WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Dissertation Examination Committee:
Roger Chamberlain, Chair
Jeremy Buhler
Ron Cytron
Christopher Gill
I-Ting Angelina Lee
Xuan Zhang

Efficient Computation Using Near-Memory Processing and High-Level Synthesis

by
Chenfeng Zhao

A dissertation presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2024
St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

This journey toward earning my PhD is like a phoenix rising from the ashes—challenging yet immensely rewarding. I would like to express my deep appreciation to the many individuals whose support and guidance were essential to my success. Without them, this achievement would not have been possible.

I would like to express my profound gratitude to my advisor, Dr. Roger Chamberlain, whose invaluable support has been instrumental throughout my PhD journey. Dr. Chamberlain is an exceptional mentor, always available to address my queries, no matter how trivial they may seem. His guidance has been crucial in helping me navigate challenges, while his trust and respect have provided me with the freedom to explore the unknown. He has consistently encouraged me when doubts arose about my direction, and has been forthright in correcting my course whenever necessary. With his support, I have not only grasped the essence of research but also sustained my passion for it. Dr. Chamberlain is more than an academic mentor; he is a role model in my life.

I give my sincere gratitude to my co-advisor, Dr. Xuan Zhang, for her invaluable guidance throughout this journey. She consistently offered constructive suggestions on my research projects, from research methodologies to scientific writing. Dr. Zhang's insights and critiques have been crucial in refining my arguments and enhancing the coherence of my findings. Her meticulous attention to detail, passion for discovery, and steadfast dedication to advancing knowledge have profoundly influenced my approach to research. I am deeply grateful for her mentorship and inspired by her commitment to scholarly excellence.

I want to thank my committee members, Dr. Jeremy Buhler, Dr. Ron Cytron, Dr. Christopher Gill, Dr. I-Ting Angelina Lee, for graciously agreeing to serve on my committee. Additionally, I give special thanks to Dr. Jeremy Buhler and Dr. Ron Cytron for their invaluable support and guidance within our research group.

I am deeply grateful for the opportunity to work with a group of exceptionally talented colleagues, including Dr. Anthony Cabrera, Dr. Clayton Faber, Dr. Weidong Cao, Dr. An Zou, Dr. Marion Sudvarg, Dr. Huifeng Zhu, Dr. Justin Deters, Dr. Stephen Timcheck, Dr. Yan Wei,

Dr. Ke Liu, Tianrui Ma, Adith Jagadish Boloor, Steven Harris, Ye Htet, Daisy Wang, and all other members who have since graduated but are not mentioned here. Their encouragement during challenging times has been invaluable. Collaborating with such brilliant individuals has been a truly unforgettable experience.

Finally, I wish to express my deepest gratitude to my parents, Jiang Zhao and Yan Zhang, for their unwavering support and companionship during the most challenging periods of my PhD journey. Our weekly discussions every Friday have been a cornerstone of my perseverance and motivation. The encouragement and comfort they provided have been invaluable, helping me navigate the ups and downs of my academic pursuits. Their belief in my abilities and their constant presence provided the strength I needed to continue, making them an integral part of my success.

This research is supported by NSF under grants CNS-1739643 and CNS-1763503 and a gift from BECS Technology, Inc. We are grateful for the use of the Open Cloud Testbed [52, 120] as an experimentation platform.

Chenfeng Zhao

Washington University in St. Louis
August 2024

ABSTRACT OF THE DISSERTATION

Efficient Computation Using Near-Memory Processing and High-Level Synthesis

by

Chenfeng Zhao

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2024

Professor Roger Chamberlain, Chair

With the diminishing of Moore's law and the end of Dennard scaling, alongside the explosion of data, the need for efficient computation—both in terms of performance and energy consumption—has become paramount. This is particularly crucial for processing large-scale data across various workloads. While a universal computation method is highly desirable, the inherent unique characteristics of different applications and datasets preclude a one-size-fits-all solution. It is usually the case that an appropriate computation technology choice could benefit a particular set of workloads and vice versa. Addressing this challenge requires designers to possess profound knowledge and insights into both hardware and software. Thus, the field of efficient computation involves many research questions driven by applications and platforms.

In this dissertation, we focus on three representative types of applications: data integration, irregular graph processing, and graph neural networks. Our work in efficient computing leverages Near-Memory Processing (NMP) with 3D-stacked memory and High-Level Synthesis (HLS) on Field-Programmable Gate Arrays (FPGAs) to accelerate these workloads. We start the dissertation by adopting NMP technology based on 3D-stacked memory to accelerate data integration applications in terms of performance and energy consumption. We then present SuperCut, a novel hardware/software graph partitioning framework for near-memory graph processing. Subsequently, we describe GNNHLS, an open-source framework for the

comprehensive evaluation of Graph Neural Network (GNN) kernels on FPGAs using high-level synthesis. Given the complexities of optimizing GNN HLS, we introduce HLPerf, an open-source, simulation-based performance evaluation framework for dataflow architectures that both supports early exploration of the design space and shortens the performance evaluation cycle.

Chapter 1

Introduction

In this era of data explosion, the 3Vs of Big Data (volume, velocity and variety) defy the traditional mechanisms of data collection, management and processing. In traditional mechanisms, a large volume of data generated from a variety of fields, such as machine learning, astronomy and bioinformatics, is collected from memory to processors, processed by CPUs, and then written back to memory. Large-volume data transfer between host-side processors and memory becomes a bottleneck of the system due to the memory wall [100]. Memory technology has not been able to keep pace with the development of CPUs in traditional architecture in terms of performance (either latency or bandwidth) and energy consumption. In order to bridge this gap, multiple levels of cache with higher speed and lower capacity are added between processors and main memory, so that data is moved to caches from the main memory first and then processed by computation cores. In practice, many data-intensive applications with sufficient cache locality/utilization benefit from this multi-level memory hierarchy because a number of expensive memory accesses are avoided and replaced with less expensive cache accesses to process reused data. However, for some other data-intensive applications, the conventional CPU-centric system with multi-level memory hierarchy is not able to improve or even exacerbates the performance and energy consumption during execution. There are several challenges in these systems:

- Narrow memory channels consisting of limited number of pins on the memory package cannot provide sufficient bandwidth to meet the ever-growing demands of modern multi-processor designs.
- Long data path as well as cache miss overhead caused by applications with low data reuse running on a multi-level memory hierarchy results in high latency.
- Energy consumption of data movement between host processors and memory becomes a key contributor to the total energy consumption of the system for data-intensive

applications. For example, 66.7% of total system energy on average when implementing Google workloads on customer devices is spent on the host-memory data movement [12].

To solve the challenges above, the concept of near-memory processing (NMP), also called processing-in-memory (PIM), has been proposed. The idea of NMP is to implement computation closer to where data resides. To realize this idea, light-weight NMP cores are integrated into the memory chip. By offloading specific computation targets to these cores near the memory and running them in parallel, the expensive cost of host-memory data movement can potentially be eliminated.

The emergence of 3-D stacked memory technology has opened the door for practical deployment of processor cores near the physical DRAM. The structure of these memories has multiple DRAM chips stacked on top of a single logic chip. These chip layers are connected by vertical high-bandwidth and low-power through-silicon vias (TSVs). The logic layer at the bottom consists of both interconnections and controller logic. In current commercial implementations, the logic layer is not fully utilized (i.e., there is a portion of unused area on the chip). Therefore, the research community has considered integrating general-purpose processor cores or custom accelerators into the logic layer as an approach to implementing a near-memory processing strategy.

Given the variety of characteristics of different applications, the utility of Near-Memory Processing (NMP) technology, particularly when applied to workloads involving 3D-stacked memory cubes, remains uncertain in terms of performance enhancement and energy efficiency for specific applications. This dissertation investigates the application of NMP technology to accelerate two representative types of applications: 1) data integration applications characterized by regular patterns, and 2) conventional graph processing workloads, which typically exhibit irregular patterns.

Data Integration is a term frequently used for the general problem of taking data in some initial form and transforming it into a desired form. While the individual transforms are each (mostly) quite straightforward, the task is quickly complicated by the fact that individual data streams can be quite large and there are frequently many streams, each requiring a distinct transformation specification. Tens to hundreds of multi-gigabyte data streams must be concurrently integrated, and this must be done prior to the real data analysis, the ultimate

goal. The issue of how to effectively achieve data integration is a pain point for enterprise data, sensor data, scientific data, financial data, to name a few.

Here, we investigate the use of near-memory processing to execute data integration tasks. In particular, we seek to exploit two properties that are common to many data integration workloads and are well-suited to execution on near-memory processing architectures.

1. **Abundant parallelism** – data integration workloads are, for the most part, embarrassingly parallel, so their performance scales well with large numbers of processor cores. Relative to traditional computing systems, the majority of near-memory processing architectures employ a larger number of smaller cores.

2. **Substantial data movement** – they are also characterized by having a large fraction of their operations data movement instructions, implying a strong sensitivity to the architecture of the memory subsystem. A central feature of near-memory processing systems is the proximity of the processor cores to the memory.

We explore the impact of each of these properties, quantifying both the performance implications and energy savings achievable though the use of near-memory processing architectures for executing data integration workloads. Overall, for the particular near-memory processing system modeled, the performance improved an average 3.5× and the energy reduced an average 4.2× (76%) relative to a traditional baseline system.

Our second application class of interest is *graph processing*. Due to their ability to capture the complex dependencies and relationships among individual data elements, graphs constitute an important data structure that have been widely used to represent social networks, citation networks, road networks, genome sequences, etc. The recent proliferation of graph processing applications, including machine learning [99], recommendation systems [67], and social network analysis [77], has heightened the need for efficiently processing graphs, both in terms of performance and energy consumption. Hence, a number of approaches have been proposed to efficiently process large-scale graphs [36, 61, 64, 70, 118].

The inherent properties of graph analytic applications pose challenges for conventional memory and communications systems, which in turn become performance bottlenecks. First, the operation of traversing neighbourhood vertices shows poor locality due to random memory accesses. Second, many graph algorithms have high memory bandwidth requirements because

the node-level computation is relatively simple. Third, when executing in parallel, frequent data movement across the system puts pressure on the communications network.

Since the demand for higher memory bandwidth is an important part of accelerating large-scale graph processing, Near-Memory Processing has been proposed to accelerate these tasks. Figure 1.1 illustrates a general abstraction of near-memory graph processing systems. Real-world information is abstracted into graph data structures. Graph processing applications are deployed to computing units inside memory chips and executed in parallel. Interaction between memory chips is communicated via an interconnection network.

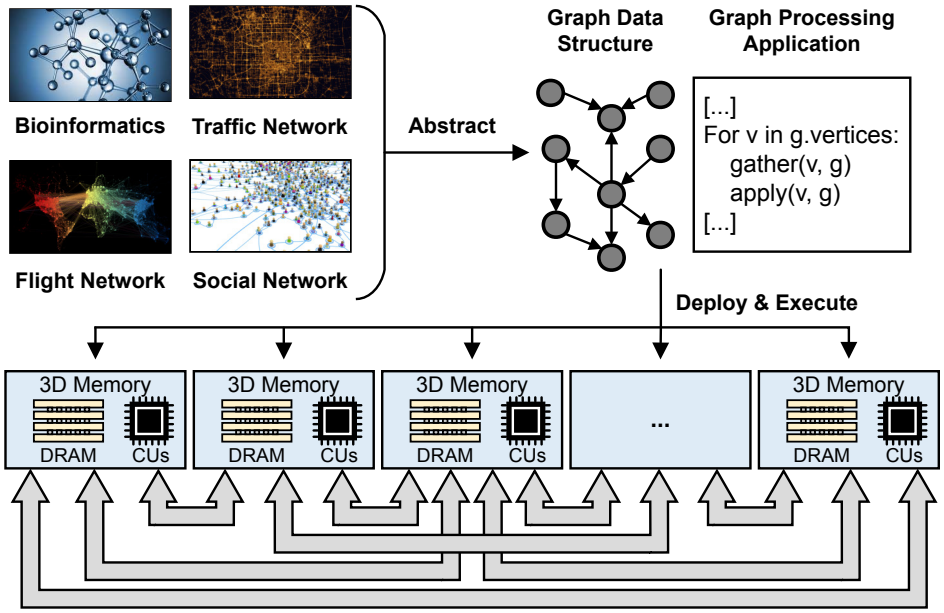


Figure 1.1: Near-memory graph processing system.

Executing graph processing applications on NMP architectures is distinct from traditional systems for a pair of reasons. First, the number of compute nodes can scale up substantially in a shared-memory paradigm since cache coherence is often not needed. Second, delivering information between compute nodes utilizes mechanisms that are substantially less heavy-weight than traditional message-passing protocols.

Subsequent works have proposed to diminish communication bottlenecks by alternative preprocessing of the graph [108] or by run-time adaptations [7, 8]. GraphP [108] proposes *source-cut* partitioning, in which replicas of the source vertex of each cross-cube edge are deployed in destination cubes, so that multiple cross-cube edges sent from a common source

vertex to the same destination cube can be reduced to one. Therefore, lower cross-cube communication volume is required relative to Tesseract.

Despite the promising results from source-cut partitioning, there is still room for improvement. We observe that after performing source-cut partitioning, cross-cube communication still takes a significant portion of execution time (12%-78%) and energy consumption (14%-73%). This invites the open research question: *how effectively can partitioning algorithms reduce communications overheads while maintaining computational balance in an NMP system*? To further explore this question, we introduce a hardware/software co-design framework for near-memory graph processing, called *SuperCut*, and evaluate its effectiveness. To evaluate our framework, we build a multi-cube, near-memory processing simulation platform with reconfigurable logic kernels as computing units by extending `gem5-SALAM` [76]. Our evaluation results show that SuperCut provides up to 1.8× total energy reduction and 2.6× speedup with 45% lower extra memory footprint relative to GraphP.

Besides traditional graph processing applications, machine learning (ML) on graphs has experienced a surge of popularity in the past decade, since traditional ML models, which are designed to process Euclidean data with regular structures, are ineffective at performing prediction tasks on graphs. Due to their simplicity and superior representation learning ability, graph neural networks (GNNs) [29, 49, 90, 103, 107] have achieved impressive performance on various graph learning tasks, such as node classification, graph classification, etc.

To implement GNNs, a set of widespread libraries, such as PyTorch Geometric (PYG) [34] and Deep Graph Library (DGL) [93], are built upon general-purpose ML frameworks (e.g. PyTorch [71]) targeting CPU and GPU platforms. However, the performance and energy consumption of GNN implementations are hindered by both hardware platforms and software frameworks:

1. Distinct from traditional NNs and graph processing workloads, GNNs combine the irregular communication-intensive patterns of graph processing and the regular computation-intensive patterns of NNs. This feature can lead to ineffectual computation on CPUs and GPUs.

2. Since these frameworks assemble functions in a sequential way, one function will not start until the previous one finishes. This execution model leads to extra memory

accesses, footprint, and implicit barriers for intermediate results, limiting the potential performance, energy consumption and the scale of graph datasets.

Field-Programmable Gate Arrays (FPGAs) are potentially an attractive approach to GNN inference acceleration. FPGAs' massive fined-grained parallelism provides opportunities to exploit GNNs' inherent parallelism. They also deliver better performance per watt than general-purpose computing platforms. In addition, FPGAs' reconfigurability and concurrency provide great flexibility to solve the challenges of hybrid computing patterns and ineffectual execution. Most of the prior works investigating FPGAs focus on accelerating a specific GNN model implemented using hardware description languages (HDL). AWB-GCN [35], as one of the earliest FPGA-based works, proposes a GCN accelerator using HDL to solve the workload imbalance problem due to the distinct sparsity of different components. BoostGCN [106] proposes a graph partition algorithm in a preprocessing step to address workload imbalance issues. Despite these promising results, HDL design methodology is not suitable for widespread adoption for GNN implementations due to the conflict between the non-trivial development efforts with HDL and the rapid emergence of new GNN models. To address this challenge, high-level synthesis (HLS) tools are proposed to create GNN kernels using popular languages such as C/C++. With the help of HLS, development time is substantially shortened relative to HDL designs. Lin et al. [60], as one of the first works, proposes an HLS-based accelerator for GCN with separated sparse-dense matrix multiplication units and dense matrix multiplication units which are connected by shared memory and execute sequentially. GenGNN [2] proposes a framework to accelerate GNNs for real-time requirements where the whole graph and corresponding intermediate results are stored in on-chip resources on the FPGA. Despite these promising results, this work is limited to small-scale graphs with low edge-to-node ratio due to on-chip memory usage being proportional to graph scale and feature dimensions.

Distinct from pure software programming, HLS developers need to adopt multiple optimization pragmas and follow certain coding styles to achieve best performance and energy cost. As reported in [15], the performance difference between a well-optimized version and a non-optimized version of the same kernel can be two to three orders of magnitude. This invites an open question: *how effectively can modern HLS tools accelerate GNN inference?*

We introduce GNNHLS an open-source framework for comprehensive evaluation of GNN kernels on FPGAs via HLS. GNNHLS contains a software stack extended from a prior GNN benchmark [31] based on PyTorch and DGL for input data generation and conventional

platform baseline deployments (i.e., CPUs and GPUs). It also contains six well-optimized general-purpose GNN applications. These kernels can be classified into 2 classes: (1) isotropic GNNs in which every neighbor contributes equally to the update of the target vertex, and (2) anisotropic GNNs in which edges and neighbors contribute differently to the update due to the adoption of operations such as attention and gating mechanisms. Our evaluation results show that GNNHLS provides up to 50.8× speedup and 423× energy reduction relative to the multicore CPU baseline. Compared with the GPU baselines, GNNHLS achieves up to 5.16× speedup and 74.5× energy reduction.

Although HLS bridges the gap between software and hardware development, optimizing HLS codes is substantially distinct from conventional software programming. In fact, due to the FPGAs' inherent attributes, such as lack of built-in cache mechanisms, low clock frequency (relative to traditional processor cores), and fine-grained configurability, the performance difference between a well-optimized version and naive version of the same kernel can be two or three orders of magnitude [15, 32, 85]. Therefore, to achieve the best performance, HLS developers need to explore a large optimization space for HLS designs with various optimization pragmas, coding paradigms, etc.

As the complexity of kernels increases, optimizing (or auto-optimizing) such kernels is difficult via conventional HLS workflows for several reasons:

1. Since pure C emulation is only designed for functionality verification, current HLS developers have to use RTL simulation to understand performance by manually mapping the results of individual signals in the generated waveform back to the HLS code. However, since all the signal names are auto-generated, they are not easily comprehensible by users. Besides, RTL simulation usually takes a very long time, making the tuning effort arduous. Even worse, it is exacerbated by the fact that tuning with a small example data set is less meaningful for GNN kernels in terms of performance estimation because of the inherent irregularity of graph datasets and algorithms. In other words, distinct graph topologies can significantly impact the final performance achieved. Therefore, when it comes to large-scale graphs, RTL simulation is impractical to be used to optimize GNN kernels with these graphs.

2. The notion of dataflow architectures which exploit task-level parallelism, where multiple functions are connected by FIFOs and executed concurrently instead of sequentially, further mystifies the optimization process because it induces a wider set of design

space challenges including: task partitioning, FIFO depth tuning, and bottleneck identification, which are distinct from conventional computation platforms.

The critical missing piece in the optimization task is the availability of fast, high-quality understanding of the performance implications of the design choices that are made. In this work we seek to address this missing element, providing the the designer (whether it be a human or an automatic design space exploration tool [80]) with performance predictions both quickly and with sufficient accuracy that they can be used effectively.

Traditional approaches to performance assessment either involve static assessment (i.e., compile-time analysis) or cycle-accurate simulation. In this work, we propose a different method, effectively between the approaches of static estimation and cycle-accurate simulation, to investigate the impact of irregularity of data and algorithms on performance. Due to the existence of other HLS tools for functional verification (e.g., software emulation in Vitis), our method decouples functional verification from performance estimation, so that the runtime of the estimation process is independent of the computational details of the FPGA algorithms.

We introduce HLPerf, a performance evaluation methodology that supports the performance variations inherent in data-dependent algorithms (it is simulation based), but relaxes the notion of cycle accuracy and replaces it with "approximate" cycle accuracy. The result is a simulation-based performance estimate that is two orders-of-magnitude faster than state-of-the-art simulations that perform cycle-accurate functional verification.

1.1 Research Questions

In this dissertation, we seek to address the following questions:

- To what extent does NMP benefit data integration applications?
- To what extent can NMP benefit graph processing applications?
- How effectively can modern HLS tools accelerate GNN inference in terms of performance and energy consumption?

- How can we provide the developer of GNN HLS designs with fast performance prediction and sufficient accuracy?

1.2 Contributions

In this dissertation, we make the following specific contributions:

1. Contributions to data integration applications.

    - Characterize data integration workloads to gain insights on the suitability and potential performance benefits of executing data integration near the memory [115].
    - Quantitatively evaluate near-memory processing for the execution of data integration workloads [115].
    - Assess both the performance improvement and energy savings that are achievable, and separately examine the distinct implications of wider parallelism (i.e., a larger number of simpler cores) and lower memory access overheads (i.e., physical location of the cores near the memory) [115].

2. Contributions to graph processing applications.

    - Propose a set of graph partitioning algorithms, containing: (1) a *mixed-cut* partitioning method which reduces communication volume by recognizing more cross-cube edge patterns, and (2) a *vertex-swapping-based greedy* algorithm to further reduce communication volume by iteratively changing the vertex distribution [110, 111].
    - Propose a *three-phase programming model* that is expressive for general vertex programs and explicitly handles computation and communication via user-defined functions along with a custom graph representation to bridge the software and hardware design while diminishing the irregularity of vertex traversal and communication [110, 111].
    - Generate specialized accelerators via high-level synthesis (HLS) and map them to FPGA resources on the logic layer of 3D-stacked memory cubes [110, 111].

3. Contributions to graph neural networks.

   - Propose GNNHLS[1], a framework to evaluate GNN inference acceleration via HLS, containing: (a) a software stack based on PyTorch and DGL for data generation and baseline deployment, and (b) FPGA implementation including 6 well-tuned GNN HLS kernels with host and configuration files which can also be used as benchmarks [112].
   - Characterize the GNN kernels in terms of locality scores and instruction mix to obtain insight into their memory access and computational properties [112].
   - Provide a comprehensive evaluation of our GNN HLS implementations on 4 graph datasets, assessing both performance improvement and energy reduction [112].
   - Propose HLPerf, an open-source[2], approximately-cycle-accurate performance evaluation method, to estimate the dynamic performance of GNN HLS kernels with a dataflow architecture. It gives useful performance guidance with dramatically better simulation speed than both RTL simulation and more recently developed cycle-accurate simulators [112].
   - Describe an approach to automatically transform the HLS C-based source code describing several GNN operations into simulation components [114].
   - Propose a set of high-level quantitative expressions in HLPerf to model the performance impact of various optimization techniques. Decoupling performance estimation from functional verification, HLPerf is faster and can be used to guide dataflow pipeline designs even prior to the authoring of the constituent HLS kernels [114].
   - Provide a comprehensive evaluation of HLPerf using 6 different GNN models on 4 graph datasets plus several additional general-purpose applications, assessing both accuracy of the performance predictions and performance of the simulator itself. Our evaluation results show that the error rate of HLPerf is 7% on average and it is 13 500× faster than RTL simulation and over 400× faster than a state-of-the-art cycle-accurate simulator [114].

[1]Released as a benchmark suite [113] and also available at https://github.com/ChenfengZhao/GNNHLS
[2]Available at https://github.com/ChenfengZhao/HLPerf

1.3 Outline

The dissertation is organized as follows. Chapter 2 gives related work and background information on data integration, near-memory processing, graph processing, GNNs, and HLS. Chapter 3 describes the work in which we adopt near-memory processing to both accelerate the execution of data integration workloads and reduce their energy needs. Chapter 4 introduces SuperCut, a graph partitioning framework for near-memory architectures to effectively reduce communication overheads while maintaining computational balance. Chapter 5 presents GNNHLS, a framework to evaluate GNN inference acceleration via HLS. Chapter 6 provides HLPerf, an open-source, simulation-based performance evaluation framework for dataflow architectures that both supports early exploration of the design space and shortens the performance evaluation cycle. Chapter 7 gives conclusions and future work.

Chapter 2

Background and Related Work

2.1 Data Integration

Data Integration is a term frequently used for the general problem of taking data in some initial form and transforming it into a desired form. While the individual transforms are each (mostly) quite straightforward, the task is quickly complicated by the fact that individual data streams can be quite large and there are frequently many streams, each requiring a distinct transformation specification. Tens to hundreds of multi-gigabyte data streams must be concurrently integrated, and this must be done prior to the real data analysis, the ultimate goal.

In graph analysis, for example, Malicevic et al. [65] describe an improvement to a breadth first search algorithm that results in a 3× improvement in execution time for the breadth first search in isolation. However, it requires the graph data to be in a different form, and when one includes the necessary pre-processing in the performance measurement, the overall execution increases by 1.5×. In cloud micro-services, Pourhabibi et al. [73] report that up to 30% of execution time is currently spent in the data format transforming process, and as protocol processing is improved by the use of smart NICs that fraction will only increase. The issue of how to effectively achieve data integration is a pain point for enterprise data, sensor data, scientific data, financial data, etc.

In this dissertation, we explore the degree to which near-memory processing techniques (the technology for which is described next) can benefit data integration applications.

2.2 Near-Memory Processing

Early NMP systems were based on traditional 2D memory chip and proposed in the 1990s. One of the early proposals is called IRAM [51], the key idea of which is putting a vector processor inside a 2D embedded DRAM chip so that multimedia applications could exploit data parallelism with benefits such as high bandwidth, low latency and low energy consumption. In spite of the promising results, the adoption of these 2D-memory-based NMP architectures was limited by the difference in process technology between computation cores and memory logic.

The emergence of 3-D stacked memory technology has provided a practical opportunity for realizing this vision [83]. These 3-D memories consist of multiple DRAM chips stacked on top of a single logic chip. The chips are connected by multiple vertical through-silicon vias (TSVs). The logic layer at the bottom consists of both interconnection and controller logic. The controller logic serves as memory controller to access DRAM and interconnection logic is designed to manage the data transfer between the stacked memory and the processor chip. In this way, the DRAM layers can be accessed with higher bandwidth and lower power than conventional off-chip memory channels. The underutilized logic layer has both area and power available for integrating compute functions [74, 115]. Commercial offerings include the early Hybrid Memory Cube (HMC) [43] as well as High Bandwidth Memory (HBM) [44]. Figure 2.1 shows the interconnection techniques used by HBM and HMC. Illustrated in Figure 2.1(a), HBM utilizes 2.5D system-in-package (SiP) memory technology, in which a silicon interposer offering an I/O density of 1024 bits connects the memory stack with the underlying circuit board and the logic processor. In contrast, HMC adopts four on-board SerDes links as interconnection to provide high bandwidth, shown in Figure 2.1(b). While there are variations across the specific implementations, the core technology is common. To facilitate fair comparison with earlier work, we use technology parameters from HMC in our simulation models.

Figure 2.2 (on the left) shows the structure of a single memory cube. Each cube is divided into 32 vertical partitions called vaults and has 4 SerDes high-speed links to implement off-chip accesses. With each cube having a capacity of 8 GB, each vault has a capacity of 256 MB. The logic layer at the bottom of the stack consists of both interconnections and vault controller logic. Each vault can provide 10 GB/s bandwidth. Therefore, the internal bandwidth of each cube is 320 GB/s. For off-chip access implemented by the SerDes links,

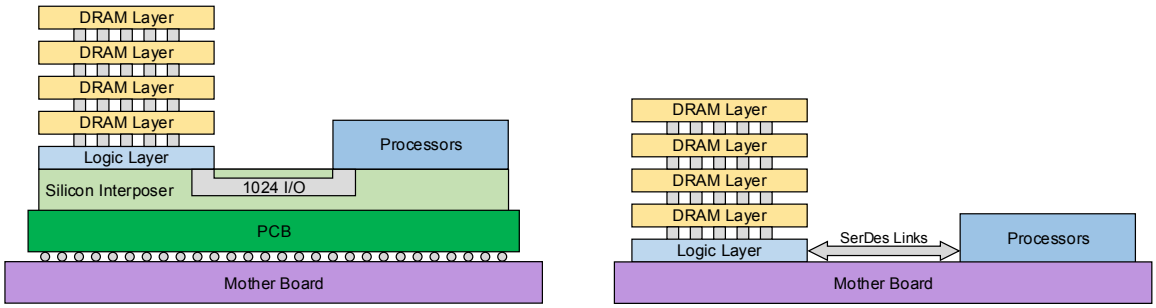


Figure 2.1: Overview of interconnection technology of 3D-stacked memory chips. (a) 2.5D system in package (SiP) technology adopted by HBM and (b) SerDes links adopted by HMC.

each link can provide a bandwidth of 120 GB/s. Each cube then has an external bandwidth of 480 GB/s. In addition, each cube has unused area on its logic layer. Previous works [3, 12] report that the spare area is about 60 mm^2, comprising 26.5% of the the total die area (226 mm^2 per cube [82]). Since the logic layer is not fully utilized in current commercial implementations (i.e., there is a portion of unused area on the chip), the research community has considered integrating general-purpose processor cores or custom accelerators into the logic layer as an approach to implementing a near-memory processing strategy.

