Title
Generalizing Programmable Accelerators for Irregularity

Author
Dadu, Vidushi

Publication Date
2022

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Generalizing Programmable Accelerators for Irregularity

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Vidushi Dadu

2022

ABSTRACT OF THE DISSERTATION

Generalizing Programmable Accelerators for Irregularity

by

Vidushi Dadu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Tony Nowatzki, Chair

Specialized accelerators are increasingly attractive solutions to continue expected generational performance scaling with slowing technology scaling. Existing programmable accelerators like GPUs are limited to regular algorithms; however, supporting irregularity becomes necessary to accelerate more advanced algorithms or those from challenging domains. Irregularity occurs when aspects of the execution depend on the data – these aspects can be control, memory, parallelism, and reuse. An example of data-dependent execution is joining two sparse lists where the branch outcome depends on data; this inherently couples computation with memory and precludes efficient vectorization – defeating the traditional mechanisms of programmable accelerators (e.g., GPUs). Data dependencies in computations like producer-consumer streaming dependencies require detrimental synchronization overhead in shared memory systems.

We aim to develop a family of compatible dataflow accelerator mechanisms. Our critical insight is that it is unnecessary to support arbitrary forms of irregularity because specializable data dependence forms are sufficient for many workloads (despite being more restrictive). We expose these data dependence forms as first-class citizens in the instruction set architecture (ISA) so that the corresponding hardware is specialized for efficiency while remaining flexible

to the requirements of applications.

This dissertation studies various applications from machine learning, graph processing, databases, and signal processing. First, we identified specializable forms for instruction and task-level irregularity (where a task is coarse-grained). These primitives are designed to be composable and extensible to enable continual evolution. Then, we used these forms to design a family of spatial architectures supporting irregularity with compatible features. Leveraging hardware flexibility, we also studied different algorithmic implementations and developed insights into the complex relationship between input, workload, and algorithmic choices on performance.

We evaluate the architecture by integrating a custom accelerator simulator into the cycle-level gem5 simulator. The input programs are written in a combination of C and intrinsics of traditional dataflow and our data dependence forms. The hardware components are implemented in Chisel with an industry 28-nm technology. Overall, our architectures achieve order-of-magnitude speedups over a similarly provisioned GPU while remaining within 30% of application-specific architectures. In graph processing, we achieved 5.7× speedup over the best prior domain-specific accelerator, using the flexibility to choose the optimal algorithm given the input workload and data characteristics. Our designed hardware can be a strong alternative to GPUs; the main compute offload accelerators today. Besides, our proposed features target irregular primitives and perform similarly to prior domain-specific architectures. We believe our work can be the first step to unifying the rapidly evolving irregular acceleration space.

The dissertation of Vidushi Dadu is approved.

Jingsheng Jason Cong

Puneet Gupta

Daniel Sanchez

Harry Guoqing Xu

Tony Nowatzki, Committee Chair

University of California, Los Angeles

2022

To Papa, Mummy, Shrey, and Anant

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation is possible with the support and guidance of many people.

First among them is my advisor, Tony Nowatzki. I have been extremely fortunate to be advised by Tony, who taught me to think about long-term vision, ask the right questions, and present my ideas effectively and passionately. I admire his skill for quickly filtering out bad ideas. He always motivated me to give my best and encouraged me even if things were not going well. I will miss our brainstorming discussions and barging at his office at random times. I hope I can continue doing impactful research the way he taught me.

My committee members have been instrumental in polishing my work. Thanks to Jason for giving his valuable suggestions throughout my Ph.D.; I also learned a lot from the discussions in RTML meetings. Jason gave me the opportunity to present at CDSC, which was a great platform to get feedback from a broader audience. Thanks to Harry and Puneet for providing me with critical feedback on my work and pushing me to evaluate the practical implications of my research. I really like Daniel's work; his papers have indirectly taught me a lot about graph processing, leading to one of my most rigorous work. Also, his feedback on my dissertation helped me to be more concise on my definitions.

Thanks to all PolyArch lab members – our reading groups were fun. I want to thank Jian for our collaborations and his life suggestions on the tragedies of having little kids. We never collaborated directly, but Zhengrong helped me with his gem5 skills frequently. It is fun to hang out with Sihao, Dylan, and Chris. I am grateful to my research collaborators: David Ott and Pratap Subramanium, for giving their industrial perspective during our meetings. I would like to thank Guy Van den Broeck and Arthur Choi for their insights and help with machine learning workloads.

My friends at UCLA made my journey enjoyable: Atefeh Sohrabizadeh, Akshay Utture, Arjun Akula, Siva Kesava, Aayush Jain, Ashutosh Kumar, Aishwarya Sivaraman, and Pradeep Dogga. I enjoyed my outings with Siva and Akshay. Arjun and Siva have always been helpful in all kinds of queries I had during my Ph.D.

Finally, this dissertation is dedicated to my parents (Alok and Sapna) and my brothers (Shrey and Anant). I am indebted to them for their love, support, and continuous encouragement throughout my years of study.

VITA

2015	Research Intern, Carnegie Mellon University, Pittsburgh, USA.
2016	Research Intern, Carnegie Mellon University, Pittsburgh, USA.
2017	B. Tech. in Electronics and Communication Engineering with Minors in Computer Science, IIT Roorkee, Roorkee, India.
2018	Teaching Assistant ("Computer Systems Architecture" course), Computer Science Department, UCLA.
2019	Research Intern, Intel, Hudson, MA.
2020	IEEE Micro Top Picks Award.
2020	HPCA 2021 Student Reviewer.
2021	Research Intern, Microsoft, Redmond, USA.
2021	IEEE Micro Honorable Mention Award.
2022	Software Engineering Intern, Google, Mountain View, USA.
2022	IEEE Micro Top Picks Award.
2022	Google Peer Bonus Award.
2022	CAL Reviewer.
2022	Outstanding Graduate Student Research Award.

PUBLICATIONS

Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In 2019 International Symposium on Microarchitecture (MICRO '52), ACM, New York, NY, USA, pages 924-939.

Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. "Towards General-Purpose Acceleration: Finding Structure in Irregularity," in IEEE Micro, 2020.

Jian Weng, Sihao Liu, Zhengrong Wang, **Vidushi Dadu**, and Tony Nowatzki. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 703-716.

Jian Weng, Sihao Liu, **Vidushi Dadu**, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: synthesizing programmable spatial accelerators. In 2020 IEEE 47th International Symposium on Computer Architecture (ISCA), pages 268-281.

Vidushi Dadu, Sihao Liu, and Tony Nowatzki. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). pages 595-608.

Vidushi Dadu, Sihao Liu and Tony Nowatzki. "Systematically Understanding Graph Accelerator Dimensions and the Value of Hardware Flexibility," in IEEE Micro, 2022.

Vidushi Dadu and Tony Nowatzki. Taskstream: accelerating task-parallel workloads by recovering program structure. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022.

CHAPTER 1

Introduction

Hardware specialization is a promising solution to continue generational scaling of performance. The idea is to tailor the underlying architecture according to the requirements of the target applications, resulting in orders-of-magnitude performance gains.

Specialization has already seen a spur in the research community as well as significant adoption in the industry. In research, examples pervade many domains, including graphs [94, 221, 274, 68, 15, 53, 209], AI/ML [265, 104, 17, 195, 210, 237, 114, 117, 142, 223, 242], databases [122, 258, 259, 115], systems [282, 75, 76, 78], and genomics [233, 81, 234, 50, 121, 269]). The industry has examples for molecular dynamics [215], cryptography accelerators in CPUs [108], digital signal processing (DSP) accelerators on mobile SoCs [21], and most predominantly, deep learning accelerators. On the one hand, matrix-multiply acceleration units have been integrated into existing programmable accelerators like CPUs/GPUs [31, 184, 190, 245, 159, 168]. On the other hand, there are several standalone accelerators, including TPU [113], Groq [12], and Amazon Inferentia [19]. GraphCore [116], Cerebras [131], etc. A common trend in academia and industry is focusing unwaveringly on application-specific accelerators.

Designs which are performance-robust across algorithms and domains would be valuable for economies of scale. For motivation, consider how drastically the deep learning domain has evolved in recent years. During the deep learning boom in 2012, AlexNet was the primary target [18]. Then, deep compression algorithms were developed, leading to sparse networks [99], newer layers like depth-wise separable convolution [55], and more recently, graph-based convolution networks [96] are popular. Besides, other evolving domains are equally important.

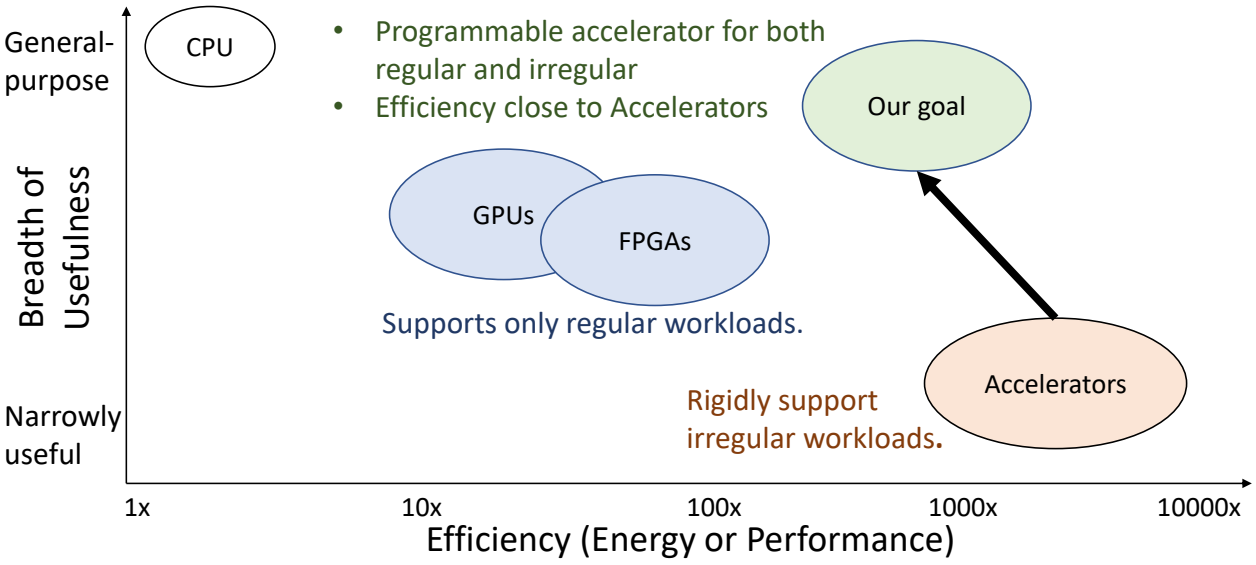


Figure 1.1: Generality vs Efficiency

For example, graph processing is used in pervasive google maps, SQL databases are critical for the cloud, signal processing has new requirements with 5G, and video processing is vital for AR/VR revolution [28, 74, 240]. An accelerator that can quickly adapt to changing workloads will reduce time-to-market and improve architecture longevity [93].

The success of the application-specific accelerators suggests that existing and prior general-purpose data-processing hardware (e.g., GPGPUs [168], Intel MIC [72] & KNL [219]) are orders-of-magnitude lower in performance and/or energy efficiency. The reason is irregularity: GPUs rely heavily on vectorization, which limits support for irregularity; thus, their efficiency is limited to regular programs. Similar is true for even more efficient spatial accelerators, which get efficiency by exposing their network in the ISA. However, both GPUs and academic architectures like Plasticine [188], LSSD [164], Vector-threads [125] and Softbrain [162] that are mostly limited to dense algebra. But why?

Irregularity is when the decision of which operation to perform or what value to operate on is influenced by data read by the program – more simply, it is data-dependent control flow or memory access. Conventional architecture mechanisms specialize in data independence and, therefore, are often inefficient in the presence of irregularity. Consider

a data-dependent branch, e.g., if(a[i]<4), where a[i] is a random variable with uniform distribution. Here the branch outcome does not follow any pattern, making branch prediction useless, leading to stalls in CPUs. In GPUs, a branch would cause SIMT lanes to diverge. We believe that future workloads will use complex data structures and thus be irregular. For example, newer deep learning models take more general graphs as input instead of regular images). Hence for accelerators to be future-proof, we must **broaden the scope of programmable accelerators to irregularity**. Figure 1.1 pictorially represents our goal to improve applicability of domain-agnostic accelerators to more workloads while achieving the efficiency of application-specific accelerators.

This dissertation explores critical irregular workloads and ultimately suggests that it is possible to accelerate broad irregular workloads without giving up efficiency. We identified forms of data-dependencies that are specializable and cover a wide variety of data-dependent kernels. We call these **specializable irregularity forms**. To this end, we designed a family of reconfigurable hardware features that provide flexibility across data-processing domains, and different algorithmic implementations of these domains (e.g., inner product or outer product in matrix multiplication).

We studied machine learning, graph processing, databases, and signal processing domains; however, our solutions may also apply to more domains. For example, the solution that we designed for data-dependent control in databases and sparse linear algebra also apply to the un-studied workload of graph mining (GM), and several subsequent works used this insight to build GM accelerators [194, 227, 69]. We also performed design space explorations that provided insights into the relationship between inputs, algorithms, and architecture techniques.

The remainder of this chapter covers the background of irregular algorithms and how different styles of architecture deal with irregularity (Section 1.1). Then, we will explain how specializable irregularity forms are a powerful tool to systematize the specialization of irregular workloads (Section 1.2). Finally, we describe our main contributions (Section 1.3) and conclude with the organization of this dissertation (Section 1.4).

Program Segment Gran.	Dependent Data Gran.	Execution Dim.
Instruction	Vector/Scalar	Control, memory
Fine-grained Task	Scalar	Schedule, Mem. Footprint
Coarse-grained Task	Vector	Task size, Inter-task dep.

Table 1.1: Classifying Irregularity by Data-dependence Granularity

1.1 Background

Here we define irregular workloads, describe kinds of irregularities, and provide insights on how irregularity occurs in practical workloads. Then, we provide background on existing architectures and discuss how they deal with irregularity.

1.1.1 Irregular workloads

Definition Irregularity is when the decision of which operation to perform or what value to operate on is influenced by data read by the program – more simply, it is data-dependent control flow or memory access. Conventional architecture mechanisms in interactions with computation, memory, and network, specialize for data-independent behaviors and, therefore, are often inefficient in the presence of irregularity. For example, the arbitrary branch outcome may cause pipeline stalls, random memory access locations may prevent efficient memory bandwidth/capacity utilization, and random remote traffic may cause congestion in the network.

Kinds of Irregularity Our insight is that data dependencies cause irregularity, and the key distinguishing factor for hardware design tradeoffs is the dependence granularity. For example, we can resolve fine-grained data dependence by passing data as live operands through on-chip queues, while coarse-grained data needs to be buffered in memory.

Thus, we classify irregularity by the granularity of the data and dependent program

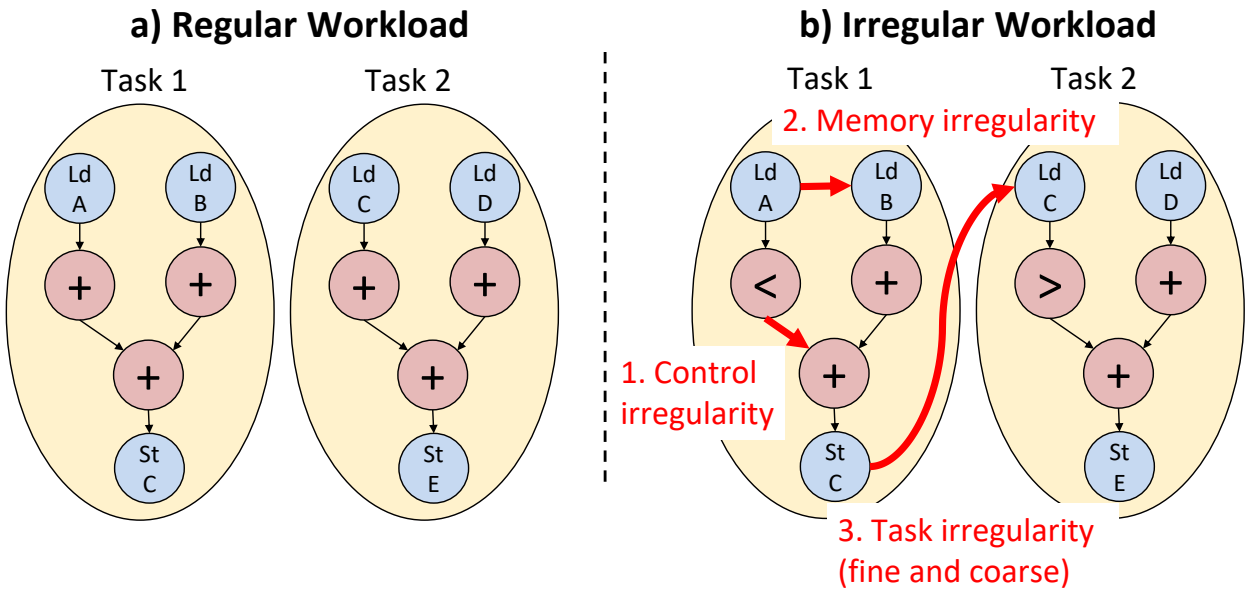


Figure 1.2: Example Regular and Irregular Programs

segments. Table 1.1 shows the dimensions: the data can be a scalar value or vector of data. The program segment can be a fine-grained instruction (e.g., ld,st,add) or a coarse-granularity task – a unit of work (involving several instructions) that can be scheduled in time and space. Figure 1.2 demonstrates the classification visually. A regular program has several independent tasks containing instructions with read-write dependence between only memory-to-compute and compute-to-compute. An irregular program, however, can have three additional dependence types: conditional operations, inter-task dependencies and cases when the address operand of a memory operation is transitively dependent on another memory load. Table 1.1 also lists the execution dimensions that may be impacted by each kind of irregularity. For example, irregularity at instruction granularity can be caused due to data-dependent control and memory. The scheduling order of fine-grained tasks and their memory footprint is critical. For coarse-grained tasks, their size and inter-task dependencies are more important factors.

One advantage of classifying by hardware requirements is that the resulting hardware

support is modular (i.e., specialization for data-dependence forms are in independent hardware components). For example, the irregularities in instruction granularity are resolved by enhancing the computation cores and on-chip memory for irregularity. While for tasks, the solution is the task management unit that interfaces to the accelerator's execution resources and enables better coarse-grained communication across cores. We will expand on the types of irregularity in Chapter 2.

Sources of Irregularity Irregularity may occur due to several reasons. We list four of them below with example algorithms in Figure 1.3:

Dataset Sparsity: One source of irregularity is a sparse dataset. Sparsity is common in machine learning: either due to practical data missing values during its collection or the data having redundancy (e.g., compression in deep learning [99]). Typical sparse workloads are deep neural networks, gradient boosting decision trees, and databases.

At instruction granularity, accessing only non-zero values may involve conditionals, and indirect memory accesses to work on only the non-zero data. At task granularity, the total number of non-zeros (hence required computations) may vary across tasks, causing load imbalance.

Graph Data Structures: Graphs model sparse connections and represent several practical scenarios like social media networks, road networks, drug models, etc. Two things differentiate graphs from traditional dataset sparsity: large size (usually have billions of non-zero values), and they are ultra-sparse (usually >99% sparsity). Thus, indirect memory access and effective on-chip memory use are more critical factors than conditional operations[1].

Data Reordering: When the purpose of an algorithm is to reorder data, random memory reads and writes cause memory irregularity (at instruction granularity). Examples include database algorithms like sorting and joins.

[1]The reason is that sparse computations require matching indices, and the more sparse a dataset is, lower the relative cost of using a single indirect access to locate a match versus performing an iterative search over many non-matching values.

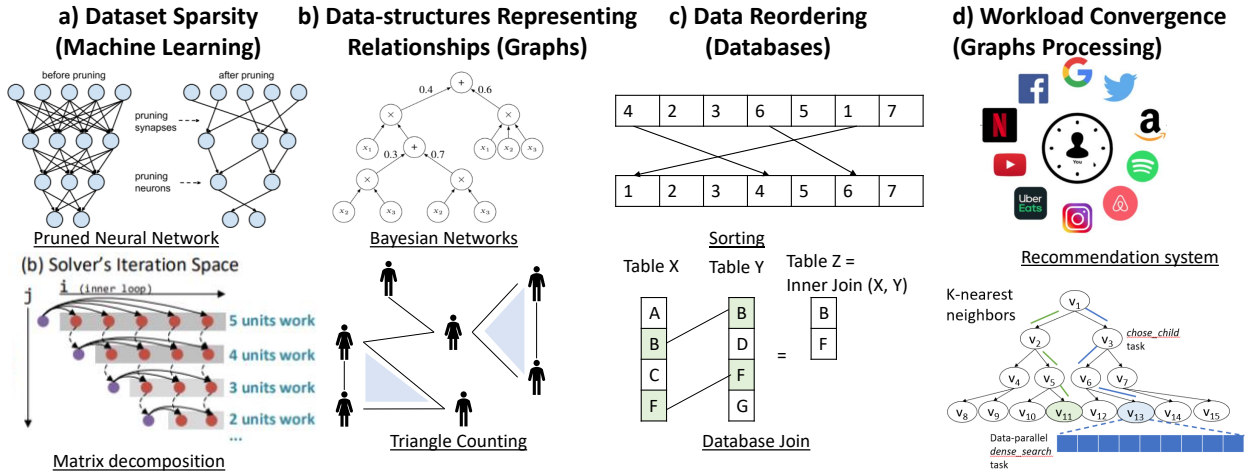


Figure 1.3: Sources of Irregularity and Example Workloads

Workload Convergence: One class of workloads is learning-based: they execute the next iteration based on the outcome of a conditional (while loop). An iteration usually involves a group of instructions and can be abstracted as a task; thus, a while loop would conditionally create a task, causing task irregularity. Examples include deep learning training and traditional graph processing workloads like shortest path and page rank.

Even though the workloads in Figure 1.3 and their source of irregularity are different, we show later that the data dependencies in these workloads manifest themselves in the hardware in common forms. Also, note that applications often display one or more forms of irregularity, therefore comprehensive support for data-dependence forms is beneficial.

1.1.2 Existing Architectures for Irregularity

Existing architectures do not perform well in the presence of irregularity: either they lose performance or resort to expensive general-purpose mechanisms. Figure 1.4 classifies architectures by their support for irregularity (higher "Efficiency" means better support) and applicability to multiple domains (higher "Generality" means broad applicability). This section will discuss how these architectures deal with different kinds of irregularities and give insights into their approach's pros and cons.

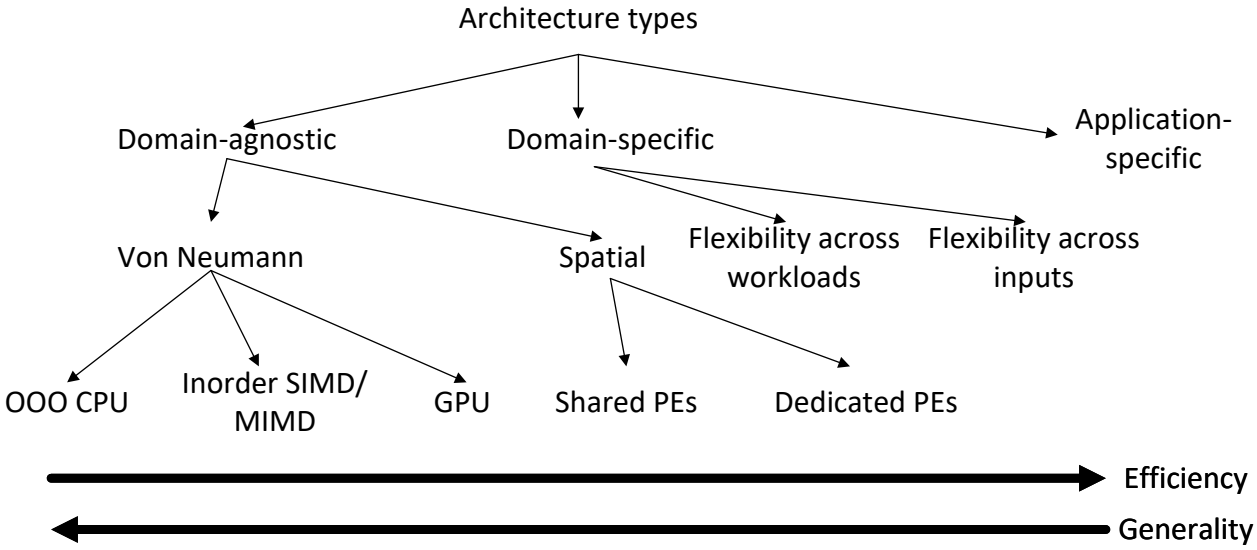


Figure 1.4: Fundamental Architecture Styles (Please note that we do not show hybrid architectures where application-specific modules are added in domain-agnostic architectures.)

1.1.2.1 Domain-agnostic Architectures

Domain-agnostic architectures have an ISA and can execute several workloads. Existing domain-agnostic architectures do not support irregularity well (either lose performance or resort to expensive general-purpose mechanisms). We classify them into instruction-based/vonNeumann and spatial architectures.

vonNeumann Architectures Here we discuss out-of-order, single-instruction multiple-data (SIMD), in-order, and multiple-instruction multiple-data (MIMD) processors.

Out-of-order (OoO) Processors: provide a view of sequential instruction execution but internally reorders the instructions for extracting parallelism. OoO mechanisms rely on patterns; for example, the branch predictor assumes history behavior, memory prefetcher can identify linear patterns, etc. However, data-dependent branches are unpredictable and adding a large instruction window and reorder buffer to hide stalls are expensive. In the case of unknown memory aliases, the number of concurrent memory accesses is limited by the size of load-store queues.

General-purpose Computing on Graphics Processing Units (GPGPUs) use the Single Instruction Multiple Threads (SIMT) execution model. The SIMT model exploits data-level parallelism using vector instructions and hides memory latency using multiple concurrent threads. In the presence of data dependencies in computation instructions, the vector lanes associated with a group of threads (warp) may diverge. Across warps, the threading model is fairly general and provides good performance for data-dependent memory accesses. However, hiding memory latency requires maintaining many threads, which is expensive: large register files are needed for the live state, and there is a burden on the programmer to express more parallelism. Finally, CPU and GPU do not support efficient communication among cores/streaming multiprocessors (SMs). In the case of dependencies among thread groups, either excessive coherence traffic or synchronization barrier lead to poor performance.

A **large array of in-order cores** is attractive for irregular algorithms, as it seems to alleviate the control problem (each lane/core is independent), the memory problem (each core reads a different memory item), and the computation problem (no vectors to pack). The downside is that data-processing algorithms still exhibit high locality and massive parallelism over data items, which cannot be exploited at low overhead in a general purpose in-order core. Moreover, they do not have spatial abstractions for computation, means that scalar multicores give up an extra dimension of efficient parallelism. Finally, there is no hardware support for task irregularity, and software scheduling is often too slow to scale to large amounts of parallelism.

Hybrid architectures like Vector-threads (VT) [125, 134, 198] propose a flexible SIMD/MIMD execution model. GANAX [265] applies some of the same principles but is specialized for ML. SIMD-mode has similar inefficiencies as GPU, while MIMD-mode also does not solve the problem because of the lower compute density of in-order processors.

Summary: Even though vonNeumann execution provides maximum flexibility, it is limited due to low compute density and the reliance on expensive mechanisms. Moreover, all the operand communication is done via centralized register files rather than an efficient

distributed communication approach. Thus, direct data forwarding is inefficient.

Spatial Architectures Here we discuss coarse-grained (e.g., Coarse-Grained Reconfigurable Architecture (CGRAs)) and fine-grained spatial architectures (e.g., Field Gate Programmable Array (FPGA)). The coarse-grained architectures use programmable ALUs instead of fine-grained programmable elements like LUTs in FPGAs. Among coarse-grained spatial architectures, we categorize them by traditional dataflow that can execute arbitrary codes and recent specialized CGRAs. Specialized CGRAs only support small dataflow graphs using dedicated processing elements (PEs) instead of temporally shared PEs in the traditional dataflow. Dedicated PEs allow static scheduling of work, resulting in improved efficiency.

FPGA: is an array of programmable logic blocks that can be configured into many application dataflows. The flexibility of FPGAs enables custom datapaths, thereby achieving near-ASIC cycle count (given sufficient resources). However, the execution time is usually higher due to the lower frequency because of the bit-level programming granularity of LUTs and the high overhead of programmable interconnects.

FPGA's mechanisms are helpful for fine-grained irregularities: e.g., arbitrary datatypes within an instruction. The programmable logic blocks allow fine-grained datatypes while the flexible network can model arbitrary-width datapaths. However, FPGA mechanisms for instruction and task irregularity are generally expensive. For example, overlaying a crossbar for routing indirect accesses over the FPGA network will consume many resources and incur long latencies [169, 100].

Traditional Coarse-Grained Spatial Accelerators: These architectures use coarse-grained programmable ALUs and hence achieve higher frequency than FPGAs. The traditional coarse-grained or dataflow architecture retains the flexibility to execute arbitrary code by using complex mechanisms for ensuring sequential program semantics, e.g., ordering memory access. Examples of traditional spatial-dataflow architectures include TRIPS [38] that achieve "when-ready" semantics using large reservation stations at each PEs. In WaveScalar [225],

the results across PEs are communicated via network. In the presence of irregularity, the latency of data communication can be on the critical path [37].

Triggered instructions [177] and Intel's CSA [256] use a data-triggered execution at each PE, with the capability of reordering instructions. Data-triggered execution enables them to be tolerant to variable latencies, which is useful in the presence of data dependencies. However, out-of-order execution at each processing element (PE) requires tag matching hardware (>3× higher area [197]).

Specialized Coarse-Grained Spatial Accelerators: These spatial architectures optimize over traditional coarse-grained spatial accelerators to improve area/power efficiency at the cost of generality. Specifically, they have a systolic execution array with a set of processing tiles forming a deep pipeline. Each tile executes a single logical instruction and only communicates with its neighbors. PEs are *not* time-multiplexed; hence, only small dataflow graphs can be mapped to this systolic execution array. Examples include LSSD [164], Plasticine [188], Softbrain [162], CGRA-ME [60, 54], and SNAFU [84].

Plasticine [188, 186] is a tiled spatial architecture composed of SIMD compute and scratchpad tiles. It uses a parallel pattern programming interface [187, 123]. SARA [277] proposes a hierarchical dataflow graph mapping to scale to large Plasticine fabrics [277]. LSSD and Softbrain has a spatial compute array (systolic-CGRA) with simple control cores [164, 162]. Compared to parallel patterns, Softbrain is lower-level ISA; however, the control core's ISA can more flexibly implement various computation/memory access patterns. In contrast, Plasticine's spatial distribution of both memory and compute enables higher flexibility in communication across cores. These architectures specialize for regular workloads and, as of today, do not work well for instruction and task irregularity. This dissertation focuses on systolic-style CGRA architectures with a RISCV ISA control core (similar to Softbrain), so that we extend the control core's ISA with our data-dependence dimensions and add corresponding support in the systolic-CGRA architecture.

Hybrid architectures propose a mix of static and dynamic scheduling. CHARM [58] composes multiple pre-defined coarse-grained building blocks at compile time according to the

algorithm requirements, but the data movement and resource allocation are performed at runtime. CAMEL [57] proposes to use reconfigurable FPGA fabrics for uncommon building blocks. In STITCH [228], data is statically scheduled using a multi-hop bufferless network. SEED [163] is a CPU-dataflow architecture that dynamically switches between speculative and dataflow modes of execution according to code requirements. Dynamic scheduling helps in better resource utilization for irregular workloads. Fifer [158] dynamically time-multiplexes multiple pipeline stages onto the same CGRA. We gain inspiration from these works and design a hierarchical architecture with static scheduling within cores and dynamic scheduling across cores (Chapter 8. However, we support only specific types of dynamic scheduling primitives to avoid expensive hardware mechanisms.

Summary: FPGAs and traditional coarse-grained spatial architectures use expensive mechanisms to tolerate variable latencies, common in the presence of irregularity. While specialized coarse-grained architectures remove expensive mechanisms and achieve near-ASIC efficiency, they do not support irregularity and are far less general. We want to broaden the scope of specialized accelerators without losing their efficiency.

1.1.3 Domain-specific Architectures

These architectures specialize for workloads in a single domain – usually for the common data structure and computation patterns. Efficiency is obtained by embedding memory and computation dependencies in the accelerator's datapath. These architectures typically require flexibility for different input dimensions and computations. We will gain inspiration from these architectures for acceleration techniques and will seek to design domain-agnostic generalizations/alternatives.

Flexibility across Inputs In domains like machine learning, the input dimensions may not match the architecture's vector width, resulting in under-utilization, or certain parameters may change the memory-to-compute ratios, thereby shifting the bottleneck to either side. An example of input flexibility is Maeri [129] which supports non-powers-of-two re-

duction using a flexible reduction tree. Hypar concurrently executes multiple DNN layers to achieve a balanced execution [220]. Other examples include SparseAdapt for sparse tensor algebra [175] and GCNAX for Graph Convolution Networks [138]. Many of these techniques are applicable beyond their target domain – for example, database queries have kernels with different bottlenecks, and running them together like HyPar, can result in balanced execution.

Flexibility across Computations: Besides input variability, different algorithms may impose additional flexibility requirements on the accelerator hardware. For example, Extensor [103] optimizes for various sparse tensor algebra operations using hierarchical elimination of computations in general intersection dataflow graphs. Master of None [145] proposes a programmable and systolic-style homogeneous reconfigurable array that can execute various operations in database queries (e.g., join, sort, etc.). Gorgon identified parallel patterns in databases (e.g., partition, merge, sort, filter, etc.) and integrated them into Plasticine [239]. We are looking for domain-agnostic patterns – for example, "join" in Master of None and "merge" in Gorgon require similar control operations. And we will show in Chapter 5 that we can support both using the same stream-join irregularity form.

Summary: Domain-agnostic architectures are flexible along input and computation dimensions. We want to incorporate the ideas in a flexible execution model so that the hardware features are not tied to a particular domain/data structure.

1.1.4 Application-specific Architectures

An ASIC (application-specific integrated chip) is a specialized integrated chip designed to only run the targeted application at high performance and area efficiency. ASICs are on the extreme end of the specialization spectrum and tend to be more specific to one data structure or algorithm.

Table 1.2 shows accelerators for different algorithms, with each giving multiple orders-of-magnitude performance gains over CPUs/GPUs. These accelerators inspire opportunities for

Algorithm	Application-specific Accelerator	Speedup over CPU	Speedup over GPU
Compression	UDP [76]	20×	N/A
Sparse inner product	Sparten [85]	24.9×	4.7×
Sparse outer product	EIE [98], SCNN [179]	7.9×	14×
Async graph	Intel-asynch [173], GraphPulse [193]	74×	N/A
Sync graph	Graphicionado [94], GraphDyns [262]	6.5×	4.4×

Table 1.2: Algorithms and Specialized Accelerators Note we put the maximum speedup numbers, so they should be considered as approximate speedup range.

software-hardware codesign where specific algorithms can be tweaked for hardware friendliness.

Summary: In the specialized spectrum, application-specific accelerators are designed for a fixed algorithm and data structure. Domain-specific accelerators only assume the data structure but provide flexibility for different algorithms and input dimensions. Still, they both suffer from the problem/challenges of over-specialization and often do so unnecessarily. For example, Q100 [259] has an inner-joiner similar to the dot product unit in Extensor [103]. However, still, Q100 still cannot efficiently perform nested tensor computations, as it does not reuse its output as a new input (necessary to chain tensor computations), and it will require a long latency trip to memory.

1.1.5 Our Focus

To balance flexibility and efficiency, we focus on specialized spatial architectures and enhance them for irregularity. We will use inspiration from domain-specific and application-specific accelerators to design the features for our accelerators that are domain-agnostic and achieve ASIC-like efficiency.

Also, once we identify how application-specific accelerators gain their efficiency, we can

map them into this programmable accelerator. Occassionally, challenging workloads emerge that break the mold and require additional features for irregularity instead of a brand new accelerator, helping to unify today's fragmented irregular accelerator space.

We see the primary practical benefit our programmable accelerators as a potential replacement for highly restrictive accelerators that are popular in industry today. Looking at industry designs, they use different architectural styles depending on the application. For example, TPUs [113] are mainly designed for Google's datacenters to run their TensorFlow-based applications. Therefore, they are domain-specific accelerators and programmed using TensorFlow operations. Other examples include Amazon Inferentia [19], Baidu Kunlun [171], Cerebras [131], Graphcore [116], etc. For the cloud, Microsoft uses reconfigurable FPGAs to accommodate more applications. Therefore, their design looks closer to an application-specific accelerator: it does not have an ISA, uses a systolic-style fabric for convolution, and a SIMD unit for scalar operations [79]. A domain-agnostic programmable accelerator will be reusable in both datacenters and the cloud.

1.2 Key Insight and Approach

In this section, we explain our key insight to systematically target irregularity. Then, we discuss our approach to taking this insight to the final architecture design (selecting target workloads and integrating proposed features in flexible hardware).

1.2.1 Insight: Irregularity Can be Specialized

Our key observation is that it is unnecessary to handle arbitrary irregularity because data dependence manifests in common forms across domains. This work suggests that specific *specializable* forms of irregularity are easier to specialize for and sufficient for critical data-processing workloads.

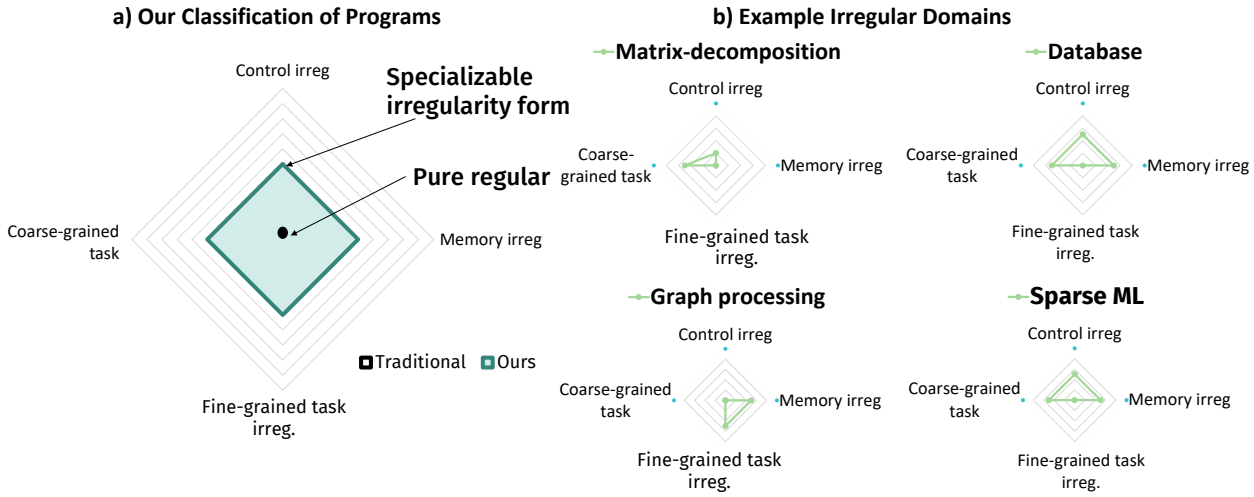


Figure 1.5: Insight: View Irregularity as a Set of Specializable Irregularity Primitives

Requirements Specializable irregularity form has three desirable properties: common, specializable, and composable. We also give intuition about forms that do not qualify for specializable irregularity forms.

- **Common:** implies that data-dependent behavior occurs in many big data processing algorithms and does not assume any characteristics specific to an algorithm.
- **Specializable:** suggests that architecture designers should be able to exploit the unique properties of the data-dependence behavior to achieve efficiency using hardware-software codesign.
- **Composable:** refers to the property of a hardware feature being integrable into a flexible execution model like SIMD, stream-dataflow [162], etc. Some forms may be a more natural fit for one execution model or another.

Some program aspects are more domain-specific, making it less easy to see how they can be composable and common. The more vague and broad a property is, the less specializable it is. Therefore, in some way, these are competing demands, and the choice of specializable form has to balance these demands. Consider two examples: inner-join and approximate algorithms. An inner-joiner module [95] may be too domain-specific/uncommon

Workload Type	Domains Covered	Spec. Irregularity
Static Parallel (control, memory)	ML, synch. GP, DB	Stream-join, AF-indirect (Chapter 5)
Fine-grained Task (scalar data)	Graph processing	Priority order, Working set (Chapter 7)
Coarse-grained Task (vector data)	kNN, Cholesky, GCN	Streaming, Multicast (Chapter 8)

Table 1.3: Irregularity Categories, Supported Domains, and Irregularity Forms

to be assigned a special instruction; however, a single lookup table to enforce *various* control operations in every PE is both common and composable. Another example is achieving high memory parallelism by dropping conflicting indirect memory requests – this occurs in point cloud [77] and compression. However, algorithm-specific tricks are necessary to ensure minimal overhead of approximation; therefore, they may not always be easily specializable.

Vision With the concept of a specializable irregularity form defined, we can explain how a comprehensive set of these forms can help to significantly broaden the scope of programmable accelerators in a systematic way. Figure 1.5a) shows the space of programs categorized by our specializable irregularity forms: control, memory, fine-grained, and coarse-grained task irregularity. The regular workloads lie at the center and cover a few programs like dense linear algebra. Using specializable irregularity forms, we can cover a larger subset of algorithms (see the green diamond in Figure 1.5a)). Figure 1.5b) shows examples of domains that lie within the specializable irregularity range: matrix decomposition, databases, graphs, and sparse machine learning. We skip general irregular workloads like billiards simulation [101], discrete event simulation [11], graph k-core [185], and others because either 1. they require general purpose mechanisms, or 2. they do not require general purpose mechanisms, but we have not figured out a specializable irregular form to express them yet.

We give examples of workloads with varying degrees of irregularities and discuss their tradeoffs in detail in Chapter 2. Later chapters will cover our specializable irregularity forms and our composable hardware features to accelerate those forms.

1.2.2 Approach

Here we discuss how we select target workloads to analyze for specializable irregularity forms and the underlying execution model we enhance for irregularity.

Selecting Target Workloads In the rest of the dissertation, we study workloads in increasing order of their complexity – this is beneficial as more challenging workloads tend to demonstrate a superset of previously identified data dependence forms. From Table 7.1, we first look at static-parallel applications: these can have control/memory irregularity, but at task granularity, they are regular and have no dependencies. Second, dynamic task-parallel applications can exhibit irregularity at both instruction and task granularity. Here tasks are created at runtime and are dependent on other tasks by scalar or vector data. Table 7.1 shows the domains we studied for each category and the specializable irregularity forms that we found.

Decoupled-Spatial Execution Model For our solutions to be composable, we could build on the vector/scalar vonNeumann execution model, but we instead choose to use spatial architectures, specifically the decoupled spatial model. Decoupled spatial architectures have two key properties. They are decoupled in that the memory address generation and computation is performed separately. By design, spatial architectures expose lower-level hardware execution features, making it more natural to modify their ISAs to convey specializable properties. Here we list four characteristics of spatial architectures that make them attractive:

1. **No Total-order on Instructions:** VonNeumann processors enforce a total order on all instructions within a thread. This can cause "artificial" dependencies between control and memory operations if the control operation determines when the load would have happened and not whether it will happen. Another example is ordering requirement among concurrent loads and stores even when they access mutually exclusive

memory locations and could be safely performed out-of-order. In a decoupled spatial model, independence is implicitly assumed, and the programmer may altogether avoid such dependence. Also, the hardware only enforces dependencies specified by the compiler/programmer.

2. **Explicit Memory Dependences:** A spatial execution model makes all dependencies explicit, meaning the programmer has control over optimizing the hardware for the cases when a lack of dependence is known. Or for instances where dependencies follow a known pattern (e.g., for producer-consumer dependence, the data can be directly forwarded).

3. **No Fixed Vector Width:** A spatial model parallelizes across data items in space and in time through dependent computations, rather than across a fixed number of data items alone as with a vector model. In the presence of irregularity (e.g., control flow), supporting heterogeneous data types with a fixed vector width is inefficient, as the control is only applied at instruction granularity, and only a single operand in subword-SIMD can be used. With a spatial model, programmers can schedule multiple different granularity instructions to occur in a pipeline parallel fashion.

4. **Spatial Locality across Tasks:** Traditional SIMT execution is optimized for data parallelism, which assumes that cores (streaming multiprocessor (SM) in GPU terminology) work on independent data. Thus, they only communicate through a monolithic shared cache. However, data is often shared across tasks (running on different cores), exacerbating the overheads of task irregularity. Here, spatial communication across cores may enable optimized network traffic distribution and other optimizations like near-data scheduling.

Overall, decoupled spatial architectures are an attractive candidate for flexible acceleration of irregular workloads.

1.3 Contributions

This section outlines the key contributions of this dissertation. We have designed reconfigurable accelerators, which can run regular and irregular workloads with ASIC-level performance and competitive area and energy efficiency. We identified common program idioms in essential and emerging applications, created novel execution models, and enhanced programmable accelerators. We also developed heuristic-based techniques to decide the optimal algorithmic implementations of this hardware. Below we provide details of each of these contributions:

Made a Case for Applying Reconfigurable Accelerators to Irregular Workloads We dispel the conventional wisdom that irregular workloads are overly complex for specialization and demonstrate that common program idioms *do* exist in irregular domains like sparse machine learning, graph processing, databases, and signal processing. We elucidated how many applications can be composed of a subset of fundamental specializable irregularity forms. We identify these forms for irregular memory access, irregular control flow [66], fine [63] and coarse-grained irregular parallelism [65]. We used irregularity forms to design a family of spatial architectures with features corresponding to each form. These features are modular as the irregularity forms correspond to different execution dimensions and one-to-one correspondence with hardware modules. Overall, our general-purpose accelerator can run workloads from many domains efficiently while retaining modularity that allows optionally removing costly hardware components that may not be required.

Unified Task-Dataflow Execution Model We propose a novel execution model that unifies reconfigurable dataflow accelerators (RDAs) and specialization for task parallelism, which has implications for the applicability of future programmable accelerators. Up to this point, RDAs (e.g., [162, 66, 188]) were only suitable for workloads with static parallelism; dynamic task parallelism on such architectures would have required centralized coordination and pipeline fill/drain overheads that overwhelm short tasks.

We propose the TaskStream execution model that makes tasks a first-class primitive in a dataflow model: task nodes introduce a breaking point in the pipelined dataflow to reorder tasks in time (for work efficiency) or in space (to enable near-data processing) [63]. TaskStream provides a general framework to embrace irregularity in task granularity while retaining efficiency at the instruction granularity. On top of this framework, we propose optimizations for dynamic data streaming to recover inter-task communication structure and dynamic task batching of reused inputs [65] to recover the temporal and spatial locality structure. An approach similar to TaskStream will likely need to be adopted by future reconfigurable accelerators to support broader workloads with dynamic parallelism.

This work also makes a significant contribution for graph-processing accelerators; our proposed task-dataflow accelerator, Polygraph [63], out-performed prior state-of-the-art architectures by several integer factors.

Insights into Optimal Algorithms for Input and Workload Types Data processing workloads are becoming more complex due to algorithm advancement and the integration of machine learning (e.g., GCN [96]). We observe that different algorithms work best for various combinations of inputs and workloads. These algorithms arise due to different computation orders, synchronization granularity, and the choice of data structures.

We use our flexible execution model to implement various algorithms and develop insights into the relationship between different inputs, algorithms, and architecture techniques. Besides comparing algorithms in a single framework, we can model and compare many prior algorithm-specific accelerators without getting misled by specific implementation details.

1.4 Organization

This dissertation builds reconfigurable hardware optimized for different kinds of data dependencies. In the next chapters, we will first give detailed examples of dimensions of irregularity (Chapter 2) and then describe the underlying programmable hardware that we generalize for

Chap.	Topic	Author's related prior work
2	Systematizing and Characterizing Irregularity Forms	MICRO '19 [66, 67], ISCA '21 [63, 64], ASPLOS '22 [65]
3	Decoupled Spatial ISA and Hardware	ISCA '17* [162]
4	Challenges to Design Domain-agnostic Reconfig. Accel.	MICRO '19 [66, 67], ISCA '21 [63, 64], ASPLOS '22 [65]
5	Accelerating Workloads with Data-Dep. Control/Memory	MICRO '19 [66]
6	TaskStream: General Task Framework for Accelerators	ISCA '21 [63], ASPLOS '22 [65]
7	Understanding Fine-Grain Task-Parallel Workloads	ISCA '21 [63]
8	Accel. Task-Parallel Workloads with Coarse-Grained Dep.	ASPLOS '22 [65]
9	Discussion	MICRO '19 [66, 67], ISCA '21 [63, 64] ASPLOS '22 [65]

Table 1.4: Dissertation Outline (* represents background papers.)

irregularity (Chapter 3). Since CGRA-based programmable acceleration is not yet mature, we will discuss the challenges that we faced during this research (Chapter 4). Then, we describe our accelerator for workloads with control and memory irregularity (Chapter 5). For task parallel workloads, we first motivate our general TaskStream framework (Chapter 6) and then describe our optimizations for fine (Chapter 7) and coarse-grained (Chapter 8) dependencies in task parallel workloads. We conclude Chapters 5, 7, 8 with alternate design decisions and associated tradeoffs. Chapter 9 concludes the dissertation with discussion of key insights from this research and pointers to future work. Table 1.4 summarizes the chapters and associated publications.

CHAPTER 2

Systematizing and Characterizing Irregularity Forms

Irregularity is when the decision of which operation to perform or what value to operate on is influenced by data read by the program – more simply, it is data-dependent control flow or memory access. The data dependence can impact performance in different ways – Figures 2.1, 2.2, 2.3 show pseudo-code examples, where inefficiency increases to the right. We only target *specializable* forms and will use this chapter to define the scope of our programmable accelerator.

Specifically, this chapter discusses irregularity along three dimensions: within-instruction dependencies and fine-grained and coarse-grained dependencies among tasks. For each category, we give examples and explain the inefficiencies of existing CPUs (see the rightmost column in figures). We conclude with the approach that we follow dealing with irregularity in later chapters.

2.1 Irregularities due to Control and Memory Dependencies

Data-dependencies may occur within instructions in cases when whether to execute an instruction depends on the outcome of a branch (irregular control) or the operand/output of a memory instruction depends on the data (irregular memory).

Irregular Control If the runtime outcome of a branch determines what computation (Figure 2.1.1a), memory access (Figure 2.1.1b), or task (Figure 2.1.1a) should be performed next, it is defined as control irregularity.

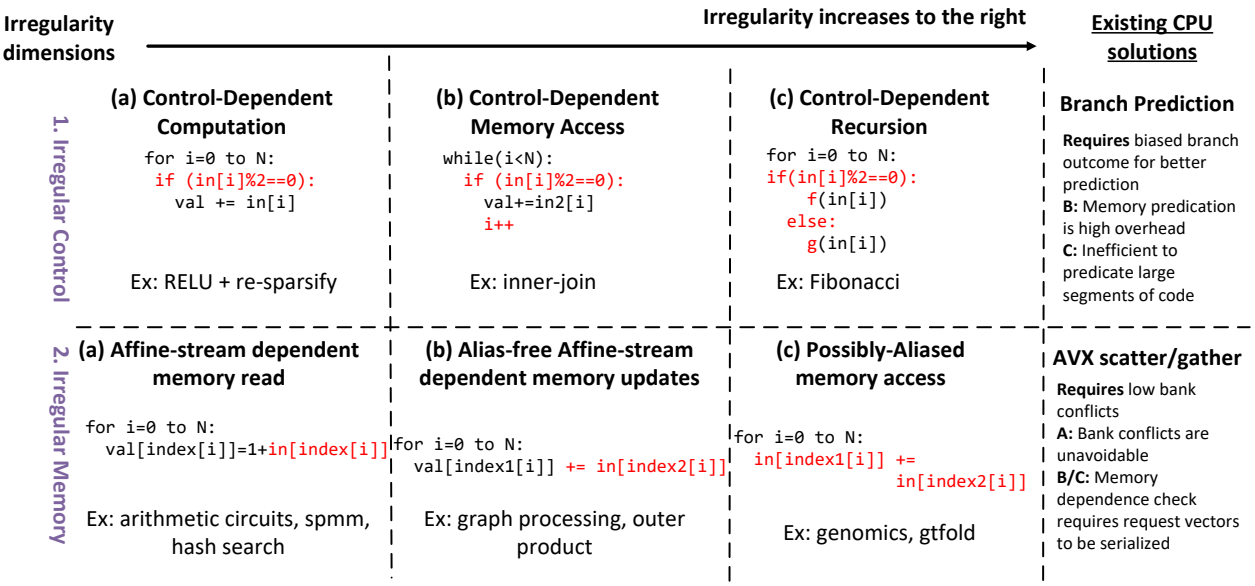


Figure 2.1: Irregularities due to Control and Memory Dependencies

CPUs rely on branch prediction to hide branch resolution latency; however, a data-dependent branch is hard to predict. In such cases, the execution of the branch and the dependent instruction is serialized, causing stalls and pipeline flushes. These stalls can significantly hurt performance, especially when the branch outcome impacts long latency instructions like memory accesses.

CPUs also use "predication"; both branch directions speculatively start in parallel, and results from either direction are discarded after branch resolution. Predication also does not help in case of dependence on memory accesses, as sending more memory requests will put unnecessary pressure on the memory hierarchy. The problem gets worse when the branch outcome decides the execution of a task (see Figure 2.1.1c), where predication would put unnecessary pressure on both memory and computation units.

While GPU's SIMT makes vectorization easier, the branch divergence across lanes makes it less efficient. Also, memory pattern divergence can reduce the effectiveness of coalescing and multi-banking, causing cache/scratchpad bank conflicts.

Irregular memory When the memory access location (Figure 2.1.2a,b) or dependencies (Figure 2.1.2c) are determined from input data, it is defined as memory irregularity.

Memory bandwidth utilization depends on the spatial locality in the memory access pattern, which may be lost due to data dependence. Also, predicting access patterns is complex, making prefetchers useless. Moreover, architectures must consider the possibility of memory dependencies among subsequent requests, requiring expensive load-store queues.

CPUs have banked caches and support scatter/gather for indirect memory. However, the throughput from scatter/gather is quite limited given the limited ports to read/write the vector-length number of cache lines simultaneously. Intel AVX512 recently added support for conflict detection instructions. These instructions *do not* improve the cache-port throughput problem, only the instruction overhead – as still any conflicts within the vector are *handled serially* with no reordering across vectors [108].

GPUs can use scratchpads to avoid the cache port problem, but they still do not reorder requests across subsequent vector warp accesses to avoid costly dependence check [255].

Ultimately, we would like to make a point that if we could guarantee alias-freedom (i.e., consecutive elements of the array are independent), we could simplify the hardware. The dependence check could be completely avoided; enabling aggressive reordering across vector requests is important. Figure 2.1.2b shows an example of alias-free indirect updates. Figure 2.1.2c is much harder because it is not alias-free.

2.2 Task Irregularities

This section discusses the inefficiencies that occur when tasks are created at runtime. Therefore, decisions like when and where to schedule this new task become critical for performance. Here we discuss the challenges separately when the data dependencies are caused due to fine-grained/scalar data or coarse-grained/vector data.

