International Journal of Parallel Programming (2023) 51:43–60
https://doi.org/10.1007/s10766-022-00745-2

Interruptible Nodes: Reducing Queueing Costs in Irregular Streaming Dataflow Applications on Wide-SIMD Architectures

Stephen Timcheck[1] · Jeremy Buhler[1]

Received: 9 September 2022 / Accepted: 21 November 2022 / Published online: 5 December 2022
© The Author(s) 2022

Abstract

Streaming dataflow applications are an attractive target to parallelize on wide-SIMD processors such as GPUs. These applications can be expressed as a pipeline of compute nodes connected by edges, which feed outputs from one node to the next. Streaming applications often exhibit irregular dataflow, where the amount of output produced for one input is unknown *a priori*. Inserting finite queues between pipeline nodes can ameliorate the impact of irregularity and improve SIMD lane occupancy. The sizing of these queues is driven by both performance and safety considerations- relative queue sizes should be chosen to reduce runtime overhead and maximize throughput, but each node's output queue must be large enough to accommodate the maximum number of outputs produced by one SIMD vector of inputs to the node. When safety and performance considerations conflict, the application may incur excessive memory usage and runtime overhead. In this work, we identify properties of applications that lead to such undesirable behaviors, with examples from applications implemented in our MERCATOR framework for irregular streaming on GPUs. To address these issues, we propose extensions to support *interruptible nodes* that can be suspended mid-execution if their output queues fill. We illustrate the impacts of adding interruptible nodes to the MERCATOR framework on representative irregular streaming applications from the domains of branching search and bioinformatics.

Keywords Irregular · Dataflow · Streaming · Queueing · Interrupt · SIMD

✉ Stephen Timcheck
stimcheck@wustl.edu

Jeremy Buhler
jbuhler@wustl.edu

[1] McKelvey School of Engineering: Department of Computer Science and Engineering, Washington University in St. Louis, One Brookings Drive, St. Louis 63105, Missouri, USA

1 Introduction

Streaming computations arise in numerous domains, including bioinformatics [1], astrophysics [2], data integration [3], network packet inspection [4], branch-and-bound search [5], and decision cascades in machine learning [6]. Applications of this type process a long stream of independent data items, which makes them amenable to running on wide-SIMD devices such as GPUs. Streaming applications can be expressed as a pipeline of *compute nodes* connected by *edges*, where incoming data is processed by a node and then sent to the next node via a connecting edge. High-level programming support for streaming is important to manage node execution and data storage on edges transparently to the application designer; in this work, such support is provided by MERCATOR [7, 8], a framework we previously constructed to support streaming applications on GPUs.

Many streaming applications of interest — including those mentioned above — exhibit *irregular dataflow*. In an irregular application, the number of output data items produced per input to a node is not fixed *a priori* but rather varies dynamically and unpredictably for each data item. Irregular dataflow means that data must be queued on edges between nodes, both for parallelism (i.e., accumulating a full SIMD vector of inputs to the next node) and for safety if a node generates more output than the next node can consume at once. The sizes of inter-node queues should be carefully chosen based on considerations of average and worst-case node behavior [8], and scheduling of an application's nodes must then be cognizant of inter-node queue occupancy to ensure safe and efficient execution [9].

A basic property of streaming pipelines in MERCATOR is that execution of a compute node is *uninterruptible*: once the node begins to consume a SIMD vector of inputs, it must finish before another node can be scheduled to execute. This behavior arises because existing GPU runtimes do not support preemptive scheduling of compute kernels or of functions within one kernel. As a result, for each node, there is a minimum safe size for its output queue, namely the space needed to hold the most output that could be generated by one vector of input. This safety constraint must override queue-sizing decisions driven by average-case performance considerations, which can result in applications that allocate much more queue space than they typically use and that may be inefficiently scheduled.

In this work, we first identify performance and memory usage problems that appear in irregular streaming applications due to safety constraints arising from uninterruptible nodes. We then describe modifications to our MERCATOR framework that enable application programmers to cooperatively support suspension and resumption of a node, which requires saving and restoring its execution state. Finally, we benchmark some representative applications to evaluate the impact of interruptibility on application performance and memory usage.

The rest of the paper is divided as follows. Section 2 examines related work, while Sect. 3 describes our application model. Section 4 looks at examples of irregular streaming applications and the impact minimum queue size restrictions have on their performance and memory usage. Section 5 describes MERCATOR's

new interruptible node facility and the challenges of implementing interruptible nodes. Section 6 empirically evaluates interruptible nodes on two applications, NQueens and BLAST. Finally, Sect. 7 concludes and considers future work.

2 Related Work

Our own prior work on MERCATOR includes the design of its node scheduler [9] and techniques for reducing overhead caused by switching from one pipeline node to another during execution [8]. This work focuses on a mechanism to allow a node to suspend and later resume execution and shows how, in the context of a wide-SIMD execution model for streaming pipelines, such a mechanism can have important benefits for throughput and for memory usage.

Prior work in scheduling multiple tasks on the GPU includes work on cooperative CPU-GPU scheduling. Hyoseung et al. [10] considered a single GPU shared between multiple non-preemptive tasks that must be scheduled sequentially. The CPU determines which GPU task to run at any given time. Kato et al.'s TimeGraph system [11] similarly includes a CPU-side scheduling mechanism for GPU tasks, each of which may have multiple components, and can make scheduling decisions based on task priority. MERCATOR also manages multiple non-preemptively executing tasks, in the form of different compute nodes in a pipeline, but the nodes along with their scheduler are all functions within one GPU kernel. Hence, we cannot use the facilities that might be used by CPU-side schedulers, such as multithreading or timer interrupts. Moreover, nodes communicate through the pipeline edges between them, which raises a different set of scheduling considerations than for independent tasks.

