Domain-specific Architectures for Data-intensive Applications

by

Abraham Lamesgin Addisie

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Doctoral Committee:

Professor Valeria M. Bertacco, Chair
Professor Todd M. Austin
Assistant Professor Reetuparna Das
Professor Wei D. Lu

Abraham Lamesgin Addisie

abrahad@umich.edu

ORCID iD: 0000-0001-6285-1714

© Abraham Lamesgin Addisie 2020

For my family

ACKNOWLEDGMENTS

It is my great pleasure to acknowledge those who extended their unreserved support and encouragement throughout my time working on this dissertation. First and foremost, I would like to thank my advisor, Prof. Valeria Bertacco, for her extensive mentorship while working on this dissertation and for giving me the opportunity to work on research areas that I found to be interesting. I am also grateful for the extensive feedback and advice that I got from my committee members, Prof. Todd Austin, Prof. Reetuparna Das, and Prof. Wei Lu. I feel fortunate to have been taught by the excellent faculty at UofM, who laid down the foundation of my research knowledge through extensive course lectures and projects.

I also would like to extend my gratitude to a lot of talented students at the university who provided their help while working on this dissertation. Specifically, I would like to thank former ABresearch-lab members, Doowon Lee, Biruk Mammo, Salessawi Yitbarek, Rawan Abdel Khalik, and Ritesh Parikh, for your insightful discussions, including holding a multitude of brainstorming sessions at the early stage of my research. I am also thankful for my collaborators, Leul Belayneh, Hiwot Kassa, and Luwa Matthews. This dissertation would not have been possible without your help. I am particularly grateful for Annika Pattenaude for reviewing my research papers, which helped me improve my writing skills. I also would like to acknowledge the many other friends for making themselves available for discussions about the ups and downs of my graduate student life. I am particularly grateful for Helen Hagos, Misiker Aga, and Zelalem Aweke. I am also thankful for other members of the ABresearch-lab members, Vidushi Goyal and Andrew McCrabb, and the many other students at CSE whom I didn't mention by name but have been of great help.

Furthermore, I would like to acknowledge the many staff members at CSE who made themselves available whenever I needed administrative-related assistance. I am particularly grateful for Magdalena Calvillo, Alice Melloni, Ashley Andreae, Sara Snay, Christine Boltz, Ava Burnham, and Stephen Reger. Also, Denise Duprie and Dawn Freysinger have been of tremendous help during my transition to the university as a new incoming student.

I do want to take this chance to acknowledge the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, for funding my research.

Finally, I am forever indebted to my family. I could not have finished this dissertation without their continuous support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Graphs' versatile ability to represent diverse relationships, make them effective for a wide range of applications. For instance, search engines use graph-based applications to provide high-quality search results. Medical centers use them to aid in patient diagnosis. Most recently, graphs are also being employed to support the management of viral pandemics. Looking forward, they are showing promise of being critical in unlocking several other opportunities, including combating the spread of fake content in social networks, detecting and preventing fraudulent online transactions in a timely fashion, and in ensuring collision avoidance in autonomous vehicle navigation, to name a few. Unfortunately, all these applications require more computational power than what can be provided by conventional computing systems. The key reason is that graph applications present large working sets that fail to fit in the small on-chip storage of existing computing systems, while at the same time they access data in seemingly unpredictable patterns, thus cannot draw benefit from traditional on-chip storage.

In this dissertation, we set out to address the performance limitations of existing computing systems so to enable emerging graph applications like those described above. To achieve this, we identified three key strategies: 1) specializing memory architecture, 2) processing data near its storage, and 3) message coalescing in the network. Based on these strategies, this dissertation develops several solutions: OMEGA, which employs specialized on-chip storage units, with co-located specialized compute engines to accelerate the computation; MessageFusion, which coalesces messages in the interconnect; and Centaur, providing an architecture that optimizes the processing of infrequently-accessed data.

Overall, these solutions provide 2$\times$ in performance improvements, with negligible hardware overheads, across a wide range of applications.

Finally, we demonstrate the applicability of our strategies to other data-intensive domains, by exploring an acceleration solution for MapReduce applications, which achieves a 4$\times$ performance speedup, also with negligible area and power overheads.

CHAPTER 1

Introduction

1.1 Graph and Graph-based Applications

Since their introduction in the eighteenth century, graphs have been recognized as an effective mechanism to capture complex relationships. Graphs represent relationships by means of vertices connected by edges. Take, for example, webgraphs, for which webpages correspond to vertices, and edges represent the web links among them. Or consider social networks, for which vertices correspond to people, and edges represent the relationship among them. Graphs' versatile ability to represent diverse relationships, make them effective for a wide range of applications. For instance, several commercial search engines – such as Google, Microsoft's Bing, and Baidu – rely heavily on analyzing webgraphs to provide high-quality search results [91]. Social networks companies– such as Facebook, Twitter, and Youtube – perform analytics tasks on their networks, which allow them to offer improved services to their users, and targeted advertising to client companies. Medical institutes and hospitals, such as the Allen Institute for Brain Science, utilize brain functional connectivity networks to study and understand animal and human brains, leading to better insights in patient diagnosis [47]. Many recent ride-share and map-service providers – such as Uber, Lyft, Google maps, and Waze – have flourished by employing analyses of road networks that provide efficient route selections, thus reducing traffic congestion and fuel consumption for their users. In addition, graph analytics have been extensively utilized in AI-based applications, such as movie recommendation solutions, product recommendations, and natural language processing. Most recently, graphs are being employed to support contact tracing in the management of viral pandemics [11].

1.2 Current Performance Limitations of Graph Applications

In light of their applications mentioned above, graphs are poised to address an increasingly broader range of challenges and applications; however, at the same time, their computational requirements are unmet by conventional computing systems. Modern computing systems require a significant amount of time to complete most graph-based applications. For example, the increase in fake content in several social networks jeopardizes the credibility of these platforms. In this case, graphs can be utilized to identify fake content by analyzing the credibility of content sources (users) in the network; specifically, by ranking users in the network [7]. The rank value depends on how they are connected with other users. A high rank-value indicates that the user is connected from a large number of users, that is, many people follow this user's opinions. Alternatively, the user could be connected with just a few other users, who, in turn, have many followers. Both these scenarios indicate a credible user, whereas a user with a low rank-value is deemed as not-as-credible, and hence a potential source of fake content. In short, in this scenario, the content from users with a low rank-value will be identified as fake and, consequently, removed from the network. Today, to complete this type of task, conventional computing systems can take minutes to hours, depending on the social network's size [66]. In contrast, to inhibit the dissemination of fake content to a significant number of users, the content must be identified, tagged as fake, and removed within just one second [10].

As another example, financial institutions lose trillions of dollars annually due to online-transaction fraud. In this scenario, graphs can be utilized to detect and prevent such fraud by analyzing users' transactions along with their social networks' presence and activity, implementing a ranking method that, first, takes into account users who have a history of committing fraud. This set of users could be gathered from other sources, such as governmental criminal records [9]. The analysis then essentially identifies users that have strong ties with individuals in the existing set of people known to have committed fraud. Based on the idea that users tend to form strong relationships among themselves in a social network, users with a high rank-value will be considered more likely to commit fraud and, consequently, their transactions will be inspected carefully. Note that this approach may lead to biased assessments. To complete this type of task, current computing systems take minutes to hours, based on the size of the social network [66]; however, the detection process must be completed within a few milliseconds to flag the potentially fraudulent even before it is approved [13].

Finally, to offer one last example, graph applications are thought to sit at the heart of the 7-trillion-dollar autonomous vehicle industry [8]. In this domain, graphs could enable self-driving vehicles to utilize their geospatial and geolocation data, not only to identify efficient routes at a fine granularity, but also to prevent collisions by analyzing their position relative to other nearby obstacles, including other vehicles [8]. These tasks essentially involve finding the shortest route from the current location of the vehicle to a destination location. They typically take seconds to complete, when using existing computing platforms [102]; in contrast, vehicles must be able to complete the tasks within milliseconds so to attain the enhanced goals described above [8]. Overall, these stringent-time limits cause existing computing systems to be unfit for such tasks.

1.3 Mismatch of Today's Conventional Computing Systems

Figure 1.1 shows a high-level architecture of a typical modern processor, which comprises multiple cores, enabling concurrent execution within an application, so to boost its performance. The cores are connected among themselves via interconnect routers. They are also connected to off-chip memory via a similar interconnection mechanism. The cores access data from off-chip memory, an activity that incurs high latency (*i.e.,* delay) and

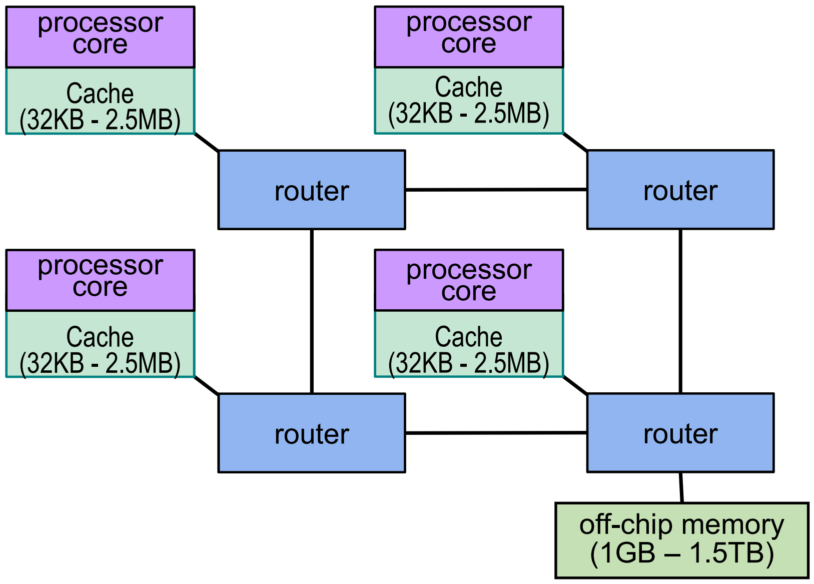


Figure 1.1: High-level architecture of a conventional processor. A typical conventional processor comprises multiple cores, each equipped with cache memory. Cores are connected to the rest of the system and the off-chip memory via an interconnection network completed by routers.

consumes much energy. To limit the impact of memory accesses in terms of energy and performance costs, a small amount of on-chip storage, called cache, is typically co-located with each core. Caches act as a buffer, temporally storing and delivering data from the much larger off-chip memory with much less latency and energy. Applications that have predictable data access patterns and/or small working datasets are ideal in this context. In contrast, typical graph applications present large working datasets that do not fit in such small on-chip storages. In addition, they access data in a seemingly random pattern; hence, they cannot leverage the reuse opportunities offered by small and fast on-chip storage. For those two reasons, when running a graph application on a conventional multicore architecture, a large amount of data is continuously transferred from off-chip main memory to the processor and back. This data movement is extremely costly, as it incurs high latency and it consumes a major fraction of the energy required to execute the application.

Case study. To analyze the magnitude of the traffic between processors and off-chip memory, we profiled a few graph applications running on an Intel Xeon E5-2630 processor

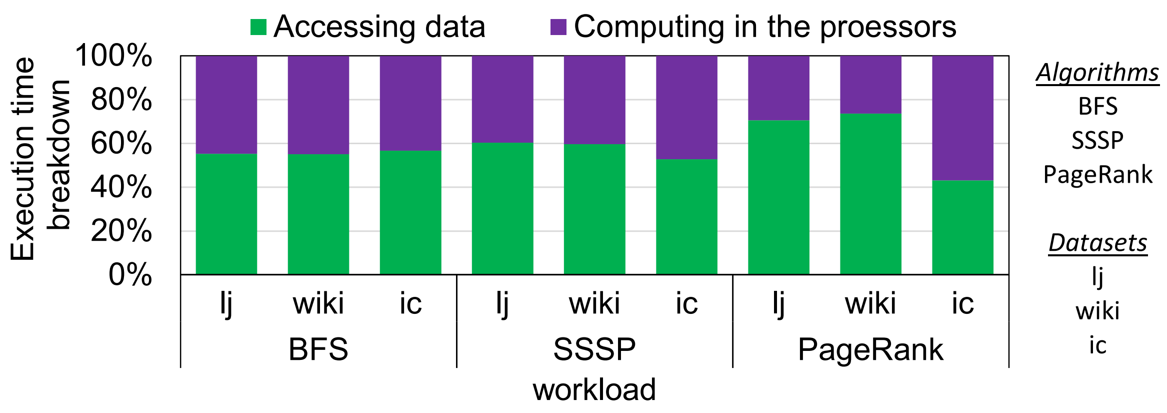


Figure 1.2: Execution time breakdown of graph applications. By profiling several graph applications using vTune on an Intel Xeon E5-2630 v3 processor with a last-level-cache of 20MB, we learned that performance is mostly limited by time spent accessing data.

equipped with a 20MB (last-level) cache. Our findings, plotted in Figure 1.2 and discussed in more detail in Chapter 3, show that the main cause of performance bottlenecks in those applications is indeed due to the inability of the processor cores to access off-chip memory efficiently.

Below we briefly discuss sources of performance and energy inefficiencies beyond off-chip memory accesses.

Inefficiencies due to high interconnect traffic. As discussed above, graph applications' random-access patterns are often responsible for much of the traffic between cores and off-chip memory traveling on the interconnect. In addition, modern processors' cores often access data stored in remote cores' caches, further exacerbating the density of interconnect traffic. High interconnect traffic creates, in turn, congestion in the network, leading to poor performance and energy inefficiencies.

Inefficiencies due to high overhead of atomic operations computation. Multicore processors [102, 105] distribute the execution of an application processing an input graph across multiple cores, which operate in parallel. The input graph is first partitioned across cores, and then each core is tasked with the processing of its own portion of the input graph. Processing a graph entails performing update operations on its vertices. While processing their own portions, multiple distinct cores often must update a same destination vertex. To correctly perform this update, the operation is performed atomically, requiring the cores to serialize their access to the vertex among themselves, thus executing only one such operation at a time. This type of operations is referred to as an atomic operation. As one can imagine, the execution of these atomic operations leads to inefficient utilization of the processor's hardware resources, as multiple cores are suspended, waiting for other cores' accesses to complete.

1.4 Our Strategies

In this dissertation, we set out to boost the performance of graph-based applications, so to enable the critical opportunities discussed in 1.2. We do this by exploring several strategies. Specifically, we devised three strategies to tackle the challenges presented in Section 1.3 in a systematic manner. These strategies are 1) **specializing the memory structure** to improve the efficacy of small on-chip storage, and thus fit the characteristics of graph applications, 2) **processing data near where it is stored** as much as possible to reduce interconnect traffic, and 3) **coalescing data messages** in the network, which should also reduce the amount of interconnect traffic. We discuss these strategies below, along with their potential benefits.

Our first strategy strives to specialize the on-chip memory architecture of a typical conventional processor to alleviate traffic generated because of their inefficient use. In line with this, we make the key observation that many graph applications do not access data in a completely random manner, but rather present domain-specific access patterns. For instance, they access some vertices more frequently than others, a trait especially pronounced in graphs that follow a power-law distribution [85, 26, 37]; that is, approximately 20% of the vertices in a graph are connected to approximately 80% of the edges in that graph. Because of their high connectivity, it is likely that this 20% of the vertices is accessed much more frequently than the rest. We realized that such access patterns can provide an opportunity to devise specialized on-chip storage. Specifically, a small amount of storage could be designed to house the most-connected vertices, which are often responsible for the majority of the accesses, thus reducing the traffic from/to the off-chip memory.

Our second strategy is to investigate if processing data near its storage mitigates interconnect traffic. Because graph applications rarely reuse recently accessed data, this strategy may reduce the interconnect traffic that otherwise would take place between a core and a remote storage unit. A byproduct of this strategy is that data processing could be performed using specialized compute engines, instead of general-purpose cores. As discussed in Section 1.3, general-purpose cores frequently face idle time when executing atomic operations. Offloading atomic operations from cores to specialized compute engines would minimize this inefficiency, as the processor cores would be freed up and could advance to other computation tasks.

Our final strategy is to explore the benefit of coalescing graph analytics data messages in the network. Offloading atomic operations from general-purpose cores to specialized compute engines may lead to a high-level of interconnect traffic. Due to the power-law characteristics described above, it is common for multiple atomic operations originating from different cores to update a same vertex. Since these operations are similar within a given graph application (see Chapter 4), it should be possible to coalesce multiple operations to a same destination vertex in the network, thus reducing the interconnect traffic.

In the next section, we present several solutions that we developed in pursuing the strategies just described.

1.5 Our Solutions

This section presents solutions that employ the strategies described in Section 1.4. Figure 1.3 provides an overview of the proposed solutions. We name the solutions along the horizontal axis, and indicate on the vertical axis the strategies that we explored in developing them.

Strategies \ Solutions	OMEGA	MessageFusion	Centaur
Specialized memory architecture	✓		✓
Processing data near storage	✓		✓
Coalescing data messages		✓	

Figure 1.3: Overview of the proposed solutions.

In our first solution, OMEGA [18], we worked to specialize the memory architecture. As discussed in Section 1.4, many graphs follow a power-law distribution, hence some vertices are accessed more frequently than others. Based on this characteristic, we developed specialized on-chip storage that houses this frequently-accessed data. To make our solution scalable, we distribute the specialized on-chip storage by co-locating one storage unit with each core. This technique reduces the traffic between the processor and off-chip memory. In addition, in striving to follow our second strategy, we augment each specialized on-chip storage unit with a specialized compute engine that carries out the atomic operations affecting the vertices in the local on-chip storage, and which have been offloaded from the cores. The specialized compute engines help reduce both interconnect traffic and computational demands on the cores.

Although the solution described above manages to offload atomic operations from cores to specialized compute engines, the messages for these operations must be transferred via the interconnect. Due to the power-law characteristics discussed in the prior section, most often these operations are to a same vertex, and can be coalesced in the network. Our next solution, MessageFusion, augments each router in the network with specialized hardware to carry out such message coalescing, thus reducing network traffic.

The solutions described thus far seek to optimize the execution of atomic operations on frequently-accessed data. However, the computation of atomic operations on infrequently-accessed data still takes place through conventional mechanisms, that is, transferring the data from off-chip memory to the relevant processor core and computing the updates in the core. Our third solution, Centaur, addresses precisely this infrequently-accessed data: it optimizes the computation of atomic operations on infrequently-accessed data as well. In particular, it augments off-chip memory units with specialized compute engines, which are tasked with executing atomic operations related to infrequently-accessed data. The computation by these specialized engines reduces the traffic between cores and off-chip memory and, at the same time, further frees up the cores from executing atomic operations.

1.6 Beyond Graph Analytics

In addition to graph analytics, we recognize that there are many important families of applications dominated by data movement. Three key domains in this space include machine learning applications, streaming applications, and MapReduce applications. Machine learning applications, such as deep neural networks, are being utilized for a wide range of applications, *e.g.*, product recommendation and language translation to name just a few. These applications move a large amount of data, mainly due to accessing the parameters of the neural network models. Streaming applications provide important insights, for instance, identifying trending topics in social networks and in stocks trading. They often process a large amount of input data incoming at a continuous pace. Finally, MapReduce applications are suited for processing a large amount of accumulated data. They are utilized for various purposes, such as indexing webpages for improving search results, analyzing log files for providing summary data, such as website usage. We believe that the strategies we outlined in Section 1.4 are beneficial for a wide range of applications, with traits like those just described. To demonstrate their applicability to these applications domains, in the dissertation, we explore solutions for MapReduce applications because of their wide adoption. We recognize other domains of applications as further research directions. Chapter 6 is dedicated to the discussion of our solution for the MapReduce domain.

1.7 Dissertation Document Organization

This dissertation document is organized as follows. Chapter 2 provides background in graph analytics. Chapter 3 describes OMEGA, a memory subsystem for the efficient execution of graph analytics on multicore machines. The chapter provides architectural details of OMEGA and explains how OMEGA exploits the power-law traits of many graphs to identify frequently-accessed vertex data, thus maximizing on-chip storage utilization. Chapter 4 discusses MessageFusion, a technique that coalesces data messages in the network, and thus scales the performance and energy efficiencies of a computing system with the number of processing elements. Chapter 5 presents Centaur, a technique that complements the other solutions by optimizing the processing of infrequently-accessed data. In exploring application domains beyond graph applications, Chapter 6 discusses CASM, an architecture that demonstrates the applicability of our strategies to MapReduce applications. Finally, Chapter 7 provides conclusions to this dissertation, as well as future research directions that can further improve the techniques developed in this dissertation.

1.8 Summary of Contributions

The goal of this dissertation is to address the performance limitations of existing computing systems so to enable emerging graph applications like those described in Section 1.2. To achieve this goal, we identify three key strategies: 1) specializing memory architecture, 2) processing data near its storage, and 3) message coalescing in the network. Based on these strategies, this dissertation develops several solutions: OMEGA, which employs specialized on-chip storage units with co-located specialized compute engines to accelerate the computation; MessageFusion, which coalesces messages in the interconnect, Centaur, which optimizes the processing of infrequently-accessed data. Overall, the solutions presented in this dissertation provide 2$\times$ in performance improvements, with negligible hardware overheads, across a wide range of graph-based applications. Finally, we demonstrate the benefits of deploying our strategies to other data-intensive domains by proposing a solution in one such domain, MapReduce, where we attain a $4\times$ performance speedup with negligible area and power overheads.

The next chapter provides a review of background concepts in graph analytics, setting the stage for a detailed discussion of the key techniques presented in the dissertation.

CHAPTER 2

Background on Graph Analytics

This chapter reviews background concepts in graph analytics. Graph analytics emerged with the eighteenth-century implementation of the graph in Kaliningrad, Russia, when the locals pondered to solve a fun mathematical problem [14]. The problem was to determine whether it was possible to walk through the city by crossing each of the seven bridges once and only once. The seven bridges connected the four landmasses of the city together, as shown in Figure 2.1. In 1735, Leonhard Euler solved this problem by relying on a simple representation that abstracted the detailed map of the city in terms of its four landmasses and seven bridges. In doing so, Euler pioneered the concept of graph. In the current field of graph theory, a landmass corresponds to a vertex and a bridge corresponds to an edge.

Since 1735, graphs have been extensively utilized as a way to simplify the representation of rather complex relationships. For instance, web graphs provide simple representation for the connectivity of web pages [91], and human brain functional connectivity networks capture complex interactions among the various regions of human brains [47]. Although their simplified representation mechanism makes them a great candidate in various application areas, efficiently executing them on existing computing systems is a challenging task, as these systems were not designed with graph-based applications in mind.

To achieve well-suited computing systems, it is crucial that we first understand some relevant characteristics of graphs. First, different graphs represent different types of rela-

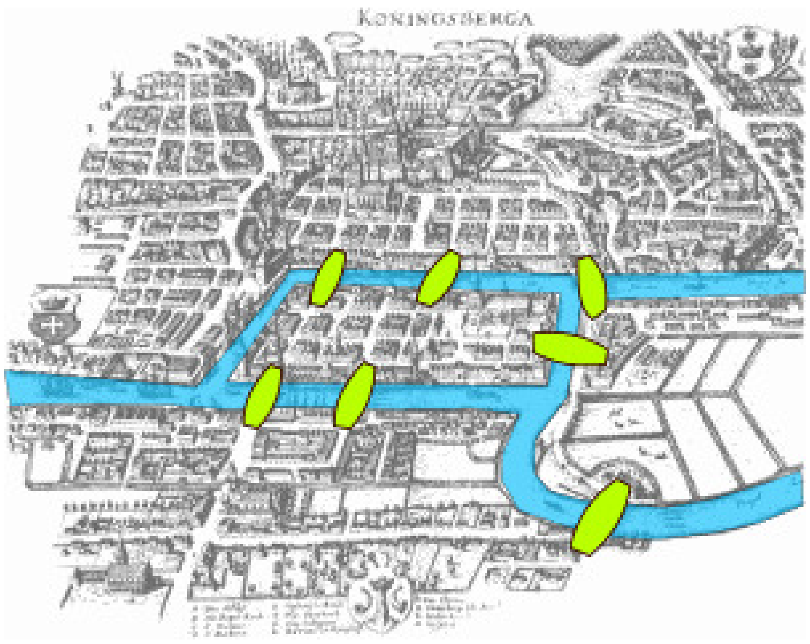


Figure 2.1: An 18th century map of Kaliningrad, highlighting its river and bridges [14].

tionships and, consequently, have different structures (*e.g.*, edge to vertex ratio). Second, a graph can be stored in various ways in memory (*e.g.*, adjacency matrix). In addition, depending on the types of insights required, a graph can be processed by using multiple algorithms. Note, finally, that these algorithms are typically implemented on a range of possible high-level software frameworks, for the benefit of the programmer's productivity. A basic grasp of these characteristics is crucial to understand the solutions presented later in this dissertation, thus this chapter provides a brief review.

2.1 Graph Structures

A graph can represent various types of relationships. An example is a graph that represents a social network, which conceptualizes interactions among people. Consequently, each graph has its own unique structure (*i.e.*, the unique way of how its vertices are interconnected). Generally speaking, there are four types of graph structures: power-law, community-based, regular, and random, described as follows.

Note also, that we refer to graphs arising in practical, real-world applications as natural

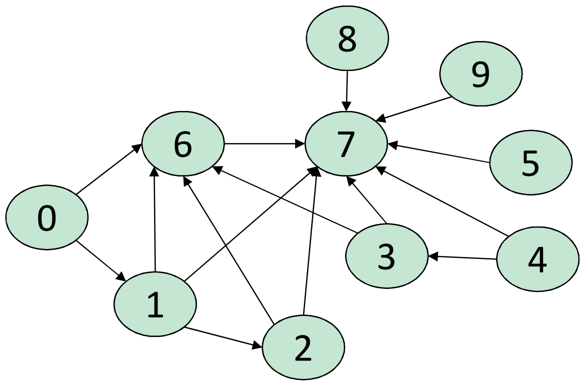


Figure 2.2: Power-law graph. Vertex 6 and 7 are connected to approximately 80% of the edges.

graphs, in contrast with synthetic graphs, which are generated to present specified characteristics.

Power-law graphs. Power-law graphs exhibit power-law distribution. A power law is a functional relationship such that the output varies with the power of the input. In the context of graphs, power-law graphs follow the power law in their vertex-to-edge connectivity; that is, the majority of the edges are connected to just a few vertices, with a long tail of vertices connected to very few edges. In practical approximations, a graph is said to follow the power law if 20% of its vertices are connected to approximately 80% of the edges [85]. Many natural graphs follow the power law, including web-derived graphs, electrical power grids, citation networks, collaboration networks of movie actors [30, 26], graphs that represent social networks [37], biological networks [25], and human brain functional networks [47]. The authors of [30, 26] argue that the reason for such abundant occurrence of power-law distributions in graphs is a mechanism called “preferential attachment”: a new vertex joining a graph would most likely connect to an already popular vertex. Figure 2.2 presents an example of a power-law graph. As shown in the figure, vertices 6 and 7, which account for 20% of the vertices, are connected to approximately 80% of the edges.

Community-based graphs. Community-based graphs [50, 84] have a set of vertices that are densely connected among themselves, while being sparsely connected to other similar set of vertices (other communities). Each group of densely connected vertices is

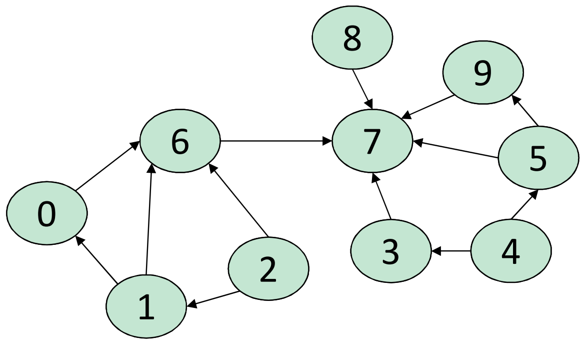


Figure 2.3: Community-based graph. The graph contains two communities.

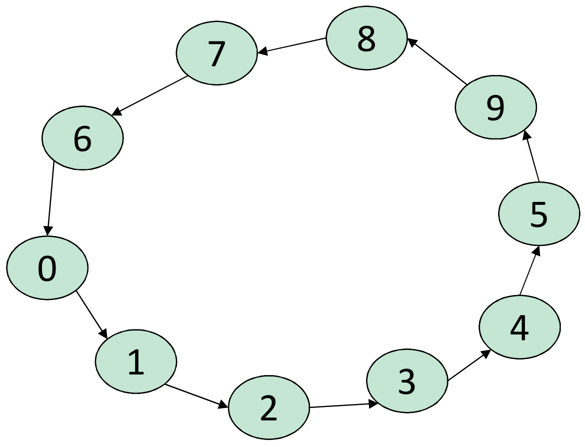


Figure 2.4: Regular graph. Each vertex is connected to two edges.

called a community. Community-based graphs are common in natural graphs [52]. For instance, in social networks, it is common to have friends closely connected with each other, while being sparsely connected with other groups of friends. Figure 2.3 provides an example of a community-based graph. The figure highlights two communities: one community comprises vertices 0, 1, 2, and 6, whereas the other community includes all other vertices. Recent work [28, 29] has proposed optimizations for community-based graphs. The key idea behind these optimizations is to segment the graph at a community-granularity and then process it, one segment at a time, thus maximizing the utilization of the fast-access storage.

Regular graphs. Regular graphs [75] have vertices with the same number of edges. This type of graph is not common in natural graphs; however, it is widely utilized for scientific studies. Figure 2.4 presents an example of a regular graph. As shown, each

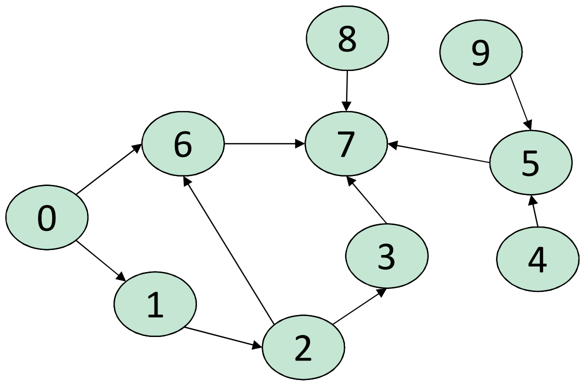


Figure 2.5: Random graph. Each vertex has an arbitrarily number of edges.

vertex has two edges, which means that the graph is regular.

Random graphs. Random graphs [86] are mostly useful for studying graph theory. Each of their vertices is connected to a random number of edges. Figure 2.5 provides an example of a random graph. The figure shows that each vertex has a randomized number of edges.

Note that graphs can be *weighted*, *i.e.*, their edges can have associated weight values. For instance, if the graph represents a road network, then the edge weights could represent the distance between cities.

Among these four types of graphs (power-law, community-based, regular, and random), the properties of power-law graphs are central to this dissertation, because the power-law trait enables us to maximize on-chip memory utilization; hence it is further discussed throughout the rest of the dissertation.

2.2 Graph Layouts

There are various ways to organize data related to graphs in memory: adjacency matrix, adjacency list, coordinate list, compressed sparse row, and compressed sparse column. Below, we provide a brief overview of each of them, and use the example graph of Figure 2.6 to illustrate them.

Adjacency matrix. An adjacency matrix stores a graph as a bidimensional V*V array,

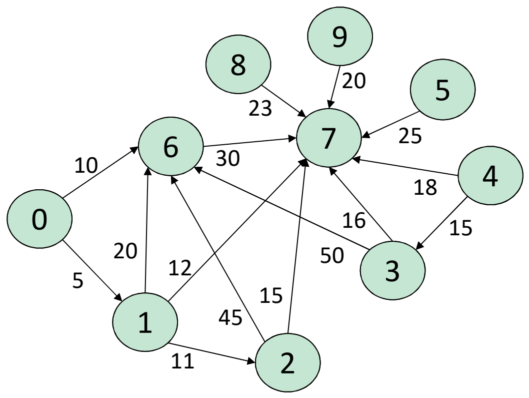


Figure 2.6: An example weighted graph, which we use to demonstrate the graph representations discussed in this chapter.

	0	1	2	3	4	5	6	7	8	9
0	∞	5	∞	∞	∞	∞	10	∞	∞	∞
1	∞	∞	11	∞	∞	∞	20	12	∞	∞
2	∞	∞	∞	∞	∞	∞	45	15	∞	∞
3	∞	∞	∞	∞	∞	∞	50	16	∞	∞
4	∞	∞	∞	15	∞	∞	∞	18	∞	∞
5	∞	∞	∞	∞	∞	∞	∞	25	∞	∞
6	∞	∞	∞	∞	∞	∞	∞	30	∞	∞
7	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
8	∞	∞	∞	∞	∞	∞	∞	23	∞	∞
9	∞	∞	∞	∞	∞	∞	∞	20	∞	∞

Figure 2.7: Adjacency matrix representation for the graph of Figure 2.6. Most entries are ∞.

where V is the number of vertices. Each entry in the array has either a weight value or the symbol ∞. The ∞ symbol at <row, column> indicates that there is no edge between the vertices with index <row> and index <column>. Otherwise, the value indicates the weight of the edge. The adjacency matrix for the example graph of Figure 2.6 is presented in the Figure 2.7. This representation is useful to represent dense graphs, but it is an inefficient choice for sparse graphs, because it entails unnecessary storage for pairs of vertices that are not connected to each other.

Adjacency list. An adjacency list maintains a per-vertex list, which stores, for each vertex, the IDs of vertices that are connected to it by an edge, and the weight value of

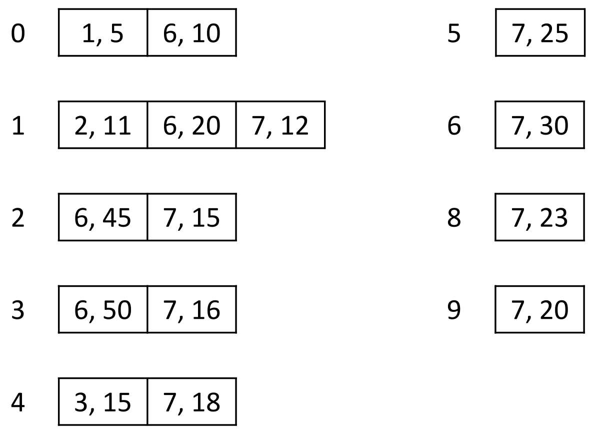


Figure 2.8: Adjacency list representation for the graph of Figure 2.6.

the associated edge. For sparse graphs, adjacency lists are more efficient compared to adjacency matrices as they only require space proportional to the number of edges in the graph. However, for dense graphs, it might not be as efficient, as it requires maintaining multiple lists, instead of the compact structure of an adjacency matrix. An example of the adjacency list for the example graph of 2.6 is shown in Figure 2.8.

Coordinate list. Instead of storing the edge information for a graph with a list for each vertex, as with the adjacency list, the coordinate list represents a graph by maintaining the list of edges, each edge listing the pair of vertices that it connects, as well as the weight value of the edge. This layout is less efficient than the adjacency list, as each vertex ID is reported multiple times, once for each of its incident edges. It is also less efficient to identify all vertices adjacent to a given vertex. Most often, the coordinate list is sorted by source vertex, followed by the destination vertex, to partially address this type of inefficiencies. Note, however, that, this layout has also some merits. For instance, it stores the graph information in a uniform data structure as the adjacency matrix, and it simplifies the process of iterating through all the edges. The coordinate list for the example graph of Figure 2.6 is (0,1,5), (0,6,10), (1,2,11), (1,6,20), (1,7,12), (2,6,45), (2,7,15), (3,6,50), (3,7,16), (4,3,15), (4,7,18), (5,7,25), (6,7,30), (8,7,23), (9,7,20).