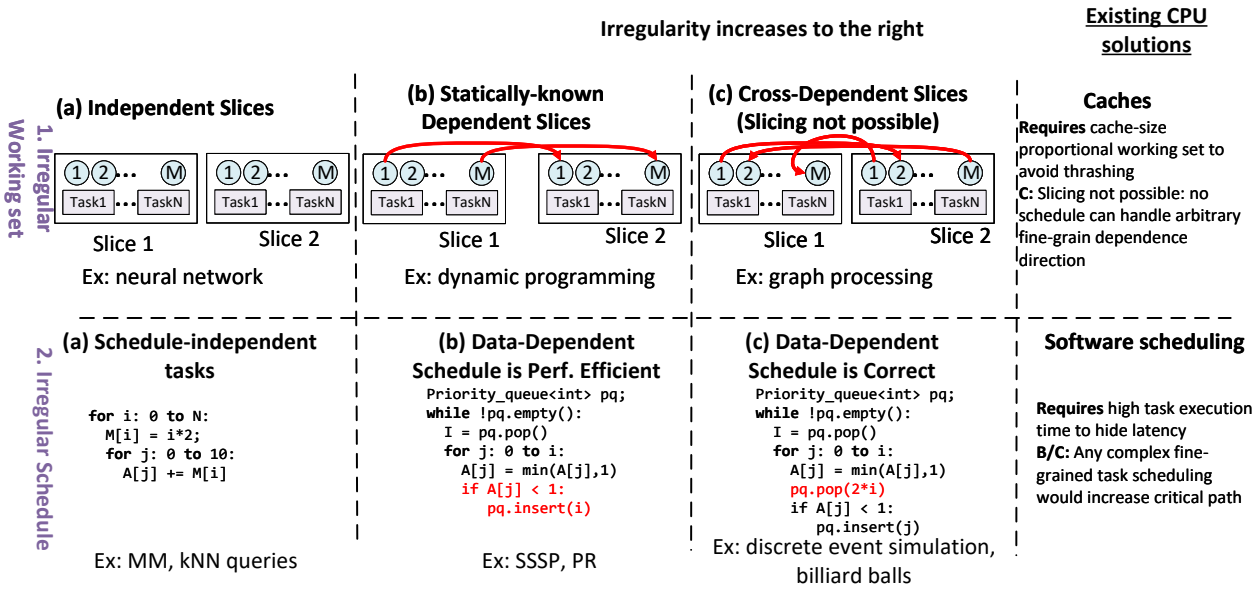


Figure 2.2: Task Irregularities with Fine-Grained Data Dependencies

2.2.1 Task Irregularity with Fine-grained Data Dependencies

The scalar data dependencies are usually satisfied by passing the dependent data as task arguments. The irregularity mainly occurs in how the dynamic task execution order determines the memory access pattern (irregular working set) and many times the algorithm convergence rate (irregular scheduling).

Irregular Working Set Working set refers to the unique cache lines accessed during a program phase. When the working set lacks a closed form representation, or a geometric representation, as they typically do in dense linear algebra, it is defined as working set irregularity. Here will use "slice" terminology – a slice is a cache-sized data partition. In this context, tasks are often designed to operate over a slice, since that optimizes the reuse capability of that program phase. Thus, the working set working set depends on which slice's task is being executed.

The working set may switch due to data dependencies across slices: these dependencies may be in a single direction (e.g., Slice 1->Slice 2) in Figure 2.2.1b or arbitrary directions

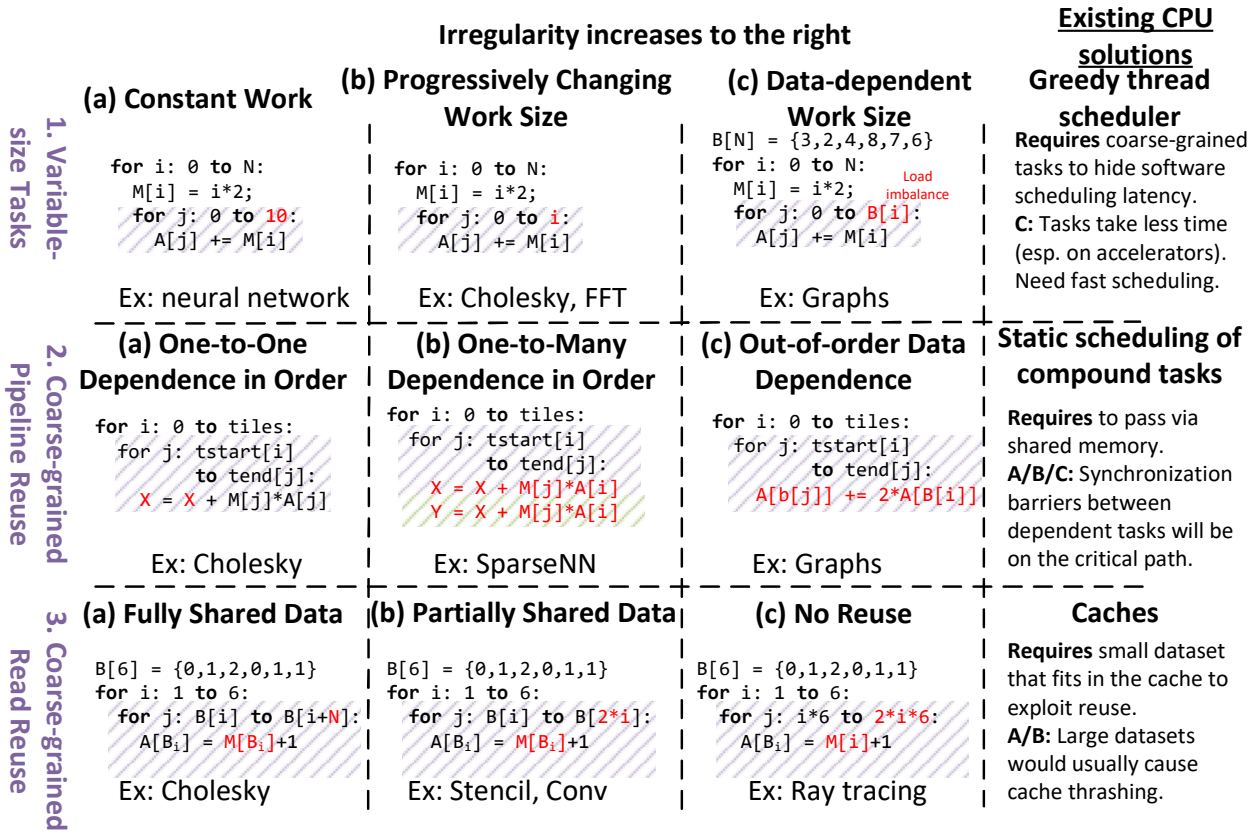


Figure 2.3: Task Irregularities with Coarse-Grained Data Dependencies

(e.g., Figure 2.2.1c). Figure 2.2.1a can be easily made regular; as slices are independent, one may batch tasks for a single slice to execute together. CPUs and GPGPUs only support the independent-slice case using software (commonly known as tiling) to batch tasks. Caches will incur high miss rates for the other two cases due to large working sets.

We will conclude with a point that when the slice dependence direction is fixed, it is possible to schedule the slice's tasks in an order such that the working set is limited for a long time.

Irregular Scheduling Scheduling is deciding when and where a task should be executed. If the preferred (for correctness/performance) schedule depends on the data (for example, sorting tasks based on task argument), it is defined as scheduling irregularity. Scheduling may be required across coarse-grained tasks (Figure 2.2.2a), fine-grained tasks where the priority

schedule is performance-efficient (Figure 2.2.2b) or required for correctness (Figure 2.2.2c). For example, shortest path algorithm prefers distance-based priority for faster convergence, thus better performance. For the example in Figure 2.2.2c, the execution of a task may impact other pending tasks in the priority queue, thus the order of task execution impacts the final result, and hence exact priority ordering is required for correctness.

CPUs usually rely on software data structures, and the latency of accessing these data structures dominates the execution time for fine-grained tasks. GPUs support hardware scheduling using threads, but switching among fine-grained tasks adds start-up overhead to initializing the large register states. Overall, fast hardware scheduling with minimal context switch overheads is desired.

2.2.2 Task Irregularity with Coarse-grained Data Dependencies

In the presence of coarse-grained dependencies, the parent writes data to shared memory, followed by a synchronization barrier, and then the child reads the dependent data. Barrier inserts long stalls, causing a significant bottleneck. In this section, we discuss irregularities caused due to variable task size and coarse-grained inter-task communication (Figure 2.3).

Variable-sized Tasks A variety of task-parallel workloads have task types[1] whose amount of work is either constant (Figure 2.3.1a), progressively changing over its instances (Figure 2.3.1b), or data-dependent (Figure 2.3.1c). The last two cases exhibit variable size irregularity.

Figure 2.3.1c shows an example where inner loop tasks have a data-dependent length based on B[i]. A naïve CPU/GPU thread scheduler would assign the inner loop tasks irrespective of the work involved in a task. Work-stealing is possible but requires extra inter-core communication latency and bandwidth.

The opportunity here is to distribute tasks with the knowledge of the work involved. In

[1]A "task type" is the static definition of a task, including computation and memory accesses, while the dynamic instantiation of a task is a task instance.

the example, core 1 gets the smallest and second-largest task (i.e., with total work = 3+7 = 10), , so all cores get similar total work. This model is synergistic with accelerators, which have quite predictable execution times. When task size distribution is arbitrary, knowledge of work involved may not be enough and often optimizations like dynamically splitting tasks is required.

Inter-task communication When concurrent tasks work on shared data, it is defined as irregularity due to inter-task communication. The communication may be of three forms: shared read-only data (read reuse), producer-consumer communication (pipeline reuse), and shared update-only data. We will discuss each of them below.

Producer-consumer communication: In data processing algorithms, the dependencies between two tasks can be ordered (for example, where one task produces an array that the other uses in the same order) or unordered (Figure 2.3.2c). The ordered dependence may exist between two tasks (Figure 2.3.2a) or many tasks (purple and green tasks in Figure 2.3.2b).

CPUs and GPUs use a shared memory model, which forces strict ordering between dependence tasks, e.g., using barriers in CPUs. For example, see Figure 2.3.2b that demonstrates a global reduction example where each core gets a tile of data. In the naïve task parallel implementation, all cores need to perform updates on the reduction variable through memory.

The opportunity for examples in Figure 2.3.2a,b is to identify the ordered reuse and pipeline or *stream* the data from a producer to one or more consumer tasks. Pipelining transforms the memory traffic into direct network traffic, reduces shared-memory overhead from coherence, and allows overlapping tasks execution for more concurrency. In the example, the pipelined reduction can be performed without accessing memory (except for writing the final value).

Coarse-grain Read Reuse: Another kind of inter-task communication occurs when different subsets of tasks read the same data. Tasks may be reading only a part of the shared data (Figure 2.3.3a) or the whole data structure (Figure 2.3.3b,c).

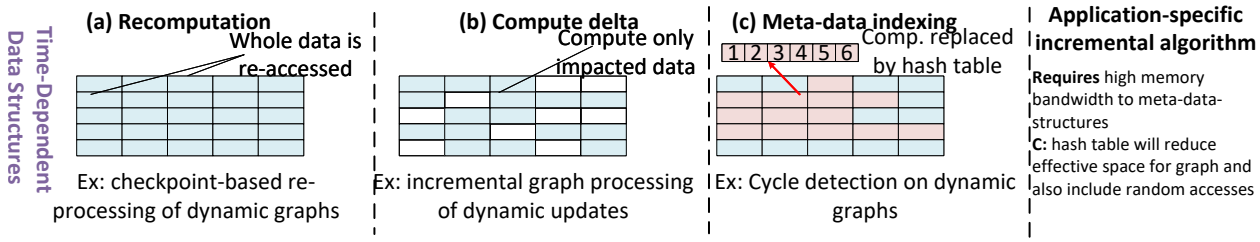


Figure 2.4: Irregularities due to Time-Dependent Data Structures

If such tasks are not scheduled together in time or space, the opportunity to exploit read reuse can be lost. Figure 2.33a demonstrates this with an algorithm that traverses and modifies a compressed sparse row (CSR)-like data structure, and is representative of common algorithms that rely on range-based indirection. Here the duplicates in B are expected to create multiple tasks with shared read data, providing an opportunity for reuse. CPUs and GPUs use cache only and cannot control the task scheduling order; thus, the data may be evicted before it reused. Exposing no-reuse property in Figure 2.33c an avoid cache pollution by bypassing caches for no-reuse data [247].

2.3 Other Irregularities

Earlier, we discussed the fine-grained and task irregularities we explored in this dissertation. In this section, we will discuss examples of alternate kinds of irregularities that could be plausibly useful in many domains and thus, lead to new opportunities for specialization.

Data-dependent Data Types Often the data-type size is chosen to meet the precision requirements, which may be changing during different phases of the algorithm [139]. The naive solution is to upcast new data types to hardware support, negating many potential benefits of variable data types.

Irregular Iteration Space The iteration space refers to the iteration range of a loop. Commonly, the loops are parallelized by splitting into iterations proportional to the hard-

ware vectorization width. Parallel loop execution is correct, given that there are no cross-iteration dependencies. In CPUs, SIMD instructions can efficiently exploit the loop-level parallelism [144].

Inefficiencies may be caused due to different reasons. For example, the loop bound is not a multiple of the vectorization width, causing the SIMD array to remain underutilized in the last iteration. A harder case is when the loop bound depends on the value of the outer loop index. Therefore, most of the parallel iterations of the inner loop would not be multiple, exacerbating the under-utilization (Figure 2.3.1.c) shows an example).

Time-Dependent Data Structures When data structures change with time, incremental algorithms are used only to update the impacted part of the computation. The extent of irregularity depends on the type of incremental algorithm; figure 2.4 shows three examples.

We will explain these using an example of google maps, where the shortest path to the destination has to be updated for every new traffic update. Figure 2.4a shows the most straightforward approach where the algorithm is re-executed. Thus, the whole road graph will be involved in the computation. This involves redundant computation for parts of the graph that are not affected. In (b), however, only the input updates are applied to the previous output. In our example, the traffic change will be considered only when it is connected to the destination. Accessing arbitrary parts of the input data will introduce randomness/irregularity in both the working set and reuse. Also, incrementally building upon some previous computation means serializing execution across updates. Figure 2.4c shows another common strategy to reduce redundancy. Here the results of common computations are pre-stored (e.g., distances between hot edges for the shortest path algorithm), and this data structure is indexed during incremental computation. For desirable performance, the hardware should be able to index into such a structure with high efficiency.

2.4 Summary and our Solution Approach

From the above analysis, we conclude that existing general-purpose processors are limited because they have to support arbitrary behaviors. Even though they provide limited support for irregularity, they are ineffective due to vonNeumann and the shared memory model. In this dissertation, we designed reconfigurable hardware for specializable irregularity forms that cover many domains. Specifically, we proposed stream-join control and alias-free indirection memory for machine learning and databases (Chapter 5). For graph processing and matrix decomposition algorithms, we proposed a task-based execution model with specialized data-dependent ordering (Chapter 7 and techniques to recover locality across tasks (Chapter 8).

CHAPTER 3

Decoupled Spatial ISA and Hardware

To design a flexible irregular accelerator, we base our proposals on a typical decoupled spatial ISA and hardware architecture. The hardware and ISA are designed for efficient acceleration of regular workloads. For ISA, we build upon the open-source stream dataflow ISA [162]. This ISA is sufficiently flexible for regular implementations. The hardware uses a systolic CGRA for spatial computation and has a flexible address generation unit for decoupled memory access.

3.1 Execution Model and ISA

3.1.1 Decoupled Spatial Execution Model

Here we first define preliminary terms and then explain the execution model and its corresponding ISA support.

- **Dataflow Graph (DFG)** The DFG is an acyclic graph where nodes are instructions while edges represent dependencies. Dataflow nodes maintain a single state item. This enables them to be mapped to systolic like fabrics [60, 188, 89, 66, 162] for high efficiency.

- **Streams** Streams are simply an ordered sequence of values, used as architecture primitives in many prior designs [199, 56, 162, 259, 250, 106, 60]. Relevant to this work are memory streams, which are sequences of loads or stores. Streams are similar to vector accesses, but have no fixed length.

(a) Original C code

```
1: def matrix_vector(uint64_t
mat[N][M], vec[M])
2: uint64_t out_vec[M]
3: for n = 0 to N:
4:    sum=0
5:    for m = 0 to M:
6:       sum+=mat[n][m]*vec[m]
7:    out_vec[n]=sum
```

(b) Decoupled-Spatial Stream Code

```
1: def matrix_vector(uint64_t mat[N][M],
vec[M])
2: uint64_t out_vec[M]
3: Config(matrix_vector_dfg,
matrix_vector_size)
4: # Load vector from memory to scratchpad
5: Mem_Scratch(&vec[0], 8, 8, N, 0)
6: # Load matrix from memory
7: Mem_Port(&mat[0][0], 8, 8, N*M, port_A)
8: Barrier_Scr_Wr()
9: Scratch_Port(0, 0, 8*M, N, port_B)
10: Port_Mem(port_C, 8, 8, 8, &out_vec[0])
11: Barrier_All()
```

(c)Decoupled-Spatial Execution

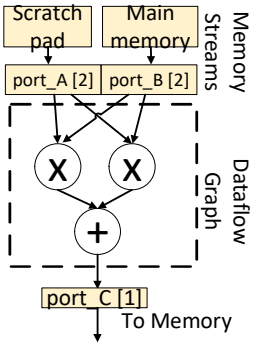


Figure 3.1: Decoupled Spatial Execution Model

Execution Model In a decoupled-spatial execution, the programs execute in phases, each starting with a loading multiple memory streams, the loaded data items are input to the dataflow graph and ending at a final barrier. Each phase consists of many computation instances. Figure 3.1c) depicts the conceptual execution model: the data streams may be read either from off-chip main memory or on-chip memory. Before communication to dataflow graph, they may be optionally buffered. Using the dataflow execution, output will be streamed to an output buffer while is then stored back using another data stream. For the matrix-vector multiplication example, the matrix is read from memory while the reuse vector is read from scratchpad. The computation involves accumulating the multiplication of a row of the matrix and the vector. The output is written back to memory.

Comparison with vonNeumann Model The key difference with vonNeumann model is that there is no total order on instructions in spatial dataflow. The programmer explicitly defines the computation, memory accesses, and the dependencies among them. This enables efficient pipeline parallelism without instruction scheduling overheads. The decoupled execution also removes address generation from the critical path.

Command Name	Parameters	Description
`Config`	Address, Size	Set systolic-CGRA configuration
`Config`	Address, Size	from the given address
`Mem_Scratch`	Source Mem. Addr., Stride, Access Size	Read from memory
	Num Strides, Dest. Scratch Addr	with pattern to scratchpad
`Scratch_Port`	Source Scratch. Addr., Stride, Access Size,	Read from scratchpad
`Scratch_Port`	Num Strides, Input Port #	with pattern to input port
`Mem_Port`	Source Mem. Addr., Stride, Access Size,	Read from memory
`Mem_Port`	Strides, Input Port #	with pattern to input port
`Const_Port`	Constant Value, Num Elements, Input Port #	Read constant value to input port
`Port_Port`	Output Port #, Num Elements, Input Port #	Issue recurrence between input/output ports
`Port_Mem`	Output Port #, Stride, Access Size, Strides,	Write from port
`Port_Mem`	Destination Mem. Addr.	with pattern to memory
`Barrier_Scratch_Rd`	-	Barrier for Scratch Reads
`Barrier_Scratch_Wr`	-	Barrier for Scratch Writes
`Barrier_All`	-	Barriers for all commands

Table 3.1: Decoupled Spatial ISA [162]

3.1.2 ISA Specification

Table 3.1 summarizes the stream commands supported in the Stream-dataflow ISA [162]. These commands can be grouped into three types of specifications: dataflow graph, memory streams and barriers. Here we explain each:

Dataflow Graph Specification The computation is expressed using the dataflow graph, which can be expressed using any graph representation. The dataflow graph has these components: 1. The number of "vector ports" (input/output of the computation) 2. their width (maximum data words transferable per cycle), 3. their depth (the associated buffer size), and 4. their connections to the computation substrate. The `Config` instruction reads the stored dataflow graph from memory and maps it to the computation fabric.

Stream Specification For efficient decoupled memory access, the ISA supports affine address patterns to/from different memory types. The supported pattern is a two-dimensional pattern defined by an access size (size of lowest level access), stride (size between consecutive accesses), and number of strides. More formally, these are accesses of the form a[C*i+j], where induction variables i and j increment from 0 to an upper bound.

Example Program Figure 3.1 shows how a matrix-vector multiply C code can be written using the decoupled-spatial ISA. The computation is translated to the dataflow graph (shown in Figure 3.1c) and the memory streams are defined using the decoupled-spatial stream code (Figure 3.1b). The stream code first configures the dataflow graph (line 3), then defines the streams, and end with barrier (line 11). To exploit reuse on the vector, vector is first written to the scratchpad (line 5) and then after a scratchpad write barrier (line 8), it can be read into the dataflow graph (line 9). Finally, an output stream writes the computed output vector to memory (line 10).

In the next chapters, we will propose extensions to different parts of the ISA. For example, dataflow specification can be modified to include specializable control irregularity information, and specialized memory streams that can encode specializable memory irregularity.

3.2 Decoupled Spatial Hardware Design

Figure 3.2 shows the typical accelerator core that implements decoupled spatial execution model. It can be viewed as consisting of two planes: 1. A Control plane manages the accelerator components and handles less frequent tasks, e.g., setup between phases, and 2. A Data Plane executes the accelerator code. In our template, the control plane is a simple inorder core while the dataplane include a systolic-style CGRA compute fabric (extremely common, e.g., [113, 48, 51, 17, 237, 143]) with a streaming address generator. Many regular algorithms can be supported with this hardware design like dense linear algebra and some

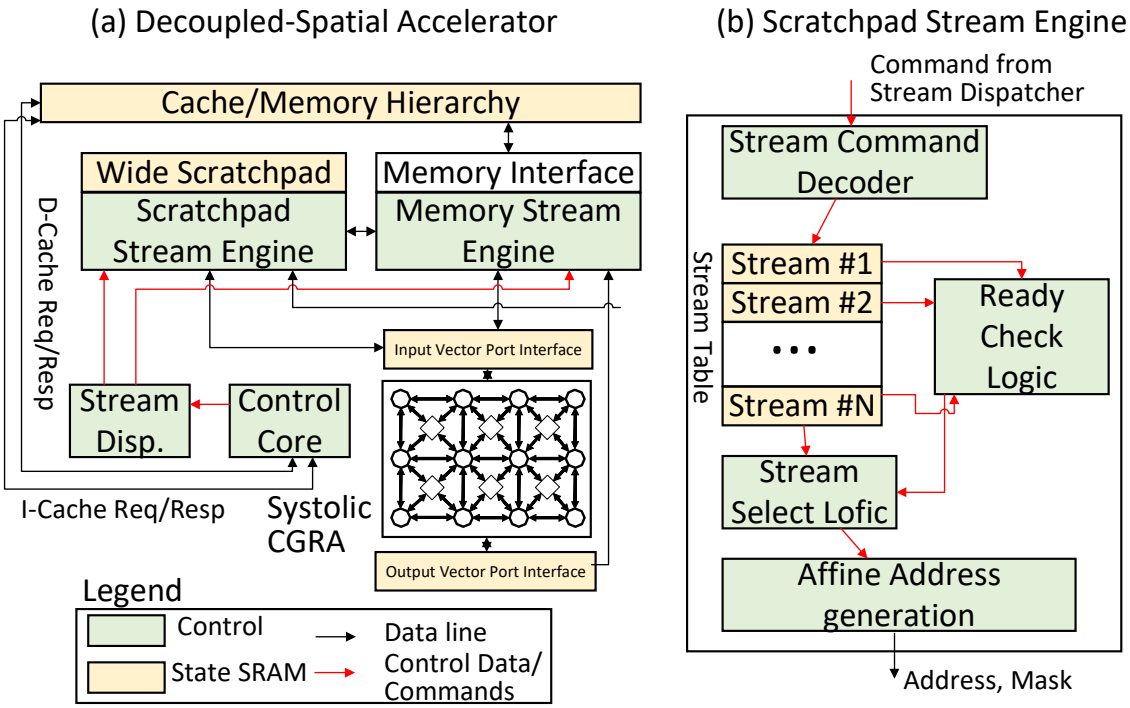


Figure 3.2: Decoupled Spatial Accelerator Core

deep learning workloads [162].

3.2.1 Control Plane

Each accelerator core has a simple inorder core that is responsible for managing the data plane hardware components of the accelerator. The jobs include configuration of dataflow graph and memory streams, and managing synchronization barriers. The control code is also responsible for the execution of any scalar code (code which is inefficient on data plane) in the program.

3.2.2 Data Plane

Data plane consists of a systolic-CGRA to implement dataflow graph and a stream controller to generate memory requests corresponding to different stream patterns. It also contains on-chip scratchpad memory for caching critical updates. Finally, the core may be connected to

a router for the scaled-up version of the accelerator. We detail the design of systolic CGRA and stream controller below:

Systolic CGRA Computation In a systolic-CGRA, the processing elements configured to repeatedly perform one operation, are fully pipelined to execute one operation per cycle, and only communicate with neighbors. In our implementation, these processing elements are connected by a circuit-switched mesh network where each router can transfer upto 64 bits data (the width of systolic CGRA). A component of the offline compiler is responsible to schedule PEs at different locations in the mesh. It should ensure perfectly pipelined communication among PEs. At a very high-level, our approach combines the principle of stochastic scheduling [161].

Stream Controller Address Generation Figure 3.2b shows the microarchitecture of the stream controller. This hardware has two responsibilities: 1. address generation according to the stream type, 2. collecting data from memory/scratchpad and sending it to the control fabric in the correct order. Since there can many memory streams, it has a stream table that maintains information about all active streams (its parameters that determine the access pattern and the number of elements that are left to be served). The address generator is a simple finite state machine that takes in access_size, num_elements, stride to produce the desired access pattern. Since the data from memory can arrive out-of-order, the data for each stream is reordered using a single vectorized buffer.

Finally, the stream controller communicates with the control core to inform when any barrier command can be revoked. For example, when waiting on a barrier for scratchpad write streams, the stream controller would inform the core that all pending scratch write streams are done.

System integration In this dissertation, we consider the accelerators to be "standalone", meaning that the accelerator has its own memory space. To integrate with the system, there

are several alternatives like PCIe, bus like NVIDIA's SXM2 [83] or even a cache coherent interconnect is possible as well.

CHAPTER 4

Challenges to Design Domain-Agnostic Reconfigurable Accelerators

Specialized accelerators have been the subject of intense research for many decades, but particularly in recent years; examples pervade domains, including graphs [94, 221, 274, 68, 15], AI/ML [265, 104, 17, 195, 210, 237, 114], databases [122, 258, 259, 115], systems [282, 75, 76, 78], and genomics [233, 81, 234, 50]). This trend is also true in industry [113, 190, 245, 159, 168]. This field is fairly mature; application-specific devices are in mobile SoCs [21, 20], IoT devices [195] and CPU chips [31]. There are even tools like HLS to help generate application-specific hardware.

On the other hand, our focus is on programmable accelerators that are robust to change, shorten chip design cycles, and will be valuable for economies of scale. Unfortunately, domain-agnostic reconfigurable acceleration space is not well understood, especially when designing new programmable accelerators. Varying proposals come from adhoc assumptions: for the degree of flexibility: e.g., FPGAs attempt flexibility across broad big data workloads while recent CGRAs [164, 60, 54, 188, 162, 84] target mostly data-parallel workloads). And the performance metrics may be different (e.g., FPGAs optimize for execution cycles while CGRAs focus on performance with minimum hardware overhead). These assumptions result in drastically different designs and tools. For example, FPGA uses LUT-level reconfigurability, and its HLS-based compilers embed an application's low-level dataflow graph onto FPGA resources. Instead, Softbrain [162], a representative example for CGRAs, uses a vectorized-dataflow model, where the compiler searches for supported patterns in computation (e.g., 32-bit MACC) and memory (e.g., linear access) to map applications to hardware. Overall,

Domains	Benchmarks
ML	MLPerf [196], Fathom [14], CortexSuite [230]
Databases	TPCH [232], TPCDS [183], TPC-C [137]
Bioinformatics	BioBench [16]
Genomics	GenomicsBench [222]
AR/VR	Illixir [107]
Graphs*	GAP [26], SympleGraph [284], RStream [244]
Matrix Decomposition*	5G [43, 253]
Point clouds*	KD-tree-based [29, 261]
AI/DB*	Gorgon [239]

Table 4.1: Benchmarks in Data-processing Domains (* specifies no standard exists.)

these are important design decisions and need more attention.

Our goals are reasonably broad: we would like to support regular and irregular workloads and explore various performance vs. area tradeoffs. This chapter will systematically discuss the challenges of designing domain-agnostic accelerator architectures at various stages. For each challenge, we briefly describe our approach, which we follow in our designs in the subsequent chapters. These challenges include: 1. Picking target workload benchmarks to specialize for, 2. Selecting what *algorithmic* implementations of those workloads to target, 3. Developing hardware and software features to achieve efficiency for the target algorithms, and 4. Model and evaluate the performance of the proposal.

4.1 Pick Workload Benchmarks

Traditional benchmark suites are designed to stress different hardware components with minimum redundancy. For example, CPU SPEC's `mcf` is bounded by front-end while `xz` is bounded by DRAM bandwidth [176]. Domain-specific benchmark suites are also designed on the same principle; for example, MLBench's ResNet model requires higher throughput while the GNMT model needs high memory capacity [196], TPC-H has both sort-heavy and non-sort-heavy queries [232]. Table 4.1 summarizes the available benchmarks for different

Dimensions	Workloads
Dense	Dense conv/FC, Graph MM, stencil
Data-dependent control	Sparse conv/FC inner, Sparse GBDT, KSVM, Database join
	Graph pull, filter, Graph mining, Genomics
Data-dependent memory	GBDT histo, Sparse conv/FC outer, Arithmetic ckt, Graph push
Fine-grained task parallelism	Graphs, Tree traversal
Coarse-grained task parallelism	matrix decomposition, Point clouds, GCN
	Sparse-MM, AR/VR, Knapsack, K-queens

Table 4.2: Workloads Classified by Data Dependence Forms

domains being studied for acceleration.

No such benchmark exists for reconfigurable accelerators. CPU's SPEC is unsuitable because their instruction footprint is too high to configure all the resources statically. Others are limited to a domain. One option is to pick a union of the benchmarks in Table 4.1; however, it will result in a large benchmark suite with extreme redundancy. Besides, it does not make sense to design an accelerator for an arbitrary combination of domains, as the resulting hardware complexity can be exponentially high. Our insight is that we need to identify benchmarks according to the hardware component they stress.

Our approach: The problem with the existing way is that it is not insightful to study random applications. For the benchmarks to be meaningful, our idea is to embody workloads in certain forms of irregularity. We need to profile and deeply understand kernels across domains to do that. Table 4.2 shows the benchmarks we classified by their data-dependence dimensions. Consider database joins and the sparse tensor dot product in deep learning: they both demonstrate data-dependent control. Therefore, we specialize in only one of them and hope it will be effective for the other. In our work, we constructed our benchmark suites out of programs that have the irregularity types in Section 4.2, corresponding to the remaining main chapters in the dissertation (Chapters 5, 7, 8).

Note that filtering workloads by data-dependence dimensions may miss out on some

workload-specific opportunities. For example, deep learning layers often involve a ReLU post-processing step, and a separate pipeline would prevent sharing bandwidth with inputs. Such requirement is not common in other domains, but in deep learning, an independent pipeline can improve performance by 30%. Nevertheless, this number is relatively small compared to 7× speedup over GPUs by just specializing in data-dependent control. Please note that most workload-specific software optimizations are still applicable. For example, Tigris [261], an accelerator for k-nearest neighbors, proposed to pick an optimal tree size to balance the tree traversal and dense search kernels. Our flexible accelerator supports tree traversal and dense search and can adapt to the proposed algorithm. **In conclusion, new benchmark suites need to be written for less redundant accelerators.**

4.2 Pick Algorithms to Accelerate

After picking a benchmark suite, the workload may be implemented differently; we call these "algorithms". For example, sparse matrix-multiply can be implemented as an inner-product or outer-product [174, 179]. Often, these algorithms have different requirements from the hardware. Thus, accelerator designs pick a single algorithmic implementation based on simplicity [94, 173, 174, 179] and specialize for it. Such an approach may not be ideal – recent works have shown flexibility across algorithms [129, 175, 138] may improve performance. This is expected: on the one hand, algorithms create tradeoffs in which data-structure to optimize for reuse [128], whether to do more work but architecture-friendly or vice-versa [94], and which order to execute to maximize sparsity benefits [138]. On the other hand, these tradeoffs' inclination may depend on the size of inputs (e.g., higher input channels mean we should optimize for input reuse) and workload properties (e.g., convergence rate may change with the algorithm). The key takeaway is that we *must* look at all different algorithm options of every workload to study the workloads effectively.

Note that our advice to study all algorithms is almost contradictory to Section 4.1 because we are saying that we need to go back and gather more implementations of each workload,

increasing the complexity of our benchmark suite. However, it is worth the complexity, partly because of performance benefits as explained above, and also because we must consider the possibility of choosing the wrong form of irregularity or the bad implementation of it to specialize. The only way out is to at least in some way model the alternative forms through exploration.

Our approach: We evaluate the cross-product of algorithm, input, and workload. The analysis output is a subset of algorithms to support and insights of picking the optimal algorithm for a given input and workload type. This analysis creates three challenges. First, there can be 10s of algorithms, and manually analyzing each can be overwhelming. Therefore, a systematic approach is required: e.g., we can have orthogonal algorithm dimensions where a combination of choices represents an algorithm. Second, performance estimation during the analysis will require modeling a hypothetical application-specific accelerator. We believe a modular execution model and hardware (where modules correspond to the algorithm dimensions) will minimize unnecessary redesign. Finally, we will need to estimate the performance of the ASICs generated above quickly. Existing analysis approaches use analytical models [178, 128] and, as such, are not sufficient for irregular workloads. The reason is that the next step in irregular workloads depends on the data, and simple formulae are insufficient. We propose abstract simulators to model *only* critical components at the cycle-level (see our GraphSim simulator for graph processing (Appendix A)).

4.3 Designing Hardware-Software Interface

We analyze each selected algorithm's bottlenecks in traditional reconfigurable architecture and determine specialization opportunities. The solution can either be software-only (e.g., tiling), hardware-only (e.g., a dedicated path to preprocess data), or programmable hardware solution (e.g., a dedicated path where the programmer may pick an arbitrary preprocessing function). The third option usually involves updating the ISA. We focus on programmable hardware solution as the hardware support is usually required for high-performance and

multiple domains creates the need for programmability.

Our key insight is that we must look past the domain to more fundamental application properties to find commonality. To this end, we define three *required* properties for any programmable hardware feature: 1. **Common:** applicable across a range of data-processing workloads, 2. **Specializable:** they should fundamentally have parallelism, which can be efficiently exploited using efficient software and hardware mechanisms[1], and **Composable:** meaning the solution is integrated into a flexible execution model, e.g., SIMD or dataflow. For example, an inner-joiner module [95] may be too domain-specific/uncommon to be assigned a special instruction; however, a single lookup table to enforce *various* control operations in every PE is both common and composable. This dissertation looks for these forms in *irregular* workloads.

These new hardware features often need to be exposed in the ISA for programmability. One design choice is granularity; coarse-granularity minimizes hardware complexity, while fine-granularity improves robustness to future algorithms. Examples include CPU's x86, which is extremely fine-grained (e.g., `ld,st, add`, etc.) and TPU, which has coarse-grained instructions from TensorFlow (e.g., conv, depth-wise conv, etc.). We are looking for a middle ground for domain-agnostic acceleration: where the ISA is coarse-grained for efficiency but with smaller granularity than Tensorflow to cover more domains. An example of prior work is Microsoft's Brainwave which uses sparse-matrix-vector in the ISA instead of sparse-matrix-multiply in TPU to allow flexibility to batch size of 1 [79]. Another example is LUTs in FPGAs, which are too fine-grained; in contrast, CGRAs use ALU operations that provide coarse granularity and are still flexible as most domains require a subset of common operations.

Our approach: Our workload benchmark selection has classified the workloads by common data-dependence forms; now, we look for specialization opportunities. There are two options: standard workload analysis [165, 133] and learning from prior accelerator works.

[1]Note that we define specializable for which we could find solutions, it does not mean that others are impossible to specialize for

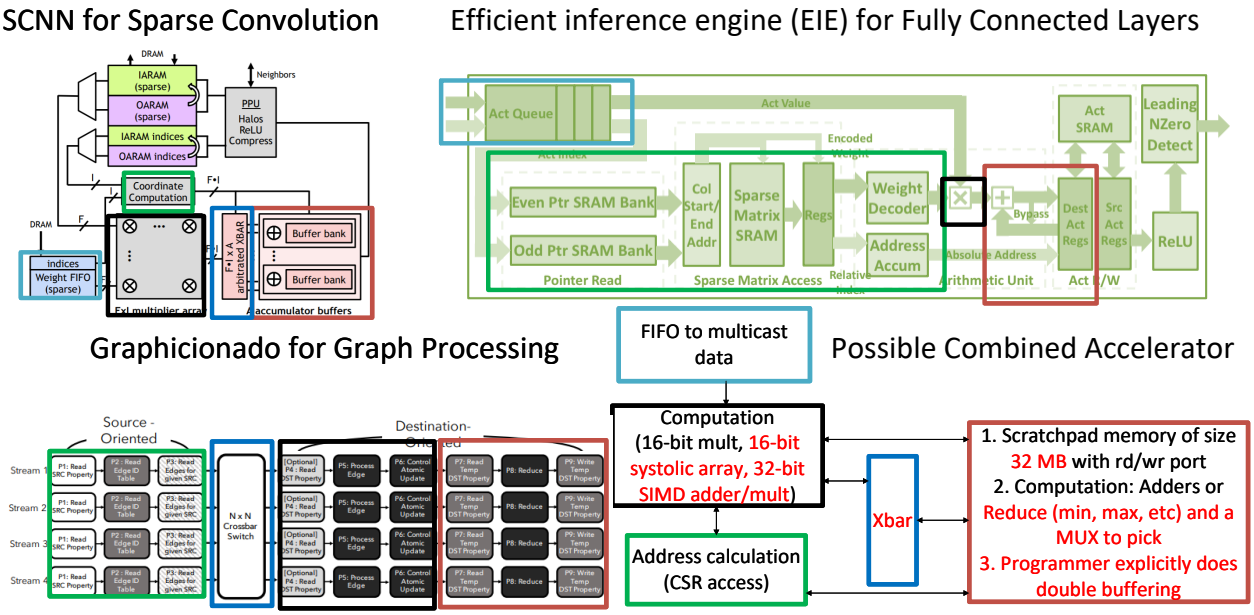


Figure 4.1: Approaches to find commonalities across kernels

Figure 4.1 demonstrates the opportunity to learn from other accelerator works. It shows three domain-specific accelerators: SCNN for sparse convolution [179], EIE for fully connected layers [98], and Graphicionado for graph processing [94]. Figure 4.1 also shows a possible combined accelerator that executes all three domains: convolution, fully connected, and graph processing. The box outline colors correspond to hardware components. Here we present the opportunity by discussing the similarities in accelerators and caveats to our approach.

Similarities in Accelerators Below we discuss the common hardware components.

- **Decoupled Computation:** All architectures have embedded spatial computation modules with application datapath (black box in the figure). A flexible spatial fabric will require switches between PEs for different datapaths, but it can use the common area-consuming routers.

- **Atomic Update Unit:** All architectures support aggressive atomic update: vectors of requests read data from scratchpad in parallel, perform computation (e.g., `adders`

in SCNN, `min` in Graphicionado), and write the updated data back to the scratchpad.

- **Address Calculation:** The green box represents the module that generates addresses to prefetch a known pattern. The architecture is similar because they all generate access patterns for the CSR data structure.

- **Multicast Units:** SCNN and EIE read data at a buffer (shown by the light blue box in Figure 4.1, and multicast to all compute units. Even though Graphicionado does not support multicast due to relying on a crossbar network, other works show that data multicast is beneficial in graph processing as well [27]. Note that we will need programmability for the multicast data source: SCNN is receiving data from memory while EIE is receiving from the output of another core.

In conclusion, application-specific work is not useless for building programmable accelerators – on the contrary, it is essential. The process of building programmable accelerators requires understanding specialization taken to the extreme. Then we are taking a step back and trying to generalize without giving up efficiency.

Caveats for the Programmable Accelerators It is also critical to understand the overheads if we combine these three accelerators. First, certain features cannot be helpful everywhere; for example, SCNN and Graphicionado need a crossbar, while EIE does not need it because of enough parallelism in SpMV to meet the target throughput. Second, the *amount* of required resources may not be consistent across domains. For example, SCNN usually requires higher scratchpad bandwidth due to involving three data structures (weights, input, and output activation) for computation. Another example is scratchpad capacity: graph datasets are usually much larger than machine learning. Finally, computation almost always requires some flexibility; for example, SCNN and EIE use 16-bit adders and multipliers, while Graphicionado needs 32-bit units. Despite these challenges, the commonality is still powerful to save engineering effort to redesign and area compared to naively concatenating the accelerators.

4.4 Evaluation Methodology

In the CPU world, cycle-accurate simulators like gem5 [32] have been extremely useful for fast exploration and providing comparability against proposals. However, for accelerators, there are not any accepted simulators. The reason is apparent: all accelerators look vastly different and thus require a custom simulator. Developing custom simulators takes significant time, and without a common framework, it is hard to compare accelerators [128, 63].

We can imagine a framework that captures common abstractions across accelerators, but it needs to be fast – this requirement again comes from the fact that accelerators are designed from scratch. Thus, there are high degrees of freedom in the choice of hardware parameters, connections, algorithmic choices, etc. Therefore, many accelerator works perform cross-product analysis across many design points to find the Pareto optimal [178, 128]. The simulators need to be fast to evaluate design points in a reasonable time.

Our approach: This dissertation uses a combination of abstract and exact simulators. First, we performed the cross-product analysis using the abstract simulator. Once the high-level hardware features and parameter space are narrowed down, we used a cycle-accurate simulator to capture practical issues. See the details below:

- **Abstract Simulator for Fast cross-product analysis:** Our abstract simulator models accelerator architecture as a dataflow pipeline where each stage may perform computation or memory/scratchpad access. The simulator implements a variety of algorithm templates, which can be customized for each application. This is useful in the graph domain as many workloads can be implemented with slight variation. It also enables the accelerator control to be abstracted for efficient simulation. We developed an abstract simulator for graph processing (see Appendix A for more details). We used it to evaluate seven workloads and four dimensions of algorithm variants in graph processing with low development and simulation time (around 10× lower time than the detailed one).

- **Detailed Simulator for Architecture Validation:** We extended gem5 RISCV in-order core with our ISA extensions and reused gem5's memory and network model. Our detailed simulator (implemented as part of the DSAGEN framework [251]) helped capture practical issues; for example, our implementation created a deadlock by sharing a single network for memory and task packets. We also realized the programming complexity when writing programs for the detailed simulator.

  Interestingly, another outcome of our programmable accelerator research is a reusable simulator called DSAGEN [251]. Our simulator models an architecture description graph (ADG) where nodes can be either processing elements, switches, memory, delay, or synchronization element [251]. We find that by connecting these components in different fashions, we can model various accelerators. This framework is open-source [111] and recent work on wearable processors used it for modeling the MAERI accelerator using ADG graphs [33, 129]. We hope this framework provides a more accessible and consistent way to evaluate accelerators.

CHAPTER 5

Accelerating Workloads with Data-Dependent Control and Memory

Our baseline decoupled spatial hardware works well on regular workloads. This chapter focuses on the most basic category of irregular workloads with control and memory data dependencies, but the parallelism is known statically at a coarse granularity. Hence, the work is scheduled spatially and temporally to ensure a high locality. We identify two specializable forms of data dependence that are common and specializable: stream-join control and alias-free indirect memory.

We study machine learning (ML) as our primary domain and graph processing and databases to demonstrate generality. Chosen ML workloads cover the sparse and dense versions of the top-5 ML algorithms used by Facebook in 2018 [102]. SVM and GBDT are general workhorses of ML, with GBDT approaches often winning ML competitions (e.g., Kaggle). FC and CNN are the core kernels in state-of-the-art speech and image/video recognition. Arithmetic Circuits are graphical model representations that can be used to answer inference questions on probability distributions [216]. From graph processing, we study page rank and BFS. From databases, we study a subset of TPC-H. Table 5.1 outlines the algorithms from these workloads, which rely on data-dependent control or memory.

We evaluate our approach across three domains:

- **AI/ML:** SPU achieves 1.8-7× speedup over a similar GPU (NVIDIA P4000), using 24% power. Further, retaining the capability to express dense algorithms led to up to 4.5× speedup.

	Kernel	Stream Join (Irreg. Control)	Indirect Memory (Irreg. Memory)	Non-data-dep. (Regular)
Machine Learning (ML)	Conv	N/A	Outer-prod. [179] (sparsity)	Dense Conv [71]
	FC/ KSVM	Inner-prod. [153] (Sparsity+better at skewed dist.)	Outer-prod. [98] (Sparsity+better for large datasets)	Dense MV
	GBDT	Sort-based [46] (Sparsity+accuracy on weighted data)	Histo [275] (Sparsity+accuracy on unweighted)	Not Possible
	Arith. Circits.	N/A	DAG Travers. (dag sparsity)	Chain of MM
Database	Join	Sort [259] O(Nlog(N))	Hash join [122] O(N)	Cartesian $O(N^2)$
	Sort	Merge O(Nlog(N))	Radix O(N)	Not Possible
	Filter	Gen filtered Col [259] (sparsity)	Gen. Column Indices (sparsity)	Maintain Bitvector
Graph	Page Rank	Pull [5] (O(VE) + no rd/wr dependency)	Push [94] (O(VE) + No latency stalls)	Dense MM ($O(V^2)$)
	BFS	Pull [7] (O(VE) + better middle iters)	Push [94] (O(VE) + better beg/end iters)	Dense MM ($O(V^2)$)

Table 5.1: Data-Dependence Forms Across Algorithms

- **Graph:** For both ordered and unordered algorithms, we achieve 14.2× performance over a 24-core SKL CPU, competitive with the scaled-up Graphicionado [94] accelerator.
- **Database:** We achieve 10.3× over the CPU for database workloads, which is competitive with the Q100 [259] accelerator.

In this chapter, we first define the specializable irregularity forms and describe how we

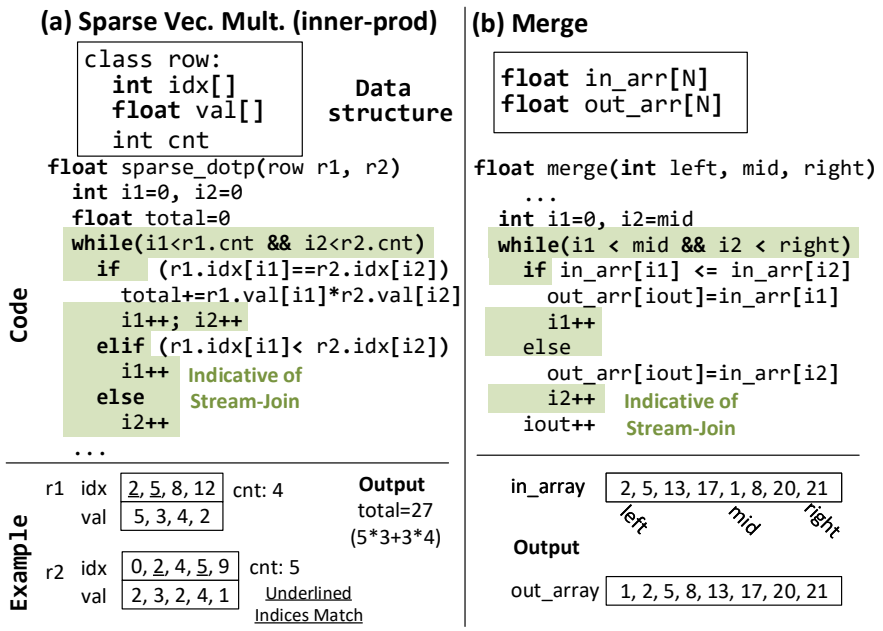


Figure 5.1: Example Stream-Join Algorithms

extend the decoupled spatial ISA to specialize for these forms (Section 5.1). Then, we discuss our enhancements to the baseline hardware to support stream-join and decomposable data types in the systolic-CGRA and compute-enabled scratchpad with aggressive stream reordering (Section 5.4). We discuss evaluation (Section 5.5, 5.6) and conclude with alternative decisions and pointers to future work (Section 5.8).

5.1 Specializable Data-Dependence Forms

We observe that two specializable forms of data-dependence are sufficient to cover many algorithms: *stream-join* and *alias-free indirection*. In this section, we first define these forms and give some intuition on their performance challenges for existing architectures, then explain how they guide our design.

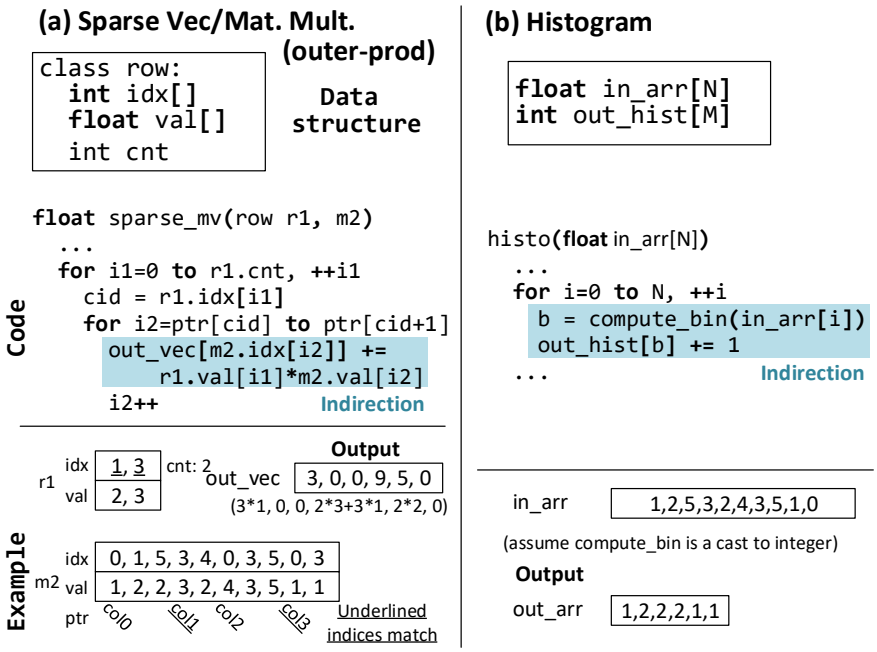


Figure 5.2: Example Alias-Free Scatter/Gather Algorithms

5.1.1 Stream-Join

An interesting class of algorithms iterates over each input (each stream) in order, but the total order of operations (and perhaps whether an output is produced) is data-dependent. Two relevant kernels are shown in Figure 5.1. Sparse vector multiplication (a) iterates over two sparse rows (in CSR format) where indices are stored in sorted order, and performs the multiplication if there is a match. The core of the merge kernel (b) iterates over two sorted lists, and at each step outputs the smaller item. Even though the data-structures, datatypes and purpose are very different, their relationship to data-dependence is the same: they both have stream access, but the relative ordering of stream consumption is data-dependent (they reuse data from some stream multiple times).

Stream Join Definition A program region which is regular, except that the re-use of stream data and production of outputs may depend on the data.

Problem for CPUs/GPUs Because of their data-dependent nature, Stream-joins introduce branch-mispredictions for CPUs. For GPGPUs, control dependence makes vectorization difficult due to control divergence of SIMT lanes; also the memory pattern can diverge between lanes, causing L1 cache bank conflicts.

Our Goal for stream-join Create a dataflow control model which can execute stream-join at full bandwidth and utilization.

5.1.2 Alias-Free Indirection (AF-Indirect)

Many algorithms rely on indirect read, write, and update to memory, often showing up as `a[f(b[i])]`. Figure 5.2 shows two examples: The sparse-vector/sparse-matrix outer product (a) works by performing all combinations of non-zero multiplications, and accumulating in the correct location in a dense output vector. Histogram (b) is straightforward. The similarities here are clear: both perform an access to an indirect location. This can be viewed as two dependent streams. Another important observation is that there are no unknown aliases between streams – the only dependence is between the load and store of the indirect update.

Alias-Free Indirection Definition A program region which is regular (including no implicit dependences), except that memory streams may be dependent on each other.

Problem for CPUs/GPUs On CPUs, indirect memory is possible with scatter/gather, however the throughput is quite limited given the limited ports to read/write vector-length number of cache lines simultaneously. As for indirect update, Intel AVX512 recently added support for conflict detection instructions. These *do not* improve the above cache-port throughput problem, only the instruction overhead – yet still any conflicts within the vector are *handled serially* with no reordering across vectors [108]. Also, not leveraging alias-freedom means a reliance on expensive load-store queues.

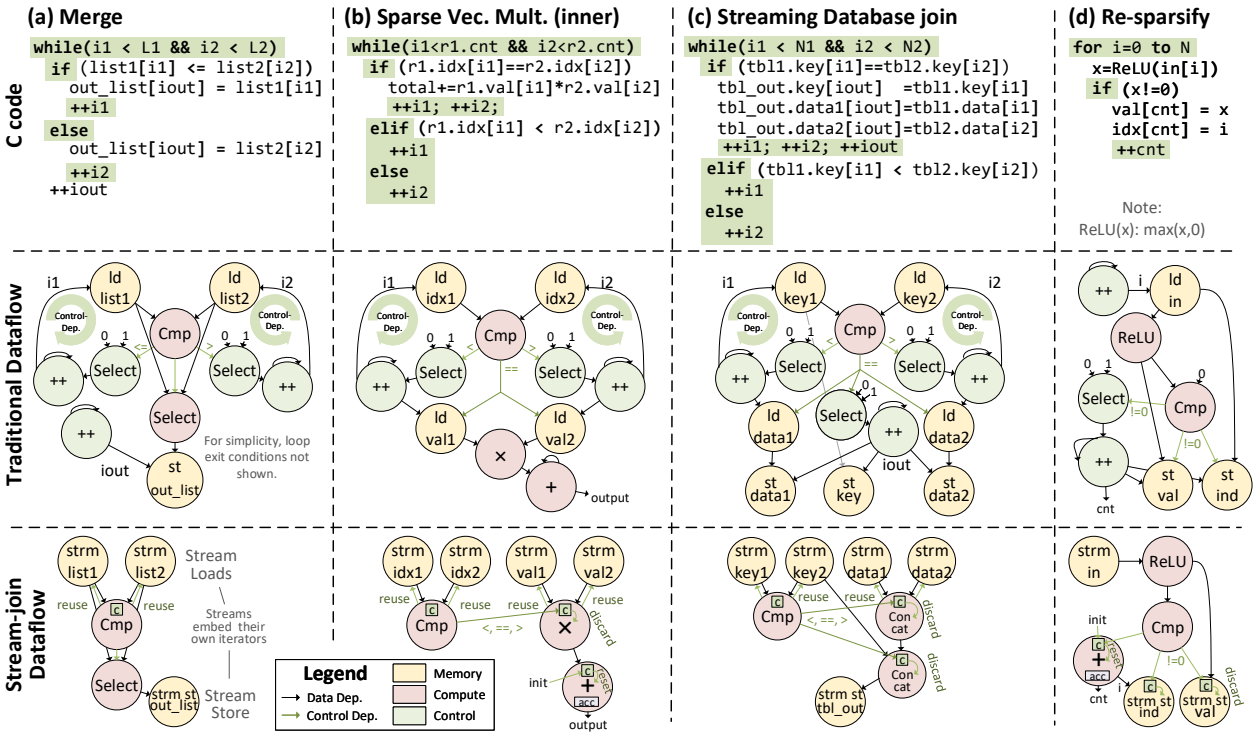


Figure 5.3: Stream-Join Control Model

While GPUs have similar throughput issues for caches, their scratchpads are banked for faster indirect access. However, they do not reorder requests across subsequent vector warp accesses [255], which is important to get high indirect throughput. Doing so in a GPU would require dependence-checking of in-flight accesses, as they cannot guarantee alias freedom.

