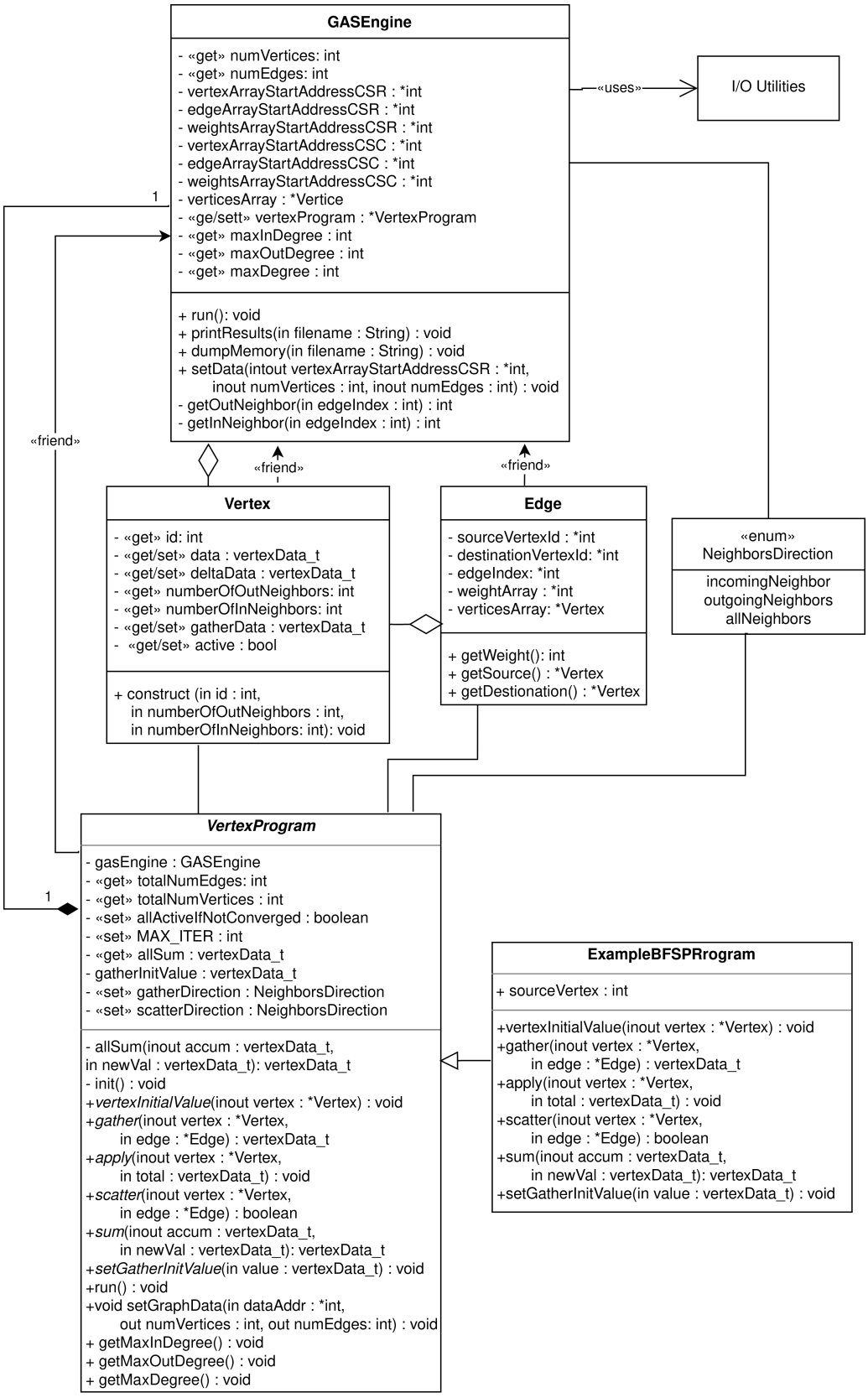


Figure 6.1: UML class diagram of GAS library.

An example program for breadth-first search (BFS) algorithm is given in Listing 6.1. Note that the default directions for gather and scatter are incoming and outgoing, respectively.

```
/*
 * bfs.cpp
 */

#include <gasip_interface.h>
#include <util.h>

using vertexData_t = int32_t;

class BFSExampleProgram: public VertexProgram {
public:
  void vertexInitialValue(Vertex *vertex) {
      if (vertex->getId() == sourceVertex) { //Source vertex
        vertex->setData(0);
      } else {
        vertex->setData(INT32_MAX); //To represent infinity
      }
  }

  vertexData_t gather(Vertex *vertex, Edge *inComingEdge) {
      if (inComingEdge->getSource()->getData() < INT32_MAX
          && inComingEdge->getSource()->getData() + 1 < vertex->
   ↪ getData()) {
        return inComingEdge->getSource()->getData() + 1;
      } else {
        return vertex->getData();
      }
  }

  void apply(Vertex *vertex, const vertexData_t &total) {
      if (total < vertex->getData()) {
        vertex->setData(total);
      }
  }

  bool scatter(Vertex *vertex, Edge *outGoingEdge) {
```

```
36        if (outGoingEdge->getDestination()->getData() == INT32_MAX
37            && vertex->getData() != INT32_MAX) {
38          return true;
39        }
40        return false;
41    }
42
43    //Choose the min
44    vertexData_t sum(const vertexData_t &accum, const vertexData_t
      ↪ &newVal) {
45        if (newVal < accum) {
46          return newVal;
47        } else {
48          return accum;
49        }
50      }
51  private:
52    uint32_t sourceVertex = 0;
53  };
54
55  extern "C" int main(int argc, char **argv) {
56    uint32_t numVertices, numEdges, *dataStartAddress;
57
58    numVertices = 1024;
59    numEdges = 17698;
60
61    char inputFile[] = "./example/data/graph_1024v_17698e.txt";
62    int fileReadSucceded = read_file(inputFile, &numVertices, &
      ↪ numEdges, &dataStartAddress);
63    if (fileReadSucceded) {
64      VertexProgram myVertexProgram;
65      myVertexProgram.setSourceVertex(67);
66      myVertexProgram.setGatherInitValue(INT32_MAX); //INT32_MAX to
      ↪  represent infinity
67
68      myVertexProgram.setGraphData(dataStartAddress, numVertices,
      ↪ numEdges);
69      myVertexProgram.run();
70    }
71
```

```
72    return 0;
73 }
```

Listing 6.1: GAS library example implementation for Breadth-First Search (BFS) algorithm.

Chapter 7

Evaluation

We evaluated the architecture for its run-time performance in various graph applications. We first executed the benchmarks on IBEX Core [51] without using any custom instructions; thus ESP Controller remained inactive. Then, we also tested the same applications with default parameters, but this time by adding the custom instructions to utilize the complete implementation. We recorded the running times of the benchmarks for speedup measurements using the simulation tool of Xilinx's Vivado Design Suite 2020.2 [53]. We also evaluated the architectural parameters such as SPM size, prefetch-buffer size, MSHR-size, etc., to see their effects on speedup. In addition to system evaluation, the Gather-Apply-Scatter library is evaluated for its utilization of the custom instructions. Lastly, we evaluated the compiler optimization pass to verify that the insertions of custom instructions by the compiler optimization pass have similar performance if they were inserted manually.

7.1 Benchmarks

We implemented six benchmarks commonly used in related work as computation kernels of many graph applications. These benchmarks and their properties are

explained below.

- **Breadth-First Search (BFS)**: Breadth-First Search algorithm is a well-known traversal-centric algorithm used to find the edge-distances and parent vertices on a path to a given source vertex in an unweighted directed graph. We used the textbook implementation for our experiments [54]. The insertions of the custom instructions are done similar to the previous work with slight modifications [6].