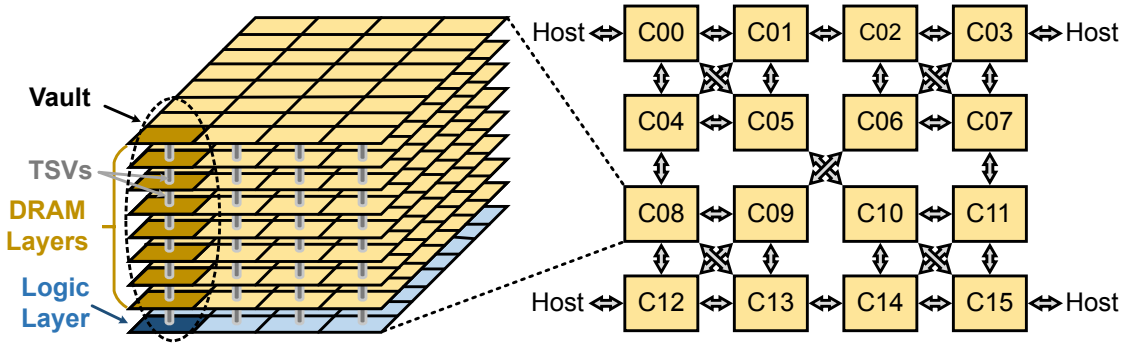


Figure 2.2: A single memory cube (left), and topology of the Dragonfly cube-to-cube interconnection network (right).

Due to the flexibility of the SerDes links, various interconnect topologies can be considered. One prevalent topology is Dragonfly [48], which has high connectivity and a low network diameter. Figure 2.2 (on the right) shows a Dragonfly interconnection network with 16 cubes. Unused links are used to provide connectivity to host cores. We use the Dragonfly network topology in this work. For the 16 cube system, the aggregated internal memory bandwidth is 5 TB/s while the bisection bandwidth of the interconnection network is only 480 GB/s,

implying that inter-cube communications can easily become a performance limiter [24, 108]. In addition to performance, prior work [7] reports that cross-cube communication is also the primary source of energy consumption in graph processing applications, taking up 62% of the total.

There have been a number of proposed near-memory processing (NMP) architectures. Drumond et al. [30] proposed an architecture that utilizes general-purpose Arm Cortex-A35 CPUs as near-memory processing cores. In addition, they altered the execution of common data analytic operators to be more NMP-friendly by optimizing for sequential memory accesses over random memory accesses. Boroumand et al. [12] proposed a near-memory processing architecture with either general-purpose cores or programmable accelerators for Google workloads. Peng et al. [72] explored the suitability of HPC scientific applications on an NMP architecture with general-purpose cores as the execution unit. Our work also utilizes general-purpose programmable cores with small caches in the NMP architecture. However, compared to these prior works, we focus on a different application domain. There has also been recent work utilizing custom logic as execution units. For example, Jang et al. [42] present a accelerator-based NMP architecture for a set of primitives in garbage collection workloads. Singh et al. [83] recently published a survey of the field. In this dissertation, we seek to exploit these ideas for acceleration of data integration workloads.

2.3 Graph Processing

The recent proliferation of graph processing applications, including machine learning [99], recommendation systems [67], and social network analysis [77], has heightened the need for efficiently processing graphs, both in terms of performance and energy consumption. Hence, a number of approaches have been proposed to efficiently process large-scale graphs [36, 61, 64, 70, 118]. The inherent properties of graph analytic applications pose challenges for conventional memory and communications systems, which in turn become performance bottlenecks. First, the operation of traversing neighbourhood vertices shows poor locality due to random memory accesses. Second, many graph algorithms have high memory bandwidth requirements because the node-level computation is relatively simple. Third, when executing in parallel, frequent data movement across the system puts pressure on the communications network. Since the demand for higher memory bandwidth is an important part of accelerating large-scale

graph processing, Near-Memory Processing (NMP) has been proposed to accelerate these tasks. There are several NMP architectures based on multiple memory cubes proposed to improve irregular graph processing applications' performance and energy consumption. Tesseract [3] leverages the large internal bandwidth provided by 16 memory cubes connected in a Dragonfly topology. A single-issue in-order CPU and a prefetcher, serving as computation units, are deployed on the logic layer of each vault. Tesseract adopts Pregel [64] and provides a vertex-centric programming model. The authors report 9× speedups relative to a traditional multicore system using out-of-order cores. While this performance gain is substantial, Dai et al. [24] and Zhang et al. [108] indicate that Tesseract's overall memory bandwidth utilization is less than 40%, implying there are additional performance gains to be had. The reason for this bandwidth utilization limit is cross-cube memory accesses, which Tesseract did not try to optimize.

To reduce cross-cube communications, Zhang et al. [108] proposed a graph partitioning method, called *source-cut*. If two or more cross-edges share the same source vertex but have different destination vertices in a common cube, a replica of the source vertex is placed in the destination cube. Therefore, the data of the source vertex need only be transferred once. To realize the source-cut partitioning method, Zhang et al. proposed a *Two-Phase Vertex Programming* model. The `GenUpdate` phase generates the update for each replica and the `ApplyUpdate` phase updates each replica. Cross-cube communication will only happen before `ApplyUpdate`. To hide the remote latency of cross-cube communication, `GenUpdate` and communication are overlapped asynchronously. A barrier after each phase ensures that hardware cache coherence is not required.

Despite of the promising results of Zhang et al. [108], there are two inherent properties of source-cut partitioning that limits its potential: (1) Since only one cross-cube edge pattern is considered, it only considers a fraction of the cross-cube edges that might potentially be eliminated. (2) It adopts a fixed initial vertex distribution, which limits its options for reducing cross-cube communication overheads. To address these limitations, we introduce a novel software/hardware co-design framework for multi-cube NMP systems, called *SuperCut*, to effectively reduce cross-cube communication overheads while maintaining workload balance.

Besides the comparison baselines of Tesseract and GraphP, there are other NMP architectures designed to accelerate large-scale graph processing. GraphPIM [69] proposes an instruction offloading mechanism to computation units on the logic layer of a single HMC device instead

of a network consisting of multiple cubes. MessageFusion [7] proposes an NMP architecture to reduce cross-cube communication in transit by coalescing multiple cross-cube messages before reaching the same destination vertex. We take inspiration from this technique in our destination-cut static partitioning algorithm. GraphVine [8] explores another way to reduce HMC network congestion at runtime using multicast techniques. Both these works failed to optimize the distribution of vertices, limiting their efficiency. GraphH [24], GraphQ [119] and GraphRing [57] tried to regularize communication overhead by proposing reconfigurable HMC interconnection, a vertex reordering mechanism, and a ring-structured memory network, respectively. None of them directly reduced communication volume or considered graph distribution.

For general distributed graph processing systems, graph partitioning strategies also play a vital role in communication optimization and workload balance, which can be classified [25] into edge-cut and vertex-cut. PowerGraph [36] and PowerLyra [19] adopt vertex-cut to minimize vertex numbers across partitions by assigning edges to replicas in different machines. Although vertex-cut shows good load balance for skewed graphs, it is not suitable for near-memory graph processing because it leads to higher communication cost and requires more complicated implementation mechanisms. Therefore, the partitioning algorithms designed for near-memory graph processing, including the algorithms proposed in this work, are edge-cut [61, 64, 118] in which vertices of the graph are evenly assigned to minimize the number of edges across partitions. Pregel [64] is an early distributed graph processing system which adopts random edge-cut partitioning and provides the message-passing mechanism to deliver updates between machines. Tesseract adopts this approach. The partitioning proposed in GraphP [108] is also an edge-cut method in essence, in which out-going edges across memory cubes are partitioned. The basic principle of destination-cut partitioning as an edge-cut method where edges sharing a destination are combined has been adopted for traditional systems [36, 118].

In this work, we are interested in its effectiveness on near-memory systems, in which the overheads of a cross-cube data transfer are very different than a message-passing send/receive pair.

2.4 Graph Neural Networks

Machine Learning (ML) on graphs has experienced a surge of popularity in the past decade, since traditional ML models, which are designed to process Euclidean data with regular structures, are ineffective at performing prediction tasks on graphs. Due to their simplicity and superior representation learning ability, Graph Neural Networks (GNNs) [29, 49, 90, 103, 107] have achieved impressive performance on various graph learning tasks, such as node classification, graph classification, etc. In this dissertation, we focus on 6 GNN models featuring diverse architectures.

Graph Convolutional Network (GCN) [49] is one of the earliest GNN models and has a simple structure. It updates node features by aggregating neighboring node features and performing linear projection. The formula is given as follows:

$$h_i^{l+1} = \text{ReLU}\left(U^l \sum_{j \in N_i} h_j^l\right) \tag{2.1}$$

Where $U^l \in \mathbb{R}^{d \times d}$ is the learnable weight matrix of the linear projection, which performs vector-matrix multiplication. $h_i^l \in \mathbb{R}^{d \times 1}$ is the feature vector of vertex i in layer l, and N_i represents the neighboring vertices of vertex i.

GraphSage (GS) [38] introduces an inductive framework to improve the scalability over GCN by aggregating information from the fixed-size set of neighbors via uniform sampling, explicitly incorporating feature vectors of both the target vertex and its source neighbors. The mathematical expression of GraphSage with a mean aggregator is formulated as follows:

$$\begin{aligned} h_i^{l+1} &= \text{ReLU}\left(U^l \text{Concat}\left(h_i^l, \frac{1}{|N_i|}\sum_{j \in N_i} h_j^l\right)\right) \\ &= \text{ReLU}\left(V^l h_i^l + W^l \frac{1}{|N_i|}\sum_{j \in N_i} h_j^l\right) \end{aligned} \tag{2.2}$$

Where N_i is the set of source neighbors of vertex i, and $h_i^l \in \mathbb{R}^{d \times 1}$ is the feature vector of vertex i in layer l. The learnable weight matrix of the linear projection, $U^l \in \mathbb{R}^{d \times 2d}$, is stored in on-chip memory. Given that distinct weight parameters are used for the target vertex and

source neighbors, U^l is divided into $V^l \in \mathbb{R}^{d\times d}$ and $W^l \in \mathbb{R}^{d\times d}$, enabling parallel execution of both paths to hide the latency of linear projection for the target vertex.

Graph Isomorphism Network (GIN) [103] employs the Weisfeiler-Lehman Isomorphism Test [96] as its foundation to investigate the discriminative ability of GNNs. The formula of GIN is described as follows:

$$h_i^{l+1} = \text{ReLU}\left(U^l \text{ReLU}\left(V^l\left((1+\epsilon)h_i^l + \sum_{j\in N_i} h_j^l\right)\right)\right) \tag{2.3}$$

where ϵ is a learnable scalar weight, U^l and $V^l \in \mathbb{R}^{d\times d}$ denote learnable weight matrices of cascaded VMM modules, $h_i^l \in \mathbb{R}^{d\times 1}$ again refers to the feature vector of vertex i in layer l, and N_i is again the source neighbors of vertex i.

Graph Attention Network (GAT) [90] is an anisotopic GNN model that uses self-attention mechanisms to weight and learn representations of neighbor vertices unequally. The equation is described as follows:

$$h_i^{l+1} = \text{Concat}_{k=1}^{K}\left(\text{ELU}\left(\sum_{j\in N_i} \alpha_{ij}^{k,l} U^{k,l} h_j^l\right)\right) \tag{2.4}$$

$$\alpha_{ij}^{k,l} = \text{Softmax}(e_{ij}^{k,l}) = \frac{\exp(e_{ij}^{k,l})}{\sum_{j'\in N_i}\exp(e_{ij'}^{k,l})} \tag{2.5}$$

$$\begin{aligned} e_{ij}^{k,l} &= \text{LeakyReLU}(\vec{a}^T Concat(U^{k,l}h_i^l, U^{k,l}h_j^l)) \\ &= \text{LeakyReLU}(a_{src}^{k,l} U^{k,l} h_i^l + a_{dest}^{k,l} U^{k,l} h_j^l) \end{aligned} \tag{2.6}$$

where $\alpha_{ij}^l \in \mathbb{R}^K$ is the attention score between vertex i and vertex j of layer l, $U^{k,l} \in \mathbb{R}^{d\times d}$ and $\vec{a} \in \mathbb{R}^{2d}$ are learnable parameters. Note that the weight parameter $\vec{a}^T$ is decomposed into a_{src}^l and $a_{dest}^l \in \mathbb{R}^d$ in the DGL library, because it is more efficient in terms of performance and memory footprint by transferring VMM between $U^{k,l}$ and h^l from edge-wise to node-wise operations, especially for sparse graphs where the edge number is larger than the vertex number.

Mixture Model Networks (MoNet, MN) [68] is a general anisotopic GNN framework designed for graph and node classification tasks using Bayesian Gaussian Mixture Model (GMM) [27]. The model is formulated as follow:

$$
\begin{aligned}
h_i^{l+1} &= \mathrm{ReLU}\left(\sum_{k=1}^{K}\sum_{j\in N_i} w_k(u_{ij})U^{k,l}h_j^l\right) \\
&= \mathrm{ReLU}\left(\sum_{k=1}^{K} U^{k,l}\sum_{j\in N_i} w_k(u_{ij})h_j^l\right) \qquad (2.7)\\
w_k(u_{ij}) &= \exp\left(-\frac{1}{2}(u_{ij}^l-\mu_k^l)^T(\textstyle\sum_k^l)^{-1}(u_{ij}^l-\mu_k^l)\right) \qquad (2.8)\\
u_{ij}^l &= \mathrm{Tanh}(V^l pseudo_{ij}^l + v^l) \qquad (2.9)\\
pseudo_{ij}^l &= \mathrm{Concat}(deg_i^{-0.5}, deg_j^{0.5}) \qquad (2.10)
\end{aligned}
$$

where $v^l \in \mathbb{R}^2$, $V^l \in \mathbb{R}^{2\times 2}$, $\mu \in \mathbb{R}^{K\times 2}$, $(\sum_k^l)^{-1} \in \mathbb{R}^{K\times 2}$, and $U^l \in \mathbb{R}^{d\times d}$ are learnable parameters of GMM. v^l and V^l represent the pseudo-coordinates between the target vertex and its neighbors, $\mu \in \mathbb{R}^{K\times 2}$ and $(\sum_k^l)^{-1} \in \mathbb{R}^{K\times 2}$ denote the mean vector and covariance matrix. $U^{k,l}$ is the weight matrix.

The Gated Graph ConvNet (GatedGCN, GGCN) [13] is a type of anisotropic graph neural network model that employs a gating mechanism to regulate the flow of information during message passing, allowing the model to emphasize relevant information and filter out irrelevant one. The gating mechanism utilizes gate functions (e.g., sigmoid) to control the flow of messages at each layer. The mathematical expression for GatedGCN is provided below:

$$
\begin{aligned}
h_i^{l+1} &= \mathrm{ReLU}\left(A^l h_i^l + \frac{\sum_{j'\in N_i} B^l h_{j'}^l \odot \sigma(e_{ij'}^{l+1})}{\sum_{j'\in N_i}\sigma(e_{ij'}^{l+1})+\epsilon}\right) \qquad (2.11)\\
e_{ij}^{l+1} &= E^l h_i^l + D^l h_j^l + C^l e_{ij}^l \qquad (2.12)
\end{aligned}
$$

where A^l, B^l, D^l, E^l and $C^l \in \mathbb{R}^{d\times d}$ are learnable matrix parameters, $e_{ij}^l \in \mathbb{R}^{1\times d}$ denote the edge features from vertex i to j layer l, h_i^l represents node features of vertex i in layer l, $\odot$ denotes Hadamard product, σ denotes the sigmoid function, and ϵ is a constant for numerical stability.

2.5 High-Level Synthesis

The basics of a High-Level Synthesis (HLS) workflow, which is adopted by most of the mainstream HLS tools, such as Xilinx Vitis and Intel OpenCL HLS, is illustrated in Figure 2.3. In the dissertation, we use Xilinx Vitis [4] as the running example, but the underlying principles are the same for many other HLS tools. It contains the following steps.

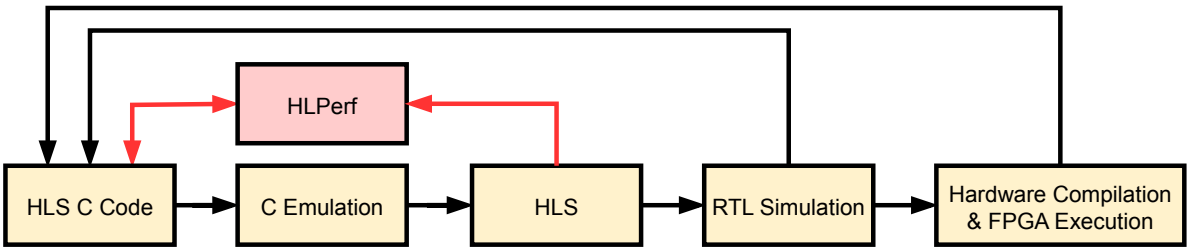


Figure 2.3: Conventional HLS workflow (in black) and our contribution (in red).

(1) **HLS C Code**: HLS users are first required to build HLS kernels based on target algorithms using high-level languages (e.g., HLS C) annotated with pragmas that can have substantial impact on the final performance. Among all the various optimization techniques, `#pragma HLS dataflow` is frequently first used to build a dataflow architecture in which all the functions in the dataflow scope are connected by FIFOs to form a pipeline-style architecture and scheduled to be performed concurrently. In our GNN applications, each function contains one or more loops. Then they use other pragmas such as `Pipeline`, `Loop Unroll`, `Loop Merge`, `Burst Memory Access`, `Memory Port Widening`, etc., to optimize each function or loop in terms of throughput, iteration, and memory accesses. In addition, distinct coding paradigms with the same functionality can also influence the final performance of the HLS kernel.

(2) **C Emulation**: The HLS kernel file, along with the host file, configuration files, and input dataset are compiled and executed under software emulation mode. Note that C emulation only focuses on functionality verification of the HLS kernel. Thus, it doesn't involve performance estimation.

(3) **High-Level Synthesis**: In this step, HLS tools convert the HLS Kernel from the high-level C-like description to an RTL-level hardware description language (e.g., Verilog, VHDL).

To achieve this goal, the HLS tool initially preprocesses the source code of the HLS kernel and conducts transformations based on user-defined pragmas. Subsequently, operations are scheduled in accordance with the corresponding dependency and optimization techniques, and then bound to hardware resources. After this process, coarse-grained control flows are typically implemented as finite-state machines, while fine-grained instruction collections are realized as variants of pipelines. Following these steps, the HLS tool provides estimated intermediate results of the scheduling such as the latency and throughput of pipelines. Finally, RTL code is generated. Note that due to the inherent dynamic characteristics of GNN applications and their significant dependence on input graph datasets, the HLS tool is incapable of providing conclusive outcomes through static performance estimation.

(4) **RTL Simulation**: To obtain estimated cycle-accurate performance results of HLS kernels, the generated RTL code, along with host files, configuration files, and graph datasets, are compiled and executed under hardware emulation mode for RTL simulation. Due to the inherent irregularity of GNNs and graph datasets, the simulation needs to include at least a representative subset of the input graphs to be processed. The results are stored in a waveform file containing cycle-accurate transitions of all the signals in the RTL code. To debug or improve the performance of the HLS kernels, users are required to trace these RTL signals (often with incomprehensible names) back to HLS code, modify the HLS code, and repeat the procedure until the performance goals are satisfied. However, the whole procedure takes a significant amount of time due to the incorporation of extensive architectural details and the desire to be cycle accurate. Consequently, RTL simulation is usually impractical to be used for estimating the performance of complicated GNN HLS kernels with large-scale graph datasets.

(5) **Hardware Compilation & FPGA Execution**: To get the physical layout, the generated RTL code is converted into a gate-level representation (i.e., netlist) for a specific architecture and then mapped to specific locations of the target device via the place & route process. A series of back-end strategies on physical implementations are performed to get a trade-off among design's performance, area, and power. The finalized circuit description is encapsulated into a bitstream file, which is then executed on an FPGA, enabling the measurement of the actual execution times. This step is quite time consuming (e.g., 4.5-12 hours for compilation of GNN HLS kernels), and even though direct execution is clearly the gold standard for performance understanding, it is the length of these build times that makes the inclusion of direct execution in the iterative design cycle unattractive.

Although HLS bridges the gap between software and hardware development, optimizing HLS codes is substantially distinct from conventional software programming. In fact, due to the FPGAs' inherent attributes, such as lack of built-in cache mechanisms, low clock frequency (relative to traditional processor cores), and fine-grained configurability, the performance difference between a well-optimized version and naive version of the same kernel can be two or three orders of magnitude [15, 32, 85]. Therefore, to achieve the best performance, HLS developers need to explore a large optimization space for HLS designs with various optimization pragmas, coding paradigms, etc. As the complexity of kernels increases, optimizing (or auto-optimizing) such kernels is difficult via conventional HLS workflows.

To address the performance evaluation challenge, several works have been proposed, which can be classified into 2 main classes: static estimation [18, 23, 26, 63] and cycle-accurate simulation [1, 22, 79]. Static estimation is performed at compile time so has difficulty with performance that is input dependent, and cycle-accurate simulation substantially accelerates the speed of RTL-level simulation while maintaining quality of the performance predictions. We will report research happening in each area in turn.

Legup [18] estimates the speedup of the accelerated function in the HLS kernel in the straightforward way of multiplying the number of iterations recorded by software profiling tools and the single-iteration execution time extracted from RTL simulation. This method does depend on RTL simulation. Additionally, it assumes the FPGA algorithms are performed sequentially, so it doesn't consider a number of HLS optimization techniques, such as pipelined execution. HLScope+ [23] proposes a method to perform pipelined loop analysis by inserting hooks to HLS C code and extracting HLS abstraction information. However, it fails to capture the irregularity of data sets (e.g., graph topologies) and respond to the dynamic properties of irregular algorithms and HLS kernels. Pyramid [63] uses machine learning techniques to estimate both FPGA area requirements and achievable clock rates. De Fine Licht et al. [26] propose a static expression for pipelined loop analysis with some optimization techniques. However, it is also not sufficient for irregular data and algorithms.

On the other hand, Flash [22] uses scheduling information to build a C cycle-accurate simulation model. FastSim [1] translates generated RTL code to an equivalent C++ cycle-accurate model. LightningSim [79] proposes a LLVM-IR-trace-based method to reconstruct a cycle-accurate model. In spite of the ability of these cycle-accurate methods to analyze the dynamic behavior of many FPGA algorithms, there are still some drawbacks: (1) These works

are designed to provide both functional correctness verification and performance estimation, increasing the workload of the evaluation process. (2) Since these methods are related to the construction and execution of cycle-accurate time models with many low-level details, the simulation speed is limited.

Design space search and custom architectures aimed specifically at GNNs have also received attention recently [35, 39, 58, 59, 104, 109, 116, 117]. HGNAS [116, 117] targets edge devices for execution of GNNs, explicitly considering reduction in memory requirements as well as execution speed. DeepBurning-GL [58] proposes an automated framework to convert specific component of GNN models based on DGL to RTL codes using pre-defined hardware templates. G-CoS [109] works to match GNN structure with the available execution platform(s), and Hao et al. [39] exploit reinforcement learning as part of the design space search. HyGCN [104] is a custom ASIC design aimed at graph neural network inference that is evaluated using the TSMC 12 nm CMOS process. EnGN [59] targets the need to scale up to large graphs by introducing a ring-edge-reduce dataflow to handle graphs with arbitrary dimensions. AWB-GCN [35] is a custom FPGA design that addresses the variability in graph topology by auto-tuning the accelerator during the execution of the GNN application itself. While the authors of all of the above studies evaluate performance on a variety of graphs, none of the design space exploration investigations incorporated simulation into the performance evaluation that specifically guides the search of the design space, and only AWB-GCN has explicit mechanisms for adapting to variations in properties of the input graphs. Our intention is to make simulation sufficiently fast that it can be seriously considered in an automated design space search context.

GNN models are but one example of applications that execute on graphs. Chen et al. [20] introduce ThunderGP, a framework for developing general graph applications for deployment on FPGAs. ThunderGP uses a dataflow architecture for its designs, so a GNN model developed using ThunderGP could likely utilize HLPerf as a companion performance evaluation tool.

We are interested in the effectiveness of modern HLS tools to accelerate GNN inference, and the availability of fast, high-quality understanding of the performance implications of the design choices that are made in the HLS optimization task.

Chapter 3

Executing Data Integration Near the Memory

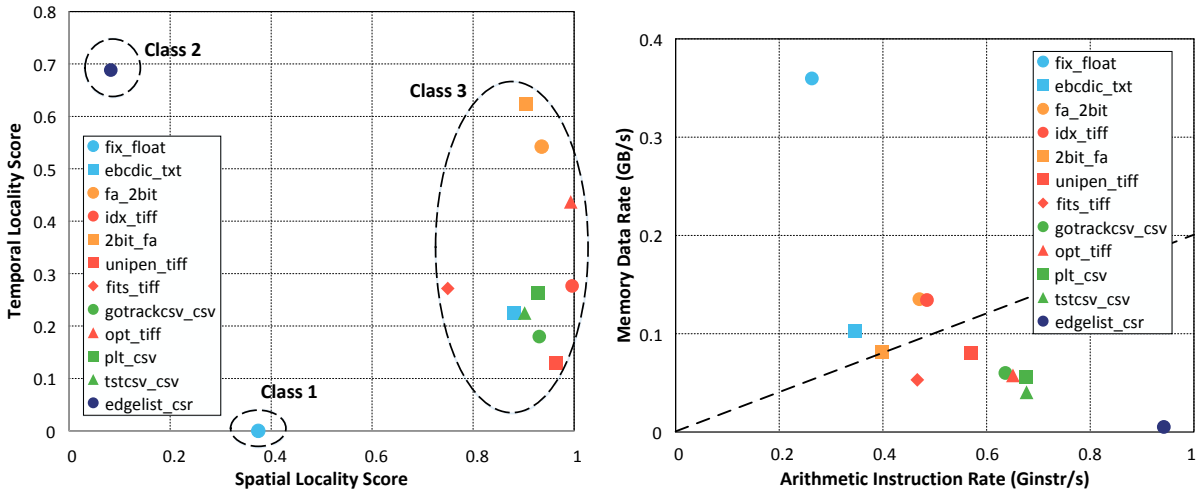
According to profiling results presented in the data integration benchmark suite (DIBS) [17], the fraction of data movement instructions are more than 50% in 8 of 12 data integration workloads, implying a strong sensitivity to the architecture of the memory subsystem and opportunities to reduce expensive data movement with near-memory processing techniques. Meanwhile, since data integration workloads are embarrassingly parallel, targeting at every independent data individual in the stream, data parallelism provided by NMP architecture is potentially beneficial as performance scales with large numbers of NMP cores. Here, we propose the question: *To what extent does NMP benefit data integration applications?*

3.1 Workload Characterization

To answer the question, We characterize these DIBS workloads using four metrics: temporal locality, spatial locality, memory access rate, and arithmetic instruction rate. These workloads have a high degree of data movement, motivating the emphasis on memory in the application characterization.

Temporal and spatial locality are quantified using the techniques proposed by Weinberg et al. [95]. Each locality score is on a normalized range [0,1], with higher scores indicating a greater degree of locality.

Figure 3.1(a) shows the temporal and spatial locality of each of the DIBS applications. We classify the applications into 3 classes: (1) low spatial and low temporal locality, (2) low spatial and high temporal locality, and (3) high spatial locality.



(a) (b)

Figure 3.1: Workload characterization of data integration workloads. (a) Temporal and spatial locality scores. (b) Memory data rate vs. arithmetic instruction rate.

The `fix_float` application, with relatively low spatial locality and lowest temporal locality belongs to class 1. Others have shown that this type of workload can benefit from near-memory processing techniques, e.g., see [83].

The `edgelist_csr` application, with low spatial locality and high temporal locality, belongs to class 2. In this case, the high temporal locality implies that a deep cache hierarchy can benefit performance, so it might not do as well on a near-memory architecture.

The remaining 10 out of the 12 applications, belonging to class 3 with high spatial locality, lie at the right of Figure 3.1(a). These locality scores give mixed signals as to the suitability of these workloads for near-memory processing techniques.

We measure the memory accesses and arithmetic instructions to further characterize our data integration workloads. In addition, we will discuss this characterization in terms of their ratio, which we call the *memory/ops* ratio.

Figure 3.1(b) shows the memory data rate and arithmetic instruction rate in the two-dimensional space. From the figure, it can be observed that `fix_float` has the highest memory/ops ratio, indicating that this workload is the most promising one that could benefit from a near-memory processing architecture.

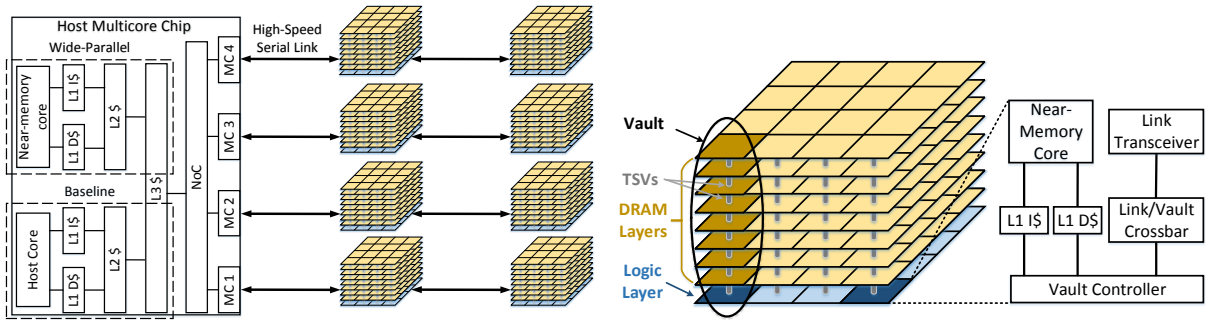
Consistent with the locality characterization of Figure 3.1(a), 10 workloads with relatively high spatial locality scores also show medium memory/ops ratios. Among these 10 workloads, `ebcdic_txt`, `fa_2bit`, and `idx_tiff` have higher memory/ops ratios.