Our Goal for AF-Indirect Create a stream-based hardware/software interface and microarchitecture enabling indirect access at full bandwidth through aggressive reordering.

Dependence Form Relationship Finally, note that dependence forms are not mutually exclusive. An example is the histogram-based Sparse GBDT (Figure 5.12 on Page 69). Alias-free indirection is used for updating the histogram count, while a stream-join is used for iterating over the sparse feature values. Other examples include deep neural networks (indirection for matrix-multiply and stream-join to subsequently resparsify the output vector) and triangle counting in graphs (indirection to traverse the graph and stream-join to find

intersecting neighbors).

5.2 Specializing Data-Dependent Control

A conventional computational fabric which is proven to perform well for non-data-dependent codes is a systolic-execution array [113, 48, 51, 143, 237], as they are quite simple. Note we define a systolic-execution array as a set of processing tiles which together form a deep pipeline, where each tile executes a single logical instruction and only communicates with its neighbors. This definition is general enough that such designs can include a circuit-switched network [218, 89, 151, 241, 164], so we refer to these as systolic CGRAs.

In this section, we propose a novel control model to enhance a conventional systolic CGRA for stream-join. We also discuss supporting finer-grain datatypes at low overhead and high hardware utilization through decomposability.

5.2.1 Stream-join Control

Existing systolic arrays are unable to make control decisions beyond simple predication, as they do not account for data dependences in deciding when and how to produce or consume data.

We discuss a number of examples in Figure 5.3, for which show the original code and a traditional dataflow representation. Here, black arrows represent data dependence, and green arrows indicate control. The dataflow representation is quite similar to what is executed on an OOO core. These examples motivate the need for a new dataflow-control model; one which can express the data-dependence without expensive throughput-limiting control dependence loops; this figure also shows these codes represented in our stream-join dataflow model.

Merge Example Consider the pseudo-code in Figure 5.3(a), which shows a simple merge kernel (only the part where both lists have data), for use in merge-sort for example. An item is selected and stored based on which of two items is smaller. This dataflow can be mapped

to a systolic array, but only at low throughput.

To explain, note that there is a loop-carried dependence through the control-dependent increment and memory access. This prevents perfect pipelining, and the throughput is limited to one instance of this computation every n cycles, where n is the total latency of these instructions. Note that the same problem exists for the out-of-order core, and it is made even worse with the unpredictable data-dependent branch which would increase the average latency due to mispredictions.

However, note that from the perspective of the memory, the control dependence is unnecessary, as all loads will be performed anyways. Therefore, to break the dependence, we need to separate the loads from computation (luckily, decoupled streams do this already), then expose a mechanism for controlling the order of data consumption. Intuitively for this example, if the model treats incoming values like a queue, it is possible to "pop" the values as they are consumed. Essentially, what we require here within the computation fabric is the ability to perform *data-dependent reuse*.

Sparse Inner-Product Example Figure 5.3(b) is a sparse-vector multiplication. Here, two pointers are maintained based on the comparison of corresponding item indices. Compared to merge, there is a similar control dependence and overhead. A similar approach could work here as well, decouple the streams and conditionally reuse indices (and values). The difference is that we only apply the multiply accumulate on matching indices, so we should discard some of this data. Therefore, in addition to data-dependent reuse, we also require *data-dependent discard*.

Database Join Example Figure 5.3(c) shows an inner equijoin. It iterates over sorted keys, and concatenates equivalent keys and corresponding columns. It has a surprisingly similar form and control dependence loop to the sparse multiplication, where the computation is replaced with concatenation. A similar approach of decoupling streams and applying data-dependent reuse and discard will break the control dependence loop and enable high

throughput.

Re-sparsification Example Re-sparsification (Figure 5.3(d)) produces a sparse row from a dense stream. The dataflow version has a predicated increment and store, and can achieve a pipelined schedule (so can the OOO core if it has predicated stores, otherwise it would be serialized by mispredictions). This example demonstrates that the ability to discard (ie. filter) is useful on its own. It is also an example where predication is enough, whereas predication is insufficient in the other examples.

Our Stream-join Proposal We find the desired behavior can be accomplished with a simple and novel control flow model for full-throughput systolic execution. The basic idea is to allow each instruction to perform the following control operations: re-use inputs, discard instructions or reset a register based on a dataflow input.

Figure 5.4 shows the execution flow of the sparse vector multiplication when expressed as a stream-join, showing the fully-pipelined execution over several cycles. Dataflow values are represented as circles, and for simplicity they take one cycle to flow along a dependence. Sentinel values (infinity for indices and zero for values) are used to indicate the end of a stream; these allow the other stream to drain on stream completion. Also, the subsequent vector multiplications can begin without draining the pipeline.

Figure 5.3 shows all of the examples written in this model. Data-dependent operand re-use is useful in (a,b,c) to iterate over input streams in correct relative order. Data-dependent *discard* is also useful in (b,c,d) for ignoring data which is not needed. The data-dependent *reset* is useful in (b,d) for resetting the accumulator. In both examples, adding stream-join primitives to instruction execution either shrinks the throughput-limiting dependence chain or eliminates it completely, enabling a fully-pipelined dataflow.

To enable flexible control interpretation, each instruction embeds a simple configurable mapping function from the instruction output and control input to the control operations:

$f(\text{inst\_out}, \text{control\_in}) \rightarrow \text{reuse1}, \text{reuse2}, \text{discard}, \text{reset}$

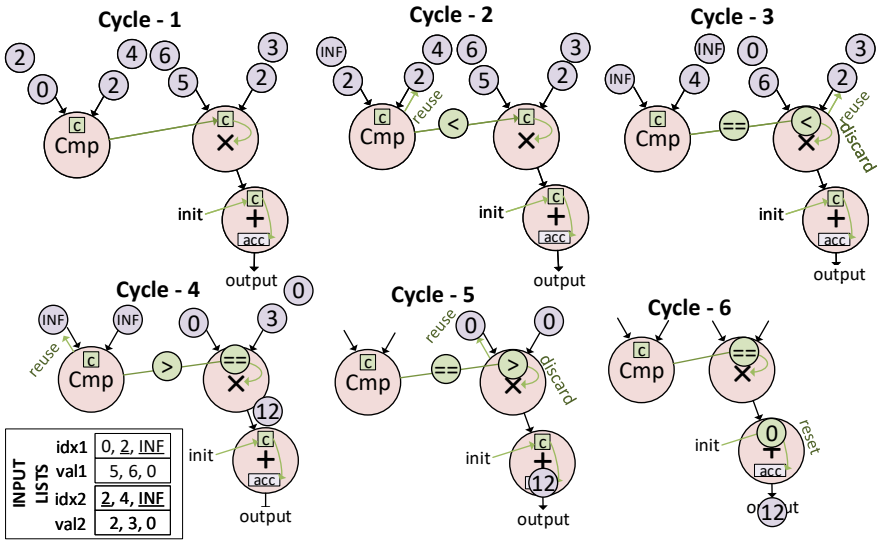


Figure 5.4: Execution diagram for join of two sorted lists.

Stream-join Overheads In kernels where input data is discarded (eg. sparse-matrix multiply and database join), the transformation to stream-joins can cause additional loads. Theoretically the worst case extra overhead compared to the original is $(value\_size+key\_size)/(key\_size)$. This can happen if there are extremely sparse matches, for example in databases. For such cases, we could only load values for matching indices; this would increase the latency of accessing values, but this can usually be hidden. For this, SPU provides efficient support for indirect accesses (Section 5.3).

5.2.2 Stream-join Compute Fabric: DGRA

Here we explain how we augment a systolic CGRA to support stream-join control. Its network is decomposable to support control semantics for smaller datatypes; thus we refer to the design as the decomposable granularity reconfigurable architecture: DGRA.

Stream-join Processing Element (PE) Implementation The stream-join control model enables an instruction to 1. treat its inputs as queues that it can conditionally reuse, 2. conditionally discard its output value, and/or 3. conditionally reset its accumulator. Instructions may use their output or a control input to specify the conditions (ie. the control info).

To implement, we add a control lookup table (CLT) to each FU (Figure 5.5), which determines a mapping between the control inputs and possible control operations. For the inputs of this table, we use the lower two bits of either the instruction output or the control input. For the outputs, there are four possible control actions: reuse-first-input, reuse-second-input, discard-operation, reset-accumulator. Therefore, for a fully configurable mapping between the 2-bit input (four combinations) and 4-bit outputs, we require a 16-bit table, and one extra bit to specify whether the instruction output or control input should be used as input. This becomes additional instruction configuration.

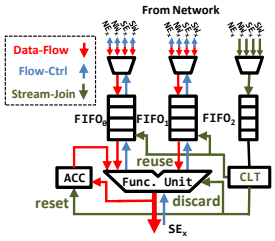


Figure 5.5: CLT integration

Supporting Decomposability To support stream-join semantics with arbitrary datatypes, our approach is to support the principle of *decomposability* – the ability to use a coarse grain resource as multiple finer grain resources. Therefore, the network of the DGRA is decomposable into multiple parallel finer-grain sub-networks. It provides limited connectivity between these sub-networks. For this we require both a decomposable switch and PE.

DGRA Switch Figure 5.6 compares a CGRA switch to our DGRA switch. On the left is an implementation of a coarse grain switch, which has one Mux per-output. The DGRA switch decomposes inputs and outputs, and separately routes each 16-bit sub-network. Flow control is maintained separately with a credit path (not shown) for each subnetwork. For flexible routing, we add the ability for incoming values to change sub-networks. In the design this is done by adding an additional input to each output Mux, which uses the latched output of the previous Mux. This forms a ring, as shown by the "X" inputs.

DGRA PE The decomposable PE (Figure 5.7) follows the same principles as the switch. Each coarse grain input of a FU can be decomposed into two finer-grain inputs which are used to feed two separate lower-granularity FUs. We replicate the CLT for each subnetwork so that each can have their own control semantics.

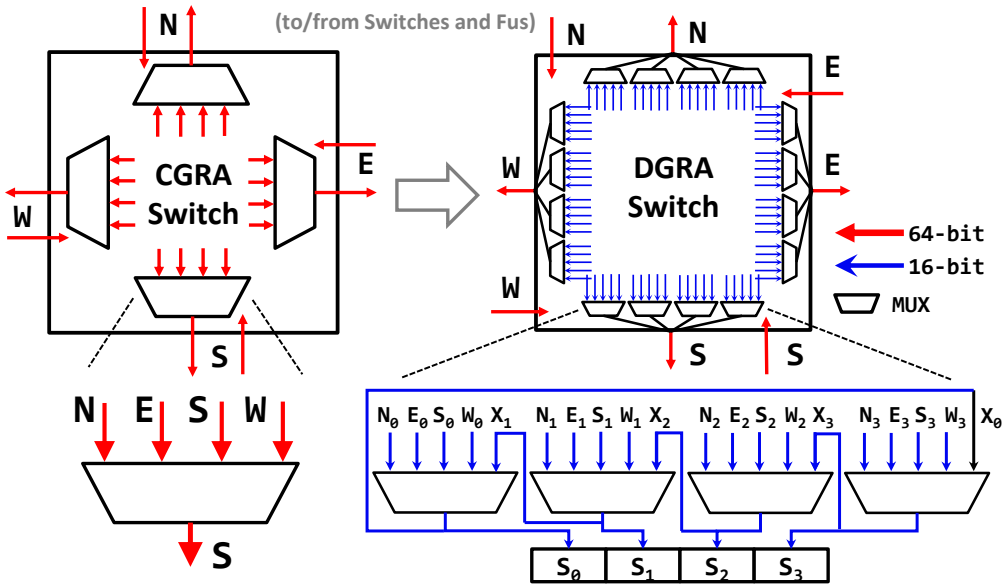


Figure 5.6: DGRA Switch

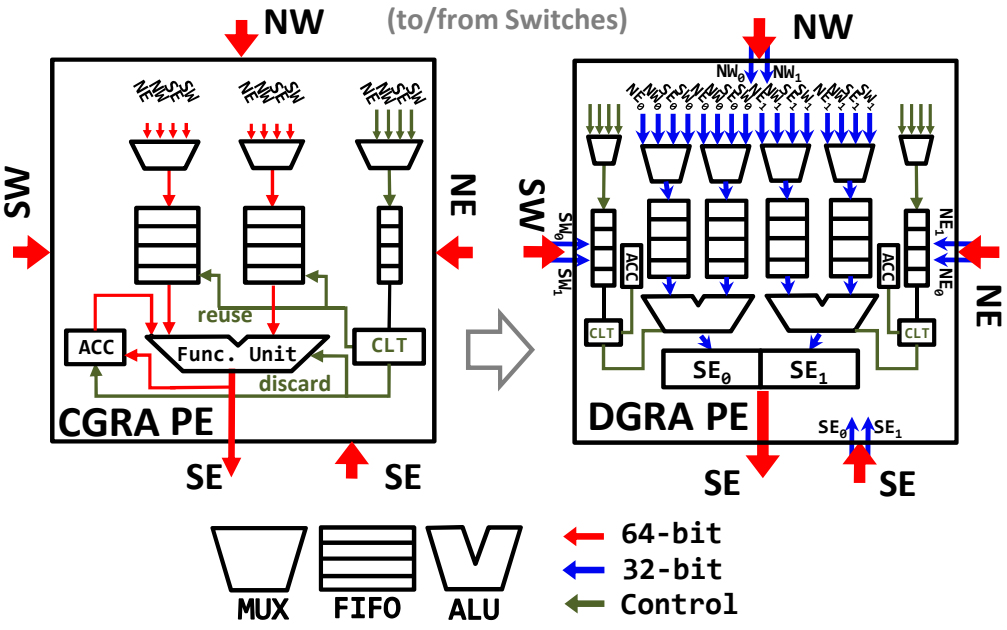


Figure 5.7: DGRA Processing Element

Mixed-precision Scheduling Mapping dataflow graphs onto reconfigurable architecture is known as spatial scheduling (eg. [180, 166, 181, 276]). Adding decomposability increases the complexity due to managing more routing decisions due to subnetworks. At a very high-level, our approach combines the principle of stochastic-scheduling [161] and over-provisioning (eg. within Pathfinder [150]). At each iteration, we attempt to map (or re-map) a dataflow instruction and its dependences onto several different positions on the DGRA; the algorithm will typically choose the position with the highest objective, but will occasionally select a random position. To avoid getting stuck in local minima, we allow over-provisioning compute and network resources and penalize over-provisioning in the objective function.

5.3 Specializing Data-Dependent Memory

The main challenge for specializing for alias-free indirection is creating a high-bandwidth memory pipeline which aggressively reorders accesses. In order to explain our proposed microarchitecture, we first discuss the set of stream abstractions that are expressed to the hardware.

5.3.1 Sparse Memory Abstractions

As we explained earlier, we start with a simple non-data-dependent contiguous stream (Listing 5.1 shows an example). For specifying indirect loads or stores, we enable one streams' addresses to be dependent on another streams' values. Often, *array-of-structs* style data structures require several lookups offset from the base address. We add this capability with an "offset list", shown in the example in Listing 5.2.

Indirect updates (as in histogramming) could hypothetically be supported by using an indirect load stream as above, performing the reduction operation, and finally using an indirect store stream to the same series of addresses. However, this requires dynamic alias detection or eschewing pipeline parallelism to prevent aliasing read/write pairs from being mis-ordered. Instead, we can leverage the alias-free property to add a specialized interface for

```
for i=0 to n                          →  load(a[0:n])
  ... = a[i]
```

Listing 5.1: Linear Stream

```
struct{int f1, f2} a[n]
for i=0 to n                              str1 = load(index[0:n])
  ind = index[i]                          ind_load(addr=str1, offset_list={0,4})
  = a[ind].field1                    →    ind_store(addr=str1, value = ...,
  = a[ind].field2                                          offset_list={0})
  c[ind] = ...
```

Listing 5.2: Indirect Load and Store Streams

```
for i=0 to n                              str_ind = load(index[0:n])
  ind = index[i]                          str_val = load(value[0:n])
  val = value[i]                     →    update(addr=str_ind, val = str_val,
  histo[ind] += val                              opcode="add", offset_list={0})
```

Listing 5.3: Indirect Update Stream

```
for i=0 to n
  size = sub_size[i]                      str_size = load(sub_size[0:n])
  for j=0 to size                    →    data_dep_load(value[0:len=str_size])
    val = value[j]
```

Listing 5.4: Data-dependent Load Stream

indirect update. In our implementation, indirect update may perform common operations like add, sub, max, and min directly on the indirect-addressed data item. Listing 5.3 shows an example.

Often, streams consist of sub-streams with data-dependent length. For example, indirect matrix-vector multiplication requires access to columns with varying size (Figure 5.2, page 53). We enable streams to specify a data-dependent *length*, as in Listing 5.4.

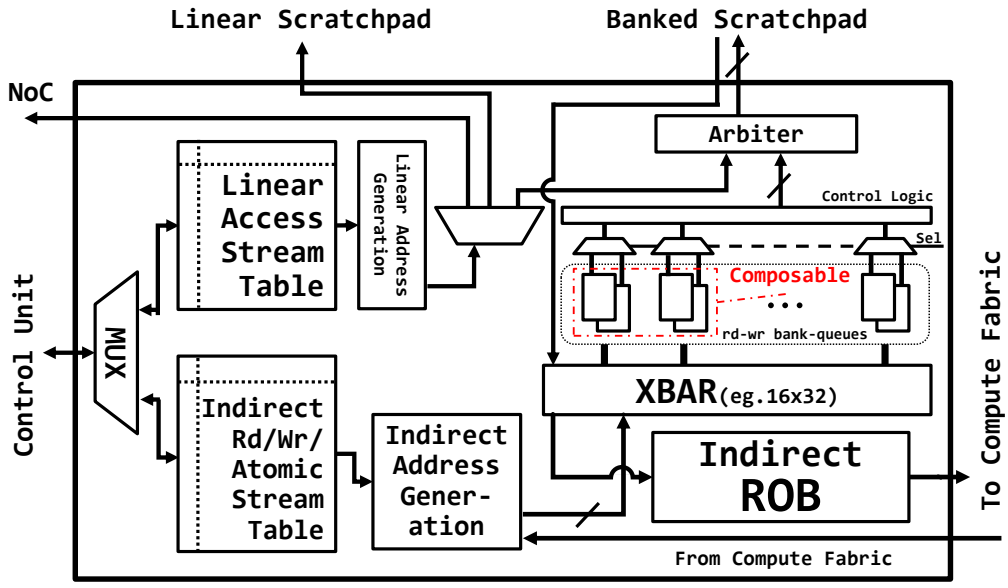


Figure 5.8: Scratchpad Controller

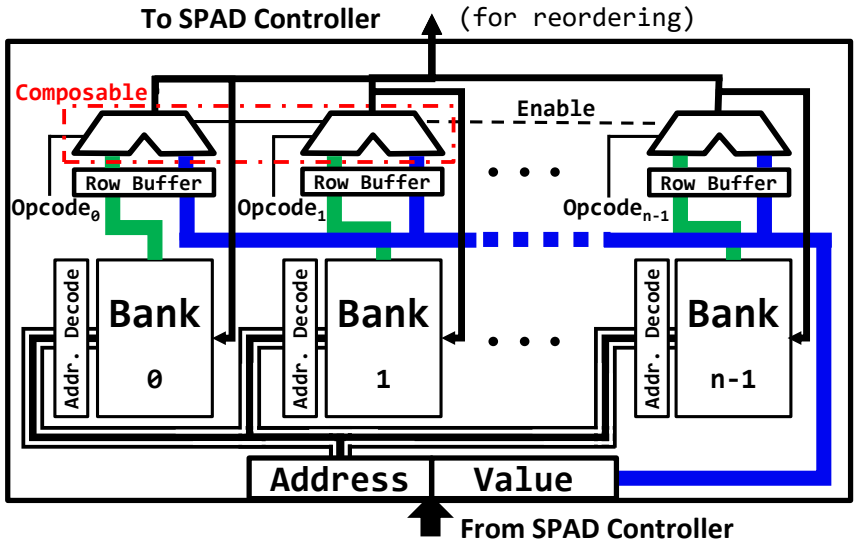


Figure 5.9: Compute-enabled Banked Scratchpad

5.3.2 Data-Dependent Memory Microarchitecture

Armed with expressive abstractions, we develop a high-bandwidth and flexible scratchpad controller capable of high-bandwidth indirect access. Because our workloads often require a mix of linear and indirect arrays simultaneously, for example streaming read of indices (direct) and associated values (indirect), we begin our design with two logical scratchpad memories, one highly *banked* and one *linear*. In this design, both exist within the same address space.

The role of the scratchpad controller (eventual design in Figure 5.8) is to generate requests for reads/writes to the linear scratchpad, and reads/writes/updates to the indirect scratchpad. A control unit assigns the scratchpad streams, and their state is maintained in either linear or indirect stream tables. The controller should then select between any concurrent stream for address generation and send to the associated scratchpad to maximize expected bandwidth. The linear address generator's operation is simple – create wide scratchpad requests using the linear access pattern.

The indirect address generator creates a vector of requests by combining each element of the stream of addresses (coming from the compute fabric, explained in Section 5.2) with each element in the stream description's offset list. This vector of requests is sent to an arbitrated crossbar for distribution to banks, and a set of queues buffer requests for each SRAM bank (Figure 5.9) until they can be serviced. Reads, writes and updates are explained as follows:

Indirect Writes Bank queues buffer both address and values. Importantly, because writes are not ordered with respect to anything besides barriers, requests originating from within the stream and across streams can be "mixed" within the bank queues without any additional hardware support. Mixing requests across multiple request-vectors helps to hide bank contention, a critical feature enabling higher throughput than traditional memories.

Indirect Updates Indirect updates use the compute units within the scratchpad. To explain, the bank queues buffer the address, operation type and the operand for the update.

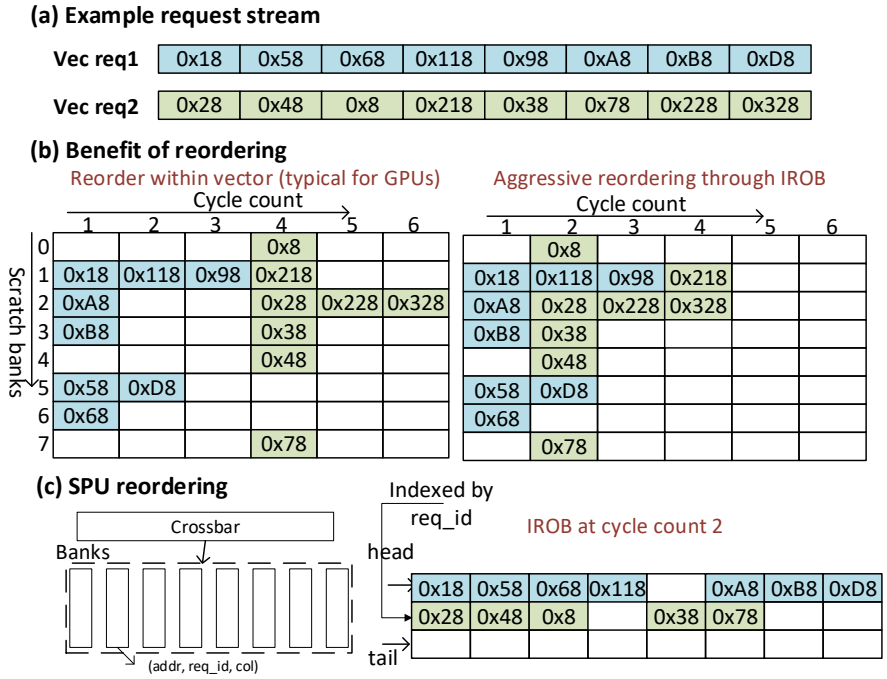


Figure 5.10: Functioning of IROB. (bits<6..4> indicate bank number)

Within the banked scratchpad, after the value is read, the associated compute unit executes, then writes the value back to the same location in the next cycle. We support only common integer operations within this pipeline (add, sub, min, max). The pipeline stalls only if subsequent updates are to the same address (max 2-cycle bubble).

Indirect Reads In contrast to the above, the order of *reads* must be preserved. For performance, we would like to maintain the ability to mix requests from subsequent accesses to hide bank contention. This actually goes beyond what even modern GPUs are capable of, as they only reorder a single vector of requests at one time [255, 152, 9]. We believe the reason for this limitation on GPUs is the challenge in handling potential memory dependences. To explain, Figure 5.10(a) shows an example of two parallel indirect read requests. Figure 5.10(b) shows the difference between how a typical GPU approach would schedule transactions, and how an aggressive reordering approach would work. The ability to intermingle parallel requests can significantly increase throughput.

To accomplish this, we maintain ordering in a structure called an indirect read reorder-

buffer (IROB), which maintains incomplete requests in a circular buffer. It is allocated an entry whenever a request is generated from the indirect address generator. For indirect reads, the bank queues maintain the address and row & column of the IROB. As results return from the banked scratchpad, they use this row & column to update the IROB. IROB entries are de-allocated in-order when a request's data is sent to the compute unit. Overall, our abstractions enable expression of the alias-free property of indirect reads in hardware, which is what allows a simple hardware structure like the IROB to aggressively reorder across multiple requests without memory dependence checking.

Decomposability The indirect scratchpad also requires decomposability to various datatypes. Multiple contiguous lanes are used in lock-step to support larger datatypes. Consider indirect store bandwidth for example: the 16×32 crossbar either supports 16 16-bit stores (to 32 logical banks), 8 32-bit stores (to 16 logical banks), or 4 64-bit stores (to 8 logical banks). We use the same approach for accessing the SRAM banks of the indirect scratchpad.

5.4 Sparse Processing Unit

The sparse processing unit (SPU) is our overall proposed design. Each SPU core is composed of the specialized memory and compute fabric (DGRA), together with a control core for coordination among streams. In this section, we overview the primary aspects of the design, then discuss how we map our workloads to SPU's computation, memory, and network abstractions. Finally, we discuss the role of the compiler and possible framework integration.

SPU Organization Figure 5.11 shows how SPU cores would be integrated into a mesh network-on-chip (NoC), along with the high-level block diagram of the core. The basic operation of each core is that the control core will first configure the DGRA for a particular dataflow computation, and then send stream commands to the scratchpad controller to read data or write to the DGRA, which itself has an input and output "port interface" to buffer data.

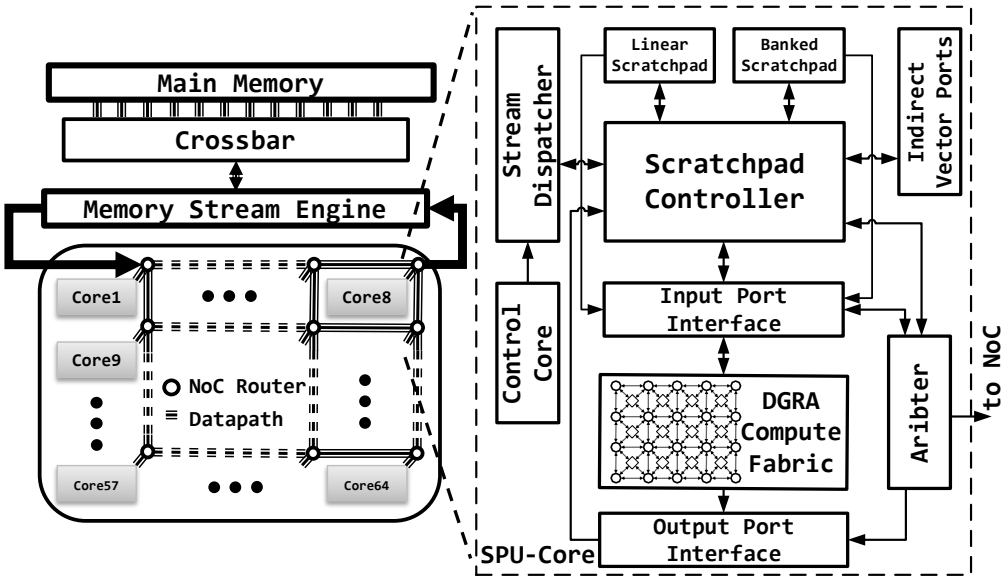


Figure 5.11: SPU Overview

Memory Integration These workloads require shared access to a larger pool of on-chip memory. To enable this, our approach was to rely on software support, rather than expensive general purpose caches and coherence. In particular, SPU uses a partitioned global address space for scratchpad. Data should be partitioned for locality if possible. Streams may access remote memory over the NoC. We add remote versions of the indirect read, write, and update streams. Indirect write and update are generally one-way communication operations, but we provide support to synchronize on the last write/update of a stream for barrier synchronization. Other synchronization is described next.

Communication/synchronization SPU provides two specialized mechanisms for communication. First, we include multicast capability in the network. Data can be broadcast to a subset of cores, using the same relative offset in scratchpad. As a specialization for loading main memory, cores issue their load requests to a centralized memory stream engine, and data can be multi-cast from there to relevant cores. For synchronizing on for data-readiness, SPU uses a dataflow-tracker-like [237] mechanism to wait on a count of remote-scratchpad writes.

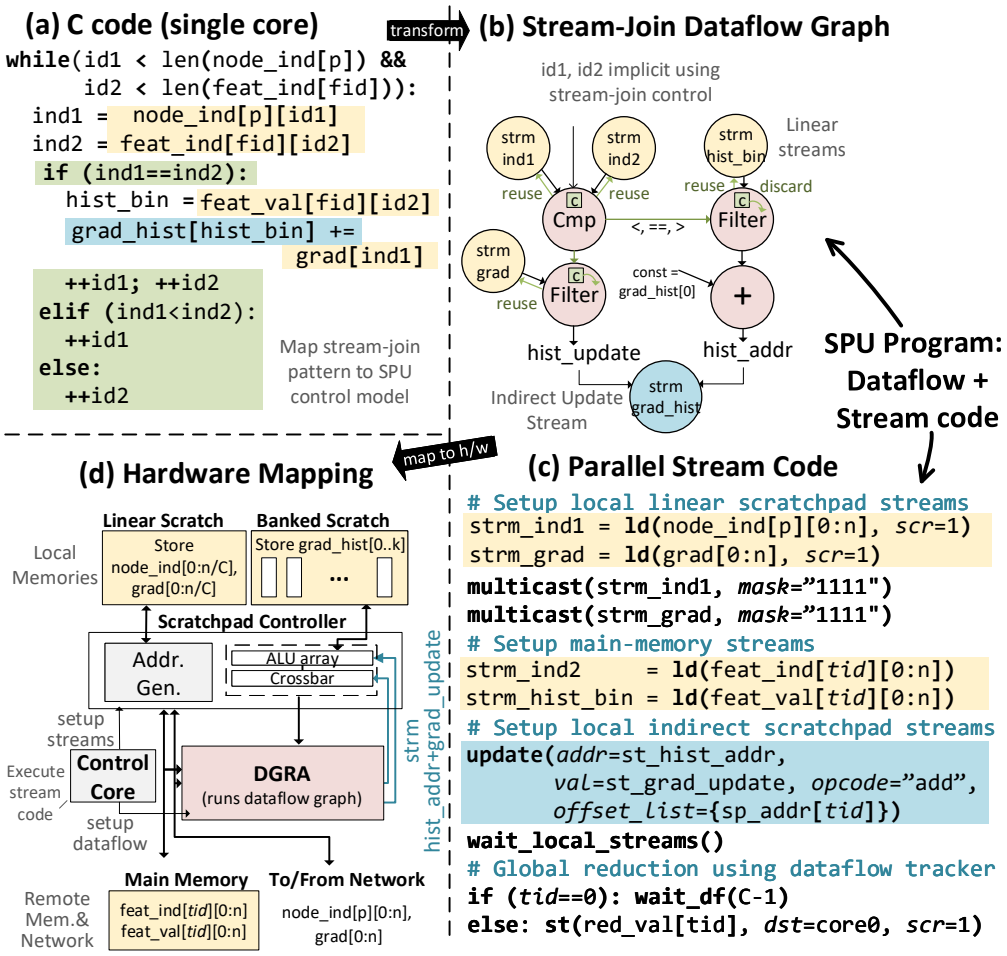


Figure 5.12: Example SPU Program Transformation: GBDT (Each core gets a subset of features to process i.e. fid=*tid*)

Control ISA We leverage an open-source stream-dataflow ISA [160, 162] for the control core's implementation of streams, and add support for indirect reads/writes/updates, stream-join dataflow model, and typed dataflow graph. The ISA contains stream instructions for the data transfer, including reading/writing to main memory and scratchpad.

Programming Model Programming SPU involves the following tasks: 1. partitioning work to multiple cores and data to the scratchpads to preserve locality, 2. extracting the dataflow graph, and possibly re-writing data-dependent control as a stream-join, 3. extracting streaming memory accesses, and 4. inserting communication/synchronization.

In terms of programming abstractions, an SPU's program consists of a dataflow graph language describing the computation (compiled to DGRA), along with a control program which contains the commands for streams (similar to stream-dataflow [162]). When a control program is instantiated, it is made aware of its spatial location, for efficient communication with its neighbors.

Example Program To explain how to map programs to SPU abstractions, we use the example of GBDT in Figure 5.12. We show the key kernel of this workload, which is a histogram over sparse lists. Figure 5.12(a) shows the original kernel's C code. Figure 5.12(b) shows the extracted stream-join dataflow, and (c) shows the control program where memory accesses are represented as streams, which is expressed as C + intrinsics. Each stream loads (or stores) data to an input (or output) in the dataflow. Figure 5.12(d) shows how the SPU program is mapped to hardware for this algorithm. In hardware, the stream code executes on the control core, which creates streams to be executed on the scratchpad controller. In turn, the controller will deliver/receive data to/from the DGRA compute unit, which executes the dataflow.

The basic parallelization strategy is that each SPU core independently builds histograms corresponding to its allotted subset of features. As for how memory is distributed, the dataset is stored in main memory in sparse CSR format (feat_ind and feat_val). Accessing these requires linear memory streams. The linear scratchpads store the subset of instances which belong to the current working node. As node indices are common across all features, the corresponding data is broadcast across all cores. This is done in synchronous phases: in each phase, the stream is loaded from a predetermined core. Phases are not shown in the figure for brevity.

As for the dataflow, stream-join is used to iterate sparse feature indices and the indices generated due to subsetting the data at each decision tree node.

Indirection is used for histogramming: the histogram address and update values are produced in the dataflow, which are then consumed by the indirect update stream. In

	Mech./ Wkld	Indirect memory	Stream join	Work partition across cores	Synchronization
Machine Learning (ML)	GBDT	Create feature histogram	Join train inst subset	Split features	Hierarch. reduce + broadcast
	KSVM	—	Matrix-mult for error calc	Split training instances	Hierarch. reduce + broadcast
	AC	Read + update child parameters	—	Split DAG levels	Pipelined communication
	FC	Accumulate activations	Resparsify	Split weight matrix rows	Broadcast of i/p activations
	CONV	Accumulate activations	Resparsify	Split weight matrix rows	Nearest neighbor comm.
Database	Merge Sort	—	Merge of 2 sorted lists	Uniform partition	Hierarchical Merge
	Hash Join	Cuckoo hash lookup	—	Smaller col replicated	Barrier until each core completes
	Sort Join	—	Join sorted lists	Equal-range partition	Same as above
Graph	Page Rank	Accumulate vert. rank	—	All vertices	Remote ind. 'add' update
	BFS	Relax vert. distance	—	Active vertices	Remote ind. 'min' update

Table 5.2: Mapping of Algorithms on SPU

hardware, the stream is mapped to the indirect stream table in the scratchpad controller.

Workload Mapping Table 5.2 details how we map each algorithm to the SPU architecture in terms of control, memory and communication ISA primitives, as well as the partitioning strategy.

Framework Integration We envision that SPU can be targeted from frameworks like TensorFlow [10], Tensor Comp. [236], TVM [47] for machine learning, or from a DBMS or graph analytics framework [5, 224]. For integration, a simple library-based approach can be used, where programmers manually write code for a given machine learning kernel. This is the approach we take in this work. Automated compilation approaches, eg. XLA [4] or RStream [189] are also possible if extended for data-dependent algorithms.

5.5 Methodology

SPU We implemented SPU's DGRA in Chisel [24], and synthesized using Synopsys DC with a 28nm UMC technology library. We use Cacti [156] for SRAMs and other components. When comparing to GPU power, we omit memory and DMA controllers.

Simulator Methodology We built an SPU simulator in gem5 [32, 200, 226], using a RISCV ISA [22] for the control core. The input to our simulator is a RISCV in-order core binary, and the output is cycle count. The simulator implements the hardware components and their connections according to the architecture diagram in Figure 5.11. We reused gem5's DRAM implementation that models the request scheduling and internal DRAM hierarchy including channels, banks, subarrays and row buffers at cycle-level. The simulator ticks each hardware component based on latency and throughput until termination to get the cycle count. Table 5.3 lists the latency and throughput of some hardware modules that we assumed based on our knowledge and prior work.

Architecture Comparison Points Table 7.3 shows the characteristics of the architectures we compare against, including their on-chip memory sizes, FU composition, and memory bandwidth. As for SPU, we provisioned the size of the DGRA to match the combined throughput of the scratchpads. We provisioned the total amount of memory on-chip for the working-sets of ML workloads, as they were our primary focus; this has some impact on workloads which have large working-sets and are expensive to tile.

Hardware	Latency	Throughput
Integer ALU	3	1
Float Div	12	1
Crossbar arbit	1	1
Spad latency	1	1
Addr. gen.	1	1

Table 5.3: Latency and Throughput of a Subset of Simulated Hardware Components.

Characteristics	**GPU** [61]	**SPU-inorder**	**SPU**
Processor	GP104	in-order	SPU-core
Cache+Scratch	4064KB	2560KB	2560KB
Cores	1792	512	64 SPU cores
FP32 Unit	3584	2048	2432
FP64 Unit	112	512	160
Max Bw	243GB/s	256GB/s	256GB/s

Table 5.4: Architecture characteristics of GPU, SPU-inorder and SPU

Workloads	**CPU**	**GPU**
GBDT	LightGBM [275]	LightGBM [275]
Kernel-SVM	LibSVM [42]	hand-written [23]
AC	hand-written [216]	hand-written [216]
FC	Intel MKL SPBLAS [2]	cuSPARSE [157]
Conv layer	Intel MKL-DNN [6]	cuDNN [52]
Graph Alg.	Graphmat [224]	-
TPCH	MonetDB [35]	-

Table 5.5: Baseline workload implementations

	Dataset	Size	Density	Dataset	Size	Density
GBDT	Cifar10-bn	50k,3k	1	Yahoo-bn	723k,136	0.05
#inst,#feat.	Higgs-bn	10M,28	0.28	Ltrc-bn	34k,700	0.008
KSVM	Higgs	10M,28	0.92	Connect	67k,700	0.33
#inst,#feat.	Yahoo	723k,136	0.59	Ltrc	34k,700	0.24
CONV	Vgg-3	802k,73k	0.47,0.4	Vgg-4	1.6M,147k	0.4,0.35
#act,#wgt	Alex-2	46k,307k	0.68,0.17	Res-1	150k,9.4k	0.99,0.1
FC	Res-fc	512,512k	0.26,0.84	Vgg-13	4K,16.8M	0.14,0.3
#act,#wgt	Alex-6	9K,37.7M	0.29,0.09	Vgg-12	25k,103M	0.42,0.06
AC	Pigs	622k	NA	Munin	3.1M	NA
#nodes	Andes	727k	NA	Mildew	3.7M	NA
Graph	Flickr	820K,9.8M	0.000015	NY-road	260K,730K	0.00005
#node,#edge	Fb-artist	50K,1.63M	0.0064	LiveJournal	4.8M,68.9M	0.000003

Table 5.6: Datasets

As for comparison to real hardware, the GPU is the most relevant. We choose the NVIDIA P4000, as it has a slightly larger total throughput and similar memory bandwidth to SPU. We do not include CPU-GPU data-transfer time.

We also address whether an inorder processor is sufficient by comparing against "SPU-inorder", where the DGRA is replaced by an array of 8 inorder cores (total of 512 cores). For reference, we also compared against a dual socket Intel Skylake CPU (Xeon 4116), with 24 total cores.

Workload Implementations We implement SPU kernels for each workload, and use a combination of libraries and hand-written code to compare against CPU/GPU versions. We compared against the best implementation (that we were aware of) for each workload on real hardware (Table 5.5). We implement kernels using both dense and sparse data-structures wherever possible (shown as SPU-dense/sparse).

Our choice of modest on-chip memory affects the implementations of graph processing and database workloads. For processing larger graphs, we follow a similar technique as

proposed in Graphicionado [94] to split the graph into "slices" that fit in on-chip memory. Edges with a corresponding vertex in another slice are instead connected to a copy of that vertex; duplicates are kept consistent. An architecture with a larger on-chip memory (such as Graphicionado [94], which has 32MB on-chip memory) means less duplicates, and less overhead. The tradeoff is also relevant for database workloads. Hash-joins require the hash-table to fit on-chip to perform well.

Benchmarks We used the datasets specified in Table 8.3. The uncompressed DNN model is obtained from Pytorch model zoo and the compression is done as described in [99] using distiller [8].

Domain-Specific Accelerator Modeling We model all domain-specific accelerators using optimistic models appropriate to the domain, always considering memory and throughput limitations of actual data.

1. **SCNN [179]:** We use a compute-bound model of SCNN according to the dataset density, assuming no pipeline overhead besides memory conflicts.

2. **EIE [98]:** Mechanistic model of EIE at maximum throughput. We compare against the scaled version of EIE with 256 cores.

3. **Graphicionado [94]:** We modeled a cycle-level approximation of its pipeline stages. We also compare against a version of this accelerator with the same peak-memory bandwidth as SPU by scaling Graphicionado to 32-cores and 32x32 crossbar.

4. **Q100 [259]:** For fair comparison to Q100, we restrict SPU to 4 cores (approximately the same area as Q100). We hand-coded query plans for Q100, specified as a directed acyclic graph in which each node indicates a database operation (join, sort, etc.) supported by the Q100 hardware, and edges indicate producer-consumer dependencies. Our model of Q100 is an optimistic execution of this query plan under memory and compute bandwidth constraints, which we verified against baseline execution time and

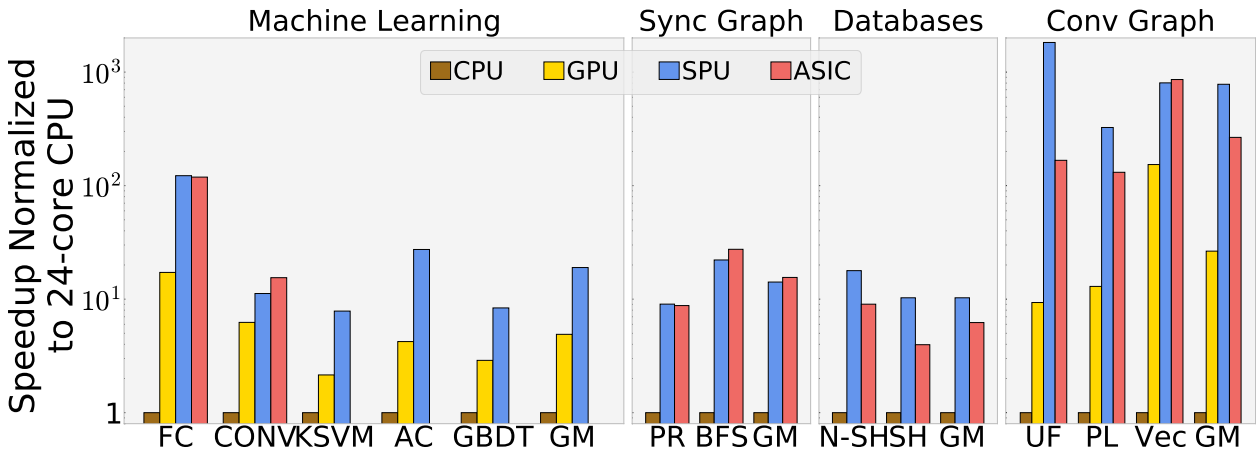


Figure 5.13: Overall Performance

speedups given by Q100's authors. This query plan is used as a reference for the SPU version, so SPU and Q100 implement the same algorithm as much as possible.

5.6 Evaluation

Our evaluation broadly addresses the question of whether data-dependencies exposed to an ISA (and exploited in hardware) can help achieve general-purpose acceleration. Here are the key takeaways, in part based on the overall performance results in Figure 5.13.

1. SPU achieves high speedup over CPUs (for ML:16.5×, Graph:14.2× and DB:10.3×), and GPUs (ML:3.87×).

2. Performance is competitive with domain-accelerators.

3. Relying on inorder cores only for supporting data-dependence is insufficient.

4. Architectural generality provides the flexibility to choose algorithmic variants depending on the algorithm and dataset.

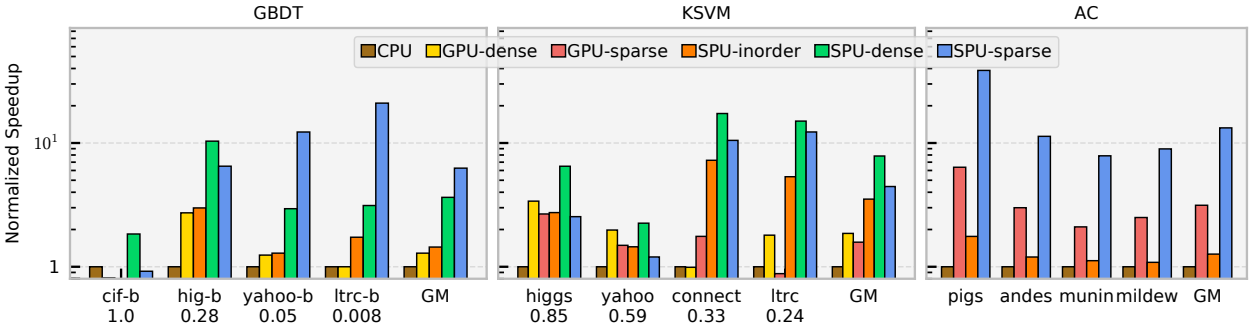


Figure 5.14: Performance on GBDT, KSVM, AC. (Computation density under benchmark name)

5.6.1 Performance on Machine Learning

Here we discuss the per-workload performance results on ML workloads, the breakdown for GBDT/KSVM/AC is in Figure 5.14 and for DNN is in Figure 5.15. During our analysis, we refer to Figure 5.16, which describes the utilization of compute, scratchpad, network, and memory within SPU.

GBDT Both GPUs and SPUs use a histogram-based approach, but SPU's aggressive reordering of indirect updates in the compute-enabled scratchpad far outperforms the limited reordering which GPUs can perform within a vector request. Further, SPU makes efficient use of multicast for communication of gradients. SPU-dense outperforms GPU dense, because histogramming is still required even with dense datasets. On a highly dense dataset like cifar, SPU-dense outperforms SPU sparse because of the extra bandwidth consumed by sparse data structures, which is an example of the benefit of having a flexible architecture.

KSVM SPU's network enables efficient broadcast and reduction. Since the dense version of KSVM is quite regular and the datasets are not sufficiently sparse, SPU-dense is generally better than its sparse version.

Arithmetic Circuits AC heavily uses indirect memory in the DAG traversal and data-dependent control (actions depend on node type) that we support efficiently. SPU's network

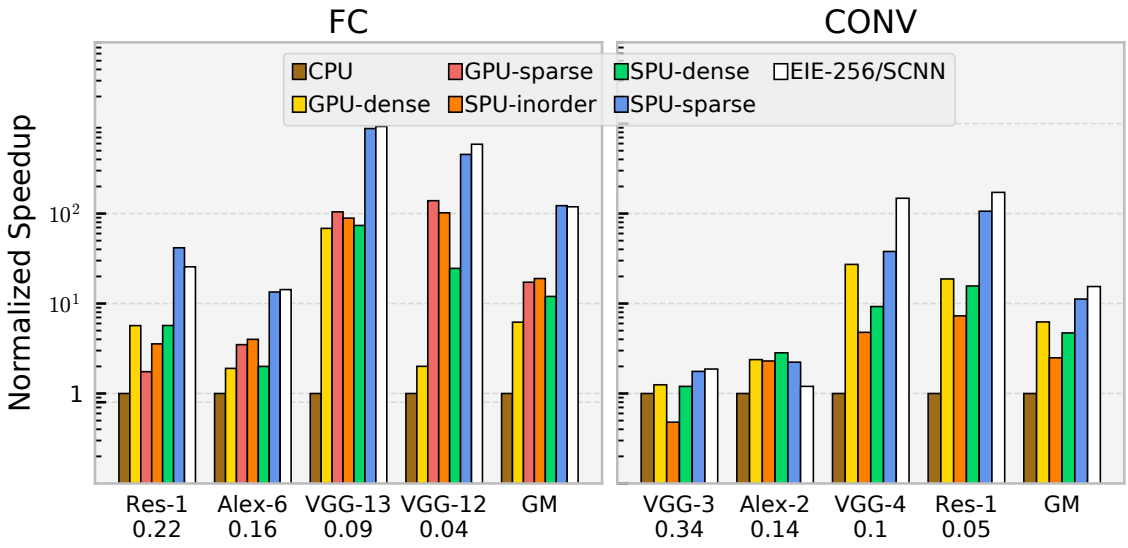


Figure 5.15: Performance on DNN. (Compute density under benchmark name)

enables efficient communication for model parallelism, which would otherwise need to go through global memory on a GPU.

Sparse Fully Connected Layers Figure 5.15 shows the per-workload performance for DNN. Using the alias-free indirection approach, we achieve high hardware utilization of the compute-enabled scratchpad. SPU outperforms GPU-sparse because it can also exploit dynamic sparsity of activations using stream-join.

Domain-accelerator Comparison: Compared to the EIE accelerator, SPU devotes more area to computation bandwidth and for providing high-throughput access to banked scratchpad, thus attaining similar performance at around half the area. Since the primary design goal of EIE is energy, it stores all weights in SRAM to save DRAM access energy; SPU trades-off lower area for higher energy.

Sparse Convolution The best GPU algorithm was a dense winograd-based CNN. SPU is able to save computations by exploiting sparsity through the outer-product convolution using indirect memory, and dynamic resparsification.

Domain-accelerator Comparison: The performance of SPU on average is 0.76× that

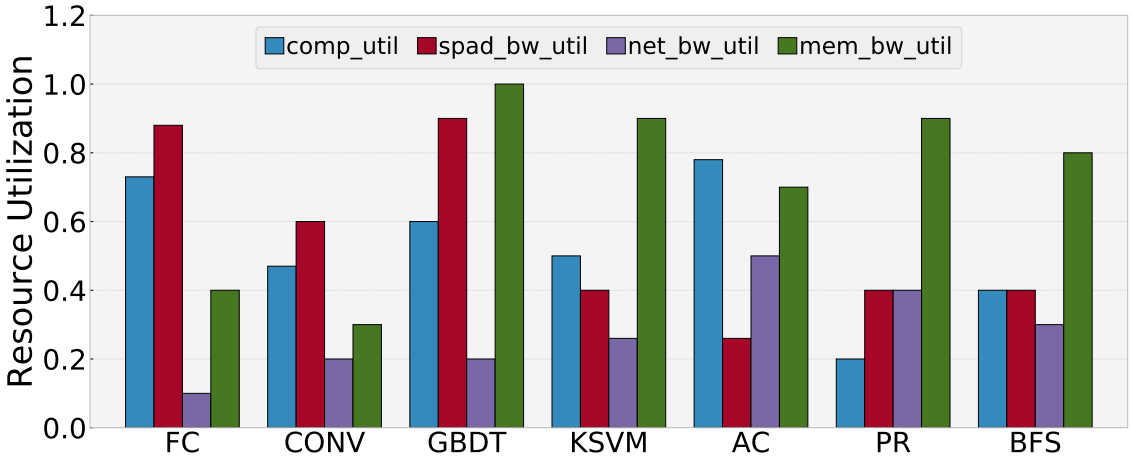


Figure 5.16: SPU Bottleneck on Machine Learning/Graph Workloads.

of SCNN. This is due to bandwidth sharing of the compute-enabled-memory scratchpad between computation and re-sparsification, whereas SCNN uses a separate non-configurable datapath. The performance difference increases for layers where re-sparsification is more intense. While comparing area is difficult, a simple scaling of SCNN's area suggests only 1.5× higher area for SPU (Section 5.6.5), a small price for significant generality.

SPU's Performance Bottleneck Figure 5.16 shows the utilization for primary bottlenecks in SPU. Bank conflicts are the bottleneck for DNN workloads, and the effect is reduced for the fully-connected layer. GBDT is bottlenecked by scratchpad and memory bandwidth. Since AC uses model parallelism, it is bottlenecked by the network.

5.6.2 Performance on Graph and Databases

Here we discuss the per-workload performance results on graph (Figure 5.17) and database (Figure 5.18) domains.

Graph Workloads SPU specializes the alias-free indirect updates to the destination vertices which would otherwise both be stalled due to load-store dependencies, and limited by inefficient bandwidth utilization due to accessing whole cache line for single accesses. For

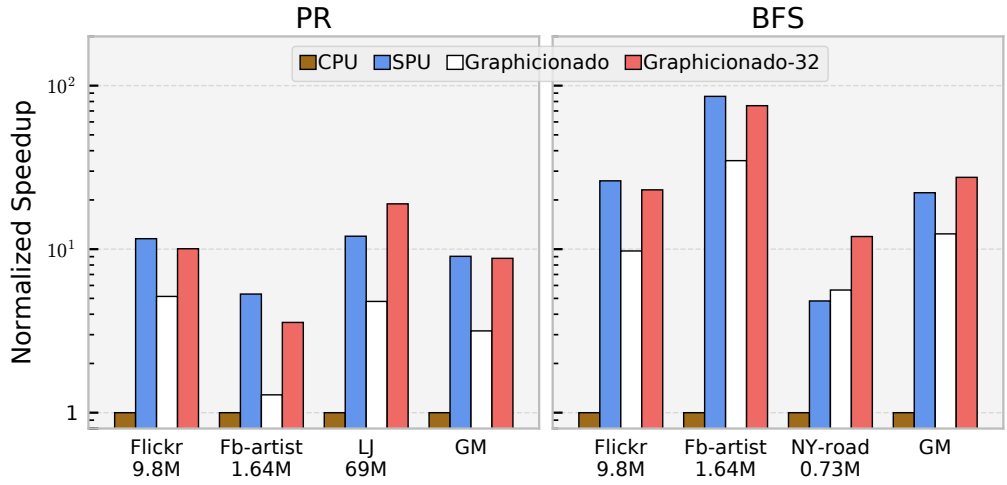


Figure 5.17: Performance on PR, BFS. (Edges under benchmark name)

SPU, the network experiences high traffic because of remote indirect updates (Figure 5.16).

Domain-accelerator Comparison: While the designs are quite different, SPU's performance is similar to Graphicionado (8-cores) as it is exploiting similar parallelism strategies: both have a way to efficiently execute indirect memory access on a globally-addressed scratchpad. Even for the scaled-up version of Graphicionado (32-cores), it only exceeds SPU slightly for road graph due to the network contention on SPU's mesh. LiveJournal graph is an example where the graph fits in on-chip memory of Graphicionado but needs to be broken in 10 slices to be able to run on SPU. Here, SPU is 57% slower than the scaled-up Graphicionado due to both network contention and extra memory accesses for vertices which are duplicated while slicing.

