Scheduling Irregular Dataflow Pipelines on SIMD Architectures

Tom Plano
planot@wustl.edu
Washington University in St. Louis
St. Louis, Missouri

Jeremy Buhler
jbuhler@wustl.edu
Washington University in St. Louis
St. Louis, Missouri

Abstract

Streaming computations often exhibit substantial data parallelism that makes them well-suited to SIMD architectures. However, many such computations also exhibit *irregularity*, in the form of data-dependent, dynamic data rates, that makes efficient SIMD execution challenging. One aspect of this challenge is the need to schedule execution of a computation realized as a pipeline of stages connected by finite queues. A scheduler must both ensure high SIMD occupancy by gathering queued items into vectors and minimize costs associated with switching execution between stages.

In this work, we present the **AFIE** (Active Full, Inactive Empty) scheduling policy for irregular streaming applications on SIMD processors. AFIE provably groups inputs to each stage of a pipeline into a minimal number of SIMD vectors while incurring a bounded number of switches relative to the best possible policy. These results apply even though irregularity forbids *a priori* knowledge of how many outputs will be generated from each input to each stage.

We have implemented AFIE as an extension to the MERCATOR system [6] for building irregular streaming applications on NVIDIA GPUs. We describe how the AFIE scheduler simplifies MERCATOR's runtime code and empirically measure the new scheduler's improved performance on irregular streaming applications.

Keywords SIMD,irregular applications,streaming,scheduling

ACM Reference Format:
Tom Plano and Jeremy Buhler. 2020. Scheduling Irregular Dataflow Pipelines on SIMD Architectures. In *Workshop on Programming Models for SIMD/Vector Processing (WPMVP'20), February 22, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3380479.3380480

WPMVP'20, February 22, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7520-7/20/02…$15.00
https://doi.org/10.1145/3380479.3380480

1 Introduction

Many high-impact computations can be parallelized using a streaming dataflow model. In this model, data items move through a pipeline (more generally, a tree or DAG) of compute nodes, each of which performs some transformation on each item. Examples of computations that fit this model include biological sequence comparison [2], network packet filtering [13], and decision cascades in machine learning [17]. When the individual items in a stream can be processed independently, streaming computations naturally map onto SIMD architectures such as GPUs, which can execute the computation of a node on many items from a stream in data-parallel fashion.

While streaming computations often treat each item independently, they may not process items identically. In particular, each item consumed by a node may require a different amount of work and may produce a variable, data-dependent number of outputs. For example, a node may filter its input stream, allowing only a subset of items through. We refer to applications with such dynamic data rates (including, e.g., the three high-impact tasks described above) as having *irregular data flow*.

Mapping an irregular streaming computation onto a SIMD processor is challenging for two reasons. First, a static assignment of data to SIMD lanes causes some lanes to become idle earlier than others, reducing SIMD *occupancy* (fraction of active SIMD lanes) and hence the efficiency with which the device is used. Runtime support for irregular streaming must therefore remap data dynamically between SIMD lanes to maintain high occupancy. The MERCATOR system for GPUs [6] is an example of a framework that performs such remapping transparently to the application developer. Second, the unpredictable rates at which output is generated from each node pose a challenge for scheduling nodes' execution. The present work addresses this scheduling problem.

Unlike traditional streaming models such as Synchronous Data Flow [10], whose fixed data rates admit throughput-optimal static schedules, irregular streaming requires a dynamic, data-dependent scheduling algorithm. To maximize application throughput, a scheduler must execute nodes in an order that both ensures high SIMD occupancy (i.e., nodes should run only when they have a full SIMD-width's worth of input) and minimizes overhead, both from the scheduling decisions themselves and from costs such as cache misses

when switching from one node's computation to another's. A throughput-optimal schedule may not be practically achievable because the behavior of each data item at each node is discovered only at runtime. However, we can still design a scheduling policy whose decisions are, under certain assumptions, close to optimal.

In this paper, we present a scheduling policy, *AFIE* (Active Full, Inactive Empty), for irregular streaming dataflow computations on a wide-SIMD processor. The AFIE policy provably optimizes SIMD occupancy while limiting the number of switches between nodes. In terms of number of switches, AFIE with only a trivial amount of resource augmentation is 2-competitive vs a clairvoyant scheduler that knows in advance how many outputs will be produced by each node for each of its inputs. We implement AFIE as a replacement for MERCATOR's existing scheduler, in the process realizing additional benefits of the policy for code simplification and overhead reduction in MERCATOR's runtime. Finally, we investigate the behavior of our scheduler on two types of irregular streaming application.

The remainder of this work is organized as follows. Section 2 describes the application and architectural assumptions underlying our work. Section 3 introduces the AFIE scheduling policy and verifies that it is both safe and efficient. Section 4 describes our implementation of AFIE in the MERCATOR system and the resulting benefits for simplification of its runtime. In Section 5, we validate the performance of AFIE. Finally, Section 6 discusses related work, while Section 7 concludes and identifies future work.

2 Background

In this section, we describe the application model that we study and how that model is realized on a particular SIMD platform, namely an NVIDIA GPU. The constraints of model and platform determine the form of our scheduling problem.

Our application model focuses on pipelines for simplicity, but our work extends straightforwardly to tree-structured topologies. In contrast, DAG-structured irregular applications lack consistent, well-defined semantics for how to match up inputs on multiple input edges to a node. For one approach to this problem, see [11].

