Reducing Queuing Impact in Irregular Data Streaming Applications

Stephen W. Timcheck
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
stimcheck@wustl.edu

Jeremy D. Buhler
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
jbuhler@wustl.edu

***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 data flow, i.e., those where the number of outputs produced per input to a node is data-dependent and unknown *a priori*. Moreover, we target these applications to architectures (GPUs) where different nodes of the pipeline execute cooperatively on a single wide-SIMD 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 queue reading, writing, and management. 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, we consider which pairs of successive nodes in a pipeline should have queues between them to maximize overall application throughput. Second, given 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. We formulate a dynamic programming approach to the first problem and give an empirically useful approximation to the second that allows for an analytical solution. Finally, we validate our theoretical results using real-world irregular streaming computations.

***Index Terms*—queuing, SIMD, irregular, dataflow, streaming**

I. Introduction

Streaming applications with irregular dataflow exist in numerous high-impact fields, ranging from biosequence analysis [1] to astrophysics [10]. These applications are commonly represented as pipelines of computational stages connected by dataflow edges. For instance, the BLAST tool for biological sequence comparison [1] can be described as a four-stage pipeline that ingests a large database of DNA or protein sequence and filters its input to retain just the portions that approximately match a given query sequence. Irregular dataflow arises when the work performed by a computational stage, and in particular the amount of output it generates per item in its input stream, is variable, data-dependent, and therefore unknown *a priori*.

When a streaming computation performs largely independent computations on successive inputs in the stream, the computation's throughput can be increased by exploiting data parallelism across its inputs. Wide-SIMD processors such as GPUs are therefore tempting targets for such applications. However, irregularity interferes with SIMD parallelism because different inputs to a pipeline may require different amounts of work or may even be filtered away 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 lanes doing useful work) will suffer.

To improve the occupancy of irregular dataflow pipelines, one may employ intermediate *queues* between successive compute stages. A queue functions as a staging area where data from the previous compute stage, otherwise known as a compute node, can be accumulated, compacted, and then redistributed across SIMD lanes to ensure full occupancy of the next stage. However, reading, writing, and remapping through 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 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. 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. GPUs also offer limited support for preemption, so the stages of the pipeline must be cooperatively

Funded by NSF award CNS-1763503.

scheduled on the single processing resource. Finally, the number of processors is large enough that tens or even hundreds of pipeline copies may be running at once.

Each pipeline replica on a GPU needs its own queue space. Given a large number of copies, it becomes important to limit the amount of queue space allocated to each replica, 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. First, we formalize the tradeoff of when to insert queues between successive pipeline stages. Using easily-obtained performance metrics from an application's profile, we formulate an algorithm for selecting which stages should be merged together and which should have queues between them. Second, we consider the problem of dividing limited space among queues in a pipeline. Assuming a simple, effective scheduling policy for pipeline stages [7], 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. These optimizations can have material impact on an application's overall throughput.

The rest of the paper is organized as follows. Section 2 examines related work. Section 3 provides a detailed explanation of our application model and the metrics used for later sections. Section 4 describes a method for determining whether queues should be added between compute nodes and a method for choosing where to place queues. Section 5 provides a method for determining how much space to allocate for individual queues given a limited space budget. Section 6 evaluates both methods on irregular streaming applications implemented on an NVIDIA GPU platform. Finally, section 7 concludes and explores future work.

II. Related Work

Many application frameworks have been developed to support *regular* streaming dataflow applications on parallel systems. A prominent example, StreamIt [9], was built around the synchronous data flow (SDF) [5] 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 where and when, thus creating a need for data-driven decisions about queue placement and sizing.

Subhlok and Vondran [8] 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 various forms of 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 [2] 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 requirements, which in our case would be reading, writing, and managing an intermediate queue. Although their model does not take into consideration communication costs between processors, it works with a more general purpose MIMD processor in mind. Hence, their communication cost would be equivalent to the scheduler cost we consider in our model.