- **PageRank (PR)**: PageRank algorithm [55] is proposed to find the importance of web pages given the topological link network structures within the pages. It is not only a basis for Google's web search engine but also a graph kernel that applies to many other applications. Even though there are many algorithm versions, we used iterative implementation in our experiments. The insertions of the custom instructions are done similarly to the previous work [6].

- **Single-Source Shortest Path (SSSP)**: Single-Source Shortest Path algorithm is similar to the BFS algorithm, but edge weights are considered for finding paths to a given source vertex. We used two implementations of SSSP, where the first one uses the Dijkstra's algorithm (**SSSP-D**) [56] with a Priority Queue using Binary Heap. The second implementation (**SSSP-BF**) uses Bellman-Ford algorithm [57, 58] as an iterative solution. In inserting the custom instructions, elements of the *weight* array are accessed with *spmreg* instructions for both SSSP-D and SSSP-BF. In SSSP-D, *memspm* and *delspm* instructions are not used since the priority-queue (considered as a work-list) is subject to change during computation. Yet, *spmreg* alone can utilize the SPM as shown experimentally. For the SSSP-BF implementation, *memspm* and *delspm* instructions are inserted similar to the PageRank algorithm considering its iterative structure.

- **Depth-First Search (DFS)**: Depth-First Search is another traversal-centric algorithm in that starting from parents, all vertices on a path are traversed, just like in BFS. However, this time paths are found in a depth-first manner. It is a computational kernel that is used in many graph

algorithms. We implemented the textbook recursive solution for our experiments [54]. For the algorithm version with the custom instructions, we used *spmreg* to access the edge array as in the other benchmarks. *memspm* instruction is used at the start of processing the neighbors of a vertex. But as soon as a recursive call is needed, the *memspm* instruction is complemented with *delspm* instruction as the depth of the recursive calls might cause to fill up the SPM.

- **Sparse General Matrix Multiply (SpGEMM)**: We used the sparse matrix-graph duality to implement the matrix multiplication. Since we base our design on CSR representation, we implemented the sequential `CSR-SpGEMM` algorithm presented in [59] using a sparse accumulator. Even though matrix multiplication is not directly a graph algorithm, it is widely used as a primitive in graph algorithms. We multiply the graph with itself in this kernel. For inserting the custom instructions, we used the *weights* array as the base address for SPM and the sparse accumulator's value array as the *data* array of CSR considering the number of accesses to these arrays in the algorithm. *weights* and *columns* arrays are accessed with *spmreg* instruction. *memspm* and *delspm* instructions are used at the outer loop of the algorithm.

Graph Coloring (GC): Graph Coloring is the problem of assigning colors to vertices such that no adjacent vertices share the same colors. As finding an optimal solution (using the minimum number of unique colors) is an NP-complete problem, we followed the greedy graph coloring approach. In this implementation, the number of colors is guaranteed to be less than the maximum degree of a given graph. Because we need both incoming edge and outgoing edge information in this algorithm, we use the CSC format in addition to the CSR format of the input graphs. Since we iteratively assign colors to vertices, *memspm* and *delspm* instructions are used effectively. We used a set implementation similar to the sparse accumulator used in the SpGEMM algorithm to keep the colors of neighboring vertices of a vertex in each iteration. The set keeps an array to hold color values of the neighboring vertices, and the array is used as the *data* array for configuring with *spmcon*

and *columns* array of CSR representation. We try exchanging CSC and CSR representations during the iterations. But due to the additional number of *spmcon* instructions, swapping the configuration within an iteration is inefficient.

We implemented Breadth-First Search (BFS), PageRank (PR), Single-Source Shortest Path (SSSP), and Graph Coloring (GC) in our Gather-Apply-Scatter library. In contrast, the other benchmarks are not suitable for the Gather-Apply-Scatter approach. Implementations of these algorithms are summarized in Tables 7.1 and 7.2. Note that minor differences exist in the actual implementation. For example, we handled spider traps and dead ends in the actual implementation of PR-G. For GC-G, we apply a greedy solution, and we call the *setData* method at the gather stage. Even though this is unusual and might not be valid for some GAS implementations, the algorithm converges with this approach using minimal memory space. In PowerGraph [45], the graph coloring implementation gathers the set of colors of neighboring vertices for each vertex and applies the color for the first available one. It runs only on asynchronous mode of the parallel version, and maintaining a set for each vertex requires significant memory space. Since we currently run our gather-apply-scatter library sequentially, we cannot run in an asynchronous mode and implement the GC-G with the approach described in Tables 7.1 and 7.2.

	gather(vertex, edge)	apply(vertex,total)	scatter(vertex,edge)
PR-G	**return** $\frac{\text{edge.source.data}}{\text{edge.source.OutNeighborsNum}}$	newVal = 0.15+0.85*total vertex.setDelta(|vertex.data-newVal|) vertex.setData(newVal)	**if**vertex.delta >$\mathcal{E}$ **return** true **else** **return** false
BFS-G	**if** edge.source.data + 1 <vertex.data **return** edge.source.data + 1 **else** **return** vertex.data	**if** total <vertex.data vertex.setData(total)	**if** edge.destination.data==∞ & vertex.data$\neq \infty$ **return** true **else** **return** false
SSSP-G	**if** edge.source.data+edge.weight<vertex.data **return** edge.source.data+edge.weight **else** **return** vertex.data	**if** total <vertex.data vertex.setDelta(|vertex.data - total|) vertex.setData(total)	**if**(vertex.delta >0) **return** true **else** **return** false
GC-G	**if** edge.source.data==edge.destination.data vertex.setData((vertex.data+1)% (maxDegree+1))	(Ignore)	**if** edge.source.data==edge.destination.data **return** true **else** **return** false

Table 7.1: GAS benchmarks: Gather, apply, and scatter methods for GAS benchmarks.

	sum(accum,newVal)	vertexInitialValue(vertex)	gatherInitialValue	gatherDirection	scatterDirection
PR-G	**return** accum+newVal	vertex.setData(1/totalNumEdges)	0	incoming edges	outgoing edges
BFS-G	**return** min(acuum,newVal)	**if** vertex.id == sourceVertex vertex.setData(0) **else** vertex.setData(∞)	∞	incoming edges	outgoing edges
SSSP-G	**return** min(acuum,newVal)	**if** vertex.id == sourceVertex vertex.setData(0) **else** vertex.setData(∞)	∞	incoming edges	outgoing edges
GC-G	(Ignore)	vertex.setData(0)	(Ignore)	all edges	all edges

Table 7.2: GAS benchmarks: Sum, vertexInitialValue, gatherInitialValue methods, and gatherDirection-scatterDirection values used in GAS benchmarks.

7.2 Graph Data

We used five graphs with various vertex and edge numbers. Two of these graphs are synthetically generated using Graph500 Kronecker Generator [60]. The other three are real-life graphs taken from SNAP Datasets [61]. Because our hardware simulation times are too long for large graphs, we scaled down the graphs. Table 7.3 summarizes the graph data we used in this study. For algorithms with required weight values, we generated random weight values as integers between

1 and 100.

	Number of Vertices	Number of Edges	Abbreviation
Graph1024	1024	17698	G1
CollegeMsg [62]	1900	20296	CM
Email-Eu-Core [63]	1005	25571	EC
Facebook [64]	4039	88234	FB
Graph4096	4096	129890	G4

Table 7.3: Input graphs used in our experiments.

7.3 Default Parameters

Because our micro-architecture depends on many parameters, we evaluated the important ones and observed their effects on performance. An essential goal in doing so is to minimize the cost of the components in the design. The critical parameters we observed are as follows:

- **Memory Size** is the size of the external memory that has high latency compared to RAM. We keep the size of memory constant at 128 MB.
- **Cache Size** is the size of the cache. Since we are using a single-level cache with small graphs compared to a real-life setting, we apply scaling in the cache size as well. Accordingly, the cache size is scaled according to the ratio of the number of vertices in graphs to LLC cache sizes in related work [33, 38, 34, 22, 39]. The default cache size used in our study is 1KB.
- **Cache Line Size** is the number of bits that each cache block holds. We also kept this parameter constant as 128 bits (4 words). This is also the bus-width of the external memory. That is, when we make a memory request, we receive 128 bits of data.
- **SPM Block Size** is the number of words in a chunk in the SPM. This parameter determines the number of block requests needed to be done for

memspm instruction requests. For example, if SPM block size is set to 32 words, as each memory block holds 4 words, we need to make $32/4 = 8$ block requests. This parameter is evaluated in our experiments.

