CODE GENERATION AND OPTIMIZATION OF GRAPH PROGRAMS ON A MANYCORE ARCHITECTURE

EMILY FURST

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2021

Reading Committee:

Mark Oskin, Chair

Saman Amarasinghe

Michael Taylor

Zach Tatlock

Program Authorized to Offer Degree:

Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

CODE GENERATION AND OPTIMIZATION OF GRAPH PROGRAMS ON A MANYCORE ARCHITECTURE

Emily Furst

Chair of the Supervisory Committee:
Associate Professor Mark Oskin
Paul G. Allen School of Computer Science & Engineering

Graph processing is an area of increasing importance in domains such as networking, super computing, public health, and more. However, large scale graph processing presents many challenges. Graph applications are difficult to optimize because they are memory intensive and suffer from poor data locality. Uniform work distribution on parallel implementations is also challenging. Further, heuristics that address these issues are highly sensitive to graph structure. As the scale of graph data continues to grow, new techniques and architectures are needed to continue to achieve high performance. Manycore architectures are flexible compute substrates that provide parallelism through hundreds to thousands of general purpose cores. Past work has demonstrated their robust compute performance on regular compute intensive tasks such as machine-learning and scientific computing.

This dissertation explores the ability of manycore architectures to efficiently compute on graph data structures. In order to study the performance of graph algorithms on a manycore, we design a code generation backend for the GraphIt domain-specific language targeting a representative manycore architecture. We explore the performance of several existing optimizations and present new manycore specific optimizations. We further show how these optimizations improve manycore performance through increased locality, improved load balancing, and leveraging of graph structure.

To my family.

ACKNOWLEDGMENTS

I am fortunate to have had an incredible community of people who have supported me throughout grad school.

First, I must thank my advisor, Mark Oskin. Mark has guided me through many different research directions, provided me with invaluable advice, and helped me grow into the researcher I am today. Further thanks to Michael Taylor for his guidance and advice on the architecture side of things. I am deeply grateful for their encouragement and support. Additional thanks to Bill Howe. Although our work together is not included in this dissertation, his mentorship during the early years of my PhD helped guide me to the space of programmability and code generation frameworks.

This dissertation presents work done by a large number of researchers that I have been lucky enough to collaborate with over the past six years. Special thanks to Dustin Richmond and Max Ruttenberg for their help in developing the code generation framework at the heart of my dissertation work and for answering my many questions about hardware simulation. Thank you to my collaborators on the GraphIt side of things: Saman Amarasinghe, Julian Shun, Daniel Sanchez, Yunming Zhang, Ajay Brahmakshatriya, Victor Ying, Claire Hsu, and Changwan Hong. Further thanks to Scott Davidson, Borna Ehsani, Dai Cheol Jung, Bandhav Veluri, and the rest of my collaborators on the HammerBlade project.

Many thanks to all of my labmates for not only their technical help and advice but also for making grad school fun; especially thanks to my friends Meghan Cowan, Ming Liu, Liang Luo, Amrita Mazumdar, Luis Vega, and Eddie Yang. Thank you to my non-architect friends Kira Goldner, Lucy Lin, and Maarten Sap for the countless espresso room chats and weekend brunches.

Thank you to the entire Allen School staff for making the administrative and operational side of grad school run as smoothly as possible. Many thanks to Elise Dorough especially for all of her support.

Thank you to my undergraduate mentors and advisors: Michael Heroux, Imad Rahal, and Kris Nairn. They introduced me to research and convinced me to pursue graduate studies. Further thanks are due to Lynn Ziegler for encouraging me to take my first Computer Science course and forever changing the direction of my studies.

Thank you to my family: Kate, Terry, and John. Thank you for believing in me and encouraging me as I moved across the country to pursue a PhD. To Beth, Joel, Hannah, and Julia, thank you for your support and for welcoming me so completely into your family.

Finally, immeasurable thanks to my wonderful partner, Andrew C. Schoen; I cannot imagine how different my graduate studies might have been without his encouragement and support.

CONTENTS

1 INTRODUCTION

Sparse graph data structures that capture relationships between elements are ubiquitous. In recent years, the size of graph datasets have exploded [91] with data coming from fields like networking [57], social networks [32, 55, 95], public health [51, 112], science [84, 111], and finance [16]. As graph sizes have grown, so too have the processing requirements. Recommender systems must respond in sub-second latencies [32, 95], and financial algorithms trade on the order of microseconds [72]. These challenges have increased the demand for high-performance graph processing systems.

However, implementing high-performance graph processing systems is not as straightforward as simply increasing the number of compute cores or adding more memory bandwidth. Further, optimizations used in serial applications or implementations with limited parallelism often do not continue to provide performance when parallelism is increased [11]. This has resulted in a wide design space for algorithmic and architectural optimizations to increase the performance of these parallel graph processing systems at scale.

1.1 CHALLENGES FOR PARALLEL GRAPH PROCESSING

Graph processing is notoriously difficult to optimize. Performance depends on optimizing locality within sparse data structures, random-access performance, minimizing high-cost communication, and load balancing between parallel threads [11, 68, 92]. Worse, the structure of graphs varies widely, both between graphs and between iterations within graph algorithms [12, 68]. Heuristic optimizations do not benefit all inputs datasets. Therefore, graph frameworks must be descriptive enough, and processing hardware must be flexible enough to support them.

These challenges have led to the development of many parallel graph processing frameworks: GraphIt [20, 123, 124], GraphLab [66, 67], Grappa [78], GraphChi [56], Green-Marl [45], and Pregel [69]. These frameworks take a user application description and emit parallel code for general-purpose hardware. Traditionally graph frameworks have targeted CPU and GPU architectures, but newer frameworks also target cloud-resident FPGAs [25, 33]. These frameworks allow users to focus on exploring optimizations by abstracting the hardware and parallel infrastructure.

Graph processing frameworks are limited by the flexibility and parallelism provided by the hardware they currently target. Server-class CPUs are widespread and support flexible execution models, but they have limited memory and compute parallelism and poor random-access bandwidth [11]. GPUs are also widespread and expose memory parallelism through banking and multiple memory channels, but are limited by vector-like execution models [96, 117], and have poor random-access bandwidth [1]. FPGAs are inherently more flexible than either architecture, but are difficult to optimize when a single recompilation can take hours.

Manycore architectures are composed of hundreds to thousands of efficient general-purpose cores to form a flexible parallel compute fabric. Past manycore architectures [5, 43, 88] have been limited by available memory bandwidth and parallelism [65]. We employ a manycore design connected to High Bandwidth Memory (HBM) [47, 49] to overcome this issue. In my dissertation, I will show how this manycore architecture can be leveraged to provide high performance on a variety of graph processing applications.

1.2 THESIS OVERVIEW

This dissertation presents a code generator for graph programs targeting a representative manycore architecture, the HammerBlade manycore. Within this code generator, I implement and evaluate manycore specific optimizations to improve graph processing performance. While graph processing frameworks and optimizations on parallel systems is an active area of research, this work targets a novel manycore architecture. Further, the optimizations for this architecture expand the space and provide a new class of techniques for graph applications on this emerging architecture.

In this work, I focus on two different aspects of the problem of efficiently utilizing the HammerBlade manycore for graph processing: ease of programming and manycore-specific optimizations. Reasoning about the implementation of a graph application and the details of the underlying architecture can be difficult especially as techniques for obtaining performance on architectures become increasingly complex. I address this by building a code generator that allows the user to reason about the graph application and optimizations at a high level without requiring knowledge of the underlying manycore architecture. I use observations about the HammerBlade architecture to implement existing graph processing optimizations and introduce several manycore specific optimizations that build off of the observation that most graph applications are memory bound.

1.3 ORGANIZATION

I begin this dissertation by providing background information on graph processing and introduce the algorithms and graphs that are used for evaluation. I also give an overview of the GraphIt DSL and the HammerBlade manycore architecture that I target in Chapter 2.

In Chapter 3, I present the set of optimizations that we implement and explore in this work. This includes several existing graph processing optimizations as well as the HammerBlade manycore-specific optimizations introduced in this dissertation. These optimizations target things such as load balancing, graph traversal, and the HammerBlade memory system.

I then introduce the code generation framework that I implemented as a backend to the GraphIt DSL in Chapter 4. This chapter describes how optimizations are applied to programs and what the resulting generated manycore code includes.

Next, Chapter 5 presents an evaluation of the code generation framework and the set of optimizations introduced in Chapters 3 and 4. We evaluate the performance of each optimization on a variety of input graphs and algorithms. Before concluding, I contextualize my work within the space of graph processing frameworks and optimizations in Chapter 6. I then conclude and provide some insights into areas of future work in Chapter 7.

2 BACKGROUND

In this chapter, we provide background on graph processing including descriptions of the algorithms and input graphs used for evaluation in this dissertation. We then describe the GraphIt DSL which allows programmers to describe graph algorithms separate from their scheduling optimizations. Finally, we provide background on manycore architectures and describe the representative manycore that we target with our code generator.

2.1 GRAPH DEFINITIONS AND PRELIMINARIES

The graph abstraction stores data in a way that maintains connections and relationships between data objects. A graph $g(V, E)$ is defined as a set of vertices V and a set of edges between vertices E. An edge connecting vertices u and v would be referred to as (u, v). An edge can also have a weight associated with it; in this case, an edge would be defined (u, v, w) where w is the weight associated with the edge connecting vertices u and v. Edges can be directed or undirected; an undirected edge can also be thought of as being bidirectional. The number of edges connected to a vertex is a vertex's degree. Graphs are often stored in compressed sparse row (CSR) format. We discuss this storage format in more detail in Chapter 2.5.

2.2 GRAPH TOPOLOGY

Graph application performance is dependent on both the algorithm and the structure of the input graph. In this work we consider two broad classes of graph topologies; scale-free or social network graphs and planar or road graphs. A scale-free or social network graph has a degree distribution that follows a power-law, at least asymptotically [8]. Most social network graphs have a small diameter and exhibit the small-world property. In a small-world graph, most vertices are not connected to one another, but the majority of vertices can be reached from every other vertex in a small number of hops [116]. Planar or road graphs have a large diameter and low average degree. The sparsity of a graph is determined by the average degree of the graph. All of the graphs we examine in this dissertation are relatively sparse, but the planar graphs we study are the most sparse. Both scale-free and planar graphs present their own unique challenges for optimization.

2.3 GRAPHS USED IN THIS THESIS

In this work, we use a diverse set of graphs for evaluation. These graphs and their properties are listed in Table 2.1. A graph's topology greatly impacts a workload's characteristics and performance, so it is important to perform evaluation on a diverse set of graphs [11]. We use a mix of real-world and synthetic graphs in our evaluation and primarily focus on social network topologies.

The real-world graphs we use come from a variety of sources. We examine two real-world social network graphs: Pokec and LiveJournal which represent the links between users in their online communities. We also consider the Hollywood graph which considers professional relationships. It links together actors that have performed together. These are all representative of a scale free or social network topology as they all have a low effective diameter and power-law degree distribution. In contrast, we also use three road network graphs in our evaluations. These are mesh topologies with low average degree and high diameter.

In addition to these real-world graphs, we fill out our set of graphs with four synthetic graphs. Like in the Graph500 benchmark, we generate several Kronecker graphs of varying size using the Kronecker generator [76]. These graphs model a social network and follow a power-law degree distribution.

Name	**Description**	**Vertices**	**Edges**	**Degree**
Kron18	Kronecker generated [58, 60]	262,144	4,194,304	16
Kron19	Kronecker generated [58, 60]	524,288	8,388,608	16
Kron20	Kronecker generated [58, 60]	1,048,576	16,777,216	16
Kron22	Kronecker generated [58, 60]	4,194,304	67,108,864	16
Pokec	social network [59]	1,632,803	30,622,564	18.8
LiveJournal	social network [29, 74]	4,847,571	85,702,474	17.6
Hollywood	movie collaborations [17, 18, 29]	1,139,905	112,751,422	98.9
RoadCA	CA road network [29]	1,971,281	5,533,214	2.8
RoadCentral	Central road network [29]	14,081,816	33,866,826	2.4
RoadUSA	USA road network [30]	23,947,347	57,708,624	2.4

Table 2.1: List of graphs used in this dissertation and their properties.All of the graphs come from real-world data except the three Kronecker graphs. Throughout our evaluation, we list the subsets of these graphs that are being evaluated.

2.4 ALGORITHMS USED IN THIS THESIS

In this dissertation we explore a variety of graph processing algorithms. We select them based on their popularity in graph-processing evaluations [10] and for the different behaviors that they exhibit. The algorithms we examine can be classified as traversal-centric or compute-centric algorithms. Traversal-centric or frontier-based algorithms start from a given source vertex and perform computation on vertices by traversing outwards from the source vertex. Compute-centric algorithms operate on the entire graph in parallel and tend to iteratively apply updates until the algorithm converges. Of the algorithms studied in this work, BFS, SSSP, and BC are traversal-centric algorithms. PageRank and CC are compute centric.

BREADTH FIRST SEARCH (BFS)

BFS is a building block of many graph algorithms. It is not an algorithm but really a graph traversal order. BFS visits every vertex at a given depth of the graph before moving on to the next depth level. There has been considerable work done to accelerate BFS and increase the computational work through algorithmic and data structure modification [4, 12, 22, 46, 121] We turn it into an algorithm by discovering and tracking the parent vertex ID of each vertex reachable from a given source vertex.

SINGLE SOURCE SHORTEST PATH (SSSP)

SSSP is an algorithm that builds off of BFS to compute the distances of the shortest paths from a given source vertex to every other reachable vertex in the graph. This is usually performed on a weighted graph, so the weights of edges are used in calculating the distance of a path. We only consider graphs with non-negative edge weights in this work. We examine two different SSSP algorithms, frontier-based Bellman-Ford and Delta-stepping.

Frontier-based Bellman-Ford trades off repeated accesses to edges for increased parallelism. It uses relaxation where approximations of the distances to each vertex are replaced by shorter distances until the algorithm converges on the correct solution. Unlike Dijkstra's, the classical SSSP algorithm, there is no notion of priority in Bellman-Ford and all edges active in the frontier are relaxed in every iteration.

The delta-stepping algorithm [73] increases parallelism by using a notion of relaxed priority. Delta-stepping coarsely sorts the vertices of the graph by distance into buckets of width Δ. This allows for all vertices in a bucket to be processed in parallel. Like Bellman-Ford, this does result in some vertices being processed multiple times, but the frequency with which this occurs can be reduced by reducing the value of Δ. If $\Delta = 1$, the algorithm effectively becomes Dijkstra's, and if $\Delta = \inf$, the algorithm behaves like Bellman-Ford.

PAGERANK (PR)

PR calculates the importance or "popularity" of each vertex in a graph and was originally developed to sort web search results [82]. It calculates the popularity of a vertex v by considering both the number of vertices that point to v and the importance of those vertices that point to v. PR iteratively updates the PageRank score for each vertex in the graph until the scores converge within some specified tolerance. It is a topology-driven algorithm where all the edges are traversed in each iteration. PR also exhibits massive parallelism in each round. There has been considerable work on optimizing PageRank and finding ways to improve the convergence rate [15, 54, 67, 97]

CONNECTED COMPONENTS (CC)

The Connected Components algorithm labels all of the components in a graph. A connected component is a subgraph in which all vertices are connected to each other. If an edge exists between two vertices, they are connected. In a directed graph, connections between vertices can be asymmetric. This means that components of a directed graph can be strongly or weakly connected. In this work, we only consider undirected graphs and do not need to consider asymmetry of connections. The CC algorithm labels vertices so that all vertices in the same component have the same label. Like PR, CC is also topology-driven and traverses every edge in each iteration.

BETWEENNESS CENTRALITY (BC)

BC is another algorithm that attempts to measure the importance of the vertices in a graph. It calculates a score for each vertex that measures the fraction of shortest paths that pass through the vertex. This can be computationally expensive as the algorithm needs to compute the shortest paths for all pairs of vertices in the graph. This is often done by computing the All Pairs Shortest Path algorithm which executes SSSP for every vertex in the graph as a source vertex. Calculating all of the shortest paths can be both compute and memory intensive. The Brandes algorithm reduces the memory requirements by compacting the critical information from a SSSP execution into a single variable [21]. We also compute BC on an unweighted graph in this work, which allows us to use BFS traversals to compute the shortest paths.

2.5 GRAPHIT: A GRAPH PROCESSING DOMAIN-SPECIFIC LANGUAGE

GraphIt is a domain-specific language for graph processing applications [20, 123, 124]. Like Halide [87], GraphIt separates the description of the algorithm from the scheduling of the computation. A separate algorithm language and scheduling language are used to specify graph programs in GraphIt. This separation of description and schedule allows users to express computation more flexibly.