The `edgelist_csr` application is the most computationally intensive workload, making it an outlier in both characterizations (a fact it has in common with `fix_float`).

To distinguish the workloads by the potential benefits provided by near-memory processing, we draw a dashed line in Figure 3.1(b). The workloads above the line are more memory intensive and have the greater chance for performance improvement. While the particular slope of the line is arbitrary, we return to this point in Section 3.4.

3.2 Proposed Near-Memory System

The high-level architecture of our near-memory processing system is illustrated in Figure 3.2. As previously proposed by Pugsley et al. [74], a number of memory channels (four in Figure 3.2(a)) are used to connect the host multicore chip to a set of 3-D memory stacks (eight in the figure), using the "far memory" topology described by Micron for HMC. In the performance analysis that follows, we assume that the host multicore chip contains 16 traditional, out-of-order processor cores with private L1 I&D caches, private L2 caches, and a shared L3 last-level cache (for the time being, ignore the dashed wide-parallel box in the figure).



(a) (b)

Figure 3.2: Architecture of near-memory processing system. (a) Stacked memory connection diagram. (b) Near-memory processor diagram in each vault.

Figure 3.2(b) shows the design of the near-memory processing subsystem. Associated with each vault is a low-power, in-order core that has immediate access to the memory within the vault, talking directly to the vault controller. This in-order core is fabricated on the logic layer chip, using previously empty chip area. It has L1 I&D caches, but no L2 or L3 cache. This arrangement yields 16 near-memory cores for each memory stack, resulting in 128 near-memory cores for a system with 8 memory stacks. The link/vault crossbar and vault controller are unchanged from a typical HMC-like memory stack.

If we are to utilize otherwise empty chip area on the logic layer, we must ensure that not only area, but power constraints are not exceeded. Jeddeloh and Keeth [43] report 68 mm^2 of area on the logic layer, or 4.3 mm^2 per vault. Using the Arm Cortex-A7 as a candidate near-memory processor core, the core plus 32 KB L1 I&D caches have an area of 0.45 mm^2 and a total power consumption below 100 mW.[3] This represents a usage of only about 10% of the available area.

A traditional HMC stack has 4 memory channels that consume a total of 5.8 W [74]. However, the far memory topology illustrated in Figure 3.2(a) only utilizes 2 channels per memory stack. If the remaining 2 channels are powered off (or not fabricated at all), this makes 2.9 W available for the near-memory cores. 16 cores at 100 mW each only requires 1.6 W, which is well within the power budget.

The advantages of this architecture for executing data integration workloads are two-fold. First, because the workloads are straightforward to parallelize, a larger number of smaller cores are well matched to the computational requirements. Using the same reasoning as the designers of the IBM BlueGene family of supercomputers, a larger number of smaller cores can yield both performance benefits and energy savings if the problem has sufficient parallelism. This is the same notion that motivated the big.LITTLE systems from Arm. Second, because the memory access patterns of the workloads are primarily local, associating a near-memory core with each memory vault results in the bulk of memory accesses being local. Not only do the accesses not have to traverse the memory channel(s) across the circuit board, but most don't even have to traverse the internal crossbar of the memory stack.

To execute the data integration workloads, the data are divided into smaller partitions and mapped to each computation unit by the host processors. As each partition is independent,

[3] https://developer.arm.com/ip-products/processors/cortex-a/cortex-a7

they can be processed by the near-memory cores in parallel. After the data integration computations are complete, the transformed data will be aggregated and processed by the host processors. This allows the downstream application processing to take advantage of the complex cache system and high computation ability available on the host side.

3.3 Methodology

We evaluate the performance and energy consumption of data integration workloads via architectural simulation (using the `gem5` simulator). First, we describe three distinct architectures that we use to perform the evaluations: (1) the near-memory processing system that is the primary target of the investigation, (2) a traditional, out-of-order core system used as the baseline/host for comparison purposes, and (3) a wide-parallel system that utilizes low-power, in-order cores which serves as an intermediate system between the baseline/host system and the near-memory processing system. It is used to distinguish between various factors in the performance assessment. After this, we describe additional details of the simulation, including the methods for performance calibration and energy estimation.

3.3.1 Near-Memory Processing System

The structure of the architectural simulation model for the near-memory processing system essentially follows Figure 3.2. The particulars of the host processor cores and memory stacks are described below in Section 3.3.2. Here, we provide the particulars of the near-memory processors.

As alluded to in the previous section, we use an Arm Cortex-A7 in-order, pipelined core as the processor model for the near-memory computation. Each core is clocked at 1.2 GHz and has 32 KB L1 I&D caches, and each vault of the memory stack is allocated one near-memory core. With 8 memory stacks and 16 vaults per stack, there are 128 near-memory cores.

The close association of each near-memory core with a particular vault of the memory stack implies a non-uniform memory access (NUMA) latency. Accesses to remote vaults on the same stack must traverse the on-logic-chip crossbar, and accesses to vaults on remote stacks must access the topological path(s) shown in Figure 3.2(a). On the other hand, accesses to

the local vault need only talk to the local vault controller. Each vault has a 32-bit vertical interface with 2 Gb/s TSV signaling rate [43]. Thus, an internal bandwidth of 8 GB/s can be achieved in each vault, and overall memory bandwidth is 1 TB/s.

We do not assume coherent caches across the near-memory cores, but rather insert explicit cache flush instructions to enforce memory consistency.

3.3.2 Baseline/Host System

The baseline/host processors have a common design, but serve two distinct purposes in our evaluations. First, they serve as the baseline for both performance and energy comparison purposes. As such, the quantitative evaluation is normalized to the baseline system's performance and energy usage. Figure 3.2(a) illustrates this baseline system if one assumes there are no near-memory cores in the logic layer of the memories. Second, they serve as the host processor(s) for the near-memory processing system. In this circumstance, they are available to execute other tasks concurrently with the near-memory subsystem.

To maintain a common ISA across the entire system, we employ 16 Cortex-A15 out-of-order cores as the processor model for the baseline/host computation. Each core is clocked at 2 GHz and has its own 32 KB L1 I&D caches, 256 KB L2 cache, and shared L3 8 MB last-level cache. Cache coherence is maintained using the traditional MESI-style protocol.

The baseline/host multicore chip has 4 memory channels, each modeled after HMC stacked memory, giving a total memory bandwidth of 160 GB/s. With 8 memory stacks of 4 GB each, the memory capacity is 32 GB. This is straightforward to alter given the flexibility of the "far memory" topology. The bandwidth to/from the baseline/host cores, however, is limited by the number of memory channels available on the baseline/host.

3.3.3 Wide-Parallel System

When comparing the proposed near-memory processing system with the traditional baseline system, there are two substantial differences that can (and should) be evaluated separately. One, a larger number of simple cores are utilized in place of a smaller number of complex cores, and two, each simple core has lower-latency, higher-bandwidth access to (a subset of)

the memory. To give us the ability to query the performance and energy usage implications of each of these two features separately, we consider an intermediate, wide-parallel system that only includes the first of the above two substantial differences from the baseline system.

The wide-parallel system has the same cache and memory configurations as the aforementioned host/baseline system. However, this system replaces the 16 Cortex-A15 cores with 128 Cortex-A7 cores, the same type and number of cores used in the near-memory system. Figure 3.2(a) illustrates this wide-parallel system via the dashed box showing a near-memory core.

Fortunately, the two substantial differences between the near-memory processing architecture and the baseline traditional architecture also correspond closely to the two primary characteristics that are common across the target data integration workloads. The abundant parallelism in the workloads can benefit from the larger number of simple cores, and the substantial (local) data movement can benefit from the positioning of those cores close to memory.

By comparing the baseline system to the wide-parallel system, we can discern the impact and importance of the parallelism in both the workloads and the execution architecture. By comparing the wide-parallel system to the near-memory system, we can discern the impact and importance of the memory bandwidth and latency. Finally, we can see the overall impact of the near-memory system by comparing it to the baseline system.

The parameters of all three of these systems are shown in Table 3.1.

3.3.4 Simulation

All of our simulation models are built in `gem5`. The standard distribution contains a stacked memory model based upon HMC and processor core models for both the Cortex-A15 cores (`ex5_big.py`) and Cortex-A7 cores (`ex5_LITTLE.py`).

The data integration workloads come from DIBS [17], which provides both source code and input data sets. All are compiled with `gcc` version 5.4.0 utilizing `-O3` optimizations. To enforce cache coherence between the host caches and the near-memory caches, we extended the core model to support cache flushing. In the cache flushing API, the corresponding cache

Table 3.1: Evaluated system configurations.

Baseline/Host System	
Core configuration	Arm Cortex-A15, out-of-order, 2 GHz
Number of cores	16
L1 cache	32 KB I&D private
L2 cache	256 KB private
L3 cache	8 MB shared
Memory	32 GB
Wide-Parallel System	
Core configuration	Arm Cortex-A7, in-order, 1.2 GHz
Number of cores	128
Caches & memory	same as baseline
Near-Memory System	
Host cores & caches	same as baseline
Near-memory cores	Arm Cortex-A7, in-order, 1.2 GHz
Number of cores	128
L1 cache	32 KB I&D private
Memory	32 GB

block is determined based on the physical address which is re-translated from the virtual address. If dirty, the cache block is flushed to the memory stack.

The energy model computes the energy due to dynamic power consumption by summing the contributions from the following elements: cores, caches, NoCs, memory channel transceivers, logic layer of the memory stack (without the near-memory processing elements), and the memory stack DRAM layers. Processor core energy (both for Cortex-A7 and Cortex-A15) is computed using energy per instruction measurements provided by [89] and instruction counts from `gem5`. Cache and NoC energy is modeled using McPAT assuming a 28 nm process node.

The memory subsystem is based upon an HMC model, using energy data provided by [43]. The total memory stack energy is 10.38 pJ/bit accessed. Of this, the DRAM layers consume 3.7 pJ/bit and the logic layer consumes 6.78 pJ/bit (of which, 43% is consumed by the transceivers) [74].

3.4 Evaluation

In this section, we examine how data integration workloads benefit from the near-memory processing system described above. We compare both performance and energy consumption for the baseline, wide-parallel, and near-memory target systems.

3.4.1 Performance Improvement

We first examine performance improvement by showing speedup relative to the baseline in Figure 3.3. The applications to the left of `2bit_fa` on the graph are above the dashed line in Figure 3.1(b) and those to the right of `2bit_fa` are below the line.

Examining the middle bar for each application (the speedup of the wide-parallel system), we observe that all of applications improve. While the geometric mean speedup (for all applications) is 3.04× and the minimum speedup is 2.48×, 5 of the applications exceed 3.4× speedup. We can conclude that the abundant parallelism in the data integration workloads can effectively be exploited by a larger number of individually less-powerful cores, and that the effectiveness of this approach is strong for all of the applications.

Note that while this wide-parallel system is used to assess the degree to which parallel execution can benefit data integration applications, it does not represent a realistically viable system in any practical sense for general workloads. At this scale, cache coherence overheads are often dominant, a fact that isn't an issue here simply because the data integration applications are, in effect, embarrassingly parallel, so they generate minimal coherence traffic.

The right-most bar for each application indicates the speedup for the target near-data processing architecture relative to the baseline architecture. During this execution, the host cores are essentially idle (only responsible for startup and termination) and therefore available to execute other applications such as the data analysis task that is downstream of data integration in the workflow of interest.

Again, the overall results reflect significant performance improvement. The geometric mean speedup is 3.46×, and the individual application speedup ranges from 2.57× up to 5.82×. The 4 applications with the largest memory/ops ratio exhibit the greatest performance

improvement relative to the wide-parallel system, each improving an additional 1.21× or more relative to the wide-parallel system. In other words, they benefit from the cores' close proximity to memory and are not hurt by the lack of L2 and L3 caches associated with each near-memory core.

The outlier here is `edgelist_csr`. Its performance is slightly worse (0.99×) when transitioning from the wide-parallel system to the near-memory system, mostly due to the lack of L2 and L3 caches for the near-memory cores. This however is not surprising and predicted by its outlier position in Figure 3.1(a).

Across the board, we see fairly good performance gains for the near-memory processing architecture by leveraging highly parallel near-memory computing units and accounting for the impact of small memory-side caches. It is also worth noting that a large fraction of this performance gain is attributable to the benefits of parallelism and a smaller fraction due to the benefits of the processor cores' physical proximity to the memory.

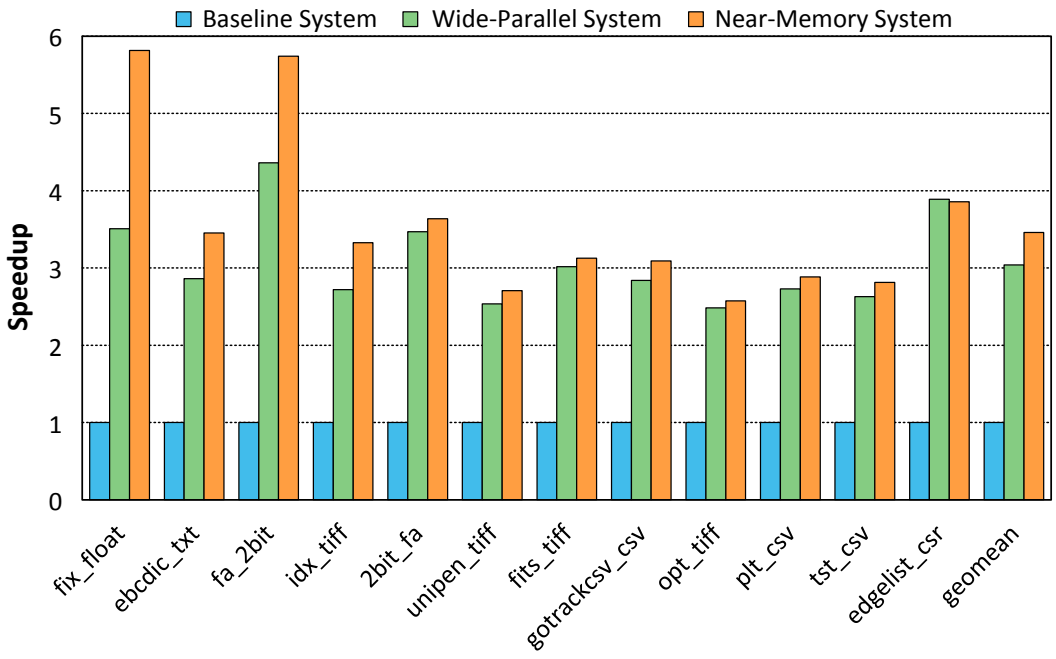


Figure 3.3: Speedup of data integration workloads for wide-parallel and near-memory system relative to baseline.

We next return to the point that, in the near-memory system, the host cores are essentially idle while the data integration application is executing, freeing them up for other tasks. For example, when executing the initial data analysis algorithm described by Malicevic et al. [65],

three-quarters of the execution time is consumed by pre-processing (i.e,. data integration) and one-quarter of the execution time is consumed by the algorithm (i.e., data analysis). If the data integration is accelerated by a factor of 3 (less than the geometric mean of our benchmark applications), the overall memory bandwidth is underutilized (TSV utilization is no more than 5% across the board), and the data analysis is executed concurrently with data integration (in a pipelined manner), then the overall performance gain is a factor of 4. In other words, the entire execution time of the data integration is overlapped by the data analysis time. Given that the near-data cores were integrated onto otherwise unused chip area on the logic layer of the stacked memory, this is almost "data integration for free."

3.4.2 Energy Consumption

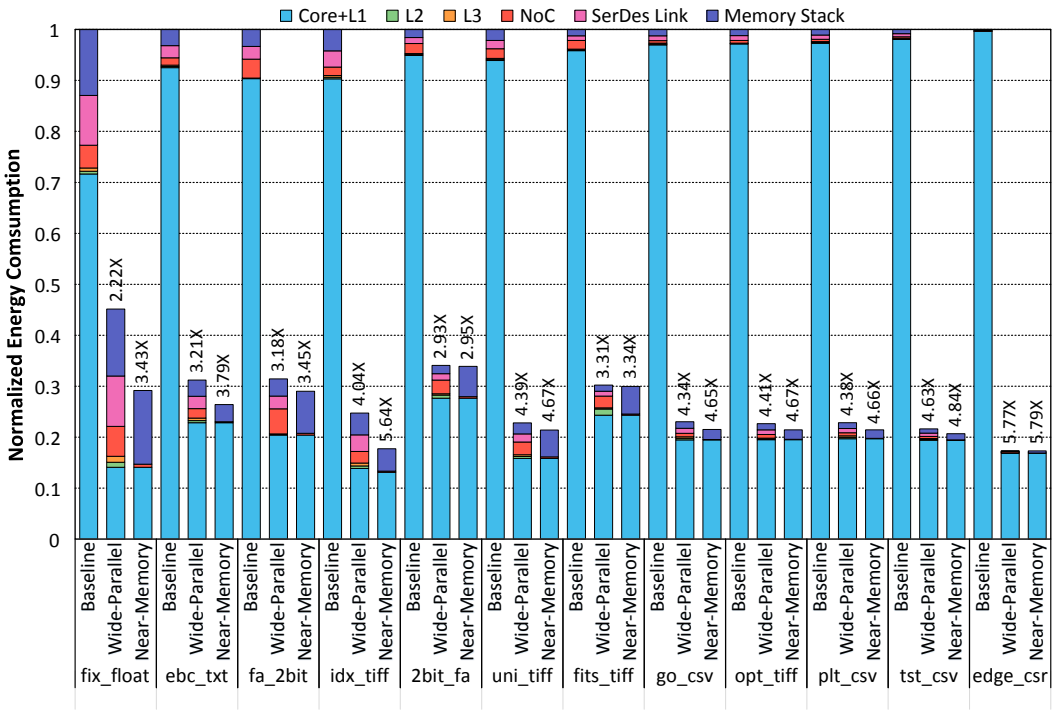


Figure 3.4: Energy consumption of data integration workloads obtained from the baseline, wide-parallel and near-memory system, normalized to the baseline.

We next quantitatively examine the energy consumption benefits of the near-memory architecture. Figure 3.4 shows the energy savings for the wide-parallel and the near-memory architectures. The relative energy improvement (number above each bar) is computed by

Table 3.2: Energy consumption of data integration workloads, expressed as a fraction of total energy consumption.

		Core+L1	L2	L3	NoC	SerDes Link	Memory Stack
fix_float	Baseline	71.6%	0.6%	0.6%	4.4%	9.8%	13.0%
	Wide-Parallel	31.2%	2.2%	2.5%	13.0%	22.0%	29.1%
	Near-Memory	48.3%	0.0%	0.0%	2.0%	0.0%	49.7%
ebc_txt	Baseline	92.5%	0.2%	0.3%	1.4%	2.4%	3.2%
	Wide-Parallel	73.2%	1.3%	1.5%	6.0%	7.7%	10.2%
	Near-Memory	86.5%	0.0%	0.0%	0.7%	0.0%	12.7%
fa_2bit	Baseline	90.3%	0.1%	0.0%	3.7%	2.5%	3.3%
	Wide-Parallel	65.0%	0.5%	0.2%	15.7%	8.0%	10.6%
	Near-Memory	70.4%	0.0%	0.0%	1.2%	0.0%	28.4%
idx_tiff	Baseline	90.3%	0.3%	0.4%	1.7%	3.2%	4.2%
	Wide-Parallel	56.0%	2.0%	2.4%	9.1%	13.1%	17.4%
	Near-Memory	73.9%	0.0%	0.0%	1.4%	0.0%	24.7%
2bit_fa	Baseline	94.9%	0.2%	0.1%	2.0%	1.2%	1.6%
	Wide-Parallel	81.1%	1.8%	0.8%	7.9%	3.6%	4.8%
	Near-Memory	81.5%	0.0%	0.0%	0.9%	0.0%	17.6%
uni_tiff	Baseline	93.9%	0.2%	0.2%	1.9%	1.6%	2.1%
	Wide-Parallel	69.5%	1.5%	1.6%	10.9%	7.1%	9.4%
	Near-Memory	74.0%	0.0%	0.0%	1.3%	0.0%	24.7%
fits_tiff	Baseline	95.8%	0.2%	0.1%	1.7%	0.9%	1.2%
	Wide-Parallel	80.4%	4.1%	0.7%	7.6%	3.1%	4.1%
	Near-Memory	81.1%	0.0%	0.0%	0.8%	0.0%	18.1%
go_csv	Baseline	97.0%	0.2%	0.2%	0.5%	0.9%	1.2%
	Wide-Parallel	84.6%	1.2%	1.4%	3.0%	4.2%	5.6%
	Near-Memory	90.5%	0.0%	0.0%	0.4%	0.0%	9.1%
opt_tiff	Baseline	97.2%	0.1%	0.1%	0.5%	0.9%	1.2%
	Wide-Parallel	86.1%	0.6%	0.7%	3.1%	4.1%	5.4%
	Near-Memory	91.1%	0.0%	0.0%	0.3%	0.0%	8.6%
plt_csv	Baseline	97.3%	0.1%	0.2%	0.5%	0.8%	1.1%
	Wide-Parallel	86.2%	1.1%	1.3%	2.7%	3.7%	5.0%
	Near-Memory	91.8%	0.0%	0.0%	0.4%	0.0%	7.8%
tst_csv	Baseline	98.1%	0.1%	0.1%	0.3%	0.6%	0.8%
	Wide-Parallel	89.7%	0.7%	0.8%	2.0%	2.9%	3.9%
	Near-Memory	93.9%	0.0%	0.0%	0.3%	0.0%	5.9%
edge_csr	Baseline	99.7%	0.0%	0.0%	0.2%	0.1%	0.1%
	Wide-Parallel	97.4%	0.2%	0.1%	1.4%	0.3%	0.5%
	Near-Memory	97.6%	0.0%	0.0%	0.1%	0.0%	2.3%

dividing the baseline system energy consumption by the wide-parallel system energy or the near-memory processing system energy, respectively.

Focusing first on the comparison between the baseline system and the near-memory target system, we see that the overall energy reduction is quite significant, with a geometric mean of 4.23×. Even the application with the least benefit, `2bit_fa`, requires just over one-third of the energy of the baseline system, for an energy savings of almost 3× when executed on the near-memory system.

Turning our attention to how that energy savings is attributable to the wide-parallel aspects of the system versus the processing proximity to the memory, we once again observe an important relationship between applications with a high memory/ops ratio and energy reduction attributable to the physical proximity of the memory. The four applications that sit above the dashed line in Figure 3.1(b) all have less energy savings for the intermediate wide-parallel system with improved energy savings when transitioning to the near-memory target system. The remaining applications show substantial energy savings, with the majority of that savings being attributable to the wide-parallel nature of the applications' execution.

We can discern why this is the case by examining the energy breakdown, shown in Table 3.2. The table decomposes the energy consumption into 6 categories: cores plus L1 caches, L2 caches, L3 caches, NoC, transceivers for the memory channels, and memory stack (including both vault controllers and the DRAM chips). They are indicated as a percentage relative to the total energy.

The energy that is saved by moving from the wide-parallel design to the near-memory design is primarily energy attributed to L2 and L3 caches and memory channel transceivers. This is largest in the applications with the highest memory/ops ratio, and is much smaller in those applications with a lower memory/ops ratio.

A final observation is that, across the board, all of the applications' energy consumption is dominated by core+L1 energy, a fact that is true for all three architectures we consider. This points to a potential approach (left for future work) for executing data integration workloads that exploits alternative computational platforms, such as reconfigurable logic.

3.5 Conclusion

Data integration is an important yet not well-explored bottleneck for data analysis flows. In this paper, we characterize data integration workloads based on localities and memory/arithmetic operation intensity. Our characterization reveals that most of data integration workloads have regular memory access patterns and varying computation intensity.

We find that a near-memory processing architecture can benefit data integration workloads both in terms of performance and energy consumption. Our proposed near-memory system outperforms the baseline/host system with 16 Arm Cortex-A15 cores, exhibiting an average 3.5× speedup and 4.2× energy efficiency improvement, by utilizing its highly parallel 128 Arm Cortex-A7 cores inside the stacked memory logic layer. In addition, by comparing the baseline system and near-memory system with an intermediate wide-parallel system, we are able to attribute benefits separately to the availability of abundant parallelism and memory proximity. While all of the applications benefit from wide-parallel execution, the benefits of memory proximity are more concentrated on applications that have a high memory/ops ratio.

We conclude that near-memory processing is a promising strategy to improve the performance and reduce energy consumption for data integration workloads.

Chapter 4

Partitioning for Near-Memory Graph Processing

The proliferation of graph processing applications, including machine learning [99], recommendation systems [92], social network analysis [87] and bioinformatics [98], has heightened the need for efficiently processing graphs, both in terms of performance and energy consumption. Hence, a number of graph analytic works for parallel computing on various systems have been proposed to efficiently process large-scale graphs [6, 21, 36, 37, 61, 62, 64, 70, 118]. However, the inherent properties of graph analytic applications pose challenges for conventional memory and communications systems, which in turn become performance bottlenecks. First, the operation of traversing neighbourhood vertices shows poor locality due to random memory accesses. Second, most graph algorithms have high memory bandwidth requirements because the node-level computation is relatively simple [10]. Third, unlike data integration workloads, when graph applications are executed in parallel, frequent data movement across the system puts pressure on the communications network. Since higher memory bandwidth is an important part of improving the performance of large-scale graph processing, near memory processing (NMP) has been proposed to accelerate these tasks.

Tesseract [3] proposes an NMP architecture for parallel graph processing with 16 cubes. While providing substantial performance gains over conventional DRAM-based architectures, its performance is ultimately limited by cross-cube communications. To alleviate such communication overhead, prior works [3, 108] tried METIS [46] to execute graph partitioning. However, the results were not promising. It was reported that there are several factors limiting the performance of METIS on such cases: (1) it leads to substantial variance between maximum and average communication; (2) it exacerbates intra-cube computational balance.

Subsequent works have proposed to diminish communication bottlenecks by alternative preprocessing of the graph [108] or by run-time adaptations [7, 8]. GraphP [108] proposes

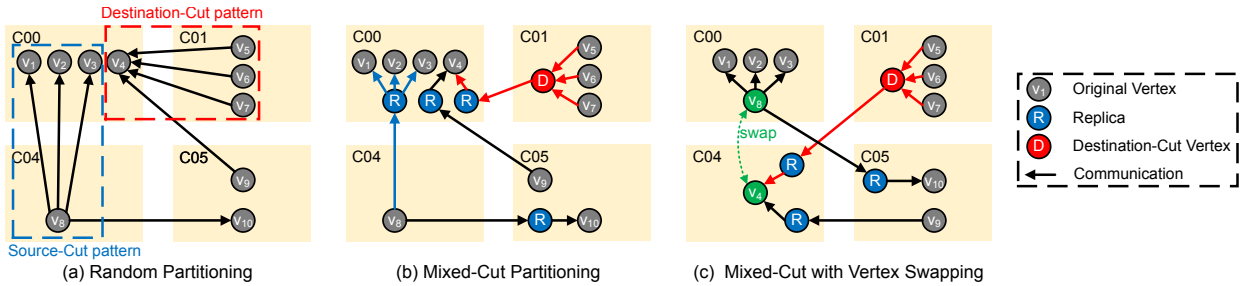


Figure 4.1: (a) A synthetic graph as an example to illustrate source-cut and destination-cut patterns. (b) The mixed-cut partitioned graph. (c) The mixed-cut example graph after swapping v_4 and v_8.

source-cut partitioning, in which replicas of the source vertex of each cross-cube edge are deployed in destination cubes, so that multiple cross-cube edges sent from a common source vertex to the same destination cube can be reduced to one. Therefore, lower cross-cube communication volume is required relative to Tesseract. Despite the promising results from source-cut partitioning, there is still room for improvement. We observe that after performing source-cut partitioning, cross-cube communication still takes a significant portion of execution time (12%-78%) and energy consumption (14%-73%). This invites the open research question: *To what extent can NMP benefit graph processing applications?* More specifically, *how effectively can partitioning algorithms reduce communications overheads while maintaining computational balance in an NMP system?*

4.1 SuperCut Framework

Here, we describe SuperCut, our co-design framework for near-memory graph processing. First, the graph dataset is pre-processed by graph partitioning algorithms (Sec. 4.1.1 to 4.1.3). Then the three-phase programming model (Sec. 4.1.4) is built with user-defined functions to express the graph processing applications. Next, NMP accelerators are generated via HLS (Sec. 4.1.5). The partitioned graph is stored using the custom graph representation (Sec. 4.1.6). Both the graph and the accelerator design is fed into the NMP simulation.

We repeatedly refer to Figure 4.1 to illustrate several points related to the graph partitioning. Figure 4.1(a) shows an example of a small synthetic graph. This graph has 8 cross-cube edges, each of which initially represents one cross-cube communication.

4.1.1 Mixed-Cut Partitioning

Our initial partitioning algorithm is called *mixed-cut*, which is a combination of source-cut partitioning and destination-cut partitioning. The source-cut pattern (described by Zhang et al. [108]) is illustrated by the blue dashed rectangle in Figure 4.1(a). After the transformation, the revised graph is shown in blue in Figure 4.1(b).

