Title

Hardware Architectures for Scalable Graph Processing

Author

Fariborz, Marjan

Publication Date

2023

Peer reviewed|Thesis/dissertation

Hardware Architectures for Scalable Graph Processing

By

Marjan Fariborz
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

S.J. Ben Yoo, Chair

Venkatesh Akella

Jason Lowe-Power

Committee in Charge

2023

To My Parents

Contents

Abstract

Graph analytics has become a popular method for understanding the relationships between data collected from various sources, such as social media, sensor feeds, and scientific data. This allows analysts to identify patterns in the data and answer difficult questions that were previously unanswerable. A more complete picture of the problem can be understood by understanding the complex relationships between different data feeds. However, due to the sparse nature of real-world graphs, these applications tend to show random memory access patterns and low locality. Current computing and memory systems are optimized for applications that exhibit high locality by utilizing deep cache hierarchy and large memory access support. Throughout this work, we show the limitations of running graph workloads on general-purpose systems in terms of both performance and scalability. This work presents scalable solutions for processing graph applications. This dissertation presents the following case studies:

First, designing a graph processing system that can scale to graph sizes that are orders of magnitude larger than what is possible on a single accelerator requires a careful codesign of accelerator memory bandwidth and capacity, the interconnect bandwidth between accelerators, and the overall system architecture. We present a high-level bottleneck-analysis model for the design and evaluation of scalable and balanced accelerators for graph processing. We further studied the scalability limitations of previous graph accelerators and designed an accelerator that overcame those limitations. We used the analytical model to capture the system-level requirements, such as the memory systems.

Second, we analyzed the network requirements of graphs as we scaled up the system to larger nodes and more memory. We demonstrate that the traffic pattern between graph processing nodes has a uniform random distribution, and while the bandwidth requirements are not significant, having a low-diameter interconnect can improve the performance.

The third part of this work proposes a new memory system that is optimized for applications with random memory patterns. Our goal was to create a new memory that reduces the amount of contention on the data path since contention on the shared resources was the main source of performance bottleneck. By utilizing 2.5D/3D integration and multi-wavelength (WDM) silicon photonic (SiPh) technologies, we create a new memory system with low memory access latency and

latency variation in addition to 4× higher bandwidth compared to high bandwidth memories such as HBM2 given the same amount of capacity.

Acknowledgments

I would like to express my heartfelt gratitude to Professor S.J. Ben Yoo for giving me the opportunity to be part of his diverse research team at UC Davis.

I would like to extend my deepest appreciation to my esteemed committee members, Professor Venkatesh Akella and Jason Lowe-Power. Their invaluable guidance, expertise, patience, and time have profoundly influenced my growth as a researcher. The insightful discussions I shared with them have been instrumental in shaping my academic journey.

I am also indebted to my colleagues at UC Davis, Mahyar Samani, Dr. Pouya Fotouhi, and Dr. Roberto Proietti for their invaluable contributions. Collaborating with such exceptional scientists and wonderful individuals has been a pleasure.

Finally, I wish to extend my deepest appreciation to my parents and my brother, Mehri, Shahriar, and Jamshid. Your unwavering belief in me, your support, and your encouragement have been pivotal in every success I have achieved. I consider myself incredibly fortunate to have in my life.

CHAPTER 1

Introduction

Large-scale data analytics is fueling breakthroughs in a broad range of applications such as logistics and transportation, advertising, security, understanding social behavior, tax policy, drug discovery, cosmology, etc. The problem sizes in these applications are increasing exponentially in volume, variety, and complexity. Data for these applications is often sparse and stored in structures with poor data locality. These applications are represented in terms of sparse matrix dense vector operations and can be modeled by graph operations. Processing and storing these large-scale data can no longer be ignored as in many scenarios they account for a significant fraction of the overall execution time and consume a significant portion of machine resources. Thus, we need to understand the characteristics of these workloads.

The memory access patterns are very irregular (especially in large-scale graph data analytics) which results in poor data locality and makes the deep hierarchy of caches used in the current computing systems ineffective. In addition, moving the data through a multilevel cache hierarchy consumes a significant amount of energy. Graph analytics applications are often memory-bound with little data reuse and low spatial and temporal locality with fine-grain random accesses to memory. Due to these characteristics, the current general-purpose computing systems are not efficient. These inefficiencies include:

- Graph traversals often require many memory accesses relative to only small amounts of computation. Ham et al. [40] shows that only 6% of instructions are dedicated to computation and the rest 94% of instructions are used for traversing the graph and loading the graph-specific arguments.
- In graph applications data is accessed at a fine granularity which is smaller than the memory access granularity. This discrepancy between the data access demand and memory access granularity causes a significant amount of off-chip memory bandwidth waste.
- Ineffective on-chip memory usage caused by lack of temporal and spatial locality in graph workloads. Figure 1.1 illustrates workloads from both benchmark suits benefit similarly

from L1 data caches with average hit rates of 99% for NPB and 95% for GAP. However, the lower levels of the cache hierarchy are much less useful for these large workloads. The irregular graph workloads (GAP) show even more ineffective use of cache than the HPC workloads (NPB) with below a 5% hit ratio at the L2 cache. As the size of the workloads grows in the future large caches will continue to be wasteful in terms of area and power.

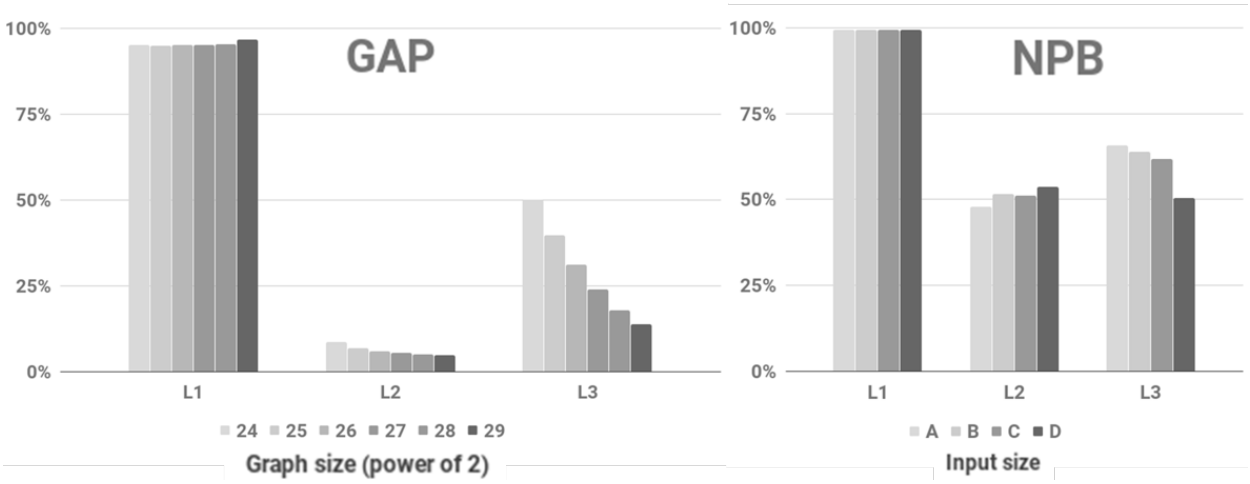


FIGURE 1.1. Cache hit rates (average values of all workloads) for NPB and GAP benchmark suites. We used input sets A to D for workloads in NPB, and graphs with 224 to 229 vertices for GAP kernels.

With advances in 2.5D/3D integration and multi-wavelength (WDM) silicon photonic (SiPh) technologies over the past decade, it is time to rethink and redesign computing systems and memory subsystems to make them more amenable to applications with random memory access. Additionally, we aim to design new computing systems that can scale to meet the demands of ever-growing data.

One approach to reducing the data movement energy for these sparse applications is to flatten both the memory hierarchy and the network interconnect. By flattening the memory hierarchy as far as possible by eliminating the cache hierarchy to realize a memory system that exhibits low and predictable latency. Photonic networks with devices such as high port count chip-scale AWGRs can be used to flatten the network topology (basically reduce the hop count) by increasing the radix of the intermediate switches and/or adding auxiliary direct links (express links) to create a custom topology that matches the unique dataflow of the application during different phases of execution. This can unleash new computing architectures that are not only more energy efficient but also easier to program.

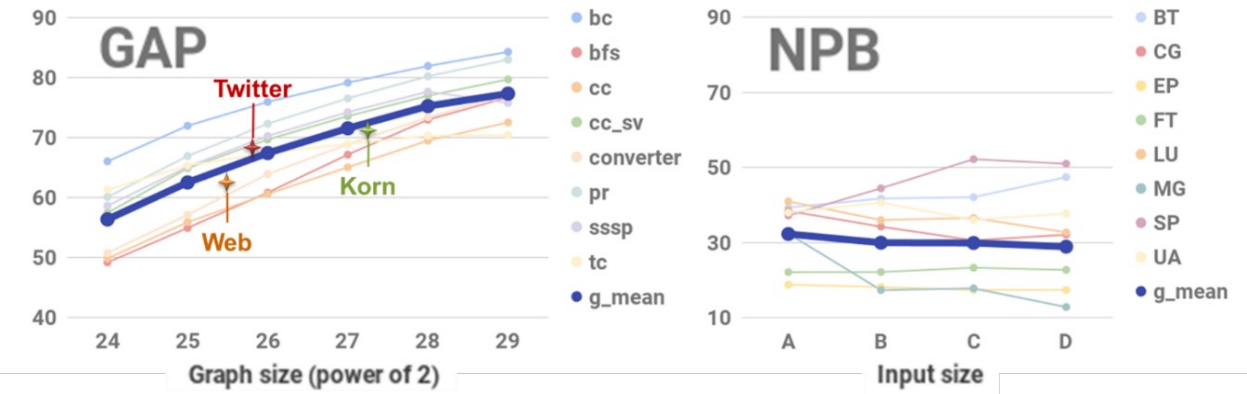


FIGURE 1.2. Target DRAM latency (in nano-seconds) to achieve the same AMAT in a system with no L2/L3 caches for GAP and NPB workloads. Using workloads from GAP benchmark suite [17] The results of this analysis for Web [30] with 50.6 M nodes, Twitter [49] with 61.6 M nodes, and Korn [3] with 134.2 M nodes are presented as single data points (geometric means across different kernels).

To improve the performance of these sparse applications, we need to rethink the architecture of the memory subsystem. By reducing contention for the memory data access for sparse workloads, we can reduce the memory access latency. Also, by taking advantage of WDM silicon photonic interconnects, we can increase the memory access bandwidth and reduce the data movement energy consumption. The hypothesis is that this low latency and high bandwidth memory sub-system allows us to ultimately flatten the memory system. Therefore, we codesigned the processor, the DRAM microarchitecture, and the memory controller by taking advantage of 2.5D/3D integration with WDM silicon photonic interconnects.

Figure 1.2 presents an initial study of the required memory latency required to remove these ineffective L2 and L3 caches. As our initial analysis, we determine the desired latency of DRAM for achieving the exact same performance (average memory access time, AMAT) with no L2 and L3 caches. We conservatively assume 90ns memory latency (unloaded local DRAM access time), 12 cycles L2 hit, and 34 cycle L3 hit latencies. As Figure 1.2 shows, workloads in NPB require a DRAM access latency of 30ns (66% improvement goal) for a system with no L2/L3. For GAP kernels using synthetic graphs, the ineffectiveness of caches translates into a modest required DRAM latency of 77ns (15% goal) for a system with no L2/L3.

To support the fast processing of the growing data structures, we create a large-scale graph accelerator. Designing a graph processing system that can scale to graph sizes that are orders of

magnitude larger than what is possible on a single accelerator. However, this system requires a careful codesign of accelerator memory bandwidth and capacity, the interconnect bandwidth between accelerators, and the overall system architecture. We present a high-level bottleneck-analysis model for the design and evaluation of scalable and balanced accelerators for graph processing. This model chooses the right mix of different memory types, network topology, network bisection bandwidth, and system-level architecture to match the access patterns and capacity requirements of different data structures for a given graph and a performance target. Designers can use this model to help them understand the bandwidth, capacity, and microarchitectural needs of their system to achieve desirable performance. This analytical model can also be used to evaluate the performance of emerging memory and interconnect technologies in graph accelerators.

While the proposed analytical model helps with understanding the system requirements of designing scalable graph accelerators, we need the accelerators' processing elements to be able to process large graphs with high performance. Previous accelerators improved performance and reduced memory access costs by designating the data structure with random access patterns (vertex information) and storing this data structure on-chip. However, given that the amount of on-chip memory by definition is limited, most of the designs reported in recent literature rely on partitioning the graph into smaller slices and time-share the on-chip memory between different graph partitions (temporal partitioning). Although effective for relatively small graphs with a few million vertices, their performance tends to degrade for larger graphs, making them unscalable and inefficient for processing large graphs. Due to the limitations of temporal partitioning and the dependence of previous graph accelerators on temporal partitioning to exploit locality and achieve high performance, we propose a new graph accelerator node that doesn't rely on locality to achieve high performance. Instead, this new accelerator relies on the on-chip memory to hide the off-chip random memory access latency.

This proposed graph accelerator is composed of multiple processing elements (PEs). Each processing element processes a subset of vertices and edges that are stored in the off-chip memory designated for each processing element. Each processing element can process any graph size without temporal partitioning as long as it fits on its off-chip memory. To scale to significantly large graphs, we propose scaling out the number of PEs, partitioning the graph, and assigning each partition to

a designated PE. With the new graph accelerator's processing element design, we can utilize the analytical model to help us understand the system needs of designing a large-scale graph accelerator.

The organization of this dissertation is as follows. Chapter 2 presents background on the graph application, background on previous studies on accelerating graph workloads, and their limitations. of graph with previous graph accelerators and their scalability challenges. Chapter 3 presents the overview of our proposed graph accelerator and an analytical model that can be used to understand the system-level requirement to scale to a specific graph size. Chapter 4 presents the network requirements of scaling to large graphs and explores the traffic pattern on the system. Chapter 5 discusses the architecture of a state-of-the-art memory system with low latency and high throughput customized for applications with random memory access patterns. At last, Chapter 6 presents a summary of the work presented in this dissertation and illustrates future perspectives.

CHAPTER 2

Background and Related Work on Graph Workloads

2.1. Background on Graph Workloads

Graph processing frameworks follow the **vertex-centric** or **edge-centric** paradigm for sequencing their computation. In the edge centric model, edges are streamed into the processor through vertex indices. The processor needs to read both source and destination addresses. This programming model suffers from poor spatial locality when reading vertices since there is a little chance of both source and destination vertices being stored in consecutive memory locations.

The vertex-centric paradigm is more popular than edge-centric due to its implementation simplicity and lower number of redundant vertex reads. The vertex-centric paradigm is designed from a vertex's perspective, i.e., a vertex property value is updated by a computation based on the property of its neighbors [61].

Vertex-centric computation model follows one of two approaches: **pull** or **push**. In the pull approach, each vertex reads the properties of all its incoming neighbors and updates their values. Thus, it involves random and redundant reads of the neighboring vertices, resulting in poor memory bandwidth utilization and wasted parallelism due to memory latency. On the other hand, the push approach each vertex performs a read-modify-write operation for each of its outgoing neighbors.

Vertices can be scheduled using **bulk-synchronous** or **asynchronous** model. In the bulk-synchronous parallel model (BSP) [90] the workload is abstracted into iterations. During each iteration vertex values get updated and new vertices get activated for the next iteration. The asynchronous programming model [91] there are no definition of iteration which allows asynchronous processing of vertices and thus substantially increasing available parallelism.

Graph workloads use two main data structures to represent a graph: 1) an array-like data structure containing vertex specific information, and 2) an array-like data structure containing edge specific information. We refer to vertex and edge data structures as *worklist* and *edgelist*

respectively. While the worklist represents the property of each vertex in the graph, the edgelist represents the structure and connectivity or the relationship of the vertices in the graph.

In addition, graph workloads use a tertiary data structure that contains vertices that need to be processed. We refer to these vertices as active and this data structure as *active set* or *frontier*.

Traditionally, graphs are processed on CPUs and GPUs. Libraries such as Galois [73], Ligra [81], GAPBS [17], Gunrock [93] are designed to optimize graph applications on CPUs and GPUs. The goal of these libraries is to improve load-balancing and improving work efficiency by exploiting locality. Galois [73] provides flexibility in the programming model by using different operator abstraction which allows them to support topology aware priority scheduling. Similar to Galois, Ligra [81] uses a load-balancing strategy that is based on CilkPlus [50] to support multicore servers. Ligra uses hybrid data representation for sparse and dense active sets. Both Galois and Ligra are the highest performing lightweighted and generalized software framework for shared-memory machines. Gunrock is a GPU-based library for graph processing. It allows programmers to develop graph primitives to GPU systems. Gurock improve the performance of graph analytic by improving the load-balancing and improving memory efficiency. Almost all these platform support the hybrid push-pull based traversal based on the sparsity of their active sets.

2.2. Related Work

In this chapter, we present previous works that focused on improving the performance of graph workloads. In the past decade, there has been a plethora of work on graph accelerators [10, 14, 40, 59, 63, 64, 66, 68, 75, 76]. Studies focusing on enhancing the performance of graph applications either through software improvements on general-purpose systems or hardware accelerators. The primary goal of these studies has been to reduce data movement either by moving to compute near data or optimizing locality.

State-of-the-art graph accelerators such as PolyGraph and GraphPulse [29, 75] utilize on-chip memory to enhance locality. These accelerators use temporal partitioning to scale graphs larger than their on-chip memory capacity and spatial partitioning for graphs larger than their off-chip memory capacity.

In *temporal partitioning*, on-chip memory can be shared by different partitions sequentially over time, while in *spatial partitioning*, multiple accelerator chips can house all slices while an interconnection network streams inter-slice events in real-time.

Other graph accelerators use a traditional memory hierarchy with multi-levels of private and shared caches. However, these architectures suffer from an overwhelming amount of data movement due to poor data-reuse in graph workloads. As a result, cache thrashing leads to more than 50% of all memory accesses missing in the last level cache [59].

Accelerators such as PolyGraph [29], GraphPulse [75], and ScalaGraph [99] utilize high-bandwidth off-chip memory for processing edges while storing partitions of vertices on-chip. This approach enables these accelerators to achieve high performance by leveraging the benefits of high-bandwidth memory. However, we demonstrate that larger graphs with more partitions can adversely impact the performance.

In this chapter, we review the literature related to accelerators, specifically PIM-based accelerators and hardware-based graph accelerators. Then, we focus on the limitations of these previous works for processing large graphs.

2.2.1. Hardware Accelerators. Previous hardware graph accelerators improve performance by creating customized pipelining mechanisms [10, 14, 40, 63, 64, 68, 75, 76] or by decoupling data access and computation [59, 66, 100]. All of these studies focus on improving off-chip memory efficiency for graph processing.

Mukkara et al. [62] reduces random off-chip memory accesses using a hardware-accelerated traversal scheduler that allows the system to improve locality. GraphDyns [98] represents a new programming model to extract data dependencies in graph processing dynamically. It uses a load-balanced scheduling mechanism, and a specialized prefetcher for off-chip edge data access. ScalaGraph [99] proposes the use of high bandwidth memory and a distributed memory assignment to improve the edge throughput while eliminating the atomic memory accesses. Chronos [10] avoids temporal partitioning and uses an on-chip cache and speculative execution model to avoid coherency overheads. Ozdal [68] defines a custom cache corresponding to each different data type (edge indices, edge data, vertex info, vertex data) of each graph object type to reduce the data access energy. Graphlily [43], ScalaGraph [99], and PolyGraph [29] utilized HBM memory for higher bandwidth access to the edge memory. However, in all these designs, HBM was used only to

improve the edge bandwidth throughput. Dalorex [67] stores the entire data in the on-chip memory, and they scale their performance significantly as they scale their processing tiles. However, for them to maintain their performance, they require to have a significant amount of on-chip memory.

PolyGraph [29] proposes a flexible accelerator that supports multiple variants, including non-slice, sliced, synchronous, and asynchronous modes. In their experiments, the non-slice variant is observed to be effective only for small graphs or phases of the workload with a low number of active vertices. For instance, they propose switching to the non-sliced variant at the beginning/ending iterations of the Breath-First Search algorithm (BFS).

Dalorex [67] eliminated the off-chip memory access by storing the entire graph in on-chip memory. The on-chip bandwidth dictates their performance, allowing them to achieve much higher performance compared to PolyGraph. To scale to larger graphs (beyond what is supported by their on-chip memory), they spatially partition the graph and increase the number of accelerator cores to support large graphs. They require gigabytes of on-chip memory to store large-scale graphs without temporal partitioning. Similar to Cerebras WSE-2 Delorax consists of multiple graph processing nodes and gigabytes of on-chip memory that are uniformly distributed between these cores. Leaving only kilobytes of capacity for each graph processing nod. Therefore, each node only processes a small fraction of the graph. In addition to scaling to terabyte or petabyte graphs, Dalorex is required to perform temporal partitioning on the disk or scale the number of graph processing boards. The size of these accelerators is quite large (in a 16×16 Dalorex with 67.2MiB on-chip memory capacity, the area is (305mm2)), which results in an interconnection with long latency making this off-chip accesses expensive.

It should be mentioned that in all these studies, the goal has been to improve performance by reducing off-chip memory.

2.2.2. PIM-based Accelerators. A promising solution to remove the memory wall challenges in the graph workloads is to use processing in the memory (PIM). PIM-based acceleration is possible due to the recent advancement of the 3D integration technology that facilitates stacking logic and memory dies in a single package. PIM-based architectures provide high levels of parallelism, low memory access latency, and large aggregate memory bandwidth.

Some studies rely on emerging memory technologies such as ReRAMs [12] to perform computing in memory in addition to storing data [23, 82, 106]. Other PIM-based architectures use 3D

stacked memory technologies, such as Hybrid Memory Cube (HMC) [72] to eliminate the irregular data movement [11, 102, 110]. Tesseract [11] is a PIM-based graph processing architecture that supports vertex programming model with architectural primitives to enable inter-memory cube communication. Graphq [82] proposed the first multi-node PIM-based graph processing architecture built on the recent work Tesseract [11]. Graphq uses a two-phase programming model enabling efficient partitioning of graphs between different memory cubes.

These accelerators propose using Hybrid Memory Cube and non-volatile memories such as ReRAM to store their randomly accessed data structures. Whereas in our proposed scalable graph accelerator, depending on the capacity, performance, and available resources, the architect can decide on the vertex or edge memories.

2.3. Limitations of Previous Work

Prior accelerator designs rely on improving spatial and temporal locality. In previous accelerators, the graph is divided into temporal slices that fit into on-chip memory and they time share the limited on-chip memory for all the temporal slices. Temporal partitioning of the graph ensures all the vertices that need processing reside in the on-chip memory during the processing of a partition. Thus, these designs remove random accesses to the off-chip memory and reduce the access time to vertices. However, temporal partitions come with additional costs in terms of preprocessing, portability, and overhead of switching between slices, and underutilization of hardware resources. Next, we will describe these costs in more detail.

2.3.1. Preprocessing Cost and Portability. Ideally, the graph should be partitioned into slices that are individually highly connected and mostly independent. However, optimal algorithms for such preprocessing can exceed the computation complexity of basic graph processing algorithms such as BFS. For example, polynomial time solutions for min-cut have not been found yet, while BFS has a complexity of $O(\|V\|+\|E\|)$ [13,15,46]. In recent studies, Balaji et al. show that RABBIT++ requires 1047 iterations of SpMV (Sparse Matrix times Dense Vector) kernels to amortize for preprocessing costs [15]. Therefore, works such as PolyGraph rely on simpler partitioning heuristics such as chunking the vertex set based on the *id* of the vertices [29,109]. Chunking has a complexity of $O(V)$, but it comes at the cost of reducing the independence of the partitions that are generated. Moreover, for every vertex that has an edge that crosses partitions, that vertex has to be replicated

in the destination partition. The replicated vertices are used to communicate information from one partition to another. Furthermore, temporal partitioning of graphs is not a portable approach because the amount of available on-chip memory determines the size of each slice. This means a graph partitioned for an accelerator with 8 MiB of on-chip memory has to be repartitioned for an accelerator with a different amount of on-chip memory.

2.3.2. Switching Cost and Resource Utilization. In addition to the preprocessing cost, there is also the cost of increased processing time due to switching and underutilization of hardware resources. There are three processing steps when switching between temporal slices: (1) The current partition's vertices must be written back to the off-chip memory and the new partition's vertices must be read from off-chip memory. (2) Each slice has a set of replicated vertices to facilitate temporal partitioning, and any vertex updated by the current temporal slice which resides in other slices must be read and updated in off-chip memory. (3) All replicated vertices of the new partition must be read to create the inter-slice messages. Finally, each slice must be processed multiple times because of the inter-slice edges, which adds to work inefficiency.

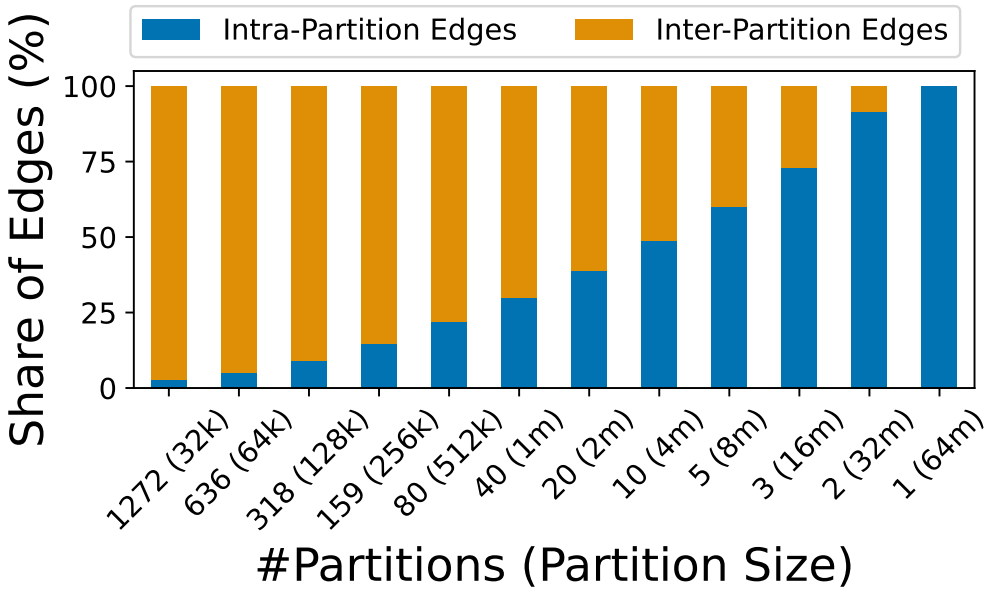


FIGURE 2.1. Distribution of edges within and between partitions as the size of a partition grows for Twitter graph.

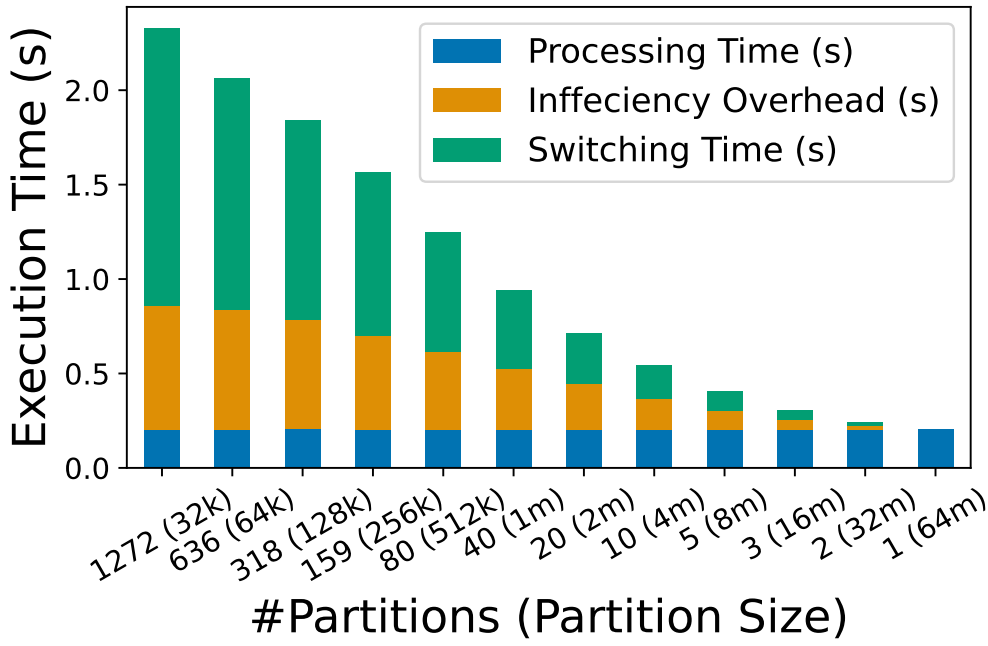


FIGURE 2.2. Overhead of temporal partitioning as the number of partitions increases using BFS and Twitter graph.

