WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Dissertation Examination Committee:
Jeremy Buhler, Chair
Sanjoy Baruah
Roger Chamberlain
Shyam Dwaraknath
Christopher Gill

Throughput Optimizations for Irregular Dataflow Streaming Applications on Wide-SIMD Architectures

by
Stephen Timcheck

A dissertation presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2023
St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am gracious to the many individuals who gave me guidance and support in developing this work. Most of all, I would like to thank Jeremy Buhler, my research advisor, who was always there for me, providing access to his extensive knowledge, sage guidance, and unyielding support. I would also like to thank my labmates for listening to my many talks, providing feedback on my work, and helping me to dig in areas I would not have otherwise thought. I would like to thank my dissertation committee for giving me valuable feedback on this work. Finally, I thank all my family and friends for supporting me throughout my work.

This work was supported by NSF awards CNS-1763503 and CNS-1500173.

Stephen Timcheck

Washington University in St. Louis
May 2023

ABSTRACT OF THE DISSERTATION

Throughput Optimizations for Irregular Dataflow Streaming Applications on Wide-SIMD Architectures

by

Stephen Timcheck

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2023

Professor Jeremy Buhler, Chair

Streaming dataflow applications exist in numerous fields of study including, but not limited to, bio-sequence analysis, astrophysics, network packet analysis, and data integration. The inputs for these applications are independent of one another, making the problems prime candidates for parallel acceleration using wide-SIMD vector processors, such as graphics cards. However, many of these streaming applications exhibit *irregular dataflow*, where the number of outputs per input from any computation within the application is data dependent and unknown *a priori*. To ameliorate the throughput issues caused by irregular dataflow on wide-SIMD systems, we utilize a framework such as MERCATOR, which *queues* data between computational stages, ensuring that each computational stage has full-width SIMD vectors with which to work.

Implementing queues within the application do not come without costs however. Overheads are incurred for queueing between compute nodes including physically moving data between queues, determining how many inputs from the queue can be safely consumed at one time, and memory usage of the queues themselves. This dissertation looks to *optimize irregular dataflow application throughput* through four pieces of work, with direct analysis of the queues and feature additions to enable lower overheads with greater queue functionality. We first examine queues in the context of region-based state, and how to efficiently make execution boundaries within a data stream. Next, we explore how much space should be assigned to queues within an application and whether the overhead of queueing is worthwhile based on runtime statistics. We then cover how minimum queue size requirements affect application performance and limit viable application implementations, which we solve using interruptible execution. Finally, we examine how to efficiently implement iterative applications for throughput.

Chapter 1

Introduction

1.1 Introduction

In recent years, computing has reached an inflection point. The end of Moore's Law and Dennard Scaling have ended years of large incremental improvements to serial execution. Many applications at the time, and still today, perform tasks that can be done in parallel with one another. To gain more performance, chip manufacturers and programmers alike have gravitated towards *parallel computing* via multicore CPUs and distributed computing over networks.

Various computing paradigms exist including: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data (MIMD). First, SISD encompasses serial tasks that work on a single data item at a time; an example is the classic *von Neumann* processor model. Next, SIMD stands for single operations which work on multiple data items simultaneously and is most commonly found on *vector processors*. MIMD is the standard model for multi-core CPUs and distributed systems, where multiple processors work on multiple data items at the same time.

SIMD processors first saw usage in the supercomputing sphere, with vector processors such as the Cray-1 [12]. These supercomputers would take in large amounts of data divided into *vectors* of some number of data elements and would perform the same calculation for each input in the vector. Quite often, all the required operations on the input data were independent of one another, making SIMD parallelization a simple and effective way to process large amounts of data, even if singular processor speeds were not the fastest at the time.

One ideal application for SIMD processing is *computer graphics*. On the simplest of levels, one can think of an image as a matrix of pixels consisting of red, green, and blue values. Pictures of high resolution have thousands of pixels that need to be drawn or (if being processed by an application) modified at the same time. A SISD processor would need to sequentially update red, green, and blue values for each pixel, whereas a SIMD processor could modify all three at the same time as a single vector. Thus, *graphics cards* (GPUs) adopted SIMD processing techniques for updating pixel component values at the same time, cutting computation times. Further, by having *wide-SIMD* with multiple vector processors available, a GPU could update pixel values for thousands of pixels at the same time. Eventually, the SIMD nature of GPU computations lead to GPUs being repurposed for general purpose computing rather than only graphics [5].

Since the scope of all parallel computing applications is too large and diverse to enumerate, this work will focus on *irregular dataflow streaming applications*; those where one does not know how much data is produced at any given point of the application *a priori*. Irregular dataflow streaming applications are widespread and exist in fields such as biosequence analysis [1], astrophysics [37], data integration tasks [6], decision cascades in machine learning [38], network intrusion detection [29], and n-body simulations [2]. As an example, say one wants to find if a specific chicken gene has any DNA subsequences similar to those in the human genome. *BLAST*, or Basic Local Alignment Search Tool [1], takes a query DNA sequence and performs inexact matching to a database of DNA sequences. Since the database can be large (on the order of gigabases), and the query DNA sequence can have multiple matching DNA subsequences throughout the database, the problem size is large enough to require acceleration through parallelism. However, the number of matches between certain parts of the query DNA sequence and database DNA sequence can vary greatly, which means the number of outputs is highly input-dependent. The goal of each application is to process at the highest dataflow rates possible, thus **maximizing application throughput**.

Although SIMD accelerators such as GPUs are amenable to tasks which perform the same operation on multiple data items at once, there are a few limitations with SIMD architectures which make achieving high throughput more difficult. *Wide-SIMD* processors — those with a vector size on the order of hundreds to thousands — expect large quantities of *dense* data to maintain a high *occupancy*, which is a high number of processors with useful work to do. Attaining high occupancy is much easier when one knows how many inputs will arrive at different stages of the application, a fact which is taken full advantage of in wide-SIMD frameworks such as StreamIt [31]. However, with irregular dataflow applications, the number of outputs for every input is unknown *a priori* and is data-dependent. Therefore, to achieve high occupancy, *intermediate data queues* are utilized between *computational stages* of the application to accumulate data which then provides full occupancy datasets to process. This work builds irregular dataflow streaming applications in the MERCATOR framework, which provides facilities to easily queue data between computational stages.

Throughput depends not only on high occupancy but also on minimizing the overhead of queuing between compute nodes. One limitation of wide-SIMD processors like GPUs is that dynamic memory allocation is a costly operation, which means that queue sizes must remain largely static during application execution. Choosing queue locations and queue sizes can drastically affect how often an application must pause execution to manage its queues. By reducing the number of times a processor needs to examine different queues, one can increase throughput. Depending on the application, data expansion/filtering rates affect overall application size on a SIMD device and can impact throughput.

1.1.1 Application Model

We first examine wide-SIMD architectures, both how they function and a brief history of their origins. We then explore the properties of dataflow streaming applications and irregularity in their context. Finally, we look at example applications within the space, providing high-level implementation details.

Wide-SIMD Architecture

SIMD performs a single operation on many inputs given to a processor at once. The input set to a SIMD processor is called a *vector* and is of a certain *vector width*, or number of inputs that can be operated on simultaneously. All data of a vector is worked on in *lock-step*, where the vector elements are processed at the same time. When two inputs in a given vector require divergent sequences of operations, two sequential execution passes are made over the vector.

While graphics applications were the driving force for SIMD processors, vector instructions were identified for use in *high performance computing* applications. Noticing the high throughput of GPUs, applications that adapted existing texture mapping and vertex shading operations provided by graphical libraries [39] were used to perform non-graphics computations. Such adoption was coined as *general-purpose GPU* (GPGPU) by Harris [16] because of the variety of applications being ported to GPUs.

Originally, GPGPU programming had to be done in assembly, with some higher-level implementations such as Microsoft's HLSL [22] and NVIDIA's Cg [21] languages providing better programming abstractions. However, extensive knowledge of graphics APIs was still required for things such as allocating texture memory and loading shader programs [5]. GPGPU programming languages such as Brook were developed to make the abstraction between GPU assembly and graphics APIs more amenable to general-purpose programming [5].

The Brook language utilizes the concept of a *data stream*, or a collection of data items that can be processed in parallel, in the context of GPGPU programming. Collections of data-parallel operations, or programs on the GPU that process data streams in parallel, are called *kernels*. A similar advancement building upon Brook's data stream model is NVIDIA's CUDA language, which removed the need for implementing graphical constructs to build high performance computing applications. This abstraction created a software environment that explicitly provided access to general-purpose operations on a GPU, which further broadened the reach of wide-SIMD execution.

At the same time GPGPU programming began, mainstream CPUs became *multi-core* with the MIMD processing paradigm. Distributed systems effectively run as a MIMD system, so multi-core CPUs were a natural target for expanding parallelism in those systems. An advantage of a MIMD system is that distinct input data items can be processed in independent *instruction streams*. However, the ability to process independent data streams requires more die space to replicate logic elements such as program counters.

While MIMD processors may at first seem to be the ideal choice for all applications, there is extra hardware required to maintain independent parallel processors. The processors must keep to memory coherence protocols which require communication infrastructure in hardware. Each processor also has separate program counters, registers, and potentially differing operating frequencies depending on the task or system. This functionality comes at the cost of processor die area, meaning processors are larger overall.

Systems using SIMD forego the independent processing nature of MIMD and instead focus on vector instructions where each input is a vector rather than a single item. Since many applications do not require independent instruction streams, one can simplify the processor e.g., by sharing a program counter between processors, to save die space and reduce processor complexity. By saving die space, more simultaneous data lanes can be implemented per processor, making the processor *wider* and increasing processor throughput. Multi-core CPUs also support SIMD vector instructions; however the instructions are not as wide as what is possible on a SIMD specific processor like a GPU.

GPU Execution Model

The design of a GPGPU application is similar to that of typical serial applications but has a few important differences that allows the programmer to specify the intended vector width and structure of the application. For this work, the terminology of high level descriptions for GPU hardware will follow the definitions of NVIDIA GPUs. Similar terminology exists for the same concepts on AMD GPUs using OpenCL. Figure 1.1 shows the software abstraction of the GPU processor.

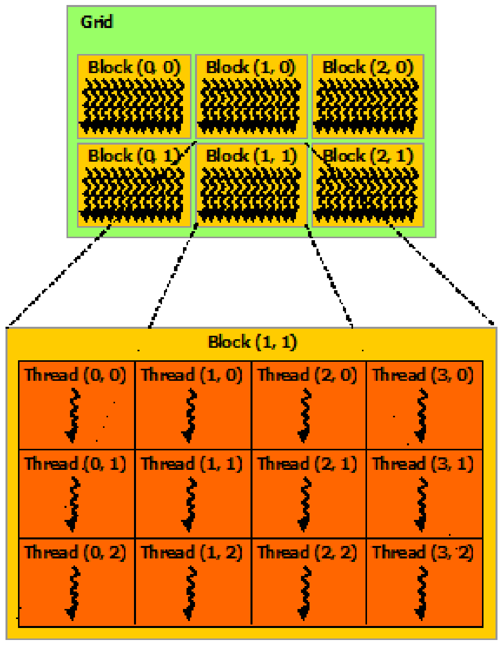


Figure 1.1: Grid of blocks with threads on a GPU. The threads within a block run in SIMD lock-step with one another [25].

For the purposes of this work, one can think of a GPU *kernel* as the entire application, which contains a *grid* of multiple *blocks* running the user's application. Each block executes a SIMD vector of *threads*. All blocks of a kernel execute asynchronously of one another.

Blocks have access to a common GPU global memory. Limited communication between blocks is possible through global memory, but since blocks execute asynchronously, communication introduces challenging concurrency issues. The memory of the CPU and GPU are separate from one another, so any updates between the two spaces must be done manually. It is best practice to transfer large chunks (on the order of hundreds of MBs) between the host (CPU) and device (GPU) since the GPU memory is designed for large data transfers.

In terms of physical hardware, the GPU has multiple *streaming multiprocessors* (SMs), with each SM executing a block at a time and switching between the execution of different blocks as needed. In general, SMs are waiting on various resources for the blocks currently running. To saturate the computational units of the SMs, more blocks than SMs are instantiated per kernel. Depending on the application, resources may be allocated per block, so using more blocks than the number of SMs occupies more of the GPUs memory.

1.1.2 Streaming Dataflow Computations

The streaming dataflow abstraction defines applications as a sequence of *independent operations* which stream data to one another at some regular interval [31]. Each independent operation performs some type of work, such as a filter on input data; the results of an operation are then sent to later operations in the sequence. In the particular model described in [31], applications have a *stable computation pattern* that does not change much if at all over time, which ensures dataflow rates remain static over the lifetime of the application. This form of dataflow application is described as *Synchronous Dataflow* (SDF) [19], where dataflow rates are statically mapped to the operations and known *a priori*. Large (sometimes infinite) streams of data enter the application from some external source. Example applications include the Bluetooth communication protocol [11] and the GSM Vocoder [24].