Compressed sparse row. A compressed sparse row representation optimizes the adjacency matrix format by leveraging three separate arrays. The first array lists all the valid

edge weights:	5	10	11	20	12	45	15	50	16	15	18	25	30	23	20
column index:	1	6	2	6	7	6	7	6	7	3	7	7	7	7	7
row index:	0	2	5	7	9	11	12	12	13	14	15				

Figure 2.9: Compressed sparse row representation for the graph of Figure 2.6.

edge weights:	5	11	15	10	20	45	50	12	15	16	18	25	30	23	20
row index:	0	1	4	0	1	2	3	1	2	3	4	5	6	8	9
column index:	0	0	1	2	2	2	3	7	14	14	15				

Figure 2.10: Compressed sparse column representation for the graph of Figure 2.6.

weight values in the adjacency matrix in row order. The second array, called the column index array, stores the column indices in the adjacency matrix of the valid weights in the first array. Finally, the third array, called row index array, stores the indices from the edge-weights array corresponding to the first element of each row in the adjacency matrix. In some implementations, this array contains an additional value at the end, storing the total number of non-zero weights. As an example, Figure 2.9 presents a compressed sparse row representation for the example graph of Figure 2.6. With reference to the Figure, the first entry in the edge-weights array contains the value 5, corresponding to the first edge-weight value in the row order traversal of the adjacency matrix. This value is found in column 1 of the adjacency matrix; hence, the first entry of the column index array shows the value 1. Correspondingly, the row index array has a value of 0, as the edge-weight values for the first row of the adjacency matrix are stored starting at index 0 of the edge-weights array.

The compressed sparse row representation has several advantages over coordinate list. First, vertex IDs are implicit, leading to additional memory efficiency. Second, it provides a way to quickly identify all the neighbors of a vertex, as neighbors' data is stored in contiguous memory locations.

Compressed sparse column. The compressed sparse column representation is similar to compressed sparse row, with the roles of row and column index arrays reversed. The first array, that is, the edge-weights array, lists all the valid weight values in the adjacency matrix in column order. The second array, called the row index array, stores the row indices in the adjacency matrix of the weights in the edge-weights array. Finally, the third array, called column index array, stores the indices from the edge-weights array of the first element of each column in the adjacency matrix. Similar to the compressed sparse row representation, in some implementations, this array contains an extra value at the end, storing the number of non-zero weights. Figure 2.10 illustrates compressed sparse column for the example graph of Figure 2.6. Once again, the first entry in the edge-weights array is 5, corresponding to the first edge-weight value in the column order traversal of the adjacency matrix representation. This value is found in row 0 in the adjacency matrix; hence, the first entry in the row index array has a value of 0. The first entry of the column index array is 0, as there are no valid edge-weight values in the first column of the adjacency matrix. It also has a value of 0 as its second entry, since the first edge-weight value of the second column of the adjacency matrix is stored at index 0 of the edge-weights array. The compressed sparse column representation enjoys similar advantages as the compressed sparse row representation, as mentioned above.

In this dissertation, our solutions are evaluated by leveraging direct C/C++ implementations and the graph software framework, Ligra [102], which stores graph data using representations that closely resemble the compressed sparse row and column representations. Note, however, that it would be straightforward to adapt them to software frameworks that store graphs in some of the other representations described.

2.3 Graph Algorithms

Graphs are processed with graph algorithms. Each graph algorithm provides unique insights into the graph. The type of graph algorithm determines how the graph is traversed and, consequently, the performance of the underlying computing systems. Hence, improving the design of the existing computing system first requires an understanding of the different types of graph algorithms. To this end, in this chapter, we considered several popular graph algorithms, which are discussed below. These are also the algorithms that we used to evaluate the solutions presented in the following chapters.

PageRank (*PageRank*) iteratively calculates a rank value for each vertex, based on the number and popularity of its neighbors, until the value converges. *PageRank* was originally introduced by Google to rank web pages in web graphs [91] and prioritize web pages provided in search results. A higher rank value indicates that a web page is more popular, as there are more websites that link to it, thus it has a higher chance of appearing at the top of the search results. Since then, it has been applied to a wide range of other domains, for instance, to determine the popularity of users in social networks [56] or to extract representative keywords/sentences from a document in natural language processing [76]. In Section 2.5, we show a possible implementation of *PageRank* using pseudo-code.

Breadth-First Search (*BFS*) traverses the graph breadth-first, starting from an assigned root node and assigning a parent to each reachable vertex. *BFS* is applied in a range of applications, such as to determine optimal wire routes in Very Large Scale Circuit (VLSI) designs [42] or to crawl the web to identify webpages that are within k-levels from a particular source page [83].

Single-Source Shortest-Path (*SSSP*) traverses a graph as *BFS*, while computing the shortest distance from the root vertex to each vertex in the graph. It is widely used, for instance, to find the shortest distance between two locations in online map applications, or to find the minimum delay of a communication path in telecommunication networks [15].

Betweenness Centrality (*BC*) computes, for each vertex, the number of shortest paths that traverse that vertex. It is used for various purposes, such as identifying a set of individuals who provide quick connections to a large portion of users in social networks [116], or finding a central location point to most other locations in transportation networks applications [70].

Radii (*Radii*) estimates the maximum radius of the graph, that is, the longest of the minimum distances between any pair of vertices in the graph. It is utilized in various areas, such as, to estimate the maximum number of clicks required to reach to a web page from another in web graphs [98], or to determine the worst-case communication delay between any two nodes in internet routing networks [106].

Connected Components (*CC*) in an undirected graph, identifies all subgraphs such that, within each subgraph, all vertices can reach one another, and none of the connecting paths belong to different subgraphs. It is widely used in various application domains, such as to identify people with similar interests in social networks so to perform content recommendations [43], or to segment images based on some similarity property when partitioning images in image analytics [110].

Triangle Counting (*TC*), in an undirected graph, computes the number of triangles, *i.e.*, the number of vertices that have two adjacent vertices that are also adjacent to each other. *TC* is widely used in various areas, such as for spam identification in large scale computer networks, or for link recommendation in social networks [108].

k-Core (*KC*), in an undirected graph, identifies a maximum-size connected subgraph comprising only vertices of degree $\geq k$. It is widely applied in various applications, such as for visualization of large-scale graphs [27], or for analyzing biological protein structures via the identification of the highest density portion of the structure [112].

Sparse Matrix-Vector Multiplication (SpMV) computes the product of the sparse adjacency matrix of a graph with a vector of values. *SpMV* is a general technique that provides an alternative approach to solve several graph algorithms [35, 104]. For instance, the *BFS* algorithm described above can be solved using *SpMV*. The key operation in *BFS* involves finding the set of vertices reachable in one step from the set of vertices already discovered. This operation can be performed by multiplying a sparse adjacency matrix representation of the graph with a 0-1 vector, where 1 corresponds to vertices that have been reached in the prior step of the traversal. Figure 2.11 shows the first iteration of *BFS*

$$\begin{array}{c|cccccccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\;\mathrm{X}\;
\begin{array}{|c|}
\hline 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \hline
\end{array}
\;=\;
\begin{array}{|c|}
\hline 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \hline
\end{array}$$

Figure 2.11: First iteration of *BFS* starting at vertex 0 using SpMV, multiplying the transpose of the unweighted adjacency matrix representation of the graph of Figure 2.6 with a vector where only index zero is set to 1.

starting at vertex 0 for the adjacency matrix representation of the graph of Figure 2.6. This iteration involves multiplying the unweighted version (all the ∞ values are replaced with 0 and the non-zero values with 1) of the transpose of the adjacency matrix of the graph with the vector [1,0,0,0,0,0,0,0,0,0]. The operation produces the vector [0,1,0,0,0,0,1,0,0,0], indicating that the vertices that can be reached in one step of the traversal are 1 and 6. The algorithm will then continue to multiply the matrix with the result vector until all entries of the result vector are zero.

The key characteristics of the algorithms outlined are reported in Table 2.1. In some cases, we do not have the characteristics; hence, we marked them with '-'. The table reports the type of atomic (used during the reduce phase) operation (atomic/reduce op type). The algorithms we considered range from leveraging a floating-point addition as their atomic operation, or an unsigned or signed operation, which may include a Boolean comparison, or a minimum or add operation. Our table also reports a qualitative evaluation of the portion of vertex accesses that entail an atomic operation over the total set of accesses (%atomic op). Similarly, we also report about the portion of access that exhibit poor temporal and spatial locality (% random access). The table also reports the size of per-vertex entry (vtxProp entry size), in Bytes, in the vtxProp structure, and the number of such structures

Table 2.1: Graph-based algorithm characterization

Characteristic	***PageRank***	***BFS***	***SSSP***	***BC***	***Radii***
atomic/reduce op type	fp add	unsigned comp.	signed min & Bool comp.	fp add	signed or & signed min
%atomic op	high	low	high	medium	high
%random access	high	high	high	high	high
vtxProp entry size	8	4	8	8	12
#vtxProp	1	1	2	1	3
active-list	no	yes	yes	yes	yes
read src vtx's *vtxProp*	no	no	yes	yes	yes
msg size	8	4	4	-	-

Characteristic	***CC***	***TC***	***KC***	***SpMV***
atomic/reduce op type	unsigned min	signed add	signed add	signed add
%atomic op	high	low	low	high
%random access	high	low	low	high
vtxProp entry size	8	8	4	4
#vtxProp	2	1	1	1
active-list	yes	no	no	no
read src vtx's *vtxProp*	yes	no	no	no
msg size	4	-	-	4

(# vtxProp) that the algorithm maintains. Many graph algorithms process only a subset of the vertices per iteration; hence, we provide whether an algorithm maintains such list or not (active-list). Moreover, the table reports whether an algorithm requires to access a vertex's vtxProp to generate a corresponding atomic operation message or not (read src vtx's vtxProp). Finally, the table provides the size of an atomic operation's message that crosses the network (msg size).

2.4 Graph Datasets

Graph algorithms process input graph datasets. Graph datasets represent complex relationships among data. They are used in a wide range of applications, including social networks, collaboration networks, web graphs, and road networks. To this end, we considered several types of graph datasets, which are discussed below. These are also the datasets that we utilized to evaluate the solutions discussed in this dissertation.

wiki-Vote (*wv*) is a snapshot of the network of Wikipedia users voting each other to become administrators [67].

soc-Slashdot0811 (*sd*) and **soc-Slashdot0922 (*sd2*)** are snapshots of the network of Slashdot users' friendship/foeship relationships [67].

ca-AstroPh (*ap*) is a network mapping co-authorship of manuscripts uploaded on Arxiv in the field of Astro-physics [67].

soc-Pokec (*pk*) is a snapshot of the network of Pokec users' friendship relationships.

rMat (*rMat*) is a synthetic graph that follows power-law distribution, which we generated using the rMat graph generator tool from [5].

orkut-2007 (*orkut*) and **com-Orkut (*orkut2*)** are snapshots of the network of Orkut users' friendship relationships [6, 67].

ljournal-2008 (*lj*) and **soc-LiveJournal1 (*lj2*)** are snapshots of the network of LiveJournal users' friendship relationships [6, 67].

enwiki-2013 (*wiki*) is a snapshot of the network of the English portion of the Wikipedia web pages, interconnected by hyperlinks [6].

indochina-2004 (*ic*) is a crawl of the network of the country domains of Indochina's web pages, interconnected by hyperlinks [6].

uk-2002 (*uk*) is a crawl of the network of .uk domain's web pages, interconnected by hyperlinks [6].

twitter-2010 (*twitter*) is a snapshot of the network of Twitter's users' follower relationships [6].

roadNet-CA (*rCA*) is a road network of California [67].

roadNet-PA (*rPA*) is a road network of Pennsylvania [67].

western-USA (*USA*) is a road network of the Western USA region [3].

Table 2.2: Graph dataset characterization

Characteristic	***wv***	***sd***	***sd2***	***ap***	***pk***	***rMat***	***orkut***	***orkut2***	***wiki***
#vertices (M)	0.01	0.07	0.08	0.13	1.6	2	3	3.1	4.2
#edges (M)	0.1	0.9	1	0.39	30.6	25	234	117.2	101
type	-	dir.	-	undir.	-	dir.	dir.	-	dir.
in-degree con.	-	62.8	-	100	-	93	58.73	-	84.69
out-degree con.	-	78.05	-	100	-	93.8	58.73	-	60.97
power law	-	yes	-	yes	-	yes	yes	-	yes
degree	14.6	-	11.5	-	18.8	-	-	38.1	-
reference	[67]	[67]	[67]	[67]	[67]	[67]	[6]	[67]	[6]
Characteristic	***lj2***	***lj***	***ic***	***uk***	***twitter***	***rPA***	***rCA***	***USA***	
#vertices (M)	4.8	5.3	7.4	18.5	41.6	1	1.9	6.2	
#edges (M)	69	79	194	298	1468	3	5.5	15	
type	-	dir.	dir.	dir.	dir.	undir.	undir.	undir.	
in-degree con.	-	77.35	93.26	84.45	85.9	28.6	28.8	29.35	
out-degree con.	-	75.56	73.37	44.05	74.9	28.6	28.8	29.35	
power law	-	yes	yes	yes	yes	no	no	no	
degree	14.2	-	-	-	-	-	1.4	2.4	
reference	[67]	[6]	[6]	[6]	[6]	[67]	[67]	[3]	

The key characteristics of the datasets just described are reported in Table 2.2. In some cases, we do not have the characteristics; hence, we marked the corresponding items with '-'. The table reports the number of vertices, which ranges from 0.01 million to 41.6 million, and the number of edges, which ranges from 0.1 million to 1.5 billion. The table also provides the type of graph, which can be either directed or undirected. Moreover, it reports in-degree and out-degree connectivity of the 20% most-connected vertices. The in-degree/out-degree connectivity measures the fraction of incoming/outgoing edges connected to the 20% most-connected vertices. Finally, it provides whether the datasets abide by the power law or not in their structures, followed by their graphs' degrees (edge-to-vertex ratio), and the source references from where we collected them.

2.5 Graph Software Frameworks

The implementation of graph algorithms involve several data structures. It is common to provide three data structures to represent a graph: "vertex property" (***vtxProp***), "edge list" (***edgeArray***), and non-graph data (***nGraphData***). *vtxProp* stores the values that must be computed for each vertex, *e.g.*, rank values of vertices in *PageRank* algorithm; *edgeArray* maintains both sets of outgoing and incoming edges per each vertex; and *non-graph* data includes all data-structures that are not part of *vtxProp* or *edgeArray*. For many graph processing frameworks, accesses to *vtxProp* often exhibit poor locality, as the access order depends on the ID values of neighboring vertices, which exhibits irregular patterns. In addition, in multicore processors, multiple distinct cores typically update *vtxProp* concurrently. To correctly perform these update operations, they must be serialized, thus incurring high-performance overheads, as cores wait in line for their turn to update. In contrast, the *edgeArray* and *nGraphData* data structures are stored in contiguous memory locations, with accesses that typically follow sequential patterns, thus exhibiting good cache locality.

Many aspects of the execution of graph algorithms – such as parallelization, thread scheduling, and data partitioning – are similar across many graph algorithms. Hence, graph algorithms are most often implemented using high-level software frameworks to boost the productivity of developing a new graph algorithm. There are two variants of graph software frameworks. The first type is based on distributed systems, running on networks of computers. Examples of this type include PowerGraph [53], GraphChi [66], and Pregel [73]. The key limitation of this type of frameworks is that they incur high network communication costs. This limitation is addressed by the second type of frameworks, which rely on single-node multicore architectures. Examples of this latter type include Ligra [102], GraphMat [105], Polymer [117], X-Stream [99], GridGraph [122], MOSAIC [72], and GraphBIG [82]. In this dissertation, our focus is on single-node solutions, as powerful, high-end servers with large and low-cost memory often have sufficient capability to run many real-world graph algorithms in single-node machines.

Single-node frameworks can be broadly classified as either *edge-centric* (X-Stream, GridGraph) or *vertex-centric* (Ligra, GraphMat, Polymer, GraphBIG). The processing of most graphs by both types of frameworks involves iteratively computing values associated with vertices that are adjacent to a set of active vertices. Active vertices most often account for a portion of the entire set of vertices, and they most often vary across iterations. During each iteration, only the vertices that are active in that iteration are processed. As an example, in *BFS*, during the first iteration, only the assigned root vertex is part of the active vertices and only this vertex is thus processed. The differences among frameworks is primarily related to how they process graphs.

Edge-centric frameworks access the complete list of edges of the graph, each edge connecting one vertex (called source vertex) to another vertex (called destination vertex). For each edge, if its source vertex is part of the active set, then the value associated with the source vertex is used to update the value of the corresponding destination vertex. Whereas vertex-centric frameworks traverse the set of active vertices (instead of the complete list of edges) and update the values of the adjacent vertices. For instance, for *BFS*, during the first iteration, starting at root vertex 0 for the graph in Figure 2.6, an edge-centric framework steps through the complete list of edges ((0,1,5,1), (0,6,10,1), (1,2,11,0), (1,6,20,0), (1,7,12,0), (2,6,45,0), (2,7,15,0), (3,6,50,0), (3,7,16,0), (4,3,15,0), (4,7,18,0), (5,7,25,0), (6,7,30,0), (8,7,23,0), (9,7,20,0)). Note that the edge list representation is similar to the coordinate list representation discussed above, except that it includes an additional entry per edge, indicating whether the source vertex is active (1) or not (0). While carrying out this edge traversal, the framework updates the values of all the edges that have vertex 1 or 6 as sources, as they are the only vertices connected to vertex 0. Whereas, a vertex-centric framework only accesses the active vertex (vertex 0) and updates its neighbors (vertex 1 and 6).

During the second iteration of *BFS*, the edge-centric framework accesses the entire edge list above again, and updates the destination vertices corresponding to edges connected to

```
1   for V in vertices                                   //Initialization phase
2       outDegree[V] = numEdgesOfV
3   for V in vertices
4       next_rank[V] = 0;
5   for V in vertices
6       curr_rank[V] = 1/numVertices;
7   for V in vertices                                   //Reduce/Update  phase
8       for edge(V,U) in outgoingEdges[V]
9           next_rank[U] += curr_rank[V]/outDegree[V];
10  for V in vertices                                   //Apply phase
11      next_rank[V] = c*next_rank[V] + (1-c)/numVertices;
```

Figure 2.12: Pseudo-code for PageRank algorithm in a vertex-centric framework, computing the rank of each vertex (*next_rank*) based on the current rank values (*curr_rank*) and a vertex's degree (*outDegree*).

vertex 1 and 6; whereas the vertex-centric framework accesses the two set of neighbors of vertex 1 and 6 and updates their values. Note that, in a vertex-centric framework, the neighbors of a vertex are typically stored contiguously in memory; whereas neighbors of different vertices are not co-located.

In summary, edge-centric frameworks access the entire edge list sequentially; hence, they benefit from high memory-access bandwidth, as sequential-accesses require higher bandwidth than random-accesses in modern memory systems. In contrast, vertex-centric frameworks access adjacent vertices in irregular patterns, as the locations of the neighbors of different active vertices are typically non-contiguous. However, vertex-centric frameworks access only those few edges that are incident to active vertices. In this dissertation, we target vertex-centric frameworks: these frameworks have gained popularity mainly because of their simpler programming paradigm.

To illustrate how graph algorithms are implemented in vertex-centric frameworks, consider *PageRank*. Figure 2.12 provides an example of pseudo-code for *PageRank*. As discussed above, *PageRank* iteratively calculates the rank values of vertices in a graph. To do so, it maintains several types of data structures: 1) *outDegree* (line 2) stores the number of a vertex's outgoing edges; 2) *next_rank* (line 4) maintains rank values for vertices active in the current iteration, 3) *curr_rank* (line 6) stores a copy of the ranks associated with each vertex, as calculated during the previous iteration, and 4) *outgoingEdge* (line 8) maintains neighboring vertex information for each vertex.

PageRank is executed in three phases: *initialization phase*, *reduce/update phase*, and *apply phase*. As shown in Figure 2.12, it starts with the *initialization phase* by initializing *outDegree*, *next_rank* (line 3-4), and *curr_rank* (line 5-6). It then enters into the *reduce/update* phase, where it iterates through all the outgoing edges of each vertex (*outgoingEdge*) to find its adjacent vertices (line 7-8). Following this step, it reads the rank (*curr_rank*) and degree (*outDegree*) of the vertex, and accrues the new rank value (*next_rank*) at the adjacent vertices (line 9). Then, it enters in the *apply* phase, where it updates *curr_rank* by performing a user-defined *apply* function (line 10-11). Note that, since typical accesses for the *outgoingEdge*, *curr_rank*, and *outDegree* data structures follow sequential patterns, these accesses exhibit good cache locality. The same holds true for accesses to the *next_rank* data structure, but only during the *initialization* and *apply* phases. In contrast, accesses to *next_rank* often exhibit poor locality during the *reduce/update* phase, as access order exhibits irregular patterns. In addition, in multicore processors, multiple distinct cores typically update *next_rank*. To correctly perform the update operations, these updates must be executed one at a time, serializing their execution at each of the general-purpose cores, thus incurring high-performance overheads. One of our goals, as described in Chapter 1.3, is to alleviate the above overheads while, at the same time, continue to support the high-level software frameworks available today, so not negatively impact developers' productivity.

2.6 Summary

In this chapter, we reviewed background concepts in graph analytics. We described various types of graph structures, presented several ways to store graphs in memory, and described various graph algorithms and software frameworks that are typically used to implement the graph algorithms. These concepts are useful to understand the solutions discussed in the remaining parts of the dissertation.

The next chapter outlines the first of such solutions: OMEGA, a memory architecture for the efficient execution of graph analytics on multicore machines. OMEGA leverages the power-law graph characteristic of many naturally existing graphs to maximize on-chip memory utilization. It was designed to operate with Ligra, a vertex-centric framework. However, as we mentioned earlier, the OMEGA solution can be easily adapted to operate with other graph-programming frameworks.

CHAPTER 3

Heterogeneous Memory Subsystem for Graph Analytics

As discussed in the introduction chapter, the execution of graph analytics on multicore processors provides limited performance and energy efficiencies primarily because of unstructured and unpredictable data access patterns. These access patterns cause a high amount of traffic between multicore processors' cores and off-chip memory. In addition, modern processors' cores often access data stored in remote cores' caches, creating high interconnect traffic. High interconnect traffic creates, in turn, congestion in the network, leading to poor performance and energy inefficiencies. Furthermore, multicore processors distribute the execution of an application processing an input graph across multiple cores, which operate in parallel. This mechanism typically entails multiple distinct cores updating a same data. To correctly perform the update operations, the updates must be executed one at a time, serializing their execution at each of the general-purpose cores, thus incurring high-performance overheads. As described in Chapter 2, many real-world data collections are structured as natural graphs; that is, they follow the power-law distribution: 80% of the edges are incident to approximately 20% of the vertices [48, 53]. Examples include web connectivity graphs [91], social networks [37], biological networks such as protein-to-protein interactions [25], and human brain functional networks [47]. This characteristic can be leveraged to devise solutions that combat the limitations described above.

In this chapter, we strive to overcome the limitations mentioned above by designing an

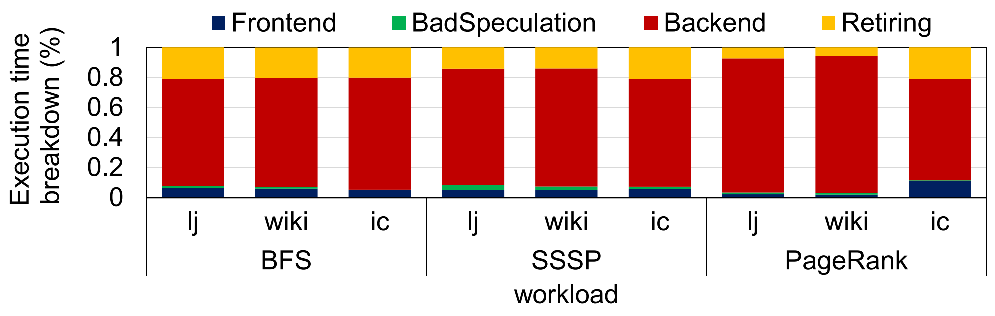
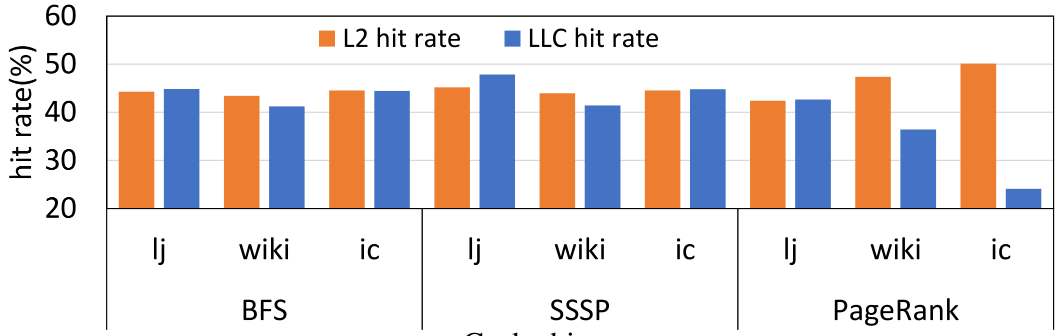


Figure 3.1: Execution breakdown using TMAM[113] metrics. Profiling several real-world graph workloads using vTune on an Intel Xeon E5-2630 v3 processor with a last-level-cache of 20MB shows that the performance is mostly limited by the execution pipeline's throughput (backend).

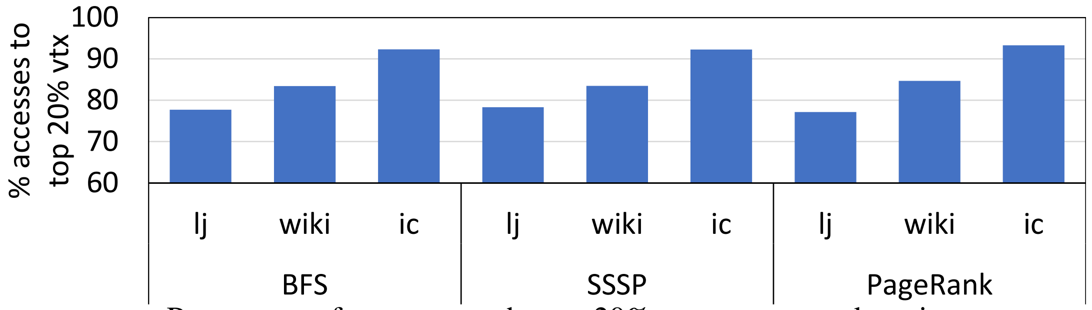
architecture that can accelerate graphs, which follow the power-law distribution. In pursuing this goal, we rely on our strategies, discussed in Section 1.4. Specifically, we explore a specialized memory architecture that houses those vertices with high incident edges, which are often responsible for the majority of the accesses, thus reducing the traffic from/to the off-chip memory. We also investigate the benefits of processing data near storage. Because graph applications rarely reuse recently-accessed data, this strategy reduces the interconnect traffic that otherwise would take place between a core and a remote storage unit in reading the vertex data, updating it, and then storing it back. A byproduct of this strategy is that some of the data processing is performed using specialized compute engines, instead of general-purpose cores. As discussed in Section 1.3, general-purpose cores frequently face idle time when executing atomic operations: offloading atomic operations from cores to specialized compute engines reduces this inefficiency, as the processor cores are freed up and can advance to other computation tasks.

3.1 Motivational Study

To analyze where the bottlenecks lie in graph processing, we performed several studies. In our first study, we profiled a number of graph-based applications over Ligra, a framework optimized for natural graphs [31], on an Intel Xeon E5-2630 processor with a 20MB last-



(a) Cache hit rate



(b) Percentage of accesses to the top 20% most-connected vertices

Figure 3.2: Cache profiling on traditional CMP architectures. (a) profiling several graph workloads on vTune on Intel Xeon reports hit rates below 50% on L2 and LLC. (b) Over 75% of the accesses to *vtxProp* target the 20% most connected vertices, a frequency-based access pattern not captured by regular caches.

level cache. We used Intel's VTune to gather the "Top-down Microarchitecture Analysis Method" (TMAM) metrics [113]. For this analysis, we used representative graph-based algorithms running on real-world datasets, as discussed in Section 2.3 and 2.4. The results, plotted in Figure 3.1, show that applications are significantly *backend bounded*; that is, the main cause of performance bottlenecks is the inability of the cores' pipelines to efficiently complete instructions. Furthermore, the execution time break down of the *backend* portion reveals that they are mainly bottlenecked by memory wait time (memory bounded) with an average value of 71%. From this analysis, we can infer that the biggest optimization opportunity lies in improving the memory structure to enable faster access completions. Note that other prior works have also reached similar conclusions [81, 23]. Hence, our primary goal in this chapter is to develop an optimized architecture for the memory system of a CMP to boost the performance of graph processing.

Our proposed approach is unique when compared with prior works because i) we strive to fully exploit the frequency-based access pattern that exists in many graph algorithms when processing natural graphs, and ii) we want to maintain transparency to the user as much as possible; thus, we strive to minimize – or better, avoid – architectural modifications that require modifications to the graph-processing framework or the graph applications running on it.

Inefficiencies in locality exploitation in natural graphs. We then performed a more in-depth analysis of the bottlenecks in the memory subsystem by monitoring last-level cache hit rates for the same graph-based applications. Figure 3.2(a) shows that all workloads experience a relatively low hit rate. We suspect that the main culprit is the lack of locality in the *vtxProp* accesses, as also suggested by [82, 23]. Moreover, in analyzing the distribution of accesses within *vtxProp*, Figure 3.2(b) reports which fraction of the accesses targeted the 20% of vertices with highest in-degree: this subset of the vertices is consistently responsible for over 75% of the accesses. The study overall suggests that if we could accelerate the access of this 20% of vertices in *vtxProp*, then we could significantly reduce the memory access bottleneck and, in turn, the pipeline backend bottleneck pinpointed by the TMAM analysis.

On-chip communication and atomic instruction overheads. We also observed that, when using conventional caches to access the vertex data structure, each access must transfer data at cache-line granularity, potentially leading to up to 8x overhead in on-chip traffic, since the vertex's *vtxProp* entry most often fits in one word and a cache line is 64 bytes. In addition, when a vertex's data is stored on a remote L2 bank, the transfer may incur significant on-chip communication latency overhead (17 cycles with the baseline setup of our evaluation). Finally, atomic accesses may incur the most significant performance overhead of all those discussed. Based on the approach in [81], we estimated the overhead entailed by the use of atomic instructions by comparing overall performance against that of an identical PageRank application where we replaced each atomic instruction with a regular read/write. The result reveals an overhead of up to 50%. Note that these on-chip communication and atomic instructions overheads are particularly high, even for small and medium graph datasets, which could comfortably fit in on-chip storage. We found these kinds of datasets to be abundant in practice, *e.g.*, the *vtxProp* of over 50% of the datasets in [67] can fit on an Intel Xeon E5-2630's 20MB of on-chip storage. Hence, we believe that a viable memory subsystem architecture must holistically address atomic instructions, on-chip and off-chip communication overheads.

Limitations of graph pre-processing solutions. Prior works have exploited the frequency-based access pattern inherent in graph algorithms by relying on offline vertex-reordering mechanisms. We also have deployed some of these solutions on natural graphs, including in-degree, out-degree, and SlashBurn-based reordering [69], but have found limited benefits. More specifically, in-degree- and out-degree-based reorderings provide higher last-level cache hit rates, +12% and +2% respectively, as frequently accessed vertices are stored together in a cache block; however, they also create high load imbalance. For all of the reordering algorithms, we perform load balancing by fine-tuning the scheduling of the OpenMP implementation. We found that the best speedup over the original ordering was 8% for in-degree, 6.3% for out-degree, and no improvement with SlashBurn. Other works [119] have reported similar results, including a slowdown from in-degree-based ordering. Another common graph preprocessing technique is graph slicing/partitioning, which provides good performance at the expense of requiring significant changes to the graph framework [119].

3.2 OMEGA Overview

Figure 3.3 outlines the solution discussed in this chapter, called OMEGA, which leverages scratchpad storage, with one scratchpad unit co-located with each core. The scratchpads accommodate highly-connected vertices (i.e., vertices that are connected with a large

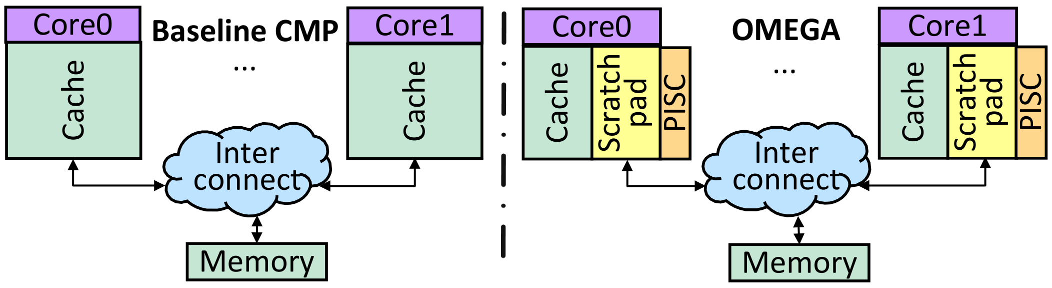


Figure 3.3: OMEGA overview. OMEGA provides a specialized memory architecture to support graph-based computation in CMPs. The architecture is implemented by replacing a portion of each core's cache with a scratchpad memory, organized to store the highly-connected vertices from the graph, and augmented with a simple data-processing unit (PISC) to carry out vertex computations in-situ.

number of edges), satisfying most of the memory requests, and consequently reducing off-chip accesses during an algorithms execution. Note that, generally speaking, accesses to vertex data lack spatial locality, as these frequently accessed vertices are unlikely to be placed adjacent to each other. Because of this reason, traditional caches are not effective. As discussed above, many graphs follow the power-law distribution: approximately 20% of the vertices are connected to 80% of the edges. We identify the highly-connected vertices so to store them in scratchpads.

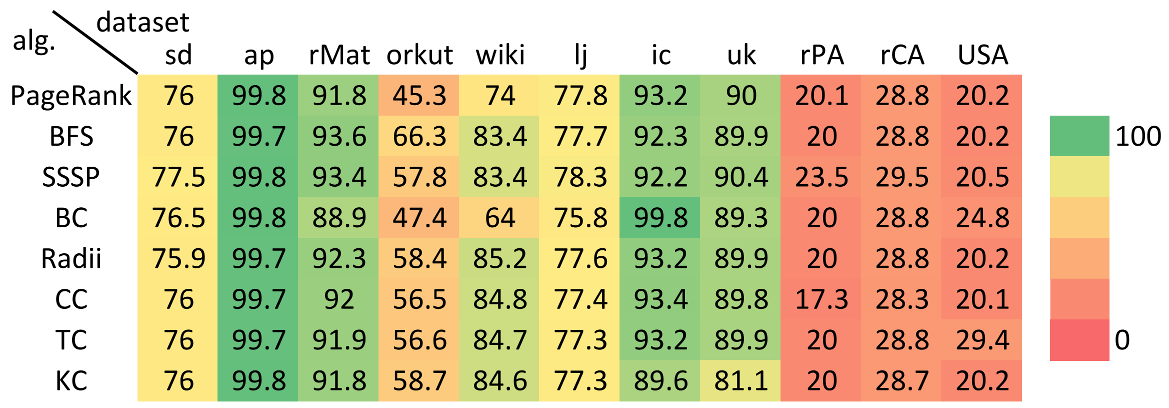
In addition, each scratchpad is augmented with a lightweight compute engine, called the Processing in Scratchpad (PISC) unit, to which computation can be offloaded. PISCs are particularly useful as many graph-based applications are bottlenecked by frequent but simple operations (including atomic) on the graphs data structures. Since we have mapped the most frequently accessed data to our on-chip scratchpads, augmenting each scratchpad with a PISC significantly reduces latency by eliminating many scratchpad-core transfers.

