Parallel Computing 109 (2022) 102863

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Reducing queuing impact in streaming applications with irregular dataflow

Stephen Timcheck *, Jeremy Buhler

Washington University in St. Louis, St. Louis, MO, United States of America

ARTICLE INFO

Keywords:
Queuing
SIMD
Irregular
Dataflow
Streaming

ABSTRACT

Throughput-oriented streaming applications on massive data sets are a prime candidate for parallelization on wide-SIMD platforms, especially when inputs are independent of one another. Many such applications are represented as a pipeline of compute nodes connected by directed edges. Here, we study applications with irregular dataflow, i.e., those where the number of outputs produced per input to a node is data-dependent and unknown *a priori*. We consider how to implement such applications on wide-SIMD architectures, such as GPUs, where different nodes of the pipeline execute cooperatively on a single processor.

To promote greater SIMD parallelism, irregular application pipelines can utilize queues to gather and compact multiple data items between nodes. However, the decision to introduce a queue between two nodes must trade off benefits to occupancy against costs associated with managing the queue and scheduling the nodes at its endpoints. Moreover, once queues are introduced to an application, their relative sizes impact the frequency with which the application switches between nodes, incurring scheduling and context-switching overhead.

This work examines two optimization problems associated with queues. First, given a pipeline with queues between each two nodes and a fixed total budget for queue space, we consider how to choose the relative sizes of inter-node queues to minimize the frequency of switching between nodes. Second, we consider which pairs of successive nodes in a pipeline should have queues between them to maximize overall application throughput. We give an empirically useful approximation to the first problem that allows for an analytical solution and formulate a performance model for the second that directs implementation toward higher-performing strategies.

We implemented our analyses and resulting optimizations in applications built using Mercator, a framework we designed to support irregular streaming applications on NVIDIA GPUs. We demonstrate that these optimizations yield meaningful performance improvements for several benchmark Mercator applications.

1. Introduction

Streaming computations process large datasets as a sequence of input items that are transformed by a pipeline (more generally, a graph) of computational stages, or *nodes*. Such computations arise in numerous high-impact applications, ranging from biosequence analysis [1] to network packet inspection [2] to astrophysics [3]. Because streaming computations exhibit multiple forms of parallelism, they have been intensively studied to find efficient strategies for parallel implementation.

A key property of a streaming computation is whether the data volume changes in a predictable way from a node's input to its output. If each j inputs to a node result in exactly k outputs, the node's behavior is said to be *regular*. Pipelines of regular nodes can be implemented efficiently using limited buffering between nodes and static scheduling [4]. But some streaming computations, including the examples cited above, exhibit *irregular* dataflow: the amount of output generated by a node per input item is variable, data-dependent, and therefore unknown *a priori*. This work focuses on strategies to effectively parallelize such irregular streaming computations.

When a streaming computation performs largely independent operations on successive inputs in the stream, the computation's throughput can be increased by exploiting fine-grained data parallelism across its inputs. Wide-SIMD processors such as GPUs are designed to exploit data parallelism and hence are tempting targets for such applications. However, irregular dataflow interferes with SIMD parallelism because different inputs to a pipeline may require different amounts of work or may even be filtered away entirely at different stages of the pipeline. If data cannot be remapped from one SIMD lane to another in mid-computation, the *occupancy* of the processor (that is, the fraction of SIMD lanes doing useful work) will suffer.

* Corresponding author.
E-mail address: stimcheck@wustl.edu (S. Timcheck).

https://doi.org/10.1016/j.parco.2021.102863
Received 28 March 2021; Received in revised form 30 August 2021; Accepted 22 October 2021
Available online 3 November 2021

To improve the occupancy of irregular dataflow pipelines, one may place intermediate *queues* between successive compute nodes. A queue following a node functions as a staging area where data from the node can be accumulated, compacted, and then redistributed across SIMD lanes to ensure full occupancy of the next node. However, the insertion of queues adds overhead to the application that may negate the performance benefits of higher SIMD occupancy. In cases where data production rates are low or the application exhibits locally almost-regular data flow, it may not be worth improving occupancy by adding a queue between stages.

A second challenge arises when adding queues to pipelines implemented on modern GPU devices, today's most popular wide-SIMD architectures. The processors of a GPU typically run asynchronously, and existing APIs offer little support for synchronization between them. Hence, a natural application mapping to a GPU runs a separate replica of the pipeline on each processor, rather than dividing the pipeline's nodes across processors. GPUs also offer limited support for preemption, so the stages of the pipeline must be cooperatively scheduled on the single processor. Finally, the number of processors is large enough to run hundreds of pipeline replicas at once.

Each pipeline replica on a GPU needs its own queue space. Given a large number of replicas, it becomes important to limit the amount of queue space allocated to each, and therefore to divide that space wisely among the queues between different pipeline stages. As we will show, the algorithm used to schedule execution of different pipeline stages can interact with the differing rates of data production from each stage, creating an opportunity to allocate queue space in a way that minimizes the application's overhead due to pipeline scheduling.

This work addresses two questions in the setting of irregular streaming dataflow pipelines on GPUs and other wide-SIMD processors. First, we consider the problem of dividing a limited memory budget among queues in a pipeline. Assuming a simple, effective scheduling policy for pipeline stages [5], we show how to divide space among queues so as to roughly minimize the frequency with which the scheduler must be invoked while processing a data stream. Second, we formalize a trade-off for when to insert queues between successive pipeline stages. Using easily-obtained performance metrics from an application's profile, we formulate a performance model that can aid in selecting which stages should be merged together and which should have queues between them.

We deploy the optimizations described in this work in the Mercator framework for irregular streaming computation. We developed Mercator to enable high-performance implementation of irregular streaming applications in the CUDA language on NVIDIA GPUs, though its basic approach is suitable for a variety of wide-SIMD processors. We quantify the empirical utility of our optimizations for irregular streaming applications written using Mercator, showing that the optimizations can have a material impact on an application's overall throughput. We also identify limits to optimization, particularly optimal queue sizing, imposed by the need to ensure certain minimal queue sizes for safe execution.

The rest of the paper is organized as follows. Section 2 examines related work. Section 3 explains our application model and associated performance metrics, while Section 4 briefly describes our Mercator system, which realizes this application model for NVIDIA GPUs. Section 5 provides a method for partitioning space among queues in an application so as to minimize overhead given a limited space budget. Section 6 describes a model for estimating the performance consequences of adding queues between compute nodes. Section 7 evaluates both of our techniques on irregular streaming applications implemented in Mercator. Finally, Section 8 concludes and explores future work.

2. Related work

2.1. Queuing optimizations for streaming pipelines

Many application frameworks have been developed to support *regular* streaming dataflow applications on parallel systems. A prominent example, StreamIt [4], was built around the synchronous data flow (SDF) [6] model of computation. In StreamIt, the number of outputs per input data item for each node is fixed at compile time, which allows effective static scheduling of nodes with minimal queue space allocation and no remapping. In contrast, the irregular problems we target do not have the luxury of knowing how much data will be generated at each stage, thus creating a need for data-driven decisions about queue placement and sizing.

Subhlok and Vondran [7] examined a similar problem to the merging of compute stages presented in this paper. They consider a pipeline of tasks, equivalent to compute stages in our model. Each task can be mapped to processors with data- and task-parallel mapping. However, their model allows for forking inputs to different replicas and re-converging to a single replica, which is not part of our model. They explore combining tasks into *modules*, which are collections of two or more tasks. These modules are then evenly assigned to processors on the system. Our model similarly considers merging compute stages, but a single pipeline cannot be split across processors on our target platform, creating different design problems. Our work models impacts due to wide-SIMD execution, while their work focuses on general-purpose MIMD processing.

Benoit and Robert [8] considered a similar problem, trying to optimize for both latency and throughput. Their work explores how to map data-parallel pipelines on parallel platforms. In their work, as in ours, merging compute nodes increases the computational load on a processor but may decrease communication, which in our case would be reading, writing, and managing an intermediate queue. Although their model does not consider communication costs between processors, it works with a more general purpose MIMD processor. Hence, their communication cost would be equivalent to the scheduler cost we consider in our model.