III. 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 Figure 1, an application is a linear pipeline of *compute nodes* $n_1 \dots n_m$ 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 two parameters: its *service time* t_i and its *gain* g_i. The service time t_i is the time for a node to process a vector of input items; the time is the same for any number of items $\leq v$. The gain g_i is the *average* number of outputs produced per input item consumed, which may be greater or less than 1. For convenience, we also define the *cumulative gain* $G_k = \prod_{i=1}^{k} g_i$ to be the average number of outputs from node n_k for each input consumed by n_1. We focus on the mean-value behavior of nodes, leaving consideration of other moments of the outputs-per-input distribution for future work. We seek to optimize the *throughput* of the pipeline, or equivalently the total time to completely process a large number of inputs to n_1.

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 it pulls contiguous vectors of up to v items from the queue. Because dynamic resizing of queues on GPUs is expensive, we assume that each queue has a fixed size that is determined at compile time.

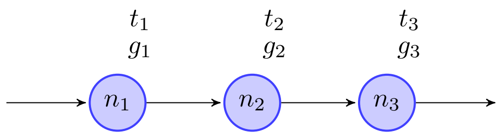

Fig. 1. A simple pipeline application topology. Node n_1 feeds into n_2, and n_2 feeds into n_3. Node n_i has service time t_i and average gain g_i.

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 executes that node, consuming some amount of input. The application ends when no node has any inputs remaining.

The particular scheduling protocol we consider in Section 5 is the AFIE (active-full, inactive-empty) scheduler [7]. 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 downstream neighbor, if any, is 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. It was shown in [7] that AFIE is deadlock-free (that is, a node with input eventually becomes eligible to execute), that it ensures that nodes almost always execute with a full vector-width of inputs, and that it incurs no more than about twice as many *switches* (calls into the scheduler to choose a new node to execute) as would a clairvoyant protocol that knew in advance the number of outputs produced by each node for each input.

IV. Determining When to Use Queues

Queuing on an edge between nodes is valuable for irregular streaming applications as a tool to improve SIMD occupancy. Even if only a subset of SIMD lanes in the upstream node produce outputs, or if different lanes produce different numbers of outputs, the queue allows items to be remapped into contiguous lanes for execution by the downstream node. However, queues introduce a certain amount of overhead to applications for reading and writing items on the edge and for compaction and remapping. Moreover, applications with more edges, and hence more queues, impose a greater load on the scheduler, which must be called more often to ensure that all nodes have an opportunity to run.

Remapping between nodes is not necessary for correct execution. Given nodes n_i and n_{i+1}, 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. (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}. Figure 2 illustrates the merge operation on the last two nodes of the pipeline from Figure 1.

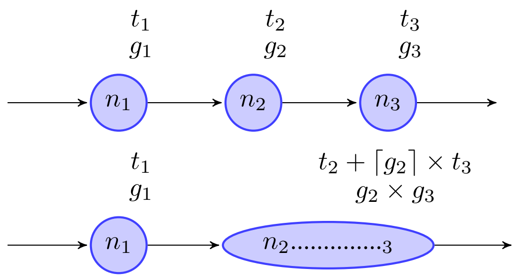

Fig. 2. Topological view of merging compute nodes. Combining nodes n_2 and n_3 may incur multiple calls to n_3 per call to n_2, each with fewer inputs, because inputs to n_3 are no longer queued. The average output gain is now the cumulative gain between the two combined nodes.

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. However, we lose the benefits of remapping for SIMD occupancy, so that it may be necessary to call n_{i+1} more often 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 will 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 $n_1 \ldots n_m$ be a pipeline of nodes of common vector width v, with n_i having output gain g_i and service time t_i. For convenience, we expand the definition of cumulative gain G_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 $G_{j,j-1} = 1$.