TPCH Queries Our primary goal in evaluating TPCH was to demonstrate generality. Figure 5.18 shows the per-query speedups of a 4-core SPU versus Q100, with three versions. SPU-dense allows only data-dependent discards (no joins or indirect memory on CGRA). Here, Joins and Sorts are performed on the control core. SPU-sparse (Join only) adds support for using the compute fabric for accelerating Sort (using merge-sort) and Join. When indirect-memory support is added, we additionally support hash-join if the smaller column fits within the scratchpad. With indirection enabled, we use a sort algorithm which

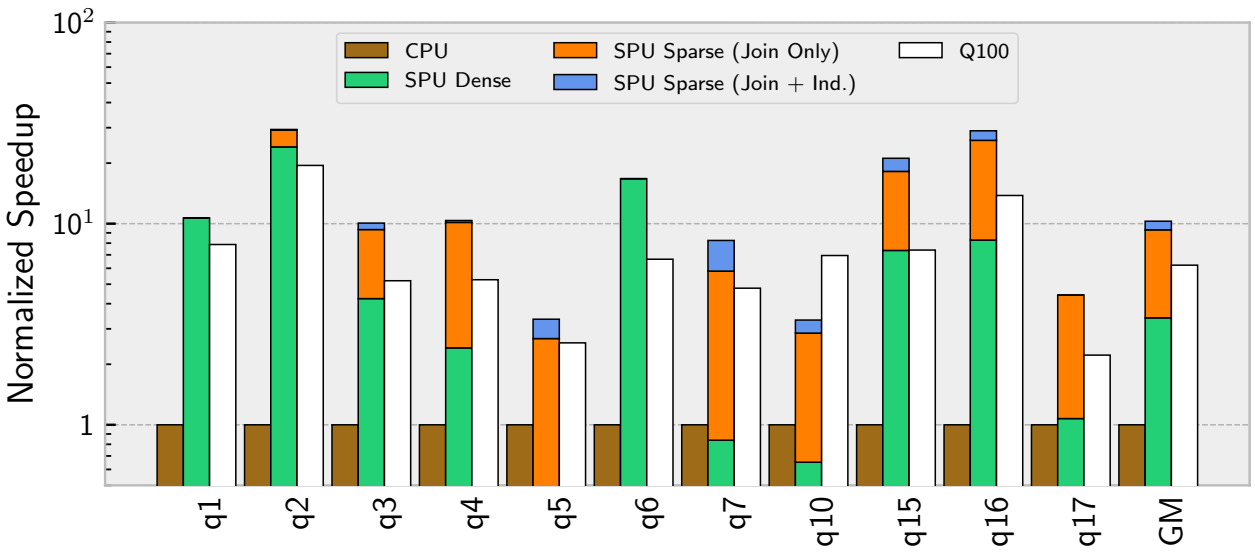


Figure 5.18: TPCH Performance comparison

applies radix-sort locally within local scratchpads, then use a merge-sort to aggregate across cores. Compared to CPU, SPU is significantly faster (10×), which is sensible given the significant data-dependence in queries, which serializes CPU execution.

Domain-accelerator Comparison: In queries which are non-sort heavy (Q1,Q2,Q6), the dense version of SPU performs adequately, and similar to the accelerator. On sort-heavy queries, stream-join within DGRA significantly reduces computation overhead, allowing SPU to catch up to Q100. Indirect access support helps to slightly improve sort's performance. Hash joins are significantly faster, but do not contribute much to speedup due to limited applicability (because of limited scratchpad size).

5.6.3 Sensitivity to Dataset Density

We demonstrate that it is useful to have both non-data-dependent and data-dependent support by studying performance sensitivity of a FC layer (Alex-6). Specifically we vary the dataset density with synthetic data, assuming uniform distribution of non-zero values. Figure 5.19 shows the performance comparison of different architectures executing matrix-vector multiply using dense and sparse data structures. At densities lower than 0.5, sparse

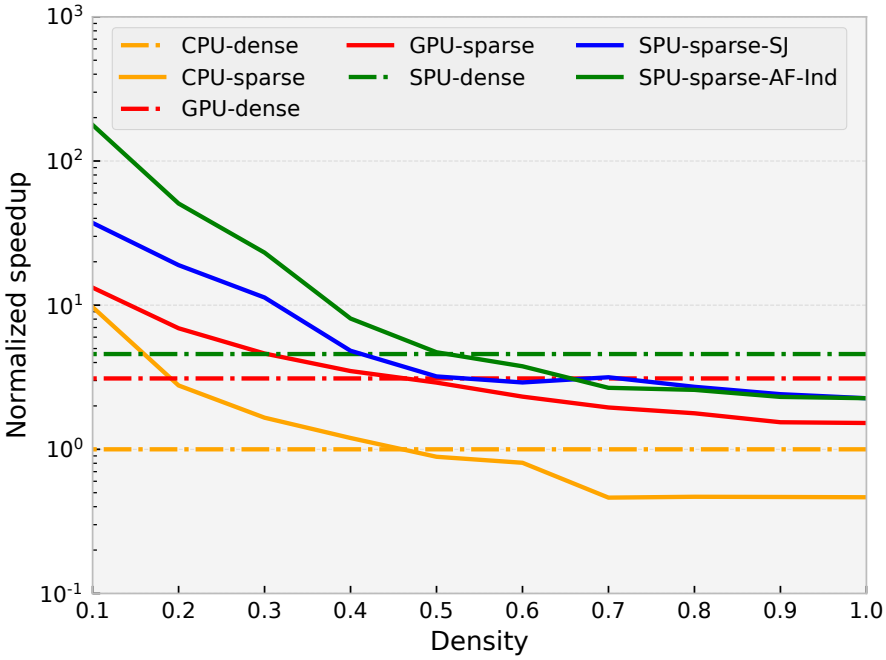


Figure 5.19: Performance Sensitivity (Matrix Multiply, dim: 9216×4096)

versions perform better as they avoid superfluous computation and memory access. However, for densities greater than 0.5, extra memory accesses due to using sparse data-structures reduces the benefit.

As for the different sparse implementations for SPU, alias-free indirection outperforms stream-join at low densities, because it can avoid unnecessary index loads. Their performance converges at higher densities when this overhead is relatively less important. We observe that for the problems which can be expressed using indirection or stream-join, it is often the case that indirection works better. We believe this is the reason why recent accelerators which exploit sparsity use alias-free indirection [179, 98]. However, there are kernels which might have an algorithmic advantage when expressed as a stream-join (examples in Table 5.1, Page 51). Indirection can also be inferior if there is not enough on-chip memory to hold the working-set, especially if the data is difficult to tile effectively.

5.6.4 Benefit of Decomposability

The speedup from decomposability is given below in Table 5.7.

Alg.	GBDT	Conv.	FC	KSVM	AC	BFS	PR
Speedup	2.27	2.67	2.67	2	3	2	1

Table 5.7: Performance Speedup With Adding Decomposability

To explain, GBDT uses 16-bit datatypes and gets 2.27× speedup, because although we can increase the compute throughput by 4, memory bandwidth becomes a bottleneck. Conv, and FC use 16-bit datatypes, but only see a 2.6× improvement: although the multiplications could be done using subword-SIMD alone, decoding run-length encoding of indices involves control serializing computation, which needs decomposability. AC involves various bitwidths (ranging from 1-bit boolean to 32-bit fixed point) coupled with control flow. As DGRA allows bitwidths as small as 8-bit, we can merge instructions with smaller bitwidths even if their control flow is different, to achieve 3× throughput. As BFS and KSVM use 32-bit datatypes, they can be fully combined using DGRA, for 2× improvement.

5.6.5 Area and Power

Sources of area and power Table 8.7 shows the sources of area and power for SPU at 28nm. The two major sources of area are the scratchpad banks and DGRA, together occupying more than 2/3 of the total; DGRA is the major contributor to power (assuming all PEs are active).

Overhead with decomposability Figure 5.20 shows the power and area cost of implementing the stream-join control and decomposability (simplest design is a standard 64-bit systolic CGRA). The stream-join control model costs about 1.7× area and power, and this is mostly due the complexity of dynamic flow control (rather than the control table). On top of this, decomposability costs around 1.2× area and power. Overall these are reasonable overheads given the performance benefits.

		Area (mm^2)	Power (mW)
Control Core		0.041	10.1
SRAM (banked+linear)		0.196	21.2
Data Vector ports		0.012	1.4
Scratchpad Controller		0.094	18.1
DGRA	Network	0.107	130.2
	FUs (4x5)	0.124	115.9
	Total DGRA	0.230	246.1
1 SPU Total		0.573	297.0

Table 5.8: Area and Power breakdown for SPU (28nm)

SPU's power and area comparison to the GPU Estimates below show SPU has 4× lower power.

Alg.	**GBDT**	**Conv.**	**FC**	**KSVM**	**AC**
SPU (W)	21.16	20.73	21.18	21.43	16.48
GPU (W)	84.87	84.02	84.92	85.42	75.60

Table 5.9: Power Comparison between SPU and GPU

5.7 Related Work

Table 5.10 gives a high-level overview of how we position SPU relative to select related work. In general, domain-specific accelerators target up to one form of data-dependence, while SPU has efficient support for both, and a domain-agnostic interface.

Domain-specific Accelerators Pudiannao [143] is an accelerator for multiple dense ML kernels. Several designs specialize sparse-matrix computations, including many for FP-

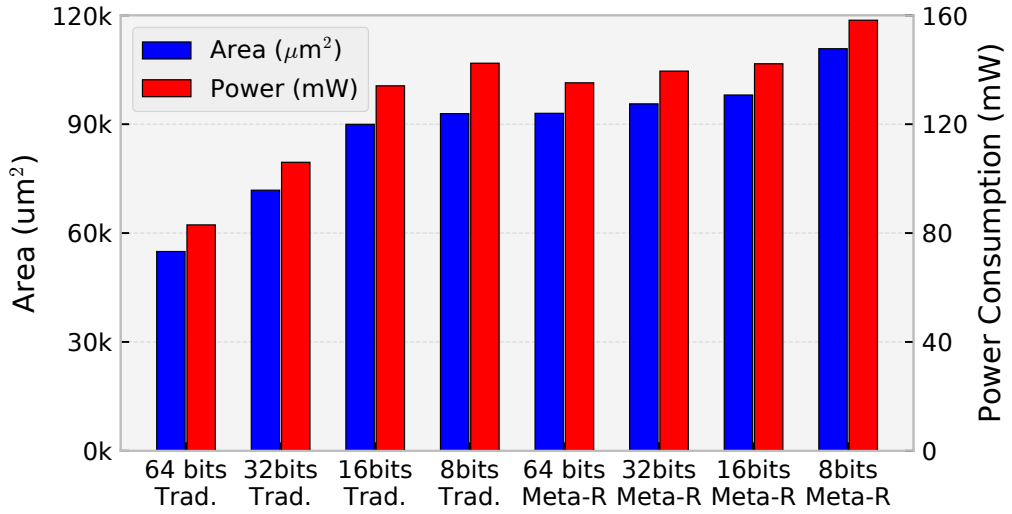


Figure 5.20: DGRA Area and Power Sensitivity

Architecture	Domain	Stream-Join	Alias-free Ind.	Non-data-dependent
Scnn/EIE[179, 98]	Sparse-NN	-	Very-High	-
Q100 [259]	DB	Very-High	-	-
Graphicion. [94]	Graph Alg.	-	Very-High	-
Sparse ML [153]	Sparse-MM	Very-High	-	-
PuDianNao [143]	NN	-	-	Very-High
Outersp [174]	Sparse-MM	-	Very-High	-
LSSD [164]	Agnostic	Low	Low	Very-high
Plasticine [188]	Agnostic	-	High	Very-high
SPU (ours)	**Agnostic**	**Very-High**	**Very-High**	**Very-High**
VT [125, 134]	Agnostic	High	High	High
Dataflow [225, 38]	Agnostic	Medium	Medium	High
GPU	Agnostic	Low	Medium	High
CPU	Agnostic	Low	Medium	Medium

Table 5.10: Analysis of Related Works (roughly least to most general)

GAs [283, 80, 90]. Nurvitadhi et al. propose a sparse-matrix accelerator specialized for SVM [167]. Mishra et al. develop an in-core accelerator for sparse matrices, and demonstrate generality to many ML workloads [153].

From the database accelerator domain, we draw inspiration from Q100's ability to perform pipelined join and filtering [259] to create our general purpose stream-join model. DB-Mesh [40] is a systolic-style homogeneous array for executing nested-loop joins. WIDX [122] is database index accelerator focusing on indirect memory access. UDP [76] targets encoding and compression workloads, which both express data-dependence.

Domain-agnostic Vector Accelerators Vector-threads (VT) architectures [125, 134, 198] have a flexible SIMD/MIMD execution model, where vector lanes can be decomposed into independent lanes to enable parallel execution for data-dependent codes. (GANAX [265] applies some of the same principles, but is specialized to ML). VT does not have spatial abstractions for computation, which SPU uses to expose an extra dimension of parallelism: pipeline parallelism. For example on VT, stream-joins would not execute at one item per cycle due to instruction overhead, but these computations can be pipelined on SPU. There are other less fundamental differences like SPU's support for programmer-controlled scratchpads with global address space.

Domain-agnostic Spatial Architectures LSSD is a domain-agnostic multi-tile accelerator with CGRAs and simple control cores [164]. However, it lacks support for data-dependent control or memory, so is far less general.

General spatial-dataflow architectures (eg. WaveScalar [225], TRIPS [38]) can perform stream joins, but at much lower throughput (due to control dependence loop, see Figure 5.3). Triggered instructions [177] and Intel's CSA [256] can perform pipelined stream-joins, but require much more complex non-systolic PEs (>3× higher area [197]), and are also not capable of decomposability. They are also not specialized for alias-free indirection, and require parallel dependence checking. In concurrent work, Master of None [145] proposes

a programmable systolic-style homogeneous reconfigurable array that can execute general database queries, as well as pipelined stream-join through a dedicated control network.

Plasticine [186, 188] is a tiled spatial architecture, composed of SIMD compute tiles and scratchpad tiles. Plasticine does not support stream-join dataflow, so would not be able to efficiently execute algorithms with this form of control-dependence. Plasticine also does not have compute-enabled globally-addressed scratchpads for high-bandwidth atomic update and flexible data sharing. Plasticine uses a parallel pattern programming interface [187, 123], while SPU provides a general purpose dataflow ISA, based on stream-dataflow [162]. While lower-level, SPU's ISA can more flexibly implement various computation/communication patterns. Recent work demonstrates efficient hash-joins for Plasticine [170]; such techniques could improve the performance and applicability of hash-joins in SPU.

Finally, lower precision control divergence is not supported in these architectures; SPU has the strongest support for arbitrary datatypes through its decomposable CGRA and memory. Note that while decomposability has been applied in other contexts, e.g. supporting multiple datatypes on a domain-specific CGRA [113, 214], we believe we are the first to apply decomposability to preserve independent flow-control.

General Purpose Processor Specialization The decoupled-stream ISA [247] allows expression of decoupled indirect streams for general-purpose ISAs. It also enables decoupling of memory from control flow in stream-joins. However, it does not specialize for the stream-join computation or high-bandwidth indirect access.

5.8 Discussion

In this section, we will discuss the alternate decisions that we could have made issues with the current design and the future work to address those issues.

Exploration of Algorithms on SPU SPU's flexible design allows us to explore various algorithm alternatives. For example, we studied three variants of sparse-matrix multiplication in Figure 5.19: dense, inner product (stream-join), and outer-product (alias-free indirection). Another interesting algorithm would be to perform stream-join on the indices and access the matched values using alias-free indirection. On a matrix of 9216x4096 size, we found that for densities lower than 0.2, the new algorithm gives 1.27× speedup over just stream-join by avoiding unnecessary memory accesses to values. For higher densities, stream-join on both indices and values performs better because streaming memory access is a more critical factor than avoiding unnecessary loads. Our new algorithm consistently performs close to the outer product implementation. At the same time, the new algorithm significantly reduces area requirements – LUT for stream-join consumes much less area than the on-chip crossbar for indirect updates. A future extension can perform a more systematic and aggressive analysis of novel algorithms enabled by SPU's design.

Vectorized Stream-join One of the limitations of stream-join is that it cannot parallelize within the joined lists. Therefore, parallelism must exist across the number of parallel joins, which may often be infeasible. Future works after SPU's publication studied bitvector-based comparison to achieve parallel stream-join (e.g., Capstan [202], Sparten [85]). The challenge will be to integrate bitvector-based comparison into the flexible execution model.

CHAPTER 6

TaskStream: General Task Framework For Accelerators

SPU, developed in the previous chapter, shows relatively broad applicability by improving flexibility for data-dependent control and memory. Yet many essential data processing workloads cannot be described (or are not efficient) because of the limitations of static scheduling. In the following three chapters, we will focus on irregular parallelism, commonly known as *task parallelism*: this occurs when a program's work is created and scheduled to "execution resources" dynamically, based on runtime computations. Task parallelism has applications at all scales: across cores within architecture and distributing work across accelerators.

The existing solution is compilation-based – the tasks are statically assigned to resources, and inter-task dependencies are satisfied using explicit synchronization barriers. The problem with this solution is that static data orchestration provides limited benefits because of a lack of runtime information like amount of work, time of creation, etc. Also, the synchronization barriers are especially problematic in an accelerator system where tasks are short. Finally, these compilers are often proprietary and do adhoc application-specific analysis. For example, REVEL supports primitive for transferring data to the right neighbor [253], FAST provides model parallelism in DNNs [270], and network multicast in SCNN [179] and EIE [98]. We observe that task scheduling primitives occur broadly in applications, and a shared interface will be handy to systematize the work.

Insight Our insight is that *task* abstractions will be helpful first-class aspects of the ISA. A task is the smallest unit of work that can be dynamically scheduled in space or time.

The annotations about relationships between tasks can encode communication patterns that appear in many workloads (and are supported in accelerators), providing broad applicability while improving performance using specialized hardware.

Besides systematizing task scheduling, handling tasks dynamically in the hardware can help in at least three additional scenarios. First, many workloads have inherent data dependencies in forming parallel work (e.g., creating tasks for all outgoing edges of a graph's vertex), enabling broader applicability. Second, sometimes the amount of work per task can only be determined at runtime (e.g., number of elements matched in a join), so dynamic assignment can balance the load. Third, many irregular workloads have multiple task types. Each type stresses the system differs in terms of compute, memory, or network (e.g., memory-bound graph aggregation and compute-bound multiplication in Graph Convolution Networks - GCNs). One could dynamically overlap the execution of different task types to balance shared resource usage.

Requirements from Task Interface Using this task framework, we would like to enable optimizations along three dimensions below (Figure 6.1b pictorially represents these):

- **Inter-task reordering** The flexibility to reorder tasks can allow controlling the memory access/reuse pattern and convergence rate of learning algorithms. We find that "priority scheduling" is suitable for fine-grained tasks as it enables faster convergence, while for coarse-grained tasks, we exploit reuse by batching tasks that read the same data.

- **Resource assignment** It determines the spatial location where a task is executed. A flexible framework can optimize locality by scheduling near-data (critical for fine-grained tasks due to scalar random accesses) or load balance by uniformly distributing work (critical for coarse-grained tasks due to fewer tasks).

- **Inter-task communication** Dynamic distribution of work precludes several locality optimizations. For example, a programmer could place data in the local scratchpad

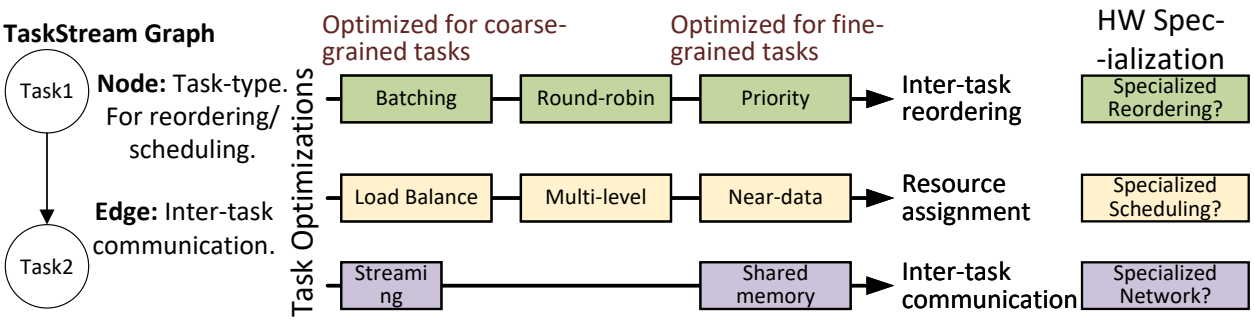


Figure 6.1: Overview of optimizations in TaskStream.

bank, but the access may still be remote since tasks are moving around. However, the hardware can dynamically coordinate tasks for specific inter-task communication patterns. Specifically, we find data streaming and multicast are essential for the locality.

In this chapter, we first describe the TaskStream framework and then discuss the requirements from the execution model and hardware.

6.1 TaskStream Execution Model

Challenges The critical challenge in designing a flexible task-parallel framework is to do so without losing accelerator-level efficiency. Reflecting on SPU, it uses two fundamental principles for efficiency: 1. **Infrequent reconfiguration:** CGRAs need long computation phases to avoid stalls during frequent reconfiguration. With tasks, resources will need to be configured repeatedly for each new task, so the hardware performs correct computation. 2. **Spatial abstractions:** Accelerators support data multicast to exploit spatial reuse on data that does not fit in on-chip memory. Multicast requires all tasks to be scheduled concurrently. When tasks are moving at runtime, dynamic coordination is required in the hardware.

Existing task parallel frameworks are not suitable for reconfigurable architectures. For example, Cilk dynamically generates task graphs, where new tasks may execute arbitrary functions (and may not necessarily be already known/configured). Thus, they cannot be immediately applied to reconfigurable accelerators. Moreover, Cilk's runtime scheduling

techniques fundamentally rely on non-accelerator-friendly features. For example, they do not support data multicast; instead, they rely on cache coherence for the locality. Coherence transparently moves data close to where the task is executed (i.e., brings the data to the private cache). Accelerators usually have scratchpads or shared caches.

6.1.1 TaskStream Program Representation

Two major requirements drive TaskStream's design: no need for frequent reconfiguration and flexibility to expose inter-task reordering, resource assignment, and inter-task communication primitives. Here we define the TaskStream graph, describe a task's lifetime written in this program, show an example program, and conclude with the limitation of TaskStream program representation.

TaskStream Definition A program in TaskStream is represented as a set of nodes, one for each task type and edges for inter-task dependencies (see Figure 6.1a). A task type corresponds to a specific dataflow graph that can be configured on a reconfigurable accelerator. We use a type rather than a function because data processing workloads typically only require 3 or 4 tasks types, and this makes supporting reconfigurable hardware straightforward (one can configure 3-4 unique dataflows). Internally, task types may be implemented using arbitrary execution models. Figure 6.1 shows an hypothetical example – here, Task1 uses dataflow (e.g., [88, 162]), and Task2 uses SIMD (e.g., [110, 62]).

Nodes and edges are typed – node type indicates inter-task reordering and resource assignment, and edge type means the inter-task communication optimization. Tasks are created when they receive values for all incoming edges. In the case of inter-task communication edges, the new task may wait for the producer task that will supply data. Ready tasks are scheduled using node-type information. The hardware implicitly handles any contention for task execution resources (using flow control protocol or overflowing extra tasks to memory).

One phase of the program completes when all tasks are completed. A program may consist of multiple phases. An example of a TaskStream program phase is shown in Figure 6.2, where

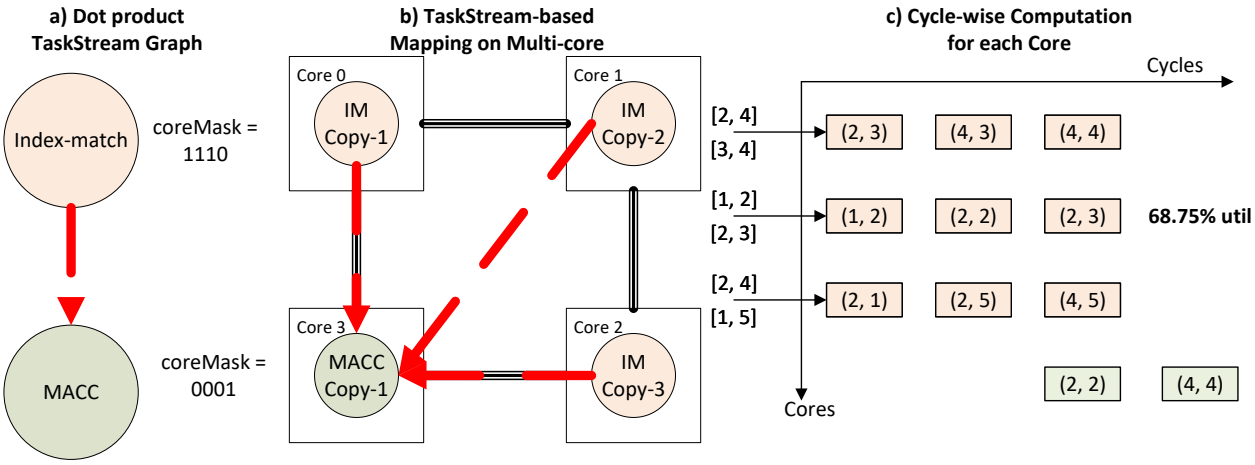


Figure 6.2: Sparse Dot Product Example Written in TaskStream.

task types are distinguished by color (and shading). We will provide details of how we port a C/C++ workload to the TaskStream programming model in Section 8.2.2.

Example Program in TaskStream Figure 6.2 shows the sparse dot product written in TaskStream. It has two task types: Index-match and MACC operations. Tasks are annotated with *coreMask*, a bitset where a set bit indicates the legal locations where a task can be scheduled (Chapter 8). For this example, Index-match tasks can be scheduled at three cores (Core 0,1,2), while the MACC operations are required only for matched indices; hence MACC tasks will always be executed on Core 3. During execution, three comparisons can keep a single MACC core busy, achieving better utilization than static-parallel SPU implementation (where each index-match is assigned its MACC operation). Figure 6.2c) shows an execution timeline where each of the Cores 0,1,2 are joining two lists. Core 3 works on the matched indices from either of Cores 0,1,2.

Since the TaskStream graph is exposed as a first-class citizen in the ISA, it enables us to use specialized hardware support for task management (scheduling and inter-task communication). We believe TaskStream opens the door for efficient task parallelism in accelerators.

	Fork	Dep-chain	Fork-join	May-alias
Aurochs [238]	✓			
Fifer [158]	✓			
PolyGraph [63]	✓			
TaskStream [65]	✓	✓		
ParallelXL [45]	✓	✓	✓	
Swarm [110]	✓	✓	✓	✓

Table 6.1: Dynamic Task Parallel Patterns Supported by Prior Works

6.1.2 Limitations of TaskStream

Task parallelism can cover a broad space. Table 6.1 broadly classifies existing work by supporting four task dependence patterns: 1. **Fork, i.e., Asynchronous Tasks:** The parent task forks/spawns an asynchronous task but the child task's completion is not tracked, 2. **Fork-join Patterns:** The parent task may spawn multiple child tasks and can choose to wait on their completion, 3. **Linear Dependence Chain:** It is a particular case of fork-join where the parent only spawns a single child task. When the spawned task in dependence chain receives and/or produces a single sequence of values, we call it streaming across tasks, 4. **Implicitly Dependent Tasks:** The tasks are assigned a timestamp order. These tasks do not require explicit synchronization, but they *may* have memory dependencies. The hardware must ensure sequential view of execution for correctness. TaskStream supports a subset of critical patterns (fork and linear dependence chain) to minimize the overhead while improving applicability. Below we list the *required* properties for efficiency on TaskStream:

1. **Explicit Dependencies:** Our streaming memory model assumes explicit dependencies. Thus, the hardware only enforces known dependencies and does not need to track them using expensive load-store queues [110, 101]. Consequently, we cannot support workloads like billiard simulation [101], where tasks are implicitly dependent due to shared data.

2. **Sufficient Parallelism within a TaskStream Phase:** If any data dependence is not supported in TaskStream edges (i.e., not dependence chain/streaming), it will be enforced by a synchronization barrier. Between barriers, there should be enough parallelism to achieve high utilization. An example is discrete event simulation: here, the parallelism between independent events are insufficient, and an architecture like Swarm that supports speculation will be required [110].

Overall, TaskStream provides a flexible program representation with broad applicability while retaining accelerator-like efficiency.

6.2 Fundamentals of the Approach

This section describes the fundamental requirements for the ISA/runtime and their implications for the hardware.

1. **Task Type and Resource Allocation:** The ISA should expose the task types, their functionality, and dependencies. In our implementation using coreMask (Chapter 8), we assigned physically different cores/architectures to different task types. Thus, our ISA also exposed the resource distribution information. The coreMask allows the programmers to specify the number of computation resources for each task type; hardware must be reconfigurable to support coreMask. An alternative method to perform load balance is to time-multiplex each core across different task types, as in Fifer [158] but we leave this to future work.

2. **Data Mapping:** The ISA should expose information on data mapping to support near-data scheduling. The hardware will require a unit for spatial task assignment that utilizes data mapping information to determine where a task should be scheduled. One could expose this information at either distributed (Chapter 7) or centralized task queues.

3. **Inter-task Communication:** Applications often share data between threads at a fine grain; therefore, the ISA should expose the index to the shared data structure for communicating tasks. The hardware would need a reordering unit to co-schedule communicating tasks. For exploiting reuse, either a network multicast support for spatial locality (Chapter 8) or on-chip cache storage for temporal locality is required.

4. **Globally Unique TaskID and Message Passing:** The hardware must assign globally unique TaskIDs for dynamically coordination among tasks (Chapter 8). The runtime identifies these communicating tasks and exchanges handshaking messages (e.g., for resource availability). Note that the runtimes of standard distributed frameworks already support message passing in software. For example, Apache Spark keeps a master node coordinating work/tasks to other "slave" nodes [268].

Our Implementation Applying TaskStream to our reconfigurable accelerator naturally creates a hierarchical-dataflow representation: one higher-level dataflow of task management and communication and one lower-level dataflow of instruction execution. In the following two chapters, we will discuss our implementation of TaskStream and its optimizations. We discuss the performance implications on workloads with fine-grained tasks in Chapter 7 and on coarse-grained tasks in Chapter 8.

CHAPTER 7

Understanding Fine-Grain Task-Parallel Workloads Through Accelerating Graph Processing

In this chapter, we explore the applicability of the TaskStream framework on task parallel workloads with fine-grained data dependencies[1]. We use graph processing as the representative domain because of its importance and complexity. Because SPU is limited to irregular data parallelism, we had to implement graph processing kernels using the synchronous algorithm. While synchronous algorithm requires lower hardware support, this leaves performance potential from flexibility on the table and complicates understanding the relationship between graph types, workloads, algorithms, and specialization. Our goal is to accelerate dynamic task parallel graph algorithms (i.e., asynchronous) and do so in a manner that we can study the value of flexibility in graph processing accelerators.

We identify a taxonomy of key algorithm variants for graph processing that dramatically affect performance (determined from the tradeoff between throughput and work-efficiency[2]). Below we list these variants along with their specific requirements from the hardware:

1. **Update Visibility:** It is granularity when graph updates become visible.
2. **Vertex Scheduling:** The scheduling policy for vertex updates – should be high throughput to hide the short latency vertex tasks.

[1]TaskStream was originally called TaskFlow, but we unified it with the TaskStream representation for presentation in this dissertation.

[2]Work-efficiency is the work required by the optimized sequential execution (in terms of edges processed) over the work performed in parallel execution.

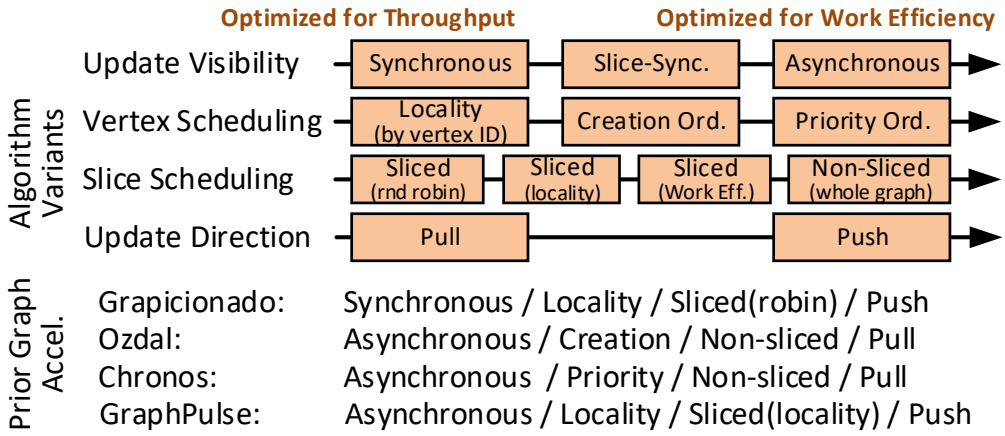


Figure 7.1: Algorithm Variant Dimensions & Prior Accelerators

3. **Slice Scheduling:** It determines whether the graph is sliced and the scheduling order of graph slices.

4. **Update Direction:** It determines whether vertices update their or neighbors' properties (pull/push). Push variants require support for efficient atomic updates.

Each variant has profoundly different implications on hardware, which can vary by graph and workload type.

Most prior graph-accelerators have each focused on one *algorithm variant* (see Figure 7.1). Because each algorithm variant causes different tradeoffs in work efficiency, locality/memory efficiency, and load balance, different accelerators perform well for different workloads and input graphs. Figure 7.2 shows the throughput (in giga-traversed edges per second - GTEPS) versus work efficiency measured by our model of these accelerators. Performance is a product of work efficiency and throughput, so we also show equi-performance curves in this figure. Our insight is that we can optimize performance by having the flexibility to use the correct algorithm variant for the right graph and workload (or even workload phase).

Selecting the correct variant is simple; the challenge is to design an architecture with sufficient algorithm/architecture flexibility and little performance, area, and power overhead. This flexibility requires supporting different granularity tasks (synchronous vs. asynchronous

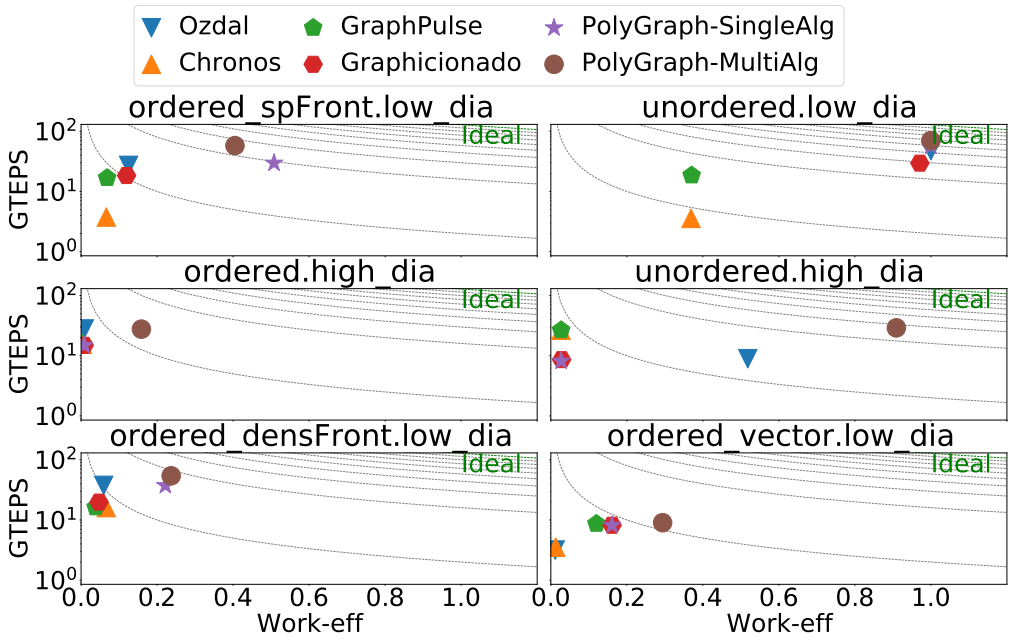


Figure 7.2: Work-efficiency and Throughput Tradeoffs

updates), fine-grain task scheduling, flexibly controlling the working set, and flexibility for different data structures.

TaskStream framework already supports fully-pipelined execution of tasks and conditional creation of new tasks at runtime. Supporting algorithm variants involved more challenges. First, integrating asynchronous variants with sliced execution required new mechanisms for deciding when to switch slices and how to orchestrate data during slice transition. Next, because tasks can be short-lived due to pipelined dataflow execution, we needed to develop a high-throughput task scheduler. Also, because we relied on a mesh interconnect (for scalability and high local bandwidth), a new multi-level spatial partitioning scheme was critical to ensure locality and a balanced load. Subsequently, we enhanced SPU to design a template architecture, PolyGraph, that is flexible across these variants (using task and scheduling primitives) while modularly integrating specialization features for each.

Evaluation and Results We evaluated PolyGraph with cycle-level simulation, supporting the design space of architectures with features encompassing many prior works [11, 173, 262].

We evaluated traditional and ML-based graph workloads (Table 7.1) on graphs with up to 1.5 billion edges. The best algorithm-specific PolyGraph design (PG-singleAlg) is 16.79× (up to 275× for high diameter graphs) faster than a Titan V GPU. More importantly, we find that flexibility is critical. By statically choosing the best algorithm variant, we gain 2.71× speedup. Dynamic flexibility provides 1.09× further speedup.

In this chapter, we first give background on key workload/graph properties (Section 7.1). We then discuss a taxonomy of algorithm variants (Section 7.2). We develop the TaskStream optimizations for these variants (Section 7.3) and describe the support within PolyGraph (Section 7.4), as well as our novel spatial partitioning (Section 7.5). Finally, we evaluate and discuss related work (Section 8.4,8.5,8.6). We will conclude with the discussion of PolyGraph's scalability and pointers to future work (Section 7.9).

7.1 Graph Acceleration Background

Here we describe our computational paradigm, mapping to a template architecture, and key graph/workload properties.

7.1.1 Vertex-centric, Sliced Graph Execution Model

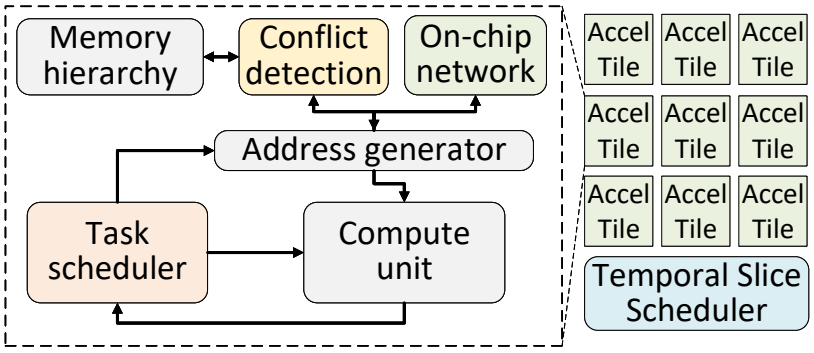
In vertex-centric graph execution [147, 146, 97, 5, 260, 243], a user-defined function is executed over vertices. This function accesses properties from adjacent vertices and/or edges, and execution continues until these properties have converged.

Preprocessing the graph can offer better spatial and/or temporal locality. Commonly, the graph is divided into *temporal slices* (or T-slices) which fit into on-chip memory. Further preprocessing may divide the graph among cores for load-balance or locality; these are *spatial slices* (S-slices).

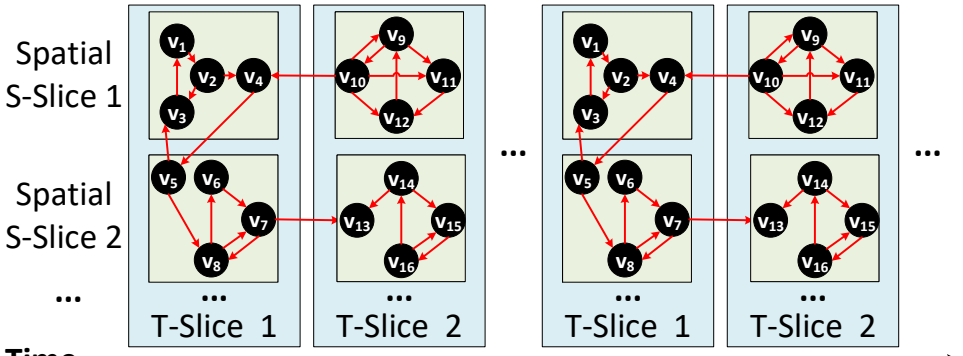
Example: Shortest Path (SSSP) Figure 7.3(a) shows an example code (SSSP) written in this model. An active list is maintained for each temporal and spatial slice. After initial-

```
1: procedure CALC_SHORTEST_PATH(GRAPH=(V,E,
   T:Temporal Slices (optional); S:Spatial Slices))
2: vert_prop[V_init] = 0 # initialize vertex prop
3: active_set[T][S] = {} # initialize active sets
4: add V_init to active_set
5: while any active slice left
6:   Choose temporal slice t
7:   for all spatial slices s in t: do in parallel
8:     while any active vertex active_set[t][s]
9:       Chose vertex v
10:      for all outgoing vertices w: do
11:          new_prop = vert_prop[v]+edge_wgt[v->w]
13:          if vert_prop[w]-new_prop[w] > 0 then
14:            vert_prop[w] = new_prop[w]
15:            add w to active_set
```

(a) SSSP Algorithm (temporally sliced algorithm variant)



(b) Accelerator Tile Template



(c) Logical execution of slices over time

Figure 7.3: Algorithm (SSSP) and Mapping to Arch. Template

	Priority	Vertex comp.	Vertex update	Prop(B)
SSSP [224]	dist	src_dist+edge_wgt	min(dist,dst_dist)	4B
BFS [224]	depth	src_depth+1	min(depth,dst_depth)	4B
CC [224]	comp. ID	-	min(src_id,dst_id)	4B
PR [254]	res/deg	src_res*α/src_deg	new_res+dst_res	2x4B
CF [280]	grad	f(src_prop.dst_prop)	new_vec+dst_vec	32x4B
GCN-Inf [120]	-	matrix-mult	src_vec+dst_vec	128x4B

Table 7.1: Graph Workloads (Prop: vertex prop. size).

ization, the algorithm iterates over all active temporal slices until no vertex is active. Within each temporal slice, the corresponding spatial slices execute concurrently (see Figure 7.3(c)). Within each spatial slice, the vertices are scheduled iteratively. Each vertex execution updates its neighbor's vertex properties. If the destination's vertex property is changed, it is activated[3].

Different graph workloads can be implemented by changing: 1. initial active vertices, 2. the function computed for each vertex, 3. the update function for the destination vertex. An optional characteristic is a priority hint for vertex scheduling (see Figure 7.3(c), Line 11). Examples are shown in Table 7.1.

Figure 7.3(b) shows a high-level decoupled-spatial template architecture, colored to indicate the relationship between algorithm and accelerator. The temporal-slice scheduler chooses a T-slice in each coarse graph phase. The spatial slices (S-slices) are executed in parallel across all cores. A task scheduler picks a vertex to execute at each step, and the per-vertex computation (lines 10-15) is mapped to the Atomic updates to vertex properties (lines 13-14) are enforced using conflict detection and stalling.

Graph Data-structures Figure 7.4 overviews the essential data structures. A vertex-list stores the first edge index for each vertex. Edges are stored contiguously in the edge list, containing a destination vertex_id (and optional edge weight). Each vertex has a property

[3]This describes the *push*-based update direction. For *pull*, the incoming edges are used to update the vertex's own property.

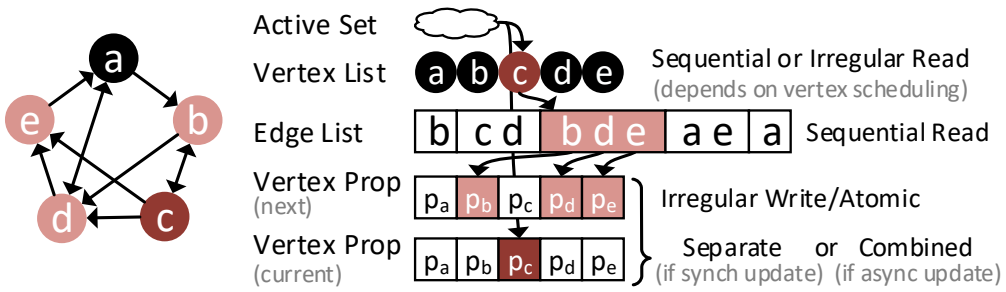


Figure 7.4: Graph Data Structures

which the algorithm computes. The active list data structure, and whether we double buffer the vertex properties, depends on the algorithm variant (Section 7.2).

7.1.2 Key Workload/Graph Properties

Graph Property: Diameter is the largest distance between two vertices. *Uniform-degree graphs* (eg. roads) have a similar/low number of edges-per-vertex, and thus a high diameter. Oppositely, *power-law graphs* (eg. social networks) have low diameter, as some vertices are highly connected.

Workload Property: Order Sensitivity Many graph workloads are *iterative* and *converging*. This converging nature tends to make such workloads functionally resilient to the order in which computation occurs. However, some of these workloads may require a different amount of work depending on the order tasks are performed; ie. their work-efficiency is *order-sensitive*. For example, in a shortest path algorithm, exploring a farther away vertex before a near vertex can lead to redundant work, because the distance of the farther vertices may be updated if a shorter path is found. Sensitivity varies with workload. Breadth-first search (BFS), is less sensitive as many paths are of equal depth, making it less likely to find a wrong path. Non-converging workload like GCN is order insensitive.

Workload Property: Frontier Density Dense frontier workloads like PR, CF usually have more than 50% active vertices while sparse frontier workloads (eg. SSSP, BFS) require

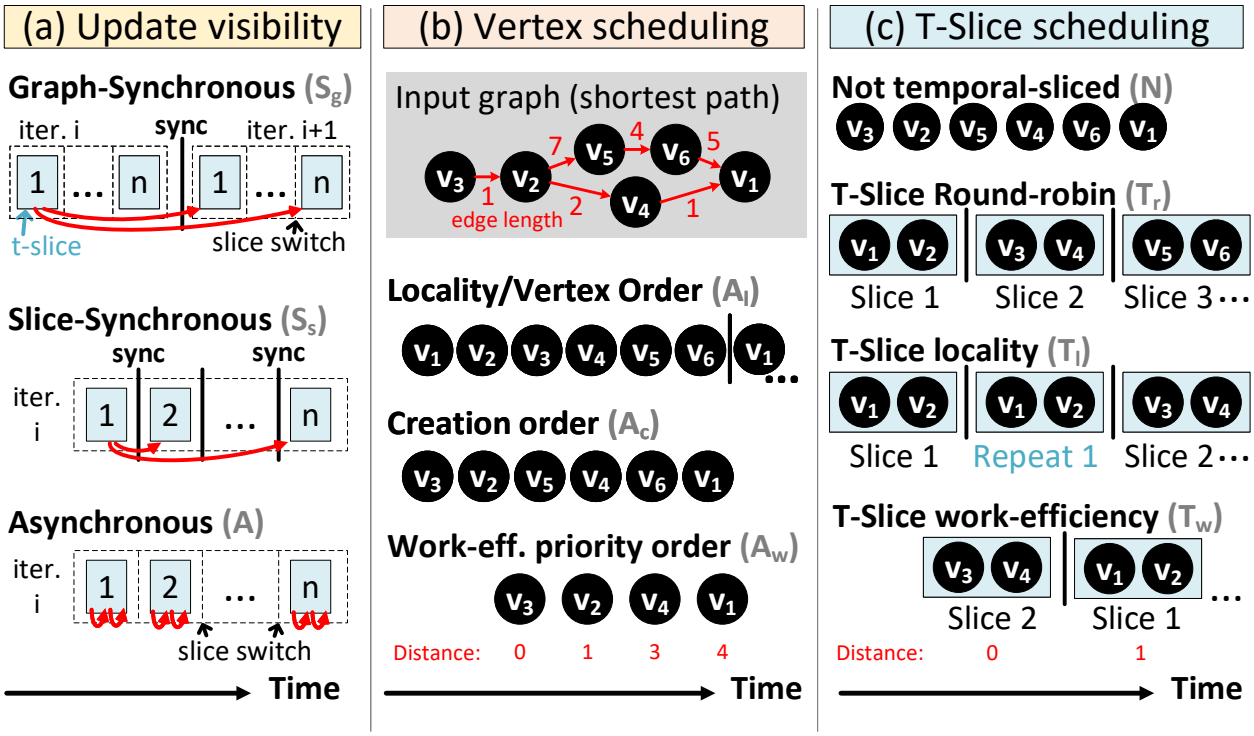


Figure 7.5: Key Variants of Graph Processing Algorithms

much fewer. In general, sparse frontier workloads require fewer passes through the graph until convergence.

7.2 Graph Algorithm Taxonomy

To systematically study the value of flexibility, we create a taxonomy of four key dimensions and discuss tradeoffs.

Update Visibility defines when writes become visible to other computations, and hence this affects the granularity at which new tasks are created. Writes may become visible after one pass through the graph (*graph-synchronous*), after each slice (*slice-synchronous*)[4], or immediately (*asynchronous*). Barriers are used to synchronize update propagation in

[4]Graph/slice synchronous are bulk-synchronous [224] at different granularity.

synchronous variants. Figure 7.5(a) visualizes dependences in a slice-based execution of graph processing (blue boxes are graph slices which fit in on-chip memory). The figure shows how the dependence distance shrinks (red arrows) moving from synchronous to asynchronous.

Tradeoffs: While synchronous algorithms naturally provide sequential consistency of vertex updates, fast implementations of asynchronous variants do not lock neighboring vertices, and hence do not provide sequential consistency of tasks. While many converging workloads do not require this, some workloads may not be expressible or converge slower (eg. ALS [146]). Also, barriers in synchronous variants can be a high overhead when the work between them is low (eg. due to few active vertices).

Vertex Scheduling defines the processing order for active vertices, relevant for asynchronous variants. Figure 7.5(b) depicts the variants for shortest path: *Locality order:* To improve the vertex access locality, schedule by vertex-id. *Creation order:* Schedule vertices in the order they are activated (breadth-first in the figure). *Work-efficiency order:* Schedule vertices in an order which reduces redundant work (by distance in the figure). Section 7.1.2 explains the intuition. Table 7.1 lists the priority metric for each workload.

Tradeoffs: When active vertices are accessed in their storage order, spatial locality enables high memory bandwidth; However, this costs work-efficiency, as it requires critical updates to be delayed. Creation order requires simple FIFO logic, while Work-efficiency order requires dynamic sorting. Note that distributed ordering is sufficient [254].

Temporal Slicing defines whether the working set may be limited to a predefined *slice* of all graph vertices. Slices are determined during offline partitioning and are generally sized to fit on-chip memory. Updates to data outside the current slice are deferred, and an explicit phase is required to switch slices (combined with barriers in synchronous variant). Slices can be scheduled in different orders, forming new variants. Figure 7.5(c) depicts each: *Round-robin:* iterate through all slices. *Locality:* similar, but repeatedly process each slice. *Work-efficiency:* prioritize slices whose properties change most [264]. In the example, slice

		Throughput benefit			Work-eff. benefit		Inputs optimized for	
		No Sync	Mem b/w	Latency	Prio.	Order Sens.	Graph	Active Tasks
Update-Vis.	S_g		✓				Low dia	More active
	S_s		✓	✓		✓	Low dia	More active
	A_c	✓		✓		✓	High dia	Less active
	A_l	✓	✓	✓			High dia	Less active
	A_w	✓		✓	✓	✓	High dia	Less active
T-Slicing	N	✓		✓		✓	High dia	Less active
	T_r		✓				Low dia	More active
	T_l		✓				Low dia	More active
	T_w		✓		✓	✓	Low dia	More active

Table 7.2: Algorithm Variant Tradeoffs

1 is chosen second, as its properties changed most (v1 and v2).

Tradeoffs: Non-sliced avoids barriers and slice-switching data movement, which is costly if there are few active vertices. Slicing can also harm the optimal ordering by restricting the scheduling scope. The key benefit of temporal-slicing is more effective on-chip memory use. Locality scheduling improves intra-slice reuse but may delay cross-slice updates. Sliced-work-efficiency ordering optimizes for work-efficiency without requiring hardware support for fine-grained scheduling.

Update Direction defines whether a task updates its own property (pull/remote read), or whether a task updates its neighbor's properties (push/remote atomic update).

Tradeoffs: Push reduces communication bandwidth by using one-way communications (push updates to neighbors) and efficient multicast [27], rather than the remote memory requests in pull. The request latency is hidden due to the access reordering potential of push. Finally, pull often requires more work while reading all incoming edges of each active vertex. However, there are techniques that optimize pull by eliminating edge accesses based on vertex convergence [25, 284]: their effectiveness depends on prefetching capability.

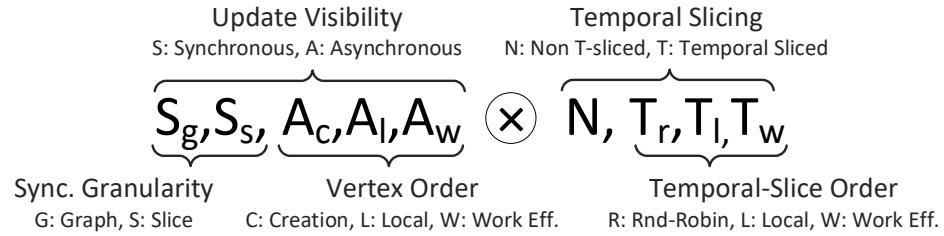


Figure 7.6: Shorthand for Algorithm Variants

Notation We use a two-letter shorthand (sometimes expanded) to denote each variant combination (Figure 7.6). By default we assume push; we notate pull with an explicit suffix.

Summary Table 7.2 summarizes the throughput (lacks synchronization or improves locality) or work-efficiency benefit (reduces update latency or has priority ordering); it also shows which graph type and #active-vertices it is best suited for.

7.3 Unified Graph Processing Representation

To support variants efficiently on a unified hardware, we need a program representation that is flexible, fast, and is specialized for graph workloads. Because of their different needs, we develop separate approaches for the data-plane (pipelined task execution) and the control plane (slice scheduling).

7.3.1 Data plane Representation: TaskStream

There are three major requirements from the execution model: 1. Need for fully pipelined execution of per-vertex computation. 2. Need to support data-dependent creation of new tasks, including programmatically specifying and updating the priority ordering. 3. Need for streaming/memory reuse. Fortunately, TaskStream already provides a fully pipelined execution of tasks. The dynamic task creation can be implemented by hierarchically integrating TaskStream with the dataflow model and encoding task creation in the dataflow

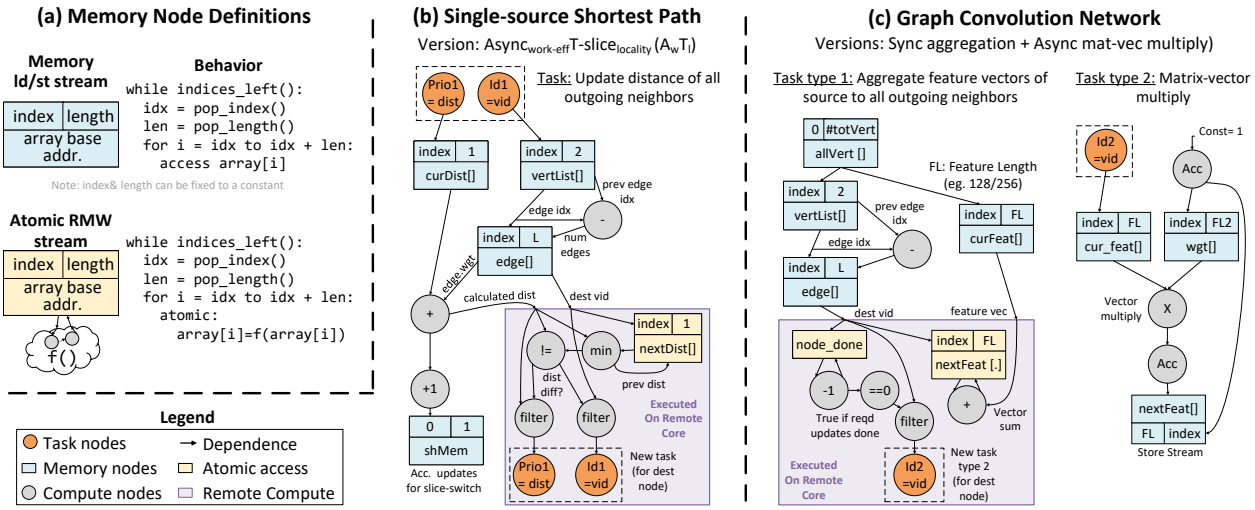


Figure 7.7: TaskStream Examples

graphs for task types. Finally, we augment node annotations to specify hints for priority ordering and near-data scheduling. Since TaskStream nodes are exposed in the ISA, we can add specialized hardware for priority ordering.