The Unified GraphIt Compiler Framework provides functionality for adding new backends by further decoupling the hardware-independent transformations from the hardware-dependent compiler passes [19]. The design of Unified GraphIt Compiler Framework makes it easy to write and compose optimizations that make use of each backend architecture's unique features. Figure 2.1 demonstrates the overall compilation flow with various analyses and lowering passes to generate GraphIR, GraphIt's graph specific intermediate representation. GraphIR is an in-memory representation of a program that allows optimiza-

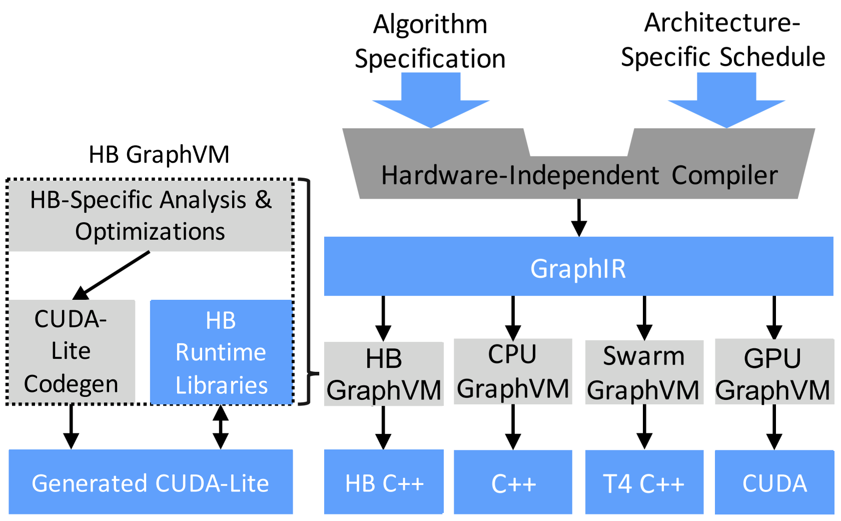


Figure 2.1: The Unified GraphIt Compiler Framework (UGC). GraphIR decouples the hardware-independent part of the compiler from the hardware-dependent GraphVMs. Grey blocks denote parts of the compiler, and blue blocks denote code (inputs, intermediates, libraries, or generated).

tions through IR-to-IR transformations before final code generation. GraphIR is lowered into code for different architectures using an architecture-specific Graph Virtual Machine (GraphVM), which performs the hardware-specific transformations and code generation.

Listing 2.1 shows an example GraphIt program implementing Breadth-First Search (BFS) along with a HammerBlade-specific schedule.

```
const edges : edgeset{Edge}(Vertex,Vertex);
var frontier : vertexset{Vertex} = new vertexset{Vertex}(0);
const parent : vector{Vertex}(int) = -1;

func updateEdge(src : Vertex, dst : Vertex)
    parent[dst] = src;
end

func toFilter(v : Vertex) -> output : bool
    output = parent[v] == -1;
end

func main()
    // frontier setup
    while (frontier.getVertexSetSize() != 0)
      #s1# frontier =
           edges.from(frontier).to(toFilter).applyModified(updateEdge,parent,true);
    end
end
```

```
   schedule:
21   SimpleHBSchedule sched1;
       sched1.configDirection(PULL);
     program->applyHBSchedule("s1", sched1);
```

Listing 2.1: GraphIt code for Breadth-First Search (BFS) with a HammerBlade manycore schedule.

2.5.1 *Graph Representation*

Graphs are represented by `edgeset` and `vertexset` structures. The edges between nodes are stored as an `edgeset`, and any vertex specific data is stored as a `vertexset`. Line 1 and Line 2 of Listing 2.1 demonstrate the declaration of the `edgeset` variable `edges` and `vertexset` variable `frontier`. Any data associated with elements in an `edgeset` or `vertexset` is stored as a `vector`. Line 3 of Listing 2.1 shows the declaration of a `vector` variable, `parent`, that contains data associated with the vertices in the graph.

GraphIt uses Compressed Sparse Row (CSR) graph format for these data structures, illustrated in Figure 2.2. CSR format represents a graph with two arrays: a vertex list and an edge list. The edge list contains the destination vertex for each edge in the graph and contains $|E|$ elements where E is the set of edges in the graph. In the case of a weighted graph, the edge list is stored as an array of tuples, with each element in the array containing the destination vertex id and the weight of the edge. The vertex list contains $|V|$ elements where V is the set of vertices in the graph. Each element in the vertex list is an index into the edge list array. This index corresponds to the start of the edges for which that vertex is the source vertex. The number of edges for a vertex v, or its degree, is $vertex\_list[v+1] - vertex\_list[v]$. The generated code operates on these arrays to perform computation.

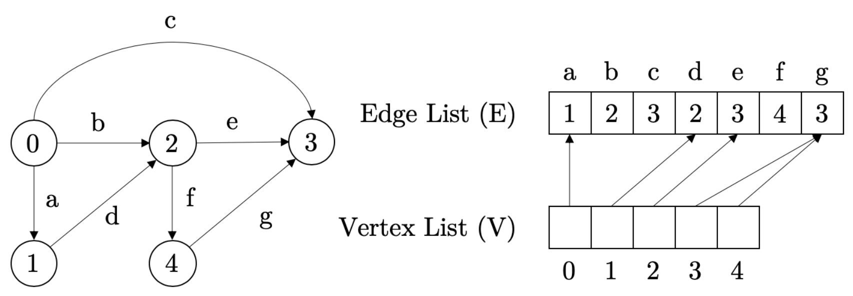


Figure 2.2: The CSR graph format that we use to store graph data on the manycore. For weighted graphs, the weight is stored with each element in the edge array.

2.5.2 *Algorithm Representation*

An algorithm in GraphIt is written as operations on `edgeset` and `vertexset` datatypes. GraphIt provides several operators that perform computation on these types, but the most common are the `filter` operator and **`apply`** operator. The `filter` operator takes a function and a set, applies the function to each element in the set, and returns a set of elements where the function returned true. The **`apply`** operator takes a function and set as input and applies the function to each element in the set. GraphIt also provides other built-in operators like `applyModified`, that takes a function and a set, applies the function to each element in the set, and returns a vertexset that contains vertices that were updated by the input function. This operator is used to construct frontiers in iterative graph algorithms. Listing 2.1 demonstrates how these operators can be used to write BFS.

Line 16 of Listing 2.1 demonstrates two uses of the `filter` operator and one use of the `applyModified` operator. The first filter application (**`from`**`(frontier)`) selects only edges with source vertices that are in the current frontier. The second filter application (**`to`**`(toFilter)`) uses the `toFilter` function, defined on line 9, to select edges with destination vertices that haven't been previously visited. Next, an `applyModified` operator is applied to this filtered set of edges. This operator computes the edge traversal of BFS and returns a vertexset containing the vertices that were updated by the `updateEdge` function defined on line 5. This `vertexset` becomes the new frontier for the next iteration of the while-loop.

2.5.3 *Schedule Representation*

One of the key benefits of GraphIt is that it decouples the algorithm description from the schedule which allows developers to iterate through different scheduling optimizations. To support a diverse set of GraphVMs each with their own set of optimizations, a new scheduling language is written for each hardware target. UGC provides an abstract interface that provides virtual functions for all of the information that the hardware-independent compiler needs. New scheduling object classes are implemented for each GraphVM by inheriting from this abstract interface. These new classes have members and functions to configure various scheduling options specific to optimizations supported for their GraphVMs. These classes implement the virtual functions from the abstract interface to provide the hardware-independent part of UGC with the information that it needs.

A wide range of scheduling options can be implemented in each GraphVM, i.e., by specifying traversal direction, by specifying the level of parallelism, or by using load balancing techniques for parallel operators. Labels are used to indicate which operators in the algorithm description the scheduling optimizations should be applied to. Further, the architecture-specific schedule tells the compiler to generate code for the target device. The schedule description always follows the algorithm description in GraphIt.

The schedule for BFS starts with the `schedule` declaration on Line 20 of Listing 2.1. Line 21 instantiates a HammerBlade scheduling object indicating that the code generator should generate code for the target manycore. Line 22 specifies that the traversal schedule for label `#s1#` should be "Dense Pull". This optimizations that the HammerBlade GraphVM and scheduling language support are described in Chapter 3.

2.5.4 *Code Generation*

Unified GraphIt Compiler Framework takes an algorithm and schedule and generates device-specific code. The compiler first parses the algorithm and schedule using the frontend to produce the GraphIR. UGC adapts the domain-specific transformations from the GraphIt DSL compiler, such as dependence analysis to insert atomics in the user-defined functions (UDFs), liveness analysis to find frontier memory reuse opportunities, and other transformations to UDFs for traversal direction, parallelism, and data structure choices. These passes also add metadata to the GraphIR for the GraphVMs to use during code generation. After these hardware-independent transformations are completed, the program is passed to the GraphVM for hardware-dependent analysis and code generation. Once the hardware-dependent passes have finished, the GraphVM produces code for the target device and host (if necessary). Chapter 4 describes how our HammerBlade GraphVM was implemented.

2.6 MANYCORE ARCHITECTURES

Manycore architectures provide thread-level parallelism and flexibility with hundreds to thousands of general-purpose cores [5, 28, 43, 88, 102]. Cores are arranged in two-dimensional arrays and interconnected with mesh-style networks for communication. This network of cores is surrounded by multiple channels of memory to provide sufficient bandwidth and parallelism to sustain computation. Cores within the architecture communicate explicitly through memory [28] or message passing [43], implicitly through coherence protocols [88], or both using inter-core result networks [102]. Communication allows cores to cooperate and solve large parallel tasks. The quantity of cores and diverse communication patterns means that manycore architectures provide a flexible parallel computation fabric that can be tailored to fit application requirements or the structure of graph input data [68].

In this section, I describe the HammerBlade manycore, a representative manycore architecture that we use to evaluate our code generator. The cores are tiny, performance-optimized scalar cores that implement a RISC-V instruction set. Each core has a software-managed scratchpad memory for low-latency storage and inter-thread communication. Data cache lines do not move between cores, eliminating coherence overhead and false sharing. The memory system is designed to expose memory parallelism and bandwidth to service memory requests from hundreds of concurrent threads. This architecture is

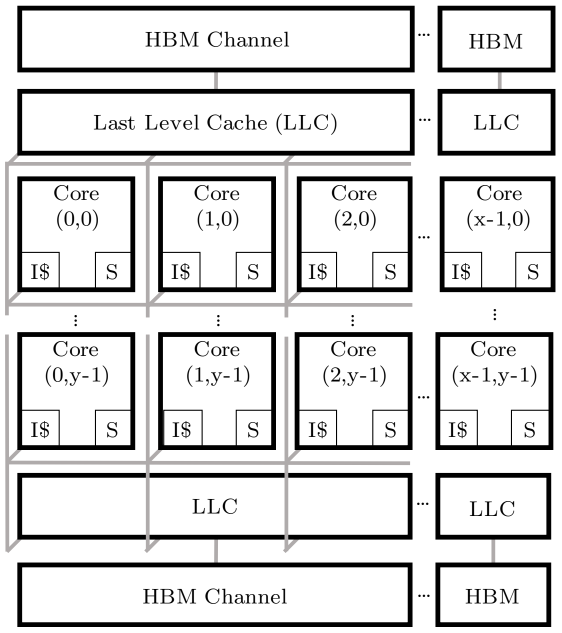


Figure 2.3: Block diagram of the general manycore architecture targeted in this work. A 2D-Mesh Network-on-Chip connects general-purpose computation cores and Last Level Caches (LLC). Each LLC connects to a single independent High-Bandwidth Memory (HBM) channel.

emblematic of Single Program Multiple Data (SPMD) parallel machines where collections of cores are loaded with the same program, but each core computes on different input data.

This section provides a overview of the target manycore architecture depicted in Figure 2.3. The architecture is composed of efficient general-purpose processor cores, Last Level Caches (LLCs), a 2D-Mesh Network on Chip, and High-Bandwidth Memory (HBM) channels.

2.6.1 *Network-On-Chip*

Communication is provided by a 2D-Mesh Network-on-Chip (NoC) that interconnects Last Level Caches and general-purpose cores. The NoC supports point-to-point communication between endpoints using shared memory. All addressable endpoints within the network are assigned a unique global address range. This provides a PGAS-like (Partitioned Global Address Space) memory model for execution.

2.6.2 *Memory System*

The manycore's memory system is designed to provide the high-bandwidth memory-level parallelism required by manycore architectures. The memory system has four hierarchy levels: High-Bandwidth Memory (HBM), Last Level Cache (LLC), core-remote scratchpad (S), and core-local scratchpad (S). Each level is designed to exploit memory parallelism and exposes a trade-off between latency, capacity, as shown in Figure 2.3.

Manycore architectures require many high-bandwidth and independent memory channels to supply data for computation, so we use second-generation High Bandwidth Memory (HBM, or HBM2). HBM provides two sources of memory-level parallelism: First, HBM provides channel-level parallelism with 8 independent physical channel interfaces per chiplet. Each channel has a maximum data transfer rate of 32 GB/s. Second, HBM provides bank-level parallelism through pipelining commands. Commands for opening/closing banks and reading data are overlapped to hide the latency of long-running commands. Compared to traditional SDRAM/DIMM-based devices, HBM provides more bandwidth per-package and parallelism and is ideal for manycore architectures, but performance depends on exploiting channel-level and bank-level parallelism.

The banked Last Level Caches (LLC) shown in Figure 2.3 are designed to exploit bank-level parallelism within a channel. The LLCs are located at the top and bottom of the network to reduce memory access latency. Each bank of the LLCs is connected to a column and is mapped to a unique address range in the NoC, and each port maps to an exclusive set of HBM banks within the LLC's channel to avoid concurrency and conflicts. Linear traversals through the memory space of a LLC expose bank-level parallelism to software.

2.6.3 *Computational Cores*

The computational cores in the manycore are throughput-optimized, general purpose processors that communicate over the NoC. Each core is a modified Harvard architecture with an instruction cache (I$), and a small program-controlled data scratchpad (S). Local scratchpad, remote scratchpads of other tiles, and main memory are mapped to contiguous segments in the memory space of each core. This allows nearby cores to communicate using shared memory with extremely low overhead.

Each core can issue a series of non-blocking, pipelined loads and stores to the NoC until a register hazard occurs. Multiple loads in flight exploit instruction-based memory parallelism and hide the access latency of a single access traversing the network (Table, Figure 2.3). Operations to sequential ports exploit bank-level memory parallelism, and operations to unique caches exploit channel-level memory parallelism. Manycore architectures with HBM are challenging to generate code for because of the interplay between bank and channel-level memory parallelism required to maximize performance for an application.

2.6.4 *Execution Model*

The manycore architecture operates under a Single Program Multiple Data (SPMD) execution model where each core executes the same program independently from all other cores. Multiple cores are aggregated into rectangular *groups* to perform computations that may require cooperation. Cores in a group communicate through shared memory and synchronize using dedicated barrier primitives. Groups can be executed in parallel if resources are available, or sequentially if not. Ordering between groups is not guaranteed. This programming abstraction can exploit the memory parallelism available in manycore architectures.

The manycore architecture described here is connected to a host CPU, and program execution is managed by a driver. Host programs allocate regions within the memory system and copy data from the host to the device, launch parallel or sequential groups, and copy data back to the host. This process continues until all groups have launched and finished execution. Once completed, the result can be copied back from the device to host.

2.7 CHAPTER SUMMARY

In this chapter, we first described graph processing at a high level before describing the behavior and structure of algorithms and input graphs used for evaluation in this dissertation. We then described the GraphIt DSL and its benefits. In Chapter 4 we use GraphIt to implement our code generation framework as a backend to the language. We also provided a description of the HammerBlade manycore architecture that we target in this work.

3 PERFORMANCE OPTIMIZATIONS

In this chapter, we explore optimizations to improve graph application performance on the HammerBlade manycore system described in Chapter 2.6. First, we discuss a blocked access method for graph data to better exploit memory-level parallelism (MLP) in software. Next, we present three different work partitioning schemes. We then introduce a new frontier storage format that targets very sparse frontiers. We follow this with an exploration of graph traversal directions.

3.1 BLOCKED ACCESS METHOD

In the naive execution model, cores request single words of application data at the time of use. The LLC responds to these requests, fetching data from HBM main memory when there is a cache miss. We discover from profiling that applications spend most cycles waiting on serialized read requests from LLC and DRAM. Many of these loads occur in iterations of the outer loop between which there are no data dependencies.

Software can reschedule these long-latency requests to execute in parallel by formatting work items into blocks. Cores iterate over their assigned blocks, prefetching the entire block at once. The block data is stored in the core's scratchpad memory, re-purposing it as a software managed L1 cache. Figure 3.1 illustrates the blocked memory access technique. Work items are stored in HBM in contiguous cache-aligned blocks. The LLC fetches a block from DRAM after receiving an initial load request from software. Cores issue these memory requests with an explicit call to `memcpy` to exploit pipelined non-blocking loads and hide memory latency. After fetching the blocked data, cores complete the loaded work before continuing to the next block. Cores must flush modified blocks back to main memory before fetching the next block.

This blocked access method gives software fine grain control over what data is read at the block and word granularity allowing the application to only prefetch data it is likely to use. This improves effective cache bandwidth by eliminating false sharing between work items and improves core memory access latency by improving locality for a work item.