To quantify the overhead of partition switching, we implemented the partitioning technique used in PolyGraph [29, 109]. For every switch operation, we chose the partition that has the most updated vertices to process as the next partition. We used the Twitter graph as an input to the BFS workload and broke the execution time down into two components. (1) *Processing time* accounts for time spent processing partitions. (2) *Switching time* accounts for time spent switching partitions including writing the current partition, reading the new partition, and reading/writing inter-partition replicated vertices. (3) *Inefficiency overhead* accounts for the time spent processing partitions more than once (i.e., the extra time from the decreased work efficiency[1]).

Figure 2.1 shows the distribution of edges within and between partitions for various graph sizes. This figure shows that as the number of partitions grows, the number of intr-partition edges outgrows the number of edges inside of partitions. With ten partitions, almost 50% of edges are inter-partition edges. This increase in the number of inter-partition edges results in higher switching time since more replicated vertices need to be updated.

[1]Work required by the optimized sequential execution (in terms of edges processed) over the work performed in parallel execution

Figure 2.2 shows the breakdown of execution time between processing time, inefficiency overhead, and switching time as the number of partitions grows (larger graphs). Inefficiency overhead and switching time constitute approximately 20% of the execution time when there are less than three partitions. As the number of partitions grows, the inefficiency overhead increases quickly. For instance, in the case of 318 partitions, inefficiency overhead makes up more than 75% of the execution time. Figure 2.2 shows that temporal partitioning of larger graphs with more partitions can reduce the performance, making the accelerators that use this method unscalable.

When using temporal slicing, there is no way to eliminate inter-tile events, even with an optimal tiling strategy (e.g., min-cut). Furthermore, as the number of tiles increases either due to using larger graphs or smaller on-chip memory, the overhead of inter-tile events will increase, and the work efficiency will decrease.

CHAPTER 3

Scalable Engine for Graph Acceleration

Modern data-driven scientific discovery is based on understanding relationships between data entities that are modeled as graphs—i.e., graph processing. Traditionally, graphs are processed on CPUs and GPUs using optimized libraries such as Galois [73], Ligra [81], GAPBS [17], Gunrock [93], etc., but the performance of software implementations on these large graphs is unacceptably poor in many applications. As a result, there has been strong interest in new hardware accelerators for graph processing. Starting with early work in FPGA-based graph exploration [54,107,108], numerous architectures for graph processing hardware have been reported in the literature over the past few years, including accelerators which leverage processing in/near memory and run at the wafer-scale. At the same time, the size of these graphs is growing quickly as it is becoming increasingly easier to collect large amounts of data. For instance, WDC12 [3] has 3.6 billion vertices and 128.7 billion edges, requiring over a terabyte of working memory.

There is currently a mismatch between graph accelerator designs and the growing graph data scale, which this paper aims to address. Most graph accelerators leverage vertex-centric programmers where processing vertices exhibit low spatial and temporal locality, leading to frequent random memory accesses.

To address this, many prior studies [29, 40, 75] have sought to exploit locality and reduce data movement overheads. These prior works use large on-chip memories, in/near memory processing [11, 83], and the addition of hardware to improve locality for graph processing [10]. Some prior work [29, 40, 67, 75] optimizes performance by storing vertices in the on-chip memory while storing the graph structure (i.e., edges) higher capacity off-chip memory like HBM or DDR4. In these accelerator designs, the data is explicitly moved from off-chip memory to on-chip memory, and their efficiency hinges on finding locality within the vertex updates. However, as the size of the graphs processed increases, the vertex set will exceed the on-chip memory capacity. To address this problem, prior work relies on partitioning the graph into slices that fit in the available on-chip memory and temporally sharing the on-chip resources. This temporal portioning is akin to memory

swapping and time-shared multiplexing of the processor in operating systems. Like these related techniques, temporal partitioning in graph accelerators imposes significance when switching between partitions. Furthermore, the generated partitions often exhibit inter-dependencies, requiring multiple processing of each generated partition. Our analysis (described in Chapter 2) shows that as the number of partitions grows, the overheads constitute an increased share of the execution time, up to 90% of the execution time. The overheads from temporal partitioning, which is required for larger graphs, cause current graph accelerator designs to exhibit poor throughput scaling as the graph size grows.

Figure 3.1 shows how PolyGraph's performance varies as the graph size grows, given a fixed-size on-chip memory. As the graph size increases, the throughput drops, indicating that the locality improvement depends heavily on the on-chip resources. As discussed in Section 2, temporal partitioning is the main limitation preventing PolyGraph and similar systems from scaling to large graphs efficiently.

This work presents a new graph processing architecture that enables a consistent graph processing throughput independent of graph input size. Instead of focusing on capturing locality, SEGA uses on-chip resources to enable enough parallelism—abundantly available in large graphs—to hide the latency of accessing the off-chip memory. The intuition behind our design is that for each reduction operation (i.e., message to update a vertex property), multiple propagations take place (i.e., many neighbors need to be updated). As a result, the throughput of the propagation procedure is more resilient to the latency of accesses to the memory. In our design, we introduce a runtime procedure to efficiently orchestrate vertex information between reduction and propagation. Our design enables leveraging the capacity of both on-chip and off-chip memory to store the active working set. In turn, we increase our coalescing space to the whole vertex set, resulting in increased work efficiency. Furthermore, our design is scalable because we leverage the idea of [message-driven processing] to push updates to the local processing element responsible for each vertex. CF Graph is a microarchitectural and system-level design for scalable graph processing based on the idea of active messages. Instead of storing active vertices on the on-chip memory for faster access and better coalescing, we move the entire active vertices to the off-chip memory. Moving data to the off-chip memory removes the use of temporal partitioning in our design. Additionally, by enabling

all vertices to be active simultaneously, we can coalesce more updates and increase work efficiency compared to prior work with temporal partitioning.

This chapter focuses on the following:

- An analytical model that helps designers understand the system-level requirements of large graph analysis. This proposed model can also predict performance based on the network, memory, reuse distance, and tiling schemes.
- We identify the scalability limitations of prior work because of their reliance on temporal partitioning to achieve high performance.
- We introduce a new graph processing microarchitecture: SEGA, and we describe multiple architectural tradeoffs in its design.
- We evaluate our new microarchitecture, at multiple scales, and show 1.3× to 1.84× performance improvement compared to PolyGraph. We observe that for smaller graphs, PolyGraph outperforms SEGA. However, as the size of the graph increases, PolyGraph's performance drops.

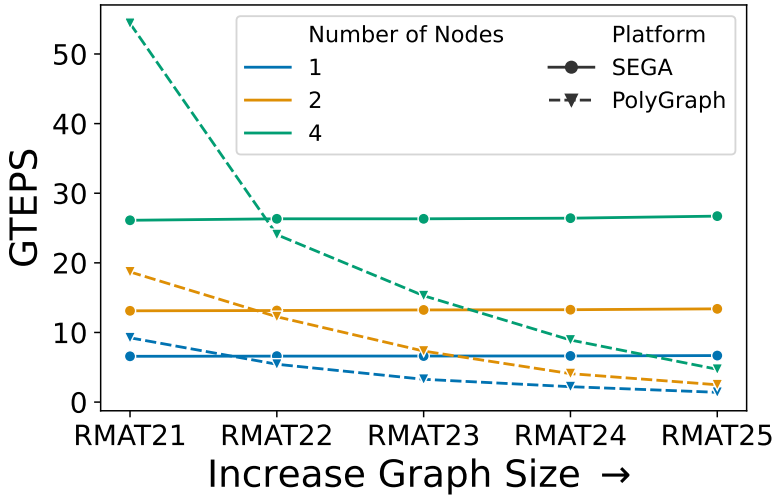


FIGURE 3.1. Both systems have 1.5 MiB on-chip memory and 332.8 GB/s for BFS workload. GTEPS (giga traversed edge per second) represents the system throughput. Higher GTEPS shows higher throughput.

3.1. Background on Message-driven Graph Applications

Vertex-centric programming is a common paradigm for implementing graph algorithms [10,11,17,29,40,73,75,76,81,85]. In this computation paradigm, programmers describe graph algorithms

as a series of operations from the perspective of a vertex [61]. Some graph processing workloads can be implemented using the *message-driven* paradigm, which can be thought of as each vertex sending *messages* to its neighbors. Each *message*, such as $< u, \delta >$, has two main attributes: a destination vertex (u) and an update (δ). In this *message-driven* model, an active vertex (such as u) calculates an update for each of its neighbors and sends the update through the message.

In the message-driven model, every workload is described using two main functions: **Reduce**, which determines the new property for a vertex using its current property and a message for that vertex, and **Propagate**, which determines the update of a message using the property of the vertex and the weight of an edge. Algorithm 1 shows an example of a message-driven vertex-centric algorithm used in SEGA.

3.2. A Model for Scalable Accelerator

The proposed performance model calculates the system-level requirements assuming an asynchronous implementation of the widely used vertex-centric programming paradigm for graph processing. We assume a system is constructed by combining a set of accelerator tiles (the basic building blocks) with an appropriate interconnection network. The requirements are in terms of *memory capacity* and *memory bandwidth* for different types of memory, the *network bandwidth*, and *number of tiles*. These requirements depend on the structure of the graph, representation of different data structures, and physical constraints such as the maximum I/O on each tile. With the assumption that graph algorithms are throughput limited, these requirements capture the behavior of many graph accelerators. Additional constraints could be added to the model if this assumption does not hold (e.g., the compute capability is a limiting factor).

To calculate the bandwidth for each component, we consider the maximum performance required from each component individually. Equation 3.1 shows Graph Algorithm Iron Law (GAIL) [18] which calculates the execution time for graph workloads. GAIL separates algorithm and hardware performance. Traversed Edge Per Second (TEPS) shows the hardware performance. For the same algorithm implementation, larger TEPS will result in a smaller execution time. Our model focuses on maximizing this TEPS performance metric.

$$\frac{\text{time}}{\text{kernel}} = \frac{\text{number of edges}}{\text{kernel}} \times \frac{1}{\text{TEPS}} \tag{3.1}$$

3.2.1. Memory System. Next we describe what data structures to store in the memory and the capacity and bandwidth requirement of these data structures. Both edges and vertices have different characteristics. The proposed model calculates the capacity and bandwidth requirements of edges and vertices separately. For simplicity we only consider vertex and edge data structures in the description below, but the model can extend with other data structures easily.

3.2.1.1. *Edge.* In general, edges require a larger memory capacity compared to vertices (graphs such as WDC12, Twitter, and LiveJournal have 36×, 34×, and 15× more edges than vertices). In most graph programming models [41, 56, 73] access to edges are sequential and read-only. The model should determine the number of memory devices that provides enough capacity for a targeted graph. Table 3.1 Row 1 shows the required capacity for the edge memory which depends on the number of bits representing edge information. The size of the edge is different between accelerators. At a minimum it should include the vertex ID (destination or source) and the weight of the edge. The other constraint for the edge memory is the required bandwidth to achieve a certain performance in TEPS (shown in Table 3.1 Row 2).

3.2.1.2. *Vertex.* In contrast to edges, vertices have low spatial and temporal locality in most graph algorithms. This lack of locality causes inefficient off-chip memory access and low memory access throughput. Prior work exploits locality through graph pre-processing [40] or by online traversal scheduling [?] to improve the on-chip cache usage. Other hardware accelerators use on-chip SRAMs to store vertices and/or events to reduce off-chip memory access [75].

The required vertex bandwidth depends on the performance of the edge memory. For every edge read there is at most one vertex read and one vertex update. Therefore, vertex access rate is 2× higher than edge access rate. The maximum supported bandwidth also depends on the access granularity to the vertex memory system (a.k.a. *atom size*). On-chip caches reduce the off-chip vertex memory access rate. Therefore in our model we use a parameter α to model accelerators with on-chip memory assigned to vertices. α indicates the fraction of the off-chip memory bandwidth that is needed by the accelerator. α is a value between zero and one. An accelerator that exploits locality/reuse will have a smaller α which means it will have lower off-chip memory bandwidth requirement. Table 3.1 Row 3, shows the maximum required bandwidth for vertex memory given a TEPS (usually from the peak edge bandwidth (Table 3.1 Row 2). In addition to the bandwidth,

we must also meet the capacity requirement of the vertex memory. Our model needs to consider the capacity of vertices (Table 3.1 Row 4).

3.2.2. Network Requirements. After finding the best memory technology for vertex and edge data structures, we must ensure that data movement among accelerators will not cause a performance bottleneck. We consider the *network* as the third constraint in our model. What we consider as network in our scaled-out system is the interconnection fabric between accelerators. Accelerators use this network to communicate inter-slice updates. A well-partitioned graph with a low inter-slice event rate does not require a high bandwidth network. The proposed model takes into account the bisection bandwidth, port bandwidth, and topology of interconnection network.

3.2.2.1. *Bisection Bandwidth.* The required bisection bandwidth is dependent on the performance of each accelerator and the size of the message communicated through the network. With every edge read a new message is created to another vertex. Hence, TEPS indicate the maximum rate of messages generated from the aggregate edge memory across all accelerators. We use γ to indicate the fraction of the messages targeting vertices in remote accelerators and have to be communicated over the network interconnect. Table 3.1 Row 5 shows the required network bisection bandwidth.

Data representation dictates the size of the message communicated in the system. It depends on the implementation of the accelerator and is a parameter in our model.

3.2.2.2. *Port Bandwidth.* The required network bandwidth for each accelerator can also be a limiting factor. Port bandwidth depends on the required bisection bandwidth and the number of accelerators in the system. See Table 3.1 Row 6.

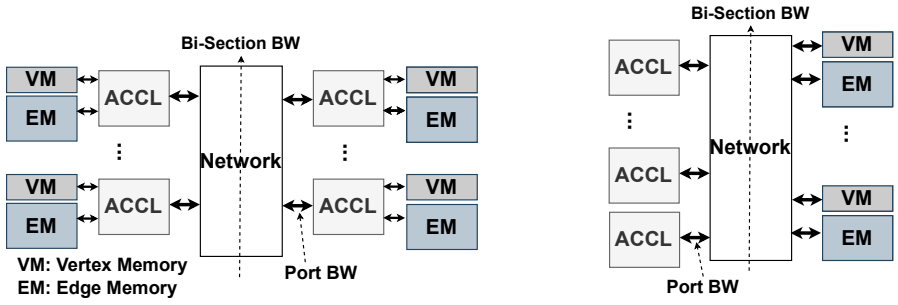


FIGURE 3.2. System architecture for the scaled-out accelerator. a) Near memory processing. b) disaggregated memory-based.

3.2.2.3. *System Architecture.* The system architecture dictates the interconnection traffic patterns and depends on the port bandwidth and the interconnection network topology. We model two systems: near memory processing architectures and disaggregated memory-based architectures as shown in Figure 3.2. In the near-memory processing system, each accelerator tile is connected to a local memory system, and the interconnection network is between the accelerators. Here, each accelerator is operating on its local vertex and edge information, and only events are communicated through the interconnect (Table 3.1 Row 5). Whereas in the disaggregated memory-based system the accelerators do not communicate directly with each other—the communication is through the memory. In this topology, accelerators use interconnects for communicating events and for reading/writing vertex and edge information. Hence, it required a larger bisection bandwidth in the network (Table 3.1 Row 7).

1	Edge Capacity	= Number of edges × Sizeof(Edge)
2	Edge Bandwidth	= Maximum TEPS × Sizeof(Edge)
3	Vertex Bandwidth	= 2× atom size × TEPS ×α
4	Vertex Capacity	= Number of vertices × Sizeof(Vertex)
5	Bisection BW (Near-memory)	= TEPS × Sizeof(message) ×γ
6	Port Bandwidth	= $\frac{\text{Bisection Bandwidth}}{\text{Number of accelerators}}$
7	Bisection BW (disagg.)	= Vertex BW + Edge BW

TABLE 3.1. System constraints used in the proposed model. α is the miss ratio of vertex on-chip memory. γ is the percentage of inter-accelerator to intra-accelerator communication.

3.2.3. Accelerator Node. The internal architectural parameters of accelerators are another constraint in our model. These parameters are: 1) *Number of bits* for representing data structures. This parameter impacts the network's bisection and port bandwidth and the edge memory bandwidth requirement. 2) *On-chip resources* such as caches and SRAMs. These on-chip memory systems will impact the required bandwidth from the main memory (α). 3) *The number of I/O pins* connected to each accelerator and their *data rate* limits the number of memory nodes connected to the accelerators (total capacity supported by a single accelerator) and the port bandwidth of the

interconnect. Given the high memory access and communication-to-computation ratios of graph workloads, we do not consider the computation within the accelerator as a bottleneck in our model. Ham et al. show that only 6% of instructions executed in a graph workload are responsible for custom graph computations, and the rest of the instructions are used for loading and traversing the graph [40].

3.3. SEGA Architecture

In the previous section, we demonstrated that depending on temporal partitioning for enhancing locality is a limiting factor for processing large graphs. Therefore, we design SEGA, a scalable graph accelerator that can achieve high performance without exploiting locality and temporal partitioning. In this section, we will explore the microarchitectural and system-level considerations that enable the scalability of graph processing to handle large-scale graphs. Our main objectives are twofold: first, to decouple performance from the on-chip memory size and eliminate the need for temporal partitioning for large graphs, and second, we will investigate a system-level approach to creating a balanced graph accelerator that maximizes performance while minimizing cost (i.e., with the least over-provisioning of bandwidth and capacity) as we scale to large graphs.

Prior work achieves high (though not full) utilization of their edge memory bandwidth by placing vertices in on-chip memory. However, since on-chip memory is limited, it must be temporally shared between different partitions of the graph. We eliminate the need for temporal partitioning by storing and processing vertices in *off-chip* storage, mitigating the performance degradation caused by temporal partitioning. Furthermore, we architect a *balanced* design such that messages are processed at the rate they are generated, and active vertices are available to generate messages at the rate they are processed. Figure 3.3 shows a logical diagram of SEGA where we decouple the reduction operations from the message propagation. The rest of this section describes how we decouple vertex processing from edge-based message propagation so they can run at different rates through the message processing unit (Section 3.3.2), the message generation unit (Section 3.3.3), and the vertex management unit (Section 3.3.4).

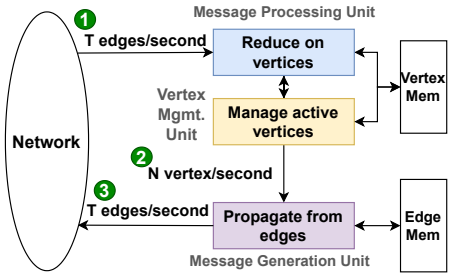


FIGURE 3.3. Logical demonstration of a balanced system.

Algorithm 1 Decoupled message-driven implementation for single source shortest path.

```
dist[:||V||] = ∞;
messages.append({u,0});
while not messages.empty() do
    for u, δ in messages do
        old_dist = dist[u];
        dist[u] = min(old_dist, δ);
        if old_dist != dist[u] then
            active_list.append(u);
while not active_list.empty() do
    for v in active_list do
        start, end = row_ptr[v], row_ptr[v+1];
        active_vertex_info.append({dist[v],start,end});
while not active_vertex_info.empty() do
    for α, start, end in active_vertex_info do
        for i in range(start, end) do
            dest = edge_dests[i];
            ϵ = add(α, edge_wgt[i]);
            messages.append({dest,ϵ});
```

3.3.1. System-level design of SEGA. SEGA consists of one or many graph processing nodes (GPNs). The building block of a GPN is a processing elements (PE). A single PE is a message-driven processor capable of executing algorithms expressed using the model described in Algorithm 1. Algorithm 1 shows the decoupled message-driven implementation of the SSSP workload. For SSSP, the reduce function is the **minimum** function, and the propagate function is the **addition** function. Decoupling the reduction from message propagation enables SEGA to support both asynchronous [104] and bulk synchronous parallel [90] execution models. In the bulk synchronous parallel execution, the red block (message propagation) is executed serially after there

are no more messages to process in the blue block (message processing). This serial execution is enforced by the decoupled active list (shown in yellow). When executing in bulk-synchronous mode, these three blocks continue to execute until no messages are generated, i.e. the program has converged. In asynchronous [**104**] execution, all blocks are executed simultaneously until there are no more messages.

A PE consists of three main units which correspond to the three parts of Algorithm 1. The colors of the units in Figure 3.3 and Figure 3.4 correspond the colors in Algorithm 1 as well.

- *Message processing unit* processes messages and updates vertices. It determines the new property of a vertex using the reduce function.
- *Vertex management unit* keeps track of active vertices from the message processing unit and sends active vertices to the message generation unit.
- *Message generation unit* uses active vertices and their edges to generate new messages. It produces the update in a message using the propagate function.

For every vertex processed by the message processing unit, there can be many messages generated by the message generation unit since each vertex can have many outgoing edges. This one-to-many relationship where one message creates many messages can cause the buffers on the message processing units to become full which introduces backpressure and queuing at the message generation unit. However, to empty the incoming message queue we must have available buffers in the message generation unit. Thus, we need a way to deal with the back pressure in the system without causing deadlock when the system is overwhelmed with messages. To break this deadlock, we *decouple* message processing from message generation with the active vertex management unit. The combination of active vertex management unit and the decoupling allows us to support bulk synchronous execution.

The message processing unit *pushes* new active vertices into the `active_list` in the vertex management unit, and when the message generation unit is ready to accept more work, it *pulls* from the vertex management unit. The vertex management unit allows the active vertices to *spill* into the vertex memory if the work generation and work consumption are not balanced. By decoupling the push and the pull from the message processing and generation units, the vertex management unit allows SEGA to balance the work generation and fully utilize the off-chip bandwidth resources.

3.3.2. Message Processing Unit. The primary function of the message processing unit is to process vertices based on incoming network messages. To process a message ($< u, \delta >$), the destination vertex (u) has to be read. After the vertex has been read, its property (e.g., distance in case of SSSP) and the message's update (δ) are used by the reduce function (e.g., minimum in case of SSSP) to determine the new property of the vertex. In addition, the read, modify, and write steps on each vertex should be done atomically to ensure correctness.

Due to a lack of locality in accesses, reading vertices from DRAM can result in long access latency. The message processing unit utilizes a hardware managed buffer to track multiple atomic operations and hide the access latency. Nevertheless, due to the massive size of some graphs, it is unlikely that the buffer can capture much locality. In our implementation, we have configured this buffer with a size of 64 KiB for each PE (512 KiB per GPN). We have evaluated the effect of the size of this buffer in Section 3.5.

When the read data arrives from the memory, the whole memory block is placed in this buffer and the destination vertex (u) is sent to the reduction engine. After the reduction, the memory block data in the buffer is updated with the new copy of the vertex.

SEGA manages this buffer as a direct-mapped cache, and the following steps are taken upon insertion and eviction.

Insertion: blocks of memory are placed in this buffer every time a vertex has to be read. The location of a block of memory in this buffer is determined using a direct-mapped function based on vertex's address. Therefore, it is possible that two read requests collide with each other, in which case the two requests have to be serialized.

Eviction: blocks of memory are evicted from the buffer in two circumstances. (1) The whole buffer is flushed back to the memory upon the completion of execution. (2) Blocks of memory are written back to memory upon a read conflict. Before a block is evicted, if it contains active vertices, the message processing unit notifies the vertex management unit of this eviction. The vertex management unit will track this memory block as an active block for use by the message generation unit.

The choice of a direct-mapped function for the placement is an appropriate decision because conflicts are not frequent enough for an associative design to exhibit significant benefits. We found

that ∼4% of the accesses result in conflict. Therefore, the choice of direct-mapped placement results in lower latency and simpler design.

3.3.3. Message Generation Unit. The main purpose of the message generation unit is to propagate messages to the destination vertex. The messages are generated using the edges[1] of active vertices from the vertex management unit (an active vertex is an updated vertex that needs to communicate its value to its neighbors). The unit initiates its process by reading an entry from the active vertex buffer (as described in the next subsection). Each entry in the active vertex buffer has three members $< \alpha, start, end >$. α denotes the property of the active vertex, and $< start, end >$ identifies the location of its edges in the edge memory.

For each edge, $< v, w >$, in $< start, end >$, a message is generated. The destination of the message is determined by the destination vertex of the edge (v), and the message update is calculated using the propagate function on the property of the vertex (α) and the weight of the edge (w). Subsequently, the message is sent to the network, where it is received by its designated message processing unit. The addressing function is assigned at initialization time since vertices are statically assigned to PEs.

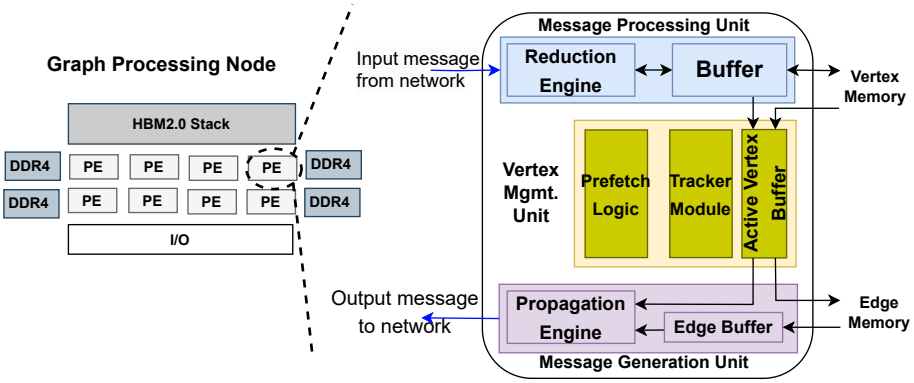


FIGURE 3.4. Hardware implementation of processing elements.

3.3.4. Vertex Management Unit. The main role of the vertex management unit is to: 1) keep track of active vertices assigned to a PE and 2) send active vertices to the message generation unit.

[1]We store the destination vertex and edge weight in the edge memory and the vertex property values and other CSR metadata such as index array pointers in the vertex memory.

The vertex management unit has three main components. (1) An on-chip memory to keep track of active vertices in the vertex memory (*tracker module* in Figure 3.4). (2) A first-in-first out buffer to send active vertices to the message generation unit (*active vertex buffer* in Figure 3.4). (3) Control logic for searching and recovering active vertices from the vertex memory (*prefetch logic* in Figure 3.4). In the following section, we discuss the details of the implementation of each component and how the vertex management unit accomplishes its tasks.

During the execution of our graph accelerator, new active vertices can be generated faster than messages created from prior active vertices because each active vertex generates multiple messages (i.e., one vertex can connect to multiple edges). Therefore, the active vertices will quickly outgrow the on-chip memory size in the message processing unit. Due to the one-to-many relationship between vertices and edges and the presence of fixed-size queues in the system, storing all active vertices on limited on-chip storage causes a deadlock. In this scenario, active vertices are spilled to the backing vertex memory to prevent stalls or potential deadlocks. The active vertices spilled from the buffer can overwrite their previous values in the vertex memory. However, to accurately execute the application, these spilled active vertices will have to be accessed and sent to the message generation unit.

When the message generation unit has the bandwidth for creating and sending new messages, spilled active vertices are accessed and sent to the message generation unit. To access the spilled active vertices, we can perform an associative search in the vertex memory or track the spilled active vertices. Performing associative searches in DRAM is significantly more expensive compared to SRAM. Hence, we developed a tracking mechanism to allow the vertex management unit to track the location of these off-chip active vertices.

Naively, keeping a single bit for every vertex in the graph would enable us to track the active vertices, but it would result in significant on-chip capacity. For example, in WDC12, a structure that keeps 1 bit per vertex would require approximately 440 MiB of on-chip storage. In addition, the size of the vertex data structure can vary greatly depending on the program or programmer, making it difficult to predict the number of vertices that need to be tracked with a constant capacity for tracking information.

We take three approaches to track active vertices efficiently: (1) We track locations in memory with active vertices, not the vertices themselves, (2) we group these locations in memory (memory

blocks) into superblocks to reduce the tracking metadata, and (3) prefetch active vertices before they are requested from the message generation unit.

While graph workloads operate at the granularity of vertices, any transaction with the vertex memory happens at a fixed size referred to as the block size. We will refer to a block of the memory that stores **at least** one active vertex as an *active block*. Tracking active blocks instead of active vertices has two main benefits. (1) Unlike vertices, which are program-specific data structures, a memory block has a fixed size; separating the size of the vertex management unit from the size of the vertex data structure. (2) It is safe to assume that a block of the vertex memory is bigger than the size of a vertex. Tracking active blocks instead of active vertices allows us to reduce the size of the tracking information.

In SEGA, we track active vertices based on a *superblock* of N blocks and count the number of active blocks in the superblock. Therefore, the capacity to implement the tracker module could be significantly reduced by grouping more blocks into a superblock. The total capacity required by such an implementation could be calculated using Equation 3.2 and Equation 3.3 where $super\_block\_dim$ denotes the number of memory blocks grouped into a superblock, and $block\_size$ denotes the block size for the vertex memory. In our implementation, we use $super\_block\_dim = 128$ and HBM2 ($block\_size = 32\ B$) as our vertex memory. Given these parameters, only 1 MiB of on-chip storage is required to track a 4 Hi stack of HBM2 with a capacity of 4 GiB (128 KiB per PE). Storing WDC12 vertices with 4 B for each vertex requires 4 HBM2 stacks and only 4 MiB of on-chip storage across four GPNs.

