RISC-V Instruction Set Extension for Graph Applications

Mehmetali Semi Yenimol
semi.yenimol@bilkent.edu.tr
Bilkent University
Ankara, Turkey

Gülce Pulat
gulce.pulat@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

Ozcan Ozturk
ozturk@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

Abstract

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 are proposed to mitigate performance issues and provide energy efficiency, programmability and flexibility are not addressed well. A domain-specific processor design based on extending the RISC-V Instruction Set Architecture (ISA) is proposed. The design uses new instructions that are supported by the compiler and software library.

Keywords: RISC-V, instruction set, iterative, graph parallel, ISA, extension

ACM Reference Format:
Mehmetali Semi Yenimol, Gülce Pulat, and Ozcan Ozturk. 2022. RISC-V Instruction Set Extension for Graph Applications. In *Proceedings of Sixth Workshop on Computer Architecture Research with RISC-V (CARRV'22).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

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.

CARRV'22, June 19, 2022, New York, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

The main challenges in improving the performance and power usage of graph applications are [1, 6, 7]:

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

Gui et al. show that hardware acceleration for 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 [1]. Furthermore, they have high energy consumption. They evaluate 37 such hardware solutions in terms of design techniques, hardware platform, performance, and energy efficiency. They point out programmability as a major 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 [3, 4]. Due to the programmability advantage, we propose a custom architecture based on RISC-V instruction set architecture (ISA) for iterative graph applications domain. Our initinal study shows that blocking memory accesses are reduced by up to 70%, with sufficient software support and a novel micro-architectural design. 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 the minimal cost.

The rest of the paper is organized as follows: In Section 2, we give some background and describe the general behavior of graph applications, then show the main challenges that are

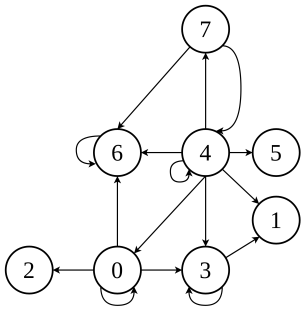

Figure 1. Sample graph with 8 vertices and 16 edges.

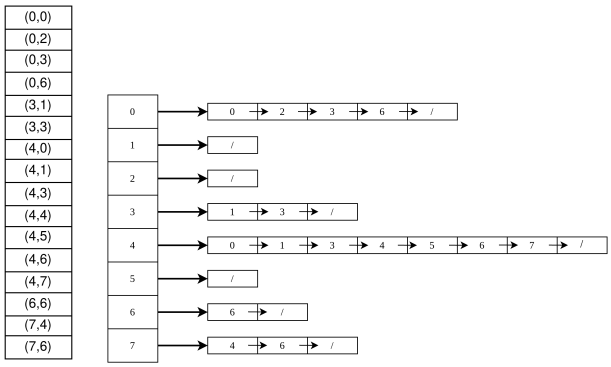

Figure 2. Edge-list and Adjacency-list representations of the sample graph given in Figure 1.

needed to be addressed. Section 3 elaborates on the design and the custom instruction set architecture. In Section 4, we explain the micro-architectural design of our system. The paper is concluded in Section 5.

2 Nature of Graph Applications

A graph is a data structure to show relations between a set of objects. The objects are often called *vertices* and relations between the vertices are referred to as *edges*. If the set of vertices are V and the set of edges are E, then we define a graph G as $G = (V, E)$. Algorithms running on a graph need to have the representation of the graph structure. Vertices of a graph are usually indexed and identified by an integer i in the open interval $[0, |V|)$. 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 it limits scheduling potential for parallel systems [1].

Another common way to represent graphs is using an adjacency list, where an array of the size of vertices 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, each adjacency list of v holds all the vertices u such that $(v, u) \in E$. An example graph is shown in Figure 1 where its edge-list and adjacency list representations are shown in Figure 2.

In practice, adjacency-list representation is often used in compressed structures. The most used compact representation is Compressed Sparse Row (CSR), introduced in [2] 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, CSR representation of the graph in Figure 1 is given in Figure 2.