Other work has investigated preemptive scheduling of multiple kernels on a shared GPU, which would be desirable for, e.g., GPU virtualization and would also help ameliorate the problems we identify with non-preemptive node execution in MERCATOR. One such system, Chimera [12], assumes the existence of hardware support for kernel preemption (which was simulated using GPGPU-Sim) and focuses on how to lower the throughput and latency impacts of context switching between kernels. While MERCATOR does not consider applications with strong latency constraints, our prior work also focuses on reducing throughput impacts, specifically the frequency of required inter-node switches (which, unlike in the case of independent tasks, are unavoidable for nodes in a streaming pipeline with finite queues). The present work furthers the goal of switching reduction by using node interruptibility to enable optimizations that further reduce switches and so improve throughput.

Much like Chimera, FLEP [13] seeks to enable kernel preemption on the GPU, with a focus on speeding up high-priority kernels as well as fairly distributing time between kernels. The authors design preemption both for the entire GPU and for specific processors on the GPU. Unlike Chimera, FLEP's preemption does not assume hardware support but rather is achieved partly via compiler-side transformations on kernel code, wrapping the kernel's interior with conditions to exit on setting a variable available to the CPU. FLEP further uses timing information

gathered from applications to estimate preemption overhead and guide decisions on when to preempt a kernel. While MERCATOR's scheduling decisions are not motivated by priorities, the present work must also contend with code transformations to enable nodes to suspend at certain strategic points so that another node can run. We presently offer only low-level facilities that enable application developers to write preemptible code manually, but future work will consider the feasibility of automated, higher-level program transformations to support node suspension and resumption.

3 Application Model

In this section, we more formally describe streaming dataflow applications and how we map them onto a wide-SIMD execution platform. Our target for MERCATOR applications is an NVIDIA GPU running applications written in the CUDA language; however, this platform's properties and limitations are typical of other wide-SIMD targets such as AMD GPUs running OpenCL. Details of MERCATOR's usage and application mapping beyond those described here may be found in [8].

3.1 Application Mapping

An application is represented as a pipeline of *compute nodes* $n_0, n_1, \ldots$ with successive nodes connected by dataflow *edges* as shown in Fig. 1. Each compute node n_i consumes a vector of up to v inputs at a time and produces a variable number of outputs per input for the downstream node n_{i+1} to process later. The number of outputs produced per input to a node, which we call its *gain*, may vary dynamically in a data-dependent fashion up to some known maximum.

An edge between two nodes has a finite *queue* in which data produced by the upstream node is stored until it can be consumed by the downstream node. We assume that queue sizes are fixed for the duration of an application's execution, or at least for the time needed to process a large number of inputs, due to the high cost of dynamic memory allocation on our target platform. When a node n_i begins to consume input from its upstream queue, it does so in SIMD vectors of up to v items at a time until either its upstream queue empties or it cannot consume another vector of inputs without potentially overflowing the remaining output space in its downstream queue. At that point, n_i must yield control to a global *node scheduler*, which selects

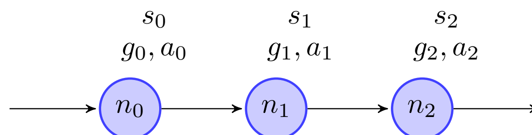

Fig. 1 A simple pipeline application topology. Node n_0 feeds into n_1, and n_1 feeds into n_2. Node n_i has service time s_i, average gain g_i, and maximum gain a_i

other nodes to execute until n_i again has both available input data and available output space.

A GPU platform typically contains multiple processors, each of which may support multiple, asynchronous, non-communicating execution contexts (*GPU blocks* in CUDA). MERCATOR runs an independent replica of the application's pipeline within each context, with all contexts pulling data competitively from a single shared input stream, running asynchronously in parallel, and writing to a single shared output stream. Each context's pipeline replica has its own set of queues and its own scheduler instance that runs nodes *sequentially* within that context. In what follows, we focus on the behavior and memory usage of *one* pipeline replica, which processes SIMD vectors but whose nodes are sequentially scheduled, with the understanding that a GPU executing an application may run (and allocate queue memory for) hundreds of pipeline replicas concurrently.

Finally, we emphasize that node scheduling is *non-preemptive*: once a node starts to consume a vector of inputs, it cannot yield to the scheduler until those inputs have been completely processed and any outputs from them emitted downstream. This lack of preemption is a limitation of our target platform — CUDA does not support preemptive scheduling of different GPU kernels or of different functions within a single GPU kernel. Consequently, a node cannot safely consume a vector of inputs unless it has space for the most output it could possibly generate from these inputs in its downstream queue; otherwise, it might either overrun that queue or deadlock the application because it cannot finish execution.

3.2 Application Performance and Queue Optimizations

A node n_i's behavior is characterized by its *service time* s_i, *average gain* g_i, and *maximum gain* a_i. The service time s_i is the average time n_i takes to process a vector containing between 1 and v inputs, which is assumed to be constant due to the SIMD target architecture. The gain of a node defines how many data items are output (on average for g_i, or in the worst case for a_i) for each item input to n_i. The average number of outputs from n_i per input to the *first* node n_0 in the pipeline is n_i's *average cumulative gain*, computed as $G_i = \prod_{k=0}^{i} g_k$.