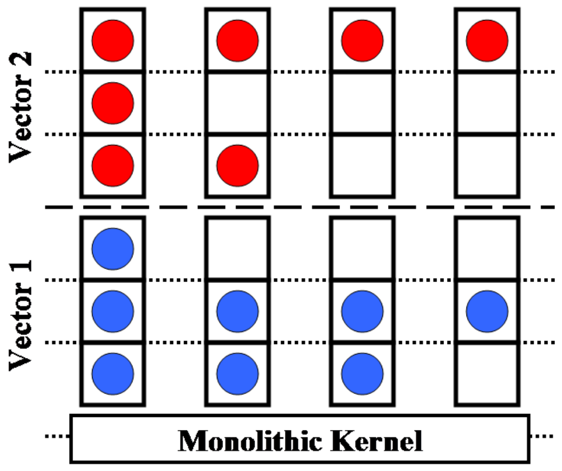
Since the SDF paradigm is designed for very large datasets, and operations are independent of one another with each producing a known number of outputs, a SIMD target processor like a GPU is well-suited as a target for SDF applications. The authors of Brook for GPUs [5] identified this use case and implemented the Brook language, an SDF implementation for GPGPU programming. Here, streams are collections of data to be processed by kernels (functions on the GPU). Streams also have *dimensions* which specify the layout and number of data elements processed by a kernel. Typing is effectively restricted to only floats, reflecting the GPU's graphics usage, but today's GPGPU programming languages (CUDA and OpenCL) allow for arbitrary typing.

StreamIT [31] utilizes SDF to statically compute scheduling of specific operations and sizing of intermediate data queues within the application. Statically computing the schedule requires knowledge of how much data is processed and if that data will expand or filter to more or less results at a predictable rate within the application.

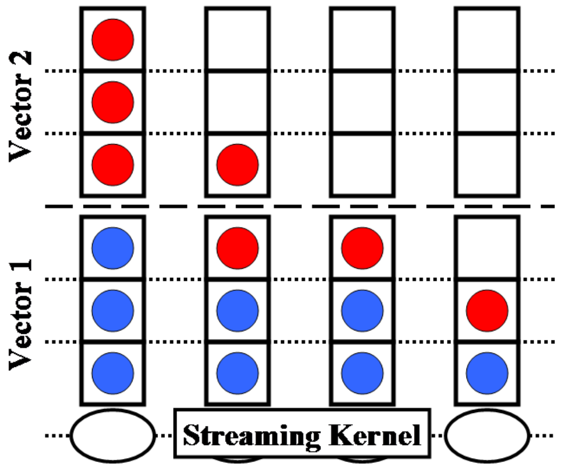
1.1.3 Irregularity in Streaming Dataflow Computations

In contrast to SDF, we define *irregular* dataflow as when the amount of output data items produced per input data item is unknown *a priori*, and varies dynamically and unpredictably from input to input. Because we do not know how many data items will be output from a given input, we cannot determine how large an output queue needs to be when an input is processed. To solve this, we restrict the problem by imposing a maximum number of outputs per input to each operation in the application. Thus, one can at least bound the worst case execution scenario in terms of scheduling.

SIMD by nature wants to have large and dense datasets on which to work, thereby maximizing throughput. When irregularity is introduced, many parts of the input vectors are empty and do not perform useful work, defined as the *occupancy* of a SIMD device. Figure 1.2a shows how irregularity affects the occupancy of kernels. With sparser input vectors the throughput of a SIMD processor is reduced. In Figure 1.2a, the application requires running two separate vectors worth of input at all times, defeating the purpose of using parallel processing in the first place. Mismanaging intermediate data in these applications (i.e. doing nothing to ameliorate the reduced occupancy) further exacerbates the issue, as uncertainty in the filtering and expansion of data by certain operations compounds as each operation feeds into another.



(a) Monolithic Kernel



(b) Streaming Kernel

Figure 1.2: Irregular dataflow and its effects on SIMD occupancy in a monolithic versus streaming kernel. The application filters data throughout the pipeline, causing vacancies to appear in each input vector. In the monolithic kernel, SIMD lanes within the vector are left vacant, requiring two separate passes on the input data at all times. In the streaming kernel, data is queued between computational stages (white ovals), which compacts the results allowing the kernel to run full vector widths of input near the end of the application.

To ease the occupancy concerns within an irregular streaming dataflow application, one can use *intermediate data queues*, which act as a staging area for outputs between operations, such as those seen in our framework [9] and shown in Figure 1.2b. The queues hold data from being processed until a *full* vector width of data is available for the operation. Queueing allows SIMD processors to operate at a high occupancy, which for a fixed volume of data and processing time per vector will maximize throughput. We see in Figure 1.2b with the introduction of intermediate data queues in the streaming kernel implementation, that the number of vectors to process now reduces as data items are filtered rather than staying constant throughout the computation. Intermediate queueing comes at a cost both in time to manage the queues and in memory for storing intermediate data items.

1.1.4 Pipeline Execution Model

Irregular dataflow streaming applications can be described as a *pipeline* of *compute nodes* $n_1, n_2, ..., n_N$ connected by edges that have *queues*. An example is shown in Figure 1.3 below:

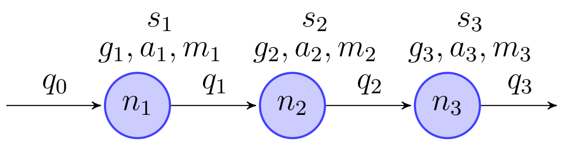


Figure 1.3: A simple pipeline application topology. Node n_1 feeds into n_2, and n_2 feeds into n_3. Node n_i has service time s_i, average gain g_i, maximum gain a_i, and maximum vector gain m_i.

A compute node n_i processes vector widths v of inputs from queue q_{i-1} at a time. Every time a node executes, it consumes up to a vector width v of input and produces a data-dependent number of outputs, potentially of different size and type.

Each edge between nodes has an associated queue q_{i-1}, where the upstream node n_{i-1}'s items accumulate in contiguous slots for node n_i. Once the downstream node executes, the downstream node will pull vectors of up to v inputs from the queue. We assume the queue is of fixed size. While it is possible to resize the queues periodically during execution, the overhead per adjustment is on the order of tens of milliseconds on our platform, so we treat queue sizes as fixed within a single "epoch" of execution for the purposes of analysis.

We instantiate multiple *pipeline replicas* on the GPU at one time. Each pipeline replica has an independent copy of the application pipeline, intermediate data queues, and a *scheduler*. The input data is split evenly among all pipeline replicas. Each pipeline replica runs independently of the others, and using the AFIE scheduling protocol, the replica's scheduler determines which node's code to run based on all the nodes' current available input items and output queue space [27]. Inter-block communication is minimal, which is a design amenable to GPU architectures since little support for cross-block synchronization exists. Each time the scheduler is called, it chooses a node to *fire*. The fired node then consumes vectors of input until there are no inputs available or the downstream queue space has filled. The application finishes when no inputs are left in any queue.

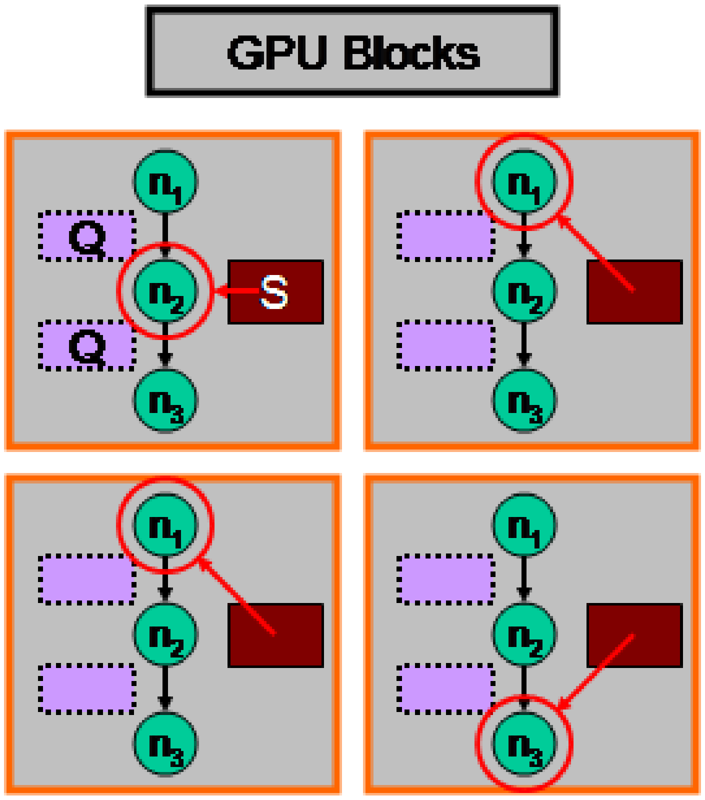


Figure 1.4: Pipeline replicas running on separate blocks. Each block has its own inputs, queue spaces (Q), and scheduler (S). Each block independently decides which node from the pipeline should be run next depending on the number of inputs and the downstream queue space available [27] [9].

When examining runtime characteristics of node n_i there is: **(1)** *service time* s_i, **(2)** *average gain* g_i, **(3)** *maximum gain* a_i, and **(4)** *maximum vector gain* m_i. The service time s_i

defines how long n_i takes to process a vector width v of inputs. If the vector being processed has fewer than v elements, then the node still requires s_i time to process the inputs. The average gain g_i is the average number of outputs per input item consumed, which could be any non-negative integer value. The maximum gain a_i is the maximum possible number of outputs per input consumed. The maximum vector gain m_i is the mode of the maximum number of outputs that any one input in a SIMD vector will produce. Recall, because of the lock-step nature of wide-SIMD architectures, all threads within a block must wait until all other threads are finished processing their input, sans thread-to-work remapping. This makes processing every vector of input take time proportional to the longest running lane m, since lanes within the vector must wait for others to finish.

When modeling application throughput, the *average cumulative gain* $G_k = \prod_{i=1}^{k} g_i$, *maximum cumulative gain* $A_k = \prod_{i=1}^{k} a_i$, and *cumulative max vector gain* $M_k = \prod_{i=1}^{k} m_i$ provide the average, maximum, and mode vector maximum number of outputs from node n_k per input to node n_1. The mean-value behavior of the pipeline is accurately summarized by G_k, with worst case behavior visualized by A_k, and M_k is a heuristic value since it cannot be simply computed even with a detailed distribution of gains. The maximum gain a_i is used to calculate the *minimum safe queue size* downstream of node n_i. For this work, m_i is estimated as the empirical *mode* of n_i's maximum vector gain.

We look at the *average gain* as our characteristic of choice when maximizing throughput because it is easy to collect and it provides a good approximation to actual execution behavior. Depending on the application, there may be better a characterization, such as the *distribution of the gains over time*, but this requires much more in-depth monitoring of the data stream than simply taking the average. Interpretation of other characterizations would provide data stream information at runtime, which our optimizations could use, but our work focuses on optimizing at compile time. For this work, we assume that the average gain is a reasonable snapshot of the execution for all applications, and defer other characterizations to future work discussed in Chapter 6.

1.1.5 Optimization Metrics

When examining irregular dataflow streaming applications, we try to maximize *application throughput*. To this end, we identify some properties of application behavior that are correlated to overall throughput of SIMD applications, taking into consideration our application and architecture models.

First, we want to maintain *high occupancy*, meaning all processors are doing useful work. So long as the overhead of achieving high occupancy is low relative to how much work is done in the application, more work will be accomplished in the same time-span. The AFIE scheduler showed that we can maximize throughput irrespective of overhead costs [27]. We achieve this via the implementation of intermediate data queues which compress data for later execution, as done in previous work [9].

Next, we look at reducing the overall number of *node switches*. A node switch occurs when a node in an application topology must yield execution to the scheduler, such as when the node runs out of inputs to process or a downstream queue has completely filled. Reducing the number of switches reduces the times the queue overhead cost is incurred. Designing a schedule, such as in AFIE, one can reduce the number of node switches to at most a factor of 2 times a clairvoyant schedule [27]. The number of node switches is dependent on the size of queues.

Finally, we examine reducing the number of host-device switches, where the device application must yield to the host. Transferring large datasets between the host (CPU) and device (GPU) is not an expensive operation, but frequent transfers are. This problem affects applications with large search frontiers, where the input data cannot fit in the entire device memory.

As a side problem, we examine reducing the *memory usage* of irregular dataflow streaming applications. There is a question of how small queues can be, where each queue must maintain a minimum queue size based on the scheduling protocol [27] and maintaining high occupancy for high throughput. Minimizing the queue sizes has benefits of not wasting memory on intermediate queues but trades off with an increased number of node switches.

1.1.6 MERCATOR

While the model in the previous section defines pipeline execution for irregular dataflow streaming applications, many streaming frameworks do not accommodate for said irregularity. We use the MERCATOR framework for irregular dataflow streaming applications which is written for NVIDIA GPUs using the CUDA language [9]. The purpose of the framework is to automatically build infrastructure for compacting irregular dataflow outputs from upstream nodes to those downstream via intermediate queue implementations. MERCATOR uses an *uberkernel* model, where a single kernel is run. Blocks within the uberkernel run a separate application pipeline replica as described in 1.1.4. Once all the pipeline replicas have processed all their input data, the outputs are coalesced by MERCATOR and returned back to the host.

