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

Quan M. Nguyen  
MIT CSAIL  
qunn@csail.mit.edu

Daniel Sanchez  
MIT CSAIL  
sanchez@csail.mit.edu

Abstract—Irregular applications are increasingly common in diverse domains, like graph analytics and sparse linear algebra. Accelerating these applications is challenging because of their unpredictable data reuse and control flow. Recent work has proposed hardware support for fine-grain pipeline parallelism, hiding long latencies by decoupling irregular applications into pipeline stages. However, this prior work requires programmers to manually decouple applications. This tedious and error-prone process limits the usefulness of such architectural support.

We address this problem with Phloem, a compiler that automatically discovers and exploits pipeline parallelism in irregular applications. Prior compilers for pipeline parallelism target regular applications, which contain simple pipeline stages with known latencies and fixed buffering needs. Designing Phloem to target irregular applications, where these properties do not hold, requires treating their unique challenges as first-class considerations throughout its design. Phloem breaks down this complex transformation into a series of simple passes that together encode the insights that have been previously applied by hand, producing code that targets architectures with support for queue-based communication.

We evaluate Phloem by generating efficient pipelines on a variety of irregular applications. Phloem’s contributions improve performance by 1.7× on average, approaching (and sometimes exceeding) the performance of manually optimized pipeline-parallel code. These results show that, for the first time, automatic parallelization for irregular applications is not only feasible, but also profitable.

I. INTRODUCTION

Irregular applications are those with data-dependent memory accesses and control flow. Irregular applications are the norm in many domains, like graph analytics and sparse linear/tensor algebra. Their data-dependent accesses and control are often unpredictable, causing poor performance on CPUs and GPUs. For concreteness, 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 cycles per call (e.g., about 10), and that it does not modify arrays A[] or B[]. This code runs 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 IPC. Moreover, the indirect access B[A[i]] causes 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 previous code snippet:

```
```

Each stage runs in parallel, e.g., in a separate 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 mispredictions no longer fill 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 dequeuing a value every 5–10 instructions. Thus, software-only queues (which take hundreds of cycles per operation [16, 44]) would add very high overheads. To enable fine-grained pipelining, much prior work has proposed adding hardware queues across cores or threads [9, 12, 17, 19, 34, 35, 38, 43, 47, 48, 52, 55, 60]. But most of these systems only work well when every pipeline stage 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 would limit throughput. Recent work addresses load imbalance by dynamically time-multiplexing stages over the same processing element, like the threads of a multithreaded core [34] or the contexts of a reconfigurable fabric [35, 38, 58].

While the above techniques provide hardware support to pipeline irregular applications, there is currently no automatic way to generate efficient pipelines for irregular applications. Existing compilers only produce regular pipelines [11, 18, 25, 39, 53, 54], 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.

Specifically, pipelining an irregular application involves three challenges. First, it requires decoupling straight-line code into...
pipeline stages, e.g., producing our example pipeline from the code snippet. Second, and more importantly, it requires selecting the right pipeline, which depends on the application and architecture. For example, 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.

We present Phloem, 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 compiles serial C/C++ code, unlike prior work requiring programmers to rewrite their applications in a new language. Phloem is a standalone compiler, but can also be combined with existing domain-specific compilers to produce efficient pipeline-parallel applications from high-level code. We demonstrate this by combining Phloem with Taco [23] to automatically pipeline sparse linear algebra kernels.

Phloem is the first technique that makes irregular applications efficient in out-of-order cores through hardware-compiler co-design. Much prior work has explored some of the techniques used by Phloem, such as decoupling, time-multiplexing, and prefetching (Sec. II-C). But prior techniques targeted regular applications or were hampered by software overheads or a limited execution model. Phloem’s key novelty is in showing the right combination of hardware and compiler techniques that results in efficient acceleration.