TaskStream Implementation In our implementation of TaskStream, each task type is defined by a graph of compute, memory, and task nodes which extend the previous definition of dataflow graphs (Section 3.1):

- **Compute nodes:** are passive and may maintain a single state item. This enables them to be mapped to systolic-like fabrics [60, 188, 89, 66, 162] for high efficiency.
- **Memory nodes:** represent decoupled patterns of memory access, called streams [119, 56, 162, 66, 247, 249]. Stream parameters can either be constant (set at stream creation) or dynamic (consumed from another node with a FIFO interface). Figure 7.7(a) defines stream parameters and behaviors.
- **Task nodes:** represent arguments and are ingress and egress points of the graph. A task is started by delivering arguments to the dependent compute and memory nodes, and a new task is created when values arrive at the complete set of egress task nodes.

An instance of a task type (i.e., task) is started by providing a value to each ingress task node. And a task is created when values arrive at all egress task nodes.

Atomics Shared-memory atomics are critical due to the need for correct handling of memory conflicts on vertex updates. In taskstream, a memory stream can be marked as "read-modify-write" (RMW) and will be atomic. See example in Figure 7.7(a).

Priority Scheduling and Coalescing One task argument may be designated at compile time as the task's priority. The parent task computes the priority prior to task creation, and this serves as a hint to schedule tasks with higher priority first.

Another task argument indicates the ID, which is unique for all active tasks. The ID generally is the vertex_id, and may be used for task coalescing and sliced execution. When two tasks with the same ID are created, this is treated as an update to the priority of the original task, and the two tasks are "coalesced" into one higher priority task. The upper bits of the ID indicate the task's T-slice and S-slice. Tasks are deferred until their T-slice is active and are scheduled at their S-slice core.

Graph Workloads Written in TaskStream Figure 7.7(b) shows an example TaskStream graph for SSSP, implemented as $\text{Async}_{work-eff}$. This workload has two tasks, each associated with different vertex data: Task type 1 iterates over outgoing edges of a source vertex to compute distances, and creates a type 2 task for each destination vertex to carry out distance updates. The ID of task 1 and 2 is the vertex-ID of source and destination vertices respectively, and they execute on the core corresponding to the ID. Type 1 tasks are prioritized by vertex distance for work-efficiency. Type 2 tasks also check if the vertex should become active, and if so, a new type 1 task will be created. The number of task 1 invocations is accumulated in a shared memory location to identify slice-switching (see Section 7.3.2).

In GCN, graph aggregation is synchronous, while matrix-vector multiply is asynchronous – this enables overlapping the memory-intense aggregation tasks with compute-intensive

matrix multiply. Here, task 1 is *coarse-grained*, and iterates over all vertices (in a slice), and a single task node trigger creates many cycles of work. It pushes source vector features to task 2 to aggregate them together for incoming edges of each vertex. Aggregation (task 2) asynchronously triggers a matrix-vector multiply (task 3) when aggregation is complete (identified using node_done). Note that this overlap splits matrix-matrix into multiple matrix-vector computations: this prevents the broadcast of weight matrix. To localize the traffic for weight responses, we duplicate weight matrix at each core (this is low overhead: only consumes 3% of scratch space).

TaskStream Flexibility Summary Synchronous variants (S_{graph},S_{slice}) use coarse grain tasks that pass through the (per graph/per slice) active list. Asynchrony is supported with explicit fine-grain tasks, optionally with priority hint argument.

7.3.2 Slice Scheduling Interface and Operation

The responsibility of the temporal slice scheduler is to configure on-chip memory, decide which slice to execute next, and manage data/task orchestration. The slice scheduler is invoked infrequently, and can be executed on a simple control core with limited extensions for data pinning operations. The slice scheduler also has a mechanism for creating initial tasks.

Data Pinning Depending on the algorithm variant, we may know which data we want to most reuse. For eg. for synchronous non-sliced (S_gN), edges have large reuse distance, but vertices with high-degree are reused many times. In sliced-locality variants, edges are also reused, to a lesser extent. To help the slice-scheduler optimize for reuse, we provide the slice scheduler an interface to *pin* a range of data to the on-chip memory at a particular offset, essentially reserving a portion of the cache (eg. pin the region of vertex properties that have high reuse). Non-pinned data is treated like normal cache access.

Slice Switching for Asynchronous Variants The decision of when to switch slices for asynchronous variants is a tradeoff between work-efficiency (switch sooner) and reuse (switch later). To explain, information at the slice's boundary becomes "stale" over time, as it may depend on an inactive slice's execution, thus hurting work-efficiency. It is impossible to calculate "staleness", as it depends on the future execution. Therefore, we approximate it by counting the number of vertex updates, and switch slices when these exceed a threshold. When switching slices, the slice scheduler gives all cores a highest priority stop task, which disallows any new tasks to be issued and it may also perform the slice transition as explained below.

Slice Preprocessing Slices are preprocessed to keep all edges (and hence updates) within each slice. This is accomplished by creating a mirror vertex for any cross-slice edge in the destination slice, and removing the original cross-slice edge from the source slice. For example, if edge A→B crosses a slice, then a mirror vertex for A would be created in B's slice. Mirror A retains the edge A→B, while this is removed in the original vertex A (in A's slice). The mirror vertices properties are only updated during slice transition, as explained next.

Slice Transition Figure 7.8 shows how data is orchestrated during slice transition. Two slices are shown, and only one may reside in on-chip memory at a time. Main memory contains the graph vertex properties, pending tasks for each slice, and a copy of each mirror vertex. Transition works as follows:

Step 1. Stream in cross-slice vertex properties (eg. vertex 10, K), and scatter to their S-slice core; compare old and new properties to see which vertices changed. If such a property changed, a new task is created in the destination slice by pushing the task's arguments to that slice's pending tasks list (10, K to slice 2).

Step 2. Meanwhile, the updated cross-slice vertex properties are stored in the copy of mirror vertex properties. Also, the current slice's pending tasks and updated vertex properties are

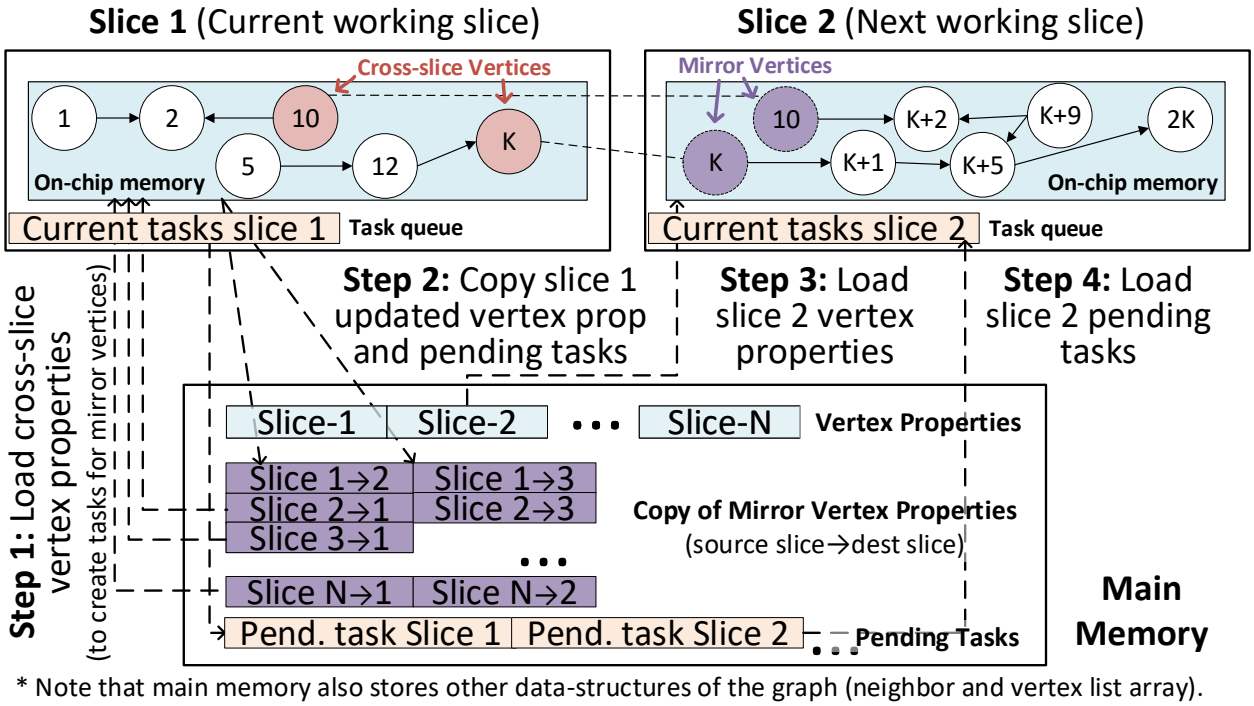


Figure 7.8: T-Slicing for Large Graphs (N slices, K vertices each)

streamed to memory.

Step 3. In parallel with step 2, using double buffering, vertex properties in the next slice are streamed to pinned memory.

Step 4. Stream next slice's pending tasks from main memory.

7.3.3 Scheduling of Algorithm Variants

Quantitative Motivation Figure 7.9 shows the fine-grained effective throughput (normalized to work-efficiency) over time for several algorithm variants. Time on the x-axis is normalized to the percentage of total execution time, as time-scales vary significantly between variants. Notice that the highest performance variant changes during the execution:

Low Diameter Graph (lj): For order-sensitive SSSP (`SP` in figures), $\text{Async}_{\text{work-eff}}$ dominates due to work-efficiency gains, while for less order-sensitive BFS, both $\text{Async}_{\text{work-eff}}$ and $\text{Sync}_{\text{graph}}$ are similar. The sliced versions improve on-chip hit rate, however switching to

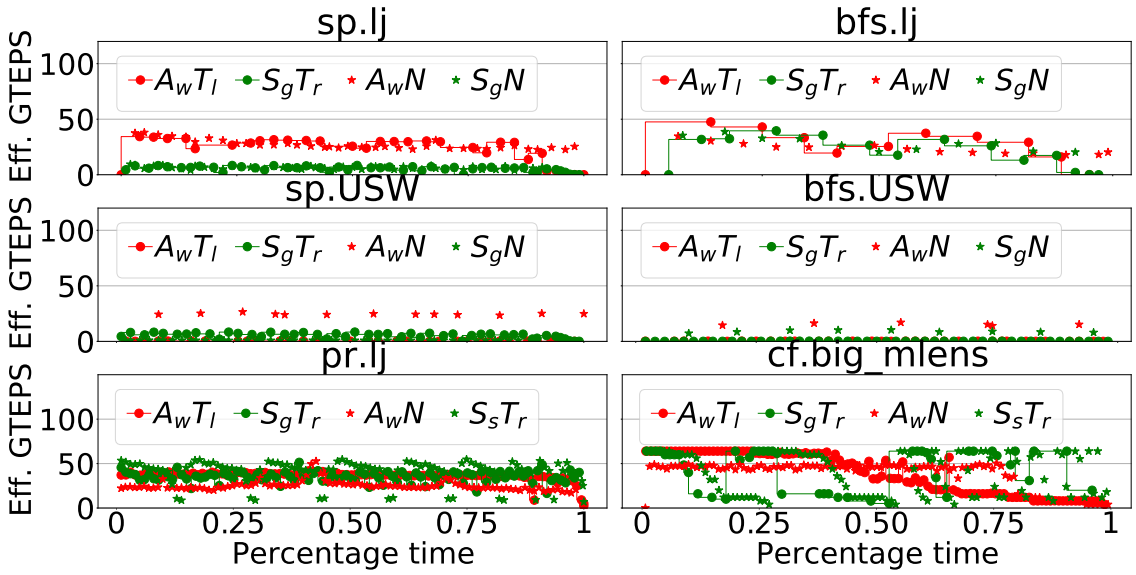


Figure 7.9: Potential of Dynamically Switching Variants (Effective GTEPS is the useful throughput – "work-done-per-second"/"work-efficiency". Here the work-efficiency is normalized to A_wN and thus, the area under the curve is "A_wN-work"/"total-execution time".)

non-sliced at beginning/ending iterations has significant potential.

High Diameter Graph (USW): Since high diameter graphs have only a few vertices active in each iteration, synchronization overhead is critical. Therefore, $\text{Async}_{\text{work-eff}}$No-slice consistently dominates.

Dense frontier workloads (pr,cf): For PageRank, there is a tension between memory efficiency due to high active vertices and work-efficiency due to order sensitivity. Therefore, $\text{Slice}_{\text{sync}}$ balances tradeoffs and is optimal. For CF, $\text{Async}_{\text{work-eff}}\text{T}_{\text{locality}}$ is optimal in initial iterations, however $\text{Async}_{\text{work-eff}}$No-slice dominates in the later iterations when #active vertices are low.

We found that switching between synchronous/asynchronous hurts work-efficiency, as they proceed differently: asynchronous leaves many vertices active because it focuses on high priority vertices, while synchronous conservatively tries to complete the work for all active vertices in every iteration.

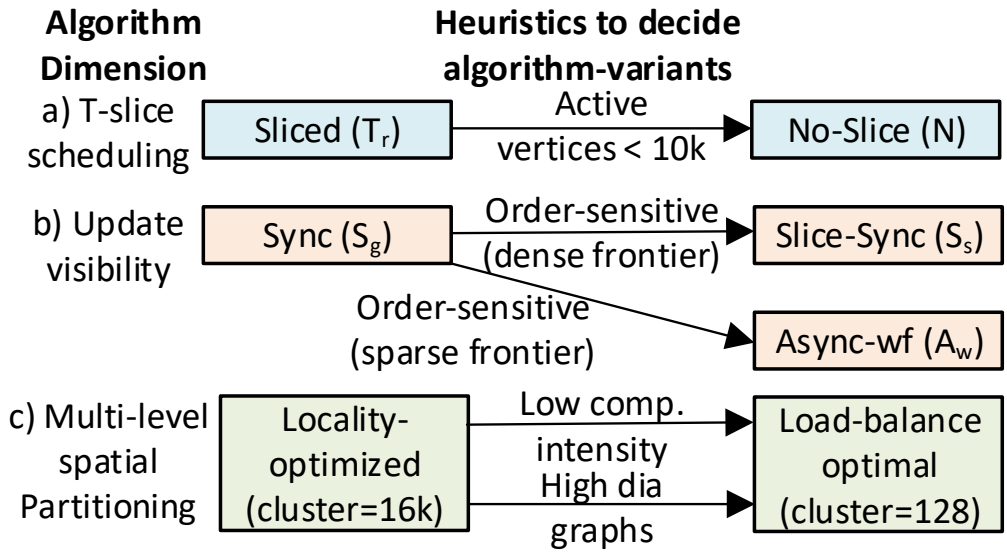


Figure 7.10: Algorithm Variant Scheduling

Heuristics for Algorithm Variant Scheduling Figure 7.10 shows how we decide the algorithm variant. For slicing, the effective throughput depends on whether the work during phase is sufficient to hide barrier overhead. This work can be approximated from active vertices – for example, non-sliced outperforms at the beginning and end of the algorithm, as active vertices are fewer (Figure 7.10, `lj`). For uniform graph (`rdUSE`), non-sliced consistently outperforms as active vertices are low due to their high diameter. Therefore, our algorithm switches to non-sliced when number of active vertices are below a threshold. In the update visibility dimension, the decision depends on the workload characteristics. Asynchronous versions are preferred for order-sensitive workloads (See Figure 7.10, `sp`). Dense frontier algorithms (those with usually >50% active vertices) have high inherent spatial locality, so $\text{Slice}_{\text{sync}}$ is preferred, as it is memory efficient while maintaining moderate work-efficiency. We also propose a flexible multi-level spatial partitioning that can optimize for load and locality depending on the workload and graph type (details explained in Section 7.5). Note that the scheduling decisions may change depending on hardware parameters, as we discuss in Section 8.5E.

Variant Transition Variant selection is performed after a single round of graph/slice in S_g/S_s, or after every 100k cycles for asynchronous variants (long enough to amortize the latency of switching). To transition, given the algorithm variant, the control core will: 1. initialize data-structures and configure TaskStream graph, and 2. perform pinning operations. If a dynamic switch is invoked, on-chip memories are flushed, and TaskStream may require reconfiguration. The pending tasks are managed as during slice transition data orchestration.

7.4 Polygraph Hardware Implementation

PolyGraph is a multicore decoupled-spatial accelerator connected by 2D triple mesh networks[5], overviewed in Figure 8.6.

The **data plane** is comprised of all modules besides the control core, and is responsible for executing taskstream graphs. The decoupled execution of memory/compute is similar to prior decoupled-spatial accelerators: memory nodes are maintained on stream address generators, and accesses are decoupled to hide memory latency. A Softbrain-like CGRA [162] executes compute nodes in pipelined fashion. Between the stream controller and CGRA are several "ports" or FIFOs, providing latency insensitive communication. The novel aspect is task management: A priority-ordered task queue holds waiting tasks. Task nodes define how incoming task arguments from the queue are consumed by the stream controller to perform memory requests for new tasks.

The basic operation is as follows: If the stream controller can accept a new task, the task queue will issue the highest-priority task. The stream controller will issue memory requests from memory nodes of any active task. The CGRA will pipeline the computation of any compute nodes. The CGRA can also create new tasks by forwarding data to output ports designated for task creation, and these are consumed by the task management hardware. Tasks may be triggered remotely to perform processing near data. Initial tasks may be

[5]Multiple networks enable efficient scalar remote accesses.

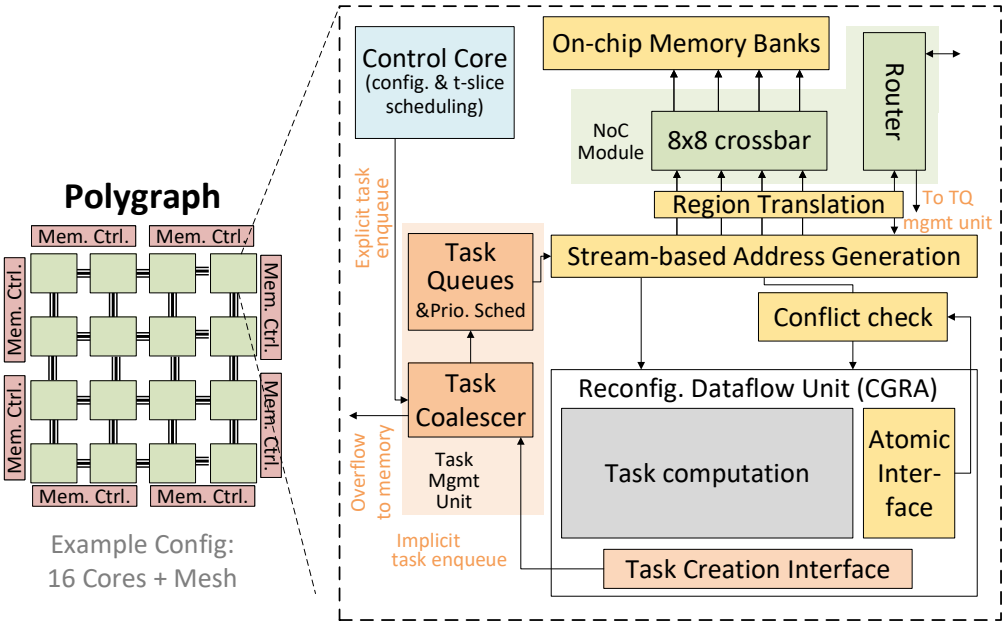


Figure 7.11: PolyGraph Modular Hardware Implementation

created by the control core, by explicitly pushing task arguments to the task creation ports.

The **task management unit** enables high-throughput priority-ordered task dispatch. This is critical, as many tasks are short-lived. Therefore, the requests of multiple tasks should be pipelined. The task unit also coalesces superfluous tasks at high throughput (for priority update), by maintaining a bitvector of in-flight tasks IDs. Tasks can overflow the task queue, which is handled by an overflow protocol.

Finally, **slice scheduling** is implemented on core 0's control core (a simple inorder core). The PolyGraph cores communicate with the slice scheduler through shared memory atomics to coordinate at phase completion. The slice scheduler orchestrates data when switching slices and may initiate a stop task on remote cores to prevent any new tasks to be issued.

7.4.1 Task Hardware Details

Task Queue and Priority Scheduling A task argument buffer maintains the arguments of each task instance before their execution. Statically, it is split into the number of task

types and each partition is configured to the size of its corresponding task type arguments. The task argument pointers to ready tasks (ie. whose all arguments are available) are stored in the task scheduler. Note that we use the priority task scheduler (described next) only for graph access tasks and FIFO scheduling for others (eg. vertex update).

Our task scheduler uses a pipelined hardware-based heap [30], where each memory bank represents a level in the priority heap. Push/pop operations move nodes across levels, locking in a hand-over-hand fashion. This delivers a throughput of one enqueue-dequeue operation every two cycles. For low degree graphs, this throughput is insufficient. Therefore, we use multiple priority-heaps per core, and alternate between them. This implies imperfect ordering, because two consecutive highest-priority elements could be in the same queue. In practice this does not significantly hurt work efficiency.

Overflow and Reserved Entries If the task queue is full, new tasks will overflow into a buffer in main memory (32kB is sufficient). This buffer is drained to the queue as entries are freed, and the priority then is re-calculated by using the updated *vertex_prop*. Re-calculation is required as the priority might have been updated due to coalescing.

Overflow unfortunately disrupts the priority order, and can hurt work-efficiency due to delaying a high priority task. To mitigate, we reserve some task queue entries for latent high-priority tasks (eg. 32 for 256 sized queue). During overflow, a new task is allocated a reserved entry if it is equal/higher priority than the current highest-priority task.

Task Coalescing To reduce active tasks, we allow coalescing of tasks with the same ID. We implement this with an SRAM bitvector, where each bit corresponds to a task ID. A set bit means the corresponding task is present in the overflow buffer, and this will prevent task insertion for that ID.

For graph processing, task ID corresponds to vertex_id. We size the bitvector to be 32 kB, as this covers the maximum vertices in temporally sliced variants. For non-sliced, PolyGraph only coalesces the first 32K vertex_id. This is sufficient in practice, as the partitioner can

simply move critical high-degree vertices to the beginning of the vertex list.

7.4.2 Memory Architecture

Shared Memory Our on-chip memory is a shared address-partitioned cache, with multiple banks per PolyGraph core. Each cache bank has a region translation unit that maintains the mapping of virtual address ranges to the pinned addresses.

Data-structure pinning is supported similar to prior buffer-in-NUCA techniques [59]. When the slice-scheduler in core 0 pins a memory region, the region translation unit in all cores are sent the base/bound of the region, along with an offset. This causes some of the sets of the cache to be set aside for pinned data. The core will generate requests for that region and send an acknowledgment when complete. Subsequent memory requests check the range registers to determine if they are to be mapped to the cache or pinned region. Pinning a new data-structure flushes all cache regions in the newly pinned addresses. Re-pinning is only required during transition between variants: this happens at most twice (when active vertices go above or below a threshold), thus the overhead is low compared to the 10s/100s of slice switchings.

Atomic Updates When update requests are received from the local core or the network, they are pushed to the pending atomic requests queue at its corresponding bank. The conflict check logic uses a small CAM (8 entries, to cover atomic latency) to detect and delay aliasing requests.

7.5 Spatial Partitioning

While offline partitioning is common for creating temporal-slices, we find that spatial partitioning makes the mesh-based network highly effective, as we describe next.

Spatial partitioning introduces a tradeoff between locality and load balance. Naively clustering connected vertices will reduce network traffic, but may hurt load balance, espe-

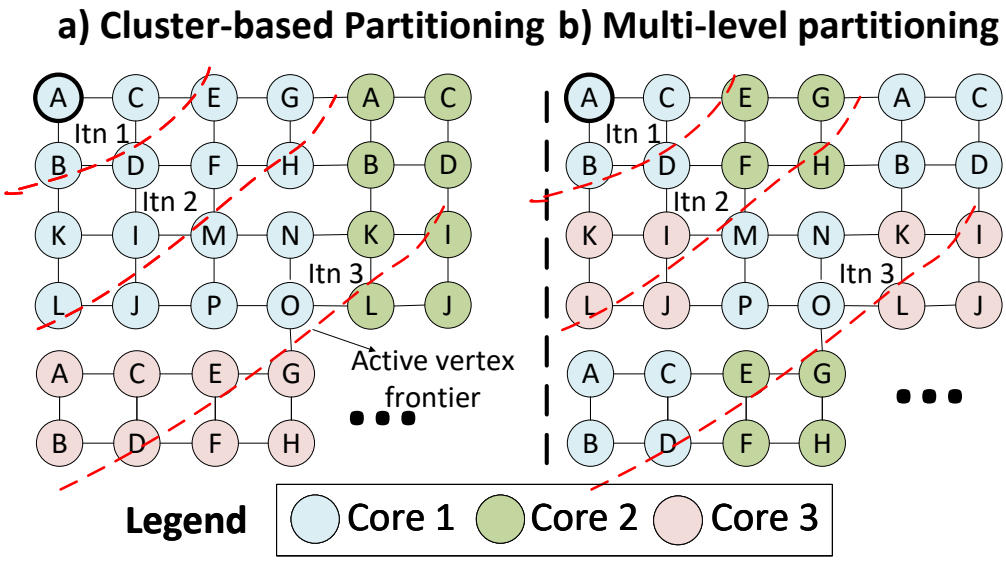


Figure 7.12: Cluster-based vs Novel Multi-level Spatial Partition

cially for sparse frontier workloads. We explain with a simple grid-graph[6] in Figure 7.12, but observations apply more broadly. In Figure 7.12(a), we use clustering-only for creating S-slices; the dotted red lines show the progression of a sparse frontier workload like BFS. What we can observe is that the frontier in several iterations is limited to a single slice (allocated in one core); hence, this strategy has extremely poor load balance.

Multi-level Scheme Figure 7.12(b) visually shows our proposed "multi-level" slicing scheme that respects both load balance and locality. First, the graph is split into many small clusters of fixed size to preserve locality, then these clusters are distributed equally among cores for balanced load. To implement, we use a simple bounded-depth first search (with depth=8) to find small clusters (of a parameterizable size), then distribute these round-robin to different S-slices. It requires O(V) time.

Note the effect is proportional to the number of active vertices, hence high diameter graphs prefer load balanced multi-level as active vertices is usually low across iterations. For low diameter, larger clusters are helpful for locality.

[6]Grid-graphs are somewhat representative of common road networks.

	GPU [61]	**Graphcnd.** S_gT_r[94]	**GraphPls** A_wT_l[193]	**Ozdal** A_cN[173]	**Chronos** A_wN[11]	**PG** (ours)
Compute	GP104	SIMT-16	ASIC	D-flow	ASIC	CGRA
$+Spad	14.5MB	32MB	32MB	32MB	32MB	32MB
FP Unit	5120	1024	1024	1024	1024	1024
Mem GB/s	652	512	512	512	512	512
Net. type	Bus	XBAR	XBAR	XBAR	3 mesh	
Net. radix	-	128	128		5 core/8 mem	

Table 7.3: Architecture Characteristics of Baselines

	Graphs	**Vertices**	**Edges**	**Dia**	**Structure**	**#T-Slices**
All	orkut	3M	106.3M	9	Power-law	1/2
	LiveJournal	4.8M	68.9M	16	Power-law	2/4
	twitter	41.6M	1.4B	9	Power-law	5/10
Search	indoChina	7.4M	194M	200	Random	2
	rdUSE	3.5M	8.7M	2897	Uniform	1
	rdUSW	6.3M	15.2M	10206	Uniform	2
cf	big-mlens	0.2M	2.5M	5	Power-law	2
	mlens	82k	10M	5	Power-law	1
gcn	pubmed	0.02M	0.09M	9	Power-law	2
	cora	2.7k	10k	20	Power-law	1

Table 7.4: Input Graphs (Left column is the domain. PR requires double #T-slices; #T-slices for CF/GCN depends on feature size.)

7.6 Methodology

PolyGraph Power/Area We prototyped PolyGraph by extending DSAGEN [251] with task scheduling hardware, and extended the stream-dataflow [162] ISA as described. We synthesized PolyGraph cores and NoC at 1GHz, with a 28nm UMC library. We used Cacti 7.0 [156] for modeling eDRAM.

Baseline Architectures For reference, we use a 24-core SKL CPU and GAP benchmarks [26]. For CF and GCN (unavailable in GAP), we used Graphmat [224] and Gunrock [246] respectively. We evaluated Gunrock [246] graph processing library on a Titan V GPU. Gunrock does not implement CC/CF, so we calculate GPU means without these workloads.

For performance modeling across variants, we developed a custom cycle-level modular simulator (see Appendix A for details). Main memory is modeled using DRAMSim2 [201]. Accelerator configurations are in Table 7.3, and have similar memory capacity, bandwidth, and max throughput. We assume preprocessing is done offline and reused across queries. Note that both temporal partitioning (chunk-based Gemini [281]) and spatial partitioning are O(V).

For fairness and consistency with our simulation framework, we make the following provisions for prior accelerators: For Graphicionado [94], we did not implement capacity optimizations like extended graph slicing and coarsened edge table. For Ozdal [173], we bypassed the sequential consistency module; it is not required on our workloads. For Chronos [11], we modeled priority-order speculative execution with their no-rollback optimization. Since SLOT requires a single read-write object per task, we implemented the pull variant; this allows vertex granularity tasks instead of fine-grained edge tasks in push. We used PolyGraph's NoC, as Chronos is FPGA-based. For GraphPulse [193], we model their task coalescing, but for consistency with our simulator, we used distributed scheduling.

For Polygraph (PG) we evaluate: 1. PG-singleAlg: fixed-variant configuration that

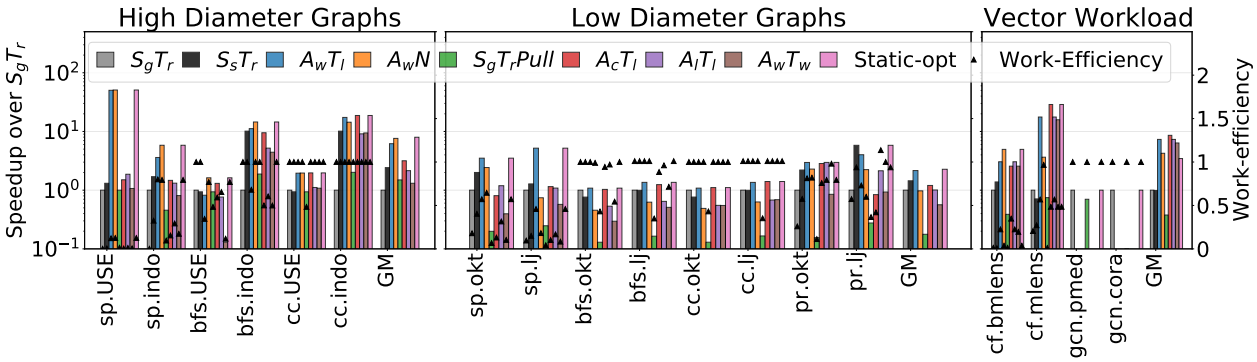


Figure 7.13: Algorithm Variant Performance Analysis

provides the best geomean speedup. 2. PG-multiAlg: Here flexibility is incrementally added for per-workload (static) and per-phase (dynamic).

Datasets Table 8.3 summarizes input graphs. For SSSP on unweighted graphs, random weights are assigned from [1:256). Due to prohibitive simulation times of exploration, we evaluate large graphs (twitter, rdUSW) only for overall performance.

Workloads Table 7.1 lists the evaluated workloads. For graph-based machine learning workloads, PR and CF, we optimize for convergence by choosing a different learning rate for each algorithm variant (higher for asynchronous variants). For GCN inference, we only implement the graph synchronous variant, as asynchronous benefits are minimal due to GCN's non-converging behavior.

7.7 Evaluation

Our objective is to evaluate how much and which kinds of flexibility are useful, across graph and workload types. First, we analyze algorithm variants (Section 7.7.1) and compare against prior accelerators (Section 7.7.2). Then, we discuss sensitivity to algorithm and hardware parameters (Section 7.7.3,7.7.4).

7.7.1 Algorithm Variants Performance Comparison

Figure 7.14 compares *strong algorithm variants* – those which perform well on at least one workload/graph type. Overall, we find that asynchronous-sliced, $\text{Async}_{\text{work}}\text{Sliced}_{\text{loc}}$, is the optimal variant (2.91× geomean speedup over typical $\text{Sync}_{\text{graph}}\text{Sliced}_{\text{rndrbn}}$), while static flexibility can further improve speedup by 3×. We explain the trends below, grouped by their choice of the best algorithm variant:

1. **High Diameter Graphs:** Here the synchronization overheads of synchronous/sliced variants (eg. $\text{Sync}_{\text{graph}}$, $\text{Sync}_{\text{slice}}$, $\text{Async}_{\text{work}}$Sliced) are the critical bottleneck. Therefore, $\text{Async}_{\text{work}}$Non-Sliced performs best/similar for all workloads.

2. **Low Diameter Graphs:** The power-law degree distribution makes random accesses more critical than synchronization. Sliced variants improve reuse, and thus perform better. For order-sensitive workloads (e.g. SSSP), faster updates in asynchronous variants lead to faster convergence. Among vertex-scheduling schemes, $\text{Async}_{\text{work}}$ performs best while $\text{Async}_{\text{creation}}$/$\text{Async}_{\text{locality}}$ provides only modest work-efficiency. Overall $\text{Async}_{\text{work}}\text{Sliced}_{\text{rndrbn}}$ is sufficient.

3. **Dense Frontier workloads:** For PageRank, which has a dense frontier, $\text{Sync}_{\text{slice}}\text{Sliced}_{\text{rndrbn}}$ provides speedups through memory efficiency while retaining some work-efficiency benefits of asynchronous updates within a graph slice.

4. **Vector Workloads:** With asynchrony, vector workload, CF sees high gains, however priority scheduling is not required. Non-sliced is similar to sliced as large vertex properties have high spatial locality that reduces cache miss overhead.

Less Competitive Variants: We generally find that with sufficient hardware for asynchronous priority scheduling, sliced-work-efficiency does not help as asynchronous variants require less iterations due to dynamic task creation.

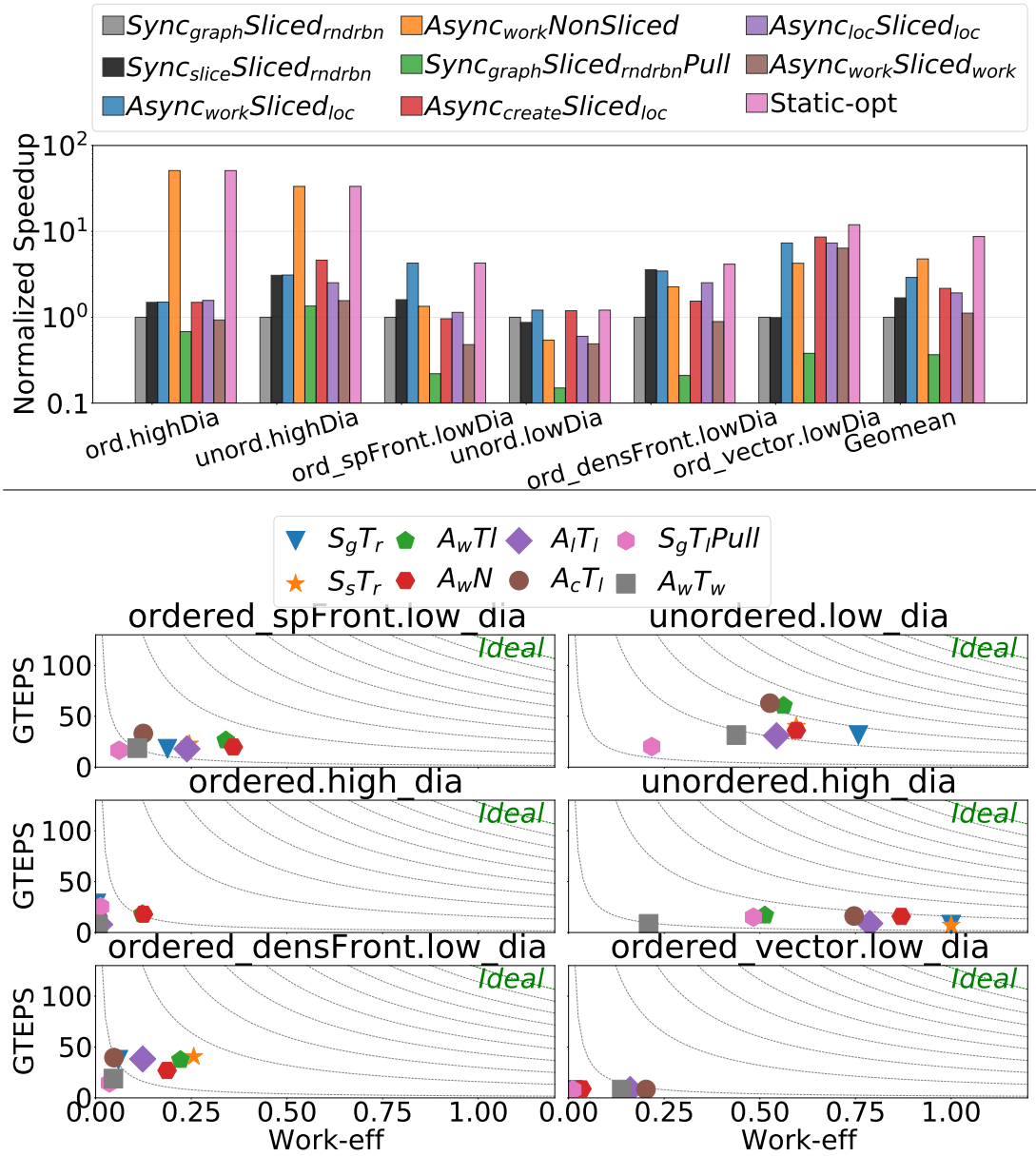


Figure 7.14: Comparison of Algorithm Variants.

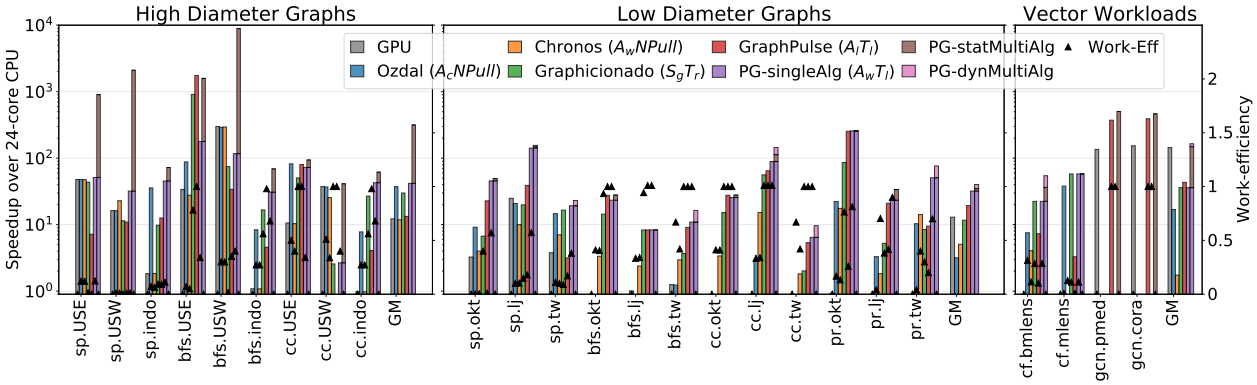


Figure 7.15: Overall Performance Comparison (Gunrock does not implement CC, CF; GCN does not have pure asynchronous implementation.)

Finally, the best pull variant, $\text{Sync}_{\text{graph}}\text{Sliced}_{\text{rndrbn}}\text{Pull}$ consistently performs worse due to pipeline stalls waiting on random reads and work-efficiency loss from accessing all incoming edges irrespective of whether they are active.

Work-efficiency vs Throughput for Algorithm-Variants Figure 7.14 further explains the workload and graph type tradeoffs. Slicing improves memory efficiency for low diameter graphs, while $\text{Async}_{\text{work}}$ improves work-efficiency for order-sensitive workloads. Since high diameter graphs are regular, Non-sliced is superior as it achieves high hit rate while avoiding barrier overheads. For dense frontier workloads, slice synchronous balances memory and work-efficiency. For the vector workload, CF, memory efficiency is implicitly high, thus asynchronous variants dominate due to faster convergence.

7.7.2 Comparison to Prior Accelerators

Figure 7.15 shows the overall comparison. PG-statMultiAlg allows variant flexibility at the workload level, and PG-dynMultiAlg enables dynamic switching. Overall, PolyGraph outperforms CPU by 105.7× and GPU by 49.4×. Over the fastest prior accelerator, GraphPulse, it achieves 5.7× speedup.

High Diameter Graphs: PolyGraph achieves work-efficiency-proportional gains over Gun-

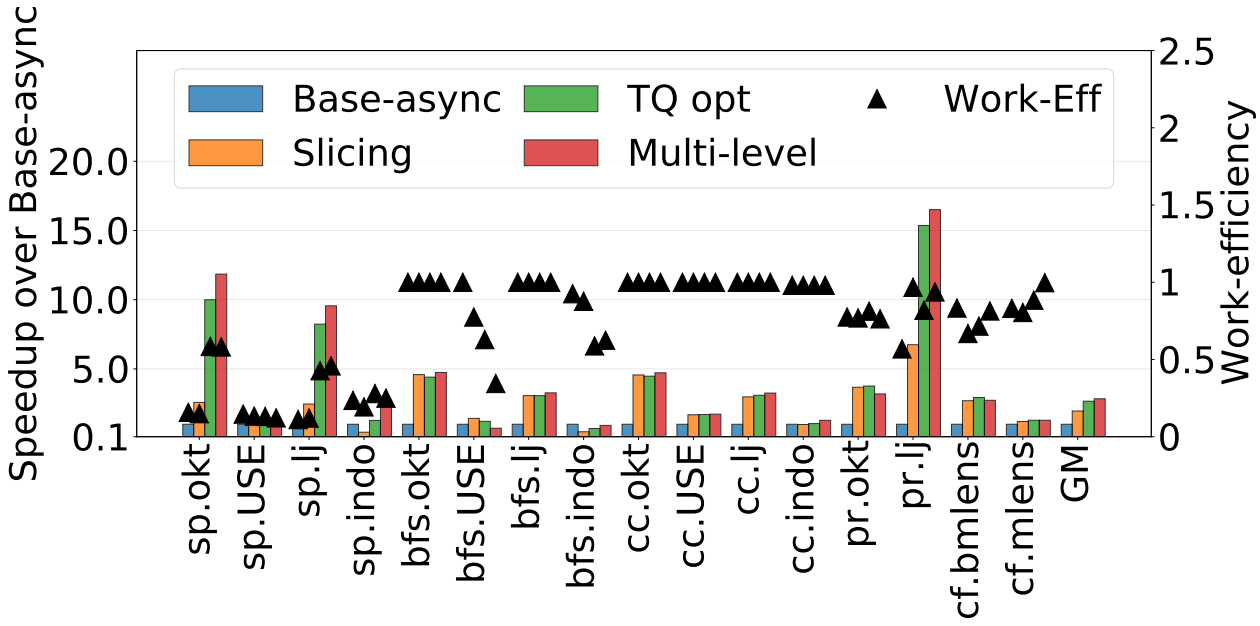


Figure 7.16: Cumulative Speedup of Novel Features

rock's $\text{Sync}_{\text{graph}}$ GPU version. Even through Chronos and Ozdal use non-sliced implementations, they also use the inefficient pull variant. For the `USW` graph that does not fit in on-chip memory, GraphPulse and PG-singleAlg lose due to slicing overheads of switching time and work-efficiency loss due to delayed cross-slice vertices. A non-sliced variant can avoid these overheads.

Low Diameter Graphs: For order-sensitive SSSP and PR, PG-singleAlg gains work-efficiency due to vertex scheduling and slicing significantly improving hit rate. Since BFS and CC are less order-sensitive, PG-singleAlg behaves similar to Graphicionado. However, dynamic switching improves performance (eg. `bfs.lj,bfs.tw`).

Vector Workloads: For GCN, accelerators provide high speedup, as they support efficient broadcast of weight matrices, while GPU is bottlenecked by the unified cache bandwidth [246]. PolyGraph gets about 25% speedup by overlapping the communication-intensive aggregation and computation-intensive multiplication tasks. CF gains work-efficiency with synchrony and throughput with switching to the non-sliced variant in later iterations.

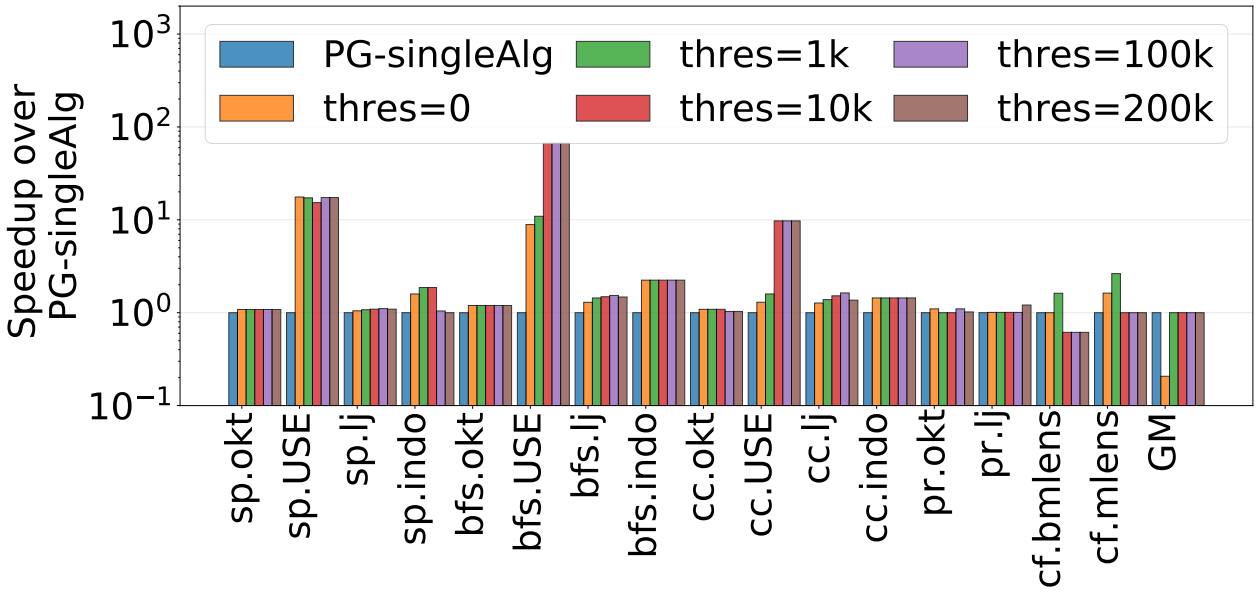


Figure 7.17: Dynamic Switching and Threshold Sensitivity

Novel Features Figure 7.16 shows the cumulative speedup of each novel features over an asynchronous-push accelerator. In general, slicing significantly improves memory efficiency on low diameter graphs (`okt,lj`), while it causes slowdown in high diameter graphs due to work-efficiency loss (`bfs.indo`). Task coalescing particularly benefits low diameter graphs by eliminating superfluous updates to high degree vertices. The locality optimized multi-level partitioning with a fixed cluster size provides 20% benefit. For `BFS.USE`, multi-level is worse than naive partitioning because load is the primary bottleneck when locality is available. Thus, optimal cluster size depends on the input.

7.7.3 Algorithm Sensitivity

Dynamic Switching Heuristic Figure 7.17 shows the results of dynamic switching, with the heuristic described in Section 7.3.3.

Here, we compare performance as we sweep the switching threshold of active vertices. The initial variant is the single-variant-optimal (A_wT_l). For low diameter graphs (`lj,okt,mlens`), performance improves to some point due to avoiding slice-switch overheads (up to 28%

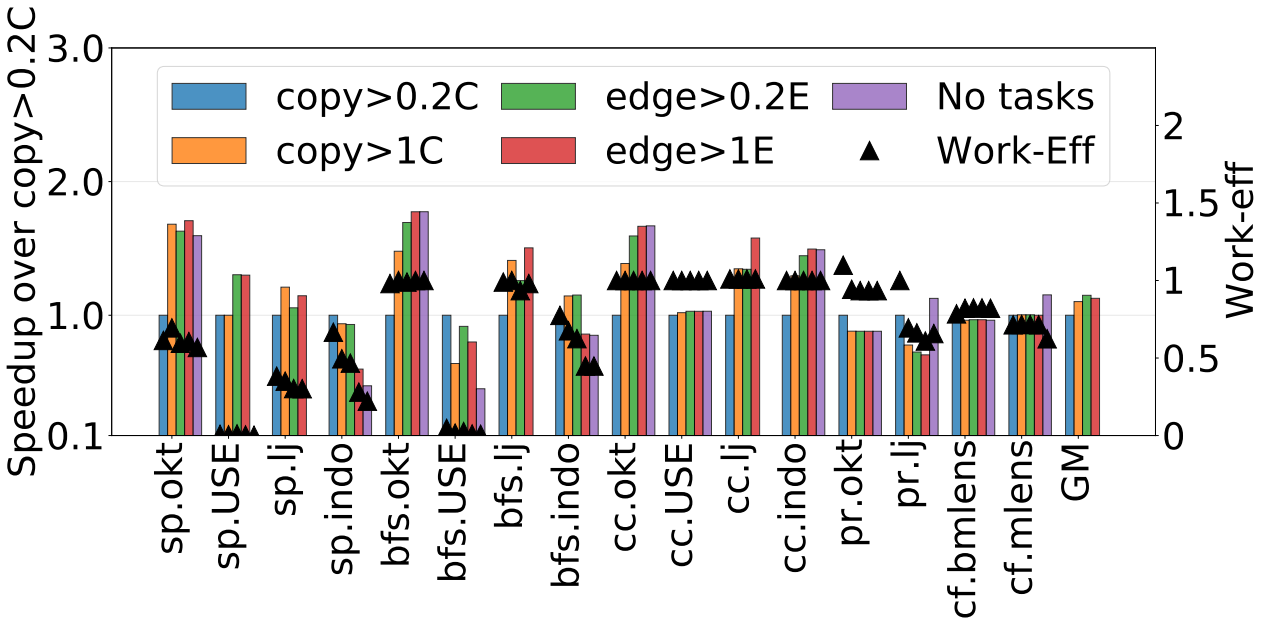


Figure 7.18: Slice Switch Heuristics (C: cross-slice vert., E: edges/slice)

here); after this point performance reduces due to low memory efficiency of non-sliced. The effect is more dominant on non-computation intensive workloads like BFS. For high diameter graphs, the speedup plateaus, as non-sliced can achieve similar memory efficiency; thus, static flexibility is sufficient for them. Overall, dynamic switching helps primarily low diameter graphs.

Slice Switching Heuristic Figure 7.18 compares slice switching heuristics on order-sensitive workloads[7]. "No tasks" represents switching when no outstanding tasks are left; it is either worse performance (`sp.indo,bfs.indo`) or does not converge (`sp.use,sp.lj`). Low diameter graphs (`okt,lj, mlens`) prefer a larger threshold because their clustered structure allows higher ratio of intra-slice vs inter-slice updates. High diameter graphs (`use,indo`) are highly sensitive to delayed updates and prefer to switch earlier. Note that for larger high diameter graphs, the work-efficiency loss still dominates, and no-slicing wins if flexibility is available. We chose the edge-based heuristic, with a slicing threshold of 0.25*E for high

[7]Note that since `okt,USE` fits in PolyGraph's on-chip memory, we reduced the on-chip memory size to half to make this experiment interesting.

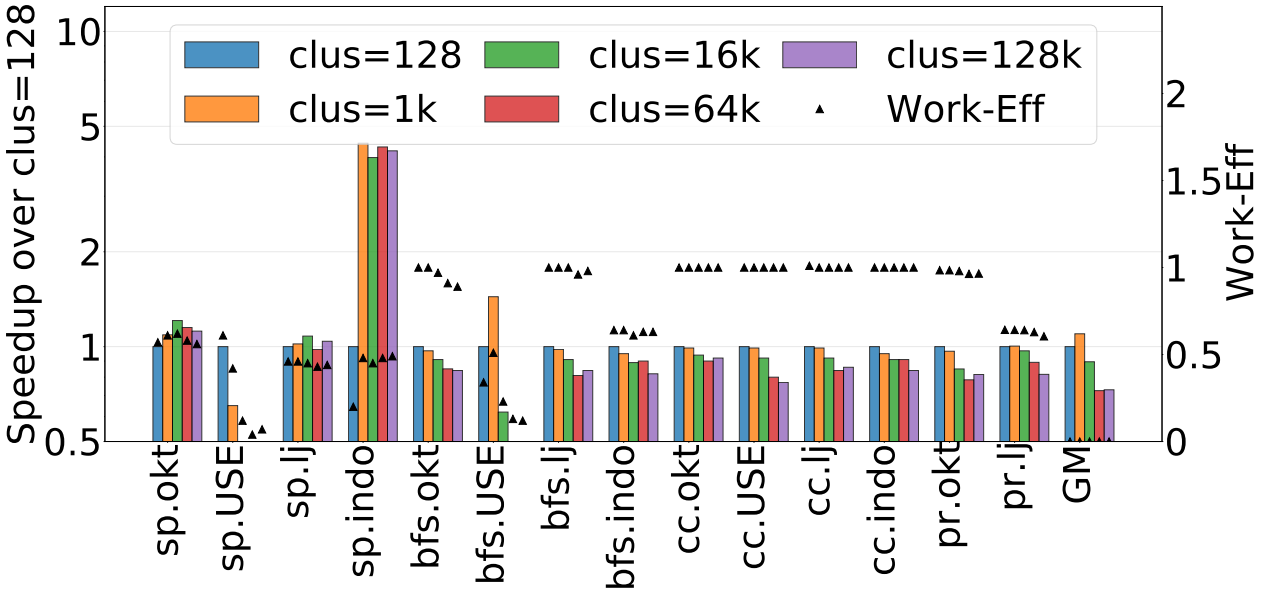


Figure 7.19: Sensitivity to Spatial Partitioning Cluster Size

diameter graphs and 1*E for low diameter graphs.

Spatial Partitioning Figure 7.19 evaluates the multi-level spatial partitioning policy for different cluster sizes (on A_wT_l variant)[8]. The results suggest small clusters optimize for dynamic load balance, while larger clusters improve locality. SSSP has higher computation intensity, and is thus more bottlenecked by locality than load balance; hence larger cluster sizes are better. BFS prefers smaller cluster sizes as memory level parallelism is more critical due to its low computation intensity. The default cluster size is 128; we use 16k for low diameter graphs (except BFS). Overall, flexible multi-level spatial partitioning provides 40% performance gain over conventional clustering.

Per Data-structure Reuse Figure 7.20 shows the per-phase access frequency of edge and vertex data-structures for a subset of interesting variants. The access counts are averaged for a single phase for synchronous or 100k cycles for asynchronous. In general, vertices

[8]We do not evaluate this for CF, GCN because, due to their large vertex properties, their maximum cluster size is too small.