2.1 Application Model

An application is a linear pipeline of $m+1$ nodes, $n_0 \ldots n_m$, that process data items. For each input item that node n_i consumes, it emits a variable, data-dependent number of items between 0 and some maximum g_i. A node n_i can process a SIMD-parallel vector of up to v_i inputs at a time, where v_i the *vector width* of n_i.

Nodes are connected by dataflow edges, each of which has a finite queue. These queues are important for achieving high SIMD occupancy in an irregular streaming application because they allow outputs from one node to accumulate over time until a full vector of inputs is available for processing at the next node. Node n_i, $1 \le i \le m$, has an input queue q_i. Node n_i can *fire*, consuming and processing $k \le v_i$ inputs from its input queue, only if there are at least kg_i slots available in the downstream queue q_{i+1}, i.e. q_{i+1} has sufficient free space to hold the maximum number of outputs that n_i could produce from k inputs. The last node n_m in the pipeline has an effectively infinite output queue and so can always consume up to v_m inputs in one firing.

As discussed below, we focus on a pipeline executing sequentially on a single wide-SIMD processor. Only a single node fires at a time, and nodes cannot be preempted. Initially, all queues are empty except for that of the head node n_0, which contains the entire input stream. Nodes are then fired in some order, consuming some number of inputs each time, until no node has any pending input, at which point the computation ends. When execution begins firing a node n_i after previously firing some other node (or at the beginning of execution), it is said to *switch* to n_i.

The *schedule* of a pipeline's execution specifies both the sequence in which its nodes are fired and the number of inputs consumed by each firing. For a given stream of inputs, there may be many feasible schedules if execution results in multiple nodes being ready to fire at a given time. We seek a schedule that maximizes application throughput or, equivalently, minimizes the time for the pipeline to process all inputs. We assume that node n_i requires a fixed service time to process a vector of (any size up to) v_i inputs and produce any associated outputs. Switching from one node to another incurs some fixed time overhead.

2.2 Architectural Realization

We use NVIDIA GPUs as our target SIMD architecture. However, while NVIDIA's compiler and runtime support are a convenient base for our research, the scheduling policy and realization described in this work are not restricted to GPUs. Our approach for realizing streaming applications applies broadly to wide-SIMD multiprocessors.

A GPU contains a number of processors, each with some fixed SIMD width[1]. To minimize the cost of coordination between the GPU and its host system, an application pipeline runs entirely on the GPU, processing an input stream that is initially copied to the GPU's memory. Control does not return to the host until the input stream has been completely processed. Queues between nodes are stored in the GPU's DRAM memory; to avoid the need for dynamic allocation, they have fixed sizes, typically on the order of a few hundred to a few thousand elements.

Communication and synchronization between code running on different processors has nontrivial overhead and

[1]For this work, we treat the CUDA block size as our effective SIMD width, ignoring its realization as a collection of hardware threads, each with a smaller SIMD width (the "warp size").

complexity; on a GPU, it typically requires expensive interaction with the host to ensure correctness. We therefore instantiate the full application pipeline separately on each processor of the GPU. Pipelines on different processors compete to consume the shared input stream and process their portions of the stream independently of each other. As noted above, each processor's pipeline schedules its node firings sequentially and non-preemptively.

3 The AFIE Pipeline Scheduler

We now define a scheduling policy for a pipeline with the goal of achieving near-optimal throughput. We associate with each node a *status*, which is either "active" or "inactive." Informally, our policy ensures that active nodes have abundant inputs ready to consume, while inactive nodes have abundant free space in their input queues. This intuition explains the name of the policy, "Active Full Inactive Empty" (AFIE). Nodes that are both active *and* have inactive downstream neighbors therefore have both abundant inputs to process and plenty of space to write any resulting outputs. Under the AFIE policy, *only* such nodes are eligible to fire.

More formally, the AFIE policy for a pipeline $n_0 \dots n_m$ is as follows.

1. A node n_i may fire iff n_i is active and either
   - n_{i+1} is inactive, or
   - $i = m$ (i.e., n_i is the tail of the pipeline).
2. Once a node n_i begins to fire, it continues firing, consuming a full-width vector of v_i inputs from q_i each time, until either
   - q_i has fewer than v_i items remaining, or
   - q_{i+1} has fewer than $v_i g_i$ free slots remaining.

   We refer to these two circumstances as n_i being *blocked* on its input and output queues, respectively.
3. An inactive node n_i becomes active when its input queue q_i has fewer than $v_{i-1}g_{i-1}$ spaces remaining.
4. An active node n_i becomes inactive when its input queue q_i has fewer than v_i items remaining.

An *AFIE schedule* is one that complies with this policy.

At the start of execution, n_0, which has the entire input stream queued, is active, and all other nodes are inactive. We handle the end of execution, when the input stream is exhausted, as follows. If the end of the input stream reaches q_i while it is nonempty, n_i becomes active if q_i has any remaining items and remains so until q_i is empty. If q_i has fewer than v_i items after the end of stream reaches q_i, the last firing of n_i consumes them all.