3.2 WORK PARTITIONING

We implement and explore the two load balancing methods: vertex-based and edge-aware vertex-based partitioning. We also propose our own manycore specific work partitioning method: alignment-based partitioning.

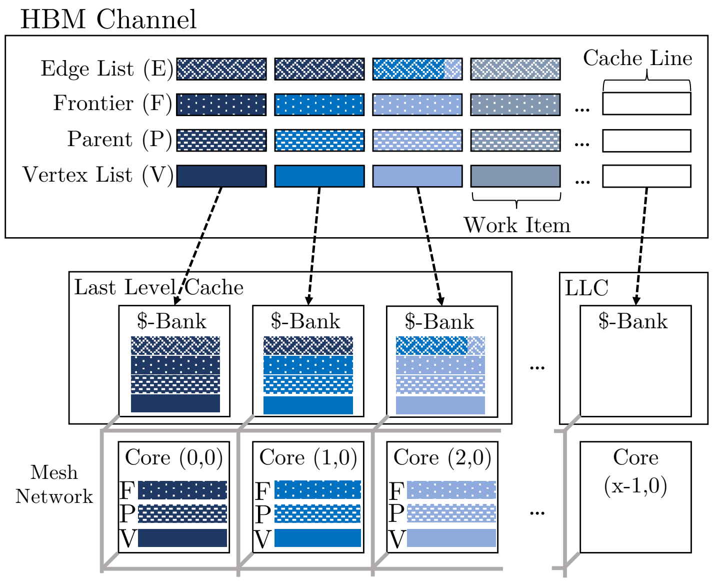


Figure 3.1: Data layout of BFS with blocked accesses. Cores are assigned cache aligned work items. Cores prefetch entire line-sized blocks of data to hide request latency.

VERTEX-BASED

In vertex-based partitioning, all cores receive an equal number of vertices to traverse. In real world graphs, the number of edges per vertex can vary drastically, and due to this, the vertex-based approach can lead to workload imbalance.

ALIGNMENT-BASED

In alignment-based partitioning, cores work instead on smaller work blocks of vertices that better align with cache lines in the LLC and make better use of the entire memory system by increasing effective bandwidth. In this method, the vertices in the graph are split into V/b work blocks where b is the number of vertices contained in each work block and V is the total number of vertices in the graph. We select b to be a multiple of the cache line size. By doing this and by reducing the size of the active set of vertices that each core is working on, we are able to increase the cache hit rate and reduce cache line contention.

EDGE-AWARE VERTEX-BASED

Edge-aware vertex partitioning scheme considers the number of edges being assigned to each core when partitioning the vertices. Figure 3.2 demonstrates how edge-aware partitioning differs from vertex-based.

In Figure 3.2, Partition 1 shows a vertex-based partitioning and Partition 2 represents an edge-aware partitioning of the graph. In both cases, the number of vertices assigned to each core is 5, but in the vertex-based partitioning scheme the first core is assigned 8 edges and the second core is assigned 21 edges. This could lead to a large workload imbalance. In the

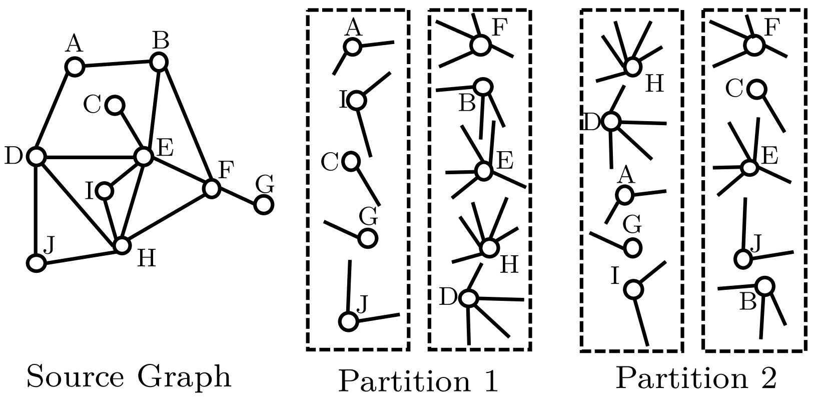


Figure 3.2: Depiction of vertex-based and edge-aware vertex-based partitioning. Partition 1 shows an example vertex-based partitioning between two cores, and Partition 2 shows an example edge-aware partitioning that provides a better workload balance between the two cores.

```
1: function RECURSIVE RANGE($start$, $end$)
2:     if $index[end] - index[start] < grainsize$ then
3:         $core.start \leftarrow start$
4:         $core.end \leftarrow end$
5:     else
6:         $recursiverange(start, end/2)$
7:         $recursiverange(end/2, start)$
8:     end if
9: end function
```

Figure 3.3: Pseudocode for the recursive call portion of the edge-aware vertex partitioning scheme.

edge-aware partition, the first core is assigned 14 edges and the second core is assigned 15. By considering the number of edges being assigned to each core, edge-aware partitioning is able to create a more balanced workload across cores.

In order to accomplish edge-aware vertex-based partitioning, a maximum number of edges that can be assigned to a core is set before execution, we refer to this maximum value as the grain size. Edge-aware partitioning effectively does a binary search for partitions of vertices that do not exceed this maximum number of edges. Initially, all vertices are considered in one large partition. If this partition of vertices exceeds the number of edges that can be assigned to a core, it is split in half, and each of those partitions are examined as possible candidates. This process continues recursively until all vertices have been assigned to a core. The pseudocode for this recursive call is shown in Figure 3.3.

In our implementation, a single core is tasked with performing the recursive assignment of work while the remaining cores wait to receive their work assignment. Further, the grain size in our implementation is set to $(E/N) * 1.5$ where E is the total number of edges in the

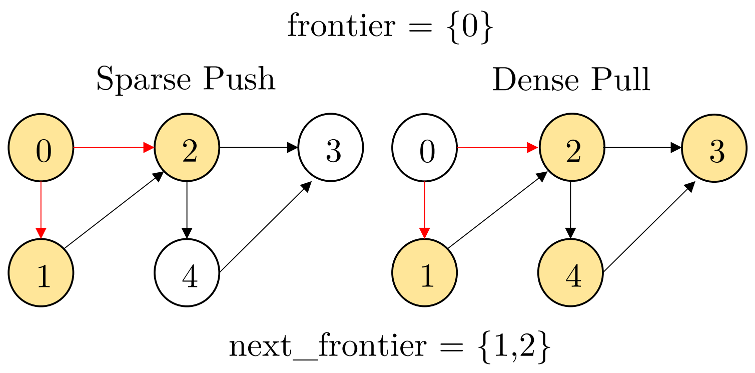


Figure 3.4: Representation of one iteration of BFS in both the PUSH and PULL directions. Nodes highlighted in yellow are visited during the iteration. Edges in red are edges that are traversed during execution.

graph and N is the number of cores in the manycore. We found that a buffer value of 1.5 was necessary to ensure that all of the vertices were assigned across the cores for all sizes of E and N that we examined. While there is overhead introduced by this partitioning scheme, we expect that a more balanced split of work will improve load balance and memory system performance on graph programs with unbalanced input graphs.

3.3 BUCKETED SPARSE FRONTIER

A common optimization when traversing in the PUSH direction is to use a sparse frontier of length m where m is the number of active vertices and each element corresponds to an active vertex. In contrast, a dense frontier is often used in the PULL direction where the frontier is stored as a boolean list of length $|V|$ and a value of `true` indicates an active vertex.

In order to maintain some locality and to easily partition work across cores, our code generator defaults to a dense frontier in both directions. However, this can lead to unnecessary cycles spent checking every vertex to see if it is in the frontier. To address this, we propose a bucketed sparse frontier storage format. In this format, the frontier is still of length $|V$ and is split into n buckets where n is the total number of cores. Each bucket stores a list of the active vertices in that range followed by -1s to indicate the end of active vertices in that bucket. This allows us to use our vertex-based partitioning scheme while reducing the number of reads to the frontier.

3.4 GRAPH TRAVERSAL DIRECTIONS

In traditional Breadth-First Search, the graph is traversed in a top-down manner by examining the outgoing edges of vertices in the frontier. For each vertex in the frontier, its edges are traversed and previously unvisited destination vertices are added to the next frontier. As the size of the frontier grows, the number of edges that need to be traversed also increases.

Beamer et al. [12] introduced a new method of traversal that reverses the direction in which edges are examined, and showed that this bottom up PULL approach can greatly reduce the number of edges traversed and improve performance. In this case, the edges of every vertex in the graph are examined, looking for vertices that are in the current frontier. There are algorithmic optimizations that can be made in the PULL direction to reduce the number of vertices and edges that are examined though. In BFS for instance, if a node has already been visited in a previous iteration, its edges do not need to be examined in any subsequent iteration.

Figure 3.4 shows both the PUSH and PULL directions for one iteration of BFS on a small graph. In this example, we start the iteration with vertex 0 in the frontier. In the PUSH case, the outgoing edges of vertex 0 are examined first. Because vertices 1 and 2 have not been visited before, they are added to the next frontier. In the PULL case, all vertices that have not yet been visited are examined. In this case, the incoming edges of all vertices except vertex 0 are traversed. If an edge originates at a vertex that is in the current frontier, the destination vertex is added to the next frontier. In Figure 3.4, vertices 1 and 2 have edges that originate at vertex 0, so they are added to the next frontier.

Beamer et al. [12] find that while the PULL direction increases performance on dense frontiers, traversing the graph in the traditional top down PUSH direction is still the better option on sparse frontiers and propose a HYBRID traversal method. The HYBRID approach examines the size of the frontier for each iteration in order to determine whether to execute in the PUSH or PULL direction for that iteration. Our code generator supports generating code in the PULL, PUSH, and HYBRID directions.

The original hybrid traversal method proposed by Beamer et al. involves two different heuristic conditions [12]. One for initially switching from PUSH to PULL and one to switch back to PUSH again towards the end of execution. These heuristics are based on: the number of vertices in the graph n, the number of edges in the graph m, the number of vertices in the frontier (n_f), the number of edges to check from the frontier (m_f), and the number of edges connected to unvisited vertices (m_u). Each of these conditions also have parameters that can be tuned to best fit the target architecture. The first condition is defined as:

$$m_f > \frac{m_u}{\alpha}$$

In their initial tuning study on a dual-socket Intel Ivy Bridge server, Beamer et al. select $\alpha = 15$ for optimal performance. The second condition to switch back to the PUSH direction at the end of execution is:

$$n_f < \frac{n}{\beta}$$

Here, the original tuning study found $\beta = 18$ to be optimal in most cases. With these α and β values, Beamer et al. found that their hybrid heuristics provided an average speedup of 3.4× over a PUSH implementation [10].

The Ligra system [97] introduced a simplified heuristic to switch between the PUSH and PULL directions. Ligra also added functionality to switch between dense and sparse frontier representations given the traversal direction. The condition for switching directions in Ligra is defined as:

$$n_f + m_f > \frac{m}{\alpha}$$

Where again n_f is the number of vertices in the frontier, m_f is the number of edges to check from the frontier, and m is the number of edges in the graph. In the Ligra implementation $\alpha = 20$. This heuristic doesn't require the application to maintain extra state to track whether the application has switched from PUSH to PULL, and removes the calculation of m_u. With this much simpler code, Ligra is able to achieve close to the same performance as the hybrid traversal method introduced by Beamer et al [97].

3.5 CHAPTER SUMMARY

In this chapter, we described graph processing optimizations that seek to improve performance on the HammerBlade manycore through data layout, memory access patterns, and graph traversal direction. First, we introduced the blocked access method for graph data that utilizes scratchpad memory and better exploits memory-level parallelism (MLP) in software. Next, we discussed three different work partitioning schemes: vertex-based, edge-aware vertex-based, and alignment-based. Vertex-based and edge-aware vertex-based are existing optimizations that we adapted for the HammerBlade architecture, and alignment-based partitioning is a scheme that we developed to better exploit MLP. We then introduced the bucketed sparse frontier that reorganizes frontier data to improve performance on very sparse frontiers. Finally, we discussed graph traversal directions and two different hybrid traversal algorithms.

4 GENERATING CODE FOR GRAPH PROGRAMS ON A MANYCORE ARCHITECTURE

To harness the benefits of manycore architectures and reduce programming complexity, we develop a code generation backend for GraphIt, a flexible domain-specific language (DSL) for graph computations [124]. We implement our backend as a GraphVM using the Unified GraphIt Compiler Framework. This GraphVM generates code targeting the representative manycore architecture described in Chapter 2.6. We use this new backend to implement and explore the performance of the optimizations discussed in Chapter 3. An overview of our approach is shown in Figure 4.1.

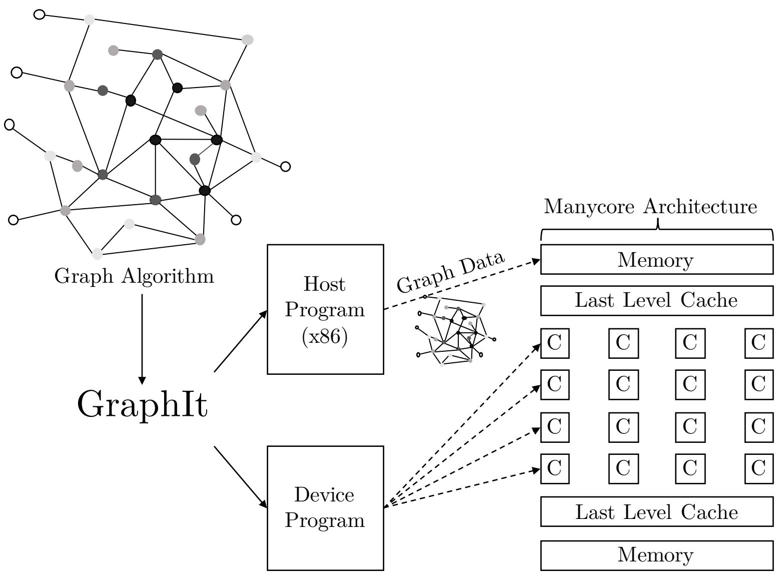


Figure 4.1: This figure shows the flow from Graph Algorithm (e.g. PageRank) to GraphIt code to execution on the manycore architecture. Code is generated for both the host and the device. The graph data structure is loaded into device memory by the host program, and the device program is executed in parallel on the manycore architecture.

4.1 MANYCORE CODE GENERATION

Reasoning about these performance optimizations and manycore-specific considerations is a challenging task. To address this, we developed a code generation framework targeting the HammerBlade manycore architecture. This code generator alleviates the need for the programmer to have extensive knowledge of the underlying architecture.

We add support for our backend and the manycore specific optimizations discussed in Chapter 3 by implementing a HammerBlade scheduling language using the abstract language provided by UGC. This scheduling language signals to the GraphIt compiler that it should generate code for the HammerBlade manycore architecture. We also add support for each of the optimizations described in Chapter 3 including our blocked access method and alignment-based partitioning scheme.

In order to generate code that runs on the manycore, we must generate code for both the host CPU and the manycore device. We chose to have the host code handle setup and coordination and to have the device code handle the core graph work. This is a natural split as it allows the manycore to handle the parallel portions of execution and lets the CPU handle the serial portions of work that require an operating system.

4.1.1 *Host Code*

The host code that we generate handles most of the setup and coordination of the graph program. We leverage a set of runtime libraries that we wrote to simplify and generalize the code generation. These runtime libraries provide wrappers over our host driver API and handle loading of program data into manycore memory, initializing the manycore and initiating kernel execution on the manycore. These runtime libraries provide a level of abstraction which allows for seamless portability of our framework. In order to target a different manycore architecture, we would only need to modify these wrapper functions and would not need to modify our code generator.

```
Graph edges = loadEdgesFromFile(file_path) ;
Vector<int> frontier = new Vector<int>(
        edges.num_nodes(), 0);
Device::Ptr device = Device::GetInstance();
addVertexOnDevice(frontier, root );
while (getVertexSetSizeDevice(frontier) != 0){
      device->enqueueJob("bfs_pull_call",
          {edges.getInIndicesAddr() ,
           edges.getInNeighborsAddr(),
           frontier.getAddr()});
      device->runJobs();
}
```

Listing 4.1: Generated HammerBlade host code for the Breadth-First Search (BFS) program shown in Listing 2.1.

Listing 4.1 shows a subset of the host code generated for the BFS program shown in Listing 2.1 and highlights some of our host side runtime libraries for coordinating placement of data and scheduling of work.

The function `loadEdgesFromFile` on line 1 loads a graph from an edgelist, formats the graph into the CSR storage format, and loads it into HBM. We also provide functions such as `addVertexOnDevice` shown on line 4, which handles the insertion of values into vectors that are stored in HBM. Finally, on lines 7 and 11 we have functions for scheduling kernel execution on the manycore: `enqueueJob()` and `runJobs()`. The enqueue job function takes the name of a kernel along with a list of parameters for that kernel and schedules it for execution. Lines 8-10 show the functions we provide to find the address of data structures stored in HBM. For vector types, we provide the `getAddr()` method. For graphs, we provide `getInIndicesAddr()` and `getInNeighborsAddr()` to get the addresses of the index and neighbor arrays respectively. The `runJobs()` function tells the manycore to execute all jobs that have been scheduled through calls to `enqueueJob()`.

