

Nested-Parallelism PageRank on RISC-V Vector Multi-Processors

*Alon Amid
Borivoje Nikolic, Ed.
Krste Asanović, Ed.*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-6

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-6.html>

April 19, 2019



Copyright © 2019, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to thank Albert Ou and Colin Schmidt for their invaluable assistance and indispensable expertise with the Hwacha vector accelerator architecture, micro-architecture, and supporting toolchain. I would also like to thank the DARPA CRAFT and PERFECT programs for supporting this research. This work was partially funded by DARPA Award Number HR0011-12-2-0016, and ADEPT/ASPIRE Lab industrial sponsors and affiliates Intel, HP, Huawei, NVIDIA, and SK Hynix. Any opinions, findings, conclusions, or recommendations in this report are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Nested-Parallelism PageRank on RISC-V Vector Multi-Processors

by Alon Amid

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Borivoje Nikolić
Research Advisor

(Date)

* * * * *

Professor Krste Asanović
Research Advisor

(Date)

Nested-Parallelism PageRank on RISC-V Vector Multi-Processors

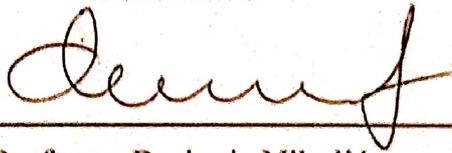
by Alon Amid

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

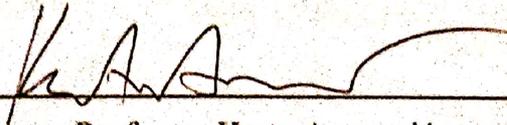
Committee:



Professor Borivoje Nikolić
Research Advisor

April 8, 2019

(Date)



Professor Krste Asanović
Research Advisor

April 10, 2019

(Date)

Abstract

Nested-Parallelism PageRank on RISC-V Vector Multi-Processors

by

Alon Amid

Master of Sciences in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

Professor Krste Asanović, Co-chair

Graph processing kernels and sparse-representation linear algebra workloads such as PageRank are increasingly used in machine learning and graph analytics contexts. While data-parallel processing and chip-multiprocessors have both been used in recent years as complementary mitigations to the slowing rate of single-thread performance improvements, they have been used together most efficiently on dense data-structure representations as opposed to sparse representations. This work presents nested-parallelism implementations of PageRank for RISC-V multi-processor Rocket chip SoCs with vector architecture accelerators. These software implementations are used for hardware and software design-space exploration using FPGA-accelerated simulation with multiple silicon-proven multi-processor SoC configurations. The design space includes a variety of scalar cores, vector accelerator cores, and cache parameters, as well as multiple software implementations with tunable parallelism parameters. This report shows the benefits of the loop-raking vectorizing technique compared to an alternative vectoring technique, and presents up to a 14x run-time speedup relative to a parallel-scalar implementation running on the same SoC configuration. A 25x speedup is demonstrated in a dual-tile SoC with dual-lanes-per-tile vector accelerators, compared to a minimal scalar implementation, demonstrating the scalability of the proposed nested-parallelism techniques.

Contents

Contents	i
List of Tables	iii
1 Background	1
1.1 Graph Processing	1
1.2 PageRank	2
1.3 Sparse Matrix Representations	4
1.4 Graph Processing Frameworks	6
1.5 GraphMat Graph Processing Framework	7
1.6 Vector Machine Architectures	8
2 Nested Parallelism Using Vector Architectures	10
2.1 Nested Parallelism in Graph Processing	10
2.2 Parallel Techniques for Sparse Matrices	11
3 Experimental Setup	21
3.1 Graph Processing Framework Infrastructure	21
3.2 Agile Hardware Development	21
3.3 Software Development	22
3.4 Validation and Verification	23
3.5 Performance Evaluation and Design Space Exploration	24
4 Evaluation and Design Space Exploration	25
4.1 Evaluation	25
4.2 L2 Cache Size	28
4.3 Total Number of Tiles	28
4.4 Total Number of Vector Lanes	28
4.5 Number of Tiles vs. Number of Vector Lanes	29
4.6 Packed-Stripmining vs. Loop-Raking	31
4.7 Graph Size and Structure	33
4.8 Vector Accelerator vs. Multi-Core Scalar Processors	36

4.9	Bottlenecks	38
4.10	Related Hardware Improvements	38
4.11	Generalization	39
4.12	Future Work	40
5	Conclusion	42
A	Measurement Results	43
	Bibliography	50

List of Tables

4.1	Properties of Evaluation Graphs	25
4.2	Simulated SoC Hardware Configurations	26
A.1	Measurements for the wikiVote Graph	43
A.2	Measurements for the roadNet-CA Graph	46
A.3	Measurements for the amazon0302 Graph	48

Acknowledgments

I would like to thank Albert Ou and Colin Schmidt for their invaluable assistance and indispensable expertise with the Hwacha vector accelerator architecture, micro-architecture, and supporting tool-chain. I would also like to thank the DARPA CRAFT [53] and PERFECT [45] programs for supporting this research. This work was partially funded by DARPA Award Number HR0011-12-2-0016, and ADEPT/ASPIRE Lab industrial sponsors and affiliates Intel, HP, Huawei, NVIDIA, and SK Hynix. Any opinions, findings, conclusions, or recommendations in this report are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Chapter 1

Background

1.1 Graph Processing

Graph processing has been a topic of recent interest in high performance computing, systems, and architecture research. While graph abstractions have long been of interest in mathematical and numerical computing communities, the rise of data analytics and the big-data revolution have exposed the various use-cases of graph processing to many additional domains. Computing statistical properties of graphs is required for many scientific, data-analysis, and machine learning applications, including recommendation systems [9], fraud detection [4] and biochemical processes [51, 3].

Graphs are a popular way of representing mathematical problems and algorithms. Graph theory is a continuously developing mathematical field, which includes several hard problems that have challenged mathematicians and computer scientists for many years. Many of these problems represent real-world problems such as the traveling-salesman problem [48], task-graph scheduling [33], graph coloring, subgraph isomorphism, and others [16].

A common perception is that graph processing problems present irregular data layouts and a high degree of implicit data-level parallelism, which make them a challenging form of research in the computer science community. This is likely, since many graph problems require traversing the graph, which can have an unpredictable structure. This perception has led to the grouping of many of these problems under the "graph processing" domain, and has led many researchers in this domain to focus on optimizing memory bandwidth utilization. Some publications further identify that graph-processing problems have additional common characteristics, such as little data-locality, fine-granularity fixed memory accesses, and low arithmetic intensity [19]. However, one could usually find several counter examples for each of these properties: most graph Triangle-Counting kernel implementations do not have fixed-size fine-granularity memory accesses, while certain implementations of PageRank may have high arithmetic intensity and high data-locality [14].

Nevertheless, there have been several new paradigm proposals and standardization attempts for graph-related workloads. One such paradigm is vertex-programming [41]. This approach treats every vertex as an individual entity with a set of incoming edges and outgoing edges. The program

and algorithms are written from the point of view of a single vertex, and may continue running indefinitely until convergence. This approach assumes that all vertices run the same program in parallel. Further abstraction nicknamed the "Gather-Apply-Scatter" interpretation [18] of vertex programming adds additional structure to this approach, by assigning three stages (Gather, Apply, and Scatter) to the vertex program. Another such paradigm is the linear-algebra based approach for graph processing. This approach treats the graph representation as an adjacency matrix, and defines various linear algebra operations that can be performed using this matrix. Once recent attempt to standardize this approach is GraphBLAS [31]. GraphBLAS attempts to provide the graph processing field a more structured-nature, by mapping common graph algorithms to sparse linear algebra operations. This approach requires overloading the algebraic operators with specific actions performed by the graph processing algorithm.

These programming paradigms attempt to assist with the previously mentioned graph processing challenges of irregular structure and implicit data-parallelism. However, previous works by both Beamer [10] and Eisenman [15] have found that in single-node server CPUs, memory bandwidth is not a bottleneck for graph processing. They found that server nodes do not saturate their memory bandwidth as expected from graph workloads with irregular memory accesses. It is important to understand the context and reference point when analyzing these types of conclusions. While server processors may indeed not saturate their memory bandwidth, this may not be the case for other data-parallel processors such as GPUs. However, these accelerators and data-parallel processors come at a cost - this can be the data-transfer cost between the host processor and the discrete accelerator, or the energy efficiency cost of the accelerator itself. Therefore, it is worth exploring methods of optimizing the compute-pipeline in server-class processors.

1.2 PageRank

One particular instance of a common graph processing kernel is PageRank [47]. PageRank is an algorithm originally used by Google to measure the importance of websites, with the purpose of ranking them. Each website is modeled as a node (or vertex) in a graph, and hyperlinks between websites are modeled as edges in the graph. After running the PageRank algorithm, each vertex (representing a website) is assigned a PageRank score, which allows it to be compared and ordered against other websites, hence - creating a ranking. The PageRank score is effectively a probability distribution that represents the likelihood of a random walker (or a random "hyperlink clicker") to arrive at a particular vertex (or web page)

There are various methods for computing the probability distribution of this random walk. In its most simplified form, the PageRank value of a vertex u ($PR(u)$) is computed by summing the PageRank values of its neighbors ($PR(v)$) divided by the number of incoming edges to each neighbor (N_v), and using a dampening factor (d):

$$PR(u) = (1 - d) + d \sum_{v \in B_u} \frac{PR(v)}{N_v}$$

By viewing the PageRank problem as an irreducible Markov chain, this probability distribution can be computed as an eigenvector problem or a homogeneous linear system [34, 20]. Using the power method, this results in an iterative process of Sparse Matrix-Vector multiplication (SpMV) operations. Each iteration computes its PageRank values by multiplying the transition probability matrix with the previous iteration’s PageRank values. The transition probability matrix is in fact the graph adjacency matrix, factored by a dampening factor and divided by the relevant vertex’s degree (with several exceptions to guarantee that the matrix will be stochastic and irreducible). The processes is repeated iteratively until the convergence of the PageRank vector. Convergence is guaranteed due to the primitive properties of the PageRank transition probability matrix, which is stochastic and irreducible, and therefore guarantees convergence to a unique dominant eigenvector. In formal terms, if A is the adjacency matrix, P is the transition probability matrix, d is the damping factor, N is the number of ongoing edges, $|V|$ is the number of incoming edges, and $y^{(k)}$ is the PageRank values vector at the k -th iteration, then the iterative SpMV formulation can be written as:

$$P = A * \frac{1}{N}$$

$$y^{(k)} = d * P * y^{(k-1)} + \frac{(1-d)}{|V|}$$

While additional alternative methods haven been proposed for efficient PageRank computation [34], this report will focus on the previously mentioned iterative SpMV method.

PageRank has evolved with many variations. For example, Personalized or Topic-Sensitive PageRank [21] proposes multiple PageRank vectors with biases towards specific topics, while Weighted-PageRank [58] accounts for both incoming link and outgoing links when computing the PageRank value. Nevertheless, all variations follow the basic ideas of an iterative random-walk process, in which iterative SpMV is the main component.

The use of iterative SpMV operations makes PageRank a useful and interesting benchmark. It allows for an interesting application while extensively using and demonstrating the performance of a primitive and elementary linear algebra operation. While other graph algorithms are commonly formalized and represented as sparse linear algebra operations, PageRank is unique in that it uses a simple SpMV, with no overloaded operators. This is unlike other graph processing kernels implemented using a sparse linear algebra abstractions (such as those using GraphBLAS), which require overloading the algebraic addition and multiplication operators.

The quantification of the importance of a node in a graph is a common problem. Hence, the use of PageRank has naturally expanded beyond the ranking of web pages. It has been used for urban planning [27], semantic analysis [50], and studying protein folding [24]. It is therefore not a surprise that PageRank is a common benchmark for graph-processing optimizations in software and hardware. PageRank will be the benchmark of choice for the purposes of this work as well.

1.3 Sparse Matrix Representations

Graphs are commonly represented in the form a adjacency lists or adjacency matrices. Since most real-world graphs are not fully connected, graphs represented using adjacency matrices usually result in sparse adjacency matrices. Sparse matrices can be represented in memory using a variety of formats. Unlike dense matrices, which are commonly represented as contiguous column-major or row-major arrays, sparse matrices present an additional degree of information, which allows for memory-efficient representations. To demonstrate some of the different popular sparse matrix representation, the example matrix in figure 1.1 will be used throughout this section.

0	81	0	0	0	0	0	0
0	5	0	0	0	0	0	0
0	0	0	0	0	0	0	0
61	0	9	0	0	0	34	11
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	42
0	0	0	0	0	0	17	0
0	92	0	0	0	0	0	70

Figure 1.1: Example sparse matrix to be used throughout the explanation.

In the coordinate format (sometimes known as "edge list" or COO), the non-zero elements of the matrix are represented as a list triplets, each representing a row and column coordinate in the matrix associated with a value. These triplets can be implemented using 3 arrays, each one dedicated to a specific type, or as a single array of triplets - depending on memory locality considerations. In COO format, the elements are not sorted by row, column or value.

row_indices	0	1	3	3	3	3	5	6	7	7
column_indices	1	1	0	2	6	7	7	6	1	7
values	81	5	61	9	34	11	42	17	92	70

Figure 1.2: Coordinate (COO) representation of the example matrix. Each non-zero values is represented by corresponding elements of the two indices arrays and the values array.

Alternatively, formats such as Compressed Sparse Row (CSR) and the inverse equivalent Compressed Sparse Column (CSC) provide improved element random access time complexity. In the CSR format, sparse rows are compressed into a single array, while an array of row pointers allows

for constant-time access to each row. Hence, 3 arrays overall are used for the CSR representation. Similarly to the COO format, two of the arrays are used for values and column indices. However, the third array consists of pointers to indices of the other two arrays, rather than the actual indices of the columns.

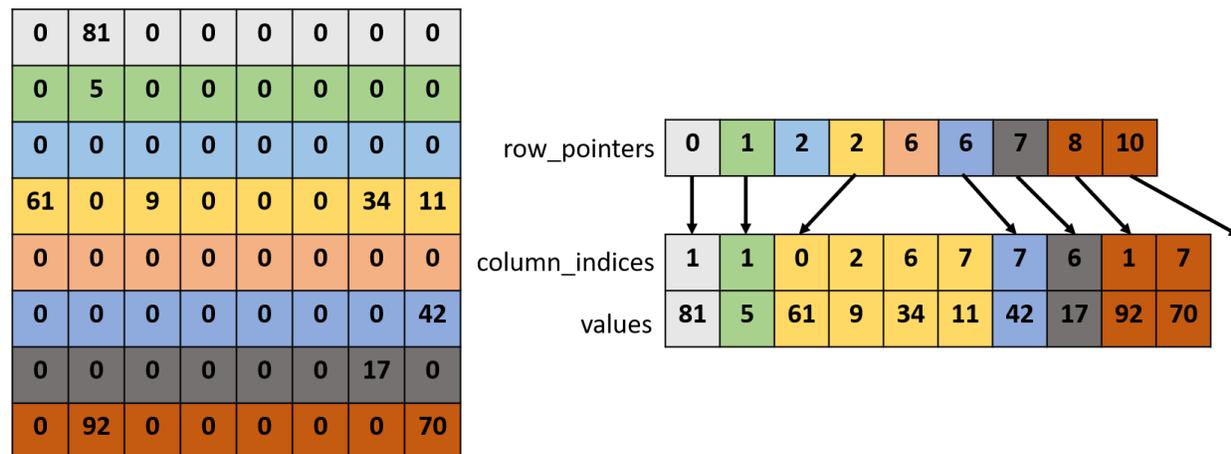


Figure 1.3: CSR representation of the example matrix. The compressed rows are represented using the column_indices and values arrays.

In the equivalent CSC format, the sparse columns are compressed into a single array, while an array of column pointers allows for constant-time access to each column. It is important to note that The CSR and CSC formats have an access-time bias towards one direction of edges (either incoming or outgoing, depending on the interpretation of the matrix). Therefore, some graph processing implementations which require access to both incoming and outgoing edges of a vertex may choose to implement both the CSR representation of the matrix as well as the CSC representation of the matrix.

An additional sparse matrix representation, which is less common than the previously mentioned formats, is the Double Compressed Sparse Column (DCSC) [12] representation. This format is useful for the cases of hyper-sparse matrices, since it provides an additional level of compression on the column indices (in addition to the row indices). Nevertheless, this additional level of compression comes at the cost of an explicit column indices array, and an auxiliary column pointers array to reduce access time complexity. It is clear that in our running-example matrix, DCSC is not a memory-efficient representation as compared to CSC or COO. Nevertheless, many graph adjacency matrices are indeed hyper-sparse, and therefore DCSC is a fitting representation for those cases. Naturally, the inverse equivalent of this representation also exists, in the form of DCSR (Double Compressed Sparse Row).

Depending on the characteristics of the non-zero values of the matrix, different sparse matrix representation may be appropriate from space complexity perspective and access time complexity perspective. This work will focus on the DCSC and DCSR formats.

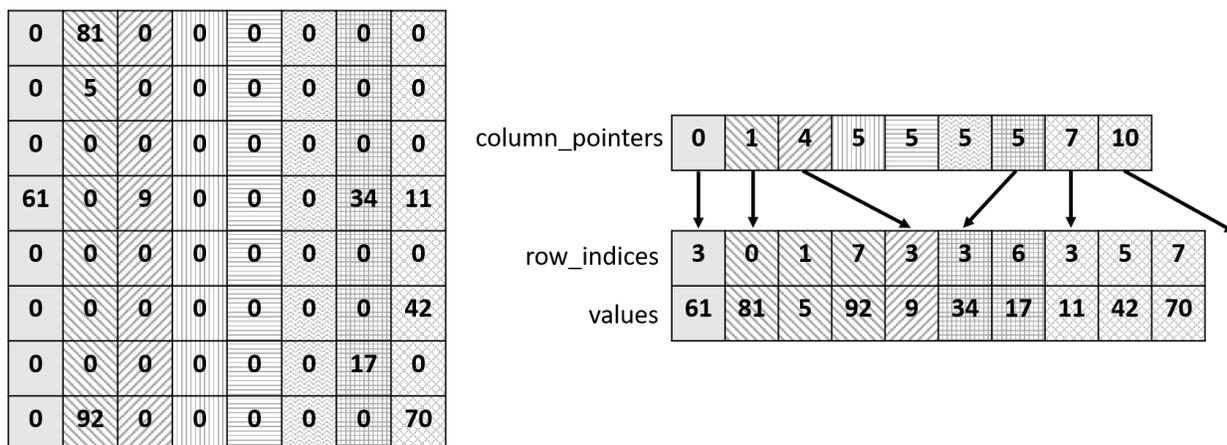


Figure 1.4: CSC representation of the example matrix. The compressed columns are represented using the row_indices and values arrays.