- **SPM Capacity** is the number of chunks in the SPM which is also evaluated in our experiments. Note that (SPM Block Size)×(SPM Capacity) gives the number of words that SPM holds.

- **SPM Read Buffer Size** is the size of the Read Buffer at the ESP Controller. We kept this parameter constant at 8 requests, large enough to not cause any buffer-overflow in our experiments.

- **ESP Queue Size** is the size of ESP Result Queue at Request Arbiter. We set this parameter large enough to not cause any stall at ESP Controller's data requests. Its value is determined relative to the SPM block size.

- **CPU Queue Size** is the size of CPU Result Queue at Request Arbiter. Since our CPU is in-order and can only make at most two requests for outstanding misses (occasionally on misaligned accesses[1]), we kept this parameter constant at 4 queue values, which is more than enough for our experiments.

- **Prefetch Buffer Size** is the size of prefetch buffer in Non-blocking Cache. This parameter is evaluated in our experiments.

- **MSHR Table Size** is the size of the MSHR table (number of rows) that holds the outstanding misses. We experimentally evaluated the effects of MSHR size.

- **Transportation Delay** is the latency of external memory requests in the number of clock cycles. This parameter is also evaluated. In our architecture, cache access time for data (on a hit) is 5 clock cycles. On a miss, response time varies but depends mostly on the transportation delay parameter.

[1] https://ibex-core.readthedocs.io/en/latest/03_reference/load_store_unit.html

The default parameter values are given in Table 7.4. These default values are used in performance evaluations with the minimal-cost settings. When some parameters are varied for the architectural parameter evaluations, the others remained in default values.

Parameter	Value
Memory Size	128 MB
Cache Size	1 KB
Cache Line Size	128 bits
SPM Block Size	16 words
SPM Capacity	4 chunks
SPM Read Buffer Size	8 requests
ESP Queue Size	16 requests
CPU Queue Size	4 requests
Prefetch Buffer Size	2 blocks
MSHR Table Size	8 MSHRs
Transportation Delay	100 clock cycles

Table 7.4: Default parameter values.

7.4 Performance Evaluation

We set the default architectural parameters for our system as explained in Chapter 7.3 for the performance evaluations. We first present the performance results for the benchmarks in Figure 7.1. As can be seen from this figure, traversal-centric graph algorithms like BFS, DFS, SSSP-BF, and GC have higher speedup when compared to PR and SpGEMM algorithms. These traversal-centric algorithms benefit more from the architecture when the program has more memory access to edge data. Even though SSSP-D is a traversal-centric algorithm like SSSP-BF, it does not exhibit a similar pattern as SSSP-BF. One reason for this is that computations of SSSP-D are not only done over graph data structure but also on a priority queue. Thus, the SSSP-D benchmark includes a significant portion

of non-graph-related computations. PR benchmark shows mostly stable performance improvements of around 25% due to its iterative and compute-intensive structure. Even though we access edges and weight values of the input graph using *spmreg* instruction for the SpGEMM benchmark, a significant portion of misses occurs due to non-graph computation, such as the sparse accumulator. Therefore, we have less speedup compared to traversal-centric algorithms. The average performance of the tested benchmarks is between 27% and 34% for various graph data. On the other hand, for individual benchmarks, the least speedup is observed for SSSP-D with an average 10% improvement in performance, whereas the highest is observed for BFS with an average 49% improvement. We also note that we observed a 73% performance gain on EC data for the BFS benchmark when an SPM capacity of 4 and SPM block size of 64 are used.

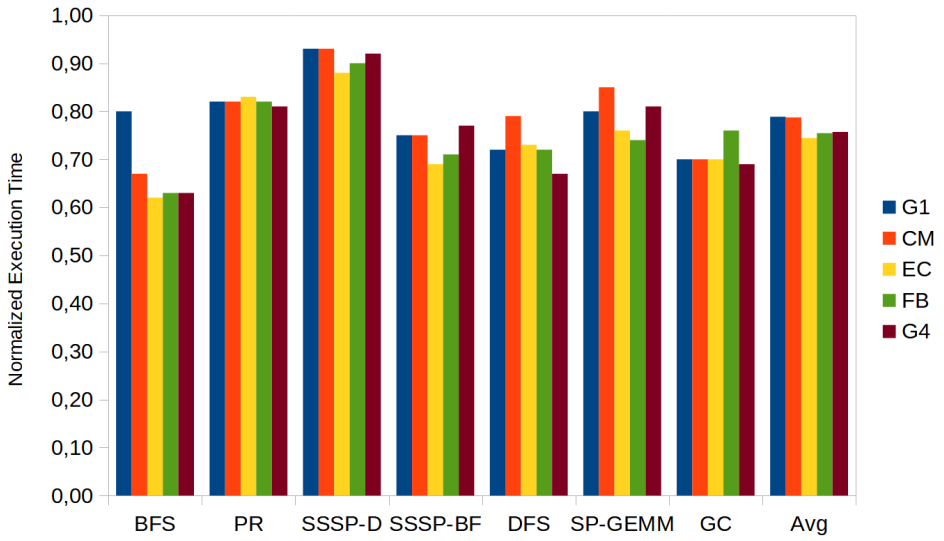


Figure 7.1: The normalized execution times for different data sets.

Similar evaluations for the GAS library are shown in Figure 7.2. Note that we could not implement all algorithms in the Gather-Apply-Scatter paradigm; thus we do not have the DFS and the SpGEMM benchmarks for the GAS library. We have a single implementation for the SSSP algorithm for the same reason. BFS-G has the best results, whereas SSSP-G has the worst. The speedups are overall more stable and depend less on the data for the GAS library. On the

other hand, benchmark-wise speedups differ from 17% (SSSP-G on FB) to 60% (BFS-G on G4). Individual benchmark performance averaged on different graph data varies between 22% and 51%. BFS-G, PR-G, and GC-G benchmarks show similar behavior. On the other hand, the SSSP-G benchmark shows a slight speedup when compared with the SSSP-BF benchmark. One reason is that we access elements of the weights array with *spmreg* instruction. This causes under-utilization of prefetch buffer as random *data* array values are prefetched that are pointed by weight value (which is not meaningful). We discuss the prefetch utilization in the next section with a possible solution. The average speedup of benchmarks for the GAS library is between 32% and 43% considering the different graph data.

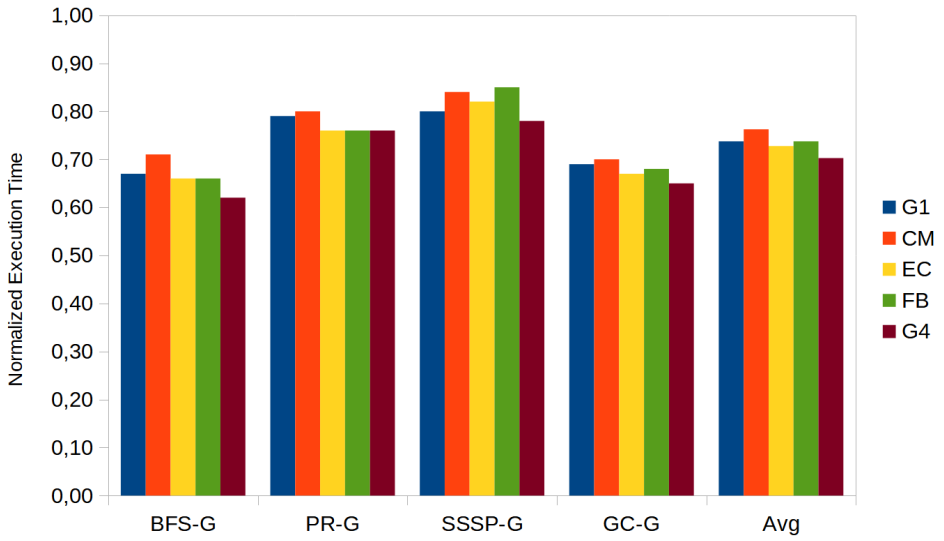


Figure 7.2: The normalized execution times for different data sets using the GAS library.