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 would require knowing the full distributions of the gains of each n_i, rather than just their average gains. For simplicity, we assume that a node's gain in each lane remains close to its average but round this gain up to the next integer to account for some lanes having more outputs than others. With this simplification, the service time t_{jk} of n_{jk} is given by

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

Now suppose we insert a queue between original nodes n_i and n_{i+1}, $j \leq i < k$, 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 output vectors per input vector to n_{ji} is just $G_{j,i}$. We additionally charge a fixed time overhead p each time n_{ji} runs to account for the costs of writing, remapping, and scheduling at this node. Hence, the total running time of the node pair per input vector to n_j is now

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

More generally, let T_j be the least total running time per input vector to n_1 obtained by inserting queues on any subset of the edges in the range $n_j \ldots n_m$. Either no such queue is inserted, or the first such queue occurs after node i, $j \leq i < m$. We therefore derive the recurrence

$$T_j = \min \begin{cases} t_{jm} \\ \min_{j \leq i < m} t_{ji} + p + G_{j,i} T_{i+1}. \end{cases}$$

This recurrence can be solved by dynamic programming for the entire pipeline in time $O(m^2)$, following precomputation of the t_{jk} values, to obtain the optimal subset of edges on which to insert queues.

V. Choosing Sizes for Finite Inter-node Queues

Assume now that a division of the pipeline into (possibly merged) nodes has been determined. Post-merging, we assume that the pipeline has nodes $n_1 \ldots n_m$, with a queue between each successive pair of nodes. As noted in Section 3, we assume that nodes in the pipeline are scheduled using the AFIE scheduler. Because AFIE waits until a node's upstream queue is full to activate it, 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*, which may be on the same order as node service times.

However, as discussed earlier, an efficient GPU implementation of the application may require a large number of copies of the pipeline – at least one per processor to avoid complex inter-processor communication, and possibly multiple copies per processor 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 is likely unacceptable. 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.

A. Bounding Rate of Switches under AFIE 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.

Intuitively, execution must switch away from node n_i (and hence back to the scheduler) whenever its input queue becomes empty or its output queue becomes full. In either case, n_i cannot continue executing until some other node runs, either to produce more input or to consume some output. Because AFIE ensures that n_i becomes eligible to execute only when its input queue fills and its output queue empties, its 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. However, occasionally, these two events (emptying of input and filling of output queues) occur concurrently

— about once per $\mathrm{lcm}(c_{i-1}, c_i/g_i)$ inputs consumed[1] — which results in only one rather than two switches. In short, we can establish the following lemma (proof provided in the appendix):

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

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

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

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

B. 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 $q_1 \ldots q_{m-1}$ 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 with some probability slightly advance or retard the filling of q_i relative to the emptying of q_{i-1}, turning one resonant switch into two ordinary ones.

[1]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 $lcm(a, b)/gcd(c, d)$.

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 constraints. R' is an upper bound on the true switching rate R, 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 10-20%.

Replacing R by R' yields a much more tractable 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

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

In practice, we round the c_i values thus obtained to integers that permit each queue to hold a whole number of items. Moreover, safety considerations dictate a minimum allowable size for each queue: q_i must be large enough to hold all the output produced by one worst-case execution of n_i. If optimization yields an infeasibly small queue size, we round it up to this feasible minimum.

VI. Empirical Evaluation

We tested our optimization methods for queue placement and queue sizing on irregular streaming applications implemented in the MERCATOR [4] framework for NVIDIA GPUs. Applications were benchmarked on an NVIDIA GTX 1080Ti with 28 processors, using CUDA 11 under Linux. For all tests, we used a width of 128 threads per block. With this configuration, full utilization of the GPU (as recommended by NVIDIA's runtime API) usually entailed creating several hundred blocks, each with one replica of the application pipeline. We allocated between 64 KB and 256 KB of space to the queues for each application. Smaller amounts of space led to violations of the safety considerations mentioned in the previous section, while larger amounts were observed to have negligible performance impact. For all experiments, the reported averages had negligible variation across multiple trials.