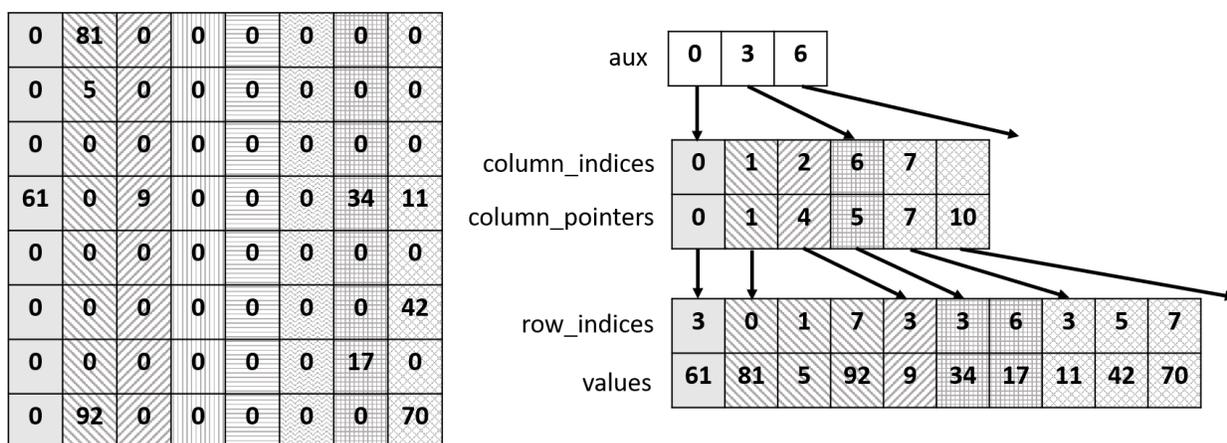


Figure 1.5: DCSC representation of the example matrix. The additional level of compression over the CSC representation is provided by the additional column_indices array and auxiliary array

1.4 Graph Processing Frameworks

Concurrent with the big-data revolution, a plethora of software frameworks have emerged for many types of graph processing use-cases [54, 42]. These frameworks range from in-memory distributed graph processing such as GraphX, Giraph and Pregel [17, 8, 41], to single node shared-memory graph processing such as Ligra and GraphMat [55, 56], to single node graph processing with secondary storage such as MOSAIC and X-Stream [40, 52].

Most frameworks provide novel contributions in the form of new abstractions, optimized kernels, data-structure management techniques, and various distributed capabilities to improve the performance of common graph-processing kernels such as Breadth-First-Search (BFS), PageR-

ank, Connected-Components (CC), and Single-Source-Shortest-Path (SSSP). Some of these frameworks have presented benchmark results on graphs with billions [55] and even a trillion edges [40] on a single computing node.

While these achievements present important advances in software utilization of existing hardware for graph processing, it is interesting to note that many of these publications were required to benchmark these frameworks on synthetically generated graphs, due to the lack of public large-scale data-sets. While each of these frameworks present impressive performance and scalability in their domain, it is unclear which graph-processing-related application each of them addresses. This adds to the interesting observation in [44] regarding the trade-off between the scalability of these frameworks to their actual performance on various hardware platforms.

Nevertheless, these frameworks provide structured methods and abstractions for addressing graph processing problems, and industrial publications indicate that they have been used in practice in various settings. While the contributions of these frameworks to high-performance graph kernel execution may be arguable, there is little doubt regarding their contribution to the increasing use of graph processing algorithms, especially at scale. Therefore, they should be useful platforms to be studied for architectural research purposes.

1.5 GraphMat Graph Processing Framework

GraphMat [56] is a graph processing framework that presents an interesting mix between graph processing abstractions and performance optimization. GraphMat provides a hybrid approach in which the algorithms and kernels are written using a user-facing vertex-centric API (with a structure similar to a Gather-Apply-Scatter paradigm), and those algorithms are then transformed and applied as overloaded sparse linear algebra operations in the back-end of the framework. This provides the advantage of the simple and scalable vertex-programming abstraction that popularized the early graph processing frameworks, while enabling a structured back-end that can be more easily manipulated for architecture-specific performance optimization. Linear algebra operations have been a historical target of many optimization libraries and techniques, and therefore mapping to these operations has the potential for future use of some of these techniques.

GraphMat has been shown to have competitive performance with state-of-the-art shared memory graph processing framework, and it uses various libraries to extract parallelism through OpenMP and MPI interfaces. It has also been used as a reference framework for other architecture-related research works [19]

GraphMat uses the Double Compressed Sparse Column/Row (DCSC/DCSR) data structures to represent its graphs. Unlike the more commonly used CSR/CSC data structure, DCSC/DCSR provide two levels of indirection rather than only one. This feature is originally designed to reduce memory access times under the assumption of real-world hyper-sparse graphs, but it also allows more flexibility in experimenting with nested parallelism methods for the purposes of this work.

1.6 Vector Machine Architectures

Data-parallel accelerators were previously mentioned to be one potential solution to the lack of memory-bandwidth saturation in graph processing kernels presented in [10, 15]. Nevertheless, there is a wide spectrum of data-parallel accelerator architectures, each accompanied by a set of characteristics and constraints. The most commonly studied data-parallel architectures are packed-SIMD, GPUs and vector architectures [22]. The packed-SIMD ISA extensions (such as Intel’s AVX extensions) were highly influenced by their original hardware implementations, and therefore current implementations are many times limited by the original programming model which encodes the register widths into the instructions [49]. This leads to algorithms and kernels that must be designed and optimized around specific vector lengths, regardless of the program or data characteristics. Nevertheless, the tight integration of many packed-SIMD units into modern processors with a single shared memory address space can lead to quick performance improvements through code changes. GPUs are designed as throughput processors that provide a large amount of compute resources. However, many of these compute resources require lock-step coordination throughout the advancement of the program. When there is divergence in the program, this results in under-utilized resources that still consume large amounts of energy. Furthermore, the SIMT programming model exposed through CUDA for NVIDIA GPUs does not expose these diverging constraints, which can lead to further energy-inefficiency with the use of GPUs as data-parallel accelerators. Vector architectures use deeply-pipelined execution of many operations to hide memory-access latency and increase throughput. Their programming model with vector-length registers and predicate masks allows for efficient compilation which does not depend on the vector-processor micro-architecture. Vector processors can saturate memory-bandwidth easily, in an energy-efficient manner, by providing an explicit programming model and using latency-hiding techniques which do not require over-provisioning of resources.

Vector architectures were popular during the early age of super-computing, but have been mostly abandoned in favor of out-of-order processing, multi-core processing, and discrete data-parallel accelerators. However, with the increasing availability of on-chip transistor area and DRAM bandwidth, vector processors have recently been reexplored alongside micro-processors.

The Hwacha micro-architecture [37] is a decoupled vector-machine implementation developed in Berkeley, associated with the RISC-V and Rocket-Chip SoC generator [7] infrastructure. It is an evolution of previous decoupled vector-fetch projects such as Maven [36], and uses the Hwacha RISC-V non-standard ISA extension. Hwacha’s main micro-architectural features include a master sequencer to control multi-lane operation, an expander to deconstruct instructions into micro-operations, a vector run-ahead unit to take advantage of the decoupled interface, and a banked vector register file (figures 2 and 3 of [37]). The banked vector-register file allows systolic execution of vector instructions using an SRAM array for low latency register-file access. Current implementations include 4-banked register files. The master sequencer tracks the dependency information between different vector instructions, and the expander decomposes vector instruction into fine-grained operations to be executed in different parts of the vector-machine. These elements allow for the deep-pipeline and efficient utilization of the execution units. Hwacha can be integrated in the Rocket-Chip SoC generator, and uses a similar TileLink based [13] cache-coherent memory

system.

The Hwacha ISA extension provides instruction primitives historically associated with vector architecture. These include configuration instructions for the vector machine (vector register lengths and element widths), vector arithmetic operations, vector memory operations, atomic memory operations, and several other unique vector instructions. It includes separate scalar register file, vector register file, and predicate mask register file. This requires explicit programming of data movement between the scalar processors and the vector processors. It also allows for explicit programming of the predicate mask operations. Hwacha also allows for mixed-precision arithmetic operations, allowing higher performance and energy efficiency through software optimizations.

The Hwacha micro-architecture has been optimized for, and mostly been evaluated on, dense linear algebra kernels such as general matrix multiplication (DGEMM). Specifically, It has not been designed or optimized for sparse and irregular workloads. The properties of the Hwacha vector architecture have not yet been explored using sparse linear algebra kernels. An evaluation of the bottlenecks of sparse linear algebra workloads on this micro-architecture can provide additional insight into future design choices.

Chapter 2

Nested Parallelism Using Vector Architectures

2.1 Nested Parallelism in Graph Processing

Throughout this report, "nested parallelism" is considered to be the use of multiple parallel execution methods in a hierarchical manner. Nested parallelism is used extensively in various software libraries to maximize the amount of exploited parallelism given a set of execution resources. Furthermore, it allows for tuning and finer-grained load-balancing between parallelizable elements [23].

The ideas of exploiting nested parallelism in graph processing have shown encouraging results in several previous attempts. Nested parallelism within a single GPU has been studied to efficiently utilize GPU architectures for general data-parallel workloads [25, 43]. Nested parallelism using multiple GPUs has been demonstrated on graph processing algorithms, but requires careful dynamic load balancing due to the high cost of transferring data between host processors and CPUs [26]. Nested parallelism in graph processing has also been explored using packed-SIMD approaches with Intel AVX extensions and multi-core processors [39, 28].

Recent work by research groups at Cornell [32] provides significant contributions in the study and taxonomy of loop-level parallelism through nested parallel hardware elements. Loop-task parallel programs are a major use-case for nested-parallelism implementations. This work identified challenges in combining multi-threading and packed-SIMD abstractions for loop-task parallel programs. This challenge is embodied through reduced programmer productivity, and marginal speedups resulting from the combined parallel methods, as opposed to using each of the parallelism techniques separately. This work also proposes a unique hardware-software interface to expose loop-task level parallelism to a hardware loop-task-accelerator (LTA), with the goal of resolving these challenges. Using a software hint, the LTA can identify tasks, partition them into micro-tasks, and group them into micro-threads based on the LTA's lane-configuration and real-time load. The rest of the LTA is designed similarly to historic vector machines architectures, with a mix of coupled (chimes) and decoupled (lanes) elements. This work provides significant

insights into the behavior of hardware-based loop-task-parallelism management, and the design space between lock-step and decoupled hardware task execution.

The methods under investigation in this report attempt to find a middle-way between the proposals presented in the aforementioned LTA work [32], and existing hardware and software infrastructure. It will explore the nested parallelism of chip multi-processors (CMPs) with decoupled vector-fetch machines integrated into SoCs, based on the tape-out-proven Hwacha [37] micro-architecture. This involves both hand-tuned optimization of the internal vector-architecture code for the consideration of the particular sparse data-structures representations, as well as an additional layer of OpenMP for CMP multi-threading and load-balancing modeling and management. To our knowledge, this nested-parallelism approach for PageRank has not been previously attempted using vector architecture instructions sets and vector-machine micro-architectures.

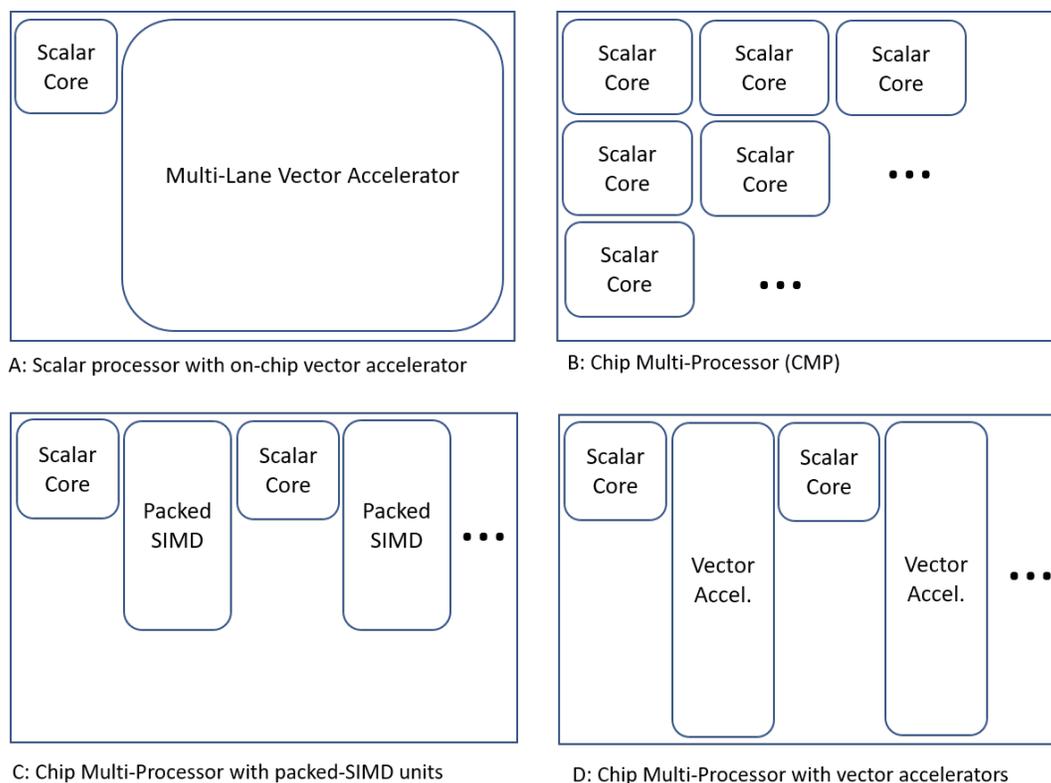


Figure 2.1: Various SoC Configurations for Data-Parallel Workloads

2.2 Parallel Techniques for Sparse Matrices

Sparse matrices are commonly represented using multiple levels of indirection, making the exploitation of parallelism within a sparse matrix for linear algebra operations highly dependent upon the data-structure representation. While some representations such as COO may allow

embarrassingly-parallel execution at the cost of data-locality, other representations such as CSR/CSC improve data-locality and constant-time accesses at the cost of creating dependencies between different parts of the data-structure (hence, reducing parallelism).

For the purposes of nested-parallelism experimentation, DCSC/DCSR representations are a useful data structure, since they expose two levels of indirection, which provide a natural boundary between two levels of parallelism. Therefore, the following examples and implementations will focus on DCSC/DCSR matrix representations. The use of nested parallelism is equivalent between DCSC and DCSR representation, and the choice of data-structure depends on the application use-case. Therefore, the rest of this explanation will focus mostly on DCSC representation, while maintaining generality for both cases.

The top level of the DCSC data structure can be thought of as a coarse-grain parallel layer. This layer can be parallelized in multi-threaded hardware implementations using parallel hardware threads. Specifically, this evaluation will use OpenMP threads to parallelize across CMP cores. This is demonstrated on the example matrix for the DCSC case in figure 2.2

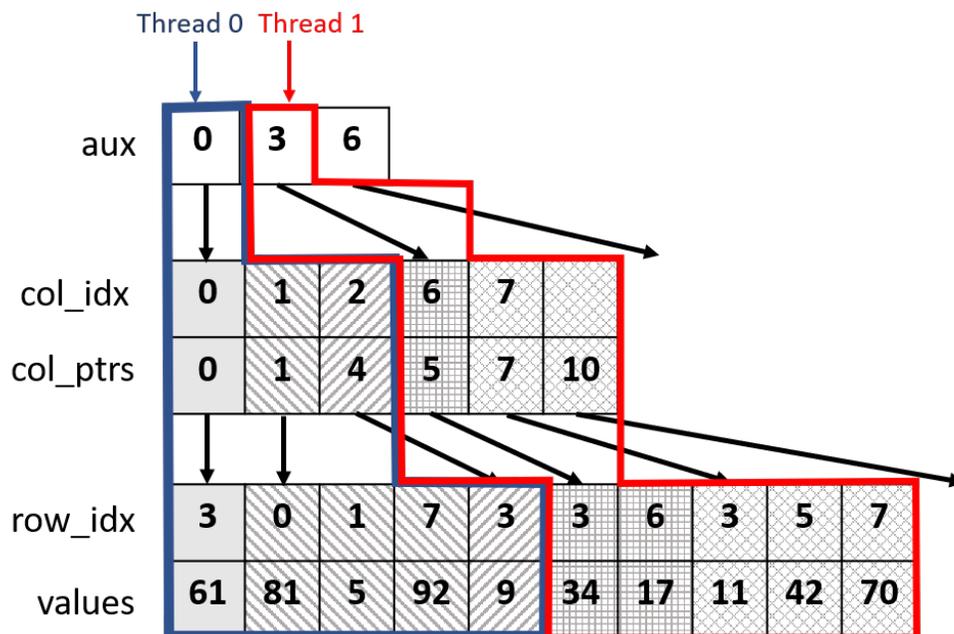


Figure 2.2: OpenMP thread assignments on the DCSC representation of the example matrix

Noticeably, each thread will in-fact process a sub-section of the matrix which is represented in CSC format (with the additional column indices array). In the case of the first thread processing our example matrix, the CSC matrix that will be processed is demonstrated in figure 2.3

The second level of our nested-parallelism scheme will therefore process the internal CSC sub-matrices. Processing the CSC sub-matrix will be implemented using three different methods. Due to the small size of the example matrix, the illustrations demonstrate the concept assuming

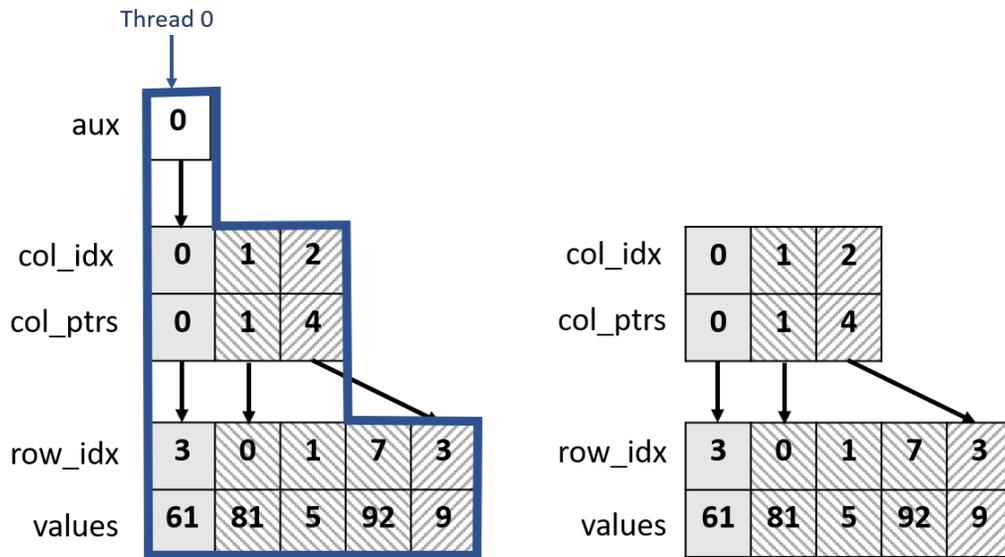


Figure 2.3: Data processed by the first thread, which is in-fact a CSC representation of part of our example matrix (with an additional column indices array)

a vector unit with a vector-length of two elements. However, for efficient use of vector-machines these concepts should be applied with longer vector lengths.