3.3 Workload Characterization

Graph datasets. To guide the design of our proposed architecture, we considered all datasets reported in Section 2.4 except *wv*, *pk*, *sd2*, *lj2*, and *orkut2*. We chose such a large pool of datasets because we wished to perform a detailed workload characterization. The pool we selected offered a broad representation of graph sizes, types, and both graphs that follow the power law and graphs that do not. The key characteristics of the graphs selected are reported in Table 2.2.

Graph-based algorithms. We considered all the algorithms discussed in Section 2.3 except *SpMV*. We did not consider *SpMV* because it was not implemented in the Ligra software framework [102] that we utilized for our evaluation. The key characteristics of those algorithms are outlined in Table 2.1. As shown in Table 2.1, the size of the *vtxProp* entry varies based on the algorithm, from 4 to 12 bytes per vertex. In addition, several algorithms, *e.g.*, *SSSP*, require multiple *vtxProp* structures of variable entry size and stride. Note that the size of *vtxProp* entry determines the amount of on-chip storage required to process a given graph efficiently. Moreover, Table 2.1 also highlights that most algorithms perform a high fraction of random accesses and atomic operations, *e.g.*, *PageRank*. However, depending on the algorithm, the framework might employ techniques to reduce the number of atomic operations. For example, in the case of *BFS*, Ligra performs atomic operations only after checking if a parent was not assigned to a vertex. In this case, the algorithm performs many random accesses, often just as expensive. Note that there are graph frameworks that do not rely upon atomic operations, *e.g.*, GraphMat. Such frameworks partition the dataset so that only a single thread modifies *vtxProp* at a time, and thus the optimization targets the specific operations performed on *vtxProp*. Finally, note that many graph algorithms process only a subset of the vertices per iteration; hence, they maintain a list of active vertices (*active-list*), which incurs a significant performance penalty. Consequently, it is crucial to differentiate algorithms by whether they need to maintain such a list or not.

Many algorithms operating on natural graphs experience $<50\%$ hit-rate on the last-level cache (Figure 3.2(a)), despite a highly-skewed connectivity among the vertices of the graph. Our graph workload characterization, reported in Figure 3.4, reveals that a hit rate of 50% or greater can be achieved on the randomly-accessed portion of the vertex data (*vtxProp*). As shown in the figure, for graphs that follow the power law, up to 99% of



alg. \ dataset	sd	ap	rMat	orkut	wiki	lj	ic	uk	rPA	rCA	USA
PageRank	76	99.8	91.8	45.3	74	77.8	93.2	90	20.1	28.8	20.2
BFS	76	99.7	93.6	66.3	83.4	77.7	92.3	89.9	20	28.8	20.2
SSSP	77.5	99.8	93.4	57.8	83.4	78.3	92.2	90.4	23.5	29.5	20.5
BC	76.5	99.8	88.9	47.4	64	75.8	99.8	89.3	20	28.8	24.8
Radii	75.9	99.7	92.3	58.4	85.2	77.6	93.2	89.9	20	28.8	20.2
CC	76	99.7	92	56.5	84.8	77.4	93.4	89.8	17.3	28.3	20.1
TC	76	99.7	91.9	56.6	84.7	77.3	93.2	89.9	20	28.8	29.4
KC	76	99.8	91.8	58.7	84.6	77.3	89.6	81.1	20	28.7	20.2

Figure 3.4: Accesses to the 20% most connected vertices. The heat map shows the fraction of accesses to *vtxProp* that refer to the 20% most-connected vertices. 100 indicates that all accesses are to those vertices. Data for *twitter* is omitted because of its extremely large runtime.

the *vtxProp* requests can be served by accommodating just the top 20% most-connected vertices in on-chip storage, which is practical for many graphs encountered in real-world applications. The per-vertex storage requirement depends on the algorithm: for example, Ligra uses 4 bytes for *BFS*, 8 bytes for *PageRank*, and 12 bytes for *Radii* (Table 2.1). Our analysis of storage requirements reveals that 20% of the vertices for the graphs in Table 2.2 can be mapped to a fraction of today's on-chip storage sizes. Among the graphs considered, only *uk* and *twitter* would require more than 16MB of on-chip storage to attain that goal. *uk* requires up to 42MB, whereas *twitter* requires 64MB. For instance, by re-purposing IBM Power9 processor's total L2 + L3 caches (132MB) [12] to store 20% vertices, up to 164 million vertices (1.64 billion edges, assuming R-MAT's [36] default edge-vertex ratio) could be allocated to on-chip storage.

3.4 OMEGA Architecture

The OMEGA architecture, shown in Figure 3.5, employs a heterogeneous storage architecture that comprises both conventional caches and distributed scratchpads. Each scratchpad is augmented with a lightweight compute-engine (PISC), which executes atomic operations offloaded from the core. The scratchpads store the most-accessed portion of the

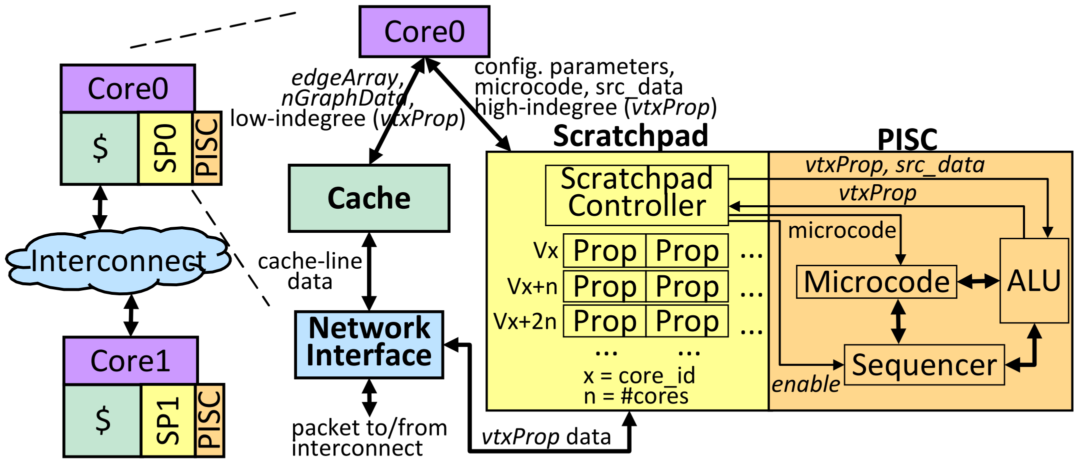


Figure 3.5: OMEGA architecture. OMEGA's heterogeneous cache/SP architecture: for each core, OMEGA adds a scratchpad and a PISC. For the sample graph of Figure 3.6, the *vtxProp* for the most-connected vertices (V0 to V3) is partitioned across all on-chip scratchpads. The rest of the vertices are stored in regular caches. The PISCs execute atomic instructions that are offloaded from the cores.

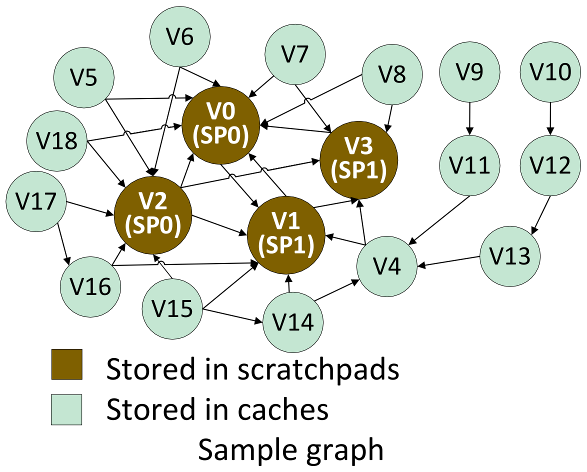


Figure 3.6: Sample power law graph. Vertices are ordered based on their in-degree, with lower IDs corresponding to higher connectivity.

vtxProp data structure, which is where most of the random accesses occur. By doing so, most accesses to *vtxProp* are served from on-chip storage, and off-chip memory accesses are minimized. In contrast, the *edgeArray* and *nGraphData* data structures are stored in conventional caches, since they are mostly accessed sequentially, thus benefiting from cache optimization techniques. While the scratchpads help minimize off-chip memory accesses, the PISCs reduce overheads of executing atomic operations on general-purpose cores. These overheads are due to on-chip communication latency related to frequent *vtx-Prop* accesses from remote scratchpads and atomic operations, causing the core's pipeline to be on-hold until their completion[81]. The PISCs, each co-located with a scratchpad, minimize these overheads by providing computational capability near the relevant data location, thus facilitating the atomic update by leveraging the two co-located units. In contrast, a number of inefficiencies arise if a core is used for this computation, even a local one: first, the significant performance penalty for entering and exiting an interrupt service routine, up to 25ns [23]. Second, since a majority of the atomic operations are simple, a simple PISC can complete them using less energy than a full-blown CPU.

To illustrate the operation of the OMEGA architecture, we use the sample graph in Figure 3.6. The vertices are ordered by their decreasing in-degree value, with V0 being first. OMEGA partitions the *vtxProp* of the most-connected vertices, in our case V0 to V3, among all on-chip scratchpads. All other data – that is, the *vtxProp* for the remaining vertices (V4 to V18), *edgeArray*, and *nGraphData* – is stored in the regular cache hierarchy. This partitioning scheme enables OMEGA to serve most *vtxProp* accesses from on-chip scratchpads by providing storage for just a small portion of the structure. For graphs that follow the power law, it should be sufficient to store just about 20% of *vtxProp* in scratchpads for OMEGA to serve 80% of the accesses from there.

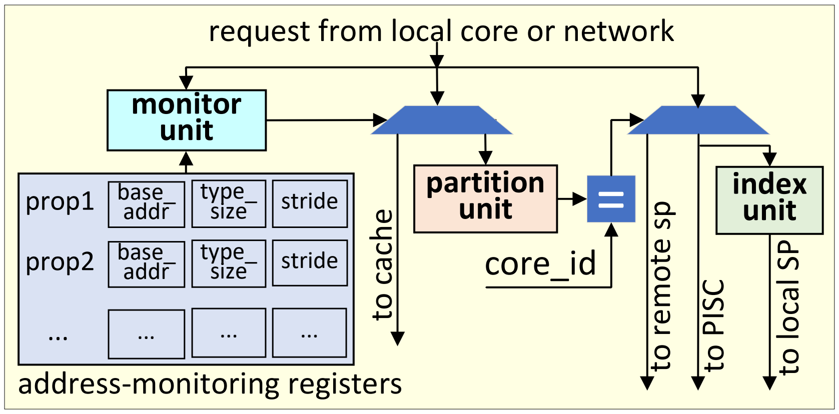


Figure 3.7: Scratchpad controller. The scratchpad controller orchestrates access to the scratchpad. It uses a pre-configured set of address monitoring registers to filter requests destined to the scratchpads. The partition unit determines if a request is to a local scratchpad or a remote one. To identify the scratchpad line of a request, the *index unit* is employed.

3.4.1 Scratchpad architecture

A high-level scratchpad architecture is also shown in Figure 3.5. The scratchpad is organized as a directly-mapped storage. For each line, we store all *vtxProp* entries (*Prop*s) of a vertex; thus, all *Prop*s of a vertex can be retrieved with a single access, which is beneficial for the efficient execution of atomic operations. In addition, an extra bit is added for each *vtxProp* entry to track the *active-list* using a dense representation [102]. The access to the scratchpad is managed by the scratchpad controller, as discussed below.

Scratchpad controller. Figure 3.7 shows the scratchpad controller. It includes a set of *address-monitoring registers*, the *monitor unit*, the *partition unit*, and the *index unit*. The scratchpad controller receives normal read and write requests, as well as atomic-operation requests from both the local core and other scratchpads via the interconnect. Upon receiving them, the *monitor unit* determines if the request should be routed to the regular caches or to the scratchpads. For this purpose, the *monitor unit* relies on a set of *address monitoring registers*, which are shown on the left-side of Figure 3.7. For each *vtxProp*, we maintain its *start_addr*, *type_size*, and *stride*. The *start_addr* is the same as the base address of the *vtxProp*. The *type_size* is the size of the primitive data type stored by the *vtxProp*. For instance, for *PageRank*, the "next_rank" *vtxProp* maintains a primitive data type of "dou-

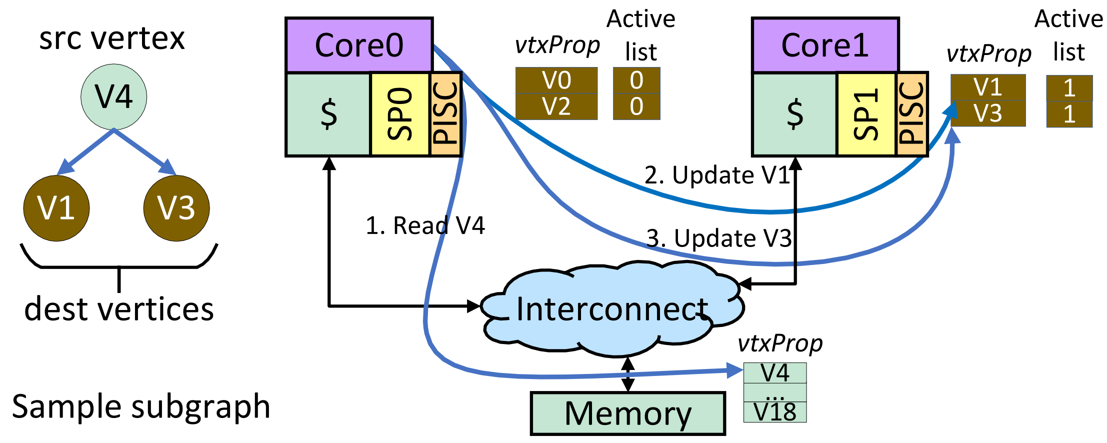


Figure 3.8: Atomic operation offloading. The PISC executes atomic operations offloaded by the cores. The figure shows Core0 executing *PageRank* on the sample graph shown in the left-side. Core0 reads the *vtxProp* of V4 from memory and utilizes this value to update the *vtxProp* of all of V4's neighbors (V1 and V3).

ble"; hence, its *type_size* would be 8 bytes. The *stride* is usually the same as the *type_size*, except when the *vtxProp* is part of a "struct" data structure. In that case, it can be determined by subtracting the first two consecutive *vtxProp* addresses. All of these registers are configured by the graph framework at the beginning of an application's execution. If the scratchpad controller associates a request with the regular cache, it ignores it, as it will be handled by the regular cache controller. If it is for the scratchpads, the *partition unit* is used to determine whether the request targets a local scratchpad or a remote one. If the request is for a remote scratchpad, the scratchpad controller forwards the request to it via the interconnect using a special packet. The *index unit* identifies the line number of the request to perform the actual scratchpad access. For an atomic operation request, the scratchpad controller reads the required *vtxProp* of the request from the scratchpad, initiates its atomic execution on the PISC engine, and then writes the result back to the scratchpad. Note that, while the execution of an atomic operation is in progress, the scratchpad controller blocks all requests that are issued to the same vertex.

3.4.2 PISC (Processing-in-Scratchpad) unit

When we deployed the on-chip scratchpads to store the most-accessed portion of the *vtxProp* and, thus, bound most of the random-access requests to on-chip storage, we found that these requests were most often to remote scratchpads. That is, the general-purpose cores would issue requests that, for the most part, were served by remote scratchpads. This effect creates bottlenecks due to interconnect transfer latency. Thus, to address this issue, we augmented our solution with simple PISC engines to offload such requests from the processor cores. Due to their simplicity, PISCs enable lower energy and latency for data accesses, similar to the benefits of Processing-in-Memory (PIM) units [81]. Each scratchpad is augmented with a PISC engine that executes the atomic operations of the algorithm in execution, *e.g.*, floating-point addition for *PageRank* algorithm. Figure 3.8 illustrates how PISCs work. The figure shows the steps involved when Core0 executes the *PageRank* algorithm on the sample graph shown in the left-side of the figure. First, Core0 reads the *vtxProp* of the source vertex V4 from memory. Then, it utilizes this value to update the *vtxProp* of all its neighbors (V1 and V3). Without a PISC, Core0 needs to read the *vtxProp* of both V1 and V3 from Core1's remote scratchpad, incurring remote scratchpad access latency. However, with a PISC, Core0 can construct an atomic operation message, corresponding to V1 or V3, and send this message to Core1's PISC, which executes the operation on behalf of Core0.

PISC architecture. A high-level architecture of a PISC unit is shown in Figure 3.9. One of the main components of a PISC unit is an ALU engine that implements the atomic operations of the algorithms discussed in Section 2.3. Different algorithms may require different types of operations. For instance, *PageRank* requires "floating-point addition," *BFS* requires "unsigned integer comparison," and *SSSP* requires "signed integer min" and "Bool comparison." The atomic operations of some algorithms, *e.g., SSSP*, are composed of multiple simpler operations, called "microcode." The PISC includes also *microcode registers* to store the microcode that corresponds to an atomic operation. Moreover, it incorporates a

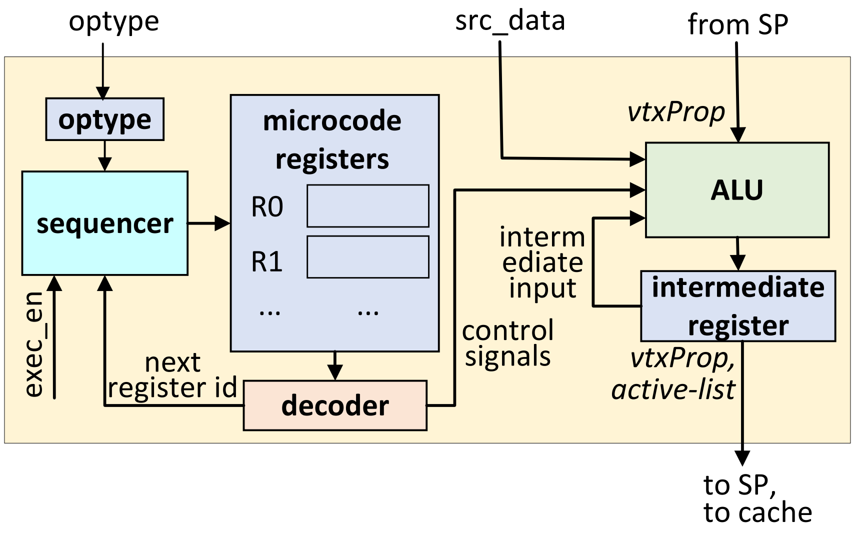


Figure 3.9: PISC architecture. A PISC includes a simple ALU engine that implements atomic operations of a wide range of graph workloads. It also includes a *sequencer* logic that controls the execution of the atomic operation's microcode stored in the *microcode registers*, based on the atomic operation type (*optype*).

```
for s in Vertices
    for d in s.outGoingEdge
        update(s,d,edgeLen[s][d]);
update (s, d, edgeLen) {
        newDist = ShortestLen[s] + edgeLen;
        [atomic_]ShortestLen[d] = min(ShortestLen[d], newDist);
        [atomic_]Visited[d] = 1;
}
```

Figure 3.10: Pseudo-code for *SSSP*. For each pair of vertices, "s" and "d," *SSSP* adds the current shortest length ("ShortestLen") of "s" and the length from "s" to "d" and use the result to update the "ShortestLen" of "d." *SSSP* also marks the vertex "d" as visited.

sequencer module that is responsible for interpreting the incoming atomic operation command, and controlling the execution of the operation's microcode. The *sequencer* is also responsible to read the *vtxProp* of the vertex being processed from the locally-attached scratchpad and input this value along with the *src_data* to the ALU module. Finally, it carries out the task of writing the result of the operation back to the scratchpad.

Maintaining the *active-list*. As discussed previously, several graph-based algorithms leverage an *active-list* to keep track of the set of vertices that must be processed in the next iteration of the algorithm. As the algorithm proceeds, the *active-list* is frequently

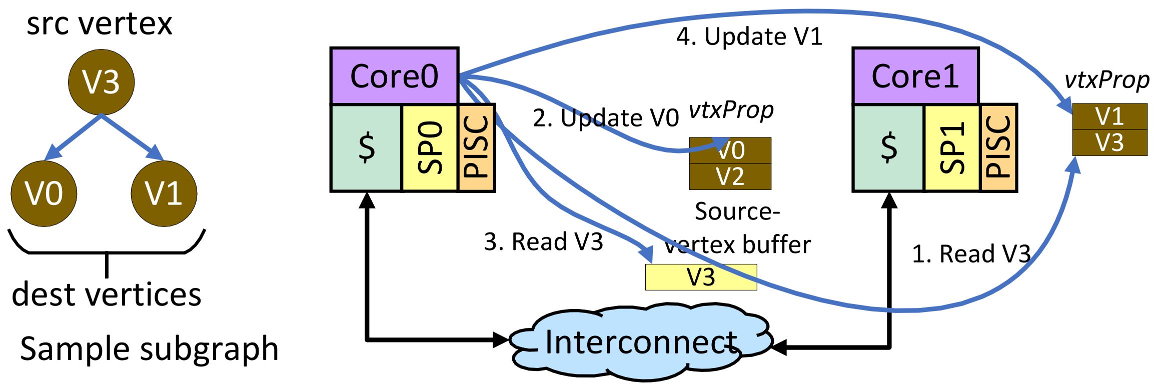


Figure 3.11: Source vertex buffer. While processing the graph shown in the left-side, a typical graph algorithm first read V3's *vtxProp* from a remote scratchpad (SP1). OMEGA then keeps a local copy of it in the "source vertex buffer." Subsequent reads to the same vertex's *vtxProp* are then served from this buffer.

updated; we offloaded this activity from processor cores, too, to avoid a penalty in on-chip latency due to cores waiting on PISC engines' completion. The *active-list* is stored either in scratchpad or in memory, depending on its type. There are two types of *active-list* data structures: *dense-active-lists* and *sparse-active-lists* [102]. As indicated in Figure 3.9, *active-list* updates are among the I/O signals of the PISC. Indeed, a PISC sets a bit in the scratchpad corresponding to the vertex entry on which the atomic operation is operating when updating a dense active list. To update the *sparse-active-list*, the PISC writes the ID of the active vertex to a list structure, stored in memory via the L1 data cache.

3.4.3 Source vertex buffer

Many graph algorithms first read a source vertex's *vtxProp* (see Table 2.1), apply one or more operations to it, and then use the result (*src_data*) to update the *vtxProp* of the source's adjacent vertices. For instance, the *SSSP* algorithm (pseudo-code shown in Figure 3.10) reads a source vertex's *vtxProp*, in this case *ShortestLen*, adds the *edgeLen* to it, and then uses the result to update the *ShortestLen* of all its adjacent vertices. When executing this sequence of operations using the regular cache hierarchy, a single read to the source vertex will bring the data to L1 cache, and all subsequent read requests (up to the number

```
#pragma omp parallel for
for(int i = 0; i < 4; i++) {curr_pagerank[i] = next_pagerank[i];}
```

Figure 3.12: A code snippet of *PageRank* performing a sequential access.

of outgoing edges of the source vertex) will be served from there. However, when using the scratchpads, distributed across all the cores, a read access to a vertex's information by a remote core could incur a significant interconnect-transfer latency (an average of 17 processor cycles in our implementation). To minimize this latency, we introduce a new read-only small storage-structure called the *source vertex buffer*. Every read request to a source vertex's *vtxProp* is first checked against the contents of this buffer. If the request cannot be served from it, the request is forwarded to the remote scratchpad. Upon a successful read from a remote scratchpad, a copy of the vertex data retrieved will be placed in the buffer to serve future requests to the same vertex. Note that, since all buffer's entries are invalidated at the end of each algorithm's iteration, and since the source vertex's *vtxProp* is not updated until that point, there is no need to maintain coherence between this buffer and the scratchpads. Figure 3.11 illustrates how the buffer works. The figure shows the process that is undertaken when Core0 executes *SSSP* on the sample graph shown on the left-side of the figure, starting with a source vertex of V3. The main task of Core0 is to update the next *ShortestLen* of V3's neighbors. To perform the update for V0, Core0 reads the *ShortestLen* of V3 from SP1, entailing a remote scratchpad access. Upon a successful read, a copy of V3's *ShortestLen* is stored in the buffer. Then, when Core0 attempts to read the *ShortestLen* of V3 again, in this case, to update V1, the read is satisfied from the buffer.

3.4.4 Reconfigurable scratchpad mapping

As previously mentioned, the performance of most graph algorithms is limited by their random access patterns to the *vtxProp* data structure. However, some algorithms also perform a significant number of sequential accesses to the same data structure. For the portion of the *vtxProp* that is mapped to the scratchpads, OMEGA's mapping scheme affects the ef-

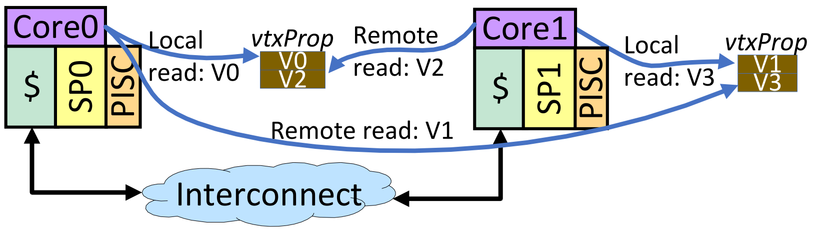


Figure 3.13: Cost of mismatched memory interval for scratchpad mapping and OpenMP scheduling. Given a scratchpad mapping based on interleaving with a memory interval of 1 and an OpenMP scheduling with a memory interval of 2 (for the code snippet shown in Figure 3.12), the mismatch in memory interval causes half of the accesses to *vtxProp* to be served from remote scratchpads.

ficiency of such sequential accesses. To understand this scenario in more detail, we provide a code snippet of *PageRank* in Figure 3.12, which performs a sequential access to *vtxPorp*. The code snippet involves copying *vtxProp* (*next_rank*) to another temporary data structure (*curr_pagerank*). The *vtxProp* is stored on the scratchpad while the temporary structure is stored in cache. Figure 3.13 illustrates how the snippet data accesses are mapped to scratchpad activities when a scratchpad data is not mapped with a matching memory interval. In this figure, OMEGA's interleaving-based mapping is configured with a memory interval of 1, and the OpenMP scheduling configuration assigns an equally-sized memory interval for each thread (a memory interval of 2 for the code snippet shown in Figure 3.12). This mismatch in memory interval causes Core0's read request to the *vtxProp* of V1 and Core1's read request to the *vtxProp* of V2 to be served from remote scratchpads, leading to half of the overall scratchpad accesses to be served remotely. To avoid these remote accesses, the memory interval that OMEGA uses to map the *vtxProp* to the scratchpads can be configured to match that of the size used by the OpenMP scheduling scheme. This setting makes it so potentially remote scratchpad accesses become local ones when performing sequential accesses.

3.4.5 On-chip communication

Communication granularity of scratchpad. Although a large portion of the accesses to *vtxProp* are served from the scratchpads, these accesses still lack spatial locality. Such a deficiency causes most accesses from the cores to be served from remote scratchpads. Consequently, frequent accesses to the remote scratchpads at a cache-line granularity waste on-chip communication bandwidth. To address this aspect, OMEGA accesses the scratchpads at a word-level granularity. The actual size of the scratchpad access depends on the *vtxProp* entry type, and it ranges from 1 byte (corresponding to a "Bool" *vtxProp* entry) to 8 bytes (corresponding to a "double" *vtxProp* entry) in the workloads that we considered (see Table 2.1). For communication with the remote scratchpads, OMEGA uses custom packets with a size of up to 64-bits, as the maximum *type_size* of a *vtxProp* entry is 8 bytes. Note that this size is smaller than the bus-width of a typical interconnect architecture (128 bits in our evaluation), and its size closely resembles the control messages of conventional coherence protocols (*e.g.*, "ack" messages). Using a word-level granularity instead of a conventional cache-block size enables OMEGA to reduce the on-chip traffic by a factor of up to 2x.

3.4.6 Adopting software frameworks

Lightweight source-to-source translation. High-level frameworks, such as Ligra, must be slightly adapted to benefit from OMEGA. To this end, we developed a lightweight source-to-source translation tool. Note that source-to-source translation implies that the resulting framework will be in the same programming language as the original one; hence, no special compilation techniques is required. The tool performs two main kinds of translation. First, it generates code to configure OMEGA's microcode and other registers. To configure the microcode registers, it parses the "update" function, pre-annotated by the framework developers (an example for *SSSP* is shown in Figure 3.10), and generates code comprising a series of store instructions to a set of memory-mapped registers. This code is

```
update (s, d, edgeLen) {
    *mem_mapped_reg1 = ShortestLen[s] + edgeLen;
    *mem_mapped_reg2 = d;
}
```

Figure 3.14: Code generated for *SSSP* with the source-to-source translation tool. The figure shows a microcode for the update function of Figure 3.10. As shown, the new computed shortest length value is written to memory-mapped register, *mem_mapped_reg1*, and the ID of the destination vertex is written to another memory-mapped register, *mem_mapped_reg2*.

the microcode to be written to each PISC. Note that the microcode contains a relatively simple set of operations, mostly implementing the algorithm's atomic operations. For example, the microcode for *PageRank* would involve reading the stored page-rank value from the scratchpad, performing floating-point addition, and writing the result back to the scratchpad. In addition to the microcode, the tool generates code for configuring OMEGA, including the *optype* (the atomic operation type), the start address of *vtxProp*, the number of vertices, the per-vertex entry size, and its stride. This configuration code is executed at the beginning of the application execution. To highlight the functionality of the tool, we show the "update" function of *SSSP* algorithm in Figure 3.10 and its translated code in Figure 3.14. The translated code shows that the new computed shortest length value is written to memory-mapped register, *mem_mapped_reg1*, and the ID of the destination vertex is written to another memory-mapped register, *mem_mapped_reg2*. We also verified the tool's functionality across multiple frameworks by applying it to GraphMat [105], in addition to Ligra [102].

3.5 Graph Preprocessing

OMEGA's benefits rely on identifying highly-connected vertices of a graph dataset and mapping their *vtxProp* to the on-chip scratchpads. Broadly speaking, there are two approaches to achieve this purpose: dynamic and static approaches. With a dynamic approach, the highly-connected vertices can be identified by using a hardware cache with a replacement policy based on vertex connectivity and a word granularity cache-block size, as suggested in other works [57, 109, 87]. However, this solution incurs significant area and power overheads, as each vertex must store tag information. For instance, for *BFS*, the area overhead can reach up to 2x, assuming 32 bits per tag entry and another 32 bits per *vtxProp* entry. To remedy the high overhead of the dynamic approach, a static approach can be utilized. An important example of a static approach is reordering the graph using offline algorithms. Any kind of reordering algorithm is beneficial to OMEGA, as long as it produces a monotonically decreasing/increasing ordering of popularity of vertices. Both in-degree- and out-degree-based reordering algorithms provide such ordering. We found in practice that in-degree-based reordering captures a larger portion of the natural graphs' connectivity, as shown in Table 2.2. Slashburn-based reordering, however, produces suboptimal results for our solution, as it strives to create community structures instead of a monotonically reordered dataset based on vertex connectivity. We considered three variants of in-degree-based reordering algorithms: 1) sorting the complete set of vertices, which has an average-case complexity of *vlogv*, where *v* is the number of vertices; 2) sorting only the top 20% of the vertices, which has the same average-case time complexity; and 3) using an "n-th element" algorithm that reorders a list of vertices so that all vertices stored before the n-th index in the list have connectivity higher than those after (in our case, *n* would be the 20% index mark). This algorithm has a linear average-case time complexity. We chose the third option in our evaluation since it provides slightly better performance and has a very-low reordering overhead. However, a user might find the first option more beneficial if the storage requirement for 20% of the vertices is significantly larger than the available storage.

3.6 Scaling Scratchpad Usage to Large Graphs

Our prior discussion regarding the scratchpad architecture assumes that OMEGA's storage can accommodate a significant portion ($\approx$20%) of the *vtxProp* of a graph. However, for larger graphs, the *vtxProp* of their most-connected vertices does not fit into the scratchpads. Even in this case, OMEGA's scratchpad architecture continues to provide significant benefits primarily because storing the most-connected vertices is the best investment of resources compared to re-purposing the same storage for conventional caches. However, as the size of the graph continues to increase, there is a point where OMEGA's scratchpads would become too small to store a meaningful portion of *vtxProp*, and, consequently, the benefit would be negligible. Below, we discuss two approaches that could enable OMEGA to continue providing benefits, even in this scenario.

1) Graph slicing. [54, 119] proposed a technique, called graph slicing/segmentation, to scale the scratchpad/cache usage of their architecture. In this technique, a large graph is partitioned into multiple slices, so that each slice fits in on-chip storage. Then, one slice is processed at a time, and the result is merged at the end. While the same technique can be employed to scale the scratchpad usage for OMEGA, there are several performance overheads associated with this approach: 1) the time required to partition a large graph to smaller slices, and 2) the time required to combine the results of the processed slices. These overheads increase with the number of slices, a challenge addressed by the next approach.

2) Graph slicing and exploiting power law. Instead of slicing the complete graph, slicing can be applied to the portion of the graph that contains the 20% of the most-connected vertices, which is sufficient to serve most *vtxProp* accesses. This approach can reduce the number of graph slices by up to 5x, and consequently, it reduces the overheads associated with slicing.

3.7 Memory Semantics

Cache coherence. Each *vtxProp* entry of the most-connected vertices is mapped to and handled by only one scratchpad; hence, it does not require access to the conventional caches, whereas all the remaining vertices and other data structures are only managed by the conventional caches. Under this approach, there is no need to maintain data coherence

Table 3.1: Experimental testbed setup

Common configuration
Core: 16 OoO cores, 2GHZ, 8-wide, 192-entry ROB
L1 I/D cache per core: 16KB, 4/8-way, private
cache block size: 64 bytes
Coherence protocol: MESI_Two_Level
memory: 4xDDR3-1600, 12GB/s per channel
interconnect topology: crossbar, 128-bits bus-width
Baseline-specific configuration
L2 cache per core: 2MB, 8-way, shared
OMEGA-specific configuration
L2 cache per core: 1MB, 8-way, shared
SP per core: 1MB, direct, lat. 3-cycles
SP access granularity: 1-8 bytes

among scratchpads or between scratchpads and conventional caches.

Virtual address translation. The local scratchpad controller maps the virtual address of an incoming request into a vertex ID, and it uses this ID to orchestrate scratchpad access, including identifying which scratchpad to access, whether this scratchpad is local or remote. This approach avoids the necessity of incorporating a virtual to physical address translation mechanism into the scratchpads.