Our performance metric of interest for streaming dataflow applications is *throughput*. Throughput depends on the node scheduler, which should schedule nodes so as to ensure that they have full vectors of input ready to consume whenever possible. Moreover, because switching execution between nodes incurs runtime overhead, scheduling should ideally ensure that a node can run for as long as possible before an empty input queue or full output queue requires switching to another node. Irregular dataflow forbids *a priori* computation of a static optimal schedule as in regular streaming dataflow models [14, 15], but MERCATOR uses a scheduling policy, Active-Full/Inactive-Empty (AFIE) [9], that ensures that nodes run with full input vectors and limits inter-node switches to within a small constant factor of the fewest possible even under a clairvoyant schedule which knows in advance how many outputs will be produced by each input to each node.

Even with AFIE scheduling, application throughput is sensitive to the relative sizes of inter-node queues. For example, a node with high average gain benefits from a larger output queue because it can write more outputs before filling the queue and forcing a return to the scheduler. Given a fixed total budget of queue memory for one pipeline replica and the average cumulative gains G_i of each node, one may formulate the problem of how to allocate the budgeted memory among the queues in the pipeline so as to minimize the expected number of times the application must return to the scheduler. By solving this problem analytically, we may perform *queue space distribution* [8] to minimize scheduling overhead.

Finally, it may be that the overhead of queueing and dequeueing data between two adjacent pipeline nodes n_i, n_{i+1} exceeds the benefit to SIMD occupancy obtained by having the queue in the first place. In such cases, it may be better to *merge* the two nodes into one that performs their combined computations for each vector of inputs to n_i. Merging analysis [8] relies on knowledge of both the average gain g_i and of the service time s_i for each node.

4 Impact of Minimum Safe Queue Sizes

The need to enforce minimum safe sizes to accommodate uninterruptible nodes has consequences for application performance and resource utilization. In this section, we identify these consequences and illustrate them through two representative irregular streaming applications from the domains of branching search and bioinformatics. A key feature in these applications is a large gap between a node's average gain, which is most relevant for performance analysis, and its maximum gain, which determines the minimum queue size needed for safety.

4.1 Example Applications

In this section and Sect. 6, we study two irregular streaming applications whose pipelines exhibit the impacts of minimum safe queue sizes: NQueens and BLAST.

4.1.1 NQueens

The NQueens application enumerates all possible ways of placing N queens on an $N \times N$ chess board such that no queen can attack another. The problem can be solved using a *branching search tree*, in which a node at level i of the tree determines all the feasible ways to place a queen on row i of the board given fixed placements of queens on rows $1 \ldots i-1$. Similar branching structures appear in branch-and-bound combinatorial optimization and other tree traversal applications.

To make NQueens a streaming application, we create a pipeline of N nodes, where node n_i enumerates feasible ways to place the $i+1$st queen. An initially empty board is passed to the first node, which outputs N partial boards, each with a possible placement of a queen on the first row. For each such board, the second node places a queen in all feasible ways on the second row and passes the resulting partial boards to the third

node, and so on until all complete feasible boards are enumerated. Node n_i can produce up to $a_i = N - i$ outputs per input, though some of these possibilities are infeasible and so are discarded.

Our benchmark computation enumerates all feasible solutions for $N = 18$. To ensure adequate parallelism to occupy all GPU blocks, we precompute feasible placements of the first four queens on the CPU and pass these partial boards as the input stream to a GPU pipeline of 14 nodes.

We may quantify the irregularity of NQueens as follows. Let γ_i be the total gain observed from one full vector of inputs to node n_i; our implementation uses vectors of size 128. The quantity γ_i is a random variable, for which we may compute the average and standard deviation over many input vectors, and thence its coefficient of variation $C(\gamma_i)$. Larger values of this coefficient indicate greater irregularity from one vector to the next.

We found empirically that $C(\gamma_i)$ ranged from 0.086 for the first node n_1 to 0.279 for the last node n_{14}. In other words, the *typical* variation in total gain from one vector to another is 10–30% of the average gain. Some vectors, and some inputs within a vector, may be expected to exhibit a much larger range of irregularity; in particular, we found that a vector of 128 inputs almost always has at least one input that produces the maximum possible gain, even though the average gain is much less than this maximum. Overall, NQueens instances exhibit substantial irregularity in their branching behavior.

4.1.2 BLAST

The Basic Local Alignment Search Tool for nucleotide sequence [1] performs pattern matching in a large DNA database. A small DNA *query* sequence is compared to the database to identify substrings that match it to within a small edit distance. The application extracts successive substrings of length $k = 8$ from the database and compares them to a hash table of all k-mers in the query; when a k-mer matches, the locations of all matching k-mers in the query are enumerated, and each such match is further verified using a series of increasingly complex filters to retain the small fraction of matches that are biologically significant. BLAST is representative of a large class of decision filter cascades that can be implemented as irregular streaming applications; other examples include Viola-Jones face recognition [6] and Snort packet inspection [4].

The four nodes of the BLAST pipeline check the hash table, enumerate the positions of matches in the query, and implement successive filters. Of particular interest to our work is node n_1, which is responsible for enumerating k-mer matches and can list up to 16 matching query positions for each database position. We test BLAST on a query of length 30 Kbases from a bacterial genome against a database containing two copies of the human genome, equalling 6.4 Gbases.