2.2. The Mercator framework

An early version of our Mercator system was described in [9,10]. Many techniques to combine work across distinct nodes that perform the same computation were used in the initial version. The newer version described here eschews such combining, which empirically incurred a very high overhead, and instead focuses on minimizing overheads associated with queuing and scheduling as much as possible. Empirically, the new version of Mercator reduces overheads by at least an order of magnitude relative to its predecessor. One key aspect of Mercator developed since the earlier version is its runtime node scheduler [5], which provably reduces scheduling overhead to within a constant factor of the best possible, even for a clairvoyant algorithm that knows how many outputs will be produced for each input at each node. This scheduler interacts with our queue resizing optimization as described in Section 5.

Irregularity on GPUs is well-documented [11], as is the need for dynamic data-lane mapping to overcome it [12]. Domain-specific techniques to address irregular dataflow have been described for, e.g., graph algorithms [13–16], n-body simulation [17], data-flow analysis [18], and graphics rendering [19]. Mercator seeks to provide abstractions that capture a wide variety of irregular computations in a way that exposes data parallelism to the application developer. Because it does not remap work from one processor to another, Mercator preserves guarantees of FIFO processing of inputs within one processor, which allows for optimizations supporting, e.g., stateful computation [20]. The uniprocessor streaming model also allows analytical reasoning supporting optimization strategies, as this work shows.

Mercator chooses to combine scheduling and all node computations of a pipeline in a single kernel, in contrast to frameworks like Thrust and OpenACC that implement computations as separate GPU kernels managed by a host-side driver. Mercator's "uberkernel" design [19] minimizes the latency and synchronization associated with host-device communication. We note, however, that the queue-related optimizations in this work apply independently of whether nodes are scheduled by the host or by a device-side scheduler. Similarly, although Mercator is built for devices targeted via CUDA, its design and our optimizations are compatible with an implementation in, e.g., OpenCL to target other classes of wide-SIMD processors.

3. Application model

In this section, we define the abstract properties of our streaming dataflow applications, as well as the characteristics of our target wide-SIMD architectures. As shown in Fig. 1, an application is a linear pipeline of *compute nodes* $n_1 \ldots n_m$ with dataflow *edges* connecting each n_i to the next n_{i+1}. Each time a node executes, it consumes a vector of up to v items from its input and produces a data-dependent number of items, perhaps of a different size/type, on its output.

The runtime behavior of a node n_i is characterized by three parameters: its *service time* t_i, its *average gain* g_i, and its *maximum vector gain* m_i. The service time t_i is the time for a node to process a vector of input items v; the time is the same for any number of items $\leq v$. The average gain g_i is the average number of outputs produced per input item consumed, which may be greater or less than 1. In contrast, the maximum vector gain m_i is the typical value of the *maximum* number of outputs produced in any one SIMD lane from an input vector containing a single input per lane. We note that if a node consumes a vector of inputs and produces a vector of outputs with maximum vector gain m, the lock-step nature of wide-SIMD execution ensures that, absent work-to-thread remapping, every SIMD lane processing this output will take time proportional to m, as lanes with fewer inputs must wait for the longest-running lane to complete. Hence, as discussed in Section 6, max vector gain can be a useful predictor of computational cost in the absence of remapping through queues.

For convenience, we also define the *cumulative average gain* $G_k = \prod_{i=1}^{k} g_i$ and *cumulative max vector gain* $M_k = \prod_{i=1}^{k} m_i$ to be the average number of outputs per SIMD lane and the typical maximum number of outputs in any one lane from node n_k per input consumed by n_1 respectively. Formally, G_k accurately summarizes the behavior of a mean-value model of the pipeline; however, M_k is a more heuristic quantity because average per-node values for m_i cannot simply be composed, irrespective of the detailed distribution of gains. We estimate m_i as the empirical *mode* of n_i's max vector gain, which for the unimodal distributions we observe in practice is robust to occasional outliers, and multiply these modes for successive nodes to estimate M_k. More accurate modeling of gain distributions is left to future work. Finally, node n_i has a *gain limit* u_i that defines the largest number of outputs that could *ever* be produced for a single input; this quantity is used to compute the minimum amount of downstream queue space needed for a node to run safely.

Each edge between two nodes has an associated queue. Items from an edge's upstream node accumulate in contiguous slots of the queue until the downstream node executes, at which point the downstream node pulls contiguous vectors of up to v items from the queue. We assume that each queue has a fixed size; while this size could be periodically adjusted during execution, doing so incurs overhead (on the order of tens of milliseconds per adjustment on our target platform), so we treat queue sizes as fixed within a single "epoch" of execution for purposes of our analyses.

As discussed earlier, we assume that our target architecture runs a replica of the complete pipeline on each of its processors, with different copies sharing a global input stream and output buffer, but not intermediate data structures such as queues. This design is compatible with modern GPUs, which offer limited support for inter-processor synchronization. Each processor runs a *scheduler* that manages execution of its pipeline replica's nodes. Each time the scheduler is called, it selects a node with items in its input queue and space in its output queue and causes that node to *fire*, consuming some amount of input. The application ends when no node has any inputs remaining.

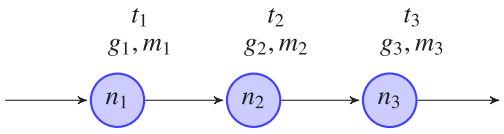

Fig. 1. A simple pipeline application topology. Node n_1 feeds into n_2, and n_2 feeds into n_3. Node n_i has service time t_i, average gain g_i, and max vector gain m_i.

In optimizing a pipeline's execution, we seek to maximize its *throughput*, or equivalently to minimize the total time to completely process a large number of inputs to n_1.

4. Mercator: a framework for irregular streaming computation on NVIDIA GPUs

In this section, we describe Mercator, a development framework that realizes our computational model for NVIDIA GPUs. Mercator both motivates the present work, as it supports rapid development of irregular streaming dataflow applications for GPUs, and serves as our testbed for benchmarking the optimizations described in subsequent sections. An early version of Mercator was previously described in [9], but the present version (2.0) has been simplified and tuned for performance. The software is publicly available[1] under an Apache license.

4.1. Specifying computations in Mercator

A Mercator application is defined by its *topology specification* and a set of functions, written in NVIDIA's CUDA language, that implement the operations at each node. The topology specification is parsed by the Mercator compiler, which emits both a CUDA *skeleton* with code stubs for the functions to be implemented at each node and supporting code to integrate these functions into a complete application. The application developer fills in the bodies of the node functions and then uses NVIDIA's CUDA compiler to build the full device-side application, which can be invoked by code running on the host processor.

4.1.1. Application topology

Fig. 2(a) illustrates the topology specification for a simple Mercator application. The type of a node is given by a *module* signature, which specifies the node's input and output data types and its gain limit (the u_i of the previous section). A module may have multiple output *channels*, each with its own data type and gain limit; hence, Mercator applications may be structured as trees, not just as linear pipelines. Multiple nodes in an application may share the same module type. Each module has an associated `run()` function that implements its operation; all nodes with this module type execute the same code, albeit at different points in the application's topology.

A topology is defined by specifying nodes, each with its module type, and *edges* that connect an output channel of one node to the input of another with a compatible data type. A single node is designated the *source* of the application. By default, the source receives a stream of consecutive integers, but can be specified to instead consume a buffer of objects in GPU memory or the output of a user-specified function. Any output channel of a node may be connected to a *sink* that stores objects emitted on the channel in a buffer in GPU memory.

