Accelerating Irregular Applications with Pipeline Parallelism

by

Quan Minh Nguyen

B.S., University of California, Berkeley (2014)
S.M., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by .....
Daniel Sanchez
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Accelerating Irregular Applications with Pipeline Parallelism

by

Quan Minh Nguyen

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Irregular applications have frequent data-dependent memory accesses and control flow. They arise in many emerging and important domains, including sparse deep learning, graph analytics, and database processing. Conventional architectures cannot handle irregular applications efficiently because their techniques for improving performance, like exploiting instruction-level or data-level parallelism, are not tailored to them. Thus, continued progress in these crucial domains depends on exploring new avenues of parallelism.

Fortunately, irregular applications contain abundant but untapped *pipeline parallelism*: they can be divided into networks of stages. Pipelining not only exposes parallelism but also enables *decoupling*, which hides the latency of long events by allowing producer stages to run ahead of consumer stages. To properly decouple these applications, though, this pipeline parallelism must be exploited at *fine-grain*, with few operations per stage. Prior work has proposed architectures, compilers, and languages for pipelines, but focus on *regular* pipelines, and thus are unable to overcome several challenges of irregular applications. First, architectures need to support the efficient execution of many fine-grain pipeline stages. Second, such irregular pipelines suffer from load imbalance, as the amount of work in each stage varies rapidly as the program runs. Finally, these stages must communicate and coordinate changes in control flow.

This thesis demonstrates that exploiting fine-grain pipeline parallelism in irregular applications is effective and practical. To this end, this thesis proposes two hardware architectures and a compiler: Pipette, the first architecture, reuses existing structures in modern out-of-order cores to implement load-balanced decoupled communication between stages; and Fifer, the second architecture, makes the acceleration benefits of coarse-grain reconfigurable arrays available to irregular applications. Pipette achieves gmean 1.9× speedup over a data-parallel implementation, and Fifer achieves up to 47× speedup over an out-of-order multicore while using considerably less area. Both architectures also further accelerate challenging memory accesses and resolve the load balancing and control flow challenges that are ubiquitous in irregular applications. Finally, Phloem is a compiler that makes it easy for programmers to use these architectures by producing high-performance pipeline-parallel implementations of irregular applications from serial code. Phloem automatically achieves 85% of the performance of manually pipelined versions.

Thesis Supervisor: Daniel Sanchez
Title: Associate Professor of Electrical Engineering and Computer Science

To my teachers

Acknowledgments

I am so lucky.

My fortune starts with my adviser, Daniel Sanchez, model scholar and without whom this thesis would not have been possible. Simply stated, he has taught me to be a better researcher. He kept me focused on the task at hand and has never hesitated to help me understand something, whether it's the first, second, or umpteenth time he's had to explain it. His breadth of knowledge, and his even bigger heart, is inimitable. Most of all, I will never forget the grace with which he welcomed me into his group, during a time in which I didn't believe in myself. I hope this thesis attests to how much I do now.

I would like to extend my gratitude to my thesis committee members, Joel Emer and Vivienne Sze. Joel has taught me that finding the answer to something often starts with finding the right name for it. I hope to emulate his ability to place every technological advance in the context of every one that came before it. Vivienne, who also served on my RQE committee, provided me with valuable perspectives and helped me truly understand the hardware-software interface.

Many professors have been instrumental to my growth and development as a student and as a person. Srini Devadas was the first to welcome me to MIT. Silvina Hanono Wachman and Arvind taught me what it meant to be a good teacher. Saman Amarasinghe and Mengjia Yan provided important insights, about programming languages, security, and more. Back at Berkeley, Krste Asanović started me down the road of computer architecture research. My thanks extend not only to these professors but also the teachers from Troy High School, Brea Junior High, and Fanning Elementary, who have all inspired me and motivated me and whose lessons I seek to pass on to my own students some day.

I extend my sincere appreciation:

To the fine gentlemen of the central committee of 32-G884: Victor Ying, Hyun Ryong (Ryan) Lee, Axel Feldmann, and Yifan Yang. You all made it worth going to the office every day. Within those four surprisingly square walls we learned much more than just computer architecture, and I will always remember the banter and laughter we shared together.

And the members of the broader Sanchez group: Nathan Beckmann, Suvinay Subramanian, Nosayba El-Sayed, Po-An Tsai, Mark Jeffrey, Guowei Zhang, Anurag Mukkara, Domenic Nutile (:fist-bump:), Maleen Abeydeera, Nikola Samardzic, Shabnam Sheikhha, Fares Elsabbagh, Robert

Durfee, Kendall Garner, and Aleksandar Krastev. There has been no more gratifying experience than to be in the presence of such caring, intelligent people. Whether it was through giving paper feedback or being in the mountains of Maine, you guys helped me find where I was (and not just in GeoGuessr). You all are part of this thesis, too.

To Ilia Lebedev and Kyle Hogan: thanks for all those coffee walks to Area Four and, more importantly, helping me through a low point in my graduate career. You two showed me the subtle power of a simple conversation. To Chris Fletcher, Albert Kwon, Alin Tomescu, Hsin-Jung Yang, Zachary Newman, Xiangyao Yu, Sabrina Neuman, Sang-Woo Jun, Andy Wright, Thomas Bourgeat, Sizhuo Zhang, and Murali Vijayaraghavan: you all provided mentorship and friendship during my many years here at the Institute. To Sally Lee for your administrative support and tolerating my chatter.

To the students I have had the privilege to teach and mentor, including SuperUROPs Parker Huntington, Shahir Rahman, and Nithya Attaluri and my 6.175 and 6.004 students. Nothing has made me feel quite as complete as the chance to teach all of you.

To the friends I've made while at MIT, Jon Zhu, Matt Melissa, Sumit Dutta, Jon Gjengset, and Greg Izatt: all of you made our fair city a terrific home for the last eight years. All too brief, but Daniel Mark, Victoria Preston, Sara Sheffels, and Anna Zeng of the Lindy Hop Society helped me find great joy in this last year at MIT. Willie Boag and Vishwak Srinivasan of the CSAIL Student Committee made it a delight to run SIGTBD not just once, but twice. Thanks to these folks, I indeed did have fun at MIT.

To *Thầy Bình*, *Nhi* (Jacquie), and *Tuấn-Anh*: you helped me learn my mother tongue at Harvard.

To my pals from Troy High School: Fady Barsoum, Sen Hirano, Aadil Hussaini, Jennifer Kadowaki, Albert Ou, Lisa Park, Amy Shieh, Aaron Tran, Nolan Vorck, Vivian Wang, and Wendy Yang, you guys are filled with fun medical facts and not everyone has the luck of having such a close group of friends that have stuck together for over a decade and going. Y'all raised me.

To my fellow Berkeley bears, who helped get me here: Mehrdad Niknami (hello), Michael Stephen Ting, Zachary Hargreaves, Stephen Twigg, Yunsup Lee, Andrew Waterman, and the Lindy on Sproul club, you helped set me up for success.

To Liz Chen, my partner, who helped get me out: thank you for your love and support, and for sticking it out with me for all these years. Thanks to Evie and Milo (*Felis catus*), who helped me prepare this thesis, kind of.

To *Uyên*, my loving, hardworking, intelligent baby sister: I know you will always have my back and I'll always have yours. To *Cô Hai*: thank you, *cám ơn*, for making it possible for me to go to grad school. To my cousins, aunts, uncles, and grandparents: you always supported my ambitions and gave indispensable life advice to take along with me.

Finally, to *Ba* and *Má*: my parents, my original teachers.

Q.M.N.
Cambridge, Mass.

Contents

1 Introduction

Advances in computer architecture have enabled significant improvements in the applications that run on them. In turn, these applications produce significant benefits for society in countless ways; as a result, computing is now embedded in all aspects of modern life.

Progress in architecture, silicon fabrication, and other fields have been important drivers of increased performance; in this thesis, we focus on parallelism. *Parallelizing* a program consists of dividing it into smaller units of work, or *tasks*, that can be run at the same time while preserving the semantics of the original program. By exploiting parallelism and performing these tasks simultaneously, we can improve performance by completing the workload faster and reaping its attendant benefits, such as consuming less energy.

As processors have become faster, the amount of data has ballooned and the ways we process it have become much more sophisticated. The result is that *sparse* applications have become much more common. A *sparse* application operates on sparse data structures, like sparse matrices, tensors, graphs, trees, hash tables, and more. These structures exploit the observation that the relationships represented by this data are themselves sparse. For example, a graph's millions of vertices may each have just a few edges to other vertices. Thus, efficiency benefits arise from not having to represent nonexistent relationships.

Leveraging sparsity this way introduces *irregularity*. In this thesis, we define an *irregular application* as one that has frequent, unpredictable (i.e., not statically known or easily computed ahead) data-dependent memory accesses and control flow. However, improvements in computer architecture over the last several decades have primarily benefited only *regular* applications—those that work on *dense, uncompressed* data structures, like dense vectors of matrices. Furthermore, with Moore's Law ending [34], we must be judicious in the way we use transistor budgets

to extract performance from irregular applications.

In this thesis, we consider organizing an application as a *pipeline*. Structuring an application as a pipeline involves further breaking down an algorithm into multiple smaller tasks, called *stages*, which are ordered to preserve the order of operations performed by the original algorithm. Each stage now receives operands from one or more preceding *producer* stages, carries out a subset of the original operations, and sends the results to one or more following *consumer* stages.

This division into multiple stages yields more tasks and exposes additional parallelism. Specifically, *pipeline parallelism* is the property of a pipeline that each stage can carry out its operations using only values received by the stage as well as that stage's internal state. Thus, stages can run independently of each other. There are plenty of examples of exploiting pipeline parallelism: processor pipelines [95] split the execution of each instruction across multiple stages. Image and video processing applications are also structured as pipelines, where each stage operates on an intermediate image or video frame, and application-specific integrated circuit (ASIC) accelerators [115] mirror this organization by structuring their compute as pipelines as well. Finally, pipelines are a common organization in software [103] as well, where an application can be structured as stages that communicate values through memory.

For this thesis, we focus on pipelines in which the stages are implemented as programmable (or configurable) hardware units that can perform these tasks. Stages in these pipelines are connected by queues: storage structures that contain outputs written (enqueued) by a producer stage that are read (dequeued) by a consumer stage. These stages are now *decoupled* from each other. If an output queue becomes full or an input queue becomes empty, the stage stalls, but other stages can continue to run. Decoupling grants a crucial property: *latency tolerance*—the ability to overlap long-latency events with useful work.

Finally, pipelines may operate at different *granularities*. The number of operations in each stage, the amount of data communicated between stages, and the frequency of communication, determine the granularity of the stages and the resulting pipeline. Granularity is relative; in this thesis, we set the finest unit of granularity to the fundamental operations (add, multiply, load, store) on a modern machine's word width (32 or 64 bits) that can be carried out in a single clock cycle. This granularity is comparable to an instruction executed by a RISC pipeline or a micro-op executed on a modern out-of-order processor. Thus, a *fine-grain* stage

carries out few operations on few values and communicates extremely frequently—once every few cycles.

We can leverage pipeline parallelism to accelerate irregular applications, but a careful choice of granularity is essential. Fine-grain pipelines confer numerous advantages: first, the ability to divide applications along sources of irregularity yields ample opportunities to insert latency tolerance mechanisms where needed. Second, dividing an application into fine-grain stages results in stages that contain substantially less work, making them simpler to implement and accelerate. Finally, even with queues of many tens or hundreds of elements, the overall storage area needed remains reasonably sized (comparable to today's L1 data caches on modern CPUs) for physical implementation.

1.1 Challenges

Adding hardware queues between compute units is not new: starting with James Smith's seminal 1982 paper on decoupled access-execute (DAE) [108], many systems [25, 30, 41, 45, 82, 83, 92, 102, 109, 119, 124, 138] also propose connecting compute units with hardware queues. Unfortunately, prior work fails to effectively support irregular applications structured as fine-grain pipelines. This lack of support is due to several challenges:

Applications need the flexibility to be decoupled at any point into arbitrarily many fine-grain stages. Irregular applications contain many sources of irregularity, and they may occur chained together, such as a series of memory accesses. To achieve good performance, sources of irregularity that happen frequently must be decoupled from each other. While fine-grain decoupling enables us to separate sources of irregularity, the underlying hardware must also be able to support these numerous stages. Many earlier designs, including DAE, only support two stages. Moreover, many prior designs limit the types of operations a stage can perform. For example, DAE requires memory accesses to be performed specifically by the access stage.

Stages must coordinate changes in control flow. Activity in one stage could affect control flow decisions made by another stage. For instance, stages may need to synchronize before moving onto the next phase of a program. For performance reasons, nested loops may need to be further decoupled into stages, but these stages need to match the semantics of

the original loop. The need to communicate control flow information between stages introduces additional complexity.

Fine-grain pipelines suffer from load imbalance, as the amount of work in each stage varies rapidly as the program runs. Related to the previous challenge, stages may experience rapid fluctuations in the amount of work in their input queues. One input element could produce many output elements for downstream stages, but the next one could produce few elements. Thus, it is not possible to statically scale or replicate stages to maintain balanced throughputs. Fine-grain pipelines thus have *load imbalance*, meaning that some stages may remain busy with lots of work and other stages may not have work to do at all. Prior work has suffered from poor utilization due to load imbalance, chiefly because they map stages to different compute units.

No tool automatically creates fine-grain pipelines of irregular applications from serial code, limiting the benefits of hardware support. Prior work in compilers of pipeline-parallel programs limit their tasks to be no larger than the innermost loop of an application [19, 43, 102]. However, irregular applications contain many nested control structures with possibly several long-latency events in each level. In order to make full use of hardware support for fine-grain pipeline parallelism, the compiler must recognize opportunities for creating pipelines that span control structures (like multiple loop nests) and properly decouple all long-latency events from each other.

1.2 Contributions

This thesis demonstrates that exploiting fine-grain pipeline parallelism in irregular applications is effective and practical. These are the key enabling insights:

1. Irregular applications can be structured as fine-grain pipelines.

2. Time-division multiplexing is cheap to implement and can be used to efficiently execute these pipelines while maximizing utilization.

3. These pipelines can be built systematically, and therefore automatically, from serial code, making this type of parallelism more accessible to programmers.

This thesis presents two novel hardware architectures and a compiler that leverage these three insights.

Pipette: The first hardware architecture, *Pipette* [82], exploits fine-grain pipeline parallelism using the threads of a multithreaded general-purpose core. Each thread executes a stage in the pipeline, and stages communicate using architecturally visible queues. An interface for passing control flow information between stages, as well as hardware support to make changing control flow efficient, make it practical to implement irregular applications as fine-grain pipelines. To cope with load imbalance, cores reuse the time-division multiplexing feature of modern out-of-order (OOO) cores—simultaneous multithreading—to always have work to issue, even if some stages are blocked. Pipette reduces implementation complexity by reusing other components of existing OOO cores, such as backing queue storage by reusing the physical register file. Pipette achieves gmean 1.9× speedup over a variety of challenging irregular applications.

Fifer: The second hardware architecture, *Fifer* [83], extends Pipette's insights to specialized architectures. Fifer targets coarse-grain reconfigurable arrays (CGRAs), which have been traditionally used for accelerating regular computations but have been ineffective for accelerating irregular applications because of their rigid internal pipelining. Fifer demonstrates that CGRAs can be efficiently leveraged for irregular applications. Like Pipette, Fifer runs networks of pipeline stages and time-multiplexes many stages onto a single CGRA to achieve high utilization. To cope with irregularity, Fifer not only buffers the inputs and outputs to each CGRA, but also changes the currently executing stage in response to varying load. Support for rapid reconfiguration keeps the overhead of time-multiplexing low, and support for passing control information makes it feasible to implement an irregular application's complex control structures within a CGRA. Thanks to the increased compute density of the specialized fabric and improved utilization, Fifer achieves up to 47× speedup over an OOO multicore while using much less area.

Phloem: Finally, the compiler, *Phloem* [84], automatically transforms serial programs into high-performance pipeline-parallel implementations. It systematizes the process of creating pipelines, including accelerating memory accesses and coordinating changes in control flow across stages. Phloem achieves 85% of the performance attained through manual parallelization. Thus, a wide variety of applications can now benefit from hardware support for fine-grain pipeline parallelism.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 reviews background and related work. Chapter 3 presents Pipette, which leverages fine-grain pipeline parallelism within a multithreaded core. Chapter 4 presents Fifer, which extends Pipette's insights to specialized architectures. Chapter 5 presents Phloem, a compiler that transforms serial programs into efficient pipeline-parallel implementations. Chapter 6 concludes this thesis and discusses future work.

2 Background and Related Work

Irregular applications have frequent data-dependent memory accesses and control flow. They are the norm in many domains, like graph analytics and sparse linear/tensor algebra, because irregularity arises from sparse data structures, like graphs and sparse matrices. Their data-dependent accesses and control are often unpredictable, causing poor performance on CPUs and GPUs. To drive the discussion of the background and related work, consider the following code:

```
for (int i = 0; i < N; i++)
  if (A[i] > 0)
    work(B[A[i]]);
```

This simple snippet is representative of the challenges of irregular applications (we will see fuller examples later on). Assume that `work()` takes few operations per call (e.g., about 10), and that it does not modify arrays `A[]` or `B[]`. This code will run very poorly on a CPU: if `A[i]` frequently alternates between positive and negative, the `if (A[i] > 0)` branch is unpredictable, serializing iterations and inducing a very low instructions per cycle (IPC). Moreover, the indirect access `B[A[i]]` will cause frequent memory misses that are hard to prefetch, making execution memory latency-bound. Data parallelism is of limited help: on a GPU or vector processor, `if (A[i] > 0)` induces conditional/masked execution that limits lane utilization, and the frequent memory gather `B[A[i]]` causes expensive uncoalesced accesses.

Instead, consider the following *pipeline-parallel* implementation of the above code snippet:

Each stage runs in parallel, e.g., in a separate CPU core. Stages produce streams of values and communicate them to other stages through queues.

This *decouples* their execution, allowing producers to run ahead of consumers. This decoupling also hides latencies and uses resources better. For example, each branch in the filter stage is resolved more quickly, since `A[i]` comes from a fast queue instead of main memory; and each misprediction in filter no longer fills the core with misspeculated instructions from `work()` or fetches from array `B[]`.

The above pipeline is *fine-grained*: it has very frequent communication, with each stage enqueueing or dequeueing a value every 5–10 instructions. Thus, software-only queues (which take hundreds of cycles per operation [39, 104]) would add very high overheads. To enable fine-grained pipelining, much prior work has proposed adding hardware queues across cores or threads [25, 30, 41, 45, 82, 83, 92, 102, 108, 109, 119, 124, 138]. However, many of these systems only work well when every stage in the pipeline proceeds at a *regular*, predictable rate. By contrast, in an irregular application, stages undergo rapid variations in the amount of work, creating *load imbalance*. For instance, consecutive runs of positive or negative `A[i]` values affect the output rate of filter, quickly changing the ratio of work between the first and last two stages. If these stages were distributed spatially (e.g., scheduled on separate cores), some would idle often while others limit throughput.

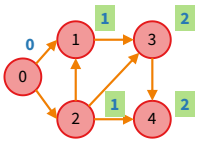
2.1 A Fully Fledged Example: Breadth-First Search

Although the previous example contains enough irregularity to cause serious problems, we turn to a more realistic example, breadth-first search (BFS), to fully illustrate the challenges of irregular applications. BFS is a common graph analytics algorithm that visits all vertices reachable from a source vertex `src` in a given input graph and tags them with the shortest distance to the source, in number of edges.

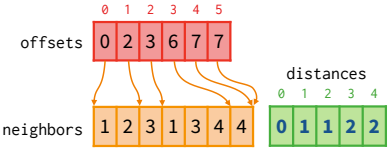
Figure 2-1(a) shows a serial implementation of BFS. This algorithm iteratively tags all vertices at a given distance from the source, `cur_dist`, before moving on to the next distance. A *fringe* tracks the set of all vertices at the previous distance (`cur_dist-1`). As BFS visits the neighbors of each vertex in the fringe, it checks whether the neighbor's distance has been set. If not, BFS sets its distance and adds it to the next iteration's fringe (`next_fringe`). In the next iteration, BFS processes the vertices of the next fringe, continuing until an iteration results in an empty fringe—indicating that all vertices reachable from the source have been visited.

```
define bfs(src):
  distances[src] = 0
  cur_fringe = [src]
  cur_dist = 1
  while not cur_fringe.empty():
    for v in cur_fringe:
      start, end = offsets[v], offsets[v+1]
      for e in range(start, end):
        ngh = neighbors[e]
        dist = distances[ngh]
        if dist is unset:
          distances[ngh] = cur_dist
          next_fringe.push(ngh)
    cur_fringe = []
    swap(cur_fringe, next_fringe)
    cur_dist += 1
```

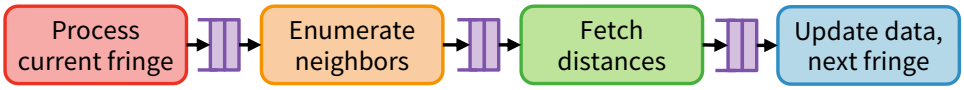
(a) Pseudocode for serial BFS.



(b) An example graph G.



(c) G's CSR representation and, at right, the output (distances) produced by BFS.



(d) A pipeline-parallel implementation of BFS.

Figure 2-1: An implementation of breadth-first search (BFS).

The BFS implementation in Figure 2-1(a) uses a graph in compressed sparse row (CSR) format, the most commonly used representation [80, 106, 112]. Figure 2-1(b) shows an example graph and Figure 2-1(c) shows its CSR representation. CSR stores the graph using two arrays, `offsets` and `neighbors`. For each vertex id, the `offsets` array stores where its neighbors begin in the `neighbors` array. Thus, vertex v has edges to `neighbors`$[i]$ for `i` in the half-open interval [`offsets`$[v]$, `offsets`$[v+1]$). The `neighbors` array stores the vertex id of each neighbor.

Obtaining and setting the distance of a neighbor vertex thus manifests as four indirections in a three-level loop: reading the fringe for the current vertex, reading that vertex's edge list, finding the neighbor of each edge, and finally loading the distance of that neighbor.

Like before, we mitigate the effects of these variable-latency indirections by separating BFS into a feed-forward network of pipeline stages. The stages are then *decoupled* with queues, which buffer work between stages and allow them to run ahead of each other. This decoupling is made possible by the observation that lookups of neighbor distances (at a given `cur_dist`) do not depend on each other.

However, simply decoupling BFS and assigning each stage to a separate core or thread presents a new problem: load imbalance. Load

Name	Flexible number of stages > 2	Independent control flow	Dynamic load balancing	Reuses core structures
DAE [108], DeSC [45]	✘	✘	✘	✘
MT-DCAE [114]	✘	✘	✔	✘
Raw [119], MPPA [30]	✔	✔	✘	✘
Triggered instructions [92]	✔	✔	✔	✘
DSWP [102]	✔	✘	✘	✘
Outrider [25]	✔	✘	✔	✘

Table 2-1: Feature comparison of related work.

imbalance occurs because work varies quickly throughout phases of execution. In BFS, load imbalance could occur because a vertex could have many neighbors, or no neighbors at all. This leads to poor utilization, in which some cores or threads remain idle while others run at full tilt.

2.2 Related Work

We break our discussion of related work into three major parts. First, we discuss techniques that employ decoupled execution. Second, we review prior specialized architectures and how they improve compute density over general-purpose cores but also why they fall short of our goals. Third, we explore why compiler support for these architectures is lacking, limiting the accessibility and effectiveness of hardware support.

2.2.1 Decoupled Architectures

In general, prior decoupled architectures suffer from two limitations: *(i)* their queue-based communication and control mechanisms target applications with regular control flow, and impose restrictions on the number of stages or the types of activities within each stage, so they are *insufficient to decouple stages* in irregular applications; and *(ii)* most of this prior work places each stage on a different core, which causes *high load imbalance* in irregular programs. Table 2-1 summarizes the shortcomings of this prior work.

Decoupled Access-Execute (DAE) architectures [108] feature two specialized units: an access core that performs memory operations and an execute core that performs compute operations. The cores are linked

to each other by queues, allowing the access core to run ahead. Software techniques for DAE [54] leverage the same insights, but without the need for specialized hardware. Unfortunately, DAE-based architectures and its descendants, including PIPE [41], ZS-1 [109], ACRI-1 [124], MT-DCAE [114], and DeSC [45], suffer from loss of decoupling because they allow only two stages, access and execute, and each stage has a limited set of operations, which causes tight two-way dependences between these stages. For example, DAE cannot decouple BFS as described.

Streaming multicores like Raw [119], Imagine [58], Merrimac [27], and Kalray's MPPA [30] introduce hardware support for decoupled communication between cores, which can stream values over the network [27, 29, 65, 120]. Unlike DAE, streaming multicores allow more than two pipeline stages and let each core execute arbitrary instructions. However, this cross-core decoupling is inefficient for irregular workloads due to *load imbalance*: since the work per stage varies quickly, cores incur many idle cycles. In fact, these streaming multicores were only used for *regular* pipeline-parallel applications. These systems relied on precise knowledge of the execution time and communication requirements of all stages, gathered through static analysis or annotations, to statically map stages to cores [65, 67, 94].

Decoupled multithreaded cores introduce support for queue-based communication among cores. Decoupled software pipelining (DSWP) [102] proposes the synchronization array, a hardware structure to facilitate communication between cores or the threads of a multithreaded core. But DSWP focuses on pipelining a single loop across different threads, which is too limiting for irregular applications. For example, BFS uses three levels of nested loops, with stages across several loop levels (and because inner loops are short, decoupling only within the inner loop is insufficient).

In addition to DSWP, Outrider [25] introduces hardware queues to decouple the threads of a single multithreaded core. In principle, this makes load-balancing across stages easy. However, Outrider was designed for applications with regular control flow, and *lacks the control-flow mechanisms needed to accelerate irregular applications*. Specifically, Outrider uses a global queue for control decisions and requires that all control instructions reside within the first thread to achieve any decoupling. For example, Outrider would not work for BFS, as three out of the four stages have control flow.

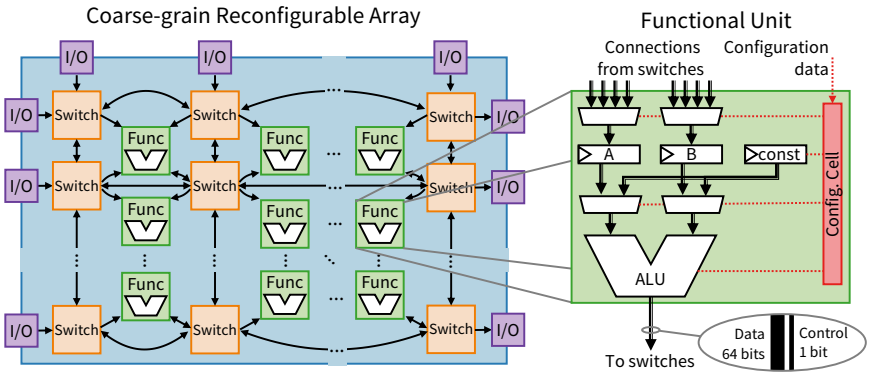


Figure 2-2: Coarse-grain reconfigurable array (CGRA, left) and functional unit design (right).