7.5 Architectural Parameters Evaluation

We used the BFS benchmark for the architectural parameter evaluations. However, all benchmarks show similar results with the tested parameters. The first evaluation is done for SPM block size and SPM capacity. We keep the prefetch

buffer Size, MSHR size, and transportation delay constant at 16, 16, and 100, respectively for this evaluation. Figure 7.3 shows the evaluation results for changing the SPM block size parameter, for an SPM capacity of 2. Figures 7.4 and 7.5 repeat the experiment with SPM capacity of 4 and 8, respectively.

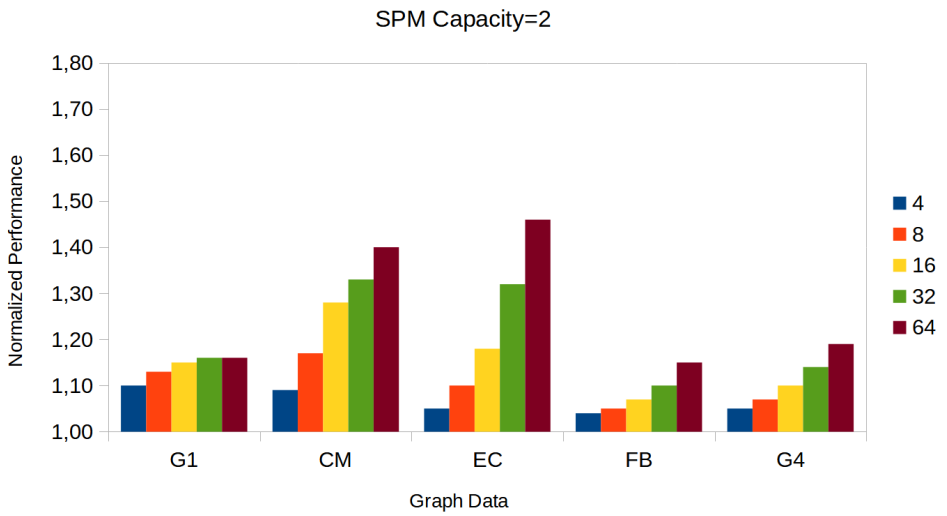


Figure 7.3: The normalized performance with varying SPM block size, with an SPM capacity of 2.

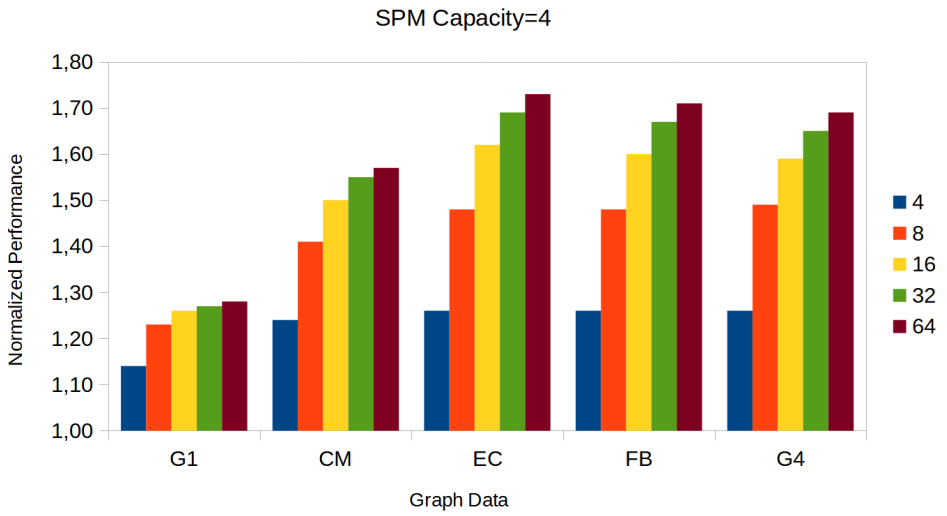


Figure 7.4: The normalized performance with varying SPM block size, with an SPM capacity of 4.

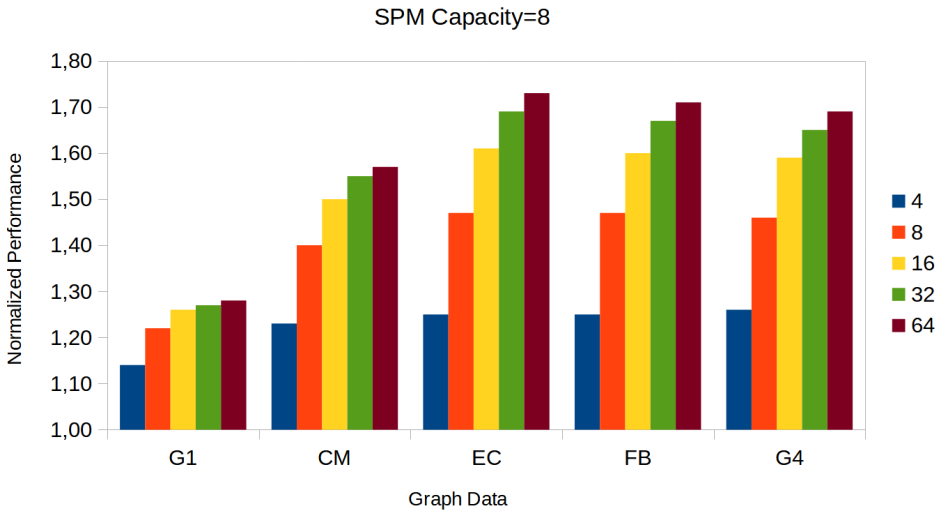


Figure 7.5: The normalized performance with varying SPM block size, with an SPM capacity of 8.

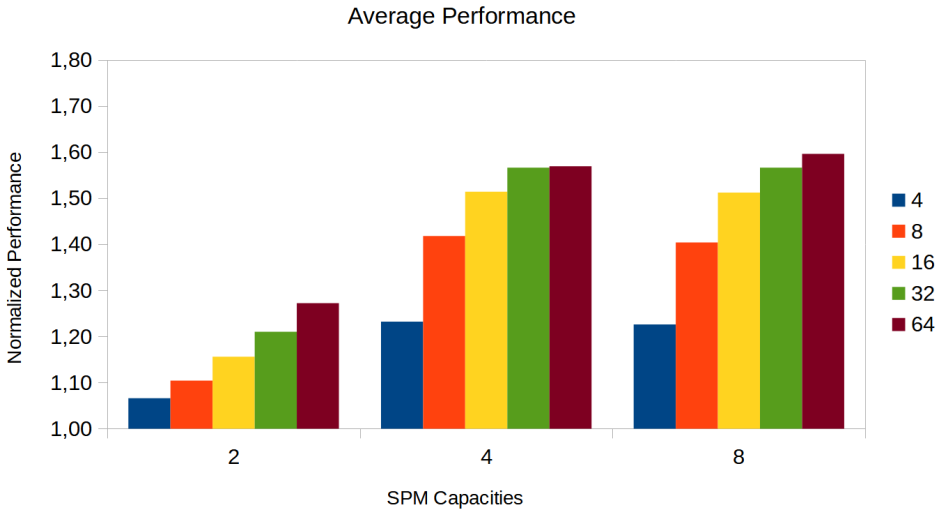


Figure 7.6: The average normalized performance with varying SPM block size, with all SPM capacity values.