[1] https://github.com/jdbuhler/Mercator

```
Application Filter;

Module Parser : int -> float : 4;

Module Threshold : float -> float;

Node p : Parser;
Node t : Threshold;
Node s: Sink<float>;

Source p buffer;
Edge p -> t;
Edge t -> s;

NodeParam Parser::table : float *;
NodeParam Threshold::thresh : float;
```

(a) topology specification

```
__device__ void
Parser::run(int i, unsigned int n) {
  const float *entry =
    getParams()->table + 4*i;
  bool stop = (threadIdx.x < n);
  for (int j = 0; j < 4; j++) {
    float v;
    if (!stop)
      {v = entry[j]; stop = (v==0.0); }
    push(v, !stop);
  }
}

__device__ void
Threshold::run(float f, unsigned int n) {
   float result;
   if (threadIdx.x < n)
     result = compute(f);

   push(result,
        threadIdx.x < n &&
        result >= getParams()->thresh);
}
```

(b) GPU-side CUDA code

Fig. 2. Mercator topology specification and code sample. The first node p reads selected sets of up to four consecutive floats from an entry in a table, stopping if it encounters a zero. The second node t performs a computation on each value read and emits the result if it is greater than or equal to a user-defined threshold value. The table and threshold are specified by the host. Each node's `run()` function takes a parameter n that indicates the number of threads that receive inputs. Note that `push()` is always called with all threads but is predicated to indicate which threads actually emit outputs.

Nodes may be assigned *parameters*, which are set on the host prior to invoking an application and are read-only from within it. Parameters may be used to alter the functions of different nodes that share the same module type, e.g., by providing different coefficients to different copies of a filter. Nodes may also be assigned *state* variables, allowing them to implement stateful operations such as reductions over their input stream.

4.1.2. Application skeleton

Fig. 2(b) shows a simplified view of the application skeleton corresponding to the given topology. Each module's `run()` function is assumed to be executed with all available CUDA threads. Each thread is given its own input if one is available; the second argument to `run()` specifies the number of threads (starting from thread 0) that have valid inputs. A future version of Mercator will offer the option to take input as an array of values visible to all threads, so that the `run()` function may map inputs to threads arbitrarily.

A `run()` function emits outputs to a channel by invoking a *push* operation. Push operations are always called with all threads but take a predicate to indicate which threads actually have an output to write. Multiple push operations may be invoked in one call to `run()`. The developer is responsible for ensuring that one call to `run()` does not push more outputs than the number of inputs received times the channel's gain limit.

Other functions may be generated as part of an application's skeleton. For example, a node with state variables will have an `init()` function, which is called once at application startup to initialize its state.

4.1.3. Invocation from the host

A Mercator application is compiled into a library that can be linked against a CUDA host program. The host program instantiates the application as an object, which can be configured through a host-side API to set parameters and specify buffers associated with the source and sink(s). Once configured, the host calls the application object's `run()` method, which launches a CUDA kernel containing the entire application to consume its input stream and emit any output streams. Fig. 3 shows an example of instantiating and invoking a Mercator application.

```
Filter filterApp;

Buffer<int> inputs(100000);
inputs.set(...); // set from host data

Buffer<float> outputs(400000);

app.p.table = /* device pointer */;
app.t.thresh = 3.75;

app.setSource(inputs);
app.s.setSink(outputs);

app.run();

outputs.get(...); // read back to host
```

Fig. 3. Host-side API of the sample application of Fig. 2. Mercator provides a `Buffer` type that serves as an input or output stream.

Mercator applications can be configured and invoked asynchronously from the host and can be associated with CUDA streams. Users requiring the highest performance may thereby use streams to overlap a Mercator application's computation with movement of input and output data to and from buffers on the GPU.

4.2. Runtime realization of Mercator applications

A Mercator application exists on the GPU as a single kernel containing the code for all nodes along with a high-level application driver. The entire application is replicated and runs separately in each CUDA block. Instances of the application asynchronously claim chunks of the shared input stream by atomically incrementing an input pointer and similarly emit outputs asynchronously to shared output buffers. Within one replica, items flow through the pipeline of nodes in FIFO order; however, writes to the shared output by different replicas are unordered. A future version of Mercator will add the ability to synchronize replicas to emit outputs in the global order of the input stream.

Edges between nodes become *queues*, implemented as circular buffers whose size is fixed at the time the application is instantiated. When the application's scheduler begins to fire a node, the node continues running until either its input queue (the input stream, for the source node) is exhausted or one of its output queues fills.

Mercator schedules nodes at runtime using a protocol developed in our prior work, the AFIE ("Active Full, Inactive Empty") scheduler [5]. Briefly, AFIE marks a node "active" when it has a full input queue and "inactive" when this queue is emptied. A node is eligible to execute when it is active and its immediate downstream neighbors, if any, are inactive. As with any flow control protocol involving execution of multiple entities with finite queues between them, the scheduling policy must ensure that a node with input must eventually be able to make progress. We proved in [5] that AFIE is deadlock-free (that is, a node with input eventually becomes eligible to execute), that it ensures that nodes always execute with a full vector-width of inputs, and that it incurs only about twice as many *switches* (calls into the scheduler to choose a new node to execute) as would a clairvoyant scheduler that knew in advance the number of outputs produced by each node for each input.

4.2.1. Motivation for optimizations

As previously discussed, the overhead of dynamic reallocation of queues motivates us to consider how to choose an appropriate static size for each queue, at least for a given epoch of execution. Larger queues allow an application's nodes to run longer before returning to the scheduler, which reduces overhead and therefore boosts application throughput. However, real GPUs impose practical limits on queue size. To fully utilize a GPU's processors and hide latency, a Mercator application may be instantiated in several hundred copies per device (one per CUDA block). Even if one instance's queues hold only a few thousand elements apiece, that can result in tens of megabytes devoted to queues overall. More aggressive queue sizing can consume hundreds of megabytes or even gigabytes of GPU global memory, which may interfere with the space needed to store the application's input and output streams. Hence, we are motivated to consider how best to allocate a limited amount of queue space among the nodes of an application so as to achieve the highest possible throughput. We formalize and solve this problem in Section 5.

While queues are useful to ensure that nodes in an irregular application receive an input for every SIMD lane, they incur significant costs. These costs include not only reads and writes of queue memory but also overhead associated with scheduling execution of the nodes at either end of a queue. The performance gains due to improved SIMD lane occupancy when a queue is inserted between nodes must therefore be weighed against the overhead incurred by its presence. Section 6 seeks to model this tradeoff and quantitatively guide where queues should be placed.

5. Choosing sizes for finite inter-node queues

Consider a pipeline with nodes $n_1 \dots n_h$, with a queue between each successive pair of nodes. In what follows, we make two key assumptions about how the pipeline behaves. First, we assume that once a node starts firing, it continues consuming full vector-widths of inputs for as long as possible, i.e., until either its input queue empties or its output queue fills. Second, the number of elements in any single queue q_i cycles between full and empty. In other words, the number of elements in q_i starts at zero, increases monotonically until q_i becomes full, then decreases monotonically back to zero before again starting to increase. Not every possible schedule of node execution satisfies these two conditions; for example, a schedule that optimized execution latency might fire a node as soon as any input is available, never allowing its queue to fill. However, the AFIE scheduler used by Mercator *does* satisfy both conditions, and we have shown [5] that such behavior is consistent with a nearly throughput-optimal schedule. In what follows, we refer to a schedule satisfying these two conditions as *efficient*.

Because an efficient scheduler does not switch away from a node until necessary, the larger the inter-node queues, the more input vectors a node can typically consume before control returns to the scheduler. Hence, larger queues are desirable because they reduce the overhead associated with scheduler invocations, or *switches*, whose cost can be on the same order as node service times.

However, as discussed earlier, a good GPU implementation of the application may require a large number of replicas of the pipeline — at least one per processor to avoid complex inter-processor communication, and usually multiple replicas per processor (via multiple active CUDA blocks) to take advantage of GPUs' ability to hide memory access latency by switching among multiple computations. For this reason, the cumulative memory cost of using arbitrarily large queues for each pipeline can be infeasible. Moreover, the number of scheduler invocations varies inversely with queue size, so at some point, the reduction in scheduling overhead from increasing queue sizes reaches a point of diminishing returns. We therefore assume that each replica of the pipeline receives only a small, fixed amount of memory to divide among all its queues.