Simple Scalar Processing

In simple scalar processing, the DCSC data structure is traversed by following the pointers in their original order. This implementation traverses the elements of the matrix in a column-major order.

```

for p in 0 to num_column_partitions
  for j in col_starts[p] to col_starts[p+1]
    col_index = col_indices[col_starts[p] + j]
    for nz_idx in col_ptrs[j] to col_ptrs[j+1]
      row_index = row_indices[nz_idx]
      val = values[nz_idx]
      do something with val, row_index and col_index

```

And specifically for SpMV:

```

for p in 0 to num_column_partitions
  for j in col_starts[p] to col_starts[p+1]
    col_index = col_indices[col_starts[p] + j]
    for nz_idx in col_ptrs[j] to col_ptrs[j+1]
      row_index = row_indices[nz_idx]
      A_val = values[nz_idx]
      X_val = x_vec[col_index]
      Y_val = y_vec[row_index] + X_val*A_val
      y_vec[row_index] = Y_val

```

Analogously, for the DCSR representation, the SpMV will be implemented as:

```

for p in 0 to num_row_partitions
  for i in row_starts[p] to row_starts[p+1]
    row_index = row_indices[row_starts[p] + i]
    for nz_idx in row_ptrs[i] to row_ptrs[i+1]
      col_index = col_indices[nz_idx]
      A_val = values[nz_idx]
      X_val = x_vec[col_index]
      Y_val = y_vec[row_index] + X_val*A_val
      y_vec[row_index] = Y_val

```

Virtual Processors View

A popular way of thinking of the parallel nature of vector machines is as multiple concurrent "virtual processors" [60, 6]. Since a CSC matrix data-structure is composed of two arrays, the virtual processors can operate in-parallel either on the pointers array, or on the values array. In graph-processing terms, these two approaches have been described in [25] as the node-parallel

approach and the edge-parallel approach. This work attempts to apply these approaches by comparing two vectorizing techniques: the first technique, nicknamed "packed-stripmining", attempts parallel processing of the pointers array elements (node-centric). The second technique, known as "loop-raking", focuses on parallel processing of the values array (edge-centric). Note that "packed stripmining" and "loop raking" are not complementary approaches used together, but rather alternative approaches to parallelizing the same problem across different parts of the data-structure.

Packed-Stripmining

Stripmining is a common technique for vectorization of dense loops using vector-length-agnostic code. This means that the code is not aware of the size of the hardware vector registers during compile-time. Therefore, a stripmining loop attempts to configure the maximum possible vector length, and treats the accommodated vector length as a variable. The stripmining loop then "strips" a layer of the actual vector register length, and repeats the process for the remainder. Basic stripmining is templated as follows:

```

        source_addr = source_base_addr
        dest_addr = dest_base_addr
        req_vl = total_num_elements
stripmine: vl = set_vlr(req_vl)
        ...
        load vl elements into vector register from source_addr
        vector operations over vl elements
        store vl elements from vector register to dest_addr
        ...
        source_addr = source_addr + vl*(elem_size)
        dest_addr = dest_addr + vl*(elem_size)
        req_vl = req_vl - vl
        if req_vl>0 then goto stripmine

```

However, as the template above demonstrates, stripmining works best on a continuous array of elements in order to exploit parallelism efficiently. In the case of a CSC sparse matrix representation, the imbalance of the number of non-zero elements in each sparse column of a CSC matrix requires additional manipulation for efficient stripmining. The progress of the stripmining loop over the pointers array depends on the number of non-zero elements each pointer in the pointers array is pointing to. This imbalance results in idle processing elements, waiting for the "worst case" virtual processor to finish. A possible solution to this scenario is to pack only "active" (non-idle) pointers from the pointers array. We therefore introduce the "Packed-Stripmining" approach. The "Packed-Stripmining" approach for CSC matrices "packs" an array of unbalanced column pointers into a dense array, at the cost of using control-flow within each iteration of the stripmining loop. By re-packing the column-pointers every iteration of the vector-processing loop, this "pseudo-stripmining" loop can operate on the packed array as it would commonly operate in

balanced dense scenarios. Figure 2.4 illustrates the traversal order of the virtual processors across the CSC data-structure using this approach.

When composing together the thread-level parallelization level across DCSC partitions with the vector-level parallelization approach within partitions, the implementation of nested-parallelism SpMV on a DCSC representation using packed-stripmining would resemble the following template:

```

for p in 0 to num_column_partitions
  initialize packed_tracker[0:vector_length]
  initialize packed_size_tracker[0:vector_length]
  initialize packed_col_idx[0:vector_length]
  initialize track_idx=row_starts[p]
  initialize done=False
  while done==False
    for j in 0 to vector_length
      nz_idx = packed_size_tracker[j] - packed_tracker[j]
      col_index = packed_col_idx[j]
      A_val = values[nz_idx]
      row_index = row_indices[nz_idx]
      X_val = x_vec[col_index]
      Y_val = y_vec[row_index] + X_val*A_val
      y_vec[row_index] = Y_val
      packed_tracker[j]--
    for j in 0 to vector_length
      while (packed_tracker[j] <= 0 && track_idx < col_starts[p+1])
        packed_col_idx[j] = col_idx[track_idx]
        packed_tracker[j] = col_ptrs[track_idx+1] - col_ptrs[track_idx]
        packed_size_tracker[j] = col_ptrs[track_idx+1]
        track_idx++
    if all elements in packed_tracker are 0, then done=True

```

Similarly, the implementation of nested-parallelism SpMV on a DCSR representation using packed-stripmining will use the same concepts by swapping equivalent row and columns variables:

```

for p in 0 to num_row_partitions
  initialize packed_tracker[0:vector_length]
  initialize packed_size_tracker[0:vector_length]
  initialize packed_row_idx[0:vector_length]
  initialize track_idx=row_starts[p]
  initialize done=False
  while done==False
    for j in 0 to vector_length
      nz_idx = packed_size_tracker[j] - packed_tracker[j]
      row_index = packed_row_idx[j]
      A_val = values[nz_idx]
      col_index = col_indices[nz_idx]
      X_val = x_vec[col_index]
      Y_val = y_vec[row_index] + X_val*A_val
      y_vec[row_index] = Y_val
      packed_tracker[j]--
    for j in 0 to vector_length
      while (packed_tracker[j] <= 0 && track_idx < row_starts[p+1])
        packed_row_idx[j] = row_idx[track_idx]
        packed_tracker[j] = row_ptrs[track_idx+1] - row_ptrs[track_idx]
        packed_size_tracker[j] = row_ptrs[track_idx+1]
        track_idx++
    if all elements in packed_tracker are 0, then done=True

```

Note, that the packing stage itself (the second internal **for** loop) cannot be vectorized due to the **while** loop which is nested within it. Conditional **while** loops break the vectorization (or

result in an inefficient implementation), and therefore the packing process was separated from the vectorized chunk.

Furthermore, while this approach is designed to "fix" the problem of imbalances sparse matrices (and transitively, imbalanced and power-law graphs), this approach still encounters a difficulty if the imbalance is extreme (for example: one column has more elements than all other columns combined), or if there is significant imbalance towards the last rows/columns of the matrix. Each virtual-processor handles only one column (or "vertex" in the case of a graph). Therefore, if all the virtual-processors are done working, but there is one column with many elements that still need to be processed, this column will only be processed by a single virtual-processor while leaving the remaining virtual processors idle. This attribute likely has a negative impact on the performance of this method on power-law graphs, since there is no guarantee of the location of the populous vertices in power-law graphs.

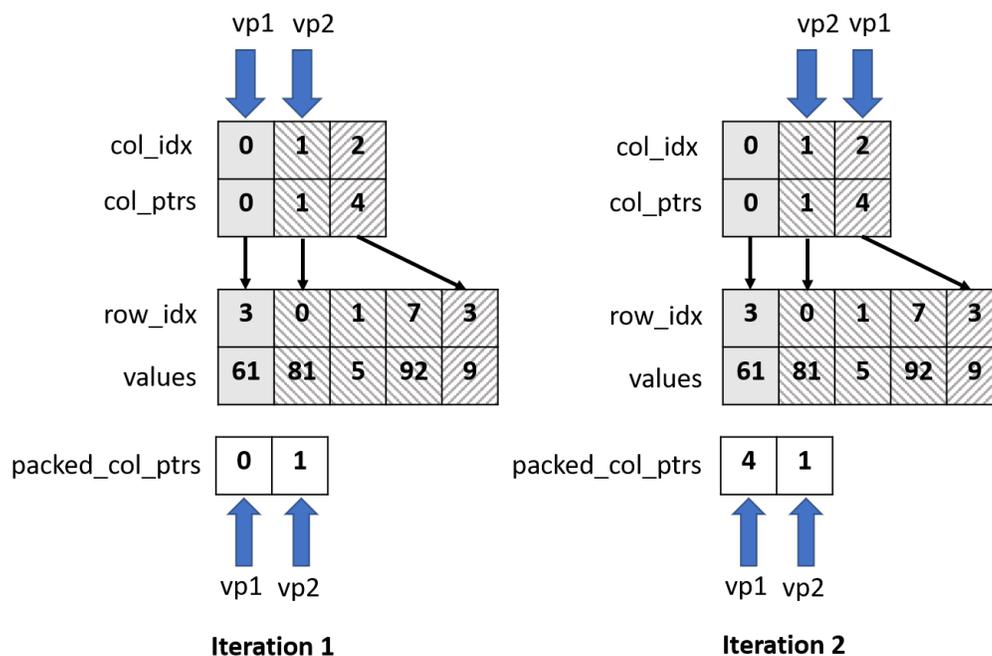


Figure 2.4: Illustration of the first two iterations of a packed-stripmining traversal of the CSC sparse matrix components from the first thread of the running example, using a hypothetical vector register length of 2

Loop-Raking

The Loop Raking vectorizing approach was originally proposed for sorting algorithms [60]. The raking access pattern is a common vector pattern used for two-dimensional data-structures [6]. It has been commonly used in dense data-structure scenarios such as dense matrix multiplication

and data compression. It allows contiguous elements to be processed by the same virtual processor, which may have implications regarding spatial data-locality, memory consistency and atomic operations. In the raking access pattern, virtual-processors process array-elements in intervals of *array_size/vector_length*.

This approach proposes a partial solution for the imbalance problems that appear in the packed-stripmining approach. In loop-raking, all virtual processors can be utilized in every iteration of the loop (with the possible exception of the last iteration). Since the vectorization is performed across the values array rather than the pointers array, loop raking results in an inherently more load-balanced scheme when performing an SpMV. Figure 2.5 illustrates the traversal order of the virtual processors across the CSC data-structure. Nevertheless, checking row sizes and boundaries is still required in order to have full information about each matrix element for the purposes of linear algebra operations. Therefore, unlike the original loop-raking use-cases, a "tracker" vector register is still required in the sparse matrix case in order to maintain information about progress through each column in a CSC structure. This tracker vector somewhat limits the possible load-balancing, since the current implementation under evaluation in this work chooses to define the rake interval as the size of the largest column. Therefore, while loop-raking resolves the utilization problem of processing a large row at the tail-end of the matrix, it does not solve the problem of an extremely large column which composes the majority of elements in the matrix (as may be the case in a power-law graph).

When composing together the thread-level parallelization level across DCSC partitions with the vector-level parallelization approach within partitions, the implementation of nested-parallelism SpMV on a DCSC representation using loop-raking would resemble the following template:

```

for p in 0 to num_column_partitions
  rake_interval = sizeof(A_val) / vector_length
  initialize rake_col_ind[0:vector_length]
  initialize rake_col_ind_ptr[0:vector_length]
  initialize tracker[0:vector_length]
  for offset in 0 to (nnz / vector_length)
    rake_offset = col_ptrs[row_starts[p]] + offset
    for j in 0 to vector_length
      if tracker[j] == 0
        rake_col_ind_ptr[j]++
        rake_col_ind[j] = col_indices[rake_col_ind_ptr[j]]
        tracker[j] = col_ptrs[rake_col_ind_ptr[j]+1]-col_ptrs[rake_col_ind_ptr[j]]
        col_index = rake_col_ind[j]
        nz_idx = rake_offset + j*rake_interval
        A_val = values[nz_idx]
        row_index = row_indices[nz_idx]
        X_val = x_vec[col_index]
        Y_val = y_vec[row_index] + X_val*A_val
        y_vec[row_index] = Y_val
        tracker[j]--

```

Similarly, the implementation of nested-parallelism SpMV on a DCSR representation using loop-raking will use the same concepts by swapping equivalent row and columns variables:

```

for p in 0 to num_row_partitions
  rake_interval = sizeof(A_val) / vector_length
  initialize rake_row_ind[0:vector_length]
  initialize rake_row_ind_ptr[0:vector_length]
  initialize tracker[0:vector_length]
  for offset in 0 to (nnz / vector_length)
    rake_offset = row_ptrs[row_starts[p]] + offset
    for j in 0 to vector_length
      if tracker[j] == 0
        rake_row_ind_ptr[j]++
        rake_row_ind[j] = row_indices[rake_row_ind_ptr[j]]
        tracker[j] = row_ptrs[rake_row_ind_ptr[j]+1]-row_ptrs[rake_row_ind_ptr[j]]
        row_index = rake_row_ind[j]
        nz_idx = rake_offset + j*rake_interval
        A_val = values[nz_idx]
        col_index = col_indices[nz_idx]
        X_val = x_vec[col_index]
        Y_val = y_vec[row_index] + X_val*A_val
        y_vec[row_index] = Y_val
        tracker[j]--

```

Note that in the pseudo-code examples above, several checks for corner-cases are omitted (padding, and checking edge conditions).

In the packed-stripmining approach, the "virtual processors" process the packed array, and perform checks to track the progress of elements in the non-zero elements array. This is as opposed to the loop-raking approach, in which the "virtual processors" process the non-zero elements array (both the indices and the values), while performing checks to track the status of the pointers array.

Due to the increased use of constant-stride loads and stores, loop-raking allows a reduction in the number scatter/gather operations, and it supports the systolic bank execution model. This makes it a good fit to the nested-parallelism approach with vector-machines. However, given Hwacha's single address-generation unit per lane, non-unit-stride loads and stores are serialized in a similar manner to scatter/gather operations.

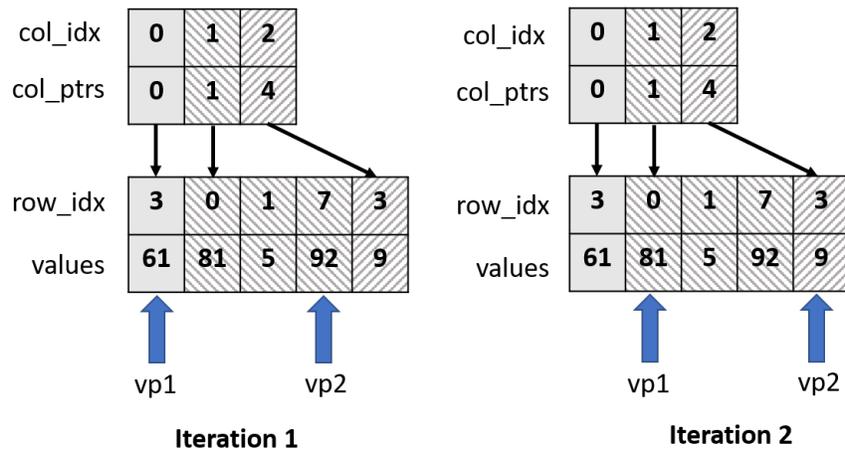


Figure 2.5: Illustration of the first two iterations of a loop-raking traversal of the CSC sparse matrix components from the first thread of the running example, using a hypothetical vector register length of 2

Chapter 3

Experimental Setup

3.1 Graph Processing Framework Infrastructure

GraphMat [56] was chosen as the base graph-processing framework infrastructure in this work for several reasons. GraphMat uses DCSC and DCSR data-structures to represent the graph adjacency matrices. Not many frameworks use this data-structure, which provides a natural boundary between the external parallelism abstraction and the internal parallelism abstraction used in nested-parallelism. Since common graph processing benchmark data-sets are usually provided in edge-list format, the use of the GraphMat infrastructure abstracts away the complications of constructing the efficient DCSC and DCSR graph representations out of these edge-list formats.

Additionally, the use of the linear-algebra representation in the back-end for the implementation of the graph-processing kernels provides a rigid structure. This means that the results and conclusions of this work can likely be transferred to other graph processing kernels implemented in GraphMat, since they all utilize similar ordering and execution patterns. Furthermore, the conclusions can potentially be generalized to other sparse linear algebra problems that are not related to the graph-processing domain.

Finally, GraphMat uses bit-vectors in-order to help represent sparse vectors. The use of bit-vector is very similar to the use of vector predicate registers in the Hwacha vector accelerator. While the Hwacha architecture is not able to load bit-vectors directly into predicate registers, this implementation is still helpful for it's equivalent representation.

At the time of selection, GraphMat was one of the fastest shared-memory graph-processing frameworks published in the academic community. It has been used for a variety of experiments and workloads, including in the architecture research community [19], and has proven to be consistently high-performing while maintaining it's unique abstractions.

3.2 Agile Hardware Development

This work was performed as part of an attempt in implementing the Agile Hardware Development approach described in [35]. It was done as part of the design process for a test SoC (named

EAGLE), developed as part of the agile hardware development methodology research programs. The nested-parallelism model described in this work is based on the micro-architecture of this test SoC: a chip multi-processor (CMP) with eight general purpose in-order scalar processors and 8 Hwacha vector-accelerators arranged into four clusters (each cluster consisting of two Rocket cores and two Hwacha Vector Accelerators with a shared L2 cache) with a 3-level cache-hierarchy.

In accordance in the Agile Hardware development approach, these software benchmarks were initially developed and verified using the Spike RISC-V ISA simulator, and then verified and evaluated during design space exploration against the RTL design using the MIDAS and FireSim FPGA simulation platforms. Nevertheless, due to a variety of technical and scheduling constraints, the FPGA-based evaluation was performed only after the chip was taped-out, and therefore was not able to contribute to the micro-architectural optimization of the chip as envisioned in the full agile hardware development manifest. However, the conclusions of this work provide valuable insights to micro-architectural choices for future chips.

3.3 Software Development

As PageRank can be implemented in its simplest form as an iterative SpMV, the experimentation with PageRank did not require many of GraphMat's advanced features of transforming the vertex-centric abstraction to the linear-algebra backend. Therefore, the first step in the development was the isolation of the GraphMat execution kernel from the supporting transformation mechanism, which will allow for focusing on the main PageRank kernel rather than the surrounding features of the framework. This isolation was verified by running several benchmarks and comparing the results with the full-featured GraphMat framework.

As a result of the isolation the GraphMat transformation mechanism, the main remaining features of GraphMat are its graph data-structures and data-ingestion components. As mentioned previously, many graph-processing frameworks maintain both a CSR and CSC representation of the graph adjacency matrix and its transpose for fast access to both outgoing edges and incoming edges [18]. Notably, the PageRank SpMV operates on the transpose of the graph adjacency matrix due to its focus on incoming edges. Therefore, the vectorized SpMV implementation actually operates on the DCSR representation of the transposed adjacency matrix, rather than the DCSC representation of the original adjacency matrix. Luckily, as described throughout the previous chapter, the DCSR SpMV implementations are analogous to the DCSC implementations, and only require switching between the different array pointers.