A. Node Merging Optimization

We studied the impact of node merging on the core computation of NCBI BLAST [1], a genomic sequence database search tool. BLAST comprises a pipeline of four stages as shown in Table I, plus a source and a sink node. We instrumented the application to measure each node's average gains and service time per input vector, as well as the average overhead of scheduling and execution management per input vector. Data shown in the table was profiled from a comparison of a 30 Kbase DNA query (from the *Salmonella* genome) against a stream containing the human genome with repetitive elements removed, which was around 2 Gbases in size. Service times were measured from the node's main firing loop, which includes getting data from node's upstream queue, running the node's function on its input, and writing its output to the downstream queue. The scheduler overhead p per input vector was computed by adding all setup and teardown costs of calling any compute node plus the total number of cycles spent in the scheduler, then dividing by the total number of input vectors processed by all compute nodes. These measurements were averaged over five runs of the program.

TABLE I
COMPUTE NODE ANALYSIS OF BLAST

Compute Node	**Avg Gain Out**	**Avg t_i (cycles)**[a]
Seed Match	0.365619	28
Seed Enumeration	1.715568	64
Small Extension	0.023177	30
Ungapped Extension	0.125698	196
Scheduler[b]	-	4

[a] Service times are from 64KB even distribution, per input vector consumed.
[b] Estimated overhead (p) per input vector consumed

We used the dynamic programming algorithm of Section 4 together with the BLAST profile to compute a queue insertion strategy that minimized the application's estimated running time per input vector consumed. The solution merged the first two stages and the last two stages of the BLAST pipeline, leaving a queue between Seed Enumeration and Small Extension. We then compared the merged implementation to a baseline with a queue between each pair of nodes. For this test, we held the number of GPU blocks constant at 336 between the merged and unmerged implementations and held the total queue space used by each pipeline replica constant, allocating the space of the queues removed by merging to the remaining queue in the merged pipeline. For the unmerged implementation, we divided the queue space equally among all queues. We report total execution time (time until last block finishes) of the merged and unmerged implementations.

Figure 3 illustrates the impact of merging on BLAST's execution time. The merged implementation ran 10-20% faster than the unmerged implementation, depending on total queue space. Increases in performance due to merging were larger with smaller queues, suggesting that reduction in scheduler switching overhead was an important benefit of merging.

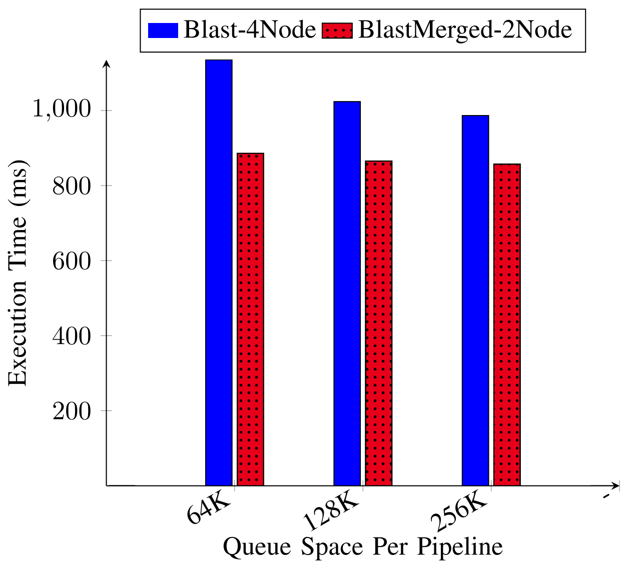

Fig. 3. Total execution time for unmerged vs merged BLAST application, averaged over 50 trials.

B. Queue Sizing Optimization

We tested the impact of the queue sizing optimization from Section 5 on two applications: the BLAST pipeline of the previous section (unmerged version), and a CSV parsing and rewriting application, tstcsv→csv, from the DIBS data integration benchmark set [3]. We refer to the latter application as "Taxi" below. Its pipeline includes three stages plus a source and sink node. We tested Taxi on a 1.8 GB input file.