One designs their application pipeline via a *skeleton file*, defining compute nodes with their maximum gains, edges connecting the nodes, other node attributes, and other application data. Based on the skeleton file, MERCATOR generates code stubs for the *run* function of each node that is called by the *scheduler* for each pipeline replica, which the user fills out. An example MERCATOR application is listed in Figure 1.5.

A MERCATOR application decides how large intermediate queues should be based on the maximum gain of a given node's output edge multiplied by some constant (default is 4). The user can change this constant at compile time along with the number of blocks, and hence pipeline replicas, the application will use. Changing the exact queue size can currently only be done directly in generated code files, but future implementations will allow this feature in the skeleton file. We address how to determine both queue placement and size in Chapter 3.

From the host side, the user can run and send/receive data from the MERCATOR application. When running input sets that do not entirely fit on the device, one can call a MERCATOR application multiple times with different input data sets without the need to completely tear down and rebuild the kernel on the device.

```
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

Figure 1.5: 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.

1.1.7 Runtime Realization of MERCATOR Applications

MERCATOR schedules nodes at runtime using a protocol developed in our prior work, the AFIE ("Active Full, Inactive Empty") scheduler [27]. 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 will eventually be able to make progress. We proved in [27] 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.

1.1.8 Irregular Streaming Dataflow Application Examples

Many irregular streaming dataflow applications exist across multiple domains, ranging from biosequence analysis, to data integration, to branching search. Listed below are relevant application domains that have been examined to explore the optimizations in this work. Each domain either has irregular dataflow features in some number of applications in that domain (data transformations) or outright exhibits irregular dataflow (filtering applications and tree search). We describe the application domain's task and provide general structural information about inputs/outputs as well as internal data movement.

- **Data Transformations:** Every application requires inputs be structured a specific way. When looking at new, faster algorithms, it is especially important to examine prerequisites for input data structures, as the data pre-processing can take more time than the new algorithm versus the baseline with less pre-processing [6]. Transforming large quantities of data in a timely manner for later, more expensive analysis is critical to the previously cited *Data Integration* tasks.

  Large quantities of independent input data items means there is great potential parallelism to be had. However, some data transformations require either filtering out or expanding input data items, which as stated previously, makes efficient SIMD-parallel data processing more difficult to implement.

  To examine irregular dataflow in this field, we look at one of the applications from the DIBS benchmark which we call Taxi [6]. We describe Taxi in more depth in Chapter 2. Of note in this data transformation, there is a computational stage which enumerates an unknown number of outputs from a single input and others which filter those results, creating both inflationary and filtering dataflows respectively.

- **Filtering Applications:** There are a number of applications which sift through large quantities of data to find good candidate data points. These candidates are then sent through more comprehensive exact analysis to determine if they are valid solutions to the problem at hand. Successive filters reduce the search space but filtering at various data-dependent rates requires extra care when SIMD parallelizing these applications.

  An example in this domain is BLAST [1], or Basic Local Alignment Search Tool, which compares DNA sequences. When comparing two DNA subsequences, one is not searching for exact matches but for long maybe semi-broken matching subsequences such as those that occur with minor genetic mutations. To perform this inexact matching, the Smith-Waterman algorithm is applied, which is a quadratic time operation. Since the Smith-Waterman algorithm is expensive, BLAST looks for candidate subsequence comparisons using cheaper methods first before sending those candidates for processing by the Smith-Waterman algorithm. Multiple different filters are used to determine subsequence comparison candidacy, with the resulting number of outputs from each of the filters unknown *a priori*.

- **Tree Search:** Many applications exhibit tree structured execution paths, with each juncture being an independent sub-problem with a set of possible next actions. These trees can expand exponentially, depending on the choices available for each sub-problem, providing large amounts of data to process. A SIMD-parallel implementation of a tree search can handle the large quantities of data, however maintaining high occupancy is difficult due to the expansionary nature of the problem, especially when the choices available to each sub-problem are variable.

  Branching search applications fall neatly into this category. The classic problem of N-Queens is an example where the application breaks a problem into sub-problems and searches for a solution. N-Queens takes an $N \times N$ chess board and tries to place N queens such that no queen is in line of sight of another. Each sub-problem enumerates all the next viable queen placements based on its current queen placements, both expanding and filtering on the search space. A more detailed description can be found in Chapter 3.

  Branch-and-bound optimizations also exhibit tree search behavior. Here, the problem is broken down into sub-problems. A theoretical bound on the input sub-problem is calculated and used to either prune or continue processing the sub-problem. Another classic problem, the Traveling Salesman Problem (TSP), takes a list of distances between cities and builds the shortest path between all the cities only visiting each city once. TSP can be broken into sub-problems, where each sub-problem tries to add another city to a partial tour of already added cities. If a lower bound calculated based on the current path and the remaining cities to add is greater than the length of the current best tour, then the sub-problem can be pruned from the search since it cannot lead to a better solution. Filtering, or pruning, of sub-problems is highly data-dependent, and expansionary factors near the top of the tree are high with a large number of cities. Further discussion of TSP branch-and-bound optimization is found in Chapter 5.

1.1.9 Research Questions

When examining throughput optimizations for irregular dataflow streaming applications, there are multiple factors which play important roles. **The overarching goal of this work is to determine what characteristics of irregular dataflow streaming applications affect application throughput and how one can maximize their throughput**. To improve the throughput of irregular dataflow streaming applications, the following questions are raised:

How does stateful execution affect irregular dataflow application throughput?
Certain applications require carrying state, such as an item's origin, associated with specific inputs throughout the application. While it is possible to individually assign those values to each input, the amount of space required is very large. Additionally, the same state is commonly shared among multiple inputs, such as in the case of expanding an input to its smaller parts. The time taken to enumerate small objects and copy relevant state to each, which we call *item tagging*, can become an expensive operation, depending on how many items need state information. The state information then needs to be carried by the subsequent smaller parts, adding to copy costs both in time and memory further down the application pipeline.

One can instead centrally share state between constituent parts of a larger object rather than carry a copy of the state with each part, eliminating both the extra memory usage and copying time overhead. However, centralizing the state necessitates that all inputs in a vector have the same originating object, which can reduce occupancy. This work examines the effects of tagging each expanded object's items individually with the origin's values against centrally storing those values.

How do changes to queue sizes affect application throughput?
In our application model, queues are *statically sized* because dynamic allocation is expensive (on the order of 10ms per re-allocation, whereas some applications can finish in hundreds of

milliseconds [35]). Since queues are statically sized, when a queue fills with data one must *switch* execution away from said node, incurring some overhead. When giving queues more memory, the number of times a node needs to switch away to another node is reduced since the node has more space to place outputs. However, giving an arbitrarily large amount of memory to queues is undesirable as that memory could be used for application data such as inputs instead. Thus, based on each application's memory usage (how many queues per pipeline, how many pipelines running concurrently, input data size, etc.), we may allocate more or less memory to intermediate queues. We show how to optimize for the highest application throughput given a fixed size memory allocation to intermediate queues.

What are the tradeoffs of having queues versus not?
The purpose of queueing in irregular dataflow applications is to *compact* results from upstream nodes so that the GPU maintains high *occupancy.* That is, each thread in each block of the application should have input to work with at all times. High occupancy generally leads to higher application throughput. However, queues require maintenance, including compacting results and the subsequent switch to executing a different node. Hence, there exists a tradeoff between having queues (high occupancy, additional overhead) and not (lower occupancy, no additional overhead). We build a performance model to quantify this tradeoff.

What are the effects of minimum queue sizes on performance?
When assigning queue sizes based on the AFIE scheduler to maximize application throughput a minimum queue size must be assigned to prevent deadlock. The minimum queue size of $|q_i| \geq v_{i-1}g_{i-1} + v_i - 1$ must be assigned to each queue q_i [27], which is dominated by the gain of the previous node times the vector width. Depending on the application, the queue sizing optimization may want to assign less queue space than what is safe [35]. In this case, the user must give those queues the necessary space or else risk deadlock. The safety constraint exists because once a node begins running, it cannot stop until all the outputs for the current vector of inputs are sent downstream. If the minimum queue size was not present, then memory could be more accurately assigned to which queues need it the most. This problem is particularly acute with nodes with large maximum gains but few total inputs.

Scheduling behavior using AFIE is pessimistic, in terms of the total number of outputs from a given input vector. Thus, when the maximum gain is very large, but the average gain is much smaller, minimum queue sizes are greatly inflated from what the queue size optimization suggests. This also leads to more switches away from a node, since the queue is considered “full” after a single vector of input.

How do input size and application topology affect iterative expansionary applications?
Iterative expansionary applications can be designed as a pipeline of compute nodes where each node represents an iteration of the application. As the name implies, they produce an exponentially growing number of outputs as data is processed. Branching search, and in particular *branch-and-bound* optimization problems, fall into this category and have irregular dataflows that are influenced by the value of the current best solution at any point during execution. These applications have large search frontiers that cannot fit in GPU memory all at once. Hence, after a certain number of iterations, intermediate results must be returned to the host and enqueued there for later investigation. Previously, we focused on optimizing the throughput of a single run of the GPU application; for this question, we must additionally consider the impact of host-device interaction and constraints on the GPU’s space to store intermediate results, as well as how fast better solutions are found, which increase pruning during the search.

1.1.10 Contributions

To answer the aforementioned research questions, the following contributions of this work are made. Throughout, implementations and empirical validation were performed with MERCATOR on NVIDIA GPUs; however, our models and designs are not solely tied to MERCATOR but are of general applicability to streaming irregular dataflow on a SIMD architecture.

A correctness-validated stateful execution paradigm
Input data in irregular dataflow streaming applications typically has complete independence from one another. However, sometimes a stream is divided into regions whose boundaries are determined by events, such as a parameter change. Execution may depend on which region of the stream an item is in. With this work, we introduce a *stateful execution* paradigm for irregular dataflow streaming applications where execution state is data-dependent and