Next, SpMV assembly kernels were written based on the structure of the GraphMat DCSR data-structures. The kernels were initially tested as single iterations of SpMV. After verification of successful single iterations, the additional elements of PageRank were added to the SpMV kernel - division by vertex degree, etc. The final segment of the custom assembly code is a function that applies the damping factors for the PageRank values, and checks for convergence of the PageRank values.

The convergence check requires the PageRank values of all the vertices to be under a convergence threshold. One challenge when writing the PageRank assembly kernel involved determining

the PageRank convergence using vectorized vector-fetch code rather than using a scalar loop. This convergence indication using a decoupled vector unit requires passing a signal or flag between the vector-fetch unit and the scalar processor. Since there is no dedicated method for this type of operation in the Hwacha vector architecture, this issue was mitigated by setting an index vector, and using the `vfirst` instruction to return a non-zero value from the index vector if a value which did not converge (otherwise a default value of 0 is returned). The result of `vfirst` is then stored in a pre-determined shared memory location used to pass the value from the vector-fetch block execution to the scalar processor for further processing.

From an optimization perspective, it is important to note that GraphMat represents the vertex properties using an array of records, in which each record includes the vertex PageRank value and the degree of the vertex (rather than an array of PageRank values and an additional array of vertex degrees). This representation has an impact both on the number of instructions in the assembly code, and on the spatial-locality of the memory accesses for the relevant values. This representation was not changed in this work, in order to attempt to maintain the generality of the results to other potential GraphMat workloads.

Finally, OpenMP pragmas were added around the external `for` loops that call the SpMV assembly kernel in order to exploit the multi-level nested parallelism. OpenMP is a common method for applications to exploit nested parallelism. However, it requires careful implementation [57]. The use of OpenMP required a full Linux stack, as opposed to the bare-metal testing that has been traditionally attempted with the Hwacha RTL implementation. This could possibly expose issues of memory consistency if threads running Hwacha vector-fetch code are preempted and replaced with other threads running Hwacha vector-fetch code. Vector-fetch blocks currently do not support precise exceptions and Linux context switching. This issue is avoided during testing and evaluation by pinning threads to cores and not running more threads than the actual number of hardware threads in the system.

3.4 Validation and Verification

Development and functional verification were performed using the Spike RISC-V ISA Simulator with Hwacha vector extensions. Spike allows us to verify the functional correctness of the code, under the mitigating assumptions of an IPC of 1, and single-cycle memory access latency.

Initial verification was performed by comparing the converged PageRank results between the vectorized version and the simple scalar processed version (using the original GraphMat kernel). This approach indeed worked well for verifying the packed-stripmining approach. However, this trivial verification approach did not work for the loop-raking approach due to floating point rounding differences resulting from the different accumulation order used in the loop-raking approach (compared to simple-scalar and packed-stripmining). Therefore, the loop-raking code was verified by identifying mismatched results, and verifying these results by re-ordering the partially-accumulated temporary sums. These floating point mismatches proved to be an important consideration when evaluating different vectorization techniques.

3.5 Performance Evaluation and Design Space Exploration

Evaluation and design space exploration are based on the Rocket Chip SoC generator with the Hwacha vector accelerator. The SoC setup includes configurations with single-core and dual-core Rocket-Chip in-order cores, each accompanied by various configurations of single-lane or dual-lane Hwacha vector accelerators. The SoC configurations included a memory hierarchy with 2 levels of cache, of which the vector-accelerator is connected directly to the L2 cache. The size of the L2 cache is also a configurable parameter across the test configurations.

Performance evaluation was executed using FPGA-accelerated cycle-exact simulation on the FireSim platform [29]. The FireSim platform allows for FPGA-accelerated cycle-exact simulation on the public cloud using Amazon Web Services (AWS) EC2 FPGA instances. This FPGA-accelerated simulation enables running application benchmarks on top of a fully functional Linux system in a cycle-accurate simulation with only a 500x slow-down compared to real time execution on actual taped-out silicon. Similar experiments would require multiple weeks using a standard software RTL simulator. Furthermore, the FireSim framework also includes elaborate memory models which can simulate a full DDR3 backing memory system and last-level caches (LLC) with high accuracy timing models, while maintaining the performance level of FPGA-accelerated simulation [11].

FPGA-accelerated simulation allows for the use of actual silicon-worthy RTL for design space exploration, while still maintaining high simulation speed. The use of the production-quality Hwacha RTL means that there is a single-source-of-truth for test-chip production as well as simulation. This single-source-of-truth allows for bridging the modeling gap between high level simulation and silicon implementation. High simulation speed is especially important for the nested-parallelism experiments in this work, since the use of standard OpenMP programming interfaces requires a full operating system with resource management capabilities (for example, Linux). Running the experiments used in this reports on standard RTL software simulation would take multiple days to multiple weeks.

Since we use the actual processor and vector accelerator RTL to perform application-level evaluation in FireSim (rather than high-level abstract processor models), this RTL-based evaluation was able to expose Hwacha RTL bugs that were previously unknown due to un-exercised codes paths involving predicated instructions and atomic memory operations. These bugs were not represented in the functional ISA-level simulation model, and therefore required careful RTL debugging procedures. These issues were identified and mitigated through both RTL fixes and assembly kernel fixes. This demonstrates the importance of full-system level testing and evaluation of hardware designs using a variety of target applications.

Finally, full Linux-based evaluation of the Hwacha micro-architecture faces difficulties involving Hwacha's inability to recover from a page-fault within vector-fetch kernels. This means that all memory accesses must be paged-in by the scalar processor before calling the vector-fetch code. While this requirement can be mitigated in software, it may have a performance penalty which needs to be considered.

Chapter 4

Evaluation and Design Space Exploration

4.1 Evaluation

Performance was measured on three sample graphs (table 4.1), selected from the Stanford Network Analysis Project [38]. The graphs were selected to represent different use-cases and characteristics, while still maintaining a size which allows for testing at reasonable times across the design-space. Other than the properties presented in table 4.1, some additional distinguishing characteristics between the graphs includes the ratio of edges-per-vertex: the wikiVote graph has an average of 15 edges per vertex, while the roadNet-CA has an average of 1.5 edges per vertex and the amazon0302 graph has an average of 5 edges per vertex. It is possible that these properties may have an impact on the performance of the packed-stripmining technique vs. the loop-raking technique.

Twelve different SoC hardware configurations were simulated, by varying the number of tiles, the number of vector accelerator lanes per tile, and the size of the L2 cache, as specified in table 4.2 (Note that a *tile* consists of a scalar core and vector unit. The vector-lanes count is per-vector unit). The simulations of all SoC configurations were run at a simulated SoC frequency of 1033 MHz. The backing-memory model used for the simulations was a DDR3 memory model with speed-grade of 14-14-14. Figure 4.1 shows block diagrams of the evaluated SoC configurations.

Performance was also evaluated using an additional software parameter which controls the number of DCSR partitions in relation to the number of hardware threads. This DCSR partition factor is multiplied by the number of hardware threads to determine the number of overall DCSR

Table 4.1: Properties of Evaluation Graphs

Name	Vertices	Edges	Description
wikiVote	7115	103689	Wikipedia who-votes-on-whom network
roadNet-CA	1965206	2766607	Road network of California
amazon0302	262111	1234877	Amazon product co-purchasing network from March 2 2003

Table 4.2: Simulated SoC Hardware Configurations

Name	Tiles	Vector Lanes	L2 Cache Size
T1L1C512	1	1	512 KB
T1L1C1024	1	1	1024 KB
T1L1C2048	1	1	2048 KB
T1L2C512	1	2	512 KB
T1L2C1024	1	2	1024 KB
T1L2C2048	1	2	2048 KB
T2L1C512	2	1	512 KB
T2L1C1024	2	1	1024 KB
T2L1C2048	2	1	2048 KB
T2L2C512	2	2	512 KB
T2L2C1024	2	2	1024 KB
T2L2C2048	2	2	2048 KB

partitions. For example, if the DCSR partition factor is 4, and the number of hardware threads is 2 (in a dual-tile configuration), then the graph DCSR representation will have 8 DCSR partitions. Since the OpenMP external parallelization scheme parallelizes across cores using units of DCSR partitions, increasing this factor increases the granularity of the dynamic allocation of partitions between cores. However, while this factor increases the dynamic allocation of kernels to hardware threads, it may also decrease vector lengths used in the vectorized code if a large number of partitions results in smaller graph sections per-partition.

Run-time results measured using the FireSim cycle-exact FPGA-accelerated simulations can be found in the appendix in tables A.1, A.2, A.3. The design space is analyzed using multiple cuts: The benefits of multiple tiles vs. multiple vector lane (figures 4.4,4.5), the benefits of L2 cache size compared to the number of tiles or vector lanes (figures 4.2, 4.3), and the effects of the software DCSR partition factors (figures 4.6, 4.7, 4.8, 4.9). The entire design space is evaluated on both nested-parallelism techniques presented previously (packed-stripmining and loop-raking).

The measurement results use relative-speedup and absolute-speedup as evaluation metrics. The term "relative speedup" is used to refer to the speedup of parallel-vectorized code relative to a parallel-scalar code implementation on the same hardware configuration. As an example, for a dual-tile with single vector-lane configuration such as T2L1C2048, the relative speedup of a loop-raking kernel is $\frac{scalar_time_T2L1C2048}{loop_raking_time_T2L1C2048}$. The term "absolute speedup" is used to refer to the speedup of a kernel (scalar or vectorized) compared to the scalar implementation on the minimal SoC configuration under evaluation (single scalar core, with an L2 cache size of 512 KB). As an example, for a dual-tile with single vector-lane configuration such as T2L1C2048, the absolute speedup of a loop-raking kernel is $\frac{scalar_time_T1L1C512}{loop_raking_time_T2L1C2048}$.

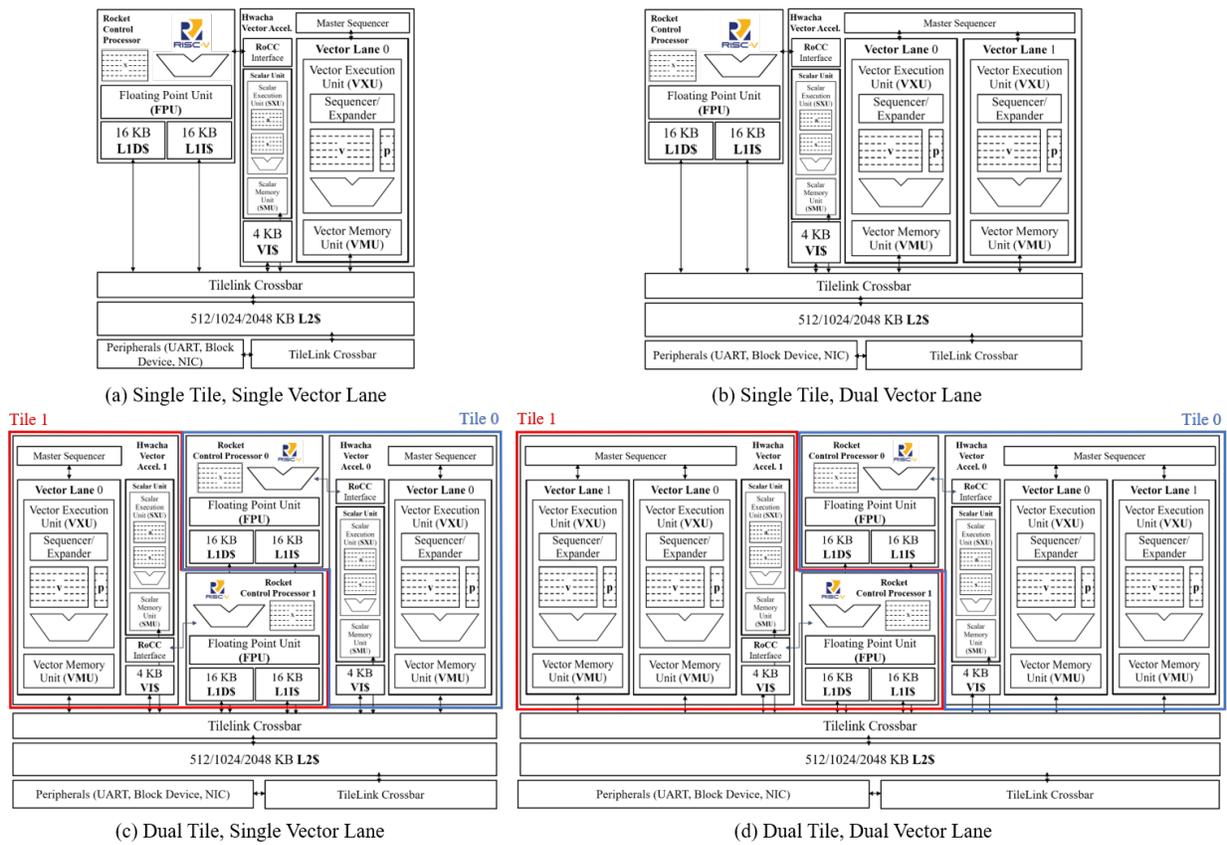


Figure 4.1: SoC configurations under evaluation.

The measured results present several interesting patterns. As mentioned previously, we attempt to address two different speedups when analyzing the results: the overall-speedup compared to a reference minimal scalar design (single-tile, L2 size of 512 KB), and relative-speedup compared to an equivalent design without a vector accelerator (i.e. a dual-tile-single-lane design would be compared against a scalar implementation of a dual-tile design with the same cache size)

4.2 L2 Cache Size

Unsurprisingly, different cache sizes have little to no effect on the performance of PageRank on all graph types in the various hardware configurations. This behavior is consistent both when varying the number of tiles and when varying the number of vector lanes. While the roadNet-CA and amazon0302 graphs are larger graphs which cannot fit in any of the L2 cache size configurations, the wikiVote graph is small enough to fit in all of the evaluated L2 cache configurations. Hence, it is not surprising that we do not observe changes in behavior across the evaluated L2 cache sizes for the wikiVote graph. At the same time, we also do not observe an improvement across different cache sizes for the larger graphs. This behavior is somewhat expected of graph workloads, which have been known to have poor spatial and temporal locality.

4.3 Total Number of Tiles

As expected, increasing the number of tiles improves the absolute performance of all graphs and software configurations compared to a minimal single tile configuration. The scalar reference implementation obtains near-linear scaling from a single tile to two tiles. However, when comparing the relative-speedup of the vectorized kernels versus the reference scalar kernel (figure 4.5), it is noticeable that the raking technique obtains a higher relative-speedup in the dual-tile case compared to the single-tile case. On the other hand, the packed-stripmining approach obtains the same, and sometimes even smaller, relative-speedup in the dual-tile case compared to the single-tile case. When considering the absolute-speedup between the single-tile and dual-tile case (4.4), the loop-raking technique obtains near-linear absolute scaling between one-tile to two-tiles, similar to the scaling of the scalar implementation. The packed stripmining approach presents less consistent scaling behavior, especially on the amazon0302 graph. We can conclude from these observations that the loop-raking method is more scalable, in relation to the number of tiles, than the packed-stripmining method.

4.4 Total Number of Vector Lanes

As expected, increasing the number vector lanes per tile generally improves the performance of most graph and software configurations, compared to a single-lane or single-scalar-tile configurations. Similarly to the multi-tile case, the loop-raking technique obtains a higher relative-speedup in the dual-lane case compared to a single-lane case. However, this observation is less-informative

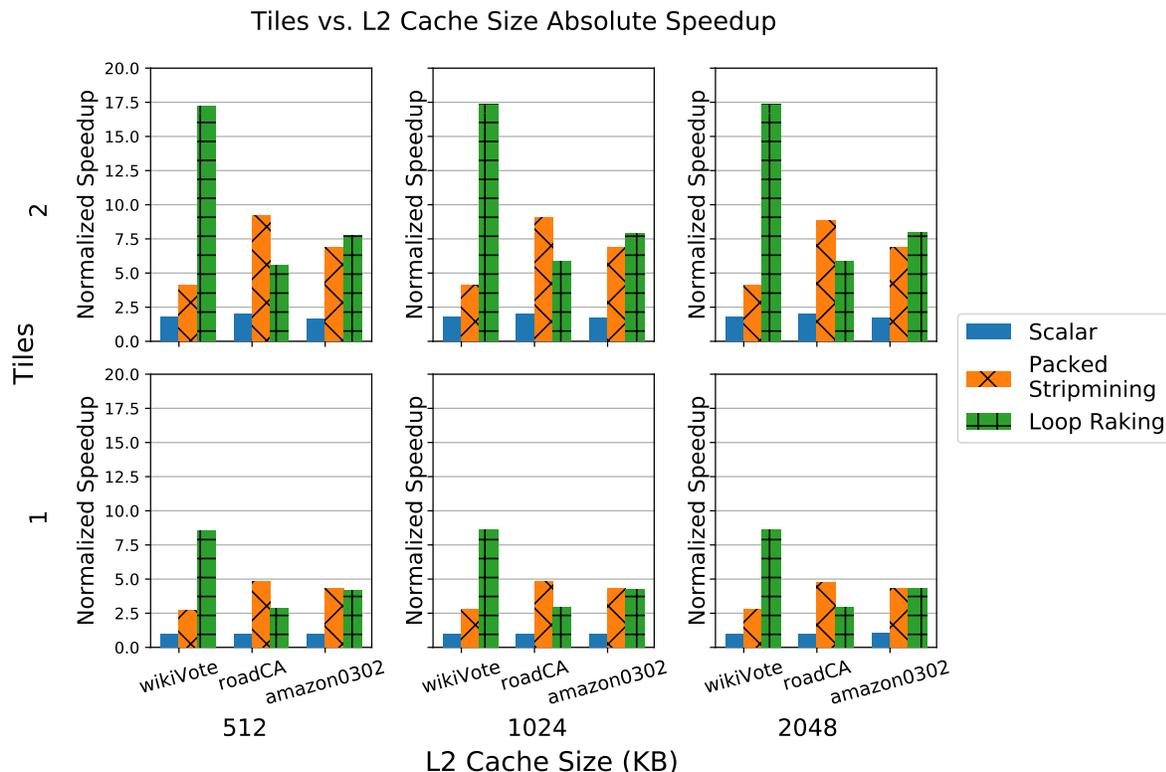


Figure 4.2: Tiles vs. L2 cache size comparison of average PageRank iteration speedup, normalized to the run-time of a minimal scalar hardware configuration (Single tile, 512 KB L2 Cache). To observe effects of number of lanes vs. L2 cache size, the results were cut with a single vector lane and with a software DCSR partition factor of 1 partition-per-hardware-thread.

than in the dual-tile case, since the multi-lane scenario relative-speedup is equivalent to the multi-lane absolute-speedup since there is only a single scalar core in both the single-lane and dual-lane cases. The absolute-speedup between the single-lane configuration to the dual-lane configuration does not scale as well as it did in the multi-tile comparison. For the packed-stripmining method, an additional lane provides a minimal speedup gain. Furthermore, the packed-stripmining implementation actually exhibits a smaller speedup in the dual-lane configuration on the wikiVote graph compared to the single-lane configuration. Nevertheless, it is clear that an additional lane indeed provides additional significant speedup for the loop-raking method.