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 8 vertices (indexed from 0 to 7), an extra index in the *rows* array is used to point to the one plus last index of *columns* array. The symmetric of CSR is called Compressed Sparse Column (CSC) which is used when we are concerned with incoming edges of vertices. For practicality reasons, we choose CSR to represent graphs in this work. A third array (*values*) which is similar to *columns* is mostly used for non-zero values of the matrix A. The non-zero values are often referred to 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 with poor cache locality still occur due to the irregular and data-driven nature of graph applications. 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 that is given by, a possibly dynamic, working list. 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 on directly vertex v. The *work-list* can be a simple list of all vertices as in some of the iterative algorithms such as PageRank. Similarly, it can be a queue for a breadth-first search, a stack for a depth-first search, or a priority queue for Dijkstra's shortest path algorithm, etc. On the other hand, function f is used in representing

the operation on the vertex and/or edge. It depends on the algorithm and can interact with the *work-list*. Even though such abstraction potentially might not always cover all graph algorithms, most of them show a similar behavior.

3 Custom Architecture and Instruction Set

Our design is based on an implementation which uses scratch-pad memories (SPM). We use an SPM, referred 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). The other part is called Vertex Scratch-Pad (VSP) 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 for an address, if the address lies on the given range, data is to be located in VSP, otherwise in GSP. The internal workings of VSP and GSP are like a modern-cache system.

The instruction set of the architecture is defined by extending the RISC-V ISA [8]. The 4 new custom instructions are added to RISC-V instruction set, namely, *spmcon, memspm, spmreg*, and *delspm*. *spmcon* instruction is related to configuration of SPMs in the architecture. On the other hand, *memspm, spmreg*, and *delspm* instructions are defined to manage ESP. The micro-architecture uses 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 found to be suitable rather than dividing the cache into two parts, thus providing more space for the cache. The micro-architectural design is explained in Section 4.

At the ISA level, functionalities of custom instructions are modified to reflect the changes in micro-architecture. Graph data is assumed to be represented in CSR format and elements of *rows* and *columns* are assumed to be 4 byte integer values. The descriptions of these instructions and their functionality 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 edge-related data array (*columns* or *values* array at CSR representation) given as an argument. The second call marks the start address of vertex-related data array given as an argument. It could be *rows* or *data* arrays in CSR representation or any other array depending on the application. 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.
- **memspm**
  - **Description**: Used to place 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 that is 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.
- **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, the data is sent to the register file if found in ESP. If not found, a memory request is issued as if it is an ordinary load instruction.
- **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. If the offset of a chunk of edge data is previously fetched from memory by a *memspm* instruction matches with the given argument of *delspm*, the chunk can become free to be replaced by another chunk of edge data.

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. *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