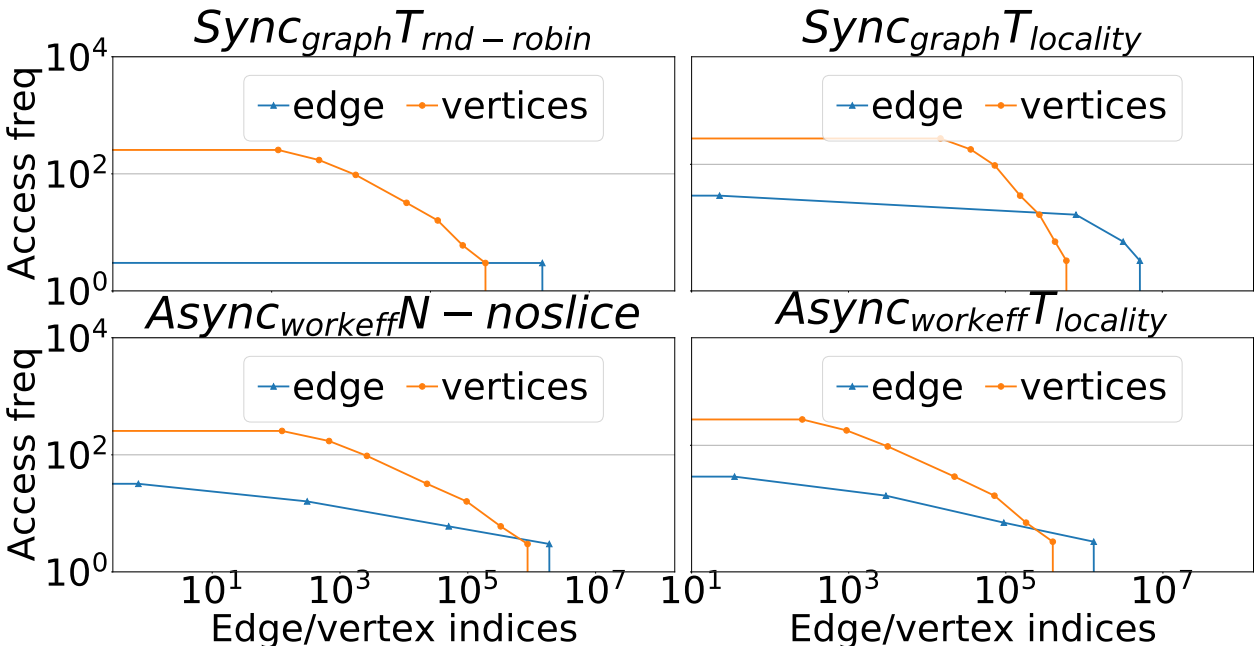


Figure 7.20: Access Patterns in Algorithm Variants (for SP.lj)

are more critical to cache on-chip because they have higher reuse and also require finer-grained random accesses. Although edge reuse is also significant because of iterating over a single slice multiple times (T-slice$_{\text{locality}}$ variants), we found that caching edges is not always beneficial. This is because it reduces vertex slice size too much, which hurts work-efficiency. Finally, the vertex_list data-structure has similar access behavior as vertex_prop, therefore we pin vertex_prop and vertex_list on-chip for sliced variants.

7.7.4 Hardware Sensitivity

In this section, we discuss the performance sensitivity of graph algorithm variants to hardware resources. Note ordered workloads are geomean of SSSP, PR and CF and unordered include BFS and CC. We discuss GCN separately.

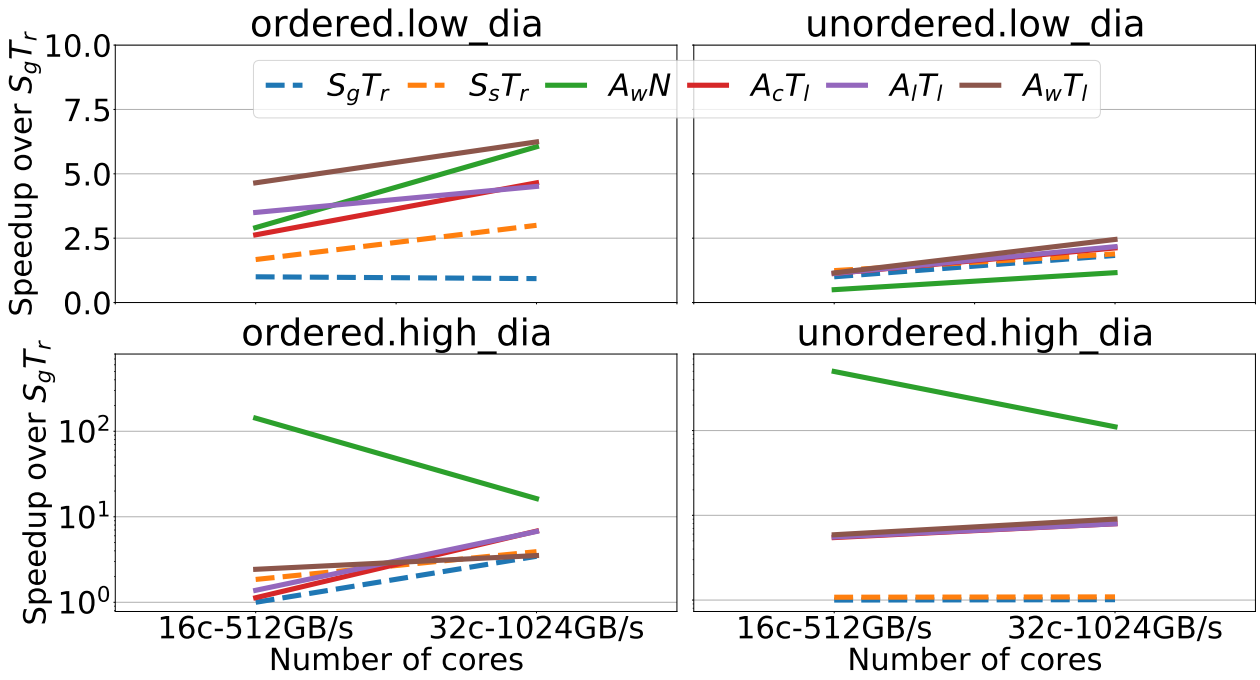
PolyGraph Scaling With 2x cores (and 2x memory bandwidth), the performance scales well (limited by available parallelism) as shown in Figure 7.21(a). Even though low diameter graphs are highly sensitive to memory bandwidth, scaling on unordered workloads is limited

by parallelism while ordered workloads suffer due to loss in work-efficiency with larger network latency. Since high diameter graphs are easy-to-partition, they ensure high hit rate and thus reduced dependence on memory bandwidth. A_wN `high_dia` case shows performance loss with more cores: this is because larger working set means higher sensitivity to factors affecting work-efficiency, like larger network latency for 32 cores.

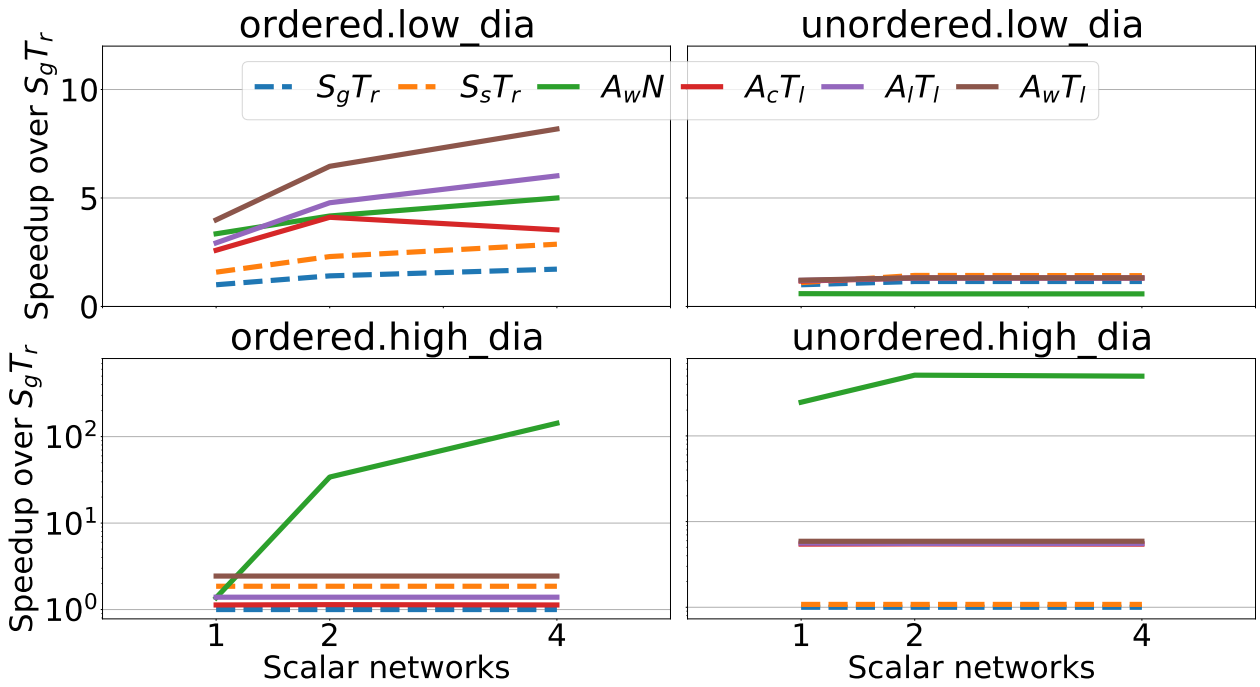
Network bandwidth Figure 7.21(b) sweeps over scalar network bandwidth. The sliced variants of low diameter graphs have good on-chip memory locality, and are bottlenecked by network bandwidth. For high diameter graphs, both sliced/non-sliced variants achieve high hit rate and memory locality. Therefore, sliced variants are bottlenecked by load imbalance due to frequent synchronization, while for non-sliced, work-efficiency is proportional to the network bandwidth.

On-chip memory size Figure 7.22 shows the performance sensitivity to on-chip memory size: this is a proxy for scaling up graph-size (more slices required and more cache pressure), while using a consistent input. The data is presented for specific input graphs as the saturation point depends on the ratio of graph and on-chip memory size. For ordered workloads, a larger memory size improves work-efficiency with lesser cross-slice edges while reducing the required number of barriers. The latter is a small factor, as can be seen for `unordered` cases. `Ordered.rdUSW` A_wN is an exception where performance degrades with larger on-chip memory; this happens because a larger working set may cause cores to become too unsynchronized, hurting working efficiency.

GCN Sensitivity When doubling core count, GCN's performance scales by 1.9×, with little loss due to load imbalance in aggregation phase. For scaling network bandwidth, GCN improves linearly up to 64-byte bandwidth, after which computation becomes the bottleneck. For scaling down on-chip memory, only if we scale down to 4MB does the memory bandwidth become the bottleneck; this happens with 7 slices, since our GCN's graphs require maximum 28 MB.



(a) Cores and Memory-bandwidth



(b) Number of 8-byte Networks

Figure 7.21: Sensitivity to Hardware Resources

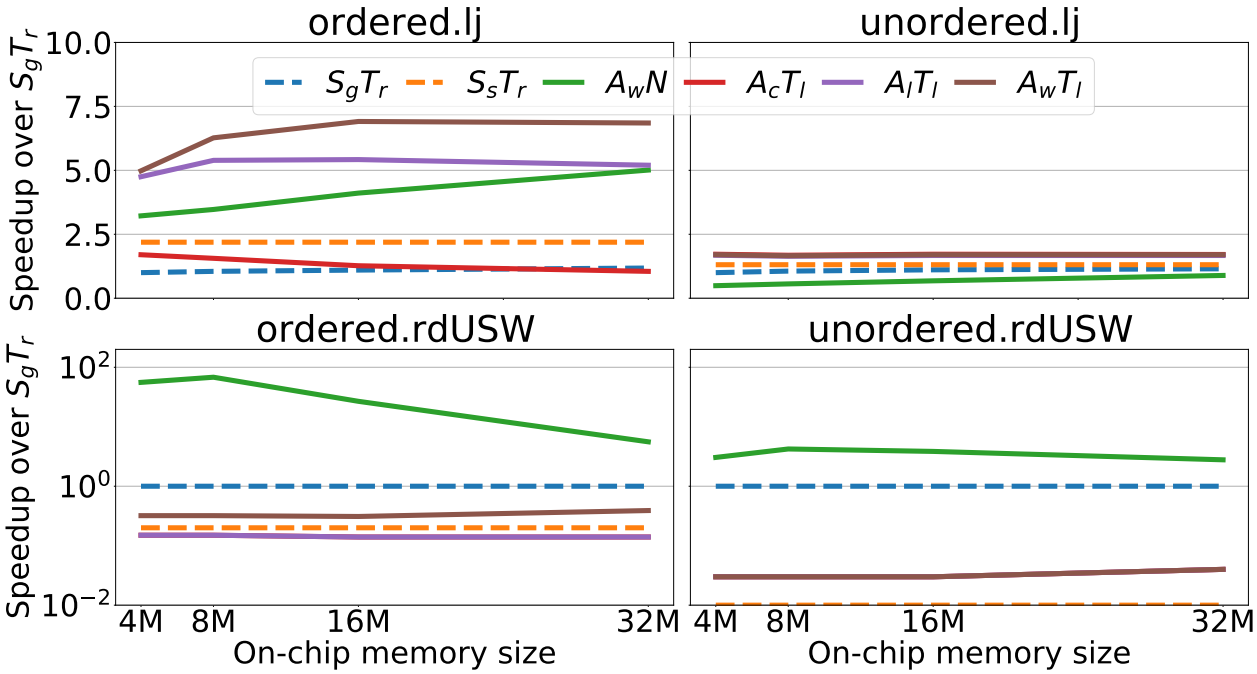


Figure 7.22: Sensitivity to Memory Size

Area Tradeoffs Table 8.7 shows PolyGraph's area breakdown. It occupies 72.56mm^2, with eDRAM consuming 91.1% of the total area. Compared to Graphicionado, PolyGraph is similar area with 84% power due to using a mesh instead of a crossbar.

Figure 7.23 compares accelerator speedup and area. Overall, PolyGraph has similar area as Graphicionado while achieving 7.2× speedup due to its optimizations and flexibility. We also examine area tradeoffs for PolyGraph by removing the components that consume significant area (eliminating certain variant options). Without caches, memory flexibility is not available, hurting high diameter graphs. Without a priority queue, the gains on order-sensitive workloads is reduced. With no dynamic tasks, S_sT_r is the best variant, as it provides some work-efficiency by conveying updates sooner, with high memory efficiency of locality scheduling.

	Area (mm^2)	Power (mW)
Control Cores	0.053	11.5
Priority Task Queue	0.05	15.86
Task Coalescer	0.05	3.4
On-chip mem+ctrl	4.15	25.64
CGRA (4x5)	0.21	80
8x8 8-byte crossbar	0.002	1.92
1 PG-Core	4.51	138.3
4x4 32 byte mesh (1)	0.2	44.7
4x4 8 byte mesh (3)	0.2	34.22
PG Total	72.56	2292.12

Table 7.5: Area and Power breakdown for PG-flex (28nm)

7.8 Additional Related Work

Table 7.6 categorizes prior work by variant. All variant combinations are supported by PolyGraph.

Graph Frameworks with Flexibility While some software graph frameworks focus on a single algorithm variant (eg. GraphMat [224]: S_gN), several others allow certain amount of flexibility in their programming model. For example, Galois [182] provides optional support for scheduling vertex buckets by a data-dependent priority (Minnow [271] provides hardware support for CPUs). Salvador et. al. [203] studies the interaction of update direction and memory coherence/consistency models for GPUs, and demonstrates the usefulness of flexibility. We further show the usefulness of flexibility across dimensions of update visibility and vertex/slice scheduling.

Powergraph [87] supports both synchronous and asynchronous variants. Powerswitch [260] adds heuristics to switch between sync/async dynamically, which we find is not effective for accelerators. X-Stream is "edge-centric", where edges are streamed without sparse access through vertex indices. Even though edge-centric can be supported by PolyGraph, we did

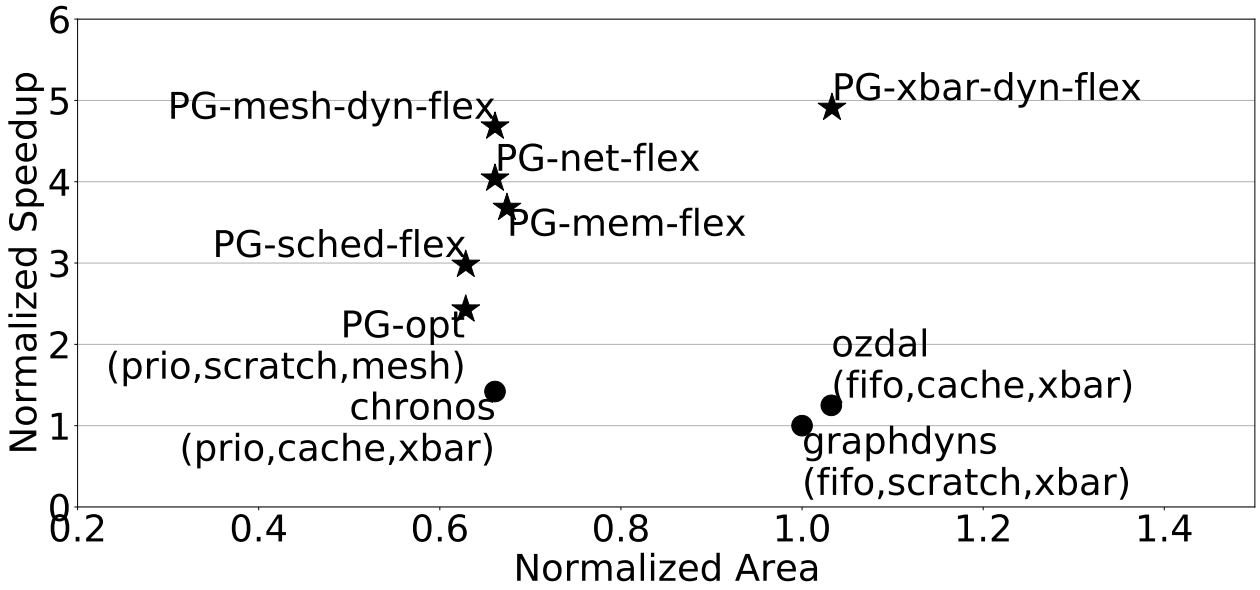


Figure 7.23: Accelerator Performance vs Area

not consider it as it is incompatible with key optimizations like priority ordering and vertex-based dynamic tasks.

Graph Taxonomies McCune et al. classified distributed graph frameworks [149]. Lenharth's taxonomy [135] identifies factors that impact graph execution: (topology, synchronicity, re-ordering, graph operators). These works do not consider hardware specialization or slicing.

Hardware Accelerators Graphicionado [94] accelerates push-based synchronous variant. GraphDyns [262] adds dynamic work-distribution. Ozdal et. al. supports sequential consistency in its asynchronous graph processing ASIC template [173]. Our evaluated workloads do not require sequential consistency guarantees for correctness; only CF may converge faster with consistency [146].

Digraph [278] is a multi-GPU system for asynchronous graph processing. Chronos [11] is the only prior asynchronous accelerator that supports fine-grained priority scheduling. Several designs exploit multiple HMC nodes (eg. Tesseract [15]). GraphP [274] extends Tesseract with a two-phase programming model enabling efficient partitioning. GraphQ [285]

	Non-sliced(N)	Temporally-sliced (T)
$\text{Sync}_{graph}(\text{S}_g)$	Tesseract [15]	Graphicionado [94] (T_{robin})
	GraphMat* [224]	GraphDyns [262] (T_{robin})
$\text{Sync}_{slice}(\text{S}_s)$		GraphQ* [285] (T_{robin})
		GraphABCD* [264] ($\text{T}_{work-eff}$)
$\text{Async}_{local}(\text{A}_l)$	Ozdal [173]	GraphPulse [193] ($\text{T}_{locality}$)
	Giraph* [97]	
$\text{Async}_{creat.}(\text{A}_c)$	Graphlab* [146]	
$\text{Async}_{Work}(\text{A}_w)$	Chronos [11]	Digraph* [278] ($\text{T}_{work-eff}$)
	Galois* [182]	
	Minnow [271]	

Table 7.6: Prior Works in Taxonomy (*software frameworks)

has a hybrid execution model; asynchronous within each HMC.

DepGraph [279] combines updates across frequently accessed paths, reducing re-execution overhead. This is an alternative way of improving work-efficiency.

Graph Spatial Locality Techniques Graph preprocessing is employed to improve spatial locality [118, 70, 235], which is especially useful if graphs are executed in locality/vertex order. HATS [155] is a CPU offload accelerator which dynamically discovers graph locality. Polymer [273] explores spatial placement and replication of vertices in a distributed system.

7.9 Discussion

In this work, we proposed a flexible graph processing accelerator. PolyGraph demonstrated promising performance on a wide range of graph inputs and workloads. However, for practical adoption, it must scale to large graphs gracefully. In this section, we first discuss our limit study that demonstrates the potential of PolyGraph's techniques and the challenges we will

need to overcome for large systems. We will also discuss preliminary ideas to mitigate the bottlenecks. Finally, we will discuss additional scheduling policies and their interplay with work efficiency for asynchronous variants.

7.9.1 Limit Study

Figure 7.24 shows the results for PolyGraph's scalability on three graph workloads: BFS (on Orkut graph (Table 8.3)), and Triangle Counting (TC) on Flickr graph, which has 820K vertices and 9.8M edges). We included Triangle Counting (TC) and Jaccard Coefficients (JC) because they represent workloads with better reuse and will produce different tradeoffs with scaling. We implemented TC and JC using our specialization for edge-list intersection (i.e., stream-join from Chapter 5) as well as symmetry-breaking optimizations. Several increasingly realistic designs were evaluated, as follows:

- **Ideal:** Ideal architecture with instantaneous communication and load-rebalance among cores, infinite memory bandwidth, and infinite network bandwidth.
- **+Distributed processing:** work split across cores corresponding to the static assignment of vertices/edges using multi-level spatial partitioning policy.
- **+Constrained memory bandwidth:** Memory bandwidth restricted to 1 TB/s for 16 cores.
- **Constrained network bandwidth (128 GB/s):** Link bandwidth restricted to 2x64B/cycle@1GHz (double mesh).
- **Constrained network bandwidth (512 GB/s):** Bandwidth increased by 4×.

Figure 7.24 shows performance scaling from 1 to 256 cores. The high performance of the Ideal (i.e., unrealistic) system indicates that PolyGraph's asynchronous algorithm exposes sufficient and scalable parallelism, even with the modest graph sizes used in this experiment. Adding Distributed processing reveals that load imbalance can be a critical overhead

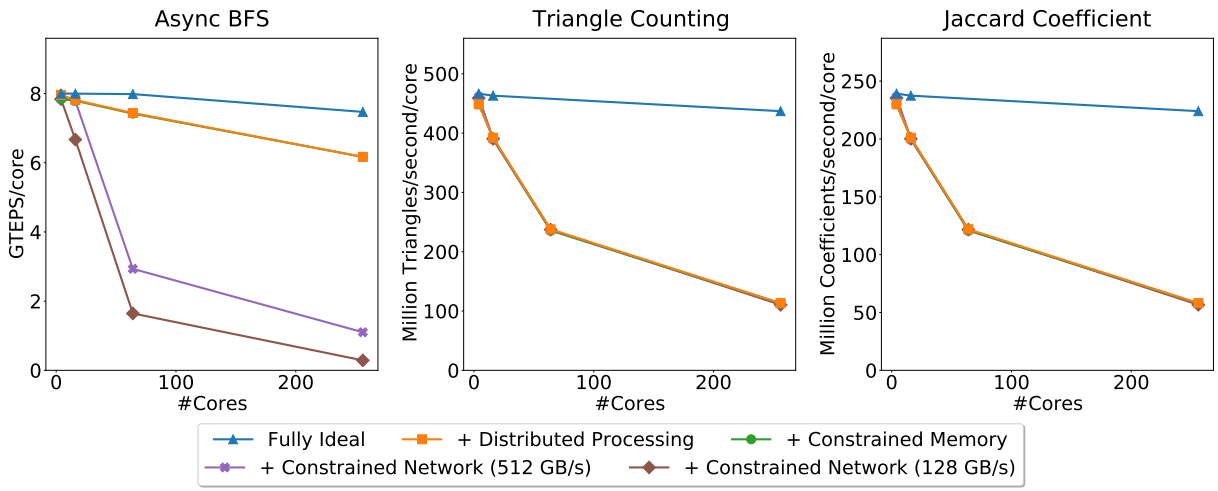


Figure 7.24: Bottlenecks with More Cores

for vector workloads like TC and JC. Next, adding a Constrained memory bandwidth of 1 TB/s for 16 cores does not lower performance. Even though the memory bandwidth utilization problem does not worsen with scaling, it is unsolved even for lesser cores. For BFS, it only achieves 25% utilization. Reducing the network bandwidth (Constrained network bandwidth) will greatly reduce the performance of BFS, which indicates that scalar vertices in BFS put pressure on the network.

Unsurprisingly, the biggest challenge is communication. The network is used to send update task arguments across spatial graph partitions, so more cores mean the graph is cut into more slices (more inter-core edges), and the network is more utilized due to more hops/message on average. To keep the resource requirements practical, the utilization needs to be improved using better spatial partitioning policies or nodel techniques for optimized traffic.

7.9.2 Factors Impacting Convergence Rate

Here we give pointers to improved scheduling policies and discuss the convergence unpredictability with the asynchronous variant. Finally, we will hint toward the applicability of

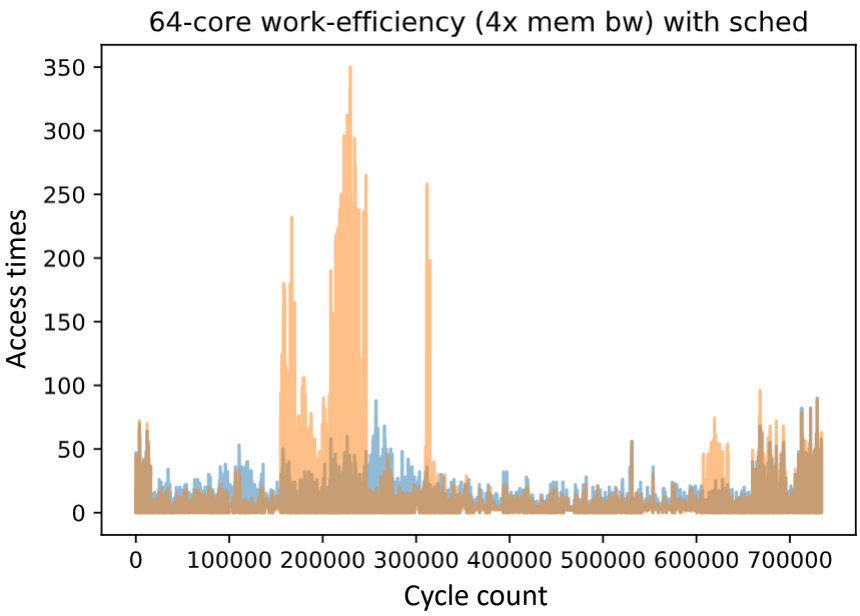


Figure 7.25: Skewed Execution of Edges in SSSP

insights for broader learning-based workloads.

Alternate Priority Scheduling Techniques PolyGraph explores various scheduling strategies that tradeoff between locality and work efficiency. We used a standard priority order for work efficiency: completely runtime-based and with priority ordering by distance or error). One can improve the priority metric according to architectural support. See the experiment in Figure 7.25, the x-axis represents edge ID, and the y-axis represents the number of times each edge is executed. The brown histogram shows the first-in-first-out ordering where specific edges are executed an exponentially higher number of times – this is expected as high-degree vertices will propagate data more often. If we could de-prioritize these high-degree vertices (blue histogram in Figure 7.25), more tasks for high-degree vertices can be coalesced, resulting in improved work efficiency. The challenge will be to devise a combined scheduling metric that considers the original priority metric and vertex degree.

Implications on Deep Learning Training In our experience, asynchronous algorithms can be unstable and hard to reason about, especially at scale. In PolyGraph, we had to pick

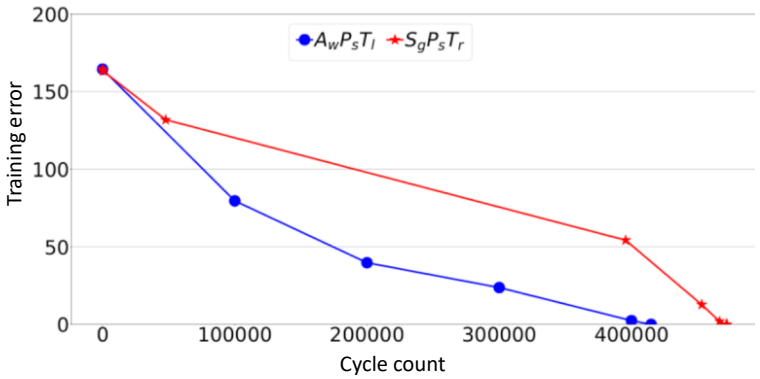


Figure 7.26: Convergence Analysis of Synchronous and Asynchronous Algorithms (for BFS on LiveJournal graph)

a larger learning rate for asynchronous variants to ensure they converge. Figure 7.26 plots how the error reduces with time for synchronous (S_g) and asynchronous (A_w) variants when running BFS on a Flickr graph, which has 820K vertices and 9.8M edges. The synchronous variant shows slow convergence initially because it has not yet seen the whole graph. However, asynchronous shows a more consistent drop in error due to covering a larger footprint. The larger footprint may also cause the asynchronous algorithm to deviate. Thus, we used a larger learning rate to make the algorithm closer to the converging point faster.

Our algorithm variants may apply to deep learning (DL) training algorithms that use gradient-based techniques similar to Collaborative Filtering [146]. However, further research will be required when analyzing large feature vectors in DL compared to graph processing.

CHAPTER 8

Accelerating Task-Parallel Workloads with Coarse-Grained Dependencies

This chapter focuses on accelerating task parallel workloads with coarse-grained data dependencies. There are two critical differences compared to graph processing: 1. tasks are coarse-grained: hence, careful distribution of computation resources is required, 2. dependencies are also coarse-grained: thus, dependencies can no longer be resolved using live operands, and they will need to be buffered in memory.

PolyGraph hardware, developed in the previous chapter, is inefficient for coarse-grained tasks as writing and reading dependent data from shared memory requires a synchronization barrier. The barrier limits the available concurrency and slows the execution. On the other hand, using static scheduling (as in SPU), one could co-schedule dependent tasks and directly stream data between them. However, SPU is only applicable when task information is known offline. Our goal is to retain the dynamic scheduling benefits of tasks while recovering the lost locality structure.

Our insight is that if we expose the tasks' potential for communication structure are first-class primitives in TaskStream, it is possible to recover program structure with extremely low overhead. Specifically, we augment TaskStream (and PolyGraph) with three optimizations: work-aware load balancing, recovery of pipelined inter-task dependencies, and inter-task read sharing through data multicast.

We chose five challenging task-parallel workloads with unique opportunities for specializable communication: K-nearest neighbor (kNN) using kd-tree traversal, an ML-oriented

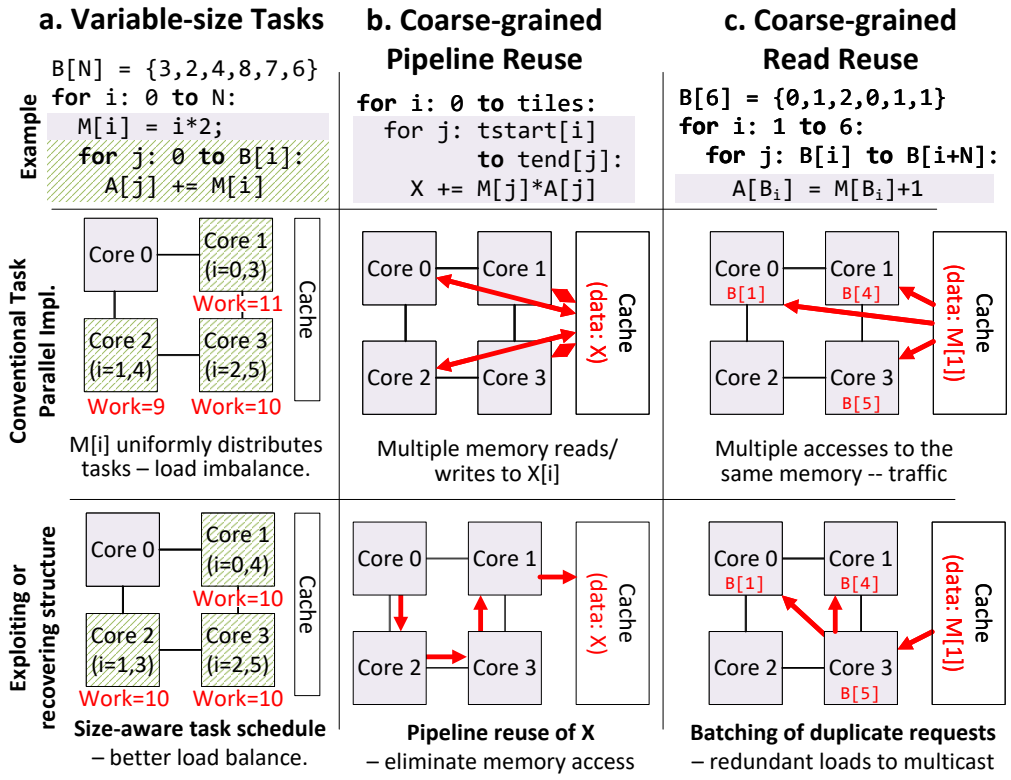


Figure 8.1: Opportunities in Naïve Task Parallelism

database query, sparse matrix-multiply, Cholesky decomposition, and Graph Convolution Networks (GCN).

Overall, we achieve 81.3× speedup over multicore CPUs. Over an efficient static-parallel CGRA baseline (without TaskStream), we achieve 2.2× speedup, with load-balancing optimizations yielding only 1.3× performance.

In this chapter, we first motivate and describe the key optimizations to TaskStream in Section 8.1. In Section 8.2, we discuss the execution model, describe the mapping of our evaluated workloads, and discuss limitations and possible extensions. Section 8.3 describes the accelerator hardware implementation, and then Sections 8.4 and 8.5 provide methodology and evaluation. Then, Section 8.6 describes the relationship to existing software and hardware systems that optimize for task locality. Finally, Section 8.7 discusses alternate decisions and pointers to future work.

8.1 TaskStream Optimizations

We first motivate the optimizations in TaskStream by discussing opportunities to exploit certain forms of program structure. Then, we will present our proposed TaskStream model, abstract from any particular architecture implementation.

8.1.1 Opportunities for Structure Recovery

The context for our proposed system is a tiled multicore architecture, in which a task scheduler assigns tasks to cores. We assume a mesh-based network on chip (NoC), but optimizations apply to other topologies. To elaborate our optimizations, we discuss three program idioms from Figure 8.1, where locality structure is lost due to exploiting task parallelism.

Variable-sized Tasks A variety of task-parallel workloads have task types[1] whose amount of work is either data-dependent or progressively changing over its instances. Figure 8.1a shows an example where inner loop tasks have a data-dependent length, based on B[i]. A naive task parallel model would assign the inner loop tasks irrespective of the work involved in a task. Work-stealing is possible, but requires extra inter-core communication latency and bandwidth.

The opportunity here is to distribute tasks with the knowledge of the work involved. In the example, core 1 gets the smallest and second-largest task (i.e. with total work = 3+7 = 10), so that all cores get similar total work. This model is synergistic with accelerators, which have quite predictable execution times.

Coarse-grain Pipeline Reuse A common behavior in data processing algorithms is ordered dependences between one task and another, for example where one task produces an array which the other uses in the same order. Figure 8.1b demonstrates a global reduction

[1]A "task type" is the static definition of a task, including computation and memory accesses, while the dynamic instantiation of a task is a task instance.

example where each core gets a tile of data. In the NSAïve task parallel implementation, all cores need to perform updates on the reduction variable through memory.

The opportunity here is to identify the ordered reuse, and pipeline or *stream* the data from a producer to one or more consumer tasks. This transforms the memory traffic into direct network traffic, reduces shared-memory overhead from coherence, and also allows overlapped execution of tasks for more concurrency. In the example, the pipelined reduction can be performed without accessing memory (except for writing the final value).

Coarse-grain Read Reuse Another common idiom is when different subsets of tasks read the same data. If such tasks are not scheduled together in time or space, the opportunity to exploit this form of reuse can be lost. Figure 8.1c demonstrates this with an algorithm that traverses and modifies a compressed sparse row (CSR)-like data structure, and is representative of common algorithms that rely on range-based indirection. Here the duplicates in B are expected to create multiple tasks with shared read data, providing an opportunity for reuse. A naïve task parallel model schedules tasks without respecting locality, so tasks that access the same data may not be scheduled on the same core or at the same time. The reuse cannot be exploited to save network traffic and cache/memory bandwidth.

Such coarse-grain reuse can be exploited by identifying tasks that access the same data, and reordering them to execute at the same time on different cores; the responses can then be multicast to significantly reduce network traffic and memory bandwidth usage. We call this optimization *task batching*.

An alternate opportunity, used in a variety of other contexts [41, 105, 109, 266, 11, 63], is to use a "spatial hint" to assign tasks that access the same data to the same cores. While this reduces memory access for data that fits in private cache, it also restricts the allowed scheduling locations, which could restrict load balancing optimizations. We compare against `spatialhint` in evaluation.

8.1.2 TaskStream Model

An appropriate task-parallel execution model should exploit memory locality within coarse-grained tasks and programmatically specify structure-recovery opportunities. Reconfigurable accelerators already exploit memory locality and prefetching using streams [162]. Therefore, we can reuse and augment the support that already exists in the TaskSTream framework. the TaskStream framework that naturally integrates with reconfigurable accelerators due to its load balance approach is an attractive choice. It is also easy – specifically, we augment TaskStream to expose structure-recovery optimizations by introducing new inter-task edge types. Here, we first discuss the basics and then cover how each optimization is applied.

TaskStream Basics We augment TaskStream edge types to indicate the potential for structure-recovery: creation (standard), streaming (for pipeline reuse), and batching (read reuse). Tasks can be in one of three states: *1. Created:* the arguments for a task instance are constructed on the originating core; *2. Scheduled:* the task is bound to execution resources, *3. Executing:* task computation is in progress.

Tasks are created when they receive values for all incoming creation (standard) edges. Next, a task is scheduled to storage/execution resources (e.g., buffer/core), after which it is assigned a TaskID that represents this location; the TaskID may be returned to the parent if a streaming communication will be established. Tasks may only be scheduled to a core configured for its task type. To convey the configuration information, each task node is annotated with a `coreMask`: a bitmap that describes the legal mapping locations. Some task types can be co-located on the same core, provided sufficient resources exist. Tasks that are not yet *ready* to execute may be waiting on streaming or batched data, and we call these tasks *pending*.

One phase of the program completes when all tasks are completed. A program may consist of multiple phases. Examples of a TaskStream program phase are shown in Figure 8.2, where task types are distinguished by color (and shading), and we will discuss next.

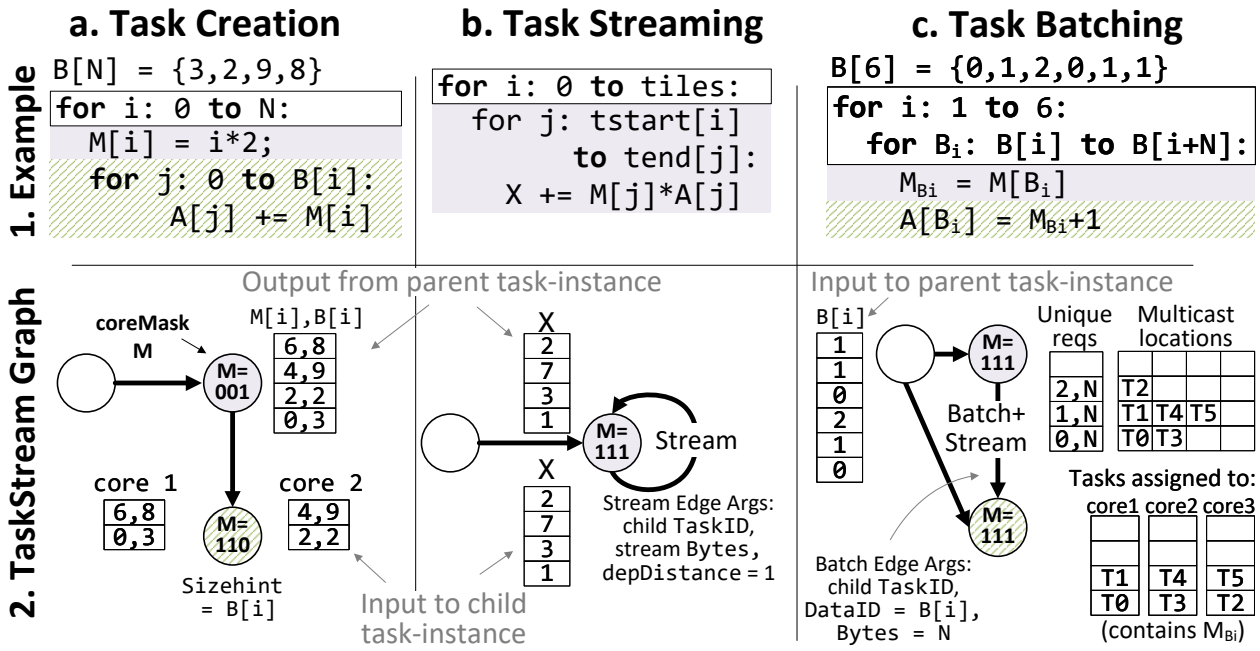


Figure 8.2: TaskStream Graph Abstractions

Task Creation & Work-aware Load Balance Figure 8.2a demonstrates the basics, as well as annotations for load balancing. A single task creation edge connects the outer-loop multiplication task, and inner-loop accumulation. When two task nodes are connected by a *task creation* edge, it means that some outputs from the source node task are used to activate the creation edge and will be inputs to the destination node task. In Figure 8.2, the interface for activations is represented by task buffers, and the data order indicates the order of producing and receiving tasks. The outer-loop task gets core 1 (`coreMask:001`) while accumulation gets the remaining 2 cores (`coreMask:110`) because it has a ratio of `B[i]` times more work compared to the outer-loop.

TaskStream provides annotations to aid load balancing at task scheduling time. Task creation edges may be annotated with a `sizehint`, which is a task argument that describes the relative amount of work for the task. This enables a simple size-aware scheduling policy, where a new task is assigned to the core with the least cumulative work. In the example, `B[i]` is the number of iterations of the inner loop, and is therefore used for that task's `sizehint`. The scheduler could then assign tasks of size 3, 8 to core 1 and tasks of size 2, 9

to core 2, resulting in a balanced load.

Task Streaming To facilitate dynamic pipelining between tasks, edges may be of *task streaming* type. When two nodes are connected by such an edge, the output at the source node task triggers a streaming communication with the assigned children (stored as child `TaskID`). For a streaming edge, the programmer can specify a dependence distance (`depDistance`), which allows developing a streaming relationship between task-instances separated by a fixed number of tasks. In the example, the `depDistance` is 1. To close the communication, an *end-of-stream* message is sent when the required number of `bytes` have been streamed in, this parameter must be specified by the producer. To set up the communication, *start-of-stream* handshaking messages are exchanged to ensure that the children are ready, and producers have their scheduling information. The data is streamed in between these messages.

Figure 8.2b demonstrates the task streaming edge: here the dependent instances of the reduction task type are scheduled in mutually exclusive locations – this is required to ensure that tasks involved in streaming are concurrently scheduled. When data is available at the parent task, the *start-of-stream* message is sent to the destination node and data will be streamed in. When finished, *end-of-stream* messages will free resources.

Task Batching To enable multicasting of shared reads, we implement a *task batching edge*. This edge requires three parameters when it is activated: `DataID` indicates whether the reads are to the same data, `TaskID` indicates the dependent dynamic task, and `bytes` indicates the length of these reads. The task scheduler can use this information to record which tasks are dependent on the same reads, and reorder them to schedule them together. The advantage is that data can be multicast to all co-scheduled tasks.

Figure 8.2c demonstrates the task batching edge: here we split the program into a CSR-traversal task and an addition task type. The outputs from `B[i]` are batched, resulting in only 3 unique requests instead of 6. For each unique request, the TaskIDs of the corresponding batched tasks are shown in the "Multicast locations" table. We are able to perform

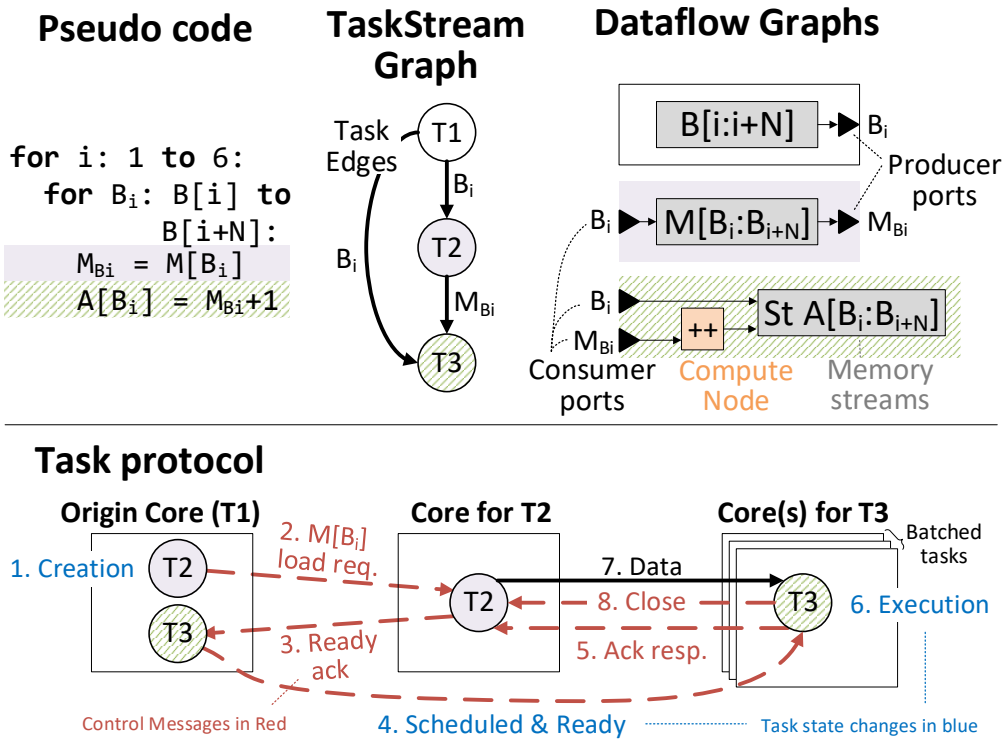


Figure 8.3: TaskStream + Dataflow (T2 state changes omitted.)

this reordering by exploiting the fact that the addition tasks are commutative and can be executed in any order without affecting correctness.

8.2 TaskStream for Reconfigurable Accelerators

Applying TaskStream to a reconfigurable accelerator naturally creates a hierarchical-dataflow representation: one higher-level dataflow of task management and communication, and one lower-level dataflow of instruction execution. We developed this hierarchical representation already in Chapter 7. Here we describe the integrated abstractions on TaskStream, the process of programming, and mapping of evaluated workloads. Finally, we discuss the limitations and possible extensions of our programming model.

8.2.1 Hierarchical TaskStream Dataflow

Figure 8.3 depicts an example program written using hierarchical taskstream dataflow program representation. In our updated task protocol, we add optimizations for sizehint-aware schedule, streaming and batching reuse. We first discuss these optimizations and then describe the techniques to overcome the possible bottlenecks.

Task Protocol Figure 8.3 also shows the task protocol for the example, demonstrating all three task operations: scheduling, streaming and batching. We refer to the figure as we detail the protocol.

Task Protocol – Scheduling: After a task is created, it is scheduled both spatially and temporally (step 1 and 2 in Figure 8.3). For spatial scheduling, TaskStream checks whether any task argument is annotated with `sizehint`, and the task is sent to the core with the least cumulative work until now, and this value is incremented. If no argument is annotated as `sizehint`, round-robin ordering is used. To minimize the response traffic, tasks which only access memory (e.g. *T2* in Figure 8.3) are scheduled differently; they are instead scheduled where the data is located (e.g. by determining the shared cache bank of the start address).

Task instances (identified by their arguments) may either be held in a ready state if all arguments are available, or in a pending state otherwise. For example, in Figure 8.3, the *T2* task is ready after receiving `B[i]`, however the *T3* task will be in the pending state, as it is still waiting on `M[j]`. In the pending case, the producer must provide an explicit "acknowledgment" (ack) of data readiness to trigger the task.

Task Protocol – Streaming: Whenever there is data at the producer port of a task streaming edge, an ack is sent to the child tasks along with the producer port information (Figure 8.3, step 3). This ack should trigger a check as to whether the child task can be concurrently scheduled; this requires that the current task has finished and the consumer port is free. When both conditions are met: the child task is set ready and scheduled (Figure 8.3, step 4), the consumer port is set busy (i.e. it is acquired), and the ack response is sent back

to start streaming (Figure 8.3, step 4, 5). After the last data is sent, the producer sends another ack to close the communication and free the remote port (Figure 8.3, step 8).

Task Protocol – Batching: Batched tasks will be held temporarily, and those identified to have the same `DataID` will be scheduled simultaneously across multiple cores; the responses of batched requests are multicast to all co-scheduled tasks. Batched requests also supply a `bytes` argument that indicates the length of the data to be streamed. In Figure 8.3, step 3, ready acks are sent to all batched tasks, which will then be scheduled on different cores (step 4). Then, after the ack response is received from all cores and the tasks are set for execution (step 5, 6), data will be multicast in step 7, before the communication is closed (step 8).

Deadlock Prevention During streaming inter-task communication, the ports involved must be acquired before data is sent, and held until the stream is complete. This ensures that data for multiple streams is not interleaved. Port acquisition has deadlock hazards, which we describe, along with solutions, as follows:

Self-loop in the TaskStream Graph: Consider the scenario when the parent and child tasks, setup for streaming communication, are scheduled for execution on the same core. The child task may never be able to lock the consumer port if the parent is already using it, and the parent cannot release the producer port until the streaming data is sent to the child, as it is waiting for the child to get lock of the consumer port. Our solution is to allocate a mutually exclusive set of resources/cores to the parent and child tasks. More specifically, in the case that streaming exists within tasks of the same type, tasks can form a dependence chain. We create virtual partitions of the cores to divide the cores into a number of sets equal to a maximum dependence-chain length. A child is always scheduled to a different virtual partition than any of its parents.

Capturing multiple ports for multicast: For multicast, the parent task needs to acquire multiple ports, one for each core. Here we break the possibility of cyclic deadlock by ensuring the ports are acquired in the order of core ID (a unique ID assigned to each core).

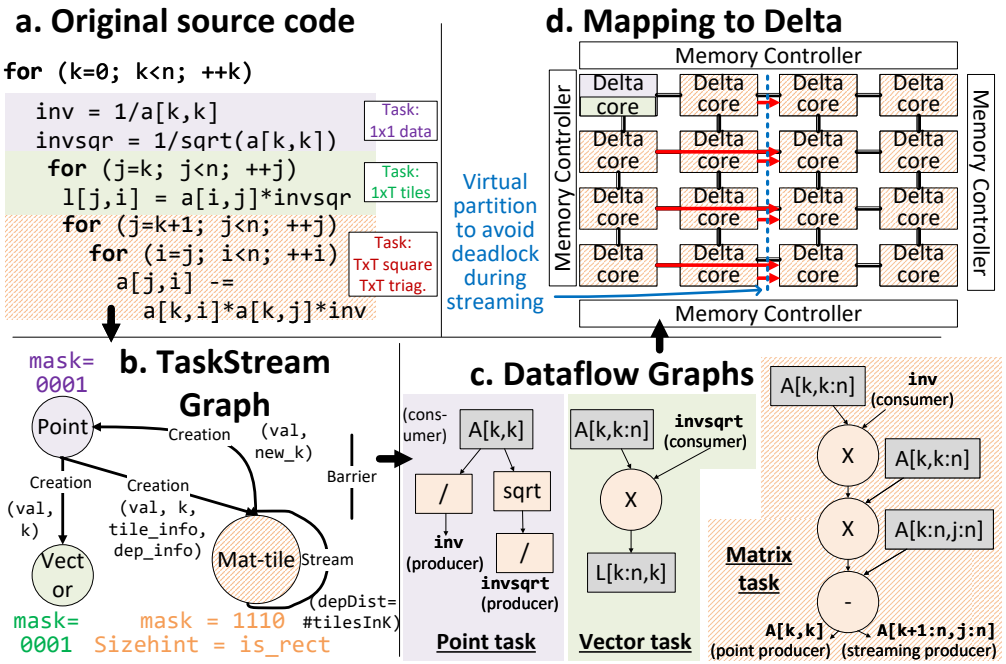


Figure 8.4: Cholesky Implemented in TaskStream (for brevity, only two outer loop iters. run in parallel in one program phase)

8.2.2 Programming

One of the key advantages of programming in TaskStream is that it manages task scheduling at high performance, with only modest programmer help. The process to port a C/C++ workload to TaskStream programming model involves three steps: 1. Defining task types and their functionality, 2. Determining the dependencies among task types to form a taskstream graph, and 3. Managing the start and stop of a program phase. We will discuss each of these in detail below, using Cholesky as an example, as depicted in Figure 8.4.

Defining Task Types When porting a program, a code region should be assigned to a new task type if it performs different computations, the same computations at a different rate, or has different locality behavior. For example, two computations may either be combined into a single task so that the task granularity is higher or may be split if the data associated with two computations are not expected to have similar locality behavior.

For each task type, the programmer first defines the instruction-level dataflow graphs.

Description of Node Property	
coreMask	Bitmask indicating cores a task may be assigned to.
sizehint	Task argument that indicates relative task length.
spatialhint	Task argument indicating a preferred core (for locality)

Table 8.1: Node properties in Task Graph

The programmer assigns certain cores to that task type using the `coreMask`, and may also define a task argument as a `sizehint` or `spatialhint`. The node characteristics are as defined in Table 8.1.

For the example in Figure 8.4a, Cholesky has three task types, one for each loop nesting degree: 1. *Point:* performs only one inverse and square root for every outer loop iteration. 2. *Vector:* performs O(n) multiplication operations. 3. *Matrix:* performs O(n×n) multiplication and subtraction operations. Since the work required for *Point* and *Vector* is much smaller than *Matrix*, the `coreMask` is set to assign one core to both *Point* and *Vector*, while all other cores are assigned to the matrix task. Figure 8.4d shows the mapping of Cholesky to our accelerator, called Delta, which will be explained in the next section.

Task granularity is also a significant choice. Smaller tasks may suffer task management overheads, while larger ones are more difficult to load balance. If the task-size distribution is too wide, even the `size-hint` optimization may not be sufficient. Cholesky is challenging to tile into tasks, because the iteration domain is triangular. Therefore, we split the task into square and triangle tiles (along the diagonal); these are still of relatively different sizes. The new task types are *Point*, *Vector-tile*, and *Matrix-tile*. The tile information is passed as arguments on the *Point*-to-*Matrix* edge.

Defining the TaskStream Graph Next, the programmer uses algorithmic knowledge to identify edges among task type nodes. These include determining whether any task computation is triggering another computation (task creation), whether the tasks have pipelined reuse among them, and whether there is shared read-data among tasks. Another important

Table 8.2: Edge properties in Task Graph

			Description of Edge Property
Program	Exposed	Edge_type	Either Creation, Streaming, or Batching
		Producer & consumer ports	Interfaces for TaskStream I/O
Program	Hints	depDistance	By comparing the distance (port), it identifies a task as parent/child.
		DataID	ID used to batch shared-read data/requests
		Bytes	Either used as size for streaming or meta-data for batching
Hardware	Managed	TaskID	Stores location where a task is buffered
		Ack	Ack buffer maintains TaskIDs of tasks whose ready signal is waiting to be served
		Sched	Stores TaskIDs of child tasks, as identified using depDistance
		SchedParent	Stores TaskIDs of parent tasks, as identified using depDistance

component is to decide the task arguments for a type and then create an edge interface from producer ports at the source task to consumer ports at the destination task. The list of supported edge characteristics is defined in Table 8.2; we list hardware managed aspects as well, to make it clear what the programmer needs to reason about.