We consider the following question: how does the allocation of memory among an application's queues impact the rate at which it must switch between nodes? We will quantify this switching rate for a given allocation, then show how to select an allocation that roughly minimizes switches for a given total amount of memory.

5.1. Bounding rate of switches under efficient scheduling

Let q_i be the queue between n_i and n_{i+1}, and suppose this queue can hold c_i items. Define the *scaled capacity* d_i of queue q_i by $d_i = c_i/G_i$. Scaled capacity normalizes the size of each queue to units of "inputs to node n_1". For example, if n_1 has gain 2, then each input to n_1 results in an average of two items inserted into q_1. The results that follow are more easily expressed in terms of scaled capacities.

Under an efficient schedule, n_i's input queue empties once per c_{i-1} items it consumes, and its output queue fills on average once per c_i/g_i items it consumes. These two events (emptying of input or filling of output queues) are the only reasons that execution switches away from n_i, so their frequency determines the number of such switches. However, the two events can sometimes occur concurrently — about once per $\mathrm{lcm}(c_{i-1}, c_i/g_i)$ inputs consumed[2] — which results in only one rather than two switches. In short, we can establish the following lemma:

[2] This result holds even for arbitrary rational g_i for the least common multiple of two rational values a/b, c/d; defined to be the smallest rational number that is a multiple of each; assuming both values are in lowest form, this LCM is computed as $lcm(a,b)/gcd(c,d)$.

Lemma 5.1. *For $1 < i < h$, the rate R_i of switches away from n_i per item consumed by n_i is given by*

$$R_i = \frac{1}{G_{i-1}}\left[\frac{1}{d_{i-1}} + \frac{1}{d_i} - \frac{1}{\mathrm{lcm}(d_{i-1}, d_i)}\right].$$

Proof. We first observe that, because each input to n_i produces g_i outputs, node n_i needs c_i/g_i inputs to fill its output queue. n_i begins firing for the first time with a full input queue and an empty output queue. The number of items processed before returning to this initial state (full input queue, empty output queue) must be a multiple of *both* c_{i-1}, the number of inputs needed to fill q_{i-1}, and c_i/g_i, the number of inputs needed to fill q_i, so that q_i fills exactly when q_{i-1} empties (after which the former empties and the latter fills). This event first occurs after processing $z = \mathrm{lcm}(c_{i-1}, c_i/g_i)$ items. Since $g_i = G_i/G_{i-1}$, we can rewrite z as follows:

$$\begin{aligned} z &= \mathrm{lcm}(c_{i-1}, c_i/g_i) \\ &= \mathrm{lcm}(c_{i-1}, G_{i-1}c_i/G_i) \\ &= G_{i-1}\,\mathrm{lcm}(c_{i-1}/G_{i-1}, c_i/G_i) \\ &= G_{i-1}\,\mathrm{lcm}(d_{i-1}, d_i). \end{aligned}$$

To compute the number of switches away from n_i during one cycle of processing these z items, we make three observations. First, the output queue fills $z/(c_i/g_i) = z/(G_{i-1}d_i)$ times, each of which incurs a switch. Second, the input queue empties $z/c_{i-1} = z/(G_{i-1}d_{i-1})$ times, each of which also incurs a switch. Third, only once (after processing all z items) do these two conditions coincide. Hence, the total number of switches S_i away from n_i in one cycle is given by

$$S_i = \frac{z}{G_{i-1}d_{i-1}} + \frac{z}{G_{i-1}d_i} - 1.$$

Conclude that over one cycle from the initial state of n_i's queues back to this state, the rate of switches away from n_i per item consumed by it is given by

$$\begin{aligned} R_i &= S_i/z \\ &= \frac{1}{G_{i-1}d_{i-1}} + \frac{1}{G_{i-1}d_i} - \frac{1}{b} \\ &= \frac{1}{G_{i-1}}\left[\frac{1}{d_{i-1}} + \frac{1}{d_i} - \frac{1}{lcm(d_{i-1}, d_i)}\right]. \end{aligned}$$

Hence, R_i is also the asymptotic switching rate observed for n_i over an unbounded number of inputs to it. □

Combining the results of Lemma 5.1 over all nodes in the pipeline and simplifying, we obtain that

Corollary 5.1.1. *The total rate R of switches across all pipeline nodes per input consumed by n_1 is given by*

$$R = \sum_{i=1}^{h-1} \frac{2}{d_i} - \sum_{i=2}^{h-1} \frac{1}{\mathrm{lcm}(d_{i-1}, d_i)}.$$

5.2. Allocating queue space to minimize switches

We now consider how to minimize the rate of switches R, and therefore the scheduling overhead, incurred by an application through manipulation of its relative queue sizes. Suppose that the items output by node n_i each have size b_i bytes, and that we wish to partition a fixed total number of bytes T among all queues in the pipeline. How can we divide these T bytes among the queues $q_1 \ldots q_{h-1}$ so as to minimize the switching rate R? We could attempt to optimize the switching rate by directly minimizing the function R subject to the constraint $\sum_i b_i c_i = \sum_i b_i G_i d_i = T$. Unfortunately, the presence of LCM terms in R makes it difficult to minimize analytically.

We argue informally that the objective R can be simplified in practice. The LCM terms arise because the number of switches away from node n_i includes a correction of -1 switch per $z = \mathrm{lcm}(c_{i-1}, c_i/g_i)$ inputs. This correction reflects the fact that, in the mean-value model, the input queue empties and the output queue fills simultaneously once per z items. We call such doubly-motivated switches *resonant*. In fact, the actual frequency of resonant switches is likely to be lower than $1/z$, even under the best *achievable* set of c_i's, for two reasons. First, the optimal rational-valued queue sizes for the mean-value model may not be integer numbers of bytes. When we round these sizes to the nearest integer, we will likely increase the LCMs between adjacent sizes and so reduce the frequency of resonances. Second, any random variation in the number of outputs per input produced by the node will likely advance or retard the filling of q_i relative to the emptying of q_{i-1}, turning one resonant switch into two ordinary ones.

If we assume that the frequency of resonant switches is negligible compared to non-resonant switches, then we may eliminate the LCM terms entirely, leaving the objective as

$$R' = \sum_{i=1}^{m-1} \frac{2}{d_i}$$

subject to the same constraint. R' is an upper bound on the true switching rate R that we seek to minimize, and it can be shown to be at most twice R. Empirically, we found that over a large number of different combinations of gains, R' overestimates R by only 10%–20%, so seeking to minimize R' rather than the actual R is still likely to be productive as an optimization strategy.

Replacing R by R' yields a much more tractable constrained optimization problem, which can be solved analytically over the reals by the method of Lagrange multipliers. It can thereby be shown that

Lemma 5.2. *The real-valued choice of queue sizes that minimizes R' subject to $\sum_i b_i c_i = T$ and $c_i \geq 0$ is given by*

$$c_i = \sqrt{\frac{G_i}{b_i}} \cdot \frac{T}{\sum_{j=1}^{h-1} \sqrt{b_j G_j}}.$$

5.3. Practical considerations: Rounding and safety

While Lemma 5.2 gives optimal real-valued queue sizes, real queues must hold an integer number of items. We must therefore round the obtained c_i values to integers, which potentially degrades performance relative to the optimum. Moreover, safety considerations dictate a minimum allowable size for each queue. The queue q_i downstream of n_i must be large enough to hold all the output produced by consuming one vector of input, which could in the worst case be $u_i v$ items. A smaller queue would be unsafe, as there would be nowhere to write output in the worst case.

We may address the safety concern either by reducing the effective vector width v for node n_i, which allows a smaller queue size but reduces available parallelism, or by raising c_i to at least the minimum safe size. We take the latter approach here, leaving the former for future work. While inflating queue sizes to ensure safety is straightforward, it can result in sizes that deviate substantially from the predicted optima. This effect is particularly pronounced when the total available space T for all queues is small or when the gain limit u_i associated with a node is greater than 1.