We first kept the SPM capacity at 2 chunks and varied the SPM block sizes in Figure 7.3. We see that increasing the SPM block size continuously increases the speedup. Doing the same experiment for setting the SPM capacity at 4 and

8, we see that the impact of increasing the SPM block size diminishes after an SPM block size of 16, as can be seen in Figures 7.4 and 7.5. Moreover, we see that when SPM capacity is set to 4 and 8, we almost always get speedups greater, neutralizing the effect of SPM block size with an SPM capacity of 2 (Figure 7.6). However, there is not much difference between the setting SPM capacity to 4 and 8, setting it to 8 might even worsen the performance slightly. This is most likely due to additional memory requests with added capacity as *memspm* instructions are ignored when no space is found in SPM. Another observation on SPM block size with SPM capacity 4 and 8 is that, the effect of increasing the SPM block size decreases for larger values. Thus, we do not need to set a large value for SPM block size as most vertices have a small number of neighboring vertices considering Power Law distribution [65]. Yet, when a vertex has many neighbors, accessing the edges with *spmreg* instruction will cause the ESP Controller to make additional direct *memspm* requests, thus the SPM will still be utilized. Utilization SPM (ratio of immediate core hits for *spmreg* instructions to total number of *spmreg* instruction calls) is summarized in Figure 7.7, 7.8, 7.9, and 7.10. We see from those figures that, SPM is highly underutilized for an SPM capacity of 2 unless the SPM block size is very large. However, if SPM capacity is at least 4, the utilization is above 90% when ESP block size is set to 16 and above.

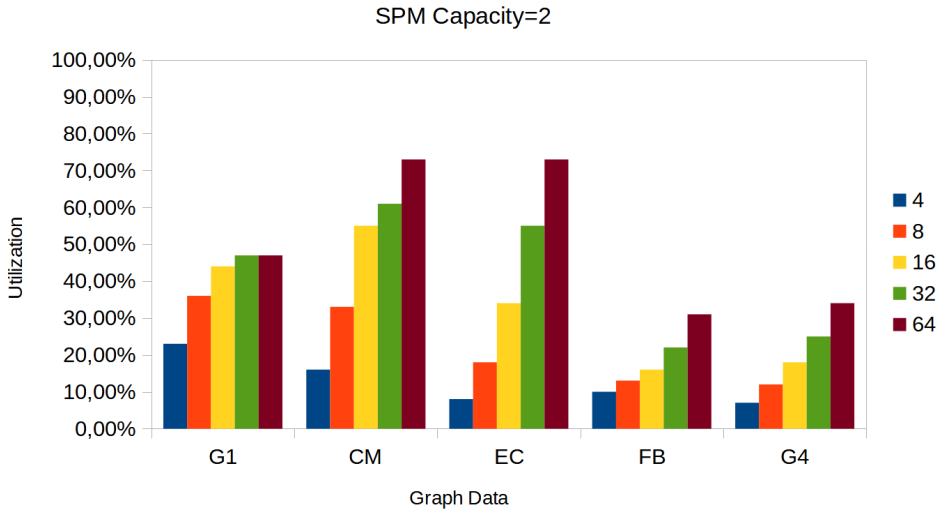


Figure 7.7: SPM utilization with varying SPM block sizes fo an SPM capacity of 2.

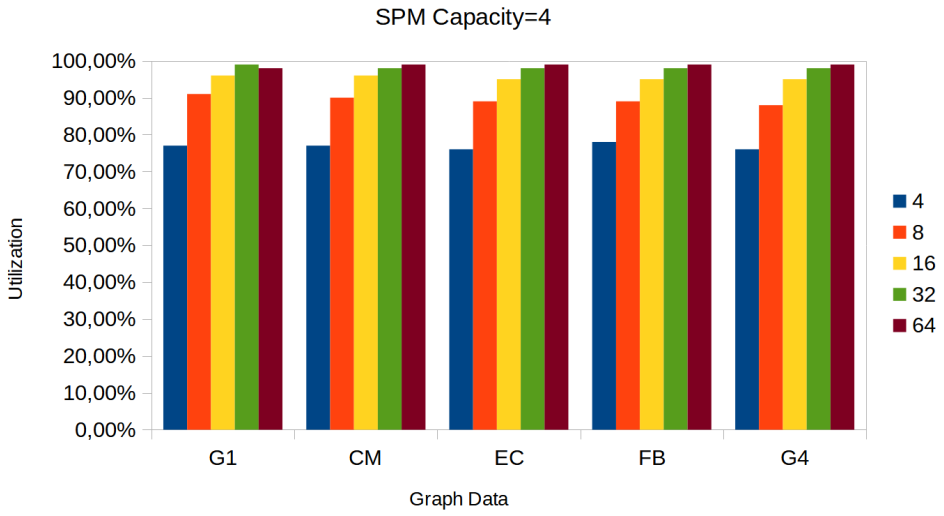


Figure 7.8: SPM utilization with varying SPM block sizes fo an SPM capacity of 4.

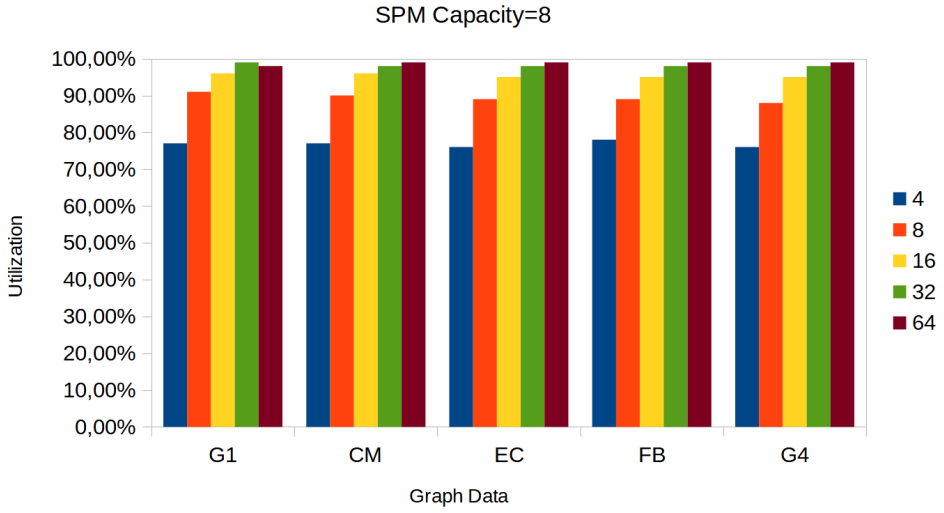


Figure 7.9: SPM utilization with varying SPM block sizes fo an SPM capacity of 8.

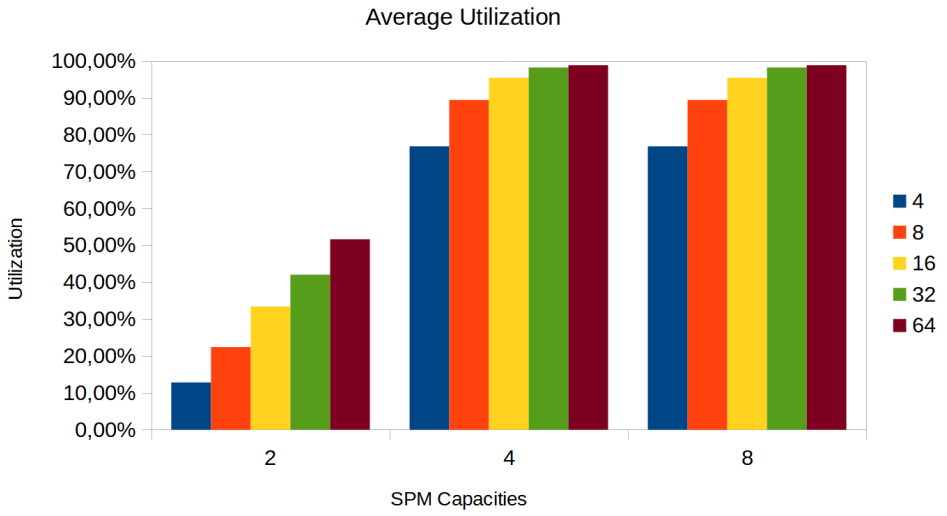


Figure 7.10: Summary of SPM capacity and SPM block size evaluations for SPM utilization.