For cross-cube updates with a common destination vertex, MessageFusion [7] dynamically merges these updates at the source cube before transferring them to the destination cube. Inspired by MessageFusion, we propose a static graph partitioning method, called *destination-cut*, to reduce cross-cube edges exhibiting this same pattern (multiple cross-cube edges which have the same destination vertex and distinct source vertices, all from the same cube). Figure 4.1(a) illustrates an example of the cross-cube edge pattern of destination-cut partitioning (marked in red). In this example, there are three cross-cube edges: $v_5 \rightarrow v_4$, $v_6 \rightarrow v_4$, and $v_7 \rightarrow v_4$ which have the same destination vertex, v_4.

In mixed-cut partitioning, we first implement source-cut partitioning as the initial partitioning method and then implement destination-cut partitioning as the secondary method. Figure 4.1(b) illustrates the results of mixed-cut partitioning on the example graph. Here, 6 out of 8 original cross-cube edges (75%) have been reduced, which is higher than source-cut partitioning alone (37.5%).

4.1.2 Vertex-Swapping Greedy Algorithm

The partitioning algorithms discussed so far do not consider moving vertices between cubes. The next element of SuperCut partitioning is a stochastic, greedy optimization algorithm that explicitly moves vertices across cube boundaries.

Inspired by the iterative placement algorithms of IC physical design [45], which continuously modify the placement of circuits by exchanging randomly-selected cells, we propose a greedy algorithm to diminish the cost of communication while maintaining workload balance across the cubes. In order to implement a greedy algorithm, a cost function is needed.

The goal of the cost function is to capture both the execution time and energy consumption of cross-cube communication, described as follows:

$$cost(G) = \alpha_1 \times \max(cost_{comm}) + \alpha_2 \times \text{mean}(cost_{comm})$$

where $cost_{comm}$ is the cube-level communication cost in graph G calculated by multiplying the number of cross-cube edges and the number of SerDes links between each cube pair, and $\max(cost_{comm})$ is the maximum communication cost among all the cube pairs while $\text{mean}(cost_{comm})$ is the average communication cost among all the cube pairs. The first term represents the worst-case runtime of cross-cube data transfer and the latter term represents the aggregated energy consumption of cross-cube communications. Parameters α_1 and α_2 are adjusted to balance these two different goals into a single metric, which are set on the basis of the significance of performance and energy as required in different scenarios.

In order to avoid introducing workload imbalance, we implement a vertex-swapping strategy as the perturbation function in the greedy algorithm, shown in Algorithm 1. In this algorithm, G represents the original graph before the swapping operation, H represents the mixed-cut graph of the original graph before the swapping operation, and graph names with suffix $'$ represent graphs after the swapping operation (e.g., G' represents the original graph after the swapping operation) Initially, all the vertices are mapped into cubes using a hash function to get an initial vertex distribution (line 1), e.g., with a modulo function:

$$\text{cube\_index} = \text{vertex\_index } \textbf{modulo } \text{total\_number\_of\_cubes}$$

which is widely used in prior works [24, 64, 108]. Next, initialize the cost value based on the mixed-cut graph H (line 2). In each iteration (lines 3-9) , the greedy algorithm will randomly swap two vertices in different cubes (line 4) . Then mixed-cut partitioning is applied to the graph and the cost is calculated based on the mixed-cut graph (line 5). If the cost increases, then undo the swap of the selected vertices (line 6); otherwise, keep the change (lines 7-9).

Figure 4.1(c) illustrates the example graph after being processed by one iteration of the greedy algorithm. After swapping v_4 and v_8, the number of cross-cube edges is reduced from 4 to 3, compared to mixed-cut partitioning. Since the workload of each vertex is related to the vertex degree, the total degree of cube C04 remains unchanged and the total degree of C00 is reduced from 5 to 4. Therefore, by swapping a pair of vertices in different cubes

Algorithm 1: Vertex-Swapping Greedy Mixed-Cut Alg.

input : G: The original graph
output : H: The partitioned graph

```
1 assign vertices to cubes by a simple hash function;
2 H, cost_old = mixed_cut(G);
3 for i = 1; i < max_iterations; i++ do
4     G' = swap random pair of vertices in distinct cubes in G;
5     H', cost_new = mixed_cut(G', H);
6     if cost_old < cost_new then undo_swap(G) ;
7     else
8         cost_old = cost_new;
9         update H with H' and G with G';
```

instead of moving a single vertex cross cubes, the greedy algorithm can maintain some degree of workload balance.

4.1.3 Partial Graph Repartitioning

Since the execution time of the iterative optimization algorithm is proportional to the number of iterations, it can be slow when the iteration number is large. Even worse, since the mixed-cut partitioning and cost calculation is performed every iteration (lines 4-5 in Algorithm 1), the larger the scale of the graph, the slower the iterative algorithm. Therefore, implementing mixed-cut partitioning to the whole graph every time after swapping the selected node pair is inefficient, especially for large-scale graphs. Fortunately, we observe that only a portion of the graph is modified after exchanging a pair of vertices. This observation provides an opportunity to increase the efficiency of the vertex-swapping strategy by only processing the influence scope of the swapping operation, instead of the whole graph, in each iteration.

Based on this observation, we propose a method, called *partial graph repartitioning*, in which the scope of the input graph considered is reduced to a smaller-scale subgraph (except for the first iteration). This method is inspired by FPGA partial reconfiguration techniques [91] which change the logic for a particular region in an FPGA without impacting operation in areas outside this region. This method is illustrated in Figure 4.2. It consists of 4 steps: ❶ Find the influence scope of the swapping operation (a small-scale subgraph called G'_{sub}) from the graph G' in which a target vertex pair has been swapped. ❷ Find H_{sub}, the influence

scope of the swap perturbation in H which is the mixed-cut graph of G. ❸ Extract G'_{sub} from G' and implement mixed-cut partitioning to the subgraph G'_{sub}. Then we can get H'_{sub}, the mixed-cut graph of G'_{sub}. ❹ Generate H', the mixed-cut graph of G', by removing H_{sub} from H and embedding H'_{sub} into H. The cost of H' can also be calculated in the same way. In this way, we process the small-scale subgraph representing the influence scope of the swap operation instead of processing the entire graph from scratch each iteration.

To find G'_{sub} and H_{sub}, we process all of the edges $e \in G'$ that are incident with at least one of the swapped vertices, enumerating all the possible cases. After checking all the possible scenarios, one of the key observations is that the boundary of the influence scope will not be expanded to the whole graph due to the fixed pattern of mixed-cut. Instead, the distance from any vertex in the influence scope to one of the swapped vertices is no more than 3. In other words, the scale of the influence scope is smaller than the whole graph for datasets with depth greater than 3.

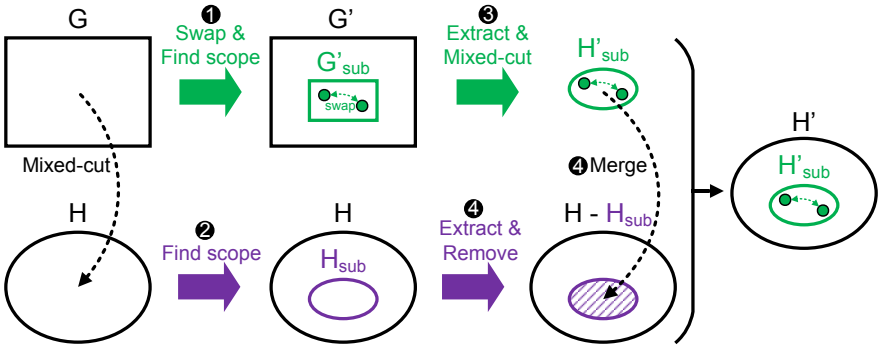


Figure 4.2: The partial graph repartitioning method. G_{sub} represents the influence scope of the target vertex pair before the swapping operation. H_{sub} is the mixed-cut graph of G_{sub}.

4.1.4 Three-Phase Programming Model

The two-phase programming model of GraphP has two limitations: (1) it supports source-cut partitioning, however it doesn't support destination-cut partitioning; (2) because the cross-cube data transfer procedure is scheduled by the operating system kernel rather than explicitly exposed to users, there is a lack of flexibility for optimizing and/or measuring cross-cube communication. In order to implement mixed-cut partitioning while maintaining compatibility with the source-cut method, we propose a new three-phase programming model shown in Algorithm 2, which has 3 steps:

Algorithm 2: Pseudocode of Three-Phase Programming Model (one iteration).

input : The SuperCut graph H and original graph G
output : Results of graph processing applications

1 **for** *each original vertex* $v_{org} \in G$ **do**
2 ⌊ gather_combine(v_{org})
3 **for** *each cross-cube edge* $e = (u, v) \in H$ **do**
4 ⌊ *update* ← gather_combine(u); scatter(*update*)
5 **for** *each original vertex* v_{org} *and replica* v_r **do**
6 ⌊ apply(v_{org}); apply(v_r)

Original Vertex Update (OVU) phase (lines 1-2): the original vertices are processed locally by collecting data from incoming neighbours and combining these data via computation operations packaged in the `gather_combine()` function which is customized to adapt to various graph applications. E.g., in the PageRank application, `gather_combine()` is an accumulation operation.

Remote Vertex Update (RVU) phase (lines 3-4): remote updates are generated and transferred across cubes. These updates are generated using user-defined function `gather_combine()` where: (1) destination-cut vertices are processed by combining data from incoming neighbors, and (2) each cross-cube edge starting from original vertices is traversed to get its source vertex data directly. After generating the updates, the user-defined function `scatter()` is invoked to transfer these updates across cubes.

Due to the inherent parallelism of graph applications, the OVU and RVU phase are executed in parallel so that the cross-cube communication latency is somewhat masked. Once OVU and RVU phase finish, all the updates are at their targets. It should be noted that cross-cube communication only happens during the RVU phase.

Apply phase (lines 5-6): In this phase, these updates are processed locally by the user-defined `apply()` function to generate the result for the current iteration, which also serves as the initial value of the next iteration.

Distinct from the two-phase programming model in GraphP, our programming model introduces the RVU phase for remote updates. If performing source-cut alone, the RVU phase is only responsible for data movement across cubes without the combining procedure. In this way, our programming model is not only suitable for mixed-cut partitioning but also

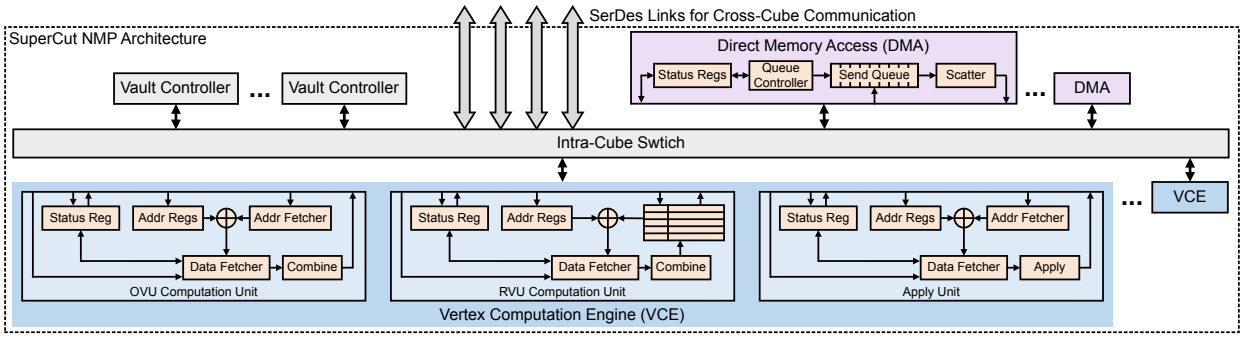


Figure 4.3: SuperCut near-memory processing architecture in the logic layer of each memory cube, composed of Vertex Computation Engines (VCEs) for intra-cube computation and DMAs for cross-cube communication. VCE consists of OVU Computation Unit, RVU Computation Unit, and Apply Unit.

compatible with source-cut. In addition, the cross-cube communication in our programming model is explicitly handled by the user-defined function scatter(), broadening the opportunity for communication functionality. A barrier before and after the Apply phase ensures that hardware cache coherence is not required.

4.1.5 Proposed Near-Memory System

To assess the benefits of SuperCut, we describe a near-memory system architecture that is similar, in many respects, to the near-memory systems of previous works. We use an HMC-like cube as our 3-D stacked memory with 8 GB DRAM capacity and 32 vaults per cube. Consistent with other multi-stack near-memory architectures, we utilize a Dragonfly topology (see Figure 2.2) to build a system with 16 memory cubes, in which each cube is connected to its neighbor cubes via SerDes links. We put FPGA resources on the logic layer of each cube, to which the 512 compute engines are mapped via HLS. These resources only take 0.26 mm^2 per cube (i.e., 0.12% of the total area), which is comparable to prior work.

Figure 4.3 illustrates the SuperCut NMP architecture on the logic layer of each memory cube. For intra-cube computation and communication (via the existing intra-cube switch), we include one Vertex Computation Engine (VCE) per vault consisting of 3 components: OVU Computation Unit, RVU Computation Unit and Apply Unit. We also design DMAs to implement cross-cube communication.

OVU Computation Unit: The OVU computation unit consists of a status register, address registers, an address fetcher, a data fetcher and a combine module. The status register includes the trigger and status bits, while the address registers are used to store the starting address of input vectors. The data addresses calculated by summing starting addresses and offsets fetched by the address fetcher are fed to the data fetcher. Then the fetched data is combined, performing the `gather_combine()` function. E.g., To implement the PageRank application, the `gather_combine()` is defined by users as an accumulation operation. Thus the combine module is synthesized to be an accumulator by the HLS compiler.

RVU Computation Unit: The RVU computation unit implements remote update generation. Distinct from the OVU computation unit, the address fetcher is replaced with a hash table of cross-cube edge information, based on which remote updates are generated by the specialized data fetcher and combine module performing the `gather_combine()` function.

Updates are transferred to specialized DMAs, along with distinct destination addresses, through the intra-cube switch. Since the OVU and RVU phases are overlapped, the OVU and RVU computation units are triggered together each iteration.

DMA: The DMAs perform the cross-cube communication. We include a send queue in each DMA to which updates with destination addresses are sent. The enqueued updates are transferred to another cube by the specialized scatter module, the realization of the user-defined `scatter()` function in the RVU phase. By default, the `scatter()` function is defined as a copy function to directly transfer data across memory cubes. It could also be defined by users with other purposes to satisfy various functionality of graph applications.

Apply Unit: The function of the apply unit is to implement the `apply()` function of the Apply phase in which updates are fetched by the data fetcher and then processed to generate results for original vertices and replicas by the apply module.

4.1.6 Graph Representation

The representation of the graphs in memory is a key link bridging the software and hardware system. We propose a new graph representation with a customized data structure stored in memory. Figure 4.4 illustrates an example of the graph representation in Cube0. In our graph representation, original vertices and replicas are stored in CSR format while cross-cube

edges information is stored in the form of a hash table where the key is edge ID and values are neighbors' IDs and destination addresses. The memory footprint of the hash table ranges from 0.17 MB to 189 MB, taking up 10%-13% of the overall memory footprint.

However, only considering the storage format is likely to introduce massive irregular memory accesses, which is more expensive than sequential memory accesses, when accessing and updating vertices in memory. To mitigate such irregularity, the order of vertices in the graph representation is rearranged during preprocessing. Original vertices and replicas are deployed within separate address ranges, so that these vertices can be accessed and updated sequentially in the appropriate phases of the programming model. In addition, since replicas in the same cube are updated by separate DMAs across cubes, to reduce irregularity of remote update, we also divide replicas into several address ranges, in the order of the index of the predecessor's memory cubes. In this way, replicas originating from the same cube can be updated contiguously.

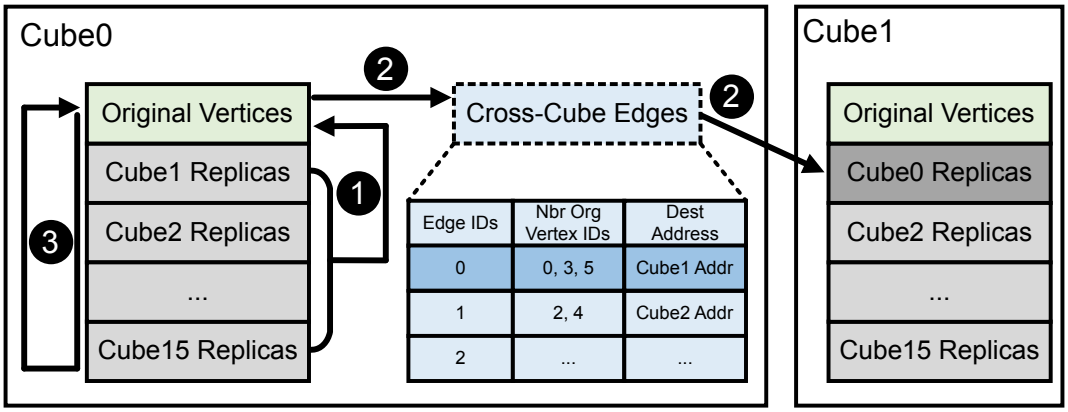


Figure 4.4: Diagram of graph representation in Cube0 and data communication (inside Cube0 and from Cube0 to Cube1).

Figure 4.4 illustrates an example of the intra-cube communication (inside Cube0) and cross-cube communication (from Cube0 to Cube1) in data layout view. ❶ In the OVU phase, data of original vertices and replicas from different address ranges are gathered and combined based on graph topology to generate updates for original vertices in Cube0. ❷ In the RVU phase, data of adjacent original vertices listed in the hash table is processed to generate remote updates. Separate from intra-cube communication, these updates are buffered in the send queue of the DMAs and then transferred to replicas originating from Cube0 in Cube1 using the destination addresses in the table. ❸ After the first two phases finish, updates are fetched from memory to apply for target vertices.

4.2 Experimental Methodology

Table 4.1: Graph dataset. Graph types are D – directed and U – undirected.

Graph Name	Graph Type	Vertex Count	Edge Count	Description	Iter. Count	Max. Deg.	Avg. Deg.
Wiki-Vote (WV)	D	7.1K	103.7K	Wikipedia who-votes-on-whom [55]	200K	893	14.6
ego-Twitter (TT)	D	81K	1.8M	Social circles from Twitter [66]	500K	1 205	21.7
Amazon0302 (AZ)	D	262.1K	1.2M	Amazon product co-purchasing [53]	3M	5	4.7
Com-Amazon (AU)	U	334.9K	925.9K	Amazon product network [105]	5M	168	2.8
Com-DBLP (DU)	U	317.1K	1M	DBLP collaboration network [105]	5M	306	3.3
soc-LiveJournal1 (LJ)	D	4.8M	69M	LiveJournal online social network [56]	20M	20 293	14.2

Simulation platform: We have adapted the `gem5-SALAM` [76] framework to build a bare-metal full-system NMP simulation platform. The host-side CPUs are based on the ARM ISA and the memory system consists of 16 HMC-like cubes to form a memory-centric network using a Dragonfly topology [48]. For our simulations, we use the standard distribution of `gem5` [11] that contains a stacked memory modeled after HMC and LLVM-based HLS accelerators to realize the computation units and programmable DMAs at 500 MHz.

Datasets: Table 4.1 shows the graph datasets used in our experiments. All these input graphs are collected from the Stanford Network Analysis Project (SNAP), a general-purpose graph library for network analysis and graph mining. These graphs have a wide range of types and fields, and are in the same scale range as prior works. In addition, Table 4.1 also shows maximum and average degree of graphs which have varying in-degree distributions, ranging from regular-like to powerlaw-like distributions.

Workloads: We code four popular graph processing applications in C using the proposed three-phase programming model. PageRank (PR) iteratively calculates the importance of web pages [14]. Average Teenage Follower (ATF) calculates the number of teenage followers of every user represented by vertices in the graph and the average number of teenage followers over K years old [40]. Breadth-First Search (BFS) searches a tree data structure, starting from a root vertex and traversing all the neighbours at the same depth iteratively. It is coded with a brute-force data parallel method to make it suitable for SIMD architecture [75]. Weakly Connected Components (WCC) finds a subgraph in which all the vertices are connected by some paths in which the direction of edges are ignored [84].

Evaluation methods: To evaluate the SuperCut framework, we simulate all the applications across all the graph datasets running on the NMP platform. We do the same for Tesseract and GraphP as well. Note that this implies we are comparing our proposed partitioning

methods to the previously described Tesseract and GraphP on a common hardware platform (described in Section 4.1.5).

The preprocessing step is implemented with Python and NetworKit library [86]. Without any optimization, the execution time of the single-thread python version ranges from several minutes to multiple hours. Since the implications of preprocessing substantially vary among different implementations, we show the number of iterations the greedy algorithm takes for each graph in Table 4.1 (executed off-line). We hope this work would inspire follow-on studies for efficient implementations that would speedup this step. In this work, we focus on exploring the on-line benefits of the partitioning methods. The parameters α_1 and α_2 in the cost function are set to $\alpha_1 = 0.2$ and $\alpha_2 = 0.8$ so as to emphasize energy savings somewhat over performance as the optimization goal.

Since the HLS accelerators are triggered and run in parallel, we use the maximum execution time across the HLS accelerators as the execution time for each iteration. The energy is computed by summing the dynamic energy consumption contributions from the local computation phases and the cross-cube communication phase. The total energy consumption of the HLS accelerators in each phase is modeled by `gem5-SALAM`. The energy consumption of the SerDes links, memory accesses to DRAM layers, and other modules on the logic layer are drawn from prior works [43, 74, 115].

4.3 Evaluation

4.3.1 Energy Consumption and Performance

We first quantitatively examine the energy consumption benefits of SuperCut, comparing SuperCut with 2 baselines: Tesseract and GraphP. Figure 4.5 shows the normalized energy consumption breakdown into computation, local memory accesses, and cross-cube communication relative to Tesseract. Focusing first on the energy consumption reduction of cross-cube communication, we observe that all the applications benefit from cross-cube communication reduction. The energy consumption reduction of cross-cube communication for each application ranges from 3.12× to 7.23× relative to Tesseract. Compared with GraphP, the energy consumption reduction of cross-cube communication ranges from 1.32× to 3.09×. This is

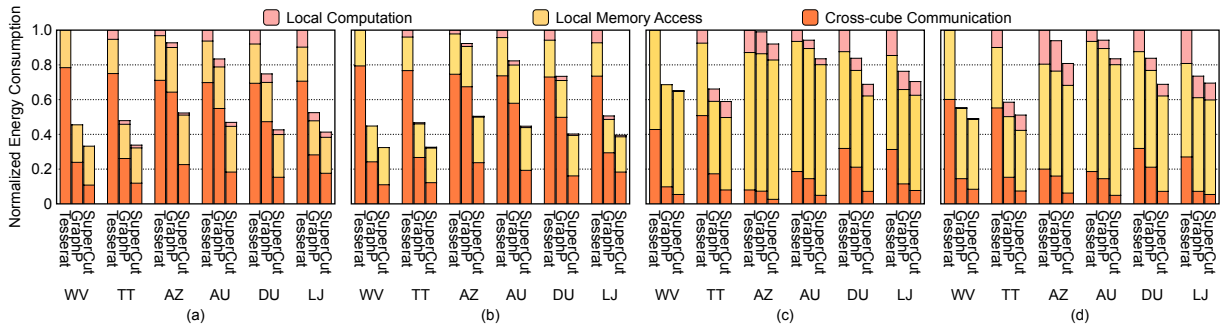


Figure 4.5: Normalized energy consumption breakdown of (a) PageRank, (b) ATF, (c) BFS, and (d) WCC applications, normalized to Tesseract. WV, TT, AZ, AU, DU and LJ are individual graphs.

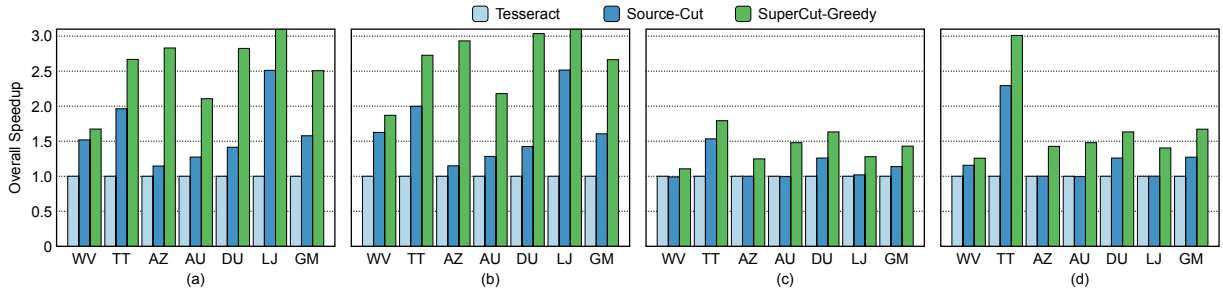


Figure 4.6: Overall speedup of (a) PageRank, (b) ATF, (c) BFS, and (d) WCC applications, normalized to Tesseract. WV, TT, AZ, AU, DU and LJ are individual graphs, GM is the geometric mean.

because SuperCut incorporates the aggregated cross-cube communication volume as one of the optimization targets. Due to the energy reduction of cross-cube communication, overall energy consumption is also reduced. The overall energy consumption reduction ranges from 1.1× to 3.09× and 1.06× to 1.84× relative to Tesseract and GraphP, respectively.

We next examine performance improvement by showing the overall speedup, defined as the execution time of the four graph applications relative to Tesseract, in Figure 4.6. Examining the last bar of each application, we observe that all the applications improve over both Tesseract and GraphP. Particularly, compared with GraphP (i.e., the state-of-the-art work), the geometric mean speedup is 1.59×, 1.64×, 1.24×, 1.33× for PageRank, ATF, BFS and WCC, respectively. We conclude that due to lower cross-cube communication volume and a balanced computational load, the performance of SuperCut is strong for all of the applications and all of the graph datasets.

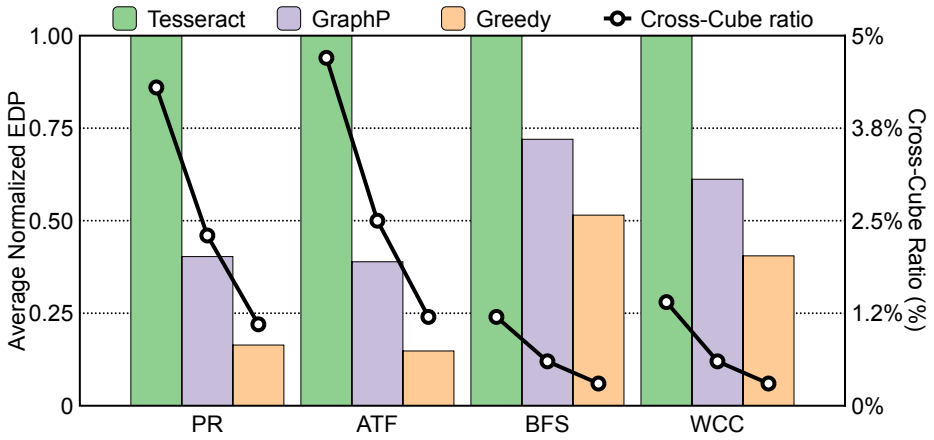


Figure 4.7: Average energy delay product and cross-cube communication ratio.

Turning our attention to how the energy consumption and performance benefit varies across applications, we observe a common relationship for both of them between applications with cross-cube communication and local memory access ratios. Figure 4.7 illustrates the average energy delay product (EDP) of each application across all the graph datasets with cross-cube communication ratio calculated as the average fraction of the data volume transferred across cubes to the overall data access volume. Here, we observe that high vertex activity applications (i.e., PageRank and ATF) with higher cross-cube communication ratio show more significant EDP reduction. This is consistent with a large communication volume within these applications. Since all the vertices in these applications are active in each iteration, the communication-to-computation ratio is high, leading to greater potential benefits achievable by SuperCut. In contrast, the property of low vertex activity applications (i.e., BFS and WCC), that only a portion of vertices participate each iteration, leads to a lower communication ratio. Thus, SuperCut achieves lower EDP reduction on these applications. We conclude that SuperCut is most beneficial for high vertex activity applications with a larger cross-cube communication ratio.

4.3.2 Mixed-Cut Partitioning

As mentioned in Section 4.1, SuperCut incorporates both the mixed-cut partitioning method and the greedy algorithm, illustrated in Figure 4.1(b) and (c) respectively. To understand how different components contribute to the benefits of SuperCut in terms of performance and energy consumption, we implement mixed-cut partitioning without greedy in SuperCut.

Since all the graphs have similar tendency, here we take AZ as an example. Figure 4.8(a) shows the energy of all four applications on AZ. We have two observations: First, mixed-cut partitioning reduces the overall energy consumption by generating less communication volume than GraphP on all the applications, validating our assumption that recognizing more edge patterns is beneficial to communication reduction. Second, the greedy algorithm combined with mixed-cut partitioning further reduces cross-cube communication volume by optimizing the vertex distribution. Figure 4.8(b) illustrates the overall speedup. From the figure, we can draw the same conclusions about mixed-cut partitioning in terms of performance. Note that the high vertex activity applications benefit the most from the inclusion of the greed algorithm.