Atomic operation. If OMEGA only employs distributed scratchpads without augmenting them with PISC units, execution of atomic operations would be handled by the cores. Since on-chip scratchpad accesses occur at word-level granularity instead of cache-line granularity, the cores lock only the required word address. Other aspects of the implementation remain the same as in a conventional cache coherence protocol.

3.8 Experimental Setup

In this section, we provide the experimental setup that we deployed to evaluate the OMEGA solution.

We modeled OMEGA on gem5 [34], a cycle-accurate simulation infrastructure. We ported the OpenMP implementation of the Ligra framework [102] to gem5 using “m5threads” and we carried out the simulation in “syscall” emulation mode. We compiled Ligra with gcc/g++ using the O3 optimization flag. Our simulation setup is summarized in Table 3.1. Our baseline design is a CMP with 16, 8-wide, out-of-order cores with private 32KB of L1 instruction and data caches, and shared L2 cache with a total storage size matching OMEGA’s hybrid scratchpad+cache architecture. For OMEGA, we maintain the same parameters for the cores and L1 caches as the baseline CMP. Each scratchpad is augmented with a PISC engine. Communication with the caches takes place at ache-line granularity (64 bytes), and communication with the scratchpads occurs at word-size granularity, with sizes ranging from 1 to 8 bytes, depending on the size of the *vtxProp* entry being accessed.

Workloads. We considered the same algorithms and datasets discussed in Section 3.3. Note that *TC* and *KC* present similar results on all our experiments; thus, we report only the results for *TC*. Because *CC* and *TC* require symmetric graphs, we run them on one of the undirected-graph datasets (*ap*). Moreover, because of the long simulation times of gem5, we simulate only a single iteration of *PageRank*. In addition, we simulate only the “first pass” of *BC*, and we use a “sample size” of 16 for *Radii*. Other algorithms are run to completion using their default settings.

3.9 Performance Evaluation

In this section, we present overall performance gains enabled by OMEGA. We then discuss a number of insights on the sources of those gains by inspecting caches hit rates, on-chip traffic analysis, and off-chip memory bandwidth utilization. Furthermore, we provide a comparison of the performance benefit of OMEGA for power-law and non-power-law graphs.

Figure 3.15 shows the performance benefit provided by OMEGA compared to Ligra running on a baseline CMP. OMEGA achieved a significant speedup, over 2x on average,

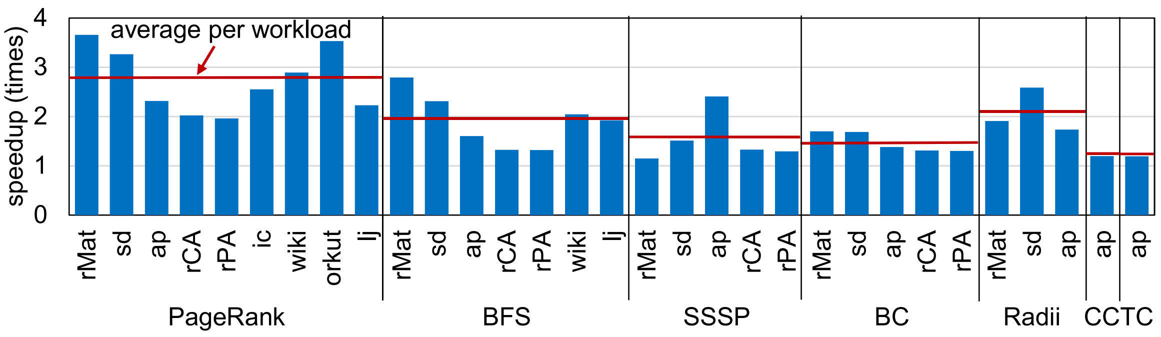


Figure 3.15: OMEGA performance speedup. OMEGA provides 2x speedup, on average, over a baseline CMP running Ligra.

across a wide range of graph algorithms (see Table 2.1) and datasets (see Table 2.2). The speedup highly depends on the graph algorithm. OMEGA achieved significantly higher speedups for *PageRank*, 2.8x on average, compared to others. The key reason behind this is that Ligra maintains various data structures to manage the iterative steps of *PageRank*. In the case of *PageRank* and *TC*, all vertices are active during each iteration; hence, the per-iteration overhead of maintaining these data structures is minimal. Unfortunately, *TC*'s speedup remains limited because the algorithm is compute-intensive; thus, random accesses contribute only a small fraction to execution time. In contrast, for the other algorithms, Ligra processes only a fraction of the vertices in each iteration; therefore, maintaining the data structures discussed above negatively affects the overall performance benefit. However, even with these overheads, OMEGA managed to achieve significant speedups: for *BFS* and *Radii*, an average of 2x, and for *SSSP*, an average of 1.6x.

In addition to the impact of the algorithm, the speedup highly varies across datasets. OMEGA manages to provide significant speedup for datasets for which at least 20% of the *vtxProp* fits in the scratchpads, *e.g.*, *lj*, and *rMat* (whose *vtxProp* fits completely). The key observation here is that, for a natural graph, OMEGA's scratchpads provide storage only for a small portion of the *vtxProp* but manage to harness most of the benefits that could have been provided by storing the complete *vtxProp* in scratchpads. Despite not following a power-law distribution, *rCA* and *rPA* achieve significant performance gains since their *vtxProp* is small enough to fit in the scratchpads. However, compared to other graphs, they

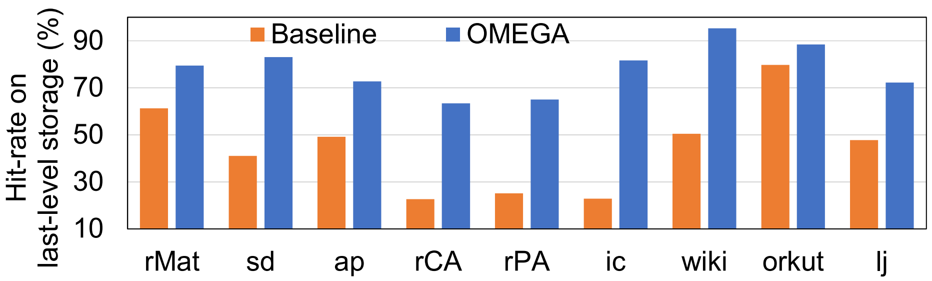


Figure 3.16: Last-level storage hit-rate in *PageRank*. OMEGA's partitioned L2 caches and scratchpads lead to a significantly larger hit-rate, compared to the baseline's L2 cache of the same size.

achieve a smaller speedup because of their low out-degree connectivity (see Table 2.2), which makes the accesses to the *edgeArray* exhibit more irregular patterns.

Cache/Scratchpad access hit-rate. In Figure 3.16, we compare the hit rate of the last-level cache (L2) of the baseline design against that of OMEGA's partitioned storage (half scratchpad and half the L2 storage of the baseline). The plot reveals that OMEGA provides an over 75% last-level "storage" hit-rate, on average, compared to a 44% hit-rate of the baseline. The key reason for OMEGA's higher hit rate is because most of the *vtxProp* requests for the power-law graphs are served from scratchpads.

Using scratchpads as storage. To isolate the benefits of OMEGA's scratchpads, without the contributions of the PISC engines, we performed an experiment running *PageRank* on the *lj* dataset with a scratchpads-only setup. OMEGA achieves only a 1.3x speedup, compared to the >3x speedup when complementing the scratchpads with the PISCs. The lower boost of the scratchpads-only solution results from foregoing improvements in on-chip communication and atomic operation overheads. Computing near the scratchpads using the PISCs alleviates these overheads.

Off- and on-chip communication analysis. As noted by prior work [31], graph workloads do not efficiently utilize the available off-chip bandwidth. We measured OMEGA's DRAM bandwitdh utilization on a range of datasets while running *PageRank* and report our findings in Figure 3.17. The plot indicates that OMEGA manages to improve the utilization of off-chip bandwidth by an average of 2.28x. Note that there is a strong correlation

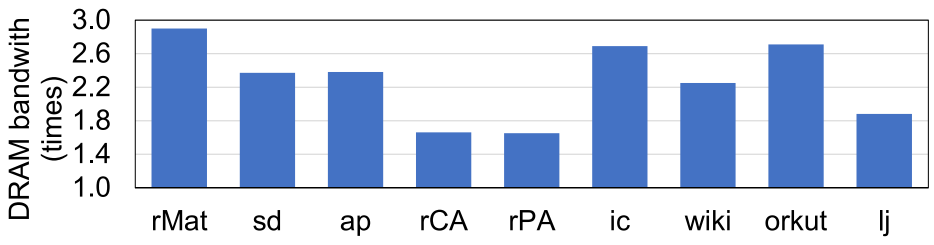


Figure 3.17: DRAM bandwidth utilization of *PageRank*. OMEGA improves off-chip bandwidth utilization by 2.28x, on average.

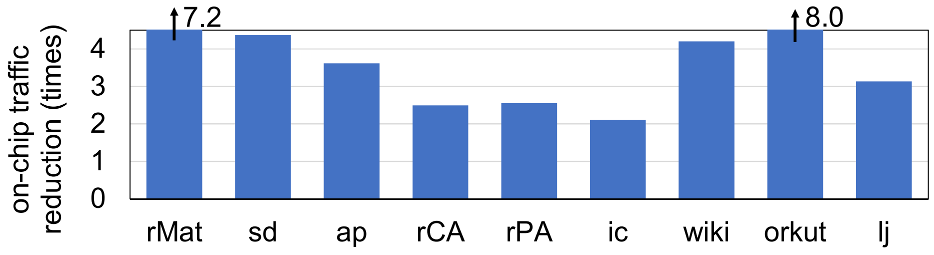


Figure 3.18: On-chip traffic analysis of *PageRank*. OMEGA reduces on-chip traffic by 3.2x, on average.

between the bandwidth utilization and the speedup reported in Figure 3.15 for *PageRank*. Indeed, the bandwidth improvement can be attributed to two key traits of OMEGA: 1) cores are freed to streamline more memory requests because atomic instructions are offloaded to the PISCs, and 2) since most of the random accesses are constrained to OMEGA's scratchpads, the cores can issue more sequential accesses to the *edgeArray* data structure. Furthermore, we found that graph workloads create lots of on-chip communication traffic. Our analysis measuring on-chip traffic volume, reported in Figure 3.18, shows that OMEGA reduces this traffic by over 4x on average. OMEGA minimizes on-chip communication by employing word-level access to scratchpad data and offloading operations to the PISCs.

Non-power-law graphs. Figure 3.19 presents a speedup comparison for two large graphs: one for a power-law graph (*lj*) and another for a non-power-law graph (*USA*), targeting two representative algorithms: *PageRank* (a graph algorithm with no *active-list*) and *BFS* (a graph algorithm with *active-list*). OMEGA's benefit for *USA* is limited, providing a maximum of 1.15x improvement. The reason is that, since *USA* is a non-power-law graph, only approximately 20% of the *vtxProp* accesses correspond to the 20% most-connected

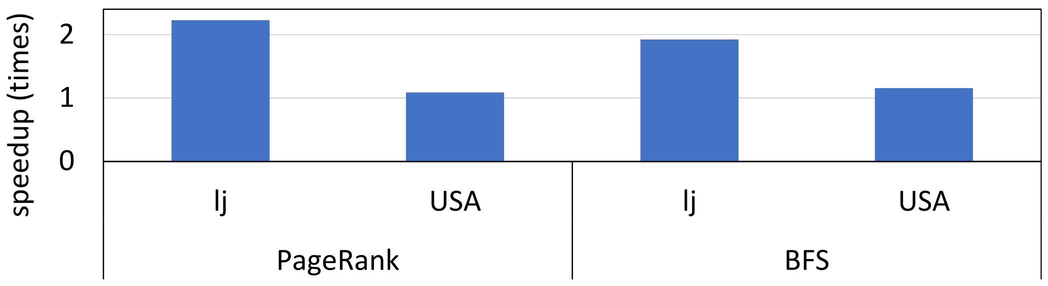


Figure 3.19: Comparison of power-law (*lj*) and non-power-law graphs (*USA*). As expected, OMEGA achieves only a limited speedup of 1.15x on a large non-power-law graph.

vertices, compared to 77% for *lj*.

3.10 Sensitivity Studies

This section provides sensitivities studies related to the sizes of scratchpad and input datasets.

Scratchpad size sensitivity. In our analyses so far, OMEGA is configured with scratchpad-sizes that enable it to accommodate around 20% or more of the *vtxProp*. In this Section, we present a scratchpad-size sensitivity study for *PageRank* and *BFS* on the *lj* dataset, over a range of scratchpad sizes: 16MB (our experimental setup), 8MB, and 4MB. We maintained identical sizes of L2 cache for all configurations, as in our experimental setup (16MB). The results, reported in Figure 3.20, show that OMEGA managed to provide a 1.4x speedup for *PageRank* and a 1.5x speedup for *BFS* even when employing only 4MB scratchpads, which accommodate only 10% of the *vtxProp* for *PageRank* and 20% of the *vtxProp* for *BFS*. As shown in the figure, 10% of the vertices are responsible for 60.3% of *vtxProp* for *PR*, and 20% of the vertices are responsible for 77.2% of the *vtxProp* for *BFS*. Note that this solution point entails significantly less storage than the total L2 cache of our baseline.

Scalability to large datasets. This study estimates the performance of OMEGA on very large datasets: *uk* and *twitter*. Since we could not carry out an accurate simulation, due to the limited performance of gem5, we modeled both the baseline and OMEGA in a

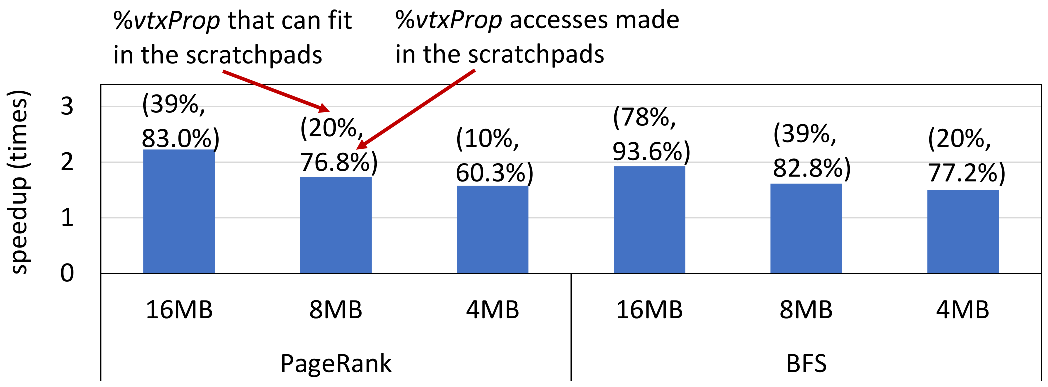


Figure 3.20: Scratchpad sensitivity study. OMEGA provides 1.4x speedup for *PageRank* and 1.5x speedup for *BFS* with only 4MB of scratchpad storage.

high-level simulator. In the simulator, we retained the same number of cores, PISC units, and scratchpad sizes, as in Table 3.1. In addition, we made two key approximations. First, the number of DRAM accesses for *vtxProp* is estimated based on the average LLC hit-rate that we obtained by running each workload on the Intel Xeon E5-2630 v3 processor and using the Intel's VTune tool. The number of cycles to access DRAM is set at 100 cycles, and we also accounted for the LLC and scratchpad access latencies. Second, the number of cycles for a remote scratchpad access is set at 17 cycles, corresponding to the average latency of the crossbar interconnect. For the baseline solution, we configured the number of cycles required to complete an atomic operation execution to match the value we measured for the PISC engines: this is a conservative approach, as the baseline's CMP cores usually take more cycles than the PISC engines. Figure 3.21 reports our findings, which also include the result from our gem5 evaluation for validation purposes. We note that the high-level estimates are within a 7% error, compared to the gem5 results. As shown in the figure, OMEGA achieves significant speedup for the two very large graphs, even if they would benefit from much larger scratchpad resources. For instance, for *twitter*, OMEGA manages to provide a 1.7x speedup on *PageRank*, by providing storage only for 5% of the *vtxProp*. Note that 5% of the most-connected vertices are responsible for 47% of the total *vtxProp* accesses. This highly-skewed connectivity is the reason why OMEGA is able to provide a valuable speedup even with relatively small storage.

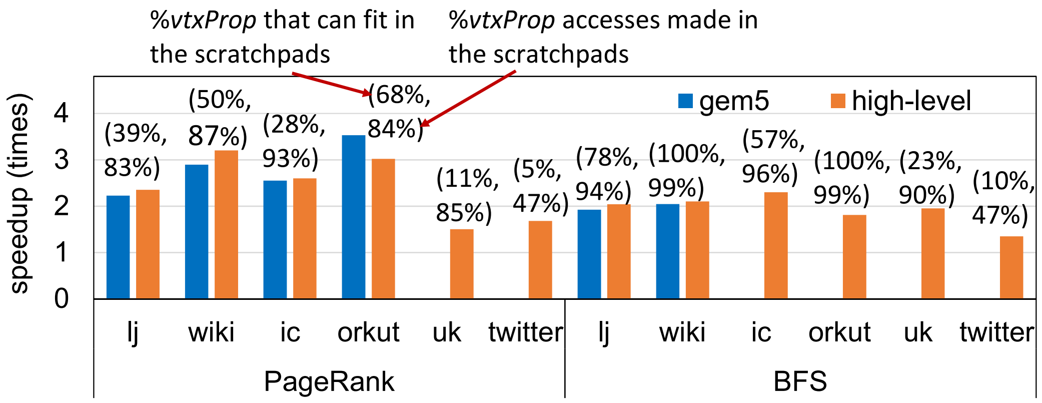


Figure 3.21: Performance on large datasets. A high-level analysis reveals that OMEGA can provide significant speedups even for very large graphs: a 1.68x for *PageRank* runnning on *twitter*, our largest graph, when storing only 5% of *vtxProp* in scratchpads; and a 1.35x for *BFS*, storing only 10% of *vtxProp*.

Table 3.2: Peak power and area for a CMP and OMEGA node

Component	**Baseline CMP node**		**OMEGA node**	
	Power (W)	Area (mm^2)	Power (W)	Area (mm^2)
Core	3.11	24.08	3.11	24.08
L1 caches	0.20	0.42	0.20	0.42
Scratchpad	N/A	N/A	1.40	3.17
PISC	N/A	N/A	0.004	0.01
L2 cache	2.86	8.41	1.50	4.47
Node total	6.17	32.91	6.21	32.15

3.11 Area, Power, and Energy Analysis

This section presents area and power overheads, as well as energy benefits provided by OMEGA, as compared to CMP-only baseline.

We used McPAT [68] to model the core, and Cacti [101] to model the scratchpads and caches. We synthesized PISC's logic in IBM 45nm SOI technology. Note that the PISC's area and power is dominated by its floating-point adder. We referred to prior works for the crossbar model [16]. We used the same technology node, 45nm, for all of our components. We re-purposed half of the baseline's L2 caches space to OMEGA's on-chip scratchpads. Table 3.2 shows the breakdown of area and peak power for both the baseline CMP and OMEGA. The OMEGA node occupies a slightly lower area (-2.31%)

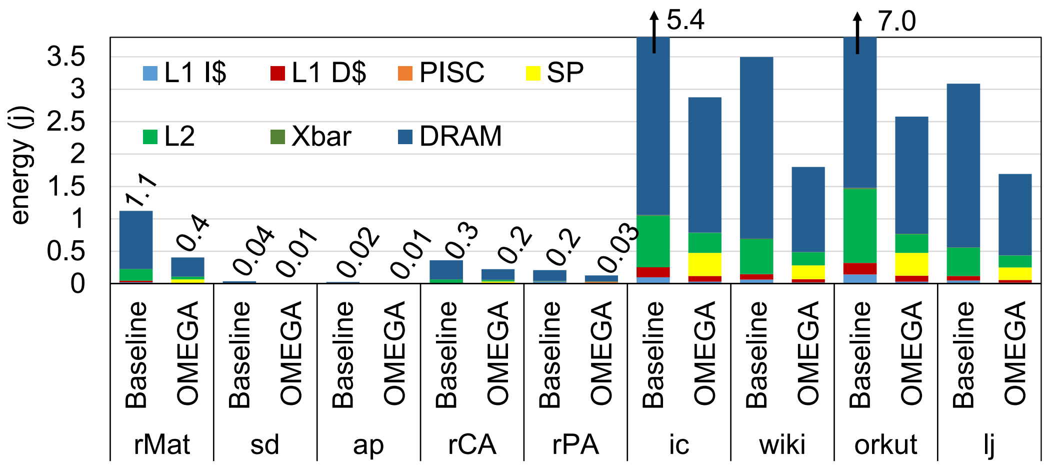


Figure 3.22: Comparison of energy spent in memory activities for *PageRank*. OMEGA requires less energy to complete the algorithm due to less DRAM traffic and shorter execution time. The energy efficiency of OMEGA's scratchpads over the caches contributes to the overall energy savings.

and consumes slightly higher in peak power (+0.65%) compared to the baseline CMP. The slightly lower area is due to OMEGA's scratchpads being directly mapped and thus not requiring cache tag information.

In Figure 3.22, we provide energy analysis on *PageRank*, considering a wide-range of datasets. Since OMEGA's modifications to the baseline are limited to the memory hierarchy, we provide a breakdown of the energy consumption only for the memory system, including the DRAM. The figure reveals that OMEGA provides 2.5x energy saving, on average, compared to a CMP-only baseline. The saving is because OMEGA's scratchpads consume less energy compared to caches. In addition, OMEGA utilizes lower DRAM energy because most of the accesses to *vtxProp* are served from on-chip scratchpads.

3.12 Bibliography

In this Section, we provide a summary of related works. We first discuss solutions that optimize the execution of graph analytics on general-purpose architectures. Then, we describe near-memory based solutions, followed by specialized architectures that target graph analytics. We also mention solutions based on GPUs and other vector processing architectures. Finally, we provide a brief discussion, which highlights the benefit of the OMEGA solution compared to other relevant works.

Optimizations on general-purpose architectures. [31] characterizes graph workloads on Intel's Ivy Bridge server. It shows that locality exists in many graphs, which we leverage in our work. [45] minimizes the overhead of synchronization operations for graph applications on a shared-memory architecture, by moving computation to dedicated threads. However, dedicated threads for computation would provide a lower performance/energy efficiency compared to our lightweight PISC architecture. [119] proposes both graph reordering and segmentation techniques for maximizing cache utilization. The former provides limited benefits for natural graphs, and the latter requires modifying the framework, which we strive to avoid. [120] proposes domain-specific languages, thus trading off application's flexibility for higher performance benefits.

Near-memory processing and instruction offloading. [24, 81] propose to execute all atomic operations on *vtxProp* in the off-chip memory because of the irregular nature of graph applications. However, our work shows that processing many natural graphs involves frequency-based access patterns, which can be exploited by using on-chip scratchpads, thus reducing the access to off-chip memory.

Domain-specific and specialized architecture. Application-specific and domain-specific architectures, including those that utilize scratchpads [90, 23, 54, 103], have recently flourished to address the increasing need of highly efficient graph analytics solutions. However, the focus of these architectures is on performance/energy efficiency, foregoing the flexibility of supporting software frameworks and applications. In addition, these solutions do not fully exploit the frequency-based access pattern exhibited by many natural graphs.

GPU and vector processing solutions. GPU and other vector processing solutions, such as [111, 22, 88], have been increasingly adopted for graph processing, mostly in the form of sparse matrix-vector multiplication. However, the diverse structure of graphs limits the viability of such architectures.

Heterogeneous cache block size architecture. [57] proposed an access mechanism of variable cache-line sizes depending on the runtime behavior of an application. Similarly, OMEGA provides variable storage access sizes (cache-line-size for accessing caches and word-size for accessing scratchpads), depending on the type of data structure being accessed.

Finally, we provide a summary of a comparison of OMEGA against the most relevant prior works in Table 3.3. As shown in the table, OMEGA surpasses previous architectures primarily because it exploits the power-law characteristics of many natural graphs to identify the most-accessed portions of the *vtxProp*, and it utilizes scratchpads to efficiently access them. In addition, most of the atomic operations on *vtxProp* are executed on lightweight PISC engines instead of CPU cores. Furthermore, OMEGA is easily deployable in new graph frameworks and applications because it maintains the existing general-purpose architecture while also providing small but effective additional components.

3.13 Summary

In this chapter, we presented OMEGA, a hardware architecture that optimizes the memory subsystem of a general-purpose processor to run graph frameworks without requiring significant additional changes from application developers. OMEGA provides on-chip distributed scratchpads to take advantage of the inherent frequency-based access patterns of graph algorithms when processing natural graphs, providing significantly more on-chip accesses for irregularly accessed data. In addition, the scratchpads are augmented with atomic operation processing engines, providing significant performance gains. OMEGA achieves on average a 2x boost in performance and a 2.5x energy savings, compared to a same-sized baseline CMP running a state-of-the-art shared-memory graph framework. The area and peak power needs of OMEGA are comparable to that of the baseline node, as it trades cache storage for equivalently sized scratchpads.

Overall OMEGA achieves its performance boost in computing on graphs by leveraging specialized storage structures, so to limit cache accesses. However, as the number of processor cores increases, other activities become preponderant in the computation, including the communication traffic among cores. The following chapter addresses specifically this issue by presenting a hardware solution that strives to limit this type of on-chip traffic.

Table 3.3: Comparison of OMEGA and prior related works

	CPU	**GPU**	**Locality Exists [31]**	**Graphicionado [54]**	**Tesseract [23]**	**GraphIt [120]**	**GraphPIM [81]**	**OMEGA**
leveraging power law	limited	limited	**yes**	no	no	limited	no	**yes**
memory subsystem	cache	cache	cache	**scratchpad**	cache	cache	cache	**cache & scratchpad**
logic for non-atomic-operation	**general**	**general**	**general**	specialized	**general**	**general**	**general**	**general**
logic for atomic-operation	general	general	general	**specialized**	general	general	**specialized**	**specialized**
on-chip communication granularity	cache-line	cache-line	cache-line	**word**	**word**	cache-line	**word**	**cache-line & word**
offloading target for atomic operation	N/A	N/A	N/A	**scratchpad**	off-chip memory	N/A	off-chip memory	**scratchpad**
compute units for atomic operation	CPU	GPU	CPU	**specialized**	CPU	CPU	**specialized**	**CPU & specialized**
framework independence&modifiability	**yes**	**yes**	**yes**	limited	**yes**	limited	**yes**	**yes**
propose software-level optimizations	**yes**	**yes**	partially	no	no	**yes**	no	no

CHAPTER 4

On-path Message Coalescing for Scalable Graph Analytics

This chapter focuses on addressing the limited scalability of the OMEGA solution, discussed in the Chapter 3, to systems with a large number of cores. OMEGA offloads the execution of atomic operations from general-purpose cores to specialized compute engines, co-located with on-chip storage units, to avoid the round-trip traffic between cores and the storage units, thus minimizing network traffic. However, when a system has many cores, even the traffic generated by these offloaded atomic operations alone is sufficient to cause network congestion and, consequently, performance and energy inefficiencies. Recent architectures have been striving to address these inefficiencies. They leverage a network of 3D-stacked memory blocks, placed above processing elements; an example of such architectures are Hybrid Memory Cubes (HMCs) [23, 118, 41]. HMCs offer high internal bandwidth between the processing elements and their corresponding local memory blocks; thus, they can access their local data efficiently. However, due to the irregular-access patterns in graph analytics, significant network traffic between local processing elements and remote memory blocks still remains. Hence, HMCs are also affected by limited performance and energy inefficiencies, even if at a lesser degree.

In this chapter, we aim to minimize the performance and energy inefficiencies described above by devising an architecture that takes advantage of the power-law characteristic. To achieve this goal, we rely on one of the strategies discussed in Section 1.4. Specifically, we explore the benefit of coalescing graph analytics data messages in the network. Due to the power-law characteristic described in Chapter 2, and also leveraged in the OMEGA solution of Chapter 3, it is common for multiple atomic operations originating from different processing elements to update the same vertex. Since these operations are similar within a given graph application, it should be possible to coalesce multiple operations to the same destination vertex in the network, thus reducing the network traffic, and consequently improving performance and energy efficiencies.

To accomplish the above goal, we must overcome a few challenges: first, it is conventionally assumed that updates to the same vertex are atomic, *i.e.*, each update to a vertex should be completed before another update to that same vertex is initiated. Second, multiple vertex-update messages can be fused only if they reach a same location in the interconnect at approximately the same time, a generally unlikely scenario.

We debunk the atomicity requirement by observing that, when update operations are associative and commutative, they can indeed be decomposed into multiple partial operations, and do not require atomic execution. We note that many popular graph-based algorithms entail updates that satisfy those two properties. To increase the opportunity for message coalescing, we reorganize the work at each compute node so that vertex-update messages are generated in destination-vertex order, so as to emit updates from multiple sources to a same destination in close sequence, before moving to the next destination. This scheduling mechanism greatly increases the opportunities for coalescing, as messages to a same destination are more likely to be en-route at the same time.

4.1 Motivational Study

A number of recently proposed works have leveraged a network of *HMC* cubes for execution of graph-based algorithms [23, 118, 41]. These proposals place a processing unit under each memory cube, so that each unit enjoys energy-efficient access to a large data footprint locally; the inter-cube interconnect is based on either a mesh or dragonfly

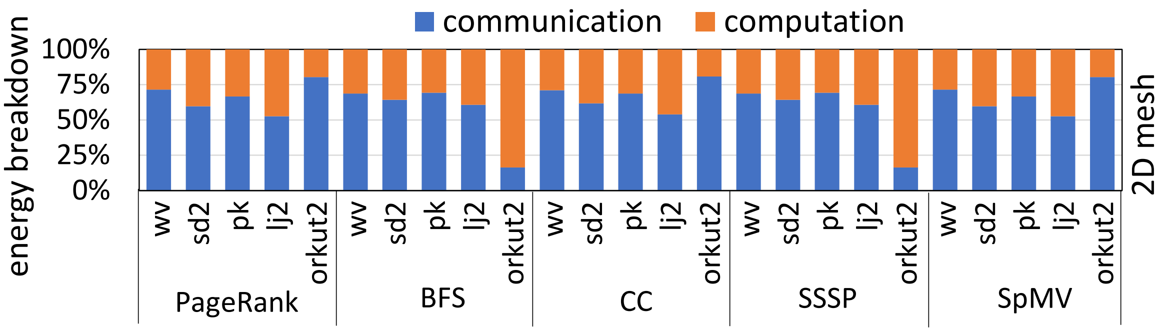


Figure 4.1: Energy breakdown: communication is responsible for 16% to 80% of energy consumption for many algorithms/datasets.

topology, the latter being particularly beneficial if the graph connectivity has a relatively higher density within the dragonfly clusters. To estimate the potential benefits of our message coalescing pursuit, we carried out a simple analysis on a HMC model. Specifically, we simulated Tesseract, a domain-specific architecture [23], with 16 HMCs and 32 vaults per HMC. We utilized CasHMC [58] to model each individual HMC, and BookSim [59] to model the network among the 16 HMCs. First, we measured the total energy consumed by the Tesseract model. Then, the energy spent in communication was measured by subtracting the computation energy, which we evaluated by setting up an ideal crossbar interconnect topology with no delay in inter-cube communication. For this evaluation, we considered several graph-based algorithms and datasets. The algorithms are discussed in Section 2.3 and the datasets are described in Section 2.4. Based on our findings, presented in Figure 4.1, communication is responsible for 62% of overall energy consumption, on average. Hence, we conclude that **inter-cube communication is the primary source of energy consumption in the execution of graph-based algorithms**, even on recent HMC-based solutions. Key contributors to this energy consumption are 1) interconnect contention, which increases execution time and, in turn, static energy consumption, and 2) high interconnect utilization, which drives dynamic energy consumption. Since message coalescing could reduce both, we carried out further analysis on the same algorithms and datasets to find that up to 80% of the total interconnect messages could be eliminated under ideal coalescing conditions, where we assume an architecture that takes full advantage of all message-coalescing opportunities.

4.2 MessageFusion Overview

The solution discussed in this chapter, called MessageFusion, is rooted in the observation that there is potential for coalescing multiple vertex-update messages traveling to a same destination, thus providing an opportunity to reduce network traffic. The reduction in network traffic reduces overall execution time and, consequently, static energy consumption. In addition, dynamic energy consumption is also affected in a favorable way, as hardware resources experience lower utilization when transferring fewer messages. To accomplish this goal, we must overcome a couple of challenges. The first challenge stems from an assumption that each update operation to a vertex must be executed atomically, *i.e.,* each operation must be completed before another operation to the same vertex is initiated. The second challenge is caused by a general trait of graph analytics' algorithms, such that opportunities for coalescing messages because they naturally reach a same interconnect node at the same time is minimal. To improve the opportunity window, one could use a fairly large buffer at each interconnect router. However, this choice entails a significant energy cost and, potentially, even a performance cost, if messages are waiting to be matched for a long time.

MessageFusion accomplishes the goal of reducing network traffic by coalescing vertex-update messages while in transit to their destination. We observe that when update operations are associative and commutative, they can indeed be decomposed into multiple partial operations, and thus do not require atomic execution. We note that many popular graph-based algorithms entail updates that satisfy these two properties, a trait that has been leveraged already in several other high-impact works for other goals, such as [60, 53].

Finally, to increase the opportunity for message coalescing, MessageFusion reorganizes the work at each compute node so that vertex-update messages are generated in destination-vertex order. Traditional graph algorithm schedulers proceed by completing all the work for one source vertex, then emitting vertex-update messages to their destinations before moving on to the next source vertex. In MessageFusion, scheduling at each node is setup so as to emit updates from multiple sources to the same destination in close sequence, before moving to the next destination. This scheduling mechanism greatly enhances the opportunities for coalescing, as messages to a same destination are more likely to be en-route at the same time. Note that edge-data information is commonly organized in a source-vertex order to promote sequential accesses to this data. Thus, we deploy reordering mechanisms to sort the edge-data structure by destination vertex offline. When beneficial to specific algorithms, we also sort vertex-update messages dynamically.

4.3 MessageFusion Architecture

In light of the analysis presented, we developed a novel architecture, called MessageFusion, based on an optimized HMC architecture. It includes two hardware modules: *reduce* and *ordering*. *Reduce* modules are deployed along with each router and each vault: they are responsible for coalescing messages emitted by the vault or traversing the router, whenever an opportunity arises. *Ordering* modules augment each vault and the overall HMC: they sort messages emitted by destination vertex, so as to maximize the opportunity of coalescing during interconnect transfers. Figure 4.2 highlights the deployment of these modules throughout the HMC architecture. Below, we briefly discuss the optimized HMC baseline, and then detail the design of the two modules.