4.5 Number of Tiles vs. Number of Vector Lanes

Given the area and power cost of additional tiles and lanes, it is interesting to investigate the trade-off between the two. We compare a single-tile-dual-lane design to a dual-tile-single-lane design. Both designs have a total of two vector lanes (albeit, split between two tiles versus concentrated

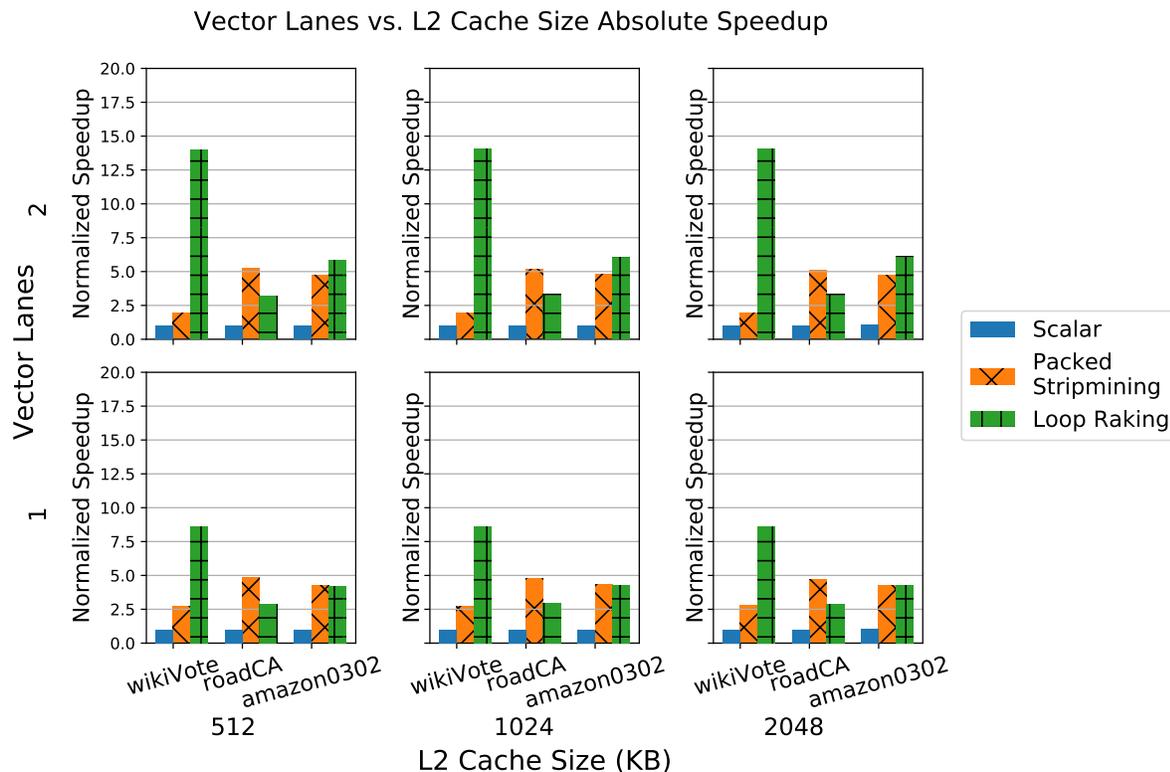


Figure 4.3: Vector lanes vs. L2 cache size comparison of average PageRank iteration speedup, normalized to the run-time of a minimal scalar hardware configuration (Single tile, 512 KB L2 Cache). To observe effects of number of lanes vs. L2 cache size, the results were cut with a single vector lane and with a software DCSR partition factor of 1 partition-per-hardware-thread.

in one tile), but the latter has an additional scalar control processor controlling the second vector lane. While the area comparison is not exact, we know from previous test-chips which include Rockets scalar processors and Hwacha vector accelerators [30] that the vector lanes dominate the area compared to scalar cores. When observing the normalized speedups in figure 4.4, it is clear that a dual-tile-single-lane design demonstrates a more significant speedup compared to the minimal scalar single-tile scalar design on all of the evaluated graphs. Figures 4.8, 4.9 show that these observations remain consistent across different software configurations as well. We can therefore conclude that multi-tile-single-lane configurations are likely a better choice for a PageRank workload (and perhaps sparse workloads in general) compared to single-tile-multi-lane configurations. Nevertheless, these observations need to be supported by supplemental energy and area simulations or measurements from a fabricated SoC.

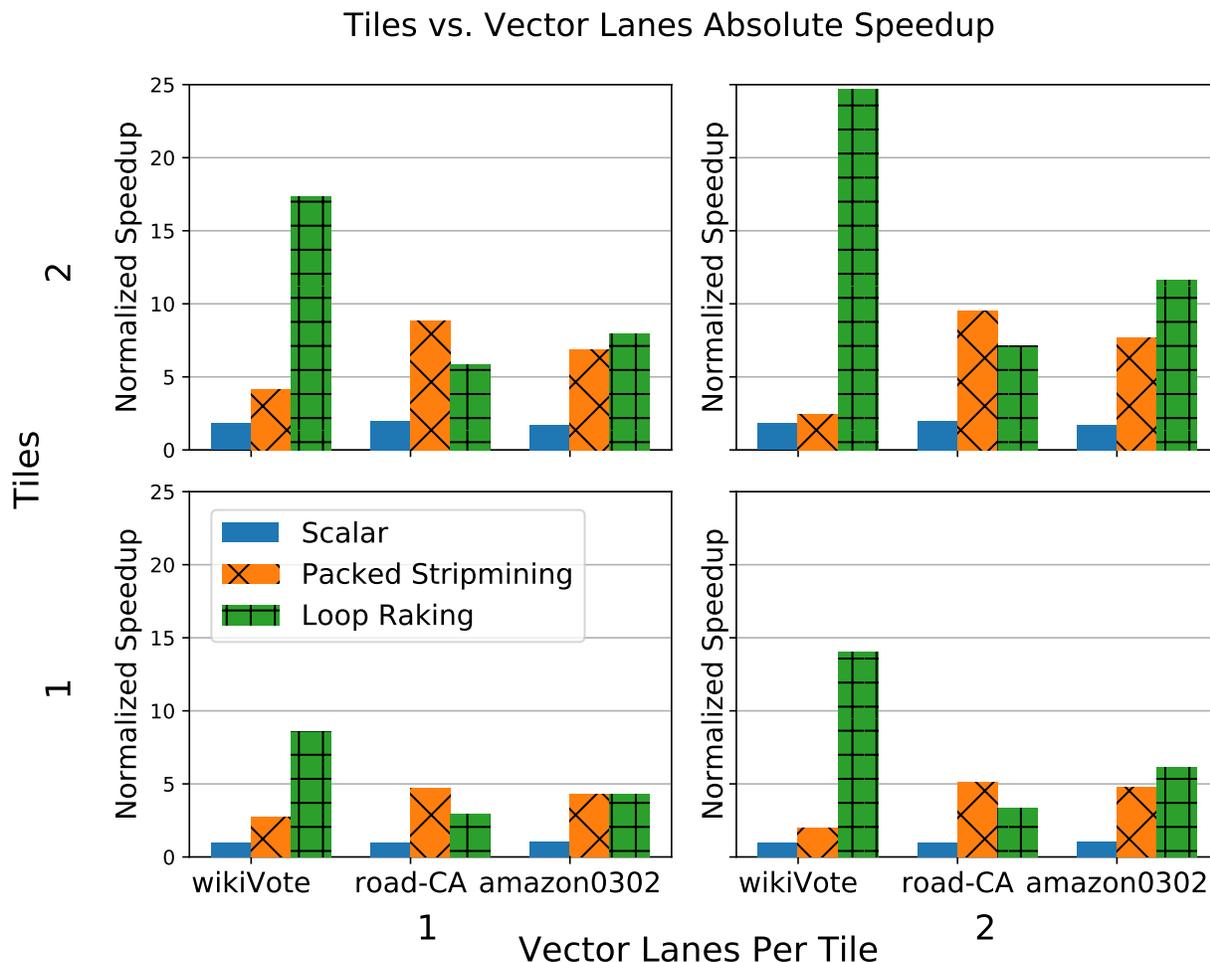


Figure 4.4: Tiles vs. vector lanes comparison of average PageRank iteration speedup, normalized to the run-time of a minimal scalar hardware configuration (Single tile, 512 KB L2 Cache). To observe effects of number of lanes vs. number of tiles, the results were cut with a cache-size of 2048KB and with a software DCSR partition factor of 1 partition-per-hardware-thread.

4.6 Packed-Stripmining vs. Loop-Raking

An initial observation of the measured results (figures 4.6, 4.7) shows that the best performing vectorized kernel depends on the choice of graph and the DCSR partitioning parameters. When observing the results with a DCSR partition factor of 1, we see that loop-raking outperforms packed-stripmining for the wikiVote and amazon0302 graphs, but performs worse in the roadNet-CA graph. However, further observation shows that the loop-raking speedup (both relative-speedup and absolute-speedup) improves as the DCSR partition factor increases for the roadNet-CA and amazon0302 graphs. Hence, for DCSR partition factors of 4, 8 and 16, loop-raking is able to

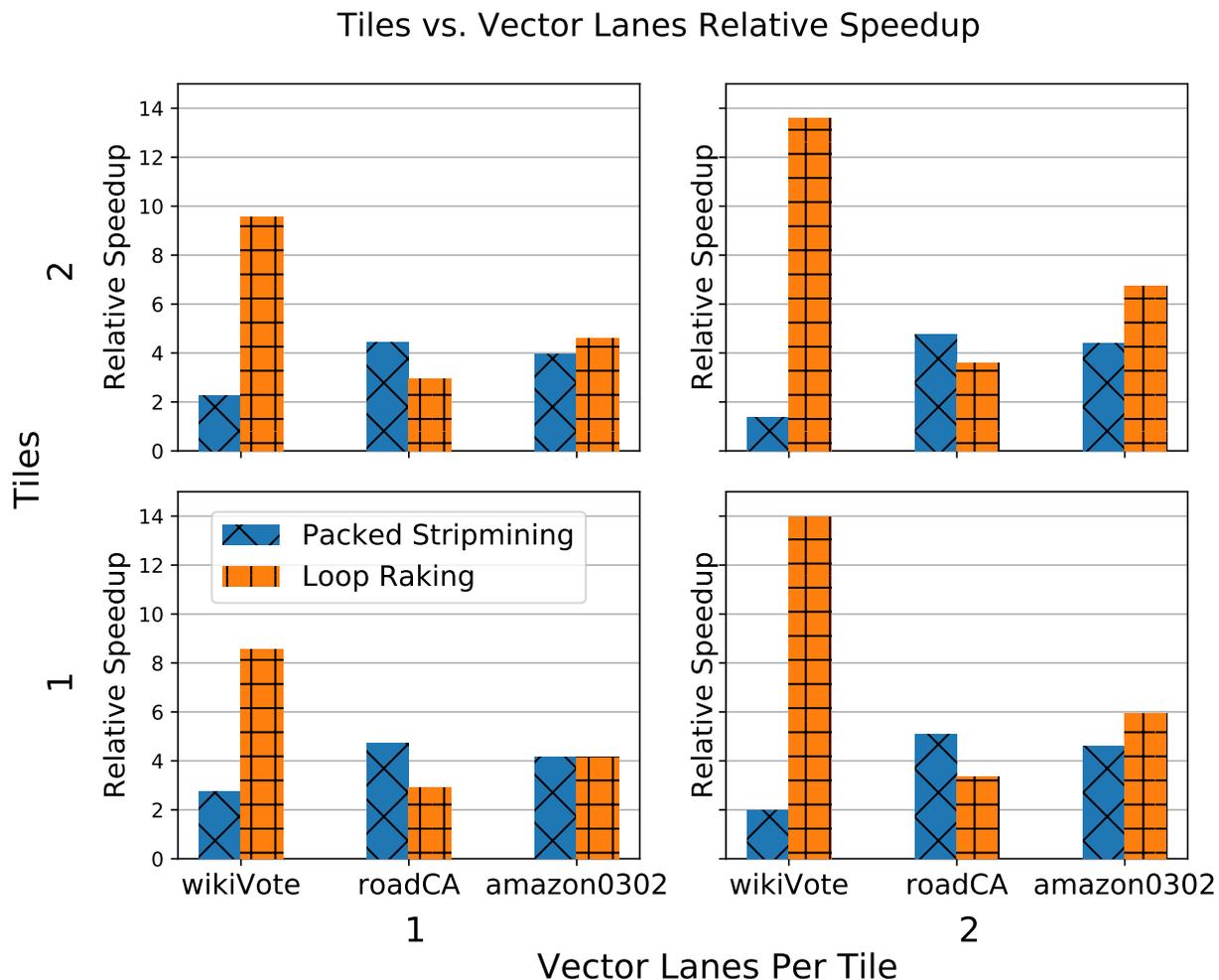


Figure 4.5: Tiles vs. vector lanes comparison of vectorized PageRank speedup relative to the scalar implementation on the equivalent SoC configuration. To observe effect of number of lanes vs. number of tiles, the results were cut with a cache-size of 2048KB and with a software DCSR partition factor of 1 partition-per-hardware-thread.

out-perform packed-stripmining for the roadNet-CA graph as well. Furthermore, the maximum observed speedups obtained by loop-raking (both relative-speedups and absolute-speedups) are significantly higher than the maximum observed speedups obtained by packed-stripmining (5.1x, 4.6x, 2.7x maximum relative speedups for the three graphs using packed-stripmining, vs. 7.3x, 9.2x, 13.9x maximum relative-speedups for the three graphs respectively using loop-raking). We can therefore conclude that when tuned correctly, loop-raking is generally a better choice of vectorizing kernel.

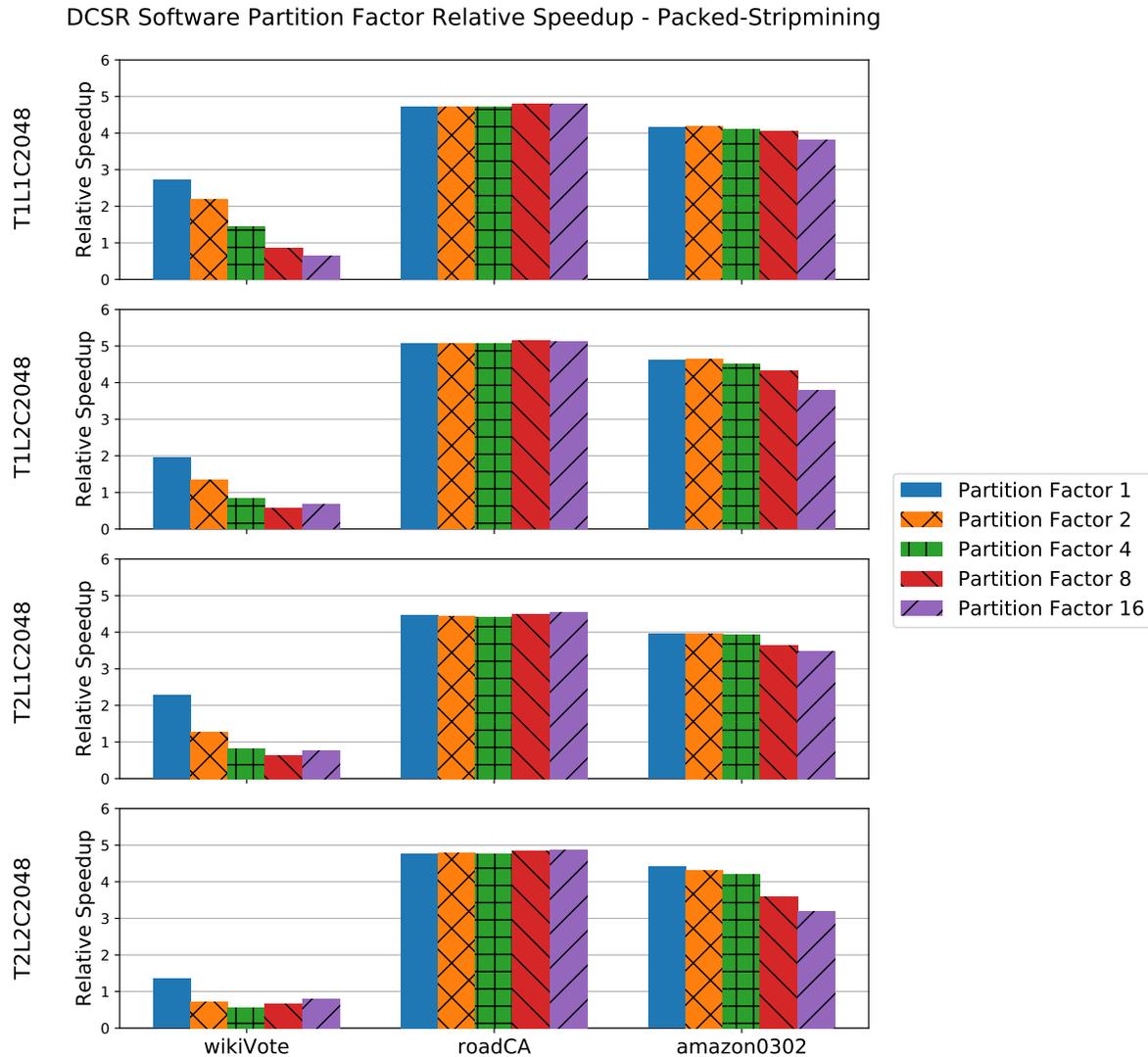


Figure 4.6: Software DCSR partitioning factor comparison of PageRank packed-stripmining speedup relative to the scalar implementation on the equivalent SoC configuration. Presented are results for the packed-stripmining vectorizing technique across four SoC configurations.