In the Cholesky example in Figure 8.4c, there is a creation edge from *Point* to *Vector* and *Matrix-tile*. There exists data dependencies among multiple matrix tasks, hence there is a streaming edge from the *Matrix-tile* task to itself. The dependence distance is the number of matrix-tiles in the $\texttt{k}^{\texttt{th}}$ iteration of the outermost loop.

Finally, we need to limit the number of recursive tasks created by the self-loop in *Matrix-tile* task – this is done by setting the maximum dependence-chain length, determined based on hardware resource limitations (the sensitivity to this length is studied in Section 8.5). The program phase will end, shown as "barrier" in the figure, after all dynamic tasks are

complete.

Managing a Program Phase A program phase starts when the programmer pushes an explicit task of any type. The phase is complete when all tasks have finished execution. Cholesky is initiated by creating a task for the outer-loop point task.

8.2.3 Workload Mapping

Here we discuss how we implemented each of the four additional evaluated workloads. Figure 8.5 shows examples (with only a `coreMask` for 4 cores, for simplicity).

k-nearest neighbors Figure 8.5a shows the TaskStream graph for kNN search. For every query, a binary kd-tree is searched. When the leaf node is reached, data associated with the leaf is accessed to perform linear search (similar to Tigris [261]). We define two task types: 1. *Tree node:* A tree traversal. Since each traversal will incur long latencies to access pointers, we split it into small tasks that compares with the current node and outputs the next tree node. 2. *Leaf search:* Here the query is searched linearly in a long vector associated with the leaf. Many queries may search in the same leaf, generating coarse-grained reuse. Hence, we separate the leaf load task and add a Batch+streaming edge to the leaf search task.

Graph Convolution Network (GCN) Figure 8.5b shows the TaskStream graph for GCN. Every vertex accumulates its feature vectors into its outgoing neighbors, and when all the incoming feature vectors are received, the accumulated vector is multiplied with a weight matrix. To enable flexible load distribution, we define three task types: graph access, feature vector updates (together performing aggregation) and matrix-vector multiplication. As updates involve irregular accesses, we use *spatialhint* based scheduling for atomic updates to ensure that remote accesses are minimized. For accesses to the vertices with varying degrees, we use `sizehint` based scheduling. For matrix-vector multiplication, different graph

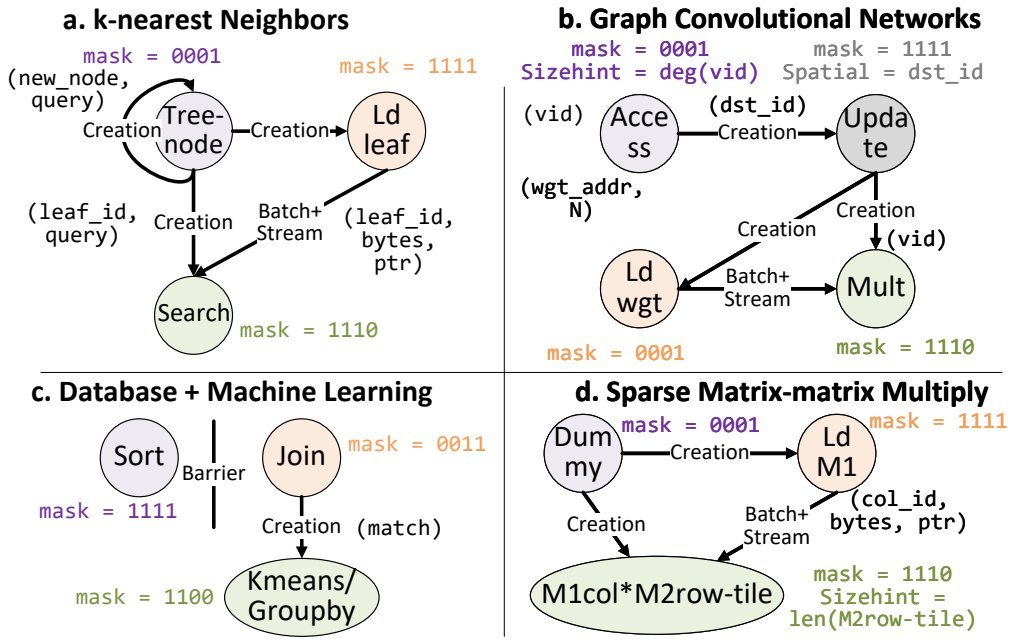


Figure 8.5: TaskStream Graphs for Evaluated Workloads

vertices access the common weight matrix, creating an opportunity for batching reuse (shown by the weight load task and the Batch+Streaming edge). We store the weight matrix in the private scratchpad, and multicast from there.

Database + Machine Learning Figure 8.5c shows the TaskStream for a Database/Machine Learning kernel from Gorgon [239], specifically query 2. We define three task types: 1. *Sort:* Sort requires sufficient work to utilize all resources. Moreover, it reuses its outputs as inputs to the next iteration of sort, and therefore, its output cannot form pipelined communication with other dependent task types. Hence, we treat the subsequent computation as another phase, executed after a task barrier. 2. *Join:* requires O(n) comparison operations. 3. *kmeans-groupby:* requires O(#matched-rows*d) operations to find the minimum distance. Since the work required among *Join* and *kmeans-groupby* may be highly different depending on the number of matched rows and the number of dimensions, we assign 2 cores to *Join* and the rest to the *kmeans-groupby* task type. These tasks are connected by the creation edge.

Sparse matrix multiply Figure 8.5d shows the TaskStream for sparse matrix-sparse matrix multiply. Here we use a tiled outer product implementation (similar to SCNN [179]). We define a task type as the product of a column of matrix 1 and a tile of matrix 2's corresponding row. Since different tiles of matrix 2's row access a common column, there is an opportunity of batching reuse (shown by "Ld M1" and the Batch+Streaming edge). Moreover, we store copies of matrix 2's row and partial sums in the private scratchpad, and use *spatialhint* based scheduling to ensure that a task is scheduled where its data is stored.

8.2.4 Discussion of Limitations and Extensions

Our Programming Experience After identifying tasks and dependences, writing TaskStream code is not overly complex. For programming TaskStream, we use a unified graph domain-specific language for both the TaskStream and dataflow graphs. For reference, tiled-Cholesky on a static-parallel accelerator (REVEL [253]) is 163 lines of code, and the TaskStream version is 210 lines.

Adapting Task-parallel Programs While we focus on specialized implementations, it is somewhat straightforward to adapt programs from languages with fork-join parallelism like Cilk [34]. The essential idea is that whenever a child synchronizes with a parent task using backward dependence in Cilk, in TaskStream, the parent-task can create a successor task, and the child tasks will now have a forward dependence to the successor task. This is possible because TaskStream allows *pending* tasks, where tasks are waiting on arguments from dependent tasks. The child tasks can communicate with the successor task using `TaskID`. Because this is implemented in hardware, it puts a limit on the number of in-flight tasks. In addition, various dataflow-inspired programming models [229, 207, 257, 124, 92] also rely on static inter-task dependencies. These could be a natural fit for expressing TaskStream programs.

A Hypothetical Compiler It is future work to automate the intuition above to construct a high-level-language compiler. In addition to the above, such a compiler would need to identify some program structure to apply the optimizations we consider: The `sizehint` could be determined by estimating loop trip counts and instruction counts. The `coreMask` can be determined by balancing load, based on the relative ratio of average per-task work, and this could be calculated similarly. For structure-recovery edges with simple nested-loop programs, loop dependence analysis could be sufficient (e.g. Cholesky). Workloads with dynamic dependencies, like sparse-matrix-multiply and kNN, may require programmer help (e.g. a loop annotation indicating dependencies).

8.3 Delta: A TaskStream Accelerator

Delta is our proposed multicore accelerator, which implements the TaskStream execution model. Delta tiles are interconnected with a mesh-based NoC, and Figure 8.6 overviews a single tile.

The computation unit is a coarse-grained reconfigurable array, connected via hardware ports (vector ports). The stream controller generates memory requests from stream access patterns, and the responses are sent to the port interface for further communication.

The novel aspect of the hardware is for task management, particularly the task-creation unit (used for storing arguments for ready and pending tasks), and the task-batching unit (used for detecting and scheduling tasks that read the same data). We next describe the design and operation of these components.

Memory Hierarchy Each core has a small private scratchpad, and all cores share access to a shared, distributed on-chip cache. To explain the rationale for this design, we consider the three predominant forms of reuse for memory access: 1. Small, read-only data shared among tasks: This data should be cached on-chip during the entire algorithm. Therefore, each core has a small private scratchpad. 2. Shared data across a subset of tasks: Here

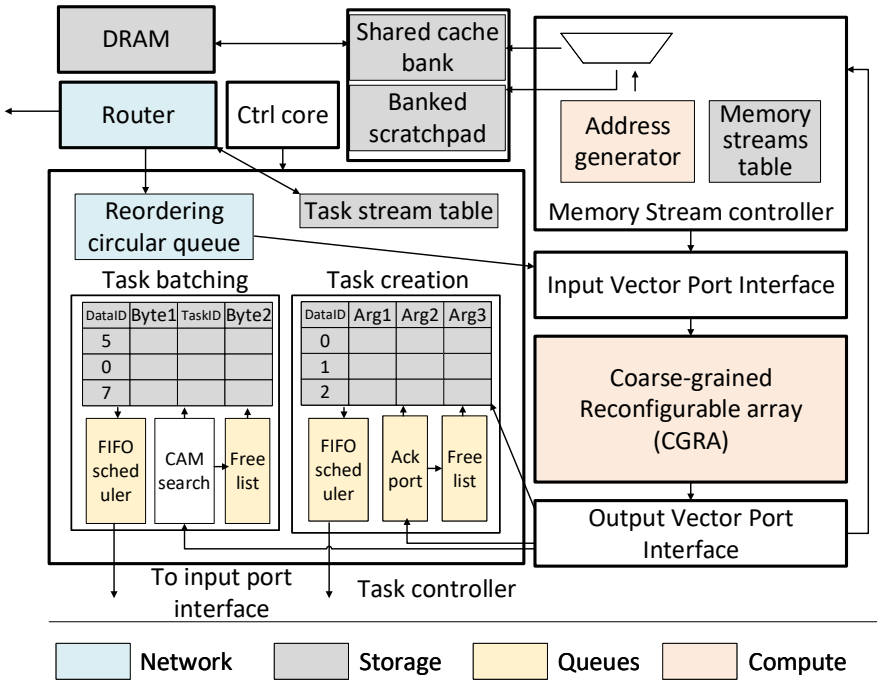


Figure 8.6: Single Tile of Delta Accelerator

the use of scratchpad would require software coherence, and would be difficult to manage. Therefore, Delta has a shared cache, and the reuse across tasks is exploited using task batching. 3. Streaming data with no reuse: This data can bypass the cache hierarchy and will be directly streamed from memory.

Memory Stream Controller The memory stream controller is designed to generate memory addresses using the access patterns and size determined by task type and arguments. Along with an address generator, the memory stream controller also has a stream table that holds their running state. For each stream in the task dataflow program, we reserve some number of entries in this stream table so that forward progress can always be guaranteed (otherwise streams for a single port could fully occupy the table, and streams for another port could not be scheduled).

Task Creation Unit This unit holds tasks until all of their arguments have been received. This unit includes a free list, task argument buffer, dedicated acknowledgment buffer, and

a FIFO scheduler (see Figure 8.6). The task argument buffer is a simple SRAM memory that stores task arguments sequentially, and the free list queue maintains free entries in this buffer.

When data at producer ports is available, if the free list has space, the arguments are pushed in the task creation buffer, and a unique `TaskID` is assigned (using current core and task buffer location information). When a task is ready (i.e. does not require an explicit acknowledgment or already has one), the address location of the current task (`TaskID`) is pushed to the FIFO scheduler. When all consumer ports have sufficient space, the FIFO scheduler will release the task arguments to the input ports in the given order. At that time, the resulting entry will be pushed into the free list.

Task Batch Buffer This unit enables dynamic batching of tasks. Similarly to the creation unit, this unit also has a free list, batching buffer, a small CAM and a FIFO scheduler. The batching buffer is a banked SRAM memory; the difference here is that the free list maintains `TaskIDs` of a "batch" of task arguments instead of just one, as in the task creation unit. Also, here all the task arguments are annotated, which includes `bytes` and `TaskID` of the dependent tasks, as shown in Figure 8.6. When the data at the producer ports is ready, the CAM is searched for the current `DataID` entry. If the entry is found, the new task arguments are stored at this `DataID`'s next empty entry. Otherwise, a new entry is popped from the free list for this purpose. The CAM is filled whenever a new entry is assigned to the task batching buffer, and is cleared whenever a task batch is full or served by the data response. The usage of free list and FIFO scheduler is similar to the task creation unit.

Task Stream Table This table maintains the streams associated with TaskStream edges. These include the streams that transfer data: 1. from producer ports to the corresponding task creation/batching buffer, 2. from task creation/batching buffer to corresponding consumer ports, 3. Any streams for streaming data to remote cores.

Workload	Dataset-size	Workload Parameters
kNN	Queries=512 Queries=1024 Queries=2048	kd-tree-depth=8 leaf-size=2048
SpMM	M1=M2=512x512 M1=M2=1k*1k M1=M2=4k*4k	density=0.10
DB-ML	T1=T2=10M T1=T2=15M	join=0.10
Cholesky	N=128 N=256	tile-size=32
GCN	cora (V=2708, E=10556) citeseer (V=3243, E=4536)	feat-len=64

Table 8.3: Datasets Used in this Work

Communication Protocol For managing streaming communication with low network overhead, we use a coarse-grained credit-based flow control. The consumer core sends credits to the producer core when some number of entries become free in the consumer port (we find 8 keeps traffic low).

Also, during streaming communication, handshaking messages are exchanged to schedule the parent and child tasks. Here, we need to reorder messages so that the correct parents are matched to the correct children, which we accomplish using the "reordering circular queue". Space in this queue is allocated when a message is sent. At responses, the router messages are passed via this queue, then finally pushed to the vector port interface in order. Finally, we use a separate virtual channel for inter-accelerator messages to avoid deadlock. Round-robin scheduling is used for fairness.

Characteristics	Value
Cores	16
FP Units	1024
Task buffer entries	16 64-byte
Memory bndwidth	256 GB/s
Shared Cache	2 MB
Private Scratch	256 kB
Network	64-byte mesh

Table 8.4: Architecture Parameters

8.4 Methodology

Delta Power/Area We implemented Delta's CGRA by extending the Chisel-based DSAGEN [251, 161] framework. Components were synthesized using a 28nm UMC library. We use Cacti 7.0 [156] for estimating the overhead of the SRAM buffers and CAM within the task creation and batching units.

Baseline Architectures For reference, we compare against a 24-core SKL CPU running optimized libraries: MKL [2] for Cholesky and SpMSpM, and MADLib [1] for DB-ML. For kNN, we use the popular FLANN [154] library. For GCN, we use the PyG library [3].

We developed a simulator for Delta and integrated it with gem5 [32, 200, 226], using a RISCV ISA [22] for the control core. For accelerator comparison points, we evaluated three designs:

- **Static Parallel**: Work is partitioned statically to each core by the programmer, and data is tiled into each core's scratchpad.
- **Delta+OnlyTasks**: In this version, Delta tasks are scheduled dynamically in hardware using size-hint scheduling policy.

Wkld	Knn	Cholesky	DB-ML	Spmspm	GCN	GM
Speedup	43.2	6.9	1729.2	65.3	105.4	81.3

Table 8.5: Speedup over 24-core SKL CPU

- **Delta-TaskStream**: This includes both load balance and locality/structure-recovery optimizations within TaskStream.

Datasets We use synthetic datasets with varying sizes and natural skewness; GCN uses popular real-world graphs. Table 8.3 shows the dataset sizes and workload parameters.

Parameters Table 8.4 shows the common hardware parameters of Delta and baselines.

8.5 Evaluation

Broadly, the goal of our evaluation is to analyze whether our Delta proposal is able to recover locality structure while retaining the benefits of task parallelism. First we compare against a CPU and an equivalent static parallel design. Then we give insight into performance benefits with stream recovery (i.e. inter-task streaming/batching) by examining network traffic and core utilization over time. Then, we explore sensitivity to the task scheduling strategy and pipelining depth. We conclude by discussing area overheads.

Overall Performance First, we validate that Delta provides accelerator-like performance over existing multicore CPUs. Table 8.5 below shows the speedup over a 24-core baseline CPU. `Cholesky` has the least speedup, as Delta exhausts all the parallelism in this workload at the evaluated array size. The other workloads have more data-parallelism: `kNN` during linear search, `DB-ML` for performing kmeans search on many data points, `spmspm` and `GCN` for matrix multiply. Thus, they achieve higher speedup, as the data-parallel resources of the accelerator can be fully realized.

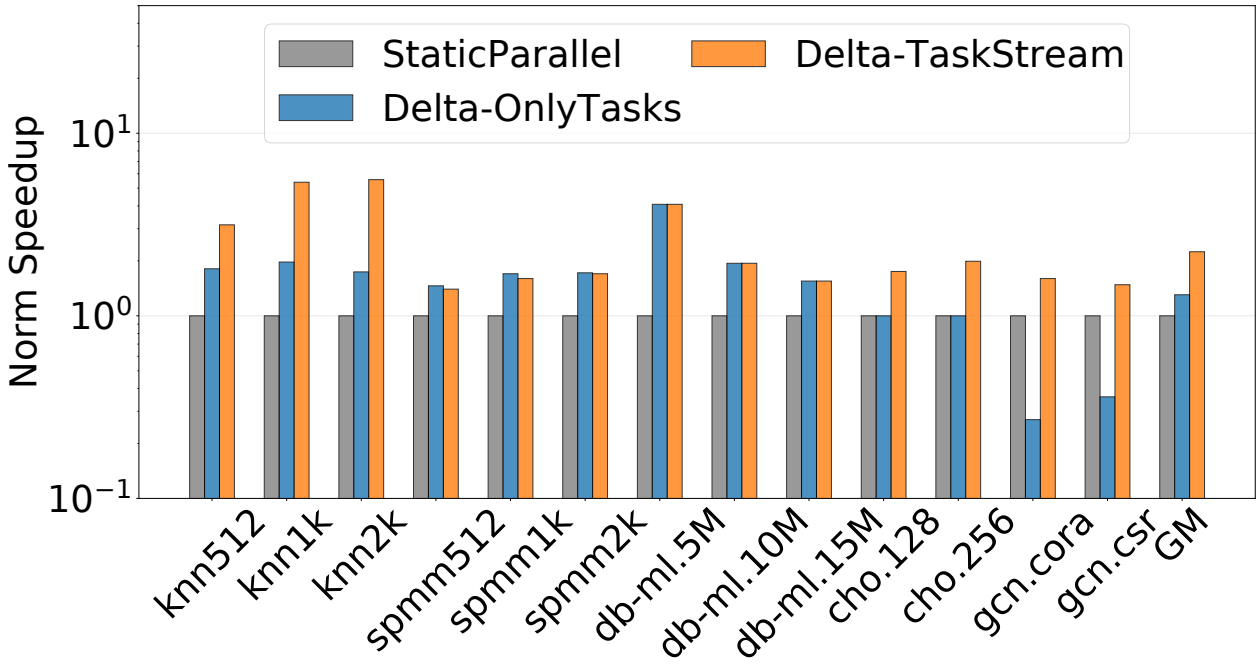


Figure 8.7: Overall Performance Comparison

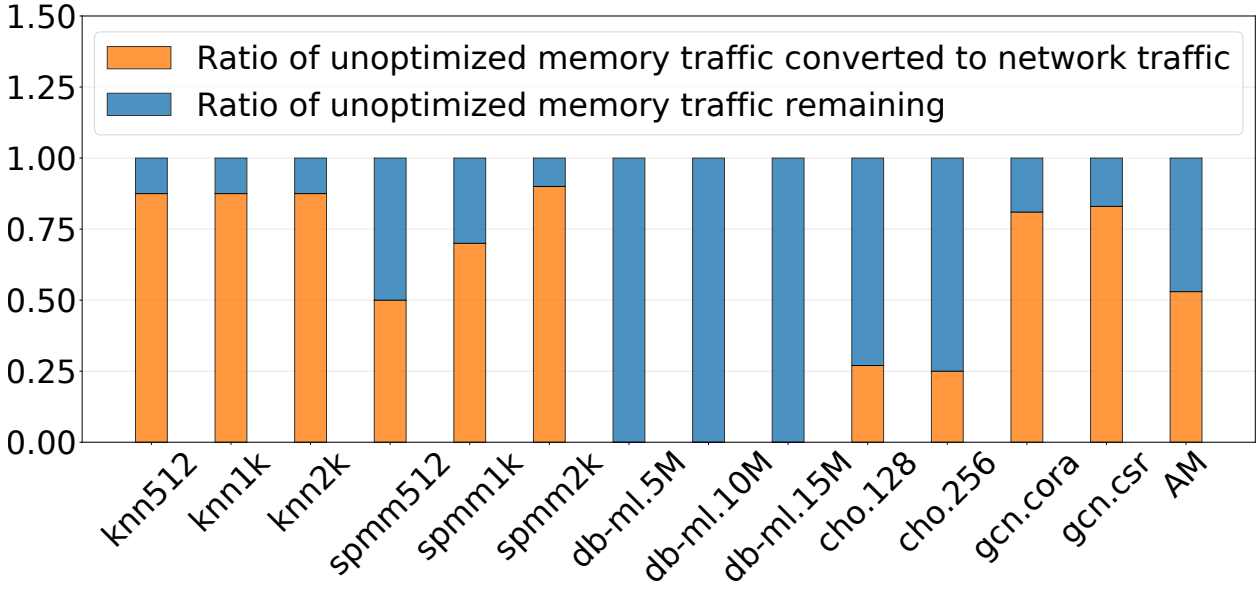


Figure 8.8: Traffic-breakdown with Stream Recovery

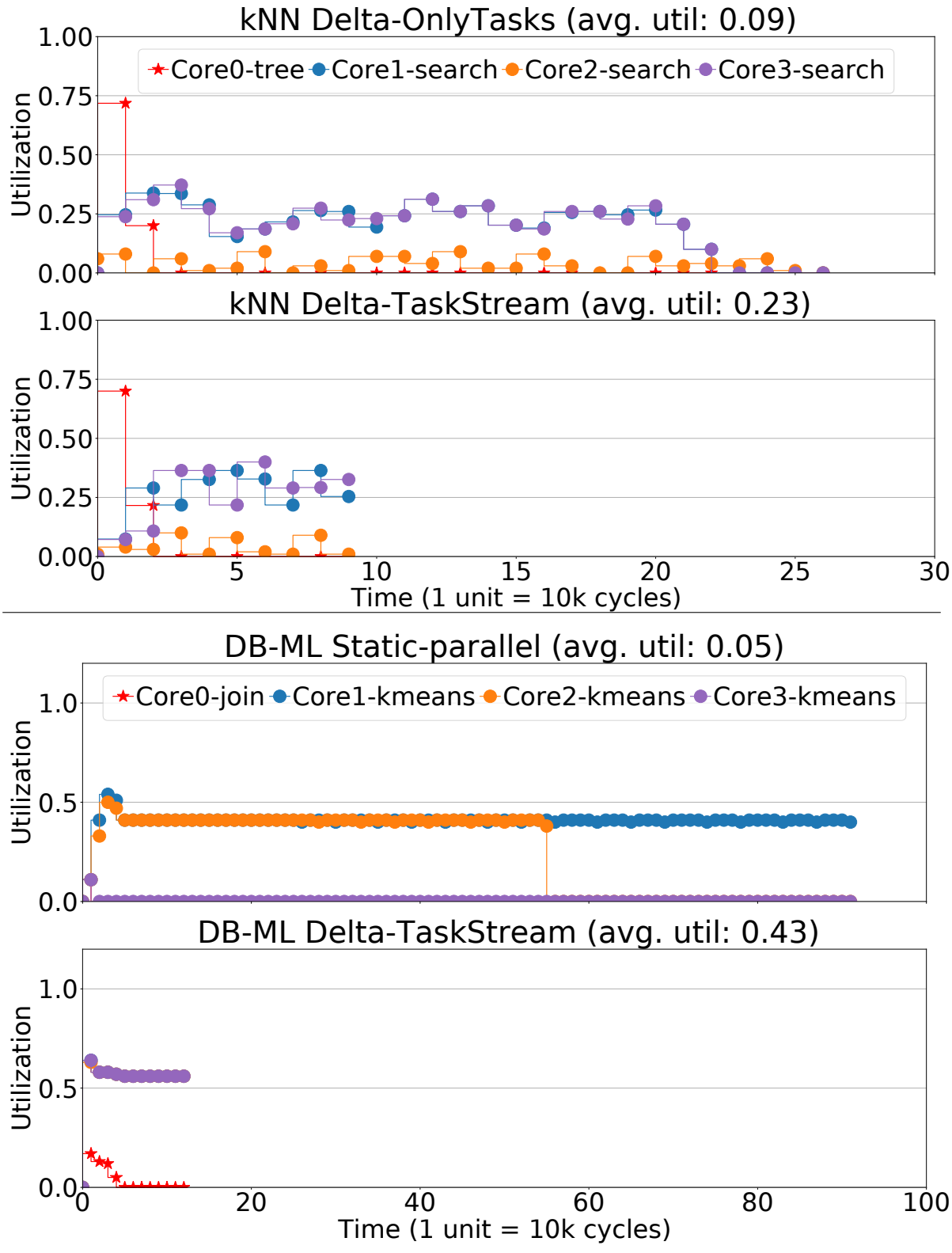


Figure 8.9: Utilization Comparison with Stream Recovery.

Benefit of Tasks Figure 8.7 shows that *Delta-OnlyTasks* achieves 1.3× speedup against the static reconfigurable accelerator that represents the state-of-the-art. *Delta-OnlyTasks* improve speedup for kNN and DB-ML by enabling better distribution of work irrespective of where the kmean's input (or leaf in kNN) is mapped statically, while it would matter in the static parallel implementation. For SpMSpM, the speedup comes from increased parallelism from overlapping execution of task types as data becomes available. Cholesky does not benefit due to low parallelism in both static-parallel and OnlyTasks without the stream-recovery optimization. GCN is an exception where there is a slowdown, as consecutive requests to access the weight matrix will create a hotspot in the network.

Benefit of Structure Recovery Figure 8.7 also shows that with stream-recovery optimizations, the speedup increases to 2.2× over the static-parallel accelerator. SpMSpM does not benefit from stream-recovery, as it is not bottlenecked by communication; in the outer-product implementation, the batched column of the first matrix has high reuse – each element in the column is multiplied with each of the elements in the second matrix's corresponding row. However, kNN and GCN have less reuse, and batching tasks can reduce memory traffic by nearly an order of magnitude. Cholesky benefits from explicit communication among dependent tasks, which alleviates the overheads of shared memory synchronization. DB-ML does not have opportunity for stream recovery.

Memory Traffic Reduction To explain the source of performance improvement using stream recovery, Figure 8.8 demonstrates what percentage of memory traffic is converted into network traffic in *Delta-TaskStream*. In most cases, more than 50% of the memory traffic is converted, with even fewer packets due to multicast.

Fine-grained Core-wise Throughput Comparison TaskStream optimizations improve the core utilization[2] by alleviating communication bottlenecks, while providing load balance.

[2]Utilization is defined as the percentage of functional units (FUs) fired in each cycle.

We give insight into its capabilities by showing the per-workload utilization over time in Figures 8.9/8.11 and discuss below. Note that we only plot for a subset of cores for clarity; also, core 0 is used for lower-rate (or less compute intensive) tasks, therefore is generally under-utilized. The title shows the average utilization across *all* cores.

- **kNN:** Tasks enable overlap of the tree search tasks, improving utilization of Core 0, which is executing tree tasks. With stream batching optimization, the accesses to the leaf data are batched and multicast to co-scheduled tasks. This increases effective bandwidth and improves utilization (see Core 1, 2, 3).

- **DB-ML:** Also in Figure 8.9, the legend lists the task-to-core mapping in our implementation. Sort is performed in a separate phase for both implementations, hence we omit from this figure. Stream recovery opportunities do not exist in DB-ML, hence we compare static parallel and *Delta-OnlyTasks*. The *Delta* implementation dynamically distributes the "kmeans" tasks that are created by "join" and hence is able to balance load much better (see how static-parallel Core 1 and 2 are heavily under-utilized in some phases).

- **Cholesky:** For Cholesky in Figure 8.10, we show 12 outer-loop iterations (1 phase in TaskStream). In *Delta-OnlyTasks*, only one outer-loop iteration can be performed at a time due to inter-loop dependencies. This both reduces the available parallelism and introduces barrier overheads. *Delta* allows many more parallel tasks with the support for chained streaming. It further ensures balanced execution by distributing load based on `sizehint`.

- **Sparse matrix-multiply:** Now referring to Figure 8.11, due to the outer-product implementation of sparse matrix-multiply, there is high reuse available. Therefore, the utilization of task-parallel versions are nearly ideal, and there is not much potential left for task batching.

- **Graph Convolution Networks:** Stream-recovery significantly improves the through-

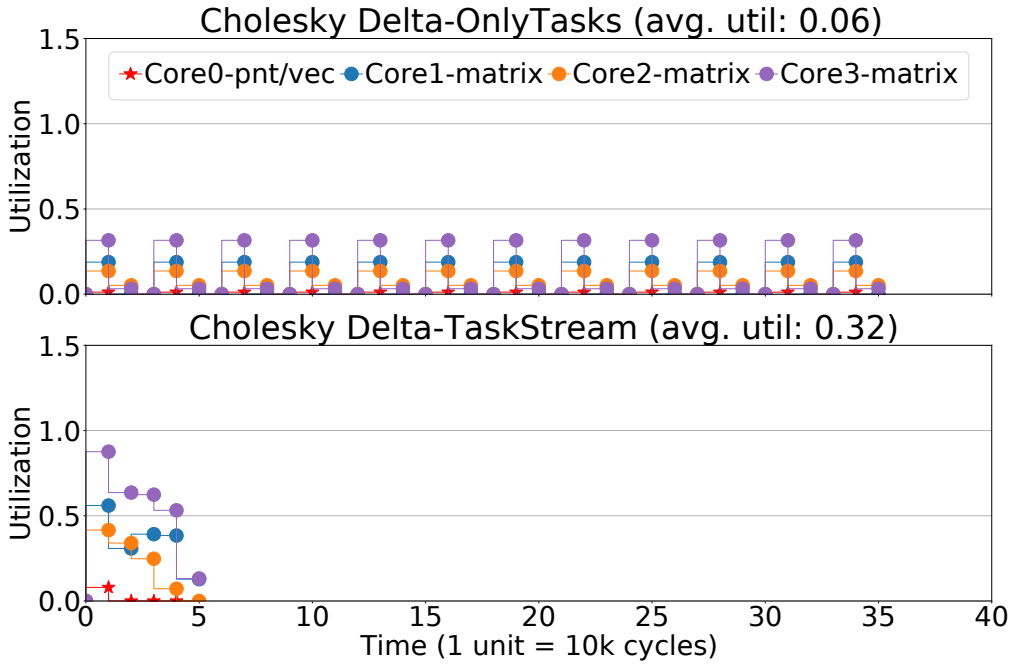


Figure 8.10: Utilization Comparison with Stream Recovery for Cholesky.

put of the matrix-multiply cores, as batched reads of the shared weight matrix enables higher effective network bandwidth. Since 13/16 cores work on the matrix-multiply, this improves the average utilization by 6.6×.

Comparing Task Scheduling Strategies Figure 8.12 compares four scheduling policies on *Delta*: round-robin, random, `spatialhint`, and `sizehint`. For *kNN*, `spatialhint` schedules tasks with the same leaf_id to the same core, so multicasting becomes irrelevant. For *Cholesky*, `sizehint` gains 1.63× performance over the baseline round-robin due to distributing square and triangular tiles better. For *DB-ML*, kmeans on all data items takes a similar amount of time, so load balance is less important, except that `spatialhint` restricts scheduling, thereby aggravating load imbalance. For *SpMSpM*, only `spatialhint` applies, as we are using an outer product, where partial sums are maintained in scratchpads, and we only allow tiles to be scheduled near their partial sums. This is because scheduling elsewhere would introduce excess remote fine-grained atomic update traffic, hurting performance. In *GCN*, the scheduling policy affects the vertex-access ordering. `sizehint` can balance load

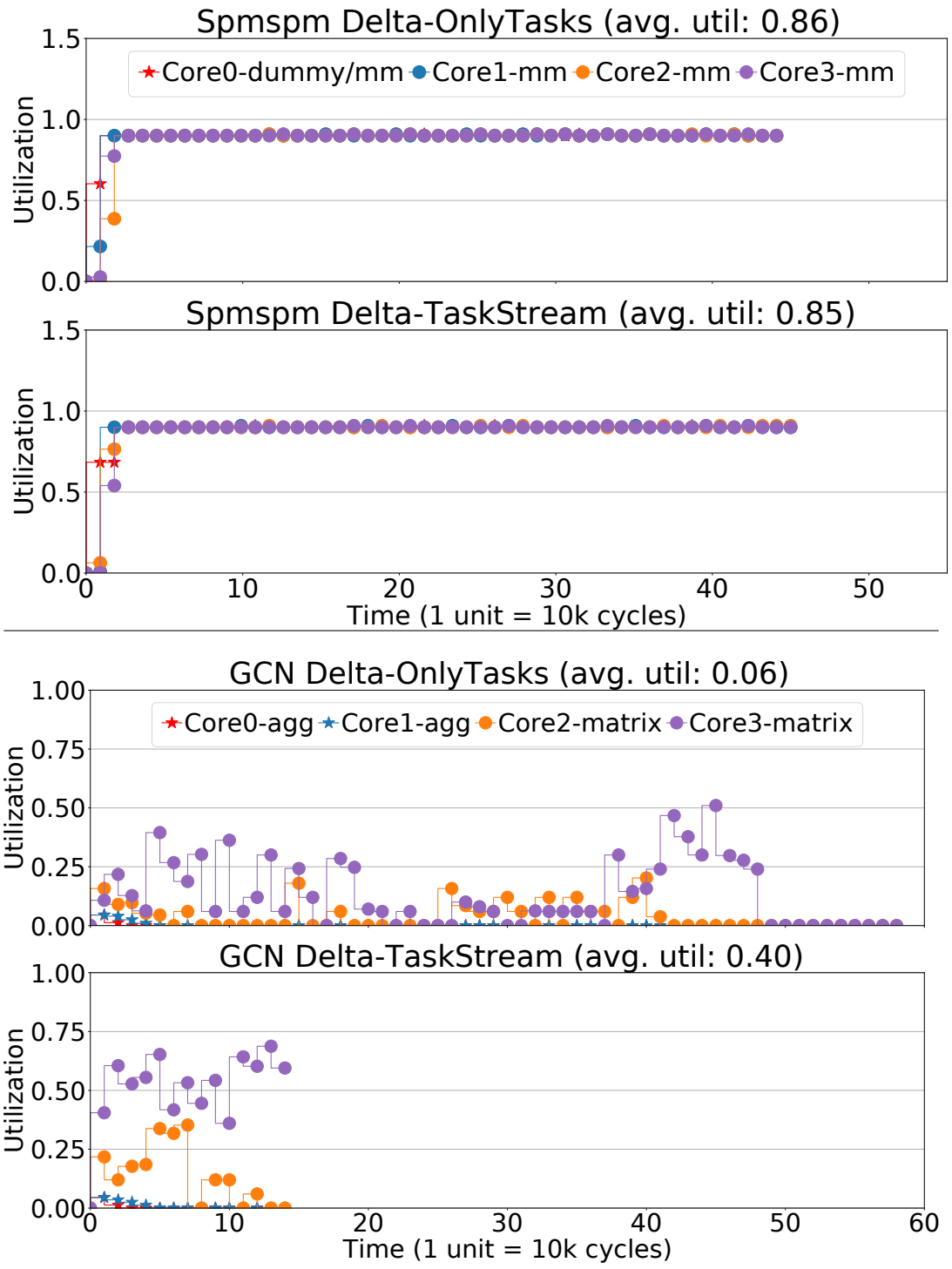


Figure 8.11: Utilization Comparison with Stream Recovery

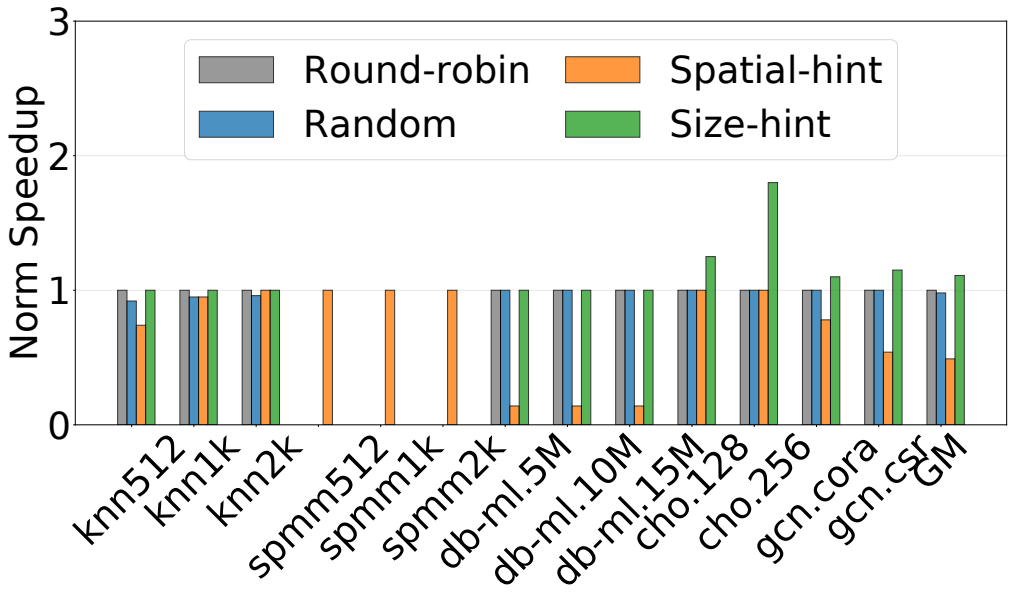


Figure 8.12: Sensitivity to Load Balancing Strategies

better by intelligently distributing vertices with varying degree. In conclusion, `sizehint` consistently outperforms the simpler random and round-robin policies. `Spatialhint` does not benefit much, as it restricts scheduling for load balancing without surpassing the locality benefits of stream recovery.

Sensitivity to the Depth of Pipeline Parallelism TaskStream gives control to programmers on how to expand different facets of parallelism. We elucidate with an experiment on Cholesky, by varying the depth of streaming dependence chain (i.e. how many tasks are allowed to chain), and the results are below in Table 8.6. Higher depth improves available parallelism at the cost of extra streaming network traffic, and the optimal point for these array sizes occurs at a depth of 12. At higher depths, latency stalls to set up the pipeline streaming communication are more compared to the benefits of improved concurrency. One interesting finding is the pathological case at a depth of 10; performance drops because on our 16-core 4x4 mesh, around half of the cores are sending data to the next half, causing much of the communication to be on a few bisecting links. Detecting and mitigating this is future work.

Table 8.6: Sensitivity to Dependence Chain Depth

depth	1	2	6	8	10	12	14
Cholesky-256	1	1.8	3.1	5.6	3.6	7.7	7.5
Cholesky-128	1	2.0	2.4	2.6	3.1	3.6	3.4

Area Overhead Table 8.7 shows the area breakdown of Delta, within and across cores. The only additional components required over the static architecture are the task management units, which consume 3.6% of the total on chip area.

Table 8.7: Area and Power breakdown for Delta (28nm)

	Area (mm^2)	**Power (mW)**
Control Cores	0.053	11.5
Task Creation	0.01	4.76
Task Batching	0.033	10
scratchpad+ctrl	0.08	11.2
CGRA (4x5)	0.21	80
1 Delta-Core	0.386	117.46
4x4 64 byte mesh (1)	0.2	44.7
Shared cache	12.39	2280
Delta Total	18.77	4203.7

8.6 Related Work

Table 8.8 compares Delta and prior works based on three critical factors: 1. *Sched-flex*: Flexibility to schedule work spatially across cores, which can help balance locality and load. 2. *Mem-sched*: Ability to exploit shared read-reuse among tasks. 3. *Inter-task-comm*: Write-read dependencies among tasks may be resolved via shared memory synchronization

or explicit communication. No prior work simultaneously supports hardware task scheduling flexibility and memory scheduling, and none supports read-reuse with task-based parallelism. The remainder of this section discusses in further detail.

Locality in Task-Runtimes An inspiration for our work is the study of locality-enhancing techniques for software-only threading and task-parallel systems [44, 13, 217, 267, 91, 141]. A prevailing mechanism is work-stealing [13, 91, 267, 141, 204], which lends itself to locality by keeping parent and child tasks together. This is effective in divide-and-conquer algorithms, but does not guarantee locality in general, and task stealing may incur non-negligible latency on the critical path (especially when ported to accelerated systems).

Of particular note are techniques which recover program structure. One approach is to annotate tasks with the dataset they may access, and use this to schedule tasks near-data. This has been explored in task-parallel languages [41], speculative parallel models [109, 266], and partitioned global address space (PGAS) systems [105]. Another approach is Splicing [140], which is a compiler optimization for recursive task-parallel programs that interleaves tasks with locality to optimize for locality.

Accelerating Task Parallel Codes Task management is a significant overhead that has been mitigated by hardware based [126, 271], or hardware assisted schedulers [206]. Anton2 [215] uses a hardware-assisted task runtime for geometry processing.

Prior work has also explored accelerating task parallel workloads in reconfigurable hardware. TAPAS [148] is a high level synthesis (HLS) toolchain that leverages the Tapir IR [208] to create application-specific hardware for task-parallel workloads. μIR [212] is a hardware design IR, also useful in the context of HLS, that supports task-parallel constructs. ParallelXL [45] is a framework that enables building custom hardware accelerators using task-parallel execution and work stealing. Chronos [11] is a framework to build task-parallel accelerators for applications with speculative parallelism. Three recent works address flexible parallelism for reconfigurable accelerators: Aurochs [238] proposes a threading model

		Sched-flex	Mem-sched	Inter-task-comm
S/w	Splicing* [140]	High	Read-reuse	Memory
	Gramps* [205]	High		Memory
Accel-assisted CPU	Stream-floating* [249]	High	Read-reuse	Memory
	Carbon [126]	High		Memory
	ADM [206]	High		Memory
	Minnow [271]	High		Memory
	Spatial-Hints [109]	Low	Near-data	Memory
Reconfigurable Accelerators	ParallelXL [45]	High		Memory
	Centaur [172]	High		Explicit
	Plasticine* [188]	Restricted	Read-reuse	Explicit
	PolyGraph [63]	Restricted	Near-data	Memory
	Aurochs [238]	High		Memory
	Fifer [158]	Restricted	Near-data	Memory
	Delta (ours)	**Medium**	**Read-reuse**	**Explicit**

Table 8.8: Related Work Comparison (* uses traditional threads for parallelism; no hardware support for tasks.)

for reconfigurable dataflow architectures. PolyGraph [63] similarly adds an integrated task/-dataflow model for a reconfigurable accelerator. Fifer [158] temporally reorders fine-grained tasks for load balancing. TaskStream's structure-recovery optimizations can be applied to any of the above systems.

Exploiting & Recovering Streaming Structure In the context of general-purpose processing, stream floating [249] has a "confluence" optimization that dynamically combines prefetching streams from different cores (confluence). Near-stream computing [248] can pipeline data between tasks offloaded to the last-level cache. Several prior reconfigurable accelerators [253, 66, 276] have primitives for multicast, but they are suitable for only for regular programs.

Prior domain-specific accelerators *can* exploit batching and pipelining in irregular pro-

	No batch	Batch + no multicast	Batch + multicast
Small cache	1×	1.5×	6×
Large cache	3×	6.5×	6×

Table 8.9: Speedups with Batch and Multicast Optimizations in kNN Normalized to Small Cache and No Batch Case

grams, but the expected structure is baked into the hardware. One example is multicasting in sparse matrix/DNN accelerators [98, 179, 49, 191, 263, 136, 272]. Other specialized architectures perform load balance optimizations while exploiting various forms of reuse in hardware, like SparTen [85], BARISTA [86] and GraphDynS [262]. MTP [36] hides memory latency for database selections by using concurrent fine-grained accelerator threads. Centaur [172] exploits streaming locality in databases by combining multiple pipelined SQL operators.

In summary, our work is the first to demonstrate techniques for exploiting specializable communication in general task-parallel accelerators.

8.7 Discussion

In Delta, we exploited inter-task communication primitives to recover the lost locality in task-parallel workloads. These optimizations are highly effective on a broad set of workloads. However, they present tradeoffs for specific input and workload properties. Below we discuss three such scenarios:

Flexibility for Spatial Multicast and Temporal Cache Reuse Delta exploits coarse-grained reuse using two steps: It reorders tasks such that tasks accessing the same data are together (i.e., batched) and simultaneously schedules these tasks to exploit spatial reuse by multicasting data. The choice of spatial multicast negates the need to buffer data while introducing the setup latency to coordinate tasks for multicast. To explain, see the experiment in Table 8.9 for kNN: large private caches with coherence outperform the multicast optimiza-

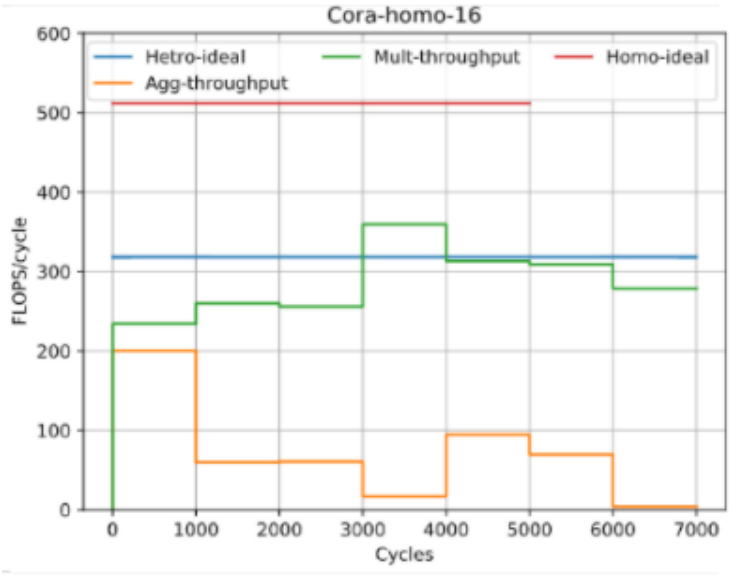


Figure 8.13: Potential Benefits of Dynamic Reconfiguration in GCN

tion by avoiding setup latency and exploiting reuse using on-chip buffers and replication. In conclusion, temporal reuse performs better for data that fits in the on-chip memory.

Dynamic Update of Task Type Distribution Our current implementation of TaskStream needs programmers to specify the task type distribution statically, using simple complexity analysis. However, in some algorithms, the ratio of work required among task types may change during the execution of a program. Figure 8.13 plots the FLOPS/cycle for GCN phases on Delta. We assume that 50% of the cores are assigned to graph aggregation (`Agg-throughput` shows their average throughput) and the rest 50% work on matrix-multiplication (shown by `Mult-throughput`). In initial phases, aggregation and multiplication cores require similar throughput, but matrix multiply becomes dominant when all vertices are aggregated. If we could assign *all* cores to matrix-multiply in later program phases, we can achieve up to 50% performance gains. The challenge would be to minimize switching overheads (e.g., reconfiguration and task redistribution). Future work should also explore switching heuristics that are broadly applicable.

	knn	spmm	DB.ML	Cho	GCN
Lazy-binding	0.72	0.8	1.00	0.61	1.25

Table 8.10: Speedups with Lazy Binding Normalized to TaskStream

Lazily Binding Tasks to Resources TaskStream currently binds a task to the execution core (and the task argument buffer) as soon as the task's initial arguments are available. However, the task's preferred execution order may change later according to when other arguments arrive. Consider a pathological case: there are ten tasks and two cores (all equal-sized). `Sizehint` schedules Tasks 1,3,5,7,9 to core 1 and Tasks 2,4,6,8,10 to core 2. Task batching may later find that Tasks 1,3,5,7,9 all share the same data; however, it will be unable to exploit spatial reuse as tasks are already scheduled at the same Core 1. A solution could be to rebind tasks to other cores, which we call lazy binding. Moreover, to ensure a balanced load at all times during execution, we require tasks to be scheduled when they use resources.

Table 8.10 shows our preliminary results. GCN achieves up to 30% performance gains because it has high degrees of spatial reuse (all vertices access weight matrix) and lazy binding distributes vertices more evenly, resulting in better multicast opportunity. For other workloads, lazy binding incurs latency on the critical path, causing little performance degradation.

CHAPTER 9

Discussion

In this dissertation, we developed a programmable accelerator that can run both regular and irregular workloads efficiently. We start with a baseline accelerator with a systolic-CGRA-based computation substrate that allows configuring the datapath with ASIC-like efficiency and a scratchpad memory for ASIC-like data placement. We support irregularity in this accelerator by enhancing hardware modules for specializable data-dependence forms - in particular, stream-join control, compute-enabled banked memory, and a new task management unit for task distribution and coordinating for inter-task dependencies.

The efficiency and generality of this system raise an interesting question: could this make a better general-purpose processing unit than a GPU? Recent mobile SoCs have a sea of accelerators on their chips [20, 21]. We envision that an irregularity-optimized programmable accelerator can replace most of the occasionally used application-specific accelerators. Lesser accelerators will help maintain fewer software stacks and enable adaptability to future algorithms. Moreover, we may be able to accelerate applications that are currently only suited only to CPUs, like gradient boosting decision trees.

We contribute to the research community by unifying the fragmented irregular acceleration space. Today, accelerators for each irregular workload are designed from scratch; thus, no comparison is possible. However, once we realize how they specialize for irregularity, we can model these in a common framework and integrate features that help with irregularity. These features may be good for the target and other similar domains. In this common framework, it is easier to compare features, and also, one can transfer innovations across domains.

This chapter presents our case for the practical adoption of domain-agnostic programmable accelerators. Then, we discuss how our work contributes to the "sea of accelerators" problem – several accelerators available but no understanding of their fundamental capabilities. Finally, we will give pointers to the future directions and conclude.

9.1 Case for Domain-Agnostic Programmable Accelerators

Today, GPUs and FPGAs provide tradeoffs between the flexibility of CPUs and the efficiency of application-specific accelerators. This dissertation makes a case for our CGRA-based programmable accelerator as a promising alternative as it has performance close to ASICs while providing decent flexibility. Four factors largely determine the efficiency:

1. **Performance:** This dissertation demonstrated that our spatial accelerator remains within 30% of ASIC's performance for deep learning, databases, and graph processing; GPGPUs may be up to 16.7× slower for these workloads. FPGAs lose 3.2× performance due to slower frequency [127].

2. **Area/Power:** The flexibility costs include flexible operand routing in CGRAs and resource over-provisioning for supporting multiple domains. SPU is 1.5× more area compared to SCNN accelerator for sparse convolution networks. SNAFU shows 2-3× energy overheads of programmability [84]. On the other hand, GPUs often consume more than 100× energy than accelerators due to large register files [143].

3. **Programming Ease:** While ASICs typically have simple domain-specific interfaces, programmable accelerators need a more expressive programming interface. Existing CUDA for GPUs and HLS are closer to C, where data chunks are assigned to computations. For example, in CUDA, data is assigned to each threadID, and in HLS, the loop iterations are parallelized across processing elements. While the direct comparison is somewhat subjective, we present a different programming model that is a lower-level and closer to assembly, but there is a path to compiler [252] and programming lan-

guage support [123]. Overall, the performance and area are competitive with ASICs (much better than existing designs), and programmability is good enough to consider replacing some ASICs.

4. **Adaptability:** The right amount of flexibility maximizes the utilization and longevity of an accelerator. However, it is hard to predict the desired amount of flexibility at design time as it depends on how applications and market will evolve [270, 112]. This dissertation presents techniques showing that the programmability costs are reasonable and that the potential longevity benefits can be safe to invest in. Moreover, the features are modular, requiring low efforts to remove/update future generations.

Overall, carefully designed programmable accelerators are an attractive alternative to application-specific accelerators due to minimal overheads and modular features, allowing cherry-picking based on the application. We hope our results encourage the industry to look into programmable accelerators and enable acceleration of more domains, which was not possible earlier due to the high costs of a new accelerator.

9.2 Systematizing Irregular Accelerator Research

Recently, there has been a surge of accelerators for every new application [85, 95, 122], and even for every new algorithmic implementation of these application [94, 173, 11]. Even though looking deep into the applications can lead to exciting innovation, it will be infeasible as the number of applications grows. Therefore, it is high time to systematize the accelerator design research and minimize redundancy. Below discuss the three dimensions of accelerator design we contributed to in this dissertation (Figure 9.1):

Systematically Dealing With Data-Dependencies Irregularity is usually thought of as random/unpredictable, making it hard to specialize. Recent accelerators achieve efficiency by hard coding dependencies in the datapath. Our specializable forms of data dependence

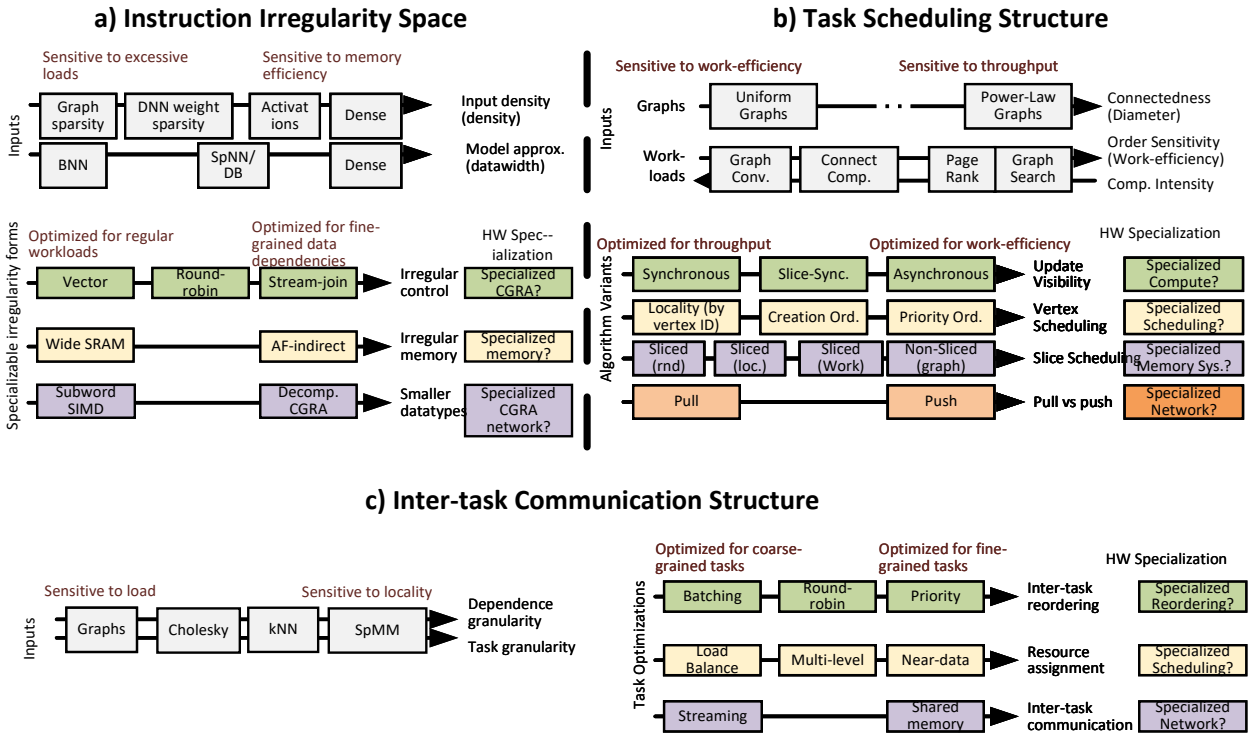


Figure 9.1: Systematizing Irregular Accelerator Research

can be used to classify and understand the fundamental capabilities of domain-specific accelerators [66, 67]. Figure 9.1 shows the data-dependence dimensions and the potential hardware support that may be required for input types (dense/sparse) and workloads (can have data-dependence in multiple dimensions).