4.1.2 *Device Code*

All `vertexset` and `edgeset` operators are generated as device code. Most importantly, this includes apply and filter operations along with the edgeset apply modified operation. By default, all of these operators are generated as parallel kernels meant to be executed across the entire manycore. To distribute work among cores, we implement a `local_range(V, start, end)` library function that takes as input the total number of vertices, a pointer to a start value, and a pointer to an end value. The function evenly splits the vertices across the cores and sets the start and end values as a contiguous subset of vertices for each core to work on. Our code generator replaces the call to `local_range` with `edge_aware_local_range` to do the recursive edge-aware work assignment described in Chapter 3.2.

```
template <typename TO_FUNC, typename APPLY_FUNC> int bfs_pull(int
    *in_indices , int *in_neighbors, int *from_vertexset, TO_FUNC
    to_func, APPLY_FUNC apply_func) {
    int start, end;
    local_range(V, &start, &end);
    for ( int d = start; d < end; d++) {
        if (to_func(d)){
            for(int s = in_indices[d]; s < in_indices[d+1]; s++) {
                if(from_vertexset[s] == 1) {
                    if(apply_func( in_neighbors[s], d )){
                        next[d] = 1;
                    }
                }
            } //end of loop on in neighbors
        } //end of to filtering
    } //end of outer for loop
15      barrier.sync();
        return 0;
    }
```

Listing 4.2: Generated HammerBlade device code for the Breadth-First Search (BFS) program shown in Listing 2.1.

Listing 4.2 shows the main kernel code generated for the inner loop of BFS. In this code, we can see the use of the `local_range` function on line 3 and the use of a barrier before each core exits the kernel on line 15. Lines 4-14 iterate through all of the vertices in the graph, traverse edges that have not yet been visited, and build the next frontier. The parallelism is achieved by each core executing their own contiguous range of vertices obtained from `local_range` at the same time. Once a core is finished with its computation, it waits at the barrier for all other cores to finish before returning the results of the iteration.

ATOMICS

While atomics can mostly be avoided when traversing in the PULL direction, they are still necessary in some cases and are always necessary during PUSH traversal. We leverage lock data structures to implement the necessary atomic operations used in our device code. We initialize one lock per cache line in the LLCs. Our profiling showed that this number of locks was sufficient to reduce contention in lock acquisition. Locks are assigned using a simple hash function on the address of the element for which an atomic operation has been called.

4.2 CHAPTER SUMMARY

In this chapter we introduced our HammerBlade code generation framework. Our code generator allows a user to easily reason about a variety of graph processing and manycore-specific optimizations. We described how we generate code for both the host processor and manycore device. We choose to have the host processor handle coordination and initialization while the device computes the graph traversals and updates.

5 EVALUATION

In this chapter, we evaluate the performance of our code generation framework and its optimizations implemented as a GraphVM using the Unified GraphIt Compiler Framework and outlined in Chapters 4 and 3.

5.1 EXPERIMENTAL SETUP

Host software executes natively on an Intel Xeon Gold 6254 CPU. The host libraries interface directly with the simulator environment using the SystemVerilog DPI interface. We compile all generated C++ code using the GNU Compiler Collection (GCC) with "-O2".

Cores	128 cores in a 16x8 grid
	RISC-V 32-bit IMAF ISA @ 1 GHz
	4KB Instruction Cache
	4KB Data Scratchpad
Cache	128KB Total Capacity
	32 Independent Cache Banks
	8-way Set Associative
NoC	Bidirectional 2D Mesh NoC (32-bit data, 64-bit address)
Memory	2 HBM2 memory channels
	32 GB/s per channel
	512 MB per channel

Table 5.1: Configuration of the HammerBlade manycore used in this evaluation.

MANYCORE ARCHITECTURE We evaluate our compiler using detailed RTL simulation of our manycore architecture. We model a manycore system running at 1GHz with 16 columns and 8 rows for a total of 128 cores. The simulation configuration details are shown in Table 5.1. The machine has a 512KB 8-way set associative LLC with 128-byte lines. The memory system uses two HBM2 memory channels. We model the HBM2 memory system with DRAMSim3[63], a timing accurate simulator for modeling DRAM. We use detailed, cycle-accurate RTL simulation to model our processor cores, network on chip, and LLC. Our simulation environment includes SystemVerilog bind modules for collecting

Name	**Scale**	**# Vertices**	**# Edges**
kron18	18	262,144	4,194,304
kron19	19	524,288	8,388,608
kron20	20	1,048,576	16,777,216
kron22	22	4,194,304	67,108,864
Pokec (pk)	20.5	1,632,803	30,622,564
LiveJournal (lj)	22	3,997,962	34,681,189
Hollywood (hw)	20.1	1,139,905	112,751,422
RoadCA (rc)	20.9	1,971,281	5,533,214
RoadCentral (rn)	23.8	14,081,816	33,866,826
RoadUSA (ru)	24.5	23,947,347	57,708,624

Table 5.2: The vertex and edge information for each of the graphs used in our evaluation. We use synthetic Kronecker graphs used in the Graph500 benchmark and three real world graphs.

performance metrics such as instruction and cycle counts. The RTL for this manycore has been validated in silicon, and this configuration occupies approximately 3.5 mm^2 of die area.

BENCHMARKS We evaluate our compiler on three common graph benchmarks: Breadth-First Search (BFS), Single Source Shortest Path (SSSP), and PageRank (PR). BFS has a high memory access to computation ratio and is a component of many graph algorithms. PageRank computes the importance of each vertex in the graph, and unlike BFS, spends most of its time performing computation on the entire graph. PR also users floating point operations. SSSP operates on a larger subset of the edges during execution and tends to be more sensitive to load balancing. We implement the Bellman-Ford variant of SSSP for our evaluation.

We implement all of these benchmarks in GraphIt and generate manycore code using our HammerBlade GraphVM. Due to the time costs of simulation, we do not run all iterations of our benchmarks. For BFS and SSSP, we run three iterations from the middle of execution with a random root node selected as the initial frontier. For PR, we run one iteration from the beginning of execution.

GRAPHS We use three types of graphs in our evaluation: synthetic Kronecker graphs used in the Graph500 benchmark [76], real-world road networks, and real-world, social network graphs. Kronecker graphs follow a power law distribution in order to simulate the small world property often seen in real world graphs [58]. We generate four Kronecker graphs of varying size. We primarily evaluate our results using three of these graphs: kron18, kron20, and kron22. We use three real-world graphs: Pokec [101], RoadCentral [29] and LiveJournal [119]. All of the graphs we use and their properties are shown in Table 5.2 and discussed in more detail in Chapter 2.3.

5.2 RESULTS

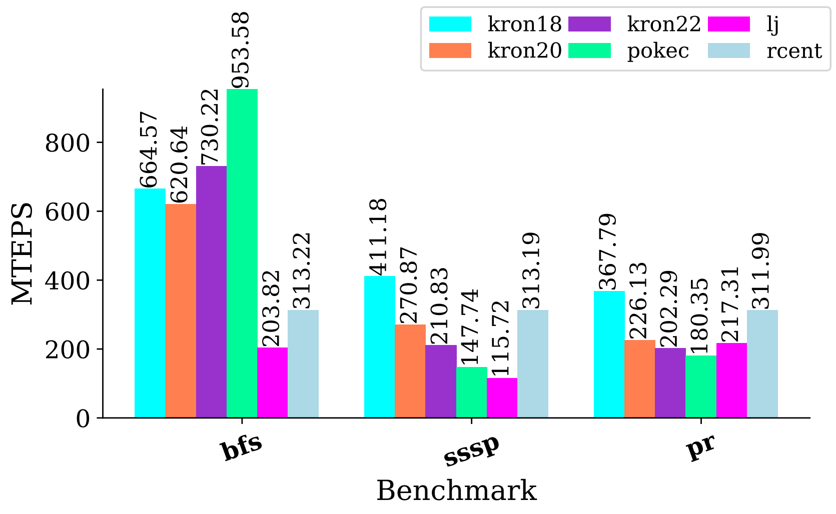


Figure 5.1: Baseline code generation results for each benchmark in the dense pull direction with no manycore specific optimizations.

To evaluate our code generator, we first present performance results for the unoptimized benchmarks. We then explore the performance trade-offs and benefits for each of the optimizations described in Chapter 3.

5.2.1 *Baseline*

To establish a baseline, we examine the performance of the code produced by our backend without any added optimizations. A key metric for our evaluation is traversed edges per second (TEPS). We report TEPS for each iteration by counting the number of edges traversed with the applied GraphIt schedule and using the cycle counts that we obtain from simulation. This metric provide an idea of execution speed that allows for comparison between different input graphs, for which the total number of traversed edges can vary significantly.

Figure 5.1 show our initial performance results for the manycore code generated by our backend without added optimizations in the PULL direction as described in Chapter 4.1. Across all results, we see a geometric mean of 520.04 MTEPS in the PULL direction.For SSSP and PR, we note that the highest performance on the Kronecker scale 18 graph. This is due to the relatively small size of the graph and the large amount of parallelism and memory

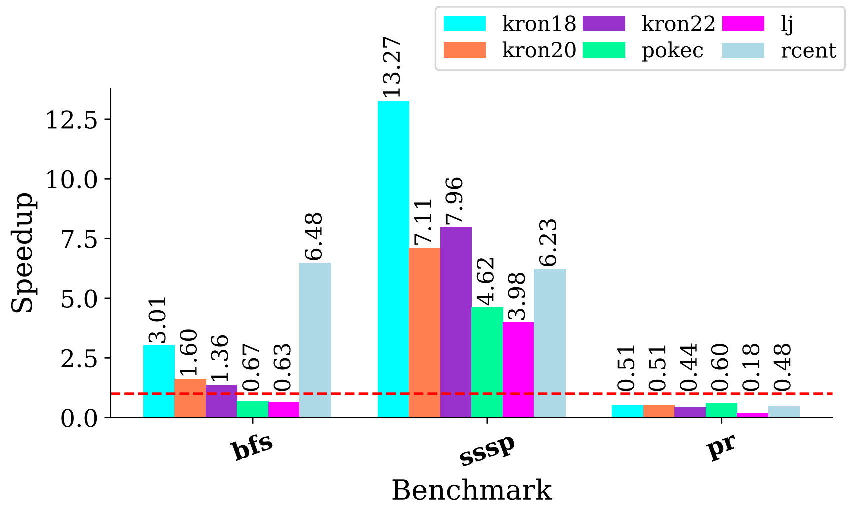


Figure 5.2: Baseline code generation results for each benchmark in the push direction with no manycore specific optimizations. Speedup over the baseline pull direction is shown.

bandwidth of the manycore architecture. For BFS we see the highest performance on the Pokec graph.

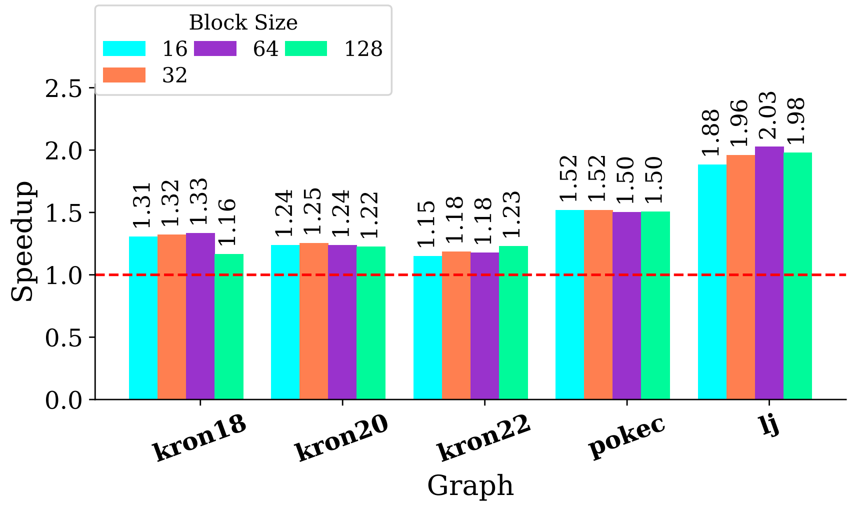
Figure 5.2 shows the performance of the manycore code generated in the PUSH direction. Because graph traversal direction is an algorithmic optimization that changes the number of traversed edges in each iteration, we report speedup over the PULL direction in terms of total execution time here.

BFS has a very high memory access to compute ratio, so the number of reads greatly affects the performance. We find that the PULL direction is optimal on our two real-world social-network graphs and that PUSH is optimal on the road-network and Kronecker graphs. The two social-network graphs have a large number of edges and contain hub nodes, and as a result, these benefit most from the early termination condition in bfs PULL where visited vertices need not be examined in subsequent iterations.

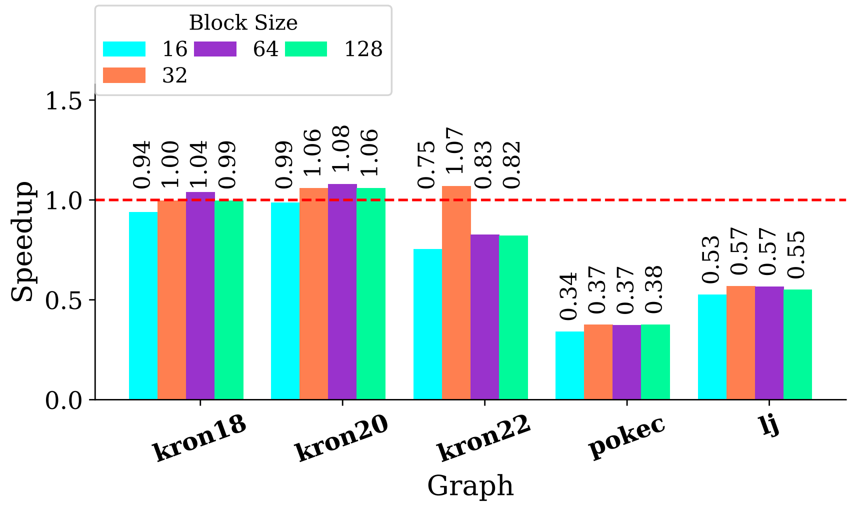
Because SSSP is not able to make use of early termination in the PULL direction, the PUSH direction results in fewer traversed edges, and thus performs less work. As a result, we observe speedup across all graphs when traversing in the PUSH direction. From our simulation results, we also determined that the graphs made better use of LLCs in the PUSH direction.

Interestingly, the PULL direction was optimal across all graphs on PageRank. Unlike BFS, PageRank must visit all vertices in all iterations regardless of traversal direction. When we examined our simulation results, we saw that the PULL direction issued fewer read requests to HBM which most likely accounts for the performance difference.

5.2.2 *Blocked Access Method*



(a) SSSP



(b) BFS

Figure 5.3: Speedup results for varying block sizes using the blocked access method on BFS and SSSP. Speedup is calculated over the baseline pull direction implementation.

Figure 5.3a shows the performance benefit of the blocking optimization across different block sizes on the SSSP benchmark in the PULL direction. Results for block sizes of 16, 32,

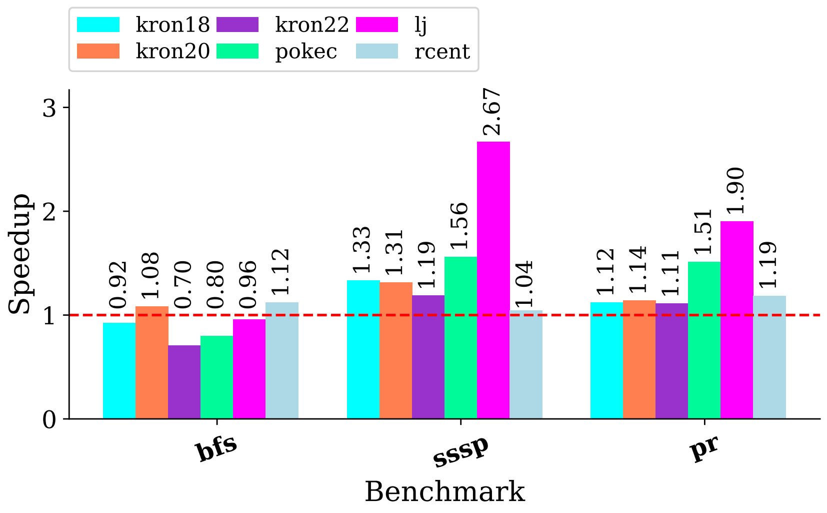


Figure 5.4: Blocked access method speedup results for each benchmark. Speedup is calculated over the baseline pull direction implementation.