3.1 Safety of AFIE

We first verify that the AFIE policy is *safe*, in the sense that it allows a pipeline to execute without the risk of deadlock. Safety of AFIE requires that the queue between nodes n_{i-1} and n_i be large enough to hold all outputs from one firing of n_{i-1} plus a nearly full vector of inputs to n_i.

Lemma 3.1. *Assume that, for $i > 0$, queue sizes satisfy*

$$|q_i| \geq v_{i-1}g_{i-1} + v_i - 1.$$

Under any AFIE schedule, so long as not all queues are empty, execution can always fire some node and so empties all queues in finite time.

Proof. Initially, n_0 is eligible to fire, and q_1, being empty, contains enough slots to receive the output from a full vector of inputs. After any number of firings, let n_i be the last active node in the pipeline. Such a node exists because, at a minimum, n_0 remains active until its input stream is exhausted, after which the earliest remaining node with a non-empty queue becomes active.

If $i = m$, q_m has either at least v_m inputs or the end of stream, and so n_m can fire. Otherwise, n_{i+1} must be inactive, and so q_{i+1} must hold $< v_{i+1}$ items. Conclude that q_{i+1} must have at least $v_i g_i$ free slots. Node n_i, being active, again has a queue q_i with either at least v_i inputs or the end of stream and so can fire. □

3.2 Efficiency of AFIE

We now analyze the efficiency of the AFIE scheduler. We can break the execution time of a pipeline into two components: time spent firing nodes, and time spent switching between them. We will analyze each of these two components of AFIE's performance separately. In what follows, let b_i be the total number of inputs consumed by node n_i during the entirety of an application's execution, which is fixed independent of the scheduling policy.

3.2.1 Time Spent Firing Nodes

We first lower-bound the number of times a node must fire to consume all its inputs under *any* schedule, then observe that AFIE meets this lower bound. AFIE therefore fires each node as few times as possible, which implies that it achieves the highest possible SIMD occupancy; otherwise, more firings would be needed to consume all of the node's inputs.

Claim 3.1. *The number of firings needed to consume all b_i inputs to n_i is at least $\left\lceil \frac{b_i}{v_i} \right\rceil$.*

Proof. If every firing of n_i uses the full vector width, except perhaps for one partial-width firing at the end of the input stream, the claimed bound is achieved. Any other firing sequence has at least as many non-full-width firings and so needs at least as many total firings as the bound. □

Lemma 3.2. *An AFIE schedule achieves the lower bound on total node firings given by Claim 3.1.*

Proof. Each firing of a node under AFIE consumes only full-width vectors, except perhaps its last firing. This policy is exactly the one given in the proof of Claim 3.1. □

3.2.2 Time Spent Switching Between Nodes

We now focus on the other component of execution time, the overhead due to switching from one node to another. In our simplified model, a switch between nodes has constant cost, so our concern is to minimize the *number* of switches between nodes during execution. For space reasons, the proofs for this section are relegated to an appendix.

We first derive a lower bound on the number of switches needed to execute a pipeline under any schedule. We quantify the minimum number of times S_i that execution must switch to a given node n_i.

Lemma 3.3. *For* $0 < i < m$,

$$S_i = \max\left(\left\lceil \frac{b_i}{|q_i|} \right\rceil, \left\lceil \frac{b_{i+1}}{|q_{i+1}|} \right\rceil\right).$$

Moreover,

$$S_0 = \left\lceil \frac{b_1}{|q_1|} \right\rceil \quad \textit{and} \quad S_m = \left\lceil \frac{b_m}{|q_m|} \right\rceil.$$

We now quantify how closely AFIE's schedules approach the lower bounds of the previous result.

Lemma 3.4. *If the best possible schedule for a given input stream uses* S^* *switches to nodes, an AFIE schedule for the same stream uses at most* $2S^* + 1$ *switches, provided that the capacity of each queue* q_i *is first augmented by* $v_{i-1}g_{i-1} + v_i - 2$ *slots.*

To summarize, we find that with modest additive resource augmentation, AFIE schedules are 2-competitive with respect to the number of switches and optimal with respect to the number of firings of each node. Hence, overall throughput is 2-competitive, no matter how large switching overhead is relative to node service times. If switching overhead is small, then throughput may be even closer to optimum.

4 Implementing the AFIE Scheduler

We implemented the AFIE scheduler as a modification to the MERCATOR system [6], an existing framework for irregular streaming computation on NVIDIA GPUs[2]. MERCATOR allows application developers to specify a pipeline structure, including the input and output types of each computational node and each node's maximum number of outputs per input. Given such a specification, MERCATOR produces a set of CUDA stubs that the developer fills in with the code to implement each node. This code is then combined with MERCATOR's runtime support to produce a complete application that executes the pipeline on the GPU.

4.1 Tracking Active Status and Fireability

To implement the AFIE policy, each node in a pipeline both maintains its active/inactive status and accesses the status of its downstream neighbor[3]. This information is sufficient to test the condition in point 1 of the policy, which determines when a node becomes eligible to fire. Fireable nodes are maintained in a FIFO queue, and the scheduler simply takes a node from this queue when it needs to find one to fire. The schedule remains AFIE no matter the order in which fireable nodes are queued and dequeued.

To maintain each node's active status, we must track changes to it according to AFIE policy points 3 and 4. Inactivation can occur only as a consequence of firing a node as described in policy point 2, so the check for inactivation is performed as part of the test needed to determine if a node can continue firing. Similarly, a node can become active only as a result of its upstream neighbor filling its input queue. Hence, after a node fires, we check whether its output has caused its downstream neighbor to activate.