It should be noted that grouping memory blocks is a middle ground between tracking one bit per vertex and no tracking at all. Therefore, instead of **eliminating** the need for performing an associative search in the vertex memory, this implementation **reduces** the amount of associative search. We analyze the sensitivity of performance to the size of the tracker module in Section 3.5.4.2.

$$cap_{bits} = (\log_2 super\_block\_dim + 1) * num\_super\_blocks \tag{3.2}$$

$$num\_super\_blocks = \frac{vertex\_memory\_capacity}{super\_block\_dim * block\_size} \tag{3.3}$$

Finally, the message generation unit depends on the vertex management unit for its execution. Therefore, the vertex management unit needs to fetch and deliver activate vertices to the message

generation unit in a timely manner. To hide the latency of searching for active vertices in the vertex memory, the vertex management unit takes advantage of a buffer (active vertex buffer) to prefetch active vertices.

The active vertex buffer is an 80-entry buffer that can be populated with active blocks of vertices. The prefetch logic is configured to read 16 blocks of memory from a superblock whenever there are 16 or more entries available in the buffer. When the data arrives from the memory, only the active blocks are placed in the buffer. The remaining blocks are dropped. In turn, this associative search in DRAM results in some wasted bandwidth. We have investigated the effect of vertex recovery on the utilization of vertex memory bandwidth in Section 3.5.4.2.

3.3.5. Choice of off-chip Memory. In graph workloads, vertices and edges have different requirements for capacity and bandwidth. In general, edges require a larger memory capacity compared to vertices. In most graph programming models [**56**, **73**], access to edges is sequential and read-only. In contrast to edges, vertices have low spatial and temporal locality in most graph algorithms. This lack of locality results in a long memory access latency. Due to these differences, we have chosen a heterogeneous system for the off-chip memory in a GPN.

We have chosen HBM2 as the off-chip memory to store vertex information. HBM2 is an appropriate choice for storing vertex because it offers substantially more bandwidth under **random** access patterns [**94**]. Moreover, HBM allows for finer granularity accesses (32 bytes), resulting in less bandwidth waste.

To store the edges, we have chosen DDR4 as the off-chip memory. This is because the edge information is accessed with a **mostly sequential** and **read-only** pattern. Lastly, for most graphs, edges require a much bigger storage than vertices. Therefore, DDR4 is an appropriate choice for storing the edges because of its high capacity density compared to HBM2 and its high bandwidth under sequential access. Fariborz et al. showed heterogeneous memory system is the most cost-effective solution for creating a system that meets the capacity and bandwidth requirements of the two data structures [**34**].

We have assigned eight PEs to one stack of HBM2 memory (eight channels). Each PE is dedicated to one channel of the HBM2 stack and operates only on the vertices stored in that channel. Fariborz et al. show that vertex memory needs to offer 4× the bandwidth of the edge

memory [34]. Therefore, we have allocated four DDR4 channels for the edge memory for each GPN ($\frac{1}{2}$ of a channel per PE) to create this balance.

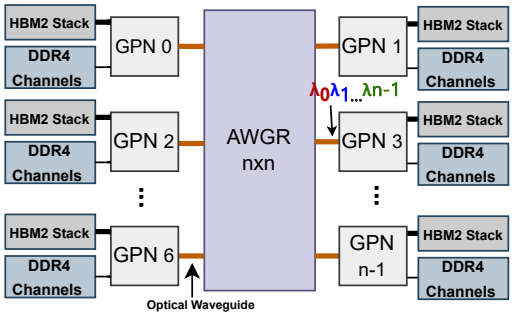


FIGURE 3.5. System-level architecture of SEGA with n GPNs. To scale to the Size of WDC12 (3.6 Billion vertices and 128.7 Billion edges) SEGA requires eight GPNs (n=8) using 8×8 AWGR with eight wavelengths per port is used. [79]

3.3.6. Graph Processing System Using GPNs. Figure 3.4 shows the layout of one SEGA node or GPN. A GPN consists of multiple PEs connected through an interconnected network and off-chip memory as storage for storing the vertices and edges. We connect PEs within one GPN through a point-to-point on-chip electrical network. Scaling to larger graphs is possible by connecting multiple GPNs together. The choice of the inter-GPN interconnect is an important design decision as it affects both the bandwidth and energy/bit. Electrical interconnects can support the bisection bandwidth for small number of GPNS. However, when scaling to more than 8 GPNs (to support WDC12) we propose using optical interconnectss.

Optical links provide low latency, high bandwidth density through wavelength division multiplexing (WDM) [65], and distance-independent energy consumption and latency. By exploiting WDM one GPN can connect to multiple GPNs through a single optical IO pin (addressing them on different wavelengths), enabling high-radix and low-diameter networks. Array Waveguide Grating Router (AWGR) [79] is a passive silicon photonic fabric with a compact layout that offers scalable all-to-all connectivity through wavelength routing. Recent advances in the fabrication process of AWGRs now enable their integration with a significantly reduced footprint (1 mm^2), crosstalk (< -38dB), and loss (< 2dB) [79]. Initial studies have shown AWGR to be a promising choice for processor-to-memory network [36, 37]. The number of ports in an AWGR can easily scale up to 64 ports [26]. We propose to use an Array Waveguide Grating Router (AWGR) based optical

network [37, 79] as the inter-GPN interconnection network as shown in Figure 3.5 that can easily scale to WDC12.

3.3.7. Spatial Vertex Mapping. In our proposed architecture, each vertex and its edges are assigned to a single PE. Choosing the vertex assignment is a tradeoff between preprocessing cost, load balancing, and locality. In a *load balanced* system, a similar number of edges are assigned to each PE for processing. The approach to optimizing load balance is to sort the vertices by their out-degree and distribute the vertices with the highest out degrees uniformly across PEs. Interleaving vertices between PEs with a fine granularity ensures load balance. In the locality-based approach, we used community detection techniques such as RABBIT [13] to detect highly connected vertices and assign sequential ids to vertices in each community. By taking advantage of locality, we can reduce network traffic at the cost of a lower load balance. We can also use the original ordering made by the graph publisher and eliminate any pre-processing. In this case, we interleave the vertices based on their vertex IDs between PEs, assigning a similar number of vertices to each PE.

3.3.8. Summary of Design. As described in this section, we create a balanced architecture by (1) decoupling vertex and edge processing to process edges and vertices at different rates; (2) spilling active vertices to HBM and tracking the active vertices to avoid expensive off-chip memory search; (3) using a heterogeneous memory system that optimizes the different memory access patterns in the graph workloads with higher vertex bandwidth to edge bandwidth; (4) proposed using a passive optical all-to-all interconnect to scale out the accelerator to multiple nodes enabling support for larger graphs.

To extract maximum throughput, active vertices need to be sent to the message generation unit at a high rate (path 1 in Figure 3.3). On the other hand, messages generated by the message generation unit should also be processed at the same rate (path 2 in Figure 3.3). Both these criteria require vertices to be processed with low latency.

In a graph workload, for every modified vertex, multiple messages will be created depending on the degree of the active vertex. Therefore, the message processing unit needs to process the incoming messages at a faster rate than the message generation unit creates them (path 3 in Figure 3.3). Separating vertex and edge processing allows the accelerator to process vertices at a higher rate. By decoupling, the message generation unit requests new work when it has enough bandwidth at

Specifications per GPN	SEGA
# PE	8 @ 2GHz
Spad	512 KB (buffer) + 1 MiB (VMU)
Vertex memory	HBM2 stack - 8GB cap. - 256GB/s
Edge memory	4 DDR4 channels - 128GB cap. - 76.8GB/s
Inter-GPN Net.	8 × 8 SiN AWGR, 8 Wavelengths per port, 4GiB/s per wavelength

TABLE 3.2. System specifications.

the edge memory. Since the vertex processing is faster, SEGA ensures that the message generation engine can pull active vertices upon request.

Now the size of on-chip memory is no longer dictated by the size of the graph. In our design, the on-chip hardware buffer simply hides the long off-chip memory access latency instead of storing all active vertices. We sized the on-chip buffer using Little's law based on the message generation rate of the edge memory and the average vertex latency when running with random traffic.

Finally, the active vertex management acts as an intermediary between message generation and processing, allowing the processing unit to store active vertices in off-chip memory without performing costly associative searches by tracking the locations of spilled vertices. Simultaneously, this unit enhances the message generation engine's ability to read active vertices at a faster rate by prefetching them from the buffer or DRAM.

3.4. Methodology

We implemented our model in gem5 v22.0 [57]. We implement cycle-level models for all the modules in a GPN (Figure 3.4) proposed message generation unit, message processing unit, vertex management unit, and a model for the point-to-point interconnect network and AWGR at 2GHz in gem5. We used gem5's models for HBM2 and DDR4 memories. The implementations of SEGA is shown in Table 3.2.

We also implemented a model of PolyGraph [29] in gem5 including the partitioning of a large graph into slices. We compared our performance to PolyGraph since it already showed better performance compared to notable graph accelerators such as GraphPulse, Chronos, Ozdal, and Grapicionado [10, 40, 68, 75], and software platforms such as Ligra, and Galois. [73, 81].

3.4.1. Workloads. We implemented five graph analytics workloads specified in GAPBS [17]. We used **b**readth **f**irst **s**earch (BFS), **c**onnected **c**omponents (CC), **s**ingle **s**ource **s**hortest **p**ath

Graph	Footprint	Vertices	Edges	Description
RoadUSA [1]	805.7 MiB	23.9M	58.3M	Road graph
Twitter [49]	14.4 GiB	41.65M	1.46B	Social network
Friendster [49]	15.4 GiB	65.6M	1.8B	Social network
Host [8]	16.6 GiB	101M	2B	Hyperlink graph
Urand [33]	34.0 GiB	134.2M	4.2B	Synthetic graph

TABLE 3.3. Graph Workloads used in evaluations.

(SSSP), **p**age **r**ank (PR) and **b**etweenness **c**enterality (BC). We have implemented BFS, CC, and SSSP in the asynchronous mode, while PR and BC are implemented in the bulk-synchronous mode. BC in its proposed asynchronous implementations requires forward and backward passes, which doubles the number of edges required to be stored. Our implementation of PR-delta as specified by [75] proved to be very sensitive to the order of the traversal of the graph. To find the optimal order of traversal requires an overall view of the graph at the timing of scheduling updates. Therefore, ordering is not a feasible solution for problem sizes that are significantly bigger than the size of on chip resources. Hence, we have chosen to implement PR in BSP mode.

3.4.2. Input Graphs. Our objective is to evaluate how SEGA performs for large graphs. Table 3.3 demonstrates the details of each of our input graphs. We have used a combination of synthetic graphs (Urand and RMAT [51]) along with real-world graphs such as Twitter. Previous accelerators used Twitter and RMAT 2^{26} as their largest graph input [29, 67]. We evaluate SEGA using Urand which has 2× more vertices and 1.5× edges compared to Twitter, as the input to our workloads.

3.5. Evaluation

To evaluate the performance of SEGA, we conducted a thorough evaluation across multiple dimensions. This includes (1) performance comparison with PolyGraph, (2) strong and weak scaling, (3) estimation of resources needed for PolyGraph, Dalorex, and SEGA to scale to WDC12 size graphs, and (4) the sensitivity analysis of GPNs to various design parameters, such as buffer size, and the size of memory used by vertex management unit to track active vertices in the main memory, and different vertex placements between PEs.

3.5.1. Scaling to Terascale Graphs. WDC12 [8] is a hyperlink graph representing 3.5 billion web pages and 128 billion hyperlinks. This is representative of future terascale graph analytics.

Accelerator	HBM Stacks	DDR Channels	SRAM/ eDRAM	Cores	# of Partitions
SEGA	14 (56 GiB)	56 (1TiB)	21 MiB	112	1
PolyGraph	272 (1.088 TiB)	-	4 GiB	2176	15
PolyGraph non-sliced	256 (1 TiB)	-	56 GiB	6400	1
Dalorex	-	-	1 TiB	249661	1

TABLE 3.4. Requirements to support WDC12. Note that SEGA has 8 cores per node, PolyGraph has 16 cores per node, and Dalorex has 256–4096 cores per node.

We compared the resource cost of SEGA to two recent works, PolyGraph [29] and Dalorex [67] for this graph.

We assumed all accelerators use the vertex size is 16 B and the edge size is 8 B. WDC12 requires 53 GiB of vertex capacity and an edge capacity of 959.15 GiB. Table 3.4 shows the system configuration to meet the minimum memory required to support WDC12. PolyGraph performs temporal partitioning that requires additional capacity, but we do not consider that. Dalorex, requires 1 TiB of on-chip capacity to support WDC12 without the need for temporal partitioning with a disk.

Table 3.4 shows that PolyGraph and Dalorex will have extremely high costs (many 100s of HBM stacks or a terabyte of SRAM) required to process tera-scale graphs. SEGA still requires significant resources to process large graphs, but by storing the high-capacity data structure in lower-cost memory (storing edges in DDR instead of HBM or SRAM) and the vertices in high-performance memory, SEGA scales to large graphs more practically than PolyGraph and Dalorex.

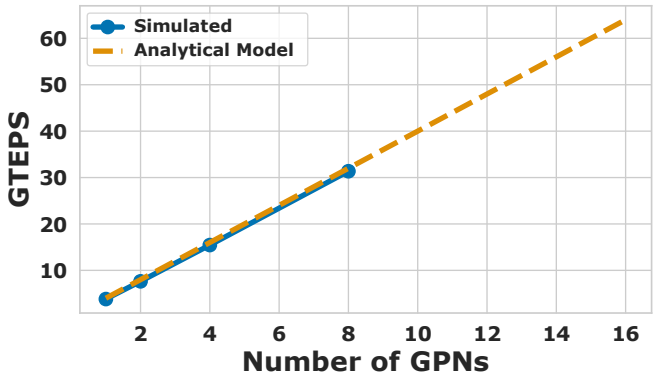


FIGURE 3.6. Comparing performance of Twitter to the analytical model.

We estimated the TEPS based on our proposed model, and the results are shown in Figure 3.6 using a dotted line. The solid blue line shows the simulated TEPS for these systems. As shown, the model predicts the performance accurately. For large graphs (high on-chip miss rate), the TEPS depends on the system and not the graph input. Therefore, we can use this model to predict the performance of large graphs based on its capacity requirements and calculate its TEPS.

3.5.2. Comparison to State-of-the-art. Figure 3.7 compares SEGA to PolyGraph with the same amount of off-chip memory bandwidth. In this comparison, both SEGA and PolyGraph are provisioned with 332.8 GB/s of off-chip memory bandwidth, which is equivalent to the aggregate bandwidth of one SEGA GPN with one HBM stack (256 GB/s) and four DDR4 channels (76.8 GB/s). While SEGA uses 1.5 MiB of on-chip memory (512 KiB for the on-chip buffer and 1 MiB for the active vertex tracker module), PolyGraph uses 32 MiB of on-chip memory.

Figure 3.7 shows that when running BFS for the Twitter graph PolyGraph is 30% faster than SEGA. In this case, PolyGraph can process Twitter graph using only 5 temporal slices. However, as discussed in Chapter 2, the overhead of switching temporal partitions grows significantly as the number of partitions grows. For other graphs (Friendster, Host, and Urand) that are larger than Twitter, SEGA outperforms PolyGraph. Overall, as the size of the graph increases SEGA gets higher speedup compared to PolyGraph, ranging from 1.12× faster for Host to 1.84× faster for Urand.

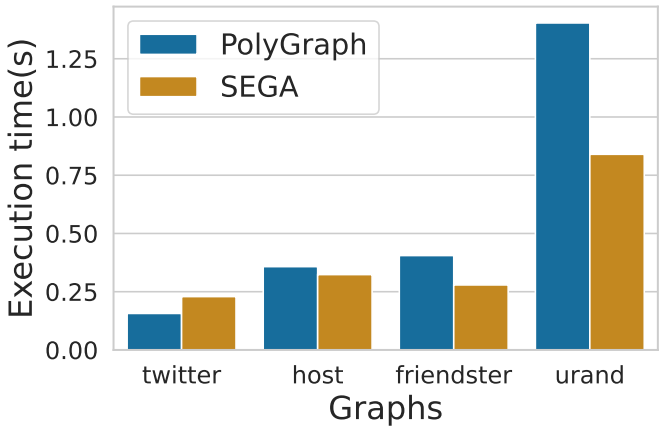


FIGURE 3.7. SEGA vs. PolyGraph (iso-bandwidth 332.8 GB/s) for different graph sizes. SEGA with 1.5 MiB on-chip memory beats SEGA with 32 MiB on-chip memory for larger graphs.

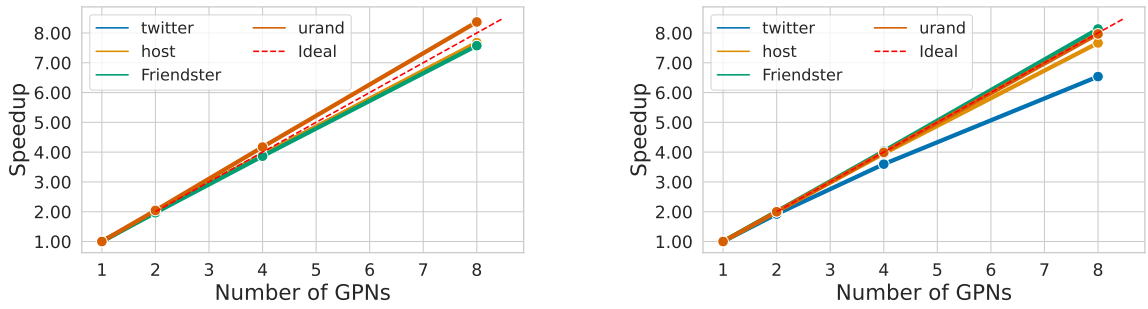
In all cases, SEGA utilizes 80% to 85% of the edge memory bandwidth. However, PolyGraph does not leverage the memory bandwidth efficiently. PolyGraph uses around 25% to 35% of the memory bandwidth for processing edges while the rest of the bandwidth is spent for switching partitions. As input graphs become larger, the time spent switching partitions constitutes a bigger part of the execution time which results in SEGA exhibiting higher performance compared to PolyGraph.

3.5.3. Scalability Analysis. We analyze both strong scaling (performance improvement as we scale the resources on a fixed size graph) and weak scaling (performance improvement as we scale both the size of the graph and resources).

For example, a system with 8 GPNs requires at most 128 GB/s bi-section bandwidth. Each port of AWGR is connected to each GPN using a 8 GiB/s bandwidth. AWGR fabric uses 8 waveguides, each carrying 8 wavelengths with 4 GB/s bandwidth per wavelength [79].

Figure 3.13 shows how the performance changes as we increase the number of GPNs for a fixed graph size (**strong scaling**). We only show BFS and BC due to limitations in space. Other workloads see a similar scaling trend. BFS is an example of a data-driven workload that experiences dynamic changes in the number of active vertices. In contrast, BC is a topology-driven workload in which the graph itself determines the active nodes.

In general, SEGA shows a near-perfect scaling in the performance as the number of GPNs grows. We observed a maximum 19% difference between the ideal scaled performance and SEGA's performance (Twitter in Figure 3.8b. For the Urand graph, performance grows beyond the ideal



(a) BFS (asynchronous)

(b) BC (bulk-synchronous)

FIGURE 3.8. Strong Scaling Analysis: How hardware resources (GPNs) affect performance for a fixed graph. This experiment shows that in SEGA strong scaling is ideal.

scaling due to increased work efficiency. Overall, SEGA achieves excellent scalability in performance as the number of GPNs increases.

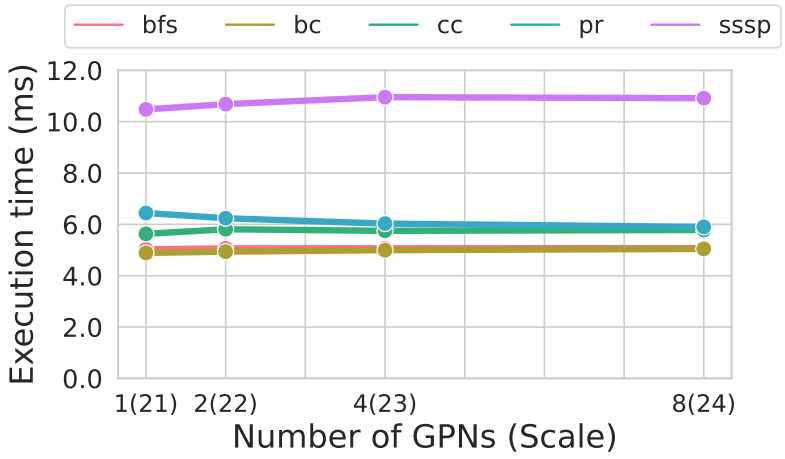


FIGURE 3.9. Weak scaling analysis when hardware resources (GPNs) and graph size increase. using synthetic graphs RMAT21-24 and BFS. Ideally, performance stays constant as the graph and GPNs grow together.

Figure 3.9 demonstrates how the performance improves as we increase the number of nodes for a fixed problem size *per node* (**weak scaling**). Weak scaling is typically employed for memory-bound applications that require a memory capacity beyond the capabilities of a single node. As illustrated in Figure 3.9, increasing both the resources and problem size will not lead to any performance degradation. In an ideal scenario of perfect weak scaling, the workload on a graph twice as large as a baseline should take the same amount of time for twice as many Graph Processing Nodes (GPNs) to execute.

3.5.4. Sensitivity Analysis.

3.5.4.1. *Sensitivity to Buffer Size.* As discussed in section 3.3 each PE uses a hardware buffer to hide the long off-chip random access latency. Due to the significant size of the graphs, it is not possible for the buffer to capture much locality in the accesses to the vertex memory. Figure 3.10 shows the buffer size does not affect performance. We ran BFS with two of our largest graphs (Twitter and Urand). Overall, we find there is less than 2% performance improvement when increasing the buffer size from 64 KiB to 4 MiB per PE (512 KiB to 32 MiB per GPN). The average bandwidth between two graphs and different buffer sizes is 6.4 GTEPS which is 80% of the peak achievable bandwidth in this system, which shows a high off-chip bandwidth utilization.

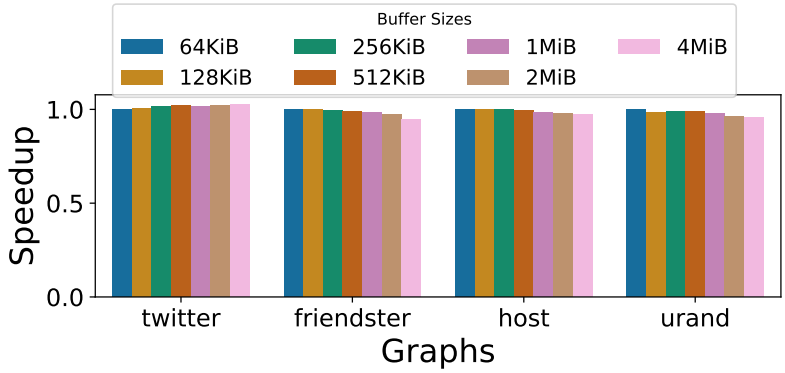


FIGURE 3.10. Performance sensitivity to buffer size for BFS

.

Moreover, Figure 3.10 demonstrates a small change in the execution time for both graphs which shows both the throughput and work efficiency in large graphs are independent of buffer size.

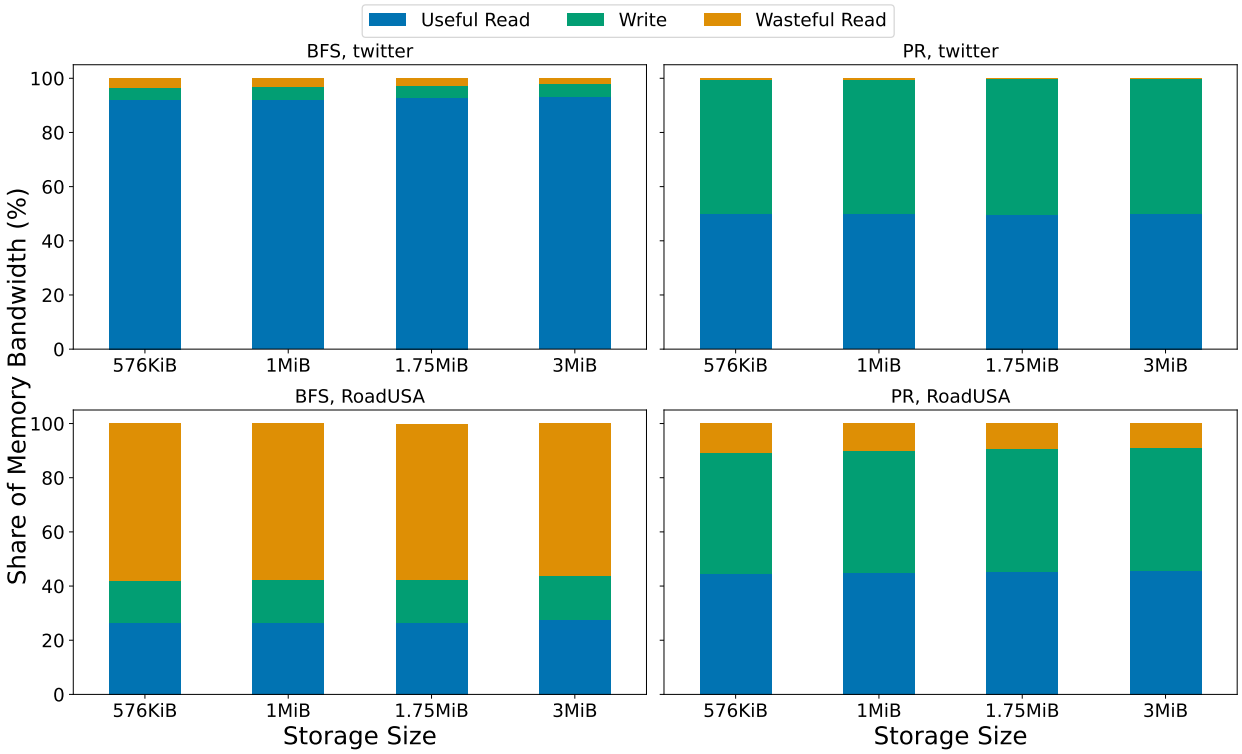


FIGURE 3.11. Breakdown of vertex memory bandwidth between useful reads (reading vertices for reduction or reading active vertices for propagation), writes, and wasteful reads (inactive vertices read while searching for active vertices). The bandwidth distribution is fairly insensitive to the storage Size.

3.5.4.2. *Sensitivity to tracker module size.* As discussed in Section 3.3.4, it is important to identify the effect of superblock dimension of the vertex management unit on the system behavior. Therefore, we have evaluated three different grouping dimensions of 32, 64, 128, and 256 for the implementation of the vertex management unit. In these cases, we are tracking 32, 64, 128, and 256 different memory locations with a single entry in the work tracking engine. These dimensions require 3 MiB, 1.75 MiB, 1 MiB, and 576 KiB of on-chip storage, respectively. We have used BFS and PR as representative programs and Twitter and RoadUSA as input for our experiments.

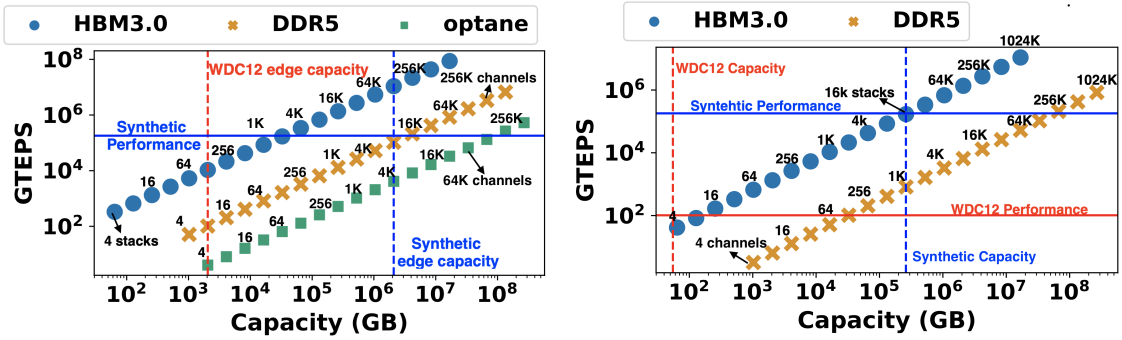
Since our blocking technique can not directly pinpoint the location of active vertices in the memory, it is important to evaluate the overheads introduced by the search for active vertices in each block. We measured the bandwidth waste of the vertex memory for different blocking dimensions. Figure 3.11 summarizes the division of the vertex memory bandwidth between useful reads, writes, and wasteful reads in proportion to the peak theoretical bandwidth of the vertex memory. The bandwidth marked as wasteful reads represent the bandwidth used to read inactive vertices while searching a superblock for active vertices. The distribution of bandwidth does not show observable change as the size of the vertex management unit changes.