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 in a way that maximizes 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.

\[ A[i] \]

\[ A[j] \]

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

void bfs(Graph* g, int* cur_fringe, int* next_fringe, int root, int* distances) {
    int cur_fringe_idx = 0, next_fringe_idx = 0;

    // Process current fringe
    while (cur_fringe_idx != 0) {
        int cur_dist = distances[cur_fringe_idx];
        // Process current fringe for
        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,
                int old_dist = distances[ngh];
                if (distances[ngh] = cur_dist;
                    next_fringe[next_fringe_idx++] = ngh;
                }
            }
        }

        // add to next fringe
        for (int i = edges[cur_fringe_idx];
            nghList[cur_fringe[i]] = ngh;
        }
    }

    swap(&cur_fringe, &next_fringe);
    cur_fringe_idx = next_fringe_idx;
    next_fringe_idx = 0;
}

Fig. 1: Decoupling BFS into a 4-stage pipeline by partitioning across sources of irregularity.

II. BACKGROUND

To see the challenges of irregular applications in more detail, consider breadth-first search (BFS), a common graph algorithm. Given a root vertex, BFS finds the distance of all vertices reachable from that root. Fig. 1 (left) shows sequential C code for BFS. This version of BFS traverses a graph stored in the commonly used Compressed Sparse Row (CSR) format [33, 46, 50]. The root vertex starts at distance zero, while all other distances are set to INT_MAX. For each vertex in the fringe, BFS accesses g->nodes to find the start and end of the vertex’s edge list. Then, neighbor vertex ids are found by accessing g->edges. Finally, if each neighbor’s current distance is less than its currently recorded distance, the distance is set to cur_dist, and the neighbor is added to the next fringe. Once the current fringe is processed, BFS switches to the next fringe, and repeats this process until no new vertex visits are recorded.

A. Challenges of irregularity

We examine two sources of irregularity in BFS that make it difficult to accelerate on modern architectures.

First, BFS has unpredictable reuse: finding the distance of a neighbor of a vertex requires four memory accesses that are dependent on each other. While the access to cur_fringe is a linear traversal, the other three accesses are extremely difficult to predict: the addresses of accesses to distances depend on values in g->edges, which in turn depend on values in g->nodes. While caches and scratchpads can capture short streaming patterns or small amounts of metadata, irregular applications often contain tricky multi-level indirections through large datasets that do not fit on-chip. In conventional out-of-order cores, the large reorder buffer (ROB) tries to keep functional units highly utilized. However, the multi-level indirections of an irregular application have hard-to-predict memory addresses, resulting in costly consecutive cache misses. To make things worse, these indirections are often followed
void bfs(Graph* g, int* cur_fringe, int* next_fringe, int root, int* distances) {
    int cur_fringe_idx = 0, next_fringe_idx = 0;
    // Add root to fringe
    cur_fringe[cur_fringe_idx++] = root;
    distances[root] = 0;
    while (cur_fringe_idx != 0) {
        int cur_dist = 0;
        // Process current fringe
        for (int i = 0; i < cur_fringe_idx; i++) {
            int v = cur_fringe[i];
            // Visit neighbors
            int edge_start = g->nodes[v];
            int edge_end = g->nodes[v+1];
            for (int e = edge_start; e < edge_end; e++) {
                int ngh = g->edges[e];
                if (cur_dist < old_dist) {
                    distances[ngh] = cur_dist;
                    next_fringe[next_fringe_idx++] = ngh;
                }
            }
            swap(&cur_fringe, &next_fringe);
            cur_fringe_idx = 0,
            // Add ngh to next fringe
            old_dist = distances[ngh];
            if (cur_dist < old_dist) {
                distances[ngh] = cur_dist;
                next_fringe[next_fringe_idx++] = ngh;
            }
        }
        cur_fringe_idx = 0;
        // Enqueue fringe
        q1_handle_ctrl(1);
    }
}

In the case of an irregular application, decoupling across sources of irregularity yields simple stages that can run at high throughput. In BFS, for example, these sources of irregularity are its multi-level indecisions; Fig. 1 shows a possible decoupling of BFS into a multi-stage pipeline.