64, and 128 elements are shown. Speedup is normalized to the baseline performance of SSSP in the PULL direction without any optimizations. Similarly, Figure 5.3b shows the performance benefit of the blocked access method with varying block sizes on BFS in the PULL direction.

Interestingly, we do not see much variation in speedup across different block sizes. LiveJournal and Kronecker 18 see the most speedup with a block size of 64, Pokec and Kronecker 20 see optimal performance with block size 32, and a block size of 128 is optimal for Kronecker 22. We observe that the optimal block size appears to be input graph dependent for the other benchmarks we studied as well.

Figure 5.4 shows the speedup for each benchmark when using the blocked access method. Again, speedup is normalized to the baseline performance of each benchmark in the PULL direction. In this figure, the optimal block size for each input graph and benchmark is used to compute the speedup. Blocking provided a mean speedup of 1.26× and a maximum speedup of 2.67×.

Blocking is primarily motivated by the microarchitecture's ability to hide memory load latency. The prefetching effect from blocking can yield significant performance improvement, but it also provides the effect of data caching. Even if the data is used once, the prefetching effect can yield significant performance improvement. We observe the most speedup with SSSP because it traverses more edges than BFS and benefits most from prefetching. We see the least speedup from blocking with BFS and see performance degradation on the Pokec and LiveJournal graphs. The PULL direction for BFS traverses the fewest edges of the three workloads thanks to a filter condition on destination vertices.

This means that much of the prefetched data is never used, and the additional memory reads become overhead.

Overall, we find that our blocking optimization provides performance improvement in all but four cases. Blocking reduces stalls from memory system requests and improves the hit rate of the LLC. We find, however, that it does not have a large impact on DRAM stalls. Making use of the low-latency scratchpad memory and coalescing accesses most improve performance on benchmarks that traverse more edges in the graph.

5.2.3 *Work Partitioning*

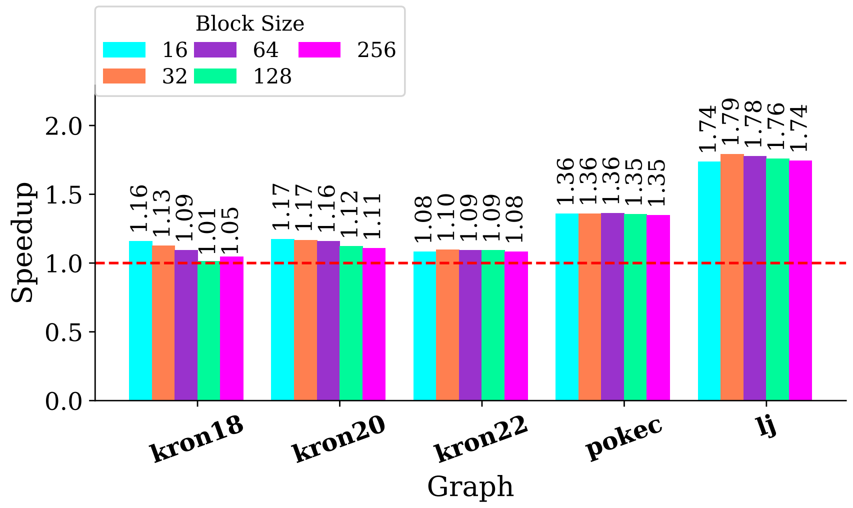
Figure 5.5 shows the performance benefit of alignment-based partitioning across different work block sizes on SSSP and BFS. Speedup is calculated over the baseline SSSP PULL implementation and baseline BFS PULL implementation. Similarly to the blocked access method results, we find that the optimal work block size for alignment-based partitioning is input graph and benchmark dependent. A work block size of 16 was optimal for Kronecker 18 and 20; liveJournal and Kronecker 22 saw optimal performance with a block size of 32 and Pokec with a block size of 64.

Figure 5.6 shows the speedup for each benchmark when using alignment-based partitioning over the baseline vertex partitioning scheme. The optimal work block size for each input graph and benchmark is used to calculate speedup. We observed a performance improvement in all input graph and benchmark combinations, with an average speedup of 1.35× and a maximum speedup of 2.68×.

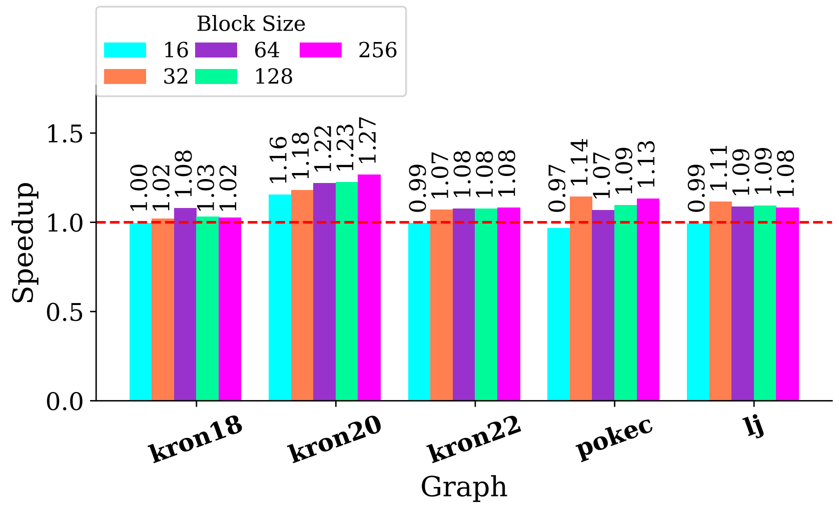
Alignment-based partitioning aims to make better use of the memory system on the manycore. By assigning smaller working sets to each core, there is less contention in the LLCs and all benchmarks achieve higher cache hit rates. Improving the LLC hit rate decreases the total amount of memory that needs to be read from HBM and decreases the amount of time spent waiting on outstanding memory requests.

Figure 5.7 shows the performance of edge-aware vertex partitioning relative to the baseline vertex-based partitioning on all benchmarks in the PULL direction. We achieved a speedup in seven benchmarks and input graph combinations, with a maximum speedup of 1.83×. If performance degraded, we observed an average slowdown of 15.32%. This scheduling optimization relies heavily on the structure of the input graph and the algorithm, so these results were somewhat expected.

Figure 5.8 shows the core cycle count probability distribution for each benchmark on Kronecker graphs of scale 18, 19, and 20. Each plot shows a probability density estimate of the total cycle counts across cores when executing both the edge-aware and vertex-based partitioning schemes for a single iteration taken from the middle of application execution. The mean for each probability distribution is also indicated. From this, we can see how the edge-aware work distribution affects the workload on each core. We observe a decrease



(a) SSSP



(b) BFS

Figure 5.5: Speedup results for varying work block sizes using the manycore aware vertex partitioning scheme on SSSP and BFS. Speedup is calculated over the baseline pull direction implementation.

in tail latency on the edge-aware probability density curve for BFS on all input graphs in Figure 5.8 indicating a more even load balance between cores.

We observe a noticeable improvement in most of the SSSP experiments. Figure 5.8(e) shows the most dramatic shift in terms of cycle distributions. PageRank sees the worst

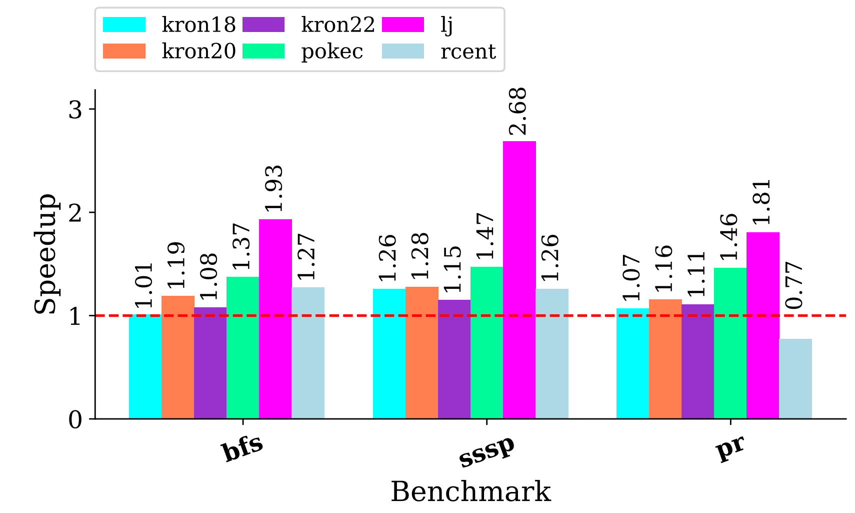


Figure 5.6: Alignment-based partitioning speedup results for each benchmark. Speedup is calculated over the baseline pull direction implementation.

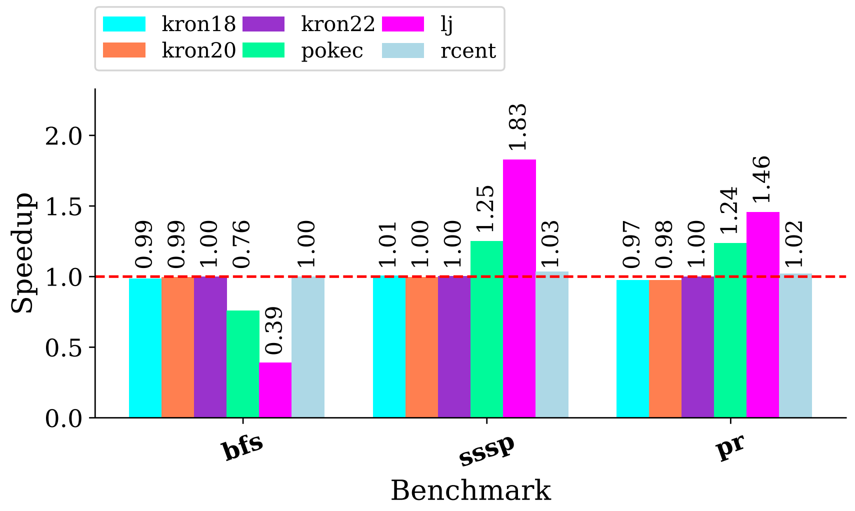


Figure 5.7: Speedup results for edge based optimization over the baseline dense pull implementation for each benchmark.

impact on cycle distribution in Figure 5.8 when switching to the edge-aware partitioning scheme and showed slight slowdowns on kron18 and kron20 in Figure 5.7. These results show that the edge-aware scheme does have a noticeable impact on cycle count and load balancing

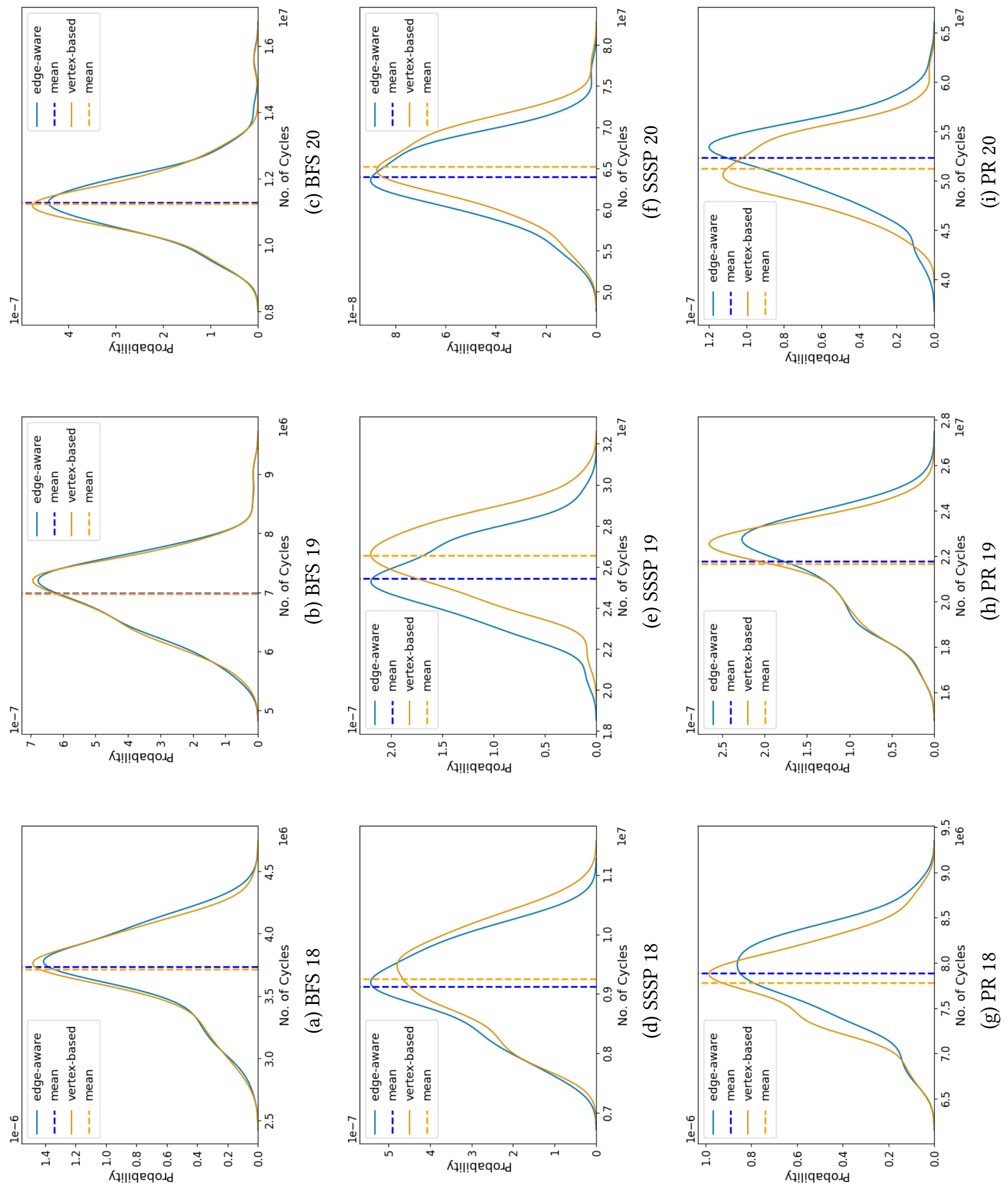


Figure 5.8: Probability distribution of cycle counts for each core for vertex-based and edge-aware partitioning.

Overall, while we see performance improvements with edge-aware vertex-based partitioning on some benchmarks, we find that alignment-based partitioning provides the most performance improvement in all cases. Despite its workload balancing benefits, edge-aware partitioning does not account for the manycore specific properties of the memory system. Because the memory is the main bottleneck of graph algorithms, the alignment-based scheme which explicitly takes into consideration the properties of the manycore's memory system is ideal.

5.2.4 *Bucketed Sparse Frontier*

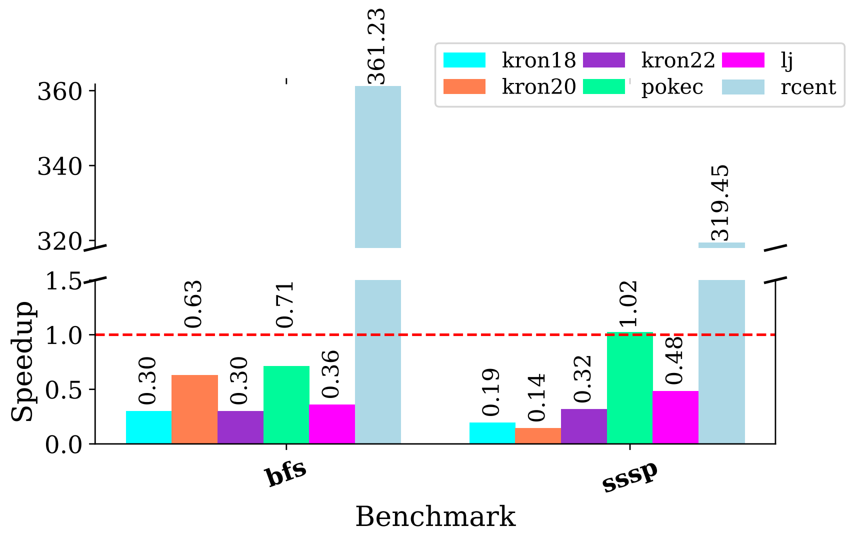


Figure 5.9: Bucketed sparse frontier speedup results for each benchmark. Speedup is calculated over the baseline PUSH direction implementation.

Figure 5.9 shows the performance of the bucketed sparse frontier optimization relative to the baseline vertex-based partitioning for all benchmarks in the PUSH direction. We only evaluate this optimization on BFS and SSSP because PR is not a frontier based algorithm. We find the largest improvement on the road_central graph with a speedup of 361.23× for BFS and 319.45× for SSSP. This is expected as it has the smallest frontiers and vertices in the graph have very low degree, and thus benefits the most from removing unnecessary reads to the frontier.

We observed a large slow down in almost every other case. This was due to an increase in cache line contention that decreased the LLC hit rate. Cores ended up issuing more read requests concurrently with this scheme than they did using a dense frontier which degraded performance.