6. Determining when to use queues

We next describe a performance model and strategy to suggest when to insert queues between nodes of an application to maximize performance. While queuing on an edge between nodes improves SIMD occupancy by remapping data items among SIMD lanes, such remapping adds overhead through queue reads and writes and additional work for the application scheduler. Moreover, remapping is not necessary for correct execution. Given nodes n_i and n_{i+1} of a pipeline, we could remove the queue between them, so that n_i simply calls n_{i+1} with whatever outputs it produced in each SIMD lane without remapping.

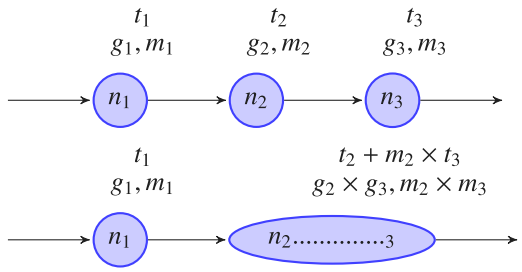

Fig. 4. Topological view of merging compute nodes. Combining nodes n_2 and n_3 is expected to incur approximately m_2 separate calls to n_3 per call to n_2 because inputs to n_3 are no longer queued. The output gains are then estimated as the cumulative gains of the two combined nodes.

(If n_i produces q outputs in a lane, they are queued in a per-lane array, and n_{i+1} must then be called q times to consume them all.) This alternative design effectively *merges* n_i and n_{i+1}. Fig. 4 illustrates the merge operation on the last two nodes of the pipeline from Fig. 1.

Merging has the advantage of no queuing or remapping overhead between the nodes, and nodes n_i and n_{i+1} may be scheduled as a unit, reducing the total number of switches executed by the scheduler. However, we lose the benefits of remapping for SIMD occupancy, so that it may be necessary to call n_{i+1} more often (typically, m_i times per call to n_i) than if we had compacted the outputs of n_i into full vectors. The decision of whether or not to merge therefore involves a tradeoff of occupancy against overhead. We now investigate how to decide quantitatively which pairs of adjacent nodes in a pipeline should have queues on their intervening edges, and which should be merged, to maximize application throughput.

Let $n_1 \ldots n_h$ be a pipeline of nodes of common vector width v, with n_i having average output gain g_i, maximum vector gain m_i, and service time t_i. For convenience, we expand the definition of cumulative average gain G_k and cumulative maximum vector gain M_k to apply to any contiguous subrange of nodes in the pipeline. Define the cumulative gain $G_{j,k}$ between nodes j and k by $G_{j,k} = \prod_{i=j}^{k} g_i$. By this definition, $G_k = G_{1,k}$, and we define $G_{j,j-1} = 1$. Similarly, $M_{j,k} = \prod_{i=j}^{k} m_i$.

We first estimate the service time of a merged node n_{jk} composed of contiguous nodes $n_j \ldots n_k$. When the merged node consumes one vector of inputs, n_j runs first, taking time t_j. Then, node n_{j+1} runs enough times to consume the maximum number of outputs produced by n_j in any SIMD lane. Similarly, node n_{j+2} then runs often enough to consume the maximum number of outputs from n_{j+1} in any lane, and so on through node n_k. An accurate estimate of average service time for the merged node requires knowing the full distributions of the gains of each n_i, so we use the maximum vector gain to estimate a typical running time for the merged nodes. The service time t_{jk} of n_{jk} is estimated as

$$t_{jk} = \sum_{i=j}^{k} M_{j,i-1} t_i.$$

Now suppose we insert a queue between original nodes n_i and n_{i+1}, $j \le i < k$, in the merged node, creating sub-nodes n_{ji} and $n_{i+1,k}$. Because the stream is remapped after node i, the number of times $n_{i+1,k}$ must execute is no longer tied to the number of executions of n_{ji}. Rather, it depends on the total number of *outputs* produced by n_{ji}. The average number of outputs per input vector to n_{ji} is just $G_{j,i}$. We additionally charge a fixed time overhead p_i for each vector of input consumed by n_{ji} to account for the overhead costs associated with this node, in particular the costs of maintaining its output queue and scheduling its execution. Hence, the total running time of the node pair per input vector to n_j is now

$$t_{ji} + p_i + G_{j,i} t_{i+1,k}.$$

We can use this performance model to compare the anticipated costs of merged versus unmerged implementations of any part of a pipeline. For example, we could consider whether to merge each adjacent pair of nodes. More generally, we may consider merging any contiguous subsequences of nodes; however, merging multiple nodes with gain limits > 1 may result in a very large gain limit for the combined node and hence may require excessive memory usage to ensure safe execution. For pipelines with small numbers of nodes, we can efficiently enumerate all feasible merging strategies and identify those predicted to have the lowest cost. To accommodate the limitations of the model, we may wish to empirically test several of the most promising strategies and choose the one with the best empirical performance.

7. Empirical evaluation

We tested our queue sizing and queue placement optimizations on irregular streaming applications implemented in Mercator. Applications were benchmarked on an NVIDIA RTX 2080 GPU with 46 streaming multiprocessors using CUDA 11.2 under Linux. With this configuration, full utilization of the GPU (as recommended by NVIDIA's runtime API) entailed creating several hundred blocks, each with one replica of the application pipeline. Limitations on the number of blocks created were dictated by register usage, which was capped to at most 32 registers per thread.

We measured the gain and running time behaviors of our test applications using a representative data set for each. We obtain the *average and maximum vector gains*, *average compute node running time*, and *average queue overhead time* based on this input data set. We also measured the cost of freeing and reallocating each application's queues to investigate the feasibility of dynamic resizing operations. Depending on the number of queues, we observed costs of approximately 10–30 ms. Hence, we anticipate that queue resizing optimizations could feasibly be performed as often as every few hundred milliseconds in response to changing characteristics of the data stream. For this work, however, we computed queue sizes and merging strategies once based on statistics gathered from the entire data stream for each application.

7.1. Benchmark applications

BLAST. Our BLAST benchmark implements the seed matching and ungapped extension stages of NCBI BLASTN [1], a tool for searching genomic DNA sequence databases. Its input stream is a list of positions in a DNA database, each of which is compared to a shorter *query* DNA sequence to detect approximate matches to parts of the query. Most stages of BLAST filter their inputs, producing fewer than one output per input; however, one stage, which enumerates locations in the query that could potentially match a location in the database, can produce up to 16 locations per input.

We tested BLAST by comparing a query sequence of 30 K DNA bases from the *Salmonella* genome to a database of 6.4 billion bases (compressed to 1.6 GB) built from multiple copies of the human genome (NCBI assembly HG38). Using 368 blocks at 128 CUDA threads/block, a full comparison requires 260–800 ms on our hardware.

N-Queens. The N-Queens benchmark enumerates all valid solutions to the problem of placing N queens on an $N \times N$ chessboard, such that no two queens share a row, column, diagonal, or antidiagonal. This well-studied computational problem, originally posed by Gauss, can be solved using a *branching tree search*, in which the ith level of branching places a queen at each feasible location on row i. We implement the search as a pipeline of N nodes, where node i accepts a partial solution containing queens on the first $i-1$ rows and enumerates feasible placements for the ith row, each of which produces a new partial solution for the next node. Node i can produce up to $N - i + 1$ outputs per input, though not all placements may be feasible for each input due to diagonal and antidiagonal conflicts.

We tested N-Queens for $N = 18$, which produces around 10^8 solutions. The first four stages are computed on the host processor to

generate sufficient inputs (a few tens of thousands of partial solutions) to the next stage to keep all GPU processors occupied. Using 368 blocks at 128 CUDA threads/block, the full computation requires around 23 s on our hardware.

Taxi. The Taxi benchmark is a parsing application drawn from the DIBS data-integration benchmark set [21]. Its input is a file of lines of text, each of which contains a variable number (from 2 to 100+) of pairs of real-valued coordinates. The application is given the starting location of each line and must parse all lines' coordinate pairs, tag each pair with its source line, and emit a stream of tagged coordinate pairs, each as a pair of single-precision floating-point numbers. We implement this application as a pipeline in three stages: the first stage enumerates the character positions on each line; the second identifies the locations of coordinate pairs within the line; and the third parses the pairs and emits them in the proper form. The application takes advantage of Mercator's stateful execution capabilities [20] to allow the last two stages to run in the context of the current line, rather than tagging each item in each intermediate queue.