Although pipeline parallelism can be exploited at many granularities, several factors make fine-grain pipeline parallelism especially practical for irregular applications. First, the amount of data communicated between stages is small (often just a single 32- or 64-bit word), but communication occurs very frequently (in BFS, one in every six operations uses a queue). Second, the amount of work between each stage is often small, such as a simple address computation. Finally, thanks to their leanness, fine-grain stages built from irregular applications have simple communication patterns of few elements from stage to stage. A successful data-parallel implementation, on the other hand, must ensure that all units perform the same computation across all elements at the same rate.

C. Compiler support for pipeline parallelism

Much prior work has proposed compiler techniques that exploit pipeline parallelism and decoupled execution. While Phloem’s passes also leverage these techniques, a combination of two key features set Phloem apart from prior work.

First, Phloem targets irregular applications. Without support for irregularity as the primary consideration, prior works wind up unable to effectively cope with irregularity. Much of this prior work, including StreamIt [18, 53, 54], Piper [26], SGMS [25], Team Scheduling [39], and some polyhedral approaches [37] targets regular programs, where the amount of work 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 amount of work and communication.

Moreover, decoupling irregular applications thus far has only been carried out by hand, a time-consuming process. While the simple serial description of BFS (Fig. 2, left) is about thirty lines of code, turning this into efficient pipeline-parallel code (right) not only doubles the size of the code, but also requires several complex transformations. With Phloem, we break down these transformations into a series of simple passes over the original source code.

Second, Phloem presents an effective end-to-end hardware-software codesign. Software-only techniques do not suffice. For example, Clairvoyance [56] is a compiler pass that reorganizes loops into access and execute phases, but achieves little performance gain: at best a 13% gmean speedup. Software prefetching (e.g., Ainsworth and Jones [2]) results in significant increases in dynamic instruction count for similar irregular performance gains. It is important to have the right architectural support: Phloem achieves much better results—1.7× gmean speedup—accelerating irregular applications.

Thus, hardware-software codesign is needed, but achieving good performance also depends on having the right interface between hardware and software. Prodigy [51], a programmable prefetcher, chooses a programming model that thrusts the burden of correctly guessing irregular memory accesses to hardware, and incorrect predictions risk polluting caches. A general programming model that supports decoupled communication allows us to explicitly issue loads for precisely the values that are needed.

Phloem is the first end-to-end solution that fully integrates all of the insights needed to create effective pipelines from irregular applications for hardware that is designed to execute them efficiently. Hardware-software codesign is paramount, because it not only affects the complexity of the software passes needed, but also dictates the potential for acceleration. Phloem shows it is possible to transform irregular applications into pipelines using a series of simple passes that are easy to reason about. Furthermore, Phloem achieves speedups that are simply not attainable with software-only approaches. Thus, we are the first to show that simple techniques, on a simple hardware model, are the right ingredients for significant speedups.

We discuss other related work, including domain-specific frameworks, other hardware-software codesign approaches, dataflow compilers, and HLS techniques in Sec. VIII.

III. Baseline Architecture

Phloem leverages hardware support for irregular application pipelines. As our baseline architecture, we use Pipette [34], a representative architecture with support for efficiently executing irregular applications cast as pipeline-parallel programs. Other architectures provide hardware support for irregular fine-grained pipelines, but they all lack a compiler; Sec. IV-E discusses these architectures and how Phloem could compile for them.

In Pipette, programmers specify individual stages in a multi-threaded code, which is then executed on the threads of an out-of-order (OOO) core with simultaneous multithreading (SMT). Each thread corresponds to a stage of the irregular application’s pipeline, and stages communicate using Pipette’s ISA support for architecturally visible queues. Inter-thread queue communication is cheap, thanks to Pipette’s reuse of the OOO core’s physical register file. Fig. 3 shows Pipette’s modifications and Table I summarizes its ISA.

Queue interface: Pipette extends the core ISA to support architecturally visible queues. An enq(q, v) operation enqueues value v into queue q. (In this paper, a queue is identified by a number, so enq(1,37) enqueues value 37 into queue 1.) Similarly, a deq(q) operation dequeues a value from queue q. Queues are first-in first-out (FIFO) and have limited size. When a thread tries to enqueue to a full queue, or dequeue from an empty queue, the thread blocks, letting the OOO core’s SMT scheduler issue work from other non-blocked threads.