Activation and inactivation checks have a small constant cost per firing. If a node has become active or its downstream neighbor inactive, we additionally check whether the condition of point 1 is now satisfied and, if so, enqueue the node for future firing. A node, once enqueued, is guaranteed to remain fireable until it is fired.

4.2 Impact of AFIE Scheduler on MERCATOR

The original scheduler described for MERCATOR in [6] used a different scheduling approach to achieve high occupancy. In MERCATOR, nodes implementing the same function are grouped into a common *module type*. The developer writes code for a module type only once; different nodes may, however, be given parameters, separate from the data stream, to modify their execution. Because they share code, multiple nodes of a given module type may be fired and may consume inputs concurrently; this is achieved by combining items from multiple nodes' queues into a single SIMD vector that is the input to the module's code.

MERCATOR's original scheduler always fires the module type whose nodes are collectively able to consume the most input items. By combining items across all nodes of the module type, the scheduler tries to ensure that the module's code receives full-width vectors of inputs, even if individual nodes have less than one vector-width of queued items. This scheduler is known to be free of deadlock but was not proved to have any particular efficiency properties. Like the AFIE scheduler, the original scheduler defers firing a module until a full SIMD width of inputs is available, though this rule is relaxed once the input stream has been exhausted.

Replacing the original scheduler with AFIE had immediate benefits for code simplification. Firstly, AFIE guarantees high occupancy without combining inputs across nodes. Consequently, we were able to remove the MERCATOR runtime's node-combining infrastructure, which is complex and incurs nontrivial overhead in gathering items from multiple input queues and scattering them to multiple output queues. (The developer-facing aspect of modules, which allows code to be written only once per module type, remains unchanged.)

[2] https://github.com/jdbuhler/mercator

[3] MERCATOR also supports applications with tree topologies, for which we maintain a count of each node's active downstream neighbors.

Secondly, the old scheduler did not have a simple, flag-based fireability criterion but rather had to calculate the number of inputs consumable by each node each time it selected a module to fire. This process took time proportional to the number of nodes in the application for each firing. In contrast, the AFIE scheduler spends only constant time per firing to select a node and to update its active flags and those of its neighbors. These changes, which reduced the MERCATOR runtime's code size by 20%, result in improved throughput for MERCATOR applications, as we verify in the next section.

We note further that the original scheduler reevaluated the best module to fire by its metric after each firing, even if it was possible to fire the same module again. This behavior is a consequence of the difficulty of efficiently determining whether sufficient inputs and queue space exist to fire again when combining data across many nodes at once. In contrast, the AFIE policy can easily check these properties for a single node and so will continue to fire a node without switching until either its input is exhausted or its output is full. AFIE therefore incurs fewer trips through the scheduler, even if both schedulers maintain high SIMD occupancy.

5 Experimental Validation

We investigated the impact of the AFIE scheduler on the performance of applications written for the MERCATOR framework. All experiments were performed on an NVIDIA GeForce GTX 1080Ti GPU using CUDA 10.1. Applications were run with the maximum possible number of active GPU blocks, with each block separately instantiating the pipeline.

For each application studied, we compared the performance of unmodified MERCATOR to our modified version with the AFIE scheduler and resultant simplifications described in Section 4.2. We compared total GPU running time as measured by CUDA's event-based timing facility as well as total trips through the runtime's scheduling loop. (Although the original scheduler, unlike AFIE, does not necessarily switch nodes after each scheduling trip, it does always incur substantial overhead per trip.) Results were averaged over 25 trials. Running times represent the cost to process the application's entire input stream in a single GPU kernel call. They do not include application setup/teardown, which was negligible compared to processing time, or host-GPU data transfers for the input and output streams, which have a fixed cost independent of the on-GPU scheduling policy.

5.1 Applications Tested

We investigated two irregular streaming applications representing two paradigms of execution. The first application, the Basic Local Alignment Search Tool (BLAST) [2], is from the domain of bioinformatics and represents a *filter cascade*. BLAST compares a genomic DNA sequence (the *query*) to a database of other sequences, looking for approximate sequence matches. It is organized as a pipeline of filtering stages, each of which receives a stream of database positions and either rules out a match to the query at that position or passes the position on to the next stage. Positions that survive all stages are deemed to contain a match.

MERCATOR's BLAST pipeline consists of four stages. Because each stage acts as an increasingly stringent filter on its input, the number of database positions surviving to each successive stage decreases vs the prior stage. Other applications that exhibit similar filtering behavior include network packet filtering [12], telescope data processing [16], and decision cascades in machine learning [17]. For an alternative GPU implementation of BLAST, see, e.g., [18].

We tested BLAST using the full human genome as a database and randomly generated DNA queries between 200 and 64200 bases. Longer queries produce more potential matches at earlier stages of the BLAST pipeline and so require more computation to process.