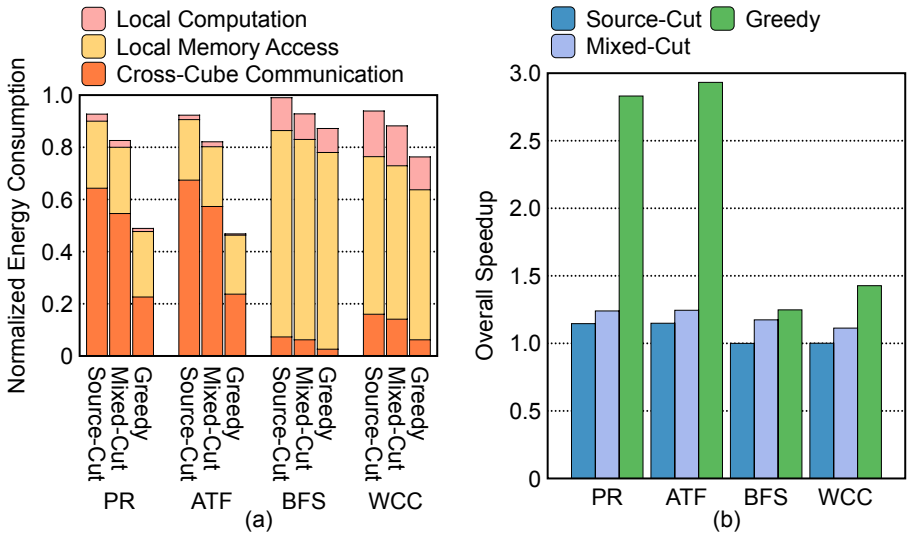


Figure 4.8: (a) Energy consumption breakdown and (b) overall speedup of mixed-cut partitioning on Amazon0302 (AZ), normalized to Tesseract.

4.3.3 Memory Footprint

Since SuperCut adopts the replica mechanisms of GraphP, both these works introduce extra memory footprint. In addition, SuperCut also generates destination-cut vertices during partitioning, the topological information of which is stored in memory. To assess the feasibility of SuperCut in terms of memory usage, we quantitatively examine the extra memory footprint of GraphP and SuperCut.

The evaluation results show that the extra memory footprint of SuperCut is 48%-75% of GraphP. This benefit comes from 3 facts: (1) SuperCut is better than GraphP at reducing the aggregated cross-cube communication volume (i.e., it introduces fewer replicas); (2) destination-cut vertices are only added for the pattern with multiple cross-cube edges, which guarantees that SuperCut has a lower memory footprint than GraphP; and (3) data for destination-cut vertices is buffered in the queue of DMAs instead of in memory.

4.3.4 Simulated Annealing

Despite the promising results of SuperCut, the greedy algorithm adopted is prone to getting trapped in local minima. In order to explore the best efficiency of data partitioning schemes proposed in SuperCut, we replaced the greedy heuristic with simulated annealing [50, 97], a more robust iterative stochastic optimization algorithm, to find a global near-optima. We adopt the same cost function in the simulated annealing algorithm as in the greedy algorithm. Figure 4.9 illustrates the cost function values of simulated annealing and greedy versus iteration number on graph AZ. From the figure, we have two observations. First, simulated annealing does achieve a lower cost function than greedy. The converged value of the cost function for simulated annealing is 53% of greedy. Second, simulated annealing takes more iterations. In this case it takes 12× more iterations than greedy. We conclude that greedy achieves good balance between efficiency and runtime.

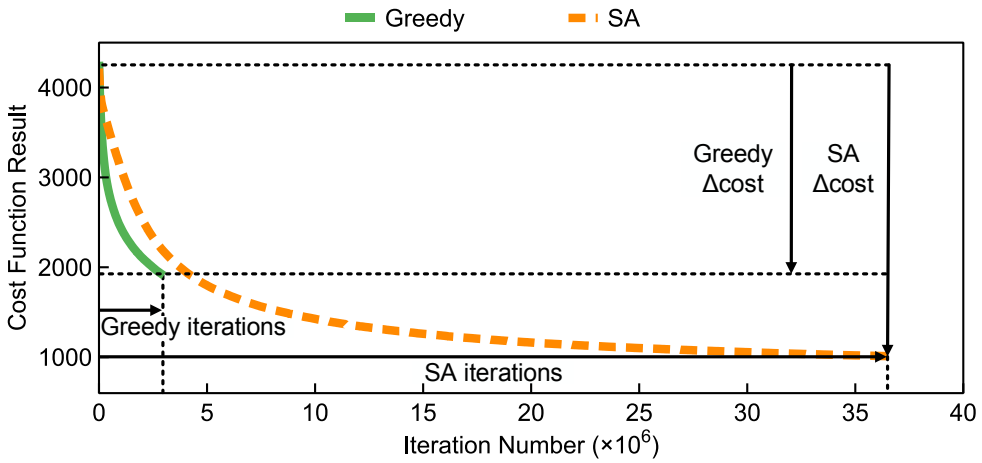


Figure 4.9: Simulated annealing (SA) and greedy cost function result on Amazon0302 (AZ).

Table 4.2 shows cost function, energy reduction (for ATF), and speedup (for ATF) of simulated annealing across all datasets, normalized to the greedy algorithm. We find that simulated annealing reduces overall energy consumption by 1.11× to 1.35× compared with the greedy as it reduces aggregated communication volume. However, for performance, we find simulated annealing degrades relative to greedy on 2 out of 5 graphs. This is because of the good job greedy-based SuperCut has done to effectively mitigate cross-cube communication. Thus, after partitioning the graphs using the greedy heuristic, the system performance bottleneck changes from cross-cube communication to computational workload balance, a topic to be explored in future work.

Table 4.2: Simulated annealing cost results, energy reduction (ATF), and speedup (ATF), normalized to greedy.

Graph	Cost Function Results	Energy Reduction (ATF)	Speedup (ATF)
WV	0.77	1.11	0.86
TT	0.55	1.25	0.85
AZ	0.53	1.35	1.16
AU	0.45	1.36	1.16
DU	0.59	1.25	1.01

4.4 Conclusions

For many graph processing applications, especially those with high vertex activity, cross-cube communication is a performance bottleneck on multi-cube NMP architectures. Here, we propose SuperCut, a framework for near-memory architectures to effectively reduce communication overheads while maintaining computational balance. We evaluate SuperCut on an NMP architecture based on reconfigurable logic using 4 representative graph applications and 6 real-world graphs. Results show that it provides up to 1.8× total energy consumption reduction and 2.6× speedup with 45% lower extra memory footprint relative to the current state-of-the-art.

Chapter 5

Graph Neural Network Inference via High-Level Synthesis

Distinct from Data integration workloads and graph processing applications, which are memory-intensive application, GNNs are a set of applications showing different characteristics. The performance and energy consumption of GNN implementations are hindered by both hardware platforms and software frameworks: (1) Distinct from traditional NNs, GNNs combine the irregular communication-intensive patterns of graph processing and the regular computation-intensive patterns of NNs. This feature can lead to ineffectual computation on CPUs and GPUs. (2) Since these frameworks assemble functions in a sequential way, one function will not start until the previous one finishes. This execution model leads to extra memory accesses, footprint, and implicit barriers for intermediate results, limiting the potential performance, energy consumption and the scale of graph datasets.

Field-Programmable Gate Arrays (FPGAs) are potentially an attractive approach to GNN inference acceleration. FPGAs' massive fined-grained parallelism provides opportunities to exploit GNNs' inherent parallelism. They also deliver better performance per watt than general-purpose computing platforms. In addition, FPGAs' reconfigurability and concurrency provide great flexibility to solve the challenges of hybrid computing patterns and ineffectual execution. Even better, the emergence of High-Level Synthesis substantially shortens FPGA development by automatically translating high-level software languages to RTL designs, bridging the gap between the non-trivial development efforts of hardware design and the rapid emergence of new GNN models. However, distinct from pure software programming, HLS developers need to adopt multiple optimization pragmas and follow certain coding styles to achieve best performance and energy cost. As reported in [15], the performance difference between a well-optimized version and a non-optimized version of the same kernel can be two

to three orders of magnitude. This invites an open question: *how effectively can modern HLS tools accelerate GNN inference?*

In this section, we introduce GNNHLS, an open-source framework for comprehensive evaluation of GNN kernels on FPGAs via HLS. GNNHLS contains a software stack extended from a prior GNN benchmark [31] based on PyTorch and DGL for input data generation and conventional platform baseline deployments (i.e., CPUs and GPUs). It also contains six well-optimized general-purpose GNN applications. These kernels can be classified into 2 classes: (1) isotropic GNNs in which every neighbor contributes equally to the update of the target vertex, and (2) anisotropic GNNs in which edges and neighbors contribute differently to the update due to the adoption of operations such as attention and gating mechanisms.

5.1 Framework Description

5.1.1 GNNHLS Overview

The GNNHLS framework, as depicted in Figure 5.1, comprises two primary components: data generation and HLS FPGA. The former is designed to generate input and output files and measure baselines on a CPU and a GPU, while the latter is designed to implement the optimized HLS applications on an FPGA. The data generation component mainly consists of the training system and the inference system, which are based on PyTorch and DGL. To account for the impact of graph topology on GNN model performance, it uses graph datasets with various topologies, including those from Open Graph Benchmarks [41]. In addition, six commonly used DGL GNN models obtained from a previous GNN benchmark [31] are incorporated. Thus, realistic model parameters, generated in the training phase, are utilized in inference.

The HLS FPGA component implements the GNN kernels on the FPGA. These kernels match the functionality of the DGL baselines and are optimized with several optimization techniques [26]. The optimized HLS kernels, with associated host files, data header files, and configuration files, are compiled by Vitis and executed on the FPGA. The optimization techniques applied in GNNHLS are described as follows:

Pipeline: Enable instruction-level concurrent execution to improve overall throughput.

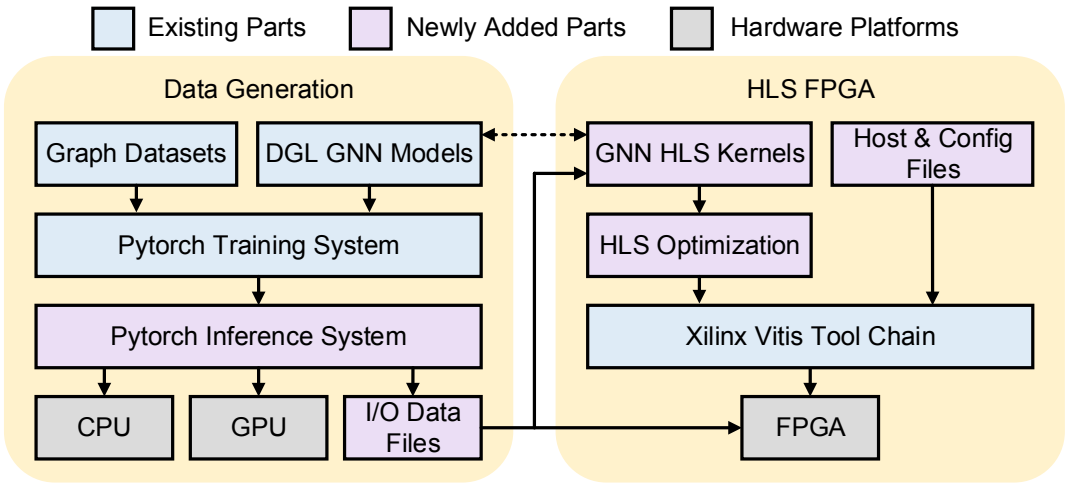


Figure 5.1: Diagram of the GNNHLS framework.

Loop Merge: Optimize the finite state machine (FSM) of nested loops to remove the impact of inner loop latency on the overall throughput.

Burst Memory Access & Memory Port Widening: access large chunks of data in contiguous addresses and increase memory port width to improve memory bandwidth.

Loop Unroll: Leverage instruction-level parallelism by executing multiple copies of loop iterations in parallel to increase throughput at the cost of resource utilization.

Dataflow: Enable task-level parallelism by connecting multiple functions with FIFOs to form a pipeline-style architecture and executing them concurrently.

Multiple Compute Units (CUs): Execute multiple kernel instances as CUs in parallel for different data portions at the cost of resource usage.

Figure 5.2 illustrates the dataflow diagrams of the GNNHLS kernels, in which memory and computation operations are divided and pipelined based on the complexity of each kernel. In general usage, the term *dataflow* architecture represents an architecture where instruction execution relies solely on the availability of input arguments rather than the program counter [5, 28, 94]. Within the FPGA community, the term dataflow architecture refers to a pipeline-style design built upon FIFOs for concurrent execution of functions. This latter usage can be considered a subset of the more general term. To mitigate the cost of dataflow, we also (1) tune the location of FIFO accesses to achieve better throughput, (2)

apply vectors for FIFO widening and associated operations, and (3) split loops to optimize the FIFO properties of loop indices.

Based on the mathematical expressions of GNN models presented in Chapter 1, we create the GCN HLS implementation, the dataflow diagram of which is depicted in Figure 5.2. Figure 5.2(a) illustrates the dataflow diagram of GCN. In addition to the memory access modules for input graphs and h, we split the computation operations into two modules: Aggregation of neighbor node vectors h_j and vector-matrix multiplication (VMM) for linear projection. We perform all the optimization techniques described previously to the GCN kernel. The memory burst length vector h is d, limited by the irregularity of the graph topology. The initiation interval (II) of the aggregation module is $4\,|N_i| + 2$. Since Vitis is not good at synthesizing tree-structured floating-point operations, we separate VMM into 2 functions in the dataflow scope for grouped VMM and sum, respectively. The II of VMM is thereby reduced from d^2 to $d + 36$. All these modules are reused in the following GNN models. Due to its simplicity, we create 2 CUs to process distinct vertices in parallel.

Figure 5.2(b) illustrates the dataflow structure of GraphSage. The memory read accesses and linear projection of the target feature, and neighbors' feature aggregation are executed simultaneously, and then summed up to update h_i.

In contrast to GraphSage, GIN illustrated in Figure 5.2(c) first sums up the aggregated vector of neighbors h_j and the target vertex vector h_i, hiding the latency of reading h_i, then performs two cascaded VMM modules with weight matrices U^l and V^l, respectively. This framework avoids the generation of long critical paths and achieves a higher clock frequency.

Figure 5.2(d) depicts the dataflow framework of GAT. Due to the unbalanced workload of the numerator and the denominator in (5), the results of $\exp(e_{ij})$, size $O(|N_i|)$, need to be temporarily stored prior to being accumulated. Considering the irregularity and large maximum $|N_i|$ of graphs, we divide the GAT model into 2 HLS kernels linked to the same memory banks for shared intermediate results: kernel 1 is designed to perform VMM with U and h, and multi-headed element-wise multiplication (MHEWM) with a_{src} and a_{dest}, respectively, in (6). After being optimized, the II of MHEWM is $k + 112$. The intermediate results are written back to memory and then read by kernel 2 to implement (4) and (5). Note that e_{ij} is computed twice in parallel to avoid performance degradation and deadlock issues. The II of aggregation, softmax, and MHEWM is $k \cdot |N_i| + 2k + 38$, $k \cdot |N_i| + k + 17$, and $k \cdot |N_i| + k + 14$, respectively.

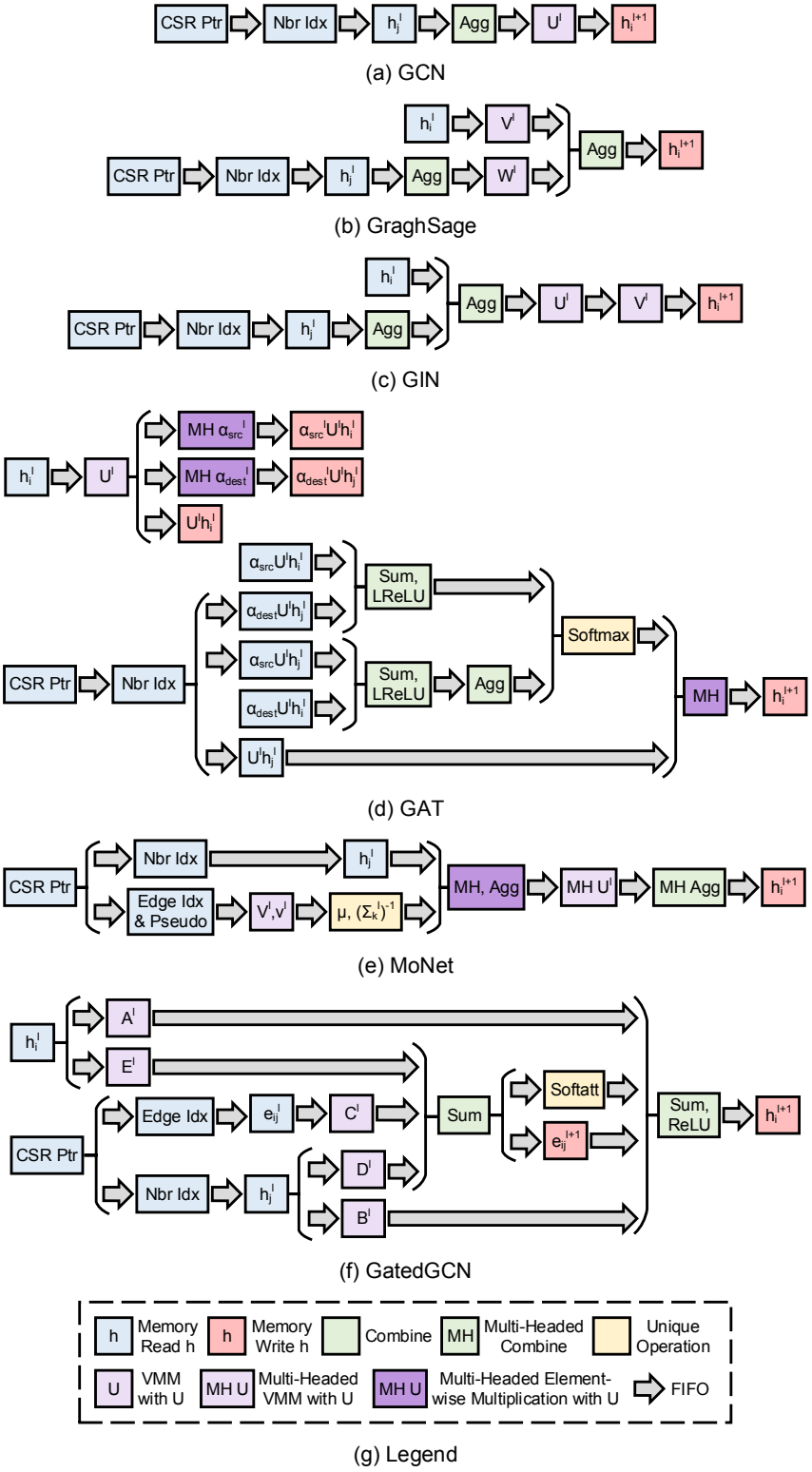


Figure 5.2: Dataflow diagrams of GNN HLS kernels in GNNHLS.

The dataflow diagram of MoNet is depicted in Figure 5.2(e). In our HLS implementation, $pseudo_{ij}$ of each edge is processed by a small VMM module with V^l and v^l in (9) and the Gaussian Weight Computation module with μ and $(\sum_k^l)^{-1}$ in (8). Meanwhile, h_j is read from memory for the subsequent MHEWM with aggregation, MHVMM with U, and MH Aggregation modules. Note that we perform the MH VMM with U after aggregation in (7), transferring it from an edge-wise to node-wise operation to reduce its occurrence. After optimization, the II of the VMM for u_{ij}, Gaussian computation, MHEWM with aggregation, MHVMM with U, and MH Aggregation are 1, 1, 4, $d + k + 28$, and $7k + 10$, respectively. We create 2 CUs for the HLS kernel to process vertices with distinct indices.

Since the soft attention of GatedGCN shown in (11) is distinct from GAT, performing accumulation operations for e_{ij} on both the numerator and denominator, we implement a single pipeline to build the HLS kernel. Figure 5.2(f) illustrates the dataflow framework of GatedGCN. To hide the latency of multiple VMM modules in GatedGCN, we perform all of them in parallel with parameters A, B, D, E, and C, respectively. Then the soft attention module is implemented to update h_i. After optimization, the II of the soft attention and sum modules to generate h_i^{l+1} are $10 \cdot |N_i| + 72$ and 31, respectively.

5.2 Experimental Methodology

Datasets: Table 5.1 shows the graph datasets used in our evaluation. All these graphs are collected from Open Graph Benchmark [41], a widely-used graph library for GNNs, and have a wide range of fields and scales. These graphs represent two classes of graphs with distinct topologies used in the GNN community: MH and MT consist of multiple small dense graphs, while AX and PT each consist of one single sparse graph. The maximum and average degree shown in Table 5.1 indicates their varying distributions ranging from regular-like to powerlaw-like. In addition, we set feature dimensions for the kernels: GCN, GraphSage, and GIN have the same input and output dimensions at 128. The input, head, and output dimensions of GAT and MoNet are (128, 8, 16) and (64, 2, 64), respectively. All the dimensions of GatedGCN are 32.

Evaluation methods: To perform evaluation, we use a Xilinx Alveo U280 FPGA card, provided by the Open Cloud Testbed [52], to execute the HLS kernels. This FPGA card provides 8 GB of HBM2 with 32 memory banks at 460 GB/s total bandwidth, 32 GB of DDR

Table 5.1: Graph datasets [41].

Dataset	Node #	Edge #	Max. Deg.	Avg. Deg.
OGBG-MOLTOX21 (MT)	145 459	302 190	6	2.1
OGBG-MOLHIV (MH)	1 049 163	2 259 376	10	2.2
OGBN-ARXIV (AX)	169 343	1 166 243	13 155	6.9
OGBN-PROTEINS (PT)	132 534	79 122 504	7 750	597.0

memory at 38 GB/s, and 3 super logic regions (SLRs) with 1205K look-up tables (LUTs), 2478K registers, 1816 BRAMs, and 9020 DSPs. We adopt 32-bit floating point as the data format. We use Vitis 2020.2 for synthesis and hardware linkage with the power-profile option enabled to perform power profiling during runtime, and Vitis Analyzer to view resource utilization, execution time and power consumption. We compare our HLS implementation with CPU and GPU baselines with PyTorch and the highly-optimized DGL library. We perform CPU baseline runs on an Intel Xeon Silver 4114 at 2.2 GHz with 10 cores, 20 threads, and 13.75 MB L3 cache. The GPU baseline is implemented on an Nvidia RTX 2080 Ti with 2994 CUDA cores at 1.5 GHz and 8 GB GDDR6 at 448 GB/s total bandwidth. We measure the energy consumption of the CPU and GPU baselines using the same technique as prior work [60].

5.3 Characterization

To capture insight into the properties of GNNHLS, we first characterize the GNN kernels using instruction mix, spatial locality, and temporal locality. We use Workload ISA-Independent Characterization (WIICA) [81], a workload characterization tool, to capture ISA-independent properties by generating and parsing a dynamic trace of runtime information. Due to the limits of disk and processing time, profiling the the full trace is impractical. Thus we use uniform random node sampling [54] to select a sequence of 500 nodes for evaluation.

5.3.1 Instruction Mix

We first take a look at the dynamic instruction mix, partitioning instructions into 3 classes: branch, memory and compute. Figure 5.3 shows the instruction mix of the HLS kernels on the

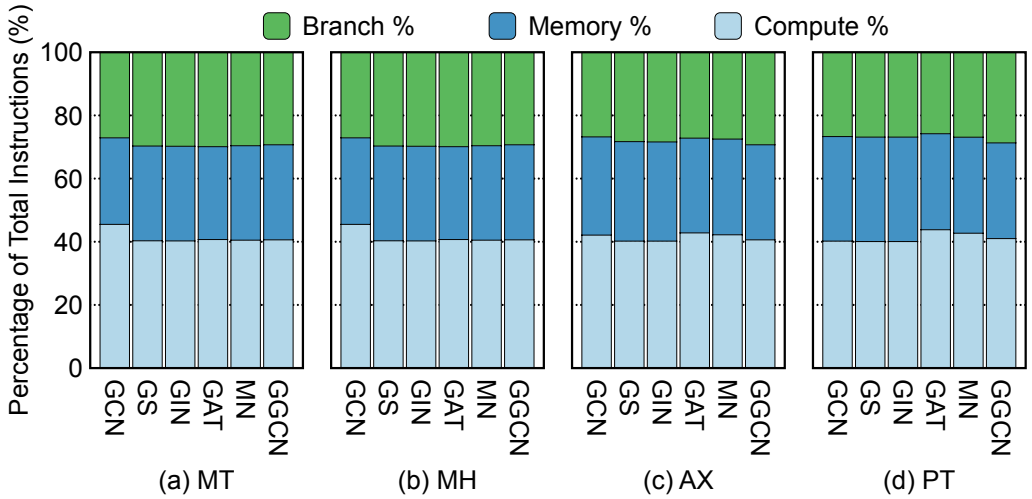


Figure 5.3: Instruction breakdown of all the HLS kernels.

4 datasets. We observe that the instruction breakdown shows a consistent tendency: (1) The computation instructions make up largest fraction (about 40%−50%) of total instructions, implying that these pipeline-style GNN HLS kernels are computation-intensive. (2) Memory instructions consume the second largest fraction (about 30% 35%), indicating the total number of memory accesses is still nontrivial even if all the kernels are in a pipeline style. (3) While branch instructions take 25%−30% of the total, most of them are due to conditional statements of for loops and irregularity of graphs. We also observe that denser graphs (e.g., AX and PT) induce a higher fraction of compute instructions for anisotropic kernels (i.e., GAT, MN, and GGCN) due to their edge-wise operations. In contrast, denser graphs induce a higher fraction of memory instructions for isotropic kernels (i.e., GCN, GS, and GIN) because their edge-wise operations are less computation intensive than node-wise update.

5.3.2 Spatial and Temporal Locality

We use spatial locality and temporal locality scores developed by Weinberg et al. [95] to quantitatively measure the memory access patterns. Spatial locality characterizes the closeness of memory references among consecutive memory accesses. For HLS accelerators, it represents the potential opportunity to optimize the efficiency of prefetching and memory burst transfer. Temporal locality measures the frequency of memory instructions accessing the same memory address. It represents the latent efficiency of caching data elements so that they can be

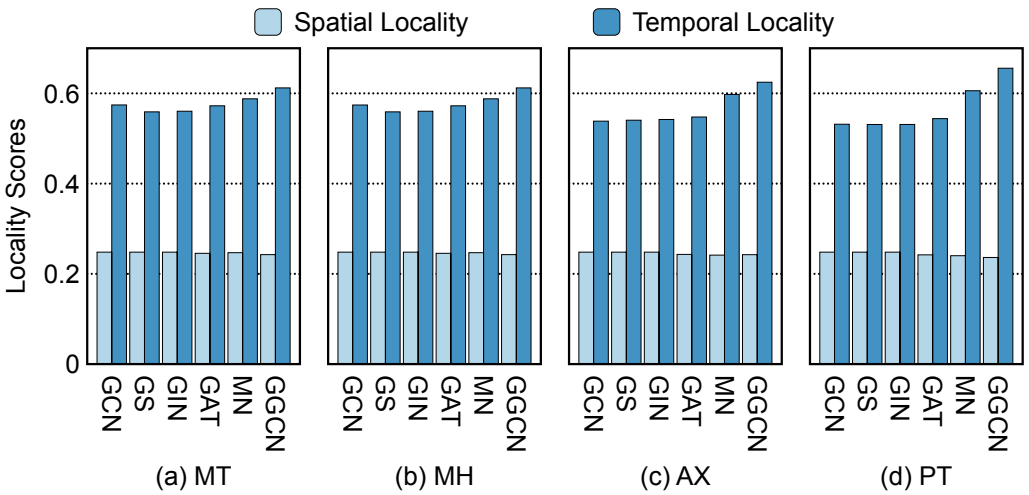


Figure 5.4: Memory locality scores of HLS kernels.

accessed repetitively with lower cost. Therefore, the higher the temporal locality, the more performance improvement due to caching mechanisms in the accelerators. Both return a score in the range $[0, 1]$.

Figure 5.4 illustrates the spatial and temporal locality scores. Focusing first on the spatial locality, we observe the score stays consistently low (about $0.23 - 0.25$) across all the kernels and datasets. It is because the irregularity of graph topology induces non-contiguous memory references, limiting memory burst transfer and prefetching to the length of feature sizes. Next examining the temporal locality, we observe that the score stays in the range of $0.5 - 0.7$, indicating the potential performance benefit of caching mechanisms, regardless of the graph topology. In addition, we observe anisotropic kernels show a higher temporal locality than isotropic kernels, due to them having more edge-wise operations.

5.4 Evaluation

5.4.1 Resource Utilization

We first examine the resource utilization and clock frequency after place & route. FPGA resources include look-up tables (LUT), flip-flops (FF), BRAM, and digital-signal-processors (DSP). Table 5.2 shows these results. From the table, we observe that the frequency of

Table 5.2: Resource Utilization of HLS GNN models.