Based on the empirical evaluation of varying SPM capacity and SPM block size parameters, we conclude that setting SPM capacity to 4 is sufficient for performance and is minimal for the cost. Similarly, setting SPM block size to 16 is sufficient as making it larger will have little impact on the performance. We next show the effect of prefetch buffer Size on performance. The evaluation is illustrated in Figure 7.11. As can be seen from this figure, having a prefetch buffer of as little as 2 blocks is sufficient for our settings as having a greater buffer size has very little effect on the performance. We fixed this size value for our overall performance evaluations.

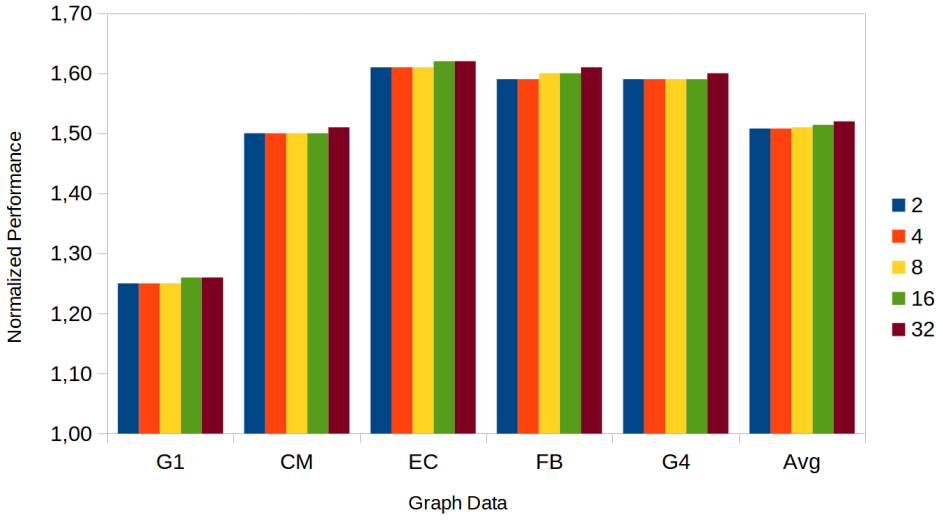


Figure 7.11: Performance for varying prefetch buffer size.

Our next evaluation is to see the effect of MSHR table size. The evaluation of MSHR table size is illustrated in Figure 7.12. We see an increase in performance up to MSHR table size of 8 but remains almost constant after that. Thus, we fixed the MSHR table size to 8 for our overall performance evaluations.

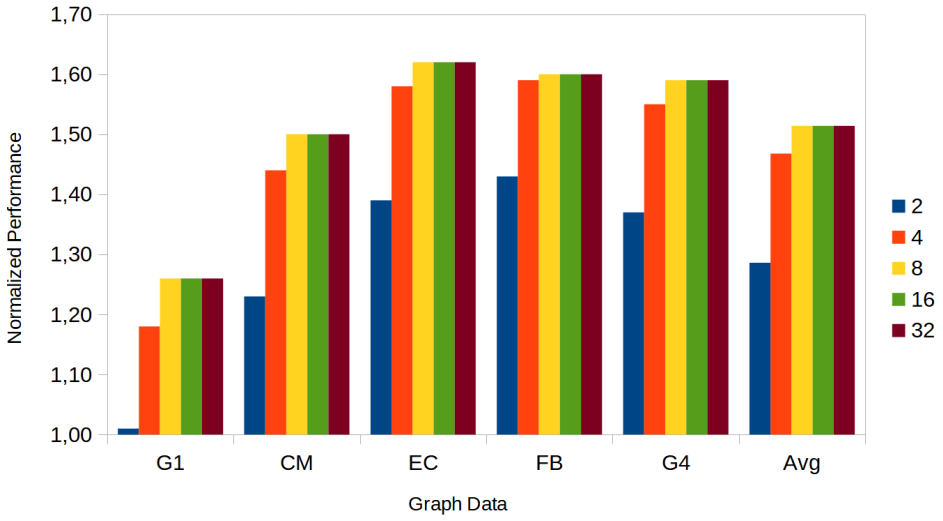


Figure 7.12: Performance for varying MSHR table size.

The last architectural parameter we evaluate is the transportation delay. For this evaluation, we set the SPM capacity to 4, SPM block size to 16, and prefetch buffer and MSHR table sizes to 16 to prevent any potential bottleneck, even though small sizes are found to be enough. The evaluation is illustrated in Figure 7.13. We do not get a significant speedup even if the when access to the external memory is immediate. The speedup increases significantly when the transportation delay is beyond 25 clock cycles. Increasing the delay after 25 clock cycles adds little to the performance. This evaluation shows that the system is still capable of having speedups even if the ratio of memory access time to cache access time is around 5 ($25/5 = 5$). Yet, we choose the transportation delay as 100 clock cycles for our overall performance evaluation considering real-life systems.

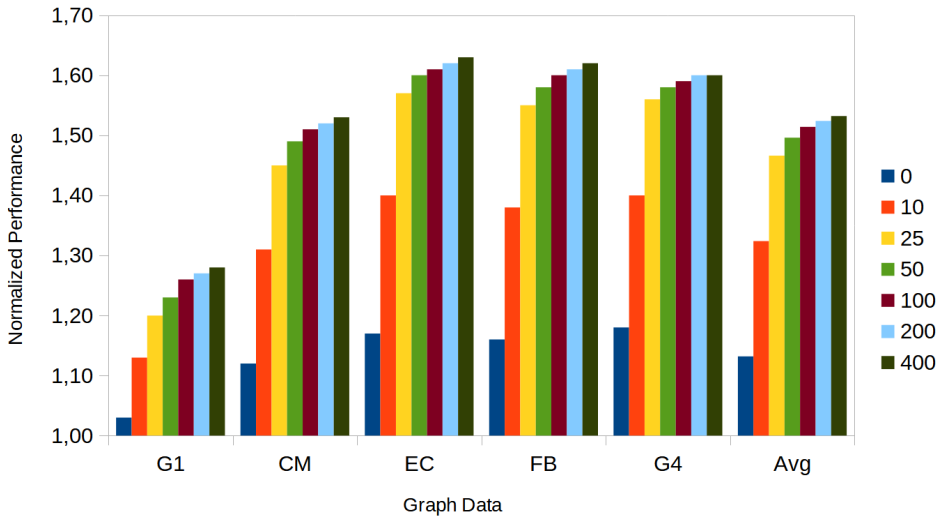


Figure 7.13: Normalized performance for varying transportation delays.

7.6 Compiler Optimization Pass Evaluation

We set the default architectural parameters for our system as explained in Chapter 7.3 for the compiler optimization pass evaluations. We executed the benchmarks as we do for the performance evaluations, but we added the custom instructions automatically using our compiler optimization pass. Then, we compared the execution times against the case where custom instructions are manually inserted. The results are presented in Figure 7.14, execution times are normalized for the case where custom instructions are manually inserted. We observe that compiler optimization pass improves the performance slightly on average. This could be explained by additional optimization passes before our custom optimization, and the insertion of the custom instructions by the compiler does not harm the performance.

In BFS, compiler optimization deteriorates the performance by an average of 0,01%, which is negligible. The average performance improvement by the custom optimization pass is 0,54%. Even though the compiler support has a slight improvement in the performance, it still provides a layer of abstraction for the programmer.