Our second application is a solver for the n Queens Problem, representing computations organized as a *tree search*. The goal is to enumerate all ways to place n queens on an $n \times n$ chess board so that no queen shares the same row, column, or diagonal with another. This problem is solved by a branching search: at each step, the solver considers all ways to place a queen on the ith row of the board given fixed positions for queens in rows $1 \ldots i-1$ and eliminates those possibilities that conflict with a previously placed queen. Only branching paths in the tree that successfully pass row n yield valid solutions. Other applications that exhibit similar tree-like structures include Delaunay mesh refinement [7, 8] and the Barnes-Hut algorithm for the n-body problem [3]. For an alternative GPU implementation of n Queens, see [15].

MERCATOR's n Queens implementation is a pipeline of n stages that process a stream of partial solutions corresponding to points in the search tree. Node i processes partial solutions that have successfully placed queens in rows $1 \ldots i-1$, or equivalently points at depth i in the tree. The computation is thus effectively breadth-first on the search tree. Nodes working near the root produce a large number (up to $n-i+1$) of outputs per input, while those toward the bottom of the tree produce few or no outputs for most inputs. Each node of the pipeline does essentially the same computation, albeit on partial solutions on a different row; hence, all nodes are of the same module type and execute the same code.

We tested n Queens for n from 13 to 16; larger values of n exhausted our GPU's DRAM memory just to store the solutions found. To distribute work across processors, the application's input stream contained all valid partial solutions for $i = 2$, which were divided equally among GPU blocks and used as roots for each block's search.

5.2 Results

Figures 1 and 2 compare the behavior of MERCATOR with original vs AFIE scheduler on the BLAST application. The AFIE implementation exhibited substantial reductions in

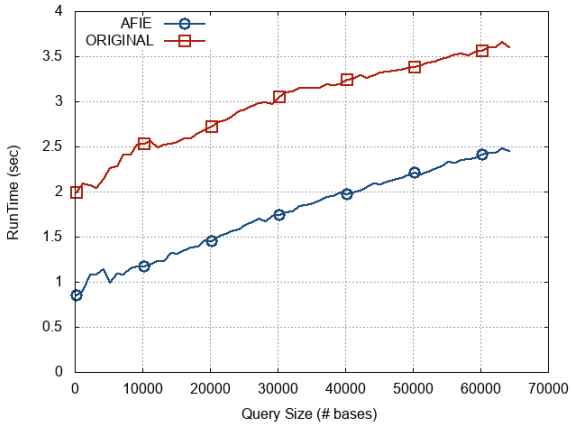

Figure 1. BLAST execution time.

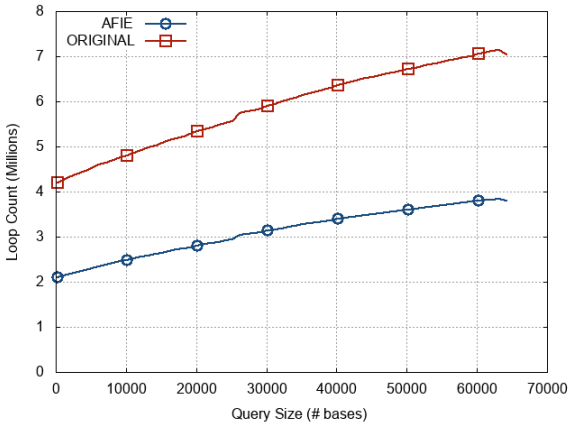

Figure 2. BLAST trips through scheduler.

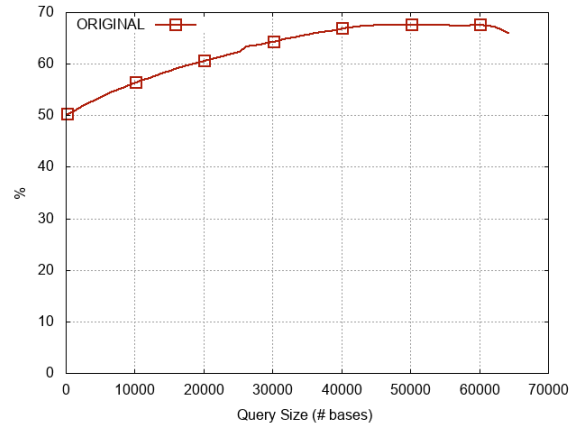

Figure 3. % of node firings in BLAST for which original MERCATOR scheduler misses an opportunity to fire the same node again.

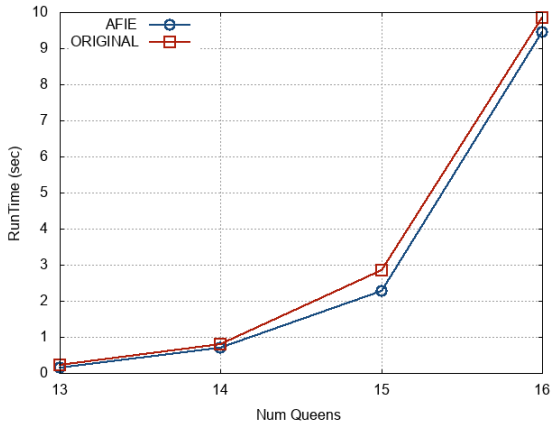

Figure 4. *n* Queens execution time.

NQueens	AFIE	Original
13	95.67%	88.80%
14	99.03%	92.67%
15	99.81%	88.69%
16	99.96%	83.57%

Table 1. Average % SIMD occupancy over all firings for *n* Queens.