However, there is a significant waste when running the RoadUSA graph. The main reason RoadUSA shows bandwidth waste is that it has a high diameter, and at most times there are a small number of vertices active Furthermore, high-diameter graphs commonly have smaller average degrees which in turn creates a smaller slack for the vertex management unit to search for active vertices in the DRAM. In this case, the prefetching mechanism of the vertex management unit aggressively over fetches resulting in this bandwidth waste. Also, as shown in Figure 3.11, dense frontier workloads such as PR (right) result in smaller wasted bandwidth as opposed to sparse frontier workloads. When the frontier is dense, the number of active vertices in a block grows, resulting in lower wasted bandwidth. Moreover, in all cases, we did not observe any change in our measured throughput as the size of the vertex management unit changed. However, for the case of running BFS with RoadUSA, we observed a drop in the work efficiency when we changed the size of the vertex management from 1 MiB to 576 KiB. For all other evaluations of our performance, we have used 1 MiB as the size of our vertex management unit.

3.5.4.3. *Sensitivity to Spatial Vertex Mapping.* Figure 3.12 shows the sensitivity to different vertex placement mechanisms. We compared a placement that is load-balanced and locality-optimized

(using RABBIT [13]) and random vertex assignment with no preprocessing cost. We observed that locality optimized shows at most 20% improvement compared to load balanced due to better overall work efficiency that is achieved from lower network traffic. However, RABBIT requires a time complexity proportional to the number of edges. in contrast to load-balanced and random that require no preprocessing cost.

3.6. Applications of Model



(a) Edge memory:
Desirable region: right-side of capacity limit.

(b) Vertex memory:
Desirable region is the Upper right quadrant.

FIGURE 3.13. Performance vs. capacity: (a) Edge memory the dotted vertical line indicates the required edge capacity. (b) Vertex memory Uses the performance (TEPS) in addition to the required vertex capacity (dotted horizontal line). We consider $\alpha = 1$.

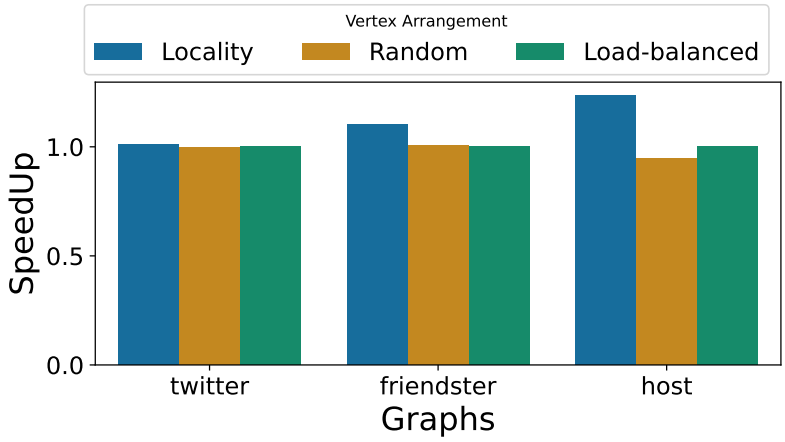


FIGURE 3.12. Performance sensitivity to spatial partitioning normalized to load balanced optimized using BFS workload.

We show three concrete use cases (or applications) for the proposed model. In the first use case, we show how to use the model to *create* a balanced system taking advantage of a heterogeneous memory such as HBM3, Intel® Optane™, and DDR5 for a given graph. Next, we show how given a homogeneous memory system, the proposed model can be used to calculate the locality required from on-chip memory to store vertices and edges in the same memory. This could be useful to someone developing the microarchitecture of the accelerator. In the third use case, we compare the system generated from our model with a high-performance computing system given a graph, algorithm, and performance.

In the first two applications, we use the WDC12 hyperlink graph [8]. We use 8 bytes to represent edges and 16 bytes for vertices. The accelerator for WDC12 requires 55 GiB for vertices and 2 TiB of edges. We consider the Breath First Search (BFS) algorithm and assume our accelerators have no on-chip resources i.e., $\alpha = 1$. In our partitioning scheme we assume 80% more inter-slice edges than intra-slice (γ). We assume our accelerators have 2000 data pins. Note, that these are just assumptions to illustrate the specific applications. The proposed model is not restricted to these assumptions.

3.6.1. Use Case 1: Creating a Scaled-out System. In this section, we address the question *How to create a balanced Scale-Out graph accelerator given graph input and a given performance target?* The proposed model provides the required memory devices for vertex and edge memory and network bisection bandwidth to answer this question.

Edge Memory: Figure 3.13a shows the capacity-performance relationship for different memory technologies using Table 3.1 Row 1 and Row 2. The right side of the red dotted line shows the region of interest for edge memory capacity. HBM3.0 requires more than 128 stacks to scale to WDC12 graph, and although it provides a significant amount of performance, it also has a high cost. Optane provides a significantly larger capacity with 32× fewer channels than HBM3.0 stacks, but with less performance due to its lower read bandwidth. Given DDR5's improved cost-performance tradeoff, we choose its performance (100 GTEPS) as the target accelerator performance for the bottleneck analysis for the rest of the system.

Vertex Memory: Figure 3.13b shows the performance-capacity trade-off for the vertex memory. The red dashed and solid red lines show the minimum capacity and performance requirements

respectively. The upper quadrant region of these lines shows the memory systems with both performance and capacity considerations.

Network: For our network constraints we consider near memory processing system architecture due to its low bisection bandwidth requirements. With 8-byte message size and using the equation in Table 3.1 Row 5, the required bisection bandwidth for WDC12 is 640GB/s.

Answer: Using our model, we find that 8 accelerators each with one stack of HBM3.0 as vertex memory and one DDR5 channel as edge memory in a near memory processing configuration is the best for the given performance target (100 GTEPS) and the graph (WDC12).

3.6.2. Use Case 2: Locality Calculation. The proposed model can be used to calculate the locality requirement (i.e., α) for the accelerator while using homogeneous memory technology for edges and vertices. In this example we consider using only DDR5 devices.

From our previous calculation we will use eight accelerators, and we will now use two channels of DDR5 each (Figure 3.13b). Eight channels of DDR5 provides the required capacity for the vertex memory (right-side of dashed red line). Eight channels of DDR5 only provides 6.4 GTEPS with $\alpha = 1$. To increase the performance to 100 GTEPS our model determines α must be less than 0.064 using Table 3.1 Row 3. Thus, if the accelerator's on-chip memory for vertex information was a simple cache it would require a hit ratio of 93.6% for a balanced system with DDR5 channels for both edge and vertex with minimum over provisioning on the capacity. Thus, our model shows that in a homogeneous memory system, unless the accelerator can find significant locality, the vertex memory will be the bottleneck.

3.6.3. Use Case 3: System Evaluation. Next, we compare the results from our model to a real example of a highly-scalable graph analytics system. We use our model to estimate the performance of the new Sunway supercomputer running BFS on a synthetic graph with 17.56 trillion vertices and 281 trillion edges and compare our result with the performance reported by Cao et al. [**21**].

Sunway supercomputer is equipped with 40 million processing cores across 103,912 processing units each with eight DDR3 channels. Using our model we predict a performance of 71,680 GTEPS with the Sunway supercomputer without any optimization ($\gamma = 1$, $\alpha = 1$). However, Cao et al.

achieves 180,792 GTEPS (2.5× better performance). This discrepancy is due to their novel 3-D partitioning method which leads to a lower γ ($\gamma \approx 0.4$).

Furthermore, we can use our model to create a *balanced* system for the same synthetic graph and a performance target of 180,792 GTEPS and compared our system against Sunway supercomputer. The proposed balanced system from our model requires 256 TiB of memory for storing the vertices and 2 PiB of memory for storing the edges. The model shows the balanced design only needs 8192 accelerators each with 2 DDR5 channels and 2 HBM3.0 stacks for edge and vertex memory, respectively. The system derived using our model uses 12× fewer processors (accelerators tiles) and 4× smaller memory by taking advantage of heterogeneous memories to better match the capacity-bandwidth tradeoffs inherent in graph processing.

3.7. Conclusion

As the size of the graphs increases, by orders of magnitude in the future, simulation is no longer viable for conducting a system-level design space exploration of graph accelerators. We need an analytical model that can help an architect make high level design decisions such as the number of tiles, what is the target network bisection bandwidth, what mix of memory technologies make sense, what are the trade-offs between a disaggregated memory configuration and a near-memory configuration, and how to build a balanced system for a given graph size and given performance target. In this chapter, we present a high-level performance model for large-scale graph processing and how to use this model to answer these questions and a scalable graph accelerator that does not rely on a large on-chip memory to mask the irregular accesses to the off-chip memory.

Currently the proposed model targets asynchronous vertex-based graph programming paradigms. In the future we plan to extend this model to linear algebra-based formulation of graph analytics [85] and dynamic graphs and use this model to drive the design of the microarchitecture of the accelerator tile itself.

SEGA achieves high performance for large graphs by creating a balanced system. Compared to previous hardware accelerators, SEGA uses off-chip memory bandwidth for both sequential edge read and random vertex read and writes. However, the throughput in SEGA for different graph sizes will remain constant, while other studies require temporal partitioning, which results in degradation of performance as the size of the graph increases. In addition, we propose a scaled-out

mechanism with an all-to-all optical interconnect that allows the design to scale to multiple cores without network overheads. Combining these two insights, the scalable SEGA architecture charts the path toward tera-scale graph analytic accelerators.

CHAPTER 4

Enabling Large Scale Graph Accelerations Using Silicon Photonics

The end of Dennard scaling restricts the ability of software frameworks to scale performance by utilizing larger processors due to the dark silicon effect. This situation encourages the development of custom hardware accelerators, designed for specific application domains, which can be significantly more efficient in terms of performance and power. As we face an exponential growth of data and an AI-driven transformation of the modern world, there is a demand for computing systems that can keep pace. Graph processing systems that can manage graphs with trillions of edges pose new challenges, necessitating innovative approaches to architecting computing systems that can scale effectively. A significant portion of industry workloads, such as graph workloads, consist of a small set of repetitive tasks that could greatly benefit from specialized units for execution. Scalability for handling large graphs is achieved by dividing larger graphs into slices that are processed either sequentially by the accelerator or concurrently using multiple accelerators.

In this work, we focus on the *scaled-out* approach, in which to process large graphs, we increase the number of accelerators. This approach is similar to going from single-core to multiple-core processors. The main challenge of scaling out is how to compose such a multi-accelerator system. To answer this question, it is important to understand the traffic pattern between the accelerators and also between the accelerator and the memory system. Furthermore, we need to detect the system requirements to package this accelerator which depends on the type and number of memory systems and the interconnect network.

We show that the interconnect requirements of large-scale graph processing systems integrate well with the unique strengths of photonic interconnects, such as high radix networks, low latency, and low energy per bit across long distances. These advantages of photonics can be synergized with emerging 3D and chiplet-based integration technology to create rack-scale or warehouse-scale systems for high-speed predictive data analytics that can enable new applications in many disciplines.

Large-scale graph processing presents three main challenges:

- To process graphs with trillions of edges, we need hundreds of terabytes of memory and hundreds of terabytes per second (TB/s) of bandwidth to move data in and out of the memory subsystem.
- Graph workloads exhibit low arithmetic intensity and the underlying memory access patterns are highly irregular with little locality. As a result, traditional architectural techniques such as caches and prefetchers do not work well.
- The size of accessed data in graph workloads is relatively small, typically 4 or 8 bytes in size. However, memory is accessed in chunks known as cachelines, which are either 64 or 32 bytes. This results in low cache utilization, out of 64Byte cacheline data only 8 bytes of that is useful.

These challenges largely mean that conventional CPUs with deep cache hierarchies and GPUs that rely on high arithmetic intensity to hide memory latency are not suitable to meet the demands of scalable graph processing. Therefore, the architecture community has been looking into new approaches such as in-memory processing and dedicated domain-specific hardware accelerators to speed up graph processing. However, the proposals in current research literature can only handle modest-size graphs (for example, the Twitter follower graph used in many benchmarks has only around 40 million vertices and 1.46 billion edges). It is not clear how these accelerators can be combined to scale to trillions of edges with good strong and weak scaling properties and energy efficiency.

In the previous chapters, we showed an analytical model that characterizes the system-level requirements to scale to a large graph and/or to achieve a certain performance. Furthermore, we showed the scalability limitations of previous graph accelerator nodes and proposed the architecture of a graph processing element that can process any size graph that fits in its memory without the previous limitations. The new accelerator design allows us to achieve strong and weak scaling as long as the network is not the bottleneck. In this chapter, we focus on the interconnect and how to organize these graph processing nodes. Furthermore, we also propose a package scheme that enables us to scale to peta-size graphs.

SEGA consists of multiple standalone processing elements (PEs) with dedicated vertex and edge memory. The contributions of this work are:

- By creating an all-to-all network between PEs we can detect the traffic distribution between PEs and calculate the bandwidth requirements (average and peak bandwidth) between PEs.
- We utilize the strengths of photonic interconnects, such as bandwidth density, low latency across long distances, and the emerging 3D and chiplet-based integration technology to create large scale graph accelerator systems.
- We evaluate the impact of different network topologies on the performance and create a scalable network between PEs.

The rest of this chapter is organized as follows: In Section 4.1 we present a background on the challenges of chiplet-based design and solutions for packaging. We analyze the network bandwidth requirements, and traffic patterns, in Sections 4.2 and 4.2.2.

4.1. Background on Chiplet-base Systems

Processing elements are small computation units. Multiple of these graph processing elements can be organized into a node which we refer to them as a graph processing node (GPN). GPNs can be considered as chiplets. Similar to chiplets we can use multiple GPNs and create a large-scale graph processing accelerator. To package the graph processing nodes we look into the packaging technologies that is used for chiplet in the HPC systems. In this section, we focus on the challenges of chiplet-based designs and their packaging solutions.

4.1.1. Challenges of scalability. To scale to tera/peta scale graphs we need to increase the number of GPNs (chiplets). To enable further scaling to large graphs the number of chiplets in the system can be increased. Scaling these chiplets has several key challenges that need to be addressed. Fotouhi et al. [**36**] categorized these challenges into different categories:

1. Interconnection Challenge. The energy and latency of electrical interconnects can be significantly impacted by distance. To overcome these disadvantages, a popular solution is to only allow interconnecting adjacent chiplets without excessive crosstalk and energy overheads. This leads to low-radix/high-diameter topologies with high average hop counts in which each inter-chiplet hop imposes tens of nanoseconds latency [**9**]. Given these latency overheads, inter-chiplet communication in general can now significantly degrade system performance and thereby limit scalability.

2. Packaging Challenge. To fit more chiplets into a single package, larger substrates are needed. While silicon interposers provide large IO density, they are too costly for the system sizes used in current HPC nodes, which mostly use less expensive organic substrates with lower IO density. However, to meet the future bandwidth demands of inter-chiplet links, high IO density is essential. Silicon bridges integrated into organic substrates can connect the edges of closely-coupled chiplets with high IO density, but they can only connect chiplets that are physically adjacent. Therefore, there is a high demand for interconnects with high IO density and energy-efficient signaling over long distances that can be integrated into a cost-effective organic package substrate.

4.1.2. Packaging Technologies.

4.1.2.1. *Multi-Chip Modules (MCMs).* MCMs mount and connect chiplets with high-density interconnects (HDIs) on the package substrate using wire-bond or flip-chip technology [84]. MCMs typically utilize organic package substrates as these are not manufactured in the foundry and therefore much cheaper. In addition, no further processing steps such as 2.5D integration and additional processing steps for the vertical interconnects are needed. Thus MCMs the cheapest option from both material and processing costs, making them an attractive option for systems of larger scale.

One of the challenges of MCMs is that flip-chip interconnects offer relatively low IO pin densities, restricting off-chip(let) bandwidth. Having a low IO pin density causes "pin wall" due to the vast majority of pins that are dedicated to power/ground, leaving few pins to satisfy off-chip communication demands. In addition, high IO pitches can also restrict the number of connected chiplets, which leads to low-radix chiplets requiring networks with high diameters and average hop counts. Inter-chiplet hop latency has a large impact on system performance and leads to complex systems with high latency variations. By connecting distant nodes we can create a topology with low diameter; however, energy grows linearly with distance for electrical links, which makes this approach infeasible for chiplet-based systems.

4.1.2.2. *2.5D Integration with Silicon (Si) Interposers.* 2.5D integration places an additional silicon die on top of the package substrate, and the chiplets on top of the interposer. Chiplets connect to each other and to the package substrate through the interposer with through-silicon vias (TSVs) and μbumps. Interposers can be passive (interconnects only) or active (interconnects and logic). The main benefit of 2.5D integration is the substantially higher interconnection density

compared to MCMs which allows for higher maximum bandwidth or for lower energy per bit by reducing the data rates of the IO transceivers. High IO density can enable higher-radix switches on the chiplets (i.e., connect each chiplet to more other chiplets), thereby reducing network diameter. 2.5D integration with Si interposers overcomes the challenges such as low pin IO density however, Si is significantly more expensive than organic substrates and 2.5D integration requires additional (and more complex) processing steps. Even with high IO density, 2.5D integration cannot overcome the limitations imposed by electrical interconnects such as length.

4.1.2.3. *Silicon bridges.* Si bridges, like Intel's EMIB technology, aim to solve the limitations of both MCM (poor interconnection density) and 2.5D integration (high cost for Si interposer) by embedding small and thin Si chips ("bridges") into an organic package substrate to interconnect the edges of adjacent chiplets. Si bridges offer high IO density with latency and energy metrics similar to on-chip wires and enable short interconnects through tight packaging with just 100μm between chiplets. Si bridges thereby offer a more scalable solution by combining the low material costs of organic substrates with the high IO bandwidth density of Si interposers. Just like Si interposers and MCMs, Si bridges utilize electrical interconnects and thus impose the same distance-related energy limitations, and thereby the same network radix/diameter problem. Consequently, Si bridges alone cannot overcome the NUMA, interconnect, and scalability challenges in chiplet-based computing systems. Systems with one large chiplet and several (much) smaller chiplets could exploit the high IO density of Si bridges to directly connect the large chiplet to each small chiplet, but inter-chiplet traffic would likely be bottlenecked by the crossbar on the large chiplet, limiting the scalability of this approach.

4.1.2.4. *Silicon Photonics.* SiPhs-enabled integrated optical interconnects have properties that can be used to address the challenges posed by electrical interconnects. Optical technology offers energy consumption that is virtually independent of distance, near speed-of-light signal propagation latency, and high bandwidth density through wavelength-division multiplexing (WDM), which allows for parallel transmission on multiple wavelengths within the same optical link. Additionally, SiPh devices can perform wavelength-selective routing, allowing data to be routed based on the wavelength channel and enabling a chiplet to connect to multiple other chiplets through a single

waveguide. Furthermore, SiPhs can be integrated into organic package substrates, providing a solution to interconnection challenges while allowing for the use of a relatively inexpensive packaging substrate (compared to a Si interposer) [4].

4.2. SEGA a Chiplet-Based System for Large Graph Acceleration

In large-scale systems, ensuring scalability is crucial. To achieve scalability, we must prevent data movement between accelerators from becoming a performance bottleneck as we add more processing elements. For this purpose, an adaptive network capable of adjusting based on traffic is essential.

In our scaled-out system, we use the interconnect network to communicate messages to neighboring vertices, also known as inter-slice updates. If a graph is well-partitioned and has a low rate of inter-slice events, it does not require a high bandwidth network. However, creating such a well-partitioned graph with few inter-slice events involves a pre-processing cost. This cost is typically higher than the cost of the graph workload itself. Therefore, we aim to use an interconnect technology that can handle the traffic when the graph is randomly distributed among the accelerators.

To understand the maximum required bandwidth needed on the interconnect, we need to understand the maximum traffic that the graph accelerators generate. We use PEs as potential accelerator cores and study their network pattern and bandwidth requirements as the size of the system grows.

One of the key characteristics of SEGA is the connection of each PE to a dedicated vertex and edge memory. By assigning each vertex to a dedicated PE, this approach not only eliminates the atomic updates resulting from multiple PEs attempting to update a single vertex but also allows each PE to access the full bandwidth of its edge and vertex memory with reduced contention from other PEs. In addition, in this scenario, we decouple the memory accesses from the message-passing network between PEs. Each PE performs vertex reads/write, and edge reads using its dedicated memory bus and only uses the network between PEs to send inter-slice messages (messages propagated between an active vertex to its remote neighbors).

Recent works from Intel, the importance of SiPh in scaling accelerator cores to 10k nodes by using SiPh technology using a hyper-x topology to reduce the diameter of this network and

therefore reduce the number of router hops and the network latency. In the scaled-out PUIMA architecture, Intel uses optical fibers and fiber routing to communicate data at high bandwidth density Similar technology can be used in a SEGA-like architecture to create a system that can scale to petascale graph size. However, we need to understand the network requirements of SEGA and use an interconnect according to our system's bandwidth and latency needs.

Making efficient use of network bandwidth is another critical aspect as the available network bandwidth directly impacts system performance, cost, and power consumption. Communication patterns between compute nodes in modern workloads are typically not evenly distributed, and they cause high variances in link utilization between low-utilization computing-intense and high-utilization communication-intense phases. It is challenging to simultaneously provide sufficient bandwidth for high-communication phases between certain node pairs without wasting energy in low-utilization phases. Unfortunately, electronic switches, and in particular transceivers connecting them, have fixed bandwidth which prevents adapting the link bandwidth to current communication demands. This has led to systems that typically deploy multi-hop topologies (like Fat-Tree, butterfly, or clos topologies) to provide load-balancing capabilities for high utilization phases without excessively over-provisioning the network resources for low utilization phases. Previous works on reconfigurable bandwidth steering show how bandwidth reconfiguration can help improve the overall execution time of different HPC applications by improving network utilization [35,55]. The applications with phases in traffic distribution reconfigurations can play an important role in improving performance without the need to overprovision the resources, whereas in applications with uniform random traffic patterns reconfiguration will not be beneficial. Therefore, it is important to understand both the bandwidth requirement of the system along with the traffic distribution between nodes.

In this section, we look into the traffic pattern and bandwidth requirements between PEs in the graph accelerator. We use both the analytical analysis and the simulation to determine the inter-PE link bandwidth, the required bisection bandwidth, and the traffic distribution among PEs. We also use experiments to help us predict how the traffic needs increase as the number of PEs scales.

To characterize the interconnect network traffic, we implemented a cycle-accurate model of the system shown above in gem5, with a point-to-point network between the accelerator nodes.

We used live-journal as the input graph and sampled the traffic on each link every 5 us. In this particular system, we assumed an infinite link bandwidth between PEs.

4.2.1. Bisection Bandwidth Requirements. In Chapter 3 we used an analytical model to predict the bisection bandwidth requirements in a scale-out graph accelerator based on TEPS (traversed edges per second). In a balanced system, TEPS can be calculated based on the edge memory bandwidth. However, realistically, the edge memory bandwidth is not fully utilized during the execution. The cause of this under-utilization is caused because the edge memory and vertex memory are accessed at different rates. TEPS is dictated by the one that causes the bottleneck. Equation 4.1 shows the system's TEPS. In cases where the TEPS is dictated by the vertex memory or the network then the edge memory bandwidth is underutilized.

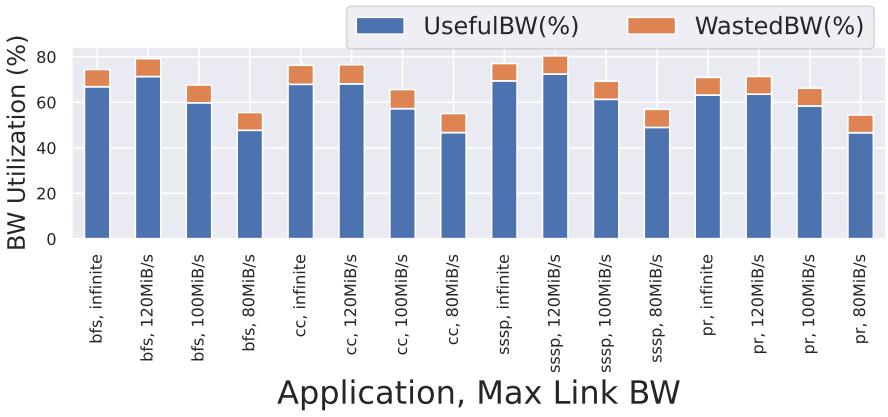


FIGURE 4.1. Edge memory bandwidth utilization of graph workload in an ideal point-to-point network when running live-journal graph.

$$\text{TEPS} = \min\left(\frac{\text{EdgeMemBW}}{SizeOf(Edge)}, \frac{\text{VertexMemBW}}{2 \times \text{atomSize} \times \alpha}, \frac{\text{BisectionBW}}{Sizeof(message)}\right) \tag{4.1}$$

Equation 4.2 can be used to calculate the realistic value of TEPS based on the edge memory when the edge memory utilization is known.

$$\text{Bisection Bandwidth} = \text{EdgeMemBW} \times \text{EdgeMemBW Utilization} \tag{4.2}$$

Figure 4.1 shows the edge bandwidth utilization of different applications in a system with 64 PEs that are connected together through a point-to-point network. We use an infinite link bandwidth in this experiment to show the network traffic pattern of graph applications without any restriction on the network. The average bandwidth utilization in a system with infinite link bandwidth is 76.1%.

Figure 4.1 also illustrates the percentage of bandwidth that is wasted. This waste occurs because not all edges read from the edge memory are connected to an active vertex. We access the edge memory based on the cache size, but the edge information is smaller than this size. As a result, there are times when the edges read are connected to neighboring vertices instead of the active vertex. Consequently, not all edges read from the edge memory lead to message propagation, resulting in wasted bandwidth as depicted in Figure 4.1. This figure highlights the percentage of wasted bandwidth caused by large memory access size, which results in reading redundant edges.

The figure also presents the bandwidth utilization when there are bandwidth limits on the interconnect links. With a link bandwidth of 120 MiB/s, the bandwidth utilization is similar or even larger (in the case of bfs, sssp, and pr) than in scenarios with infinite link bandwidth. However, as the maximum link bandwidth decreases, so does the bandwidth utilization. This reduction in bandwidth is primarily due to the interconnect becoming a bottleneck in these scenarios. As indicated in Equation 4.1, the bisection bandwidth drives the TEPS. Due to bandwidth limits, the message generation engine experiences backpressure and cannot send messages to the network at a high rate.

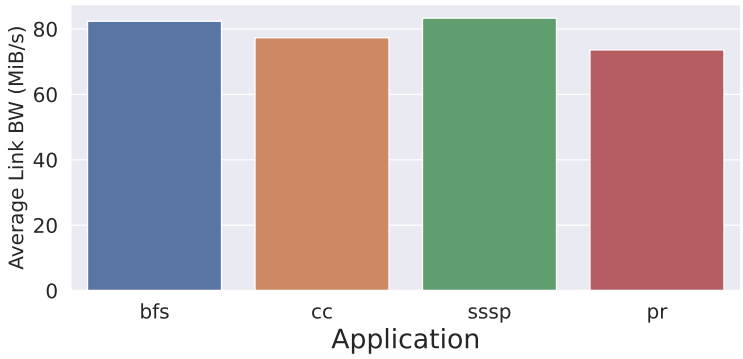


FIGURE 4.2. Average PE-to-PE traffic bandwidth with a point-to-point network with infinite bandwidth when running live-journal graph.

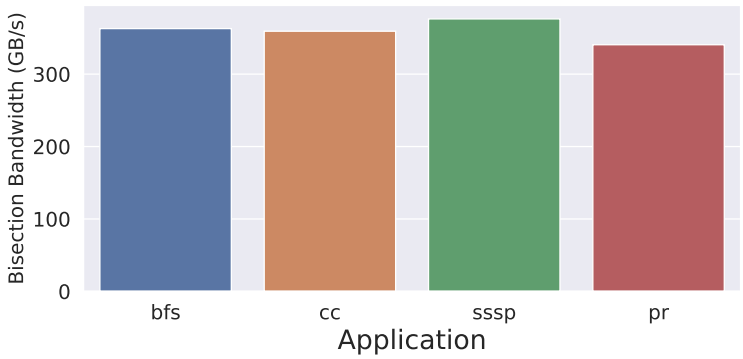


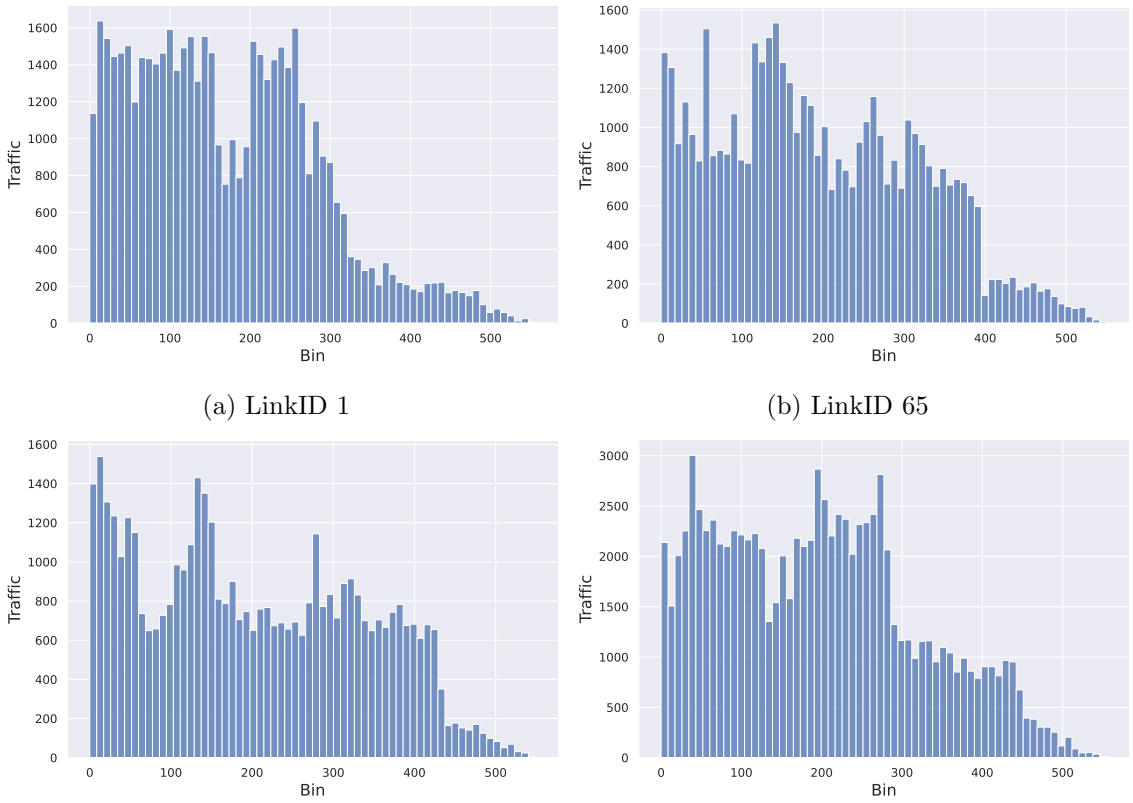
FIGURE 4.3. Required bisection bandwidth in an ideal point-to-point network with infinite bandwidth when running live-journal graph.