5.2.5 *Hybrid Traversal*

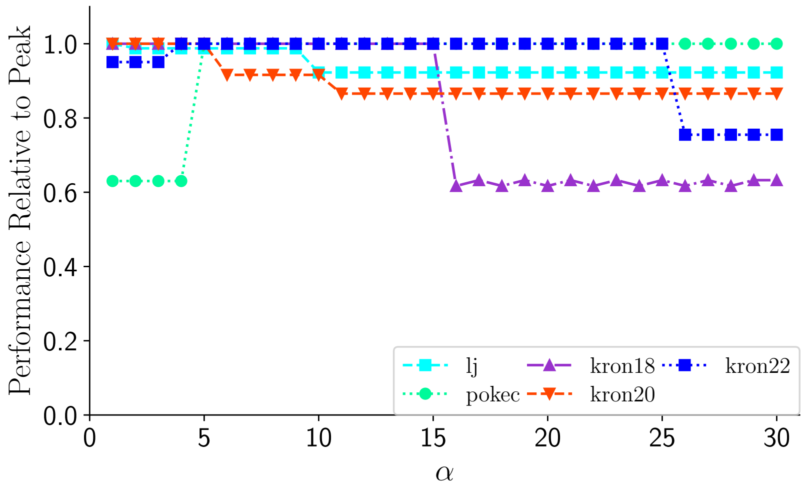


Figure 5.10: Performance results for hybrid traversal on BFS for each graph relative to its best performance for a range of α values. We find $\alpha = 5$ to be optimal on HammerBlade.

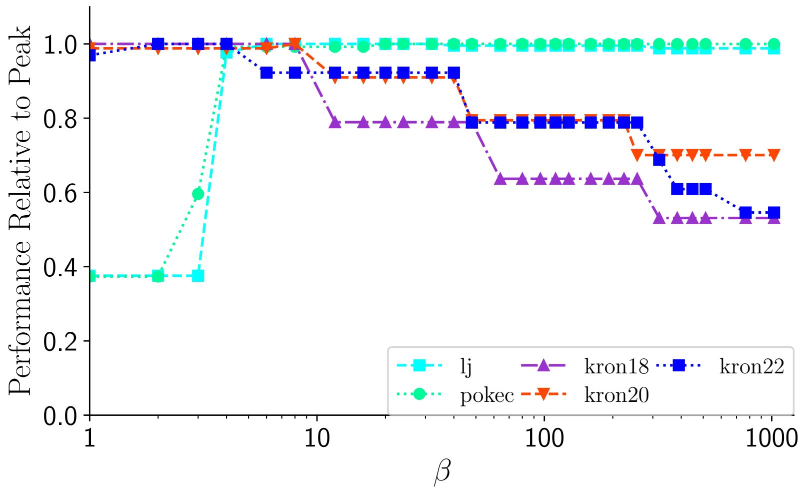


Figure 5.11: Performance results for hybrid traversal on BFS for each graph relative to its best performance for a range of β values. We find $\beta = 4$ to be optimal for these graphs on HammerBlade

To evaluate hybrid traversal on HammerBlade, we first tune the α and β variables used in the heuristics proposed by Beamer et al. Figure 5.10 shows the results for tuning α to the HammerBlade manycore on 5 different input graphs. We run BFS without optimizations for 10 iterations on each graph. We don't evaluate on any road networks as these graphs rarely benefit from hybrid traversal. We examine values of α between 1 and 30. While Beamer et al. found $\alpha = 15$ to be optimal on an Intel IvyBridge server, we find that the optimal value of α is much lower. For BFS with the graphs we studied, we find that $\alpha = 5$ provides the best performance in the most cases.

The results for tuning β are shown in Figure 5.11. Again, we run BFS for 10 iterations on the same input graphs. We choose values for β in the range 1 to 1024. Again, the optimal β value for HammerBlade is lower than the one found by Beamer et al. We find $\beta = 4$ to provide the best performance in the most cases.

Our lower α and β values result in hybrid traversal preferring the PUSH direction. Repeating this study with alignment-based partitioning in the PULL direction or repeating it for different benchmarks might yield slightly higher α and β values. However, the differences in the memory hierarchies between an Intel IvyBridge server and the HammerBlade manycore likely account for the lower values.

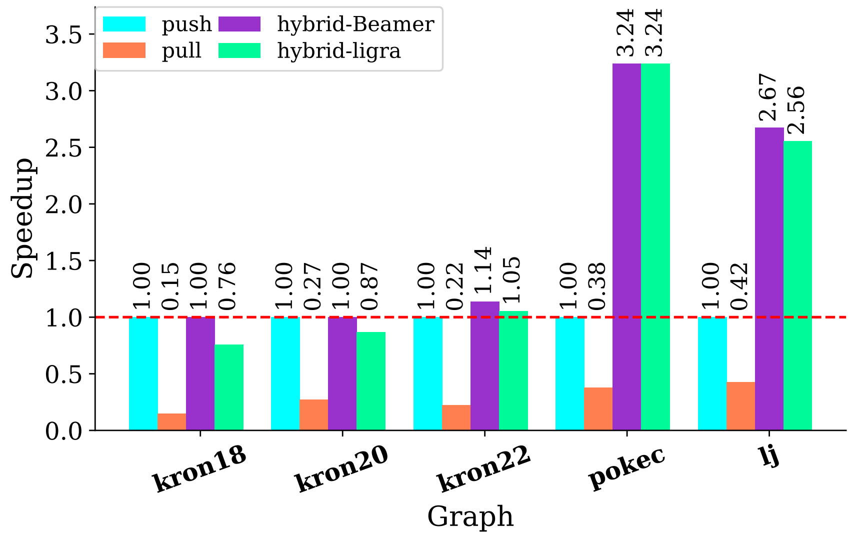


Figure 5.12: Traversal direction speedups for BFS relative to the PUSH implementation. Hybrid-beamer is the hybrid method proposed by Beamer et al. and hybrid-ligra is the method introduced in Ligra.

Finally, we present results for PUSH, PULL, Beamer's hybrid traversal (with $\alpha = 5$ and $\beta = 4$), and Ligra's hybrid traversal in Figure 5.12. We evaluate 10 iterations of BFS with the same input graphs. We report speedup relative to the PUSH implementation results. Similarly to Figure 5.2, we find that traversing in the PULL direction results in a slowdown over PUSH on the Kronecker graphs. Because we sample a larger number of iterations here, we also observe a slowdown for PULL on LiveJournal and Pokec. With the values of α and β that we use, we find that Beamer's hybrid traversal never switches to PULL on kron18 and kron20. For these graphs we see a performance decrease with the Ligra hybrid traversal method. On the two real-world graphs, we see larger performance improvements up to 3.24×. We also find that Ligra's hybrid traversal is able to achieve performance within 4.5% of the performance of Beamer's hybrid traversal implementation on the two real-world graphs and is within 12% on average across all graphs studied. Because of this and because of the simplicity of its implementation, we use the Ligra hybrid traversal heuristic in our runtime libraries.

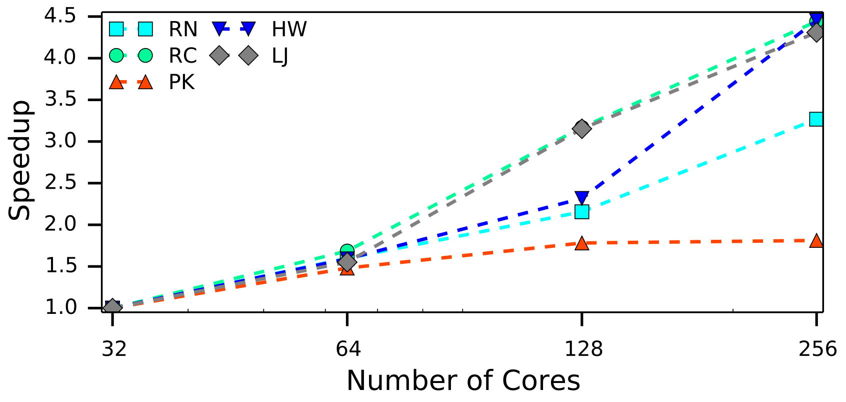


Figure 5.13: Scaling results for BFS across four HammerBlade manycore configurations. LLC capacity and the number of columns (16) is held constant while the number of rows is varied from 2 to 16.

Graph	**DRAM Stalls**	**Bandwidth**	**Speedup**
LiveJournal	0.78	3.03	1.19
Hollywood	0.79	2.17	1.53
Pokec	0.83	3.02	1.49

Table 5.3: Impact of the blocked access optimization on SSSP. Reduction in DRAM stalls, improvement in memory bandwidth utilization, and overall speedup.

5.2.6 *Performance Analysis*

Figure 5.13 shows how performance scales on the HammerBlade manycore as the number of cores are varied. Optimized BFS code was executed on four different machine configurations:

we hold the LLC capacity and number of columns (16) constant and vary the number of rows (2, 4, 8, and 16) to vary the total number of cores. We observe increased performance as we increase the number of cores, suggesting that the generated code provides sufficient parallelism to obtain high performance. The strong scaling indicates that our generated code can successfully exploit parallelism. We highlight BFS for this scaling study due to its high memory access to compute ratio. Because memory is a major bottleneck for most graph applications, we anticipate that these scaling trends will continue to hold for other applications.

Table 5.3 demonstrates performance improvements for SSSP with delta-stepping when the blocked-access optimization is applied on three selected input graphs. The table shows the overall performance speedup, the improvement in effective random access bandwidth, and the reduction in DRAM stalls during execution for the three input graphs where the this optimization was applied. The blocked-access optimization loads fixed-size blocks of vertex data into local scratchpad memory. Loading the data into scratchpad reduces access latency in exchange for bandwidth. This optimization exploits memory parallelism to hide DRAM access latency in exchange for loading unused data and reducing effective bandwidth. For SSSP, we observe that this optimization decreases DRAM stalls, increases memory bandwidth utilization, and improves overall application performance. Because SSSP is a data driven application, it benefits from this optimization which targets the memory hierarchy and utilizes the manycore's software managed scratchpads.

5.3 DISCUSSION

Figure 5.14 shows speedups obtained from applying the blocked-access method or alignment-based partitioning to a wider range of benchmarks and input graphs. Here we evaluate performance on three real-world social network graphs (LiveJournal, Pokec, and Hollywood) and three road networks (RoadCA, RoadCentral, and RoadUSA). In addition to BFS and PR we also evaluate on SSSP with delta-stepping, CC, and BC. For PR we simulate one iteration, and for the remaining applications, we simulate five representative iterations that cover a range of frontier densities and execution behavior. We use hybrid traversal in the baseline code of BFS, BC, and SSSP to decrease simulation times. BC, CC, and BFS benefit from alignment-based partitioning, while PR and SSSP use the blocking optimization due to their more compute-intensive nature. As shown throughout our evaluation, these optimizations better utilize the memory hierarchy and provide up to 4.96× speedup over unoptimized code.

Table 5.5 shows the highest achieved MTEPS for each input graph and benchmark along with the optimization that was used in the generated code. Alignment-based partitioning provides optimal performance across the most input graphs and benchmarks, and the next best performing optimization is blocking. This is not surprising, as all of our benchmarks are limited by DRAM, and blocking and alignment-based partitioning are optimizations

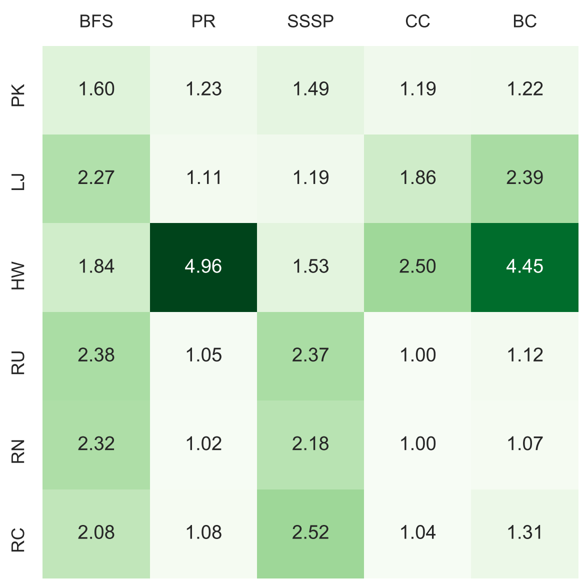


Figure 5.14: Speedups for all benchmarks and real-world input graphs attained from applying the blocked-access method or alignment-based partitioning optimization. Here we examine BFS, PR, CC, BC, and the delta-stepping variant of SSSP.

that makes better use of the memory system. By adding blocking and alignment-based partitioning, we are able to coalesce read accesses to vertex data. In addition, with blocking, we are able to make use of explicitly managed, low-latency scratchpad memory in our blocked access method. The blocked access method and alignment-based partitioning improve the memory system performance of our benchmarks by targeting a key performance limiter of graph algorithms.

Source	**Platform**	**Benchmark**	**Graph**	**MTEPS**
[98]	Xeon Phi MIC	CC	LJ (22)	240
[98]	Xeon Phi MIC	CC	Flickr (19)	140
[52]	GTX780	BFS	LJ (22)	272.4
[127]	C2050	BFS	KKT (21)	351.5
[118]	k40c	SSSP	LJ (22)	334.2
[115]	k40c	SSSP	LJ (22)	217.9

Table 5.4: Performance results in MTEPS from other graph processing frameworks. The benchmark CC is strongly connected components.

We do not include a direct comparison of our results against other systems; however, Table 5.4 shows performance for some of the frameworks mentioned in Chapter 2. This table is included to help contextualize the results reported above. It is also important to note that we only simulate a small portion of the total manycore architecture. The full manycore architecture would have eight HBM channels instead of the two that are simulated along with many more cores. As a result, the full architecture would have much higher memory bandwidth and more compute resources. Because all of our benchmarks are limited by DRAM, we anticipate that we would achieve better performance on the full system due to an increase in memory bandwidth and decrease in memory contention.

5.4 CHAPTER SUMMARY

In this chapter we presented performance results for our code generation framework introduced in Chapter 4. We showed baseline code performance and explored the performance benefits of the optimizations we presented in Chapter 3. We demonstrated speedups of up to 4.96× when our manycore-specific optimizations are applied. Our evaluation demonstrated that our GraphIt backend and optimizations can improve high performance on a wide variety of input graphs and algorithms. This is because our code generation framework enables flexible exploration of optimizations in order to find the optimal schedule for each benchmark and input graph.

Benchmark	Graph	Traversed Edges	Time (ms)	MTEPS	Speedup	Schedule
BFS	kron18	364,687	3.02	120.61	3.01	PUSH
	Kron20	3,075,348	19.82	155.13	1.60	PUSH
	Kron22	130,762,502	136.73	956.38	1.38	HYBRID
	Pokec	28,787,299	32.60	916.15	3.74	HYBRID, alignment-based partitioning on PULL
	LJ	31,773,988	80.68	393.83	1.93	PULL, alignment-based partitioning
	RoadCentral	470	0.13	3.39	2342.51	PUSH, bucketed sparse frontier
SSSP	kron18	769,042	1.77	434.36	15.68	PUSH
	Kron20	12,300,469	19.96	616.26	8.7	PUSH
	Kron22	86,836,312	155.19	559.51	5.89	PUSH
	Pokec	10,018,259	236.91	42.29	2.62	PUSH, bucketed sparse frontier
	LJ	128,114,036	515.65	248.45	3.44	PUSH,
	RoadCentral	850	0.15	5.35	2043.07	PUSH, bucketed sparse frontier
PR	kron18	3,805,022	9.23	412.34	1.12	PULL, blocked access method
	Kron20	15,700,369	60.07	261.35	1.16	PULL, alignment-based partitioning
	Kron22	64,155,711	285.29	224.88	1.11	PULL, blocked access method
	Pokec	30,622,564	112.34	272.58	1.51	PULL, blocked access method
	LJ	34,681,189	83.96	413.06	1.90	PULL, blocked access method
	RoadCentral	33,866,826	91.57	369.84	1.19	PULL, blocked access method

Table 5.5: Table containing best execution time results across all benchmarks and input graphs. The total number of traversed edges, achieved MTEPS, speedup over the baseline implementations, and the optimizations used to achieve these results are also listed above. Note that algorithmic changes can affect the total number of traversed edges. As a result, a benchmark may report speedup over the baseline but report lower overall MTEPS. This is especially noticeable on the road_central graph as discussed in Chapter 5.3

6 RELATED WORK

In this dissertation, we talk about generating optimal code for graph algorithms on a manycore architecture with HBM. We explore both general graph processing and manycore specific performance optimizations and show how to generate performant code for a manycore architecture leveraging these optimizations. There has been a large amount of work done in the space of optimizing parallel graph algorithms on various architectures. Graph processing has traditionally been done on CPU and GPU machines, and increasingly on FPGAs and custom architectures. First, we discuss work that focuses on understanding graph performance. Then, we talk about graph processing on CPUs, GPUs, FPGAs, and custom architectures. Finally, we introduce previous manycore architectures and discuss prior work on graph processing on manycore architectures.