	Target Freq.	Actual Freq.	LUT	FF	BRAM	DSP
GCN	300 MHz	250 MHz	264 485	413 197	41	2 880
GS	250 MHz	204 MHz	253 608	358 722	33	2 766
GIN	300 MHz	190 MHz	278 251	421 915	55	3 264
GAT	300 MHz	255 MHz	168 559	248 424	81	1 718
MN	300 MHz	250 MHz	289 208	428 917	212	2 236
GGCN	300 MHz	270 MHz	151 497	235 484	124	1 036

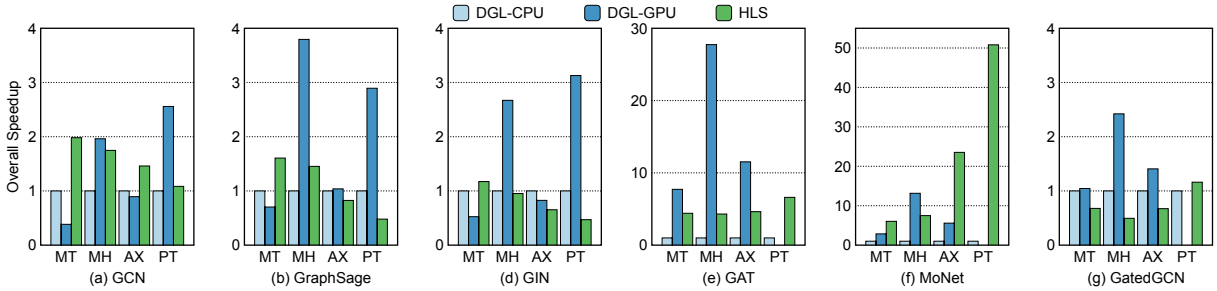


Figure 5.5: Speedup of HLS kernels relative to DGL-CPU. The higher the better.

all the kernels is lower than the target frequency, which is not unusual in FPGA designs. Among these kernels, GraphSage achieves a low frequency due to some critical paths which are unresolvable by the tool. In addition, we observe that the resources on the FPGA are not over-utilized.

Table 5.3: Execution time (sec) of DGL-CPU, DLG-GPU, and GNN HLS implementation on 4 graph datasets. DGL-CPU columns are labeled CPU, DGL-GPU columns are labeled GPU, and GNN HLS columns are labeled HLS.

	MT			MH			AX			PT		
	CPU	GPU	HLS	CPU	GPU	HLS	CPU	GPU	HLS	CPU	GPU	HLS
GCN	0.11	0.28	0.05	0.69	0.35	0.39	0.31	0.34	0.21	16.09	6.29	14.85
GS	0.21	0.30	0.13	1.42	0.38	0.98	0.43	0.42	0.52	16.45	5.68	34.29
GIN	0.15	0.29	0.13	0.93	0.35	0.98	0.34	0.41	0.52	16.11	5.15	34.29
GAT	0.91	0.12	0.21	6.52	0.24	1.51	3.10	0.27	0.67	186.93	OoM	28.28
MN	0.32	0.11	0.05	2.37	0.18	0.32	1.18	0.21	0.05	89.71	OoM	1.77
GGCN	0.12	0.11	0.17	0.62	0.26	1.26	0.36	0.26	0.54	38.93	OoM	33.55

5.4.2 Performance

We next examine the performance improvement by showing the overall speedup, defined as the execution time of the GNN HLS kernels relative to CPU-DGL (using all 10 cores on the CPU), in Figure 5.5. Table 5.3 shows the execution time of baselines and HLS kernels. Note that GPU results of GAT, MN, and GGCN on PT cannot be obtained because of running out of memory (OoM). Examining each kernel in Figure 5.5, we observe that the HLS implementation is not always outperforming corresponding CPU baselines. Compared with DGL-CPU, the speedup ranges from 0.47× to 50.8×.

Among isotropic GNN kernels, GCN achieves better performance than GraphSage and GIN, ranging from 1.08× to 1.98× because its simpler structure enables us to create two CUs to leverage spatial data parallelism. In contrast, we can only create one CU for GraphSage and GIN each because of their complex structure and heavy resource usage. In addition, we observe that the execution time of GraphSage and GIN are close. Thus, we conclude that the distinction on the structure of these two GNN models will not substantially affect HLS implementation results.

Among anisotropic kernels, MoNet achieves highest performance improvement ranging from 6.04× to 50.8× due to (1) its single pipeline structure with computation order optimization where the node-wise operations are placed behind the edge-wise operations, and (2) well-designed MHVMM modules with lower II, especially MHVMM whose II is $O(d + k)$ instead of $O(dk)$. In spite of the 2-pipeline structure of GAT, we observe that it still achieves 4.31× to 6.61× speedup relative to multi-core CPU baselines. In addition, since the feature size of GatedGCN is smaller, leading to more performance improvement for CPU baselines with time complexity of $O(d^2)$, its speedup is not comparable to other anisotropic kernels, ranging from 0.5× to 1.16×.

Turning our attention to how the performance benefit of HLS implementations varies across graph datasets, we observe that the speedup of isotropic kernels relative to DGL-CPU on regular-like graphs (i.e., MT and MH) is higher than powerlaw-like graphs (i.e., AX and PT) because (1) the edge-wise operations are less computation-intensive than node-wise operations in these kernels, making the baselines more computationally efficient on powerlaw-like graphs containing more edges than nodes; and (2) the edge-wise aggregation operations in HLS implementations are executed sequentially without leveraging edge-level

parallelism, making these HLS kernels less computationally efficient for powerlaw-like graphs. Distinct from isotropic kernels, the speedup of anisotropic kernels on powerlaw-like graphs is higher than regular-like graphs because the edge-wise operations of these kernels are more computation-intensive than isotropic kernels, making baselines less efficient on powerlaw-like graphs.

Focusing on the second and the third bar, we observe that DGL-GPU outperforms HLS implementations in many cases, due to the high-performance fixed-function accelerators in the GPU. The speedup of HLS kernels relative to the GPU baselines ranges from $0.13\times - 5.16\times$. In spite of the promising GPU performance, there are still some drawbacks of GPU compared with HLS implementations. For the execution of isotropic GNN models, DGL-GPU achieves lower speedup than HLS on small-scale graphs such as MT and AX. It is speculated that the GPU is designed to achieve high throughput in the cost of latency which plays a more important role for small-scale graphs than large-scale graphs. In addition, compared with HLS implementations on FPGA, GPU is also not suitable for the execution of anisotropic GNN models on large-scale, especially powerlaw-like graphs (e.g., PT) due to (1) the non-trivial memory footprint caused by its sequential execution paradigm to store intermediate results of edge-wise operations, and (2) insufficient memory capacity on the GPU board. That is why we failed to execute anisotropic GNNs on PT with GPU. It is solved by the HLS implementations' pipeline structure not storing the intermediate results.

Since GenGNN [2] also discusses 3 of the GNN models included in this paper (GCN, GIN, and GAT), we can make a limited comparison of our GNN HLS implementations with theirs. The two are not directly comparable for a number of reasons: (1) the feature dimensions of our GNN HLS kernels are higher, (2) we use off-chip memory instead of on-chip memory, (3) our general-purpose GNN HLS kernels focus more on throughput rather than real-time latency, and (4) the FPGAs are from the same family, but are not same part. The performance of our HLS kernels exceeds that of GenGNN, achieving overall speedup of $35\times$, $5\times$, and $6\times$ over GCN, GIN, and GAT, on MT, respectively.

5.4.3 Optimization Techniques

As described in Section 5.1, we apply multiple optimization techniques to the HLS kernels. In order to evaluate the efficacy of these techniques, we use GraphSage on MT as a case

Table 5.4: Execution time of various optimization techniques for GraphSage on MH.

Optimizations	Execution Time (s)	Speedup
No Pragmas	129.59	1.00×
Dataflow	65.11	1.99×
Loop Unroll	11.11	11.7×
Vectorization	4.44	29.2×
Split Loops	0.98	132×

study. Table 5.4 presents the execution time of GraphSage with the combined impact of optimization techniques applied. The reported execution time of each technique represents the effect of both the current technique and above techniques listed in the table. In the table, No Pragma means we don't intentionally apply any pragmas to the HLS code, except for those automatically applied by Vitis (i.e., Pipeline, Loop Merge, and Memory optimizations). Dataflow denotes that we apply dataflow pragma and FIFO streams to exploit the task-level parallelism of each application. Loop Unroll means we apply loop unroll pragmas to completely or partially unroll for loops, keeping II as low as possible while exploiting instruction parallelism. Vectorization means using vector data types to widen the width of FIFO streams and corresponding operations to decrease the cost of FIFO accesses. Split Loops means splitting the outer-most node loop and putting it inside each function connected by streams to further optimize FIFO properties inferred from loop indices.

We observe that Loop Unroll achieves the highest performance improvement. Therefore, exploiting instruction parallelism is still the primary choice for GNN HLS optimization. In order to further improve performance, exploiting task-level parallelism is necessary. Focusing on the first and second row in the table, we observe that only performing the dataflow pragma and streams in a naive way obtains 1.99× performance improvement. By applying Vectorization and Split Loops as complementary techniques of Dataflow, performance is further improved by 2.5× and 3.9×, respectively. After applying all the optimization techniques together we observe that the performance of GraphSage is improved by 132×.

5.4.4 Energy Consumption

We next present a quantitative analysis of the energy consumption. Figure 5.6 displays the energy reduction of both DGL-GPU and HLS implementations relative to DGL-CPU in logarithmic scale. Energy reduction is calculated as the energy consumption of DGL-GPU or

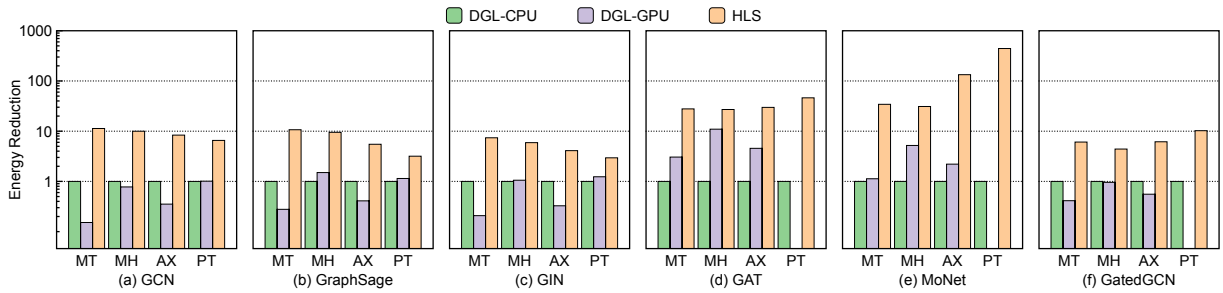


Figure 5.6: Energy consumption reduction of HLS kernels relative to DGL-CPU (logarithmic scale). The higher the better.

Table 5.5: Energy Consumption (J) of DGL-CPU, DGL-GPU, and GNN HLS implementations on 4 graph datasets. DGL-CPU columns are labeled CPU, DGL-GPU columns are labeled GPU, and GNN HLS columns are labeled HLS.

	MT			MH			AX			PT		
	CPU	GPU	HLS	CPU	GPU	HLS	CPU	GPU	HLS	CPU	GPU	HLS
GCN	9.06	59.67	0.80	58.38	75.25	5.85	25.93	73.38	3.10	1 367.75	1 352.67	208.77
GS	17.95	64.60	1.68	120.97	80.63	12.73	36.74	89.54	6.69	1 397.99	1 221.69	439.91
GIN	13.12	63.20	1.77	79.25	75.04	13.40	29.10	89.11	7.10	1 369.04	1 107.06	464.29
GAT	77.45	25.37	2.79	554.10	50.53	20.50	263.09	57.74	8.83	15 889.04	OoM	344.14
MN	27.46	24.32	0.80	201.19	38.70	6.48	100.59	45.59	0.75	7 625.48	OoM	17.22
GGCN	9.84	23.82	1.62	53.12	55.47	12.05	30.76	55.32	5.00	3 309.16	OoM	323.44

HLS divided by that of DGL-CPU. Examining the final bar of each application and dataset, we observe that HLS implementations consume less energy than CPU and GPU baselines in all cases. The energy reduction ranges from 2.95× to 423× relative to DGL-CPU and from 2.38× to 74.5× relative to DGL-GPU. It is because of the low power of FPGA logic, low clock frequency, and efficient pipeline structure of HLS implementations.

Focusing on the first and last bar, we observe a similar tendency in energy reduction as in performance: for isotropic GNN models, denser graphs result in lower energy reduction, whereas for anisotropic GNN models, denser graphs result in higher energy reduction. This leads us to conclude that improving GNN applications generally will require some degree of graph topology awareness.

5.5 Conclusions

In this chapter, we propose GNNHLS, an open-source framework to comprehensively evaluate GNN inference acceleration on FPGAs via HLS. GNNHLS consists of a software stack for data

generation and baseline deployment, and 6 well-tuned GNN HLS kernels. We characterize the HLS kernels in terms of instruction mix and memory locality scores, and evaluate them on 4 graph datasets with various topologies and scales. Results show up to 50.8× speedup and 423× energy reduction relative to the multi-core CPU baselines. Compared with GPU baselines, GNNHLS achieves up to 5.16× speedup and 74.5× energy reduction.

Chapter 6

Performance of HLS-based Graph Neural Networks

Although HLS bridges the gap between software and hardware development, optimizing HLS codes is substantially distinct from conventional software programming. In fact, due to the FPGAs' inherent attributes, such as lack of built-in cache mechanisms, low clock frequency (relative to traditional processor cores), and fine-grained configurability, the performance difference between a well-optimized version and naive version of the same kernel can be two or three orders of magnitude [15, 32, 85]. Therefore, to achieve the best performance, HLS developers need to explore a large optimization space for HLS designs with various optimization pragmas, coding paradigms, etc.

As the complexity of kernels increases, optimizing (or auto-optimizing) such kernels is difficult via conventional HLS workflows for several reasons:

1. Since pure C emulation is only designed for functionality verification, current HLS developers have to use RTL simulation to understand performance by manually mapping the results of individual signals in the generated waveform back to the HLS code. However, since all the signal names are auto-generated, they are not easily comprehensible by users. Besides, RTL simulation usually takes a very long time, making the tuning effort arduous. Even worse, it is exacerbated by the fact that tuning with a small example data set is less meaningful for GNN kernels in terms of performance estimation because of the inherent irregularity of graph datasets and algorithms. In other words, distinct graph topologies can significantly impact the final performance achieved. Therefore, when it comes to large-scale graphs, RTL simulation is impractical to be used to optimize GNN kernels with these graphs.

2. The notion of dataflow architectures which exploit task-level parallelism, where multiple functions are connected by FIFOs and executed concurrently instead of sequentially, further mystifies the optimization process because it induces a wider set of design space challenges including: task partitioning, FIFO depth tuning, and bottleneck identification, which are distinct from conventional computation platforms.

The critical missing piece in the optimization task is the availability of fast, high-quality understanding of the performance implications of the design choices that are made. Our focus in this work is to address this missing element, providing the the designer (whether it be a human or an automatic design space exploration tool [80]) with performance predictions both quickly and with sufficient accuracy that they can be used effectively.

Traditional approaches to performance assessment either involve static assessment (i.e., compile-time analysis) or cycle-accurate simulation. In this chapter, we propose a different method, effectively between the approaches of static estimation and cycle-accurate simulation, to investigate the impact of irregularity of data and algorithms on performance. Due to the existence of other HLS tools for functional verification (e.g., software emulation in Vitis), our method decouples functional verification from performance estimation, so that the runtime of the estimation process is independent of the computational details of the FPGA algorithms.

Here, we introduce HLPerf, a performance evaluation methodology that supports the performance variations inherent in data-dependent algorithms (it is simulation based), but relaxes the notion of cycle accuracy and replaces it with "approximate" cycle accuracy. The result is a simulation-based performance estimate that is two orders-of-magnitude faster than state-of-the-art simulations that perform cycle-accurate functional verification.

The chapter is organized as follows. Section 6.1 describes the methodology, including the design workflow with HLPerf, the event-driven simulation, and the modeling of HLS kernels. Section 6.2 gives two options for an application developer's experience using HLPerf. Section 6.3 articulates the evaluation methods, and Section 6.4 gives quantitative evaluation results. Section 6.5 provides conclusions.

6.1 Methodology

6.1.1 Overall Workflow

As depicted in black in Figure 2.3, the conventional design flow exhibits a deficiency in effective and practical methods for estimating the performance of GNN HLS kernels, which is crucial for rapid iterative tuning. To address this shortcoming, we devise a new workflow, HLPerf, incorporating the new element highlighted in red in Figure 2.3. The main idea is to circumvent the need for RTL simulation or FPGA execution each design iteration and build a high-level "approximately-cycle-accurate" abstraction of the HLS kernel that supports much higher simulation speed to accelerate the iterative design space exploration of HLS kernels. HLPerf is composed of 3 core elements: (1) a discrete-event simulation system built upon SimPy to emulate the inherent concurrency of the dataflow architecture and capture its dynamic execution behavior; (2) quantitative expressions of pragma-driven patterns to model the performance impact of various optimization techniques, decoupling performance estimation from functional verification; and (3) a front-end source-to-source compilation step to automatically transform the HLS C-based source code of diverse GNN applications into corresponding simulation components. These elements are used to build "approximately-cycle-accurate" models focusing only on the performance estimation of fundamental loops with distinct optimization techniques. Since HLPerf (1) decouples the performance estimation from computational intricacies of the algorithm, (2) involves fewer signals to be simulated, and (3) adopts coarser granularity of runtime simulation rather than cycle-accurate simulation, the performance estimation process and the resulting iterative tuning procedure are substantially accelerated.

The overall workflow of HLPerf is shown in Figure 6.1. It consists of 4 steps: model creation, parameter loading, HLPerf simulation, and kernel tuning.

❶ **Model Creation:** In addition to the conventional HLS workflow where HLS C code is checked with functionality via C emulation and converted to RTL code via HLS, in HLPerf we build a front-end converter to analyze the source code of HLS kernels to extract the key information, such as the code structure, the topology of dataflow scopes, high-level expression of loops and FIFOs, and HLS optimization techniques. Then the corresponding approximately-cycle-accurate HLPerf model is automatically generated by the converter.

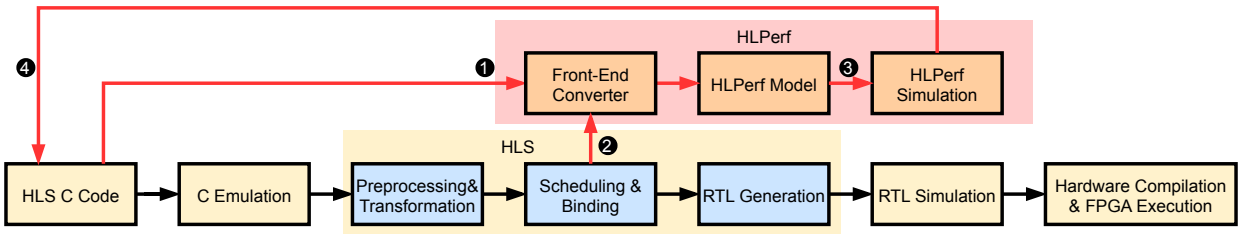


Figure 6.1: The overall workflow of HLPerf.

❷ **Parameter Loading**: After the scheduling and binding procedure of HLS and before the start of RTL generation, the front-end converter also automatically extracts realistic performance parameters (e.g., latency and initiation interval) from the synthesis log file and uses them to complete the generated HLPerf model. Note that this step does not rely on the first step, so they can be executed in parallel.

❸ **HLPerf Simulation**: After performing the simulation of the HLPerf model with graph datasets, a detailed result summary report is generated, offering a holistic view of the dataflow architecture's performance. This report provides the overall estimated performance of the GNN HLS kernel, and can help users to explore the impact of distinct graph topologies on the performance of the dataflow architecture and pinpoint performance bottlenecks.

❹ **Kernel Tuning**: With the help of the HLPerf result report, users can tune the HLS C code for one or more kernels with an alternative strategy, such as code paradigm, FIFO sizes, optimization technique choices, etc. Even better, if the code modification is not related to the parameters achieved from the scheduling and binding procedure, users can directly re-run the workflow of HLPerf without performing the HLS step, which can save additional time.

6.1.2 Discrete-Event Simulation

In cycle-accurate simulations, such as RTL simulation, activity within the design is modeled at each cycle, whether or not that activity is relevant to the performance estimation of the design (e.g., it might be relevant for functional correctness, but not performance). However, many cycles do not have activity that is relevant to the performance estimation, and can therefore be skipped in a purely performance-oriented simulation.

In contrast to the cycle-based progression described above, HLPerf's Discrete-Event Simulation (DES) adopts an event-driven paradigm. The key idea is to generate discrete sequences of coarse-grained events occurring at specific time intervals. These events represent a change of state in the system and are scheduled with a time-skipping strategy which advances the simulated time forward directly to the subsequent event's occurrence, Therefore, it enables the creation of simplified system models that concentrate on key processes of significant influence, while avoiding the intricacies of low-level details. By substantially diminishing the complexity and resource intensity of the simulation, DES is more efficient and time-effective than time-stepped simulation methodologies.

In this paper, we build the high-level performance models within HLPerf for a variety of GNN kernels using SimPy [88], a Python-based generic library for discrete-event simulation. In essence, SimPy operates as an asynchronous event dispatcher. It generates and schedules events at specified time intervals by storing all the events in a heap-based event list and ordering them by simulation time, priority, and increasing event ID. In SimPy, systems are modeled through the creation of process functions. These functions simulate entities whose behaviors evolve over time. Rather than exploiting multiple threads to replicate the inherent parallelism in processes, SimPy utilizes Python's generator functions in each process function. These generators are characterized by Python's `yield` keyword and act as suspend/resume points. This feature allows the temporary suspension of process functions and subsequent resumption at the point of last suspension. Specifically, when a generator issues a yield command within a process, the process is paused, and a new event is yielded and inserted into the event list with the position in a given order. Subsequently, SimPy's internal functions inspect the scheduled events, extracting and removing the one with the earliest simulation time. The system's simulation time, `SimTime`, is then updated to the time of that event, and the corresponding process is resumed immediately following the last executed `yield` statement. Through this mechanism, the parallelism of process functions can be simulated by alternate execution of effective co-routines.

In addition, process functions are not standalone entities. SimPy provides shared resource classes to model the interaction between processes. These resource classes serve as containers with a user-defined capacity, so that process functions can either write data into the resource instances using the `put()` method or retrieve data from it using the `get()` method. Since both of these methods are Python generator functions with `yield` statements, they can return an event that is triggered when the corresponding action is to be executed. Therefore,

when the resource is empty or full, processes are required to wait for the state change of the resource. This system ensures a controlled and orderly interaction among processes for shared resources. All aspects of the simulation, including process functions, resources, simulation time, and event scheduling, are managed by SimPy's `Environment` class. Once the `Environment.run()` method is executed, the simulation is activated.

In order to simulate dataflow architectures in HLPerf we use process functions to represent individual stages. Given that SimPy offers three types of shared resources, we choose the `Store()` class to model FIFOs due to its capability to store Python objects. Figure 6.2 illustrates an example of employing SimPy for dataflow architectural simulation. For the sake of simplicity, this example considers only Stage1, Stage2, and a connecting FIFO1 with a capacity of two. These components are correspondingly modeled as `process1`, `process2`, and `store1`. In this case, assume `process1` has data to dispatch to `store1` at times t_0, t_1, t_2, and t_7, while `process2` retrieves data at t_3, t_4, t_5, and t_6, with $t_i < t_{i+1}$. Upon initiation of the simulation, a process from the "runnable" list is selected for execution. Assume `process1` is chosen and successfully sends data at t_0 and subsequently again at t_1, the FIFO reaches its full capacity, preventing the transmission at t_2 and resulting in the suspension of `process1`. Subsequently, `process2` is activated, retrieves data at t_3, and is then suspended. As FIFO1 is no longer full, `process1` resumes and succeeds in transmitting data at t_3, and is suspended again. Next, `process2` then resumes, successfully retrieves data at t_4 and t_5, and is suspended. At t_6, `process2`'s attempt to retrieve data fails due to FIFO1 being empty, prompting `process1` to resume and successfully send data at t_7, followed by `process2` retrieving data at the same time. Note that while ping-pong buffers are also used in some cases, they are not utilized in the benchmarks discussed in this paper. As such, they are not elaborated here. However, a ping-pong buffer can be implemented using two instances of SimPy's `Store` and simple switching logic. More details on constructing performance models in HLPerf based on SimPy are discussed in Section 6.1.4.

6.1.3 HLPerf Model Converter

The front-end converter takes the source code of the target HLS kernel and the intermediate results of the HLS scheduling and binding procedure as inputs, and automatically generates HLPerf models as outputs. Figure 6.3 depicts the workflow of the front-end converter. Initially, the converter preprocesses the HLS C source code of the target GNN kernel by parsing the

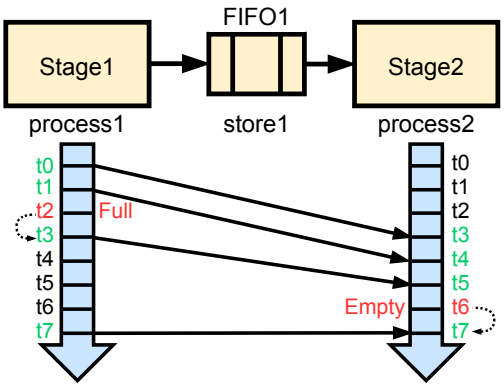


Figure 6.2: An example of SimPy for a dataflow architecture, where `process1` sends data to `store1` at t_0-t_2 and t_7; `process2` receives data from `store1` at t_3-t_6; and t_0 ¡ t_1 ¡ t_2 ¡ t_3 ¡ t_4 ¡ t_5 ¡ t_6 ¡ t_7.

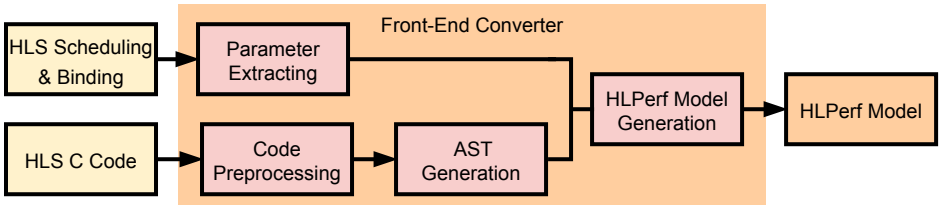


Figure 6.3: The workflow of the Front-End Converter in HLPerf.

code using `pycparser` [9], a generic python library for C language parsing. Since the source code incorporates 2 specific C++ data types: `hls::vector` and `hls::stream`, which are not inherently recognizable by `pycparser`, these data types are substituted with a placeholder C data type (e.g., `int`) during the preprocessing procedure. We note that this substitution does not influence the subsequent HLPerf model simulation. The irrelevance of vector data to the simulation stems from our approach of decoupling the simulation from functional correctness, one of the key advantages of HLPerf. Additionally, the stream type's relevance is mitigated as the front-end converter can identify FIFOs through the stream pragma (i.e., `#pragma HLS stream`), ensuring a seamless simulation process. Meanwhile, the converter also parses the the synthesis log file (e.g., `vitis_hls.log` in Vitis) to extract intermediate results of HLS scheduling and binding. For example, in `vitis_hls.log` it will report the II and latency L in the log information `Pipelining result : Target II = 1, Final II = 1, Depth = 75` where final II is 1 and latency is 75. Besides, the converter also extracts the pre-defined memory latency (e.g., set by parameter `m_axi_latency` in Vitis).

After parsing and preprocessing, the source code of the target HLS kernel is converted into an abstract syntax tree (AST) representation by `pycparser`. An AST is a tree representation of the syntactic structure of the source code, in which each node represents the information of a code part such as function, loop, statement, variable, etc, and each edge represents the relationships among different code parts. The root of the AST is the top-level kernel function. The front-end converter automatically generates the HLPerf model of the target GNN HLS kernel following AST traversal.

Algorithm 3 shows the pseudocode for the HLPerf model generation. This method accepts the AST representation of the target kernel and intermediate parameters as inputs. The process begins with the instantiation of a new model for the target High-Level Synthesis (HLS) kernel. Subsequently, the converter identifies and logs all memory-related arguments from the top-level kernel design by analyzing the `#pragma HLS INTERFACE` directive. In our case, these memory ports adhere to the `m_axi` protocol and are essential for the subsequent integration of memory latency into the HLPerf model. Notably, among these arguments, `node_src`, the array representing source vertex indices, is uniquely tied to real input data as it reflects the irregular topology of the input graph. The converter then proceeds to locate all dataflow functions, starting from the root. These functions are identified by the `#pragma HLS dataflow` directive. Typically, each HLS kernel in our GNN applications corresponds to a single dataflow function. For each identified dataflow function, a new environment within the model is established. Given that dataflow functions in our applications consist of two node types, variables with a stream pragma denoting FIFOs and functions representing stages in the dataflow architecture, they are processed distinctly. A new FIFO instance for each stream variable is created within a unified model environment. Moreover, as each function generally encompasses several loops, a new process within the model is created for each function, with a detailed examination of its arguments to discern FIFOs and memory-related arguments. Subsequently, a recursive Depth-First Search (DFS) algorithm traverses all `for` loops within the function. For each loop, the boundary is first analyzed, with distinct processing based on the boundary's type (e.g., constant or variable). The pragmas within the loop are then checked, and the loop body is replaced with pragma-driven patterns. Finally, the intermediate parameters from HLS are incorporated into the loop. Upon the comprehensive processing of all dataflow functions, the HLPerf model is automatically generated.