4.3.1 Baseline architecture

Our two proposed hardware modules are deployed alongside an HMC architecture optimized for graph processing. Specifically, we consider the Tesseract architecture [23], comprising HMCs of 32 vaults each. High-speed SerDes links connect each cube to other cubes via a router. Each vault includes a processing element (PE) and storage. The PE is an optimized graph analytics unit implementing a vertex-centric approach, following the

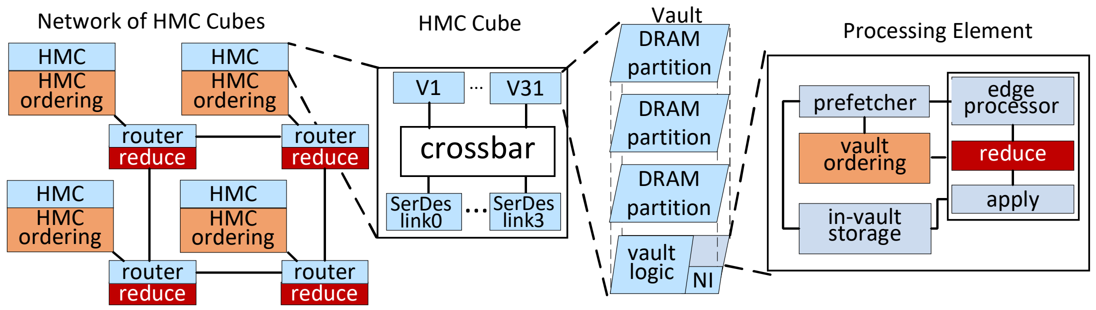


Figure 4.2: The MessageFusion Architecture entails the deployment of two novel hardware modules: *reduce* and *ordering*, which augment the HMC architecture at vaults, cubes and router locations.

types described in [54, 41]. Each PE includes three main units, sketched in Figure 4.2: *reduce* and *apply*, corresponding to the two computation phases discussed in Section 2.5; and *edge-processor* used by algorithms that process weighted-edges [54]. The PEs implement an efficient message-passing communication mechanism, and utilize specialized messages for synchronization at the completion of each iteration of a graph algorithm. Figure 4.3 outlines the in-vault storage architecture in the baseline design. As shown, storage is organized as three scratchpads (rather than conventional caches of [23]). The first two are called *source* and *destination* scratchpads and store *vertex-property* information (described in Section 2.5) in a distributed fashion across the entire system. Each entry in the *source* scratchpad holds the vertex property of a vertex accessed in the current iteration, its starting memory address, and the number of outgoing edges. The *destination* scratchpad temporally stores the vertex entry that is being computed, which will be copied to the *source* scratchpad at the end of the current iteration. When the entire set of vertices does not fit in the collective source scratchpads, graph slicing is employed [54]. The third scratchpad is an *active-list scratchpad*, and it tracks the subset of vertices that are active in each iteration [54]. Two registers support the management of the *active-list* by storing the address of the starting location and the length of the list.

Finally, our baseline architecture includes an *edge-prefetcher* as in [23] to stream the *edgeArray* from the local vault's memory. The prefetcher consults the *active-list* to identify

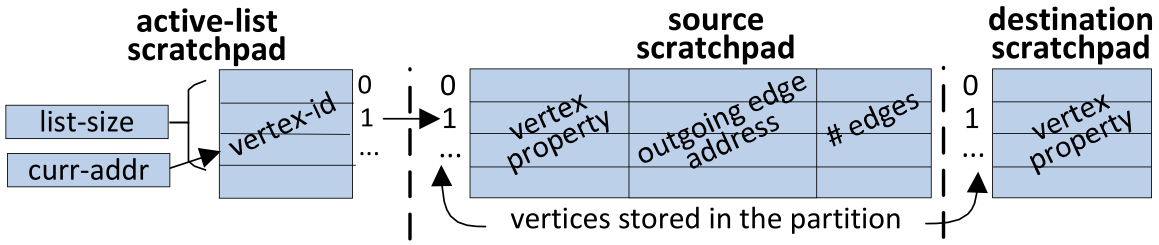


Figure 4.3: Scratchpad memory architecture in the baseline design. The *source* scratchpad stores all vertices within a partition for the current algorithm's iteration. Each entry holds vertex property, starting memory address, and number of outgoing edges. The *destination* scratchpad is temporary storage for updated vertex information. The *active-list* scratchpad keeps track of the active vertices for the current iteration.

which edges to prefetch, or simply streams the complete *edgeArray* for an algorithm that must process all vertices in each iteration.

4.3.2 Distributed message coalescing

One of MessageFusion's key contributions is enabling a distributed approach to vertex updates. While the general assumption by developers is that updates take place atomically, it is indeed possible in many situations to decompose the update operation into several incremental updates, without affecting final outcomes. We note that this decomposition is possible whenever the update operation satisfies both commutative and associative properties, a scenario common to several graph-based algorithms, including, for instance, *PageRank* (update is an addition operation, as illustrated in Figure 2.12), or *Single-Source-Shortest-Path* (update is a 'min' operation). MessageFusion coalesces vertex-update messages when they are emitted from the source vault, and again while in transit in the interconnect.

Coalescing at the source vault: the *reduce* module deployed in each vault is responsible for carrying out the reduce operation at each vault. In addition to coalescing updates for vertices stored in the local vault – a function common to most state-of-the-art solutions – our *reduce* units also strive to coalesce multiple messages generated within the vault and destined to the same remote vertex. Messages that could not be coalesced are directly

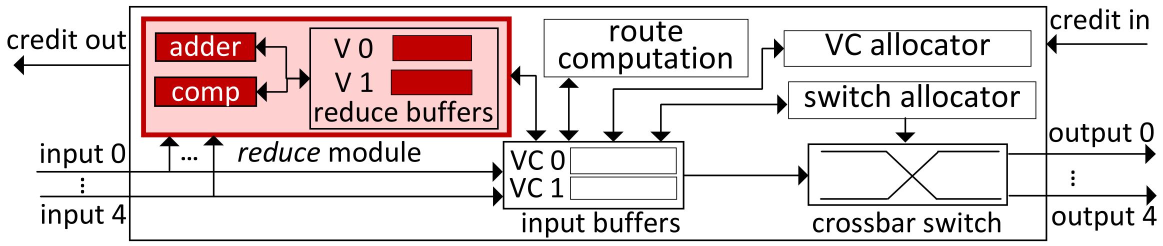


Figure 4.4: Router architecture showing the *reduce* module, which taps into all the router's input ports looking for opportunities to aggregate messages among those stored at input or *reduce* buffers.

injected into the interconnect.

Coalescing in the network: once a message is injected into the interconnect, there is an additional opportunity for coalescing with messages from other vaults while traversing the interconnect. Figure 4.4 presents a typical router augmented with our *reduce* unit to perform such coalescing. The *reduce* unit contains a comparator to match messages to the same destination vertex, compute logic for the supported reduce operations (*e.g.*, adder, comparator, *etc.*) that are selected by configuring each message with the application's reduce operation type before the message is injected to the interconnect. In addition, the *reduce* unit incorporates a few *reduce* buffers, whose role is to extend the time window available for finding coalescing opportunities. The *reduce* unit continuously scans all of the router's input ports and messages stored in the *reduce* buffers, seeking to merge messages to the same destination vertex. Note that this operation is executed in parallel with other functions of the routers; hence, it incurs a negligible penalty in latency. Also, vertex update messages are single-flit; thus, they are transferred through the router in a single operation, without requiring reassembly. If an input message is not matched immediately upon arrival, it is stored in one of the *reduce* buffers, replacing the most stale entry, which, in turn, is sent along to the destination node. Upon completion of an iteration, the HMC vaults transmit a buffer-flush message used to clear all *reduce* buffers so that all messages reach their destination. Figure 4.5 presents a simple example of the execution flow just described: two messages for vertex V4 are merged in cycle 2, and again with one more merged-message

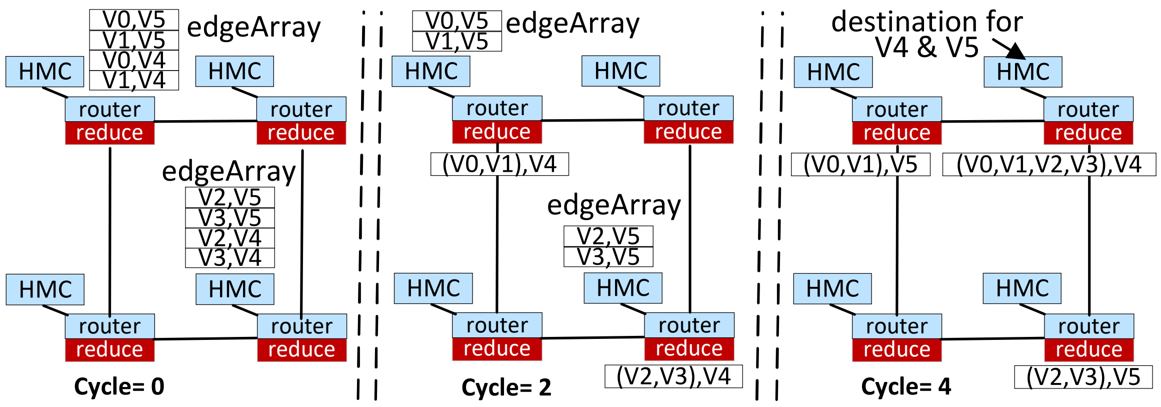


Figure 4.5: Example of MessageFusion's execution illustrating snapshots of messages being coalesced at cycle 2 and again at cycle 4.

in cycle 4.

4.3.3 Maximizing coalescing opportunities

To support and enhance coalescing opportunities, MessageFusion strives to organize the work at each vault so that vertex-update messages to same vertex are generated in close sequence. When successful in this effort, the messages can be coalesced at the source and may find additional opportunities as they traverse the interconnect, as other vaults maybe working on updates to the same vertices. To this end, we explored two types of edge reordering techniques. The first is an offline reordering that must be completed only once for each dataset and simplifies per-destination message generation. The second is an online reordering solution, which is particularly valuable for algorithms that update only a portion of the vertices during each iteration of execution. For the second solution, we devised a simple online hardware-based implementation.

Offline edge reordering. This process is very simple in that it reorganizes the *edgeArray* to be sorted by destination vertex, rather than by the more common source vertex ordering. An example is shown in the *edgeArray*s of Figure 4.5, cycle 0. With this ordering, vaults process all updates to a same destination vertex first, then move forward to the next, thus creating coalescing opportunities.

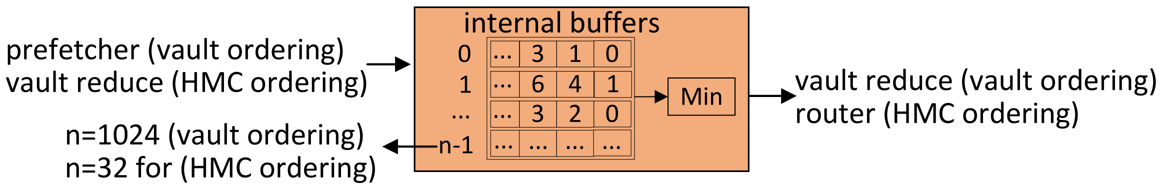


Figure 4.6: *Ordering* hardware modules. Vault- and HMC-level *ordering* modules can order messages within a pool as large as the number of buffers they maintain. During each cycle, the module selects the message with the minimum destination vertex ID among those stored in its buffers, and forwards it to a downstream *reduce* module for potential reduction.

Runtime edge reordering. For applications that require processing only a subset of vertices during each iteration of the algorithm's execution (thus, they maintain an *active-list*), the simple solution described above is not effective, as accesses to the *edgeArray* would not benefit from any spatial locality. Since only some of the source vertices are considered in each iteration, the prefetcher unit becomes ineffective (see also Section 4.3.1). To address this challenge, we leverage hardware *ordering* units, which are deployed in each vault and each HMC. *Ordering* units within a vault receive a stream of vertex-update messages and forward them to the *reduce* units, after sorting them by increasing destination vertex. Note that this sorting is only partial, as the set of incoming messages keep changing during the runtime ordering. In addition, we also deploy *ordering* units at each HMC: these units sort the vertex-update messages arriving from multiple vaults within the same HMC, and output them to the local port of the associated router. Note that HMC-level *ordering* units enhance MessageFusion's coalescing capability for all applications. The *ordering* module, shown in Figure 4.6, comprises internal buffers storing update messages generated during edge-processing. The larger the number of buffers within the unit, the better its sorting capabilities. In our implementation, we used 32 entries for HMC *ordering* units, and 1,024 for vault *ordering* units. These units include a 'min' function that extracts the message with the lowest vertex ID among those in the buffers, and sends it to the *reduce* unit within the vault (vault *ordering* modules) or to the router's local port (HMC *ordering* modules).

4.3.4 Selective power-gating

We employ power-gating selectively, based on algorithm and dataset, to limit the energy impact of MessageFusion's *reduce* and *ordering* hardware modules. Specifically, vault *ordering* modules are only useful for algorithms that maintain an *active-list* (*e.g., PageRank*); thus, they are power-gated when running all other algorithms.

Moreover, the input dataset's size drives scratchpad's utilization. The scratchpad size in our experimental setup (see Section 4.4) matches the storage size of several recent works [54, 41]. These scratchpads have low utilization when running most datasets. Indeed, utilization is less than 50% for over 85% of real-world datasets from [67]. In other words, for many datasets, a large portion of the scratchpad can be turned off using the Gated-V_{dd} technique [94]. We leverage the number of vertices of the dataset to estimate the size of the scratchpad portion that can be turned off. The technique entails negligible impact in storage access time and area of the scratchpads [94].

4.4 Experimental Setup

In this section, we present the experimental setup that we deployed to evaluate the MessageFusion solution.

We modeled the optimized Tesseract baseline using an in-house cycle-accurate simulator. To model the HMC, we used CasHMC [58], a cycle accurate HMC simulator, set up with 16 8GB HMCs interconnected with a scalable network architecture modeled in BookSim [59], as discussed in [63]. The interconnect has 128-bit links and 5-stage pipelined routers with 3 virtual channels. We considered both dragonfly and 2D mesh network topologies. The dragonfly topology is configured with 255 input buffers, whereas the 2D mesh topology is set with 8 input buffers. Four high-speed SerDes links connect each HMC to the interconnect. Each HMC comprises 8 DRAM dies stacked on top of each other and partitioned into 32 vaults, with each vault providing 16 GB/s of internal memory bandwidth, for a total bandwidth of 8 TB/s available to the processing elements (PE). Each PE is equipped with a 1KB prefetcher to stream edge data. In addition, *source* scratchpads are 128KB, *destination* scratchpads are 64KB, and *active-list* storage is 32KB. To model MessageFusion, we then augmented each router and vault with a *reduce* module. *Reduce* modules within each vault coalesce messages locally, and thus have a buffer size of 1. *Reduce* modules in the routers have a buffer size equal to the total number of HMCs. The *ordering* module in each vault performs 1,024 comparisons per cycle, while the *ordering* module in each HMC provides 32 comparisons.

Area, power, and energy modeling. We modeled the scratchpads with Cacti [101] and the interconnect with ORION 3.0 [62], assuming a 45nm technology node. We also synthesized MessageFusion's logic (PE and *ordering* logic) in IBM 45nm SOI technology at 1GHz. We refer to prior works for estimation of the logic layer, including the SerDes links, and the DRAM layers of the HMC [96, 23]. The SerDes links consume the same power whether in idle or active states [96]. To quantify overall energy consumption, we considered both static power and per-access energy for each hardware module, which we combined with execution time and module's utilization, obtained from the cycle-accurate simulator.

Graph workloads characteristics. Once again, we considered for our evaluation algorithms from the pool discussed in Section 2.3. From this pool, we selected five representative algorithms: *PageRank*, *BFS*, *CC*, *SSSP*, and *SpMV*. We provide their key characteristics in Table 2.1. We evaluated these algorithms on seven real-world datasets that we selected from a pool of datasets described in Section 2.4. The datasets are *wv*, *sd2*, *pk*, *lj2*, *orkut2*, *rCA*, and *USA*. These datasets provide a broad representation of graph size and degree (edge-to-vertex ratio). In Table 2.2, we report their size in vertices and edges, their degree, and the source reference for the dataset.

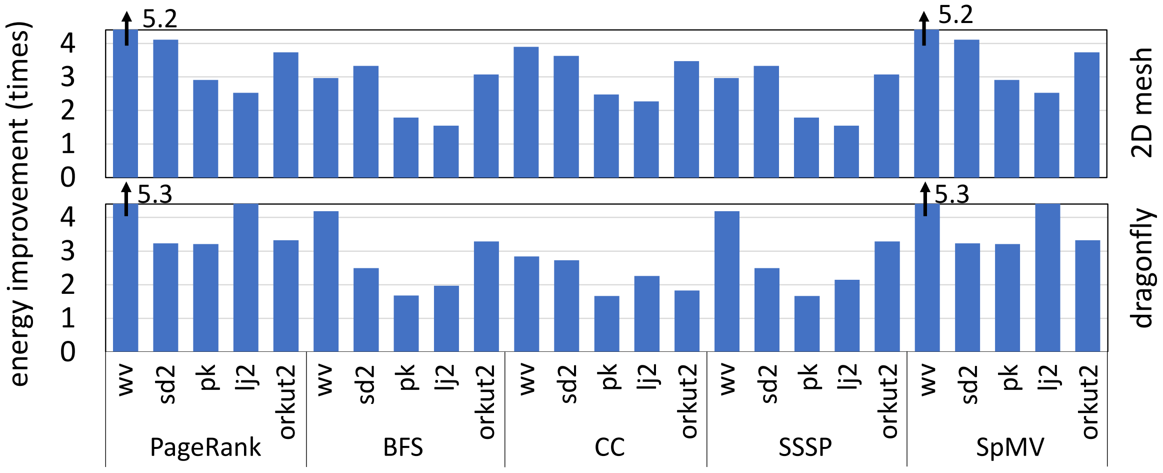


Figure 4.7: Energy savings enabled by MessageFusion over Tesseract on 2D mesh (3.1× on average) and on dragonfly (2.9× on average).

4.5 Energy Evaluation

In this section, we present overall energy benefit and the insights gained concerning message coalescing rates and sources of the energy savings.

Figure 4.7 presents the overall energy benefit of MessageFusion compared to Tesseract, on 2D mesh and dragonfly topologies. As shown, MessageFusion achieves a 3× energy reduction on average for both topologies. MessageFusion's main source of energy saving lies in its ability to coalesce messages within each vault, as well as in the routers. This property minimizes inter-cube communication bottlenecks typical of Tesseract, reducing execution time and network utilization, and consequently reducing overall energy consumption. *PageRank* and *SpMV* enable relatively better energy benefits because they process all vertices in each iteration. Hence, an offline reordering algorithm is sufficient to order edges so as to maximize coalescing opportunities, and its energy cost is amortized over all runs of the same dataset [119]. *BFS*, *SSSP*, and *CC* attain lower energy savings because they maintain an *active-list* to process a different subset of vertices at each iteration. As a result, our limited-size *ordering* module (the module can order at most 1,024 at a time) is relatively ineffective. All of the above algorithms are executed on the baseline solution with the *edgeArray* in source-order. Executing the algorithms on the baseline with the graphs

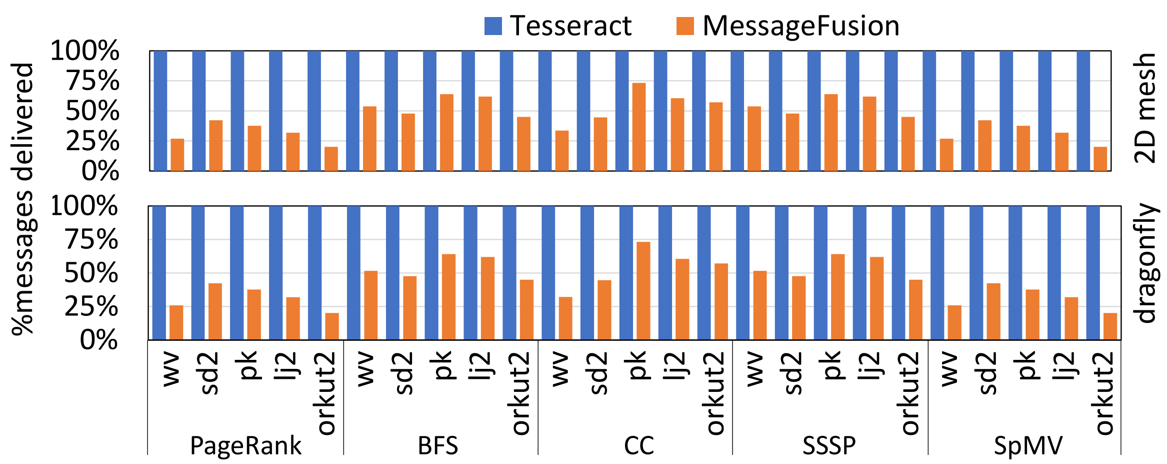


Figure 4.8: Number of messages delivered with 2D mesh and dragonfly, showing a reduction by 2.5× over the baseline.

reordered in destination-order consumes higher energy (by a factor of 1.2×). This result is because destination-order vertex update messages exacerbate network contention in the baseline. MessageFusion's *reduce* modules alone alleviate this network contention, and manage to achieve a 1.2× energy reduction.

We note that energy savings are generally dataset-size-independent, but highly sensitive to the graph's degree. Indeed, we analyzed the energy saving profile of road-network graphs (*USA* and *rCA*), which have notoriously low degrees: MessageFusion achieves only 1.3× energy savings for these datasets. This limited benefit is due to the fewer coalescing opportunities that these graphs present.

Message reduction. To gain insights on MessageFusion's effectiveness, Figure 4.8 reports the number of messages crossing each router in both MessageFusion and Tesseract, showing that MessageFusion enables a 45% reduction. Further analysis reveals that almost 50% of the messages to remote HMCs are coalesced at the source vault, while 30% of coalescing takes place in the network. MessageFusion's ability to reduce interconnect traffic is the lead factor in its energy savings.

Energy savings analysis. Figure 4.9 presents a breakdown of the sources of energy savings in MessageFusion. A primary source is the shorter execution time due to message coalescing, which alone contributes a factor of 2×, on average. In addition, power-gating unused

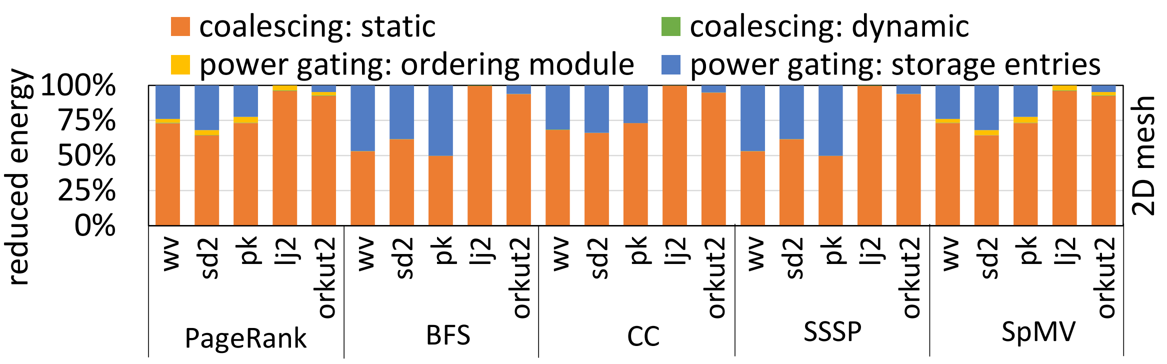


Figure 4.9: Breakdown of energy savings: for most graphs, the key contributor is static energy, due to MessageFusion's performance speedup.

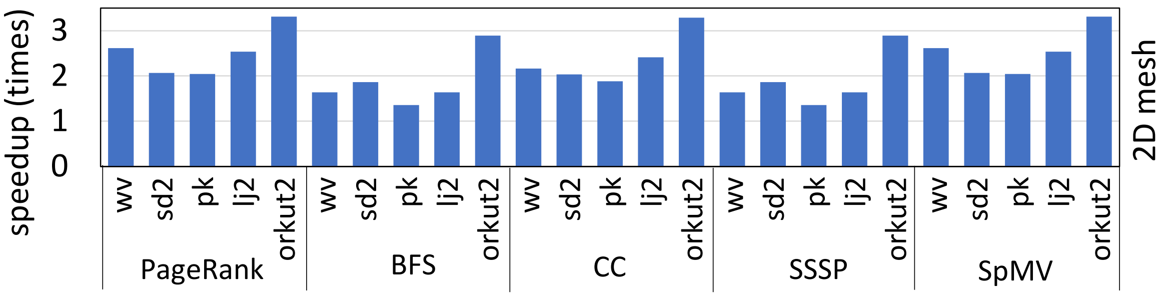


Figure 4.10: Performance improvement over the Tesseract baseline is 2.1x on average.

storage entries is significantly beneficial for small datasets, which underutilize available storage. Power-gating the vault *ordering* modules provides some benefit for algorithms with no *active-list*. Finally, the benefit of coalescing on network utilization is negligible. A major source of the remaining energy costs, not shown in the plot, is the use of the SerDes links (60% of the remaining energy spent in execution). Since MesssageFusion reduces the activity through this component significantly, future works may explore a selective disabling of these links [96].

4.6 Performance

Figure 4.10 presents the speedup achieved by MessageFusion over Tesseract. MessageFusion provides a 2.1× speedup on average, primarily because it reduces inter-cube traffic by coalescing messages within each vault, as well as in the network.

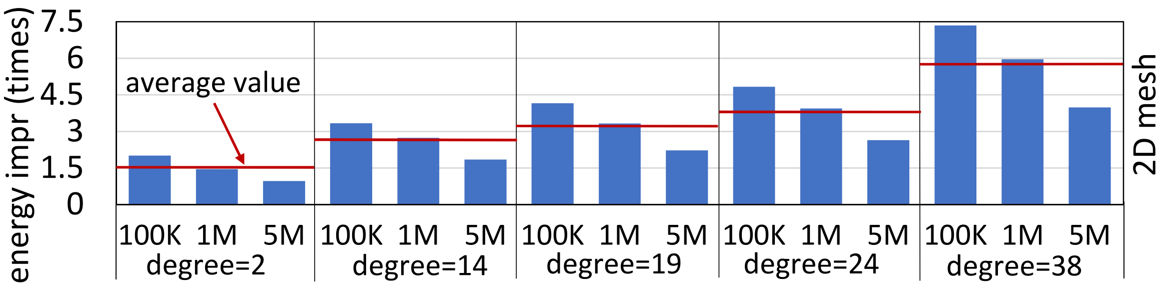


Figure 4.11: Sensitivity to input graph size and degree showing that energy savings improve with higher degrees, but degrade with larger graph sizes. The analysis is carried out for *PageRank*.

4.7 Sensitivity Studies

This section provides sensitivities studies related to input graph size and degree, router buffer size, and number of HMC cubes.

Input graph size and degree. To evaluate in depth the impact of graphs' size and degree on MessageFusion, we generated synthetic graphs using a tool from [67]. The number of vertices of the graphs generated ranges from 100K to 5M while their degree spans from 2 to 38. The number of edges for those graphs is the number of vertices times the degree. Figure 4.11 reports our findings. As shown in the figure, MessageFusion's energy benefit increases with a graph's degree, but degrades with large graph sizes.

Router resources. While a single-entry *reduce* buffer suffices at each vault, since messages are ordered by destination-vertex, router buffers require more entries as messages come from different cubes. To this end, we evaluated a range of buffer sizes for a router's *reduce* module: 8, 16, and 32 entries. 16-entry was an optimal choice over the evaluated algorithms and datasets, providing a 1.2× improvement over the 8-entry solution, and on par with the 32-entry one. Consequently, we used this value in all other experiments. With a 16-entry *reduce* buffer, the average opportunity window for coalescing is 175 clock cycles. Note that this latency does not affect overall performance, as the workloads are not latency sensitive. In addition, we evaluated the effect of re-purposing the *reduce* buffers for Tesseract's input buffers, and found only a negligible benefit. We also considered increasing the number of virtual channels (3, 6, 12) at the routers for both Tesseract and MessageFusion: the relative energy benefit of MessageFusion over Tesseract holds at 3$\times$ across these different virtual-channel configurations.

Number of cubes. Tesseract is shown to be highly scalable with the number of cubes [23]. Our evaluation indicates that MessageFusion maintains this scalability. As a data point, when quadrupling the number of cubes, MessageFusion's benefit slightly improves from 3$\times$ to 3.7$\times$, on average.

4.8 Area, Power, and Thermal Analysis

This section presents area, power, and thermal analysis of MessageFusion, as compared to the baseline Tesseract design.

Assuming that the logic die in the HMC architecture occupies the same area as the DRAM die (226 mm^2 as reported in [23]), the baseline Tesseract design occupies 10% of the logic die, while MessageFusion takes an additional 3.1%, for a total of 13.1% of the available area. MessageFusion increases the power consumption by a maximum of 2.5%, leading to a power density of 124 mW/mm^2, which is still under the thermal constraint (133 mW/mm^2) reported in [23].

4.9 Bibliography

In this section, we provide a summary of related works. We first present specialized architectures that target graph analytics. Then we describe solutions based on CPUs and other vector processing architectures. Finally, we provide a discussion on message coalescing based solutions.

Graph analytics architectures. Recently, several graph-specific architectures have been proposed. Tesseract [23] relies on 3D-stacked memory to benefit from its high bandwidth, and GraphP [118] proposes a graph partitioning strategy to improve the inter-cube communication of Tesseract, while trading its vertex programming model as well as incurring additional memory overhead. OMEGA [18] proposes a memory subsystem optimization. In Table 4.1, we present a comparison of MessageFusion with these architectures. MessageFusion improves over these architectures primarily because it coalesces messages while they are in transit. It performs coalescing locally at each vault as well as in the network's routers. Its ability to reduce the number of messages flowing through the network is the key reason for its significant improvement over prior works. While doing so, it still maintains the widely accepted vertex-centric model and is independent of partitioning strategies.

Table 4.1: Comparison of several graph analytics architectures.

	Tesseract[23]	**GraphP[118]**	**OMEGA[18]**	**MessageFusion**
`programming`	vtx-centric	2-phase upd.	vtx-centric	vtx-centric
`update location`	remote	local	remote	on-path
`partitioning`	flexible	source-cut	flexible	flexible
`preprocessing`	none	source-cut	reorder	slice & reorder
`communication`	word-size	source-cut	word-size	coalesce
`memory type`	PIM	PIM	DDR	PIM

Graph analytics on CPUs and Vector platforms. Several works have been proposed to reduce the energy of graph analytics on CPUs [119] and vector platforms [92]. However, these works do not take advantage of the high internal bandwidth provided by 3D-stacked memories.

On-path message coalescing. On-path message coalescing has been explored in single-node systems to reduce traffic due to cache coherence [20], atomic-operations [44], and key-value data in MapReduce applications [17]. However, extending these solutions to graph analytics is nontrivial.

4.10 Summary

This chapter presented MessageFusion, a specialized hardware architecture for graph analytics based on a novel mechanism to coalesce vertex update messages as they transit to their destination node, hence mitigating the interconnect bandwidth bottleneck of recent domain-specific architectures. MessageFusion maximizes message-coalescing opportunities by relying on a novel runtime graph-reordering mechanism specific to this goal. As a result, MessageFusion reduces the number of messages traversing the interconnect by a factor of 2.5 on average, achieving a 3$\times$ reduction in energy cost, and a 2.1$\times$ boost in performance, over a highly-optimized processing-in-memory solution.

MessageFusion provides a solution for our third strategy, discussed in Section 1.4, as it coalesces graph analytics data messages to the same vertex in the network. MessageFusion is effective for processing of frequently-accessed vertices. It reduces interconnect traffic, and thus improving the overall performance and energy benefits.

Both MessageFusion and our OMEGA solution discussed in Chapter 3 address the inefficiency of the execution of graph analytics by optimizing the processing of frequently-accessed data. The next chapter complements these solutions by optimizing the processing of infrequently-accessed data.

CHAPTER 5

Hybrid Processing in On/Off-chip Memory Architecture for Graph Analytics

This dissertation has thus far focused on boosting the execution of graph analytics by optimizing the processing of frequently-accessed data. Chapter 3 presented the OMEGA solution, which optimizes the processing of frequently-accessed data, by adopting our strategy of processing near storage, presented in Chapter 1. Whereas Chapter 4 provided an introduction to MessageFusion, which achieves the same goal by embracing the strategy of coalescing messages in network. Although these solutions optimize the processing of frequently-accessed data, they rely on conventional mechanisms to process infrequently-accessed data. Once the processing of frequently-accessed data was optimized, we noted that the processing of the infrequently-accessed data had become the primary cause of inefficiencies.

In this chapter, we strive to holistically optimize the processing of both frequently-accessed data as well as infrequently-accessed data. To optimize the former, we employ the OMEGA solution of Chapter 3. This technique provides specialized on-chip storage units to handle frequently-accessed data, and co-locates specialized compute engines with those storage units for processing the data in-situ. As discussed in Chapter 3, the technique reduces the traffic between processor and off-chip memory, as well as between processor cores and on-chip storage units. To optimize the processing of infrequently-accessed data, we pursue our third strategy: processing data near storage in traditional storage units. Specifically, we co-locate compute engines with off-chip memory units for processing infrequently-accessed data in-situ. This approach reduces the traffic due to the processing of infrequently-accessed data, which would have normally taken place between processor cores and off-chip memory, thus minimizing the associated performance and energy inefficiencies.

The solution discussed in this chapter equips both on-chip and off-chip memory with specialized compute engines, which accelerate the computation over all vertices in a graph, after negotiating where each vertex should be stored. For power-law graphs [18] with imbalanced connectivity among their vertices, the solution maps the high-degree vertices, which are accessed most frequently, to dedicated on-chip memory: their updates are computed by co-located lightweight compute-engines. In contrast, low-degree vertices remain in off-chip memory: their updates are processed by compute units connected to the off-chip memory blocks. This approach enables high-degree vertices to experience efficient on-chip access and all vertices to benefit from efficient execution of the related operations.

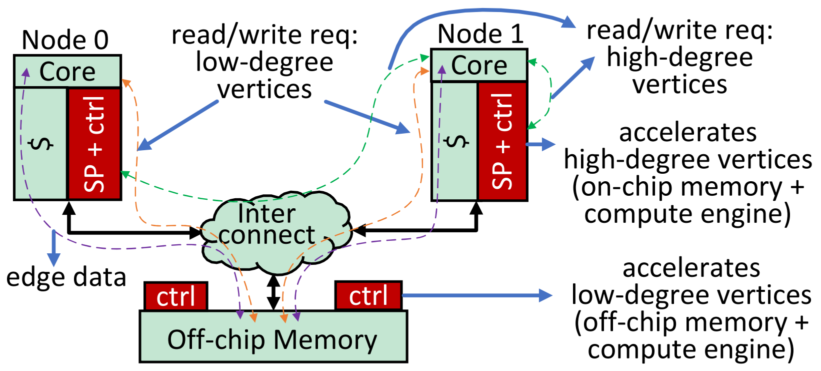
5.1 Motivational Study

As discussed in Chapter 2, accesses to vertices' property data (the *next_rank* data structure) in the *PageRank* algorithm often exhibit poor locality during the *update* phase, and those updates are typically carried out atomically in multicore architectures, incurring high performance overheads. Many recent studies strive to optimize atomic operations: GraphPIM [81] by offloading all atomic operations to off-chip memory, OMEGA [18] by executing atomic operations related to high-degree vertices in dedicated on-chip memory. Note that neither approach holistically optimizes the execution of atomic operations across a wide range of vertices' degrees or graph characteristics (*e.g., from power law to uniform graphs*).