Offloading memory accesses: Irregular applications often contain memory accesses that are easy to offload to a specialized engine. Pipette adds simple hardware engines to offload these accesses, called reference accelerators (RAs). They interpose on the queue-based interface and can launch memory requests in parallel but deliver loads in order.

Reference accelerators are simple runtime-configurable finite state machines (FSMs) that are configured with the base address of the array they are accessing, the size of the element, and the mode of access. In INDIRECT mode, the RA treats each enqueued value as an index into the array, fetching the element at that index and placing it into the input queue of the next stage. In SCAN mode, pairs of enqueued values are treated as start and stop indices of a linear access through the array.

In the BFS example, edge_start comes from loading g->nodes[v]. Now, instead of directly performing an indirec-

Fig. 3: Baseline multicore system with Pipette’s changes to the OOO cores shaded in purple and hatched.
tion, the process current fringe stage only needs to enqueue \( v \) to a reference accelerator configured to indirect on \( g \rightarrow \text{nodes} \). The enumerate neighbors stage simply dequeues the value of \( \text{edge}_{\text{start}} \) as an output of the RA.

Chained reference accelerators allow Phloem to exploit the fact that some stages simply dequeues 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 inductions. BFS contains an opportunity for chained RAs. Extending the example above, the RA performs inductions on \( g \rightarrow \text{nodes} \) to produce \( \text{edge}_{\text{start}} \) and \( \text{edge}_{\text{end}} \). These two values form the starting and ending indices for \( g \rightarrow \text{edges} \), so we can chain this to a second scanning RA to read neighbors (\( \text{ngh} \)) out of \( g \rightarrow \text{edges} \). Chained RAs free us to devote general-purpose threads and core resources to application compute, rather than manipulating queues.

Making control flow efficient: Despite the judicious use of queues, each stage’s loop could still be tightened for better performance. Computing loop bounds becomes relatively expensive as the body—the actual useful work—becomes smaller as it is decoupled from other stages. But, the loop condition can often be inferred, or sent by the producer. Pipette adds hardware support for control values, which are passed through queues just like data, but they cannot be interpreted as data. The special \( \text{enq}_{\text{ctrl}}(q, cv) \) instruction enqueues control values, which appear in-band with data values.

Now, a consumer stage using control values no longer needs to determine the trip count of the enclosing loop—in our implementation, any loop that uses a control value becomes a \( \text{while}(\text{true})\{\ldots\} \) statement. The consumer simply needs to examine whether the most recently dequeued value is a control value; if so, it acts on the value as needed by the program, such as breaking out of a loop.

Checking for control values still has overhead but control values are infrequent. Pipette adds hardware support for control value handlers, which eliminate repeated checking for control values. The core jumps to the control value handler whenever it is about to dequeue a control value, letting it process the control value externally, rather than within the inner loop.

IV. 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.

A. Phloem interface

Phloem transforms serial code, starting with a program written in C. Other interfaces are also possible, e.g., we later combine Phloem with the Taco domain-specific compiler. We do not find a C-based interface limiting: while C provides serial semantics, it has enough semantic information to divide execution into pipeline stages. Phloem’s key challenge is not to find pipeline parallelism, but to generate efficient multi-stage pipelines for irregular applications. Specifically, the bulk of the techniques we present are code analyses and transformations on already-pipelined code. These techniques do not depend on the semantics of our frontend language (serial C in our case), and would apply equally to any other frontend language.

Phloem is automatic, but programmers can control some aspects through the pragma annotations shown in Table II.

**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. Fig. 4 shows such a race in BFS: if we pipelined the lookup of neighbors, a given neighbor (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 these races 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. However, Phloem may still prefetch data in this case. In Fig. 4’s 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.

![Fig. 4: A race condition in BFS that would arise with an incorrect decoupling into pipeline-parallel stages.](image_url)
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 on to the next one.

B. Producing efficient fine-grain pipelines

Phloem produces efficient pipeline-parallel programs by systematically applying six passes. We introduce these passes with a detailed example using BFS, showing how they complement each other and progressively improve performance. Fig. 5 illustrates these passes.