It is not feasible to create a network with unlimited link bandwidth. Therefore, to impose a restriction on link bandwidth, we need to understand both the average link bandwidth between PEs and the distribution of traffic on each link. Figure 4.3 shows the average network bisection bandwidth for all the applications. This shows that in a system with 64 PEs that can scale to graphs with terabytes of memory footprint, the average bisection bandwidth is 350 GiB/s, which is 76.1% of the system's maximum bandwidth.

In a system with 64 processing elements (PEs) and full edge bandwidth utilization, the network bandwidth reaches 468 GiB/s. Opting for a 120 MiB/s link bandwidth in a point-to-point network can effectively support maximum network traffic.

A point-to-point network is particularly desirable for applications with uniform random traffic patterns. However, in scenarios where traffic distribution is uneven across links, we encounter situations with hotspot links (high demand) and underutilized links. To mitigate this, a thorough study of the traffic pattern on each link is essential.

4.2.2. Network Traffic pattern. In the previous section, we looked into the bisection bandwidth required to create a system with 64 PEs. We study 64 PEs since, based on our model, SEGA can use 64 PEs to scale to WDC12 [8], the largest publically available graph.



(a) LinkID 1

(b) LinkID 65

(c) LinkID 1025

(d) LinkID 3969

FIGURE 4.4. Histogram of network traffic on three random links.

A combination of bisection bandwidth and the distribution of traffic among PEs helps us calculate the link bandwidth among PEs. In a uniform random distribution, each PE communicates with others at a uniform rate. Therefore we can calculate the link bandwidth by simply dividing the bisection bandwidth by the number of links. Our goal is to understand the traffic distribution on the network which consequently helps us calculate the link bandwidth and type of network technology to use.

We used the same 64-PE system with infinite link bandwidth to capture the traffic pattern between PEs. Figure 4.4 shows the histogram of traffic on four random links in the system. This figure shows that in each time sample, the number of packets sent on each link is not uniform. The other observation is that the traffic does not follow any type of particular distribution. Therefore, the traffic distribution on each link is random. During the execution of the application, there are

phases in the application where the link is underutilized, and there are time samples where the traffic is heavy.

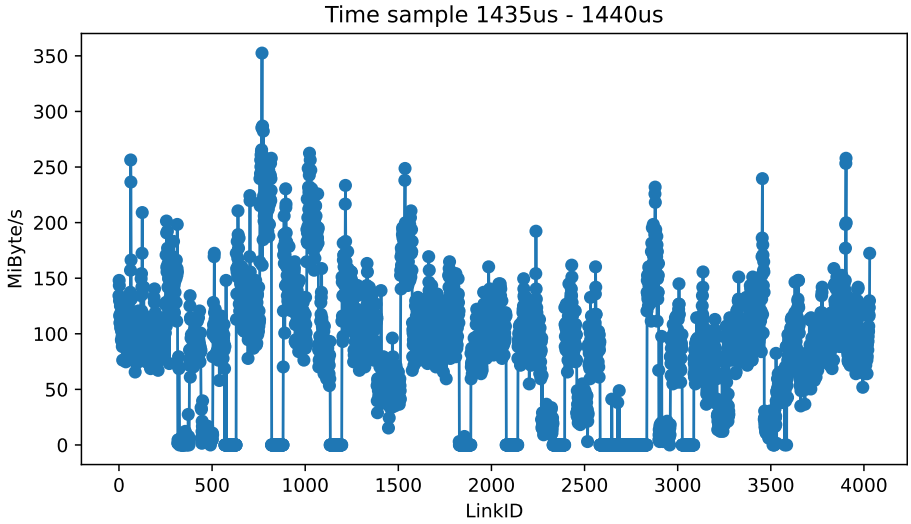
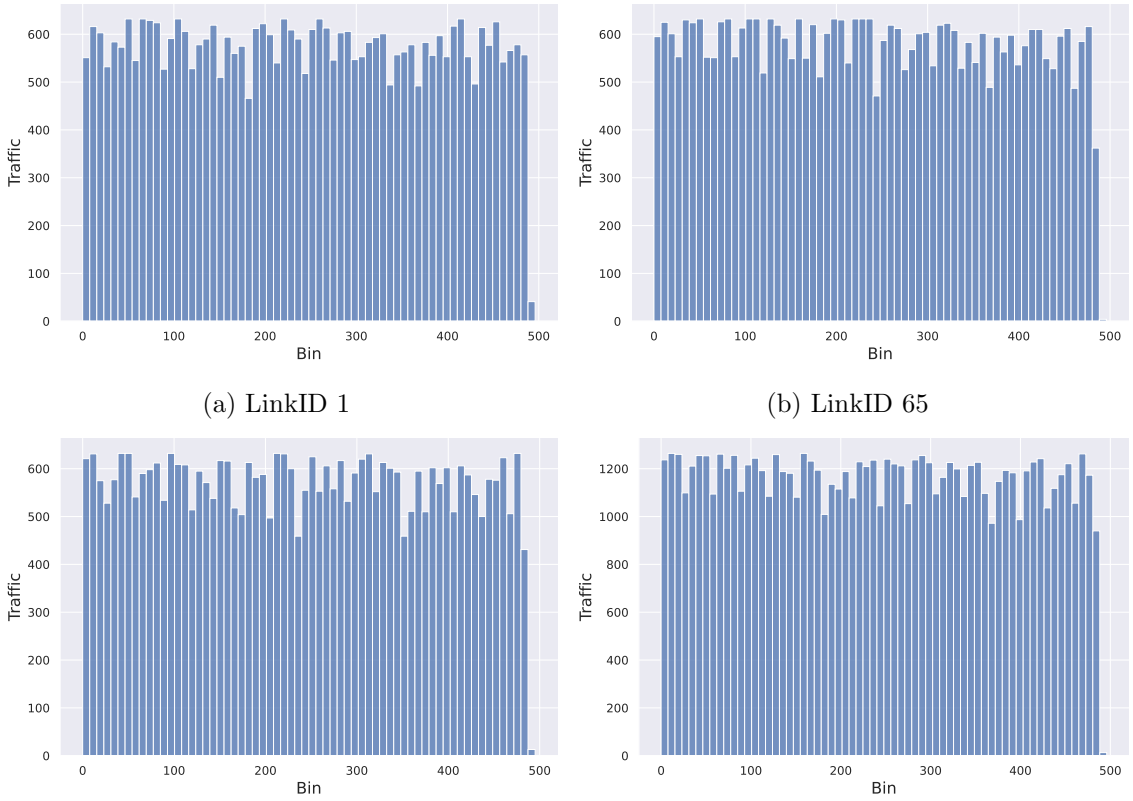


FIGURE 4.5. Average bandwidth of all the links in a certain time sample.

While Figure 4.4 shows the histogram of traffic on each link, we need to understand how traffic distribution changes between links. Are there situations where there are more inter-slice events between PE_i and PE_j than between PE_i and PE_k? Figure 4.5 shows the average link bandwidth within a random 5 μs time sample. In this time sample the average link bandwidth changes between 0 and 350 MiB/s, showing that within this time sample some links are completely underutilized. Due to this traffic nonuniformity a "hot spot," typically but not uniquely is produced on certain links which can produce effects that severely degrade all network traffic.

In such a system with irregular bandwidth assignments, the designer can maintain the performance by either using a link bandwidth that matches the traffic on the congested link, or they can use a reconfigurable interconnect to steer the bandwidth from underutilized links toward the hotspot links. In the former case, the network resources are over-provisioned for the low-utilization links. Thus reconfiguration can play a vital role in improving performance without overprovisioning bandwidth.



(a) LinkID 1

(b) LinkID 65

(c) LinkID 1025

(d) LinkID 3969

FIGURE 4.6. Histogram of network traffic on three random links.

Figure 4.2 shows the average link bandwidth for different applications. Based on these values and the bisection bandwidth, we used 120 MiB/s as the link bandwidth. By adding this restriction to the link, the traffic pattern on each link changes. Figure 4.6 shows the histogram of link traffic, showing that the traffic is uniform compared to Figure 4.4, by adding the traffic bandwidth restriction, the extra traffic from heavy time samples rolls over to the time samples with low traffic, which results in a more uniform distribution. In addition, with infinite link bandwidth, we are allowing the PEs with bursty accesses to utilize all the resources of the destination PE, preventing other PEs from sending requests to this destination PE. In contrast, by adding bandwidth restriction, we are only allowing a limited number of messages to be sent to a particular PE from other PEs. Which allows a more uniform access pattern between PEs. This is one of the reasons that in Figure 4.1 we get better bandwidth utilization with 120MiB/s compared to infinite bandwidth.

In addition to better bandwidth utilization, work efficiency can also be impacted. The work efficiency can be improved by coalescing multiple updates to the same vertex. At the message processing engine, more PEs can send their request to a particular vertex, allowing for more coalescing at both on-chip memory and the vertex memory. There is a higher degree of spatial locality in requests sent from multiple PEs compared to those sent from a single PE. Figure 4.10 shows the normalized number of coalescing compared to infinite link bandwidth. This figure shows that bandwidth restriction allows for better coalescing.

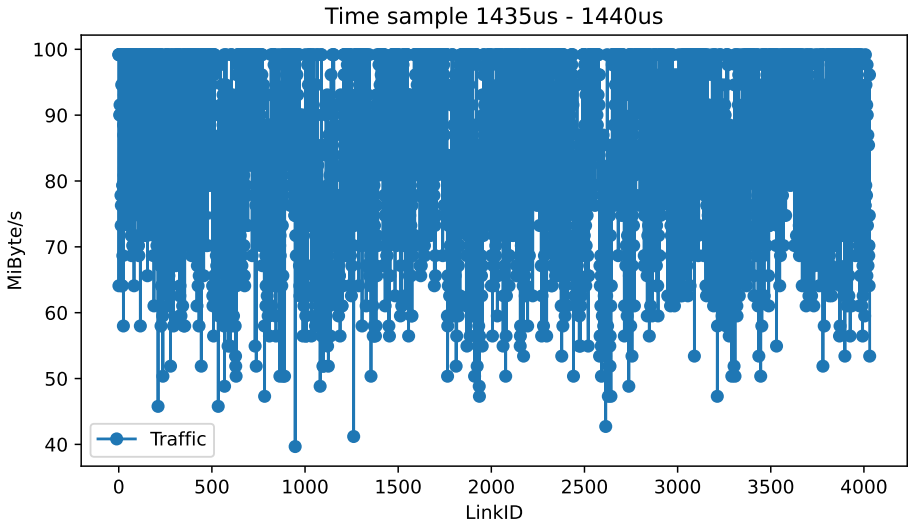


FIGURE 4.7. Average bandwidth of all the links in a certain time sample.

Opportunities for Reconfiguration: Figure 4.7 shows the average bandwidth of links compared to other links within a time sample. Compared to the infinite link bandwidth, the average link bandwidth changes between 40 MiB/s to 100 MiB/s, showing the traffic between links is uniform compared to the infinite link bandwidth where the bandwidth changes from 0 to 350 MiB/s (Figure 4.5). Thus, by adding bandwidth restriction, the traffic becomes more uniformly random compared to the infinite link bandwidth. Bandwidth reconfiguration is not a good solution for uniform random distribution traffic.

Consequently, since the traffic is uniformly random, we can assign the link bandwidth restriction by dividing the bisection bandwidth by the number of links. Figure 4.8 shows how adding link bandwidth restriction impacts the overall performance. This figure shows that 120 MiB/s maximum link bandwidth can achieve similar/higher performance compared to infinite link bandwidth. This is

due to both better edge bandwidth utilization that causes higher overall TEPS and a higher number of vertex coalescing that ultimately improves the work efficiency. Figure 4.10 shows the normalized number of coalescing compared to the infinite link bandwidth. By assigning link bandwidth, we achieve higher/similar coalescing compared to the infinite link bandwidth. However, as we decrease the link bandwidth to 100 and 80 MiB/s throughput starts to decrease due to contention on the network. 120 MiB/s shows the highest coalescing rate with the highest TEPS across different networks.

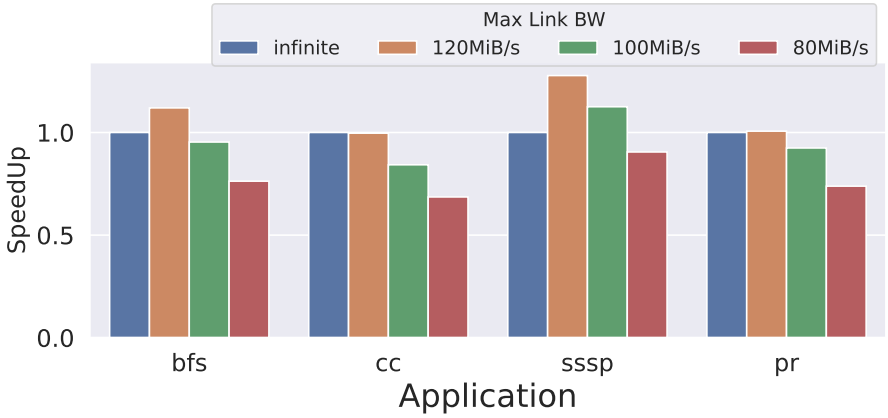


FIGURE 4.8. Performance for different link bandwidth restrictions. This figure shows that the link bandwidth of 120MiB/s shows similar or higher performance to a simulation with infinite bandwidth. Higher is better.

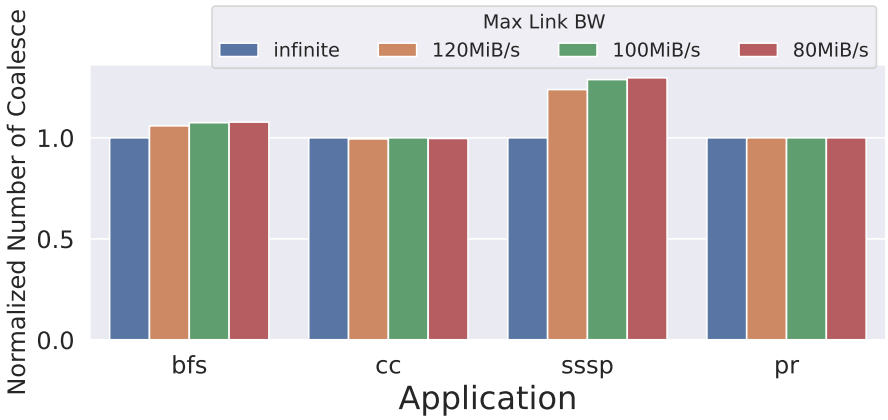


FIGURE 4.9. The normalized number of vertices coalesced. More coalesced vertices result in better work efficiency and a lower number of vertex memory reads. These values are normalized to the infinite link bandwidth

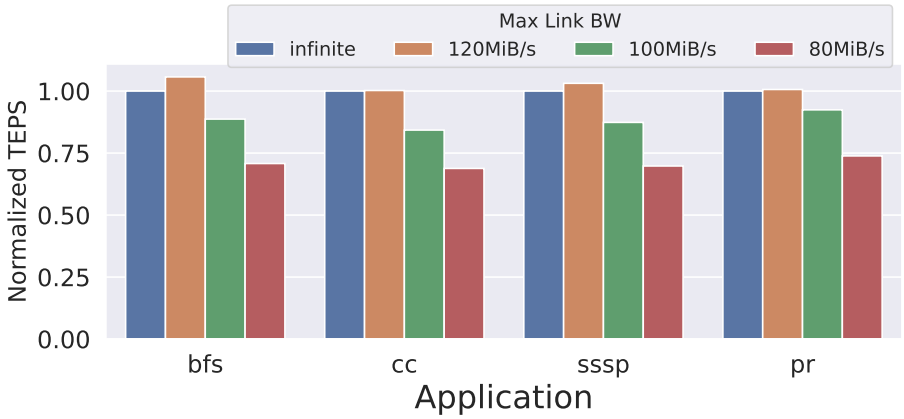


FIGURE 4.10. The normalized throughput. These values are normalized to the infinite link bandwidth

4.3. Packaging

During our initial experiments, we assumed 64 PEs all connected point-to-point in a single package. However, in such system there are 32 DDR4 channels and 8 HBM2 stacks. Such a system requires 17,408 pins. One significant bottleneck of such a system and sustaining the scalability is processing power.

To maintain the processing power within tight power envelopes, we can rely on breaking monolithic chips into smaller "chiplets." Utilizing several smaller chiplets assembled using advanced packaging technologies, instead of one large monolithic chip, reduces costs by exploiting the higher yield of smaller dies. This approach incurs low performance and energy overheads through tight integration in the same package [36].

Due to the low traffic required between PEs, we are not limited by the link bandwidth but by the number of pins connecting each PE to the off-chip memory. Due to such a large number of pins required, we propose to organize a group of PEs into GPNs of graph processing nodes and only fit a number of GPNs into a die. To scale to larger graphs we increase the number of connected dies.

In the recent PUIMA architecture from Intel, the number of pins per die is calculated to be 3275 pins [5]. A single HBM stack and a DDR channel have 1024 and more than 184 pins respectively. In this design, we can connect 8 PEs, 1 stack of HBM2, and four DDR4 channels in one GPN. connect two GPNs into a single die. Two GPNs in a single die require 3072 pins for die memory interconnection.

Since the bandwidth requirement between PEs is small. We are proposing that PEs inside one GPN be connected through a point-to-point electrical interconnect. GPNs within a die are connected through a PCIe4 $\times 4$ lane with 16 GiB/s bisection bandwidth. Each die can support a graph with a maximum of 1 billion vertices and 1 trillion edges. For scaling to graphs larger than 1 trillion edges, we can scale the number of graph accelerator dies. In graph processing communications, latency is significant to system performance. Since the network bandwidth is low between PEs, the link latency plays an important part in determining the execution time. To ensure low data movement overhead the on-die network needs to be extended to an off-die network with low latency, by reducing the link propagation time and minimizing the number of hops.

To scale to WDC12 which is the largest publicly available real-world graph, our calculations show that we need 8 GPNs. In a scale-out 4-die system, the maximum bisection bandwidth required is 360 GiB/s. to support a 4×4 system with 22.5 GiB/s link bandwidth in a 4×4 can be supported using NVLink with 50GB/s link bandwidth.

Scaling to graphs larger than WDC12 or systems with more than 4 number of dies, the number of hops can cause a bottleneck. The required bisection bandwidth increases with the order of n^2 (n is the number of dies in the system). To reduce the impact of communication on performance, a low-dimeter topology with low network transmission time is required. Although electrical interconnects such as PCIe and NVLink can support this bandwidth, to create a low-dimeter interconnection electrical interconnects require large amounts of wiring and their energy and latency are dependent on distance.

Wavelength-selective routing allows GPNs to connect to multiple other GPNs through a single optical IO pin (addressing them on different wavelengths), enabling high-radix low-diameter networks with a distance-independent latency. Optical networks provide sufficient scalability in terms of crosstalk and power consumption to enable point-to-point connectivity. The bisection bandwidth in the array waveguide grating router (AWGR) also scales with the order of n^2. The bandwidth scalability along with low latency and high radix feature makes this device an ideal candidate for the inter-die interconnection fabric between the accelerator dies.

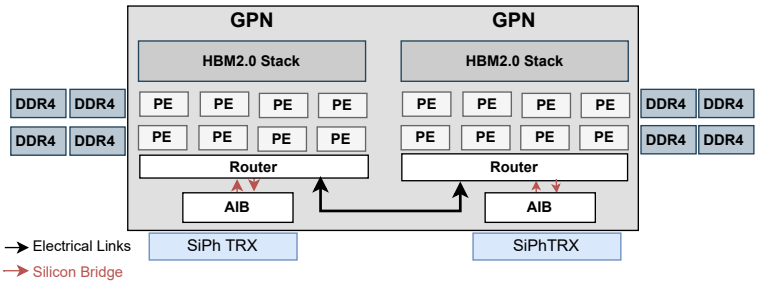


FIGURE 4.11. Architecture of single graph accelerator die

To connect the die to the SiPh transceivers we use Intel's Advanced Interface Bus (AIB). AIB moves data from microbumps on one chiplet to microbumps on another adjacent device. The fine pitch of new high-density packaging microbumps keeps the real estate required for the interface modest. Current optical I/O chiplets from Ayar labs can also be used to connect the Graph accelerator dies to the optical transceivers through PCIe [4]. PCIe uses high speed through a few wires, whereas AIB uses a wide parallel interface supported by new high-density packaging technology. By running each wire of the interface at a relatively low speed in AIB, the circuitry for each transmitter and receiver is greatly simplified and uses little silicon area. AIB connections can be made by wires on an interposer or by using a bridge technology like Intel's Multi-Die Interconnect Bridge (EMIB) bridge. We propose using EMIB to interconnect optical transceivers to the routers to extend the on-die network between dies.

4.4. Methodology and Evaluation

We have implemented a model for the intra-die network shown in Figure 4.11 and the inter-die network with AWGR in gem5. We evaluate the networks with both synthetic and application traffic to study how the networks perform under graph workloads and how they would perform based on the network injection rate (which allows us to scale to large systems without being limited by the long simulation latency). For the electrical interconnection, we are using a 4-cycle router latency working at 1GHz frequency. The internal GPN links use a 120MB/s data rate while the electrical link between routers is 7.5 GiB/s (8 routers, 8 links each, 120 MiB/s). For the optical interconnection, we are using a 32 Gb/s link bandwidth with a transmission time of 1 ns and an optical-to-electrical-optical latency of 2 ns.

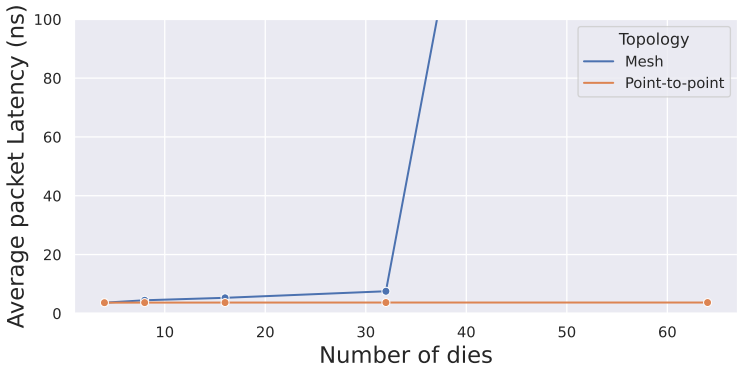


FIGURE 4.12. Iso-bisection bandwidth test: The average network packet latency for point-to-point vs. mesh topology. Higher link bandwidth for mesh compared to the point-to-point topology. The network latency increases for the mesh topology for higher number of dies due to more number of average network hops.

In the synthetic traffic evaluation, we used gem5 [57] with garnet3.0 [20] for detailed network performance simulation, and for the detailed graph simulation, we used our own implementation of gem5 implementation of SEGA and the router implementation to model the graph accelerator and the interconnect. We use uniform random traffic for the synthetic traffic simulation that matches the inter-GPN traffic distribution. We used Mesh topology as the baseline since it is the topology used in the previous graph accelerators [29]. Figure 4.12 shows the iso-bandwidth test and the average packet latency in the network as the number of dies increases. The link bandwidth in mesh topology is O(n), where n is the number of dies. In the mesh interconnect, the average number of network hops increases as the size of the network increases, resulting in higher overall network latency.

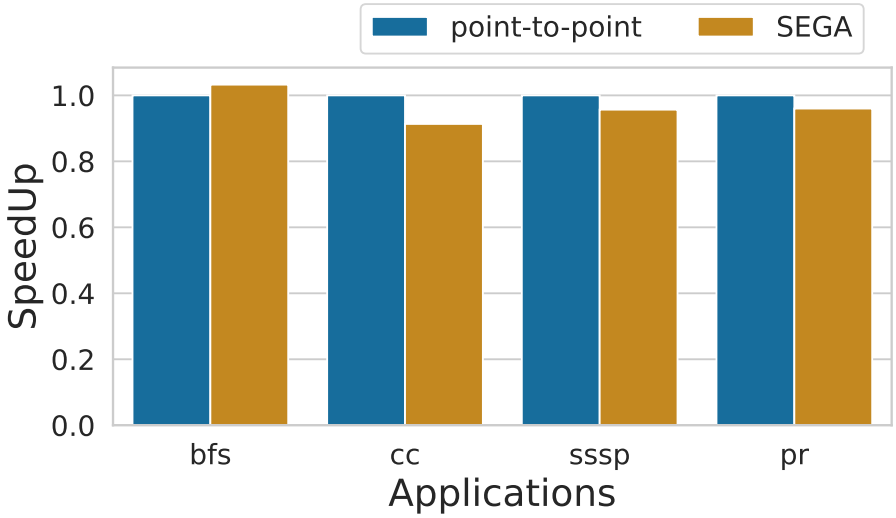


FIGURE 4.13. Iso-bisection bandwidth test in an 8 GPN system: comparing the point-to-point network in a system without pin wall vs. comparing with a SEGA packaging with two GPNs per die, connecting 4 SEGA dies

Figure 4.13 running our simulation results when running real graph workloads on our gem5 implementation of SEGA. In this case, we are comparing an ideal 64×64 PE system with point-to-point interconnection on a single die compared to a system similar to Figure 4.11 with 4 dies connecting through optical links and 16 PEs per die clustered into two GPNs.

Figure 4.13 shows that the performance of SEGA is similar to the ideal point-to-point connection mostly due to the bandwidth density and low communication latency that is provided by optical interconnections.

4.5. Conclusion

In graph applications, the data movement plays an important part in determining the performance. In addition to performance, scalability is another challenge for these applications. In this chapter, we looked at the characteristics of traffic between accelerator nodes and proposed a packaging scheme that helps the accelerator to scale out without performance degradation. While this chapter improved the data movement among the graph processing nodes, the accelerator memory interaction can still cause limitations, especially for the vertex memory since the vertices are the data structures with irregular memory access patterns. In the next chapter, we propose a high

bandwidth, low latency, and low energy memory system that can be a great candidate as the vertex memory in the graph accelerators.

In this new memory system, we rearchitect the memory bank and reduce the bank conflicts in the memory channel, furthermore, we co-design the memory controller and a silicon photonic-enabled data bus that reduces the contention on the data path and creates a system with high bandwidth and low memory access latency specialized for sparse workloads.

CHAPTER 5

Low Latency Memory

5.1. Introduction

Emerging applications, such as recommendation systems, mining large sparse graphs, etc., exhibit irregular memory access patterns with little data reuse and poor locality [38]. For these irregular workloads, the memory subsystem is increasingly becoming the bottleneck in modern computing architectures. The memory subsystem should not only provide high bandwidth but also low latency to achieve high performance for irregular applications [24, 32]. In addition, *variability* in memory latency is another concern as it limits the performance of computing systems [24] and increases the burden on the programmer. It is desirable that both the average memory access latency and its variability (e.g., as measured by the 95^{th} percentile) are low.

To address these challenges, there has been a resurgence of interest in DRAM microarchitectures and memory system designs. With the emergence of silicon photonics technologies, and chiplet-based architectures with 2.5D/3D packaging, there are new opportunities to co-design the various components of the memory subsystem. Recent advances in DRAM architecture, such as wider I/O enabled by 2.5D/3D packaging (as in HBM and its derivatives [44, 69]), higher data rates with serial links, and increased bank-level parallelism (again with HBM like technologies), have improved DRAM bandwidth significantly. However, often these bandwidth improvements come at the expense of additional latency and variability due to deeper queues in the memory controller to take advantage of the bank-level parallelism and serialization/deserialization (SerDes) latency [25]. There are also proposals [22, 42, 45, 47] in literature that explicitly address the latency question in DRAM microarchitectures, and most of these proposals simply take advantage of *locality* to reduce latency.