Helper Threads: Architectures with helper threads [74], including runahead execution [79], slipstream processing [113], and "flea-flicker" two-pass pipelining [12], perform redundant computation so that the main thread benefits from improved branch prediction and prefetched operands. Our goal is to instead create pipelines of threads whose work is never discarded.

2.2.2 Specialized Architectures

General-purpose cores, which continuously change the operations (instructions) on data kept in the same place (e.g., the register file), can be considered *temporal architectures*. On the other hand, a *spatial architecture* is structured as a spatially distributed grid of functional units that each perform a fixed, but configurable, operation. They perform computation by moving data from one functional unit to the next.

Coarse-grain reconfigurable arrays (CGRAs) are spatial architectures; they promise improved computational efficiency by arranging computation as a grid of functional units connected with configurable switches as shown in Figure 2-2. Functional units are composed to perform more complex computations, and data is moved among functional units through a network of simple switches. CGRAs achieve programmability without the need to continuously fetch and decode instructions.

Many CGRA-based architectures have been proposed, either as standalone accelerators [26, 40, 50, 77, 86, 125] or tightly integrated within the pipeline of a general-purpose processor [43, 72, 75, 107, 131]. Stream dataflow [86] provides an interface to express streaming semantics and

uses CGRAs to reduce instruction count. Plasticine [98], for instance, proposes a grid of specialized compute and memory units.

However, a purely spatial approach has disadvantages: the size of the program is limited by the number of functional units, and to achieve high performance, these units must be configured as a *pipeline*, to allow many functional units to be active simultaneously. Prior work relies on CGRAs' rigid internal pipelines for good performance, but their lack of latency tolerance mechanisms presents significant challenges when there is irregularity. Triggered Instructions [92] adds a temporal component to spatial architectures by allowing its functional units to dynamically select one of many operations to execute. Overheads are high, though, because each functional unit only supports a small number of instructions, e.g., sixteen. In Chapter 4, we show how to cheaply add latency tolerance mechanisms to make good use of highly efficient CGRAs on irregular applications.

Graphics processing units (GPUs) are massively data-parallel architectures that improve performance by simultaneously processing data over hundreds or thousands of vector lanes, and prior work has attempted to fit irregular applications to these architectures. UGC [14], G2 [15], IrGL [90], Gunrock [132], and Medusa [145] are frameworks for graph processing on GPUs, but must contend with awkward marshaling of data across GPU lanes. In BFS on a GPU, a neighbor could be added multiple times to the next fringe, but the optimal strategy is to add it just once; Gunrock, for instance, must filter out these duplicates. Some GPUs now include specialized units featuring spatial execution, like NVIDIA's Tensor Cores [23], which, though less programmable, resemble CGRAs. Inter-thread communication has also been extended to GPUs [131], but this communication is in service of the GPU's many data-parallel threads, rather than for making a pipeline.

Application-specific accelerators often employ spatial architectures to improve throughput and reuse, especially when faced with irregular applications' unpredictable compute latencies and memory access patterns. To cope with this, prior techniques specialize their hardware to the problem. For example, SCNN [93] targets compressed sparse convolutional neural networks by directly embedding knowledge of the data structure format into the accelerator. Graphicionado [47] proposes pipelines for graph processing, and Q100 [137] proposes a spatial accelerator for databases. Accelerators for sparse linear algebra, like GAMMA [140],

MatRaptor [110], ExTensor [48], SpArch [143], and OuterSPACE [91], are also organized as spatial architectures. These designs trade a high degree of specialization for high performance in a narrow problem domain. As a result, prior efforts to accelerate these irregular applications often culminate in accelerators that are heavily tuned to the specifics of an application—benefiting only those whose needs are exactly met by the technique.

Indirect prefetchers hide the latency of accesses common in irregular workloads. IMP [139] prefetches accesses of the form `A[B[i]]`, which is insufficient work for BFS, as it has several indirections. Ainsworth and Jones propose a prefetcher tailored to BFS [5] and a more general event-driven prefetcher [6] that can handle multiple levels of indirection. Prodigy [116] uses hardware-software codesign to program a prefetcher to keep its prefetches in pace with the core's execution. However, these prefetchers are complex, taking significant energy to infer dependent accesses from memory traffic; they cannot handle all accesses accurately (like fetching the right set of offsets and neighbors in BFS, which requires iteration [6]); they can handle a limited set of access patterns; and they duplicate much of the work done in the cores, hurting efficiency.

Data structure fetchers are similar to prefetchers, but feed fetched data to cores to avoid duplicating work. HATS [78] performs graph traversals; Widx [62] accelerates hash indexing; and SQRL [66] handles vector, hash table, and tree traversals. SpZip [138] combines range-based scanning, indirection, and compression operators to efficiently traverse data structures before they reach the core. Fetchers avoid the inefficiencies of prefetchers, but are limited to specific data structures and to operations where data structure traversal and computation are not interleaved. Data structures are often tailored for a specific computation; the sparse linear algebra accelerators mentioned above all have hardwired assumptions about the input data structure, e.g., CSR.

2.2.3 Compiler support for pipeline parallelism

Prior work has proposed compiler techniques to exploit pipeline parallelism, but they fail to support irregular applications. Much of this prior work, including StreamIt [42, 121, 122], Piper [68], SGMS [65], and Team Scheduling [94], targets *regular programs*, where the throughput and input/output of each stage are known ahead of time. This information is used to produce fixed thread schedules that maintain load balance and

achieve decoupling with limited buffers. This approach does not extend to irregular applications, as stages incur an unknown and highly variable throughput.

Compilers for specialized architectures [37, 71, 88, 134] help applications take advantage of hardware specialized for specific application domains. These also target neural networks [118], data analytics [38, 130], and signal processing [96, 135].

Pattern-specific compilers and frameworks focus on a specific irregular pattern or data structure. HELIX [19] and HELIX-RC [18] seek to decouple communication from execution. HELIX-RC is a co-designed compiler and architectural support for inter-core communication, but is still limited to parallelizing a single loop. Irregular applications also need the flexibility to parallelize across loop levels (which in turn helps expand the search space for finding the best mapping of irregular applications to stages).

3 Pipette: Improving Core Utilization with Intra-Core Pipeline Parallelism

3.1 Introduction

Irregular workloads, such as graph analytics and sparse linear algebra, use high-performance cores poorly: these workloads suffer from frequent long-latency memory accesses and hard-to-predict branches that limit instruction-level parallelism and render out-of-order execution mechanisms ineffective. In this chapter we focus on non-invasive modifications to existing out-of-order cores to make these challenging workloads run efficiently.

Leveraging multithreaded cores is a common way to improve core utilization. But structuring irregular applications into multiple *data-parallel* threads suffers from three key problems: *(i)* latencies are larger than what can be effectively hidden by a moderately large number of threads per core (e.g., four); *(ii)* operating on disjoint parts of the input increases pressure on the memory hierarchy, limiting performance [44]; and *(iii)* data-parallel implementations suffer from overheads because they need to synchronize through shared memory.

We explore a different and more effective approach to improve utilization in simultaneous multithreading (SMT) cores: exploiting *pipeline parallelism*. As described in previous chapters, a pipeline-parallel program is structured as a series of feed-forward pipeline stages, with each stage executing on a separate thread. Decoupling stages with queues hides latency by allowing producer stages to run far ahead of consumer stages.

Abundant prior work has proposed *decoupled architectures* to exploit pipeline parallelism:decoupled access-execute (DAE) architectures [41,

45, 108, 109, 114, 124], streaming multicores [25, 30, 119], and spatial architectures [87, 92, 100, 137] use queues as latency-insensitive interfaces between cores, threads, or specialized processing elements (Section 2.2). Unfortunately, these architectures are ineffective for our use case because they *(i)* suffer from load imbalance, as they decouple stages across separate cores or processing elements, *(ii)* lack control-flow mechanisms, preventing decoupling of irregular applications, and/or *(iii)* fail to target threads within SMT cores, so their implementations miss opportunities to reuse already-existing resources to implement decoupling cheaply.

To address these limitations, we present *Pipette*. Pipette introduces architectural support for pipeline parallelism within the threads of a multithreaded core. Pipette's novel ISA (Section 3.2) allows threads to define inter-thread queues. By exploiting pipeline parallelism within a multithreaded core, Pipette hides latencies more effectively than the same number of data-parallel threads. Pipeline parallelism's naturally smaller memory footprint alleviates cache pressure and reduces the need to synchronize through shared memory.

By using SMT to time-multiplex stages in the same core, Pipette avoids load imbalance issues that arise when decoupling stages across separate cores or processing elements. Nevertheless, Pipette allows queues to span multiple cores, avoiding limitations on the number of stages (and thus opportunities for decoupling). Pipette also adds out-of-band control flow to keep producer and consumer loops running despite complex control flow. With these features, Pipette effectively decouples irregular applications, unlike prior work.

In addition to using SMT for load balancing, Pipette's microarchitecture (Section 3.3) features two more novel aspects. First, the implementation reuses core structures: it uses the physical register file to implement queues cheaply, avoiding the storage costs of prior techniques. Second, the implementation exposes a decoupled interface that cleanly accommodates *reference accelerators*, simple hardware units that further accelerate common memory accesses like indirections. Whereas prior work proposed coarse-grain specialized units to access complex data structures [62, 66, 78], Pipette enables composable, fine-grain interleaving of accelerated accesses and general-purpose computation.

We evaluate Pipette on applications from graph analytics, sparse linear algebra, and databases (Section 3.5). It substantially outperforms prior work, by gmean 1.9× and up to 3.9× over SMT with data-parallel threads. Moreover, Pipette is more efficient because it achieves high core utilization.

In summary, we make the following contributions:

- We identify the architectural support needed to efficiently express many irregular applications as a pipeline of decoupled stages.
- We present a novel ISA and control flow primitives that enable effective decoupling in these applications.
- We present a novel implementation of this ISA that reuses existing core machinery to achieve decoupling and load-balanced execution cheaply, and adds simple, composable specialized units to accelerate common memory accesses.
- We demonstrate the effectiveness of this approach on a wide range of applications.

3.2 Pipette ISA

Design goals: Pipette's design is driven by three main goals:

1. Providing inter-thread queues at extremely low overheads, so that threads can communicate very frequently, potentially on almost every instruction. This enables a fine-grain slicing of the program into stages, which is crucial, as we saw in Chapter 2. For example, some stages of BFS have as little as one dereference per enqueue and dequeue.
2. Providing control flow primitives that avoid instruction overheads when a serial thread is split into multiple stages. For example, in BFS, stages must synchronize on distance changes. If each stage had to check on every dequeue whether a distance increase was needed, control overheads would negate the benefits of splitting work into stages.
3. Achieving an efficient implementation that reuses existing core structures and accelerates common access patterns.

Pipette's ISA is designed to achieve all these goals. We first discuss how the Pipette ISA achieves extremely low-overhead queues (Section 3.2.1), enabling the first design goal. We then present Pipette ISA's control primitives for efficient inter-stage coordination (Section 3.2.2), enabling the second goal. Section 3.3 presents Pipette's microarchitecture, which efficiently implements the Pipette ISA to achieve all design goals. Table 3-1 details the Pipette instruction set.

Mnemonic	Function
`map_enq $q, %rq`	Map writes to architectural register `%rq` as enqueues to queue `$q`.
`map_deq $q, %rq`	Map reads from architectural register `%rq` as dequeues from queue `$q`.
`unmap %rq`	Revert `%rq` to a non-queue register.
`peek %rd, %rq`	Peek top element from queue `%rq`, writing `%rd` without dequeueing `%rq`.
`enq_ctrl %rq, %rs`	Enqueue a control value (§ 3.2.2).
`skip_to_ctrl %rd, %rq`	Skip to the next occurrence of a control value (§ 3.2.2).

Table 3-1: Pipette instruction set additions. `$q` **is a queue id;** `%rd`**,** `%rs`**,** `%rq` **are general-purpose registers used as a destination, source, or queue.**

3.2.1 Enqueue and dequeue operations

Pipette provides a fixed number of FIFO queues per core (e.g., 16 in our implementation). To minimize overheads, Pipette does *not* have explicit enqueue or dequeue instructions. Instead, each thread can map the input or output of a queue to a general-purpose register. Each write to a queue input register implicitly enqueues the written value, and each read of a queue output register implicitly dequeues it. As we will see in Section 3.3, this register-mapped communication is cheap to implement through register renaming.

It is sometimes useful to read the value at the head of the queue without dequeueing it. To accomplish this, Pipette provides a peek instruction, as shown in Table 3-1.

Pipette queues have a maximum size (e.g., 32 values). To avoid full/empty checks, queues have blocking semantics: dequeue or peek operations to an empty queue block until a value is enqueued, and enqueues to a full queue block until free space is available. We later describe how producers and consumers can use *control values* to coordinate without adding instruction overheads in the common case.

Figure 3-1 shows why register-mapped, implicit enqueues and dequeues are crucial for performance in the enumerate neighbors stage of BFS (presented in Figure 2-1(a)). Figure 3-1(b) shows assembly code corresponding to the excerpt of C code in Figure 3-1(a). If a pipeline-parallel implementation used an `enq` instruction to enqueue values to queues, as done in Figure 3-1(c), it would expand the inner loop by one instruction,

```
int start = offsets[v];
int end = offsets[v+1];
for (int e = start;
     e < end;
     e++) {
  int ngh = neighbors[e];
  // fetch distances[ngh]
  // if unset:
  //   update distance
  //   add to next fringe
}
```

(a) C code enumerating neighbors of a vertex.

```
  ; vertex v's first neighbor
  a2 = &(neighbors[offsets[v]])
  ; vertex v+1's first neighbor
  a3 = &(neighbors[offsets[v+1]])
  ...
loop:
  ; ngh = neighbors[offsets[v]]
  lw    t1, 0(a2)
  ; fetch distance
  ; set if unset
  addi  a2, a2, 4 ; next neigh. addr
  blt   a2, a3, loop ; more neighs?
```

(b) Serial assembly code.

```
loop:
  lw    t1, 0(a2)
  enq   q1, t1 ; overhead
  addi  a2, a2, 4
  blt   a2, a3, loop
```

(c) Pipeline-parallel assembly code using an `enq` instruction, which does not exist in Pipette, to manipulate a queue.

```
  ; writes to t1
  ; enqueue q1
  map_enq q1, t1
  ...
loop:
  ; q1 enq ngh
  lw    t1, 0(a2)
  addi  a2, a2, 4
  blt   a2, a3, loop
```

(d) Pipette code tightens the inner loop by making writes to `t1` enqueue `q1`.

Figure 3-1: Example showing the importance of tight inner loops: adapting the code from Figure 3-1(b) to Figure 3-1(c) using explicit enqueue instructions adds an instruction to a tight inner loop. Pipette addresses this in Figure 3-1(d) with its implicit queue semantics.

a 33% increase for this short loop. This instruction would add pressure to the core frontend (to fetch and decode it) and backend (to execute and commit a micro-op that merely copies a value). Instead, Pipette uses implicit communication, implemented through register renaming, to avoid all these overheads: the Pipette code in Figure 3-1(d) maps register `t1` so that the load instruction directly enqueues `q1`.

3.2.2 Efficient control flow

Producers often need to communicate control flow changes or exceptional conditions to consumers. Doing this through normal enqueues and dequeues would be inefficient. Instead, Pipette provides *control values* (CVs). Control values are similar to other values passed through queues except that they convey changes to control flow instead of application data. To

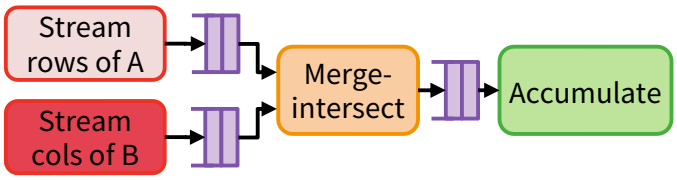


Figure 3-2: Sparse matrix-matrix multiplication (SpMM), with one stage receiving two inputs.

differentiate them from application data, each CV is enqueued with the `enq_ctrl` instruction, which sets the *control bit* for its queue entry.

CVs let programs avoid checks for infrequent conditions by using semantics similar to those of exceptions. Before starting execution, each thread registers a *dequeue control handler*, similar to an exception handler. A thread dequeueing from or peeking at a queue with a control value at its head instead jumps to the dequeue control handler (this jump happens entirely in user level and does not involve the operating system). The dequeue control handler receives the control value and the id of the queue that triggered it. The handler processes the control value, then jumps back to mainline Pipette code to continue computation.

Going back to the BFS example from Figure 3-1(d), it's easy to see why control values make execution efficient: the inner loop in Figure 3-1(d) does not check for termination or level switches in the inner loop. Instead, stages handle these conditions through control values and dequeue control handlers, leaving the inner loops to deal with data values only.

For a more sophisticated use of control values, consider the inner-product sparse matrix-matrix multiply (SpMM) kernel, shown in Figure 3-2. SpMM computes the dot product of a row of `A` and a column of `B` at a time. Both matrices are sparse, so the leftmost stages (stream rows/cols) stream the non-zero *coordinates* of a row and a column at a time. Then, the merge-intersect stage finds the matching non-zeros, which the accumulate stage fetches and accumulates.

Control values make SpMM efficient by letting the stream rows/cols stages *delineate each row and column*. For example, the stream rows stage enqueues all non-zeros for a row of A, followed by a control value denoting the index of the next row, and then the non-zeros of the next row. The merge-intersect code need not check for row or column termination, and the stream rows stage can fetch multiple rows ahead, which is useful as rows often have few non-zeros.

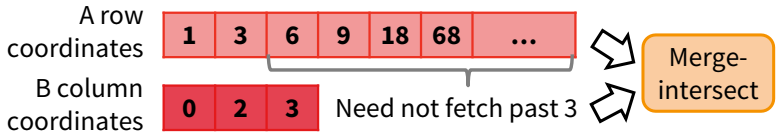


Figure 3-3: In SpMM, an inner product in which the input row and column differ greatly in length.

Consumer-producer coordination: So far, we have seen how producers can delimit data values with control values to communicate with consumers. The dual case is also desirable: consumers may need to communicate with producers. For example, a consumer may discover that the work a producer is enqueueing is no longer useful, and should alter the producer's control flow to reduce unnecessary work.

To achieve this, the `skip_to_ctrl` instruction finds and dequeues the next control value in a queue, discarding all earlier data values. If the queue does not have a control value, `skip_to_ctrl` blocks waiting for one, and the next time the producer attempts an enqueue, it jumps to an *enqueue control handler* instead. This lets the producer redirect control flow and enqueue a control value that unblocks the consumer.

SpMM shows why `skip_to_ctrl` and enqueue control handlers are useful. Figure 3-3 shows an inner product where *A*'s row is much longer than *B*'s column, and the last coordinate in *B*'s column is seen very early in *A*'s row, so no more matched coordinates are possible. It would be wasteful for stream rows to stream the full row of *A*, but only merge-intersect can detect this condition. To address this, when merge-intersect sees the end of *B*'s column (its dequeue control handler fires), it performs `skip_to_ctrl` on the rows queue to skip to the next row. If stream rows is still working on the same row, the queue has no control value, so on the next enqueue, the enqueue control handler of stream rows fires and moves to the next row. If stream rows is already on a later row, then `skip_to_ctrl` lets merge-intersect discard the current row without undue interruptions to stream rows.

In summary, control values and enqueue/dequeue control handlers enable producers and consumers to coordinate out-of-band, in a way similar to *user-level* exceptions. This avoids frequent checks on inner loops that would add significant overheads to the pipelined version. Beyond the two instructions required to enqueue and dequeue control values, this mechanism requires two control registers per thread to store the PCs of enqueue and dequeue control handlers.

3.2.3 Integrating Pipette into the system

Code transformations to use Pipette: We use a simple, systematic procedure to split applications into Pipette stages: we split programs along every long-latency indirect load, starting at the innermost loop and moving outwards. BFS in Chapter 2 demonstrated this procedure.

In this chapter, we transform applications manually (in Chapter 5 we show how to perform this transformation automatically). We transform source code rather than assembly, by using a simple C/C++ API that encapsulates Pipette functionality (e.g., abstracting the mapping and use of queue registers). Irregular applications have non-trivial considerations for pipelining, as their complex indirections may be impacted by aliasing and races.

For example, one such race condition arises in the last stage of BFS, update data. To decouple this stage from the previous one, the Pipette BFS implementation fetches distances in advance. However, this distance may be stale, as the neighbor may have been recently reached from another fringe vertex. It would be incorrect to use the distance as-is. Our manually transformed code uses this distance for an initial check, but if unset, it re-fetches the distance to ensure it was not set in the interim (this second access is cheap, as it hits in the L1).

Pipette is orthogonal to the memory consistency model, and programs behave like normal multithreaded programs.

Finally, if a Pipette application is incorrectly synchronized, it may deadlock. Deadlocks leave user-level threads blocked, but the OS can use interrupts to break those deadlocks (like e.g., a blocked `monitor/mwait` instruction).

Architectural state and context switches: Pipette queues are architectural state, and must be drained and saved across context switches. As is done for FPU state, operating systems (OSs) need not save and restore this state on every system call or interrupt, only when the process is descheduled. As these context switches occur infrequently, saving queue contents represents a negligible fraction of the time spent in OS code.

Draining and refilling queues can be done with normal Pipette instructions. In addition to the OS, debuggers could inspect queues by draining and refilling them.

Privileged code and virtualization: Since threads are an OS abstraction, and threads from multiple processes may share the same core, some of Pipette's operations must be privileged. Specifically, the map and

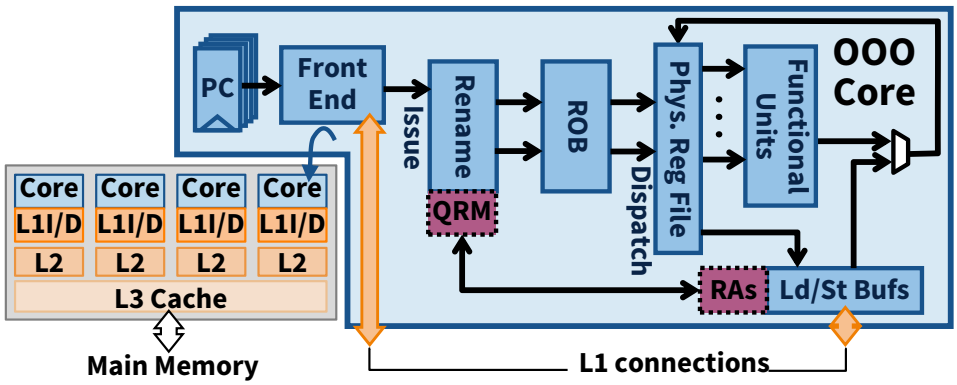


Figure 3-4: Pipette implementation overview and modified out-of-order pipeline. The key modifications, the queue register map (QRM) and reference accelerators (RAs), are shown in purple with dotted borders.

unmap operations and the registration of control handlers must happen though system calls. Similar to virtual memory, the OS can provide each process with a set of virtual queues, which it can then map to physical queue ids within each core. This allows descheduling and rescheduling individual threads in any order. Since each queue is shared between a producer and a consumer thread, the last of the two threads to be descheduled saves the queue's state. Threads can migrate across cores (using cross-core queues, Section 3.3.3). A producer/consumer thread can enqueue/dequeue to a queue while the other thread is descheduled; however, in practice it will quickly block on a full/empty queue. Thus, the OS should co-schedule the threads of a Pipette program, e.g., using gang scheduling [36].

OS-mediated queue mappings prevent accessing queues from other processes. Side channels are possible just like in normal SMT cores; to avoid them, Pipette threads should not be co-scheduled with other processes on the same core.

3.3 Pipette Microarchitecture

Figure 3-4 gives an overview of Pipette's implementation, focusing on its two distinguishing features. First, our Pipette implementation *uses the physical register file to implement queues* (Section 3.3.1). We observe that physical registers are underutilized in irregular applications, where deep out-of-order execution is not efficient. This enables a cheap imple-

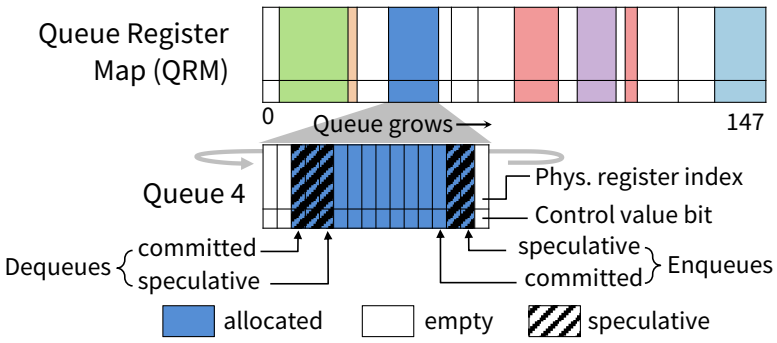


Figure 3-5: The Queue Register Map (QRM) tracks the physical registers of each Pipette queue. Each queue is managed as a circular buffer.

mentation that leverages existing OOO structures: physical registers and register renaming. Second, we introduce *reference accelerators* to speed up common access patterns (Section 3.3.2). We also introduce *connectors* to enable cross-core queues (Section 3.3.3), and evaluate Pipette's implementation costs (Section 3.3.4).

3.3.1 Register-based inter-thread queues

Pipette maintains queues within the *physical register file*, and adds minor changes to register renaming to implement FIFO queue semantics. Pipette prevents queues from starving threads of physical registers by sizing each queue and limiting the space all queues may collectively occupy. Since queues are embedded within speculatively managed structures, we first explain the basic Pipette bookkeeping structure, then how it interacts with speculative execution.

Basic operation: Figure 3-5 shows the Queue Register Map (QRM), the structure that tracks the state of all queues. The QRM has as many entries as the maximum capacity of all queues. Each queue takes a contiguous chunk of entries (shown in different colors in the figure), and manages it as a circular buffer. The chunk associated with each queue determines its capacity. This mapping is configurable by the OS, but cannot change while queues are active.

Figure 3-5 also shows how each queue is managed. Each queue has both *speculative* and *committed* pointers for *head* and *tail*. Enqueues happen to the tail of the queue, and dequeues happen from the head. We restrict each queue to be point-to-point, so there is a single enqueuer and dequeuer thread.

Each entry between the head and tail pointers tracks the physical register index that holds the enqueued value. Moreover, each entry has a *control value bit* that denotes whether the entry holds a control value (enqueued with `enq_ctrl`).

The QRM is designed to require simple changes to register renaming. Enqueue operations are nearly identical to normal register writes. On issue, the rename stage acquires a free register index from the freelist, and uses it to store the enqueued value. As usual, the reorder buffer (ROB) stores the *previous* physical register index. The only difference is that, on commit, the ROB *does not free* the previous physical register index. Instead, the QRM manages it, as it is part of a queue.

Dequeue operations are also very similar to reads. For each dequeue-mapped queue, the thread's register map simply holds the index for the *head* of the queue. A dequeue simply uses this value, and additionally modifies the register map to point to the next register in the queue, supplied by the QRM. On commit, the QRM returns the register to the freelist. Finally, peek operations are exactly like normal reads.