4.7 Graph Size and Structure

We analyze whether the size or characteristics of the graph structure have an impact on certain SoC configurations or software configurations. It is clear from figures 4.6,4.7,4.8,4.9 that the wikiVote graph presents a different behavior than the other two graphs under evaluation. As mentioned previously, wikiVote is the smallest graph under evaluation, and it is small enough to fit in the L2 cache size of all of the tested SoC configurations. Therefore, it is not bound by off-chip memory bandwidth, and it should be able to utilize all of the vector accelerator’s additional computational

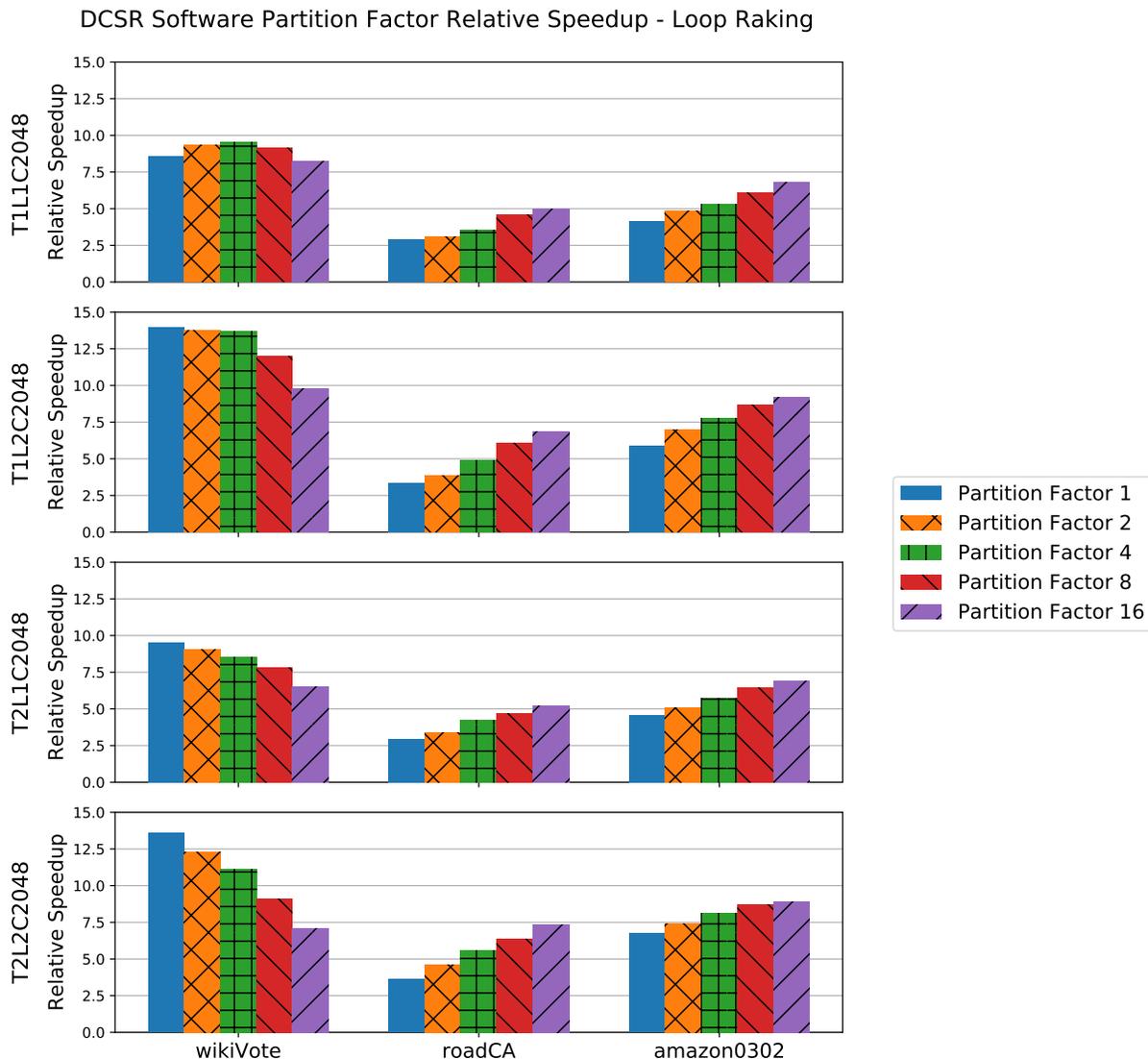


Figure 4.7: Software DCSR partitioning factor comparison of PageRank loop-raking speedup relative to the scalar implementation on the equivalent SoC configuration. Presented are results for the loop-raking vectorizing technique across four SoC configuration.

resources. This is also supported by the measurements, in which the wikiVote graph is able to utilize the additional computational resource and reach a relative-speedup of up to 14x, while the larger graphs that do not fit in the L2 cache and require interaction with the off-chip memory system are able to reach only up to a 9x relative-speedup with vectorized kernels.

The two larger graphs present better relative-speedups as the DCSR partition factor increases. This is not surprising, since a higher DCSR partition factor allows for finer-grained load-balancing of partitions between hardware threads. However, the DCSR partition factor was not expected

to have an impact on the single-tile configurations, since those configurations have only a single hardware thread. Furthermore, we also observe that the wikiVote graph presents smaller speedup with the loop-raking kernel as the DCSR partition factor increases. This is in contrast to the larger graphs which present higher speedup when the DCSR partition factor increases. In addition, when using the packed-stripmining method, the wikiVote graph actually exhibits a relative-slowdown when using higher DCSR partition factors. An initial suspicion in this case regards the vector lengths and their impact of the vector unit utilization. As the DCSR partition factor increases, the requested vector lengths decrease due to the smaller number of elements processed in each DCSR partition. However, the number of vertices and edges in all 3 graphs is significantly higher than the maximum vector length possible in both the loop-raking and packed-stripmining kernels under the evaluated SoC configuration. Based on the number of vertices and edges each of the evaluation graphs, we would expect that the active vector-length (AVL) for each configuration would be the maximum vector length (MVL) allowed by the vector unit configuration (which is determined by the number of vector registers required in the vectorized kernel). However, after further investigation, we found that for the wikiVote graph, the active vector length is less than the maximum vector length when the DCSR partition factors are 8 and 16 (for single-tile SoC configurations). The explanation for this apparent-contradiction is that while the number of vertices is significantly higher than $MVL \times \text{num_partitions}$, the de-facto compressed matrix size depends only on vertices that have outgoing edges. While there are overall 8000 vertices in the graph, only 2300 vertices have outgoing edges. Hence, in the cases of 8 and 16 partitions, $(2300/\text{num_partitions})$ turns out to be less than the maximum vector length allowed by the vector-unit register configuration. As a result, the vector unit is not utilized to the fullest extent. This situation is further exacerbated between the single-lane case and the dual-lane case: there is a smaller speedup for low DCSR partition factors, and an increased slowdown for higher DCSR partition factors. This indeed helps to provide an explanation for the behavior of the wikiVote graph results when using the loop-raking method: In the T1L1C2048 we observe higher speedups as the number of DCSR partitions increases until 8, 16 partitions in which we start observing smaller speedups. For the T1L2C2048, we start observing the lower utilization of the vector units starting with lower DCSR partition factors due to the use of two lanes. For the T2L1C2048 and T2L2C2048 configuration we observe the smaller speedup behavior in all DCSR partition factors since the actual number of partitions is double the partition factor (since the number of DCSR partitions is the DCSR partition factor times the number of hardware threads, hence resulting in double the number of DCSR partitions for dual-tile configurations).

However, the vector unit utilization does not provide a full explanation for the slow-down observed for the packed-stripmining measurements. It is important to note that the packed-stripmining approach involves a re-packing phase that is performed by the scalar processor after each stripmining iteration on the vector unit. Hence, there is a trade-off between the overhead of re-packing, and the benefits of the vector accelerator. Longer vector lengths enable better utilization of the vector units, but incur longer re-packing phases in the scalar processor. This trade-off may explain the behavior of packed-stripmining across different DCSR partition factors.

Nevertheless, to confirm these explanations regarding the run-time performance behaviors across different DCSR partition factors, further introspection and investigation are required.

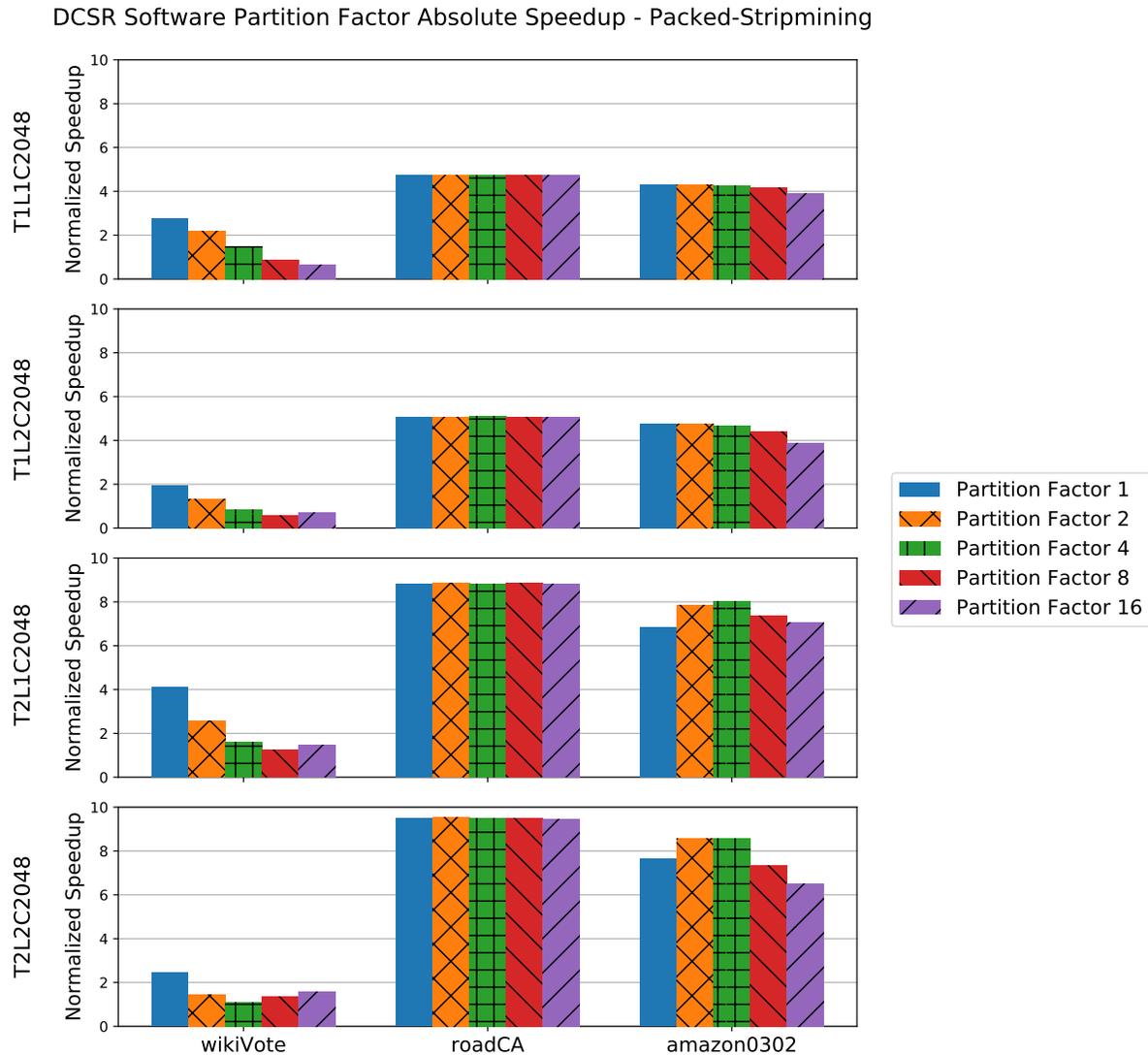


Figure 4.8: Software DCSR partitioning factor comparison of PageRank packed-stripmining absolute speedup compared to the run-time of a minimal scalar hardware configuration (Single tile, 512 KB L2 Cache). Presented are results for the packed-stripmining vectorizing technique across four SoC configurations.

4.8 Vector Accelerator vs. Multi-Core Scalar Processors

Another question of interest when addressing graph-processing and sparse workloads is regarding the benefit of data-level parallelism versus task-level parallelism. This question can be projected to the design space under evaluation by comparing the run-time of a scalar-parallel implementation to an equivalent vectorized implementation. It is important to note that while the scalar

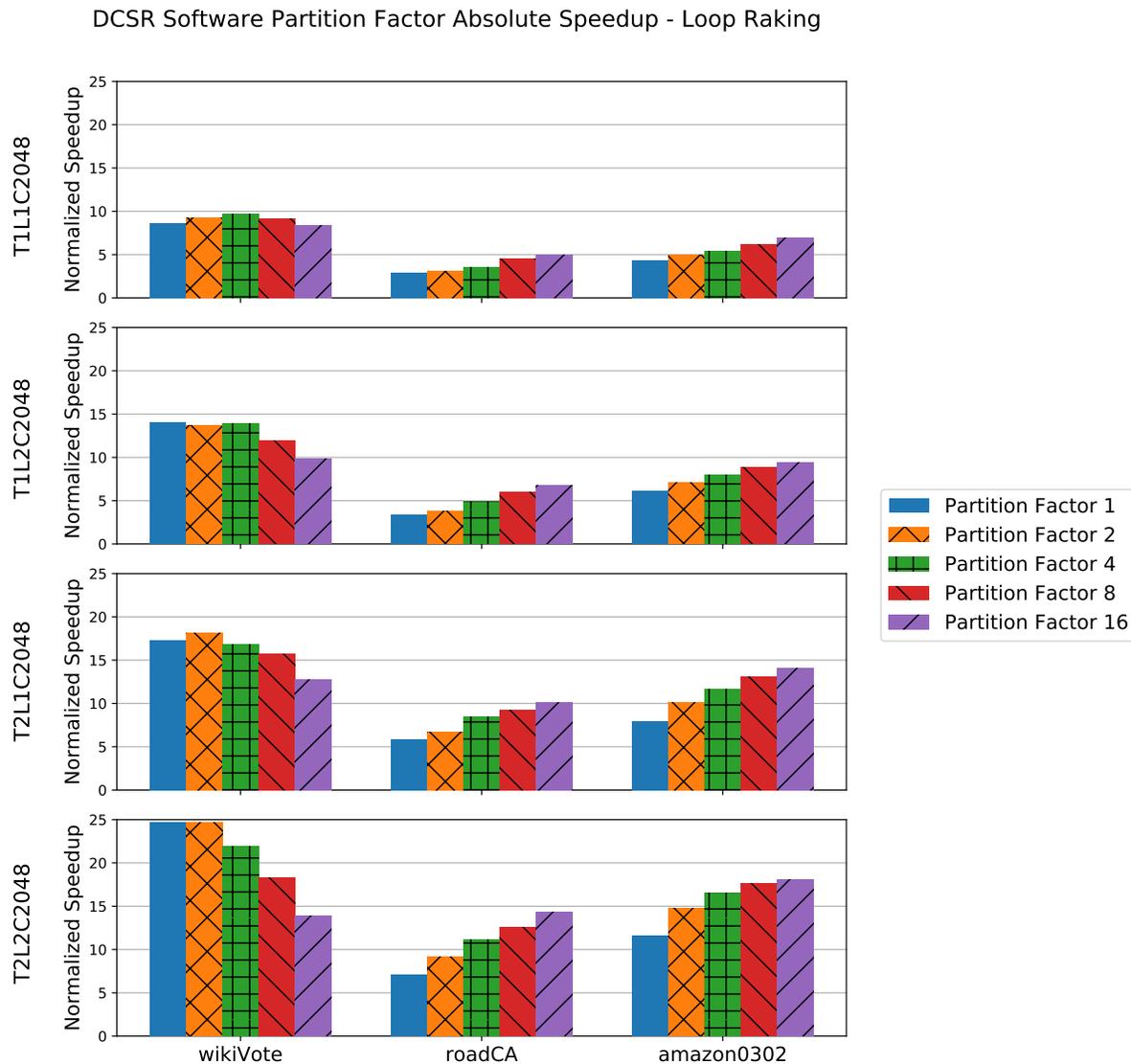


Figure 4.9: Software DCSR partitioning factor comparison of PageRank loop-raking absolute speedup compared to the run-time of a minimal scalar hardware configuration (Single tile, 512 KB L2 Cache). Presented are results for the loop-raking vectorizing technique across four SoC configuration.

implementation in this evaluation has a parallel dimension across DCSR partitions, this is only coarse-grained task-level parallelism. The internal loops of the scalar implementation were not optimized for task-level parallelism. Nevertheless, we can attempt to perform a coarse-estimate by observing the results of figure 4.4. We observe that the dual-tile configurations obtain a 2x speedup when using the scalar implementations compared to the single-tile scalar implementations. At the same time, we observe that a single-tile-single-lane vector accelerator obtains between 2.75-8.6x

speedup compared to the scalar implementation.

When analyzing the benefits of adding a vector accelerator for a sparse workload as opposed to adding a additional scalar cores, we must consider the number of additional functional units contributed by a vector accelerator over a scalar core. The Hwacha vector accelerator has four floating point functional units, while a Rocket scalar core has only one. We observe that the only cases where the vector accelerator obtains an absolute speedup lower than 4x are for the wikiVote graph in the packed-stripmining case, and for certain DCSR partition configurations of the CA-roadNet graph in the loop-raking case. Hence, it is reasonable to concluded that with the correct choice of software optimization, the vector accelerator can potentially achieve the desired speedup (greater than 4x) in all of the evaluated scenarios, and therefore data-parallel vector accelerators remain a valid choice for sparse and graph-processing workloads.

4.9 Bottlenecks

We attempt to analyze the potential performance bottleneck for the evaluated workloads. Hwacha has only a single address generation unit per lane, which can serialize indexed and non-unit-strided memory operations that are frequently used in sparse kernels. While the loop-raking kernel has a higher count of non-unit-stride memory operations compared to the packed-stripmining kernel, it is likely that the impact of this potential bottleneck is obfuscated by the scalar processing overhead of the packed-stripmining re-packing phase.

The initialization overhead of both kernels was eliminated as a potential bottleneck. The initialization overhead of both kernels was measured and found to be negligible compared to the body of the computation loops.

Further investigation of the reasons for the behavior of wikiVote require detailed Hwacha commit logs. Due to the length of simulation, these are difficult to obtain using standard RTL software simulation. However, new FPGA-based debugging features of the FireSim platform enable the extraction of such logs through FPGA-accelerated simulation. This will allow for further introspection and investigation for identifying the bottlenecks with higher confidence. After such an investigation, potential micro-architectural features, such as additional address-generation units, could be added to Hwacha to improve the run-time of sparse kernels using this vector architecture.

4.10 Related Hardware Improvements

In this work, a general-purpose vector accelerator was used to accelerate and improve the performance of a graph processing kernel. However, as the field of domain-specific acceleration is gaining momentum with the end of Dennard scaling and Moore’s law, graph processing has naturally taken a significant spotlight as a domain-specific research agenda [46, 19, 1, 2].

The desired outcome in an ideal situation would be to find a domain-specific solution that fits the entire ”graph processing” domain. However, it turns out that while graph processing problems have common data-structures, the computation functions do not have many common traits. Fur-

thermore, depending on the context, even the data-structure may not be a common characteristic: a dynamically updated graph may be represented as adjacency lists, while a static graph may be represented as a sparse matrix. Concretely, performing a Breadth First Search (BFS) to generate a spanning tree from a static graph has very few similarities to performing topic modeling using Latent Dirichlet Allocation (LDA) on a dynamic graph - not in data-structure, not in the amount of computation, and not in the level of parallelism. Hence, attempting to generalize an entire domain of graph processing problems into a single domain may not be beneficial for identifying possible hardware acceleration avenues.