We argue that the main source of latency for irregular workloads in the memory subsystem is **c**ontention caused by *sharing resources* such as buffers, ports, data/command/control buses, and the DRAM cells where the data actually resides. Increasing these resources comes at a significant cost

and may have physical limits such as the number of pins (I/O pads) that can be placed in a given space. Thus, we must consider sources of contention in the *entire end-to-end path*, which includes the processor/memory interconnect, memory controller, and DRAM microarchitecture. In the past, end-to-end optimization of the memory subsystem was not feasible in commodity CPUs (though there has been a slow transition in this direction with integrated memory controllers and special-purpose processors with GDDR SDRAM). However, chiplet-based architectures such as AMD's EPYC and recently announced Intel's Sapphire Rapids offer the opportunity to **c**o-design the off-chip(let) processor/memory interconnect, memory controller, and the DRAM microarchitecture [9].

This chapter describes our co-design approach, which we call Low Latency Memory (LLM). LLM simultaneously optimizes latency, bandwidth, and energy efficiency by taking advantage of silicon photonics (SiPh) interconnects with optical parallelism and wavelength routing to reduce contention in the entire path from chiplet to the DRAM subarrays. This co-optimization is now possible because silicon photonics offers lower energy/bit [78], high bandwidth density ($Gb/s/mm^2$) with wavelength division multiplexing (WDM) [65], and all-to-all interconnectivity with chip-scale AWGRs (Arrayed Waveguide Grating Routers) [79].

5.2. Motivation

The primary source of performance degradation for irregular applications is contention among shared resources [32]. Figure 5.1a shows the high-level schematic of a generic chiplet-based architecture such as AMD EPYC [9]. There are four major components in this system: the interconnect fabric between each chiplet and the memory controllers, usually a complex crossbar-like structure with high bisection bandwidth; the memory controller, which consists of queues to buffer read/write requests bound for the particular memory channel; and finally the DRAM device, which consists of multiple banks, with each bank itself made up of subarray of cells. It is important to note that the interconnect fabric, the queues inside the memory controllers, data buses within the channel, global sense amplifiers, and global bitlines within the DRAM devices are *shared*, which introduces the potential for contention and additional latency due to arbitration, buffering, and serialization (time multiplexed sharing).

Figure 5.1b, shows the simulation results of end-to-end latency by adding parallelism only at the DRAM microarchitecture. Here we used eight random traffic generators connected to 4-Hi stack

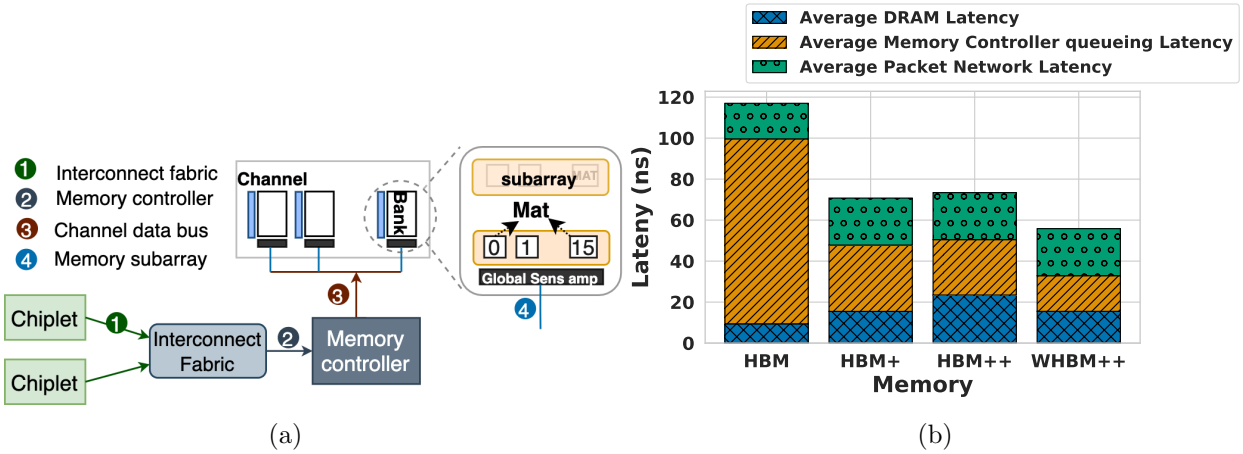


FIGURE 5.1. (a) Generic High-level Architecture of the Memory Subsystem. (b) Breakdown of end-to-end latency. HBM+ increases the pseudo-channels, HBM++ reduces the size of each bank, and WHBM++ increases the data bus width compared to HBM++.

TABLE 5.1. DRAM configuration

Category	HBM	HBM+	HBM++	WHBM++
Channels/stack	8	8	8	8
pseudo-channel/channel	2	4	16	8
Banks/channel	16	16	32	32
Pins/pseudo-channel	64	32	8	64
t_{BURST}	4	8	32	4

HBM2 (eight channels) in gem5 [57]. We used HBM as a baseline model of HBM2 working in the pseudo-channel mode, which divides each HBM2 channel into two pseudo-channels that share the channel's address/control (ADD/CMD) bus but have their own 64-bit wide I/O interface. Table 5.1 shows the specification of different memories. WHBM++ has an 8 × number of pins compared to HBM++ while providing the same number of banks and pseudo-channels as HBM++.

We divided the end-to-end latency into three categories: network latency, the queuing latency at the memory controller, and DRAM access latency. Figure 5.1b shows that for HBM, most of the latency is in the queuing at the memory controller. When we increase resources without considering co-design, the memory controller bottleneck is alleviated. Still, the other components (the device and the network latency) begin to dominate the total latency, and there are diminishing returns. Thus, a high-performance memory, not only needs higher parallelism to reduce the memory controller queuing latency, but it must also reduce the device and interconnect latency. In fact, we

propose to *re-architect the entire end-to-end system* to reduce the latency of the memory subsystem, specially as we scale the system to large number of compute units and run irregular workloads with poor data reuse and locality.

LLM makes the following ***contributions*** towards removing these sources of contention: **(a)** It proposes a ground up co-design of the entire path from the processor/memory interconnect to the DRAM microarchitecture. This co-design enables both bandwidth and latency improvement without sacrificing one for the other. LLM is composed of three pieces: a contention-less optical data plane, a low-bandwidth electrical control plane, and fine-grained memory banks with integrated photonics. **(b)** In the data plane (Figure 5.2a), LLM provides a dedicated data path from every requestor to every memory bank. An LLM-like architecture is impractical with electrical interconnects because of the energy costs of data movement and the wiring complexity of providing these dedicated data paths. We propose using a passive and contention-less optical interconnect for the data plane with no intermediate buffering, thus reducing the queuing and the interconnect latency compared to other chiplet-based architectures. **(c)** The control plane (shown in Figure 5.2b) communicates the address and command between chiplets and memory and coordinates the time that a chiplet sends or receives its data. A low bandwidth electrical network is used for carrying this control information. **(d)** LLM uses fine-grained memory units called μbanks that are exposed to the memory controller to exploit massive amounts of parallelism. LLM memory devices have integrated optics to allow low-latency high-bandwidth direct connections from the requestors to the memory μbanks.

5.3. Silicon photonic enabling technologies

Over the past decade, optical interconnects have shown great potentials in overcoming the bandwidth bottlenecks that limit inter-processor and memory performance [**16**, **36**, **95**]. Commercial products (e.g., Ayar Labs in collaboration with Intel) leveraging foundry-enabled (e.g. GlobalFoundries offers SiPh-CMOS fabrication) SiPh fabrics and WDM SiPh transceivers have been announced, making SiPh technology feasible for chiplet-based communications [**2**].

The first SiPh device we use in this study is a microring resonator. Microrings are compact and energy efficient, WDM-compatible devices that are designed to resonate when presented with specific individual wavelengths and remain quiescent at all other times. Active microrings are

designed to tune their resonance frequency as the amount of current in their base layer changes, enabling data modulation and demodulation. Microring modulators encode bits onto the optical medium (electrical-to-optical (EO) conversion), and microring filters extract the optical signal and send it to a photodetector performing optical-to-electrical (OE) conversion.

Earlier proposals used optical buses and large matrices of microrings (consisting of hundreds of microrings) for the memory-to-processor network [16, 27, 48]. In this proposal, we use AWGR [37, 74, 79, 87] which is a passive silicon photonic fabric with a compact layout that offers scalable all-to-all connectivity through wavelength routing. Recent advances in the fabrication process of AWGRs now enable their integration with a significantly reduced footprint (1 mm^2), crosstalk (< -38dB), and loss (< 2dB) [79]. This makes the AWGR a favorable candidate for energy-efficient, high bandwidth, all-to-all connectivity within HPC systems. Initial studies have shown AWGR to be promising choice for processor-to-memory network [36, 37]. Figure 5.2d shows the wavelength routing in a 5×5 AWGR; all wavelengths inside a waveguide entering one input port of AWGR are evenly distributed over all the output ports, each to a unique output port.

A Vertical Optical Interconnect (VOI) is an optical waveguide that can potentially replace through-silicon vias (TSVs) in 3D stacked memories. Unlike previously demonstrated optical TSVs [71], VOIs have 1-2 μm pitch size [105] and they can provide higher bandwidth density compared to state-of-the-art TSVs (20 μm pitch size [70]).

5.4. Architecture

In this section we present the detailed design and implementation of LLM that harnesses the benefits of silicon photonics to reduce contention in the entire memory subsystem from the requestor (chiplet or group of chiplets) to the fine grain access units called μbanks inside the DRAM.

5.4.1. Processor-Memory Interconnect. LLM reduces contention by taking advantage of the lower energy consumption and the higher bandwidth density of optical interconnects for data communication. In addition, it uses a low bandwidth all-to-all electrical interconnect to manage bank conflicts and orchestrate the data movement.

Figure 5.2a shows the *optical data plane* with an AWGR provideing an all-to-all connection. On the memory-side, each channel is connected to a port of the AWGR using a waveguide.

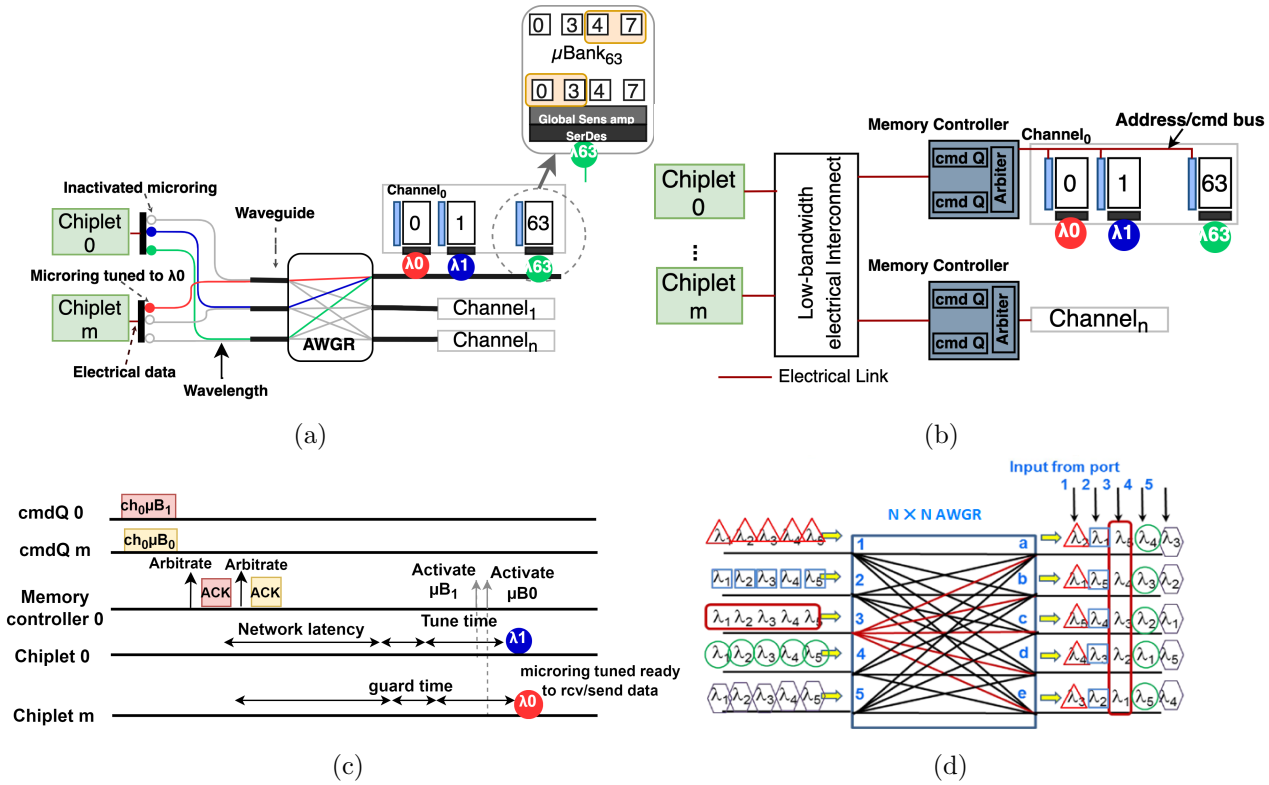


FIGURE 5.2. High-level Overview of (a) data plane, and (b) control plane, (c) demonstrates an example of routing scheme in LLM, and (d) shows the wavelength routing property illustration of AWGR.

Each waveguide carries a wavelength for each μbank. Inside the memory channel, μbanks modulate/demodulate data on the waveguide through a tuned microring which is tuned to a specific wavelength. To enable simultaneous reads/writes per channel we can assign two waveguides per channel to connect to two separate AWGRs (one for carrying read and another for write data).

While the AWGR can route the optical signal to the destination μbank, the requestors should modulate the data on the intended wavelength and send it to the correct AWGR port. Thus, each chiplet uses an array of tunable microrings where each microring in the array directly connects to a different input ports of the AWGR to send/receive the data. For an $n \times n$ AWGR, each chiplet requires n microrings.

The request's μbank address indicates the wavelength, and its channel address indicates which microring on which waveguide needs to be tuned to the corresponding wavelength. This configuration allows (a) single requestor to send requests to every bank within a single channel using a different wavelength on each of the waveguides connected to different input ports of the AWGR;

(b) at a particular time, all the requestors can send requests to different channels using different wavelengths on a single waveguide connected to a single port of the AWGR; (c) at a particular time any combination of the above could occur. Note that the *only possible contentions are bank conflicts*, which cannot proceed in parallel anyway and are stalled at the memory controllers.

The choice of the number of waveguides, the number of wavelengths per waveguide, and the data rate in the waveguide are *design parameters* which dictate the maximum number of requestors, memory channels, μbanks, and μbank bandwidth. An $n \times n$ AWGR interconnects n memory channel and n requestors (or group of requestors) each connected to n microrings using n wavelengths. The scalability of the system depends on the scalability of AWGR. The number of ports in an AWGR can easily scale up to 64 ports [**26**]. For larger systems, multiple smaller AWGRs (lower port count) can be used in parallel to provide the all-to-all interconnection as a large AWGR [**74**].
Due to the small size of control packets, an electrical interconnect can provide sufficient bandwidth for the communication of command and address bits. Therefore, LLM takes advantage of an electrical interconnect for the implementation of the *control plane*.

Figure 5.2c illustrates an example of our proposed routing scheme, where multiple chiplets are performing write operations. When request 1 from chiplet 0 wins the arbitration in the memory controller (Explained in Section 5.4.2), the memory controller sends an acknowledgment signal to chiplet 0, allowing it to send data to the memory. Chiplet 0 uses the second ring and tunes it to the wavelength of its destination (in this example μbank 1 is the destination, which operates with blue wavelength). At the same time, chiplet m can use the red wavelength on a different waveguide connected to another port on the AWGR to reach the μbank 0 in the same channel. After issuing a request to the DRAM, data will be ready in the memory at a predefined time later (which is related to the memory access latency). The requestor uses this latency to tune the correct microring (the channel and μbank address indicate which microring must be tuned to which wavelength). Therefore, the memory device needs to have a deterministic response time. Hence, LLM uses a closed-page policy, where the DRAM row buffer is closed immediately after every read or write.

5.4.2. Memory Controller. LLM redesigns the memory controller to accomplish three main tasks- (i) issuing request at a high rate to increase throughput, (ii) manage arbitration in case of

bank conflicts, and (iii) coordinate between requests and data signals (control flow scheme to enable processors to tune the microrings at a particular time).

To improve throughput, we propose reducing the head-of-line-blocking in memory controllers. In a standard memory controller, a bursty sender can overload the entire queue in the memory controller, forcing other processing units to stall. To avoid this, we assigned a single entry queue per requestor (a single or group of processing units) as shown in Figure 5.3a. These single-entry queues only store the electrical command signals and the data is buffered at the requestor. Then, instead of requiring a complex priority queue (e.g., first-ready first-come-first-serve), we use a round robin arbiter to select an available request from one queue to a free memory μbank.

To maintain consistency between data and control signal, the memory controller must let the requestors know when to tune their microrings. On an LLC miss or write-back, the requestor sends a request to the memory controller. Then, every cycle, the arbiter selects a ready request from one of the command queues. For read requests, the memory controller asserts the appropriate command and address on the electrical command bus (shown in Figure 5.3a in red). At the same time, the arbiter sends a notification back to the requestor to inform the requestor when the data will appear on the dedicated data bus for that μbank, allowing the requestor to tune its microring to an specific wavelength. We use electro-optically tunable microrings with few-nanosecond tuning speed [**31**,**60**]. The requestor can tune its microring while memory is activating the corresponding row in the memory. The microring at the requestor needs to be tuned to the corresponding wavelength once the memory row is activated. To ensure this, memory controller delays the activation request by guard time of 10 ns.

5.4.3. Memory Microarchitecture. For irregular workloads, bank conflicts could cause long latency due to their random memory access pattern. Bank conflicts happen when multiple consecutive requests target different rows in the same bank. The impact of bank conflicts on latency is quite high. For instance, in HBM2 this latency is approximately 50 ns (precharge latency plus activation latency) [**6**].

LLM reduces the probability of bank conflicts by dividing HBM banks into smaller μbanks. In both HBM and LLM, groups of DRAM cells are combined into "mats" which are planar 2D arrays of 512×512 DRAM cells. Mats inside of a subarray are connected to a local sense amplifier and a global bitline connects local sense amplifiers to a global sense amplifier. In LLM μbanks, both the

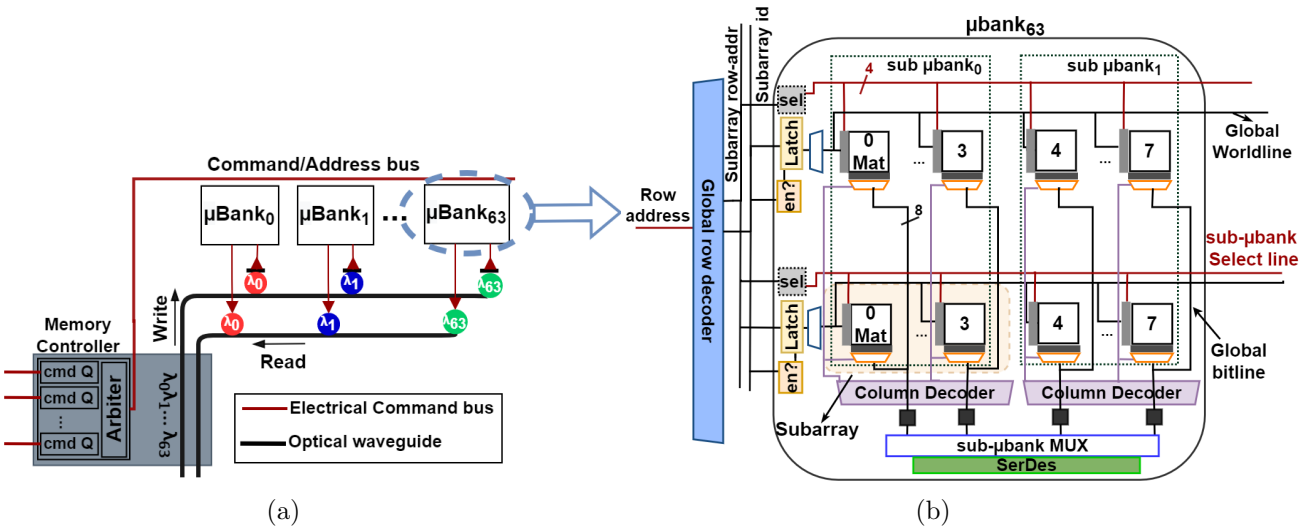


FIGURE 5.3. (a) LLM channel organization. Data and commands are communicated through optical waveguide and electrical bus respectively. (b) μbank architecture which is divided into two sub-μbank that share the same optical data bus through a multiplexer. Each μbank is connected to a microring which is tuned to a certain wavelength.

number and size of subarrays are 2× smaller than HBM banks. Lower number of subarrays in LLM μbanks results in shorter global bitlines compared to HBM since each μbank is physically smaller than the HBM banks. LLM further reduces the size of the row buffer by splitting each μbank into two sub-μbanks. This design further reduces the activation energy in LLM which allows for more parallel accesses. Figure 5.3b shows the detailed architecture of μbank. The impact of our design decisions on the DRAM die size is discussed in Section 5.5.

In addition to the increased parallelism, this new bank organization also reduces the activation energy. A series of studies have shown that the activation row size directly impacts the DRAM activation energy [**28**, **39**, **69**, **103**]. Dividing the HBM banks into μbanks and sub-μbanks, reduces the activation row size and the activation energy by 75% compared to HBM2.

The second source of contention is the data bus shared by multiple banks inside of one channel. To remove this contention requests targeting different banks need to be t_{BURST} apart. LLM removes the contention on the shared data bus inside the channels by assigning a dedicated optical wavelength to each μbank. Each μbank uses a SerDes and a tuned microring to communicate data.

These microarchitectural changes in DRAM also affect the timing constraint of the memory system. t_{CAS} or t_{CL} defines the time between the column command and the appearance of the

data at the memory interface I/O. This makes t_{CAS} the data movement latency within the memory die, which consists of pre-GSA (global sense amplifier) and post post-GSA latency. Reducing the length of the global bitline (2× smaller), lowers the capacitance which reduces the pre-GSA t_{CAS} by 2×. Post-GSA t_{CAS} also will be 1 ns [37, 97] since the banks send data to the I/O through optical wavelengths. Note that the E-O and O-E latency is discussed in Section 5.5.

t_{FAW} limits the activation rate in DRAM to limit the drawn current. Since LLM reduces the number of activated bits by 4×, it can activate 4× more rows compared to HBM2. In HBM2, t_{FAW} is 12 ns. If the command bus works at a high frequency of 2 GHz, memory controller can issue the maximum of 24 activations which is still lower than the limitations of t_{FAW} in LLM (32 activations). Therefore, the parallelism in LLM channels is not limited by the power delivery constraints.

t_{BURST} is the time to transfer the data for a single DRAM request on the I/O bus. With 32 Gb/s data bus bandwidth and 64 byte data, the t_{BURST} in LLM is 16 ns. However, since each μbank in LLM has a dedicated data bus increasing t_{BURST} does not affect the requests targeting different μbanks in one channel. In a system with a shared data bus, the long t_{BURST} increases the serialization effect, enforcing all requests going to different banks in each channel to be t_{BURST} apart. The dedicated data bus eliminates the bus contention in LLM.

5.4.4. LLM Organization and packaging. LLM dies can be organized as both 3D stacks (similar to HBMs) or non-stacked DRAMs (similar to GDDR memories). In this study, we assume that the LLM dies are organized in 3D stacks to offer increased capacity and bandwidth. To this end, we propose using the innovatively new enabling technology called Vertical Optical Interconnects (VOIs) [105] to replace the TSVs. These optical vias allow substantially higher bandwidth and scaling with number of channels, while keeping the area and number of I/O pins the same. In 3D stacked LLM, data can be moved between μbanks in different layers vertically through optical links. Thus VOIs can replace most of the electrical copper TSVs. Werner et al. explored the bandwidth and scalability advantages of VOIs in 3D stacked memories [96].

Figure 5.4 presents an overview of the packaging approach we use in our design. We place memory stacks, AWGR, and compute cores on the same package substrate and use a previously proposed technique for intra-package communication [36, 92]. This approach uses dedicated processor node chiplets, and memory node chiplets with embedded SiPh transceivers. For instance

the processor node chiplet consists of SerDes, SiPh transceivers, and the compute core dies. The dedicated SiPh transceivers are connected to the chiplets through Si bridges (which are ideal for short-distance electrical interconnection) and optically to AWGR through polymer waveguides. The memory node has SiPh transceivers embedded inside and can use polymer waveguides to connect to AWGR. The polymer waveguides are integrated on top of the organic package substrate and provide connectivity to AWGR. SiPh is ideal for long-distance, inter-package communication, enabling this system to scale out to multiple packages. The multipackage system uses a polymer waveguide for interconnecting separate packages for computing cores, AWGR, and memory stacks without performance and energy degradation.

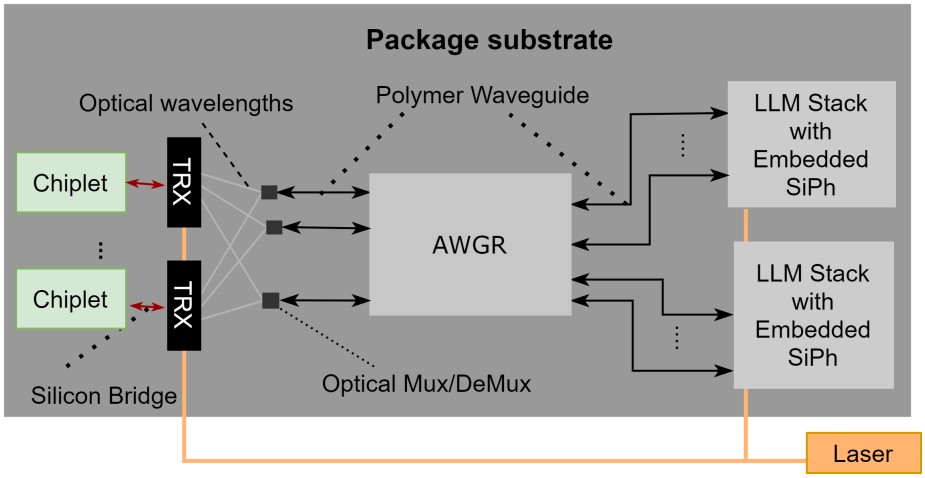


FIGURE 5.4. The package layout for a computing system using the proposed LLM memory design. SiPh transceiver chiplets are connected to the compute core dies through Si bridges, memory stacks already have embedded SiPh transceivers and can directly connect to a polymer waveguide.

5.5. Methodology

To evaluate the performance and latency of LLM, we used the gem5 simulator version 21.0 [57] with both synthetic workloads and full-system (with Linux kernel version 5.2.3). We modeled the network interconnect with Garnet3.0 and the cache hierarchy using Ruby to evaluate the system architecture.

We compared our design with high bandwidth memory systems such as HBM2. In addition, we used two state-of-the-art memory systems with more memory level parallelism. The first one is HBM2, with added subarray level parallelism for lower memory access latency. We augmented

HBM2 by adding techniques from Kim et al. [47]. Throughout the paper, we refer to this as HBMSALP. The second one is a highly concurrent memory system with 4× higher bandwidth than HBM2. In this architecture, the memory banks are finer and more independent. A narrow electrical bus with 4× higher datarate compared to HBM2 is assigned to these fine-grain memory banks. This design is our interpretation of Fine-Grained DRAM, and we refer to it as FGDRAM [69]. FGDRAM shows the benefits of incorporating μbanks without the contention-less optical data plane.

To be able to fully stress the bandwidth, we used *synthetic traffic* with different access patterns both with high and low locality. We used three different traffic patterns: *Stream*, *Random*, and *GUPS*. The Stream and Random traffic create a sequence of requests with linearly increasing and uniform random distributed addresses respectively. They both generate requests at user-specified frequencies. GUPS is a data dependent application [58] with a random distribution over memory addresses.

Using traffic generators is a *processor architecture agnostic* evaluation allowing these results to be portable whether LLM is used in a CPU, GPU, or accelerator platforms. Using traffic generators also enables experiments with different network injection rates to model memory intensive workloads that can fully stress the high bandwidth of our proposed memory system.

For the synthetic traffic simulation, we used 32 traffic generators. For this experiment, we scaled our high bandwidth baseline memories to reach the same peak bandwidth as LLM stack which is 4 TB/s (iso-bandwidth). In these iso-bandwidth experiments, both HBM and HBMSALP are given 8× the channels of LLM and FGDRAM 2× compared to LLM.

For latency and overall evaluation, we ran real workloads in the gem5 simulator. We used applications such as GAP benchmark suite (GAPBS) [17] as a representative for irregular workloads due to their random memory access pattern. Table 5.2 shows the system configuration. We used a multiple core CPU system, each with two levels of cache hierarchy.