Speculative value management: Because registers are written and read speculatively, there are multiple value management options. We choose the simplest one: *enqueued values cannot be dequeued until they are non-speculative*.

This leads to a simple implementation: QRM's *speculative* head and tail pointers are the only eagerly managed values. Each enqueue advances the speculative tail pointer on issue, and the committed tail pointer on commit; similarly, each dequeue advances the speculative head pointer on issue, and the committed head pointer on commit. The queue is considered *empty* if the speculative head is about to catch up with the committed tail, and *full* if the speculative tail is about to catch up with the committed head. The issue stage stalls the thread on enqueues to full queues or dequeues from empty queues.

Recovery from misspeculation simply requires rolling back the speculative head and tail pointers, as well as releasing the registers from rolled-back enqueues to the freelist.

A key benefit of consuming only committed values is that *misspeculation in a producer thread does not propagate to the consumer thread*. This allows us to implement Pipette with simple changes to the issue and rename stages.

We also tried a more complex variant of Pipette that allowed dequeues to consume still-speculative enqueued values. This version barely im-

proved performance (by about 1% on average), and occasionally caused minor performance degradations. Intuitively, this result makes sense because the point of Pipette is to keep threads decoupled, so while allowing dequeues to dip into the speculative region of the queue may get some out-of-order benefits, in well-decoupled programs producers should already run far ahead of consumers.

Issue logic modifications: Pipette requires minor changes to the issue logic. First, the per-thread issue logic stalls on a dequeue from an empty queue or an enqueue from a full queue. Second, every dequeue of a control value triggers a jump to the dequeue control handler. For simplicity, we reuse the exception logic to implement this redirection.

Our current Pipette implementation does not change the thread prioritization logic. We use the standard ICOUNT policy [128] to avoid issue queue clog. Further gains might be achieved by controlling thread priorities to increase decoupling, e.g., by prioritizing producers over consumers. However, we find Pipette works well with ICOUNT, and leave exploration of more advanced issue policies to future work.

3.3.2 Accelerating common access patterns

By exposing a queue-based interface, Pipette makes it easy to add specialized units to accelerate long-latency memory accesses. Pipette achieves this with *reference accelerators* (RAs), simple configurable units that perform indirect loads and communicate with threads through queues.

Benefits: BFS (in Figure 2-1(d)) showcases the two key benefits of RAs. First, some stages, like fetch distances in BFS, are very simple, performing indirect or strided accesses. Using a thread for such simple work is overkill: a simple RA can perform them much more efficiently. Second, to get decoupling, Pipette divides the code across each long-latency indirection. While this lets producers run ahead of consumers, *producers still suffer from long-latency loads*. For example, the process current fringe stage in BFS issues loads to the `offsets` array. These loads take a long time to commit and stress the ROB, limiting memory-level parallelism. RAs allow offloading these *producer* long-latency accesses. This results in producers with short, tight loops that do not stress OOO resources, improving performance. For example, in our RA-enhanced BFS, the process current fringe stage passes `v` to an RA, which fetches `offsets[v]` and `offsets[v+1]` autonomously and non-speculatively, producing `start` and `end`.

Interface: Each RA is a configurable unit with a single input and output queue. The RA takes in a stream of input elements, uses them to perform indirect accesses, and places the resulting data in its output queue. RA accesses are independent from those of general-purpose threads. With respect to consistency, programs simply see each RA as a separate thread.

RAs are configured once, by specifying which queues to use, a starting address `A`, an element size `S`, and the access *mode* with can be *indirect* or *scan*. The RA interprets `A` as an array with elements of size `S`. In *indirect* mode, the RA takes a stream of indices at its input, and for each index `i`, it fetches `A[i]`. In *scan* mode, the RA takes a stream of starting and ending indices at its input, and for each pair of input values {`start`, `end`}, it fetches elements `A[start:end-1]`.

We find that these simple modes cover most indirection patterns and benefit all our applications. For instance, in BFS, the indirect mode covers the first and third stages, and the scan mode covers the second stage.

Implementation: RAs use existing core machinery. RAs opportunistically use spare rename and register bandwidth, and manipulate the QRM like ordinary threads on enqueues and dequeues. When performing memory accesses, RAs use the load/store unit and use virtual addresses. Each RA has a small completion buffer to track outstanding loads (Section 3.3.4 presents implementation costs). On a virtual memory exception, the core interrupts the producer thread associated with the RA.

3.3.3 Extending Pipette to cross-core queues

We have so far described how queues work within a core, but allowing queues to span threads in multiple cores is desirable for three reasons. First, although Pipette's main goal is to improve core utilization, achieving effective decoupling may require more stages than a core has threads. Second, as we will see later (Section 3.5.6), Pipette can scale and balance work across cores in new ways, by using inter-core queues to improve locality and avoid shared-memory synchronization costs. Third, as Section 3.2.3 discussed, it is desirable to let the OS schedule threads individually, in separate cores if needed.

We achieve this through *connectors*, simple hardware structures that stream a queue from a producer to a consumer core. Producer and consumer threads are both given intra-core queues. The connector is a simple FSM that sits on the producer's core. It has a similar but simpler implementation than RAs: rather than interacting with the load/store

Component	QRM entries 148 × 9 bits	QRM pointers 64 × 8 bits	Handler PCs 8 × 64 bits	**Pipette total**	Int. PRF (for comparison)
Size (bits)	1,332	512	512	**2,356**	13,568

Table 3-2: Pipette storage requirements.

unit, it just sends values from the producer to the consumer core, using credit-based flow control to avoid saturating the on-chip network and strictly limits the receiver queue's state to its capacity. When descheduling a consumer thread, the OS must wait for its connectors to quiesce; this requires a simple teardown protocol on top of credit-based flow control.

3.3.4 Implementation costs

Pipette's storage and logic additions impose minimal overheads. Table 3-2 summarizes Pipette's storage requirements. In our configuration, Pipette can map up to 148 physical registers, and takes 1844 bits (231 bytes). This is only 14% of the physical register file, showing the benefits of leveraging physical registers to implement queue storage. Beyond the QRM, 512 bits (64 bytes) are required for the per-thread enqueue and dequeue control handler PCs. Overall, only 2356 bits (295 bytes) of additional storage are needed to implement Pipette, a small overhead for a modern core.

RAs, the other hardware addition, are small. We write complete RTL for RAs, including configuration registers, address generation, and a 32-entry completion buffer. We synthesize RAs using yosys [136] and the 45 nm FreePDK45 library [51]. Four RAs take 0.0014 mm^2 at 45 nm, adding an estimated 0.007 % to core area, a tiny overhead.

3.4 Experimental Methodology

3.4.1 Simulated System

We implement Pipette on a detailed event-driven, cycle-level simulator based on Pin [73]. Table 3-3 lists the parameters of our simulated system, whose cores are modeled after Intel's Skylake [32], scaled to 4 SMT threads from the usual two. Core structures are sized as in Skylake; we grow the physical register file (PRF) from 180 to 212 entries to accommodate the 32 architectural registers of the two extra threads. Thus, the

Cores	1 or 4 cores, 3.5 GHz, x86-64 ISA, Skylake-like: 6-wide out-of-order issue, 224-entry ROB, 97-entry issue window, 72-entry load buffer, 56-entry store buffer, 212 integer physical registers, 168 vector physical registers; 4-thread SMT with ICOUNT issue policy and dynamically shared ROB, issue window, PRF, and LSQs
Pipette	QRM with 148 physical register entries, 16 queues max; 4 RAs; 4 connectors; queues are sized 24 elements deep by default
L1 cache	32 KB/core, 8-way set-associative, 4 cycle latency
L2 cache	256 KB/core, 8-way set-associative, 12 cycle latency
L3 cache	2 MB/core, 16-way set-associative, 40 cycle latency
Main mem	120-cycle minimum latency, 2 controllers, 25 GB/s each

Table 3-3: Configuration parameters of the evaluated system.

PRF entries left for renaming are the same as in Skylake. We extend cores to faithfully simulate Pipette additions, with the configuration shown in Table 3-3. We use McPAT [70] to model core and uncore energy at 22 nm, and Micron DDR3L datasheets [76] to model main memory energy.

We evaluate 1- and 4-core systems. Since Pipette's main goal is to improve core utilization, we first compare Pipette and data-parallel programs on a single 4-thread SMT core. Then, we compare 1-core, 4-thread Pipette with a baseline decoupled architecture: a 4-core streaming multicore. Finally, we show that Pipette also scales across cores.

3.4.2 Benchmarks

We evaluate Pipette on six applications from graph analytics, sparse linear algebra, and databases. For each application, we start from a state-of-the-art implementation that includes serial and data-parallel versions. We derive the Pipette version of each benchmark from the serial version. The wide variety of Pipette applications highlights its generality and the abundance of pipeline parallelism.

Breadth-first search (BFS), first described in Chapter 2, determines the distance of graph vertices to a source vertex. We base our implementation on PBFS [69].

Connected components (CC), **PageRank-Delta (PRD)**, and **Radii estimation (Radii)** are graph algorithms from the Ligra framework [106]. CC uses multiple invocations of BFS to discover graph connectivity. PRD is a PageRank variant that only visits vertices whose PageRank value changes by more than a certain amount. Radii launches several breadth-

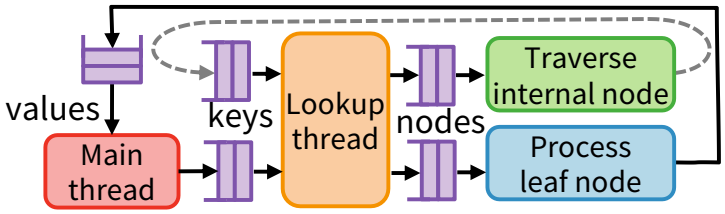


Figure 3-6: Silo, with bounded feedback loops.

Domain		Graph	Vertices	Edges
Human collaboration	(Hu)	coAuthorsDBLP-symmetric	299K	1.9M
Dynamic simulation	(Dy)	hugetrace-00000	4.6M	14M
Circuit simulation	(Ci)	Freescale1	3.4M	19M
Internet graph	(In)	as-Skitter	1.7M	22M
Road network	(Rd)	USA-road-d-USA	24M	58M

Table 3-4: Input graphs, sorted by number of edges.

first searches from random points in the graph to estimate the radii of its vertices. These algorithms process only a subset of graph vertices in each iteration, so their memory access patterns are very irregular. The pipelines for these algorithms resemble the pipeline for BFS in Figure 2-1(d).

Sparse matrix-matrix multiplication (SpMM), introduced in Section 3.2, is a key component of sparse linear algebra, and its merge-intersection parallels similar operations on databases.

Silo [127] is an in-memory database. Silo is dominated by lookups to B+tree indexes. Our Pipette implementation, shown in Figure 3-6, pipelines multiple tree traversals. The lookup thread performs tree lookups level by level, requeueing the key and lookup status in its input queue if the lookup needs to go to the next level by traversing internal nodes. This recursive process manifests as a cycle in the pipeline diagram (dashed gray line).

Silo shows that Pipette programs can feature cycles in their application graphs. As in dataflow systems, we avoid application-level deadlock as long as cycles are bounded—in this case, each lookup thread re-enqueues *at most* one element for each element it processes.

Input sets: Graph applications use five large, real-world graphs that include road networks, Web connectivity graphs, and academic collaboration graphs, listed in Table 3-4. SpMM uses six diverse sparse matrices, listed in Table 3-5. Silo uses the YCSB-C workload [24] on a 52 GB dataset.

Domain		Matrix	Size ($n \times n$)	Avg. nnz/row
File sharing	(FS)	p2p-Gnutella31	62,586	2.4
Graph as matrix	(Gr)	amazon0312	400,727	8.0
Collaboration	(Co)	ca-CondMat	23,133	8.1
Gel electrophoresis	(GE)	cage12	130,228	15.6
Electromagnetics	(EM)	2cubes_sphere	101,492	16.2
Fluid dynamics	(FD)	rma10	46,835	49.7
Structural	(St)	pwtk	217,918	52.9

Table 3-5: Input matrices, sorted by average non-zero elements per row. Fifer (Chapter 4) uses FS instead of Co; Pipette does not use FS.

On some of PRD, Radii, and SpMM's largest inputs, we use *iteration sampling* to keep simulation times reasonable: we simulate only a subset of iterations, uniformly distributed. Even with sampling, simulated periods are long (*e.g.*, ∼3 billion cycles per phase of PRD), so no warmup is needed. For all other benchmarks, we simulate the full algorithm.

Reference accelerators: We build Pipette benchmark variants with and without RAs. We apply RAs systematically, offloading every producer load to an RA as described in Section 3.3.2. All workloads benefit from RAs. We report results with RAs on by default; Section 3.5.5 studies the impact of RAs.

3.5 Evaluation

We first analyze Pipette's single-core performance and compare Pipette and data-parallel implementations. Then, we compare Pipette to a 4-core streaming multicore to show the utilization and efficiency benefits of time-multiplexing stages among threads. We then report microarchitectural efficiency metrics, study sensitivity to configuration parameters, and conclude by studying Pipette on multiple multithreaded cores.

3.5.1 Pipette vs. data-parallel implementations

Figure 3-7 (left) summarizes the performance of the serial, data-parallel, and Pipette versions of all benchmarks. These versions use a single 4-thread SMT core. Performance is reported as speedup *over the data-parallel version* (not serial), averaged (gmean) across inputs. Figure 3-7 shows that Pipette substantially outperforms the data-parallel versions, by

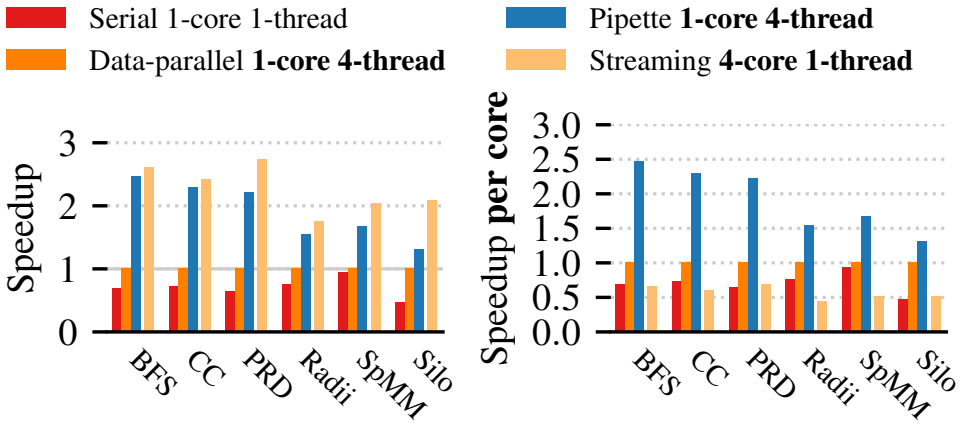


Figure 3-7: Performance of Pipette implementations on a single 4-thread core and a *4-core* streaming multicore, as well as *performance per core*.

1.9× gmean across applications; by up to 2.5× for an application (BFS); and by up to 3.9× on particular inputs (Figure 3-11).

3.5.2 Pipette vs. cross-core decoupling

Beyond data-parallel implementations, we compare Pipette against a baseline decoupled architecture, a streaming multicore. Pipette's key benefit over prior decoupled architectures is its ability to time-multiplex stages across the same core to achieve load balance and high utilization. To focus on evaluating this effect, we model the streaming multicore simply as multi-core, single-thread Pipette: the streaming multicore uses 4 single-threaded cores, and benefits from Pipette's ISA and features. *This includes reference accelerators*, even though they are our contribution.

Figure 3-7 compares the performance of Pipette on a *single multi-threaded core* to the *single-threaded 4-core* streaming multicore. To measure how effectively Pipette uses its core resources, we also show the performance *per core*.

Figure 3-7 shows that, while the streaming multicore outperforms Pipette, it does so by relatively small margins given that it uses *four times the cores*: BFS, CC, Radii, and SpMM perform similarly, and Streaming is 24% faster on PRD and 59% faster on Silo. Overall, Streaming is only 22% faster than Pipette.

This happens because *load imbalance hampers utilization of decoupled cores in irregular applications*: the right plot in Figure 3-7, which normalizes by the number of cores, shows that each core in the Streaming

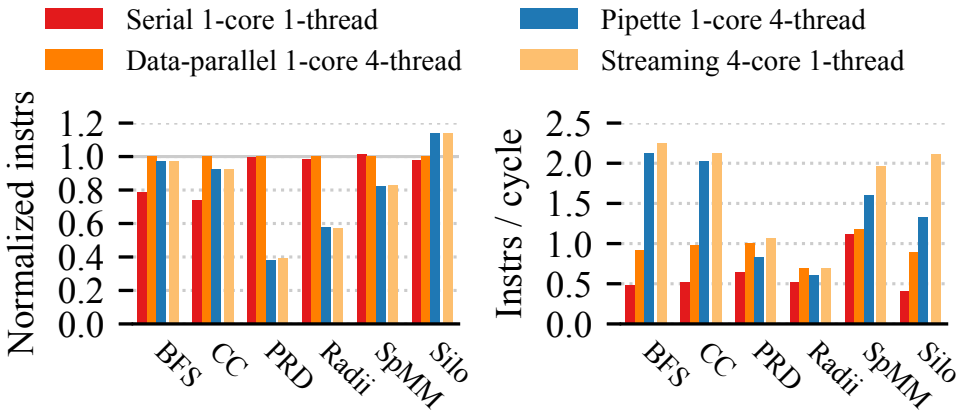


Figure 3-8: Instructions executed (lower is better) and average IPC (higher is better), averaged across program inputs, for 1-core data-parallel and Pipette as well as 4-core Streaming.

baseline contributes similar performance as Serial. This is because, unlike in regular applications where stages proceed at matched rates, irregular applications have highly variable utilization across stages.

3.5.3 Pipette is resource-efficient

To further understand these results, Figure 3-8 compares the instructions executed by each benchmark version, relative to those of the data-parallel implementation (left graph, lower is better) as well as instructions per cycle (IPC, right graph, higher is better). Each group of bars shows results for a single benchmark, averaged across all inputs.

These figures reveal that Pipette consistently uses cores efficiently, but the reasons are application-dependent:

- In BFS and CC, Pipette's improvement mainly comes from its dramatic gain in IPC. Pipette executes nearly the same instructions as the sequential code, whereas the data-parallel versions incur some synchronization overheads.

- In PRD and Radii, Pipette's benefits mainly come from reducing the number of instructions. Instruction overheads in these benchmarks stem from synchronization overheads. (These benchmarks come from Ligra, and unfortunately, the serial Ligra version also carries these overheads.) Pipette avoids this and reduces instruction count by up to 3.2×. Thus, while Pipette's IPCs are slightly lower, each instruction does more work.

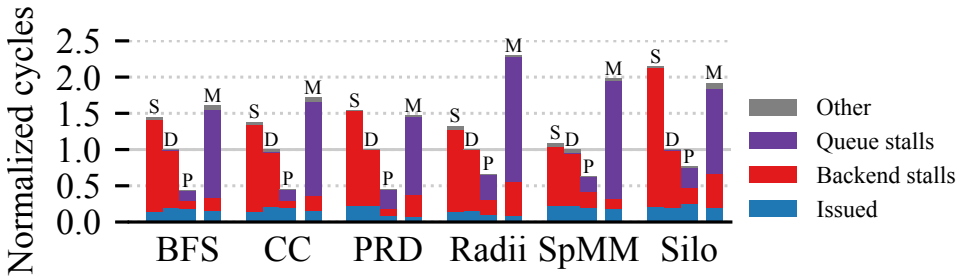


Figure 3-9: Breakdown of cycles spent executing each application, normalized to data-parallel and averaged across inputs. (S: Serial, D: Data-parallel, P: Pipette, M: streaming Multicore)

- In SpMM, Pipette's benefits stem from both increasing IPC and reducing the number of executed instructions.
- Pipette improves Silo by increasing IPC, which is slightly attenuated by a modest increase in executed instructions.

Figure 3-9 gives more insight into the factors contributing to IPC by showing a breakdown of cycles spent by cores, derived using the CPI stack methodology [35]. Each group of bars reports breakdowns of each variant across benchmarks (averaged across inputs), relative to the data-parallel baseline. Each bar within a group reports cycles for one technique, broken down in cycles spent *(i)* issuing micro-ops, and waiting on *(ii)* backend stalls (including memory latency), *(iii)* full or empty queues (for Pipette and Streaming), or *(iv)* other stalls (e.g., frontend).

Figure 3-9 shows that the serial and data-parallel versions are limited by backend stalls, which are caused by long memory accesses. Meanwhile, the streaming multicore is limited by queue stalls, i.e., load imbalance. By contrast, Pipette incurs few stalls: proper decoupling dramatically reduces backend stalls, and time-multiplexing stages in the same core keeps queue stalls low.

Finally, Figure 3-10 shows the breakdown of energy consumption of each variant across benchmarks (averaged across inputs), relative to the data-parallel baseline. Pipette is the most efficient variant for BFS, CC, PRD, Radii, and SpMM, reducing energy by up to 2.2× (PRD). Pipette's savings mainly come from reducing dynamic core energy (fewer instructions) and reducing static energy (fewer cycles, as performance is higher). Figure 3-10 shows that the streaming multicore is not efficient overall, as it suffers from high static energy due to poor core utilization.

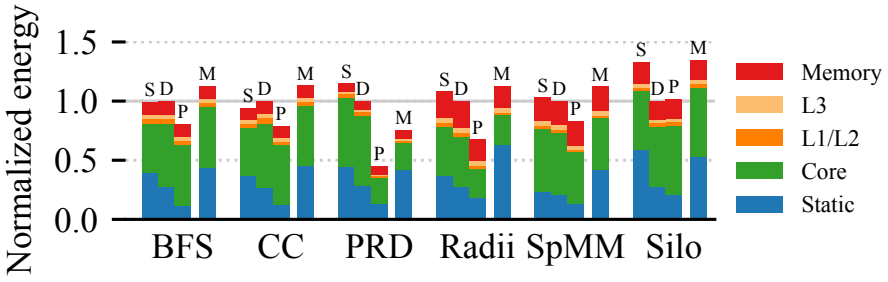


Figure 3-10: Breakdown of energy consumed by each application, normalized to data-parallel and averaged across inputs. (S: Serial, D: Data-parallel, P: Pipette, M: streaming Multicore).

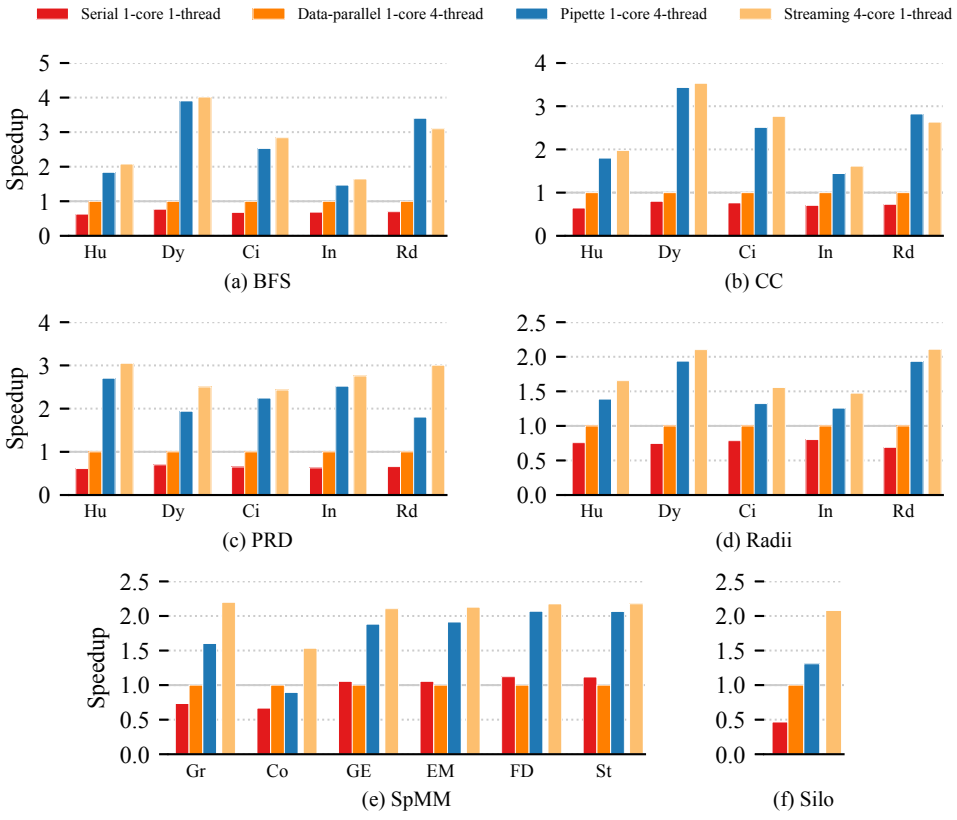


Figure 3-11: Per-input performance of all evaluated applications.

3.5.4 Per-input results

Figure 3-11 reports the performance of all variants across each input. Per-input results reveal some interesting behaviors.

BFS: Pipette widely outperforms the serial and data-parallel BFS versions, as shown in Figure 3-11(a). Pipette outperforms the data-parallel BFS by gmean 2.5× and by up to 3.9×.

Speedups mainly depend on two factors: graph size and average degree. Pipette yields more benefits in larger graphs, where misses are more frequent; and Pipette is more efficient at enumerating small sets of edges than conventional code, where hard-to-predict control flow is inefficient. Thus, Pipette achieves the best speedups in low-degree graphs (`Dy` and `Rd`).

CC, PRD, and Radii (Figure 3-11(b-d)) show similar trends to BFS: Pipette consistently outperforms the data-parallel versions, of gmean speedups of 2.3×, 2.2×, and 1.5×.

Unlike BFS, these algorithms operate on a fraction of the graph that changes slowly over iterations, so they get better reuse. However, synchronization is more complex, so the data-parallel versions suffer from costly overheads that add substantial extra work. Thus, Pipette's low instruction counts contribute substantially to speedups, as explained above.

SpMM (Figure 3-11(e)) shows more mixed performance results: Pipette outperforms data-parallel SpMM by up to 2.1×, but it is slightly slower than data-parallel SpMM on one input.

The slight slowdown on the `Co` input results from a combination of two factors. First, `Co` has only 8 non-zeros per row (Table 3-5), so control values are common and the merge-intersect stage spends a significant fraction of time in the dequeue control handler. Second, `Co` is a small matrix that fits on-chip, so decoupling yields limited benefits. This result shows that, while Pipette's control flow mechanisms work well even under frequent control flow, in some cases data parallelism (i.e., processing several row-column pairs in parallel) is slightly better. An adaptive implementation could detect this and switch between Pipette and data-parallel versions.

Finally, **Silo** (Figure 3-11(f)) yields a modest 24% gain for Pipette. Silo's high-radix B+tree is cache-friendly, and the data-parallel version hides occasional misses reasonably well, but Pipette achieves further decoupling and hence performs better.