6.1 UNDERSTANDING GRAPH PERFORMANCE

Several pieces of work have focused on understanding the bottlenecks of graph processing [9, 11]. It is clear that memory is the main bottleneck for graph algorithms. Most graph programs struggle to achieve high memory bandwidth due to insufficient instruction window sizes and memory load dependencies inherent in graph accesses. Further, many graph applications actually experience both low memory bandwidth and low compute due to memory latency [11]. We find that memory accesses are the main bottleneck in our system and propose a blocking method and an alignment-based partitioning method to coalesce accesses.

Graph processing work is often split between making algorithmic improvements and making systems improvements. Beamer et al. propose a new way to evaluate improvements in graph processing that allows for evaluation of both algorithmic and system improvements in one metric [13]. We leverage their traversed edges per second (TEPS) metric in our evaluation.

6.2 GRAPH PROCESSING ON CPUS

Multicore CPUs are an example of Multiple Instruction Multiple Data (MIMD) execution. CPUs have powerful arithmetic cores for high-performance scientific computations and handle complex control-flow. They have large, multi-level memory systems that can load large graph data structures into memory. Multicore systems can partition workloads across multiple threads of execution and perform complex load-balancing technique [77].

The benefits of CPUs can also be their downfall. Deep memory hierarchies introduce long latencies for random-access memory patterns, and coherence protocols between processors introduce significant overheads on shared datasets even with small numbers of threads [68]. Graph applications are dominated by random memory accesses with minimal computational demands, which can leave a high-performance high-energy consuming CPU core idle.

Many frameworks have been proposed for multicore CPU systems including Ligra [97], GraphLab [67], GraphChi [56], and Galois [79]. These frameworks use different approaches to address the limitations of graph processing on CPUs including priority scheduling of work and various load balancing techniques. There are also a wide range of DSLs for shared-memory graph processing including Ligra, Galois, GraphGrind [99], Polymer [122], Gemini [131], and Grazelle [39]. These frameworks adopt a frontier-based model for execution. Green-Marl [45], Socialite [94], Abelian [36], and EmptyHeaded [2] are also shared-memory graph processing DSLs, but they do not allow for creation of active vertex sets and perform worse than frontier-based frameworks [2, 92]. However, I do not spend very much time on these optimizations because I focus on a manycore with much simpler cores and a smaller memory hierarchy.

6.3 GRAPH PROCESSING ON GPUS

GPUs have shown great promise in exploiting the inherent parallelism in graph applications [64, 71]. GPUs operate within the Single Instruction Multiple Threads (SIMT) execution model. When a memory access stall occurs the executing warp is replaced by another warp from the thread pool to achieve Instruction Level Parallelism (ILP). Memory-level parallelism (MLP) is also achieved through coalescing memory accesses of threads in a warp to contiguous addresses in memory in order to form a single high-bandwidth memory transaction. These features expose more thread-level and memory-level parallelism than CPUs.

The efficiency of the SIMT model however can quickly degrade due to branch and memory divergence. Threads can take different control flow paths in a branch instruction and be forced to serially execute [35]. Threads in a warp can also access different levels of the memory hierarchy which will stall the entire warp until *all* requests have been serviced. The problem is amplified in graph analysis applications where the random pattern of memory accesses reduce the effectiveness of coalescing and result in multiple high-latency memory transactions that stall execution [96, 117].

Previous work explores the performance and engineering benefits of high-level software frameworks for graph analytics on CUDA programmable GPUs [52, 80, 115, 118, 127]. One notable example, Gunrock [115], is similar to the DSL we have chosen for our code generator, GraphIt. It uses a data-centric, frontier-based abstraction for specifying compute, and a program consists solely of a problem description (input graph, data management, etc.), user defined computation, and an enactor or entry point to computation. Similar to GraphIt, it focuses on specifying what should be done and not how that work should be scheduled. However, it does not allow users to experiment with different scheduling options. In this work I focus on targeting a broader class of manycore processors. While there are some interesting software base optimizations in this area of work, I do not spend too much time on them due to the SPMD model of our manycore architecture as opposed to the SIMT model of GPUs.

6.4 GRAPH PROCESSING ON FPGAS

FPGAs present an interesting platform for graph processing frameworks. Engelhardt et al. [33] and Dai et al. [25] both propose vertex-centric frameworks on single and multi FPGA systems. Meanwhile, Zhou et al [130] use large external memory and an edge-centric model in their FPGA framework. Dai et al. [26] propose another multi-FPGA system that partitions edge and vertex data among the FPGAs in the system. There has also been work on FPGA implementations of individual graph benchmarks [128, 129]. Because my work is focused on general purpose manycore architectures, I do not address these FPGA specific optimizations.

6.5 GRAPH PROCESSING ARCHITECTURES

There has been significant work designing custom hardware for sparse compute and graph processing [3, 6, 27, 44, 48, 62, 120]. Ozdal et al. [81] propose an asynchronous, vertex-centric hardware for graph processing. Their asynchronous model does not execute graph algorithms using iterations with strict boundaries like traditional graph processing. The goal of their work was to maintain a larger active set of vertices and edges in order to increase memory level parallelism. Ham et al. [44] design an architecture for vertex centric graph processing. They note that most of the instructions in graph processing programs are for traversing the graph (~95%) and only a small amount (~5%) are specific to the graph benchmark. As a result, they focus their design on data path specification and memory subsystem optimizations. Their architecture also makes use of small scratchpad memory. Zhuo et al. [132] and Ahn et al. [6] present graph processing architectures that leverage processing in memory (PIM). Both of these works focus on reducing irregular movement of data by reordering vertex computations and through movement of computation. These works tend to focus on graph optimizations that are not solvable in software, while my work focuses on optimizations that can be implemented on general purpose manycore architectures.

6.5.1 *Hardware Modifications*

There is also work being done on modifying small parts of existing hardware systems to improve graph processing performance. Segura et al. [93] propose adding a stream compaction unit to existing GPUs to handle memory compaction and coalescing. Their stream compaction unit uses bitmasks in order to construct a subset of the edgelist for processing on the GPU in order to reduce duplication of work. This, along with coalescing and organizing the data to improve cache efficiency, helps to improve performance on the GPU. Matam et al. [70] take the approach of customizing the memory system and making it aware of graph semantics in order to improve caching and performance. They made their SSD aware of the CSR structure of graphs by adding a graph based lookup system and attempted to store edges for a node in one SSD page whenever possible. Another memory focused solution is proposed by Mukkara et al. [75]. In their work, they design a locality aware hardware scheduler for scheduling graph traversal. Again, I do not focus on these types of optimizations in my work because my focus is on obtaining performance on a general purpose manycore.

6.6 GRAPH PROCESSING ON MANYCORE ARCHITECTURES

6.6.1 *Manycore Architectures*

Over the years, many diverse manycore architectures have been proposed. Unlike these past manycore architectures, the manycore architecture we target maximizes compute density and utilizes HBM for increased memory bandwidth. Raw [102] was an early manycore design that focused on exploitation of ILP via various forms of message-passing. After Raw, the Tilera architecture was proposed [88]. The Tilera chip had comparatively large cores that were each individually capable of running the Linux OS, and implemented directory-based cache coherent shared memory. Adapteva [43] and [5] was a manycore design that focused on compute density but lacked a parallel DRAM memory system suitable for graphs. Unlike GPUs and Xeon PHI, the target manycore architecture is a pure SPMD machine, without the complexity and thread divergence challenges of SIMT/SIMD units.

6.6.2 *Manycore Graph Processing Frameworks*

There has been previous work on optimizing graph algorithms on manycore architectures. Deveci et al [31] and Slota et al. [98] adopt an edge-centric approach for their implementation of the graph coloring algorithm and evaluate it on both a GPU and Xeon Phi. While they both leverage the Kokkos library [31] for platform portability, these works focus on highly optimizing a small handful of graph algorithms. Chavarria et al. [23] focus on optimizing the performance of community detection on the Tilera processor. Their optimizations focus on load balancing and improving locality.

While a lot of work aims at optimizing specific graph algorithms on manycores, there has also been work on developing more general graph processing frameworks for manycore architectures. GraphPhi [83] is a Xeon Phi runtime library for graph processing that uses HBM. The authors propose a hierarchical graph storage format that partitions a graph to better map to the Xeon Phi's SIMD units and to allow for better load balancing. The framework uses both vertex-centric and edge-centric processing to compute graph applications, and as a result, requires the program to store the input graph in both CSR and COO format. Unlike our approach, this approach requires modifications to the entire graph processing system stack to achieve performance on the chosen manycore. Chen et al. [24] propose a graph processing framework that splits execution between a CPU and Intel MIC. Their work focuses on exploiting SIMD performance on the MIC. Unlike our GraphIt backend, their API requires the programmer to simultaneously reason with the algorithm, the schedule, and the passing of messages between the CPU and MIC. Li et al. [61] also propose a manycore code generation framework for graph programs. They target the Intel Xeon Phi and leverage OpenMP for parallelism on the Xeon Phi. Their manycore specific optimizations are also limited to vectorization of compute and some data reuse analysis.

7 CONCLUSION

This dissertation presents solutions to the challenge of writing peformant code for graph applications on a manycore architecture. The optimizations presented in this work aim to better leverage the memory hierarchy and architectural features of a manycore in order to achieve high performance on graph applications. In this dissertation, I present a code generation backend to the GraphIt DSL that allows for flexibility in application development and optimization. I show how existing optimizations can be implemented on the HammerBlade manycore in order to improve work efficiency and decrease workload imbalance across cores. I present several manycore specific optimizations that target the memory hierarchy in order to more efficiently use memory bandwidth.

As the size of sparse graph data continues to grow, the importance of scalable, performant graph processing systems will only continue to increase. This work demonstrates ways that we can leverage the parallelism offered by emerging manycore architectures in order to maintain scalable performance on graph applications while also maintaining the flexibility and ease of programming offered through the use of the GraphIt DSL. As graph processing demands grow and users turn more and more to emerging architectures, I expect that future work will continue to build on these efforts to reduce programming complexity through code generation and increase performance through optimizations that target architectural features.

7.1 FUTURE WORK

There are several areas of exciting future work to be done in the space of graph processing on the HammerBlade manycore architecture. Much of the future work in this space focuses on better understanding application performance and on improving performance through continued memory system and data layout optimizations. I believe that the code generation framework that this dissertation presents will provide the flexibility and extensibility necessary to explore these areas of future work.

MULTI-SOURCE TRAVERSAL

Breadth-first search (BFS) is a common building block of many graph algorithms, and there has been substantial effort in optimizing BFS on parallel architectures. While quite a few algorithms such as SSSP execute BFS from a single source vertex, there also exist many algorithms which execute many different BFS traversals from different source vertices including all pairs shortest paths and betweenness centrality. There has been limited work optimizing this multi-souce BFS (MS-BFS) for parallel architectures [64, 110].

The approach outlined by Then et al. [110] leverages the properties of small-world networks that are present in many real-world graphs. They aim to optimize concurrent processing of a large number of BFS traversals in a single core. The authors observe that in graphs that exhibit small-world properties, the majority of vertices are discovered in just a few iterations. Further, they note that multiple BFS traversals over the same graph have a high likelihood of having overlapping sets of discovered vertices within the same iteration. Based on these observations, they create data structures that allow for combined accesses to the same vertices across multiple BFS instances. They use sets and set operations to concurrently execute many BFS traversals at the same time. In practice, these are implemented as bit operations over fixed width bit fields where the width of the bit field is the maximum number of concurrent BFS traversals that are supported. This approach reduces cache-contention and redundant work.

We believe that the manycore will be well suited to this style of MS-BFS processing, and that the use of bitfields will work well with the small caches and scratchpad memories present on the manycore. In addition, GraphIt has support for multi-source traversal through its `parallel_for` operator. However, this is a feature that we do not currently support in our HammerBlade code generation backend. An optimal implementation of this operator could open up our code generation framework to a larger class of graph applications.

PARAMETER TUNING AND CUSTOM DATA STRUCTURES

In graph processing, proposed optimizations are often tuned for one specific architecture. This means that when an optimization is ported to a new architecture, a performance study must be conducted to tune the optimization parameters to the specific features of the new architecture. We began a study of parameter tuning in this work with the exploration of hybrid traversal methods. From this we found that for optimal performance HammerBlade required smaller α and β values for the hybrid traversal heuristics introduced by Beamer et al.

Another area to evaluate would be tuning the Δ parameter in the delta-stepping SSSP algorithm. As discussed in Chapter 2, delta-stepping is a variant that was proposed to increase parallelism without dramatically increasing algorithmic work by introducing a bucketed priority queue [73]. The bucketed priority queue increases parallelism by relaxing the strict processing order imposed in Dijkstra's algorithm. Vertices are sorted into buckets within the priority queue of size Δ. Setting $\Delta = 1$ effectively turns delta-stepping into Dijkstra's and setting $\Delta = \infty$ turns it into Bellman-Ford.

It has been noted that the Δ parameter is sensitive to both input graph and architecture [10]. Beamer et al. found that the range of optimal Δ values for a given input graph is actually quite large [10]. However, there has been less study on how sensitive Δ values are to different architectures. A study of Δ values on road network and social network graphs to determine optimal values for the HammerBlade manycore could provide interesting insights into the class of ordered graph algorithms.

In addition, there is room to improve the implementation of the bucketed priority queue for a manycore architecture. Currently the priority queue is implemented on the host and all bucket updates must happen on the host in between each iteration. At the beginning of each iteration, the priority queue constructs a frontier to be dispatched to the manycore for processing. We believe that the manycore could be well suited to a device side implementation of the priority queue, and that moving the logic of the priority queue to the device could greatly increase performance.

7.2 TECHNICAL ACKNOWLEDGEMENTS

Portions of this dissertation work were partially supported by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement numbers FA8650-18-2-7863 and FA8650-18-2-7856; NSF grants SaTC-1563767, SaTC-1565446, and the DARPA/SRC JUMP ADA Center. This work leverages research and infrastructure created by the members of the Bespoke Silicon Group, spanning across accelerators ([14, 19, 37, 38, 100, 126]), open source hardware ([34, 105, 109]), RISC-V ([7, 28, 85, 89, 90, 113, 125]), Network-on-Chips ([50, 53, 86]) and multicore ([40–42, 53, 85, 102–104, 106–108, 114]).

BIBLIOGRAPHY

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. “General-purpose graphics processor architectures.” In: *Synthesis Lectures on Computer Architecture* 13.2 (2018), pp. 1–140.

[2] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. “Emptyheaded: A relational engine for graph processing.” In: *ACM Transactions on Database Systems (TODS)* 42.4 (2017), pp. 1–44.

[3] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. “Heterogeneous memory subsystem for natural graph analytics.” In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 134–145.

[4] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A Bader. “Scalable graph exploration on multicore processors.” In: *SC’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–11.

[5] Spiros N Agathos, Alexandros Papadogiannakis, and Vassilios V Dimakopoulos. “Targeting the parallella.” In: *European Conference on Parallel Processing*. Springer. 2015, pp. 662–674.

[6] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. “A scalable processing-in-memory accelerator for parallel graph processing.” In: *ACM SIGARCH Computer Architecture News* 43.3 (2016), pp. 105–117.

[7] Tutu Ajayi et al. “Celerity: An Open Source RISC-V Tiered Accelerator Fabric.” In: *HOTCHIPS*. Aug. 2017.

[8] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks.” In: *science* 286.5439 (1999), pp. 509–512.

[9] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. “Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads.” In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 373–386.

[10] Scott Beamer. “Understanding and improving graph algorithm performance.” PhD thesis. UC Berkeley, 2016.

[11] Scott Beamer, Krste Asanovic, and David Patterson. “Locality exists in graph processing: Workload characterization on an ivy bridge server.” In: *2015 IEEE International Symposium on Workload Characterization*. IEEE. 2015, pp. 56–65.

[12] Scott Beamer, Krste Asanović, and David Patterson. "Direction-Optimizing Breadth-First Search." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012. ISBN: 9781467308045.

[13] Scott Beamer, Krste Asanović, and David Patterson. "GAIL: The graph algorithm iron law." In: *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 2015, pp. 1–4.

[14] B. Beresini, S. Ricketts, and M.B. Taylor. "Unifying manycore and FPGA processing with the RUSH architecture." In: *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*. 2011, pp. 22–28.

[15] Pavel Berkhin. "A survey on PageRank computing." In: *Internet mathematics* 2.1 (2005), pp. 73–120.

[16] Vladimir Boginski, Sergiy Butenko, and Panos M. Pardalos. "Statistical analysis of financial networks." In: *Computational Statistics & Data Analysis* 48.2 (2005), pp. 431–443. ISSN: 0167-9473. DOI: https://doi.org/10.1016/j.csda.2004.02.004. URL: http://www.sciencedirect.com/science/article/pii/S0167947304000258.

[17] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks." In: *Proceedings of the 20th international conference on World Wide Web*. 2011, pp. 587–596.

[18] Paolo Boldi and Sebastiano Vigna. "The webgraph framework I: compression techniques." In: *Proceedings of the 13th international conference on World Wide Web*. 2004, pp. 595–602.