For each application, we compared its performance with an equal division of memory among all queues vs. the division recommended by Lemma 5.2. We measured the number of switches between nodes during a full execution using CUDA's recommended number of blocks (336 for BLAST, 448 for Taxi). We also measured execution time, this time running only one block (one replica of the pipeline) on each GPU processor. The latter measurement, while not fully utilizing the GPU, allowed us to accurately account cycles spent in the application's nodes vs. the scheduler. (Speedups observed using the much greater recommended number of blocks per processor were qualitatively similar.)

Figures 4 and 5 show the impact of queue space redistribution on the number of switches. The total number of scheduler calls was substantially reduced, by 50% or more in some cases for BLAST. As expected, the absolute number of switches declines as the overall queue space per pipeline replica increases.

Figures 6 and 7 show the impact of queue space redistribution on average time spent on compute nodes vs. scheduler overhead. As expected, time spent executing nodes is nearly unchanged, but time spent in the

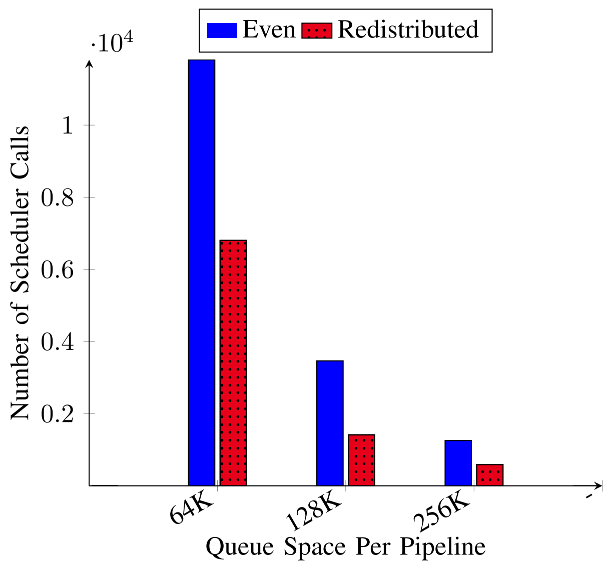

Fig. 4. Number of calls to scheduler for BLAST, averaged over 50 trials.

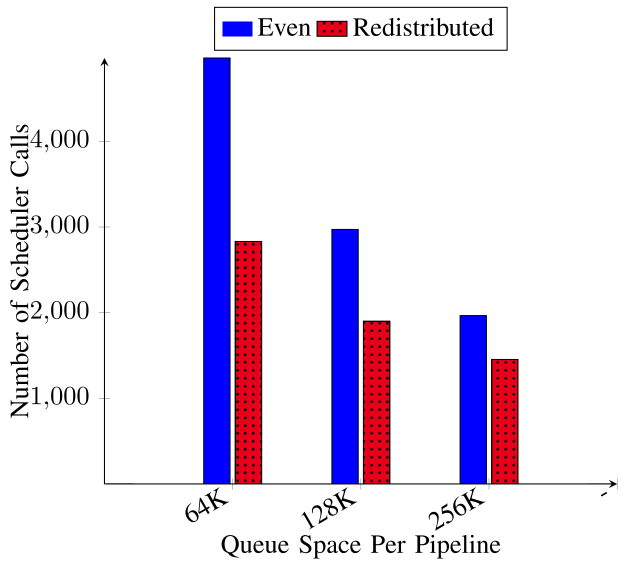

Fig. 5. Number of calls to scheduler for Taxi, averaged over 50 trials.