NQueens	AFIE	Original
13	10.45	28.69
14	11.51	91.22
15	12.45	784.43
16	13.45	6898.01

Table 2. Number of firings with non-full SIMD occupancy in one run of *n* Queens (averaged over all pipeline instances).

both total running time (hence, higher throughput) and trips through the scheduler.

The BLAST application's pipeline performs a different function in every node. Each node therefore has a different module type, and there is no opportunity to improve occupancy by merging inputs across nodes. Nevertheless, we found that node firings even with the original MERCATOR scheduler had full SIMD occupancy over 99% of the time, comparable to the results obtained with the AFIE scheduler. The BLAST application provides abundant inputs to all nodes, so high occupancy is easy to achieve.

However, we found that the AFIE scheduler substantially reduces total trips through the scheduler compared to the original MERCATOR scheduler. This improvement may occur in part because AFIE, unlike the original scheduler, offers a stronger guarantee of both available inputs and free space before firing a node. However, a second contributing cause

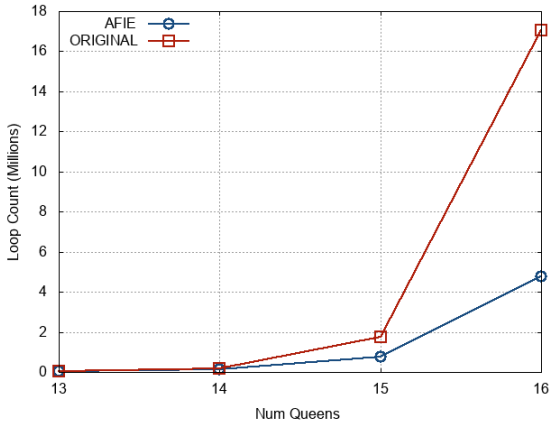

Figure 5. *n* Queens trips through scheduler.

is the fact that our AFIE implementation checks after each firing whether it is possible to continue firing the current node and does so if possible (policy point 2). In contrast, the original MERCATOR scheduler pessimistically computes the minimum number of firings before a node's downstream queue *could* become full and returns to the scheduler after that many firings, whether or not the queue actually fills. Figure 3 shows that the original scheduler often misses the opportunity to fire a node again without returning to the scheduler; in contrast, AFIE never misses such an opportunity. Because trips through the scheduler in original MERCATOR have high complexity and cost, this difference in behavior contributes to differing application throughput.

While it is in principle possible to modify MERCATOR's original scheduler to be less pessimistic like AFIE, the complexity of doing so when merging across multiple nodes of the same module type is substantially greater than for AFIE, which does not merge nodes yet still provides strong performance guarantees.

Figures 4 and 5 compare the original and AFIE schedulers on the *n* Queens problem. We see a similarly dramatic drop to BLAST in the number of trips through the scheduler. The difference in application runtime is more modest, but the AFIE scheduler consistently delivers lower running times than MERCATOR's original scheduler.

Nearly all nodes in the *n* Queens application have the same module type, so MERCATOR's scheduler *should* have ample opportunity to ensure high occupancy and reduce scheduler calculations by merging inputs across nodes. However, as shown in in Table 1, AFIE also exhibits notably greater average SIMD occupancy than the original scheduler. A likely cause of the difference in observed occupancy is that in *n* Queens, the small input stream is rapidly depleted; most work is the generated in intermediate stages of the pipeline by amplifying the small number of inputs. To avoid deadlock, MERCATOR's scheduler relaxes its requirement of full SIMD width to fire a module as soon as the input stream is depleted. But most outputs from the main module of *n* Queens feed back into the same module (albeit to another node), so this module remains active for a long time after the depletion of the input stream without enforcing full SIMD occupancy. In contrast, Our AFIE implementation enforces the full occupancy requirement for each node in the pipeline for as long as possible, until its upstream neighbor will no longer produce any more output.

Table 2 illustrates the impact of this difference: AFIE's firings with less than full SIMD occupancy, which occur only at the end of execution, stay nearly constant as the amount of work done by the application grows. In contrast, the number of such firings with the original MERCATOR scheduler increases rapidly with workload.

Here again, MERCATOR's scheduler exhibits unanticipated and undesirable behavior not seen with AFIE. It may again be possible to modify the original scheduler to avoid this pitfall. However, the complexity of doing in the presence of merging inputs across nodes seems high. In contrast, the AFIE scheduler avoids such complexity while still offering a strong occupancy guarantee.

Finally, we observed (data not shown) that the gap in overall application throughput between AFIE and the original MERCATOR scheduler becomes larger if the application is run with fewer active GPU blocks. This observation holds for both BLAST and *n* Queens. NVIDIA GPUs switch among active GPU blocks to hide memory access latencies incurred by each block during execution. We therefore hypothesize that the AFIE scheduler incurs fewer high-latency operations – perhaps due to simpler implementation or to better cache performance obtained with fewer switches between nodes. Future work will characterize the implications of AFIE scheduling for memory access patterns.

6 Related Work

Streaming is a well-studied model of computation with implementations for multiple architectures, including GPUs and other accelerators. Frameworks that support streaming include, e.g., StreamIt [14], Brook [4], and StreamC [9].