Fig. 5: 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.

Fig. 6: Speedup over the original serial BFS implementation with each added pass.

of irregularity. Because each loop level usually contains a long-latency load, decoupling an irregular application often results in placing each loop level in its own stage. Choosing decoupling points is critical for performance; Sec. V 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 and edge_end. This stage now needs the value of the current vertex v.

1. Add queues: A functionally correct pipeline can be achieved by passing 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, excess queue communication has overhead, resulting in poor performance.

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. 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 & 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. In BFS, all vertices visited in one iteration are compared to the same distance. It is unnecessary to know which vertex a particular neighbor 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.

Fig. 6 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 Fig. 6) build successively on each other, culminating in a 1.85× speedup. Note that eliminating unnecessary uses of control values and checks for them is critical. Adding control values in isolation (CV, R, Q) actually diminishes performance compared to not having them at all, because of the overheads resulting from instructions needed to check for control values (is_control() in Pass 4 of Fig. 5). Finally, reference accelerators (RA) greatly increase performance, but they depend on the other optimizations: RAs truly shine when stages are fast enough to keep them busy.

With a 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.

C. Composing data and pipeline parallelism

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 (as we saw in Sec. II) can be replicated over many cores so that each pipeline works on a specific part of the input graph, as shown in Fig. 7. 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.

Phloem lets programmers create data-parallel pipelines by marking the function with #pragma replicate and specifying the number of replicas. 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.

D. Making efficient domain-specific pipelines

C/C++ remains the lingua franca of domain-specific accelerator compilers [3, 5, 23, 42, 62] and frameworks [4, 46]: they either use C directly or emit C code. Thus, Phloem’s C-based frontend 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, a process that would take considerable time to do manually. Furthermore, compilers emit code that already meet Phloem’s input requirements; for example, their data structures are already 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) [23]. 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 $y(j) = A(i, j) \times x(i)$, and emits C code. Phloem uses this code to produce an efficient pipeline-parallel implementation.

E. Targeting other architectures

Although we evaluate Phloem on the Pipette architecture, several recent accelerator architectures, including Fifer [35], Aurochs [58], and SPU [10], also provide hardware support for fine-grain irregular pipelines. These systems are programmed manually and lack compiler support, highlighting the need for automation. Phloem could target these systems, as their ingredients are similar to Pipette.

V. 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 IR that represents fine-grain operations (e.g., load, add). This IR allows any two operations in a program to be decoupled. It is not necessary to decouple every operation; decoupling at just a few points (3 or 4) is enough for good performance. Unlike conventional IRs like LLVM’s, Phloem’s IR adds support for queue operations and controls flow conversion completely.

**Determining decoupling points statically:** Phloem’s static analysis mode finds decoupling points by ranking expensive operations (e.g., memory accesses) with a simple cost model. Then, it 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 by its (1) predicted cost and (2) frequency. The cost of the memory access depends on whether it is indirect or sequential and the presence of nearby accesses. For example, BFS has two nearby accesses to $g->\text{nodes}$. 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 $g->\text{edges}$ is considered more even more costly than to $g->\text{nodes}$, and would be prioritized for decoupling.

This simple static analysis 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. Building stages from the innermost loop outwards usually produces pipelines that decouple 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 (Sec. IV-C), it is similarly possible to generate non-replicated pipelines spanning multiple cores.

**Autotuning decoupling points:** The static approach produces reasonable pipelines, but its cost model is by necessity approximate: in irregular applications, cache misses to each data structure and 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 from combinations of these points. These pipelines are then profiled on small training inputs to find the best one. Fig. 8 shows this process, with the pipeline search and profile-guided optimization shaded in gray. This process, which completes in seconds, allows exploiting decoupling points that are statically ranked below the bar, but happen to be more profitable.

Fig. 8 (upper right) also shows the static compilation flow, in which only one pipeline is generated and no training occurs. This static compilation process also completes within seconds, and its pipelines work well in practice. Our evaluation compares the performance of both modes.