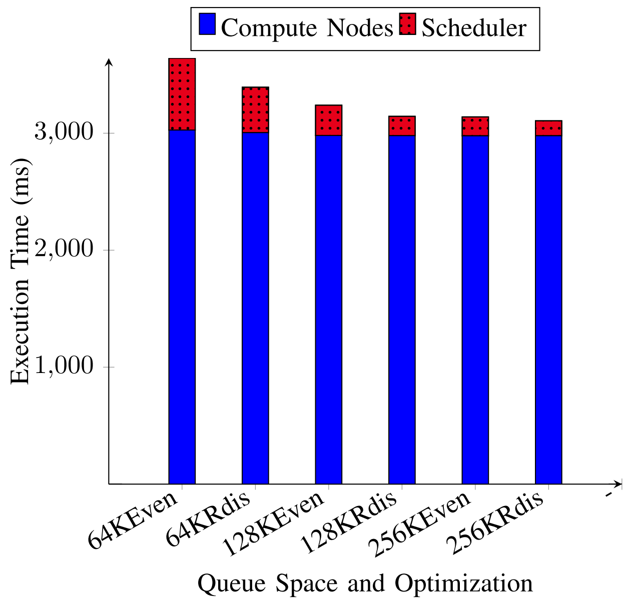

Fig. 6. Time spent by BLAST application in Scheduler (red) and Compute Nodes (blue), averaged over 140 executions of one pipeline replica.

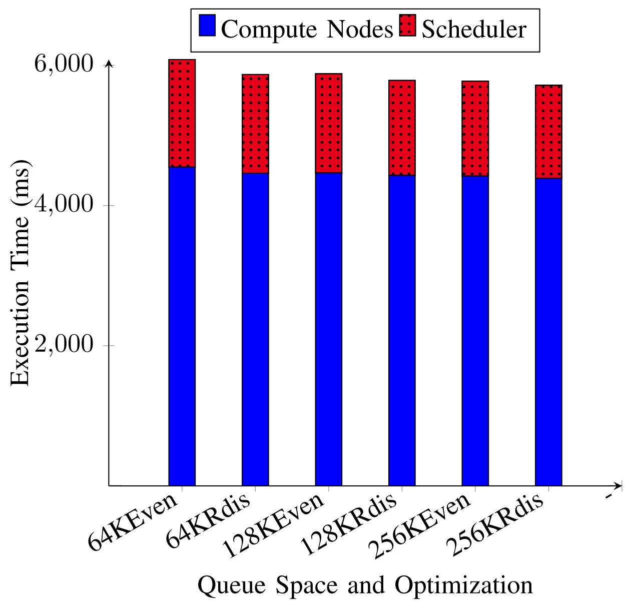

Fig. 7. Time spent by Taxi application in Scheduler (red) and Compute Nodes (blue), averaged over 140 executions of one pipeline replica.

scheduler decreased after redistribution. The effect was greater for BLAST than for Taxi and was again more pronounced with smaller overall amounts of queue space, which increased the number of switches incurred.

C. Double Optimization

After finding each of our two optimizations to be individually beneficial, we next investigated the impact of applying both to the BLAST pipeline. We applied queue space redistribution to the merged version of BLAST, then compared the doubly-optimized version against the unoptimized and both singly-optimized versions. Again, the number of GPU blocks was held constant at 336 for all versions.

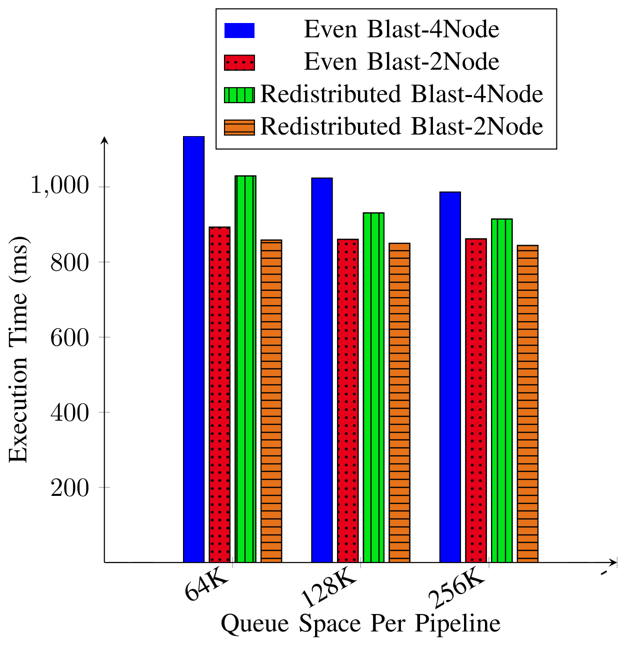

Fig. 8. Total execution time for BLAST with one or both optimizations, averaged over 50 trials.