affects decision making. We test stateful execution using *enumeration and aggregation*, a system designed to expand sub-elements from an input object, work on those individual sub-elements, and aggregate a result from the sub-elements. We develop a novel *credit system* where each node in the compute pipeline keeps track of the number of data elements in a parent object that was enumerated. The credit system carries the parent object through the pipeline to where the sub-elements are. We empirically evaluate the overhead of our credit system against an alternative sub-element tagging system on a data integration task TaxiApp [6]. We find that the overhead of our credit system is less than that of tagging every sub-element for state in most cases.

A node-merging heuristic for determining whether to implement queues, and an accompanying method for queue space allocation for maximizing throughput
Queues in irregular dataflow streaming applications are present between compute nodes to increase the occupancy of the nodes when run. While a high occupancy is ideal for wide-SIMD execution, the overhead of maintaining those queues can outweigh the benefits of fitting more inputs into a vector. Using execution statistics of the application pipeline, namely the average execution time of a node on a single vector of input and the cumulative gains, one can determine where the overhead of queues will be outweighed by the occupancy improvement for specific nodes. Once those beneficial queue placements have been found, one can use those same execution statistics to allocate queue space so as to maximize pipeline throughput, subject to an overall limit on total memory allocated to queues. We design a heuristic for determining where queues should be placed in a pipeline and show how to allocate queue space based on those same easily acquired node execution statistics. We empirically test on benchmark applications including BLAST [1] and the classic N-Queens problem. We find that our queue placement heuristic provides a good set of candidate application configurations and that our queue space allocation algorithm does indeed maximize throughput.

An interruptible node implementation for pessimistic scheduling behavior and minimizing safety constraints on queue sizes
When looking at sizing queues, the scheduling policy that is used (e.g. the AFIE [27] scheduling policy) dictates when enough input has arrived to a node for that node to run. In particular, when a vector of inputs is run on a specific node, that node must ensure that there

are enough output spaces available for what the input vector may produce. However, if a node's gain is very large, the number of output spaces available must be equally large for any time an input vector is processed. This can lead to two issues: (1) required queue space can be much larger than the amount of data a queue ever sees when most output gains are not at the maximum, and (2) scheduling behavior becomes pessimistic when the maximum number of outputs per vector of input is close to the total queue size. To alleviate both of these issues, we implement *interruptible nodes* that can pause and resume execution in the middle of a vector of inputs, allowing nodes to yield to the scheduler when their output queues actually fill, rather than speculating on how much more output they could produce. We examine pessimistic scheduling behavior in both the classic N-Queens problem and BLAST [1], which both show performance improvements when queues are small, maximum gains are large, and average gains are smaller than the maximum gains. We show that memory usage can be reduced or re-applied to other queues in the system when making nodes interruptible.

An evaluation of input size and pipeline topology for maximizing throughput of iterative expansionary applications

We design a combined CPU/GPU implementation of branch-and-bound search to solve the Travelling Salesman Problem. Because the potential number of outputs from the search is so large, performing the search as a single, monolithic GPU pipeline storing intermediate results is not feasible. Instead, we split the search among multiple GPU *steps*, each of which processes a contiguous set of levels in the search tree. Steps higher in the tree (i.e., closer to the root) generate output that is queued on the CPU and later passed to steps lower in the tree. The CPU is then responsible for scheduling execution of steps, in particular deciding which of several steps with available input should be run next. We propose a strategy to select which step to run next that balances GPU occupancy considerations with the need to reach the bottom of the tree quickly to more rapidly update the incumbent best solution.

Two application parameters – the pipeline length (i.e., number of tree levels) within one step, and the number of distinct sub-problems given as input to a step – influence the maximum number of outputs the step may produce, the GPU occupancy of the step, and the frequency with which control must be returned to the CPU. We examine empirically how input size and step pipeline length affect application throughput in our TSP implementation.

1.1.11 Outline

This dissertation is organized as follows. Chapter 1 is this chapter, and describes the application model used to describe irregular dataflow streaming applications, and related systems along with defining metrics which characterize throughput and memory requirements for irregular dataflow streaming applications and associated research questions. Chapter 2 examines how stateful execution is useful for irregular dataflow streaming applications, both for increased throughput and lower memory consumption. Chapter 3 explores when queueing overheads are worth taking for occupancy gains and how resizing queues based on mean execution information improves throughput. Chapter 4 shows how the node scheduler in our model can be overly pessimistic in queue sizing and node switching because of minimum queue sizes, with a solution to reduce the minimum queue size significantly. Chapter 5 examines the throughput of differing input size and pipeline length on branching search applications. Finally, Chapter 6 concludes and provides future work.

Chapter 2

Stateful Execution

This work originally appeared in the Proceedings of 13th International Workshop on Programmability and Architectures for Heterogeneous Multicores, Bologna, Italy, January 2020 [33].

2.1 Introduction

When items in a data stream cannot be processed independently, the computation is considered *stateful*. To avoid the need for full serialization, we focus on a common scenario in which the input stream is divided into variably-sized *regions*. Items in one region are processed independently of each other but in a common context. For example, a stream of characters may be grouped into lines or network packets; a stream of edges in a graph may be grouped by their source vertex; or a stream of measurements may be grouped by a common time window or event trigger. Region boundaries are state-change events for the stream — items after a boundary must be processed differently than items before it.

In this chapter, we investigate mechanisms to support streaming computations with region-based contextual state on SIMD-parallel platforms. Our contributions are threefold. First, we describe a low-level mechanism for precise delivery of control signals between pipeline stages of a streaming application. This mechanism, unlike those described in some prior work (e.g. [32]), supports irregular dataflow. Second, we use this mechanism to construct an abstraction, *enumeration and aggregation*, that lets application developers express region-based contextual state as part of a streaming application. Finally, we implement our designs in MERCATOR to investigate the performance implications of regional context, exposing a SIMD-specific performance tradeoff between alternate ways of implementing this behavior in applications. The two strategies we examine trade off between SIMD occupancy and representation overhead.

The remainder of this chapter is organized as follows. Section 2.2 describes the application and architectural models in which we formulate our work and considers related work in other streaming models. Section 2.3 describes our protocol for synchronizing control signals with a data stream. Section 2.4 describes the developer-facing abstraction of enumeration and aggregation, which we implement in terms of signals. Section 2.5 investigates the performance of our designs, while Section 2.6 concludes and considers future work.

2.2 Background and Related Work

2.2.1 Application Model

Expanding upon our description in Chapter 1, a streaming application consists of a pipeline of *compute nodes* connected by fixed-sized *data queues*. A node consumes a stream of *data items* from its input queue and produces a stream of data items (perhaps of a different type) at its output that are queued for processing by the next node downstream in the pipeline. When a node is executed to consume one or more inputs, we say that the node *fires*. Each input data item consumed by a node causes it to produce zero or more outputs. The number of outputs may vary for each input consumed, up to some node-dependent maximum, and is not known prior to execution. Figure 2.1a shows a simple application pipeline with three nodes. While we mainly discuss linear pipelines in this work, our contributions also apply to tree-structured topologies like those in Figure 2.1b.

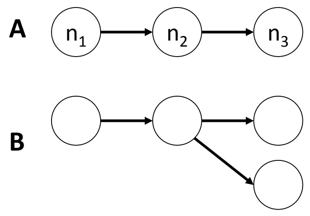


Figure 2.1: (a) Streaming computation pipeline with three compute nodes; (b) Pipeline with a tree topology.

We do not consider DAG-structured topologies because the semantics associated with convergent edges are complex under irregular dataflow, even in the absence of stateful execution [20]. Topologies with cycles have clearer streaming semantics, but in the presence of irregularity, items in the stream can be reordered if they take different numbers of trips around a cycle. For such topologies, maintaining precisely ordered control boundaries in the stream requires aggressive reordering that is beyond the scope of this work.

An application is provided with an initial stream of inputs to its *source node*. When the application executes, a global *scheduler* repeatedly chooses a node with one or more pending inputs to fire. Because of our architectural mapping below, we assume that only one node fires at a time and that a node cannot be preempted while firing. The scheduler continues to select and fire nodes until no node has any inputs remaining.

The major extension to the application model in this work is the addition of *signals*. A *signal* is a control message generated by a node for consumption by its downstream neighbor. When a node receives a signal, that node can change its state and may also generate additional signals to the next node downstream. Signals must be delivered *precisely* with respect to the stream of data items. Formally, suppose we have two successive nodes n_1 and n_2 in a pipeline. If n_1 emits a data item d, followed by a signal z, followed by another data item d', then n_2 must receive z after processing d but before processing d'.

2.2.2 Related Work

As described in Chapter 1, several systems and languages have been developed to express streaming dataflow computations. One of the most influential such systems is StreamIt [31], which implements the synchronous data flow (SDF) abstraction [19]. StreamIt assumes a fixed number of outputs per input to a compute node. This assumption allows StreamIt to offer a powerful abstraction, *teleport messaging* [32], in which signals can flow both forward and backward in a pipeline. StreamIt can also schedule a signal to be delivered to an arbitrary destination node at some precise future time, rather than forcing the signal to flow through the pipeline.

Many capabilities of teleport messaging rely on the underlying SDF model. In contrast, our model does not assume a fixed number of outputs per input and so requires a different design to ensure precise signaling. Our work therefore extends precise signaling capabilities from regular to irregular streaming applications. Like StreamIt, we choose to send signals "out of band," in our case via parallel control edges, rather than attempt to enqueue them together with data items.

Other streaming systems, such as Ptolemy [14] and Auto-Pipe [8], support a variety of dataflow semantics, including multiple, differently-typed dataflow edges between a pair of nodes. This support is in principle sufficient to implement a control channel for signals. However, except in restricted cases like SDF, the systems do not specify how multiple channels between the same two nodes are synchronized and so do not by themselves support precise signaling.

Our work is influenced by a control messaging protocol developed by Li et al. [20]. That work, however, incurs additional complexity to support asynchronous streaming dataflow and to impose well-defined semantics on convergent dataflow edges in an DAG-structured irregular application. We preserve their idea of a credit protocol for synchronizing data and control streams but realize this idea in a way that is efficient for our target model and architecture.

The idea of processing part of a stream of items in a common context is similar to facilities present in Apache Spark [42]. Spark streams consist of a sequence of RDDs (Resilient Distributed Datasets) [41], which are discrete data sets, processed as a unit, that may contain multiple elements. Our abstraction supports some operations semantically similar to Spark's but realizes them in the context of a single wide-SIMD processor rather than the multicore and distributed systems that Spark targets.