We tested Taxi on a file of 2 GB containing approximately 1.3 million lines with an average of 45 coordinate pairs per line. Using 460 blocks at 96 CUDA threads/block, the full computation requires 150–250 ms on our hardware.

7.2. Queue sizing optimization

For each of our test applications, we compared its performance with an equal distribution of memory among all queues vs. the unequal distribution recommended by Lemma 5.2 given the average gains for each node observed on our benchmark datasets. We measured the number of switches between nodes and the time to complete a full execution using CUDA's recommended number of blocks.

For BLAST and Taxi, we allocated a total of between 32 KB and 256 KB of memory to queue space per instance of the application pipeline. These applications have large inputs (and for Taxi, large outputs); devoting excessive space to queues would limit the size of the input stream that can be processed without additional host-GPU communication. In contrast, for N-Queens, the input and output stream sizes are comparatively small, so we devoted much more memory to queues — an order of magnitude more than for the other two benchmarks. We report total queue space instead of per-pipeline space for N-Queens to simplify the axis labels in our figures.

Figs. 5, 6, and 7 show the impact of queue space redistribution on the number of switches between nodes. The number of switches is expected to scale inversely with the total amount of queue space used, and this is indeed what we observed. For all applications at nearly all sizes, redistributing the queue space as dictated by our optimization does indeed result in fewer switches than an equal distribution, often reducing switches by 50% or more. The effect is most pronounced when the total memory allocated to queues is smallest, corresponding to the highest absolute numbers of switches.

These results reflect the impact of inflating some queue sizes to the minimum safe size for each node. For our benchmarks, we adjusted queue sizes to ensure that, even after rounding, the total allocation of queue space per pipeline remained the same for equal and redistributed runs. For BLAST, allocating <80 KB per queue with an equal distribution of space among queues, or <192 KB after redistribution, incurs inflation to ensure safety, particularly after the second pipeline stage (which has $u_i = 16$). For N-Queens, inflation occurs at all allocations tested, albeit a relatively small amount (3%–10%). In contrast, Taxi did not incur inflation. For the most part, the beneficial impact of redistributing queue space was still manifest even after adjusting for safety; the only exception is at the smallest allocation for BLAST.

Figs. 8, 9, and 10 show the impact of queue space redistribution on overall execution time. Qualitatively, we see improvements at all queue sizes after redistribution. The magnitude of improvement diminishes as total queue memory grows and the fraction of execution time attributable to scheduler overhead decreases. We note that the point of diminishing returns is reached sooner (i.e., for smaller total memory allocations to queues) after redistribution, which more quickly reduces the absolute number of switches compared to the equal allocation.

Table 1
Compute Node Analysis of BLAST (192 KB equal). Times are measured in GPU cycles/input vector.

Compute Node	g_i	m_i	t_i	p_i
Seed Match	0.379	1	0.23	0.01
Seed Enumeration	1.920	5	0.70	0.03
Small Extension	0.0331	1	0.24	0.1
Ungapped Extension	9×10^{-6}	1	2.47	0.01

Table 2
Predicted (cycles/input vector) vs. Empirical (ms) Results for Different Node Merging Strategies in BLAST. Queue space allocation is 192 KB/pipeline. For each strategy, "+" indicates that adjacent nodes were merged, while a comma indicates that a queue was inserted after a node. Empirical timings were repeatable to within 1–3 ms.

Merging Strategy	Model Result (Equal)	Empirical Result (Equal)	Model Result (Redistributed)	Empirical Result (Redistributed)
1,2,3,4	**0.758**	289	**0.728**	269
1,2+3,4	1.023	**227**	0.993	**231**
1+2,3+4	2.934	331	2.813	329
1+2,3,4	1.198	284	1.161	278
1,2,3+4	2.494	325	2.380	309
1+2+3,4	2.194	250	2.142	258
1,2+3+4	5.642	365	5.359	703
1+2+3+4	14.468	447	13.739	1358

7.3. Node merging optimization

We studied the impact of node merging on the BLAST and NQueens applications. We did not test merging optimizations on Taxi because, for correctness purposes, its design introduces internal barriers between computations for successive lines of input. Hence, even without merging, the application is frequently forced to run with non-full vectors, which is not yet well-modeled by our performance model. To parameterize our performance model, we used our benchmark computations to measure the average gain g_i and maximum vector gain m_i out of each pipeline stage as well as the time spent executing each stage, which we divided into *service time* (time spent executing user code and writing output) and *overhead* (time spent setting up and tearing down execution of each node each time it is called by the scheduler, as well as time spent in the scheduler itself selecting the next node to fire). We computed the t_i values for the model using average service times and computed the p_i values by summing all the overhead time observed for the application, then allocating among nodes proportionally to the number of vectors of input processed by each node. Both t_i and p_i are in units of processor cycles per vector of input to the node.

Table 1 shows and example of the resulting parameter values using 192 KB per pipeline, distributed equally among queues. We computed similar parameters for BLAST with queues redistributed as described in the previous section, and for NQueens with and without redistribution.

We used our model to assess the expected impact of merging contiguous subsets of adjacent nodes in the BLAST pipeline versus leaving all four nodes separate. For both equal and redistributed queues, the model's top two choices were first, to leave all nodes unmerged, and second, to merge the middle two nodes (seed enumeration and ungapped extension). Other strategies are predicted to be progressively worse. Intuitively, reviewing Table 1 suggests that of the possible mergers of adjacent nodes, the middle merger is likely to be better than merging the last two nodes (where the gap between average and max vector gain is amplified by the high cost of ungapped extension), and perhaps better than merging the first two nodes (where the gap is again amplified by the cost of seed enumeration). Similar observations apply

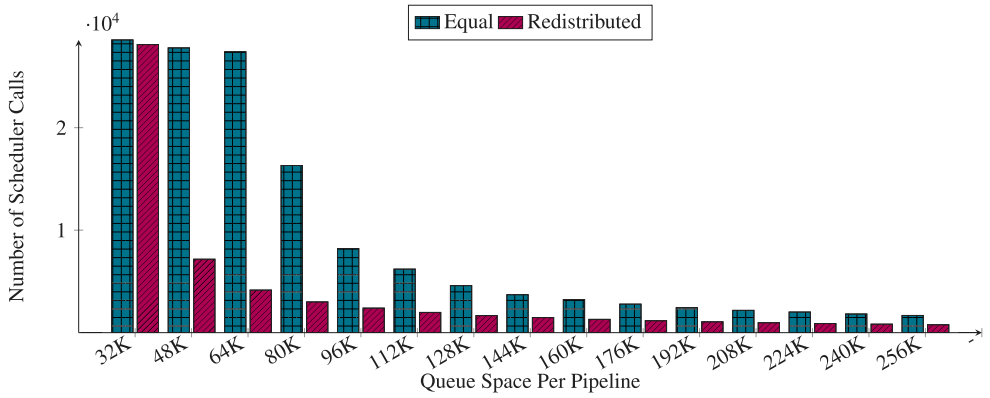

Fig. 5. Number of calls to scheduler for BLAST, averaged over 50 trials.