Given a specific graph problem (such as finding a shortest path on a static graph), fixed-function hardware solutions such as [19, 46] can provide significant power and performance benefits. However, when the exploration within the graph processing domain is expanded beyond single fixed-function problems, it is difficult to identify hardware acceleration features that encompass the entire domain. One possible reason for this is that graph processing is characterized by data-representation rather than by computation kernels. Hence, most domain-wide improvements for graph processing have focused on the memory system - since they address the inherent property of a graph which is its data structure. [59] and [2] have proposed dedicated prefetchers for graph processing, while many of the proposals in [46] revolve around partitioning of the memory system. Nevertheless, by refining the definition of the graph-processing problem domain, and identifying sub-domains with particular representations and computation patterns, it may be possible to utilize more general purpose vector accelerators with particular memory system improvements (such as multiple address generation units) in order to meet the acceleration requirements for graph-related problems. As such, large static problems such as PageRank may be categorized as one sub-domain with a particular acceleration approach (perhaps a standard vector unit with memory system improvements may be enough), while dynamic shortest-path problems may be categorized differently and use a different representations and hardware acceleration semantics. Integration of some of the micro-architectural features presented in fixed-function graph-processing accelerators (such as prefetchers) into a general-purpose vector accelerator may provide the additional desired performance for these sub-domains. It is important to note that increasing the memory system's address bandwidth is typically expensive, and therefore these types of features require further investigation.

4.11 Generalization

This work examined the effect of explicit nested parallelization on PageRank. Since PageRank is implemented using a simple SpMV kernel, it is able to utilize the advantages of basic ALU primitives such as vectorized addition and multiplication. However, generalization attempts for graph processing, such as the GraphBLAS standard [31], assume that the basic algebraic addition and multiplication operations may be overloaded by alternative functions for the implementation of other graph processing algorithms such as BFS, SSSP, or CC. These alternative functions may include minimum/maximum, or other forms of reductions.

The RISC-V vector extension presents new challenges and opportunities in this context of

vector instructions and their generalization. The most recent working draft of the vector extension specification [5] provides an option for polymorphic vector instructions a custom vector data-types. These polymorphic vector instructions may allow for different instruction semantics depending on a currently configured type representation of the vector registers. Potential future extensions based on this vector extension may allow for graph-specific representations which may provide an efficient opening for the overloading of relevant instructions based on the GraphBLAS semantics. The use of overloaded vector instructions will not necessarily reduce the burden from compilers and hand-optimization of kernels, but it may allow for better generalizations, code generation, and integration with custom hardware based on common representations. Nevertheless, the RISC-V vector extension working draft proposal does not come without its challenges to sparse and graph-related workloads. The base vector extension proposal has defined only a single predicate vector register (implemented as the least significant bit of each element of the first standard vector register). This is a point of interest since predicated instructions are a significant factor in sparse vectorized workloads. The implications of this proposal may require additional instructions to compute and move masks into this predicate register, which may generate additional register pressure.

4.12 Future Work

A common assumption in graph processing research is that the computation of the graph kernel is the expensive computation component, and therefore it is the main problem that requires research attention. This comes under the assumption that once a graph data-structure is constructed, it will be used multiple times across various kernel, hence amortizing the cost of the data-structure construction. This assumption indeed hold for most cases of a PageRank, since it is an iterative kernel, and it takes many SpMV iterations for the kernel to converge. However, in some cases, the graph construction time may be the significantly longer. The wikiVote PageRank converges in a relatively small number of iterations (20-30). For the wikiVote graph on the T1L1C512 configuration, the graph construction time was 1238 ms, while the overall PageRank computation time until convergence was 615 ms for a scalar implementation and 71 ms for a loop-raking implementation. These demonstrate that the graph construction time may be the bottleneck in cases of non-iterative graph computation kernels. While compressed data-structure provide significant data-locality which improves the kernel computation time, the graph-construction time may render this irrelevant if the time to construct the optimized data-structure is not amortized. To provide a complete solution for different types of non-iterative graph processing kernel, further work is required to optimized the graph construction stage.

Additional hybrid vectorization approaches may help further mitigate load-balancing issues in power-law graphs that the loop-raking technique is still susceptible too. A hybrid approach may apply stripmining on large vertices, while using loop-taking for the remainder of the graph. However, this type of hybrid approach requires sorting the vertices based on vertex degrees, and therefore the sorting overhead must be studies against current loop-raking performance.

This evaluation in this work was limited by the size of the FireSim FPGA platforms. Hence, it was not possible to evaluate SoC configurations of four tiles, four lanes or beyond. To further

confirm or validate the results of this evaluation, we expect that the previously mentioned EAGLE SoC will be a useful platform. The EAGLE SoC was designed based on a similar Rocket Chip configuration consisting of eight tiles arranged in the form of four clusters, each cluster equivalent to the dual-tile-single-lane configuration evaluated in this work. The EAGLE SoC will allow for further evaluation and scaling of the results up to 8 tiles.

Finally, the root causes and reasons for the performance bottlenecks were not thoroughly confirmed given the possibilities of cycle-accurate evaluations. Further investigation of the bottlenecks will allow for additional performance optimization through optimized software pipelining, instruction ordering, and minor micro-architectural features. These optimizations can be achieved through instruction commit-log analysis of the target kernel simulated on the SoC configurations. Obtaining such commit-logs from software RTL simulation takes multiple weeks. Newly integrated micro-architectural introspection features of FireSim FPGA-accelerated simulation enable extracting these instruction commit logs within several hours. Extraction of these instructions and analysis of the logs will allow for better understanding of the relevant bottlenecks, and mitigation of those bottleneck through instruction ordering or micro-architectural features.

Chapter 5

Conclusion

This work presents SW/HW co-design space exploration and evaluation of a nested-parallelism PageRank graph processing kernel. The design space was evaluated using a variety of SoC configurations of the a Rocket multi-processors with Hwacha vector accelerators and multiple software configurations. This work demonstrated the benefits of the loop-raking vectorizing technique compared to the packed-stripmining vectorizing technique for a sparse data-structure representation likely due to the overhead of additional re-packing in the scalar-processor and longer vector lengths. Furthermore, this work demonstrated that using correct data-structure partitioning, the loop-raking vectorizing technique can achieve up to 14x relative-speedup compared to equivalent scalar implementations. A 25x speedup was demonstrated using dual-tile SoC with dual-lanes-per-tile vector accelerators, compared to a minimal single-tile scalar implementation, demonstrating the scalability of the proposed nested-parallelism techniques. The results of this design space exploration shed light on the preferred SoC configuration choice required for this type of vectorized nested-parallel sparse workload: Given fixed-area constraints, a dual-tile-single-lane configuration is a higher performing configuration compared to a single-tile-dual-lane configuration for nested-parallel sparse workloads. This work also demonstrated an implementation of agile hardware development methodologies with software development using functional simulators and design space exploration using FPGA accelerated simulation for performance evaluation. Further work will include final evaluation using a fabricated SoC, and further micro-architectural optimization using additional micro-architectural introspection features of FPGA-based performance evaluation tools. The key contributions of this work include the evaluation and comparison of vectorization techniques for an SpMV kernel on a novel vector architecture, as well as the evaluation methodology for accurate design space exploration using system-level applications.

Appendix A

Measurement Results

Table A.1: Measurements for the wikiVote Graph

Config	DCSR Partition Factor	Scalar SpMV (ms)	Packed-Strip. SpMV (ms)	Loop-Raking SpMV (ms)	Scalar PageRank Average Iteration (ms)	Packed-Strip. PageRank Average Iteration (ms)	Loop-Raking PageRank Average Iteration (ms)
T1L1C512	1	27.22	10.12	3.21	29.30	10.66	3.42
T1L1C512	2	27.55	12.88	2.98	29.63	13.56	3.18
T1L1C512	4	27.00	19.15	2.84	28.97	20.08	3.04
T1L1C512	8	27.52	32.71	3.05	29.57	34.26	3.23
T1L1C512	16	27.40	43.53	3.42	29.11	45.16	3.55
T1L1C1024	1	27.16	10.11	3.20	29.11	10.64	3.41
T1L1C1024	2	27.50	12.87	2.97	29.45	13.54	3.16
T1L1C1024	4	26.96	19.12	2.83	28.79	20.05	3.02
T1L1C1024	8	27.47	32.70	3.05	29.40	34.22	3.21
T1L1C1024	16	27.32	43.55	3.41	28.93	45.13	3.50
T1L1C2048	1	27.17	10.11	3.19	29.10	10.63	3.40
T1L1C2048	2	27.51	12.86	2.96	29.44	13.53	3.16
T1L1C2048	4	26.96	19.12	2.83	28.78	20.04	3.03
T1L1C2048	8	27.46	32.72	3.05	29.38	34.22	3.20
T1L1C2048	16	27.35	43.53	3.43	28.91	45.12	3.50
T1L2C512	1	27.22	14.18	1.96	29.29	14.88	2.10
T1L2C512	2	27.53	20.89	2.03	29.62	21.92	2.16
T1L2C512	4	27.00	33.37	2.02	28.96	34.88	2.12
T1L2C512	8	27.51	49.41	2.38	29.57	51.71	2.49
T1L2C512	16	27.41	40.92	2.91	29.09	42.42	3.02

T1L2C1024	1	27.16	14.16	1.95	29.11	14.86	2.09
T1L2C1024	2	27.51	20.87	2.01	29.45	21.89	2.14
T1L2C1024	4	26.96	33.34	2.00	28.79	34.85	2.10
T1L2C1024	8	27.47	49.41	2.37	29.40	51.68	2.45
T1L2C1024	16	27.34	40.95	2.94	28.92	42.38	2.97
T1L2C2048	1	27.17	14.16	1.96	29.10	14.86	2.09
T1L2C2048	2	27.51	20.88	2.02	29.44	21.89	2.14
T1L2C2048	4	26.96	33.35	2.00	28.78	34.84	2.10
T1L2C2048	8	27.46	49.43	2.39	29.39	51.67	2.45
T1L2C2048	16	27.36	40.94	2.94	28.92	42.38	2.97
T2L1C512	1	15.08	6.70	1.54	16.25	7.11	1.70
T2L1C512	2	13.61	10.96	1.49	14.72	11.55	1.65
T2L1C512	4	13.82	17.57	1.61	14.92	18.46	1.77
T2L1C512	8	13.73	22.36	1.77	14.72	23.26	1.91
T2L1C512	16	13.89	18.97	2.20	15.02	19.89	2.36
T2L1C1024	1	15.03	6.69	1.53	16.13	7.10	1.69
T2L1C1024	2	13.59	10.94	1.47	14.61	11.52	1.62
T2L1C1024	4	13.79	17.56	1.61	14.80	18.44	1.74
T2L1C1024	8	13.69	22.36	1.77	14.60	23.22	1.87
T2L1C1024	16	13.85	18.97	2.20	14.91	19.86	2.31
T2L1C2048	1	15.03	6.69	1.52	16.12	7.09	1.69
T2L1C2048	2	13.57	10.94	1.46	14.61	11.52	1.62
T2L1C2048	4	13.78	17.57	1.60	14.79	18.43	1.74
T2L1C2048	8	13.71	22.34	1.77	14.60	23.21	1.87
T2L1C2048	16	13.85	18.98	2.20	14.91	19.86	2.30
T2L2C512	1	15.05	11.35	1.10	16.25	11.95	1.22
T2L2C512	2	13.65	19.46	1.11	14.72	20.40	1.22
T2L2C512	4	13.82	26.11	1.25	14.91	27.35	1.37
T2L2C512	8	13.73	21.04	1.54	14.72	21.86	1.65
T2L2C512	16	13.89	17.99	2.02	15.02	18.83	2.16
T2L2C1024	1	15.04	11.32	1.09	16.17	11.91	1.18
T2L2C1024	2	13.59	19.46	1.10	14.61	20.37	1.19
T2L2C1024	4	13.80	26.11	1.23	14.80	27.33	1.34
T2L2C1024	8	13.69	21.04	1.54	14.60	21.82	1.61
T2L2C1024	16	13.84	17.97	2.02	14.91	18.78	2.11
T2L2C2048	1	15.05	11.32	1.09	16.11	11.93	1.19
T2L2C2048	2	13.57	19.47	1.09	14.61	20.37	1.19
T2L2C2048	4	13.78	26.12	1.25	14.79	27.32	1.34
T2L2C2048	8	13.70	21.04	1.53	14.60	21.81	1.60

T2L2C2048	16	13.84	18.01	2.02	14.91	18.79	2.11
-----------	----	-------	-------	------	-------	-------	------

Table A.2: Measurements for the roadNet-CA Graph

Config	DCSR Partition Factor	Scalar SpMV (ms)	Packed-Strip. SpMV (ms)	Loop-Raking SpMV (ms)	Scalar PageRank Average Iteration (ms)	Packed-Strip. PageRank Average Iteration (ms)	Loop-Raking PageRank Average Iteration (ms)
T1L1C512	1	1891.68	528.95	907.47	2750.35	568.02	956.25
T1L1C512	2	1889.22	528.67	845.76	2750.18	567.32	892.77
T1L1C512	4	1889.52	527.86	731.75	2749.47	567.50	775.48
T1L1C512	8	1889.17	528.43	565.66	2790.97	567.69	605.80
T1L1C512	16	1889.23	528.90	517.23	2791.60	568.12	556.27
T1L1C1024	1	1887.72	529.22	893.77	2746.48	572.30	946.16
T1L1C1024	2	1885.51	529.18	833.52	2746.50	571.56	884.37
T1L1C1024	4	1885.57	528.23	724.15	2745.93	571.32	772.06
T1L1C1024	8	1886.48	529.49	559.69	2788.38	572.55	603.60
T1L1C1024	16	1885.16	529.36	512.85	2788.26	572.33	555.41
T1L1C2048	1	1884.31	530.35	888.14	2744.33	581.83	948.89
T1L1C2048	2	1883.41	530.69	829.14	2746.10	581.81	887.81
T1L1C2048	4	1883.75	530.09	720.32	2745.62	581.45	776.06
T1L1C2048	8	1883.73	531.19	556.22	2787.76	582.35	607.93
T1L1C2048	16	1883.22	530.95	510.18	2788.03	582.59	560.51
T1L2C512	1	1891.65	494.70	813.68	2750.27	528.05	855.00
T1L2C512	2	1889.10	494.45	703.09	2750.17	528.21	741.65
T1L2C512	4	1889.85	494.20	535.30	2750.81	527.53	569.29
T1L2C512	8	1889.22	494.46	427.88	2790.96	527.80	459.61
T1L2C512	16	1889.07	495.42	372.46	2791.69	529.38	403.41
T1L2C1024	1	1887.71	494.85	778.85	2746.42	532.30	823.62
T1L2C1024	2	1885.42	494.89	674.63	2746.50	532.45	717.30
T1L2C1024	4	1885.92	494.38	523.91	2747.24	531.57	562.23
T1L2C1024	8	1886.05	495.23	419.30	2788.39	532.76	455.14
T1L2C1024	16	1885.28	495.70	366.92	2788.27	533.57	401.65
T1L2C2048	1	1884.29	495.81	768.08	2744.27	541.60	821.60
T1L2C2048	2	1883.34	496.24	667.85	2746.11	542.41	718.03
T1L2C2048	4	1883.14	496.02	519.04	2746.29	541.54	565.05
T1L2C2048	8	1883.87	496.71	415.35	2787.75	542.59	459.05
T1L2C2048	16	1883.20	497.21	364.08	2788.04	543.41	406.45
T2L1C512	1	955.27	267.86	456.04	1392.49	299.77	493.50
T2L1C512	2	952.69	267.32	383.51	1384.66	299.23	422.35
T2L1C512	4	955.08	267.88	293.86	1371.93	299.91	326.19

T2L1C512	8	951.12	266.94	263.33	1402.86	298.89	295.21
T2L1C512	16	951.18	267.64	233.25	1409.79	299.34	264.32
T2L1C1024	1	952.43	267.74	429.31	1387.03	303.34	469.45
T2L1C1024	2	948.89	267.14	369.16	1384.82	302.49	407.23
T2L1C1024	4	951.42	267.65	284.26	1377.51	303.22	320.85
T2L1C1024	8	948.41	267.01	258.00	1398.47	302.60	293.46
T2L1C1024	16	948.58	267.58	229.91	1407.56	303.17	264.57
T2L1C2048	1	950.74	268.24	422.73	1387.84	312.04	470.53
T2L1C2048	2	947.24	267.69	363.35	1382.88	311.08	409.11
T2L1C2048	4	949.72	268.41	280.52	1375.82	311.78	324.17
T2L1C2048	8	946.38	267.71	256.00	1397.97	311.25	299.22
T2L1C2048	16	947.50	268.16	228.83	1412.93	311.87	271.29
T2L2C512	1	955.02	251.11	455.59	1394.38	278.05	491.93
T2L2C512	2	952.66	250.73	310.44	1384.45	277.31	340.09
T2L2C512	4	954.73	251.35	237.25	1370.08	277.92	265.61
T2L2C512	8	951.23	250.76	200.16	1404.09	277.42	225.44
T2L2C512	16	951.26	251.61	165.77	1409.94	278.18	190.32
T2L2C1024	1	952.47	250.79	357.76	1386.58	281.58	391.85
T2L2C1024	2	948.92	250.09	269.76	1381.58	280.68	301.10
T2L2C1024	4	959.64	250.92	215.03	1377.49	281.27	245.01
T2L2C1024	8	948.29	250.52	186.18	1401.62	281.29	215.42
T2L2C1024	16	948.56	251.29	157.68	1407.61	282.03	186.11
T2L2C2048	1	950.67	251.20	343.34	1384.59	289.93	384.91
T2L2C2048	2	947.45	250.64	262.70	1379.84	289.12	301.51
T2L2C2048	4	949.73	251.49	209.88	1375.94	289.85	247.24
T2L2C2048	8	946.13	250.98	182.95	1397.51	289.54	219.68
T2L2C2048	16	947.54	251.93	156.03	1413.13	290.55	192.37