4.2 Memory Bloat

For a node n_i that processes up to v inputs at once and emits at most a_i outputs per input, the minimum safe size for its output queue is $a_i v + v - 1$ slots [9]. Clearly, $a_i v$ slots are needed to accommodate the node's maximum outputs from one input

vector; the remaining $v - 1$ slots are needed to accommodate a residue from prior runs of less than one full vector-width of items whose consumption may have been deferred in hopes of obtaining a full vector later. In short, the minimum safe queue size scales linearly with a node's maximum gain.

In contrast, the *ideal* queue size for a node is one that minimizes the overhead incurred by the node scheduler, which is proportional to the frequency with which the scheduler must be called to switch between nodes. In [8], we showed that given a fixed total amount of memory devoted to queues, the fraction of that memory that should be allocated to a node's output queue to minimize switching scales roughly as the square root of a node's *average cumulative gain*.

When the minimum safe size for a queue exceeds its ideal size for a given memory budget, we say that the queue is *bloated*. The larger the gap between a queue's average cumulative gain and its (individual) maximum gain, the greater the bloat of its output queue.

As an example, Table 1 illustrates the ideal and safe queue sizes for all 14 nodes of the NQueens application when using SIMD vectors of size 128. Nodes early in the pipeline have large maximum gains and hence large minimum queue sizes, since the branching search for feasible boards at early stages has few constraints. In contrast, the average cumulative gain is *smallest* for nodes early in the pipeline, growing rapidly with greater tree depth except at the highly constrained final stages. Moreover, all nodes individually have average gains less than (and mostly less than half) their maximum gains.

In the table, we consider a total queue memory allocation of 600 MB across 368 pipeline replicas, which was determined to be on the low end of a reasonable total queue memory allocation based on how much memory NQueens uses for input

Table 1 Gains and implied queue sizes of NQueens application with 128-wide SIMD vectors and a target allocation of 600 MB for all queues

Node	Max gain	Avg. (cumulative) gain	safe queue size (items)	Ideal size (items)
0	14	8.87 (8.87)	1919	44
1	13	7.46 (66.13)	1791	120
2	12	6.18 (408.62)	1663	298
3	11	5.12 (2094.02)	1535	674
4	10	4.20 (8792.57)	1407	1381
5	9	3.40 (29853.91)	1279	2545
6	8	2.71 (80990.22)	1151	4191
7	7	2.14 (173673.81)	1023	6138
8	6	1.66 (288936.00)	895	7917
9	5	1.27 (366991.64)	767	8922
10	4	0.94 (344898.36)	639	8649
11	3	0.66 (227664.25)	511	7027
12	2	0.41 (94298.90)	383	4523
13	1	0.19 (18367.82)	255	3992

and output streams and how much memory was available on our GPU. The output queues for nodes early in the pipeline have smaller ideal sizes than their minimum safe sizes and hence are bloated. The queues for nodes 0 and 1 are bloated by more than a factor of ten. The application designer must therefore either increase its memory allocation to accommodate the required bloat or take away memory from later nodes' queues, incurring more scheduling overhead as a result.

The problem of bloat may be exacerbated by optimizations that attempt to merge adjacent nodes. In the case of NQueens, analysis of service times and SIMD occupancy according to [8] suggests that merging nodes 0 and 1 and eliminating the queues between them could be beneficial for throughput. However, merging two nodes with maximum gains a and a' results in a combined node with maximum gain $a \cdot a'$. For this example, the minimum safe size for the merger of nodes 0 and 1 is $14 \cdot 13 \cdot 128 + 127 = 23423$ entries—nearly 200 times the ideal queue size of 120. Adding this space to the ideal allocation shown would increase the application's overall queue memory usage by roughly 40%.

In short, when an application's nodes have a large maximum gain but a small average cumulative gain, the resulting constraint on queue sizes can lead to substantial bloat that increases memory requirements and forces deviation from the throughput-ideal pattern of queue sizing.

4.3 Pessimistic Scheduling Behavior

Even when bloat is not a substantial concern, the disparity between a node's average and maximum gain can incur additional costs to execution. To illustrate the issue, consider the queue allocations for the BLAST application shown in Table 2. The total memory allocation is much smaller than for NQueens (only 32 KB per pipeline, for 368 pipelines) because of the need to reserve as much GPU memory as possible for BLAST's sequence database. Given the application's overall memory budget and SIMD width, the minimum queue sizes do not incur substantial bloat except at the last node; overall, bloat accounts for only a small fraction of the application's overall queue space usage (either ideal or minimum safe).

However, we observe that node n_1, the enumeration node, exhibits a large disparity between its maximum gain (16) and its individual average gain (roughly 2). This node cannot consume a vector of input unless it has at least $16 \cdot 128 = 2048$ slots free in its output queue; otherwise, the vector's output might overrun the queue in

Table 2 Gains and implied queue sizes of BLAST application with 128-wide SIMD vectors and a target allocation of 11.5 MB for all queues

Node	Max gain	Avg. (cumulative) gain	safe queue size (items)	Ideal size (items)
0	1	0.38 (0.38)	255	1552
1	16	1.92 (0.73)	2175	2151
2	1	0.03 (0.02)	255	392
3	1	0.000009 (0.0000002)	255	1

the worst case. However, the node's actual gain averages 246 outputs from one input vector. Hence, after consuming one input vector, the node typically must yield to the scheduler and cannot be run again until its output queue is emptied. Yet the node's queue is large enough to hold more than 8 input vectors' worth of "typical" output!

Hence, even disregarding bloat, a node whose maximum gain greatly exceeds its average gain typically leaves most of its output queue unused. If that space could safely be used without fear of overrun, the node would encounter a full output queue (and hence would need to return the scheduler) much less often.