Algorithm 3: HLPerf Model Generation Algorithm

input : AST of kernel's source code
input : Intermediate parameters of HLS
output : Generated HLPerf model

```
// Create a new model instance
model ← createModel();
S_m ← Find all the top-level memory arguments via #pragma HLS INTERFACE;
S_d ← Find all the dataflow functions starting from root;
foreach df ∈ S_d do
    // create a new environment
    env ← model.createEnv(df);
    foreach node ∈ df.childList do
        if node.pragma is stream then
            // create a new fifo instance
            model.addFIFO(env, node);
        else if node.type is function then
            // create a new process instance
            func ← model.addProcess(env, node, S_m);
            while Recursively DFS traverse all the for loops ∈ func do
                Analyze the boundary of the loop;
                Replace loop body with pragma-driven patterns;
                Load intermediate parameters of HLS;
// Save the generated HLPerf model
model.save();
```

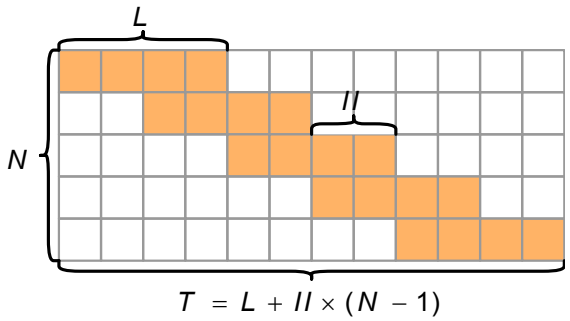


Figure 6.4: Diagram of the latency (L), initiation interval (II), and iteration number (N) of a pipelined loop.

6.1.4 Pragma-Driven Pattern Modeling

In GNN implementations, the HLS kernels are invariably loops, which can have their performance analyzed analytically without any need for full cycle accuracy. In HLPerf, we build pragma-driven patterns based on analytical expressions [26] to model the performance impact of various optimizations on loops in each stage of the dataflow architecture. A pragma driven pattern consists of 3 parameters: **Latency** (L) represents the number of cycles for an input element to propagate from the entry to exit of a pipeline. **Initiation interval** (II) denotes the number of cycles required for successive input elements to be ingested. **Iteration number** (N) of the loop represents the number of data elements to be processed by the pipeline. Figure 6.4 shows the relationship of these 3 parameters in a pipeline loop where L, II, and N are 4, 2, and 5 respectively. The overall execution time is $T = L + II \times (N - 1)$. In essence, the performance benefit of the prevalent optimization techniques will be attributed to their impact on these parameters. Thus, applying distinct optimization techniques to the same loop will result in different parameterizations of this quantitative expression. To quantify the parameterization, we use SimPy's `Environment.timeout()` method which is a `yield` statement to schedule the execution of the HLPerf model by suspending a process for a given time. Note that for loops without any pragmas, we use the number of unoptimized operations in the loop body to schedule the process. Here, we use code snippets of the GCN kernel as examples to illustrate how these optimization techniques affect the kernel performance and the mapping between HLS kernel and HLPerf models.

Pipeline is one of the essential optimization techniques in the HLS community for effective hardware acceleration. It enables instruction-level concurrent execution to to reduce II

```
1  loop1: for(uint64_t e=tmp_begin;
       e<tmp_end; e++){
2    vec vec1 = fifoIn.read();
3    loop1.1: for(int i=0; i<2; i++){
4      #pragma HLS pipeline II=1
5      //aggregate feature of each
           neighbor
6    }
7  }
8  loop2: for(int i=0; i<2; i++){
9    #pragma HLS pipeline II=1
10   fifoOut << vec_agg[i] // write the
         aggregated result.
11 }
```

Listing 6.1: Aggregation module in HLS C.

```
1  for e in range(deg):
2    _ = yield fifoIn.get()
3    yield env.timeout(L[0]+II[0])
4  for i in range(2):
5    if i == 0:
6      yield env.timeout(L[1])
7    else:
8      yield env.timeout(II[1])
9    fifoOut.put(1)
```

Listing 6.2: Aggregation module in HLPerf.

and improve the overall throughput. Vitis provides a pragma, `#pragma HLS pipeline`, to convert a regular sequential loop to a hardware pipeline whose execution is illustrated in Figure 6.4. To figure out how to map HLS C with pipelined loops to the HLPerf model, we use the aggregation module in the GCN kernel as an example, Listing 6.1 and Listing 6.2 show the corresponding code snippet in HLS C and HLPerf, respectively. There are 3 loops in Listing 6.1. The first, `loop1`, is a non-perfect loop with a variable boundary representing the in-degree of each node. After reading the neighbors' feature vectors from the input FIFO (i.e., `fifoIn`) which is implemented as a `stream` class in Vitis (line 2), these features are aggregated in `loop1.1` to which the pipeline pragma is applied. The equivalent HLPerf model is shown in Listing 6.2 (lines 1-3). The input FIFO is realized by the `store` class of SimPy and the read operation is performed via the `get()` method. We note that although

FIFO operations are performed, the accessed data is ignored because we are only focusing on performance estimation. Now that `loop1.1` doesn't contain any FIFO operations, we can use the expression illustrated in Figure 6.4 to calculate its overall execution time serving as the delay, which is simulated via the `timeout()` method of the SimPy environment.

The remaining loop in Listing 6.1, `loop2`, is also a pipelined loop designed to write the result feature vector to `fifoOut`. However, since it contains FIFO accesses, we cannot use a single formula to build the HLPerf model. Because the first input data needs to pass through the pipline before being written to the output FIFO, we establish a loop with iteration-level delay analysis, shown in Listing 6.2 (lines 4-9), where the time spent on generating the first and the rest of output data is L and II, respectively.

```
1 loop1: for(int k=0; k<FT/D; k++){
2   loop1.1: for(int kd = 0; kd < D;
        kd++){
3     #pragma HLS pipeline II=1
4     #pragma HLS unroll factor=2
5     // computation details of grouped
          VMM
6   }
7 }
```

Listing 6.3: Grouped vector-matrix multiplication in HLS C.

```
1 yield env.timeout(L + II * (FT/D *
    D/2 - 1))
```

Listing 6.4: Grouped vector-matrix multiplication in HLPerf.

Loop Unroll leverages instruction-level parallelism by executing multiple copies of loop iterations in parallel to increase throughput at the cost of resource utilization. It is enabled with `#pragma HLS unroll`. In essence, this pragma reduces the number of data elements to be processed sequentially. Thus, it improves the kernel performance by reducing N. The `factor` option represents the number of generated hardware replications. Here we use the core loop of the grouped VMM module as an example to illustrate its HLPerf Model. Listing 6.3 shows the HLS C code where `loop1.1` is unrolled by a factor of 2. The corresponding HLPerf model is shown in Listing 6.4 where the impact of the unroll pragma is to reduce the iteration number by 2.

Loop Merge optimizes the finite state machine (FSM) of nested loops to remove the impact of inner loop latency on the overall throughput. This optimization technique is usually automatically inferred by Vitis. In essence, its performance benefit is enabling the latency of inner loops to be counted at every iteration of the outer loops. Let's still take the grouped VMM module as an example. In Listing 6.3, `loop1` and `loop1.1` are merged automatically by Vitis because this nested loop is a perfect loop. Therefore, in the HLPerf model shown in Listing 6.4, L is counted only once. Note that for the loops containing FIFO operations, the HLPerf model can be built by adding the iteration index of the outer loop into the if condition of the latency.

```
1 loop1: for(uint64_t e=tmp_begin;
      e<tmp_end; e++){
2   uint64_t tmp_src = fifoIn.read();
3   loop1.1: for(int i=0; i<2; i++){
4     #pragma HLS pipeline II=1
5     fifoOut << ft_in_mat[tmp_src*2+i];
6   }
7 }
```

Listing 6.5: Memory read module in HLS C.

```
1 for e in range(deg):
2   _ = yield fifoIn.get()
3   yield env.timeout(L_mem) # memory
        latency
4   for i in range(2):
5     if i == 0:
6       yield env.timeout(L)
7     else:
8       yield env.timeout(II)
9     yield fifoOut.put(1)
```

Listing 6.6: Memory read module in HLPerf.

Memory Burst Access enables large chunks of data accesses in contiguous address to be executed in burst mode to improve the overall memory bandwidth. During memory burst accesses, the memory latency (i.e., the accumulated latency of the DDR controller, AXI interconnect, M-AXI adapter, and kernel design) is paid only for the first data element and the successive data elements are accessed at every clock cycle. This optimization technique and associated parameters (e.g., burst length) are automatically inferred by Vitis. From a

high-level perspective, the performance benefit of burst access lies in moving the request latency out of the memory access loop, which is similar to the principle of Loop Merge.

Taking a memory read module as an example, which enables memory read accesses in burst mode, the HLS C kernel and HLPerf model are shown in Listing 6.5 and Listing 6.6, respectively. In Listing 6.5, there is a nested loop. Since it is an edge-wise operation, the indices of the source node need to be read from the input FIFO `fifoIn` in the order of edges (lines 1-2). Then in `loop1.1` each feature vector is read from memory according to the source node index (i.e., `ft_in_mat`) and written to the output FIFO `fifoOut`. Because of the emergence of `fifoIn` read operation (line 2), induced by the inherent irregularity of graph topology, `loop1` is a non-perfect loop. Therefore, the burst mode is only inferred in `loop 1.1` and the burst length is the feature size d. Listing 6.6 shows the equivalent HLPerf model. Since the burst access is constrained in the inner loop (lines 4-9), the impact of memory latency L_mem is applied prior to the start of the inner loop. Therefore, the timeout statement of L_mem is placed at line 3, between the outer loop and the inner loop.

```
1 loop1: for(uint64_t e=tmp_begin;
     e<tmp_end; e++){
2   #pragma HLS pipeline II=1
3   uint64_t tmp_src = fifoIn.read();
4   fifoOut << ft_in_mat[tmp_src];
5 }
```

Listing 6.7: Pipelined memory read requests in HLS C.

```
1 yield env.timeout(L_mem) # memory
     latency
2 for e in range(deg):
3   _ = yield fifoIn.get()
4   if e == 0:
5     yield env.timeout(L)
6   else:
7     yield env.timeout(II)
8   yield fifoOut.put(1)
```

Listing 6.8: Pipelined memory read requests in HLPerf.

Memory Port Widening increases the memory port width of the kernel to improve the throughput of memory access logic. Users can enable it by defining the memory port

arguments of the top-level function using the `vector` data type, so that the kernel can fetch or store the whole vector instead of a single data element (e.g., `float`) at a time. Therefore, this optimization technique improves the performance by reducing the number of iterations N of the memory access loop, which is similar to the principle of Loop Unroll from a high-level perspective. Thus, after applying this optimization technique to loop1.1 in Listing 6.5 by setting the vector length to $d/2$, the iteration number is reduced from d to 2.

Furthermore, beyond its impact on N of the memory loop, widening memory ports provides more opportunities to improve memory access throughput for GNN implementations by enabling pipelined memory requests. For example, Listing 6.7 and Listing 6.8 show the HLS C design and the HLPerf model of the Memory Read Module after setting the vector length to d. As can be seen, the nested loop is changed to a regular pipelined loop, meaning that the memory read requests are pipelined. Therefore, in the HLPerf model in Listing 6.8, we can put the `timeout` statement of L_mem (line 1) to the outside of the loop1 (line 2) so that L_mem will not affect the performance of loop1 at each iteration. Note that given the width of physical pins on FPGA is limited (e.g., 512 bits), it will result in the increase of II if the memory port width is higher than the physical boundary. Therefore, there might be a trade-off to widening the memory ports in some cases.

6.2 Developer Experience

Here, we describe a number of potential developer experience use cases that are enabled by HLPerf. In the first use case, as indicated above, HLPerf can be helpful in the tuning process that is ubiquitous in FPGA design efforts [16, 26, 78]. In designs for which the performance is data dependent (often the case in GNN computations, which are frequently sensitive to graph topology), the time required to adjust a potentially performance impacting parameter (e.g., alter a pragma) and understand its performance implications can be quite long in conventional workflows. HLPerf's approach of simulating performance exclusively, rather than including functional correctness, supports a shorter turnaround time, which provides overall benefit to the development process. When the designer is verifying algorithm correctness, a longer evaluation method is totally appropriate. When all they want to know is, "Is this approach faster or slower, and by how much?", re-evaluating functional correctness simply slows down the process.

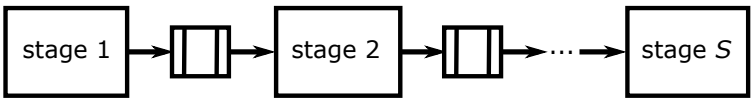


Figure 6.5: GNN dataflow pipeline.

In the second use case, HLPerf is usable prior to authoring HLS kernels. Figure 6.5 illustrates a generic computational pipeline, in which the contents of each pipeline stage are (as yet) undetermined. In fact, even the number of stages, S, has not yet been specified.

As the developer makes initial decisions about the number of pipeline stages (which will become HLS kernels) and the particular functions that will be performed at each pipeline stage, an HLPerf model can be developed that utilizes estimates of the performance parameters of each stage. These estimates might come from the developer's experience (i.e., they have written similar kernels in the past) or measurements of existing kernels (e.g., when library kernels are being invoked). This model can then be simulated to assess the performance impact of the data dependencies present in the input graph data set. In this case, it is incumbent on the developer to manually author the HLPerf models, rather than have them automatically derived from the HLS kernel code, which doesn't yet exist.

As an example of this second use case, consider a circumstance where a streaming computation is implemented across two execution platforms and data flows from the upstream portion to the downstream portion via a network link. This could be between two stages of a GNN model, or as part of any general-purpose streaming computation. To minimize the performance impact of the network, the author of the application chooses to compress the data moving across the network, and to maintain security the data is encrypted as well. The resulting communications link pipeline is shown in Figure 6.6.

Figure 6.6: Streaming data flow pipeline over a network.

Estimating the performance of individual stages of this pipeline by referencing the available AMD Xilinx libraries, one can readily build a HLPerf model of this communications link. We will show the performance implications of varying compression ratios on this pipeline below in Section 6.4.5.

As the development effort shifts to implementation of the HLS kernels themselves, the HLPerf model can be used to guide the kernel developer as to what should be the performance focus in each individual kernel's design. For example, if the HLPerf model indicates a particular stage in the pipeline is likely to be a performance bottleneck, the focus of the development can be on throughput. Alternatively, if the HLPerf model indicates some stage is unlikely to impact the overall throughput, the focus of the development can be on area savings. These types of area-speed trade-offs occur frequently in FPGA designs, and HLPerf can be helpful in providing guidance as to which direction the trade-off should be focused.

In this latter use case, the performance results from HLPerf are, of course, entirely dependent upon the quality of the individual kernel performance estimates provided by the developer. One possible path for the developer to pursue next is to prioritize the detailed design of individual HLS kernels that have significant performance uncertainty. Once authored, those kernels that previously had uncertain performance parameters can now exercise the HLS C Code to HLPerf Model link in Figure 6.1 to provide better knowledge of their individual performance to the HLPerf high-level model.

6.3 Evaluation Methodology

GNN Models: Figure 5.2 shows the 6 GNN models that are used to evaluate HLPerf. They include GCN [49], one of the earliest GNN models; GraphSage (GS) [38]; and Graph Isomorphism Network (GIN) [103] as isotropic models. Also included are Graph Attention Network (GAT) [90]; Mixture Model Networks (MoNet) [68]; and the Gated Graph ConvNet (GatedGCN) [13] as anisotropic models. We configure the feature dimensions for various GNN kernels as follows: GCN and GraphSage have the same input and output dimensions at 128. GIN is assigned input and output dimensions of 64. The input, head, and output dimensions of GAT and MoNet are (128, 8, 16) and (64, 2, 64), respectively. GatedGCN is set with a feature dimension of 32.

Datasets: Table 6.1 shows the graph datasets used in our evaluation. All these graphs are collected from Open Graph Benchmark [41], a widely-used graph library for GNNs, and have a wide range of fields and scales. These graphs represent two classes of graphs with distinct topologies used in the GNN community: MH and MT consist of multiple small dense graphs, while AX and PT each consist of one single sparse graph. The maximum and average

degree shown in Table 6.1 indicates their varying distributions ranging from regular-like to powerlaw-like. As mentioned above, we use GNNHLS, a benchmark suite of 6 GNN models described in Chapter 5, to evaluate HLPerf. HLPerf models are constructed with PyPy3 and SimPy [88] running on an Intel i7-8850H CPU at 2.6 GHz.

Table 6.7 lists the set of five general-purpose applications used to assess HLPerf on applications beyond GNN models. The data sets come from the original authors.

Table 6.1: Graph datasets [41].

Dataset	Node #	Edge #	Maximum Degree	Average Degree
OGBG-MOLTOX21 (MT)	145 459	302 190	6	2.1
OGBG-MOLHIV (MH)	1 049 163	2 259 376	10	2.2
OGBN-ARXIV (AX)	169 343	1 166 243	13 155	6.9
OGBN-PROTEINS (PT)	132 534	79 122 504	7 750	597.0

Evaluation methods: To perform evaluation, we use a Xilinx Alveo U280 FPGA card, provided by the Open Cloud Testbed [52], to execute the HLS kernels. This FPGA card provides 8 GB of HBM2 with 32 memory banks at 460 GB/s total bandwidth, 32 GB of DDR memory at 38 GB/s, and 3 super logic regions (SLRs) with 1205K look-up tables (LUTs), 2478K registers, 1816 BRAMs, and 9020 DSPs. We adopt 32-bit floating point as the data format, and Vitis 2023.1 for hardware emulation, synthesis, and hardware linkage.

6.4 Evaluation

As part of the evaluation, we will start by investigating the performance of the simulation execution within HLPerf. This will be followed by an assessment of the accuracy of the performance predictions, an illustration of the use of HLPerf for buffer size analysis, a description of HLPerf's utility in identifying and addressing performance bottlenecks within the application, and a look into the use of HLPerf on other applications.

6.4.1 Simulator Performance

We first examine the performance of HLPerf by comparing its simulation elapsed time with RTL simulation and several previous cycle-accurate simulators. We use Vitis hardware emulation mode to conduct the RTL simulation. Due to the low speed of RTL simulation, it will take a very long time on even the smallest-scale graph dataset MT used in this paper. Thus, in order to constrain the RTL simulation elapsed time to within 1 hour, we perform all the GNN kernels on just the first subgraph of MT with 16 nodes and 34 edges. Table 6.2 shows the simulation elapsed time and the speedup of HLPerf over RTL simulation. We observe that the speedup of HLPerf ranges from 1 200× to 35 700× across all the GNN applications, and the average speedup is 13 500×.

Table 6.2: Simulation elapsed time for HLPerf and RTL simulation, and the speedup of HLPerf relative to RTL simulation.

	HLPerf (s)	RTL Simulation (s)	Speedup
GCN	0.08	1 779	21 600×
GS	0.14	1 988	14 500×
GIN	0.10	3 538	35 700×
GAT	0.21	2 771	12 900×
MoNet	0.10	3 368	34 100×
GatedGCN	0.28	352	1 200×

Compared with the reported speed of the state-of-the-art cycle-accurate simulator, LightningSim [79], our work averages over 400× faster (primarily by giving up the requirement to be cycle accurate). In comparison to the reported performance of Flash [22], HLPerf exhibits an average speedup of just over 8×. Relative to FastSim [1], our methodology achieves an average speedup of over 300×. Note that although these comparisons might not be perfectly fair, for example Flash's C-based implementation is inherently more efficient than HLPerf's Python implementation and they are not all simulating the same designs, our work still achieves significant performance improvement. Note, the above comparisons are performed by examining the performance gain relative to RTL simulation as reported by the simulators' authors.

The performance benefit of HLPerf comes from 3 aspects: (1) it decouples the performance estimation from computational details of the algorithm, which is important for computation-heavy algorihtms like GNNs, (2) it simplifies the signal list to be simulated with a higher-level abstraction, and (3) HLPerf as a discrete-event simulator doesn't simulate every clock cycle.

These results indicate that the "approximately-cycle-accurate" approach of HLPerf can substantially diminish the time required to estimate performance for HLS GNN kernels.

Since increasing the graph scale leads to long simulation times, this superior speedup makes HLPerf quite suitable for design exploration of GNNs with large-scale graphs. Table 6.3 shows the elapsed time of HLPerf across all 6 GNN kernels and 4 graph datasets. In contrast, Table 6.4 presents the elapsed time of conventional HLS workflow procedures including HLS synthesis steps (preprocessing & transformation, scheduling & binding, and RTL generation) and hardware compilation. The duration of these procedures are independent of the input graphs. Comparing Table 6.3 and the first two rows of HLS synthesis from Table 6.4, we observe that for small-scale graphs the elapsed time of HLS synthesis is comparable to HLPerf simulation, while for large-scale graphs HLPerf simulation becomes increasingly dominant. A comparison between Table 6.3 and hardware compilation time in Table 6.4, reveals that the smaller the graph scale, the higher performance benefit of HLPerf to be achieved against hardware compilation. Even for the largest graph adopted in the benchmark, PT, the performance benefit of HLPerf over hardware compilation ranges from 4.4× to 122×. In addition, we note that HLPerf is based on the single-threaded Python library, and therefore its performance could potentially be further improved by utilizing multiple threads and a more efficient implementation.

Table 6.3: HLPerf simulation elapsed time of all the GNN kernels on 4 graph datasets.

	MT (s)	MH (s)	AX (s)	PT (s)
GCN	13	83	18	359
GS	31	230	42	364
GIN	24	158	36	298
GAT	68	542	158	6 097
MoNet	19	128	31	699
GGCN	71	426	285	4 808

Table 6.4: Elapsed time of conventional HLS workflow procedures, including HLS synthesis steps and hardware compilation.

Conventional HLS Workflow Procedures		GCN (s)	GS (s)	GIN (s)	GAT (s)	Monet (s)	GGCN (s)
HLS Synthesis	Preprocessing & Transformation	47	85	26	111	29	19
	Scheduling & Binding	58	95	51	146	55	59
	RTL Generation	77	168	83	251	94	139
Hardware Compilation		23 602	44 325	16 397	27 098	18 826	25 922

6.4.2 Application Performance Prediction Accuracy

We next quantitatively examine the accuracy of HLPerf by comparing the simulated execution time with the experimental execution time measured on the FPGA platform. For these experiments, and all those that follow, we are using the entire graph for performance evaluation, both in the HLPerf simulation and in the FPGA platform execution. We use measured execution time instead of RTL simulation for two reasons: first, it is the gold standard for understanding performance; and second, RTL simulation is prohibitively time-consuming and thus impractical for complex GNN HLS kernels with real-world graph datasets. We use error rate, defined as the percentage of the simulation result deviating from experimental measurement, to represent the accuracy of HLPerf. Figure 6.7 shows the normalized execution time predicted by HLPerf for all the 6 GNN kernels on 4 graph datasets relative to FPGA measurements, and Table 6.5 shows the absolute numbers for execution times and corresponding error rates. To enhance the clarity of comparison, in Table 6.5 we use the same clock frequency as on-board measurements. From the table, we observe that the error rate of HLPerf ranges from 3.3% to 14.7%, and is 7% on average. This level of imprecision is quite acceptable when assessing design alternatives that regularly exceed factors of 2 and more.

The observed inaccuracy in HLPerf's performance predictions can be be attributed to several factors. First, as HLPerf operates at a higher level of abstraction, it does not consider some low-level architecture details. For example, it cannot account for the impact of implementation strategies on onboard execution performance since it operates prior to hardware compilation. HLPerf relies on the estimated parameters (e.g., L and II) derived from the HLS report. These parameters are subject to further optimizations during the back-end compilation phase, which can affect the accuracy of HLPerf simulation. Furthermore, HLS synthesis does not provide all the detailed information needed for accurate HLPerf predictions. In scenarios with unoptimized loops where latency is not explicitly reported by HLS, operation count is used as a substitute for latency. Similarly, a default memory latency is applied in the absence of precise measurements, which are not available until actual execution on the FPGA board. Based on detailed examination of a small subset of the experimental runs, the reliance on latency and iteration interval estimates are the primary error contributor for compute-intensive kernels and the memory latency estimate is the primary error contributor for memory-intensive kernels.

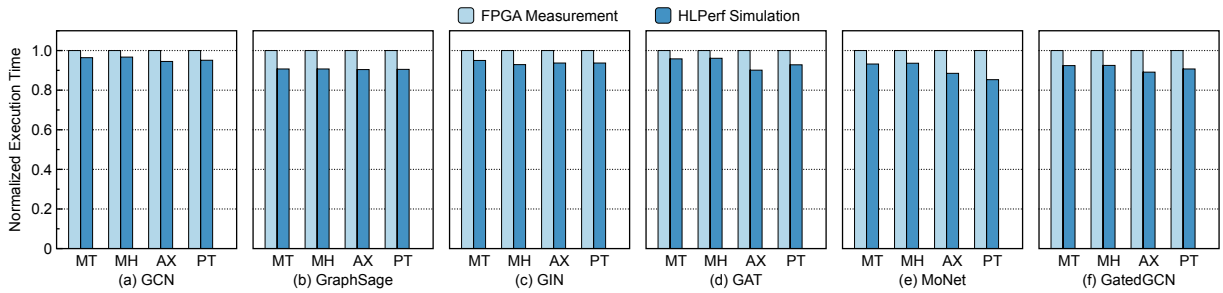


Figure 6.7: Normalized HLPerf predicted execution time relative to measurements on physical FPGA. The closer to 1 the better.

Although the simulated and experimental results are not exactly the same, the timing results of HLPerf track the FPGA execution time closely across graphs with various topologies and GNN kernels with distinct structures, indicating that HLPerf is able to recognize the inherent irregularity of the graph datasets and the algorithm.

Table 6.5: Execution time of FPGA measurements (HLS), predicted execution time from HLPerf, and corresponding error rate relative to FPGA measurements.

	MT			MH			AX			PT		
	HLS (s)	HLPerf (s)	Error Rate	HLS (s)	HLPerf (s)	Error Rate	HLS (s)	HLPerf (s)	Error Rate	HLS (s)	HLPerf (s)	Error Rate
GCN	0.10	0.10	3.6%	0.74	0.71	3.3%	0.39	0.36	5.5%	25.72	24.44	4.9%
GraphSage	0.16	0.14	9.3%	1.17	1.06	9.3%	0.62	0.56	9.6%	40.24	36.42	9.5%
GIN	0.08	0.08	5.0%	0.63	0.59	7.1%	0.33	0.31	6.3%	22.04	20.65	6.3%
GAT	0.23	0.22	4.2%	1.65	1.59	3.9%	0.74	0.66	9.9%	31.10	28.86	7.2%
MoNet	0.08	0.07	6.8%	0.55	0.51	6.4%	0.09	0.08	11.5%	1.37	1.17	14.7%
GatedGCN	0.10	0.09	7.6%	0.72	0.67	7.5%	0.39	0.34	10.9%	25.40	23.04	9.3%

Turning our attention to the impact of graph topology on the accuracy of HLPerf, we find that HLPerf achieves lower error on regular-like graphs than powerlaw-like graphs. According to Table 6.5, the average error rate of GNN applications on MT, MH, AX, and PT are 5.8%, 5.9%, 8.6%, 8.1%, respectively. We believe this to be because irregular graphs (i.e., PT and AX) with higher edge-to-node ratio and variation of degree distribution brings increased uncertainty to the performance predictions. In contrast, regular graphs contain less irregularity. Thus, the performance predictions have lower uncertainty.

Note that there is always a trade-off between simulation speed and accuracy. Therefore, although HLPerf is less accurate than full cycle-accurate simulators (e.g., LightningSim reports a 0.1% error rate on average compared with RTL simulation on an example dataset, while Flash and FastSim report 0% error rate relative to RTL simulation on their respective benchmarks), HLPerf matches the "approximately-cycle-accurate" goal in which the simulation speed

can be dramatically improved at the cost of a small accuracy loss. We contend that when design choices can have as dramatic a performance impact as three orders of magnitude (e.g., see [15, 32, 78, 85]), this is a trade-off well worth considering.

6.4.3 FIFO Size Sensitivity

One of the clear benefits of a simulation-based performance modeling approach is that one can effectively observe more than just aggregate performance. Here, we will illustrate the use of HLPerf to assess whether or not inter-stage buffers (i.e., FIFOs) have been allocated enough storage so as to not become a performance bottleneck themselves. In order to investigate the effectiveness of HLPerf for tuning FIFO size, we build a micro-benchmark based on a common property across all the GNN applications. We do this, instead of directly using GNN applications, because the applications' dataflow architecture contains sufficiently many stages and FIFOs as to make it challenging to illustrate the technique.

This micro-benchmark is a dataflow architecture with 2 functions and 1 FIFO. The first function represents an edge-wise aggregation operation and the latter one is a node-wise update operation. These two functions are connected by a FIFO. We use a synthetic graph to make the FIFO alternatively full and empty, reproducing the FIFO size issue. Figure 6.8 shows the cycle counts of RTL simulation and HLPerf running on the micro-benchmark with the FIFO size ranging from 2 entries to 15 entries. We observe that HLPerf's performance predictions are sensitive to the FIFO size in a manner that is quite close to the more detailed simulation available at the RTL level. In addition, optimized FIFO sizes achieved from RTL simulation and HLPerf are 8 and 11, respectively. Although these optimized FIFO sizes are not the same because of the accuracy loss of HLPerf, it requires less iterations for users or heuristic tuning algorithms to find the optimized FIFO size if starting from HLPerf. Note that HLPerf, as a high-level software simulator designed for performance prediction, does not encompass functional verification. Consequently, it may not identify certain potential deadlock issues related to functional correctness.