Figure 7.14: The normalized execution times for the compiler optimization pass using different data sets.

Chapter 8

Discussion and Future Work

This chapter presents our observations and possible extensions to explore as future work. The performance evaluations show that the design can increase performance on average by between 27% and 34% for different graphs. The peak speedup is observed as 73% on a special setting.

Even though our results are promising, blocking memory accesses (cache misses) is still not entirely eliminated; thus, memory bandwidth is not fully utilized. Figure 8.1 shows the miss rates for the benchmarks in architecture with and without utilizing the custom instructions. Even though architecture eliminates a substantial amount of blocking memory accesses on graph data, the remaining memory accesses do not show a significant cache hit rate. Thus, further work is needed to utilize the full bandwidth of the memory. Using a set-associative cache instead of a directly mapped cache might alleviate this situation.

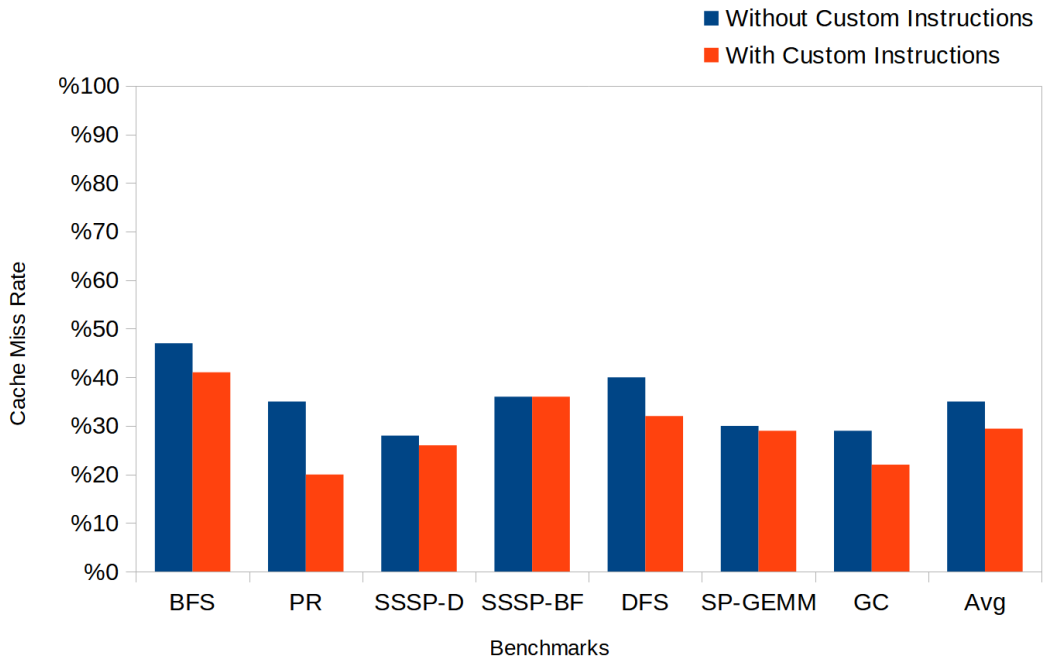


Figure 8.1: Cache miss rates for the benchmarks test with and without utilizing custom instructions.

The GAS library is capable of utilizing the architecture with an average speedup of between 32% and 43%. However, execution times of the same benchmarks with the native code (without custom instructions) are significantly less than the benchmarks in the GAS library. Thus, the library shows a significant computation overhead. In addition, mapping traversal-centric graph algorithms to the GAS approach is not straightforward and causes more computation due to the inherent iterative structure. Yet, the GAS library is still valuable for architecture abstraction and potential usage for parallel architectures. This single-core implementation can be extended to build a parallel architecture to execute on the GAS library.

We also show the prefetch utilization ((Prefetched Data in Cache)/(Number of Prefetch Requests)) values in Figure 8.2. We see that some benchmarks exhibit a high prefetch utilization in some graphs and low prefetch utilization in some others. One reason is that the CPU requests a prefetched block while it is in a pending position at MSHR, the entry in MSHR becomes a normal memory

request. Thus, differences exist for the same benchmarks on different graph data. We should also mention that prefetch size does not substantially affect prefetch utilization. Another factor that affects the prefetch utilization is using *spmreg* instruction over *weight* array values. Because *weight* array values do not predict the future access for a location in the *data* array, accessing them with *spmreg* instruction will cause prefetching unrelated data. This is especially observed in the SSSP-D and SSSP-BF benchmarks. A new custom instruction that behaves similar to *spmreg* but does not imply prefetching can be proposed as a future extension. This will increase prefetch utilization, while reducing the total number of memory requests.

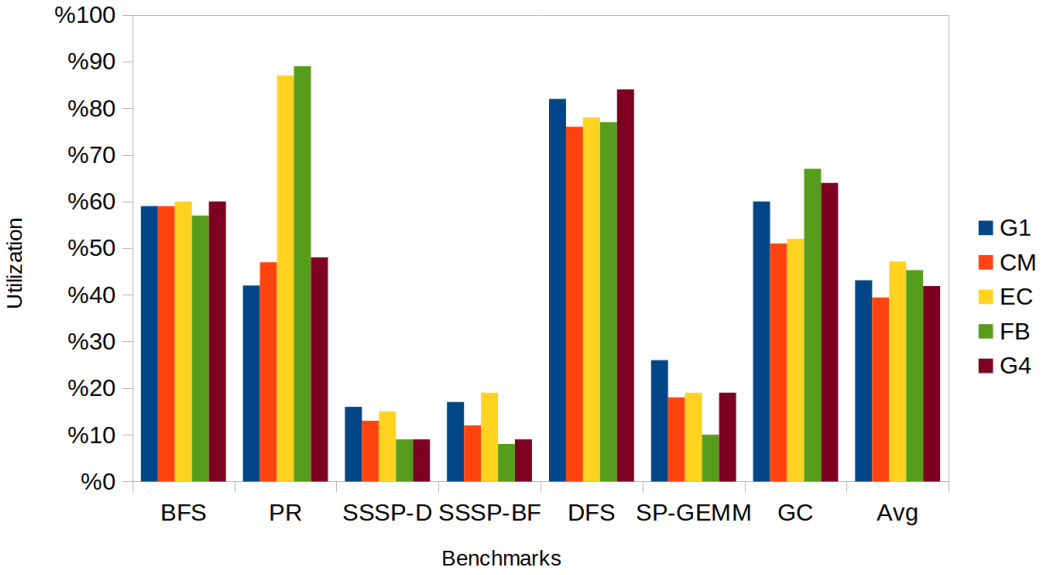


Figure 8.2: Prefetch utilization for the tested benchmarks.

Chapter 9

Conclusion

This study describes the micro-architecture of a single-core RISC-V CPU extended with custom instructions for the domain of graph applications. The software support for the architecture is provided by an LLVM compiler pass and a Gather-Apply-Scatter (GAS) library.

The custom instructions were previously designed to control multiple SPMs that are tailored for graph applications. We analyzed the graph applications and illustrated their computational irregularities and data-driven random memory access patterns. We modified the previously implemented instructions for better performance, better adaptability, and better software support. Along with the behavioral modifications on custom instructions, a novel micro-architecture that relies on a non-blocking cache and a prefetching mechanism is designed.

We performed extensive tests to decide on the default values of architectural parameters. Furthermore, we performed experimental evaluation of the custom processor with well-known graph application benchmarks and graph data scaled for our simulation environment. The average performance of the architecture varies between 10% and 49%, considering native benchmarks, and varies between 22% and 51% considering the GAS library. The peak speedup is 73% for the BFS algorithm on the EC dataset with a large SPM block size parameter.