Both bloat and pessimistic scheduling behavior are driven by nodes with maximum gains that far exceed their average individual or cumulative gains. This disparity is traceable to a basic limitation of our model: *because nodes must process their inputs without interruption*, they need enough space to write the maximum possible amount of output each time they run. In the next section, we describe a method to remove this limitation.

5 Interruptible Nodes

To overcome performance and resource issues caused by large minimum queue sizes, we extend the MERCATOR framework with support for *interruptible nodes*. We first describe the basic idea of interruptible nodes and why they address the problems identified in the previous section, then describe how we implement them given the limitations of our target platform.

5.1 Semantics of Interruptible Nodes

The key observation underlying interruptible nodes is that, while a node may produce > 1 output per item in its input vector, it cannot actually enqueue more than v items (the width of one SIMD vector) at a time. More specifically, a node in a MERCATOR application emits items to its downstream queue by calling a function `push()`, which takes a vector of items and a per-SIMD-lane flag indicating whether each item is valid (and so should be emitted). Because `push()` cannot emit more than v items at once, a node that may emit multiple outputs from a single input must call `push()` multiple times in one run.

```
__device__ void
MyNode::run(int x) {
  i = 0;
  while (i < M) {
    int v = f(x, i);
    push (v, v > 0);
    ++i;
  }
}
```

Fig. 2 A MERCATOR node function that iterates over its input to produce up to M outputs per input item. Although the code appears sequential, it runs concurrently on an entire SIMD vector of inputs, each of which maps to the variable x in a different CUDA thread

As an example, Fig. 2 illustrates CUDA code for a MERCATOR application node `MyNode`. Each SIMD lane of `MyNode` takes an integer input x and produces up to M outputs. The ith potential output for a SIMD lane is computed from the input by a function `f()`, and the result is pushed downstream iff it is a positive value. This node's maximum gain is M, but its average gain depends on the inputs and the properties of the function `f()`.

A single call to `push()` is safe so long as the downstream queue has at least v slots available to receive outputs. This is true no matter how large the node's maximum gain is. Hence, in a node that may call `push()` several times, we wish to make each such call a *yield point* – if the push would fail due to insufficient queue space, the node should be suspended, and control should return to the scheduler. Once sufficient space is available, the node may resume execution from the point of suspension.

Making a node interruptible addresses the performance and resource concerns raised in the previous section. Critically, it is no longer necessary to bloat a node's output queue to accommodate its maximum gain a_i, because the node can be suspended if it would otherwise overrun the queue. The only size constraint on the output queue is that it hold at least $2v - 1$ entries— the minimum needed by the AFIE scheduler for safety given pushes of size up to v items. Moreover, if (as in our BLAST example) a node's output queue size significantly exceeds its vector width times its average gain, the node will likely be able to consume multiple vectors of input without filling the queue and returning to the scheduler. Should the node exhaust its queue space while processing a vector, it can now be suspended and resumed later.

5.2 Implementation Challenges for Interruptibility

A MERCATOR application is specified using a high-level pipeline description that produces a CUDA skeleton, with stub functions for each node that are filled in by the application developer. These functions are compiled together with MERCATOR's node scheduler and other runtime support code to form a single GPU kernel that consumes a stream of inputs stored in GPU global memory. Adding interruptible node semantics to this model is challenging due to the limitations of CUDA and so requires cooperation from the application developer.

Ideally, CUDA device code would support a facility for saving execution state in a way that can be resumed later, analogous to `setjmp/longjmp` in C or continuations in functional languages. In the absence of such a facility, we chose to provide a minimal set of extensions to let an application recognize when node suspension is required and communicate a decision to suspend to the MERCATOR runtime. The application developer then implements state saving and restoring as part of the node's code.

We extended MERCATOR's runtime in two ways. First, the `push()` function now returns a boolean value to indicate if the *next* call to `push()` might fail due to insufficient (i.e., $< v$ items) downstream queue space. Second, a node now returns a boolean value to the MERCATOR runtime to indicate whether it

finished processing its input vector (and so can immediately be run again with another vector if one is available) or had to suspend in the middle of processing a vector. In the latter case, when a node resumes after interruption, the runtime will invoke it with the *same* input vector that it was processing when it suspended execution. MERCATOR guarantees that a suspended node will not be called again until it can successfully complete at least one push operation of up to v items and so make progress.

The application developer's code for a node is responsible for detecting that the next call to `push()` may fail, saving its state in order to suspend itself, and later restoring this state and resuming execution when it is called after a suspension. This code may take advantage of MERCATOR's per-node state facility, which lets the developer declare state variables that can be initialized at application load time and then read and written from within a node. Figure 3 illustrates a modification of the node in Fig. 2 to support suspension and resumption.

Even this simple example illustrates the challenges of user-directed suspension and resumption. The state variable is shared by all CUDA threads, so writes to it must be protected by block-wide synchronization calls to ensure that all threads see a consistent value. Real applications may need to store multiple pieces of state in order to resume execution. The more complex the control structure of the node (e.g., a `push()` inside nested loops), the more challenging it is to transform the code to behave correctly in the presence of suspension and resumption. Future work should investigate whether a CUDA language compiler can be extended to perform the transformations needed for node interruptibility or to implement an efficient `setjmp`-like facility.

```
__device__ void
MyNode::init() {
    if (threadIdx.x == 0)
      getState()->i = 0;
    __syncthreads();
}

__device__ bool
MyNode::run(int x) {
  int i = getState()->i;
  bool canContinue = true;
  while (i < M && canContinue) {
    int v = f(x, i);
    canContinue = push(v, v > 0);
    ++i;
  }
  __syncthreads();
  if (threadIdx.x == 0)
    getState()->i = (i == M ? 0 : i);
  __syncthreads();
  return (i == M);
}
```