To estimate the potential benefit of processing data both in on- and off-chip memory, we carried out an experiment using the gem5 simulation infrastructure [34]. In this experiment, we modeled the OMEGA solution, which processes data in on-chip memory, and another recent proposal, called GraphPIM [81], which computes on data in off-chip memory. We modeled both OMEGA and GraphPIM in gem5 and ran the *PageRank* algorithm with several input graphs of varying average degrees. Our findings indicate that OMEGA executes up to 26% of the atomic operations on general-purpose cores when processing power-law graphs, and up to 80% when processing relatively uniform graphs. Unfortunately, the execution of atomic operations on general-purpose cores incurs a high-performance cost, primarily due to the suspension of the cores' pipelines during the operation's execution [81, 18]. GraphPIM attempts to overcome this cost by executing all atomic operations in off-chip memory, but it also generates high traffic between compute engines and their local memory partitions: up to 6$\times$ the traffic generated by a plain chip multi-processor (CMP) solution. Indeed, the execution of each atomic operation in GraphPIM entails two memory requests (a read and a write) from the compute engines. The memory traffic could be potentially reduced by processing atomic operations both in on-chip and off-chip memory.

5.2 Centaur Overview

Figure 5.1 illustrates the solution discussed in this chapter, called Centaur. Centaur enables hybrid processing in a dedicated on-chip (SPs) / off-chip memory architecture. It stores frequently-accessed vertex data in on-chip memory and infrequently-accessed data in off-chip memory. Lightweight compute engines, attached to each memory unit, execute the related operations in-situ. Centaur's approach enables high-degree vertices to experience efficient on-chip access, and all vertices to benefit from efficient execution of the related atomic operations. Note that both the general-purpose cores and the compute-engines access vertex data. To simplify the implementation of cache coherence, Centaur employs a cache-bypassing technique for vertex-data requests from the cores. Centaur retrieves edge data via caches, as those accesses exhibit high cache-locality.



Node 0
Core
$
SP + ctrl
read/write req: low-degree vertices
Node 1
Core
$
SP + ctrl
read/write req: high-degree vertices
accelerates high-degree vertices (on-chip memory + compute engine)
Inter connect
edge data
ctrl
ctrl
Off-chip Memory
accelerates low-degree vertices (off-chip memory + compute engine)

Figure 5.1: Centaur overview. Centaur enables hybrid processing in a specialized on-chip (SPs) / off-chip memory architecture. It stores frequently-accessed vertex data in on-chip memory and infrequently-accessed data in off-chip memory. Lightweight compute engines, attached to each memory unit, execute the related operations in-situ.

5.3 Centaur Architecture

Figure 5.2 presents the architecture of Centaur, a hybrid solution that comprises processing both in on- and off-chip memory. As shown, data related to high-degree vertices is stored in dedicated on-chip memory, whereas data related to low-degree vertices is stored in off-chip memory. Atomic Compute Units (ACUs) are co-located with both types of memory units to perform related atomic operations. The on-chip memory is modeled as a specialized scratchpad (SP) architecture. Off-chip memory is modeled as a Hybrid Memory Cube (HMC), a 3D-stacked memory solution that has been shown to provide higher bandwidth to memory compared to conventional solutions, such as DDR [93]. The HMC includes 32 DRAM partitions (vaults), which are connected to the processor cores via four high-speed SerDes links. The ACUs at both scratchpads and vaults are specialized hardware units that execute the atomic operations of a wide range of graph algorithms, similar to those detailed in the HMC 2.0 specification [81]. The on-chip Vertex Management Units (O_nVMUs) filter requests to vertex data, manage their execution, and forward the results to their destination. The destination O_nVMUs control the execution of the requests on the O_nACUs and/or the scratchpads. In addition, the O_nVMUs update the *activeList* on behalf of their local cores, based on the results of atomic operations obtained from the ACUs. *ac-*

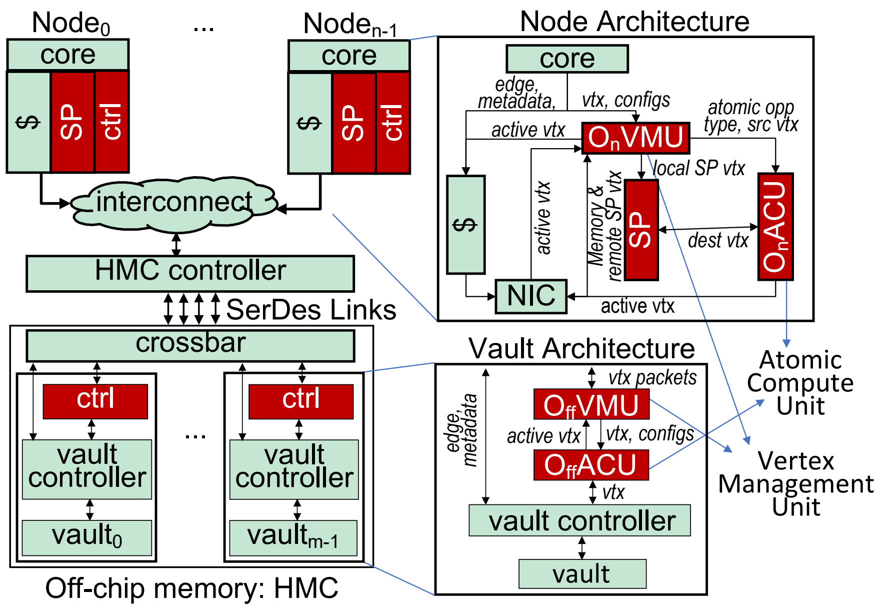


Figure 5.2: Centaur architecture. Centaur employs Atomic Compute Units (ACUs) capable of executing atomic operations at both dedicated on-chip memory units (SPs) and off-chip memory partitions (vaults). The Vertex Management Units (VMUs) attached to the SPs filter requests to vertex data and send them to their respective destinations. They also control the execution of requests to high-degree vertices, and update the *activeList* on behalf of their respective cores. The VMUs attached to the vaults manage the execution of requests to the low-degree vertices. The cores access edge and metadata via the conventional caches.

tiveList refers to a list of vertices that will be active, and thus processed in the next iteration of an algorithm [18, 102]. The off-chip Vertex Management Units (O_{ff}VMUs) manage the execution of requests related to low-degree vertices. Additionally, similar to ACUs, the cores issue read/write requests to the vertex data, *e.g.,* to initialize the vertex data and to access source-vertex data for generating atomic commands to ACUs. All of these requests bypass the conventional cache hierarchies to simplify coherence management between the cores and the ACUs; this is accomplished by leveraging a cache bypassing technique common in commercial processors [78]. This approach also reduces cache pollution, access latency, and energy costs. However, edge data and metadata are still delivered via the conventional cache hierarchy, as they are accessed only by the cores and maintain high locality.

Dedicated On-chip Memory. To minimize the traffic between cores and off-chip mem-

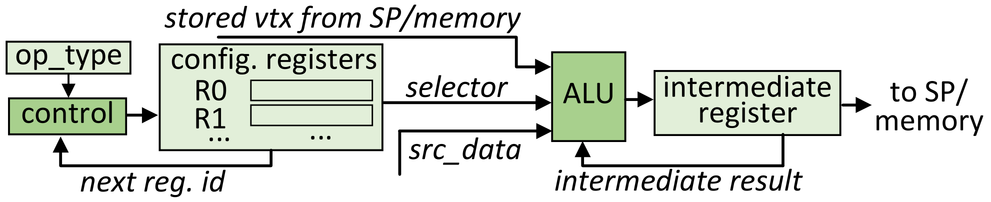


Figure 5.3: The Atomic Compute Unit (ACU) executes the atomic operations entailed by the graph algorithm. They are deployed in both on- and off-chip memory units.

ory, and thus reduce the associated bandwidth, latency, and energy costs, Centaur utilizes dedicated on-chip memory units to store high-degree vertices, partitioning them across the multiple units. The on-chip memory is modeled as a scratchpad (SP), organized as a direct-mapped memory, storing data related to high-degree vertices. Each entry in the scratchpad corresponds to the data that the algorithm stores per-vertex. In most graph algorithms, this per-vertex data ranges from 4 to 12 bytes (see Table 2.1). For instance, *PageRank* stores 8 bytes of rank values for each vertex.

Atomic Compute Unit (ACU). To execute the atomic operations entailed by graph algorithms in-situ, thus eliminating the computation overhead of atomic operations from the general-purpose cores, Centaur co-locates ACUs with both scratchpads and off-chip memory. Figure 5.3 provides the architecture of the ACU. Each ACU uses *configuration registers* to store the set of micro-operations that implement the required atomic operation. The *control* logic takes the atomic operation type (*op_type*) as input to index the *configuration registers*, and then executes the corresponding micro-operation. Other inputs to the ACUs include source-vertex data (*src_data*) from the cores and stored data from either scratchpads or off-chip memory, depending on where the ACU is deployed. Centaur configures the *configuration registers* during the application's initialization. New algorithms requiring new types of atomic operations can also be supported by generating additional micro-operation sequences.

On-chip Vertex Management Unit (O$_n$VMU). O$_n$VMUs share the cache data-port of their local cores to filter requests related to vertex-data away from the cores. During the initialization phase of the graph algorithm, the O$_n$VMU filters configuration requests, such as

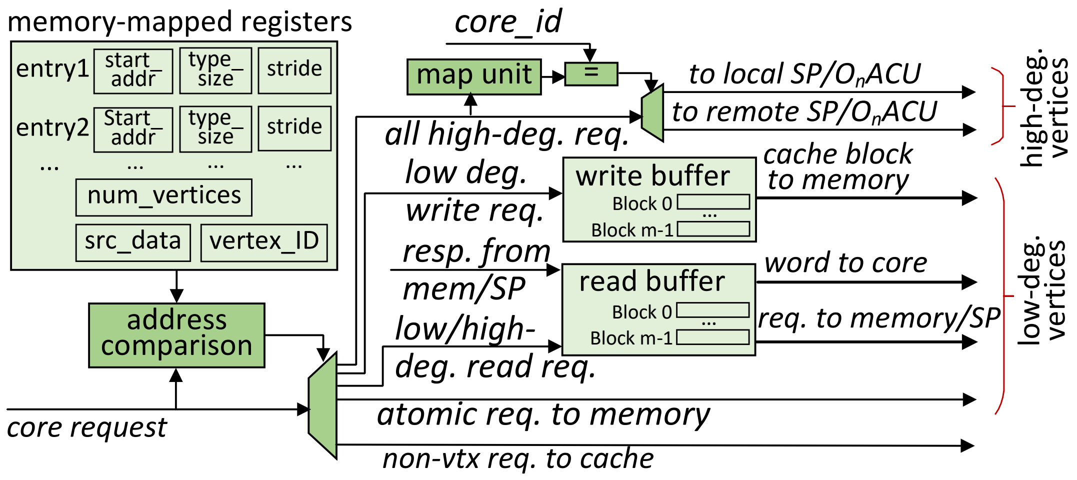


Figure 5.4: The On-chip Vertex Management Unit (O$_n$VMU) routes vertex-data requests from the local core to the correct storage unit and monitors their computation.

atomic operation type (*op_type*), and base address and range of vertex data locations (depicted in the left part of Figure 5.4). Once the execution of the graph algorithm begins, the O$_n$VMU determines if further requests are to vertex data or other data. If a request is not to vertex data, the O$_n$VMU forwards it to the cache; otherwise, it must determine whether the requested vertex is stored in a scratchpad or in off-chip memory. Simple, non atomic, read/write requests on low-degree vertices are processed through specialized *read/write buffers* within the O$_n$VMU (shown in the right part of Figure 5.4). These *read/write buffers* are useful when requests to vertex data exhibit sequential patterns: they operate as read-combining or write-combining buffers, which are already common in contemporary processors [1]. As an example, *PageRank* performs sequential read/write operations on the *next_rank* vector during the *apply* phase. Alternatively, if a request is to a high-degree vertex, the O$_n$VMU sends it forward to the relevant destination scratchpad, handling it at word granularity. Read requests are sent through the *read buffer*, before forwarding to a scratchpad, to take advantage of temporal locality in the vertex data (common when cores access source-vertex data to generate atomic commands).

Requests related to atomic operations, *e.g.*, a write request to the *src_data* memory-mapped register (discussed in the last paragraph of this section), are handled differently:

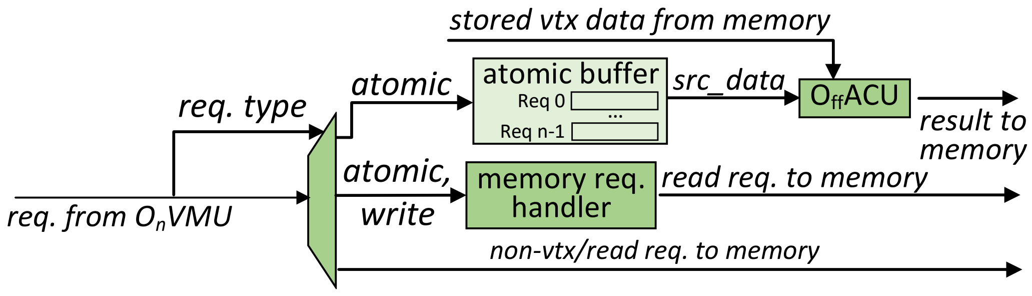


Figure 5.5: The Off-chip Vertex Management Unit (O_{ff}VMU) manages requests to low-degree vertices in off-chip memory.

requests related to low-degree vertices are forwarded to off-chip memory, while those related to high-degree vertices are sent to the destination scratchpad. The destination O_nVMU holds the incoming request in a buffer and generates a read request to its associated scratchpad to retrieve the requested value. Upon receiving the value, the destination O_nVMU initiates execution of the atomic operation on the associated O_nACU. Then, the O_nVMU writes the result back to its scratchpad and, if the operation generates a new active vertex, it sends also a command with active-vertex information to the originating O_nVMU. Upon receiving the command, the originating O_nVMU updates its *activeList* in the cache. Figure 5.4 shows this selection process and compute flow.

Off-chip Vertex Management Unit (O_{ff}VMU). Low-degree vertex accesses filtered by the O_nVMUs are forwarded to the O_{ff}VMUs via the memory controller. The memory controller inspects the request address and sends it to the corresponding memory partition (vault). The O_{ff}VMU connected to this vault is thus tasked with processing the request, as shown in Figure 5.5. First, this O_{ff}VMU determines whether the request is atomic or a simple read/write request to vertex data. For atomic requests, the O_{ff}VMU generates a read request to memory, while queuing the request in the *atomic buffer*. Upon receiving a response from memory, the O_{ff}VMU instructs the corresponding O_{ff}ACU to dequeue the request and execute the atomic operation. Once the O_{ff}ACU completes its task, the O_{ff}VMU writes the result back into memory. If the atomic operation generates an active vertex, the O_{ff}VMU forwards a command to the originating O_nVMU to update its corresponding *activeList* in the cache. For non-atomic requests, the O_{ff}VMU reads the corresponding cache block from memory and sends the block to the originating O_{n}VMU (read operations), or it reads, updates, and writes-back the block to memory (write operations).

Vertex data partitioning between on/off-chip memory. For power-law graphs that do not fit in the on-chip memory, Centaur must identify high-degree vertices to maximize the utilization of on-chip memory, and thus provide higher performance benefit. To do so, either of the following two approaches can be adopted. The first is a hardware-based vertex replacement policy that maintains the frequency of atomic operations computed on each vertex. In this approach, Centaur maintains frequency and collision bits for each vertex stored in the on-chip memory, while either increasing the associated frequency value (if an atomic operation is computed on the vertex successfully), or otherwise increasing the corresponding collision value. For each new request, if the stored frequency value is greater than that of the collision value, the stored value will be replaced by the new one and sent to off-chip memory. Otherwise, the new value will be the one forwarded to off-chip memory. This approach is similar to that employed in [17], its main drawback is that it requires maintaining extra bits to implement the replacement policy.

The second solution is a software-based graph preprocessing approach that reorders vertices based on their in-degree. Once vertices are reordered, the high degree vertices can be identified by verifying if the ID of a new vertex request is smaller than the maximum number of vertices that can be mapped to the on-chip memory unit, assuming that the highest-degree vertex has an ID of 0. Although this approach can identify the optimal set of high-degree vertices at no extra hardware cost, it entails a preprocessing overhead. Such overhead could be alleviated by reordering only vertices to be mapped to on-chip memory. In power-law graphs, approximately 20% of the vertices account for 80% of the connectivity; thus, sorting only 20% of the vertices would allow the computation of a significant fraction of atomic operations in on-chip memory. Furthermore, the cost of this reordering algorithm is amortized over the execution of various graph algorithms, repetitive

```
for V in vertices
    for edge(V,U) in outGoingEdges[V]
        *vertex_ID = U;
        *src_data = curr_rank[V]/outDegree[V];
```

Figure 5.6: Pseudo-code for a Centaur-ready version of PageRank's update phase. The new rank value of the source vertex and the destination vertex ID are written to memory-mapped registers.

execution of the same algorithm, or even the many iterations of a single execution of the algorithm on the same dataset, as suggested by [18, 119]. We employed this latter approach in this work and deferred the former to future work.

5.4 System Integration

Cache coherence, address translation, and context switching. To simplify the implementation of cache coherence, Centaur relies on uncacheable address space for vertex data: all accesses to vertex data bypass caches, avoiding the need to maintain coherence across caches, scratchpads, and off-chip memory. Centaur utilizes the cores' translation look-aside buffer (TLB) to translate virtual to physical addresses when the $\mathrm{O_n}$VMUs update the *activeList* in the cache. Context switching is supported by saving the vertex data stored in scratchpads as part of the process's context. Other functionalities, such as thread scheduling, are independent of Centaur's architecture and are performed as in a traditional CMP.

Integration with software frameworks. To enable a seamless integration of Centaur with graph-software frameworks, such as Ligra [102], GraphMAT [105], *etc*., we aimed to minimize system-level facing changes. Indeed, in Centaur, it is sufficient to annotate atomic operations (described in Section 2), a task that can be accomplished by a simple source-to-source transformation tool. For instance, Figure 5.6 shows the Centaur-ready version of the *update phase* in PageRank; the atomic operation is translated into two write operations: one to the *vertex-ID* memory-mapped register, the other to the *src_data* memory-mapped register. These operations are received by the $\mathrm{O_n}$VMUs: the first passes the ID of the target vertex for the atomic operation, and the second the corresponding new rank value. The transformation tool should also augment the software frameworks with code to pass configuration parameters from the cores, such as atomic operation type, base and range of vertex data locations, *etc*. Other aspects of these software frameworks, including multi-threading, load balancing, *etc.*, remain unchanged and are independent of the Centaur architecture.

5.5 Further Design Considerations

Dynamic graphs. Although the connectivity of vertices in dynamic graphs changes overtime, Centaur continues to provide performance benefit, as existing popular vertices tend to remain popular for a while, due to a characteristic called "preferential attachment" [18]. However, after a considerable amount of time, a large portion of the vertices stored in off-chip memory might become more popular than those in on-chip memory, in which case, the graph's vertices should be reordered periodically for optimal benefits. Alternatively, a hardware-based vertex replacement strategy, as discussed in Section 5.3, or a dynamic graph partitioning technique similar to [74] can be employed at the cost of a small hardware overhead.

Applicability to other application domains. While Centaur specifically targets graph analytics, we foresee its deployment in other application domains, such as MapReduce and database queries. For instance, in MapReduce, the word-count application determines the frequency of words in a document, which has been found to follow Zipf's law; that is, 20% of the words occur 80% of the time [79]. Centaur can take advantage of such skews in word occurrences by processing high-frequency words in the on-chip memory, and the remaining ones in off-chip memory.

On- and off-chip memory options. While using scratchpads as on-chip memory leads to high performance benefits, it also entails high design costs. To alleviate these costs, a portion of the caches can be re-purposed to operate as scratchpads, using a technique similar to Intel's Cache Allocation Technology (CAT) [4]. For off-chip memory implementations, Centaur is not limited to a HMC design; it can also leverage other architectures, such as High Bandwidth Memory.

5.6 Experimental Setup

In this section, we provide the experimental setup that we deployed to evaluate the Centaur solution.

To evaluate Centaur, we compared it to a chip multi-processor (CMP) solution and two state-of-the-art proposals: GraphPIM, a processing-in-memory solution [81], and OMEGA [18], which leverages scratchpads with associated compute units. We modeled all these solutions and Centaur in a gem5 simulator[34]. The CMP has 16, 2GHz, 8-wide O3 cores running the x86 ISA, 32KB L1 instruction and data caches, and 16MB of shared L2 cache. The off-chip memory for all of the above solutions is based on HMC, which includes 8GB of memory partitioned over 32 vaults, each vault providing a peak bandwidth of 20GB/s. The scratchpads, O_nACUs, and O_nVMUs are modeled by extending gem5's cache implementation, whereas the O_{ff}ACUs and O_{ff}VMUs are modeled by extending gem5's HMC model. We re-purposed half of the shared L2 cache as scratchpads for Centaur and OMEGA, while keeping the L2 cache intact for the CMP and GraphPIM. Finally, we mapped Ligra [102], a highly optimized software graph framework, to the simulator through the "m5threads" library, and ran the simulations in "syscall" emulation mode.

Workloads and Datasets. Once again, we considered for our evaluation all algorithms discussed in Section 2.3 except *SpMV*. We did not consider *SpMV* because it was not implemented in the Ligra software framework, which we utilized for our evaluation. We summarized the key characteristics of those algorithms in Table 2.1. We considered six representative real-world graph datasets from the pool described in Section 2.4: *lj*, *wiki*, *sd*, *USA*, *rCA*, and *rPA*. We used all of these datasets as inputs of each algorithm, except for *SSSP*, *TC*, and *KC* for which we used only the smallest datasets among those, due to their long simulation time. We chose these algorithms because they have representative

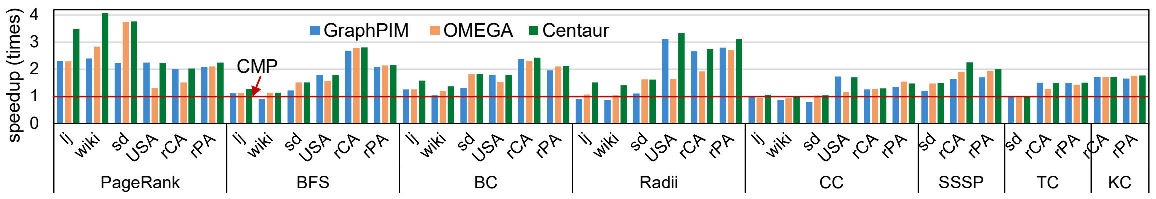


Figure 5.7: Performance improvement comparison. Centaur provides up to 4.0× speedup over a CMP baseline, while also achieving up to 1.7× speedup over GraphPIM and up to 2.0× speedup over OMEGA.

characteristics in size, degree, and power-law distribution. Table 2.2 reports their key characteristics. Note that all datasets were sorted by decreasing vertex degree, as discussed in Section 5.3, and the sorted datasets were used in evaluating all three solutions considered.

5.7 Performance Evaluation

In this section, we first present overall performance benefits of the proposed solution, followed by our insights on atomic operation overhead and internal off-chip memory traffic and bandwidth analysis. We also provide discussion of the benefits of the read/write buffers.

Figure 5.7 compares Centaur's performance against other solutions. As shown in the figure, Centaur delivers up to 4.0× speedup over the CMP reference, up to 1.7× over GraphPIM, and up to 2× over OMEGA. Note that GraphPIM excels on graphs with low-degree vertices, whereas OMEGA performs best on graphs with average high-degree. However, Centaur consistently delivers the best performance on graphs with any average degree. This trait also holds across graph sizes: OMEGA performs best on small graphs where the scratchpads can hold a valuable fraction of the vertex data, while GraphPIM works best on large non-power-law graphs. Once again, Centaur delivers high performance across the entire graph-size range. Additionally, note how Centaur performs best with algorithms that are dominated by a high density of vertex access – where Centaur can offer benefit – such as *PageRank*, which processes all vertices during each iteration. In contrast, *TC* and *KC* are more compute-intensive, and thus attain a lower speedup.

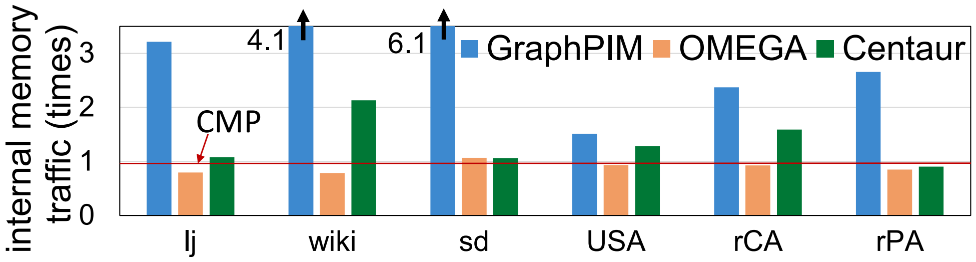


Figure 5.8: Internal off-chip memory traffic analysis. GraphPIM generates 4.7× more traffic between O_{ff}ACUs and memory than a CMP's traffic between cores and memory, on average. Centaur reduces it to 1.3×. The analysis is carried out on *PageRank*.

Atomic operation overhead. Centaur's high performance benefit is mainly due to its computing 100% of atomic operations in on-/off-chip memory units. GraphPIM achieves the same goal by executing atomic operations in off-chip memory, but at the cost of generating high internal memory traffic, as discussed in the next section. In contrast, OMEGA computes only a fraction of atomic operations in on-chip memory: up to 20% for non-power-law graphs and 74% for power-law graphs.

Internal off-chip memory traffic and bandwidth analysis. GraphPIM's main drawback is that it incurs a large amount of traffic between the O_{ff}ACUs and their associated memory partitions, as the O_{ff}ACUs generate two memory requests (one read and one write), both at a cache line granularity, for each atomic operation. Figure 5.8 shows this traffic, indicating a 4.7× increase over a baseline CMP, on average. Centaur limits the read/write requests to low-degree vertices, reducing the traffic to 1.3× of the baseline CMP, on average. Because of these transfers, which are related to atomic operations, both solutions attain high internal bandwidth utilization, 10.5× for GraphPIM, and 2.1× for Centaur, over the baseline CMP, as reported in Figure 5.9. In contrast, the other two solutions considered, the CMP and OMEGA, have much lower utilization as their cores are suspended during the execution of atomic operations. Note that Centaur's traffic could be further reduced by enabling the O_{ff}ACUs to access their associated memory partitions at a word-granularity [115].

Benefits of read/write buffers. We conducted an analysis of the impact of read/write buffers in Centaur, and found that read buffers moderately improve the speedup of *PageR-*

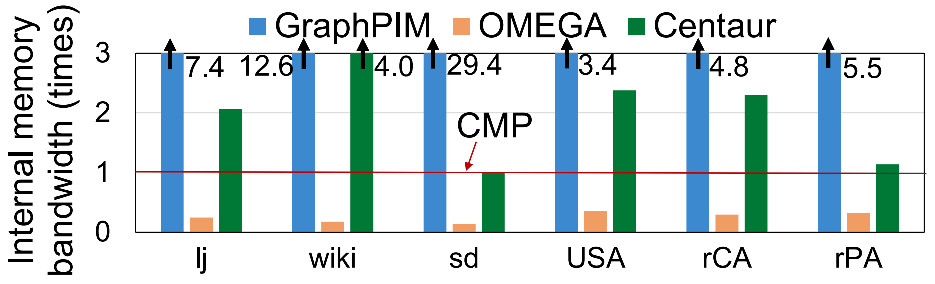


Figure 5.9: Internal off-chip memory bandwidth utilization. GraphPIM and Centaur achieve internal memory bandwidth utilization of 10.5× and 2.1×, on average, over a baseline CMP. The analysis is carried out on *PageRank*.

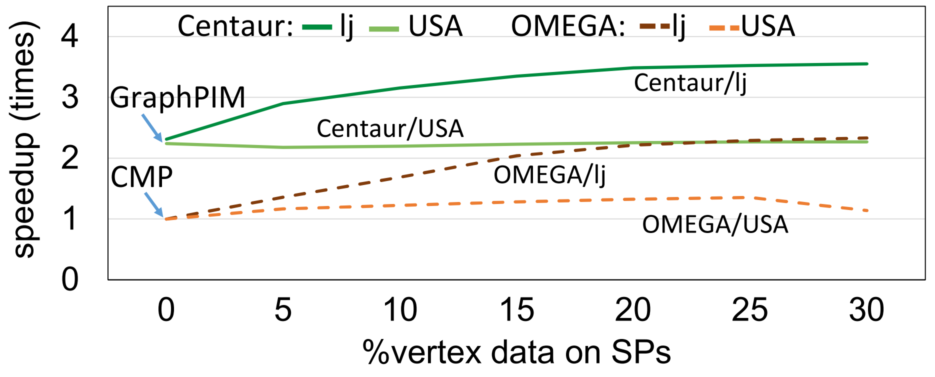


Figure 5.10: Scratchpad sensitivity analysis on *PageRrank*. Centaur's and OMEGA's speedup rates improve significantly with larger scratchpad sizes on a power-law dataset as *lj*, while a graph as *USA*, which does not follow the power law, is fairly insensitive to it. Centaur outperforms OMEGA across the entire range swept.

ank over all our datasets from an average of 2.6× to 2.8×, and that write buffers further improve it to an average of 3.0×.

5.8 Sensitivity Studies

This section provides sensitivity studies in terms of scratchpad size, number of vaults and ACUs.

Scratchpad size sensitivity. Figure 5.10 compares trends in performance speedup as we sweep the scratchpad size in Centaur and OMEGA, while keeping the total scratchpad plus shared L2 cache size similar to the shared L2 cache size of the baseline CMP and GraphPIM. We note that larger scratchpads correspond to speedup increases for a power-

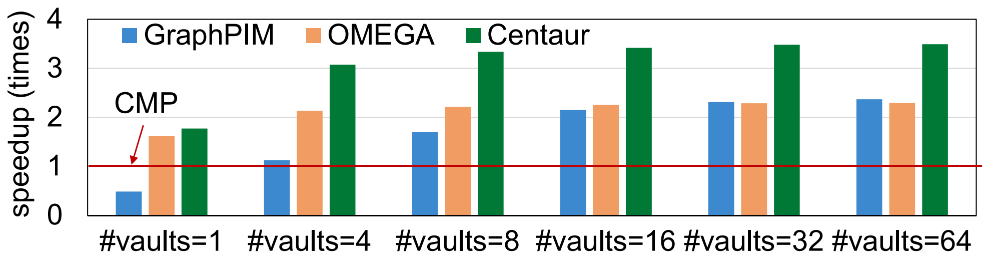


Figure 5.11: Sensitivity to the number of vaults. Centaur maintains a performance advantage over both OMEGA and GraphPIM across a wide number-of-vaults range. The analysis is carried out on *PageRank* processing *lj*.

law graph, like *lj*, while this correlation is absent for a graph that does not follow the power-law, as is *USA*.

Vaults, ACUs, and bandwidth sensitivity. Figure 5.11 evaluates the speedup attained when varying the number of vaults and, correspondingly, the number of O_{ff}ACUs and the off-chip bandwidth available. As shown, Centaur presents the best performance consistently across the spectrum of setups considered.

5.9 Area, Power, Thermal, and Energy Analysis

This section presents area, power, and thermal analysis of Centaur. It also provides discussion on Centaur's energy benefit.

To evaluate the physical impact of Centaur, we carried out an analysis on Centaur's synthesized components. To this end, we synthesized the processor cores using McPAT, the caches and scratchpads using Cacti at 45nm technology, and the ACUs and VMUs using the IBM 45nm SOI library. We referred to prior works to obtain area, power, and per-access energy cost of the HMC's SerDes links, logic layer, and DRAM layer [81, 23]. We then estimated both static and dynamic energy consumption by considering the execution time of each algorithm and the energy costs incurred by each module's per-data access, along with their utilization rate, which we obtained from our simulation infrastructure.

Our area analysis indicates that Centaur's and OMEGA's on-chip components occupy 3% less area than the CMP primarily because the scratchpad portions of on-chip storage

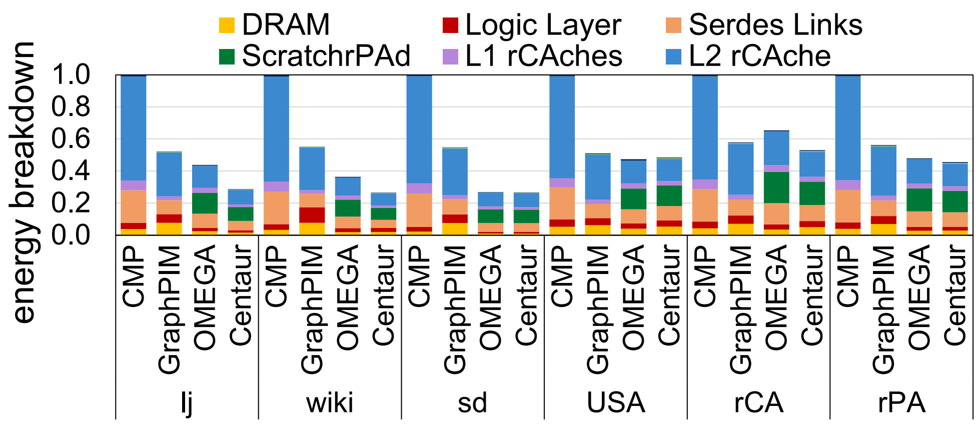


Figure 5.12: Breakdown of uncore energy consumption. Centaur provides up to 3.8× in energy reduction, compared to a CMP, and a better energy profile than both GraphPIM and OMEGA across all datasets. Note how Centaur greatly reduces the contributions to energy by L2 caches. ACUs and VMUs are not shown because of their minimal footprint. The analysis is carried out on *PageRank*.

are direct-mapped, thus requiring much simpler control logic than multi-way caches. As for power, Centaur's and OMEGA's on-chip modules draw 1% less in peak power, due to, once again, the simpler controller logic in the scratchpads. The off-chip hardware modules (O_{ff}ACUs and O_{ff}VMUs), common to both Centaur and GraphPIM, occupy $<1\%$ of the baseline HMC's logic layer. Finally, Centaur's thermal density was estimated at 62mW/mm^2, well below the reported constraint for the HMC, of 133mW/mm^2 [23].

On the energy front, since Centaur's design targets uncore components, we only provide an energy analysis for those units, and present it in Figure 5.12. As shown, Centaur provides up to 3.8× in energy improvement over a baeline CMP, primarily due to reduced execution time and less frequent cache, DRAM, and logic layer accesses.

5.10 Bibliography

In this section, we provide a summary of related works. We start with solutions that are based on CMPs, GPUs, and specialized hardware architectures. We then focus on the more relevant related works that rely on processing in on-chip and off-chip memory techniques.

Prior works looking to optimize the execution of graph analytics on CMPs have proposed reordering the input graphs [119] and/or partitioning them, so that each partition fits in on-chip memory units [119]. Solutions based on GPUs [88] and specialized hardware architectures [54] have also found some benefit, but only for a limited set of graph algorithms. Moreover, specialized-hardware solutions have limited applications because of their extreme design specialization. All these works tackle challenges orthogonal to ours.

Solutions based on processing in off-chip memory [24, 81] or on-chip memory [18, 80] are closest to our work. Among them, [24] proposes an architecture that automatically identifies the locality profile of an application, and processes it at the cores or on off-chip memory accordingly. Specific to graph analytics, [81] further optimizes [24] by eliminating the overhead incurred in automatically identify irregular accesses, as they presume that every access is random. [18] makes the observation that a certain portion of the graph data can be processed in on-chip memory for power-law graphs. Centaur brings to light the best opportunities for both approaches by processing vertex data both at on- and off-chip memory. As far as we know, it is the first solution of its kind and has the potential to be deployed in other domains.