VI. 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.

A. Evaluated systems

We evaluate Phloem’s generated benchmarks on an extended version of Pipette (Sec. III). We use Pipette’s evaluation configuration, whose cores are modeled after Intel’s Skylake [14] microarchitecture and scaled to four SMT threads. Table III
lists the parameters of our evaluation configuration. We use an event-driven, cycle-level simulator based on Pin [30]. To be consistent with Pipette’s energy models, we also model core and uncore energy at 22 nm with McPAT [28] and main memory with Micron DDR3L datasheets [31].

### B. Benchmarks

Our initial evaluation uses five diverse C benchmarks: four from graph analytics and one from sparse linear algebra. We use Phloem to automatically generate pipelines for each application from the serial code. We compare the Phloem-generated version to its original serial implementation, a competitive data-parallel implementation, as well as a manually pipelined version.

**Breadth-First Search (BFS)** is the graph algorithm introduced in Sec. II. It discovers the distance of all vertices reachable from a root vertex. The data-parallel implementation is based on work-efficient PBFS [27].

**Connected Components (CC)** assigns labels to connected components of a graph by running searches from each vertex in the graph until all vertices are assigned a label. *PageRank-Delta (PRD)* is like the PageRank algorithm in that it determines the importance of vertices by distributing weights, except that the change in weight must exceed a threshold for it to be applied. *Radii* estimates the radius of a graph by performing multiple searches from randomly sampled vertices. CC, PRD, and Radii are derived from their data-parallel implementations in the Ligra framework [46].

Finally, **Sparse Matrix-Matrix Multiplication (SpMM)** multiplies two compressed matrices. It uses an *inner-product*, or *output-stationary*, dataflow, meaning that each element of the output matrix is computed one element after another from a dot product of an input row and column. The coordinates of non-zero values are sorted, so identifying non-zero partial products of the dot product requires a *merge-intersection* that jointly iterates through these two vectors.

**Taco benchmarks:** We integrate Phloem with the Tensor Algebra Compiler (Taco) [23] to automatically compile tensor algebra expressions into 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 \( A = B \odot CD \), where \( C \) and \( D \) are dense matrices, \( A \) and \( B \) are sparse matrices, and the \( \odot \) operator represents component-wise multiplication. *Matrix-Transpose Multiplication (MTMul)* evaluates \( y = \alpha A^\top x + \beta z \), where \( \alpha \) and \( \beta \) are constants; \( x \), \( y \), and \( z \) are vectors; and \( A \) is a sparse matrix. Lastly, *Residual* evaluates \( y = b - Ax \), where \( b \), \( x \), and \( y \) are vectors and \( A \) is a sparse matrix.

**Inputs and sampling:** We execute the graph analytics workloads (BFS, CC, PRD, and Radii) on graphs listed in Table IV, which arise from many real-world domains. SpMM is evaluated using the matrices listed in Table V. We evaluated the Taco benchmarks using the same matrices as its original evaluation; Table V also lists these matrices.

In PRD and Radii, simulations of the largest graphs take tens of billions of cycles. To keep simulation times reasonable, runs of PRD and Radii on the largest graphs use *iteration sampling*, simulating a subset of iterations. Even with sampling, these applications exceed billions of simulated cycles.

### C. Automatic pipeline generation and search

When evaluating Phloem’s profile-guided compilation flow (Sec. V), we 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, email1-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.

<table>
<thead>
<tr>
<th>TABLE III</th>
<th>Configuration parameters of the evaluated system.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores</strong></td>
<td>1 or 4 cores, 3.5 GHz, x86-64 ISA, Skylake-like: 6-wide out-of-order issue; 4-thread SMT</td>
</tr>
<tr>
<td><strong>Pipette</strong></td>
<td>16 queues max; 4 RAs; queues up to 24 elements deep</td>
</tr>
<tr>
<td><strong>L1 cache</strong></td>
<td>32 KB/core, 8-way set-associative, 4 cycle latency</td>
</tr>
<tr>
<td><strong>L2 cache</strong></td>
<td>256 KB/core, 8-way set-associative, 12 cycle latency</td>
</tr>
<tr>
<td><strong>L3 cache</strong></td>
<td>2 MB/core, 16-way set-associative, 40 cycle latency</td>
</tr>
<tr>
<td><strong>Main mem</strong></td>
<td>120-cycle minimum latency, 2 controllers, 25 GB/s each</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE IV</th>
<th>Input graphs, sorted by the number of edges.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td><strong>Graph</strong></td>
</tr>
<tr>
<td>Training internet graph</td>
<td>internet</td>
</tr>
<tr>
<td>Training road network</td>
<td>USA-road-d-NY</td>
</tr>
</tbody>
</table>