5.5.1. Latency Parameters. The memory system needs to model both the network latency (which also includes the O-E and E-O and SerDes latencies) and the DRAM timing constraints. Both of these timings are included in our simulation platform. Due to the different bank and channel organizations, some timing constraints are different from LLM and HBM2. Table 5.2 illustrates the changed timing constraints between HBM, FGDRAM, and LLM. We assumed an optical traversal of 1 ns [37,52]. We are using a low-power 16 Gb/s SerDes for seriliazing/deserializing 32 bits of data

TABLE 5.2. Full System Simulation Parameters

Parameter	Description		
CPU	16 cores ; x86 @ 4GHz		
Caches	private 32 kB L1I/D, 2/8-way per core private 512 kB, 8-way L2 per core directory coherence		
Memory	8 DRAM channels		
Timing parameter (ns)	HBM2 [6]	FGDRAM [69]	LLM
tCAS	16	16	5
tBURST	4	16	16
tFAW	12	12	12
activates in tFAW	8	32	32

TABLE 5.3. Silicon Photonic device parameters

Parameter	Value	Parameter	value	Parameter	value
VOI loss	1.3 dB	Photodetector loss	0.1 dB	Modulator Insertion loss	1 dB
Waveguide loss	0.5 dB/cm	Filter through loss	0.1 dB	Power Margin	3 dB
Filter drop loss	1.5 dB	Receiver Sensitivity	-17 dBm	Laser efficiency	14 %
Coupler: Fiber-to-Package	3 dB	AWGR crosstalk	-20 dB	AWGR loss	1.8 dB

from global sense amplifiers, resulting in 2 ns latency. We assume that the E-O, O-E conversion latency takes 35 ns [31, 60]. We also modeled the electrical control plane in LLM with a network latency of 20 ns, which is a conservative assumption in our system.

5.5.2. Power Model. For the power modeling of the optical interconnects, we used values for 65 nm CMOS [52, 101] and scaled it down to 28 nm using SPICE models [52, 101]. The laser efficiency is based on commercially-available comb lasers [7]. Table 5.3 illustrates the details of our silicon photonic devices.

5.5.3. Area. We compared the area of LLM stack based on both microarchitectural changes and the optical circuitry we have added to the memory microarchitecture design. We compared the area for a 4 die stack (4Hi) LLM and HBM. The dimensions of HBM dies are typically 5.5 mm $\times$ 7.7 mm [53].

Each μbank includes SiPh transmitter and receiver circuitry (5 μm pitch size), and a 16 Gb/s serializer-deserializer (SerDes) with an area of 0.0045 mm^2 (estimated using TSMC 28 nm CMOS process). Two waveguides are connected to each memory channel, each with 2 μm pitch size [105]. A 4Hi HBM requires 1024 TSVs for data but LLM requires only 32 VOIs. Overall, optical circuitry adds 4.94% area overhead compared to a HBM stack.

LLM also requires 2$\times$ more column decoders and 4$\times$ more global sense amplifiers. Dividing each μbanks to sub-μbanks adds additional circuitry such as 4 bit wordline-select, and sub-μbank

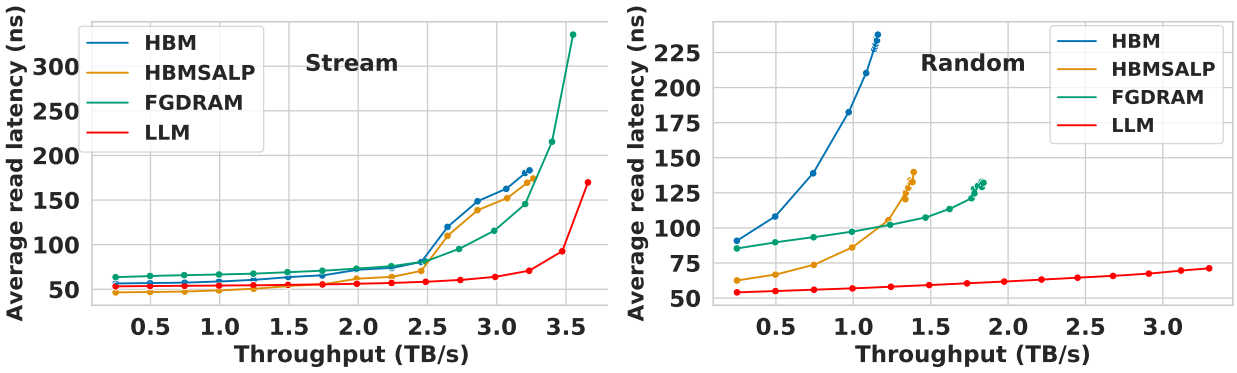
multiplexer. These area overheads are equal to FGDRAM and subchannel [25,69] which are 4.67%. LLM also requires latches to enable subarray-level parallelism. Each latch requires 2 μm^2 area. In total microarchitectural changes to DRAM adds an additional 4.8% area overhead. A 4 stack-high LLM requires 9.74% area overhead compared to HBM2.

5.6. Evaluation

5.6.1. Synthetic Traffic Evaluation. In the first experiment, we ran stream and random synthetic traffic with different traffic rates to see how latency and throughput change as we increase the traffic rate. Figure 5.5 shows both the achieved throughput and the average access latency for read-only memory requests under varying injection rates. With stream traffic, all memories can achieve high throughput. However, under high injection rates, LLM has lower latency than the other designs due to its low latency interconnect and zero data queuing at the memory controller. At very low injection rates, HBMSALP has a lower average latency due to increased page hit rate and the SALP optimizations [47]. Since LLM uses closed-page policy for applications with high locality LLM will not show significant reduction in latency compered to HBMSALP. However, at all injection rates LLM has lower latency than FGDRAM and HBM.

For random traffic, Figure 5.5b shows that LLM has much lower latency for all injection rates. The main reason HBM's latency increases even under a relatively low injection rate is due to DRAM row buffer misses which incur high latency. These row buffer misses cause contention in the memory controller which results in a high queuing delay. For LLM, reducing the size of the queues in the memory controller and using a closed-page policy leads to low latency under high injection rates. This low queuing is unlike HBM and FGDRAM which experience significant increase in latency as the traffic rate increases. Figure 5.5b shows the biggest difference between LLM and prior technologies. *LLM can achieve nearly the same throughput with random traffic as with streaming traffic.* In contrast, the best other technology, FGDRAM, can only achieve approximately 50% of its peak theoretical bandwidth under a random access pattern. The difference between LLM and FGDRAM, also shows that simply adding parallelism in the memory subsystem (μbanking) without re-architecting the entire datapath will not remove the contention in the system; it will simply move the contention to another point in the datapath.

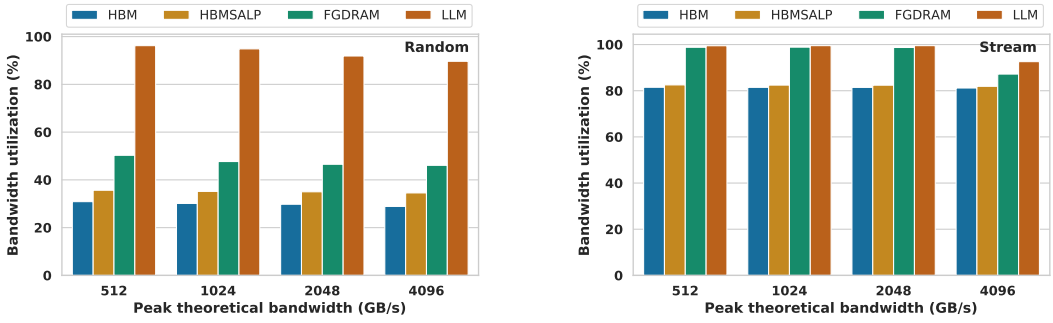
To increase complexity in our synthetic traffic experiments, we applied the Giga Updated Per Second (GUPS) benchmark which has data dependent accesses. We measured the performance of these systems based on the GUPS as defined by the benchmark. Similar to Random and stream we used iso-bandwidth test for GUPS. Figure 5.7b show that even when given significantly more I/O (and cost) HBM and FGDRAM cannot match LLM's performance for this irregular workload.



(a) (b)

FIGURE 5.5. Iso-bandwidth test with (a) Stream, (b) Random. Comparing the average read latency and throughput for different injection rates and access patterns.

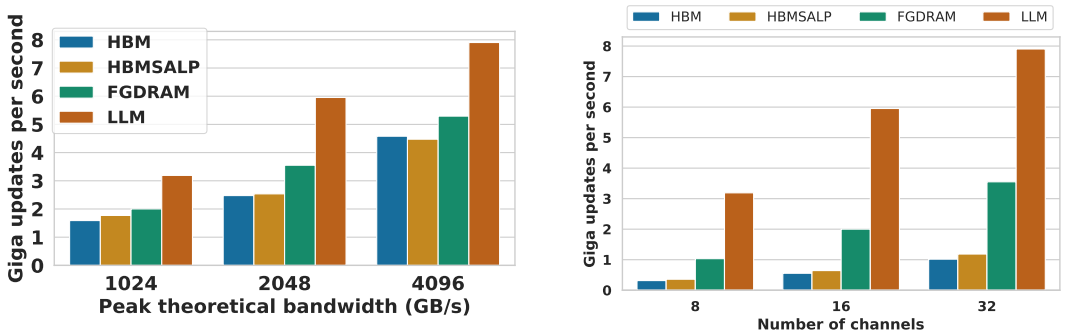
Although HBMSALP adds more intra-bank parallelism compared to HBM, Figure 5.7b shows it does not achieve considerable performance improvements. This result demonstrates the importance of optimizing the memory system for both bandwidth and latency. Even for latency-critical workloads like GUPS, the bandwidth can also be the limiting factor. Only optimizing for latency does not necessarily lead to the best performance.



(a) Random traffic (b) Linear traffic

FIGURE 5.6. The iso-bandwidth experiment: Comparing bandwidth utilization of three memories. In random traffic LLM achieves in average 1.96× better bandwidth utilization compare to FGDRAM. LLM achieves high bandwidth utilization for *stream* traffic because multiple μbanks can be activated at the same time.

Figure 5.6a shows the performance of these memory technologies with a random traffic pattern. This figure shows that HBM, HBM with SALP and FGDRAM have poor utilization with a random traffic pattern. These technologies have poor utilization because they depend on high page hit rates to saturate the theoretical bandwidth. FGDRAM does the best of these prior memory technologies since it has the smallest row buffer size and the most internal parallelism.



(a) (b)

FIGURE 5.7. (a) GUPS traffic, shows even with the same peak bandwidth LLM provides more parallelism resulting in 2× improvement on average performance compared to HBM (with 8× more channels). (b)For the same number of channels (same capacity), LLM provides higher peak bandwidth compared to HBM, HBM with SALP, and FGDRAM. The higher peak bandwidth allows for a higher number of updates and therefore better performance.

LLM shows nearly full utilization even under a random traffic pattern reaching over 90% of the peak theoretical bandwidth. The main reason LLM has a high utilization under this traffic pattern is due to the closed page policy, increased internal parallelism, and inherently low (near-zero) contention. Under random traffic LLM outperforms HBM by 23.23× and FGDRAM by 3.85× when using 32 channels.

Figure 5.6b shows the performance for the Stream workload (linear/sequential accesses). Under this workload, HBM and HBM with SALP perform much better (around 80% utilization). However, FGDRAM and LLM are able to achieve almost 100% bandwidth utilization since they have more internal parallelism and more dedicated data links (grains in FGDRAM and dedicated links per μbank in LLM).

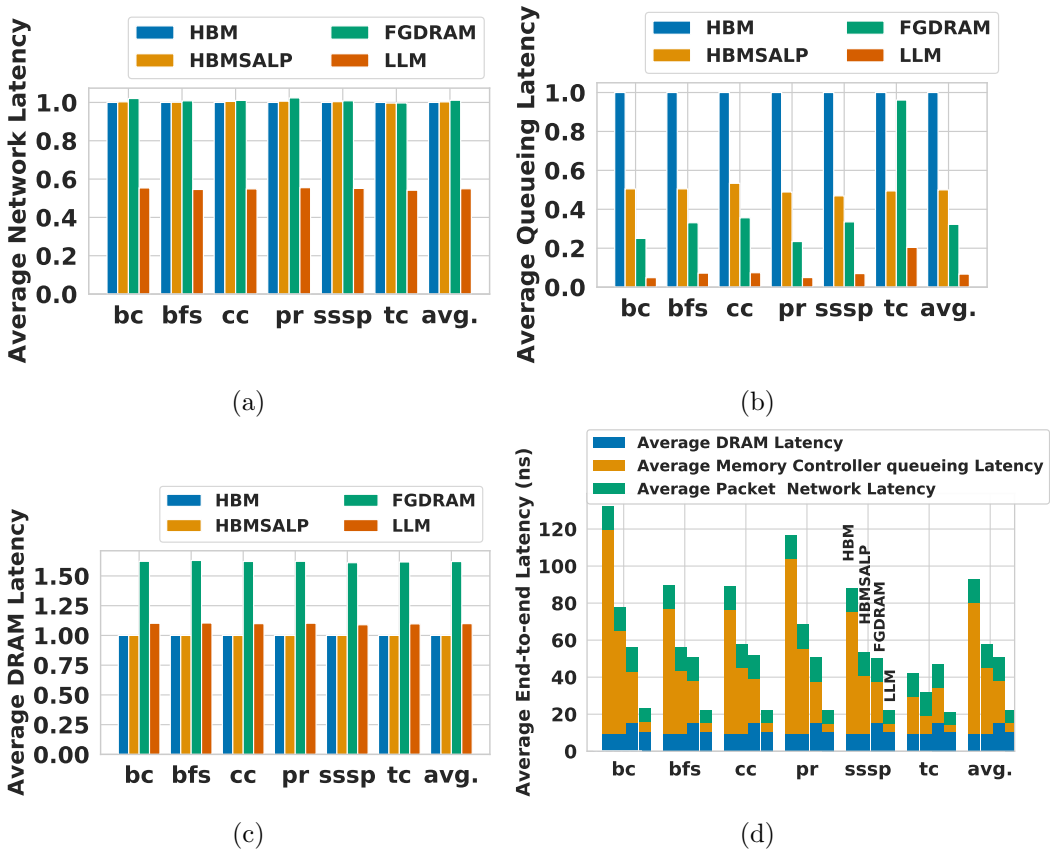


FIGURE 5.8. Average latency normalized to HBM2 for (a) network (b) queuing (c) memory device, and (d) shows the average end-to-end latency. (a) shows LLM achieves in average 2× lower network latency, 1.1× higher DRAM latency due to the long bus latency, and (b) indicates 10× lower queuing latency compared to HBM2.

5.6.2. Irregular Workloads. In a more realistic setup, we used gem5 21.0 full system mode to compare LLM with, HBM, HBMSALP, and FGDRAM in a system with 8 processing cores and 8 memory channels (iso-capacity configuration of different memory technologies) as opposed to the iso-bandwidth tests used in the synthetic traffic experiments. Though it is difficult for us to estimate the costs of each technology, this iso-capacity experiment compares the performance in a real system setting with each technology given approximately the same amount of resources. Due to the extensive time of simulation for each system configuration, we created traces for 8 core systems and extended it to 16 core configurations. This enabled us to stress the bandwidth of the system under the same traffic pattern. We used 64 × 64 AWGRs with 64 wavelengths.

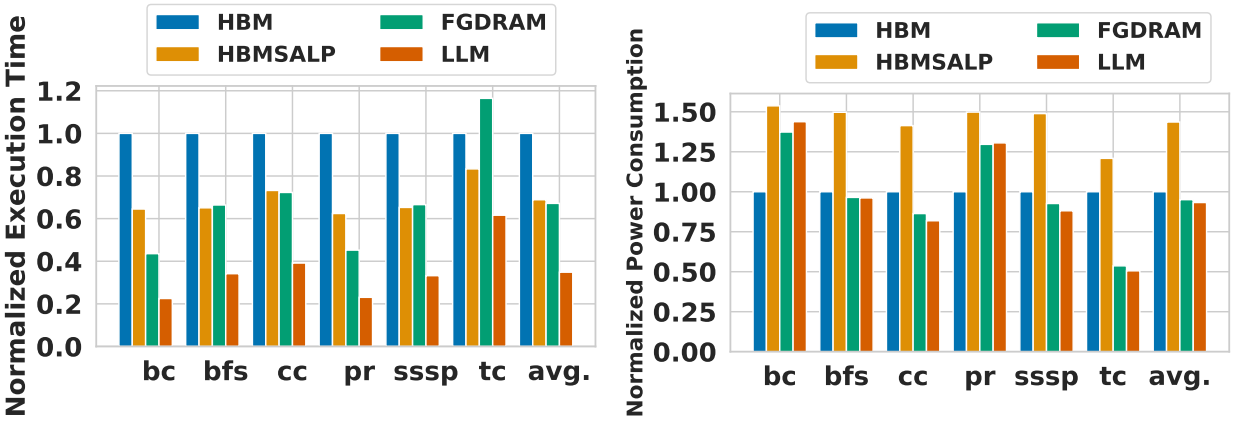
For the first experiment, we compared the average latency for DRAM access, the queuing latency at the memory controller, and the average network latency. Figures 5.9(a–c) show the normalized

comparison between these memory systems. For all workloads, LLM has significantly lower queuing at the memory controller which is what we expected based on lack of data queuing at the memory controller. Also, the network latency for LLM remains smaller for all workloads because in large-scale systems with higher crossbar radix electrical interconnect latency is higher. Compared to HBM, FGDRAM shows lower queuing latency which indicates the benefits of added parallelism at the memory microarchitecture without the optical datapath. Comparing LLM and FGDRAM, the queuing latency is on average 3× lower which shows the benefit of the co-design architecture of the memory controller, the interconnect design, and the all-optical data path. Finally, for the device latency (Figure 5.8c), all systems have approximately the same latency except FGDRAM which is higher due to the larger t_{BURST}.

Figure 5.8d shows the total average latency of the three components (device, queuing, and network latency). This shows that for all systems except LLM, queuing latency is the dominant portion of the time (broken out in Figure 5.8b). Figure 5.8d indicates the memory intensity of the workloads as well. For instance, *tc* has lower average end-to-end latency with lower queuing compared to the other workloads. Thus, optimizing just for throughput will not improve the execution time for this workload (e.g., FGDRAM does not improve performance for *tc* as shown in Figure 5.9a since it sacrifices latency for bandwidth).

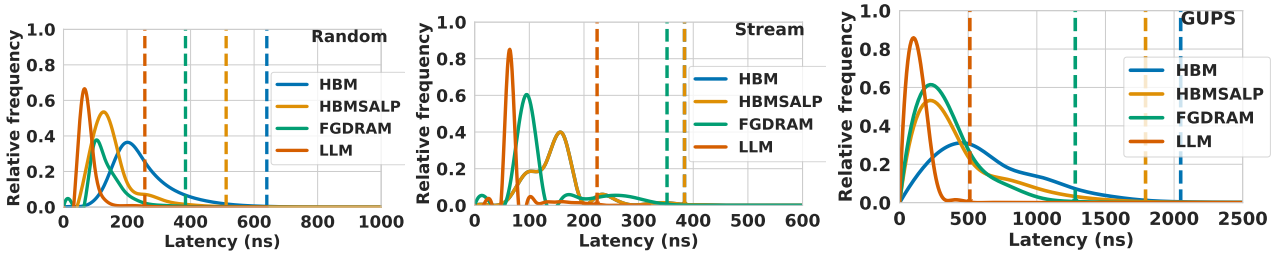
Figure 5.9a compares the execution time of GAPBS workloads for HBM, HBMSALP, FGDRAM, and LLM. Compared to HBM, LLM provides 3× reduction on average execution time. For the more memory-intensive workloads, the increased bandwidth of LLM provides reduced execution time. Importantly, for the lower-intensity workloads, LLM also provides an improvement over the other technologies (most notably FGDRAM running *tc*) due to its lower contention on the shared data bus.

5.6.3. Energy and power analysis. The DRAM access energy consists of activation energy, data movement energy, and I/O energy. We used the HBM2 energy values from O'Conner et al. [**69**]. The activation energy directly depends on the number of bits in a row that get activated. Similar to FGDRAM [**69**], LLM reduces the size of the row by a factor of 4×, and therefore, we reduce the activation energy to 227 pJ for LLM from 909 pJ in HBM 2.0. Pre-GSA energy is the energy of moving data from local and master bitlines to the global row buffer, and it depends on



(a) (b)

FIGURE 5.9. Execution time (a) and power consumption (b) normalized based on HBM2. LLM provides in average 3× lower execution time while maintaining same power consumption compared to HBM2.



(a) (b) (c)

FIGURE 5.10. The latency distribution for different memory systems under 3 types of synthetic traffic: (a) Random, (b) Stream, and (c) GUPS. LLM has a lower 95$^{\text{th}}$ percentile (shown as dashed lines) and therefore has lower latency variation. In (b) HBM and HBMSALP have the same distribution of latency.

the length of bitline. Since we are reducing the size of the global bitlines, this energy will also be reduced to 0.755 pJ/bit from 1.51 pJ/bit in HBM2.

LLM uses optical links to move data between μbanks and processing cores. Therefore, both I/O and post-global sense amplifier energy values are equal and are independent of laser, SerDes, and modulation circuitry. For this SiPh stack, we used the parameters shown in Table 5.3 to match realistic current technologies. We found the total I/O energy (including laser, SerDes, modulation circuitry) to be 760 fJ/bit. In comparison, for conventional DRAM the I/O requires 800 fJ/bit [**69**], which is expected to increase as the height of DRAM stacks increases. Figure 5.9b illustrates a comparison of overall memory power consumption normalized to HBM between a DRAM stack

interconnected electrically with TSVs against LLM with SiPh DRAM stacks. As shown, the LLM is approximately the same power as the electrically implemented FGDRAM showing the SiPh implementation is feasible to integrate in a chiplet-based package. In some cases, the power is higher, mostly due to the higher bandwidth that FGDRAM and LLM enable compared to HBM.

5.6.4. Latency Variation. Finally, we analyzed the latency variation in each memory system. In current systems, the main cause of latency variation in the system is queuing. Thus, one of the byproducts of our low contention memory system should be lower latency variation. Figure 5.10 shows the distribution of access times for each technology under stream, random, and GUPS synthetic traffics using 16 memory channels. This figure also shows the 95^{th} percentile latency with dashed vertical bars.

Figure 5.10 shows that LLM achieves significantly lower and more predictable latency compared to other technologies. In general HBM has the broadest distribution, with FGDRAM and HBMSALP having slightly less variation than HBM for Random and GUPS traffics. On average LLM has 3× lower 95^{th} percentile latency compared to HBM which can be translated into 3× lower memory latency variations. We see similar results for the full system graph workloads as well.

5.7. Related Work

Several studies have shown the benefits of using photonics to increase bandwidth and reduce data movement energy for processor/memory communication [**16**, **19**, **77**, **80**, **89**, **96**]. Although these studies reduce contention at the interconnect, they did not contribute to increasing memory performance at the microarchitectural level. LLM extends these prior works by (a) reducing in-memory activation and data movement energy, allowing for higher parallelism, and (b) integrating optics inside of the memory channel and co-designing the memory controller to facilitate both bandwidth and *latency* improvements.

Previous work on DRAM energy [**28**, **39**, **69**, **103**] showed the benefits of reducing activation energy while maintaining a higher bandwidth than HBM2. These studies are still bounded by the processor/memory data movement energy. LLM extends these prior works by exploiting silicon photonic interconnects. Optical links do not suffer from the distance/bandwidth trade-off that impacts electrical interconnects. This allows LLM to achieve a low energy data movement in a chiplet based architecture while achieving higher peak bandwidth than the previous studies.

Creating smaller channels with narrower data bus and higher datarate is the technique used both in in the industry (with HBM2 and HBM2 pseudo-channel mode, and GDDR) and research [69] to enable high throughput memory systems. However, they do not consider optimizing the memory for latency. Furthermore, they use deep queues for bandwidth improvements which will result in higher latency. In contrast, LLM is a redesign of the complete memory subsystem. Decoupling data and control signals in the LLM allows for bandwidth and latency improvement at the same time.

Previous work, has explored many different avenues for decreasing DRAM latency including changing the DRAM controller [22], segmenting and shortening bitlines [47] and caching and paging policies [45]. Although these techniques proved to be effective in reducing the DRAM access latency, they are not optimized for irregular applications and in some cases can increase memory access latency variability. Wang et al. improved latency for irregular workloads by creating a low-cost DRAM substrate that enables data relocation [86]. Although effective for irregular workloads they have not shown any benefits for applications with high locality and the effects on memory latency variations. LLM reduces the amount of data queuing on the entire path and assigns a dedicated data path between each requestor and memory μbank. This technique reduces latency in both regular and irregular workloads but it also reduces memory access variability due to low queuing on the path.

Table 5.4 shows the comparison between LLM to FGDRAM, SALP, FGDRAM, and optically interconnected 3D stack memory and how these memory systems impact bandwidth, latency, and energy. Compared to these state-to-art memory systems, LLM can achieve high bandwidth, low latency, and low energy consumption. This is due to using optical link and WDM, and reducing contention at the memory bank level, and the memory controller.

5.8. Conclusion

In this paper, we investigated a new memory system that is optimized for applications with both regular and irregular access patterns with poor spatial locality. LLM introduces lower execution time compared to the baseline HBM2 systems. It also utilizes an all-optical data communication fabric that provides a direct contention-free data link between processing cores and memory banks. The use of optical interconnects, optical links, and the new memory microarchitecture improve

TABLE 5.4. Comparison between LLM (ours) and other DRAM architecture designs that target increasing bandwidth, reducing latency, and/or reducing energy. While most prior work trades off one or two of these characteristics for another, LLM increases bandwidth, reduces latency, and reduces energy.

	Bandwidth	Latency	Energy
FGDRAM [69]	+ More channels (grains) for more inter-bank parallelism (4×) Higher peak bandwidth - Bandwidth is limited by the number and datarate of TSVs	- Similar to HBM it sacrifices latency for higher bandwidth	+ Lower row activation energy: smaller row buffer size
SALP [48]	No change in increasing peak memory bandwidth of memory	+ Enabling the possibility of having multiple rows activated within the bank. Thus, fewer bank conflicts - More latency variation	No significant change to energy
PIDRAM [19]	+ Optics provide higher bandwidth	Not focusing on the latency. The latency is is impacted by the DRAM access latency and memory controller latency.	+ Lower data movement energy due to optical links
Optically interconnected 3D stack Memory [88]	+ Can easily increase number of memory channels No improvement within the stack	The memory access latency does not change, however the data movement latency among stacks will differ	+ Lower data movement energy with optics
LLM	+ Dedicated optical bus for each μbank + More channels per stack + More inter-bank parallelism	+ Finer grain banks: Fewer bank conflicts + Less memory controller queuing delay + Less memory access variability with close-page policy and small queues	+ Smaller row buffer size + Lower data movement energy with optics

data movement, reduce activation energy, and provide higher bandwidth/mm^2. By incorporating all these methods, LLM can reduce the execution time and energy with a modest area overhead. The cost increase for optoelectronic integrated LLM would be around 30% compared to electronic-only HBM2. However, LLM achieves around 3× better execution time while maintaining the same power consumption as HBM2.

Due to low-contention data access in LLM, we believe that LLM-like designs can improve the performance of other computing systems. LLM can be a great candidate as the vertex memory in graph accelerators such as SEGA due to its low memory access latency and low contention for irregular memory access patterns.

CHAPTER 6

Conclusion and Future Works

The architectural concepts explored in this dissertation pave the way for a range of system-level designs to be investigated. This chapter summarizes the primary insights derived from this study along with limitations and proposes potential avenues for future research.

Part of our decision in the graph accelerator architecture studied in Chapter 3 is to separate the memory-processor access from the inter-processor communication. It also allocates a vertex and edge memory to each accelerator core, which we call the processing element or PE. Although this selection is effective in removing the atomic updates and coherency traffic between PEs, it can potentially result in the underutilization of memory capacity and bandwidth. A disaggregated memory system with edges and vertices stored in shared memory can be used to potentially maintain the same performance as SEGA while achieving better memory utilization.

Another path to explore is to add support for streaming graphs with the capability to add/delete vertices and edges for the graph accelerator presented in Chapter 3. Graphs in many real-world applications are inherently dynamic. For example, new users will join a social network, and users in the social network will create new relations. Users in the e-commerce platform continue interacting with new items, and new connections are established in a communication network over time.

During the execution of a workload, adding and deleting a new edge can affect the two vertices directly connected to the edge and influence other neighboring vertices. As the graph evolves, a straightforward approach is to restart the application from scratch after applying a batch of graph updates. However, if the number of vertices or edges modified in a graph is small relative to the size of the graph, the changes only modify a small subset of the graph. Therefore, much of the computation performed during reevaluation is redundant.

Lastly, in SEGA, we propose using off-the-shelf memory systems. These memory systems are accessed at cache-line size granularity to take advantage of spatial locality. However, the large memory access granularity can lead to wasted off-chip memory bandwidth in graph workloads. Specifically, we access vertex memory at a cache-line size granularity while operating on only a

portion of the data (8 bytes in our case). This inefficiency in memory access can cause significant bandwidth and energy waste. Choosing costume memory systems with fine-grain access in graph accelerators can reduce these inefficiencies.