Other frameworks supporting streaming-like behavior, such as CnC-CUDA [15], utilize an "in-band" approach with *control collections* that mix control and data into a single stream. Control collections are analogous to our regions of items with a common context. Their implementation requires a tag for each item to track the region associated with it. In contrast, our implementation keeps region boundaries synchronized with the data stream without the need for tagging. We compare these two approaches in Section 2.5.

2.3 A Mechanism for Precise Signaling

In this section, we describe how to synchronize data and control signals between two successive nodes in a streaming pipeline. The mechanism bears some similarities to control flow in networking. We state the correctness properties of our design and provide their proofs.

2.3.1 Credit Protocol for Synchronizing Signals

Let n_1 and n_2 be successive nodes in a pipeline, connected by data queue Q. We add a separate, finite-sized *signal queue* Z between the nodes, as in Figure 2.2a. Data items are moved from n_1 to n_2 on Q, while signals are moved on Z.

We must ensure that, although data and signals move on separate queues, their movement is synchronized so as to ensure precise signal delivery. For this purpose, we introduce a *credit protocol* between n_1 and n_2. Each signal created by n_1 is assigned an non-negative integer amount of *credit*, which is transmitted along with the signal on Z. Credit records a number of data items that n_2 must process before it can receive the signal.

When n_1 emits a signal z, it uses two rules to set the credit associated with z. (1) If no signal is currently queued on Z, then z gets an amount of credit equal to the number of data items queued on Q. (2) If one or more signals are queued on Z, let z' be the signal at the tail of Z. Then z gets an amount of credit equal to the number of data items emitted by n_1 since z' was enqueued. Node n_1 maintains a counter of emitted data items, which is reset each time it emits a signal, that is used to implement the second rule.

The downstream node n_2 maintains a *current credit counter*, initially set to 0, that tracks the number of items that can safely be consumed before processing the next signal. Node n_2 uses the following two rules to determine whether to process data or a signal when it fires. (1) If no signal is queued on Z, n_2 may freely consume any available data items on Q without regard to the counter. Otherwise (i.e., a signal *is* queued on Z), (2a) if the current

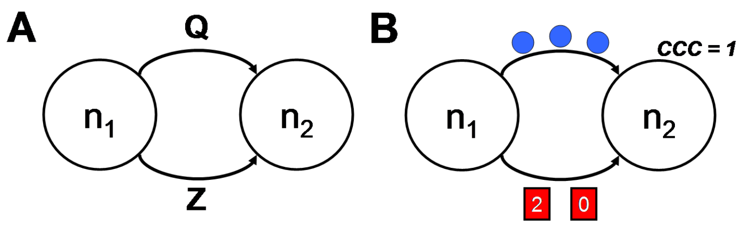


Figure 2.2: (a) Two nodes with data and signal queues Q and Z between them; (b) A possible state of the signal protocol, showing the credit associated with each signal (lower edge) and the current credit counter (right). n_2 may consume one data item from Q before the first signal, then another two before the second signal.

credit counter is non-zero, n_2 may consume only a number of data items less than or equal to the value of this counter, which is decremented once for each data item consumed. (2b) If instead the current credit counter is 0, let z be the signal at the head of queue Z. If z carries more than 0 credit, that credit is removed from z and added to the current credit counter. Otherwise, n_2 consumes z.

Figure 2.2b illustrates how the credit carried in the signals and in the receiving node's credit counter synchronizes the two queues.

Lemma 2.3.1. *A signal z emitted by n_1 is received by n_2 precisely when n_2 has consumed all data items emitted by n_1 prior to z.*

Proof. We proceed by induction on the number of signals already on the signal queue Z when z is emitted.

Suppose Z is empty when n_1 emits z. All items emitted by n_1 prior to s either have been consumed by n_2 or are present on Q. The protocol assigns z an amount of credit equal to the size of Q. This amount is then added to n_2's current credit counter, which was previously 0. Finally, n_2 consumes z precisely when its current credit counter returns to 0, which happens once n_2 consumes the items that were present on Q when z was emitted.

Now suppose z is emitted when Z is not empty. z receives a number of credits equal to the number of items added to Q since the prior signal z'. We know inductively that n_2 consumes z' precisely when all data items emitted prior to z' have been consumed. At this point, the only items on Q must be those emitted after z' but before z, and n_2's current credit counter is 0 since it just consumed a signal. Conclude that n_2 will transfer the credit in z to its current credit counter and will then consume exactly those data items emitted after z' but before z before it consumes z itself. □

2.3.2 Scheduling Applications with Signals

A firing of a node n proceeds in two phases: a *data phase* and a *signal phase*. In the data phase, n consumes as many queued data items as it can. The number of items consumed is limited to the minimum of three values: the number of queued items, the amount of space in n's downstream queue, and (if a signal is pending for n) the value of n's current credit counter. Once n can consume no more data, if its current credit counter is 0, it enters the signal phase, in which it consumes as many queued signals as it can. Signal processing ends when no queued signals remain, or when n's current credit counter becomes > 0 (and hence data must be consumed prior to the next queued signal).

A node is *fireable* if it has either data or a signal pending, and if there is sufficient space in its output queue to hold any outputs from the firing. The maximum number of output data items per input item is known *a priori* for each node, so the scheduler can determine whether at least one data item can be consumed given the available space on n's downstream data queue. The maximum number of *signals* emitted per data item or signal input is also known *a priori*, so a similar determination can be made given n's downstream signal queue. The scheduler repeatedly chooses some fireable node and fires it until no node has queued data or signals remaining.

Lemma 2.3.2. *Under the given firing/scheduling policy, an application pipeline always finishes execution in finite time and so cannot deadlock.*

Proof. Our proof of deadlock-freedom relies on the following two claims.

Claim: A node cannot have a current credit counter > 0 without a pending data item.

Proof of claim: Suppose that a node's current credit counter is > 0. The credit in the current counter was transferred from the signal z currently at the head of the signal queue; it cannot remain from prior signals because the node did not even check for z until its current credit counter last became 0. Hence, this credit was assigned to z to cover data items that were enqueued at the time that z was issued. Since not all credit has yet been consumed, at least one of these items is still enqueued.

Claim: If a node n has either pending data or a pending signal, one of the following holds: (1) n can consume a data item; (2) n can consume a signal; (3) n is blocked due to insufficient space in its downstream queues.

Proof of claim: The node either has credit or not. If it has credit, then by the previous claim it has pending data that can be consumed. If it has no credit but has a pending signal, then either the signal can be consumed, or credit can be transferred from the signal; in the latter case, there must again be data corresponding to this credit. If there is no credit and no pending signal, then there must be pending data, which can be consumed without credit in the absence of pending signals. In all cases, the node can consume some input unless its downstream queues lack sufficient space.

We now proceed to prove the original lemma. If any node in the pipeline has pending data or a signal, then let n be the last such node. Either n's downstream queues are empty (else its successor would have pending data or signals), or n has no successor, i.e., it is the last node in the pipeline, which has unbounded output space and cannot block. Hence, node n is not blocked on its downstream queues and so, by the second claim, can be fired to consume input.

We conclude that the application terminates only when all nodes have exhausted their inputs. $\square$

2.3.3 SIMD Extensions

The above description assumes that nodes process input data items one at a time. However, on a SIMD-parallel processor, a node may process an ensemble of multiple items at once. Because a signal updates the state of the receiving node, we must ensure that *items appearing before and after a signal in the stream are not processed in the same input ensemble.* Hence, if a signal is queued for a node, the system must limit the size of the node's input ensemble to the value of its current credit counter. This requirement may adversely impact SIMD occupancy if signals occur frequently; we study its impact in Section 2.5.

2.4 Regional Context via Enumeration and Aggregation

We now describe a developer-facing abstraction, *enumeration and aggregation*, that allows application developers to describe streaming computations in which regions of a stream must be processed in a common context. We have implemented this abstraction as an extension to our MERCATOR system, building on the work of the previous section.

Our abstraction assumes that regions of a stream with a common context are represented as *composite objects*, similar to the RDDs of Apache Spark [42]. The actual input provided to the application is a stream of such objects. Each object may contain zero or more *elements* of a common data type. For example, an object could be a line of text whose elements are characters, or a vertex whose elements are its adjacent edges, or a list whose elements are numbers. Objects within the same data stream may contain different numbers of elements.

At a given point in the application's pipeline, the developer may choose to "open" the stream of composite objects to create a stream of all their elements. We call this opening process *enumeration* of the objects. The enumerated element stream becomes the input to the next node in the pipeline. In this and subsequent nodes, the developer may access the *parent object* that gave rise to an input item to obtain context needed for its processing.

The opposite of enumeration is *aggregation*, which "closes" the context associated with a parent object. The developer may choose to emit a stream of results derived from individual elements, stripped of their parent context, or to aggregate values computed from the elements of each parent object (e.g. by summing them) and emit a single result per parent. Either way, the stream of results continues down the pipeline.

2.4.1 Developer Interface

To make these ideas concrete, consider the simple application whose topology is sketched in Figure 2.3. A stream of objects of type `Blob`, each containing a collection of numbers, flows from the source node. The Blobs' elements are enumerated, and node f does some computation on each number in the element stream, producing a (possibly shorter) output stream of numbers. These results are passed to node a, which sums the results from each Blob and sends a stream of per-Blob sums to a sink node.

The listing of Figure 2.4 specifies the application's topology. For each node, we specify the types of its input and output streams. The `enumerate` keyword at the input to node f indicates that Blobs are to be enumerated starting there; subsequent data types in the enumeration region of the pipeline are labeled `from Blob`, indicating that items are to be processed in the context of their parent Blobs. Aggregation occurs at the output of node

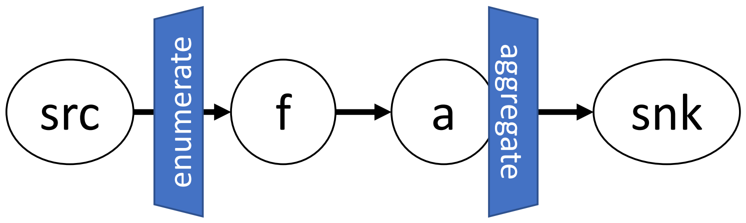


Figure 2.3: A pipeline with enumeration and aggregation. The computation enumerates composite objects drawn from an input source, acts on their elements with a filtering node f, aggregates the filtered values in an accumulator node a, and writes the accumulated value from each object to an output sink.

a, where the `aggregate` keyword indicates that a produces (up to) one `double` value per parent object rather than per element.

Figure 2.5 shows code to implement the application. For each node, there is a `run()` function that processes items in the node's input stream. Output from a node is generated via the `push()` function; because the application is irregular, not every input might produce an output.

The listing shows several functions specific to enumeration and aggregation. `findCount()` is called once per parent object to determine how many elements it contains. The `begin()` and `end()` functions, which may be defined for each node receiving enumerated inputs, are executed before and after the region of the stream associated with each parent object, respectively. The parent object associated with a node's current input is accessible via `getParent()`.

Enumeration produces a stream of sequential *indices* of elements in each parent object. However, the application developer is responsible for providing code to extract the elements from an object. This design allows MERCATOR to remain ignorant of how objects are organized internally.

2.4.2 Implementation

The MERCATOR system takes in an application topology, as shown in Figure 2.4, and produces stubs for all the functions shown in Figure 2.5. The user then fills in the function bodies with the actual code of the application.