Fig. 3 Modification of a node to support suspension and resumption. The current iteration i, which is the only state variable that must be stored on suspension, is initialized to 0 at application start and is then read from stored state each time the node is run. When a push indicates that the downstream queue is full, the loop is interrupted and its current state stored. If fewer than M iterations have completed, the node returns *false* to the MERCATOR runtime to indicate that it should be suspended

Finally, we note that the code transformations needed to support interruptibility themselves introduce overhead, in the form of additional state reads and writes and additional synchronization. The cost of this overhead must be weighed against the savings from fewer invocations of the node scheduler when evaluating the performance impact of interruptible nodes.

6 Empirical Evaluation

To evaluate the quantitative impacts of interruptible nodes on application performance and storage in MERCATOR, we implemented interruptible node support as described in the previous section, then modified the code of our example applications (BLAST and NQueens) to support saving and restoring of node state. We then investigated the behavior of these applications on an NVIDIA RTX 2080 GPU using CUDA 11.2. Applications were run using inputs as described in Sect. 4.1 with a SIMD vector width of 128 and used 368 pipeline replicas (the maximum number of CUDA blocks permitted on our GPU given the applications' register usage) to fully occupy all processors of the GPU. All reported running times represent the average over 50 trials.

6.1 Reduction of Scheduling Overhead

We first compared the NQueens application without interruptible node support ("NoInterrupt") to a version in which all 14 nodes of the pipeline were made interruptible ("AllNodeInterrupt"). We did this comparison for a range of target allocations for total queue memory, from 600 to 1400 MB summed over all queues in all replicas. For the NoInterrupt implementation, any queue bloat required for safety was allocated over and above this target value. For the AllNodeInterrupt implementation, we allocated the same total amount of memory as for the corresponding NoInterrupt version but redistributed the excess previously used for bloat across all the application's queues so as to minimize switching overhead, according to the analysis of [8].

Figure 4 shows that the net impact of interruptibility on performance was negative—the additional cost and complexity of saving and restoring node state far outweighed any savings from overhead reduction. This result was consistent over a range of possible targets for total memory allocated to queues.

Recall from Table 1 that only the first few nodes of the NQueens pipeline exhibited output queue bloat. To reduce the cost of interruptibility, we modified the AllNodeInterrupt implementation so that only the first four nodes of the pipeline were interruptible; the remaining nodes were left uninterruptible. This modified implementation ("4NodeInterrupt") exhibited a statistically significant performance *improvement* over the uninterruptible version for target allocations up to 1100 MB.

As shown in Fig. 5, redistribution of space previously needed for bloat in the first four nodes of NQueens had a salutary effect on scheduler overhead. Nodes near the middle of the the pipeline, which have the largest average cumulative

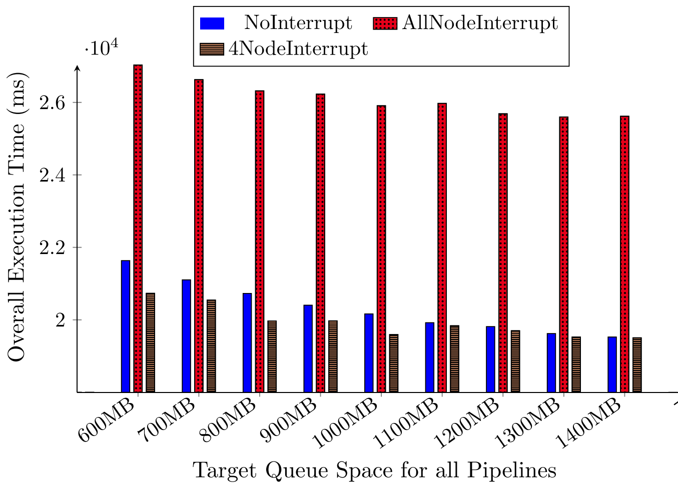

Fig. 4 Total execution time for NQueens for different target allocations for total queue memory. Measured times are the average of 50 trials and have a 95% confidence interval of ±150 ms

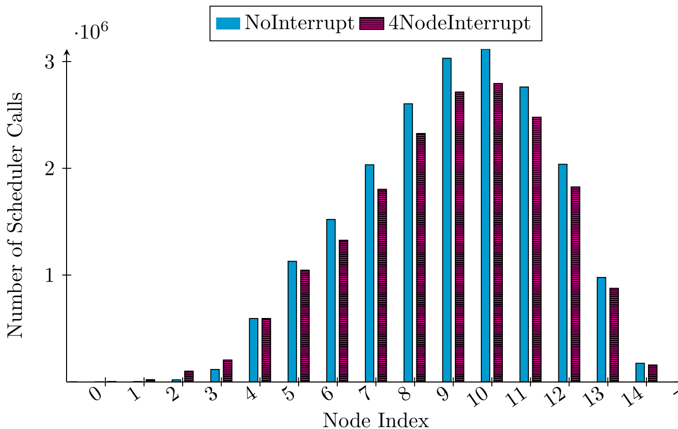

Fig. 5 Number of scheduler calls from each node for NQueens given 600MB total queue space for all pipelines

gain (i.e., process the most data) benefit the most from larger output queues through reduction in scheduler calls, which we believe to be the primary source of performance improvement. This benefit diminishes as the total memory allocated to queues grows, since bloat (and hence the memory redistributed to other queues) is the excess of a queue's minimum safe size, which is fixed, over its optimal target size, which grows with the overall memory allocation. Moreover, larger queues reduce the absolute number of times the scheduler is called, which further reduces the benefit of the redistribution optimization. Hence, we see that