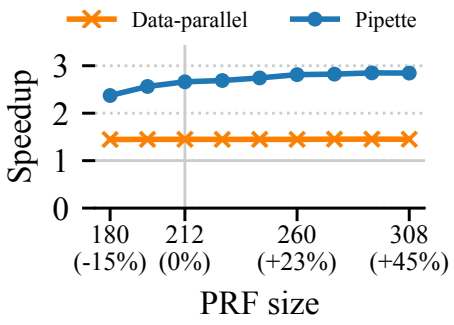


Figure 3-12: Performance of 4-thread Pipette and data-parallel over PRF sizes. Gmean speedups are over Serial with default parameters (0%).

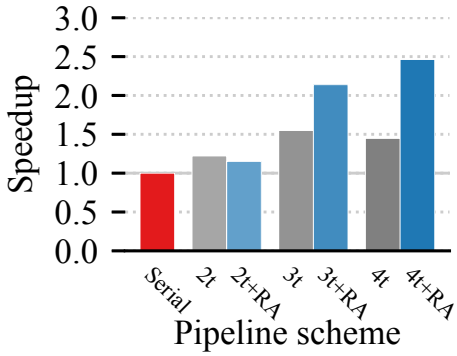


Figure 3-13: Sensitivity studies measuring performance of BFS while varying number of stages and use of RAs. Higher is better; speedups are geomean over all inputs.

3.5.5 Sensitivity studies

Sensitivity to physical register count: Figure 3-12 compares the performance of Pipette and data-parallel variants as the physical register file (PRF) changes. The graph shows the gmean speedup over all benchmarks, relative to the serial version with the default 212-entry PRF. We scale the PRF from 180 to 308 entries. We scale Pipette's queues proportionally with PRF size, so larger PRFs result in larger queues and thus more decoupling.

Figure 3-12 shows that implementing queues using physical registers is a good choice. Pipette maintains a substantial performance advantage over the full range of PRF sizes. Moreover, while data-parallel benchmarks are insensitive to PRF capacity (as they are bound by backend stalls, the issue queue and ROB limit them), Pipette can modestly benefit from larger PRFs, which improve decoupling.

Pipette vs. software techniques: Pipette programs feature fine-grain stages that communicate extremely frequently: as many as one in six register file reads/writes are enqueues/dequeues (BFS, SpMM) or as few as one in 27 (Silo). This shows the need for hardware support: state-of-the-art software queues take tens of cycles per enqueue/dequeue [7], so using them instead of Pipette would add very high overheads.

Effect of the number of stages on decoupling: Figure 3-13 examines the performance of 2-, 3-, and 4-stage versions of BFS. This shows that proper decoupling requires more than two stages: the best-performing

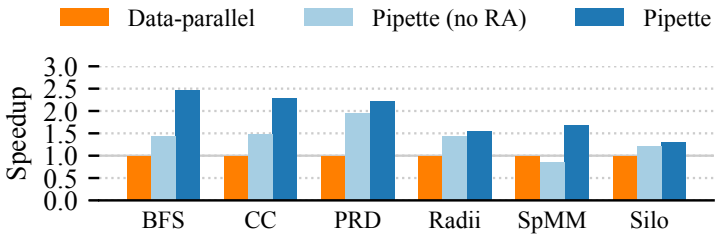


Figure 3-14: Performance of Pipette without and with reference accelerators (RAs).

implementation is the 4-stage one. We also measure the effect of RAs these pipelines. Without RAs, performance peaks in the 3-stage implementation, as the 4-stage version has higher ROB pressure. The 2-stage implementation decouples the distance updates, but leaves the fringe accesses and neighbor enumeration needlessly tightly coupled. RAs and decoupling go hand-in-hand; the 2t+RA point demonstrates the pitfalls of adding RAs without first properly decoupling the application. A distance fetched by an RA can become stale if that distance is updated later. This race condition requires an extra check in the second stage, whose latency cannot be overcome by the limited decoupling. When RAs offload *all* long-latency loads, they reduce backend pressure and enable peak performance with 4 stages—a 1.7× speedup over the conventional 4-stage pipeline.

Effect of RAs: Figure 3-14 shows Pipette's per-application performance without and with RAs. BFS, CC, and SpMM benefit substantially from RAs, whereas PRD, Radii and Silo see modest gains. Overall, RAs improve performance by gmean 38%, by reducing instruction count and core backend pressure.

3.5.6 Multicore Pipette

Finally, we compare the performance of a different Pipette BFS variant on a 4-core system, showing that data and pipeline parallelism are complementary, and that Pipette's techniques scale outside of the core.

Figure 3-15 (right) compares the performance of four different BFS implementations: Serial (1 core, 1 thread), data-parallel (with 4 cores and 4 threads/core), streaming single-threaded (with each BFS stage running on a separate core), and the Pipette multicore BFS shown in

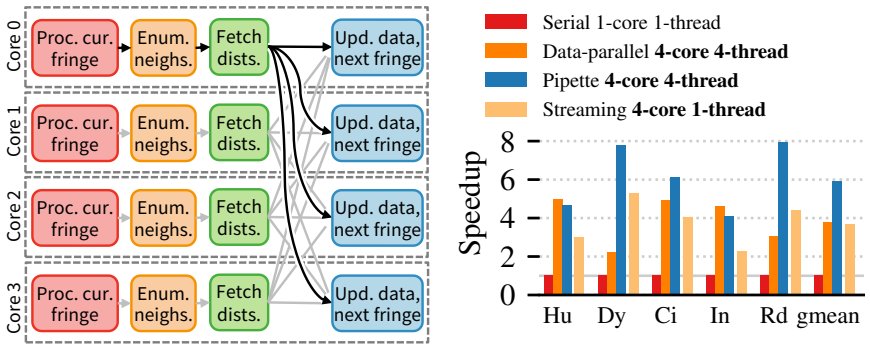


Figure 3-15: Multicore and multithreaded BFS Pipette implementation, and its performance compared to serial, data-parallel, and streaming versions.

Figure 3-15 (left). In this version, all stages are replicated across cores to achieve load balance, so each core processes a fraction of the fringe. Furthermore, instead of using shared-memory synchronization, neighbors are partitioned across cores and sent to be processed at their corresponding core, as shown by the cross-core communication in Figure 3-15 (left) among the last two stages.

Figure 3-15 shows that, like in the single-core case, the data-parallel version leaves performance on the table, with a gmean speedup of 3.8× vs. serial despite using 16 threads on 4 cores. The streaming version sometimes outperforms the data-parallel version, though it is limited by load imbalance as each core runs a single stage. Finally, the Pipette multicore version performs best, achieving a gmean speedup of 5.9×.

To analyze scalability, we also evaluate a 16-core system for a total of 64 threads. At 16 cores, the data-parallel version is 1.5× faster than the 4-core system, while Pipette is 1.8× faster (and thus 1.9× faster than 16-core data-parallel). While Pipette scales better, at 64 threads BFS suffers synchronization overheads that limit its scalability.

In summary, this result shows that Pipette continues to be attractive in multicore systems. Multicore Pipette achieves high core utilization and avoids the synchronization overheads of the data-parallel implementation by using connectors to join queues across cores.

3.6 Summary

Applications with irregular access patterns and control flow have latent pipeline parallelism that can be exploited to improve core utilization.

However, prior decoupled architectures are insufficient for these irregular applications. We have presented Pipette, which achieves high utilization by exploiting fine-grain pipeline parallelism within the threads of a multi-threaded core. This new regime not only allows fast and inexpensive local communication, but also sidesteps the load balancing issues that affect prior decoupled architectures and enables a cheap implementation that reuses otherwise-idle registers and accelerates common access patterns. As a result, Pipette achieves significant speedups on several applications over a wide variety of inputs. Pipette thus offers a high-performance, practical substrate for pipeline-parallel programs.

4 Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures

4.1 Introduction

General-purpose processors, like the ones used to implement Pipette, are woefully inefficient: they routinely spend less than 1% of their energy executing computation [49], and spend most of their energy and area on instruction interpretation overheads and general but expensive latency-tolerance mechanisms, like out-of-order execution and speculation. With Moore's Law waning, it is crucial to reduce this bloat. While specializing hardware to each application achieves maximum performance and efficiency, it is inflexible. Ideally, we want architectures that approach the efficiency of full specialization, while being programmable and capable of executing a wide range of applications.

Coarse-grain reconfigurable arrays (CGRAs) are a promising approach to achieve this goal. CGRAs implement a sea of spatially distributed functional units that can be configured and connected with switches to create high-throughput datapaths. Prior work has explored and implemented a wide range of CGRA designs, either as standalone accelerators [26, 50, 77, 98, 105] or tightly integrated coprocessors [43, 75, 107, 131].

Unfortunately, CGRAs are restricted to *regular applications*, i.e., those with structured access patterns and control flow, like dense linear algebra. These features are necessary to produce a high-performance pipeline that can be spatially and statically mapped to a CGRA fabric. By contrast, CGRAs struggle with *irregular applications*, i.e., those with unstructured

memory accesses (like indirections) and control flow (like data-dependent branches). These applications arise in many important domains, like graph analytics, sparse linear algebra, sparse deep learning, and databases. CGRAs are ill-equipped to handle these operations: faced with a long-latency operation, like a cache miss, they simply stall; and even if misses are rare, irregular control flow causes load imbalance that leaves most of the fabric idle.

In this chapter, we present *Fifer*, an architecture and compilation technique that makes irregular applications efficient on CGRAs. Fifer combines two key techniques:

Extracting regular stages from irregular applications: We show that an application's irregular accesses and control can be *decoupled* from its regular computation, which can then be efficiently processed by the CGRA. This approach divides the computation into a pipeline, i.e., a feed-forward network of stages. These stages are connected with latency-insensitive channels, like FIFO queues, to tolerate unpredictable latencies. Importantly, this approach produces *regular* stages that can be turned into high-throughput datapaths mapped to a CGRA fabric and confines irregularity to happen across pipeline stages.

Despite this transformation, conventional CGRAs are still inefficient: while each stage maps well to a CGRA fabric, irregularity causes wide variations in work across stages. CGRAs, being pure spatial architectures, cannot accommodate these variations and suffer from load imbalance. This necessitates Fifer's second key technique:

Dynamic temporal pipelining: To avoid load imbalance, Fifer *temporally pipelines* CGRA-based architectures: multiple stages are time-multiplexed into the same CGRA fabric, with a scheduler dynamically choosing which stage to run based on the availability of work. This avoids load imbalance by dedicating more cycles to stages with more work. To be efficient, reconfigurations should be infrequent (occurring every few hundred cycles), and brief (lasting tens of cycles). We introduce fast reconfiguration mechanisms to make this possible.

Prior work has also explored adding time-multiplexing to CGRAs: Triggered Instructions [92] maps multiple operations onto each element of the fabric, and each element selects a ready operation to execute each cycle. This fine-grain time-multiplexing tolerates imbalance, but requires substantial additions to a CGRA to support such frequent switching. By contrast, Fifer reconfigures at coarser granularity: switching between

large, regular chunks of operations over several cycles. Because our program transformation coarsens work into chunks that are switched less often, we can use much simpler scheduling hardware to achieve load balance on the CGRA fabric. By analogy with general-purpose cores, Triggered Instructions is the CGRA counterpart to fine-grained multithreading, whereas Fifer is the CGRA counterpart to coarse-grained multithreading.

To implement Fifer, we make three simple modifications to an existing CGRA: *(1)* frequent and rapid reconfiguration, *(2)* buffers acting as queues decoupling spatial and temporal pipelines, and *(3)* logic to further decouple irregular memory accesses. Altogether, these modifications present an interface similar to that of Pipette's and allow us to extend Pipette's insights to CGRAs. We prototype Fifer in a system with multiple *processing elements* (specialized cores), each with its own CGRA fabric and private cache. We show that Fifer scales well to large systems by combining spatial and temporal pipelining. We implement Fifer's major components in RTL and show that its additions are simple and cheap.

Our evaluation shows Fifer outperforms an OOO multicore by over gmean 17× while using much less area. We also compare Fifer to a CGRA-based architecture that cannot time-multiplex stages. Fifer outperforms this baseline by gmean 2.8×, and by up to 5.5×, across several challenging irregular applications.

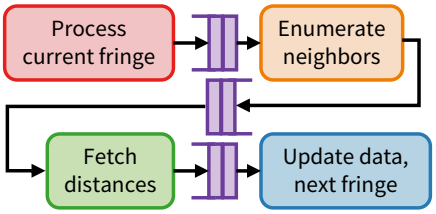
In summary, we make the following contributions:

- We identify the challenges of irregular applications on reconfigurable spatial architectures.
- We present a technique to decouple applications across sources of irregularity for effective CGRA mappings.
- We introduce Fifer, a CGRA-based architecture that time-multiplexes multiple configurations onto its processing elements to avoid load imbalance.
- We implement Fifer and evaluate its effectiveness on a wide range of applications, demonstrating its applicability.

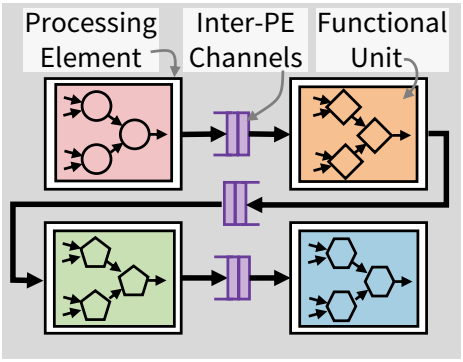
4.2 Background and Motivation

4.2.1 Challenges of irregular applications

CGRAs are amenable to creating *static spatial pipelines*, in which an application is split into pipeline stages and mapped to functional units across



(a) The pipeline-parallel implementation of BFS from Chapter 2.



(b) Mapping BFS as a spatial pipeline on current reconfigurable spatial architectures.

Figure 4-1: Mapping breadth-first search (BFS) to spatial architectures.

the fabric. To perform a particular computation, operands are passed from one functional unit to the next in this *fixed* pipeline. These are highly effective for regular applications, where the data access patterns and control flow are highly predictable.

However, mapping *irregular* applications to spatial architectures is more complex, because their unpredictable latencies and control flow can significantly impact CGRA throughput. The key insight to enabling effective mappings of irregular applications is to recognize that they are rich in otherwise-*regular* computation, but interspersed with *irregular* memory accesses and control flow. Moreover, in the previous chapter, we showed that irregular applications can be easily decomposed into pipelines and, with Pipette, efficiently executed in general-purpose cores. By extending Pipette's insights to CGRAs, we arrive at Fifer's first major contribution: enabling effective mappings to reconfigurable spatial architectures by *partitioning stages across sources of irregularity*.

Consider the pipelined BFS example from Chapter 2, but instead of each stage implemented as instructions for a general-purpose core, each stage is now mapped to a CGRA. (Keep in mind that BFS has tricky control flow; this is a non-trivial problem not handled by prior work that we will address in Section 4.3.) This transformed BFS can now be mapped to a spatial architecture, but it will suffer from poor performance due to load imbalance across stages. The baseline architecture that we use in this chapter (Section 4.3) has multiple *processing elements* (PEs), each with a separate CGRA fabric, and PEs can communicate through FIFO

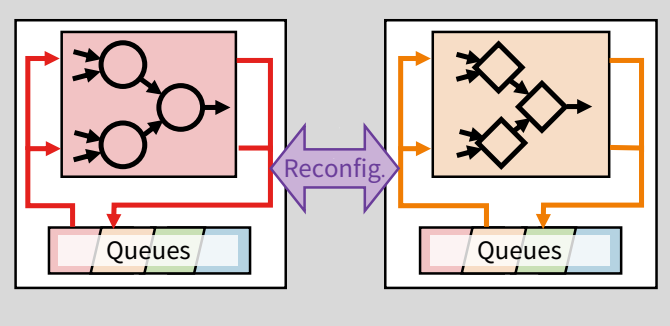


Figure 4-2: Fifer: map BFS (2 of 4 processing elements from Figure 4-1(b) shown) as a dynamic temporal pipeline to a time-multiplexed reconfigurable fabric on a *single* PE.

queues and access memory individually. Figure 4-1(b) shows how our transformed BFS can be executed in this architecture by mapping each stage to a PE. But this approach is still a *static spatial pipeline*, and quick variations in load across stages will cause stalls on empty or full queues, resulting in poor utilization.

Fifer's other key insight addresses the shortcomings of static spatial pipelines by observing that a stage need not be fixed to a PE for the lifetime of a computation. Instead of placing stages on physically distinct PEs and creating a spatial pipeline, we can *temporally pipeline stages by time-multiplexing them onto the same PE*. Figure 4-2 shows how Fifer maps BFS using this approach. We call this *dynamic* temporal pipelining, because a scheduler dynamically switches across stages based on the availability of work (e.g., when a stage runs out of work, the ready stage with the most work is switched in). Switches happen every few tens to hundreds of cycles, long enough to amortize reconfiguration overheads, and short enough to keep queue and memory footprint low (as these grow the further stages are decoupled).

4.2.2 Prior spatial and temporal CGRAs

Fifer is not the first proposal to add a temporal component to a spatial architecture, but to the best of our knowledge, it is the first to do so at this granularity.

At one extreme, prior work has proposed time-multiplexing at the cycle level: Triggered Instructions [92] is a spatial architecture with an array of PEs that communicate through latency-insensitive channels. Each

PE holds a limited number of instructions (e.g., 16), which become ready depending on runtime conditions (e.g., the availability of a value in a queue). Each PE chooses among one of the ready instructions each cycle. Though this cycle-level approach improves utilization, it comes at a cost: each PE is more complicated than the functional unit in a CGRA, and PEs communicate through queues rather than registers.

At the other extreme, run-time reconfiguration (RTR), a feature of commercial FPGAs (another kind of spatial architecture), has been used to time-multiplex configurations when a design exceeds available resources. However, this happens at coarse timescales—hundreds of microseconds [81]—much longer than the tens of cycles needed to effectively load balance stages of a pipeline or tolerate memory latency. Moreover, prior use of RTR only targeted applications that already map well to spatial architectures, like sorting [63] and streaming [21], or HLS-generated pipelines targeting applications with abundant data parallelism [117].

Fifer lies in the middle of these extremes, reconfiguring every few 10s–100s of cycles. This avoids expensive modifications to CGRAs and amortizes reconfiguration overheads, yet suffices to avoid load imbalance and achieve high utilization.

It is useful to contrast these techniques with general-purpose processors. They are analogous to *multithreading*, where a core switches among multiple threads of execution to improve utilization. Triggered Instructions is the CGRA analog to fine-grained multithreading [8, 55, 64] (FGMT), where the core time-multiplexes threads cycle by cycle; Fifer is the CGRA analog to coarse-grained multithreading [2, 3, 4, 57] (CGMT), where the core switches across threads less frequently, to tolerate long-latency events (e.g., on every L2 cache miss); and RTR is the spatial analog to software-only context-switching of threads by the operating system. Just as CGMT requires simpler core changes than FGMT, Fifer requires simpler changes than Triggered Instructions (quantitatively, the difference between these is larger since CGRAs do not already have a temporal component).

Unlike prior work, Fifer combines spatial pipelining and temporal pipelining at *coarse-grain timescales* (10s–100s of cycles) to amortize the costs of reconfiguration yet effectively tolerate latencies of the memory hierarchy, with *coarse-grain computation* (at the width of machine words, not bits) to address prior systems' limited throughput and flexibility.

We achieve better utilization by *(1)* time-multiplexing the fabric at the individual PE level for improved load balance, and *(2)* enabling fast

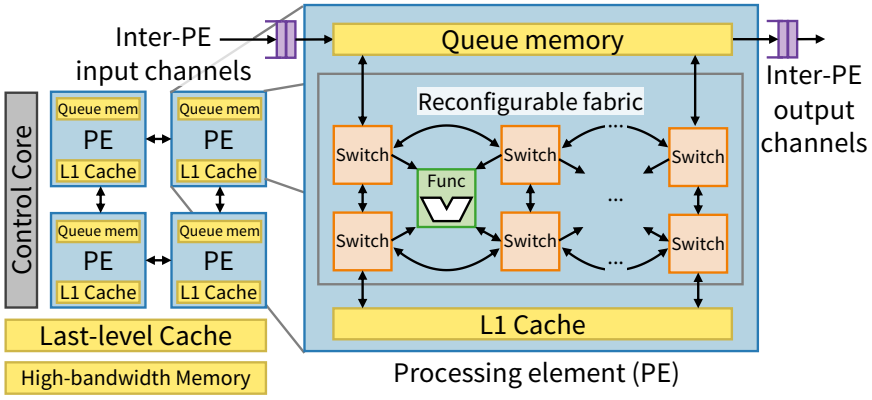


Figure 4-3: Baseline spatial architecture and design of a processing element (PE) with CGRA fabric.

communication between stages on the same PE. Our novel programming model—structuring applications as pipeline-parallel stages of computation that can be time-multiplexed on the same PE—overcomes the limitations of prior architectures and enables Fifer's performance benefits.

4.3 Baseline CGRA Architecture

Because there are many CGRA designs, we first introduce the baseline CGRA architecture to make the discussion concrete. Fifer then builds on this baseline.

CGRA fabric: As discussed in Section 2.2.2 and illustrated in Figure 4-3, a coarse-grain reconfigurable array (CGRA) is a grid of *functional units* connected together with switches. Each functional unitcontains an integer ALU similar to one in a general-purpose processor, capable of elementary operations (arithmetic, shifts, bitwise operations) at machine word width (e.g., 64 bits). Each PE also incorporates a few double-precision fused multiply-add (FMA) units to support floating-point workloads.

Configuration cells (registers) at each functional unit specify the operation of the ALU; additional configuration cells specify the connectivity of switches passing operands between functional units. Because conventional CGRAs are reconfigured rarely, their configuration cells use slow but simple write mechanisms, like register scan chains. (Fifer contributes a fast reconfiguration mechanism in Section 4.5.1.)

Inputs and outputs enter and leave the reconfigurable array through ports at the edges of the grid. The reconfigurable array is internally

pipelined; that is, functional units are separated by registers as shown in Figure 2-2, and the longest input-output path through functional units sets the latency of a given configuration. Registers also allow the CGRA to retain program state, e.g., to track loop iteration counts or accumulate values over loop iterations.

Multi-PE CGRA architecture: A single CGRA fabric is sometimes used as a functional unit or coprocessor [40, 43, 107] to accelerate small kernels. However, our goal is to handle full algorithms autonomously, without having general-purpose cores as intermediaries. To this end, our baseline architecture, shown in Figure 4-3, consists of multiple *processing elements* (PEs), each of which integrates a CGRA fabric, a private L1 cache, and mechanisms for queue-based communication. All PEs share a (highly banked) last-level cache.

This approach is preferable to having a single, very large CGRA fabric for two reasons. First, it enables having multiple independent private caches, so the system can achieve high memory throughput and exploit locality. Second, it provides decoupled communication between PEs. Because functional units inside each CGRA fabric are tightly coupled through *rigid* pipelines, a single stall would quickly propagate through the whole fabric. By contrast, PEs communicate with each other through FIFO queues, so when one incurs a stall (e.g., due to a cache miss), other PEs will not necessarily stall.

Our baseline implements inter-PE queues through a flexible interface: the switches at the edges of the fabric can dequeue from input queues and enqueue to output queues. Queues are stored in a small *queue memory* in each PE (a 16 KB SRAM in our implementation). This queue memory can be statically divided among multiple queues, each of which is managed as a circular buffer.

Mapping applications: To use this multi-PE design, applications are divided in *stages*, each of which is mapped to a PE. Then, these stages communicate through queues.

To use this baseline system well, it is crucial that all stages proceed at roughly the same rate: if one stage produces more inputs at a higher rate than its consumer, it will be bottlenecked by its consumer and frequently stall on a full output queue. Conversely, a too-fast consumer will spend many cycles waiting for input. Note that queues provide decoupling, but only against *temporary* mismatches in throughput, e.g., due to a cache miss. Long-running throughput differences will, over time, leave queues

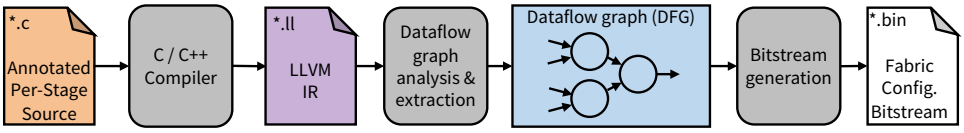


Figure 4-4: Process for transforming a pipeline stage's annotated source code into a Fifer PE fabric configuration.

full or empty, and make the whole program run at the pace of the rate-limiting stage. Because queues do not provide unbounded decoupling, we consider this design a *static* spatial pipeline, even though it is decoupled.

Prior work has proposed different techniques to use this kind of static pipeline well, such as replicating slow stages [42]. But this requires having known and stable rates, which is not the case with irregular applications.

4.4 Extracting Regular Stages from Irregular Applications

We present a new technique to map irregular applications to CGRA fabrics—this is a prerequisite for Fifer, but also for our baseline architecture. The key insight, as we explained in Section 4.2.1, is to first *partition the application into stages across sources of irregularity*. This produces *regular stages* and confines the irregularity to happen across stages. Then, these regular stages are efficiently mapped to a CGRA. In our implementation, the first step is manual, while the second is automated.

Partitioning: A program may be split at arbitrary locations into arbitrarily many stages, but judiciously decoupling programs is essential for good performance. Our overarching objective is to decouple irregular parts of the computation, such as unpredictable memory accesses and control flow, from more regular parts. Like Pipette, we split a program at every long-latency load, so that loads issued by a given stage are consumed by a different stage. Remember that for BFS (Chapter 2), each loop nest level contains such a load, so each stage corresponds to a level. Finally, stages that are too large to fit on one PE can always be divided into multiple smaller stages.

Mapping: Figure 4-4 shows the process of transforming partitioned serial code into configurations for a CGRA. We generate LLVM intermediate representation (IR) for each stage, which represents low-level operations on data and their dependences. An automated tool examines the LLVM IR

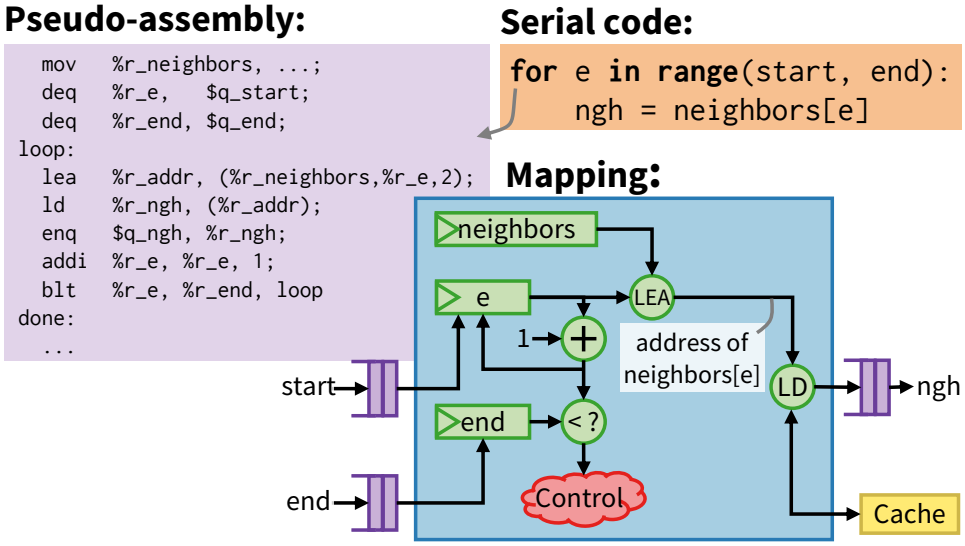


Figure 4-5: Example mapping of BFS's enumerate neighbors stage to a CGRA.

and produces a dataflow graph (DFG) using the actual operations that can be performed by a PE's functional units. The DFG is modified to receive its inputs and send its outputs via queues. A final bitstream generation step transforms the DFG into a bitstream that, when configured into a CGRA, carries out the computation represented by the DFG.