[19] Ajay Brahmakshatriya, Emily Furst, Victor A Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael B Taylor, et al. "Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures." In: *Appears in the Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA), 2021*. 2021.

[20] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. "Compiling Graph Applications for GPU s with GraphIt." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 248–261.

[21] Ulrik Brandes. "A faster algorithm for betweenness centrality." In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[22] Aydin Buluç and Kamesh Madduri. "Parallel breadth-first search on distributed memory systems." In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12.

[23] Daniel Chavarria-Miranda, Mahantesh Halappanavar, and Ananth Kalyanaraman. "Scaling graph community detection on the tilera many-core architecture." In: *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE. 2014, pp. 1–11.

[24] Linchuan Chen, Xin Huo, Bin Ren, Surabhi Jain, and Gagan Agrawal. "Efficient and simplified parallel graph processing over cpu and mic." In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 819–828.

[25] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. "Fpgp: Graph processing framework on fpga a case study of breadth-first search." In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 105–110.

[26] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. "Foregraph: Exploring large-scale graph processing on multi-fpga architecture." In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 217–226.

[27] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. "Graphh: A processing-in-memory architecture for large-scale graph processing." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.4 (2018), pp. 640–653.

[28] S. Davidson et al. "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips." In: *IEEE Micro* 38.2 (2018), pp. 30–41.

[29] Timothy A Davis and Yifan Hu. "The University of Florida sparse matrix collection." In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[30] Camil Demetrescu, Andrew Goldberg, and David Johnson. *9th DIMACS implementation challenge - shortest paths*. http://www.dis.uniroma1.it/challenge9/.

[31] Mehmet Deveci, Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. "Parallel graph coloring for manycore architectures." In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, pp. 892–901.

[32] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. "Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time." In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 1775–1784. ISBN: 9781450356398. DOI: 10.1145/3178876.3186183. URL: https://doi.org/10.1145/3178876.3186183.

[33] Nina Engelhardt and Hayden Kwok-Hay So. "Gravf: A vertex-centric distributed graph processing framework on fpgas." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–4.

[34] Hadi Esmaeilzadeh and Michael Bedford Taylor. "Open Source Hardware: Stone Soups and Not Stone Satues, Please." In: *SIGARCH Computer Architecture Today*. Dec. 2017.

[35] Wilson WL Fung and Tor M Aamodt. "Thread block compaction for efficient SIMT control flow." In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE. 2011, pp. 25–36.

[36] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. "Abelian: A compiler for graph analytics on distributed, heterogeneous platforms." In: *European Conference on Parallel Processing*. Springer. 2018, pp. 249–264.

[37] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. "GreenDroid: A Mobile Application Processor for a Future of Dark Silicon." In: *HOTCHIPS*. 2010.

[38] N. Goulding-Hotta et al. "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future." In: *Micro, IEEE* (Mar. 2011), pp. 86–95.

[39] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. "Making pull-based graph processing performant." In: *ACM SIGPLAN Notices* 53.1 (2018), pp. 246–260.

[40] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "DR-SNUCA: An Energy-Scalable Dynamically Partitioned Cache." In: *International Conference on Computer Design (ICCD)*. 2013.

[41] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "Time Cube: A Manycore Embedded Processor with Interference-Agnostic Progress Tracking." In: *International Conference On Embedded Computer Systems: Architectures, Modeling And Simulation (SAMOS)*. 2013.

[42] Anshuman Gupta, Jack Sampson, and Michael Bedford Taylor. "QualityTime: A Simple Online Technique for Quantifying Multicore Execution Efficiency." In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014.

[43] Linley Gwennap. "Adapteva: More flops, less watts." In: *Microprocessor Report* 6.13 (2011), pp. 11–02.

[44] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13.

[45] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. “Green-Marl: a DSL for easy and efficient graph analysis.” In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 2012, pp. 349–362.

[46] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. “Efficient parallel graph exploration on multi-core CPU and GPU.” In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 78–88.

[47] JEDEC. In: (Jan. 2020). URL: https://www.jedec.org/standards-documents/docs/jesd235a.

[48] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. “A scalable architecture for ordered parallelism.” In: *Proceedings of the 48th International Symposium on Microarchitecture*. 2015, pp. 228–241.

[49] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. “In-datacenter performance analysis of a tensor processing unit.” In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[50] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. “Ruche Networks: Wire-Maximal, No-Fuss NoCs.” In: *NOCS*. 2020.

[51] Matt J Keeling and Ken TD Eames. “Networks and epidemic models.” In: *Journal of the Royal Society Interface* 2.4 (2005), pp. 295–307.

[52] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. “CuSha: vertex-centric graph processing on GPUs.” In: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 2014, pp. 239–252.

[53] Jason Kim, Michael B. Taylor, Jason Miller, and David Wentzlaff. “Energy Characterization of a Tiled Architecture Processor with On-Chip Networks.” In: *International Symposium on Low Power Electronics and Design (ISLPED)*. Aug. 2003.

[54] Christian Kohlschütter, Paul-Alexandru Chirita, and Wolfgang Nejdl. “Efficient parallel computation of pagerank.” In: *European Conference on Information Retrieval*. Springer. 2006, pp. 241–252.

[55] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. “What is Twitter, a social network or a news media?” In: *Proceedings of the 19th international conference on World wide web*. 2010, pp. 591–600.

[56] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. “GraphChi: Large-Scale Graph Computation on Just a PC.” In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 31–46. ISBN: 9781931971966.

[57] Oliver Lehmberg, Robert Meusel, and Christian Bizer. "Graph Structure in the Web: Aggregated by Pay-Level Domain." In: *Proceedings of the 2014 ACM Conference on Web Science*. WebSci '14. Bloomington, Indiana, USA: Association for Computing Machinery, 2014, pp. 119–128. ISBN: 9781450326223. DOI: 10.1145/2615569.2615674. URL: https://doi.org/10.1145/2615569.2615674.

[58] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. "Kronecker graphs: an approach to modeling networks." In: *Journal of Machine Learning Research* 11.2 (2010).

[59] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data.

[60] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication." In: *European conference on principles of data mining and knowledge discovery*. Springer. 2005, pp. 133–145.

[61] Da Li, Srimat Chakradhar, and Michela Becchi. "Grapid: A compilation and runtime framework for rapid prototyping of graph applications on many-core processors." In: *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2014, pp. 174–182.

[62] Gushu Li, Guohao Dai, Shuangchen Li, Yu Wang, and Yuan Xie. "GraphIA: an in-situ accelerator for large-scale graph processing." In: *Proceedings of the International Symposium on Memory Systems*. 2018, pp. 79–84.

[63] Shang Li, Rommel Sánchez Verdejo, Petar Radojković, and Bruce Jacob. "Rethinking cycle accurate DRAM simulation." In: *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 184–191.

[64] Hang Liu, H Howie Huang, and Yang Hu. "ibfs: Concurrent breadth-first search on gpus." In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 403–416.

[65] Igor Loi and Luca Benini. "An efficient distributed memory interface for many-core platform with 3D stacked DRAM." In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE. 2010, pp. 99–104.

[66] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud." In: *Proc. VLDB Endow.* 5.8 (Apr. 2012), pp. 716–727. ISSN: 2150-8097. DOI: 10.14778/2212351.2212354. URL: https://doi.org/10.14778/2212351.2212354.

[67] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. "GraphLab: A New Framework for Parallel Machine Learning." In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence.* UAI'10. Catalina Island, CA: AUAI Press, 2010, pp. 340–349. ISBN: 9780974903965.

[68] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. "Challenges in parallel graph processing." In: *Parallel Processing Letters* 17.01 (2007), pp. 5–20.

[69] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* 2010, pp. 135–146.

[70] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. "GraphSSD: graph semantics aware SSD." In: *Proceedings of the 46th International Symposium on Computer Architecture.* ACM. 2019, pp. 116–128.

[71] Adam McLaughlin and David A. Bader. "Scalable and High Performance Betweenness Centrality on the GPU." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 572–583. ISBN: 9781479955008. DOI: 10.1109/SC.2014.52. URL: https://doi.org/10.1109/SC.2014.52.

[72] Albert Menkveld. "High-Frequency Trading as Viewed through an Electron Microscope." In: *Financial Analysts Journal* 74 (Feb. 2018), pp. 1–8. DOI: 10.2469/faj.v74.n2.1.

[73] Ulrich Meyer and Peter Sanders. "Δ-stepping: a parallelizable shortest path algorithm." In: *Journal of Algorithms* 49.1 (2003), pp. 114–152.

[74] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. "Measurement and analysis of online social networks." In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement.* 2007, pp. 29–42.

[75] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling." In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE. 2018, pp. 1–14.

[76] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. "Introducing the graph 500." In: *Cray Users Group (CUG)* 19 (2010), pp. 45–74.

[77] Brandon Myers and Brandon Holt. "Do we need a crystal ball for task migration?" In: *Presented as part of the 4th {USENIX} Workshop on Hot Topics in Parallelism.* 2012.

[78] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. “Latency-Tolerant Software Distributed Shared Memory.” In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’15. Santa Clara, CA: USENIX Association, 2015, pp. 291–305. ISBN: 9781931971225.

[79] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. “A lightweight infrastructure for graph analytics.” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 456–471.

[80] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. “Tigr: Transforming irregular graphs for gpu-friendly graph processing.” In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 622–636.

[81] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. “Energy efficient architecture for graph analytics accelerators.” In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2016, pp. 166–177.

[82] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank citation ranking: Bringing order to the web*. Tech. rep. Stanford InfoLab, 1999.

[83] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. “Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures.” In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–14.

[84] Jose B Pereira-Leal, Anton J Enright, and Christos A Ouzounis. “Detection of functional modules from protein interaction networks.” In: *PROTEINS: Structure, Function, and Bioinformatics* 54.1 (2004), pp. 49–57.

[85] D. Petrisko et al. “BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs.” In: *IEEE Micro* (July 2020), pp. 93–102.

[86] Daniel Petrisko, Chun Zhao, Scott Davidson, Paul Gao, Dustin Richmond, and Michael Bedford Taylor. “NoC Symbiosis.” In: *NOCS*. 2020.

[87] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.” In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[88] Carl Ramey. “Tile-gx100 manycore processor: Acceleration interfaces and architecture.” In: *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE. 2011, pp. 1–21.

[89] A. Rovinski et al. “A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS.” In: *2019 Symposium on VLSI Circuits*. 2019, pp. C30–C31.

[90] A. Rovinski et al. "Evaluating Celerity: A 16-nm 695 Giga-RISC-V Instructions/s Manycore Processor With Synthesizable PLL." In: *IEEE Solid-State Circuits Letters* 2.12 (2019), pp. 289–292.

[91] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing." In: *Proc. VLDB Endow.* 11.4 (Dec. 2017), pp. 420–431. ISSN: 2150-8097. DOI: 10.1145/3164135.3164139. URL: https://doi.org/10.1145/3164135.3164139.

[92] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. "Navigating the maze of graph analytics frameworks using massive graph datasets." In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 979–990.

[93] Albert Segura, Jose-Maria Arnau, and Antonio González. "SCU: a GPU stream compaction unit for graph processing." In: *Proceedings of the 46th International Symposium on Computer Architecture*. ACM. 2019, pp. 424–435.

[94] Jiwon Seo, Stephen Guo, and Monica S Lam. "SociaLite: Datalog extensions for efficient social network analysis." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 278–289.

[95] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. "GraphJet: Real-Time Content Recommendations at Twitter." In: *Proc. VLDB Endow.* 9.13 (Sept. 2016), pp. 1281–1292. ISSN: 2150-8097. DOI: 10.14778/3007263.3007267. URL: https://doi.org/10.14778/3007263.3007267.

[96] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. "Graph processing on GPUs: A survey." In: *ACM Computing Surveys (CSUR)* 50.6 (2018), pp. 1–35.

[97] Julian Shun and Guy E Blelloch. "Ligra: a lightweight graph processing framework for shared memory." In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 135–146.

[98] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. "High-performance graph analytics on manycore processors." In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 17–27.

[99] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. "Graphgrind: Addressing load imbalance of graph partitioning." In: *Proceedings of the International Conference on Supercomputing*. 2017, pp. 1–10.

[100] Steven Swanson and Michael Taylor. "GreenDroid: Exploring the next evolution for smartphone application processors." In: *IEEE Communications Magazine*. Mar. 2011.

[101] Lubos Takac and Michal Zabovsky. “Data analysis in public social networks.” In: *International scientific conference and international workshop present day trends of innovations*. Vol. 1. 6. 2012.

[102] M. B. Taylor et al. “Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams.” In: *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 2004, pp. 2–13.

[103] MB Taylor, J Kim, J Miller, F Ghodrat, B Greenwald, P Johnson, W Lee, A Ma, N Shnidman, V Strumpen, et al. “The Raw processor-a scalable 32-bit fabric for embedded and general purpose computing.” In: *Proceedings of Hot Chips XIII*. 2001.

[104] Michael Taylor. “Tiled Microprocessors.” PhD thesis. Massachusetts Institute of Technology, 2007.

[105] Michael B. Taylor. “BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design.” In: *Design Automation Conference*. June 2018.

[106] Michael B. Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. “Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures.” In: *International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2003.

[107] Michael B. Taylor et al. “The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs.” In: *IEEE Micro*. Mar. 2002.

[108] Michael B. Taylor et al. “A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network.” In: *IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2003.

[109] Michael Bedford Taylor. “Your agile open source HW stinks (because it is not a system).” In: *ICCAD*. 2020.

[110] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. “The more the merrier: Efficient multi-source graph traversal.” In: *Proceedings of the VLDB Endowment* 8.4 (2014), pp. 449–460.

[111] Masashi Tsubaki, Kentaro Tomii, and Jun Sese. “Compound–protein interaction prediction with end-to-end learning of neural networks for graphs and sequences.” In: *Bioinformatics* 35.2 (2019), pp. 309–318.

[112] Vladimir I. Ulyantsev, Sergey V. Kazakov, Veronika B. Dubinkina, Alexander V. Tyakht, and Dmitry G. Alexeev. “MetaFast: fast reference-free graph-based comparison of shotgun metagenomic data.” In: *Bioinformatics* 32.18 (June 2016), pp. 2760–2767. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw312. URL: https://doi.org/10.1093/bioinformatics/btw312.

[113] Luis Vega and Michael Bedford Taylor. " RV-IOV: Tethering RISC-V Processors via Scalable I/O Virtualization." In: *CARRV*. 2017.

[114] Elliot Waingold et al. "Baring it all to Software: Raw Machines." In: *IEEE Computer*. Sept. 1997.

[115] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. "Gunrock: A high-performance graph processing library on the GPU." In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2016, pp. 1–12.

[116] Duncan J Watts and Steven H Strogatz. "Collective dynamics of 'small-world' networks." In: *nature* 393.6684 (1998), pp. 440–442.

[117] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. "Graph processing on GPUs: Where are the bottlenecks?" In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 140–149.

[118] Carl Yang, Aydin Buluc, and John D Owens. "GraphBLAST: A high-performance linear algebra-based graph framework on the GPU." In: *arXiv preprint arXiv:1908.01407* (2019).

[119] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth." In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213.

[120] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. "An efficient graph accelerator with parallel data conflict management." In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 2018, pp. 1–12.

[121] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. "A scalable distributed parallel breadth-first search algorithm on BlueGene/L." In: *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE. 2005, pp. 25–25.

[122] Kaiyuan Zhang, Rong Chen, and Haibo Chen. "NUMA-aware graph-structured analytics." In: *Proceedings of the 20th ACM SIGPLAN symposium on principles and practice of parallel programming*. 2015, pp. 183–193.

[123] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. "Optimizing ordered graph algorithms with GraphIt." In: *arXiv preprint arXiv:1911.07260* (2019).

[124] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. "GraphIt: A High-Performance DSL for Graph Analytics." In: *arXiv preprint arXiv:1805.00923* (2018).

[125] Ritchie Zhao et al. "Celerity: An Open Source RISC-V Tiered Accelerator Fabric." In: *7th RISC-V Workshop*. 2017.

[126] Qiaoshi Zheng, Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Bedford Taylor, and Jack Sampson. "Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon." In: *Transactions on Embedded Computing Systems (TECS)* (Mar. 2014).

[127] Jianlong Zhong and Bingsheng He. "Medusa: Simplified graph processing on GPUs." In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2013), pp. 1543–1552.

[128] Shijie Zhou, Charalampos Chelmis, and Viktor Prasanna. "Accelerating large-scale single-source shortest path on FPGA." In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE. 2015, pp. 129–136.

[129] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. "Optimizing memory performance for fpga implementation of pagerank." In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2015, pp. 1–6.

[130] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. "High-throughput and energy-efficient graph processing on FPGA." In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2016, pp. 103–110.

[131] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. "Gemini: A computation-centric distributed graph processing system." In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 301–316.

[132] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. "GraphQ: Scalable PIM-Based Graph Processing." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2019, pp. 712–725.