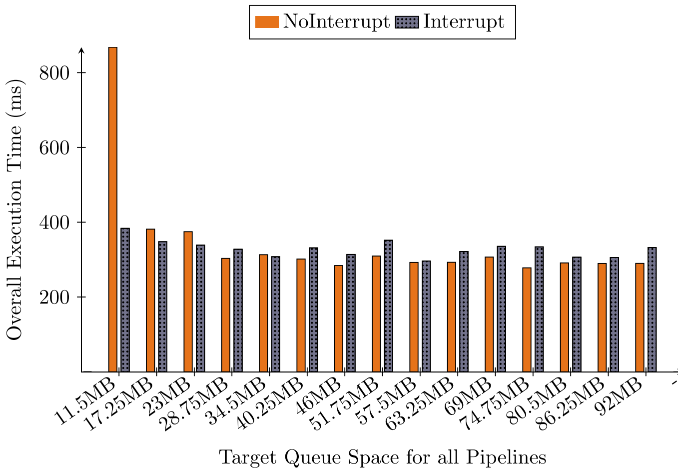

Fig. 6 Total execution time for BLAST for different target allocations of queue memory. Measured times are the average of 50 trials and have a 95% confidence interval of ±50 ms

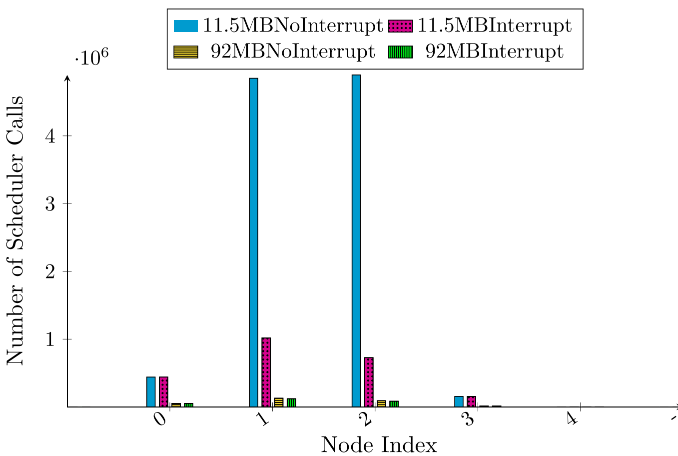

Fig. 7 Number of scheduler calls from each node for BLAST at two different target queue space allocations

the difference in running time between NoInterrupt and 4NodeInterrupt decreases with increasing target allocation.

We then investigated whether the same strategy of targeted interruptibility was effective for the BLAST application. Recall from Table 2 that BLAST was not significantly bloated even at a relatively small target allocation of queue space; however, we identified node 1, which has a max gain of 16 but an average gain of only 2, as having a queue that was significantly underutilized. We therefore made only this node interruptible and compared the performance of the modified application ("Interrupt") to that of the original, uninterruptible version ("NoInterrupt").

Figure 6 shows that at the smallest target allocation (11.5 MB across all queues), targeted interruptibility had a large beneficial effect on running time. As Fig. 7 shows, making node 1 interruptible greatly reduced the number of times it was forced to yield to the scheduler, as would be expected given that the node can now safely consume multiple vectors of input before filling its output queue. Allowing this queue to fill also reduced the frequency with which the following node, node 2, had to yield to the scheduler due to an empty input queue.

Once again, the benefits of interruptibility were highest at small target allocations, as larger allocations (e.g., 92 MB, as shown in the figure) are naturally large enough to permit node 1 to write multiple input vectors' worth of worst-case output to its output queue before yielding. Overall, we observed no statistically significant difference in performance in the interruptible vs. uninterruptible implementations for target allocations larger than 11.5 MB.

6.2 Combining Interruptibility with Node Merging

As discussed in Sect. 3, node merging to eliminate queue overhead is a potentially useful pipeline transformation. However, for nodes with large maximum gains, merging can result in excessive queue bloat for the merged node's output queue – bloat that can be ameliorated by making the merged node interruptible.

We investigated the impact of merging nodes 0 and 1 of the NQueens application, which our analysis in [8] suggested was potentially beneficial to performance. Without interruptibility, merging these two nodes increased the application's actual queue memory usage by 16-30% beyond the target, as shown in Fig. 8. Making the merged node interruptible eliminated this excess memory usage. The resulting implementation exhibited running time similar to that of the unmerged version.

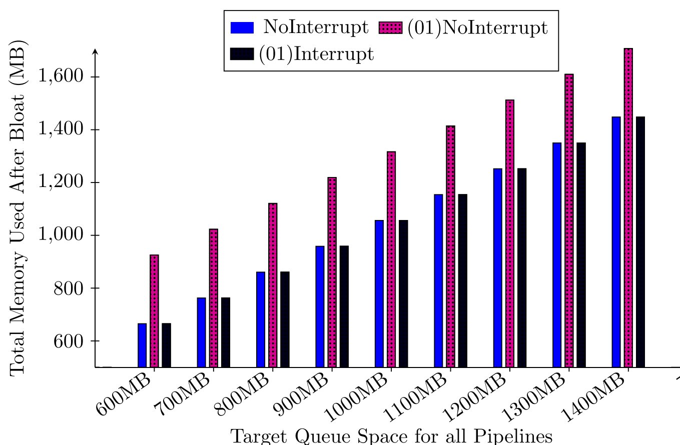

Fig. 8 Total actual memory used for each target allocation in NQueens before and after merging nodes 0 and 1

7 Conclusion and Future Work

Irregular streaming dataflow applications have great potential for wide-SIMD parallelization, but this potential can be realized only by inserting queues in the application pipeline. The "right" sizes for these queues are determined by potentially conflicting design considerations: performance, which favors certain relative queue sizes to reduce scheduling overhead, and safety, which imposes often severe minimum queue size requirements when a node can produce many outputs per input in the worst case. We have shown that the pressure of safety on queue size can be ameliorated by selectively making nodes *interruptible*, and that doing so can be a net positive for throughput and/or queue memory usage.