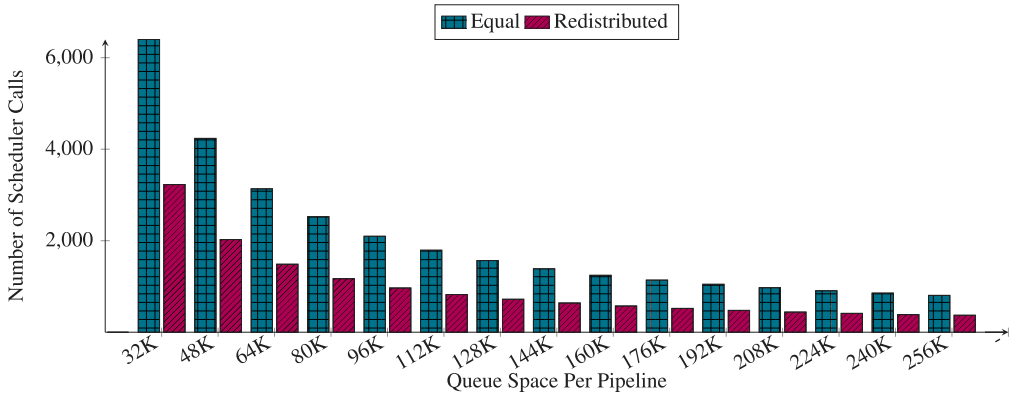

Fig. 6. Number of calls to scheduler for Taxi, averaged over 50 trials.

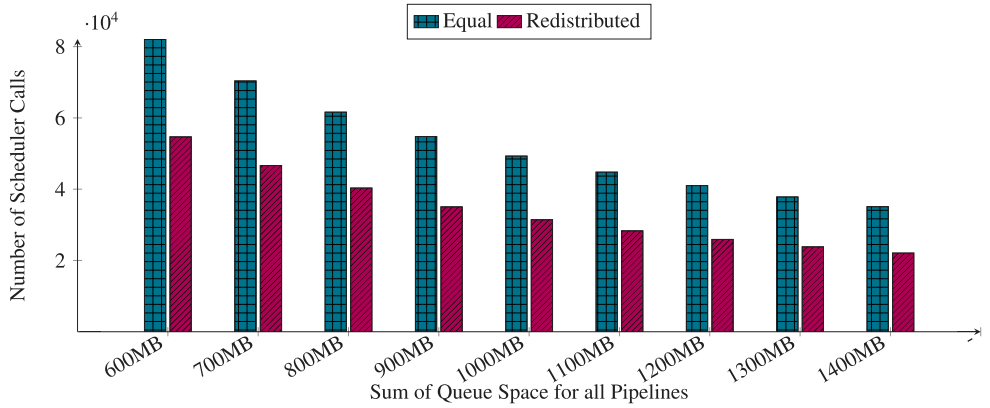

Fig. 7. Number of calls to scheduler for NQueens, averaged over 50 trials.

if we perform queue redistribution as well as merging, which changes the times but not the gains.

Table 2 compares the model's predictions to empirically measured running times for BLAST in its various merged configurations. The empirically optimal strategy was among the model's top two choices, though the model incorrectly predicted an unmerged implementation to be faster. Similar behavior was observed for both equal and redistributed queue sizes. In general, the model underestimated the benefits of merging adjacent nodes but was able to eliminate empirically bad strategies, in particular those involving the merger of the last two nodes.

Table 3 shows predictions and empirical results for several merging strategies for NQueens, this time allocating a total of 1000 MB of queue space to all pipelines. In this case, the model again predicted that a fully unmerged strategy was most efficient among all possibilities, but merging of the last two nodes was empirically faster; the best empirical solution found was again among the top two predictions. Merging additional node pairs concurrently with (and independently of) the last pair produced empirically worse results, suggesting that only pairs of stages close to the end of the pipeline, which run least frequently, are likely beneficial to merge. Moreover, the magnitude of the benefit over the unmerged implementation is small compared to the performance losses incurred when merging earlier pairs. While many more possible merging strategies exist beyond those shown in the table, the large gain limit of most stages of the pipeline($15 - i$ for node n_i, which was also its empirically observed max vector gain) mean that the amount of memory needed to implement most of these strategies safely was infeasible for our target GPU.

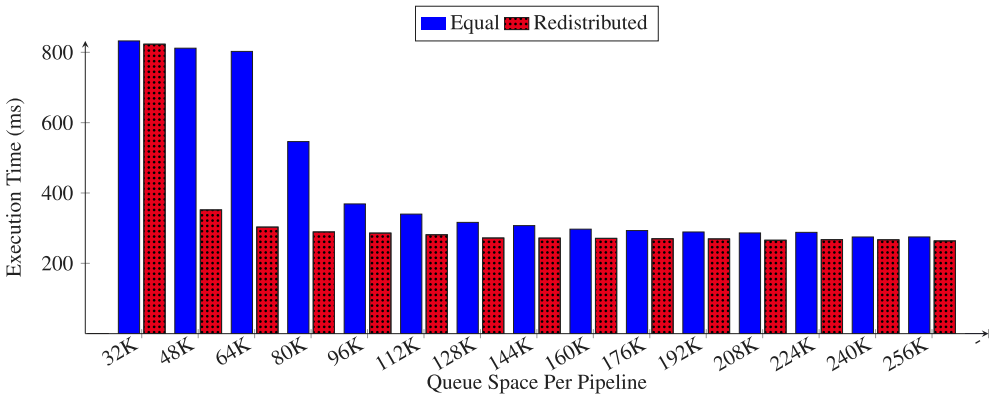

Fig. 8. Total execution time for equal vs. redistributed queue space on the BLAST application, averaged over 50 trials.

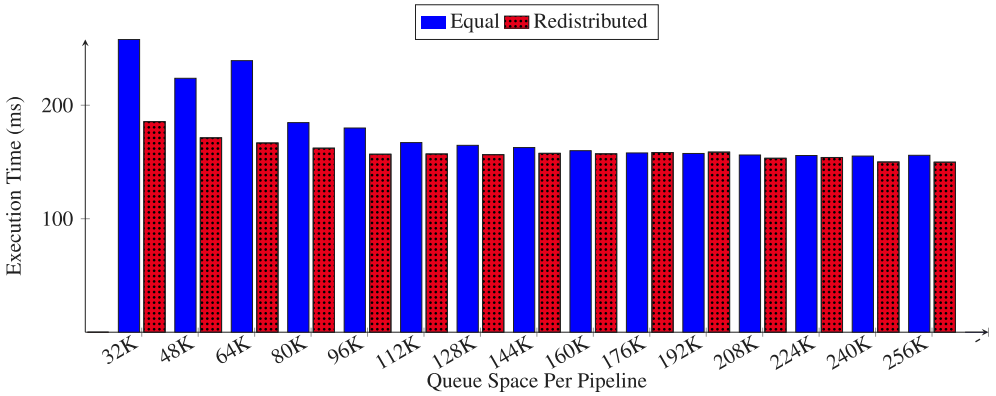

Fig. 9. Total execution time for equal vs. redistributed queue space on Taxi, averaged over 50 trials.

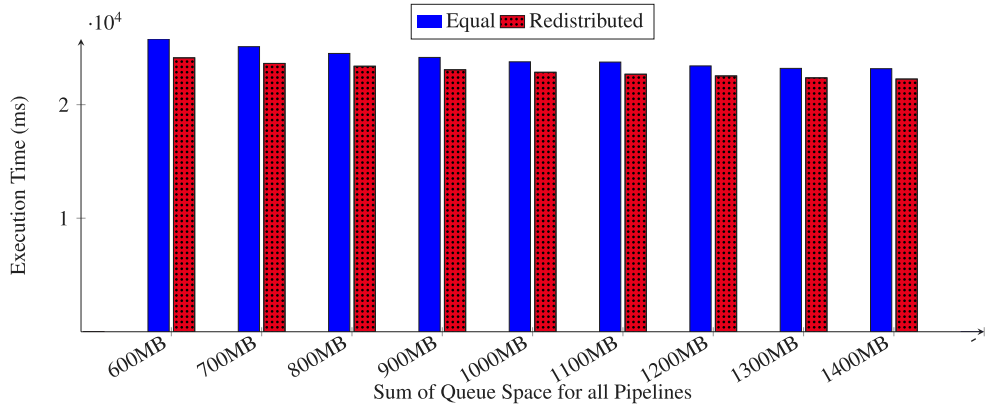

Fig. 10. Total execution time for equal vs. redistributed queue space on NQueens, averaged over 50 trials.