To concretely demonstrate this process, consider the enumerate neighbors stage from BFS, shown in Figure 4-5. This stage dequeues the `start` and `end` positions of the edge list of a vertex, and produces neighbor vertex ids. The operations required to carry out the serial code (in orange) for this stage are represented with the pseudo-assembly code (in purple) resembling the lowered LLVM IR produced by the compiler. Each pseudo-assembly instruction corresponds to an operation assigned to a functional unit in the PE.

When mapped to a DFG, this stage computes addresses (`LEA`) in the `neighbors` array, performs the dereference (`LD`), and passes the neighbor (ngh) to the next stage. Some additional logic (`+` and `< ?`) determines whether we have finished this edge list.

Values are processed in the order they arrive; once dequeued, they flow through one functional unit per cycle. Some control logic (the red cloud) triggers dequeues of new `start` and `end` values. In addition to driving enqueues and dequeues, per-PE control logic also orchestrates

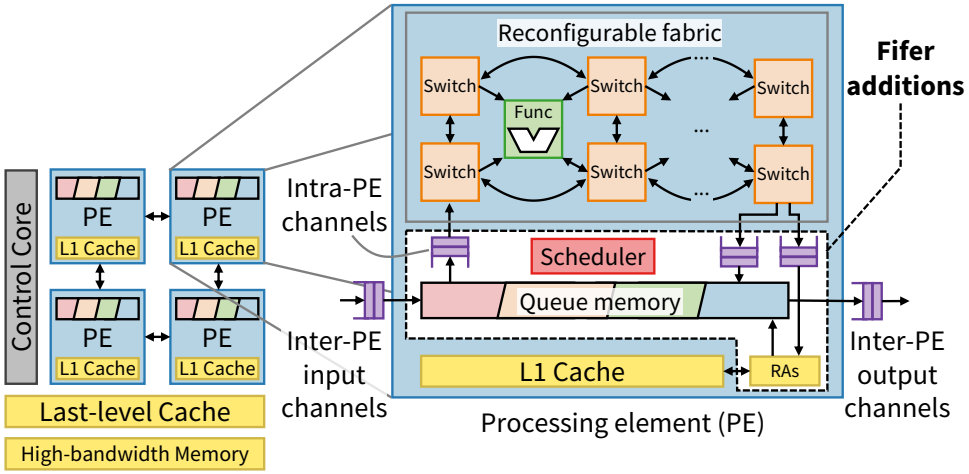


Figure 4-6: Fifer's modifications to a reconfigurable spatial architecture and processing elements, surrounded by a dashed line. (Queues, colored purple, are only illustrative and virtualized in the queue memory.)

reconfigurations (Section 4.5.1) and stalls the pipeline for cache misses on coupled loads (Section 4.5.4).

Inter-stage control flow: In irregular applications, stages often need to communicate control flow decisions to other stages; for example, in BFS, all stages need to know when the current distance from the source vertex has changed. Since stages communicate through queues, it is natural to pass this control information through the same queues. We extend queues to carry control or data values. Since control values are infrequent, we compile stages to handle either multiple input data values or one control value per cycle. Section 4.5.4 gives implementation details.

4.5 Fifer Architecture

Static spatial pipelines have several limitations reducing their effectiveness on irregular applications. Fifer overcomes these limitations by augmenting spatial pipelines with *temporal pipelines*. Figure 4-6 shows our modifications to the baseline system. We organize our discussion of Fifer to describe:

1. how multiple stages are time-multiplexed onto the same PE through the reconfiguration process,

2. how to extend queues to communicate between stages on the same PE, and

3. how to further decouple long-latency memory accesses.

We initially focus on a single-PE Fifer system that implements pure temporal pipelining. In Section 4.5.5 we discuss how multi-PE Fifer leverages both temporal and spatial pipelines.

4.5.1 Rapid reconfiguration

Each Fifer PE may select from not just one, but *many* possible configurations, so a PE may represent any part of a temporal pipeline throughout a program's execution.

Fifer is designed to transparently switch stages so that any ready stage begins executing as soon as possible. When a stage is scheduled onto a PE, functional units are configured with the operations needed by the stage. Reconfiguration also establishes connections between these functional units, as well as any input/output queues, registers, and memory connections. Any state required by the application, such as fixed constants, are loaded into the appropriate registers.

Unlike configurations for the baseline system, Fifer's configurations are stored in cacheable memory and loaded from the L1 cache. Configuration cells are chained, so that configurations can be loaded over multiple cycles. For example, our L1 supports a bandwidth of 64 bytes/cycle, and our 16×5 fabric requires about 360 bytes of configuration, so configuration cells are divided in 6 groups (with 5 groups consisting of a row of ALUs and switches, and one group being the last row of switches). Each cycle, the L1 serves 64 bytes of configuration data, which are propagated though the chained configuration cells. Thus, over 6 cycles (plus the L1 latency), the new configuration is loaded in place.

Loading the new configuration, as described by the previous paragraph, forms step (1) of a three-step reconfiguration process. Step (2) drains the in-flight operations from the current configuration. Step (3) activates the new configuration. Fifer introduces *double-buffered configuration cells* so that steps (1) and (2) take place in parallel. Consider the process in which Stage 3 is currently running on a Fifer PE and now needs to switch to run Stage 2. Figure 4-7 shows the behavior of a configuration cell (top row) and how it changes the currently executing stage of a Fifer PE (middle row).

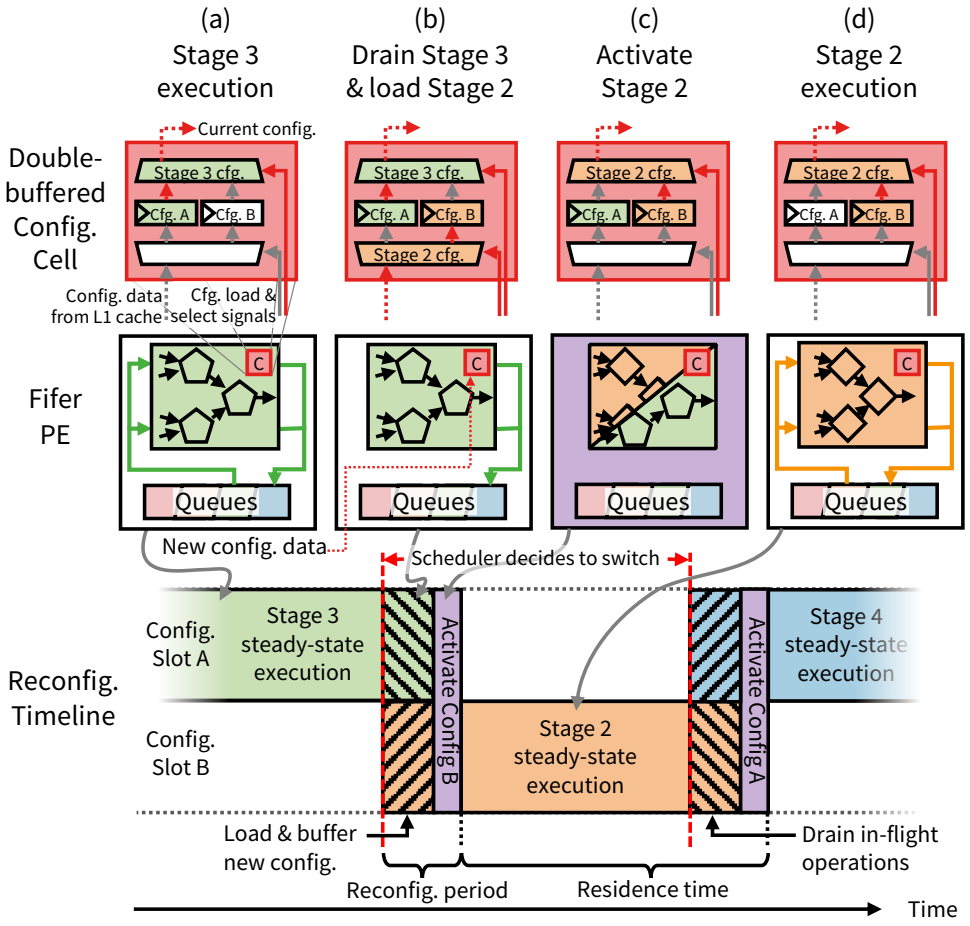


Figure 4-7: Reconfiguration process in a Fifer PE, depicting role of double-buffered configuration cells.

As soon as the scheduler begins the reconfiguration process, Stage 3 stops accepting new inputs and begins draining in-flight operations, as Figure 4-7(b) shows. In parallel with draining in-flight operations, the PE begins loading the new configuration. This parallel loading is enabled by Fifer's double-buffered cells, which offers two configuration slots (Cfg. A and Cfg. B): one containing the current configuration and one to receive the new configuration. The PE loads new configuration data into the unused configuration slot (e.g., Cfg. B, since Cfg. A is currently used for Stage 3). Double-buffering allows us to overlap execution of the current configuration with the loading of the next configuration, which is essential to reducing the cost of reconfiguration. In practice, most configurations have over 6 pipeline stages, so draining them takes longer than loading

the new configuration and is the dominant cost of reconfigurations for most applications.

Once remaining in-flight operations finish and the new configuration is loaded, the fabric *activates* the new configuration. Within the double-buffered configuration cells in Figure 4-7(c), a multiplexer switches from reading Cfg. A to Cfg. B, so now Stage 2 becomes active. We account for this process with an *activation time*, a dead time of two cycles. At last, in Figure 4-7(d), Stage 2 commences execution.

The old configuration may have written to state elements, like registers, that need be preserved across reconfigurations. As the new configuration loads, the contents of these state elements are drained out along with the old configuration to the L1 cache.

We define the *reconfiguration period* to be the time spent performing all of these operations: draining in-flight operations, loading the new configuration, and activating it. The *residence time* for a given stage is the time between activating that stage and the activation of the next stage (and thus includes the reconfiguration period). Fifer's effectiveness relies on keeping the reconfiguration period small—no more than a few dozen cycles—and maximizing residence times to many hundreds of cycles. We evaluate the effect of reconfiguration period on performance in Section 4.8.3.

4.5.2 Scheduling reconfigurations

Fifer extends each PE with a simple *scheduler* that dynamically switches among stages. Which stage is scheduled onto a particular PE depends on which input queues have values available and which output queues have space.

To keep utilization high and amortize the overall cost of reconfigurations, the scheduler follows a simple policy. First, it keeps a PE configured to the current stage until it is blocked by a full output queue or an empty input queue. Second, when it must select a new stage, the scheduler examines the occupancies of the queues used by the other stages. Of the unblocked stages (i.e., those with non-empty input queues and non-full output queues), the scheduler selects the stage with the greatest amount of work available in its input queues. By selecting stages with more work, the scheduler reduces the number of reconfigurations.

We also tried other scheduler policies, such as a round-robin scheduler or finer-grain multithreading, but found that these did not work as well.

This makes sense: the application work done is nearly constant regardless of the scheduling policy, so processing the stage with the most work reduces the amount of reconfigurations; alternative policies increase reconfiguration frequency.

4.5.3 Communicating in temporal pipelines

Because Fifer allows a producer stage to communicate with a consumer stage that may be located at the same PE, we introduce *intra-PE* queues. We augment the queue buffer described in Section 4.3 with more head/tail pointers to store these additional queues. These intra-PE queues offer high-bandwidth communication between stages located on the same PE, so it is advantageous to decouple applications such that stages communicating frequently reside on the same PE. Of course, as before, a producer stage on one Fifer PE can enqueue data destined for a consumer stage at a different PE.

For the purposes of evaluation, the baseline spatial architecture and Fifer have the same amount of queue buffer per PE. This means that Fifer, which can fit many more pipelines than the baseline, could have less effective space per queue. However, as we will see, modest decoupling suffices to achieve high utilization.

4.5.4 Decoupling memory accesses and handling control flow

As we saw in Pipette, irregular applications pose unique challenges in its long chains of memory accesses and complex control flow. We extend Pipette's insights to address them in specialized architectures as well.

Further decoupling memory accesses: Memory accesses have widely varying needs: some may be irregular and cause stalls so they need to be decoupled; others may be known to rarely cause cache misses and so they do not merit further decoupling. Thus, Fifer PEs offer both *decoupled* and *coupled* load interfaces. The conventional, coupled load interface is a simple connection to the cache and stalls the PE on cache misses. Simple memory access patterns, like streaming linearly through memory, do not need to be decoupled, and would be suitable for this interface.

However, some accesses are known to miss frequently, causing lengthy stalls. Fifer allows these accesses to be further decoupled from stages; we accomplish this by adapting Pipette's reference accelerators (RAs, Section 3.3.2) to Fifer. Reference accelerators are especially critical here,

as the CGRA fabric's rigid pipelines are especially sensitive to latency. Unlike memory accesses initiated by the fabric, RAs' memory accesses may complete out of order. Just like Pipette, RAs are configured once at initialization and continue performing accesses regardless of the currently scheduled stage.

Handling control flow: Sometimes, producers may need to communicate control flow decisions to downstream consumer stages. Control values, as done in Pipette (Section 3.2.2), lets PEs change local state (such as updating the current BFS fringe), wait to synchronize with other PEs, or switch to another configuration. Using control values cheaply implements point-to-point synchronization where global synchronization is unnecessary or cumbersome.

4.5.5 Multi-PE Fifer to exploit data parallelism

Up to this point, we have focused on temporal pipelines running on a single Fifer PE. We now explore how spatial pipelines can be used to leverage data parallelism. While our programming model focuses on making pipeline parallelism easy to exploit, data parallelism and pipeline parallelism are complementary; by exploiting them together, Fifer offers advantages that are not available when exploiting data parallelism alone. We now present two techniques that exploit data parallelism *within* a PE and *across* PEs.

SIMD-style parallelism within a PE: Because we can split applications into as many stages as needed, each stage can be arbitrarily small. When a stage implements data-parallel computation, the *datapath* obtained from its DFG can be replicated to use the PE's fabric as much as possible by filling unused functional units and switches. For example, a 16×5 grid of functional units can be configured as four copies of a datapath that fit on a smaller 4×5 grid, yielding a potential $4\times$ throughput improvement. Our applications provide abundant opportunities to take advantage of SIMD-style parallelism: for example, the many edge list accesses performed by graph applications can be launched in parallel.

These datapaths run *in lockstep*: if multiple input elements are available at once, they can be dequeued as a group and processed simultaneously, up to the number of replicated datapaths. Control values are always handled serially: in a given cycle, a PE can dequeue multiple data values but will always dequeue a single control value.

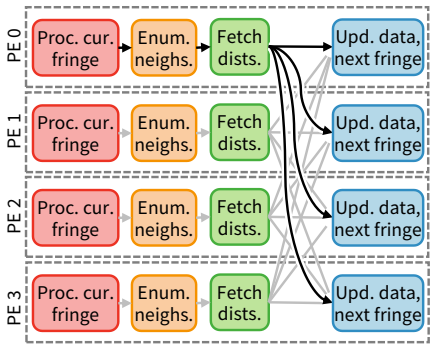


Figure 4-8: Replicated 4-PE BFS with Fifer.

This process is analogous to how vector processors, SIMD instructions, and GPUs exploit data parallelism: running multiple copies of the same operation in lockstep. When fewer than the maximum supported number of elements are available at the input, some datapaths are left unused, just like masked-off lanes in vector processing. However, Fifer's decoupled execution model makes exploiting SIMD-style data parallelism more efficient. For example, to get good GPU lane utilization in graph algorithms, edges from multiple vertices need to be processed in the same warp, leading to complex marshaling of vertex and graph metadata. Instead, Fifer's queue-based approach allows decoupling the processing of vertices and edges across stages, so they are grouped independently and easily fill the parallel datapaths.

Replicated temporal pipelines across PEs: Temporal pipelines can coexist with spatial pipelines; we spatially partition the data across multiple PEs, each running its own temporal pipeline. This partition is similar to the multicore, multithreaded implementation for Pipette in Section 3.5.6. Figure 4-8 replicates the BFS pipeline from Figure 4-1(a) across four PEs, each pipeline processing a fraction of fringe and updating separate parts of the graph.

Pipeline parallelism and queue-based communication enable a crucial optimization: instead of synchronizing through shared memory, each pipeline—that is, a processing element—sends neighbors "owned" by a different pipeline on another processing element. In BFS, this sharding is represented by the cross-PE communication between fetch distances, the third stage, and update data and next fringe, the final stage. By avoiding the need for shared-memory synchronization, applications scale better than by exploiting data parallelism alone.

Item	Area
Reconfigurable fabric, 16 × 5 func. units	0.91 mm^2
and 4× double-precision FMA units	0.15 mm^2
16 KB queue SRAM	0.054 mm^2
4× reference accelerators (RAs)	0.0029 mm^2
32 KB data cache	0.22 mm^2
Total area (per PE)	1.34 mm^2

Table 4-1: Implementation costs for major components of a Fifer PE.

Inter-PE queues use credit-based flow control [28] to implement backpressure and handle multiple producers. Credits are associated to free queue space. Each destination queue divides credits evenly across producers, and a producer stalls when it runs out of credits.

Importantly, each PE in this replicated pipeline is still a dynamic temporal pipeline, so it *independently* reconfigures itself in response to varying load. Thus, different PEs can work on different stages. This scheme offers an additional dimension of load balance: not only is work distributed across PEs, but each PE also tolerates a different distribution of work among stages. This allows us to employ simpler partitioning schemes—for example, by examining bits of the neighbor id—rather than resorting to more complex techniques like work stealing.

4.6 Fifer Implementation

We implement the Fifer architecture by writing and synthesizing RTL for its major components.

The CGRA in each PE is a 16 × 5 grid of functional units surrounded by switches, a scaled-up version of the DySER fabric [43]. We use the CGRA-ME [22] framework to generate RTL for this fabric. CGRA-ME's output Verilog only uses a simple register scan chain to implement its configuration; we replace this with double-buffered configuration cells (Section 4.5.1) to allow loading a new configuration while the current configuration's remaining in-flight operations complete. To make loading configurations fast, Fifer loads configuration data served at the L1 width, not through a serial scan chain.

Our virtualized queues are stored in a 16 KB buffer, and each PE contains a 32 KB data cache. To support the floating-point operations used

in some of our benchmarks, we synthesize several double-precision fused multiply-add (FMA) units and distribute them evenly across the fabric. The reference accelerators (RAs), which launch and track decoupled memory accesses, add little additional area cost.

We synthesized these components with Yosys [136] and the 45 nm FreePDK45 library [51], closing timing at 2 GHz, and summarize the area used in Table 4-1. The memory arrays in caches and queue storage were estimated with CACTI [11]. Overall, each PE is 4.6% of the area of a core in the same technology node (45 nm Nehalem [123]), and each PE has higher arithmetic intensity. To account for this difference, in the evaluation, CGRA-based systems use 4 PEs for each OOO core of the baseline, which is conservative area-wise. (Our evaluation in Section 4.8 uses Skylake cores with larger structures, which makes our estimate even more conservative, even considering scaling to Skylake's 14 nm node.)

We transform our evaluated applications in two steps. We first derive the pipeline-parallel version by manually dividing code into stages using the systematic approach described in Section 4.2.1 and Section 4.4. Then, we use per-stage data parallelism by replicating the dataflow graph until we fill the PE fabric.

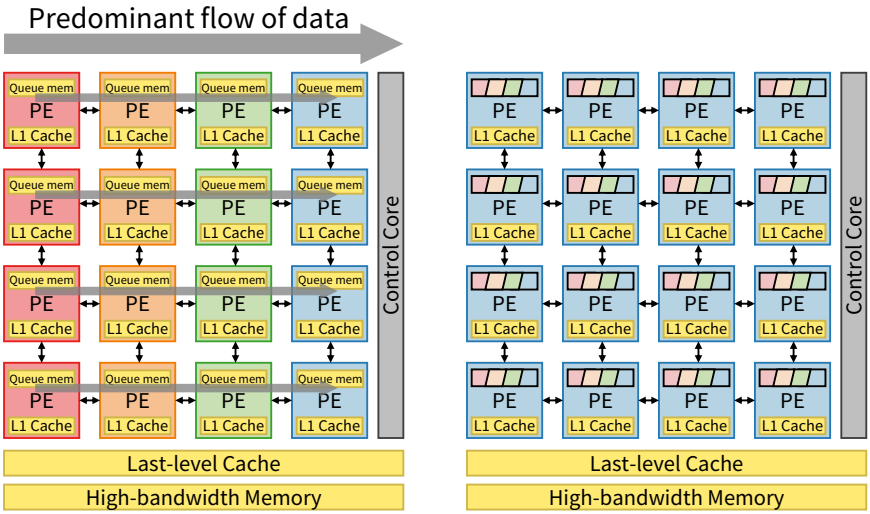
Whenever a Fifer PE changes configuration, it takes a minimum of 12 cycles (loading the new configuration from the L1 cache is 10 cycles, plus 2 cycles for the activation time), but as previously discussed in Section 4.5.1, draining in-flight operations may increase this time.

4.7 Experimental Methodology

4.7.1 Evaluated systems

We implement and evaluate Fifer using cycle-level simulation. For the serial and OOO cores, we use the same evaluation methodology as used for Pipette (but the cores do *not* use Pipette). We also create a cycle-level simulator to evaluate our CGRA-based systems; it simulates executing stages using mapping information produced by CGRA-ME [22].

We model core and uncore energy at 22 nm for the OOO systems with McPAT [70] and use prior work to estimate HBM energy consumption [89]. Energy consumption for the reconfigurable fabric is based on Synopsys Design Compiler post-synthesis power estimates and scaled from 45 nm to 22 nm.



(a) Baseline reconfigurable architecture with static spatial pipelines.

(b) Fifer architecture with dynamic temporal pipelines.

Figure 4-9: Baseline and Fifer system implementations.

Figure 4-9(b) shows our modeled spatial reconfigurable architectures, including 16 PEs with buffers serving as queue storage. A control core is responsible for initialization and teardown of a Fifer program as well as interactions requiring a general-purpose CPU (such as calls to the OS).

Comparison systems: Our baseline is the 16-PE system depicted in Figure 4-9(a). Each stage of an application is mapped to a single PE, and this mapping remains fixed throughout the run. As a result, the predominant flow of data through the pipeline is also fairly fixed; for example, in BFS the predominant flow of data would be from left to right, as indicated by the gray arrow. While the baseline and Fifer systems have the same amount of space allocated for queues on each PE, the static system notably lacks the scheduler. The baseline system also retains RAs, to focus our analysis on the effects of time-multiplexing on load balance. Table 4-2 summarizes our evaluated systems' parameters.

Benchmarks: We evaluate Fifer on the same six applications from Pipette, with each application mapped to CGRA fabric. Otherwise, we use the same pipeline configurations, inputs and overall evaluation methodology as Pipette (see Section 3.4).

PEs	16 PEs, 2 GHz, 16×5 func. unit mesh, 32 KB L1 cache (8-way set-associative, 4-cycle latency)
Fifer	Up to 16 queues per PE, virtualized on a 16 KB buffer
LLC	2 MB/core *or* 512 KB/PE, 16-way set-associative, 40-cycle latency
Main mem	120-cycle latency, 256 GB/s high-bandwidth memory

Table 4-2: Configuration parameters of the evaluated system.

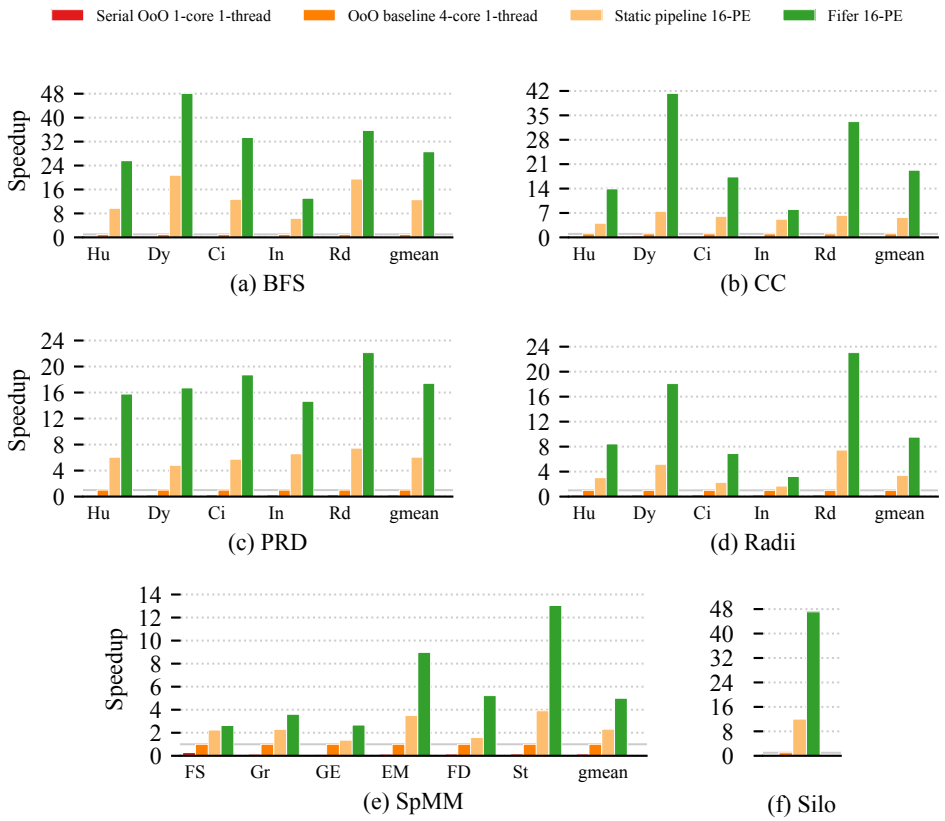


Figure 4-10: Per-input performance of all evaluated applications.

4.8 Evaluation

We compare Fifer to state-of-the-art data-parallel baseline implementations running on a generously provisioned out-of-order core. We compare our technique to an OOO baseline, then show that our approach achieves better utilization, and thus better performance, than a pipeline of static, single-stage PEs.