```
Node src  : Source<Blob>;
Node f    : enumerate Blob ->
  float from Blob;
Node a    : float from Blob ->
  aggregate double;
Node snk  : Sink<double>;

Edge src -> f;
Edge f -> a;
Edge a -> snk;
```

Figure 2.4: Application topology specification illustrating enumeration and aggregation.

```
void enumFor_f::findCount(Blob *b)
{ return b->nElements(); }

void f::run(int i)
{
  Blob* b = getParent();
  float v = b->getItem(i);
  if (isGood(v)) push(3.14 * v);
}

void a::begin(Blob *b) { acc=0.0; }
void a::run(float v) { acc+=v; }
void a::end(Blob *b) { push(acc); }
```

Figure 2.5: Application code, with stubs generated from topology and developer-supplied function bodies.

We note that *the code shown is CUDA, not C++*; hence, the `run()` functions are actually called not with a single input but with a SIMD ensemble of items in multiple threads, which execute the function body in parallel for each item. (The accumulation in node a would in practice be implemented atomically or with a SIMD-parallel reduction.) MERCATOR provides the runtime infrastructure needed to transfer data from one node to the next and to schedule nodes.

The signaling mechanism of the previous section is key to enabling enumeration and aggregation. At the point of enumeration, the runtime generates a data stream of element indices together with signals indicating the start and end of each parent object's elements. Downstream nodes intercept these signals in order to update their current parent object and to call the `begin()` and `end()` stubs at the right times. Because data before and after a signal is never processed in the same SIMD ensemble, operations on different parent objects' elements always happen in separate calls to a node's `run()` function, and the result of `getParent()` is the same for all items in an ensemble.

2.5 Results

We implemented both our precise signaling infrastructure and the enumeration and aggregation abstraction as extensions to our MERCATOR framework and studied their performance on several benchmark computations. Since the original publication of this work, many improvements have been made to both GPU hardware and to MERCATOR (namely the introduction of the AFIE scheduler [27]) to reduce the framework's overhead. We therefore reran the experiments from our earlier paper using the most recent version of MERCATOR and a newer GPU – an NVIDIA A40 GPU (84 processors) – using as many active blocks as could fit on the device and a SIMD width of 128 threads per block. Code was compiled using CUDA v11.8 under Linux. The new features were implemented such that MERCATOR applications that do not use them do not compile those features, therefore not affecting the runtimes of non-stateful applications.

Cost of Regional Context Abstraction To characterize the performance impact of regional context, we began with two simple benchmark computations. Each benchmark operates on a large array of integers in GPU memory, and each divides this array into a series of regions. The computation enumerates each region, sums its elements, and produces a stream of per-region sums. In the first benchmark, the regions are of uniform size; in the second, the size of each region is chosen uniformly at random between 0 and a specified maximum.

Figures 2.6 and 2.7 show the time to process an array of 2 billion integers in each benchmark as a function of the region size (fixed in the first figure, maximum in the second). Focusing first on the test with fixed-sized regions, we see that execution time decreases sharply as the region size grows from 32 to the SIMD width of 128, then decreases more gradually for larger sizes. This decrease reflects the lower frequency of region boundary signals relative to the data stream as the region size increases. For region sizes on the order of several hundred of elements or more, the abstraction overhead is small relative to the total cost of execution.

We see that when the region size is less than the SIMD width, *every* ensemble becomes non-full, which explains the large performance impacts seen at region sizes below 128. Going above the region size diminishes the negative affects of non-full vectors.

Figure 2.7 shows a much-reduced impact of small size variations on throughput. Unlike the previous benchmark, but more typically of real-world irregular applications, the region size is not fixed. Since the sharp peaks in execution time no longer exist from execution boundaries, the reduced throughput for worst-case region sizes have already been smoothed

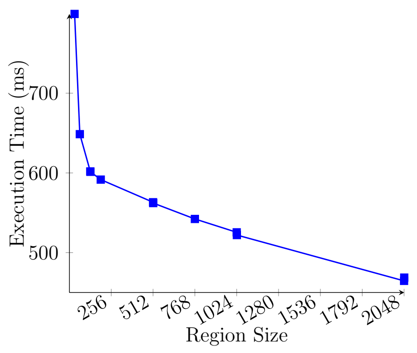


Figure 2.6: Execution time vs. region size for sum app with fixed-size regions.

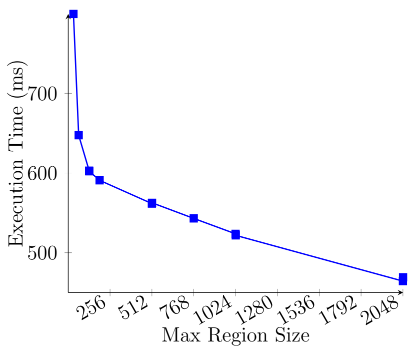


Figure 2.7: Execution time vs. max region size for sum app with variable regions.

out by improvements to GPUs and the latest version of MERCATOR like before. The dominant effect remains: larger region sizes incur less abstraction overhead.

Comparison of Mechanisms for Communicating Context For our second experiment, we implemented a real-world application taken from the DIBS benchmark set [6], a suite of applications representative of data integration workloads. The application, which DIBS calls tstcsv->csv but we refer to hereafter as "taxi," processes a sequence of lines of text, each of which contains a tag, a variable-length list of GPS locations specified as real-valued coordinate pairs, and other data. The goal is to parse each coordinate pair, swap the elements of the pair, and emit the pair together with the tag corresponding to its source line.

Our initial implementation of the taxi application operates on the raw text in GPU memory. It takes as input a stream of line start indices and line lengths. For each line, the first stage of the application enumerates the line's individual characters as a stream, checks them in parallel, and retains only those character positions (identified by an open-brace character) that likely mark the start of a coordinate pair. The second stage verifies, again in parallel, that each open-brace indeed marks a coordinate pair and, if so, parses the pair's coordinates. Each line's tag is parsed once when the line is first enumerated and is then used to mark each parsed coordinate pair for that line.

The first series of Figure 2.8 (square points) shows the execution time of the taxi app as a function of its input size. Larger file sizes were obtained by replicating the input file from DIBS multiple times. Mindful of the relationship between abstraction penalty and region size, we then investigated how the input data in the taxi app determined SIMD occupancy.

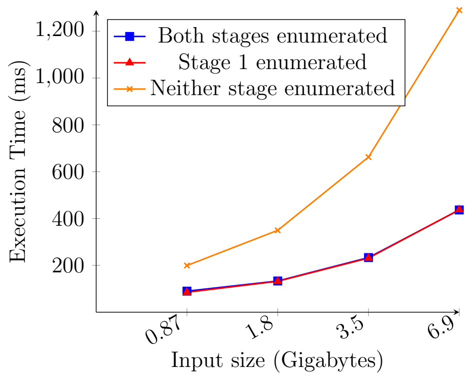


Figure 2.8: Execution time vs input size for three versions of the taxi app. Both stages enumerated and only stage 1 enumerated coincide with one another.

Input lines have an average length of 1397 characters, so regions corresponding to each line in stage 1 are large, and the penalty to occupancy is expected to be low. In contrast, lines contain on average only 45 coordinate pairs, less than the SIMD width, and so would be expected to incur a large penalty to occupancy in stage 2, whose region size is determined by the number of pairs per line. Indeed, we found that stage 1 was fired with full SIMD ensembles 93% of the time, while stage 2 had full ensembles only 14% of the time.

When regional context changes frequently relative to the SIMD width, the occupancy cost to performance of our implementation may exceed the cost (mainly extra memory accesses) of replicating this context along with every data item. We therefore developed a second version of the taxi app that used enumeration to provide context in stage 1 but explicitly marked each open-brace with its line's tag before sending it to stage 2. The latter stage does not utilize the enumeration abstraction and so can process items from multiple lines in one ensemble, achieving essentially full SIMD occupancy. The second series in Figure 2.8 (triangular points) shows that removing context for stage 2 does not provide a meaningful performance improvement, whereas our original work did. This is most likely due to a combination of the same improvements to GPU hardware and the MERCATOR scheduler. However, using the same strategy to tag each character of each line in stage 1, while it slightly improves occupancy by avoiding enumeration entirely, incurs substantially more overhead due to the much greater number of elements to be tagged per region. The third series (x points) shows that, at the largest input size tested, a pure tagging implementation is roughly 3× slower than one that judiciously uses either tagging or our design as appropriate for each stage.

We conclude that the best way to provide regional context to streaming applications on a SIMD architecture depends strongly on the performance tradeoff between reduced SIMD occupancy and reduced representation overhead. Each stage of a pipeline may represent a different point in this tradeoff, and the highest-performing implementation (dense or sparse) for regional context may therefore vary between stages. Ultimately, this choice should be made transparent to the application developer based on profile-guided feedback.

2.6 Conclusion and Future Work

We have described an abstraction, enumeration and aggregation, to support stateful streaming computation based on regional contexts. We presented an implementation of this abstraction for irregular streaming computations on SIMD-parallel architectures such as GPUs. Our abstraction relies on a sparse implementation of precise signal delivery between computational stages.

We characterized the cost of the abstraction on benchmark computations and demonstrated that the best strategy for realizing it may depend on the relationship between region size and the architecture's SIMD width. Future work will include more careful modeling and/or empirical measurement of the costs of alternative implementations, with an eye toward allowing the MERCATOR runtime to transparently choose between strategies based on the typical number of elements per region.

Another direction for future work will investigate how to lower the abstraction penalty of precise signaling for SIMD occupancy. When the effects of a signal on a node's state are limited and well-defined (e.g. changing the parent object pointer), the node may be able to compute the correct state (pre- or post-signal) to expose to the item in each SIMD lane separately. Computing the correct state per item in each node, rather than storing it with items in the queues between nodes, would offer the same efficient representation of state as in our design while eliminating signals' cost to SIMD occupancy.

Chapter 3

Queue Space Redistribution and Node Merging

This work originally appeared in the Proceedings of IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3), November, 2020 [36]. A later journal publication appeared in the Parallel Computing Journal, Volume 109, 2022 [35].

3.1 Introduction

As discussed in Chapter 1, 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 [31]. But some streaming computations, including the examples cited in Chapter 1, 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.

To improve the occupancy of irregular dataflow pipelines, the application model of Chapter 1 places 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. Our application mapping described in Chapter 1 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 [27], 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 tradeoff 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 this chapter is organized as follows. Section 3.2 examines related work. Section 3.3 explains our application model and associated performance metrics. Section 3.4 motivates our optimizations of queue space distribution and queue placement. Section 3.5 provides a method for partitioning space among queues in an application so as to minimize overhead given a limited space budget. Section 3.6 describes a model for estimating the performance consequences of adding queues between compute nodes. Section 3.7 evaluates both of our techniques on irregular streaming applications implemented in MERCATOR. Finally, Section 3.8 concludes and explores future work.

3.2 Related Work

3.2.1 Queueing Optimizations for Streaming Pipelines

As discussed in Chapter 1, many application frameworks have been developed to support *regular* streaming dataflow applications on parallel systems. A prominent example, StreamIt [31], was built around the synchronous data flow (SDF) [19] 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 [30] 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 [3] 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.

3.3 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 Figure 1.3 from Chapter 1, an application is designed as 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* s_i, its *average gain* g_i, and its *maximum vector gain* m_i. As a reminder from Chapter 1, service time s_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 less or greater 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 3.6, max vector gain can be a useful predictor of computational cost in the absence of remapping through queues.