5.11 Summary

In this chapter, we proposed Centaur, a novel solution that leverages both processing near on-chip and off-chip memory architectures. Centaur holistically optimizes the processing of both frequently-accessed data as well as infrequently-accessed data. Centaur is capable of delivering high-performance computation on a wide range of graph datasets. It provides up to 4.0$\times$ performance speedup and 3.8$\times$ improvement in energy, over a baseline chip multiprocessor. It delivers up to 1.7$\times$ speedup over a processing near off-chip memory solution, and up to 2.0$\times$ over a processing near on-chip memory solution. Centaur's on-chip components have a smaller footprint than the baseline, while off-chip ones occupy $<$1% of the baseline HMC's logic layer.

The solution presented in this chapter adopted our two strategies discussed in Section 1.4. We followed our first strategy of specializing memory architecture, where we utilized scratchpads to store frequently-accessed data, reducing the traffic between processor cores and off-chip memory. We then adopted our second strategy of processing near data storage, where we co-located the scratchpad with specialized compute engines for processing the data in-situ, reducing the traffic between processor cores and on-chip storage units. We once again embraced our second strategy of processing data near storage, where we co-located off-chip memory units with specialized compute engines for processing infrequently-accessed data in-situ, reducing the associated performance and energy inefficiencies.

The solutions presented thus far adapt our three strategies discussed in Chapter 1, providing a comprehensive set of solutions for graph analytics. In the next chapter, we show the applicability of our strategies to another domain of applications, called MapReduce.

CHAPTER 6

Optimizing Memory Architectures Beyond Graph Analytics

In this chapter, we demonstrate the applicability of our strategies discussed in Chapter 1 to another domain of applications, specifically MapReduce applications. MapReduce applications provides an effective and simple mechanism to process a large amount of data, delivering their result in a simple key-value data. They are utilized for various purposes, such as indexing webpages for improving search results and analyzing log files for providing summary data, such as website usage. In all of these applications, they maintain key-value data structures used to track the occurrence of all relevant "key"s on information related to the analysis. For instance, word count, a typical MapReduce, application maintain words as key and their frequency in input documents are value. During a typical execution of MapReduce applications on chip multiprocessor (CMP) machines, each core involved in the execution must manage its own key-value data, as a single centralized key-value data would lead to extreme access contention. In addition, each core accesses its own key-value data with random-access patterns. Unfortunately, this setup creates pressure on the memory system, causing high on- and off- chip communication, and resulting in performance and energy inefficiencies. Many real-world data inputs for MapReduce applications have key-value pairs with some keys appearing more frequently than others. This characteristic can be leveraged to develop solutions that combat the limitations described

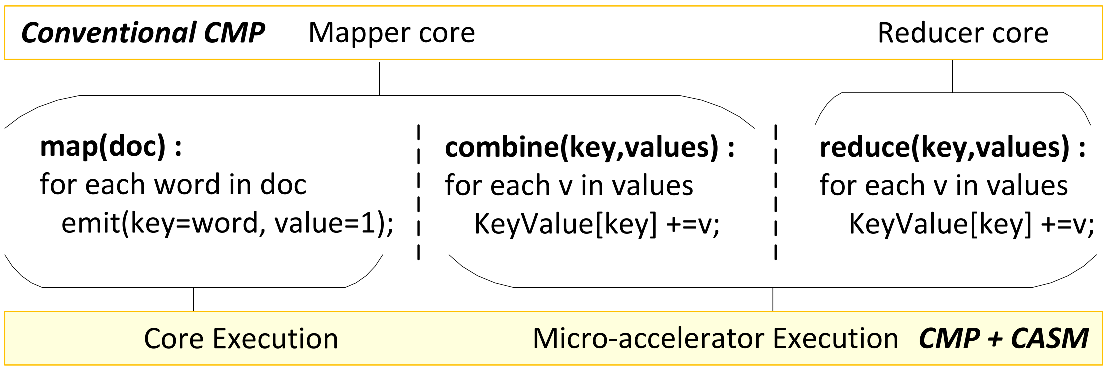


Figure 6.1: MapReduce for *wordcount*. *map* emits a kv-pair for each word, *combine* aggregates words emitted from a mapper, whereas *reduce* aggregates emitted words from all mappers. In CASM, *map* is executed by the cores, while *combine* and *reduce* are offloaded to the accelerators' network.

above.

In this chapter, we strive to overcome the above limitations by designing an architecture that can accelerate MapReduce applications, leveraging the skewed access frequency of key-value pairs. To achieve this goal, we rely on the strategies discussed in Section 1.4. Specifically, we explore a specialized memory architecture, where each core of a CMP is augmented with scratchpad memories. The scratchpads house frequently-occurring key-value pairs, which are often responsible for the majority of the accesses, thus reducing the traffic from/to the off-chip memory. We also investigate the benefits of processing data near storage, by co-locating each scratchpad with a lightweight compute engine. This second strategy reduces the interconnect traffic that otherwise would take place between a core and a remote storage unit. As a byproduct, some of the data processing is performed using specialized compute engines, instead of general-purpose cores, freeing up processor cores to perform other computation tasks. Each core's scratchpads and their locally-attached compute engines form an accelerator unit, resulting in an overall system that comprises a network of accelerators, enabling the CMP to efficiently execute MapReduce applications

6.1 Background and Motivation

MapReduce is a simple programming framework for data-intensive applications. Users can write complex parallel programs by simply defining map and reduce functions, while all the remaining parallel programming aspects, including data partitioning and data shuffle stages, are handled by the MapReduce infrastructure. In a typical scale-up MapReduce framework, the application's input data is first partitioned over the cores in the system. Then, each core runs the user-defined ***map*** function, which processes the input data and produces a list of intermediate key-value pairs (*kv-pairs*), followed by a ***combine*** stage, which aggregates keys to partially reduce local kv-pairs, conserving network bandwidth. Once this stage is complete, the intermediate data is ***partitioned*** and ***shuffled*** within the network, while the cores assume the role of reducers, so that all kv-pairs with the same key are transferred to a same reducer. In the final stage, each core executes the user-defined ***reduce*** function to complete the aggregation of its kv-pairs. As an example, Figure 6.1 provides the pseudo-code for *wc* (wordcount), a classic MapReduce application with a wide-range of uses. *wc* computes the frequency of occurrence of each word in a document. In the MapReduce framework, the map function parses the input document and identifies all the words. It then emits each word as part of a kv-pair, with the word as the key and an initial value of 1. The combine stage partially aggregates kv-pairs at each core, before they are transferred to the reducer core. Finally, the reduce function collects all kv-pairs and sums up the values for each word, producing in output a final list of unique words along with their frequency.

Phoenix++: optimizations and inefficiencies. Phoenix++ is among the most optimized scale-up MapReduce frameworks for CMPs. One major optimization adopted by Phoenix++ is the interleaving of map and combine stages, which lowers memory pressure caused by applications with many kv-pairs. In Phoenix++, kv-pairs are aggregated locally, immediately after they are mapped. The left part of Figure 6.2 illustrates the MapReduce steps for Phoenix++. At first, each core considers its own data segment, one input at a time;

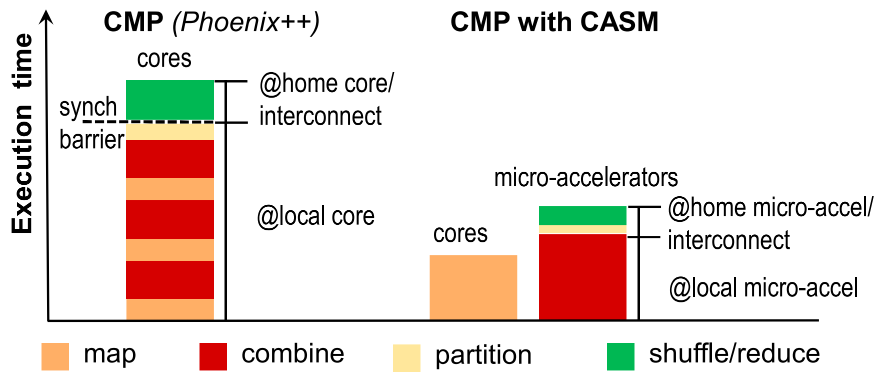


Figure 6.2: Execution flow of MapReduce. Left side - In a typical CMP-based framework, the map, combine and partition stages execute on the local core. A barrier then synchronizes the execution; kv-pairs are shuffled through the interconnect and reduced at the home core. Right-side - When deploying CASM, cores are only responsible for the map stage. They then transfer kv-pairs to the local accelerator, which combines, partitions, and transfers them to the home accelerator for the reduce stage.

maps it into a kv-pair (orange); and combines it with its current database of kv-pairs (red). Then the core partitions aggregated kv-pairs over all the cores based on their key (yellow). At this point the threads are synchronized so that all cores switch from operating as local cores to home cores. kv-pairs are then transferred through the interconnect to the home core and reduced there (green). However, there are still two major inefficiencies in this approach. i) Map and combine stages are executed in a sequential manner. If they were to run concurrently, as suggested in the right part of Figure 6.2, execution time would be significantly reduced. ii) Moreover, each combine function (one per map function) maintains its own kv data structure as a hashtable; thus, keys are replicated in the on-chip caches, creating many off-chip memory accesses when that data no longer fits in cache.

Motivating Study. To gain insights on the execution bottlenecks of a real platform, we analyzed the execution of Phoenix++ with our experimental workloads (Section 6.6) on a 16-core Intel Xeon E5-2630 (2 threads per core) machine, using a range of input data sizes. Figure 6.3 plots the breakdown of execution time by MapReduce stage. Note how map and combine dominate the overall execution time, and the combine stage contributes the majority of the total execution time for most benchmarks. We then analyzed in detail the execution of the combine stage for those benchmarks using Intel's VTune tool and col-

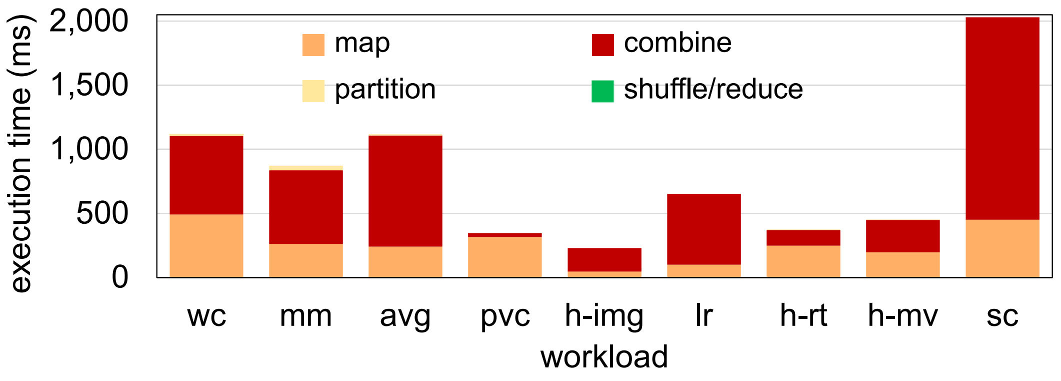


Figure 6.3: Motivating study. Execution time breakdown for our MapReduce workloads, running Phoenix++ on a Xeon E5-2630 V3 machine with the input datasets discussed in Section 6.6. Note how the combine stage dominates the overall execution for most applications.

lected the *Top-down Microarchitecture Analysis Method* (TMAM) metrics [113]. VTune reported that most workloads were primarily back-end bounded (60% average value); that is, the main cause of performance bottlenecks was the inability for instructions to progress through the pipeline. Vtune could also indicate that, among the (lack of) resources that caused back-end bottlenecks, the time overhead in accessing memory was a primary factor in our case. Based on this analysis, we suspected that the bottleneck could be due to the large number of data transfers occurring during the combine stage, driven by the need to maintain multiple, large, and irregularly accessed hashtables (one per core), which often do not fit in on-chip storage. Thus, to assess the impact of these off-chip memory accesses, we setup a 64-core system on the gem5/garnet infrastructure, where all kv-pairs could be accommodated in on-chip storage, as it would be in an ideal case. This last analysis showed that the total data-transfer footprint differential between the real and the ideal system is over 9x.

6.2 CASM Overview

Figure 6.4 outlines our solution, called CASM (Collaborative Accelerators for Streamlining Mapreduce), which comprises a network of accelerators laid out alongside the cores,

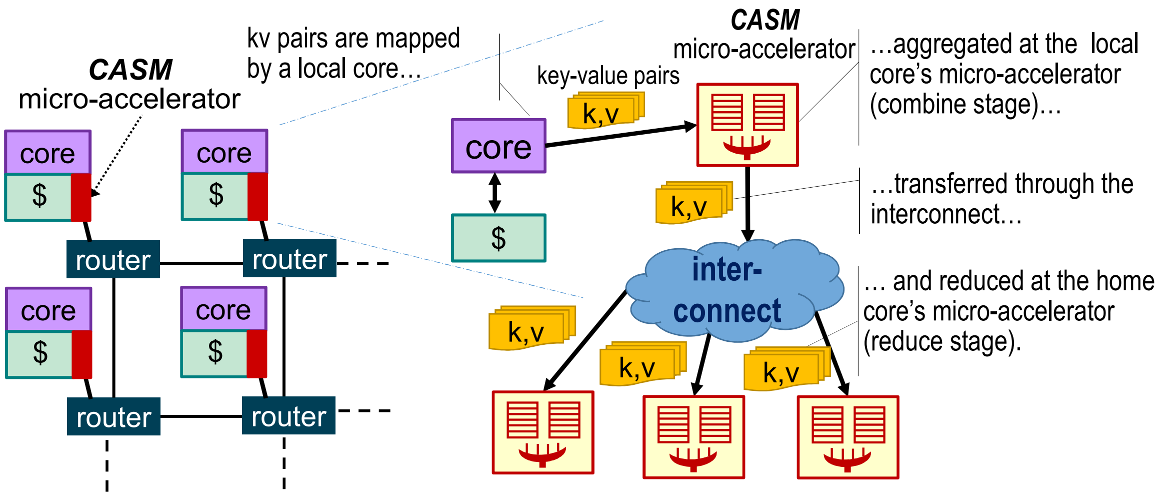


Figure 6.4: CASM deployed in a CMP architecture. Left side - CASM adds a local accelerator to each CMP's core to carry out MapReduce tasks. Right side - Each accelerator aggregates kv-pairs emitted by the local core (local aggregation). Those kv-pairs are then transferred among the accelerators through the CMP's interconnect. At the home core, accelerators execute the reduce stage, so that, in the end, there is one kv-pair per key.

capable of delivering high computation performance locally, while collaboratively managing the application's data footprint, so as to minimize data transfers among the cores and to off-chip memory. Each of CASM's accelerator contains two storage structures: a home scratchpad and a local scratchpad memory. The scratchpad memories (*SPM*) share some similarity with the home and local directories in a directory-based cache coherence protocol, but they are indexed by keys instead of memory addresses. The home SPMs collectively form a large cumulative on-chip memory to store kv-pairs. Each SPM is responsible for its own portion of the keys' space, capturing most keys of the input dataset; spilling to memory occurs rarely, only when the home SPMs cannot store all the keys. While home SPMs minimize off-chip memory access, most kv-pairs would still traverse the interconnect, potentially leading to performance degradation due to interconnect contention. Local SPMs are used to combine kv-pairs locally, so as to greatly reduce contention on the interconnect.

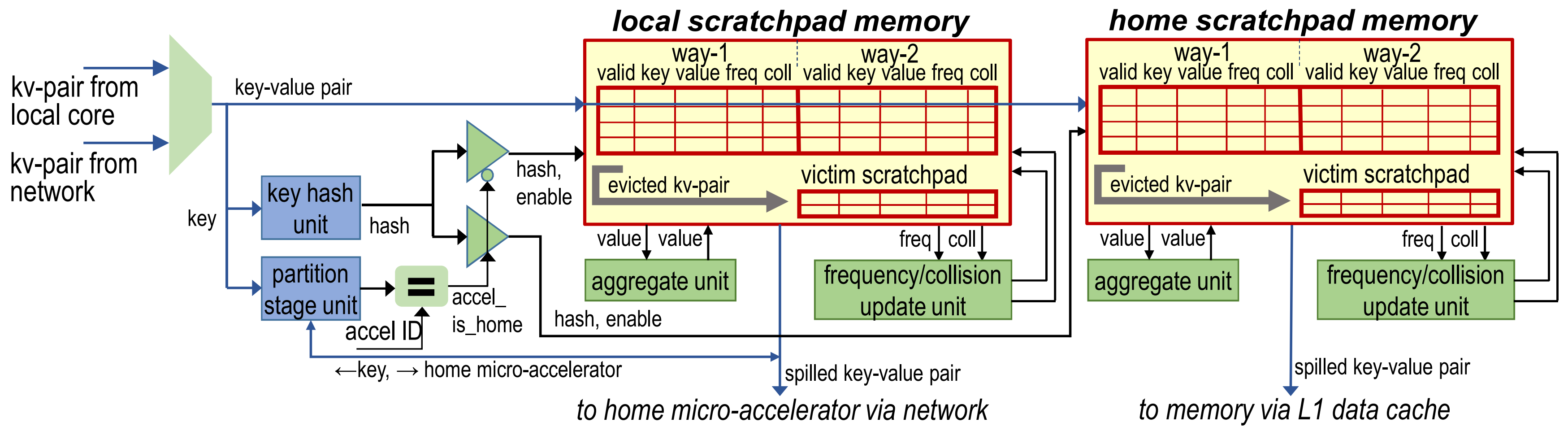


Figure 6.5: CASM's accelerator architecture. Each accelerator includes two *SPMs*, organized as 2-way associative caches with small *victim SPMs*, indexed by a hash function, which aggregates kv-pairs incoming from the local core or the interconnect. The *aggregate units* aggregate values for kv-pairs stored in the SPMs. The *frequency/collision update units* enforce our kv-pair replacement policy. Finally, each accelerator includes a *key hash unit* to compute the hash of incoming keys, and a *partition stage unit*, responsible for deriving the ID of the home core in charge of reducing each unique key. kv-pairs evicted from the SPMs are transferred to their home core (from local SPM) or the local cache (from the home SPM).

6.3 Inside a CASM accelerator

Each CASM's accelerator (see Figure 6.5) comprises a scratchpad memory (SPM), organized into two partitions, one to serve kv-pairs incoming from the local processor core (local SPM), and one to serve kv-pairs incoming from the interconnect (home SPM). Each SPM is complemented by a small "victim SPM," similar to a victim cache, which stores data recently evicted from the main SPM. The accelerator also includes dedicated hardware to compute a range of reduce functions, used both to aggregate data in the local and home SPMs. Logic to compute hash functions, both for indexing the SPMs and for partitioning kv-pairs over home accelerators, completes the design. Note that both local and home SPMs have fixed sizes; thus, it may not be always possible to store all the kv-pairs that they receive. When a local SPM cannot fit all the kv-pairs, it defers their aggregation to the reduce stage, by transferring them to the home accelerator. When a home SPM encounters this problem, it transfers its kv-pairs to the local cache, and then lets the home core carry out the last few final steps of the reduce function.

When an accelerator receives a kv-pair from either its associated core, or the network, it first processes the key through its *key hash unit* and through the *partition stage unit*. The purpose of the former is to generate a hash to use in indexing the SPM. The latter determines which home accelerator is responsible for reducing this kv-pair: if the local accelerator is also the home accelerator (*accel_is_home* signal), then we send the kv-pair to the home SPM, along with the hash value and an enable signal; otherwise, we send it to the local SPM. Note that all kv-pairs incoming from the network will be aggregated at the home SPM. Pairs incoming from the local core will be aggregated at the home SPM only if the local core is also the home one. Each SPM is organized as a 2-way cache augmented with a small victim cache. Associated with each SPM is an *aggregate unit*, responsible for deploying the specified reduce function to combine two kv-pairs with the same key. Each SPM is also equipped with a dedicated unit, called *frequency/collision update unit*, to keep up to date the replacement policy information. Finally, note that when a kv-pair is spilled from a SPM, it is transferred out through the interconnect, either to a home accelerator or to local cache. Spilling occurs because of eviction from the victim SPM.

Hash Function Units. Our accelerator includes two hash computing units: the *key hash unit* and the *partition stage unit*. The former is used to compute the index value to access the SPM, which is organized as a 2-way associative cache. The latter uses a hash function to create a unique mapping from keys to an accelerator ID (*accel ID*), so that kv-pairs from each core can be distributed among the home accelerators for the final reduce stage, and each home accelerator is responsible for the reduce stage of all the keys hashed to its ID. We used an XOR-rotate hash for both units. We considered several alternatives for the *key hash unit* and found that XOR-rotate [64] is both the most compute efficient and has a low collision rate.

SPM (Scratchpad Memory). The accelerators' SPM is a fixed-size storage organized as a 2-way associative cache, where the set is determined by the *key hash unit*. Each entry in the SPMs stores the following fields: valid bit, key, the corresponding value to that key, and frequency and collision values. When a kv-pair accessing the SPM "hits", that is, the keys of the SPM entry and the kv-pair are a match, we aggregate the two values by sending them to the aggregate unit, and then update the entry in the SPM. When a kv-pair "conflicts" in the SPM, that is, when both stored keys are different, then we leverage our replacement solution to determine which kv-pair (the incoming one or one of those already stored) should be removed from the SPM and evicted to the victim SPM as discussed below. Of course, if there is an empty entry in the SPM corresponding to the current hash index, the incoming kv-pair will be stored there. We maintain separate local and home SPMs, one to store kv-pairs undergoing local aggregation, the other for kv-pairs in their final reduce stage. We keep them separate because, particularly for applications with a large number of unique keys, the home SPMs avoid key duplication and provide the equivalent of a large unified on-chip storage, minimizing kv-pair spills to memory. Local SPMs, on the other hand, are beneficial in avoiding network congestion. We also considered an alternative direct-mapped SPM organization, but dismissed it due to much higher collision rates.

kv-pair Replacement Policy. Since the SPMs are of limited size, it is possible for two distinct keys to be mapped to the same SPM entry and collide. Our replacement policy determines if a previously-stored key must be evicted and replaced with an incoming kv-pair. Upon storing a new entry in the SPM, its collision is initialized to 0 and its frequency to 1. Each time a new kv-pair is aggregated with a current entry, its frequency value is incremented, while the collision value remains unmodified. Each time there is a key conflict, that is, the incoming kv-pair has a different key than those stored in the SPM set, the collision is incremented for both kv-pairs in the set. Whenever an incoming kv-pair conflicts in the SPM, we analyze the two entries already in the SPM set to determine if one should be replaced. If, for either entry, the frequency is greater than the collision, then the entries are frequent ones and we simply update their collision values, but no replacement occurs. In this case, we send the new kv-pair to its destination (either its home accelerator or spilled into memory through the local cache). If, instead, collision exceeds frequency for one of the entries, then that entry is deemed infrequent, and it is replaced by the incoming kv-pair. If both entries are infrequent, we evict the one with the lowest frequency. Upon replacement, the frequency and collision values of the new entry are reset.

Victim SPM. Depending on the sequence of keys accessing the SPMs, it is possible to incur thrashing, where a small set of keys keeps overwriting each other in the SPM. To limit the impact of this issue, we augment each SPM with a small, fully-associative "victim SPM": kv-pairs are stored in the victim SPM when they are denied entry to or evicted from the main SPM. All kv-pairs that are either evicted or rejected by the victim SPM are transferred to the home accelerator (from a local SPM), or to local cache (from a home SPM).

Aggregate Unit. The accelerator's *aggregate unit* implements the MapReduce reduce function. Our accelerator design supports several reduce operators, which cover a wide range of common MapReduce applications: we support addition; computing the maximum

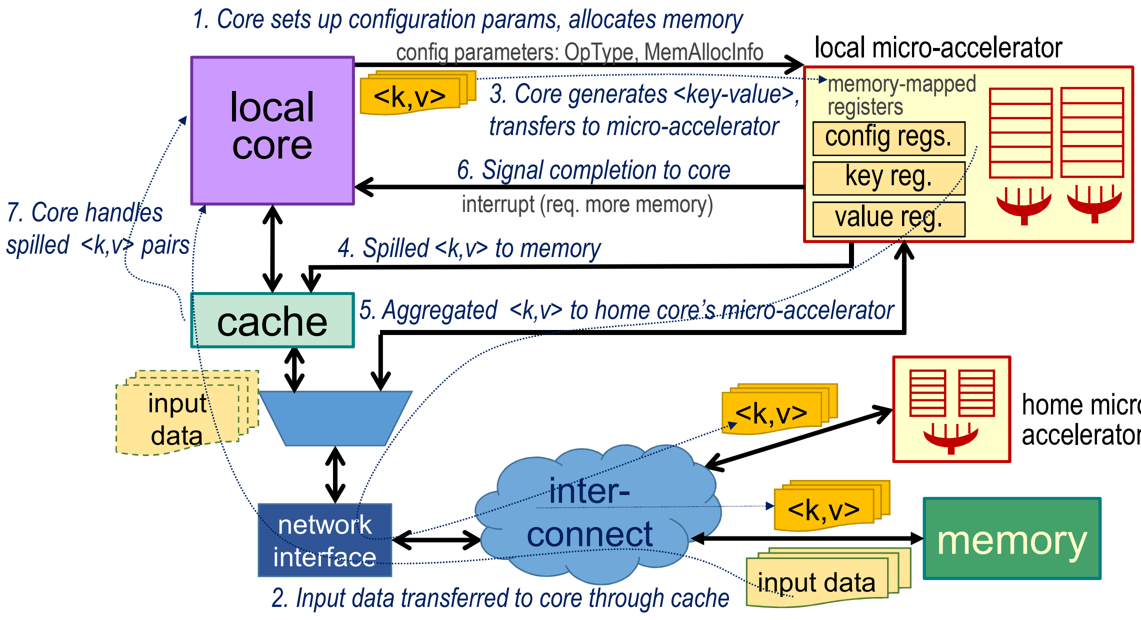

1. Core sets up configuration params, allocates memory
config parameters: OpType, MemAllocInfo
local micro-accelerator
local core
<k,v>
3. Core generates <key-value>, transfers to micro-accelerator
memory-mapped registers
config regs.
key reg.
value reg.
6. Signal completion to core
interrupt (req. more memory)
7. Core handles spilled <k,v> pairs
4. Spilled <k,v> to memory
cache
5. Aggregated <k,v> to home core's micro-accelerator
input data
<k,v>
home micro-accelerator
inter-connect
<k,v>
network interface
memory
input data
2. Input data transferred to core through cache

Figure 6.6: System integration showing the sequence of CASM execution steps.

value, the minimum value, and average; and more. The average operation is implemented by separating it into two addition operations that are stored into the two halves of the data field: the first maintains the sum of values, and the second counts the total number of values. As a general rule, CASM requires that the reduce function be both commutative and associative, since the accelerators process kv-pairs independently from each other, and thus no ordering can be enforced on the kv-pair processing in the combine stage. Note that many practical applications satisfy this requirement and employ straightforward and common operators, as those we provide [40]. It is also possible to replace the *aggregate unit* with a reconfigurable logic block to provide further flexibility.

6.4 System Integration

Figure 6.6 illustrates the interactions among all systems components during execution. At the start of an application, each core sends the type of aggregate operation and the pre-allocated memory region that the accelerator shall use for reduced kv-pairs at the end of the execution, and for potential spilled pairs. The core then begins sending mapped kv-pairs to its accelerator, completing this transfer with a *map completion* signal. Whenever an accelerator receives this signal from its local core, it sends all kv-pairs from its local SPM to their home SPMs, and then sends out a completion signal to all other accelerators in the system. After an accelerator has received completion signals from all other accelerators, it flushes out the contents of its home SPM to the local cache and signals to its core that the processing has completed. At this point, the corresponding core retrieves the final, reduced kv-pairs from memory and carries out a final reduce step, if any kv-pair was spilled during the execution. All communication from a core to its local accelerator is through store instructions to a set of memory-mapped registers, while accelerators communicate with the core via interrupts and shared memory. Each accelerator is also directly connected to the on-chip network interface to send/receive kv-pairs and synchronization commands to/from other accelerators and memory.

Cache Coherence. SPM storage is for exclusive access by its accelerator, and it is not shared with other units. Communication among the accelerators also happens via custom packets, instead of the CMP's coherence protocol's messages. Thus, CASM's read/write operations from/to the SPMs are transparent and oblivious to the CMP's coherence protocol. To handle spilling of kv-pairs to the L1 cache, each accelerator, on behalf of its local core, writes the spilled kv-pair to the cache, handled by the CMP's coherence protocol. Note that, for each kv-pair, spilling is handled by only one home accelerator.

Virtual Memory. In accessing memory storage set up by the local core, each accelerator uses the same virtual memory space as its core; thus, addresses are translated with the same page table and TLB as the process initiating the MapReduce application. Once the physical address is obtained, the access occurs by read/write to memory through the local cache.

Context Switching. During context switching of a process, the content of the local SPMs is flushed into their respective home SPMs; then, the kv-pairs in the home SPMs are spilled to memory. Once context switching is complete, the accelerators stop issuing further requests to memory, so to avoid accessing stale data in page-tables and TLBs. Note that, when the context is restored, spilled kv-pairs do not have to be re-loaded to the SPMs, as their aggregation can be handled during the reduce step, together with other spilled kv-pairs.

6.5 Composite MapReduce Applications

Many MapReduce applications map to a single MapReduce job, for instance, those considered in our evaluation (see Section 6.6). More complex MapReduce applications often involve multiple analysis steps, which depend on each other's results [121, 39, 33]. One type of such applications involves multiple MapReduce jobs organized in a pipeline fashion, with the output of one fed as input to the next. An example of such an application is *top k*, an application that determines the most popular terms within an input dataset [39]. *top k* is a pipelined MapReduce application: first, it relies on a word count job to determine each term's frequency, and then it deploys a second MapReduce job to identify the k most popular terms. Another example is *TF-IDF*, an application that identifies a list of documents that best match a given search term [33]. *TF-IDF* relies on two distinct MapReduce jobs: *TF* (term frequency) that determines the frequency of a term for each document and *IDF* (inverse document frequency) that determines the uniqueness of a term for a particular document by diminishing the frequency value of terms common across multiple documents. Beyond pipelined applications, other complex applications include those that compute their results iteratively, *e.g.*, k-means [107].

The execution flow structures of both types of applications described above are compatible with CASM. For a pipelined MapReduce application, at the end of the application's first job, the software framework instructs CASM to copy the aggregated kv-pairs from source scratchpads to their destination scratchpads, followed by copying the kv-pairs to the CMP's memory system. If any kv-pair was spilled during the execution of the job, then CASM's cores retrieve the final kv-pairs from memory, reduce those kv-pairs, and then generate the final output of the job (as described in Section 6.4). The next job of the application in the pipeline then takes the output of the first job as input and follows the same steps as that of the first job to produce its outcome. This process will continue until the final job of the application is executed. The output of the final job of the application is provided as the result of the overall application. The same process can be extended to iterative MapReduce applications except that instead of per MapReduce job, CAM initiates the above process per each iteration of the application.

6.6 Experimental Setup

In this section, we provide the experimental setup that we deployed to evaluate the CASM solution.

In order to perform a detailed micro-architectural evaluation of a CASM-augmented, large-scale CMP architecture, we implemented our design with gem5 + Garnet [34], a cycle-accurate simulation infrastructure. We ported the Phoenix++ framework [107] to gem5's x86 model using "m5threads" and carried out the simulations using the "syscall" mode. We modeled the baseline scale-up CMP solution as a 64-core CMP in a 8x8 mesh topology, with 4 DDR3 memory nodes at the corners of the mesh. Each core is OoO, 8-wide, and equipped with a 16KB private L1 I/D cache and a 128KB slice of a shared L2 cache. The cache coherence is based on a MOESI directory-based protocol. The interconnect uses 5-stage routers. CASM's SPMs are also 16KB, and use 8-entry victim SPMs. Each entry in the SPMs contains 64-bit key and value fields, along with an 8-bit field to implement our replacement policy. For the 'average' reduce operation, the value field is partitioned into two 32-bit fields, accruing sum and entry-count.

Workloads and Datasets. We considered several workloads in our evaluation: *wc* (wordcount, 257K keys), *h-img* (histogram image, 768 keys), and *lr* (linear regression, 5 keys) are gathered from the Phoenix++ framework, while *sc* (counts frequency of 64-bit hashes of 3-word sequences, 3.5M keys), *h-rt* (histogram ratings of movies, 5 keys), and

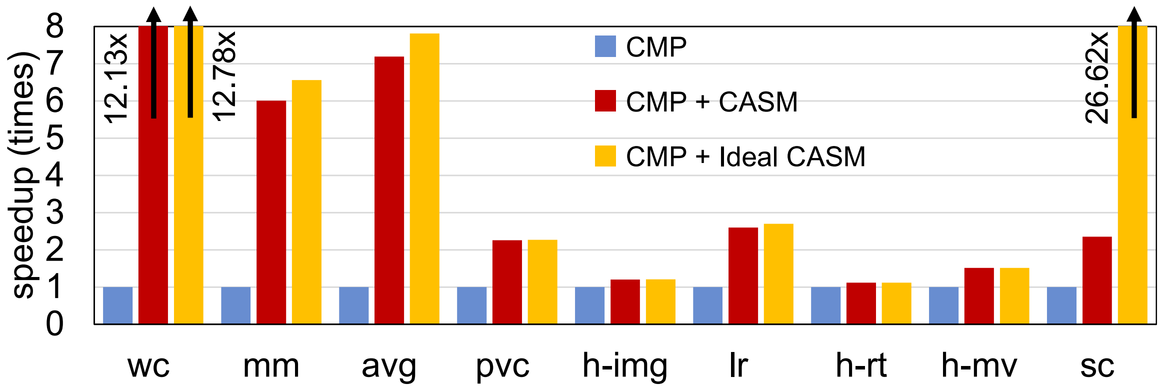


Figure 6.7: CASM performance speedup over a baseline CMP execution of MapReduce. SPMs are 16KB for CASM, and infinite for the ideal variant.