We believe that our custom architecture can provide benefits for future high-performance graph analytics architectures. Our single-core architecture is a promising initial step for the utilization of a complex parallel architecture. We expect that the evaluation of the architecture in a real system with large graph datasets will show a similar or even better performance. Adding complexity to the base CPU core such as using hierarchical cache design, out-of-order execution, etc. might further benefit our design.

Bibliography

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[2] C.-Y. Gui, L. Zheng, B. He, C. Liu, X.-Y. Chen, X.-F. Liao, and H. Jin, "A survey on graph processing accelerators: Challenges and opportunities," *Journal of Computer Science and Technology*, vol. 34, no. 2, pp. 339–371, 2019.

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

[4] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 84–90, 2002.

[5] M. Jain, M. Balakrishnan, and A. Kumar, "Asip design methodologies: Survey and issues," in *Proceedings of the Fourteenth International Conference on VLSI Design*, VLSI Design 2001, pp. 76–81, 2001.

[6] G. Pulat, "Scratch-pad memory based custom processor design for graph applications," 2020. Master's Thesis, Department of Computer Engineering, Bilkent University.

[7] F. G. Gustavson, *Some Basic Techniques for Solving Sparse Systems of Linear Equations*, pp. 41–52. Boston, MA: Springer US, 1972.

[8] "Milestones:Atanasoff-Berry Computer, 1939," 1990. Engineering and Technology History Wiki, available at `https://ethw.org/Milestones:Atanasoff-Berry_Computer,_1939`, accessed at 25 April 2022.

[9] R. Raúl and U. Hashagen, *The First Computers: History and Architectures*. MIT Press, 2002.

[10] A. Nohl, F. Schirrmeister, and D. Taussig, "Application specific processor design: Architectures, design methods and tools," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '10, pp. 349–352, 2010.

[11] P. Ienne and R. Leupers, "From prêt-á porter to tailor-made," in *Customizable Embedded Processors* (P. Lenne and R. Leupers, eds.), Systems on Silicon, pp. 3 – 9, Burlington: Morgan Kaufmann, 2007.

[12] M. Amold and H. Corporaal, "Designing domain-specific processors," in *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, CODES 2001, pp. 61–66, 2001.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 2019.

[14] K. Karuri, R. Leupers, G. Ascheid, and H. Meyr, "A generic design flow for application specific processor customization through instruction-set extensions (ISEs)," in *Embedded Computer Systems: Architectures, Modeling, and Simulation* (K. Bertels, N. Dimopoulos, C. Silvano, and S. Wong, eds.), (Berlin, Heidelberg), pp. 204–214, Springer Berlin Heidelberg, 2009.

[15] C. A. R. A. Melo and E. Barros, "Oolong: A baseband processor extension to the RISC-V ISA," in *Proceedings of the IEEE 27th International Conference on Application-specific Systems, Architectures and Processors*, ASAP '16, pp. 241–242, 2016.

[16] J. Clemons, A. Jones, R. Perricone, S. Savarese, and T. Austin, "EFFEX: An embedded processor for computer vision based feature extraction," in *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference*, DAC '11, pp. 1020–1025, 2011.

[17] O. Muller, A. Baghdadi, and M. Jezequel, "ASIP-based multiprocessor SoC design for simple and double binary turbo decoding," in *Proceedings of the Design Automation Test in Europe Conference*, vol. 1, pp. 1–6, 2006.

[18] S. Saponara, L. Fanucci, S. Marsi, G. Ramponi, D. Kammler, and E. M. Witte, "Application-specific instruction-set processor for retinex-like image and video processing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 7, pp. 596–600, 2007.

[19] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a Tensor Processing Unit," *CoRR*, vol. abs/1704.04760, 2017.

[20] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[21] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media processing with streams," *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.

[22] 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, 2016.

[23] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, p. 1214–1225, 2015.

[24] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, Article No. 8, 12 p., (New York, NY, USA), Association for Computing Machinery, 2018.

[25] A. Ayupov, S. Yesil, M. M. Ozdal, T. Kim, S. Burns, and O. Ozturk, "A template-based design methodology for graph-parallel hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 420–430, 2018.

[26] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, ISCA '16, pp. 166–177, 2016.

[27] 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*, FCCM '14, pp. 25–28, 2014.

[28] 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, 2015.

[29] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, p. 103–111, 1990.

[30] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proceedings of the ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays, FPGA '16, (New York, NY, USA), p. 111–117, Association for Computing Machinery, 2016.

[31] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 615–628, Association for Computing Machinery, 2019.

[32] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proceedings of the 57th ACM/IEEE Design Automation Conference*, DAC 2020, pp. 1–6, 2020.

[33] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, Article No. 39, 11 p., (New York, NY, USA), Association for Computing Machinery, 2016.

[34] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '14, pp. 140–149, 2014.

[35] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on GPUs: A survey," *ACM Computing Surveys*, vol. 50, no. 6, Article no. 81, 35 p., 2018.

[36] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward unified-memory-efficient high-performance graph processing on GPU," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 2, Article no. 22, 25 p., 2021.

[37] I. V. Afanasyev, V. V. Voevodin, K. Komatsu, and H. Kobayashi, "VGL: A High-performance Graph Processing Framework for the NEC SX-Aurora TSUBASA Vector Architecture," *The Journal of Supercomputing*, vol. 77, no. 8, pp. 8694–8715, 2021.

[38] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, pp. 614–625, 2014.

[39] 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 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 105–117, 2015.

[40] S. Angizi and D. Fan, "GraphiDe: A graph processing accelerator leveraging In-DRAM-Computing," in *Proceedings of the Great Lakes Symposium on VLSI*, GLSVLSI '19, (New York, NY, USA), p. 45–50, Association for Computing Machinery, 2019.

[41] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, HPCA '18, pp. 531–543, 2018.

[42] S. Angizi, J. Sun, W. Zhang, and D. Fan, "GraphS: A graph processing accelerator leveraging SOT-MRAM," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, DATE '19, pp. 378–383, 2019.

[43] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Transactions on Knowledge & Data Engineering*, vol. 30, no. 2, pp. 305–324, 2018.

[44] 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, (New York, NY, USA), p. 135–146, Association for Computing Machinery, 2010.

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of*

the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI '12, (New York, NY, USA), p. 17–30, Association for Computing Machinery, 2012.

[46] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), p. 472–488, Association for Computing Machinery, 2013.

[47] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[48] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0," Tech. Rep. UCB/EECS-2014-54, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 2014.

[49] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, (Washington, DC, USA), p. 81–87, IEEE Computer Society Press, 1981.

[50] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2013.

[51] lowRISC Company, "Ibex Github Page." `https://github.com/lowRISC/ibex`, 2021.

[52] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, (USA), p. 75, IEEE Computer Society, 2004.

[53] T. Feist, "Vivado design suite," *White Paper*, p. 14, 2012. Available at `https://docs.xilinx.com/v/u/en-US/wp416-Vivado-Design-Suite`, accessed at 24 April 2022.

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2009.

[55] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[56] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, p. 269–271, 1959.

[57] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[58] D. R. Ford and D. R. Fulkerson, *Flows in Networks*. USA: Princeton University Press, 2010.

[59] A. Buluç, *Linear Algebraic Primitives for Parallel Computing on Large Graphs*. PhD thesis, 2010. Department of Electrical Engineering and Computer Science, University of California, Berkeley.

[60] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," in *Proceedings of the Cray User Group*, CUG 2010, pp. 1–5, 2010.

[61] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, 2014.

[62] P. Panzarasa, T. Opsahl, and K. Carley, "Patterns and dynamics of users' behavior and interaction: Network analysis of an online community," *Journal of the Association for Information Science and Technology*, vol. 60, pp. 911–932, 2009.

[63] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, (New York, NY, USA), p. 555–564, Association for Computing Machinery, 2017.

[64] J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS '12, (Red Hook, NY, USA), p. 539–547, Curran Associates Inc., 2012.

[65] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of Modern Physics*, vol. 74, no. 1, p. 47, 2002.