StreamIt, which is based on the Synchronous Data Flow (SDF) model of computation [10], developed advanced methods for computing a *static* schedule for node execution that minimize the need for inter-node queuing. Their methods rely on the fact that SDF guarantees a predictable number of outputs for each input to a node, that is, regular dataflow by our definition. In contrast, our work supports irregular applications for which a static schedule is not possible.

Our focus on switching as a behavior to optimize is inspired in part by work of Agarwal et al. [1]. That work considers scheduling for pipelines with the goal of minimizing an application's cache miss rate. Cache misses arise both from switching between nodes and from the need to access items stored in inter-node queues in DRAM. We address the first

of these costs in our work, while the second is an interesting topic for future work. However, Agarwal et al. consider a very different architectural model, in which a pipeline is distributed across multiple processors that execute different nodes concurrently, and their work does not consider SIMD occupancy or irregular dataflow. In contrast, our work is motivated by a quite different, more loosely-coupled mapping of irregular streaming pipelines to a multiprocessor.

Burtscher et al. [5] collect, implement, and profile many GPU codes for irregular applications. They consider more expansive notions of irregularity that encompass both computation and memory access patterns. Our work focuses more narrowly on computations with irregular dataflow that map well to the streaming paradigm.

7 Conclusion and Future Work

We have introduced the AFIE scheduling policy for irregular streaming computations on SIMD processors. AFIE satisfies strong theoretical performance guarantees in both SIMD occupancy and minimization of node-switching overhead while offering a simple, efficient implementation. While our exposition focused on linear pipelines, AFIE extends straightforwardly to tree-structured application topologies as well. We implemented AFIE in a framework for building irregular streaming applications on NVIDIA GPUs and observed improvements in overall throughput and in the frequency of scheduler invocation. On the applications we studied, our AFIE implementation proved less prone to performance pitfalls arising from implementation complexity than MERCATOR's original scheduler.

Several opportunities exist for future work on scheduling irregular streaming applications on SIMD-parallel architectures. First, the AFIE policy assumes that every node in an application has a unique predecessor. While this property holds for pipelines and trees, it is not true of applications whose dataflow graphs include directed cycles. For example, MERCATOR supports such cyclic application topologies. Scheduling such applications safely (without deadlock) and efficiently requires further study. Second, our model of application costs focuses on two aspects: SIMD occupancy, and the overhead of switching between nodes. A model that more explicitly accounts for cache performance, as in the work of [1], might suggest a modified policy. At a minimum, it would be informative regarding the most efficient sizes for inter-node queues; AFIE, in contrast, is agnostic about the sizes of queues provided they meet the minimum threshold for safe execution. Finally, we plan to develop a wider array of irregular streaming benchmark applications to facilitate the study of scheduling and other aspects of runtime support for such applications.

Acknowledgments

The authors thank Dr. Angelina Lee for helpful discussions and the anonymous referees for their valuable comments. This work is supported by NSF CISE award CNS-1763503.

References

[1] Kunal Agrawal, Jordyn Maglalang, and Jeremy T Fineman. 2014. Cache-conscious scheduling of streaming pipelines on parallel machines with private caches. In *Proc. 21st Int'l Conf. High Performance Computing*. IEEE, 1–12.

[2] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 1990. Basic local alignment search tool. *J. Molecular Biology* 215, 3 (1990), 403–410.

[3] Josh Barnes and Piet Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature* 324, 6096 (1986), 446.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics* 23, 3 (2004), 777–786.

[5] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proc. 2012 IEEE Int'l Symp. Workload Characterization (IISWC)*. IEEE, 141–151.

[6] Stephen V Cole and Jeremy Buhler. 2017. MERCATOR: a GPGPU framework for Irregular Streaming Applications. In *Proc. 2017 Intl' Conf. High Performance Computing & Simulation*. IEEE, 727–736.

[7] Paul-Louis George and Houman Borouchaki. 1998. *Delaunay Triangulation and Meshing – Application to Finite Elements*. Hermes.

[8] Benoît Hudson, Gary L Miller, and Todd Phillips. 2007. Sparse parallel Delaunay mesh refinement. In *Proc. 19th Ann. ACM Symp. Parallelism in Algorithms and Architectures*. ACM, 339–347.

[9] Ujval J Kapasi, Scott Rixner, William J Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D Owens. 2003. Programmable stream processors. *Computer* 36, 8 (2003), 54–62.

[10] E.A. Lee and D.G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235—-1245.

[11] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. 2010. Deadlock avoidance for streaming computation with filtering. In *22nd ACM Symp. Parallelism in Algorithms and Architectures*. ACM, 243–252.

[12] Bin Ren, Gagan Agrawal, James R Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proc. 2013 IEEE/ACM Int'l Symp. Code Generation and Optimization (CGO)*. IEEE, 1–10.

[13] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks. In *Proc. 13th Systems Administration Conference (LISA)*. 229–238.

[14] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Proc. Int'l Conf. Compiler Construction*. Springer, 179–196.

[15] Krishnahari Thouti and SR Sathe. 2012. Solving N-Queens problem on GPU architecture using OpenCL with special reference to synchronization issues. In *Proc. 2nd IEEE Int'l Conf. Parallel, Distributed and Grid Computing*. 806–810.

[16] Eric J Tyson, James Buckley, Mark A Franklin, and Roger D Chamberlain. 2008. Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 595, 2 (2008), 474–479.