For convenience and as described in Chapter 1, we define the *cumulative average gain* G_k and *cumulative max vector gain* M_k. Each edge between two nodes also has an associated queue. There is a maximum gain a_i which dictates the minimum amount of downstream queue space required to run a node safely. Our target architecture, the GPU, runs multiple pipeline replicas on each of the processors, each with some share of the inputs to the application.

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.

3.4 Motivation for Optimizations

As previously discussed in Chapter 1, 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 memory 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 3.5.

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 3.6 seeks to model this tradeoff and quantitatively guide where queues should be placed.

3.5 Choosing Sizes for Finite Inter-node Queues

Consider a pipeline with nodes $n_1 \ldots n_h$, 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 [27] 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.

3.5.1 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[1] — which results in only one rather than two switches. In short, we can establish the following lemma:

Lemma 3.5.1. *For $1 < i < h$, the rate R_i of switches away from n_i per item consumed by n_i is given by*

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

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

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. $\square$

Combining the results of Lemma 3.5.1 over all nodes in the pipeline and simplifying, we obtain that

Corollary 3.5.1.1. *The total rate R of switches across all pipeline nodes per input consumed by n_1 is given by*

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

3.5.2 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

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

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 3.5.2. *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*

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

3.5.3 Practical Considerations: Rounding and Safety

While Lemma 3.5.2 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 $a_i v$ 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 a_i associated with a node is greater than 1. We address this safety concern in more depth in Chapter 4.

3.5.4 Robustness of Optimization

The robustness of these runtime characteristics in determining throughput optimal queue sizes depends on the application. For example, as described later in Section 3.7, the N-Queens application takes about $20 - 30$ seconds to complete on a problem size of $N = 18$. If we take into consideration the worst possible case of execution, where every time a node fires the execution must switch to another node, we can use the node service time and node overhead time (how long it takes to gather a vector of inputs and process their outputs on the output queue) to determine an approximate upper bound on application execution time.

Under this assumption, the total application time will be $\approx$25 minutes when distributing memory amongst all the queues equally at the minimum reasonable total queue size (600MB). Estimated worst case execution time decreases as we increase the total queue memory allocation for the application, such as with a queue space of 1400MB which will take $\approx$21 minutes when distributing memory equally amongst all queues. Using our optimizations in Chapter 3 this approximate upper bound reduces by $1 - 2$ minutes. In practice, this worst-case execution time will almost never occur because queues will be sized more appropriately for the application, providing plenty of room for nodes to run on more vectors of input at a time.

3.6 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. (If n_i produces

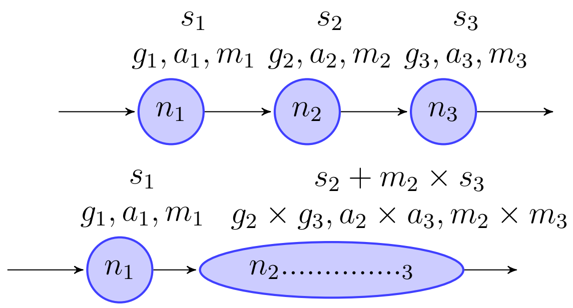


Figure 3.1: Topological view of merging compute nodes. Combining nodes n_2 and n_3 is expected to incur approximately m_2 separate calls to n_3 per call to n_2 because inputs to n_3 are no longer queued. The output gains are then estimated as the cumulative gains of the two combined nodes.

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 3.1 illustrates the merge operation on the last two nodes of the pipeline from Figure 1.3 in Chapter 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 then 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 s_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 s_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 s_{jk} of n_{jk} is estimated as

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

Now suppose we insert a queue between original nodes n_i and n_{i+1}, where $j \leq i < k$ 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 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

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

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.

3.7 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.

3.7.1 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 30K DNA bases from the *Salmonella* genome to a database of 4 billion bases (compressed to 1GB) 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 [6]. A description of this application can be found in Chapter 2.

We tested Taxi on a file of 2GB 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.

3.7.2 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 3.5.2 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.

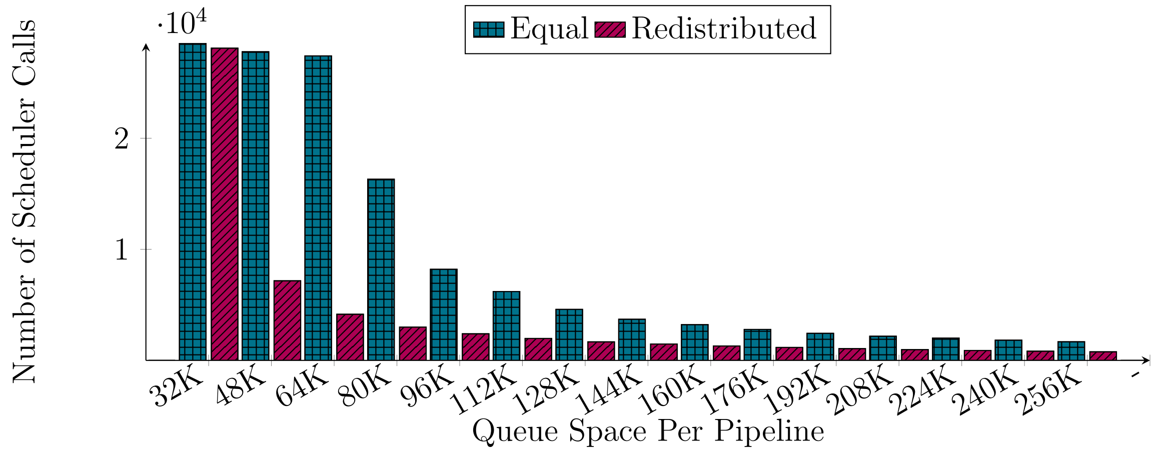


Figure 3.2: Number of calls to scheduler for BLAST, averaged over 50 trials.

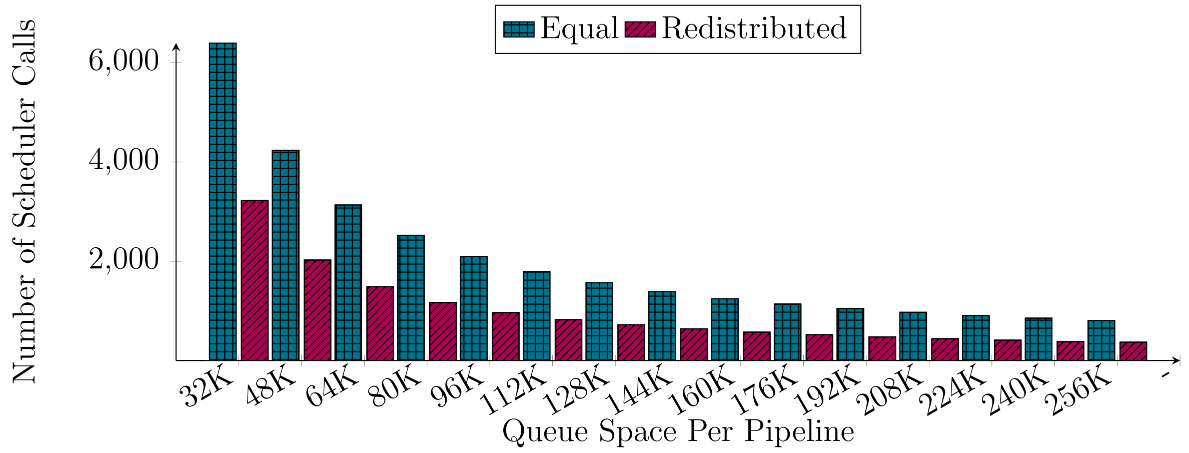


Figure 3.3: Number of calls to scheduler for Taxi, averaged over 50 trials.

Figures 3.2, 3.3, and 3.4 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

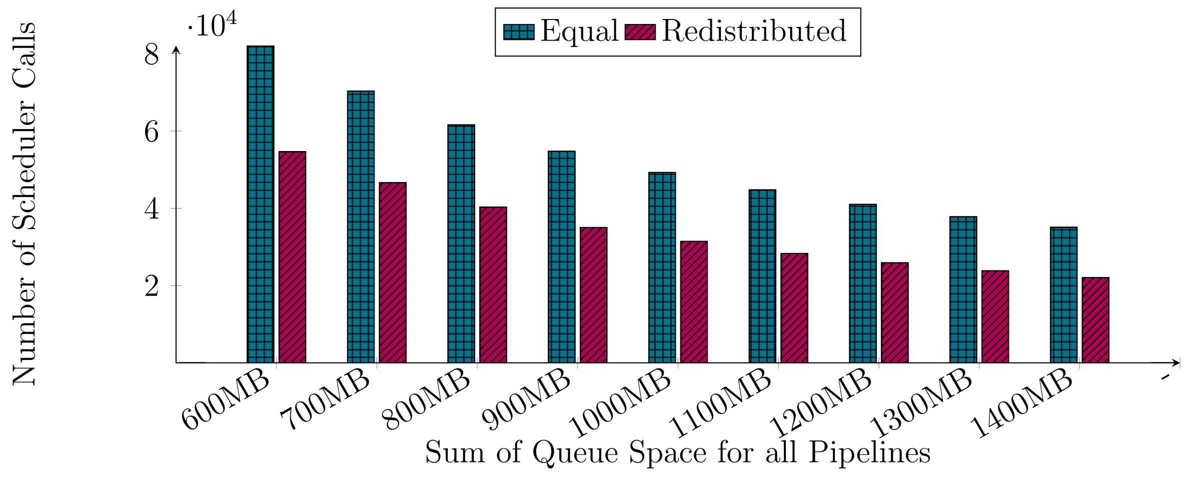


Figure 3.4: Number of calls to scheduler for N-Queens, averaged over 50 trials.

nearly all sizes, redistributing the queue space as dictated by our optimization does 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 $<$ 80KB per queue with an equal distribution of space among queues, or $<$ 192KB 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.

Figures 3.5, 3.6, and 3.7 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.

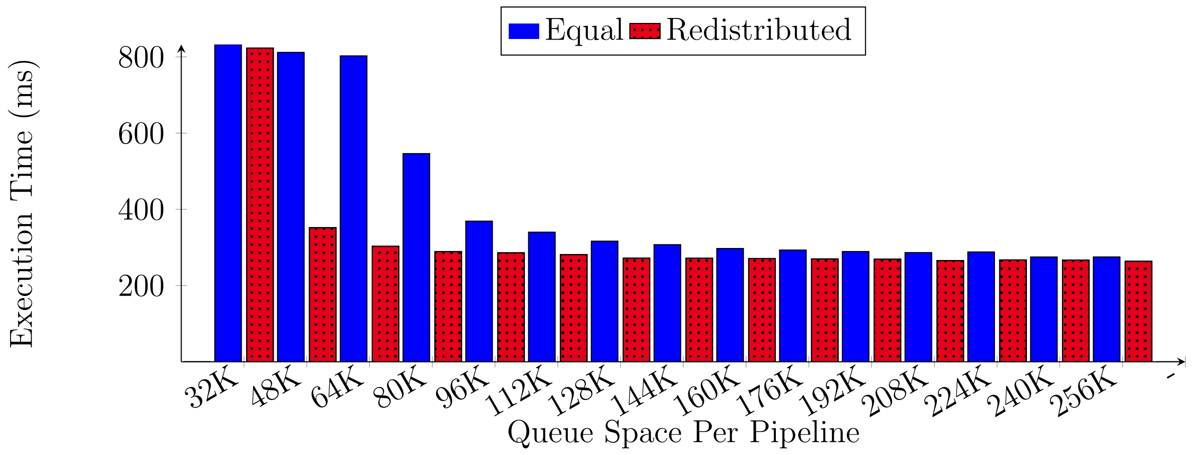


Figure 3.5: Total execution time for equal vs. redistributed queue space on the BLAST application, averaged over 50 trials.

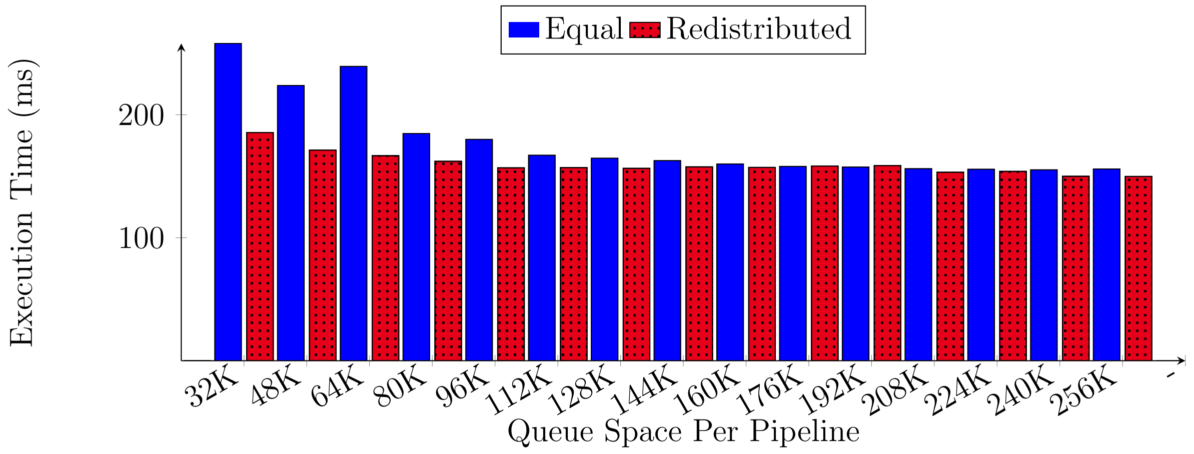


Figure 3.6: Total execution time for equal vs. redistributed queue space on Taxi, averaged over 50 trials.

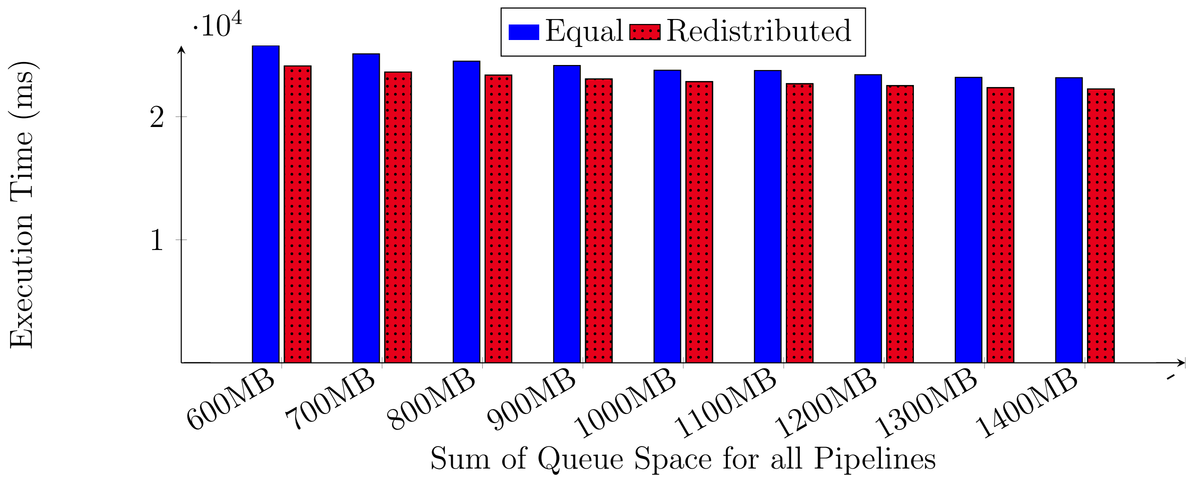


Figure 3.7: Total execution time for equal vs. redistributed queue space on N-Queens, averaged over 50 trials.

3.7.3 Node Merging Optimization

We studied the impact of node merging on the BLAST and N-Queens 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 3.1 shows an 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 N-Queens with and without redistribution.

Table 3.1: 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

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

Table 3.2 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 3.3 shows predictions and empirical results for several merging strategies for N-Queens, 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 the majority of these strategies safely was infeasible for our target GPU.

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.

Table 3.2: Predicted (cycles/input vector) vs. Empirical (ms) Results for Different Node Merging Strategies in BLAST. Queue space allocation is 192KB/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

Table 3.3: Predicted (cycles/input vector) vs. Empirical (ms) Results for Different Node Merging Strategies in N-Queens. Queue space allocation is 1000MB 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

3.8 Conclusions and Future Work

In this chapter, 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 demonstrable 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 consider an alternative strategy for dealing with safety constraints on queue size that may be less prejudicial to performance at small queue sizes in Chapter 4. Additionally, 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 for 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.

Chapter 4

Interruptible Nodes

This work was originally presented at the 15th International Symposium on High-level Parallel Programming and Applications (HLPP), Porto, Portugal, July, 2022. Original publication was made through the HLPP 2022 special issue of the International Journal of Parallel Programming, December, 2022 [34].

4.1 Introduction

In an irregular dataflow streaming application, the sizes of inter-node queues should be carefully chosen based on considerations of average and worst-case node behavior described in Chapter 3 [35]. Scheduling of an application's nodes must then be cognizant of inter-node queue occupancy to ensure safe and efficient execution [27].

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 chapter, 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 chapter is divided as follows. Section 4.2 examines related work, while Section 4.3 describes our application model. Section 4.4 looks at examples of irregular streaming applications and the impact minimum queue size restrictions have on their performance and memory usage. Section 4.5 describes MERCATOR's new interruptible node facility and the challenges of implementing interruptible nodes. Section 4.6 empirically evaluates interruptible nodes on two applications, N-Queens and BLAST. Finally, Section 4.7 concludes and considers future work.

4.2 Related Work

Prior work in scheduling multiple tasks on the GPU includes work on cooperative CPU-GPU scheduling. Hyoseung et al. [18] 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 [17] 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 [26], 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 [40] 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.

4.3 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.

4.3.1 Application Mapping

As described in Chapter 1, an application is represented as a pipeline of *compute nodes* $n_0, n_1, \ldots$ with successive nodes connected by dataflow *edges*. 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 other nodes to execute until n_i again has both available input data and available output space.

As a refresh on GPUs from Chapter 1, 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.

4.3.2 Application Performance and Queue Optimizations

As stated in Chapter 1 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 [19, 31], but MERCATOR uses a scheduling policy, Active-Full/Inactive-Empty (AFIE) [27], 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.

4.4 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.

4.4.1 Example Applications

In this section and Section 4.6, we study two irregular streaming applications whose pipelines exhibit the impacts of minimum safe queue sizes: N-Queens and BLAST.

N-Queens. As described in Chapter 3, the N-Queens application enumerates all possible ways of placing N queens on an $N \times N$ chess board such that no queen can attack another.

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 N-Queens 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, N-Queens instances exhibit substantial irregularity in their branching behavior.

BLAST. As discussed in Chapter 3, the Basic Local Alignment Search Tool for nucleotide sequence [1] performs pattern matching in a large DNA database.

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.

4.4.2 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 [27]. 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 [35], 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 4.1 illustrates the ideal and safe queue sizes for all 14 nodes of the N-Queens 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 Table 4.1, 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 N-Queens uses for input 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.

Table 4.1: Gains and implied queue sizes of N-Queens 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

The problem of bloat may be exacerbated by optimizations that attempt to merge adjacent nodes. In the case of N-Queens, analysis of service times and SIMD occupancy according to [35] 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.

4.4.3 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 4.2. The total memory allocation is much smaller than for N-Queens (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 total queue space usage (either ideal or minimum safe).

Table 4.2: 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

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 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 to 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.

4.5 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.

4.5.1 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.

As an example, Figure 4.1 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()`,

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

Figure 4.1: 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.

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.

4.5.2 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 4.2 illustrates a modification of the node in Figure 4.1 to support suspension and resumption.