Beyond simply understanding the space, this way of viewing the algorithm's interaction with architecture can improve the portability of techniques across domains. Consider the context of accelerators for sparse linear algebra. SPU's design can join sparse lists at one element per cycle (per PE). An idea proposed for a sparse ML accelerator is to vectorize the join [153], so N elements can be joined at once from each list (requiring N×N comparisons). The ExTensor accelerator [103], designed for multidimensional sparse tensor ops, goes further. It demonstrates that hierarchical list-intersection (a form of stream-join) can be more work-efficient by skipping a variable number of unmatched items of the other list in a single step. To further reduce the memory bandwidth overhead of sparsity, Sparten [85] proposed

a bit-vector representation of indices. Thus, the matched indices can be found using efficient bit-level operations. These optimizations can apply to SPU. More importantly, finding specializable opportunities and commonality in the dependence forms across domains makes it clear how to apply these for other superficially different problems. For example, a subsequent work, Sparsecore [194] used our insight that graph pattern mining (GPM) algorithms can be formulated as stream-join and accelerated them using a similar stream-based ISA.

Secondly, we observed that our insight of data-dependence forms can open directions of more general accelerators. For example, we extended our idea of data-dependent control and memory in SPU to look at data-dependent parallelism in PolyGraph and TaskStream. The follow-up works, Capstan [202] and Aurochs [238] also enhanced the Plasticine [188] spatial architecture for data-dependent parallelism. At the other end of the spectrum could be data-dependent datatypes, where at a fine grain, the datatype size is chosen to meet the precision requirements. One could imagine exposing these forms as first-class primitives in the hardware/software interface, and each could be plausibly useful in many domains.

Finally, our data dependence forms are modular and have a one-to-one correspondence to hardware modules. A follow-up work, DSAGEN [251], used these forms as primitives to perform design space exploration for programmable accelerators.

Clearly Understanding the Hardware-Software Codesign Space Accelerator designs make strong assumptions about certain aspects of a domain for simplicity, like input type (e.g., high vs. low diameter graph), workload property (e.g., order resilience, frontier density), and algorithms. Different algorithmic choices result in different accelerators making it challenging to compare accelerators with common denominators [63, 64]. Moreover, many algorithms are left unexplored because it is infeasible to track the large space without systematic understanding.

Our work shows the value of systematizing the codesign process using algorithm variants – Figure 9.1b) shows variants for graph processing. An *algorithm* is a combination of algorithm variant choices. We created a modular execution model and accelerator which can optionally

support hardware features leading to the efficient execution of specific algorithm variants (e.g., caches for non-sliced execution, remote tasks/atomics for push-based algorithms, etc.). This framework enabled us to make strong statements about the effectiveness of various algorithm variants and the value of flexibility across these codesign dimensions without being hampered by differences in evaluation (e.g., simulator & preprocessing assumptions). We show it for graph processing, but fundamental algorithm variants also exist for other workloads, as demonstrated by Maeri [129] and SparseAdapt [175].

Decoupled Abstractions for Coarse-grained Scheduling The network is a critical resource and the scheduling between coarse-grained tasks to optimize for locality are essential. Today, custom compilers are being built that analyze and identify scheduling optimizations specific to their target workloads [132]. We find that there is a standard set of these optimizations (shown in Figure 9.1c)), and the process can be systematized.

Along with workloads with different task types (fine/coarse-grained), our taxonomy is more useful because it can apply to various systems, e.g., AMD's HSA [130], NVIDIA's multi-GPU systems [211]. We achieve this using our TaskStream program representation that makes assumptions only of the communication interface and is neutral to the underlying architecture.

9.3 Future Directions and Open Questions

So far, we have described the architecture of our programmable accelerator; however, more research is required to mature the idea. First, the accelerator should be easier to program. Second, it would be essential that the accelerator applies to practical use cases. For example, real-world graphs may be large and dynamic; thus, the architecture must be scalable and work for dynamic graphs. Future SoCs will likely be heterogeneous; therefore, the inter-task communication support should be broad enough and make minimal assumptions about the internal implementation.

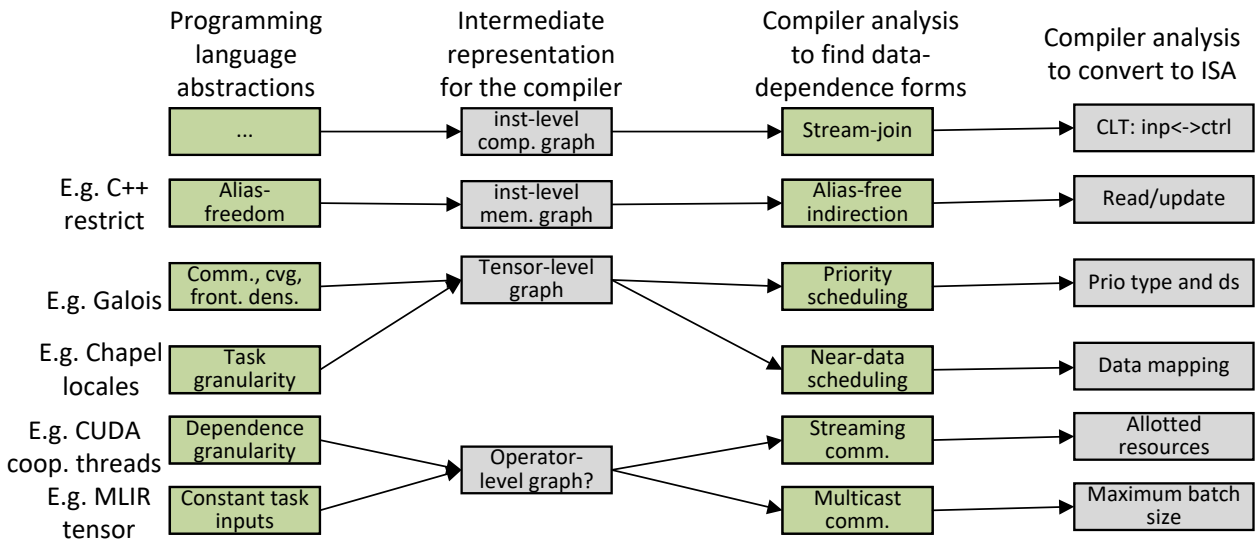


Figure 9.2: Our vision for software stack of our programmable accelerator

9.3.1 Programming language support for "Programmable Accelerators"

For an accelerator to be widely adopted, a non-ninja programmer should be able to extract the most performance by providing minimal information about program properties, e.g., prioritizing shorter distances accelerates convergence in the shortest path algorithm. However, our existing abstraction of specializable data-dependence forms may require non-trivial algorithmic transformation for the programmer to see easily. The natural solution would be that a compiler enumerates all legal transformations and identify patterns through approximate subgraph isomorphism techniques. However, for irregular workloads, enumerating legal transformations often requires programmer hints. For example, alias-freedom information is necessary to reorder updates. Overall, our software stack should expose properties in both programming language and compiler.

Figure 9.2 demonstrates our vision. The programming language would consider asynchronous tasks and mathematical properties (e.g., alias-freedom) as first-class program objects. The compiler will use the programmer information to identify and analyze valid schedules for specializable data-dependence forms. When multiple forms apply, a performance model will be required to pick the optimization. Fortunately, significant prior art

already exists: e.g., for streaming communication, CUDA's cooperative groups [73] allow specifying tasks as a group that will enable communicating via shared scratchpad memory. The compiler can then check for resources to co-schedule these groups of tasks. The challenge would be to explore the ample space of available options and develop a unified intermediate representation for irregular workloads.

9.3.2 Acceleration at scale

The dataset sizes are increasing at an alarming rate, making it critical to scale to large distributed systems. Memory bandwidth is usually the bottleneck for sparse workloads at a single accelerator level. However, using our bottleneck analysis in Chapter 7, we find that network comes on the critical path. For graph workloads with reuse (e.g., triangle counting), load imbalance is critical.

Our insight that *parameterizable solutions that can adapt to different tradeoffs will be valuable*. For example, our multi-level spatial partitioning policy in PolyGraph (Chapter 7) can optimize for locality/load by varying `cluster size`. With more cores, communication is the bottleneck; thus, larger `cluster size` improves performance (due to better locality). The challenge is to reuse our existing algorithm variants taxonomy and analysis to adapt to tradeoffs at scale. This will also require augmenting the taxonomy to at-scale algorithmic optimizations.

9.3.3 Accelerate Workloads with Dynamic Data

Data changes with time in many critical applications, and every new update often needs a real-time response. For example, fraud detection in online transactions should be performed in less than a second [192]). Along with low latency, the system also demands high throughput to cope with the high incoming update rate; for example, the average number of tweets per second is 10k, and we would require updating "trending news" for each of them [213]).

Below are the challenges and requirements introduced by dynamically changing data (we

present these in the context of graph processing): 1: *Dynamic Memory Allocation:* When the real-time response to fine-grained updates is required, allocating/deallocating memory for updates may become a significant bottleneck. For example, inserting in CSR will require shuffling to maintain sorting order. The updates should be fast while ensuring that the accesses to the graph are still sufficiently regular (e.g., linked list insertion is O(1), but it requires pointer chasing to access). Therefore, a hybrid solution is required (see example in Figure 9.3a)). 2. *Dynamic Re-partitioning:* Several graph frameworks rely on offline partitioning to alleviate the communication bottleneck. Re-partitioning overhead may be too high with changing graph structure, and hardware support for dynamically modifying partitions is required (see Figure 9.3b)). 3. *Incremental Computation:* Since the updates introduce minor changes to the large graph, the typical approach to minimize redundant work is only to compute only part of the output impacted by the input graph update. The decision of whether to compute incrementally for each update sequentially, as a batch or in parallel creates the throughput and latency tradeoffs (Figure 9.3c)). The challenge with fewer updates is lower parallelism within the computation, while for batched updates, higher parallelism *may* causes a larger memory footprint. There is an opportunity to explore the potential benefits of dynamically managed structures like caches.

Our TaskStream framework is a good fit to express an extra dimension of parallelism across time and dynamically manage reuse. One could imagine considering graph updates as a new task type that creates incremental computation tasks. The challenge would be ensuring that different graph update instances appear serially in the memory hierarchy.

9.3.4 Generalizing Taskstream Abstraction

We proposed TaskStream for reconfigurable architectures, where different cores are configured to do varying work, and TaskStream optimizations enable balanced load and high locality. However, TaskStream is general, and the scheduling abstractions can also benefit large SoCs. We discuss three examples: 1. **Work scheduling across heterogeneous architectures** [107] for balanced load and locality, 2. **Multi-tenancy** to concurrently run

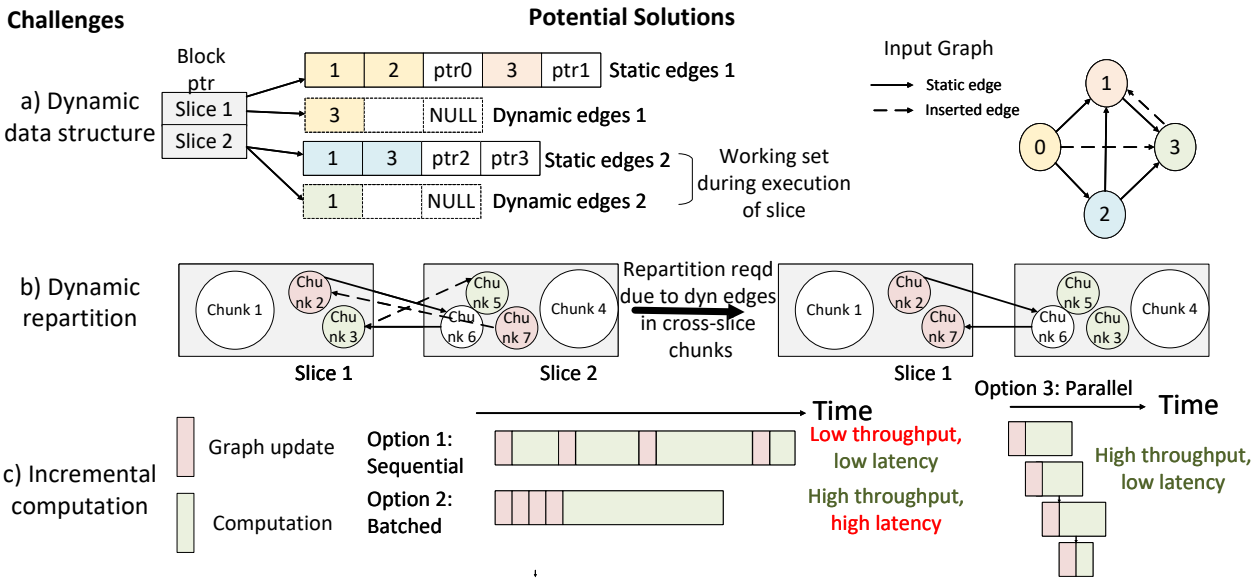


Figure 9.3: Challenges and Proposed Solutions/Insights

multiple applications on a single hardware [112], and 3. **Fault tolerance** when some hardware fails at runtime [39].

Heterogeneous Architectures In Taskstream, programmers can annotate task types with `coreMask` that specifies legal locations where a task can be scheduled to. For our reconfigurable architecture, these were cores, but one can imagine these to be different hardware in a heterogeneous system (see Figure 9.4a). Practical examples include autonomous vehicle applications that span machine learning and a fast Fourier transform accelerator. A point cloud application may use a tree traversal hardware along with a matrix-multiply accelerator [77]. We should be able to reuse TaskStream's general message-passing protocol to apply producer-consumer and multicast optimizations transparently. The challenge would be that implementation should balance network bandwidth tradeoffs in a larger system.

Multi-tenancy Here, multiple concurrent applications work on separate data, and the goal is to maximize throughput while enforcing the latency constraints of each application. The challenge is to make context switching faster. One can imagine exposing context switching,

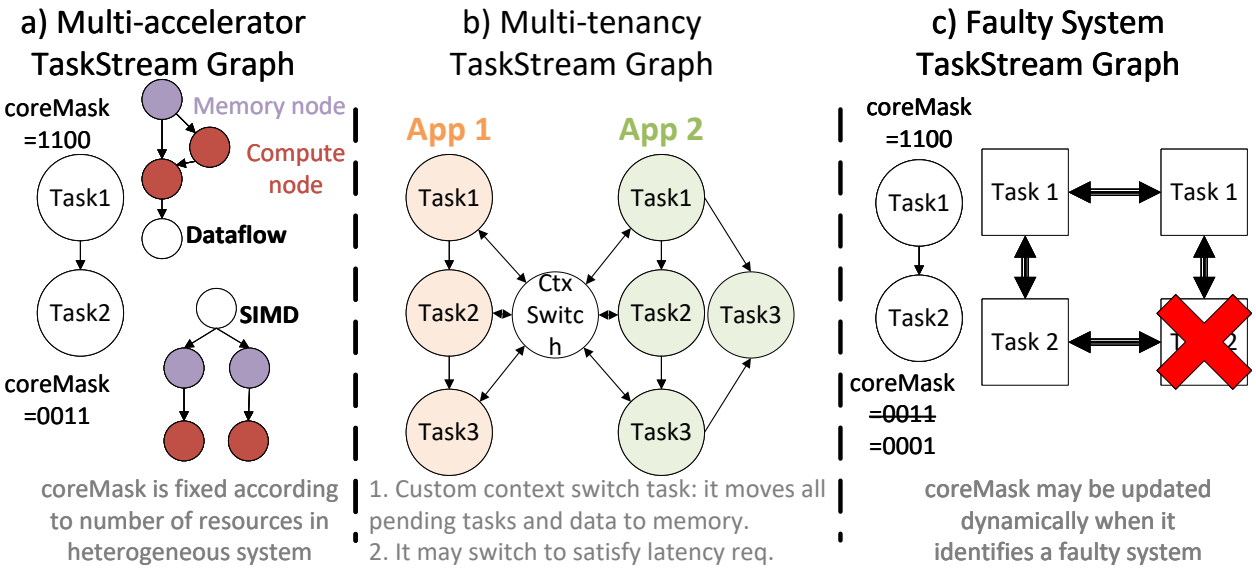


Figure 9.4: Modeling Heterogeneous Architectures and Multi-tenancy as Task Scheduling Problem

at the coarse granularity, in the hardware-software interface. Thus, we can now add special hardware for context switching: it can potentially track latency using time counters to switch at the right time and efficiently overlap communication and computation during switching. Figure 9.4b) shows an example: the applications are divided into phases (shown by Task 1,2,3) and are connected by a "context-switch" task. The applications may also be spatially distributed across cores, the challenge will be determine the optimal distribution for balanced resource utilization.

Fault Tolerance For large-scale distributed systems, machines may be faulty, and thus, the work distribution should be dynamic. In CPUs, virtual machines enable dynamic allocation. However, accelerators are fast and often have stringent switching constraints, e.g., in the order of ms. Therefore, hardware tasks as in TaskStream are attractive. A faulty machine in TaskStream would imply resetting a bit in `coreMask` (see example in Figure 9.4c)).

9.4 Conclusion

This dissertation showed that it is possible to specialize for irregular workloads using specializable data-dependence forms systematically. We used these forms to design a programmable accelerator that efficiently executes both regular and irregular workloads.

We believe this is the right time to think of architectures that are fundamentally built for ASIC-class efficiency and are programmable (e.g., reconfigurable architectures). To explain, consider how other general-purpose architectures are adding accelerator modules to cope with the AI wave: matrix-multiply assists in IBM CPUs [231], tensor cores in NVIDIA GPUs [184], and AI accelerator in Versal FPGAs [82]. Even though these specialized modules significantly improve performance, they are far from standalone application-specific accelerators due to the traditional memory hierarchy in CPUs/GPUs and LUT-level reconfigurability in FPGAs.

Despite the apparent fundamental differences in these architectural approaches, the research directions of accelerator modules in CPUs/GPUs and our specializable irregularity forms (in reconfigurable accelerators) can heavily inspire each other. For example, we were inspired by CPU's thread scheduling techniques to optimize for the locality. To explain, we annotated tasks with the dataset they may access and use this information to schedule tasks near data – this has been explored in task-parallel languages [41], speculative parallel models [109, 266], and partitioned global address space (PGAS) systems [105]. Another approach is Splicing [140], a compiler optimization for recursive task-parallel programs that interleaves tasks with the locality to optimize for the locality. Thread scheduling techniques like splicing can transparently improve cache hit rate.

Some of our insights can improve CPU and GPU performance as well. Like SIMD (e.g., Intel AVX512), a CPU could support stream-join control. An approach could be to add specialized instructions which allow treating registers as FIFOs, and the branch instructions may control the order of data consumption (using simple FSM at FIFOs). A possible extension to a GPU ISA could enable the annotation of a program region as being alias-free indirect. Such an extension would allow GPU scratchpads to achieve the same

benefits as SPU: they would not require expensive memory dependence checking. They could more effectively reorder requests, reducing bank conflicts' impact and achieve higher throughput. NVIDIA's tensor core is historical precedence that such dramatic specialization is feasible.

Irregularity fundamentally affects the efficiency of the lowest level hardware primitives, thus justifying specialization's importance. This dissertation makes progress toward a unified set of primitives for irregularity. Yet, a complete taxonomy, if one exists, will require studying increasingly challenging workloads in the context of compilers and programming languages and iteratively refining hardware/software interfaces and execution models. In other words, many remaining research questions must be answered before we can create a universal architecture for irregularity.

APPENDIX A

Abstract Graph Simulator

PolyGraph proposes four dimensions of algorithm variants (10 algorithms altogether) for seven workloads and various graphs. Thus, we need to run seventy simulations for graphs of large sizes – a fast and reasonably accurate simulator is required for practicality. Existing simulators model all hardware components in a cycle-accurate fashion and are slow for our purpose. The modeled steps involve simulating data transfer from file system to accelerator memory, an in-order control core that recognizes accelerator ISA and talks to accelerator components, and other hardware modules that tick each cycle for work. Moreover, the detailed simulator requires figuring out the ISA and programming support before evaluating an architecture, which may not always be needed for proof-of-concept analysis.

Existing fast simulators use analytical modeling to quickly estimate performance [128, 178]. These techniques work well for dense linear algebra but fail for graph processing where the reuse/memory access pattern is unknown statically. For example, MAESTRO's [128] analytical model for deep learning takes the spatial and temporal reuse information as input. However, in graph processing, the available reuse changes according to the vertices activated at runtime and cannot be determined statically.

We build an abstract simulator, GraphSim, which models accelerator architecture as a dataflow graph [66, 63, 65] where each stage may perform computation, memory/scratchpad access, network access, or reordering. The simulator implements a variety of algorithm templates, which can be customized for each application. The template approach is helpful in the graph domain as many workloads can be implemented with slight variation. It also enables the accelerator control to be abstracted for efficient simulation. Thus, the simulator

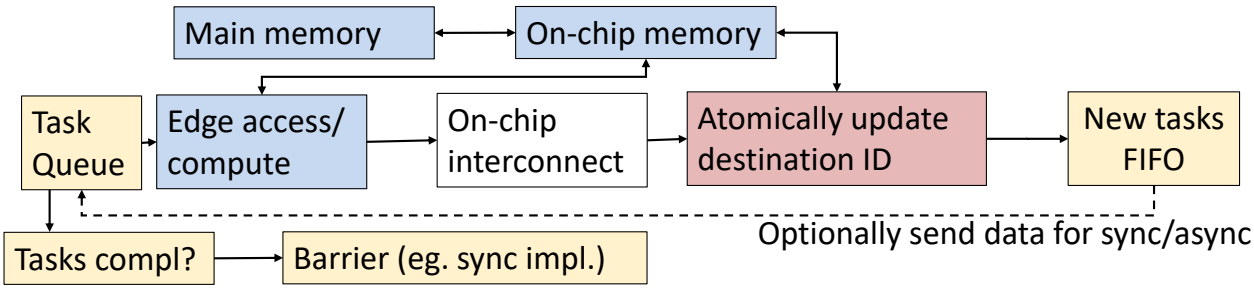


Figure A.1: Vertex-Centric Graph Processing Pipeline Template Implemented in GraphSim

takes more than 100× lower time than the detailed one.

Besides fast simulation, the dataflow graph modeling is general enough to represent various application-specific accelerators that essentially embed the application's datapath in the hardware. Finally, since the datapath node types are sufficiently common across applications, new algorithms can reuse node types and implement new dataflow graph connections.

A.1 GraphSim Implementation

GraphSim is a multi-core simulator where a set of mesh networks connects multiple cores. Each core implements the template dataflow graph for vertex-centric graph processing (see Figure A.1). The nodes of the dataflow graph may be doing one of the four things: 1. accessing memory or scratchpad, 2. sending network requests, 3. performing computation, or 4. reordering a task-packet. The edges represent data transfer – each node receives data from the previous stage, buffers them locally, and executes when execution resources are free. Figure A.2 shows two example accelerators implemented using the vertex-processing template: Graphicionado [94] uses the synchronous algorithm with on-chip scratchpads for sliced execution. Figure A.2b) demonstrates PolyGraph's pipeline when it is running the asynchronous priority and sliced execution mode.

We allow flexibility in several useful architectural features. The prominent ones include spatial data distribution techniques that may be random, may optimize for locality (linear)

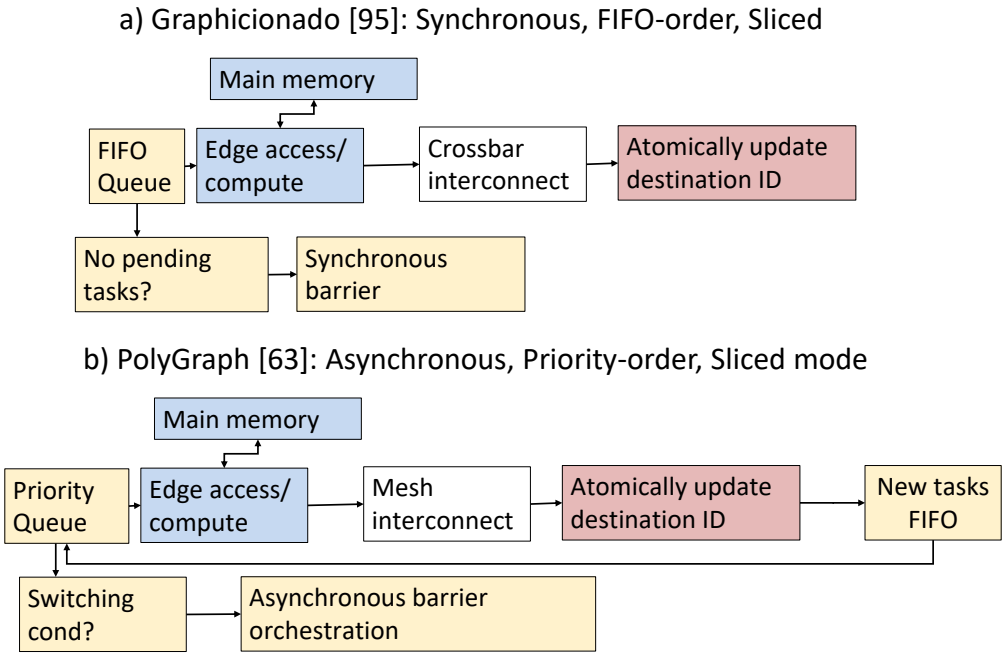


Figure A.2: Example Accelerators Implemented using GraphSim's Template

or load balance (modulo), or allow parametrically balancing between them (multi-level). The workload distribution can be near-data or use dynamic work stealing. The network traffic can be optimized using standard multicast and path-based multicast [27]. Along with optional optimizations, we support idealized dimensions of each. For network and memory, the ideal case corresponds to 0-cycle latency and infinite bandwidth, and for distributed processing, the idealized scenario assumes a giant core with an infinitely-sized task scheduler.

The latency of the computation and task scheduler are constants and configurable by the programmer. DRAMSim2 is used for DRAM simulation [201]. For mesh network simulation, we developed a detailed in-house simulator.

GraphSim Usage Our simulator is available here: https://github.com/PolyArch/graphsim-simulator. It depends on standard packages and uses MACROS to define architecture, input graphs, and workload properties. We model three categories of accelerators: prior application-specific, our TaskStream-based PolyGraph accelerator, and idealized architectures with ideal network/memory/task-scheduler. README provides details for running

Feature	Supported Input Dimensions
Graph structure	Low, high diameter, dynamically changing
Graph slices	No-slice, slice count
Workload order sensitivity	Order sens (sp, pr, cf, astar) or not (bfs, cc, gcn)
Workload frontier-size	Sparse frontier (sssp, bfs, cc), Dense frontier (pr, , cf, gcn)
Feature	**Supported Architecture Dimensions**
Network topology	Mesh, crossbar, hierarchical-mesh-crossbar, *Ideal:* crossbar with inf-sized queues
Network traffic	Std. multicast, Path multicast, *Ideal:* 0 latency, Inf. bandwidth
Cache replacement	LRU, *Ideal:* allhits, allmiss
Main memory	DRAM, *Ideal:* 0 latency, Inf. bandwidth
Spatial work distribution	Near-data, Work-stealing, *Ideal:* Single giant core
Temporal work distribution	Priority order, Creation order, *Ideal:* Inf. sized queue
Feature	**Supported Programming Dimensions**
Update visibility	Synch, Slice-sync, Asynch
Vertex scheduling	Locality, Creation, Data-dependent priority
Slice scheduling	No-slice, Round-robin, Locality, priority
Update direction	Pull, Push
Spatial data distribution	Random, Locality, Load, Multi-level
Dynamic algorithm	Recomputation, Incremental

Table A.1: Graph Accelerator Options Supported in GraphSim

these experiments, and Table A.1 shows the options currently supported by GraphSim.

A.2 Limitations

GraphSim gains its speed by limiting its simulation accuracy for things we believe are not first-order concerns during the initial exploration stages. The first is that we do not need to design the programmer interface as programmers directly express their applications in dataflow-graph form. Thus, the difficulty of programming cannot be estimated. We also do not model the configuration time of accelerator components (usually small). We assume

backpressure-based flow control in the networks and do not consider/evaluate coarse-grained credit-based flow control techniques that may be useful to prevent network congestion. Finally, we limit bandwidth only from memories but do not model bandwidth internal to the core, under the assumption that core-local networks (e.g. a CGRA network) can be provisioned not to be the limiting factor in these workloads. Nevertheless, we believe the GraphSim simulator will help perform extensive experiments to analyze the broad accelerator space practically.

REFERENCES

[1] Apache MADlib: big data machine learning in sql. `https://madlib.apache.org/`.

[2] Intel math kernel library. `http://software.intel.com/en-us/intel-mkl`.

[3] Pytorch geometric. `https://www.pyg.org/`.

[4] XLA: domain-specific compiler for linear algebra to optimizes tensorflow computations.

[5] Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[6] Intel(r) math kernel library for deep neural networks., 2016. "https://github.com/01org/mkl-dnn".

[7] Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 631–643, Berkeley, CA, USA, 2017. USENIX Association.

[8] Neural network distiller by intel ai lab., 2017. "https://github.com/NervanaSystems/distiller".

[9] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, and M. Martonosi. *General-Purpose Graphics Processor Architecture*. Morgan & Claypool, 2018.

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[11] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient speculative parallelism for accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1247–1262, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, et al. Think fast: a tensor streaming processor (TSP) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158. IEEE, 2020.

[13] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, page 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[14] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[15] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117, 2016.

[16] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 2–9. IEEE, 2005.

[17] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, June 2016.

[18] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *arXiv preprint arXiv:1803.01164*, 2018.

[19] Amazon. Amazon inferentia ml chip. *URL: https://aws.amazon.com/machine-learning/inferentia/*, 2020.

[20] Apple. Apple a12 chip. *URL: https://en.wikipedia.org/wiki/Apple_A12*.

[21] Apple. Apple m1 chip. *URL: https://www.apple.com/uk/mac/m1/*.

[22] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *UC Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[23] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. GPU acceleration for support vector machines. In *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*, 2011.

[24] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM.

[25] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. SC'12, pages 1–10. IEEE, 2012.

[26] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[27] Leul Belayneh and Valeria Bertacco. GraphVine: exploiting multicast for scalable graph analytics. In *DATE*, pages 762–767. IEEE, 2020.

[28] Imogen H Bell, Jennifer Nicholas, Mario Alvarez-Jimenez, Andrew Thompson, and Lucia Valmaggia. Virtual reality as a clinical tool in mental health research and practice. *Dialogues in clinical neuroscience*, 2022.

[29] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[30] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 2, pages 538–547 vol.2, 2000.

[31] Puneeth Bhat, Jose Moreira, and Satish Kumar Sadasivam. Matrix-multiply assist (mma) best practices guide. *IBM Corporation*, 2021.

[32] Nathan Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[33] Nathaniel Bleier, Muhammad Husnain Mubarik, Srijan Chakraborty, Shreyas Kishore, and Rakesh Kumar. Rethinking programmable earable processors. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 454–467, 2022.

[34] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 1996.

[35] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237, 2005.

[36] Prerna Budhkar, Ildar Absalyamov, Vasileios Zois, Skyler Windh, Walid A Najjar, and Vassilis J Tsotras. Accelerating in-memory database selections using latency masking hardware threads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):1–28, 2019.

[37] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein.

[38] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, July 2004.

[39] Matt Calder, Ryan Gao, Manuel Schröder, Ryan Stewart, Jitendra Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. Odin:{Microsoft's} scalable {Fault-Tolerant}{CDN} measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 501–517, 2018.

[40] Bingyi Cao, Kenneth A. Ross, Stephen A. Edwards, and Martha A. Kim. Deadlock-free joins in db-mesh, an asynchronous systolic array accelerator. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 5:1–5:8, 2017.

[41] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in cool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, page 249–259, New York, NY, USA, 1993. Association for Computing Machinery.

[42] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[43] Shanzhi Chen and Jian Zhao. The requirements, challenges, and technologies for 5g of terrestrial mobile telecommunication. *IEEE Communications Magazine*, 52(5):36–43, 2014.

[44] Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, et al. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, 2007.

[45] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 55–67, Piscataway, NJ, USA, 2018. IEEE Press.

[46] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[47] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[48] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[49] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[50] Yu-Ting Chen, Jason Cong, Jie Lei, and Peng Wei. A novel high-throughput acceleration engine for read alignment. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 199–202. IEEE, 2015.

[51] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

[52] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[53] Yuze Chi, Licheng Guo, and Jason Cong. Accelerating SSSP for power-law graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 190–200, 2022.

[54] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. CGRA-ME: a unified framework for cgra modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189, 2017.

[55] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[56] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (RSVP). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 141–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *ISLPED*, 2013.

[58] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. CHARM: a composable heterogeneous accelerator-rich microprocessor. In *ISPLED*, 2012.

[59] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. BiN: a buffer-in-nuca scheme for accelerator-rich cmps. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 225–230, 2012.

[60] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of CGRA. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16. IEEE, 2014.

[61] NVIDIA Corp. GeForce GTX 1080 Whitepaper.

[62] NVIDIA Corp. NVIDIA GPU programming guide, v2.2.1, November, 2004.

[63] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 595–608, 2021.

[64] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. Systematically understanding graph accelerator dimensions and the value of hardware flexibility. *IEEE Micro*, 42(4):87–96, 2022.

[65] Vidushi Dadu and Tony Nowatzki. Taskstream: accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, 2022.

[66] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 924–939, 2019.

[67] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general-purpose acceleration: Finding structure in irregularity. *IEEE Micro*, 40(3):37–46, 2020.

[68] 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, April 2019.

[69] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. Dimmining: pruning-efficient and parallel graph mining on near-memory-computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 130–145, 2022.

[70] TIMOTHY A Davis, WILLIAM W Hager, SCOTT P Kolodziej, and S NURI Yeralan. Algorithm XXX: mongoose, a graph coarsening and partitioning library. *ACM Trans. Math. Software*, 2019.

[71] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, June 2015.

[72] A. Duran and M. Klemm. The Intel many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, 2012.

[73] Anne C Elster and Tor A Haugdahl. Nvidia Hopper GPU and Grace CPU highlights. *Computing in Science & Engineering*, 24(2):95–100, 2022.

[74] Xiaojun Fan, Xinyu Jiang, and Nianqi Deng. Immersive technology: A meta-analysis of augmented/virtual reality applications and their impact on tourism experience. *Tourism Management*, 91:104534, 2022.

[75] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 533–545, New York, NY, USA, 2015. ACM.

[76] Yuanwei Fang, Chen Zou, Aaron Elmore, and Andrew Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[77] Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. Crescent: taming memory irregularities for accelerating deep point cloud analytics. *arXiv preprint arXiv:2204.10707*, 2022.

[78] J. Fowers, J. Y. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59, May 2015.

[79] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[80] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 36–43. IEEE, 2014.

[81] Daichi Fujiki, Aran Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. GenAx: a genome sequencing accelerator. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 69–82. IEEE Press, 2018.

[82] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: Versaltm architecture. In *Proceedings of the 2019*

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 84–93, 2019.

[83] Nitin A Gawande, Jeff A Daily, Charles Siegel, Nathan R Tallent, and Abhinav Vishnu. Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing. *Future Generation Computer Systems*, 108:1162–1172, 2020.

[84] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1027–1040. IEEE, 2021.

[85] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–165, 2019.

[86] Ashish Gondimalla, Sree Charan Gundabolu, T. N. Vijaykumar, and Mithuna Thottethodi. Barrier-free large-scale sparse tensor accelerator (BARISTA) for convolutional neural networks. *CoRR*, abs/2104.08734, 2021.

[87] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on {OSDI} 12)*, 2012.

[88] V. Govindaraju, Chen-Han Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, 2011.

[89] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, September 2012.

[90] Paul Grigoraş, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. Optimising sparse matrix vector multiplication for large scale FEM problems on FPGA. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–9. IEEE, 2016.

[91] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[92] Gagan Gupta and Gurindar S Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 59–70. IEEE, 2011.

[93] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. ACT: designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 784–799, 2022.

[94] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[95] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H Oh, Krste Asanovic, Jae W Lee, and Lisa Wu Wills. Genesis: a hardware acceleration framework for genomic data analysis. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 254–267. IEEE, 2020.

[96] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[97] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.

[98] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: efficient inference engine on compressed deep neural network. *SIGARCH Comput. Archit. News*, 44(3):243–254, June 2016.

[99] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[100] Masanori Hashimoto, Xu Bai, Naoki Banno, Munehiro Tada, Toshitsugu Sakamoto, Jaehoon Yu, Ryutaro Doi, Yusuke Araki, Hidetoshi Onodera, Takashi Imagawa, et al. 33.3 via-switch fpga: 65nm cmos implementation and architecture extension for al applications. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 502–504. IEEE, 2020.

[101] Muhammad Amber Hassaan, Donald D Nguyen, and Keshav K Pingali. Kinetic dependence graphs. *ACM SIGPLAN Notices*, 50(4):457–471, 2015.

[102] Kim Hazelwood et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA, 2018*. IEEE, 2018.

[103] Kartik Hegde et al. Extensor: An accelerator for sparse tensor algebra. In *MICRO'52*, 2019.

[104] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. UCNN: exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 674–687, Piscataway, NJ, USA, 2018. IEEE Press.

[105] Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin. Alembic: Automatic locality extraction via migration. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 879–894, New York, NY, USA, 2014. Association for Computing Machinery.

[106] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. Ayguadé, and M. Valero. Advanced pattern based memory controller for fpga based hpc applications. In *2014 International Conference on High Performance Computing Simulation (HPCS)*, pages 287–294, July 2014.

[107] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, et al. ILLIXR: enabling end-to-end extended reality research. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–38. IEEE, 2021.

[108] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[109] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez. Data-centric execution of speculative parallel programs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[110] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, Dec 2015.

[111] Vidushi Dadu Jian Weng, Sihao Liu and Tony Nowatzki. DSAGEN: democratizing spatial accelerator research. *URL: http://www.seas.ucla.edu/ jianw/dsagen/tutorial.html*, 2020.

[112] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's TPUv4i: industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[113] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt,

Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[114] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[115] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.

[116] Ilyes Kacher, Maxime Portaz, Hicham Randrianarivo, and Sylvain Peyronnet. Graphcore c2 card performance for image-based deep learning application: A report. *arXiv preprint arXiv:2002.11670*, 2020.

[117] Shinhaeng Kang, Sukhan Lee, Byeongho Kim, Hweesoo Kim, Kyomin Sohn, Nam Sung Kim, and Eojin Lee. An FPGA-based rnn-t inference accelerator with pim-hbm. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 146–152, 2022.

[118] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[119] Brucek Khailany, William J Dally, Ujval J Kapasi, Peter Mattson, Jinyung Namkoong, John D Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE micro*, 21(2):35–46, 2001.

[120] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[121] Marius Knaust, Enrico Seiler, Knut Reinert, and Thomas Steinke. Co-design for energy efficient and fast genomic search: Interleaved bloom filter on fpga. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 180–189, 2022.

[122] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[123] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[124] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–80, 2018.

[125] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 52–, Washington, DC, USA, 2004. IEEE Computer Society.

[126] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):162–173, June 2007.

[127] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[128] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[129] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. ASPLOS '18.

[130] George Kyriazis. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*, page 21, 2012.

[131] Mandy La and Andrew Chien. Cerebras systems: Journey to the wafer-scale engine. 2020.

[132] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: a compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[133] Michael A Laurenzano, Ananta Tiwari, Allyson Cauble-Chantrenne, Adam Jundt, William A Ward, Roy Campbell, and Laura Carrington. Characterization and bottleneck analysis of a 64-bit ARMv8 platform. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 36–45. IEEE, 2016.

[134] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 129–140, New York, NY, USA, 2011. ACM.

[135] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[136] Jure Leskovec and Andrej Krevl. SNAP datasets: Stanford large network dataset collection, 2014.

[137] Scott T Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. *ACM Sigmod Record*, 22(2):22–31, 1993.

[138] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788. IEEE, 2021.

[139] Cedric Lichtenau, Alper Buyuktosunoglu, Ramon Bertran, Peter Figuli, Christian Jacobi, Nikolaos Papandreou, Haris Pozidis, Anthony Saporito, Andrew Sica, and Elpida Tzortzatos. AI accelerator on IBM telum processor: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 1012–1028, 2022.

[140] Jonathan Lifflander and Sriram Krishnamoorthy. Cache locality optimization for recursive programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–16, 2017.

[141] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Optimizing data locality for fork/join programs using constrained work stealing. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 857–868, 2014.

[142] Yi-Chien Lin, Bingyi Zhang, and Viktor Prasanna. HP-GNN: generating high throughput gnn training implementation on cpu-fpga heterogeneous platform. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 123–133, 2022.

[143] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. PuDianNao: A polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[144] Chris Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23, 2011.

[145] Andrea Lottarini, João P Cerqueira, Thomas J Repetti, Stephen A Edwards, Kenneth A Ross, Mingoo Seok, and Martha A Kim. Master of none acceleration: a comparison of accelerator architectures for analytical query processing. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 762–773, 2019.

[146] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[147] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[148] S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam. TAPAS: generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 245–257, Oct 2018.

[149] Robert Ryan McCune, Tim Weninger, and Gregory R. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for distributed graph processing. *CoRR*, abs/1507.04405, 2015.

[150] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for fpgas. In *Third International ACM Symposium on Field-Programmable Gate Arrays*, pages 111–117, Feb 1995.

[151] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*, pages 61–70. Springer, 2003.

[152] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, Jan 2017.

[153] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr. Fine-grained accelerators for sparse machine learning workloads. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 635–640, Jan 2017.

[154] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[155] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal

scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2018.

[156] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[157] M Naumov, LS Chien, P Vandermersch, and U Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[158] Quan M Nguyen and Daniel Sanchez. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1064–1077, 2021.

[159] Chris Nicol. A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing. *WaveComputing WhitePaper*, 2017.

[160] Tony Nowatzki. Stream-dataflow public release. *URL: https://github.com/PolyArch/stream-dataflow*, 2017.

[161] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *27th PACT*, 2018.

[162] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. ISCA '17.

[163] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 298–310, New York, NY, USA, 2015. ACM.

[164] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 27–39, March 2016.

[165] Tony Nowatzki, Venkatraman Govindaraju, and Karthikeyan Sankaralingam. A graph-based program representation for analyzing hardware specialization approaches. *IEEE Computer Architecture Letters*, 14(2):94–98, 2015.

[166] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 495–506, New York, NY, USA, 2013. ACM.

[167] E. Nurvitadhi, A. Mishra, and D. Marr. A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 109–116, Oct 2015.

[168] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE: THE WORLD'S MOST ADVANCED DATA CENTER GPU. *NVIDIA WhitePaper*, 2017.

[169] Hiroyuki Ochi, Kosei Yamaguchi, Tetsuaki Fujimoto, Junshi Hotate, Takashi Kishimoto, Toshiki Higashi, Takashi Imagawa, Ryutaro Doi, Munehiro Tada, Tadahiko Sugibayashi, et al. Via-switch FPGA: Highly dense mixed-grained reconfigurable architecture with overlay via-switch crossbars. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(12):2723–2736, 2018.

[170] Kunle Olukotun, Raghu Prabhakar, Rekha Singhal, Jeffrey D Ullman, and Yaqi Zhang. Efficient multiway hash join on reconfigurable hardware. *arXiv preprint arXiv:1905.13376*, 2019.

[171] Jian Ouyang, Mijung Noh, Yong Wang, Wei Qi, Yin Ma, Canghai Gu, SoonGon Kim, Ki-il Hong, Wang-Keun Bae, Zhibiao Zhao, et al. Baidu Kunlun an AI processor for diversified workloads. In *Hot Chips Symposium*, pages 1–18, 2020.

[172] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218. IEEE, 2017.

[173] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In *ISCA*, pages 166–177, June 2016.

[174] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, Feb 2018.

[175] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. Sparseadapt: Runtime control for sparse linear algebra on a reconfigurable accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1005–1021, 2021.

[176] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2018.

[177] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered instructions: A

control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM.

[178] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 ISPASS*.

[179] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: an accelerator for compressed-sparse convolutional neural networks. ISCA '17.

[180] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hongseok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 166–176, 2008.

[181] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM.

[182] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, et al. The tao of parallelism in algorithms. PLDI '11.

[183] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 582–587, 2002.

[184] Jeff Pool. Accelerating sparsity in the nvidia ampere architecture. *GTC 2020*, 2020.

[185] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C Jeffrey. A scalable architecture for reprioritizing ordered parallelism. In *ISCA*, pages 437–453, 2022.

[186] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A reconfigurable accelerator for parallel patterns. *IEEE Micro*, 2018.

[187] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 651–665, New York, NY, USA, 2016. ACM.

[188] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE, 2017.

[189] Benoıt Pradelle, Benoıt Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. Polyhedral optimization of tensorflow computation graphs.

[190] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, et al. A reconfigurable fabric for accelerating large-scale datacenter services. ISCA '14.

[191] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[192] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 2018.

[193] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. GraphPulse: an event-driven hardware accelerator for asynchronous graph processing. In *MICRO*, pages 908–921. IEEE, 2020.

[194] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. Sparsecore: stream isa and processor specialization for sparse computation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–199, 2022.

[195] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Y. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, June 2016.

[196] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[197] Thomas J. Repetti, João P. Cerqueira, Martha A. Kim, and Mingoo Seok. Pipelining a triggered processing element. In *Proceedings of the 50th Annual IEEE/ACM Inter-*

national Symposium on Microarchitecture, MICRO-50 '17, pages 96–108, New York, NY, USA, 2017. ACM.

[198] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector lane threading. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 55–64, Aug 2006.

[199] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 3–13, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[200] Alec Roelke and Mircea R. Stan. RISC5: Implementing the RISC-V ISA in gem5. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[201] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: a cycle accurate memory system simulator. *IEEE computer architecture letters*, 10(1):16–19, 2011.

[202] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. Capstan: A vector rda for sparsity, 2021.

[203] Giordano Salvador, Wesley H Darvin, Muhammad Huzaifa, Johnathan Alsop, Matthew D Sinclair, and Sarita V Adve. Specializing coherence, consistency, and push/pull for gpu graph analytics. In *ISPASS*, pages 123–125. IEEE, 2020.

[204] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 22–32, 2011.

[205] Daniel Sanchez, David Lo, Richard M Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 22–32. IEEE, 2011.

[206] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. *SIGPLAN Not.*, 45(3):311–322, March 2010.

[207] Alina Sbîrlea, Yi Zou, Zoran Budimlic, Jason Cong, and Vivek Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. *ACM SIGPLAN Notices*, 47(5):61–70, 2012.

[208] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. PPoPP '17, page 249–265, New York, NY, USA, 2017. Association for Computing Machinery.

[209] Francesco Sgherzi, Alberto Parravicini, Marco Siracusa, and Marco D Santambrogio. Solving large top-k graph eigenproblems with a memory and compute-optimized fpga design. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 78–87. IEEE, 2021.

[210] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[211] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–27, 2019.

[212] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir -an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-52, pages 940–953, New York, NY, USA, 2019. Association for Computing Machinery.

[213] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. Graphjet: real-time content recommendations at twitter. *Proceedings of the VLDB Endowment*.

[214] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[215] David E Shaw, JP Grossman, Joseph A Bank, Brannon Batson, J Adam Butts, Jack C Chao, Martin M Deneroff, Ron O Dror, Amos Even, Christopher H Fenton, et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 41–53. IEEE, 2014.

[216] Yujia Shen, Arthur Choi, and Adnan Darwiche. Tractable operations for arithmetic circuits of probabilistic models. In *Advances in Neural Information Processing Systems 29 (NIPS)*, 2016.

[217] Harsha Vardhan Simhadri, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Aapo Kyrola. Experimental analysis of space-bounded schedulers. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, page 30–41, New York, NY, USA, 2014. Association for Computing Machinery.

[218] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49(5):465–481, May 2000.

[219] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24. IEEE, 2015.

[220] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68. IEEE, 2019.

[221] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. GraphR: accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2018.

[222] Arun Subramaniyan, Yufeng Gu, Timothy Dunn, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Genomicsbench: A benchmark suite for genomics. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–12. IEEE, 2021.

[223] Mengshu Sun, Zhengang Li, Alec Lu, Yanyu Li, Sung-En Chang, Xiaolong Ma, Xue Lin, and Zhenman Fang. FILM-QNN: efficient FPGA acceleration of deep neural networks with intra-layer, mixed-precision quantization. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 134–145, 2022.

[224] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[225] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 291–, Washington, DC, USA, 2003. IEEE Computer Society.

[226] Tuan Ta, Lin Cheng, and Christopher Batten. Simulating multi-core RISC-V systems in gem5. 2018.

[227] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David T Blaauw, Trevor N Mudge, and Ronald G Dreslinski. NDMiner: accelerating graph pattern mining using near data processing. In *ISCA*, pages 146–159, 2022.

[228] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 575–587. IEEE, 2018.

[229] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. General purpose task-dependence management hardware for task-based dataflow programming models. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 244–253. IEEE, 2017.

[230] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. Cortexsuite: A synthetic brain benchmark suite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 76–79. IEEE, 2014.

[231] Brian W Thompto, Dung Q Nguyen, José E Moreira, Ramon Bertran, Hans Jacobson, Richard J Eickemeyer, Rahul M Rao, Michael Goulet, Marcy Byers, Christopher J Gonzalez, et al. Energy efficiency boost in the ai-infused power10 processor. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2021.

[232] Transaction Processing Performance Council. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html*, 2008.

[233] Yatish Turakhia, Gill Bejerano, and William J Dally. Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213. ACM, 2018.

[234] Yatish Turakhia, Sneha D Goenka, Gill Bejerano, and William J Dally. Darwin-WGA: a co-processor provides increased sensitivity in whole genome alignments with high speedup.

[235] Johan Ugander and Lars Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, page 507–516, New York, NY, USA, 2013. Association for Computing Machinery.

[236] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[237] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 13–26, New York, NY, USA, 2017. ACM.

[238] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. Aurochs: An architecture for dataflow threads. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 402–415, 2021.

[239] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. Gorgon: Accelerating machine learning from relational data. In *2020 ACM/IEEE*

47th Annual International Symposium on Computer Architecture (ISCA), pages 309–321, 2020.

[240] Rafael Villena Taranilla, Ramón Cózar-Gutiérrez, José Antonio González-Calero, and Isabel López Cirugeda. Strolling through a city of the roman empire: an analysis of the potential of virtual reality to teach history in primary education. *Interactive Learning Environments*, 30(4):608–618, 2022.

[241] Dani Voitsechov and Yoav Etsion. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 205–216, Piscataway, NJ, USA, 2014. IEEE Press.

[242] Erwei Wang, James J Davis, Georgios-Ilias Stavrou, Peter YK Cheung, George A Constantinides, and Mohamed Abdelfattah. Logic shrinkage: Learned fpga netlist sparsity for efficient neural network inference. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 101–111, 2022.

[243] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[244] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. {RStream}: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 763–782, 2018.

[245] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. An overview of micron's automata processor. In *2016 international conference on hardware/software codesign and system synthesis (CODES+ ISSS)*, pages 1–3. IEEE, 2016.

[246] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016.

[247] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 736–749, New York, NY, USA, 2019. ACM.

[248] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. Near-Stream Computing: general and transparent near-cache acceleration. In *HPCA*, 2022.

[249] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. Stream Floating: enabling proactive and decentralized cache optimizations. In *HPCA*, 2021.

[250] Gabriel Weisz and James C Hoe. Coram++: Supporting data-structure-specific memory interfaces for fpga computing. In *25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2015.

[251] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: synthesizing programmable spatial accelerators. In *ISCA*, pages 268–281. IEEE, 2020.

[252] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro*, 2022.

[253] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 703–716, 2020.

[254] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S Dhillon, and Keshav Pingali. Scalable data-driven pagerank: Algorithms, system issues, and lessons learned. In *European Conference on Parallel Processing*, pages 438–450. Springer, 2015.

[255] NVIDIA Whitepaper. Cuda C best practices guide, May 2019. https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[256] WikiChip. Configurable spatial accelerator, 2019. https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator.

[257] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 95–102. IEEE, 2013.

[258] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, 2013.

[259] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. ASPLOS '14.

[260] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. *ACM SIGPLAN Notices*, 50(8):194–204, 2015.

[261] Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. Tigris: Architecture and algorithms for 3d perception in point clouds. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 629–642, 2019.

[262] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, et al. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In

Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 615–628, New York, NY, USA, 2019. ACM.

[263] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 711–724. IEEE, 2020.

[264] Yifan Yang, Zhaoshi Li, Yangdong Deng, Zhiwei Liu, Shouyi Yin, Shaojun Wei, and Leibo Liu. GraphABCD: scaling out graph analytics with asynchronous block coordinate descent. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 419–432. IEEE, 2020.

[265] Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. Ganax: A unified mimd-simd acceleration for generative adversarial networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 650–661. IEEE Press, 2018.

[266] Victor A Ying, Mark C Jeffrey, and Daniel Sanchez. T4: Compiling sequential code for effective speculative parallelization in hardware. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 159–172. IEEE, 2020.

[267] Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, page 315–325, New York, NY, USA, 2013. Association for Computing Machinery.

[268] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[269] Alberto Zeni, Guido Walter Di Donato, Lorenzo Di Tucci, Marco Rabozzi, and Marco D Santambrogio. The importance of being X-Drop: high performance genome alignment on reconfigurable hardware. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–141. IEEE, 2021.

[270] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 27–42, 2022.

[271] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 593–607, New York, NY, USA, 2018. ACM.

[272] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. ASPLOS 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[273] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193, 2015.

[274] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557. IEEE, 2018.

[275] Si; Hsieh Cho-Jui Zhang, Huan; Si. GPU-acceleration for large-scale tree boosting. *arXiv eprint arXiv:1706.08359, 2017*, 2017.

[276] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. Scalable interconnects for reconfigurable spatial architectures. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 615–628, 2019.

[277] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. Sara: Scaling a reconfigurable dataflow accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1041–1054. IEEE, 2021.

[278] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. DiGraph: an efficient path-based iterative directed graph processing system on multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 601–614, New York, NY, USA, 2019. ACM.

[279] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. DepGraph: a dependency-driven accelerator for efficient iterative graph processing. In *HPCA*. IEEE, 2021.

[280] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International conference on algorithmic applications in management*, pages 337–348. Springer, 2008.

[281] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.

[282] Yuhao Zhu and Vijay Janapa Reddi. Webcore: architectural support for mobileweb browsing. *ACM SIGARCH Computer Architecture News*, 42(3):541–552, 2014.

[283] Ling Zhuo and Viktor K Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74. ACM, 2005.

[284] Youwei Zhuo, Jingji Chen, Qinyi Luo, Yanzhi Wang, Hailong Yang, Depei Qian, and Xuehai Qian. SympleGraph: distributed graph processing with precise loop-carried dependency guarantee. In *PLDI*, page 592–607, New York, NY, USA, 2020. Association for Computing Machinery.

[285] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. GraphQ: scalable pim-based graph processing. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 712–725, New York, NY, USA, 2019. ACM.