[17] Paul Viola and Michael Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Proc. IEEE Comp. Soc. Conf. Computer Vision and Pattern Recognition*. I511–I518.

[18] Panagiotis D Vouzis and Nikolaos V Sahinidis. 2010. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics* 27, 2 (2010), 182–188.

Appendix: Proofs for Switching Competitiveness

Proof of Lemma 3.3

Proof. We prove two claims that imply the lower bound on S_i for $1 \le i \le m$. The first claim alone proves the bound for S_m, while the second proves the bound for S_0.

Claim 7.1. *Any feasible schedule for executing the pipeline must switch to node n_i, $i > 0$, at least $\left\lceil \frac{b_i}{|q_i|} \right\rceil$ times.*

Proof of claim. When execution switches to n_i, it cannot consume more than $|q_i|$ inputs without emptying its input queue. Because nodes do not execute concurrently, execution must then switch away from n_i. If execution later produces additional inputs for n_i, it must eventually switch back to n_i to consume them. The claim follows.

Claim 7.2. *Any feasible schedule for executing the pipeline must switch to node n_i, $i < m$, at least $\left\lceil \frac{b_{i+1}}{|q_{i+1}|} \right\rceil$ times.*

Proof of claim. When execution switches to n_i, it cannot produce more than $|q_{i+1}|$ outputs without filling its output queue. Because nodes do not execute concurrently, execution must then switch to another node. Hence, execution must switch to n_i at least $\left\lceil \frac{b_{i+1}}{|q_{i+1}|} \right\rceil$ times to produce all its outputs. □

Proof of Lemma 3.4

Proof. At the time a node n_i becomes active (other than at the end of stream), q_i must hold at least $|q_i| - v_{i-1}g_{i-1} + 1$ items; moreover, when n_i subsequently becomes inactive, q_i must hold at most $v_i - 1$ items. Hence, after augmenting the capacity of each queue as stated in the Lemma, the number of inputs consumed between a node's becoming active and its becoming inactive is at least

$$(|q_i| + v_{i-1}g_{i-1} + v_i - 2) - v_{i-1}g_{i-1} + 1 - (v_i - 1) = |q_i|.$$

Similarly, the number of inputs that must accumulate in q_i between n_i becoming inactive and n_i becoming active is at least $|q_i|$, except at the end of stream.

We again consider the number of switches to each node n_i. Node n_0 is always active while it has any inputs. Hence, any time an AFIE schedule switches to n_0, its output queue q_1 has $< v_i$ items, and n_0 fills q_1 until it either blocks on q_1 or runs out of input stream. Hence, execution switches to n_0 at most $\left\lceil \frac{b_1}{|q_1|} \right\rceil$ times, including switches back after blocking on q_1 and the final switch (if any) to produce any remaining output. This number of switches matches the lower bound for S_0 derived from Claim 7.2.

Node n_m never blocks, so we switch to it only when its input queue q_m fills enough to make it active (at which point n_m can drain at least $|q_m|$ inputs), plus once more at the end of execution if n_m must process any remaining items in q_m. Hence, execution switches to n_m at most $\left\lceil \frac{b_m}{|q_m|} \right\rceil$ times, which matches the lower bound for S_m derived from Claim 7.1.

For $0 < i < m$, we consider the number of times n_i becomes eligible to fire. Each time this occurs, execution must switch to n_i before it again becomes *in*eligible to fire, i.e., before either n_i blocks on its input queue and becomes inactive or n_i blocks on its output queue and causes n_{i+1} to become active.

There are three circumstances in which node n_i can become newly eligible to fire:

1. node n_i transitions from inactive to active.
2. node n_{i+1} transitions from active to inactive.
3. at the end of input stream, if there are more than zero inputs remaining in q_i but not enough to otherwise make n_i active.

Events (1) and (2) may happen concurrently or not, but the number of switches cannot exceed the sum of counts for all three events.

Event (1) occurs when q_i fills enough to make n_i active after being empty enough to make it inactive, i.e. after accumulating at least $|q_i|$ new items. Hence, this transition occurs at most $\left\lfloor \frac{b_i}{|q_i|} \right\rfloor$ times. Event (2) occurs when q_{i+1} empties enough to make n_{i+1} inactive after previously being full enough to make it active, i.e. after at least $|q_i|$ items are consumed. Hence, this transition occurs at most $\left\lfloor \frac{b_{i+1}}{|q_{i+1}|} \right\rfloor$ times. Event (3) occurs at most once.

We conclude that the total number of events (1)-(3), and hence the total number of switches to n_i, is at most

$$\begin{aligned}\left\lfloor \frac{b_i}{|q_i|} \right\rfloor + \left\lfloor \frac{b_{i+1}}{|q_{i+1}|} \right\rfloor + 1 &\le \left\lceil \frac{b_i}{|q_i|} \right\rceil + \left\lceil \frac{b_{i+1}}{|q_{i+1}|} \right\rceil + 1 \\ &\le 2 \max\left(\left\lceil \frac{b_i}{|q_i|} \right\rceil, \left\lceil \frac{b_{i+1}}{|q_{i+1}|} \right\rceil \right) + 1 \\ &= 2S_i + 1\end{aligned}$$

We conclude that the total number of switches over all nodes is at most

$$\sum_{i=0}^{m} 2S_i + 1 = 2S^* + 1$$

as claimed. □