Table A.3: Measurements for the amazon0302 Graph

Config	DCSR Partition Factor	Scalar SpMV (ms)	Packed-Strip. SpMV (ms)	Loop-Raking SpMV (ms)	Scalar PageRank Average Iteration (ms)	Packed-Strip. PageRank Average Iteration (ms)	Loop-Raking PageRank Average Iteration (ms)
T1L1C512	1	417.86	116.30	119.14	526.73	122.73	125.76
T1L1C512	2	417.30	116.38	101.61	530.25	122.69	108.27
T1L1C512	4	417.10	117.58	92.03	525.98	124.10	97.74
T1L1C512	8	417.05	120.37	79.96	530.21	126.78	85.41
T1L1C512	16	417.12	128.23	70.88	530.37	135.07	76.03
T1L1C1024	1	409.85	115.94	116.82	518.51	122.59	123.56
T1L1C1024	2	409.27	115.95	99.73	522.04	122.62	106.90
T1L1C1024	4	409.09	117.19	90.94	517.79	123.97	96.94
T1L1C1024	8	409.23	119.97	79.28	522.07	126.67	85.00
T1L1C1024	16	409.14	127.87	70.40	522.16	134.94	75.85
T1L1C2048	1	403.03	115.78	115.80	511.33	122.80	122.81
T1L1C2048	2	402.51	115.84	99.05	514.89	122.85	106.51
T1L1C2048	4	402.36	117.09	90.48	510.63	124.18	96.85
T1L1C2048	8	402.49	119.82	79.01	514.90	126.87	85.08
T1L1C2048	16	402.35	127.72	70.21	515.02	135.14	76.01
T1L2C512	1	417.88	105.28	84.82	526.72	110.69	89.92
T1L2C512	2	417.25	105.77	71.45	530.25	111.08	76.05
T1L2C512	4	417.03	107.79	62.55	525.86	113.54	66.93
T1L2C512	8	417.00	113.67	55.65	530.22	119.40	59.81
T1L2C512	16	417.14	130.03	51.97	530.37	135.93	55.95
T1L2C1024	1	409.85	104.76	81.91	518.51	110.47	87.12
T1L2C1024	2	409.30	105.23	69.00	522.04	110.92	74.32
T1L2C1024	4	409.16	107.30	61.24	517.67	113.29	65.92
T1L2C1024	8	409.21	113.14	54.82	522.07	119.17	59.27
T1L2C1024	16	409.17	129.55	51.34	522.16	135.73	55.66
T1L2C2048	1	403.07	104.63	80.84	511.33	110.66	86.26
T1L2C2048	2	402.42	105.11	68.30	514.88	111.11	73.87
T1L2C2048	4	402.29	107.14	60.85	510.59	113.46	65.86
T1L2C2048	8	402.44	112.99	54.62	514.91	119.33	59.36
T1L2C2048	16	402.49	129.36	51.20	515.01	135.90	55.84
T2L1C512	1	261.11	71.73	63.40	316.54	76.89	68.20
T2L1C512	2	219.70	62.14	49.09	274.62	67.10	53.50
T2L1C512	4	211.82	60.81	41.21	269.52	65.69	45.57

T2L1C512	8	212.39	66.33	36.25	267.99	71.32	40.55
T2L1C512	16	210.65	69.29	33.22	267.22	74.34	37.37
T2L1C1024	1	254.55	71.30	61.51	310.50	76.73	66.73
T2L1C1024	2	214.98	61.75	47.69	269.76	67.01	52.55
T2L1C1024	4	207.26	60.50	40.42	262.62	65.67	45.06
T2L1C1024	8	207.86	66.03	35.73	263.34	71.30	40.28
T2L1C1024	16	205.98	68.96	32.74	262.50	74.32	37.21
T2L1C2048	1	248.86	71.16	60.72	304.34	76.81	66.19
T2L1C2048	2	210.76	61.69	47.23	265.42	67.16	52.29
T2L1C2048	4	203.00	60.32	40.14	258.15	65.82	45.09
T2L1C2048	8	203.78	65.87	35.44	259.08	71.47	40.32
T2L1C2048	16	202.71	68.89	32.62	258.82	74.55	37.37
T2L2C512	1	260.32	65.09	45.71	316.42	69.36	49.33
T2L2C512	2	219.64	57.58	34.75	274.57	61.86	38.01
T2L2C512	4	211.88	57.48	29.97	267.40	61.66	33.26
T2L2C512	8	212.38	67.68	27.41	267.97	71.94	30.73
T2L2C512	16	210.70	76.56	26.62	267.20	81.22	29.95
T2L2C1024	1	254.42	64.41	42.03	310.51	68.95	46.05
T2L2C1024	2	214.86	57.02	32.29	269.73	61.49	36.18
T2L2C1024	4	207.18	56.95	28.24	262.63	61.43	31.91
T2L2C1024	8	207.56	67.23	26.17	263.33	71.77	29.83
T2L2C1024	16	206.11	76.12	25.33	262.48	81.11	28.98
T2L2C2048	1	248.50	64.23	41.00	304.35	68.94	45.27
T2L2C2048	2	210.61	56.88	31.74	265.21	61.54	35.77
T2L2C2048	4	202.82	56.82	27.78	257.95	61.61	31.79
T2L2C2048	8	203.80	67.07	25.90	259.11	71.98	29.87
T2L2C2048	16	202.26	75.93	25.12	258.82	81.22	29.11

Bibliography

- [1] Junwhan Ahn et al. “A scalable processing-in-memory accelerator for parallel graph processing”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. June 2015, pp. 105–117. DOI: [10.1145/2749469.2750386](https://doi.org/10.1145/2749469.2750386).
- [2] Sam Ainsworth and Timothy M. Jones. “Graph Prefetching Using Data Structure Knowledge”. In: *Proceedings of the 2016 International Conference on Supercomputing. ICS '16*. Istanbul, Turkey: ACM, 2016, 39:1–39:11. ISBN: 978-1-4503-4361-9. DOI: [10.1145/2925426.2926254](https://doi.org/10.1145/2925426.2926254). URL: <http://doi.acm.org/10.1145/2925426.2926254>.
- [3] Tero Aittokallio and Benno Schwikowski. “Graph-based methods for analysing networks in cell biology”. In: *Briefings in Bioinformatics 7.3* (2006), p. 243. DOI: [10.1093/bib/bbl022](https://doi.org/10.1093/bib/bbl022). eprint: [/oup/backfile/Content_public/Journal/bib/7/3/10.1093/bib/bbl022/2/bbl022.pdf](http://oup/backfile/Content_public/Journal/bib/7/3/10.1093/bib/bbl022/2/bbl022.pdf). URL: [+%20http://dx.doi.org/10.1093/bib/bbl022](http://dx.doi.org/10.1093/bib/bbl022).
- [4] Lemar Akoglu, Hanghang Tong, and Danai Koutra. “Graph Based Anomaly Detection and Description: A Survey”. In: *Data Min. Knowl. Discov.* 29.3 (May 2015), pp. 626–688. ISSN: 1384-5810. DOI: [10.1007/s10618-014-0365-y](https://doi.org/10.1007/s10618-014-0365-y). URL: <http://dx.doi.org/10.1007/s10618-014-0365-y>.
- [5] Krste Asanovic. *RISC-V Vector Extension Specification Draft Proposal*. 2018. URL: <https://github.com/riscv/riscv-v-spec> (visited on 11/26/2018).
- [6] Krste Asanovic and John Wawrzynek. *Vector microprocessors*. University of California, Berkeley, 1998.
- [7] Krste Asanović et al. “The Rocket Chip Generator”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).
- [8] Ching Avery. “Giraph: Large-Scale Graph Processing Infrastructure on Hadoop”. In: *Proceedings of the Hadoop Summit. Santa Clara 11* (2011).
- [9] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. “Fast Incremental and Personalized PageRank”. In: *Proc. VLDB Endow.* 4.3 (Dec. 2010), pp. 173–184. ISSN: 2150-8097. DOI: [10.14778/1929861.1929864](https://doi.org/10.14778/1929861.1929864). URL: <http://dx.doi.org/10.14778/1929861.1929864>.

- [10] Scott Beamer, Krste Asanovic, and David Patterson. “Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server”. In: *2015 IEEE International Symposium on Workload Characterization*. Oct. 2015, pp. 56–65. DOI: [10 . 1109 / IISWC . 2015 . 12](https://doi.org/10.1109/IISWC.2015.12).
- [11] David Biancolin et al. “FASSED: FPGA-Accelerated Simulation and Evaluation of DRAM”. In: *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA’19)*. FPGA ’19. Seaside, CA, USA: ACM, 2019. ISBN: 978-1-4503-6137-8/19/02.
- [12] Aydin Buluc and John R Gilbert. “On the representation and multiplication of hypersparse matrices”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. Apr. 2008, pp. 1–11. DOI: [10 . 1109 / IPDPS . 2008 . 4536313](https://doi.org/10.1109/IPDPS.2008.4536313).
- [13] Henry M Cook, Andrew S Waterman, and Yunsup Lee. *TileLink cache coherence protocol implementation*. Tech. rep. 2015.
- [14] Gianna M Del Corso, Antonio Gulli, and Francesco Romani. “Fast PageRank computation via a sparse linear system”. In: *Internet Mathematics 2.3* (2005), pp. 251–273.
- [15] Assaf Eisenman et al. “Parallel Graph Processing: Prejudice and State of the Art”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE ’16. Delft, The Netherlands: ACM, 2016, pp. 85–90. ISBN: 978-1-4503-4080-9. DOI: [10 . 1145 / 2851553 . 2851572](https://doi.org/10.1145/2851553.2851572). URL: [http : / / doi . acm . org / 10 . 1145 / 2851553 . 2851572](http://doi.acm.org/10.1145/2851553.2851572).
- [16] Michael R Garey, David S. Johnson, and Larry Stockmeyer. “Some simplified NP-complete graph problems”. In: *Theoretical computer science 1.3* (1976), pp. 237–267.
- [17] Joseph E. Gonzalez et al. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 599–613. ISBN: 978-1-931971-16-4. URL: [https : / / www . usenix . org / conference / osdi14 / technical - sessions / presentation / gonzalez](https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez).
- [18] Joseph E. Gonzalez et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 17–30. ISBN: 978-1-931971-96-6. URL: [https : / / www . usenix . org / conference / osdi12 / technical - sessions / presentation / gonzalez](https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez).
- [19] Tae Jun Ham et al. “Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics”. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 56:1–56:13. URL: [http : // dl . acm . org / citation . cfm ? id = 3195638 . 3195707](http://dl.acm.org/citation.cfm?id=3195638.3195707).
- [20] Taher Haveliwala. *Efficient computation of PageRank*. Tech. rep. Stanford, 1999.
- [21] Taher H Haveliwala. “Topic-sensitive pagerank”. In: *Proceedings of the 11th international conference on World Wide Web*. ACM. 2002, pp. 517–526.

- [22] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [23] Susan Flynn Hummel and Edith Schonberg. “Low-overhead scheduling of nested parallelism”. In: *IBM Journal of Research and Development* 35.5.6 (Sept. 1991), pp. 743–765. ISSN: 0018-8646. DOI: [10.1147/rd.355.0743](https://doi.org/10.1147/rd.355.0743).
- [24] Gábor Iván and Vince Grolmusz. “When the Web meets the cell: using personalized PageRank for analyzing protein interaction networks”. In: *Bioinformatics* 27.3 (2010), pp. 405–407.
- [25] Yuntao Jia et al. “Edge v. node parallelism for graph centrality metrics”. In: *GPU Computing Gems Jade Edition*. Elsevier, 2011, pp. 15–28.
- [26] Zhihao Jia et al. “A Distributed multi-GPU System for Fast Graph Processing”. In: *Proc. VLDB Endow.* 11.3 (Nov. 2017), pp. 297–310. ISSN: 2150-8097. DOI: [10.14778/3157794.3157799](https://doi.org/10.14778/3157794.3157799). URL: <https://doi.org/10.14778/3157794.3157799>.
- [27] Bin Jiang. “Ranking spaces for predicting human movement in an urban environment”. In: *International Journal of Geographical Information Science* 23.7 (2009), pp. 823–837.
- [28] Peng Jiang and Gagan Agrawal. “Conflict-free Vectorization of Associative Irregular Applications with Recent SIMD Architectural Advances”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: ACM, 2018, pp. 175–187. ISBN: 978-1-4503-5617-6. DOI: [10.1145/3168827](https://doi.org/10.1145/3168827). URL: <http://doi.acm.org/10.1145/3168827>.
- [29] Sagar Karandikar et al. “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. June 2018, pp. 29–42. DOI: [10.1109/ISCA.2018.00014](https://doi.org/10.1109/ISCA.2018.00014).
- [30] Ben Keller et al. “A RISC-V Processor SoC With Integrated Power Management at Sub-microsecond Timescales in 28 nm FD-SOI”. In: *IEEE Journal of Solid-State Circuits* 52.7 (July 2017), pp. 1863–1875. ISSN: 0018-9200. DOI: [10.1109/JSSC.2017.2690859](https://doi.org/10.1109/JSSC.2017.2690859).
- [31] Jeremy Kepner et al. “Graphs, matrices, and the GraphBLAS: Seven good reasons”. In: *Procedia Computer Science* 51 (2015), pp. 2453–2462.
- [32] Ji Kim et al. “Using Intra-core Loop-task Accelerators to Improve the Productivity and Performance of Task-based Parallel Programs”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 ’17. Cambridge, Massachusetts: ACM, 2017, pp. 759–773. ISBN: 978-1-4503-4952-9. DOI: [10.1145/3123939.3136952](https://doi.org/10.1145/3123939.3136952). URL: <http://doi.acm.org/10.1145/3123939.3136952>.
- [33] Yu-Kwong Kwok and Ishfaq Ahmad. “Benchmarking and comparison of the task graph scheduling algorithms”. In: *Journal of Parallel and Distributed Computing* 59.3 (1999), pp. 381–422.

- [34] Amy N Langville and Carl D Meyer. “Deeper inside pagerank”. In: *Internet Mathematics* 1.3 (2004), pp. 335–380.
- [35] Yunsup Lee et al. “An Agile Approach to Building RISC-V Microprocessors”. In: *IEEE Micro* 36.2 (Mar. 2016), pp. 8–20. ISSN: 0272-1732. DOI: 10.1109/MM.2016.11.
- [36] Yunsup Lee et al. “Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. June 2011, pp. 129–140.
- [37] Yunsup Lee et al. “The Hwacha Microarchitecture Manual, Version 3.8.” In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-263* (2015).
- [38] Jure Leskovec et al. “Stanford network analysis project”. In: *ht tp://snap. stanford. edu* (2010).
- [39] Sander Lijbrink. “Irregular algorithms on the Xeon Phi”. Thesis. Universiteit Van Amsterdam, 2015.
- [40] Steffen Maass et al. “Mosaic: Processing a Trillion-Edge Graph on a Single Machine”. In: *Proceedings of the Twelfth European Conference on Computer Systems. EuroSys ’17*. Belgrade, Serbia: ACM, 2017, pp. 527–543. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064191. URL: <http://doi.acm.org/10.1145/3064176.3064191>.
- [41] Grzegorz Malewicz et al. “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD ’10*. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: <http://doi.acm.org/10.1145/1807167.1807184>.
- [42] Robert Ryan McCune, Tim Weninger, and Greg Madey. “Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing”. In: *ACM Comput. Surv.* 48.2 (Oct. 2015), 25:1–25:39. ISSN: 0360-0300. DOI: 10.1145/2818185. URL: <http://doi.acm.org/10.1145/2818185>.
- [43] Adam McLaughlin and David A. Bader. “Accelerating GPU Betweenness Centrality”. In: *Commun. ACM* 61.8 (July 2018), pp. 85–92. ISSN: 0001-0782. DOI: 10.1145/3230485. URL: <http://doi.acm.org/10.1145/3230485>.
- [44] Frank McSherry, Michael Isard, and Derek G. Murray. “Scalability! But at what COST?” In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015. URL: <https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry>.
- [45] Andreas Olofsson. *Power Efficiency Revolution for Embedded Computing Technologies (PERFECT)*. 2016. URL: <https://www.darpa.mil/program/power-efficiency-revolution-for-embedded-computing-technologies> (visited on 04/23/2018).

- [46] Muhammet Mustafa Ozdal et al. “Energy Efficient Architecture for Graph Analytics Accelerators”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 166–177. DOI: [10.1109/ISCA.2016.24](https://doi.org/10.1109/ISCA.2016.24).
- [47] Lawrence Page et al. *The PageRank citation ranking: Bringing order to the web*. Tech. rep. Stanford InfoLab, 1999.
- [48] Christos H Papadimitriou. “The Euclidean travelling salesman problem is NP-complete”. In: *Theoretical Computer Science* 4.3 (1977), pp. 237–244.
- [49] David Patterson and Andrew Waterman. <https://www.sigarch.org/simd-instructions-considered-harmful>. 2017. URL: <https://www.sigarch.org/simd-instructions-considered-harmful/> (visited on 09/30/2018).
- [50] Mohammad Taher Pilehvar, David Jurgens, and Roberto Navigli. “Align, disambiguate and walk: A unified approach for measuring semantic similarity”. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. 1. 2013, pp. 1341–1351.
- [51] Nataša Pržulj. “Protein-protein interactions: Making sense of networks via graph-theoretic modeling”. In: *Bioessays* 33.2 (2011), pp. 115–123.
- [52] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. “X-Stream: Edge-centric Graph Processing Using Streaming Partitions”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farmington, Pennsylvania: ACM, 2013, pp. 472–488. ISBN: 978-1-4503-2388-8. DOI: [10.1145/2517349.2522740](https://doi.org/10.1145/2517349.2522740). URL: <http://doi.acm.org/10.1145/2517349.2522740>.
- [53] Linton Salmon. *Circuit Realization at Faster Timescales (CRAFT)*. 2016. URL: <https://www.darpa.mil/program/circuit-realization-at-faster-timescales> (visited on 04/23/2018).
- [54] Nadathur Satish et al. “Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’14. Snowbird, Utah, USA: ACM, 2014, pp. 979–990. ISBN: 978-1-4503-2376-5. DOI: [10.1145/2588555.2610518](https://doi.org/10.1145/2588555.2610518). URL: <http://doi.acm.org/10.1145/2588555.2610518>.
- [55] Julian Shun and Guy E. Blelloch. “Ligra: A Lightweight Graph Processing Framework for Shared Memory”. In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’13. Shenzhen, China: ACM, 2013, pp. 135–146. ISBN: 978-1-4503-1922-5. DOI: [10.1145/2442516.2442530](https://doi.org/10.1145/2442516.2442530). URL: <http://doi.acm.org/10.1145/2442516.2442530>.
- [56] Narayanan Sundaram et al. “GraphMat: High Performance Graph Analytics Made Productive”. In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1214–1225. ISSN: 2150-8097. DOI: [10.14778/2809974.2809983](https://doi.org/10.14778/2809974.2809983). URL: <https://doi.org/10.14778/2809974.2809983>.

- [57] Yoshizumi Tanaka et al. “Performance Evaluation of OpenMP Applications with Nested Parallelism”. In: *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. LCR '00. London, UK, UK: Springer-Verlag, 2000, pp. 100–112. ISBN: 3-540-41185-2. URL: <http://dl.acm.org/citation.cfm?id=648049.761156>.
- [58] Wenpu Xing and Ali Ghorbani. “Weighted PageRank algorithm”. In: *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004*. May 2004, pp. 305–314. DOI: [10.1109/DNSR.2004.1344743](https://doi.org/10.1109/DNSR.2004.1344743).
- [59] Xiangyao Yu et al. “IMP: Indirect Memory Prefetcher”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 178–190. ISBN: 978-1-4503-4034-2. DOI: [10.1145/2830772.2830807](https://doi.org/10.1145/2830772.2830807). URL: <http://doi.acm.org/10.1145/2830772.2830807>.
- [60] Marco Zagha and Guy E. Blelloch. “Radix Sort for Vector Multiprocessors”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 712–721. ISBN: 0-89791-459-7. DOI: [10.1145/125826.126164](https://doi.org/10.1145/125826.126164). URL: <http://doi.acm.org/10.1145/125826.126164>.