| **Test inputs** | |
| **Human collaboration** | coauthors-DELTA-symmetric | 299K | 1.9M | 6.4 |
| **Dynamic simulation** | hugerace-00000 | 4.6M | 14M | 3.0 |
| **Circuit simulation** | Freescale1 | 3.4M | 19M | 5.6 |
| **Internet graph** | as-skitter | 1.7M | 22M | 1.2 |
| **Road network** | USA-road-d-USA | 24M | 58M | 2.4 |

<table>
<thead>
<tr>
<th>TABLE V</th>
<th>Input matrices, sorted by average non-zeros per row. SpMM also uses pwtk.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td><strong>Matrix</strong></td>
</tr>
<tr>
<td>Training graph as matrix 1</td>
<td>email1-Enron</td>
</tr>
<tr>
<td>Training graph as matrix 2</td>
<td>wiki-Vote</td>
</tr>
</tbody>
</table>

| **SpMM test inputs** | |
| **File sharing** | p2p-Gnutella31 | 62,586 | 2.4 |
| **Graph as matrix** | amazon0312 | 400,727 | 8.0 |
| **Gel electrophoresis** | cage12 | 130,228 | 15.6 |
| **Electromagnetics** | 2cubes_sphere | 101,492 | 16.2 |
| **Fluid dynamics** | nrel8 | 46,835 | 49.7 |

| **Taco (MTMul, Residual, SpMV, SDDMM) test inputs** | |
| **Circuit simulation** | scircuit | 170,998 | 5.6 |
| **Economics** | unc_econ_fsd500 | 206,500 | 6.2 |
| **Particle physics** | cog250A | 121,192 | 21.7 |
| **Structural** | pwtk | 217,918 | 52.9 |
| **Cantilever** | cant | 62,451 | 64.2 |
VII. Evaluation

Fig. 9 reports the overall speedups of Phloem compared to the serial code, a data-parallel implementation, and a manually optimized Pipette version. Each group of bars shows results for one application; each bar is the speedup of each variant over serial, averaged (gmean) across inputs. For Phloem, the height of the bar is the speedup of the pipeline produced by profile-guided optimization, and the × mark is 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 those of the hand-tuned versions. In almost all cases, 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.

Fig. 10 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 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.

Fig. 11 shows energy breakdowns across benchmarks, averaged across inputs, relative to the serial baseline. In all benchmarks, Phloem achieves better energy efficiency than the serial and data-parallel versions, chiefly due to better core utilization. For CC, PRD, and Radii, Phloem’s energy use is comparable to the manually pipelined version. In BFS, energy usage decreases thanks to Phloem’s speedups, with its overall reduced time spent running the cores. In SpMM, while energy usage is overall lower, these gains are somewhat offset by the increase in time spent on stalls.

**Taco results:** Fig. 12 reports Phloem’s speedups when parallelizing Taco programs. (This is similar to Fig. 9, but without 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, turning to data parallelism barely improves performance because of the increase in instructions.

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.

A. Analysis of generated pipelines

We now evaluate the impact of profile-guided optimization by examining the pipelines generated by the search process.
Fig. 13 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 constructing pipelines, as adding too many stages can cause excessive communication that limits the achievable 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.

B. Replicating pipelines

We evaluate how Phloem effectively uses multicore resources by producing replicated pipelines. In addition to the replicated BFS presented in Sec. IV-C, 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 have 4 cores with 4 threads each, scaling the data-parallel versions and replicating the Phloem and manual pipelines to use all 16 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.

Fig. 14 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 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.