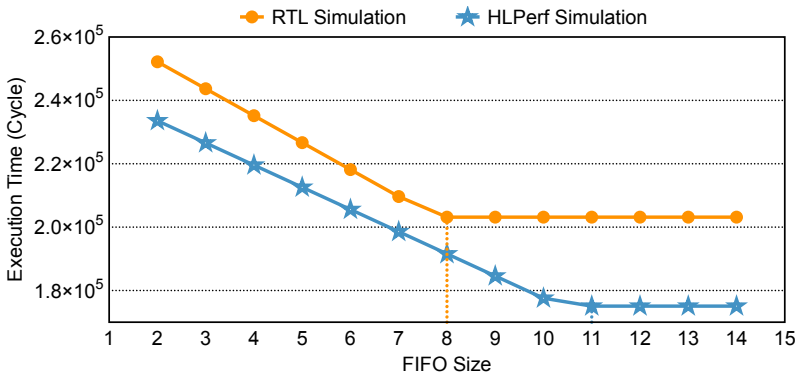


Figure 6.8: FIFO size sensitivity.

6.4.4 Identifying Performance Bottlenecks

HLPerf can not only estimate overall application performance but can also identify the bottleneck kernel of a dataflow pipeline. By comparing the simulation results of each function without stalls and the simulation results of the whole dataflow architecture, the bottleneck can be identified. Here we take GCN as an example. After performing HLPerf on GCN and 4 graph datasets, we find that the bottleneck of the GCN kernel on regular-like graphs, MT and MH, is the node-wise update operation implemented as a vector-matrix multiplication. On the other hand, on powerlaw-like graphs such as AX and PT, the bottleneck function is the edge-wise memory read module. After replacing this latter kernel with the pipelined memory read request enabled, as illustrated in Listing 6.7, the GCN kernel is accelerated by 2.6× and 5.3× on AX and PT, respectively, and the cycle counts of GCN don't change on MT and MH (the execution time changes because of distinct clock frequency inferred by the HLS tool). Table 6.6 shows the execution time, error rate, and simulation time of HLPerf for the modified GCN kernel. From the table, we observe that the error rate ranges from 1.9% to 6.9% and the average error rate is 2.8%, which matches the accuracy of HLPerf on other GNN applications. Hence we conclude that HLPerf can be used to tune the GCN HLS kernel for better performance.

6.4.5 General-Purpose Application Evaluation

While the main motivation for the development of HLPerf is understanding the performance of GNNs, the techniques used are not limited exclusively to GNNs. In general, the set

Table 6.6: Execution time, error rate, and simulation time of HLPerf for the GCN Kernel with pipelined memory requests.

	Execution Time (s)		HLPerf Performance	
Graphs	HLS	HLPerf	Error Rate	Simulation Time (s)
MT	0.09	0.09	2.3%	10
MH	0.68	0.67	1.9%	68
AX	0.11	0.11	6.9%	17
PT	3.82	3.74	2.1%	179

of applications to which HLPerf is applicable is constrained to dataflow architectures for which the developer has (or can learn) information about the performance of the constituent dataflow stages. In effect, if the performance estimates (e.g., latency and initiation interval) that come out of the high-level synthesis compilation process are accurate, HLPerf can be used to understand end-to-end performance.

In order to evaluate the ability of HLPerf to be used on other HLS kernels, beyond GNN models, we adopt 5 general-purpose benchmarks with dataflow architectures used in previous work [79]. Distinct from GNN applications, which are complicated irregular workloads with large-scale graph input datasets, these benchmarks have small enough input datasets that an RTL simulation with the full dataset is reasonable. Therefore, we use the results of RTL simulation as the baseline for comparison purposes.

Table 6.7 (left) presents both the predicted application execution time by HLPerf and RTL simulation, along with the accuracy of HLPerf relative to RTL simulation. From the table, we observe that the error rate ranges from 0.01% to 9.5% with an average of 2.0%, which is consistent with the results in Section 6.4.2.

Table 6.7: Predicted execution time and simulation elapsed time of RTL simulation and HLPerf, and corresponding error rate relative to RTL simulation for 5 general-purpose applications. The performance speedup of HLPerf and LightingSim are relative to RTL simulation.

	Predicted Results (cycles)			Simulation Elapsed Time (s)			
	RTL Sim	HLPerf	Error Rate	RTL Sim	HLPerf	HLPerf Speedup	LightingSim Speedup [79]
Vector Accumulator [101]	4 642	4 239	8.7%	37.18	0.1981	188×	15.2×
Cascade Adder [102]	9 212	8 333	9.5%	41.04	0.2292	179×	7.3×
Parallel Merge Sort [47]	73	68	6.8%	38.63	0.0027	14 062×	30.7×
Block Matrix Mult. [47]	226	216	4.4%	21.28	0.0032	6 590×	14.3×
Multi-Stage FFT [47]	7 208	7 209	0.01%	87.83	0.0139	6 338×	29.7×

Table 6.7 (right) details the simulation elapsed time of RTL simulation and HLPerf, and speedup of both HLPerf and LightingSim [79] relative to RTL simulation. From the table, we find that HLPerf speedup over RTL simulation ranges from 188× to 14 062× and it is 112× on average faster than LightingSim. Among the benchmarks, Vector Accumulator and Cascade Adder exhibit lower speedup through HLPerf relative to the other general-purpose benchmarks. This discrepancy is attributed to these kernels having a higher number of scheduling events, triggered by Python generator functions, leading to extended duration of interleaving suspension and resumption of process functions as discussed in Section 6.1.2. In addition, the speedup of HLPerf over RTL simulation for these two small benchmarks is lower than that observed for the GNN HLS kernels. This is because the performance of RTL simulation is closely related to the computational details of the HLS kernels, while HLPerf's efficiency depends on the quantity of processes and events, rather than the computational intricacies within the dataflow architecture. Hence, HLS kernels of varying computational complexity might yield similar performance under HLPerf models.

Returning to the network communications link example of Section 6.2, prior to deployment of the individual kernels, we have estimates of their performance because they are available as library elements. Similarly, we can estimate the performance of a network link from the literature (e.g., see [33]). What we may not know, however, early in the design, is the impact of the compression stage on the overall performance. How does the compression ratio achieved in the `compress` kernel influence the performance of the overall pipeline?

Since HLPerf does not perform functional simulation, and is therefore not actually executing the compression algorithm as part of the simulation process, this is clearly an example of what is being traded off to achieve faster simulation speeds. What we can do, however, is to sweep the compression ratio over a range of credible values and explore the performance implications for each value.

Figure 6.9 shows the results of simulating the pipeline of Figure 6.6, varying the compression ratio between $1:1$ and $8:1$. What we see is that the effective data rate (referenced at the input) increases as the compression ratio increases, up to a point of no additional benefit.

Here, while HLPerf cannot tell us the actual compression ratio, it does give the designer information about the performance implications of different compression ratios.

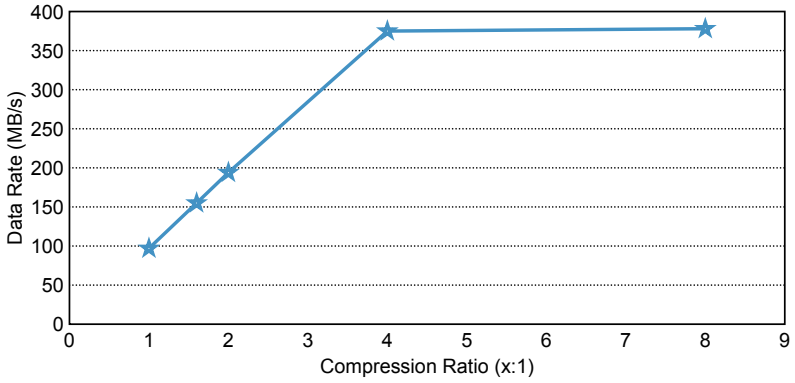


Figure 6.9: Streaming data pipeline with compression.

6.5 Conclusions

This paper has introduced HLPerf, an open-source, simulation-based performance evaluation methodology for dataflow architectures that is over 13 500× faster than RTL simulation and 400× faster than state-of-the-art cycle-accurate tools, on average, at the cost of 7% average error rate relative to FPGA measurements. This speed increase is attributable to three decisions inherent in HLPerf: (1) it does not try to be cycle-accurate, instead being satisfied with being "approximately" cycle-accurate, (2) it simplifies the signal list to be simulated with high-level abstractions, and (3) it does not try to verify functional correctness, focusing exclusively on performance prediction.

Because it is simulation-based, HLPerf can reflect the performance variations of computations that are dependent on input data sets. As such, it is well suited to evaluating the performance of graph neural network computations, which can be heavily influenced by graph topology. In the empirical evaluation, we showed that distinct HLS kernels in the dataflow architecture can be performance bottlenecks for regular-like graphs versus powerlaw-like graphs.

While the fast performance predictions supported by HLPerf are useful for optimizing the design of individual HLS kernels (e.g., choosing specific pragmas, etc.), HLPerf can also be an effective design tool prior to authoring any of the HLS code itself. By using performance estimates of individual kernels, HLPerf can analyze the performance of the dataflow pipeline as a whole, focusing the attention of the developer on potential bottleneck kernels earlier in the design cycle.

Chapter 7

Conclusions and Future Work

In this dissertation, we present our work in the field of efficient computation. We address 4 research questions regarding data integration, graph processing, and graph neural networks.

First, we characterize data integration workloads by examining their locality and the intensity of memory and arithmetic operations. Our analysis indicates that these workloads generally exhibit regular memory access patterns and variable computation intensities. We then construct three systems—a near-memory system, a baseline system, and a host system, each based on multiple Arm cores and Hybrid Memory Cube (HMC) technology—to assess the performance and energy consumption implications of each system. Our evaluations demonstrate that the near-memory system, equipped with a highly parallel architecture of 128 Arm Cortex-A7 cores integrated within the stacked memory logic layer, significantly outperforms the baseline and host systems, which utilize 16 Arm Cortex-A15 cores. Specifically, we observe an average speedup of 3.5× and an improvement in energy efficiency of 4.2×. This performance benefit can be attributed to the the availability of abundant parallelism and memory proximity. We conclude that near-memory processing holds substantial promise for enhancing performance and reducing energy consumption in data integration workloads. In the future, we anticipate that custom near-memory accelerators, specifically designed for these workloads, will achieve even greater efficiency and performance enhancements.

Second, we propose SuperCut, a framework for near-memory architectures to effectively reduce communication overheads while maintaining computational balance. SuperCut is comprised of the following 4 elements: (1) a set of graph partitioning algorithms to yield lower cross-cube communications volume and a balanced computational load; (2) a three-phase programming model to express general vertex programs on the basis of user-defined functions that is consistent with our graph partitioning algorithms; (3) an accelerator generator with which near-memory accelerators are generated via HLS and mapped to an FPGA fabric on the logic layer of the memory cubes; and (4) a custom graph representation that supports

the resulting graph applications on the NMP system while diminishing the irregularity of vertex traversal and data transfer. We evaluate SuperCut on an NMP architecture based on reconfigurable logic using 4 representative graph applications and 6 real-world graphs. Results show that it provides up to 1.8× total energy consumption reduction and 2.6× speedup with 45% lower extra memory footprint relative to the current state-of-the-art. In our future work, we aim to enhance the SuperCut framework through several improvements. (1) The iterative optimization algorithm in SuperCut is performed sequentially, leading to prolonged preprocessing times. To address this, we can parallelize the execution of multiple iterations of the algorithm, thereby significantly reducing the time required for this step. (2) The cost function being minimized is currently limited to considering communications volume alone. We would like to investigate incorporating a load balance factor into the cost function, which could enable a wider range of perturbation operations to be considered, actively pursuing load balance rather than simply trying to avoid introducing imbalance.

Third, we propose GNNHLS, an open-source framework to comprehensively evaluate GNN inference acceleration on FPGAs via HLS. GNNHLS comprises a software stack tailored for data generation and baseline deployment, alongside six finely-tuned GNN HLS kernels. We characterize these kernels in terms of instruction mix and memory locality scores and assess their performance across four graph datasets that vary in topology and scale. The results demonstrate a substantial performance improvement, with up to 50.8× speedup and 423× energy reduction when compared to multi-core CPU baselines. Against GPU baselines, GNNHLS achieves up to 5.16× speedup and 74.5× energy reduction. In the future, we plan to expand GNNHLS to include additional GNN models and graph datasets. This expansion will not only enhance the framework's utility but also serve as a benchmark for HLS researchers exploring the potential of HLS tools in accelerating GNN inference.

Forth, we introduce HLPerf, an open-source, simulation-based performance evaluation methodology for dataflow architectures that is over 13 500× faster than RTL simulation and 400× faster than state-of-the-art cycle-accurate tools, on average, at the cost of 7% average error rate relative to FPGA measurements. Future work will focus on assessing how well the techniques exploited by HLPerf can generalize to a wider set of other problems, outside the scope of GNNs, that have performance that is dependent on the input data set. This will require expansion of the model conversion (translating HLS C code to SimPy simulation models) and the pragma-driven pattern modeling (which is limited to the pragmas typically used in loop optimizations that are prevalent in GNN models). One approach is to rely

initially on manually authored HLPerf models, as we did in Section 6.4.5, which would enable us to assess the techniques without yet implementing the source-to-source compiler required to author HLPerf models automatically. With current design space exploration yielding performance variability over multiple orders of magnitude, a fast approach to performance estimation is an important tool. HLPerf seeks to do precisely that task.

References

[1] M. Abderehman, J. Patidar, J. Oza, Y. Nigam, T. A. Khader, and C. Karfa. FastSim: A fast simulation framework for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(5):1371–1385, 2021.

[2] S. Abi-Karam, Y. He, R. Sarkar, L. Sathidevi, Z. Qiao, and C. Hao. GenGNN: A generic FPGA framework for graph neural network acceleration. *arXiv preprint arXiv:2201.08475*, 2022.

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

[4] ARM Xilinx. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393). https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration, 2023. Accessed Aug. 2023.

[5] Arvind and D. E. Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1):225–253, 1986.

[6] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proc. of IEEE International Symposium on High Performance Computer Architecture*, pages 373–386. IEEE, 2019.

[7] L. Belayneh, A. Addisie, and V. Bertacco. MessageFusion: On-path message coalescing for energy efficient and scalable graph analytics. In *Proc. of IEEE/ACM International Symposium on Low Power Electronics and Design*. IEEE, 2019.

[8] L. Belayneh and V. Bertacco. GraphVine: Exploiting multicast for scalable graph analytics. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 762–767. IEEE, 2020.

[9] E. Bendersky. `pcyparser`. https://github.com/eliben/pycparser, 2023. Accessed Oct. 2023.

[10] M. Besta, D. Stanojevic, J. de Fine Licht, T. Ben-Nun, and T. Hoefler. Graph processing on FPGAs: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697*, 2019.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[12] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proc. of 23rd International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 316–331, 2018.

[13] X. Bresson and T. Laurent. Residual gated graph ConvNets. *arXiv preprint arXiv:1711.07553*, 2017.

[14] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[15] N. Brown. Exploring the acceleration of Nekbone on reconfigurable architectures. In *Proc. of IEEE/ACM Int'l Workshop on Heterogeneous High-performance Reconfigurable Computing*, pages 19–28, 2020.

[16] A. M. Cabrera and R. D. Chamberlain. Design and performance evaluation of optimizations for OpenCL FPGA kernels. In *Proc. of High-Performance Extreme Computing Conference*. IEEE, Sept. 2020.

[17] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain. DIBS: A data integration benchmark suite. In *Proc. of ACM/SPIE Int'l Conf. on Performance Engineering Companion*, pages 25–28, Apr. 2018.

[18] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson. From software to accelerators with LegUp high-level synthesis. In *Proc. of Int'l Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE, 2013.

[19] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing*, 5(3):13:1–13:39, 2019.

[20] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen. ThunderGP: HLS-based graph processing framework on FPGAs. In *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 69–80. ACM, 2021.

[21] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang. NXgraph: An efficient graph processing system on a single machine. In *Proc. of IEEE 32nd Int'l Conf. on Data Engineering*, pages 409–420. IEEE, 2016.

[22] Y.-K. Choi, Y. Chi, J. Wang, and J. Cong. Flash: Fast, parallel, and accurate simulator for HLS. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4828–4841, 2020.

[23] Y.-K. Choi, P. Zhang, P. Li, and J. Cong. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pages 691–698. IEEE, 2017.

[24] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):640–653, 2018.

[25] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proc. of 39th Conference on Programming Language Design and Implementation*, pages 752–768. ACM, 2018.

[26] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029, 2020.

[27] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[28] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[29] Z. Dong, W. Cao, M. Zhang, D. Tao, Y. Chen, and X. Zhang. CktGNN: Circuit graph neural network for electronic design automation. *arXiv preprint arXiv:2308.16406*, 2023.

[30] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The Mondrian Data Engine. In *Proc. of 44th ACM International Symposium on Computer Architecture*, page 639–651, June 2017.

[31] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *Journal of Machine Learning Research*, 23, 2020.

[32] C. J. Faber, S. D. Harris, Z. Xiao, R. D. Chamberlain, and A. M. Cabrera. Challenges designing for FPGAs using high-level synthesis. In *Proc. of High-Performance Extreme Computing Conference*. IEEE, Sept. 2022.

[33] C. J. Faber, T. Plano, S. Kodali, Z. Xiao, A. Dwaraki, J. D. Buhler, R. D. Chamberlain, and A. M. Cabrera. Platform agnostic streaming data application performance models. In *Proc. of IEEE/ACM Workshop on Redefining Scalability for Diversely Heterogeneous Architectures*. IEEE, Nov. 2021.

[34] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[35] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proc. of 53rd Int'l Symp. on Microarchitecture*, pages 922–936, 2020.

[36] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. of 10th USENIX Symp. on Operating Systems Design and Implementation*, pages 17–30. USENIX, 2012.

[37] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. of 49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[38] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *Adv. Neural Inf. Process. Syst.*, 30, 2017.

[39] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen. FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge. In *Proc. of 56th Design Automation Conference*. ACM, 2019.

[40] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying scalable graph processing with a domain-specific language. In *Proc. of IEEE/ACM Int'l Symp. on Code Generation and Optimization*, pages 208–218, New York, NY, USA, 2014. ACM.

[41] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Adv. Neural Inf. Process. Syst.*, 33, 2020.

[42] J. Jang, J. Heo, Y. Lee, J. Won, S. Kim, S. J. Jung, H. Jang, T. J. Ham, and J. W. Lee. Charon: Specialized near-memory processing architecture for clearing dead objects in memory. In *Proc. of 52nd IEEE/ACM International Symposium on Microarchitecture*, pages 726–739, 2019.

[43] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *Proc. of Symposium on VLSI Technology*, pages 87–88, 2012.

[44] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim. HBM (high bandwidth memory) DRAM technology and architecture. In *Proc. of IEEE International Memory Workshop (IMW)*. IEEE, 2017.

[45] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer, 2011.

[46] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[47] R. Kastner, J. Matai, and S. Neuendorffer. Parallel programming for FPGAs. *arXiv preprint arXiv:1805.03648*, 2018.

[48] G. Kim, J. Kim, J. H. Ahn, and J. Kim. Memory-centric system interconnect design with hybrid memory cubes. In *Proc. of 22nd Int'l Conf. on Parallel Arch. and Compilation Techniques*, pages 145–155. IEEE, 2013.

[49] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *Proc. of Int'l Conf. on Learning Rep.*, 2017.

[50] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[51] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78, 1997.

[52] M. Leeser, S. Handagala, and M. Zink. FPGAs in the cloud. *Computing in Science & Engineering*, 23(6):72–76, 2021.

[53] J. Leskovec, L. Adamic, and B. Huberman. The dynamics of viral marketing. *ACM Trans. on the Web*, 1(1):5:1–5:39, 2007.

[54] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *Proc. of 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 631–636, 2006.

[55] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *Proc. of SIGCHI Conference on Human Factors in Computing Systems*, pages 1361–1370, New York, NY, USA, 2010. ACM.

[56] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[57] Z. Li, X. Chen, and Y. Han. GraphRing: an HMC-ring based graph processing framework with optimized data movement. In *Proc. of 59th ACM/IEEE Design Automation Conference*, pages 1063–1068, New York, NY, USA, 2022. ACM.

[58] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li. DeepBurning-GL: an automated framework for generating graph neural network accelerators. In *Proc. of 39th International Conference on Computer-Aided Design*, pages 72:1–72:9. ACM, 2020.

[59] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li. EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Trans. Computers*, 70(9):1511–1525, 2020.

[60] Y. C. Lin, B. Zhang, and V. Prasanna. GCN inference acceleration using high-level synthesis. In *Proc. of IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021.

[61] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[62] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *Proc. of USENIX Annual Technical Conference (USENIX ATC)*, pages 195–207, 2017.

[63] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad. Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design. In *Proc. of 29th International Conference on Field Programmable Logic and Applications*, pages 397–403. IEEE, 2019.

[64] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of ACM Int'l Conf. on Management of Data*, pages 135–146, New York, NY, USA, 2010. ACM.

[65] J. Malicevic, B. Lepers, and W. Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proc. of USENIX Annual Technical Conference*, pages 631–643, July 2017.

[66] J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. In *Proc. of 25th International Conference on Neural Information Processing Systems*, page 539–547. Curran Associates, Inc., 2012.

[67] B. J. Mirza, B. J. Keller, and N. Ramakrishnan. Studying recommendation algorithms by graph analysis. *Journal of Intelligent Information Systems*, 20(2):131–160, 2003.

[68] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model CNNs. In *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, pages 5115–5124, 2017.

[69] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In *Proc. of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468. IEEE, 2017.

[70] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In *Proc. of 43rd International Symposium on Computer Architecture*, page 166–177. IEEE, 2016.

[71] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, et al. PyTorch: An imperative style, high-performance deep learning library. *Adv. Neural Inf. Process. Syst.*, 32, 2019.

[72] I. B. Peng, J. S. Vetter, S. Moore, R. Joydeep, and S. Markidis. Analyzing the suitability of contemporary 3D-stacked PIM architectures for HPC scientific applications. In *Proc. of 16th ACM International Conference on Computing Frontiers*, pages 256–262, 2019.

[73] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch. Optimus Prime: Accelerating data transformation in servers. In *Proc. of 25th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, page 1203–1216, 2020.

[74] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 190–200, 2014.

[75] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proc. of Int'l Symp. on Workload Characterization*, pages 110–119. IEEE, 2014.

[76] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi. gem5-SALAM: A system architecture for LLVM-based accelerator modeling. In *Proc. of 53rd IEEE/ACM Int'l Symp. on Microarchitecture*, pages 471–482. IEEE, 2020.

[77] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *Proc. of 19th Int'l Conf. on World Wide Web*, pages 861–870, New York, NY, USA, 2010. ACM.

[78] A. Sanaullah, R. Patel, and M. Herbordt. An empirically guided optimization framework for FPGA OpenCL. In *Proc. of International Conference on Field-Programmable Technology*, pages 46–53. IEEE, 2018.

[79] R. Sarkar and C. Hao. LightningSim: Fast and accurate trace-based simulation for high-level synthesis. In *Proc. of 31st IEEE International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2023.

[80] B. C. Schafer and Z. Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2019.

[81] Y. S. Shao and D. Brooks. ISA-independent workload characterization and its implications for specialized architectures. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 245–255, 2013.

[82] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. N. Udipi. Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device. In *Proc. of 46th IEEE/ACM Int'l Symp. on Microarchitecture*, pages 198–209. IEEE, 2013.

[83] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra. Near-memory computing: Past, present, and future. *Microprocessors and Microsystems*, 71, Nov. 2019.

[84] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *Proc. of IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559. IEEE, 2014.

[85] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems*, 27(4):32:1–32:27, 2022.

[86] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016.

[87] L. Tang and H. Liu. Graph mining applications to social network analysis. In *Managing and Mining Graph Data, Advances in Database Systems*, volume 40, pages 487–513. Springer, 2010.

[88] Team SimPy. SimPy: Discrete event simulation for Python. https://simpy.readthedocs.io, 2023. Accessed Aug. 2023.

[89] E. Vasilakis. An instruction level energy characterization of Arm processors. Technical Report FORTH-ICS/TR-450, Foundation of Research and Technology Hellas, Inst. of Computer Science, 2015.

[90] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. In *Proc. of Int'l Conf. on Learning Rep.*, 2017.

[91] K. Vipin and S. A. Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys*, 51(4):72:1–72:39, 2018.

[92] H. Wang, F. Zhang, M. Zhao, W. Li, X. Xie, and M. Guo. Multi-task feature learning for knowledge graph enhanced recommendation. In *Proc. of the World Wide Web Conference*, pages 2000–2010, 2019.

[93] M. Y. Wang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *Proc. of ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[94] I. Watson and J. Gurd. A practical data flow computer. *Computer*, 15(02):51–57, 1982.

[95] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of HPC applications. In *Proc. of the ACM/IEEE Conference on Supercomputing*, 2005.

[96] B. Weisfeiler and A. Leman. The reduction of a graph to canonical form and the algebra which appears therein. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

[97] E. E. Witte, R. D. Chamberlain, and M. A. Franklin. Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel & Distributed Systems*, 2(04):483–494, 1991.

[98] L. Wu, H.-F. Yu, N. Rao, J. Sharpnack, and C.-J. Hsieh. Graph DNA: Deep neighborhood aware graph encoding for collaborative filtering. In *Proc. of International Conference on Artificial Intelligence and Statistics*, pages 776–787. PMLR, 2020.

[99] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE Trans. on Neural Networks and Learning Systems*, 32(1):4–24, 2020.

[100] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[101] Xilinx. Introductory examples for Vitis HLS. https://github.com/Xilinx/Vitis-HLS-Introductory-Examples, 2023. Accessed Feb. 2024.

[102] Xilinx. Vitis Data Center Acceleration Examples. https://github.com/Xilinx/Vitis_Accel_Examples, 2023. Accessed Feb. 2024.

[103] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *Proc. of Int'l Conf. on Learning Rep.*, 2019.

[104] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. HyGCN: A GCN accelerator with hybrid architecture. In *Proc. of IEEE Int'l Symp. on High Performance Computer Architecture*, pages 15–29, 2020.

[105] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[106] B. Zhang, R. Kannan, and V. Prasanna. BoostGCN: A framework for optimizing GCN inference on FPGA. In *Proc. of 29th Int'l Symp. on Field-Programmable Custom Computing Machines*, pages 29–39, 2021.

[107] M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An end-to-end deep learning architecture for graph classification. *Proc. of AAAI Conf. on Artificial Intelligence*, 32(1):4438–4445, 2018.

[108] 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 *Proc. of Int'l Symp. on High Performance Computer Architecture*, pages 544–557. IEEE, 2018.

[109] Y. Zhang, H. You, Y. Fu, T. Geng, A. Li, and Y. Lin. G-CoS: GNN-accelerator co-search towards both better accuracy and efficiency. In *Proc. of IEEE/ACM International Conference On Computer Aided Design*. IEEE, 2021.

[110] C. Zhao, R. D. Chamberlain, and X. Zhang. Graph partitioning for near memory processing. In *In-Memory Architectures and Computing Applications Workshop (iMACAW)*, July 2022.

[111] C. Zhao, R. D. Chamberlain, and X. Zhang. Supercut: Communication-aware partitioning for near-memory graph processing. In *Proc. of 20th ACM Int'l Conf. on Computing Frontiers*, pages 44–53, 2023.

[112] C. Zhao, Z. Dong, Y. Chen, X. Zhang, and R. D. Chamberlain. GNNHLS: evaluating graph neural network inference via high-level synthesis. In *Proc. of 41st IEEE International Conference on Computer Design*, pages 574–577. IEEE, Nov. 2023.

[113] C. Zhao, Z. Dong, Y. Chen, X. Zhang, and R. D. Chamberlain. Graph Neural Network High-Level Synthesis Benchmark Suite V1. https://doi.org/10.7936/6RXS-103645, Sept. 2023.

[114] C. Zhao, C. J. Faber, R. D. Chamberlain, and X. Zhang. HLPerf: Demystifying the performance of HLS-based graph neural networks with dataflow architectures. *ACM Transactions on Reconfigurable Technology and Systems*, 2024.

[115] C. Zhao, X. Zhang, and R. D. Chamberlain. Executing data integration effectively and efficiently near the memory. *IEEE Design & Test*, 29(2):65–73, 2022.

[116] A. Zhou, J. Yang, Y. Gao, T. Qiao, Y. Qi, X. Wang, Y. Chen, P. Dai, W. Zhao, and C. Hu. Brief industry paper: Optimizing memory efficiency of graph neural networks on edge computing platforms. In *Proc. of Real-Time and Embedded Technology and Applications Symp.*, pages 445–448. IEEE, 2021.

[117] A. Zhou, J. Yang, Y. Qi, Y. Shi, T. Qiao, W. Zhao, and C. Hu. Hardware-aware graph neural network automated design for edge computing platforms. *arXiv preprint arXiv:2309.10875*, 2023.

[118] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *Proc. of USENIX Symp. on Operating Systems Design and Implementation*, pages 301–316. USENIX, 2016.

[119] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian. GraphQ: Scalable PIM-based graph processing. In *Proc. of 52nd IEEE/ACM International Symposium on Microarchitecture*, pages 712–725, New York, NY, USA, 2019. ACM.

[120] M. Zink, D. Irwin, E. Cecchet, H. Saplakoglu, O. Krieger, M. Herbordt, M. Daitzman, P. Desnoyers, M. Leeser, and S. Handagala. The Open Cloud Testbed (OCT): A platform for research into new cloud technologies. In *Proc. of IEEE 10th Int'l Conf. on Cloud Networking (CloudNet)*, pages 140–147, 2021.