6.1. Opportunities for Disaggregated Memories

In SEGA, we proposed dedicated HBM vertex and DDR edge memory per processing element, which has the following advantages:

- Removes multiple nodes from updating a single vertex. Therefore, we no longer need the coherency data.
- Reduces the contention on the memory system since only the processing element can access vertex and edge memory.
- Seperating the PE-memory traffic from the inter-PE traffic

Despite effectively removing atomic updates to the vertices and reducing contention on the memory system, this tight coupling between the number of processing elements (PEs) and memory channels can lead to underutilization of resources in the system.

Considering the structure and capacity of graph data, particularly when employing spatial partitioning methods, some memory channel capacities (especially edge memories) remain underused. The growth of the number of edges and vertices in real-world graphs is unrelated to each other. However, due to the tight coupling between the number of processing elements and memory channels, even a small addition to vertices necessitates an increase in the number of processing elements and. For instance, when a new vertex is added, it results in an additional processing element with an edge memory. However, even if some of the edge memory channels in the system remain underutilized, this tight coupling between the number of processing elements and memory channels persists.

Additionally, coupling capacity and bandwidth in both HBM and DDR-based memory technologies poses challenges. In an ideal large-scale graph processing system, we would provision bandwidth and capacity independently to meet performance targets for specific graph sizes. Unfortunately, this independence is not achievable with HBM/DDR-based memory technologies, resulting in significant overprovisioning.

As graph sizes continue to grow into the terascale or beyond, resource utilization problems become even more pronounced. Given recent advancements in technology supporting disaggregated memory systems, these solutions may offer a desirable approach to optimize memory usage for scalable graph processing.

Furthermore, the disaggregated memory system can also enable easier vertex and edge addition and thus facilitate the streaming graph processing.

In chapters 3 and 4, we showed that the maximum TEPS is calculated based on the memory bandwidth, which is the bottleneck in a balanced graph processing system. This means any memory bandwidth underutilization will affect the actual achieved TEPS in a given system. Figure 6.1 shows a distributed graph accelerator, running BFS and BC, our memory utilization ranges from 60% to 80%, meaning even in a best-case scenario, we have 20% less performance than we potentially could have. In data-driven graph workloads such as BFS, not all vertices are active, which causes some of the memory systems to have low bandwidth utilization (the ones with low active vertices) and the rest with overprovisioned bandwidth utilization (memories with active vertices).

The main challenges of disaggregated memory are resource allocation and deallocation between processing elements based on their bandwidth and capacity demand. Another interesting feature that disaggregated memory can provide is to unlock more parallelism by reallocating vertex assignments to different PEs based on the utilization of the PEs in the system

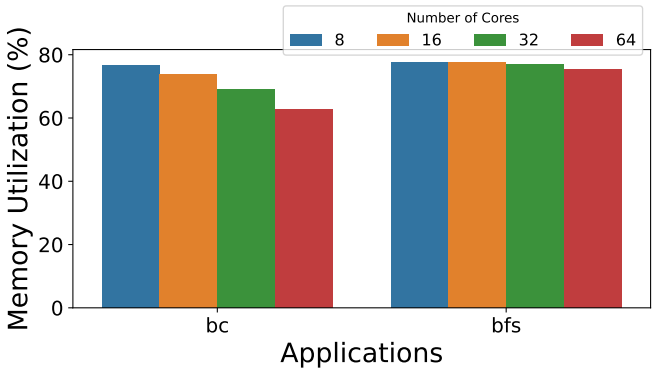


FIGURE 6.1. Bandwidth utilization of BFS with Twitter with different amounts of cores.

6.2. Support for Streaming Graphs

So far in our studies, we have focused on graph applications that ingest static data structures, where the topology of the graph remains untouched with no deletion or addition on the number of vertices and edges. SEGA, similar to other graph applications, optimizes the performance of a given application for a static graph. However, in many real-world applications, we face streaming graphs that constantly change as new attributes are created or removed or new interactions occur. A stream of updates in the form of edge/vertex additions/deletions is typically applied to the graph in batches for efficiency. For graphs with a large number of edges modified, we can restart the application from scratch after applying a batch of graph updates. However, vertices or edges modified in a batch are typically exceedingly small relative to the size of the graph. Thus, as the changes in the structure of the graph can only modify a small subset of the graph for many workloads, much of the computation performed during reevaluation is unnecessary.

Among graph accelerators, JetStream [76] is the only accelerator that supports streaming graphs. JetStream is an event-based accelerator that is built upon GraphPulse [75] and supports both the addition and deletion of edges in the graph. However, Jetstream, unlike SEGA, only supports asynchronous programming models, and since it is based on PolyGraph its performance relies on the size of on-chip memory. Therefore, it has the same scalability limitations. This new vertex

Based on the insight from JetStream, the edge addition in the graph workload is similar to adding a new message to the outgoing vertex. This new edge can only change the property of vertices that are connected to the new edges. Thus, only a subset of the large graph is affected.

Unlike edge addition, edge deletion is more complex because the contribution of the deleted edge to the previous converged state must be undone. For algorithms such as single source shortest path and page rank with accumulative updates, reversing the effect of edge deletion is easier. In such applications, we need to inverse the effect of the deleted edge by sending the inverse of its previous converged state, transformed by the propagate function, which negates the cumulative effect of all updates over this edge. Propagating the negative events from the receiver vertices leads to the rollback of all contributions from this edge and puts the graph in a recoverable state. For algorithms having selective updates such as breath-first-search and single source shortest path, it is more difficult to identify which edges contributed to a vertex. The destination vertex of a deleted

edge is reset to its initial value so that it can be updated later in the reevaluation phase. To remove the effect of deleted edges in workload with selective reduction, the system needs to identify the potentially affected vertices and efficiently reset them to their initial value. Afterward, new events are created for all the neighbors of the impacted vertices. We process the inserted edges at this point to create and queue the events for them. These new events allow the impacted vertices to set their new state using the states of their neighbors. At the end of this phase, when the queue is empty, the graph arrives at a correct state, and the process of reevaluation concludes.

6.3. State of the art Memory Systems for Graph Workloads

Current memory systems are optimized to extract locality, for instance, large memory access granularity is used to extract spatial locality. However, graph workloads have little to no locality and vertex and edge information are much smaller than memory block size. This causes inefficient use of off-chip memory bandwidth. In graph workloads, the vertex accesses are random with low locality, while edge memory accesses exhibit high spatial locality. Therefore, vertices can benefit from smaller memory access granularity, while edge memory can benefit from wider access width.

Another characteristic of these workloads is that they are memory latency and bandwidth-bound. Figure 6.2 shows the average flops/bytes for a variety of graph algorithms on the Twitter graph. Lower flops/byte translates to greater demands on external bandwidth to remote memories and remote nodes. Thus, in such systems, memory technology can play an important role in dictating the energy and performance.

The ideal memory system for graph applications is memories with low memory access granularity, high bandwidth, and low latency. Low Latency Memory, or LLM is a perfect candidate for vertex memory in graph accelerators.

The hardware buffer used in SEGA accelerates active vertex read and hides the long memory access latency to the edge memory. We can reduce the vertex memory access latency by replacing HBM with LLM as the vertex memory. By using LLM as the vertex memory, we can update vertices and read active vertices at a higher rate, and utilizing higher edge memory bandwidth results in processing elements with significantly higher throughput. We can effectively reduce data movement energy by accessing LLM at a finer granularity. Specifically, when reading data from

the vertex memory, we retrieve information about active vertices individually rather than fetching an entire block of memory containing multiple vertices.

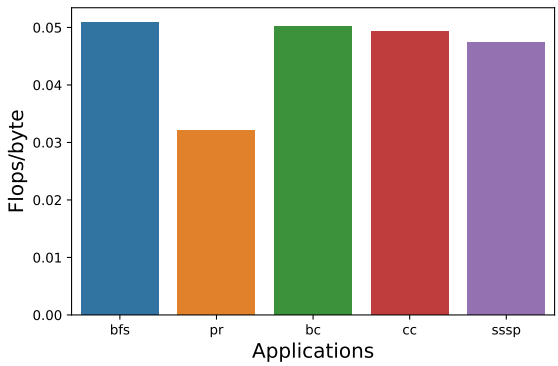


FIGURE 6.2. Compute to Memory Requirements in Graph Processing for Different Algorithms.

Due to the cost of electrical-to-optical (E-O) and optical-to-electrical (O-E) conversion, using LLM in a disaggregated memory system would be more beneficial. This is because the energy and latency of optics are independent of distance.

6.4. Conclusion

This dissertation delves into the hardware limitations of current computing systems when processing graph applications. Our focus lies in improving the performance of graph workloads, particularly for real-world graphs. Additionally, we project the capacity and bandwidth requirements of future large-scale graphs and emphasize the significance of state-of-the-art Silicon Photonics (SiPh) fabrics in scaling out graph accelerators. Furthermore, we propose a novel memory microarchitecture tailored for applications with random memory access patterns, such as graph workloads. The proposed memory system can be used to reduce the memory access latency and data movement energy consumption while providing high throughput.

Chapter 3 presents a high-level bottleneck-analysis model for designing and evaluating *scalable* and balanced accelerators for graph processing. It also shows several applications of this model, including choosing the right mix of different memory types, network topology, network bisection bandwidth, and system-level architecture to match the access patterns and capacity requirements of different data structures for a given graph and a performance target. It also proposes an architecture for a scalable graph accelerator called SEGA (Scalable Engine for Graph Acceleration).

SEGA introduces microarchitectural and system-level changes that enable scalability. SEGA is a balanced architecture that facilitates fast work generation (sending updates to neighbors) and work consumption (performing vertex reductions) at the available off-chip memory bandwidth, with a small on-chip SRAM buffer to harbor the slack between work generation and consumption and to hide the latency of updates to off-chip vertices. Unlike prior accelerators that focus on graph core microarchitecture to improve locality, SEGA offers scalable performance for large graphs without the need to increase on-chip resources.

Chapter 4 focuses on the interconnect between graph accelerators and different packaging technologies to organize the graph accelerators efficiently. It studies the network traffic pattern and bandwidth needs in a randomly assigned graph with no pre-processing costs. It shows how SiPh can enable scaling out without degrading performance.

Chapter 5 proposes LLM (Low Latency Memory), a codesign of the DRAM microarchitecture, the memory controller, and the last level cache and DRAM interconnect by leveraging embedded silicon photonics in 2.5D/3D integrated system on a chip. LLM relies on Wavelength Division Multiplexing (WDM)-based photonic interconnects to reduce the contention throughout the memory subsystem. LLM also increases the bank-level parallelism, eliminates bus conflicts by using dedicated optical data paths, and reduces the access energy per bit with shorter global bit lines and smaller row buffers. LLM exhibits low memory access latency for traffic with both regular and irregular access patterns. For irregular traffic, LLM achieves high bandwidth utilization (over 80% peak throughput compared to 20% of HBM2.0). For real workloads, LLM achieves 3× and 1.8× lower execution time compared to HBM2.0 and a state-of-the-art memory system with high memory level parallelism, respectively. This study also demonstrates that by reducing queuing on the data path, LLM can achieve on average, 3.4× lower memory latency variation compared to HBM2.0.

State-of-the-art computing systems employ increasingly complex hardware and software stacks to meet their performance goals. A large portion of these added complexities is due to limitations at the technology level (e.g., memory wall, pin wall, reticle size, energy cost of data movements, etc). These limitations shift the architects away from their ideal design choices, and result in many compromises at the design stage. The unique properties of SiPh links in terms of their energy-consumption and bandwidth-density can be utilized to change the way we think about computing

systems today. This dissertation explored a subset of the design space for computing systems enabled by SiPh, and provided pointers for several studies in the future.

Bibliography

[1] *9th dimacs implementation challenge: Shortest paths.*

[2] *Ayar Labs Realizes Co-Packaged Silicon Photonics – WikiChip Fuse.*

[3] *Graph 500 — large-scale benchmarks.*

[4] *An important milestone in delivering on the promise of optical i/o - ayar labs.*

[5] *Intel demos 528-thread chip with 1tb/s of optical bandwidth • the register.*

[6] *JEDEC.*

[7] *Thermistor Specification Fiber Specification an exemplary Eye Diagram of one F-P mode Externally modulated at 2.5GHz filtered-out single channel.*

[8] *WDC - Hyperlink Graphs.*

[9] *Zen - Microarchitectures - AMD - WikiChip.*

[10] M. Abeydeera and D. Sanchez, *Chronos: Efficient speculative parallelism for accelerators*, in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1247–1262.

[11] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, *A scalable processing-in-memory accelerator for parallel graph processing*, in Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015, pp. 105–117.

[12] H. Akinaga and H. Shima, *Resistive random access memory (reram) based on metal oxides*, Proceedings of the IEEE, 98 (2010), pp. 2237–2251.

[13] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, *Rabbit order: Just-in-time parallel reordering for fast graph analysis*, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 22–31.

[14] A. Ayupov, S. Yesil, M. M. Ozdal, T. Kim, S. Burns, and O. Ozturk, *A template-based design methodology for graph-parallel hardware accelerators*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37 (2017), pp. 420–430.

[15] V. Balaji and B. Lucia, *When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs*, in 2018 IEEE International Symposium on Workload Characterization (IISWC), 2018, pp. 203–214.

[16] C. Batten et al., *Building many-core processor-to-dram networks with monolithic cmos silicon photonics*, International Symposium on Microarchitecture (MICRO), (2009.), pp. 8–21.

[17] S. Beamer and K. Asanović, *The gap benchmark suite*, arXiv preprint arXiv:1508.03619, (2015).

[18] S. Beamer, K. Asanović, and D. Patterson, *Gail: The graph algorithm iron law*, in Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, 2015, pp. 1–4.

[19] S. Beamer et al., *Re-architecting dram memory systems with monolithically integrated silicon photonics*, in Proceedings International Symposium on Computer Architecture (ISCA)., IEEE, 2010, p. 129–140.

[20] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, *Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling*, in 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6.

[21] H. Cao et al., *Scaling graph traversal to 281 trillion edges with 40 million cores*, in PPoPP, 2022, pp. 234–245.

[22] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, et al., *Impulse: Building a smarter memory controller*, in Proceedings Fifth International Symposium on High-Performance Computer Architecture, IEEE, 1999, pp. 70–79.

[23] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan, *Gaas-x: Graph analytics accelerator supporting sparse data representation using crossbar architectures*, in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 433–445.

[24] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia, *Managing dram latency divergence in irregular gpgpu applications*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), 2014, pp. 128–139.

[25] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, *Architecting an energy-efficient dram system for gpus*, in IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2017, pp. 73–84.

[26] S. Cheung, T. Su, K. Okamoto, and S. Yoo, *Ultra-compact silicon photonic 512× 512 25 ghz arrayed waveguide grating router*, IEEE Journal of Selected Topics in Quantum Electronics, (2013), pp. 310–316.

[27] M. J. Cianchetti, J. C. Kerekes, and D. H. Albonesi, *Phastlane: a rapid transit optical routing network*, in Proceedings of the 36th annual international symposium on Computer architecture, 2009, pp. 441–450.

[28] E. Cooper-Balis and B. Jacob, *Fine-grained activation for power reduction in dram*, IEEE Micro, 30 (2010), pp. 34–47.

[29] V. Dadu, S. Liu, and T. Nowatzki, *Polygraph: Exposing the value of flexibility for graph processing accelerators*, in 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2021, pp. 595–608.

[30] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011).

[31] G. de Valicourt, J. E. Simsarian, A. Maho, R. Brenot, K. Kim, A. Melikyan, P. Dong, C.-M. Chang, and Y.-K. Chen, *Dual hybrid silicon-photonic laser with fast wavelength tuning*, in Optical Fiber Communications Conference and Exhibition (OFC), 2016, pp. 1–3.

[32] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, *Bandwidth bandit: Quantitative characterization of memory contention*, in Proceedings of the 2013 IEEE/ACM CGO, 2013, pp. 1–10.

[33] P. Erdős, A. Rényi, et al., *On the evolution of random graphs*, Publ. Math. Inst. Hung. Acad. Sci, 5 (1960), pp. 17–60.

[34] M. Fariborz, M. Samani, T. O'Neill, J. Lowe-Power, S. B. Yoo, and V. Akella, *A model for scalable and balanced accelerators for graph processing*, IEEE Computer Architecture Letters, 21 (2022), pp. 149–152.

[35] M. Fariborz, X. Xiao, P. Fotouhi, R. Proietti, and S. B. Yoo, *Silicon photonic flex-lions for reconfigurable multi-gpu systems*, Journal of Lightwave Technology, 39 (2021), pp. 1212–1220.

[36] P. Fotouhi, S. Werner, J. Lowe-Power, and S. B. Yoo, *Enabling scalable chiplet-based uniform memory architectures with silicon photonics*, in Proceedings of the International Symposium on Memory Systems, 2019, pp. 222–334.

[37] P. Grani, R. Proietti, V. Akella, and S. B. Yoo, *Design and evaluation of awgr-based photonic noc architectures for 2.5 d integrated high performance computing systems*, in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2017, pp. 289–300.

[38] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, et al., *The architectural implications of facebook's dnn-based personalized recommendation*, in IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2020, pp. 488–501.

[39] H. Ha, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, *Improving energy efficiency of dram by exploiting half page row access*, in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2016, pp. 1–12.

[40] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. 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.

[41] M. Han and K. Daudjee, *Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems*, Proceedings of the VLDB Endowment, 8 (2015), pp. 950–961.

[42] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, *Chargecache: Reducing dram latency by exploiting row access locality*, in IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2016.

[43] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, *Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas*, in 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), IEEE, 2021, pp. 1–9.

[44] J. JESD235A, *High bandwidth memory (hbm) dram*, JEDEC Solid State Technology Association, (2015).

[45] D. Kaseridis, J. Stuecheli, and L. K. John, *Minimalist open-page: A dram page-mode scheduling policy for the many-core era*, in International Symposium on Microarchitecture (MICRO), IEEE, 2011, pp. 24–35.

[46] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell System Technical Journal, 49 (1970), pp. 291–307.

[47] Y. Kim et al., *A case for exploiting subarray-level parallelism (salp) in dram*, in Proceedings of the International Symposium on Computer Architecture (ISCA)., IEEE, 2012, pp. 368–379.

[48] N. Kirman, M. Kirman, R. K. Dokania, J. F. Martinez, A. B. Apsel, M. A. Watkins, and D. H. Albonesi, *Leveraging optical technology in future bus-based chip multiprocessors*, in 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), IEEE, 2006, pp. 492–503.

[49] H. Kwak, C. Lee, H. Park, and S. 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.

[50] C. E. Leiserson, *The cilk++ concurrency platform*, in Proceedings of the 46th Annual Design Automation Conference, 2009, pp. 522–527.

[51] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, *Kronecker graphs: an approach to modeling networks.*, Journal of Machine Learning Research, 11 (2010).

[52] H. Li, Z. Xuan, A. Titriku, C. Li, K. Yu, B. Wang, A. Shafik, N. Qi, Y. Liu, R. Ding, et al., *A 25 gb/s, 4.4 v-swing, ac-coupled ring modulator-based wdm transmitter with wavelength stabilization in 65 nm cmos*, IEEE Journal of Solid-State Circuits, (2015), pp. 3145–3159.

[53] L. Li, P. Chia, P. Ton, M. Nagar, S. Patil, J. Xue, J. Delacruz, M. Voicu, J. Hellings, B. Isaacson, et al., *3d sip with organic interposer for asic and memory integration*, in IEEE 66th Electronic Components and Technology Conference (ECTC), IEEE, 2016, pp. 1445–1450.

[54] C. Liu, Z. Shao, K. Li, M. Wu, J. Chen, R. Li, X. Liao, and H. Jin, *Scalabfs: A scalable bfs accelerator on fpga-hbm platform*, in The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2021, pp. 147–147.

[55] G. Liu, R. Proietti, M. Fariborz, P. Fotouhi, X. Xiao, and S. B. Yoo, *Architecture and performance studies of 3d-hyper-flex-lion for reconfigurable all-to-all hpc networks*, in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–16.

[56] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, *Graphlab: A new framework for parallel machine learning*, arXiv preprint arXiv:1408.2041, (2014).

[57] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, et al., *The gem5 simulator: Version 20.0+*, arXiv preprint arXiv:2007.03152, (2020).

[58] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, *The hpc challenge (hpcc) benchmark suite*, in Proceedings of the 2006 ACM/IEEE conference on Supercomputing, vol. 213, 2006, p. 1.

[59] A. Manocha, T. Sorensen, E. Tureci, O. Matthews, J. L. Aragón, and M. Martonosi, *Graphattack: Optimizing data supply for graph applications on in-order multicore architectures*, ACM Transactions on Architecture and Code Optimization (TACO), 18 (2021), pp. 1–26.

[60] S. Matsuo and T. Segawa, *Microring-resonator-based widely tunable lasers*, IEEE Journal of Selected Topics in Quantum Electronics, (2009), pp. 545–554.

[61] R. R. McCune, T. Weninger, and G. Madey, *Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing*, ACM Computing Surveys (CSUR), 48 (2015), pp. 1–39.

[62] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. 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.

[63] Q. Nguyen and D. Sanchez, *Fifer: Practical acceleration of irregular applications on reconfigurable architectures*, in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 1064–1077.

[64] Q. M. Nguyen and D. Sanchez, *Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism*, in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2020, pp. 596–608.

[65] C. J. Nitta, M. Farrens, and V. Akella, *On-chip photonic interconnects: A computer architect's perspective*, Synthesis Lectures on Computer Architecture, (2013), pp. 1–111.

[66] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, *Tiny but mighty: Designing and realizing scalable latency tolerance for manycore socs*, in Proceedings of the 49th Annual International Symposium on Computer Architecture, 2022, pp. 817–830.

[67] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, *Dalorex: A data-local program execution and architecture for memory-bound applications*, in 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), IEEE, 2023, pp. 718–730.

[68] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, *Energy efficient architecture for graph analytics accelerators*, ACM SIGARCH Computer Architecture News, 44 (2016), pp. 166–177.

[69] M. O'Connor et al., *Fine-grained dram: Energy-efficient dram for extreme bandwidth systems*, in International Symposium on Microarchitecture (MICRO), IEEE, 2017, pp. 41–54.

[70] I. A. Papistas and V. F. Pavlidis, *Bandwidth-to-area comparison of through silicon vias and inductive links for 3-d ics*, in 2015 European Conference on Circuit Theory and Design (ECCTD), IEEE, 2015, pp. 1–4.

[71] M. S. Parekh, P. A. Thadesar, and M. S. Bakir, *Electrical, optical and fluidic through-silicon vias for silicon interposer applications*, in 2011 IEEE 61st Electronic Components and Technology Conference (ECTC), IEEE, 2011, pp. 1992–1998.

[72] J. T. Pawlowski, *Hybrid memory cube (hmc)*, in 2011 IEEE Hot Chips 23 Symposium (HCS), 2011, pp. 1–24.

[73] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, *The tao of parallelism in algorithms*, ACM SIGPLAN Notices, 46 (2011), pp. 12–25.

[74] R. Proietti, X. Xiao, K. Zhang, G. Liu, H. Lu, P. Fotouhi, J. Messig, and S. Yoo, *Experimental demonstration of a 64-port wavelength routing thin-clos system for data center switching architectures*, Journal of Optical Communications and Networking, 10 (2018), pp. B49–B57.

[75] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, *Graphpulse: An event-driven hardware accelerator for asynchronous graph processing*, in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2020, pp. 908–921.

[76] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, *Jetstream: Graph analytics on streaming data with event-driven hardware accelerator*, in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 1091–1105.

[77] S. RRumley, D. Nikolova, R. Hendry, Q. Li, D. Calhoun, and K. Bergman, *Silicon photonics for exascale systems*, Journal of Lightwave Technology (JLT), (2015).

[78] A. Shacham, K. Bergman, and L. P. Carloni, *Photonic networks-on-chip for future generations of chip multiprocessors*, IEEE Transactions on Computers, (2008), pp. 1246–1260.

[79] K. Shang, S. Pathak, C. Qin, and S. B. Yoo, *Low-loss compact silicon nitride arrayed waveguide gratings for photonic integrated circuits*, IEEE Photonics Journal, 9 (2017), pp. 1–5.

[80] Y. Shen, X. Meng, Q. Cheng, S. Rumley, N. Abrams, A. Gazman, E. Manzhosov, M. S. Glick, and K. Bergman, *Silicon photonics for extreme scale systems*, Journal of Lightwave Technology (JLT), (2019), pp. 245–259.

[81] J. Shun and G. 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.

[82] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, *Graphr: Accelerating graph processing using reram*, in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2018, pp. 531–543.

[83] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, *Graphr: Accelerating graph processing using reram*, vol. 2018-Febru, IEEE Computer Society, 3 2018, pp. 531–543.

[84] I. D. Sousa and L. Achard, *The future of packaging with silicon photonics*, 2016.

[85] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, *Graphmat: High performance graph analytics made productive*, arXiv preprint arXiv:1503.07241, (2015).

[86] F. I. system performance via fine-grained In-DRAM data relocation and caching, *Figaro: Improving system performance via fine-grained in-dram data relocation and caching*, in International Symposium on Microarchitecture (MICRO), IEEE, 2020, pp. 313–328.

[87] K. Takada, M. Abe, M. Shibata, M. Ishii, and K. Okamoto, *Low-crosstalk 10-ghz-spaced 512-channel arrayed-waveguide grating multi/demultiplexer fabricated on a 4-in wafer*, IEEE Photonics Technology Letters, 13 (2001), pp. 1182–1184.

[88] A. N. Udipi et al., *Combining memory and a controller with photonics through 3d-stacking to enable scalable and energy-efficient systems*, ACM SIGARCH Computer Architecture News, 39 (2011), pp. 425–436.

[89] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, *Rethinking dram design and organization for energy-constrained multi-cores*, in Proceedings of the International Symposium on Computer Architecture (ISCA)., 2010, pp. 175–186.

[90] L. G. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), p. 103–111.

[91] ——, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), p. 103–111.

[92] M. Wade, E. Anderson, S. Ardalan, P. Bhargava, S. Buchbinder, M. L. Davenport, J. Fini, H. Lu, C. Li, R. Meade, et al., *Teraphy: A chiplet technology for low-power, high-bandwidth in-package optical i/o*, International Symposium on Microarchitecture (MICRO), (2020), pp. 63–71.

[93] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. 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.

[94] Z. Wang, H. Huang, J. Zhang, and G. Alonso, *Shuhai: Benchmarking high bandwidth memory on fpgas*, in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2020, pp. 111–119.

[95] S. Werner, P. Fotouhi, R. Proietti, and S. B. Yoo, *Awgr-based optical processor-to-memory communication for low-latency, low-energy vault accesses*, in Proceedings of the International Symposium on Memory Systems (MEMSYS), 2018, pp. 269–278.

[96] S. Werner, P. Fotouhi, X. Xiao, M. Fariborz, S. B. Yoo, G. Michelogiannakis, and D. Vasudevan, *3d photonics as enabling technology for deep 3d dram stacking*, in Proceedings of the International Symposium on Memory Systems, 2019, pp. 206–221.

[97] S. Werner, J. Navaridas, and M. Luján, *Amon: An advanced mesh-like optical noc*, in IEEE 23rd Annual Symposium on High-Performance Interconnects, 2015, pp. 52–59.

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

[99] P. Yao, L. Zheng, Y. Huang, Q. Wang, C. Gui, Z. Zeng, X. Liao, H. Jin, and J. Xue, *Scalagraph: A scalable accelerator for massively parallel graph processing*, in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), IEEE, 2022, pp. 199–212.

[100] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, *An efficient graph accelerator with parallel data conflict management*, in Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, 2018, pp. 1–12.

[101] K. Yu, C. Li, H. Li, A. Titriku, A. Shafik, B. Wang, Z. Wang, R. Bai, C.-H. Chen, M. Fiorentino, et al., *A 25 gb/s hybrid-integrated silicon photonic source-synchronous receiver with microring wavelength stabilization*, IEEE Journal of Solid-State Circuits (JSSC), (2016), pp. 2129–2141.

[102] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, *Graphp: Reducing communication for pim-based graph processing with efficient data partition*, in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2018, pp. 544–557.

[103] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, *Half-dram: A high-bandwidth and low-power dram architecture from the rethinking of fine-grained activation*, ACM SIGARCH Computer Architecture News, 42 (2014), pp. 349–360.

[104] Y. Zhang, Q. Gao, L. Gao, and C. Wang, *Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation*, IEEE Transactions on Parallel and Distributed Systems, 25 (2013), pp. 2091–2100.

[105] Y. Zhang, Y.-C. Ling, Y. Zhang, K. Shang, and S. B. Yoo, *High-Density Wafer-Scale 3-D Silicon-Photonic Integrated Circuits*, IEEE Journal of Selected Topics in Quantum Electronics, (2018), pp. 1–10.

[106] M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing, *Gram: graph processing in a reram-based computational memory*, in IEEE Asia and South Pacific Design Automation Conference, 2019.

[107] S. Zhou, C. Chelmis, and V. 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.

[108] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, *An fpga framework for edge-centric graph processing*, in Proceedings of the 15th ACM International Conference on Computing Frontiers, 2018, pp. 69–77.

[109] X. Zhu, W. Chen, W. Zheng, and X. Ma, *Gemini: A computation-centric distributed graph processing system.*, in OSDI, vol. 16, 2016, pp. 301–316.

[110] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, *Graphq: Scalable pim-based graph processing*, in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 712–725.