Overall, while our performance model was not perfectly accurate in ordering different merging strategies according to empirical running time, it did rank strategies with the best empirical performance highly among its predictions. Better modeling of the costs of merging – for example, the potential savings when two CUDA functions are merged due to register reuse, common subexpressions, and so forth – could help to better reorder the top candidates. However, our model already shows promise as a tool to guide design space search among different pipeline merging strategies.

8. Conclusions and future work

In this work, we explored how to optimize irregular streaming dataflow applications on SIMD processors by controlling the placement and sizes of inter-node queues. We first devised a technique for choosing the relative sizes of queues given an overall storage budget for the pipeline. We then developed a performance model to inform where to insert queues in an application. Both optimizations were driven by profile data on the output behavior of each node in the application; the queue insertion model also utilized empirical measurements of service times and overhead. Both techniques targeted the cost of scheduling multiple nodes of an application on a single processor. Each provided demonstrated benefits in selecting configurations with lower execution time, with queue size optimization proving the more robust of the two.

Future work will examine a broader set of irregular applications and a larger variety of representative data sets. Characterization of these applications' structures will aid in development decisions for queue placement and allocation. We will consider alternative strategies

Table 3
Predicted (cycles/input vector) vs. Empirical (ms) Results for Different Node Merging Strategies in NQueens. Queue space allocation is 1000 MB for all pipelines. For each strategy, "+" indicates that adjacent nodes were merged, while a comma indicates that a queue was inserted after a node. Empirical timings were repeatable to within 90–130 ms.

Merging Strategy	Model Result (Equal)	Empirical Result (Equal)	Model Result (Redistributed)	Empirical Result (Redistributed)
1,2,3,4,5,6,7,8,9,10,11,12,13,14	$\mathbf{1.707 \times 10^6}$	23774	$\mathbf{1.616 \times 10^6}$	22851
1,2,3,4,5,6,7,8,9,10,11,12,13+14	1.777×10^6	**23108**	1.684×10^6	**22082**
1,2,3,4,5,6,7,8,9,10,11+12,13+14	2.746×10^6	28895	2.606×10^6	27682
1,2,3,4,5,6,7,8,9+10,11+12,13+14	3.694×10^6	35394	3.511×10^6	33456

for dealing with safety constraints on queue size that may be less prejudicial to performance at small queue sizes. More broadly, we will consider whether more detailed information about output gain – in particular additional moments beyond the average and mode of max vector gain per node – could result in more accurate decision making, particularly in node merging. We will also consider whether it is possible to more accurately predict the impact of merging nodes on their combined service time, which we suspect is an important phenomenon in determining throughput.

This work provides a framework for deciding the *relative* queue sizes of an application but does not address how much *absolute* queue space to allocate an application. Currently, we test a broad range of absolute queue sizes, but do not have a particular method for determining which to use. We have shown that there is a correlation between larger absolute queue sizes and faster running times. However, these faster running times come at the cost of a larger memory footprint for infrastructure and could quickly balloon out of control considering each block has its own set of queues, leaving little to no room for input and application data. For future work, analysis of when larger absolute queue sizes provide diminishing returns on running time, as well as modeling the tradeoff between larger queues and the need to process smaller chunks of input due to GPU global memory limitations, will provide guidance on how much total queue space should be given to an application.

Mercator supports efficient implementation of irregular streaming applications like those described here, including branching searches such as NQueens. We plan to continue extending the system to support broader classes of application, such as those whose topologies include DAGs and cycles. The semantics of such topologies for irregular dataflow are not entirely clear; [22] offers one possible set of semantics that lead to nontrivial safety and efficiency challenges.

Finally, we plan to automate the process of profile gathering and profile-guided optimization for Mercator applications. The execution statistics supporting our optimizations are straightforward to collect. In the near term, we will offer optimization guided by execution profiles through recompilation of the application. In the longer term, it should be possible for a long-running application (on the order of a second or more, based on our empirical measurements of reallocation costs) to dynamically reconfigure its pipeline during execution based on observed behaviors. Such automated tuning would allow an application to respond to local variations in the properties of a long input stream and thereby optimize overall application performance.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by National Science Foundation, USA awards CNS-1763503 and CNS-1500173.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, Basic local alignment search tool, J. Mol. Biol. 215 (3) (1990) 403–410.
[2] M. Roesch, Snort - lightweight intrusion detection for networks, in: Proceedings of the 13th USENIX Conference on System Administration, LISA '99, USENIX Association, USA, 1999, pp. 229–238.
[3] E. Tyson, J. Buckley, M. Franklin, R. Chamberlain, Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system, Nucl. Instrum. Methods Phys. Res. A 595 (2008) 474–479.
[4] W. Thies, M. Karczmarek, S. Amarasinghe, StreamIt: A language for streaming applications, in: R.N. Horspool (Ed.), Compiler Construction, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 179–196.
[5] T. Plano, J. Buhler, Scheduling irregular dataflow pipelines on SIMD architectures, in: Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing, WPMVP'20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–9.
[6] E. Lee, D. Messerschmitt, Synchronous data flow, Proc. IEEE 75 (1987) 1235–1245.
[7] J. Subhlok, G. Vondran, Optimal latency–throughput tradeoffs for data parallel pipelines, in: 8th Annual ACM Symposium on Parallel Algorithms and Architectures, 1997, pp. 62–71.
[8] A. Benoit, Y. Robert, Complexity results for throughput and latency optimization of replicated and data-parallel workflows, Algorithmica 57 (4) (2010) 689–724.
[9] S.V. Cole, J. Buhler, MERCATOR: A GPGPU framework for irregular streaming applications, in: 2017 International Conference on High Performance Computing Simulation, HPCS, 2017, pp. 727–736.
[10] S.V. Cole, Efficiently and Transparently Maintaining High SIMD Occupancy in the Presence of Wavefront Irregularity (Ph.D. thesis), Dept. of Computer Science and Engineering, Washington University in St. Louis, 2017.
[11] M. Burtscher, R. Nasre, K. Pingali, A quantitative study of irregular programs on GPUs, in: Proc. 2012 IEEE Int'l Symp. on Workload Characterization, 2012, pp. 141–151.
[12] K. Gupta, J.A. Stuart, J.D. Owens, A study of persistent threads style GPU programming for GPGPU workloads, in: Innovative Parallel Computing, IEEE InPar, 2012, pp. 1–14.
[13] P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: Proc. 14th Int'l Conf. on High Performance Computing, 2007, pp. 197–208.
[14] L. Luo, M. Wong, W.-M. Hwu, An effective GPU implementation of breadth-first search, in: Proc. 47th Design Automation Conference, 2010, pp. 52–55.
[15] D. Merrill, M. Garland, A. Grimshaw, Scalable GPU graph traversal, in: Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, 2012, pp. 117–127.
[16] R. Nasre, M. Burtscher, K. Pingali, Morph algorithms on GPUs, in: Proc. 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 2013, pp. 147–156.
[17] M. Burtscher, K. Pingali, An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm, in: GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011, pp. 75–92.
[18] M. Mendez-Lojo, M. Burtscher, K. Pingali, A GPU implementation of inclusion-based points-to analysis, in: Proc. 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 2012, pp. 107–116.
[19] S. Tzeng, A. Patney, J.D. Owens, Task management for irregular-parallel workloads on the GPU, in: Proc. 2010 Conf. on High Performance Graphics, 2010, pp. 29–37.
[20] S. Timcheck, J. Buhler, Streaming computations with region-based state on SIMD architectures, in: 13th Int'l Wkshp. on Programmability and Architectures for Heterogeneous Multicores, 2020, p. 1.
[21] A.M. Cabrera, C.J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R.K. Cytron, R.D. Chamberlain, DIBS: A data integration benchmark suite, in: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 25–28.
[22] P. Li, J. Beard, J. Buhler, Deadlock-free buffer configuration for stream computing, in: 2015 Int'l Workshop on Programming Models and Applications for Multicores and Manycores, 2015, pp. 164–169.