VIII. ADDITIONAL RELATED WORK

In this section, we discuss prior work not covered so far. Software-only decoupling approaches: Asynchronous Memory Access Chaining (AMAC) [24] breaks variable-length, pointer-chasing chains of memory accesses using an FSM. It cycles through multiple independent chains, issuing them access by access to improve memory-level parallelism. CIMPLE [22] and Jonathan et al. [20] leverage coroutines that yield on long-latency events to achieve similar decoupling with simpler code transformations than AMAC. While these approaches improve memory-level parallelism, they add substantial overheads, as scheduling stages and buffering of intermediate values is done in software. Consequently, while these approaches show benefits on extremely latency-dominated data structures, these overheads limit their applicability [22, Table 2]. Moreover, this prior work requires complex code changes, unlike Phloem.

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

Ligra [46] is a framework for accelerating graph traversals, Halide [42] is a domain-specific language for image processing pipelines, and T2S-Tensor [49] extends Halide to accelerate tensor kernels. However, these are all highly tuned to their specific domain. For example, irregular applications’ arbitrarily
While SpecDSWP supports more than one level of indirection, with a profile-guided optimization mode. SARA [61] exploits coarse-grain data parallelism in HLS: compilers identify parallel patterns for FPGA implementation, Compilers for spatial architectures and high-level synthesis rely on the application already being structured as a pipeline. Analytical models for pipeline parallelism [32], as such models opportunities. This flexibility lends an advantage over prior irregular application to stages and also offers more optimization enables a more comprehensive search for the best mapping of an loop levels and thus offers more flexibility in decoupling. This space of possible pipelines—something that Phloem explores decoupling at more than one point significantly increases the at every source of irregularity to run efficiently. Moreover, decoupling at more than one point significantly increases the space of possible pipelines—something that Phloem explores with a profile-guided optimization mode.

Unlike DSWP, HELIX, and CFD, Phloem parallels across loop levels and thus offers more flexibility in decoupling. This enables a more comprehensive search for the best mapping of an irregular application to stages and also offers more optimization opportunities. This flexibility lends an advantage over prior analytical models for pipeline parallelism [32], as such models rely on the application already being structured as a pipeline. Compilers for spatial architectures and high-level synthesis (HLS): SARA [61] exploits coarse-grain data parallelism in mapping applications to the Plasticine architecture [41]. Some compilers identify parallel patterns for FPGA implementation, including pipelines [29,40]. LegUp [8] is an HLS system that supports streaming semantics. Calyx [36] is an intermediate representation intended for hardware implementation. Aetherling [15] produces statically scheduled streaming hardware circuits. These prior systems all fall short for several reasons: mapping to spatial architectures works best when applications are compute-heavy regular applications with little control. They also assume that the entire pipeline can be mapped to the hardware at once, which results in different tradeoffs. Dynamic time-multiplexing of pipeline stages, as the Pipette-style baseline system does, offers considerably more flexibility in structuring stages. Finally, some systems lack support for key elements: Calyx explicitly mentions the need for compiler support for (explicit) pipeline parallelism, and Aetherling does not support variable-latency operators, a hallmark of irregular applications.

IX. Conclusion

Irregular applications use conventional architectures poorly. Software-only techniques are insufficient and many previous hardware-software codesign approaches have programming models that are ill-suited to accelerating irregular applications. An emerging class of architectures exploits fine-grain pipeline parallelism with a simple but powerful programming model, but applications needed to be written by hand. Phloem is the first system to show that a series of simple passes can systematize the insights needed to transform serial code into high-performance pipeline-parallel code. Therefore, Phloem is a comprehensive solution for automatic high performance on irregular applications.

Acknowledgments

We thank Joel Emer, Fares Elsabbagh, Axel Feldmann, Hyn Ryong (Ryan) Lee, Nikola Samardzic, Shabnam Sheikha, Yifan Yang, Victor Ying, Robert Durfee, Nithya Attaluri, Kendall Garner, Aleksandar Krastev, and the anonymous reviewers for their feedback. This work was supported in part by SRC under contract 2020-AH2985, by DARPA under contract N00014-21-1-2960, and by NSF under grant CCF-2217099. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References