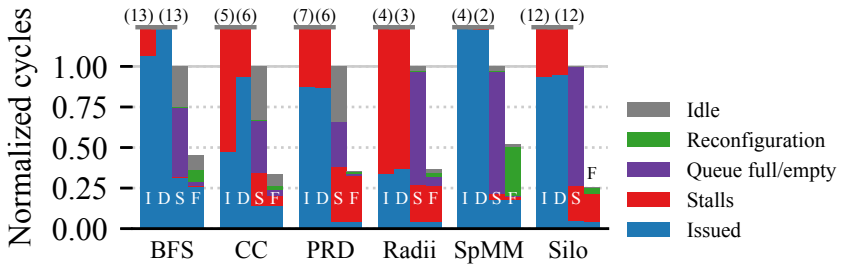


Figure 4-11: Breakdown of cycles spent executing each application, normalized to the static pipeline and averaged across inputs. (I: OOO serial, D: OOO multicore, S: Static pipeline, F: Fifer)

4.8.1 Fifer outperforms the OOO baseline

The OOO core baseline, despite its significant area overheads, fails to perform well because it cannot effectively handle these applications' unpredictable memory accesses and control flow. Because it is a temporal architecture that sequences instructions one after another, it also suffers from low arithmetic intensity compared to the CGRA fabric. As a result, the static pipeline and Fifer are 25× and 72× faster than serial, respectively.

4.8.2 Fifer outperforms static pipelines

Figure 4-10 shows the performance of our evaluated systems normalized to the performance of the *OOO multicore* (not serial). Fifer outperforms the static pipeline by gmean 2.8× and by up to 5.5× (CC with the Dy input). This speedup comes from Fifer's ability to change contexts in response to available work at each PE. For example, in BFS, speedups are best on graphs with high outdegree, where Fifer's many dynamic temporal pipelines working in parallel achieve better throughput than the baseline's few static pipelines.

To better understand how each system spends its execution time, Figure 4-11 shows the breakdown of cycles spent executing each benchmark. We report the proportion of time spent by a core using the CPI stack methodology [35]. We extend this methodology to our PEs as well. Each group of bars reports breakdowns of each variant across benchmarks (averaged across inputs), relative to the static pipeline baseline. Each bar within a group reports cycles for one system, broken down in cycles

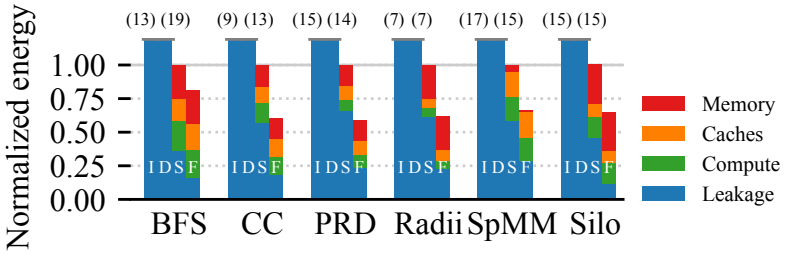


Figure 4-12: Breakdown of energy, normalized to the static pipeline and averaged across inputs. (I: OOO serial, D: OOO multicore, S: Static pipeline, F: Fifer)

spent *(1)* performing useful computation, and waiting on *(2)* backend or CGRA stalls (due to non-decoupled loads), *(3)* full or empty queues, *(4)* reconfigurations (for Fifer), or *(5)* idle stalls (when a PE is completely inactive while waiting for other PEs, e.g., a barrier).

As expected, a significant source of slowdowns in the serial and data-parallel systems is waiting on the backend (in red), which includes waiting for memory accesses and OOO core structures becoming full. In the static pipeline and Fifer systems, the breakdown also includes stalls resulting from full or empty queues (in purple). Finally, for Fifer, time spent reconfiguring is shown in green. In spite of this, Fifer performs better because it overlaps useful work, like completing memory accesses, as these reconfigurations occur.

These cycle breakdowns help us understand why Fifer performs consistently better across applications: as expected, the static pipeline spends a significant fraction of time stalled on full or empty queues (purple bars). Reconfiguration stalls are significant in SpMM because it is a *control-intensive* application: the merge-intersect stage intersects values at very high throughput, and when it reaches the end of an input row or column, it directs the producer to stop fetching unneeded data. In relatively sparse matrices, such as FS and Gr, with averages of 2.4 and 8.0 non-zero elements per row, respectively, merge-intersections complete rapidly, resulting in queues emptying more often and triggering more reconfigurations. In SpMM, despite frequent reconfigurations, Fifer is gmean 2.2× faster.

We also examine the energy consumed by each system in Figure 4-12. Each bar indicates the dynamic energy consumed by the memory hierarchy (red and orange), dynamic energy consumed by cores or PEs (green), and energy consumed due to leakage currents (blue). The systems with OOO

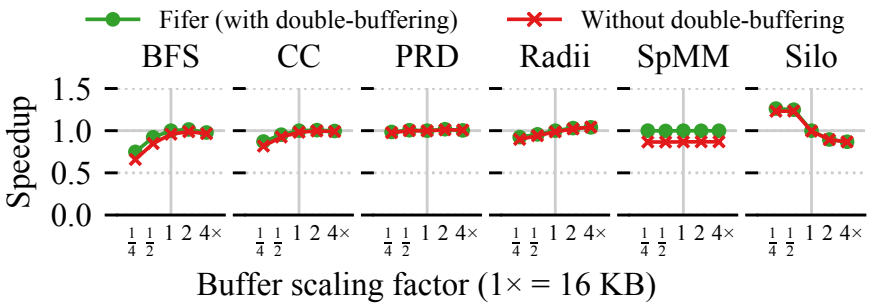


Figure 4-13: Fifer performance as the size of per-PE queue memory grows and how double-buffered configuration cells affect speedup.

cores not only suffer from considerable leakage currents but also consume significant dynamic energy per instruction. By contrast, the systems with reconfigurable PEs consume much less energy, due to their reduced area and higher performance: the static pipeline achieves gmean 12× better energy efficiency than the OOO multicore across all applications. Using Fifer further improves energy efficiency because applications complete faster and reduce the energy lost to leakage. In SpMM, the static pipeline suffers from increased main memory energy usage due to the larger memory footprint created by separate PEs working on different parts of the matrix; Fifer avoids this by keeping the entire pipeline's working set at a single PE. Otherwise, dynamic energy consumed by the reconfigurable fabric and the memory hierarchy in Fifer and the static pipeline remain largely the same. Overall, Fifer reduces gmean energy consumption by 1.5× over the static pipeline and by 19× over the 4-core OOO system.

4.8.3 Sensitivity to queue size and reconfiguration time

We now study Fifer's sensitivity to queue size. Figure 4-13 reports gmean performance relative to the default configuration, a 16 KB queue memory, as it changes from 4 KB (0.25× the default) to 64 KB (4× the default). A second line allows us to study the effect of using Fifer's double-buffered configuration cells.

Figure 4-13 shows that applications are sensitive to these parameters in different ways. First, BFS is mainly sensitive to queue sizes: its performance nearly halves with a 4 KB memory due to insufficient decoupling. CC, PRD, and Radii, which also benefit from the additional queue space, show similar trends in speedup as queue size increases. These applications do not see significant changes with the addition of double-buffered

configuration cells. Because larger queues make reconfigurations less frequent, slower reconfigurations are irrelevant for large queue sizes.

Next, SpMM is mainly sensitive to reconfiguration latency: its performance reduces by about 15% without the double-buffered configuration cells. As mentioned, SpMM is a control-intensive application; precisely because of this reason, SpMM's performance is flat across queue sizes: larger queues let producers fetch further ahead, but this is not used because merge-intersections redirect producers every few elements. Without double-buffering, PEs cannot overlap loading a new configuration from memory as they complete the old stage's in-flight operations.

Finally, while Silo is insensitive to reconfiguration latency, its performance somewhat *decreases* as queue size increases. The larger queues enable so much parallelism that it significantly strains the memory hierarchy: with a 64 KB queue buffer, the working set of a stage matches the L1 size, and L1 hit rates fall from 66% to 62%. This increases pressure on the LLC and adds non-decoupled L1 misses that stall the PEs. This shows that extremely excessive decoupling can cause memory footprint issues, just as how prefetchers can trigger extra cache misses when running too far ahead.

Table 4-3 lists the average time a configuration resides on a PE, as well as the time needed to complete a reconfiguration for each benchmark. A typical application, BFS, executes each stage for about 140 cycles before reconfiguring, and spends around 13 cycles in reconfiguration, most of which is spent completing the previous configuration's in-flight operations. Applications on average spend 448 cycles per configuration with about 19.7 of those cycles spent reconfiguring. SpMM, with its frequent switching, shows why double-buffered configuration cells are crucial: being able to load the new configuration and finishing the current one in parallel minimizes dead time on a PE. Finally, the average time spent in a configuration is correlated to queue capacity; quadrupling queue storage increases the average residence time to 1488 cycles.

Finally, we also evaluated a system that can perfectly overlap loading a new configuration with completing the previous configuration's operations, achieving zero-cost reconfiguration. This system improves performance by just 10% gmean (and up to 1.8× on SpMM's `Gr` input). We conclude that this alternative design is a poor tradeoff, as it incurs too much complexity for its limited benefits (for example, due to stalls from the outgoing stage, functional units would have to interleave the execution of multiple stages).

Application	BFS	CC	PRD	Radii	SpMM	Silo	Mean
Avg. residence time	140	279	927	564	30	1490	**448**
Avg. reconfig. period	12.5	13.9	20.4	27.7	12.6	60.1	**19.7**

Table 4-3: Average residence time of a configuration and time needed to complete reconfiguration (draining the old configuration, loading and activating the new configuration), in cycles.

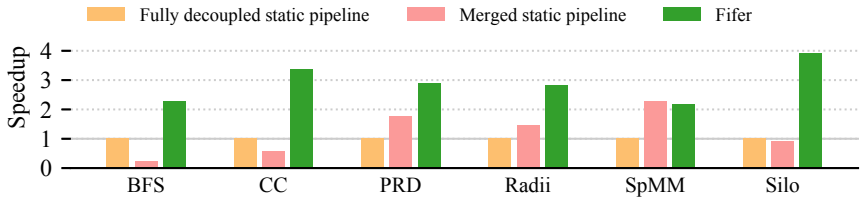


Figure 4-14: Performance of applications with merged stages.

4.8.4 Sensitivity to stage count

So far we have evaluated fully decoupled application pipelines: we have used our partitioning technique to split the application into stages across every long-latency load. This produces regular stages that can execute on a CGRA fabric efficiently. Because Fifer time-multiplexes stages, splitting the program aggressively is the right approach. But for the static pipeline, the tradeoff is less clear: because each stage runs on a separate PE, having many stages improves decoupling but may worsen load imbalance.

To study this effect, we now consider alternative application pipelines where we judiciously merge stages to try to improve the performance of the static pipeline. For example, in BFS, the source-centric stages (processing the fringe, enumerating neighbors, and fetching distances) can be merged into a single stage, reducing a 4-stage pipeline to two stages. This pipeline still decouples across the most expensive indirection and, in the static pipeline, allows twice the number of parallel pipelines to be instantiated. However, the resulting first stage of this pipeline will incur stalls due to long-latency loads. In general, we choose which stages to merge by focusing on less-frequent indirections, merging low-activity stages, and keeping stage logic small enough to still fit in a single PE.

Figure 4-14 shows the results of combining stages compared to the original (i.e., fully decoupled) Fifer and static pipeline, normalized to the performance of the *original* static pipeline and averaged across inputs.

Overall, applications see very different effects. In BFS, because combining stages reintroduces coupling, the merged static pipeline is 4.4× slower than the original. CC sees a similar tradeoff as BFS, but PRD and Radii become slightly faster. In SpMM, stages are combined so that a single PE carries out the entire matrix multiplication for its share of rows. This exploits more data-parallelism at the expense of decoupling and benefits from small matrices like FS and Gr, which, as we discussed previously, cause Fifer to switch very frequently. Due to high speedups in those matrices, the merged static pipeline is gmean 4% faster than Fifer across inputs (Fifer using this coupled pipeline for the inputs that benefit from it and the decoupled pipeline for the others is 12% faster). Finally, Silo sees a slight performance degradation from the merged pipeline.

4.9 Summary

We were able to achieve speedups on irregular applications in general-purpose cores with Pipette. However, general-purpose cores are still very inefficient due to their organization as temporal architectures. We observed that reconfigurable spatial architectures can significantly improve compute intensity. However, the unpredictable memory accesses and control flow of irregular applications are significant obstacles to good performance when using spatial architectures. As before, we structured irregular applications as spatial pipelines. Then, we proposed Fifer, which augments a spatial reconfigurable fabric with dynamic temporal pipelines, allowing us to create pipeline-parallel applications whose stages are time-multiplexed onto this fabric. Just as we used SMT in Pipette to time-multiplex many stages in a general-purpose core, we added a cheap reconfiguration mechanism that lets us layer temporal pipelines atop spatial pipelines. With this addition, and several other features like handling control flow and reference acceleration, we demonstrated significant utilization and performance improvements in key application domains. Fifer thus makes efficient acceleration of irregular applications practical for reconfigurable architectures.

5 Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism

5.1 Introduction

The two previous chapters described hardware architectures that execute irregular applications efficiently, achieving speedups of up to 47× over an OOO multicore. However, there is currently *no automatic way to generate efficient pipelines for irregular applications*. Existing compilers only produce *regular* pipelines [29, 42, 65, 94, 121, 122], and so far, *irregular* pipelines have been written by hand. Creating an irregular application pipeline requires making many choices that have significant impact on performance, and it is tedious and error-prone to do so manually.

Pipelining an irregular application involves three challenges. First, it requires decoupling straight-line code into pipeline stages, e.g., producing the two example pipelines from Chapter 2. More concretely, while the simple serial description of BFS (Figure 5-1, top) is about thirty lines of code, turning this into efficient pipeline-parallel code (bottom) not only doubles the size of code, but also requires several complex transformations. Second, and more importantly, it requires selecting the right pipeline, which depends on the application and architecture. For instance, in the example at the beginning of Chapter 2, if `A[]` is prefetched accurately, it may be better to combine the fetch `A[i]` and filter stages. Third, because irregular applications have frequent control flow and shared state, it is important to handle these efficiently when partitioning it across stages. Otherwise, the resulting overheads may negate the benefits of pipelining.

```
void bfs(Graph* g, int* cur_fringe, int* next_fringe,
         int root, int* distances) {
    int cur_fringe_idx = 0, next_fringe_idx = 0;
    int cur_dist = 0;
    // Add root to fringe
    cur_fringe[cur_fringe_idx++] = root;
    distances[root] = 0;
    while (cur_fringe_idx != 0) {
        cur_dist++;
        // Process current fringe
        for (int i = 0; i < cur_fringe_idx; i++) {
            int v = cur_fringe[i];
            // Enumerate neighbors
            int edge_start = g->nodes[v];
            int edge_end = g->nodes[v+1];
            for (int e = edge_start; e < edge_end; e++) {
                // Visit neighbor
                int ngh = g->edges[e];
                // If dist decreases, update it,
                // add ngh to next fringe
                int old_dist = distances[ngh];
                if (cur_dist < old_dist) {
                    distances[ngh] = cur_dist;
                    next_fringe[next_fringe_idx++] = ngh;
                }
            }
        }
        swap(&cur_fringe, &next_fringe);
        cur_fringe_idx = next_fringe_idx;
        next_fringe_idx = 0;
    }
}
```

```
void bfs_stage1(Graph* g, int* cur_fringe, int* next_fringe,
        int root, int* distances) {
    int cur_fringe_idx = 0;
    int cur_dist = 0;
    // Add root to fringe
    cur_fringe[cur_fringe_idx++] = root;
    distances[root] = 0;
    while (cur_fringe_idx != 0) {
        cur_dist++;
        // Process current fringe
        for (int i = 0; i < cur_fringe_idx; i++) {
            int v = cur_fringe[i];
            enq(1, v);
            enq(1, v+1);
        }
        enq_ctrl(1, NEXT);
        swap(&cur_fringe, &next_fringe);
        cur_fringe_idx = deq(5);
    }
   enq_ctrl(1, LAST);
}
```

```
void bfs_stage2(Graph* g, int* cur_fringe, int* next_fringe,
        int root, int* distances) {
    setup_reference_accelerator(1, INDIRECT, g->nodes);
    setup_control_value_handler(1, &&q1_handle_ctrl);
    while (true) {
        while (true) {
            // Enumerate neighbors
            int edge_start = deq(1);
            int edge_end = deq(1);
            for (int e = edge_start; e < edge_end; e++) {
                enq(2, e);
            }
        }
q1_handle_ctrl:
        if (deq(1) == LAST) {
            enq_ctrl(2, LAST);
            break;
        }
        enq_ctrl(2, NEXT);
    }
}
```

```
void bfs_stage3(Graph* g, int* cur_fringe, int* next_fringe,
        int root, int* distances) {
    setup_reference_accelerator(2, INDIRECT, g->edges);
    setup_control_value_handler(2, &&q2_handle_ctrl);
    while (true) {
        while (true) {
            // Visit neighbor
            int ngh = deq(2);
            enq(3, ngh);
            enq(4, ngh);
        }
q2_handle_ctrl:
        if (deq(2) == LAST) {
            enq_ctrl(3, LAST);
            break;
        }
        enq_ctrl(3, NEXT);
    }
}
```

```
void bfs_stage4(Graph* g, int* cur_fringe, int* next_fringe,
        int root, int* distances) {
    int next_fringe_idx = 0;
    int cur_dist = 0;
    setup_reference_accelerator(4, INDIRECT, distances);
    setup_control_value_handler(3, &&q3_handle_ctrl);
    while (true) {
        cur_dist++;
        while (true) {
            int ngh = deq(3);
            // If dist decreases, update it,
            // add ngh to next fringe
            int old_dist = deq(4);
            if (cur_dist < old_dist) {
                distances[ngh] = cur_dist;
                next_fringe[next_fringe_idx++] = ngh;
            }
        }
q3_handle_ctrl:
        if (deq(3) == LAST)
            break;
        swap(&cur_fringe, &next_fringe);
        enq(5, next_fringe_idx);
        next_fringe_idx = 0;
    }
}
```

Figure 5-1: Sequential BFS code (above) and hand-optimized pipeline-parallel BFS implementation (below), with changes shaded in gray. Parallelizing this multithreaded code by hand is tedious and error-prone; we automate this process with Phloem.

We present *Phloem*,[1] a compiler that automatically discovers and exploits pipeline parallelism in irregular applications. Phloem's key enabling insight is that the transformations required for pipeline parallelism can be carried out as a series of novel, simple, composable passes that leverage simple static analyses and cost models. These analyses and models help Phloem select effective decoupling points, tighten inner loops, and reduce the impact of irregular control flow. Finally, Phloem generates code that leverages hardware support that enables irregular applications to run efficiently as pipelines.

Our Phloem implementation can be used as a standalone compiler on serial C/C++ code or combined with existing domain-specific compilers to produce efficient pipeline-parallel applications from high-level code. We demonstrate this by combining Phloem with Taco [61] to automatically pipeline sparse linear algebra kernels.

Our evaluation shows that Phloem approaches the performance of manually tuned pipelines. Averaging across all evaluated applications, Phloem achieves gmean speedup 1.7× over serial code, and 85% of the performance of manually tuned code. In the best case, Phloem even *exceeds* the performance of manually tuned code by 15%. We also show that Phloem can be combined with existing domain-specific compilers to produce efficient pipeline-parallel applications.

In summary, we make the following contributions:

- We show how to systematically partition *irregular* applications into stages and how to manage state to maximize performance.
- We introduce Phloem, which automatically transforms serial source code into efficient pipeline-parallel implementations through a series of simple passes.
- We demonstrate Phloem's broad applicability by interfacing it seamlessly with a domain-specific compiler.
- We implement and evaluate Phloem, achieving performance comparable to hand-optimized code.

1 Pronounced like "flow 'em", phloem is a plant's specialized vascular tissue for conducting sugars and other metabolic products [1].

5.2 Baseline architecture

Phloem leverages hardware support for irregular application pipelines. We use Pipette (Chapter 3) as our baseline architecture. Phloem uses nearly every Pipette feature to create efficient pipelines from irregular applications. These include Pipette's support for further decoupling memory accesses (reference accelerators) as well as its mechanisms for efficiently handling control flow (control values and control value handlers).

We also augment Pipette's reference accelerators by allowing them to be chained. *Chained reference accelerators* allow Phloem to exploit the fact that some stages simply dequeue values from one RA only to enqueue it to another one. We extend Pipette to support chained RAs, which perform the work of several consecutive indirections. BFS contains an opportunity for chained RAs.

As an example, in BFS, `edge_start` comes from loading `g->nodes[v]`. Now, instead of directly performing an indirection, the process current fringe stage only needs to enqueue `v` to a reference accelerator configured to indirect on `g->nodes`. The enumerate neighbors stage simply dequeues the value of `edge_start` as an output of the RA. These two values form the starting and ending indices for `g->edges`, so we can chain this to a second scanning RA to read neighbors (`ngh`) out of `g->edges`. Chained RAs free us to devote general-purpose threads and core resources to application compute, rather than manipulating queues.

5.3 Phloem Design

We now present Phloem's design and key techniques. We first explain Phloem's programming interface, introduce Phloem's core transformations to produce efficient pipelines, and present additional features.

5.3.1 Phloem interface

Phloem transforms serial code, starting with a program written in C. Other interfaces are possible, e.g., we later introduce a TACO-based frontend. Our C-based interface is not limiting, because Phloem's key challenge is not to find pipeline parallelism, but to generate efficient pipelines.

Phloem is automatic, but programmers can control some aspects through the pragma annotations listed in Table 5-1.

#pragma	Function
phloem	Mark this function for automatic pipeline parallelization.
decouple	Separate the following instructions into a new stage.
replicate	Make copies of the pipeline to fill hardware resources.
distribute	Send values to another replica identified by a user-specified function.

Table 5-1: Summary of Phloem annotations.

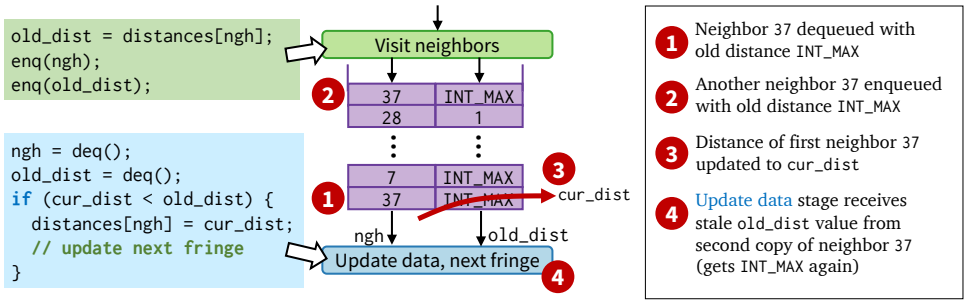


Figure 5-2: A race condition in BFS that would arise with an incorrect decoupling into pipeline-parallel stages.

Phloem transforms single procedures: Phloem currently works on a single procedure; this is not a major limitation in our experience, as the main kernel of an irregular application typically fits in a concise definition in a single function. Calls to other functions are supported, but Phloem does not decouple within those calls. Inlining could remove this limitation; we leave this to future work.

Memory and aliasing: To preserve the semantics of the serial program, Phloem requires information about memory beyond C's standard semantics. Specifically, the programmer must provide precise aliasing information, e.g., by tagging pointers with C's `restrict` keyword. Modern high-performance programs often do this already, as it enables other compiler optimizations; we shortly discuss how to handle situations without precise aliasing information. This enables Phloem to safely transform code that reads and writes memory, by ensuring involved operations cannot alias. In addition, Phloem does not attempt to track and transform value communication through memory (i.e., load-store telescoping).

One of the most significant benefits of this approach is that it prevents race conditions; nevertheless, Phloem can work with such code provided that some care is taken. Figure 5-2 shows the race in BFS described in Section 3.2.3: if we pipelined the lookup of neighbors, a given neighbor

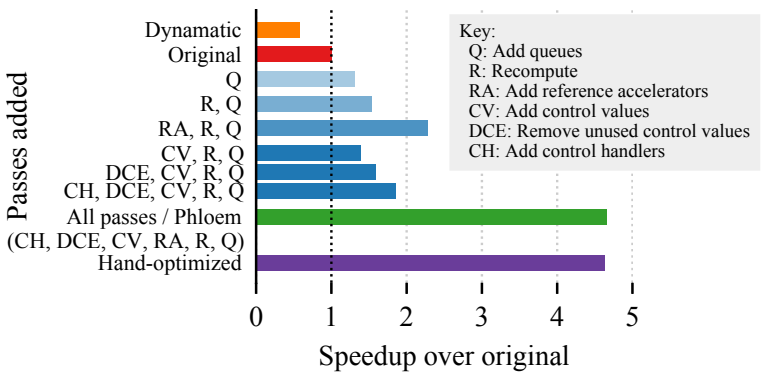


Figure 5-3: Speedup over the original serial BFS implementation with each added pass.

(for instance, neighbor `37`) may appear as a neighbor of multiple edges. If we also pipeline the lookup of `old_dist`, and the update data stage updates the distance of neighbor `37`, then any already-queued copies of neighbor `37` will have a stale value for `old_dist`, as it has changed.

Avoiding races like this one requires a simple compiler analysis: placing reads and writes to the same data structure, or doing so through pointers that may alias, in separate stages is disallowed. Phloem also allows programmers to instruct it to be more conservative around code that might cause race conditions. However, Phloem may still *prefetch* data in this case. In the above example, visit neighbors and update data can still be decoupled to prefetch neighbor distances, but update data must read and update the distances itself to avoid observing stale data.

Program phases: Phloem generally decouples one loop nest (of arbitrary depth) at a time; some programs, like PageRank-Delta, are structured as phases of several loop nests that successively build on each other. These loop nests can be decoupled individually, but in general, their execution cannot be overlapped. To ensure correctness in this case, Phloem inserts synchronization to ensure that all the stages complete the previous loop nest before moving onto the next one.

5.3.2 Producing efficient fine-grain pipelines

Phloem produces efficient pipeline-parallel programs with the systematic application of six passes. We introduce these passes with a detailed example using BFS, showing how they complement each other and progressively improve performance.

Figure 5-3 shows the performance benefits of implementing BFS on Pipette and running it on a large road network input: BFS achieves a 4.6× speedup over the serial implementation. We emphasize that these performance benefits were achieved by *manually* applying non-trivial insights about efficient execution of pipeline-parallel programs; we now describe a potential pitfall of working at the wrong abstraction level.

A more general, but costlier, approach than pipeline parallelism is to leverage a *dataflow architecture* [9, 31], where computations are expressed as a graph of operators that communicate values, and the availability of data drives the execution of operators. Dataflow execution can hide long latencies because operators do not run in any particular order. We initially studied Dynamatic [56], a state-of-the-art technique to transform serial code into dataflow graphs. We used it to map BFS to a dataflow graph, with which we simulated dataflow execution: any operation can begin as soon as its inputs are available. However, as Figure 5-3 shows, the performance of this dataflow graph is very poor: 1.7× *worse* than the serial version. The chief reason is that pipeline stages are *extremely* sensitive to overhead, and Dynamatic, which breaks down applications into basic blocks, must track all of the context and control information of each block. This is the case even if these values are unused or easily recomputed. Thus, Dynamatic's dataflow graphs propagate significant amounts of program state across stages. These extra storage overheads and operations ruin throughput, just like how extra instructions in the innermost loops in serial programs hurt throughput.

We now show how to Phloem's passes achieve an efficient pipeline-parallel BFS. To make discussion concrete, we focus on one stage of BFS, enumerate neighbors, and show how each of the six passes applies to this stage's code. Figure 5-4 shows all but the last pass (which operates across multiple stages).