Figure 8 illustrates that combining both optimizations yielded better performance than each by itself. Merging had the greatest overall impact either with or without redistribution, and redistribution was more effective by itself than when combined with merging. These observations support the hypothesis that both optimizations affect scheduling overhead, rather than targeting orthogonal aspects of application performance.

VII. Conclusions and Future Work

In this paper, we explored how to optimize irregular streaming dataflow applications on SIMD processors by controlling the placement and sizes of inter-node queues. We first gave a dynamic programming algorithm for inserting queues in an application based on a simplified model of node service times. We then devised an optimization strategy for the relative sizes of queues given an overall storage budget for the pipeline. Both optimizations were driven by profile data on the service times and output behavior of each node in the application, and both directly or indirectly targeted the cost of scheduling multiple nodes of an application on a single processor. Each optimization was observed to be useful by itself, and they gave a small additional improvement in application running time when applied together. Broadly, our results illustrate the importance of considering performance tradeoffs between improved SIMD occupancy and overhead when implementing irregular streaming dataflow applications on wide-SIMD processors.

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. Expansion of the node merging optimization to permit varying vector widths between compute nodes will allow us to apply it to more applications, including Taxi. We will more accurately model merged node service times using the full empirical distribution of gains for each node, rather than just the average, and will develop better models of queue overhead, perhaps including cache effects. Extension to dataflow graphs with DAGs and cycles is possible; however, the semantics of nodes with multiple input streams are not entirely clear in irregular applications. [6] offers one possible set of semantics that lead to nontrivial safety and efficiency challenges. Finally, we hope to build auto-tuning capabilities for frameworks such as MERCATOR by profiling and re-optimizing application pipelines automatically at run time.

References

[1] Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D.: Basic local alignment search tool. J. Molecular Biology 215(3), 403-10 (1990)
[2] Benoit, A., Robert, Y. Complexity Results for Throughput and Latency Optimization of Replicated and Data-parallel Workflows. Algorithmica 57, 689–724 (2010). https://doi.org/10.1007/s00453-008-9229-4
[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: 2018 ACM/SPEC Int'l Conf. Performance Engineering. pp. 25-28 (2018)
[4] Cole, S., Buhler, J.: MERCATOR: A GPGPU framework for irregular streaming applications. In: 2017 Int'l Conf. High Performance Computing and Simulation. pp. 727-36 (2017)
[5] Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE 75(9), 1235-245 (1987)
[6] Li, Peng & Beard, Jonathan & Buhler, Jeremy. (2015). Deadlock-free Buffer Configuration for Stream Computing. 10.1145/2712386.2712403.
[7] Plano, T., Buhler, J.: Scheduling irregular dataflow pipelines on SIMD architectures. In: 6th Wkshp. on Programming Models for SIMD/Vector Processing, San Diego, CA, Feb 2020.
[8] Subhlok, Jaspal and Vondran, Gary. (1997). Optimal Latency–Throughput Tradeoffs for Data Parallel Pipelines. Annual ACM Symposium on Parallel Algorithms and Architectures. 10.1145/237502.237508.
[9] Thies, W., Karczmarek, M., Amaransinghe, S.: StreamIt: A language for streaming applications. In: 11th Int'l Conf. Compiler Construction. pp. 179-96 (2002)
[10] Tyson, E., Buckley, J., Franklin, M., Chamberlain, R.D.: Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the AutoPipe design system. Nuclear Instruments and Methods in Physics Research Sec. A: Accelerators, Spectrometers, Detectors and Associated Equipment 595(2), 474-9 (2008)

Appendix
Proof of Lemma 5.1

Lemma: For $1 < i < m$, 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. QED