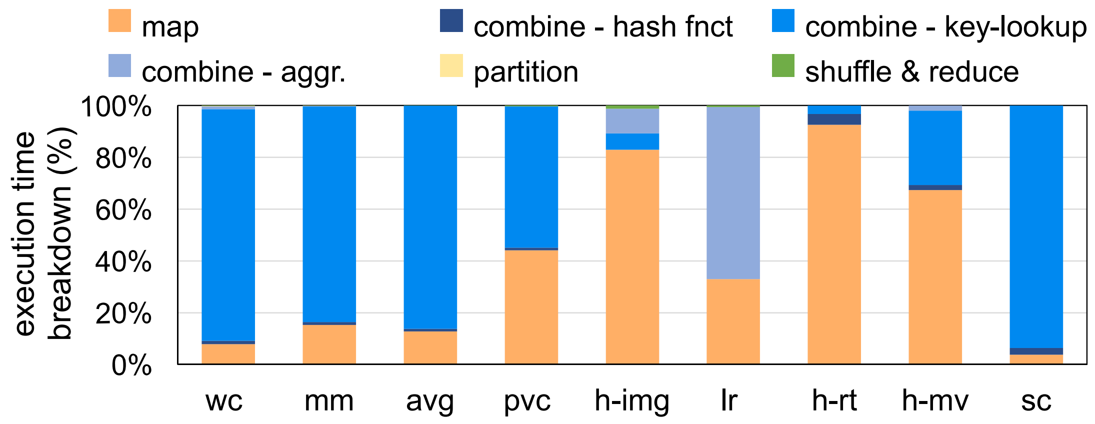
h-mv (movies by histogram rating, 20K keys) are adopted from the PUMA benchmarks [21]. Finally, we developed *pvc* (page view count, 10K keys), *mm* (min-max), and *avg* from scratch using the API of Phoenix++. The datasets come from the same framework as the workloads, except for *avg* and *mm* (we used a list of 28K cities and corresponding temperature logs) and *pvc*, which we generated. For *wc*, we preprocess its dataset to filter out words that are more than 8 characters long because of our 64-bit key-size limitation. We used two families of datasets. In running the cycle-accurate gem5 simulations, we used datasets of 1GB for *h-rt* and *h-mv*, and 100MB for all others. The dataset footprint was driven by the limited performance of the gem5 simulator (*e.g.*, 15 days of simulation time for a 1GB input file). Note that datasets of similar size have been used to evaluate prominent scale-up MapReduce frameworks [97, 114, 107]; thus, we determined that such footprint sizes are useful and provide practical insights. To further evaluate the scalability of CASM to large-scale datasets, we carried out a study with dataset sizes of 30GB from [2] for *wc* and *sc*, 100GB from [21] for *h-mv* and *h-rt*, and 1.5GB from [107] for *h-img*. For other workloads, we generated 100GB datasets with the same number of keys.

6.7 Performance Evaluation

In this section, we first present overall performance benefit, then gain insights by inspecting sources of performance gains and data transfer analysis. We also provide discussion on the performance benefit that can be achieved by employing only either the local or home scratchpads.

Figure 6.7 reports the performance speedup of CASM compared to the baseline CMP running Phoenix++. We report both the speedup that we could achieve with infinitely large local and home SPMs (*i.e.*, no kv-pair collision occurs in either of the SPMs) and that of the actual setup (*i.e.*, 16KB SPMs). Note that most applications reach fairly close to their ideal speedup. From the figure, the ideal speedup ranges from 1.1x to 26x, while the speedups we observed with our implementation settings peak at 12x and average at 4x. Note that *wc*, *mm*, and *avg* have many unique keys; yet, CASM achieves an almost ideal speedup. *wc*, in particular, is an important kernel in MapReduce applications. *h-img*, *h-rt*, and *lr* have relatively few unique keys, which comfortably fit in the baseline CMP's caches, reducing our room for improvement. The speedup of *lr* is relatively better because its map stage is less memory-intensive than that of *h-img* and *h-rt* (CASM executes concurrently with the map stage). *h-mv*'s speedup is limited despite its many unique keys, as its input data layout provides a cache-friendly access pattern, which benefits the baseline CMP. *pvc* entails fairly heavy parsing during mapping, thus limiting the speedup potential of CASM. Finally, it is clear that *sc*'s speedup is limited by the SPM size because of its vast number of distinct keys, over 3 million. Yet, CASM's 16KB SPMs were able to deliver a 2x speedup.

Speedup Breakdown. To assess source of the performance gains that CASM attains, we analyzed the execution stage breakdown of our testbed applications running on a Phoenix++/ CMP system. To gather the analysis data, this experiment leveraged our gem5-Garnet infrastructure. Figure 6.8 reports how execution time is partitioned among map, combine, partition, and reduce stages. Moreover, we dissected the combine stage into i) hash function computation; ii) hash-key lookup, which entails memory accesses to walk e



	wc	mm	avg	pvc	h-img	lr	h-rt	h-mv	sc
map concurrency	1.09x	1.18x	1.15x	1.79x	1.20x	1.59x	1.12x	1.21x	1.04x
h/w acceleration	6.17x	3.12x	3.74x	1.13x	1.00x	1.32x	1.00x	1.00x	2.16x

Figure 6.8: Performance insights. The plot reports a breakdown of MapReduce by stage for a baseline CMP framework. The table shows performance improvements provided by i) map stage concurrency alone, and ii) hardware acceleration alone.

the hashtable; and iii) data aggregation. Note that for most applications, the dominant time drain is the hash-key lookup. *pvc* presents a dominating map stage because of the extensive parsing of web addresses, while *lr* is dominated by data aggregation because it only uses five distinct keys, which can be easily mapped into registers. It can be noted that just optimizing the hash key lookup execution via a specialized hardware structure would provide significant overall performance benefits. The table in Figure 6.8 specifies the contributions of each source of performance improvement. As discussed in Figure 6.2, the map stage in CASM executes concurrently with the other MapReduce stages. The first row of the table reports the speedup we would obtain if this concurrency was the only source of performance improvement. The second row of the table reports the speedup we would obtain if the map stage was serialized with the other stages, but combine, partition, and reduce were all accelerated by CASM.

Data Transfers Analysis. To further explain sources of speedups, we tracked off-chip memory accesses for our baseline CMP, the CASM solution and an ideal solution. For the latter, we assumed that all unique keys could be accommodated in the SPMs, with no off-chip memory access for key-lookups. CASM reduces this traffic by 4.22x on average,

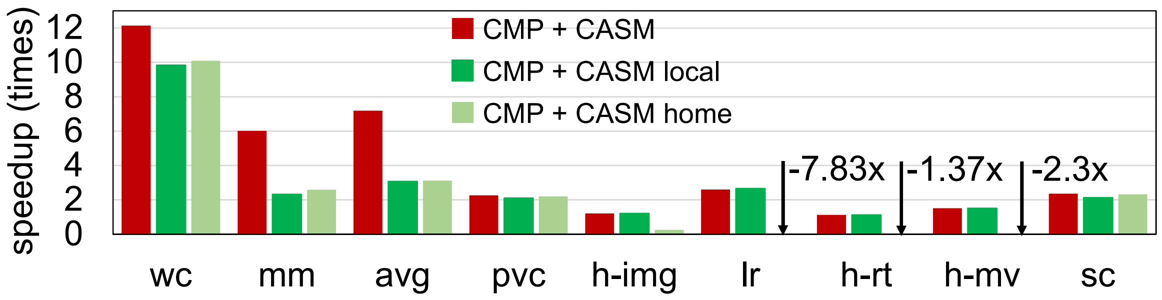


Figure 6.9: Speedup breakdown by home or local SPM. CASM with local SPMs alone shows significant speedup on applications with few unique keys (no spilling to the home SPM). Home SPMs contribute to overall speedup mostly for applications with many keys and low key-access locality.

while the ideal solution achieves a 9x traffic reduction. Such traffic reduction is possible because CASM's home SPMs experience minimal kv-pair spills (<4%), except for *sc*, which amounts to 75% because of its large number of unique keys. CASM's ability to aggressively reduce off-chip memory accesses is the root of the vast performance benefits it provides, since many of our workloads spend most of their time performing key-lookups, which entail accesses to caches and memory. Furthermore, we found that CASM reduces interconnect latency by 7% on average, peaking at 15% for workloads such as *wc*.

SPM (scratchpad memory) Architecture Analysis. In our previous analysis, we evaluated the contribution to overall speedup by home and local SPMs. Figure 6.9 provides the results of a comparative analysis when using only home SPMs or only local SPMs – note that each SPM is still 16KB in size. When CASM uses only local SPMs, the benefits of local aggregation stand, and we still obtain acceptable speedups, although to a lower extent: local SPMs contribute to 2.75x of the speedup, on average. In particular, for applications with a moderate number of keys, such that they can all fit in the local SPM, local SPMs provide the majority of the overall performance benefit of CASM. For other applications, with many distinct keys, performance becomes more challenging using only local SPMs, because of the high rate of spilling to the home SPM. On the other hand, we found that having only home SPMs provides on average a 2.26x speedup. The performance boost here is extremely variable: applications with a large number of keys provide the best benefit (*e.g.*,

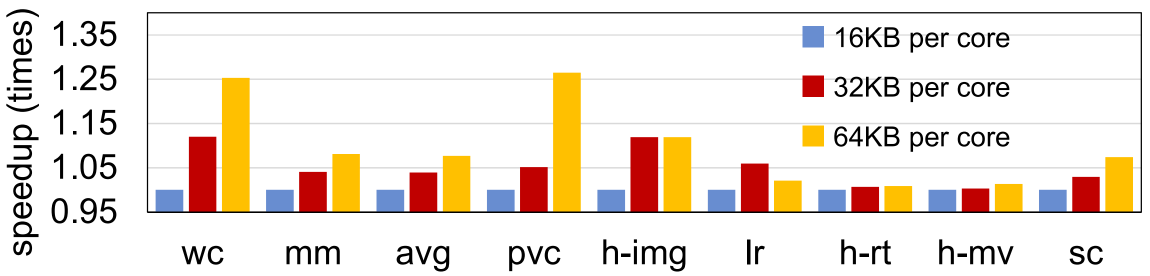


Figure 6.10: L1 cache sensitivity study. Performance speedup over the baseline CMP, for varied L1 cache sizes. The performance improves by no more than 26% at L1=64KB.

wc). Note that, in a few cases, the use of home SPMs alone leads to a slowdown, as much as -7x (*lr*), because of the high interconnect traffic generated in transferring all kv-pairs directly to the home SPMs, with no local aggregation.

6.8 Sensitivity Studies

This section provides sensitivities studies in terms of cache and input data sizes.

Cache Size Sensitivity. Since CASM entails additional storage (32KB per accelerator in our experimental setup) over the L1 data caches, one might wonder if it were possible to achieve the same performance improvement by simply increasing the capacity of the cores' caches. To this end, we performed a cache-size sensitivity study by tracking the execution time of each application over a range of cache sizes. Figure 6.10 provides a plot obtained by running our MapReduce applications on the baseline CMP, while sweeping the L1 data cache size from 16KB (the original baseline size) to 64KB. The plot shows that the largest performance gain is only 26%, corresponding to the largest L1 data cache considered, while running the *pvc* benchmark. Note that this solution point entails more storage than the total of our baseline L1 cache and the two SPMs embedded in the CASM accelerator. In contrast, with the addition of CASM, we can achieve an average of 4x speedup with less storage. Finally, we carried out a similar analysis for L2 caches. In this case, we swept the size of the total shared L2 cache from 8MB (128KB per core) to 32MB (512KB per core) and found that the largest performance gain was only 32% at 32MB when running *wc*.

Table 6.1: Collision rates over a range of dataset and SPM sizes at CASM's local and home SPMs. At 64KB, CASM achieves equal or lower collision rates with large (30-100GB) datasets than with the medium (0.1-1GB) ones at 16KB. Workloads not reported have a 0% collision rate for all datasets and SPMs sizes. *min-max* and *avg* have the same access patterns, leading to the same collisions.

dataset size/	wc		mm/avg		pvc		sc	
SPM size	local	home	local	home	local	home	local	home
0.1-1GB / 16KB	24.14	3.51	46.35	1.92	3.99	0.00	91.44	75.31
30-100GB / 16KB	30.00	5.28	47.03	**1.63**	5.24	**0.00**	94.98	**74.51**
30-100GB / 32KB	24.16	3.96	**45.83**	**0.43**	**1.04**	**0.00**	93.47	**69.57**
30-100GB / 64KB	**18.71**	**3.01**	**41.25**	**0.10**	**0.63**	**0.00**	**91.19**	**65.17**

Scalability to Large Datasets. This study estimates the performance of CASM on the large datasets discussed earlier in this section, ranging from 30 to 100GB. Since we could not carry out an accurate simulation, due to the limited performance of gem5, we used the collision rate on the local and home SPMs as a proxy for performance. Indeed, kv-pair collisions in local or home SPMs are the lead cause for contention in the interconnect; in addition, collisions in the home SPMs cause spilling to off-chip memory. As discussed above, off-chip memory accesses are the main source of performance benefits provided by CASM. In contrast, if an application generated no collisions, CASM's execution would almost completely overlap that of the map stage. Table 6.1 reports the collision rates for our applications on all datasets we considered for them, both medium (0.1-1GB) and large (30-100GB). We compute a collision rate as the average number of kv-pairs collisions at the local/home SPM, divided by the total number kv-pairs. We only report our findings for four applications because all others have a collision rate of 0%, irrespective of the dataset size. The first row of the table reports the collision rate of the medium dataset – the one simulated in gem5 and reported in Figure 6.7. The other rows report the collision rate for the large datasets, over a range of SPM sizes. For the large datasets, we report in bold the SPM sizes that lead to a collision rate below that of our original setup with the medium dataset. Assuming that the baseline CMP maps large datasets as efficiently as medium datasets – a highly conservative assumption – CASM should achieve the same or better performance speedup as reported in Figure 6.7, when the collision rates at both SPMs fall below that of the medium datasets. Note that all workloads reach this low collision rate with 64KB SPMs. Note also that our analysis is based on a very conservative assumption that the baseline CMP would be able to provide the same performance for medium and large datasets, which is unlikely, as large datasets create further pressure on the baseline-CMP's memory subsystem, providing additional benefits for CASM.

6.9 Area, Power, and Energy

This section presents area and power overheads, as well as energy benefits provided by CASM, as compared to CMP-only baseline.

We synthesized CASM's logic in IBM 45nm technology. We then setup Cacti with the same technology node to model the SPMs. We used McPAT to compute the same metrics for the other CMP node's components: cores, caches and interconnect. Our findings indicate that CASM accounts for approximately 6% of a CMP node's area and 1% of its peak power consumption. We derived energy consumption from average dynamic and leakage power and total execution time, using the tools detailed earlier and performance stats from gem5/Garnet. The performance speedup of CASM provides energy savings up to 11x, corresponding to *wc*. Since the power overhead of CASM is minimal (1%), it is natural to expect that high performance benefits translate into high energy savings.

6.10 Bibliography

This section provides a summary of related works, focusing on solutions that optimize the execution of data-intensive applications, by leveraging CMPs, GPUs, FPGAs, and domain-specific architectures.

Many MapReduce solutions targeting CMPs, GPUs and vector platforms have recently been proposed [97, 38, 107, 114, 71, 46, 49, 55]. CPU solutions rely on a traditional memory subsystem, which we found to be inefficient for most applications. Other architectures are optimized for compute-intensive applications, whereas, most MapReduce applications require large and irregularly accessed data structures, generating high off- and on-chip traffic. Several other works accelerate MapReduce using FPGA platforms: [100, 61], but their architecture does not allow for scaling to large CMP systems. Some recent works share some traits with CASM. For instance, [65] improves hash-key lookups for database applications [54, 18], and optimizes off- and on-chip communication for graph analytics. Other proposals, even if they state different goals, have proposed some architectural features that share some similarity with CASM: the software-defined caches of [32] partially resemble home scratchpads, [77] proposes a set of fine-grained accelerators, [51] provides a near-data processing architecture, [19] offers computation in cache, and [89] designs a novel reconfigurable architecture that is more flexible than domain-specific architectures. All of these solutions, while including valuable contributions, do not offer the full range of optimized design aspecs of CASM, and many have a different set of goals all together.

6.11 Summary

This chapter discussed CASM, a novel hardware architecture that accelerates MapReduce applications, leveraging the skewed access frequency of key-value pairs. CASM augments each core of a CMP with scratchpad memories. The scratchpads store frequently-occurring key-value pairs, which are often responsible for the majority of the access. It also provides processing data near storage functionality by co-locating each scratchpad with a lightweight compute engine. By doing so, CASM manages to process the majority of the key-value pairs in scratchpads. The system is highly scalable with the number of cores, and provides over a 4x speedup on average over a CMP-based software solution, as we evaluated over a wide range of input dataset, up to 100GB in size. The cost of CASM is a 6% overhead in silicon area and a 1% in peak power.

Overall in this chapter, we showed that our techniques to boost the performance and lower the energy of application domains that are data-transfer bounded are applicable and effective even in domains beyond graphs. For instance, as we discussed above, MapReduce applications access their key-value data with random-access patterns, which causes high on- and off-chip communication, which, in turn, leads to performance and energy inefficiencies. We addressed these inefficiencies by employing our two strategies discussed in Section 1.4. We adopted our first strategy of specializing memory architecture, where we utilized scratchpads to store frequently-occurring key-value pairs, which are often responsible for the majority of the off-chip memory accesses. We then adopted our second strategy of processing data near storage, by co-locating each scratchpad with a lightweight compute engine. This second strategy reduced the interconnect traffic that otherwise would take place between cores and remote on-chip storage units.

With this chapter, we conclude the discussion of the technical contributions of this dissertation. The next chapter provides the conclusions of this dissertation, followed by some future research directions that can potentially improve this existing study.

CHAPTER 7

Conclusions

In this chapter, we provide a summary of the contributions of this dissertation, followed by future potential research directions. Finally, we conclude this chapter with a brief summary of the overall dissertation.

7.1 Summary of the Contributions

This dissertation provides several solutions that address the primary limitations of graph analytics in conventional computing system architectures, specifically in terms of performance and energy efficiencies affected by unstructured and unpredictable data access patterns. We do this by relying on three strategies discussed in Section 1.4. These strategies are: 1) **specializing the memory structure** to improve the efficacy of small on-chip storage, and thus accommodating the characteristics of graph applications, 2) **processing data near its storage location, when possible**, to reduce interconnect traffic, and 3) **coalescing data messages** in the network, which should also reduce the amount of interconnect traffic. Based on these strategies, we developed several solutions, discussed below.

Processing in on-chip memory architecture for graph analytics. In Chapter 3, we discussed our first solution, OMEGA [18], based on our first strategy of specializing memory architecture. As discussed in Section 1.4, many graphs follow a power-law distribution; hence, some vertices are accessed more frequently than others. Based on that characteristic, we developed specialized on-chip storage that houses this frequently-accessed data. To make our solution scalable, we distribute the specialized on-chip storage by co-locating one storage unit with each core. This technique reduces the traffic between the processor and off-chip memory. In addition, in aiming to follow our second strategy, we augment each specialized on-chip storage unit with a specialized compute engine that carries out the atomic operations affecting the vertices in the local on-chip storage, and which have been offloaded from the cores. The specialized compute engines help reduce both interconnect traffic and computational demands on the cores.

On-path message coalescing for graph analytics. Although the OMEGA solution described above manages to offload atomic operations from cores to specialized compute engines, the messages for these operations must be transferred via the interconnect. Due to the aforementioned power-law characteristics; most often these operations target the same vertex and can therefore be coalesced in the network. Our MessageFusion solution, detailed in Chapter 4, adopts our third strategy of coalescing data messages in the network. Specifically, it augments each router in the network with specialized hardware to carry out such message coalescing. It boosts the chances of coalescing by reorganizing the work at each compute node so that messages to a same vertex are generated in close sequence before moving to the next one. By doing so, it significantly reduces network traffic, and thus maximizes performance and energy efficiencies.

Hybrid processing in on/off memory architecture for graph analytics. The two solutions described above seek to optimize the execution of atomic operations on frequently-accessed data. However, the computation of atomic operations on infrequently-accessed data still takes place through conventional mechanisms, that is, transferring the data from off-chip memory to the relevant processor core and computing updates in the core. Our Centaur solution, discussed in Chapter 5, relies on the second strategy of processing data near storage so to improve precisely the computation of infrequently-accessed data: it optimizes the computation of atomic operations on infrequently-accessed data as well. In particular, it augments off-chip memory units with specialized compute engines, which are tasked with executing atomic operations related to infrequently-accessed data. The computation by these specialized engines reduces the traffic between cores and off-chip memory and, at the same time, further frees up the cores from executing atomic operations.

Optimizing memory architecture beyond graph analytics. Finally, we applied the strategies discussed thus far to MapReduce applications, to demonstrate their applicability to multiple data-intensive application domains. In this regard, Chapter 6 detailed our work, CASM, which optimizes the execution of MapReduce, by augmenting each core in a chip multiprocessor with specialized on-chip storage units to house performance-critical data, thus reducing traffic between cores and off-chip memory. In addition, each on-chip storage unit is co-located with a lightweight compute engine that processes data in situ, limiting network traffic.

7.2 Future Directions

The contributions of this dissertation are augmented by the future research directions it illuminates, as discussed below.

Dynamic graphs. Dynamic graphs, in which the connectivity of vertices changes overtime, are increasingly becoming important in a wide range of areas, such as social networks [74]. Although this dissertation focuses on static graphs (*i.e.*, graphs with a static topology), future work might find the techniques discussed in this dissertation useful for dynamic graphs as well. New work might explore, for example, efficient techniques that maximize the on-chip memory utilization while the structure of the graph morphs over time.

Locked cache vs scratchpad. Locking cache lines allows programmers to load a cache line and disable its replacement policy [95]. This technique could be extended to capture frequently-accessed data on conventional caches by locking their corresponding cache lines, thus, minimizing the cost associated with hardware specialization, as compared to the solutions discussed in this dissertation.

Processing near on-chip memory using the general-purpose cores. In this dissertation, each scratchpad is co-located with a specialized compute unit to process the vertex-update operations for vertices stored in that scratchpad. In place of the specialized compute unit, the general-purpose core that is local to the scratchpad can be utilized for this purpose. This approach could potentially reduce the overhead of hardware specialization while maintaining an attractive performance cost.

Further memory architecture optimization using word-level granularity access. In this dissertation, the vertex data stored in the off-chip memory is accessed at cache-line granularity, and these accesses most often exhibit low spatial-locality, potentially creating a performance bottleneck. One future research direction is to evaluate the benefits of accessing the off-chip memory at word-level granularity instead of cache-line granularity.

Hybrid processing in on-/off-chip memory architecture for other applications domains. Chapter 5 discusses our work, Centaur, which targets graph analytics. Centaur's hybrid processing in on- and off-chip memory technique does not modify the on-chip memory's SRAM-cell circuitry nor the off-chip memory's DRAM-cell circuitry. Efficient hardware solutions for compute-intensive applications, such as machine learning, often require re-purposing the SRAM- or DRAM- cell circuitry as computational blocks. In this regard, future work might find it useful to employ a Centaur-like hybrid processing in on- and off-chip memory architectures that involve the aforementioned modifications.

From single-node platforms to distributed-system platforms. Finally, the techniques discussed in this dissertation target single-node platforms. For further scalability, future work might find it useful to extend them to distributed-system platforms.

7.3 Summary

In summary, this dissertation addresses the performance limitations of existing computing systems so to enable emerging graph analytics. To achieve this goal, we identified three key strategies: 1) specializing memory architecture, 2) processing data near its storage, and 3) message coalescing in the network. Based on these strategies, the dissertation presents several solutions: OMEGA, which employs specialized on-chip storage units with co-located specialized compute engines to accelerate the computation; MessageFusion, which coalesces messages in the interconnect; and Centaur, which optimizes the processing of infrequently-accessed data. Overall, the solutions discussed in this dissertation provide a 2× performance improvement, with negligible silicon area overhead, across a wide range of graph-based applications. We also demonstrate the benefits of adapting our strategies to other data-intensive domains by exploring a solution in MapReduce, where our strategies achieve a 4× performance speedup, once again with negligible area and power overheads.

Finally, we would like to conclude by noting that the limited available works in the memory-access specialization areas made our journey of working on this dissertation challenging. We hope that this dissertation will be a valuable resource to future researchers working on related areas, enabling them to focus more on developing their own novel techniques.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Intel Corporation. `https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html`, 2017.

[2] Project Gutenberg. `https:/www.gutenberg.org`, 2017.

[3] 9th DIMACS Implementation Challenge . `http://www.dis.uniroma1.it/challenge9/`, May 2018.

[4] Cache Pseudo-Locking. `https://lwn.net/Articles/738968/`, Dec. 2018.

[5] Ligra GitHub. `https://github.com/jshun/ligra`, May 2018.

[6] WebGraph. `http://webgraph.di.unimi.it/`, May 2018.

[7] Facebook Is Changing News Feed (Again) to Stop Fake News. `https://www.sicara.ai/blog/2019-01-09-fraud-detection-personalized-page-rank`, 2020.

[8] Graph Databases Lie at the Heart of $7TN Self-driving Car Opportunity. `https://www.information-age.com/graph-databases-heart-self-driving-car-opportunity-123468309/`, 2020.

[9] How to Perform Fraud Detection with Personalized Page Rank. `https://www.sicara.ai/blog/2019-01-09-fraud-detection-personalized-page-rank`, 2020.

[10] Is technology the key to combatting fake news? `https://www.itproportal.com/features/is-technology-the-key-to-combatting-fake-news/`, 2020.

[11] Neo4j COVID-19 Contact Tracing. `https://neo4j.com/blog/this-week-in-neo4j-covid-19-contact-tracing-de-duplicating-the-bbc-goodfood-graph-stored-procedures-in-neo4j-4-0-sars-cov-2-taxonomy/`, 2020.

[12] POWER9. `https://en.wikipedia.org/wiki/POWER9`, 2020.

[13] Real Time Fraud Detection: Next Generation Graph Analytics? `https://medium.com/@erez.ravid/real-time-fraud-detection-next-generation-graph-analytics-b6cefc96d1bb`, 2020.

[14] Seven Bridges of Knigsberg. `https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg`, 2020.

[15] Shortest Path Problem. `https://en.wikipedia.org/wiki/Shortest_path_problem/`, 2020.

[16] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. Dreslinski, D. Blaauw, and T. Mudge. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *Proceedings of the International Symposium on High Performance*

Computer Architecture (HPCA), 2013.

[17] A. Addisie and V. Bertacco. Collaborative accelerators for in-memory mapreduce on scale-up machines. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2019.

[18] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco. Heterogeneous memory subsystem for natural graph analytics. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2018.

[19] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[20] N. Agarwal, L.-S. Peh, and N. K. Jha. In-network coherence filtering: snoopy coherence without broadcasts. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009.

[21] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. PUMA: Purdue MapReduce benchmarks suite. Technical Report TR-ECE-12-11, School of Electrical and Computer Engineering, Purdue University, 2012.

[22] M. Ahmad and O. Khan. GPU concurrency choices in graph analytics. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2016.

[23] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[24] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[25] A. Airola, S. Pyysalo, J. Björne, T. Pahikkala, F. Ginter, and T. Salakoski. All-paths graph kernel for protein-protein interaction extraction with evaluation of cross-corpus learning. *BMC Bioinformatics*, 2008.

[26] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the world-wide web. *Nature*, 1999.

[27] J. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Proceedings of the Neural Information Processing Systems (NIPS)*, 2006.

[28] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[29] V. Balaji and B. Lucia. When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2018.

[30] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 1999.

[31] S. Beamer, K. Asanovi, and D. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2015.

[32] N. Beckmann and D. Sanchez. Jigsaw: Scalable software-defined caches. In *Pro-*

ceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013.

[33] L. Bin and G. Yuan. Improvement of tf-idf algorithm based on hadoop framework. In *Proceedings of the International Conference on Computer Application and System Modeling (ICCASM)*, 2012.

[34] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39:1–7, 2011.

[35] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.

[36] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2004.

[37] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[38] C.-T. Chu, S. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Ng. MapReduce for machine learning on multicore. In *Proceedings of the Neural Information Processing Systems (NIPS)*, 2007.

[39] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[40] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting in-network aggregation for big data applications. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[41] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38:640–653, 2018.

[42] X. Deng, Y. Yao, J. Chen, and Y. Lin. Combining breadth-first with depth-first search algorithms for VLSI wire routing. In *Proceedings of the International Conference on Advanced Computer Theory and Engineering (ICACTE)*, 2010.

[43] S. Dhingra, P. S. Dodwad, and M. Madan. Finding strongly connected components in a social network graph. *International Journal of Computer Applications*.

[44] S. Dickey and R. Kenner. Combining switches for the NYU Ultracomputer. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS)*, 1992.

[45] H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan. Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[46] K. Duraisamy, R. Kim, W. Choi, G. Liu, P. Pande, R. Marculescu, and D. Marculescu. Energy efficient MapReduce with VFI-enabled multicore platforms. In *Proceedings of the Design Automation Conference (DAC)*, 2015.

[47] V. Eguiluz, D. Chialvo, G. Cecchi, M. Baliki, and A. Apkarian. Scale-free brain functional networks. *Physical Review Letters*, 94:018102, 2005.

[48] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM Computer Communication Review (CCR)*, 29:251–262, 1999.

[49] W. Fang, B. He, Q. Luo, and N. K. Govindaraju. Mars: Accelerating MapReduce with graphics processors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 22:608–620, 2010.

[50] S. Fortunato. Community detection in graphs. *Physics Reports*, 486:75–174, 2010.

[51] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[52] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences (PNAS)*, 99:7821–7826, 2002.

[53] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[54] T. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[55] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero. Future Vector Microprocessor Extensions for Data Aggregations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[56] J. Heidemann, M. Klier, and F. Probst. Identifying key users in online social networks: A PageRank based approach. 2010.

[57] K. Inoue, K. Kai, and K. Murakami. Dynamically variable line-size cache architecture for merged DRAM/Logic LSIs. *IEICE Transactions on Information and Systems (IEICE T INF SYST)*, 83:1048–1057, 2000.

[58] D.-I. Jeon and K.-S. Chung. CasHMC: A cycle-accurate simulator for hybrid memory cube. *IEEE Computer Architecture Letters (CAL)*, 16:10–13, 2017.

[59] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. Shaw, J. Kim, and W. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[60] S.-W. Jun, A. Wright, S. Zhang, et al. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[61] C. Kachris, G. Sirakoulis, and D. Soudris. A reconfigurable MapReduce accelerator for multi-core all-programmable socs. In *Proc. ISSOC*, 2014.

[62] A. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, 2009.

[63] G. Kim, J. Kim, J. H. Ahn, and J. Kim. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the International Conference on*

Parallel Architectures and Compilation Techniques (PACT), 2013.

[64] A. Klein. *Stream ciphers*. Springer, 2013.

[65] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.

[66] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[67] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, May 2018.

[68] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009.

[69] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26:3077–3089, 2014.

[70] W. Liu, X. Li, T. Liu, and B. Liu. Approximating betweenness centrality to identify key nodes in a weighted urban complex transportation network. *Journal of Advanced Transportation (JAT)*, 2019.

[71] M. Lu, Y. Liang, H. Huynh, Z. Ong, B. He, and R. Goh. MrPhi: An Optimized MapReduce Framework on Intel Xeon Phi Coprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 26:3066–3078, 2014.

[72] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

[73] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2010.

[74] A. McCrabb, E. Winsor, and V. Bertacco. DREDGE: Dynamic Repartitioning during Dynamic Graph Execution. In *Proceedings of the Design Automation Conference (DAC)*, 2019.

[75] M. Meringer. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory (JGT)*, 30:137–146, 1999.

[76] R. Mihalcea and P. Tarau. Textrank: Bringing order into text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2004.

[77] A. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr. Fine-grained accelerators for sparse machine learning workloads. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[78] S. Mittal. A survey of cache bypassing techniques. *Journal of Low Power Electronics and Applications (JLPEA)*, 6:5, 2016.

[79] I. Moreno-Sánchez, F. Font-Clos, Á. Corral, et al. Large-Scale Analysis of Zipfs Law in English Texts. *PLOS ONE*, 11:1–19, 2016.

[80] A. Mukkara, N. Beckmann, and D. Sanchez. PHI: Architectural support for

synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[81] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *Proceedings of the High Performance Computer Architecture (HPCA)*, 2017.

[82] L. Nai, Y. Xia, I. Tanase, H. Kim, and C.-Y. Lin. GraphBIG: understanding graph computing in the context of industrial solutions. In *Proceedings of the High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[83] M. Najork and J. Wiener. Breadth-first crawling yields high-quality pages. In *Proceedings of the International Conference on World Wide Web (WWW)*, 2001.

[84] M. Newman. Detecting community structure in networks. *The European Physical Journal B (EPJ B)*, 38:321–330, 2004.

[85] M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, 2005.

[86] M. Newman et al. Random graphs as models of networks. *Handbook of Graphs and Networks*, 1:35–68, 2003.

[87] D. Nicolaescu, X. Ji, A. Veidenbaum, A. Nicolau, and R. Gupta. Compiler-Directed Cache Line Size Adaptivity. In *Proceedings of the International Workshop on Intelligent Memory Systems (IMS)*, 2000.

[88] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[89] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. Stream-dataflow acceleration. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[90] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[91] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[92] S. Pal, R. Dreslinski, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, and T. Mudge. OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[93] J. T. Pawlowski. Hybrid Memory Cube (HMC). In *Proceedings of the Hot Chips Symposium (HCS)*, 2011.

[94] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. Vijaykumar. Gated-V_{dd}: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2000.

[95] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the Design, Automation & Test in Europe (DATE)*, 2007.

[96] S. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyukto-

sunoglu, A. Davis, and F. Li. NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[97] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[98] L. Roditty and V. Vassilevska. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the Symposium on Theory of Computing (STOC)*, 2013.

[99] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2013.

[100] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce framework on FPGA. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2010.

[101] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. 2001.

[102] J. Shun and G. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[103] S. Singapura, A. Srivastava, R. Kannan, and V. Prasanna. OSCAR: Optimizing SCrAtchpad reuse for graph processing. In *Proceedings of the High Performance Extreme Computing Conference (HPEC)*, 2017.

[104] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. GraphR: Accelerating graph processing using ReRAM. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[105] N. Sundaram, N. Satish, M. Patwary, S. Dulloor, M. Anderson, S. Vadlamudi, D. Das, and P. Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment (PVLDB)*, 8:1214–1225, 2015.

[106] F. Takes and W. Kosters. Determining the diameter of small world networks. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2011.

[107] J. Talbot, R. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for shared-memory systems. In *Proceedings of the International Workshop on MapReduce and Its Applications (MapReduce)*, 2011.

[108] C. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining (SNAM)*, 1:75–81, 2011.

[109] A. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proceedings of the International Conference on Supercomputing (ICS)*, 1999.

[110] J.-P. Wang. Stochastic relaxation on partitions with connected components and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 20:619–636, 1998.

[111] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. Owens. Gunrock: A high-

performance graph processing library on the GPU. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[112] S. Wuchty and E. Almaas. Peeling the yeast protein network. *Proteomics*, 5:444–449, 2005.

[113] A. Yasin. A top-down method for performance analysis and counters architecture. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[114] R. Yoo, A. Romano, and C. Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009.

[115] D. Yoon, M. Jeong, and M. Erez. Adaptive granularity memory systems: A trade-off between storage efficiency and throughput. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[116] J. Zhang and Y. Luo. Degree centrality, betweenness centrality, and closeness centrality in social network. In *Proceedings of the International Conference on Modelling, Simulation and Applied Mathematics (MSAM)*, 2017.

[117] K. Zhang, R. Chen, and H. Chen. NUMA-aware graph-structured analytics. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[118] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. GraphP: Reducing communication for pim-based graph processing with efficient data partition. In *Proceedings of the High Performance Computer Architecture (HPCA)*, 2018.

[119] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *Proceedings of the International Conference on Big Data (Big Data)*, 2017.

[120] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. GraphIt-A High-Performance DSL for Graph Analytics. *arXiv preprint arXiv:1805.00923*, 2018.

[121] Z.-L. Zhao, C.-D. Wang, Y.-Y. Wan, Z.-W. Huang, and J.-H. Lai. Pipeline item-based collaborative filtering based on MapReduce. In *Proceedings of the International Conference on Big Data and Cloud Computing (BDCloud)*, 2015.

[122] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the Annual Technical Conference (ATC)*, 2015.