Decouple: Before applying any transformations, Phloem first identifies where to decouple code into stages. Phloem decouples across long-latency loads, because these are BFS's main source of irregularity. Because each loop level usually contains a long-latency load, decoupling an irregular application often results with each loop level to its own stage. Choosing decoupling points is critical for performance; Section 5.4 fully describes how Phloem selects these points.

The loop level in the enumerate neighbors stage iterates over the variable `e` to traverse the `g->edges` array, based on values of `edge_start`

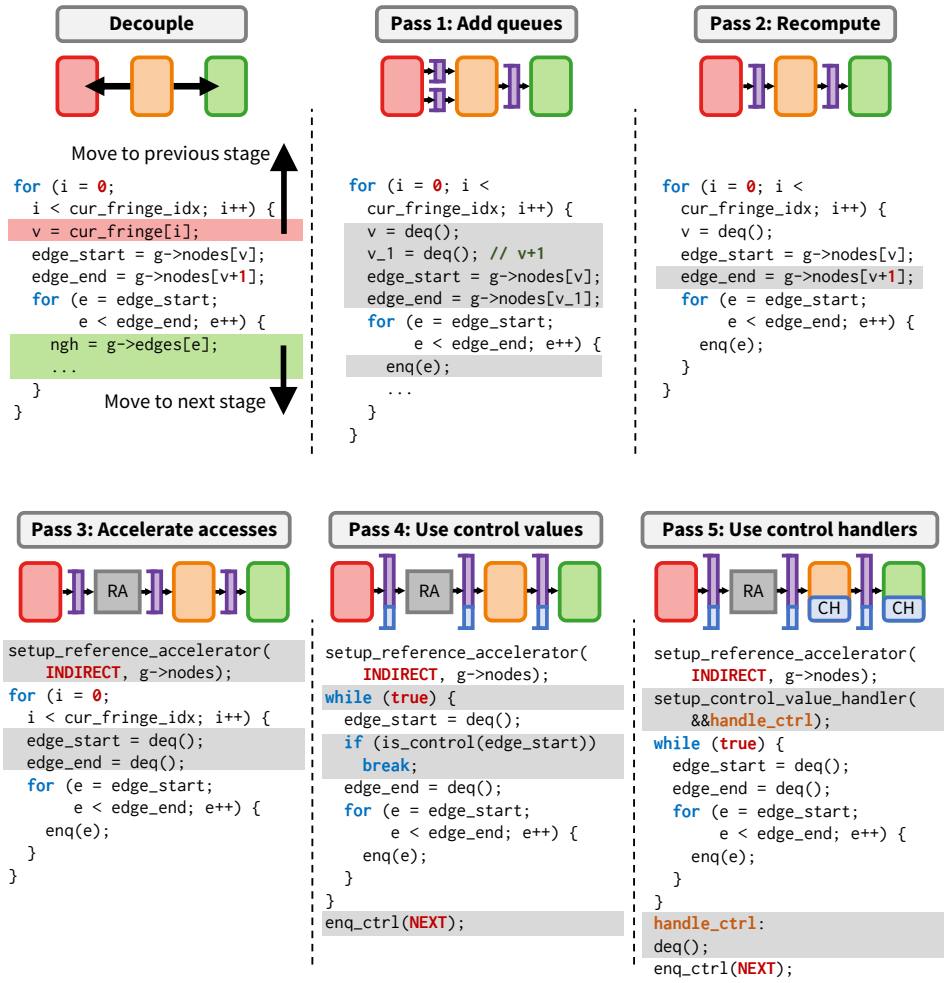


Figure 5-4: Passes to transform serial code into an efficient pipeline. After the initial decoupling step, each column shows successive transformations of the enumerate neighbors stage (changes shaded in gray). Pass 6, inter-stage dead code elimination, is not shown.

and edge_end. But now we have a context management problem: we need to communicate the current vertex v to this stage.

1. Add queues: The simplest way to achieve a functionally correct pipeline is to pass every needed value through a queue. Phloem adds queues to communicate the values of v and v+1, which were produced in the previous stage. However, passing everything through queues produces poor results because queue communication has overhead. For example, sending v+1 is not needed because it can be recomputed from v.

2. Recompute: Some values change infrequently, or can be determined without communication from another stage. In this case, we can simply recompute the value (similar to *rematerialization* in compiler literature). A great candidate for this optimization is index computations: calculating `v+1` rather than passing it through a queue is more efficient.

3. Accelerate accesses: In the enumerate neighbors stage, Phloem can use Pipette's reference accelerators to offload accesses to `g->nodes`, and since both `edge_start` and `edge_end` access this array, we can route them through the same RA: the producer simply enqueues `v` and then `v+1`.

4. Use control values: We can signal the end of an edge list by sending the `NEXT` control value, which the stage detects with the `is_control()` function. If so, the code breaks out of this loop. Now, stages simply check for a control value rather than recompute the loop condition.

5. Use control value handlers: Instead of checking for control values in the inner loop, Phloem sets up control handlers: these process control values and, if necessary, send more control values to downstream stages and break out of inner loops. At initialization, Phloem configures Pipette with the address of the control handler (the code uses the `&&` unary operator to indicate taking the address of a label).

6. Inter-stage dead code elimination: Finally, Phloem improves decoupling by performing inter-stage dead code elimination on superfluous control values. Using control values represents breaking out of inner loops in the original sequential source code. However, separating the program into stages means that some of these outer loops in some stages are empty—this means that a control value used at this loop level is unneeded. For example, all vertices visited in an iteration of BFS are compared to the same distance. It is unnecessary to distinguish which vertex a particular neighbor vertex belonged to. A naive implementation of control values, however, would send an unnecessary control value after the end of each edge list. By eliminating this control value, downstream stages can simply process all vertices until the iteration ends.

Figure 5-3 shows the impact of applying these techniques. To better understand the performance impact of each pass, we show multiple intermediate combinations of theses passes. For instance, three of the control-based passes (corresponding to CV, DCE, CH in Figure 5-3) build successively on each other. The plot shows the performance improvement of adding each of these passes to code which already contains queues (Q) and recomputation (R). The benefits of adding control values,

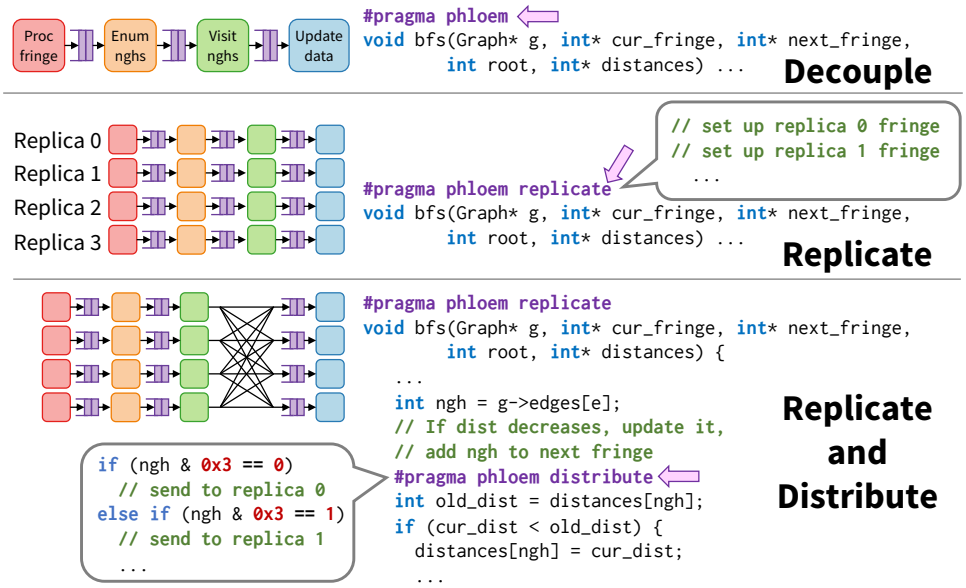


Figure 5-5: Replicating a decoupled pipeline and distributing work between replicas.

eliminating unused control values, and adding control handlers (the bar labeled CH, DCE, CV, R, Q) culminates in a 1.85× speedup over the original non-decoupled code.

With 4.7× speedup over the original code, the performance of Phloem's emitted BFS now even exceeds that of hand-optimized code. Moreover, Phloem accomplishes this performance through simple static inspection of the program, whereas the manually optimized version needed to leverage application-specific insight about communication patterns.

5.3.3 Composing data and pipeline parallelism

As we saw in Pipette (Section 3.5.6) and Fifer (Section 4.5.5), data parallelism and pipeline parallelism flexibly compose; *pipeline replication*, enabled by Pipette's support for cross-core queue communication, lets us fully exploit the resources of modern multicore systems.

For instance, a single BFS pipeline can be replicated over many cores so that each pipeline works on a specific part of the input graph as shown in Figure 5-5. Working on disjoint parts of the input eliminates the need for expensive synchronization operations across shared memory.

With time-division multiplexing of stages in each core, each replicated pipeline can also mitigate load imbalance. This implementation can then

use a simple partitioning scheme, like examining bits of the neighbor vertex id, to determine which replica to send neighbors to. This shows that exploiting pipeline and data parallelism together can lead to simpler implementations than exploiting data parallelism alone.

By marking a function with `#pragma replicate` and specifying the number of replicas, a programmer can create multiple copies of a pipeline. By default, these pipelines operate independently but over the same data; Phloem does not automatically infer which data structures are shared or replicated. Instead, by defining a simple `replicate_arguments()` function, a programmer can indicate how to partition work across the pipelines. For instance, in a replicated BFS, each replica works on its own fringe array, so this function would allocate new `cur_fringe` and `next_fringe` arrays for each one. The partition need not be complex, because each pipeline is automatically load-balanced by the underlying hardware. As a result, the replicas proceed at roughly the same rate, even with irregularities in the stages.

To better exploit locality, Phloem also allows pipelines to distribute work in a data-centric way, by allowing one replica to enqueue work to not only its next stage but also the corresponding stage of *any* replica. The programmer defines another simple function to describe how to select which replica will receive the enqueued value. In BFS, adding `#pragma distribute` between the visit neighbor and the update data stages splits the replicas into *source-centric* and *destination-centric* sections; selecting the replica simply involves inspecting bits in the neighbor id. This improves data locality in the update data stage because each replica works on separate parts of the graph.

5.3.4 Making efficient domain-specific pipelines

Phloem's implementation as a source-to-source transform on C/C++ is crucial, as C/C++ remains the lingua franca of domain-specific accelerator compilers [10, 16, 61, 101, 142] and frameworks [13, 106]. Using C as an input language to Phloem, then, makes it possible to seamlessly pass code to and from these compilers and frameworks. These compilers emit code with structure that Phloem can easily discover; this process would take considerable time for a human to do the same. Furthermore, compilers emit code that already meet Phloem's input requirements; for example, these tools emit data structures that are qualified with the C/C++ `restrict` keyword.

As a case study, we examine Phloem's performance on a variety of automatically generated sparse linear algebra kernels from the Tensor Algebra Compiler (Taco) [61]. Taco accepts a tensor expression that represents operations on sparse tensors, such as the multiplication of a matrix A by a vector x with the expression `y(j) = A(i,j) * b(i)`. Then, given the storage formats of each tensor (for instance, A could be stored in CSR format), Taco produces C code. We then process this C code with Phloem to produce an efficient pipeline-parallel implementation to be executed on Pipette.

5.4 Automatic Decoupling

The previous section presented the techniques that enable Phloem to produce efficient pipeline-parallel programs. The remaining challenge in implementing Phloem is finding decoupling points. Choosing where to decouple is crucial, as missing frequent irregular accesses hurts performance and these accesses are not always easy to identify.

Phloem intermediate representation (IR): Phloem transforms the C abstract syntax tree into a custom intermediate representation (IR) that represents fine-grain operations (e.g., load, add) within the program. It is important to represent the program as a fine-grain graph because decoupling could conceivably occur at any point in the program. That said, it is *not necessary* to decouple every operation from each other; we will show that decoupling at just a few (3 or 4) strategically chosen points is enough for good performance.

Decoupling in the Phloem IR amounts to selecting edges that will be replaced by queued communication. Importantly, and unlike conventional IRs like LLVM's, Phloem's IR supports operations to manipulate queues as previously described: to carry out communication and convey control flow changes. These have the effect of restoring definitions of values to these subgraphs.

Determining decoupling points statically: Phloem can find decoupling points using static analysis. In this mode, Phloem ranks all possible decoupling points using a cost model, and selects the $(N-1)$ highest-ranked points to build an N-stage pipeline. Each stage is assigned to a separate thread.

Phloem's cost model prioritizes decoupling points according to two factors: how expensive the memory access is predicted to be, and how

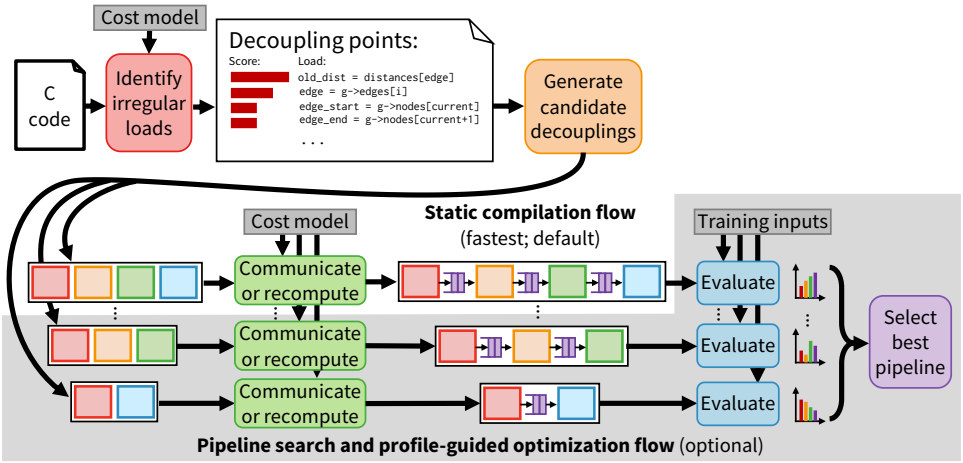


Figure 5-6: How Phloem selects decoupling points, generates pipelines, communicates data between stages, and outputs a pipeline.

frequently it happens. First, to estimate access cost, Phloem considers both whether the access is indirect (rather than sequential) and whether there are other accesses nearby. For instance, the BFS example has two nearby accesses to `g->nodes`. The first access is an indirection, so it is predicted to be costly. However, the second access touches the location after the first one, so it is very likely a cache hit, and is predicted to be cheap. This biases these two accesses to happen together, rather than in two separate stages. To estimate access frequency, Phloem gives higher weight to memory references located in the innermost loops and less weight to infrequent accesses in the outer loop. Accordingly, the access to `g->edges` is considered more even more costly than to `g->nodes`, and would be prioritized for decoupling.

Figure 5-6 (center) shows the static compilation flow, in which only one pipeline is generated and no training occurs. This simple, static approach works well and produces pipelines that approach manually optimized versions. This makes sense, as the innermost loops typically demand the highest throughput but also come at the end of the longest chains of indirections. By building stages from the innermost loop outwards, we usually produce a pipeline that decouples the most performance-critical sources of irregularity.

Phloem makes it simple to target the stage count that matches the number of threads supported by the architecture: 2, 4, or even 8. Phloem can generate pipelines with more stages than there are threads on a core;

just as we generated replicated pipelines (Section 5.3.3), it is similarly possible to generate non-replicated pipelines spanning multiple cores.

Autotuning decoupling points: Though the above approach produces reasonable pipelines, the cost model is by necessity approximate: in irregular applications, the misses to each data structure and the loop lengths are highly input-dependent, and often vary over time. To improve performance, Phloem includes a profile-guided optimization mode. In this mode, Phloem selects more than $(N-1)$ candidate decoupling points from the highest-ranked ones, and then builds the candidate pipelines stemming from all their combinations. These pipelines are then profiled on small training inputs to find the best one. Figure 5-6 also illustrates this process, with the pipeline search and profile-guided optimization shaded in gray. This process, which completes in under an hour, allows exploiting decoupling points that are statically ranked below the bar, but happen to be more profitable.

The static compilation process completes within seconds, and its pipelines also work well in practice. Our evaluation compares the performance of pipelines generated by both the static compilation and profile-guided optimization modes.

5.5 Methodology

We implement Phloem as a source-to-source compiler. For each of our benchmarks, we start with high-quality serial implementations. Phloem automatically identifies decoupling points based on a simple heuristic of the costliest operations (long chains of dependent references in deep loop nests), and produces pipeline-parallel versions. We then compile Phloem-generated code with `gcc -O3`.

5.5.1 Evaluated systems

We evaluate Phloem's generated benchmarks on an extended version of Pipette (Section 5.2) and use Pipette's evaluation configuration, including the same core parameters. We also use the same benchmarks as Pipette with the exception of Silo (as Phloem does not yet detect bounded cycles in loops).

Taco benchmarks: We integrate Phloem with the Tensor Algebra Compiler (Taco) [61] to automatically compile tensor algebra expressions into

Domain	Graph	Vertices	Edges	Avg. deg.
Training inputs				
Training internet graph	`internet`	126K	207K	1.7
Training road network	`USA-road-d-NY`	264K	734K	2.8

Table 5-2: Training graphs, sorted by the number of edges. (The test graphs are listed in Table 3-4.)

Domain	Matrix	Size ($n \times n$)	Avg. nnz/row
SpMM training inputs			
Training graph as matrix 1	`email-Enron`	36,692	10.0
Training graph as matrix 2	`wiki-Vote`	8,297	12.5
Taco (MTMul, Residual, SpMV, SDDMM) test inputs			
Circuit simulation	`scircuit`	170,998	5.6
Economics	`mac_econ_fwd500`	206,500	6.2
Particle physics	`cop20k_A`	121,192	21.7
Structural	`pwtk`	217,918	52.9
Cantilever	`cant`	62,451	64.2

Table 5-3: Input matrices, sorted by average non-zeros per row. (Test matrices used by SpMM are listed in Table 3-5.)

pipeline-parallel programs. We compare Phloem-generated pipelines with Taco-generated serial and data-parallel versions. We use the following Taco benchmarks:

Sparse Matrix-Vector Product (SpMV) evaluates $y = Ax$, where x and y are dense vectors and A is a sparse matrix.

Sampled Dense-Dense Matrix Multiplication (SDDMM) evaluates $A = B \circ (CD)$, where C and D are dense matrices, A and B are sparse matrices, and the $\circ$ operator represents component-wise multiplication.

Matrix-Transpose Multiplication (MTMul) evaluates $y = \alpha A^\intercal + \beta z$, where α and β are constants; x, y, and z are vectors; and A is a sparse matrix.

Residual evaluates $y = b - Ax$, where b, x, and y are vectors and A is a sparse matrix.

Inputs and sampling: We use the inputs evaluated by Pipette (Section 3.4) as test inputs; training inputs for the graph benchmarks are listed in Table 5-2 and training inputs for SpMM are listed in Table 5-3. (We

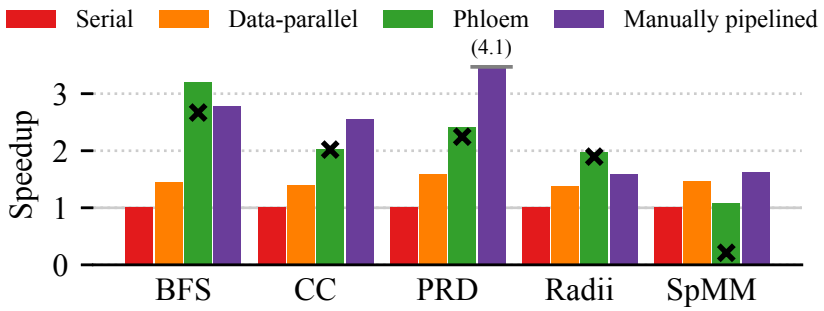


Figure 5-7: Per-benchmark speedup over the serial baseline. Each Phloem bar (green) shows the performance of a pipeline produced through profile-guided optimization; an × indicates the performance of a pipeline produced by the static cost model.

also use the same sampling methodology for PRD and Radii as Pipette.) We evaluated the Taco benchmarks using the same matrices as its original evaluation; Table 5-3 also lists these matrices.

5.5.2 Automatic pipeline generation and search

We use Phloem to automatically generate all pipelines of up to four threads. (This results in no *fewer* than fifty different pipelines for each benchmark.) We then select the best pipeline by running each pipeline configuration on a small set of training inputs: for graph applications, `internet` and `USA-road-d-NY`; for SpMM, `email-Enron` and `wiki-Vote`. We then use the best-performing pipeline as determined by these training inputs and evaluate its performance on the test input set. Importantly, we *do not* present results for the best pipeline over all inputs *a priori*; nevertheless, training may identify the globally optimal pipeline. Finally, for simplicity, we use the static compilation flow for the Taco benchmarks.

5.6 Evaluation

Figure 5-7 reports the overall speedups of Phloem compared to the non-decoupled serial code, a competitive data-parallel implementation, as well as a manually optimized version. Each group of bars shows results for one application, and each bar represents the gmean speedup across inputs of each variant over the serial version. For Phloem, the height of the bar shows the result for the pipeline produced by profile-guided

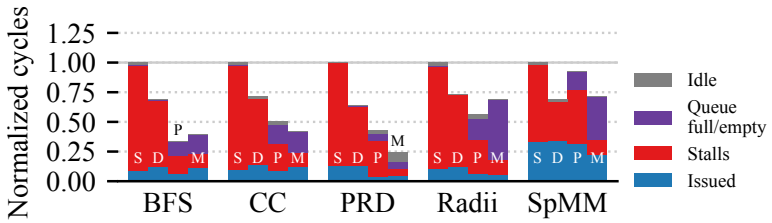


Figure 5-8: Breakdown of cycles, normalized to serial baseline (S: Serial, D: Data-parallel, P: Phloem, M: Manually pipelined).

optimization, and the × mark shows the speedup of the pipeline produced by the static compilation flow.

Phloem achieves significant speedups—on average, 1.7× over serial—which are comparable to that of the hand-tuned version. In all but one case, the performance of the Phloem version not only surpasses the serial version but also a competitive data-parallel implementation. On average, Phloem achieves 85% of the performance gains of the manually optimized version. In two applications, BFS and Radii, Phloem *outperforms* the manually optimized version. SpMM shows a negative result, where Phloem does not improve performance.

Figure 5-8 gives more insight into these results by showing a breakdown of cycles spent by cores. Each group of bars reports cycles for one application, relative to the serial baseline. Each bar is broken down in cycles spent *(i)* issuing micro-ops, and waiting on *(ii)* backend stalls (including memory latency), *(iii)* full or empty queues (for Phloem and Manually pipelined), or *(iv)* other stalls (e.g., frontend).

Comparing the manually optimized code to the Phloem-generated BFS code, the Phloem version runs slightly fewer instructions. Threads also block less often from full or empty queues in the Phloem version, keeping OOO core resources busy and resulting in Phloem beating the hand-optimized pipeline by 15%. Radii primarily benefits from a better decoupling that reduces queue stalls. CC and PRD, on the other hand, show slightly worse decouplings, both due to increased memory stalls, but still do much better than data-parallel.

Finally, Phloem does not benefit SpMM. The reason is that SpMM's manual version uses a bespoke implementation of merge-intersect where, upon finding the end of an input queue through a control value, the consumer skips the remaining values in the other input queue up to its next control value. This reduces instructions and stalls by avoiding

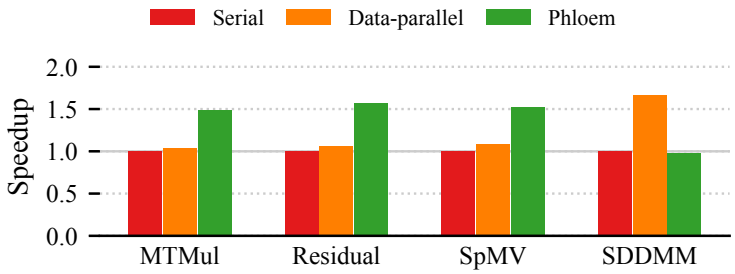


Figure 5-9: Speedups over serial baseline for Taco benchmarks.

ineffectual work. However, it is a highly application-specific insight that is hard to derive from serial code, and is thus unavailable to Phloem. This shows that there are some patterns for which a manual approach yields better performance.

Taco results: Figure 5-9 reports Phloem's speedups when parallelizing Taco programs. (This is similar to Figure 5-7, except that we lack manually optimized pipelines for these programs.) For MTMul, Residual, and SpMV, Phloem easily parallelizes the code, resulting in a gmean speedup of 1.5× over serial across their inputs for each of these benchmarks. Notably, data-parallel Taco versions barely improve performance, as the increase in instructions outweighs the benefits gained by turning to data parallelism.

SDDMM shows the opposite result: while Phloem-generated code shows no improvement over the serial version, the data-parallel version sees some speedup. This is because SDDMM, unlike the other benchmarks, has a regular innermost loop that multiplies *dense, uncompressed* matrices C and D. Conventional architectures handle this case well.

The speedups gained by simply adding Phloem as a pass to an existing domain-specific compiler showcases not only Phloem's generality, but also the applicability and effectiveness of fine-grain pipeline parallelism.

5.6.1 Analysis of generated pipelines

We now evaluate the impact of profile-guided optimization by examining the pipelines generated by the search process. Figure 5-10 shows the distribution of gmean speedups for pipelines of a given number of stages. Speedups are relative to the original serial code, and here, the number of stages *includes any reference accelerators used*. For instance, in BFS, the best 4-stage pipeline is 2.8× better than serial, while an 8-stage pipeline is only 2.4× better. This illustrates the many tradeoffs that exist when

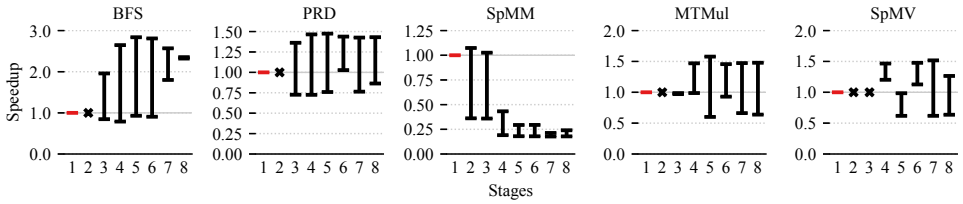


Figure 5-10: Plot showing the distribution of gmean performance over the training inputs of Phloem-generated pipelines for select benchmarks as the number of stages are varied. An × indicates that no pipeline of that length was profiled.

constructing pipelines, as adding too many stages can cause excessive communication that limits possible performance. This is underscored by SpMM, in which performance diminishes as stages are added for the reasons already discussed. Lastly, forcing a particular pipeline length could cause awkward pipeline stage boundaries that decrease performance, as seen in SpMV at 5 stages. Applying profile-guided optimization helps avoid falling into such minima. Overall, our automatic approach finds well-performing pipelines within the distribution.

5.6.2 Replicating pipelines

We evaluate how Phloem effectively uses multicore resources by producing replicated pipelines. In addition to the replicated BFS, presented in Section 5.3.3, we also evaluate replicated pipelines for CC, PRD, and Radii, all of which are also amenable to a data-centric partitioning scheme.