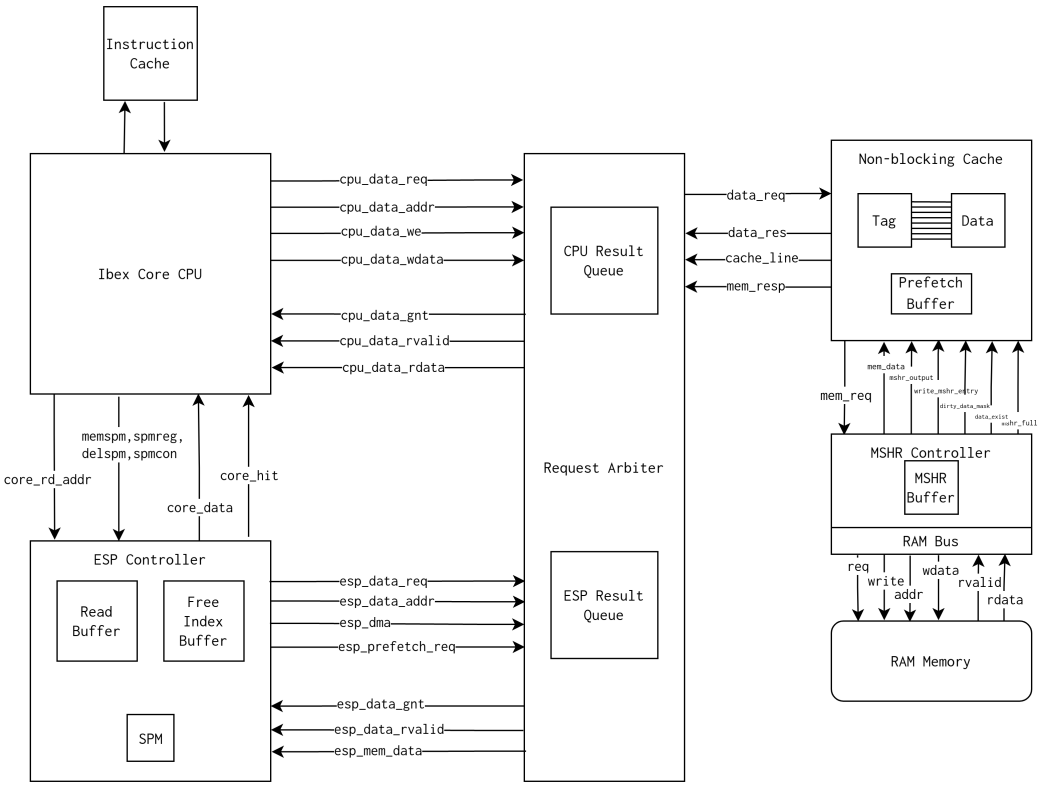

Figure 4. Overall System with the interactions of its 6 components: CPU with its Instruction Cache, ESP Controller, Request Arbiter, Non-blocking Cache, Miss Status Holding Registers(MSHR) Controller, and RAM Memory

call *delspm* on *columns*[*rows*[3] (edges of the vertex 3), a later *memspm* request can purge the edge data associated with this free chunk and place its own related edge data. *spmreg* is used to get 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 the 4 as the index of outgoing neighbor. While getting this information, we can also make a prefetching request for the next 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 later execution of the program. The prefetching scheme is explained in Section 4 with detail.

4 Micro-architectural Design

The micro-architecture of the system is explained in this section. 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. To address this challenge, two major components are designed: *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 CPU. Since both CPU and ESP Controller can issue memory requests simultaneously, a non-blocking cache structure is designed so that multiple outstanding memory requests can be handled.

The overall system consist 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 4. Core CPU interacts with ESP Controller when executing the custom instructions defined earlier in Section 3. The *spmcon*, *memspm*, and *delspm* instructions does not cause a stall in CPU and 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 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 as a miss or hit depending on the availability. The results of requests are kept in Request Arbiter queues. When the head of queues holds "ready" data (cache hit or a later memory response on cache misses), Request Arbiter directs the result to 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 immediately returns the result as a hit. When the data is not available, it allocates space and issues memory request(s) through MSHR Controller. Since it uses 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 Miss Status Holding Registers (MSHR) structure proposed by Kroft [5] 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.

The components in the design keep tables for storing data during the execution. These tables are mostly implemented as buffers that work in FIFO (First In First Out) manner.

5 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 of the architecture is expected to be provided by LLVM compiler optimization in addition to the Gather-Apply-Scatter(GAS) library.

The custom instructions are designed for controlling multiple SPMs that are tailored for graph applications. Analyzing the graph applications, we illustrated their computational irregularities and data-driven random memory access patterns. After the analysis, we made small changes to the behaviors of defined instructions for better performance and better adaptability for 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.

References

[1] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A Survey on Graph Processing Accelerators: Challenges and Opportunities. *Journal of Computer Science and Technology* 34, 2 (2019), 339–371.

[2] Fred G. Gustavson. 1972. *Some Basic Techniques for Solving Sparse Systems of Linear Equations.* Springer US, Boston, MA, 41–52. https://doi.org/10.1007/978-1-4615-8675-3_4

[3] M.K. Jain, M. Balakrishnan, and A. Kumar. 2001. ASIP Design Methodologies: Survey and Issues. In *VLSI Design 2001. Fourteenth International Conference on VLSI Design.* 76–81. https://doi.org/10.1109/ICVD.2001.902643

[4] K. Keutzer, S. Malik, and A. R. Newton. 2002. From ASIC to ASIP: the Next Design Discontinuity. In *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors.* 84–90. https://doi.org/10.1109/ICCD.2002.1106752

[5] David Kroft. 1981. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Minneapolis, Minnesota, USA) *(ISCA '81).* IEEE Computer Society Press, Washington, DC, USA, 81–87.

[6] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel Graph Analytics. *Commun. ACM* 59, 5 (April 2016), 78–87. https://doi.org/10.1145/2901919

[7] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[8] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html