Interruptibilty works best when targeted to nodes with large maximum gains but much smaller individual or cumulative average gains. The space saved from such node's output queues can be removed from the application, decreasing its memory usage, or redistributed to other nodes in the pipeline, potentially increasing throughput. Even when queue sizes do not change much after interruptibility, a node whose average gain is far below its maximum gain can benefit from reduced scheduler overhead when it is made interruptible. These benefits are most readily seen when the overall target allocation of queue space to the application is smaller, since changing queue sizes and allowing greater queue occupancy have the largest impact when the total available queue space is small.

In the future, we hope to obtain more accurate measurements of the overhead of interruptibility, in particular the cost of saving and restoring state. Timing these operations, which take place inside a function called within the CUDA kernel, is challenging, particularly because they may involve operations by multiple GPU threads that run asynchronously. We also plan to better model potential *increases* in node switching overhead when the bloat is removed from a node's output queue. Accounting for these effects would allow us to better predict whether making a node interruptible is likely to be beneficial to throughput overall and to direct the effort of optimization accordingly.

Another avenue for investigation is whether the burden of interruptibility on the application developer — in particular, the need to extensively rewrite code to support interruptions — can be reduced. Ideally, the CUDA runtime would provide support to implement suspension and resumption of nodes with appropriate saving of state in between. Because of GPUs' very large register files, naively saving all register state for a block when suspending might have a prohibitive cost in time and memory; hence, it may be preferable to leverage compiler analysis or user-provided variable tagging to identify and save only live data at the point of suspension. For example, [16] uses static metaprogramming to seamlessly implement asynchronous programming calls. Suspension and resumption of code is directly supported by the code transformations done before compilation. Alternatively, GPU support for hardware preemption would greatly aid interruptibility and might change the preferred realization of streaming applications on the GPU entirely. For now, the impacts of hardware preemption for irregular streaming could be investigated on multicore CPUs, which have robust preemptive multithreading as well as increasingly large SIMD vector widths.

Acknowledgements This work was supported by National Science Foundation award CNS-1763503.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Molecul. Biol. **215**(3), 403–410 (1990)
2. Tyson, E., Buckley, J., Franklin, M., Chamberlain, R.: Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the auto-pipe design system. Nucl. Instrum. Methods Phys. Res. Sect. A Accel. Spectrometers Detect. Assoc. Equip. **595**, 474–479 (2008)
3. Cabrera, A.M., Faber, C.J., Cepeda, K., Derber, R., Epstein, C., Zheng, J., Cytron, R.K., Chamberlain, R.D.: DIBS: a data integration benchmark suite. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. ICPE '18, pp. 25–28. Association for Computing Machinery, New York, NY, USA (2018)
4. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration. LISA '99, pp. 229–238. USENIX Association, USA (1999)
5. Kolesar, P.J.: A branch and bound algorithm for the knapsack problem. Manag. Sci. **13**(9), 723–735 (1967)
6. Viola, P., Jones, M.: Robust real-time object detection. In: International Journal of Computer Vision (2001)
7. Cole, S.V., Buhler, J.: MERCATOR: A GPGPU framework for irregular streaming applications. In: 2017 International Conference on High Performance Computing Simulation (HPCS), pp. 727–736 (2017)
8. Timcheck, S., Buhler, J.: Reducing queuing impact in streaming applications with irregular dataflow. Parallel Comput. **109**, 102863 (2022)
9. Plano, T., Buhler, J.: Scheduling irregular dataflow pipelines on SIMD architectures. In: Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing. WPMVP'20, pp. 1–9. Association for Computing Machinery, New York, NY, USA (2020)
10. Kim, H., Patel, P., Wang, S., Rajkumar, R.R.: A server-based approach for predictable GPU access control. In: 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10. IEEE (2017)
11. Kato, S., Lakshmanan, K., Rajkumar, R., Ishikawa, Y., et al: TimeGraph: GPU scheduling for real-time multi-tasking environments. In: 2011 USENIX Annual Technical Conference (USENIX ATC 11) (2011)
12. Park, J.J.K., Park, Y., Mahlke, S.: Chimera: collaborative preemption for multitasking on a shared GPU. ACM SIGARCH Comput. Arch. News **43**(1), 593–606 (2015)
13. Wu, B., Liu, X., Zhou, X., Jiang, C.: FLEP: enabling flexible and efficient preemption on GPUs. ACM SIGPLAN Not. **52**(4), 483–496 (2017)
14. Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE **75**, 1235–1245 (1987)
15. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: a language for streaming applications. In: Horspool, R.N. (ed.) International Conference on Compiler Construction, pp. 179–196. Springer, Berlin (2002)
16. Prokopec, A., Liu, F.: Theory and practice of coroutines with snapshots. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.