We scale the system to a 4-core system with 4 threads each, scaling the data-parallel system and replicating the Phloem and manual pipelines to use all threads. We use `#pragma replicate` and `distribute` annotations to direct Phloem to produce the replicated pipeline, which uses Pipette's inter-core queue communication to send work to other cores.

Figure 5-11 compares the performance of these systems to a single-core, single-thread serial configuration. In BFS and CC, the data-parallel system achieves speedups somewhat linearly with the number of cores, the manually pipelined versions of BFS and CC respectively achieve 12× and 7× better performance than the serial implementation. Phloem's automatically replicated versions of these pipelines perform 10× and 4× better than serial, and in both cases outperform the data-parallel system. Phloem's replicated Radii pipeline outperforms both the data-parallel and

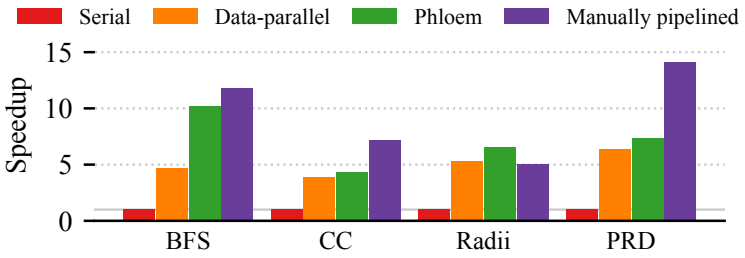


Figure 5-11: Performance of BFS, CC, PageRank-Delta, and Radii replicated over many cores, compared to serial, data-parallel, and manually pipelined versions.

manually pipelined versions. Unlike the other automatically replicated pipelines, which replicate 4 stages (plus RAs) four times across four cores, Radii's pipeline is 2 stages (plus RAs), replicated eight times across four cores. This organization better exploits data locality and reduces the impact of memory stalls. Finally, while PRD also outperforms the data-parallel implementation, its performance is about half that of the manual pipeline. The manual version merges the two middle stages together to make room for a second level of stage replication within each already-replicated pipeline. Phloem does not yet support this transformation automatically. On a final note, when replicating pipelines, Phloem also selects different pipeline configurations than merely replicating stages from the single-replica configuration. This change further reinforces the need for automatic parallelization: changes to the pipeline alter the tradeoffs needed to get good performance.

5.7 Additional Related Work

In this section, we discuss prior work not covered so far.

Domain-specific software frameworks: Several frameworks and languages may feature pipeline-like behavior, but they ultimately address different problems than Phloem.

For example, Ligra [106] accelerates graph applications by first casting them into combinations of edge-centric and vertex-centric operators. It achieves good performance on graph analytics applications due to graph-specific optimizations applied to these operators.

Halide [101] is a domain-specific language for image processing pipelines, but its optimizations are highly tuned for its specific domain.

For instance, it cannot handle arbitrarily deep chains of dependent memory accesses, which have no meaningful analogue in image processing. Like specialized hardware accelerators, these domain-specific frameworks achieve good performance within their intended domain, but would require wholly different tradeoffs to adapt to other domains.

T2S-Tensor [111] extends Halide to create spatial accelerators for dense, uncompressed tensor kernels. However, irregular applications, Phloem's target application domain, do not map to systolic arrays in a performant way.

Phloem sidesteps the limitations of domain-specific accelerators by exploiting a general technique, pipeline parallelism. Nevertheless, despite having orthogonal goals to these frameworks, we demonstrated that Phloem can be easily combined with them, serving as a backend to produce efficient pipeline-parallel code from the code emitted by these frameworks.

Partitioning dataflow graphs: Several prior techniques perform *temporal partitioning* [20, 53, 99] in the context of reconfigurable architectures, which divides a dataflow graph into multiple partitions. By executing each partition over multiple time steps, a reconfigurable architecture can perform a computation that is been otherwise too large to fit all at once.

Many crucial shortcomings of these works make them insufficient for partitioning our dataflows. First, communication between partitions is only concerned with minimizing the number of reconfigurations and the amount of communication, so it is acceptable to slowly communicate data between partitions through memory on the reconfigurable unit [99] or main memory [59]. Second, these prior works generally examine applications that give rise to regular implementations on reconfigurable architectures, like signal processing (*e.g.*, filters, FFT, and DCT). Third, these techniques *statically* schedule partitions [52, 129] and run them in a fixed order (*e.g.*, round-robin [60, 126]), so they cannot mitigate load variation by dynamically selecting a partition. Finally, these approaches target reconfigurable architectures; Phloem generalizes this approach to general-purpose architectures.

Compilers for spatial architectures solve problems adjacent to those of Phloem, as they need to map operations to architectures supporting some form of dataflow. However, they have all focused on mappings in which the configuration (and thus computation) is fixed throughout the application's execution [17, 144]. SARA [141] exploits coarse-grain data parallelism in mapping applications to the Plasticine architecture [98], but

its focus is not on irregular applications; instead, it focuses on compute-heavy regular applications with little control. Some compilers identify parallel patterns for FPGA implementation, including pipelines [97], but these all assume that the entire pipeline can be mapped to the hardware at once, resulting in different tradeoffs.

Phloem's ability to also generate configurations for spatial architectures superficially resembles high-level synthesis (HLS) and the intermediate representations (IRs) that have been recently developed in service of that goal. For instance, Calyx [85] is an IR intended for hardware implementation, but relies on unrolled loops for exploiting parallelism in regular applications. Its support for pipelines remains immature—in fact, it explicitly identifies the need for compiler support for supporting (explicit) pipeline parallelism.

Moreover, Phloem generates pipelines for dynamic time-multiplexed architectures, which have different design considerations. For instance, Aetherling [33] produces statically scheduled streaming hardware circuits, but it specifically does not support variable-latency operators—a hallmark of irregular applications. Moreover, dynamic time-multiplexed architectures contain features for efficient control and memory accesses that are required to make irregular applications run well—features that are not exploited by prior work.

5.8 Summary

Irregular applications use both conventional and specialized architectures poorly. Pipette and Fifer present simple hardware support to exploit untapped fine-grain pipeline parallelism in irregular applications by structuring them as pipelines. However, they needed manually created and painstakingly hand-tuned applications that were produced through a tricky and error-prone process. Phloem automates the extraction of efficient fine-grain pipeline parallelism in irregular applications, systematizing numerous insights in a robust and composable approach. This enables automatic high performance on irregular applications and helps make Pipette's and Fifer's hardware support more easily available to programmers.

6 Conclusion and Future Work

Advances in computer architecture have enabled significant changes in the computing landscape, but they have also increased the demand for processing more data in more complex ways. The introduction of irregularity, in the form of unpredictable memory accesses and control flow, threatens continued progress in today's important emerging applications, because current systems mainly target regular applications.

This thesis demonstrates that accelerating irregular applications with fine-grain pipeline parallelism is practical and effective with an end-to-end approach consisting of two hardware architectures and a compiler.

The first hardware architecture, *Pipette*, takes advantage of existing structures in modern OOO cores to implement fine-grain pipelines of irregular applications. Specifically, it reuses the physical register file as backing storage for decoupled communication, and simultaneous multithreading to ensure high utilization even when some pipeline stages stall. It also introduces inexpensive mechanisms for handling control flow efficiently and accelerating common memory accesses. Pipette achieves nearly two-fold gmean speedup over a variety of irregular applications.

The second hardware architecture, *Fifer*, extends Pipette's insights to specialized architectures and shows that fixed-latency compute-rich fabrics can be used to efficiently implement high-performance irregular application pipelines. Fifer's double-buffered reconfiguration scheme lowers the cost of using these highly efficient fabrics, and its adaptation of the control flow handling and memory access acceleration techniques from Pipette makes implementing irregular applications practical. Thus, Fifer makes the efficiency benefits of spatial reconfigurable architectures available to irregular applications.

Finally, *Phloem*, the compiler, makes it easy to tap into this new hardware support by transforming serial code into efficient pipeline-parallel implementations. It systematizes the creation of pipelines and their stages,

using cost models that identify chains of challenging indirections that need decoupling. With both a static compilation flow as well as a profile-guided optimization flow, Phloem automates the laborious trial-and-error process of manually creating pipelines while retaining 85% of the performance improvements gained by making pipelines by hand. Phloem thus makes the acceleration benefits of architectures like Pipette and Fifer much more accessible to programmers.

6.1 Future Work

This line of work opens exciting avenues for future research.

Expanding the repertoire of pipelined irregular applications: This thesis has demonstrated that latent pipeline parallelism is plentiful and can be exploited efficiently. Further inquiries into other irregular applications can yield more opportunities for acceleration.

Bringing dynamic pipeline parallelism to hardware: Current hardware architectures make it infeasible to automatically pipeline-parallelize arbitrary binary code due to race conditions (see Phloem's description of race conditions in Section 5.3). While Phloem offers mechanisms to automatically cope with these conditions, a programmer must often intervene with application-specific insights to maximize performance.

This opportunity can also be viewed as the difference between *explicit* and *implicit* parallelization. These architectures rely on a programmer (or compiler) to *explicitly* harness pipeline parallelism by decoupling programs into stages. Future work could add architectural support for *implicit* pipeline parallelism, to automatically infer pipelines in the same way that out-of-order cores can extract instruction-level parallelism from unmodified binary code by knowing which instructions to run out of order. Such additions could make the benefits of pipeline parallelism available without specific compiler support or programmer intervention.

Extending instruction sets for pipelines: Extensions to the ISA to support SIMD instructions, like MMX, SSE, and AVX for the Intel x86 architectures, have made exploiting data-level parallelism accessible to programmers and compilers. An extension to a modern, clean instruction set like RISC-V [133] with instructions to support pipeline semantics can improve interoperability between the many proposed systems that leverage hardware queues. Even within a single system, a standard queue-based communication interface might need to coordinate between

heterogeneous units, like accelerators and general-purpose processors, all across the memory hierarchy or within a single core.

Developing decoupled functional units: Pipette's and Fifer's reference accelerators have repeatedly proved themselves crucial in improving the performance of irregular applications. So useful, in fact, that similar decoupled, specialized structures to fetch individual values, or ranges of values, from memory appear in other work [46, 116, 138]. A full treatment of these types of units may help drive the inclusion of these units (controlled by the aforementioned instruction set extension) in future commercial architectures.

Targeting new architectures: While Pipette and Fifer target out-of-order cores and coarse-grain reconfigurable arrays, the techniques developed here can be applied to a variety of other architectures. Pipeline parallelism is especially relevant where plenty of work can be buffered and then fed through a high-throughput datapath in rapid succession. In particular, GPUs are a good candidate for future exploration.

Balancing general-purpose and specialized support for pipeline parallelism: Pipette and Fifer represent two points at opposite extremes in a tradeoff between general-purpose and specialized computation. Even in Pipette, in which any computation can be carried out by the general-purpose core, the reference accelerators provide specialization that is crucial for performance. A future system could devote more area to specialized hardware that more closely resembles RAs in functionality and size, but leave a few general-purpose cores to have the flexibility to handle stages that are better expressed as instructions.

Executing regular programs more efficiently: This thesis demonstrated the effectiveness of exploiting fine-grain pipeline parallelism on irregular applications. Future work could examine how to support both regular and irregular applications. In particular, we have shown that building temporal pipelines by time-multiplexing compute resources places less demands on chip area because it is cheaper than duplicating compute structures outright. We could accelerate regular applications in a way that is more resourceful than simply deepening speculative structures or widening vector lanes.

Bibliography

[1] "phloem, n." in *OED Online*. Oxford University Press, 2021.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. A. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife machine: Architecture and performance," in *Proceedings of the 22nd annual International Symposium on Computer Architecture (ISCA-22)*, 1995.

[3] A. Agarwal, J. Kubiatowicz, D. A. Kranz, B. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: an evolutionary processor design for large-scale multiprocessors," *IEEE Micro*, vol. 13, no. 3, 1993.

[4] A. Agarwal, B. Lim, D. A. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing," in *Proceedings of the 17th annual International Symposium on Computer Architecture (ISCA-17)*, 1990.

[5] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the International Conference on Supercomputing (ICS'16)*, 2016.

[6] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the 23rd international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018.

[7] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proceedings of the 18th International Conference, Euro-Par*, 2012.

[8] R. Alverson, D. Callahan, D. Cummings, B. D. Koblenz, A. Porterfield, and B. J. Smith, "The Tera computer system," in *Proceedings of the International Conference on Supercomputing (ICS'90)*, 1990.

[9] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Trans. Computers*, vol. 39, no. 3, 1990.

[10] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 17th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[11] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM TACO*, 2017.

[12] R. D. Barnes, J. W. Sias, E. M. Nystrom, S. J. Patel, N. Navarro, and W. W. Hwu, "Beating in-order stalls with "flea-flicker" two-pass pipelining," *IEEE Trans. Computers*, vol. 55, no. 1, 2006.

[13] A. Brahmakshatriya and S. P. Amarasinghe, "BuildIt: A type-based multi-stage programming framework for code generation in C++," in *Proceedings of the 19th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021.

[14] A. Brahmakshatriya, E. Furst, V. A. Ying, C. Hsu, C. Hong, M. Ruttenberg, Y. Zhang, D. C. Jung, D. Richmond, M. B. Taylor, J. Shun, M. Oskin, D. Sanchez, and S. P. Amarasinghe, "Taming the zoo: The unified GraphIt compiler framework for novel architectures," in *Proceedings of the 48th annual International Symposium on Computer Architecture (ISCA-48)*, 2021.

[15] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. P. Amarasinghe, "Compiling graph applications for GPUs with GraphIt," in *Proceedings of the 19th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021.

[16] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT-20)*, 2011.

[17] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *Proceedings of the 11th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004.

[18] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G. Wei, and D. M. Brooks, "HELIX-RC: an architecture-compiler co-design for automatic parallelization of irregular programs," in *Proceedings of the 41st annual International Symposium on Computer Architecture (ISCA-41)*, 2014.

[19] S. Campanoni, T. M. Jones, G. H. Holloway, V. J. Reddi, G. Wei, and D. M. Brooks, "HELIX: automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the 10th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2012.

[20] J. M. P. Cardoso, "On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures," *IEEE Trans. Computers*, vol. 52, no. 10, 2003.

[21] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE)," in *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL-2000)*, 2000.

[22] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. H. 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)*, 2017.

[23] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: performance and innovation," *IEEE Micro*, vol. 41, no. 2, 2021.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, 2010.

[25] N. C. Crago and S. J. Patel, "OUTRIDER: Efficient memory latency tolerance with decoupled strands," in *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA-38)*, 2011.

[26] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.

[27] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi, "Merrimac: Supercomputing with streams," in *Proceedings of the ACM/IEEE conference on Supercomputing (SC03)*, 2003.

[28] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*, 2004.

[29] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT-15)*, 2006.

[30] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, "A clustered manycore processor architecture for embedded and accelerated applications," 2013.

[31] J. B. Dennis and D. Misunas, "A preliminary architecture for a basic data flow processor," in *Proceedings of the 2nd annual International Symposium on Computer Architecture (ISCA-2)*, 1974.

[32] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation Intel Core: New microarchitecture code-named Skylake," *IEEE Micro*, vol. 37, no. 2, 2017.

[33] D. Durst, M. Feldman, D. Huff, D. Akeley, R. G. Daly, G. L. Bernstein, M. Patrignani, K. Fatahalian, and P. Hanrahan, "Type-directed scheduling of streaming accelerators," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[34] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA-38)*, 2011.

[35] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[36] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, 1992.

[37] S. Friedman, A. Carroll, B. V. Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceedings of the 2009 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-17)*, 2009.

[38] M. Gao and C. Kozyrakis, "HRL: efficient and flexible reconfigurable logic for near-data processing," in *Proceedings of the 22nd IEEE international symposium on High Performance Computer Architecture (HPCA-22)*, 2016.

[39] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[40] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, 2000.

[41] J. R. Goodman, J. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "PIPE: A VLSI decoupled architecture," in *Proceedings of the 12th annual International Symposium on Computer Architecture (ISCA-12)*, 1985.

[42] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[43] V. Govindaraju, C. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proceedings of the*

17th IEEE international symposium on High Performance Computer Architecture (HPCA-17), 2011.

[44] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Comput. Archit. Lett.*, vol. 8, no. 1, Jan 2009.

[45] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: decoupled supply-compute communication management for heterogeneous architectures," in *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*, 2015.

[46] T. J. Ham, J. L. Aragón, and M. Martonosi, "Efficient data supply for parallel heterogeneous architectures," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, 2019.

[47] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49)*, 2016.

[48] K. Hegde, H. A. Moghaddam, M. Pellauer, N. C. Crago, A. Jaleel, E. Solomonik, J. S. Emer, and C. W. Fletcher, "ExTensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.

[49] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2014.

[50] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in *Proceedings of the 2013 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-21)*, 2013.

[51] N. Inc., "The NanGate 45nm open cell library," http://www.nangate.com/?page_id=2325, 2008.

[52] R. Jain, A. Mujumdar, A. Sharma, and H. Wang, "Empirical evaluation of some high-level synthesis scheduling heuristics," in *Proceedings of the 28th Design Automation Conference (DAC-28)*, 1991.

[53] Y. Jiang and J. Wang, "Temporal partitioning data flow graphs for dynamically reconfigurable computing," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 12, 2007.

[54] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proceedings of the 12th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[55] H. F. Jordan, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*. MIT Press, 1985.

[56] L. Josipovic, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-26)*, 2018.

[57] R. N. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 chip: A dual-core multithreaded processor," *IEEE Micro*, 2004.

[58] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine stream processor," in *Proceedings of the 20th International Conference on Computer Design (ICCD)*, 2002.

[59] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," in *Proceedings of the Design, Automation and Test in Europe (DATE)*, 1998.

[60] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications," in *Proceedings of the 36th Design Automation Conference (DAC-36)*, 1999.

[61] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. P. Amarasinghe, "The tensor algebra compiler," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.

[62] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th annual IEEE/ACM international symposium on Microarchitecture (MICRO-46)*, 2013.

[63] D. Koch and J. Tørresen, "FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proceedings of the 2011 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-19)*, 2011.

[64] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded Sparc processor," *IEEE Micro*, vol. 25, no. 2, 2005.

[65] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[66] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips, "SQRL: hardware accelerator for collecting software data structures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT-23)*, 2014.

[67] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, 1987.

[68] I. A. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang, "On-the-fly pipeline parallelism," in *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[69] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[70] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-42)*, 2009.

[71] D. Liu, S. Yin, G. Luo, J. Shang, L. Liu, S. Wei, Y. Feng, and S. Zhou, "Data-flow graph mapping optimization for CGRA with deep reinforcement learning," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2019.

[72] F. Liu, H. Ahn, S. R. Beard, T. Oh, and D. I. August, "DynaSpAM: dynamic spatial architecture mapping using out of order instruction schedules," in *Proceedings of the 42nd annual International Symposium on Computer Architecture (ISCA-42)*, 2015.

[73] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[74] P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative multithreaded processors," in *Proceedings of the International Conference on Supercomputing (ICS'98)*, 1998.

[75] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL-2003)*, 2003.

[76] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.

[77] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[78] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*, 2018.

[79] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th IEEE international symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[80] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the 24th Symposium on Operating System Principles (SOSP-24)*, 2013.

[81] M. Nguyen and J. C. Hoe, "Time-shared execution of realtime computer vision pipelines by dynamic partial reconfiguration," in *Proceedings of the 28th International Conference on Field-Programmable Logic and Applications (FPL-2018)*, 2018.

[82] Q. Nguyen and D. Sanchez, "Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism," in *Proceedings of the 53rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-53)*, 2020.

[83] Q. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *Proceedings of the 54th annual IEEE/ACM international symposium on Microarchitecture (MICRO-54)*, 2021.

[84] Q. Nguyen and D. Sanchez, "Phloem: Automatic acceleration of irregular applications with fine-grain pipeline parallelism," under submission.

[85] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proceedings of the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, 2021.

[86] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*, 2017.

[87] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *Proceedings of the 22nd IEEE international symposium on High Performance Computer Architecture (HPCA-22)*, 2016.

[88] T. Nowatzki, M. Sartin-Tarm, L. D. Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[89] M. O'Connor, N. Chatterjee, D. Lee, J. M. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems," in *Proceedings of the 50th*

annual IEEE/ACM international symposium on Microarchitecture (MICRO-50), 2017.

[90] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[91] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. T. Blaauw, T. N. Mudge, and R. G. Dreslinski, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proceedings of the 24th IEEE international symposium on High Performance Computer Architecture (HPCA-24)*, 2018.

[92] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. C. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. L. Allmon, R. Rayess, S. Maresh, and J. S. Emer, "Triggered instructions: a control paradigm for spatially-programmed architectures," in *Proceedings of the 40th annual International Symposium on Computer Architecture (ISCA-40)*, 2013.

[93] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. S. Emer, S. W. Keckler, and W. J. Dally, "SCNN: an accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*, 2017.

[94] J. Park and W. J. Dally, "Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures," in *Proc. SPAA-22*, 2010.

[95] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware / Software Interface*, 4th ed., 2008.

[96] G. Peng, L. Liu, S. Zhou, S. Yin, and S. Wei, "A 2.92-Gb/s/W and 0.43-Gb/s/MG flexible and scalable CGRA-based baseband processor for massive MIMO detection," *IEEE J. Solid State Circuits*, 2020.

[97] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. D. Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the 21st international*

conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI), 2016.

[98] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*, 2017.

[99] K. M. G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Trans. Computers*, vol. 48, no. 6, 1999.

[100] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th annual International Symposium on Computer Architecture (ISCA-40)*, 2013.

[101] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, 2012.

[102] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT-13)*, 2004.

[103] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT-20)*, 2011.

[104] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, 2010.

[105] A. Shrivastava, J. Pager, R. Jeyapaul, M. Hamzeh, and S. B. K. Vrudhula, "Enabling multithreading on CGRAs," in *Proceedings of the 40th International Conference on Parallel Processing (ICPP-2011)*, 2011.

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

[107] H. Singh, G. Lu, E. M. C. Filho, R. Maestre, M. Lee, F. J. Kurdahi, and N. Bagherzadeh, "MorphoSys: case study of a reconfigurable computing system targeting multimedia applications," in *Proceedings of the 37th Design Automation Conference (DAC-37)*, 2000.

[108] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th annual International Symposium on Computer Architecture (ISCA-9)*, 1982.

[109] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. Laudon, "The ZS-1 central processor," in *Proceedings of the 2nd international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, 1987.

[110] N. K. Srivastava, H. Jin, J. Liu, D. H. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proceedings of the 53rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-53)*, 2020.

[111] N. K. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. H. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. Herr, C. J. Hughes, T. G. Mattson, and P. Dubey, "T2S-Tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *Proceedings of the 27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM-27)*, 2019.

[112] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proc. VLDB*, 2015.

[113] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," in *Proceedings of the 9th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[114] M. Sung, R. Krashinsky, and K. Asanović, "Multithreading decoupled architectures for complexity-effective general purpose computing," *SIGARCH Comput. Archit. News*, 2001.

[115] V. Sze, D. F. Finchelstein, M. E. Sinangil, and A. P. Chandrakasan, "A 0.7-V 1.8-mW H.264/AVC 720p Video Decoder," *IEEE J. Solid State Circuits*, vol. 44, no. 11, 2009.

[116] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. M. Austin, M. F. P. O'Boyle, S. A. Mahlke, T. N. Mudge, and R. G. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *Proceedings of the 27th IEEE international symposium on High Performance Computer Architecture (HPCA-27)*, 2021.

[117] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded pipeline synthesis for data-parallel kernels," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2014.

[118] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima, "A CGRA-based approach for accelerating convolutional neural networks," in *Proceedings of the 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2015.

[119] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," in *Proceedings of the 35th annual IEEE/ACM international symposium on Microarchitecture (MICRO-35)*, 2002.

[120] M. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar operand networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, 2005.

[121] W. Thies, V. Chandrasekhar, and S. P. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *Proceedings of the 40th annual IEEE/ACM international symposium on Microarchitecture (MICRO-40)*, 2007.

[122] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, 2002.

[123] M. E. Thomadakis, "The architecture of the Nehalem processor and Nehalem-EP SMP platforms," 2008, Hot Chips.

[124] N. P. Topham and K. McDougall, "Performance of the decoupled ACRI-1 architecture: the perfect club," in *Proceedings of the International Conference on High-Performance Computing and Networking (HPCN)*, 1995.

[125] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic CGRAs for irregular loop specialization," in *Proceedings of the 27th IEEE international symposium on High Performance Computer Architecture (HPCA-27)*, 2021.

[126] S. Trimberger, "Scheduling designs into a time-multiplexed FPGA," in *Proceedings of the 1998 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-6)*, 1998.

[127] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the 24th Symposium on Operating System Principles (SOSP-24)*, 2013.

[128] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd annual International Symposium on Computer Architecture (ISCA-23)*, 1996.

[129] M. Vasilko and D. Ait-Boudaoud, "Architectural synthesis techniques for dynamically reconfigurable logic," in *Proceedings of the 6th International Conference on Field-Programmable Logic and Applications (FPL-1996)*, 1996.

[130] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data," in *Proceedings of the 47th annual International Symposium on Computer Architecture (ISCA-47)*, 2020.

[131] D. Voitsechov, O. Port, and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," in *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*, 2018.

[132] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[133] A. S. Waterman, "Design of the RISC-V instruction set architecture," Ph.D. dissertation, University of California, Berkeley, 2016.

[134] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: synthesizing programmable spatial accelerators," in *Proceedings of the 47th annual International Symposium on Computer Architecture (ISCA-47)*, 2020.

[135] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*, 2020.

[136] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/, 2014.

[137] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: the architecture and design of a database processing unit," in *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014.

[138] Y. Yang, J. S. Emer, and D. Sanchez, "SpZip: Architectural support for effective data compression in irregular applications," in *Proceedings of the 48th annual International Symposium on Computer Architecture (ISCA-48)*, 2021.

[139] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: indirect memory prefetcher," in *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*, 2015.

[140] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th international conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI), 2021.

[141] Y. Zhang, N. Zhang, T. Zhao, M. Vilim, M. Shahbaz, and K. Olukotun, "SARA: scaling a reconfigurable dataflow accelerator," in *Proceedings of the 48th annual International Symposium on Computer Architecture (ISCA-48)*, 2021.

[142] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe, "GraphIt - A high-performance DSL for graph analytics," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.

[143] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*, 2020.

[144] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for CGRA modulo scheduling," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 9, 2020.

[145] J. Zhong and B. He, "Medusa: A parallel graph processing system on graphics processors," *SIGMOD Rec.*, vol. 43, no. 2, 2014.

Colophon

The text typeface is Bitstream Charter, designed by Matthew Carter. The sans-serif typeface is Source Sans Pro, and the fixed-width typeface is Inconsolata. This dissertation was compiled with pdfTEX 1.40.16 with LATEX 2_ε <2016/03/31> on a Macintosh. Figures were drawn in Microsoft PowerPoint for Mac, and plots were generated by Matplotlib. The author's signature on the original copy of this thesis was signed with a Pilot Metropolitan fountain pen.