```
__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);
}
```

Figure 4.2: 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.

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.

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.

4.6 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 N-Queens) 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 Section 4.4.1 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.

4.6.1 Reduction of Scheduling Overhead

We first compared the N-Queens 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 [35].

Figure 4.3 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.

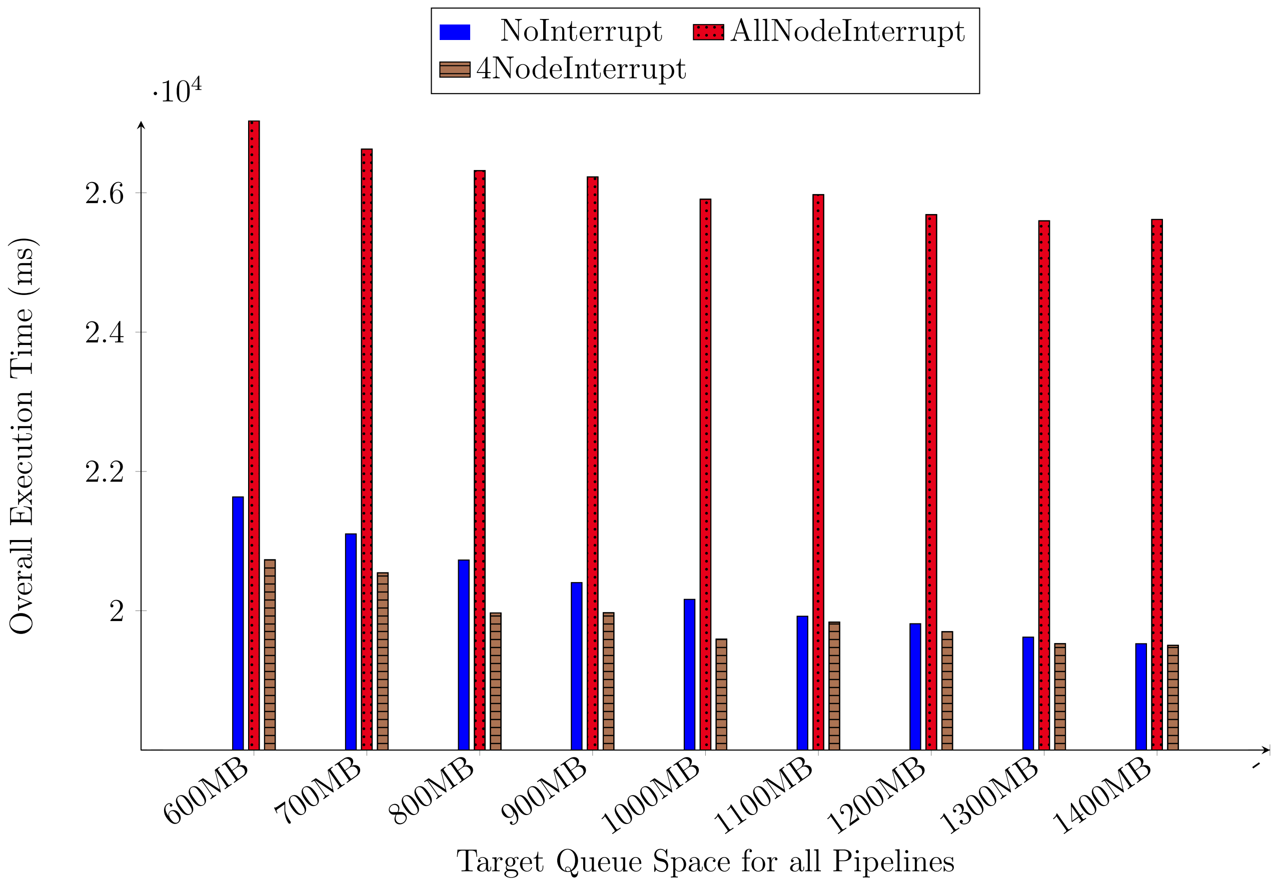


Figure 4.3: Total execution time for N-Queens for different target allocations for total queue memory. Measured times are the average of 50 trials and have a 95% confidence interval of ±150 ms.

Recall from Table 4.1 that only the first few nodes of the N-Queens 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. The impact of re-allocating queue space to nodes that could use it more was expected to diminish as total queue allocation increased, since larger queues require less switching.

As shown in Figure 4.4, redistribution of space previously needed for bloat in the first four nodes of N-Queens had a salutary effect on scheduler overhead. Nodes near the middle of the the pipeline, which have the largest average cumulative 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 the difference in running time between NoInterrupt and 4NodeInterrupt decreases with increasing target allocation.

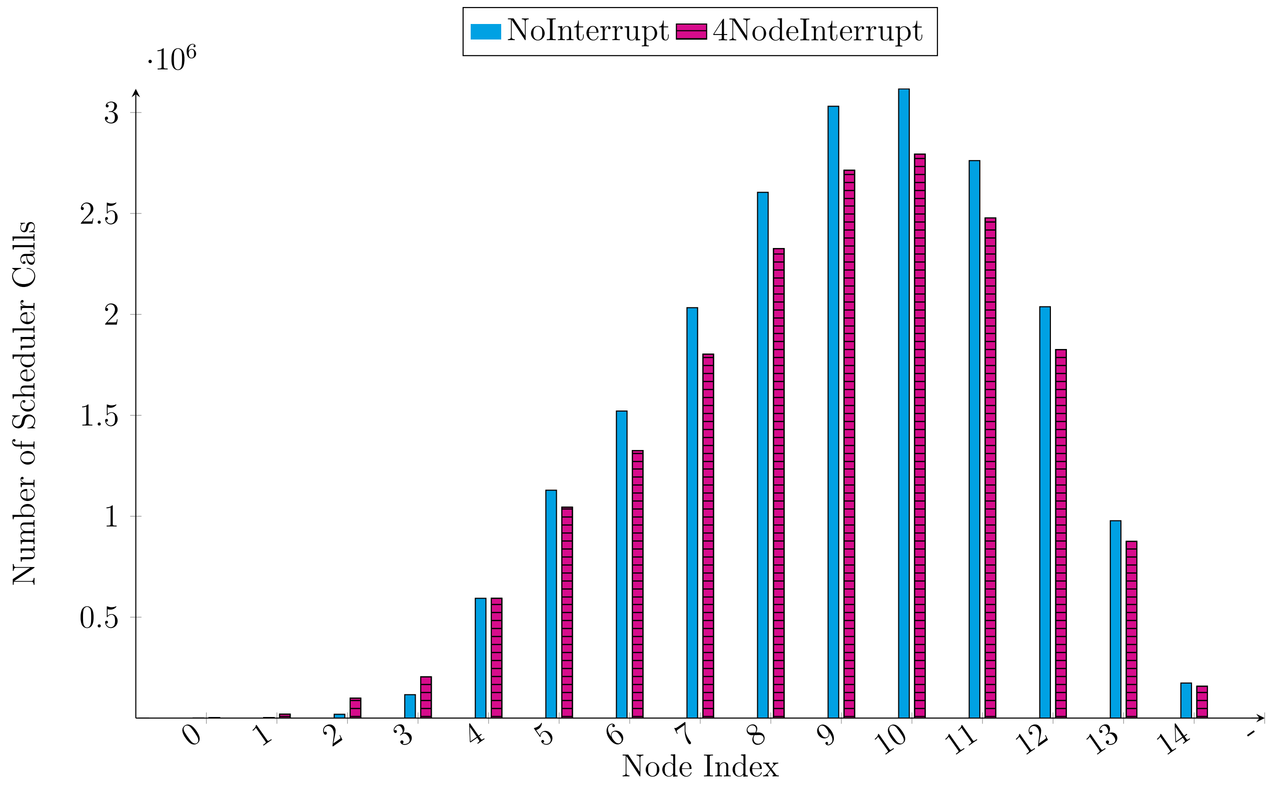


Figure 4.4: Number of scheduler calls from each node for N-Queens given 600MB total queue space for all pipelines.

We then investigated whether the same strategy of targeted interruptibility was effective for the BLAST application. Recall from Table 4.2 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 4.5 shows that at the smallest target allocation (11.5 MB across all queues), targeted interruptibility had a large beneficial effect on running time. As Figure 4.6 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. Again, because total queue space directly impacts the overall number of node switches, we see that the affects of re-allocating queue space diminishes as total queue space increases.

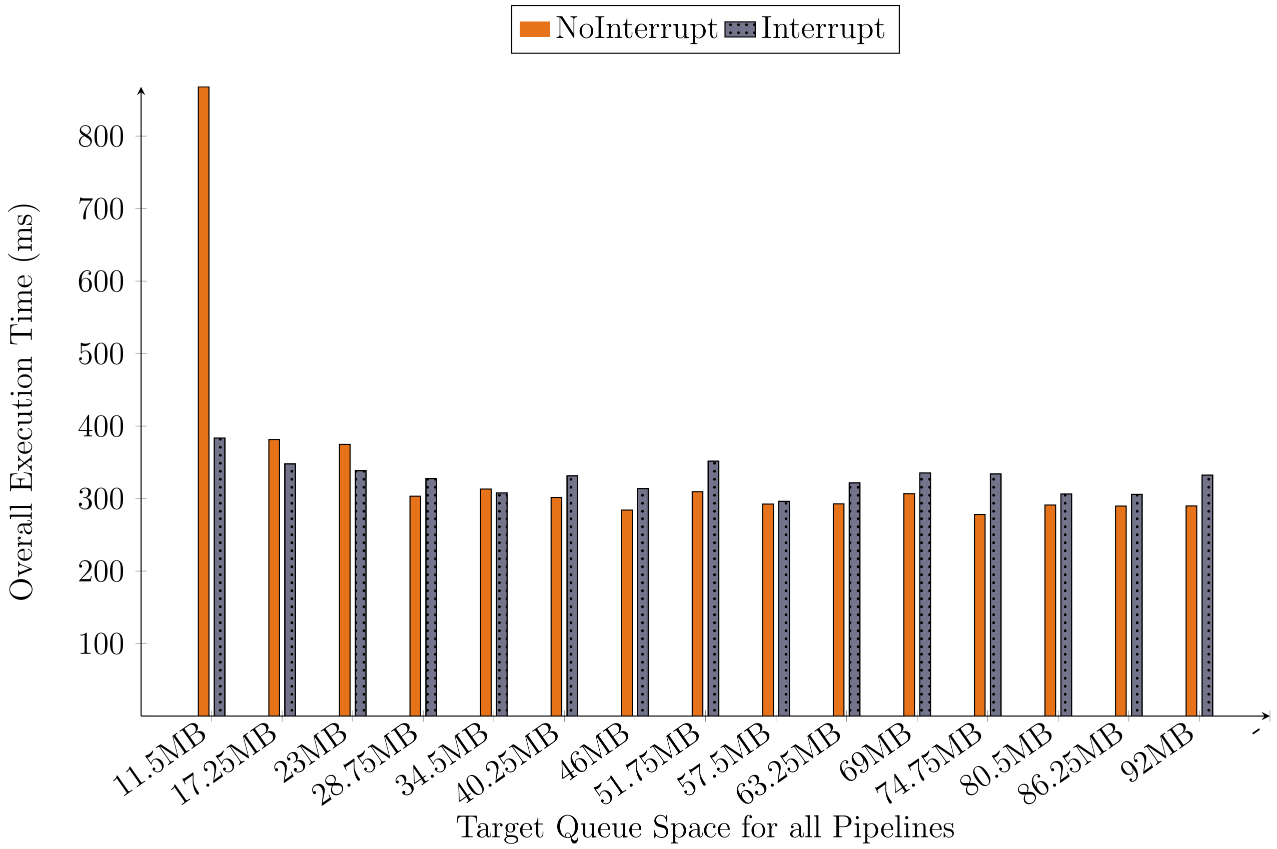


Figure 4.5: Total execution time for BLAST for different target allocations of queue memory. Measured times are the average of 50 trials and have a 95% confidence interval of ±50 ms.

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.

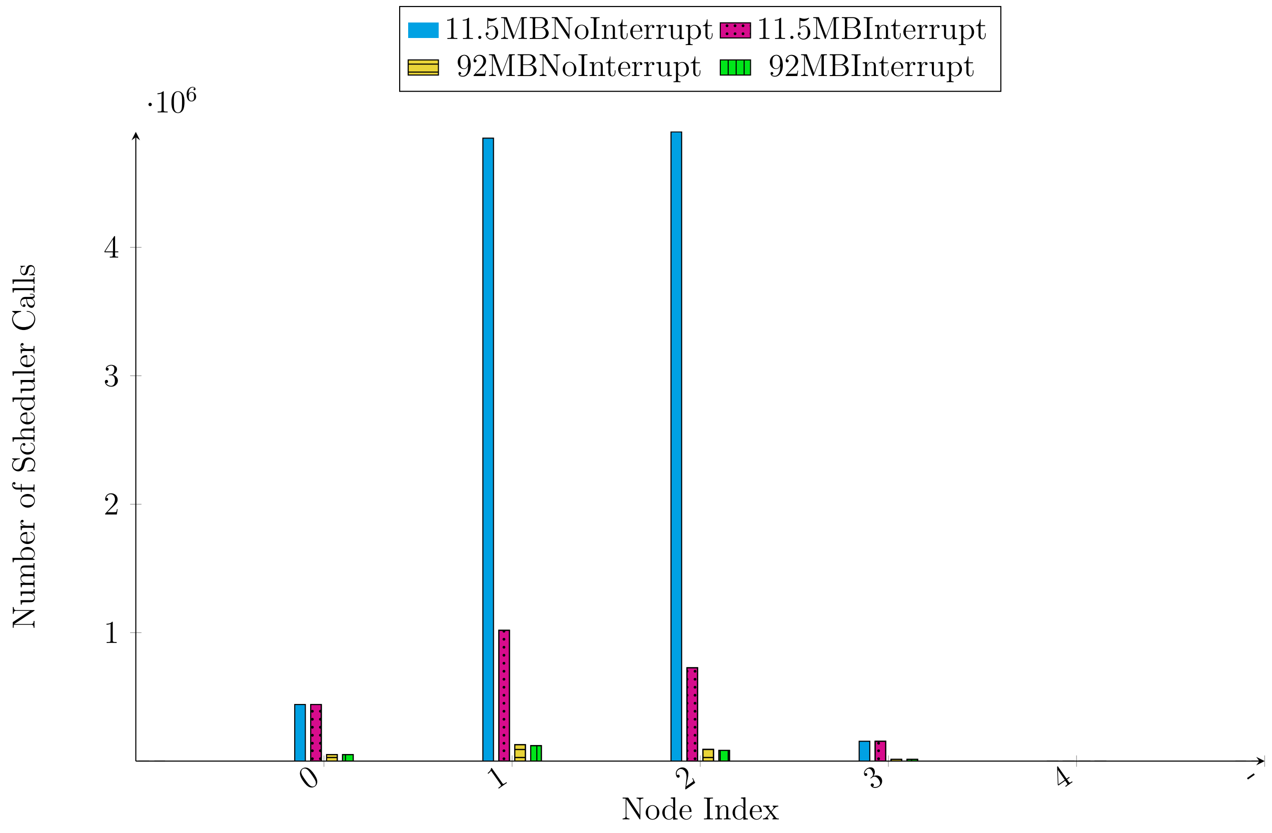


Figure 4.6: Number of scheduler calls from each node for BLAST at two different target queue space allocations.

4.6.2 Combining Interruptibility with Node Merging

As discussed in Section 4.3, 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 N-Queens application, which our analysis in [35] 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 Figure 4.7. Making the merged node interruptible eliminated this excess memory usage. The resulting implementation exhibited running time similar to that of the unmerged version.

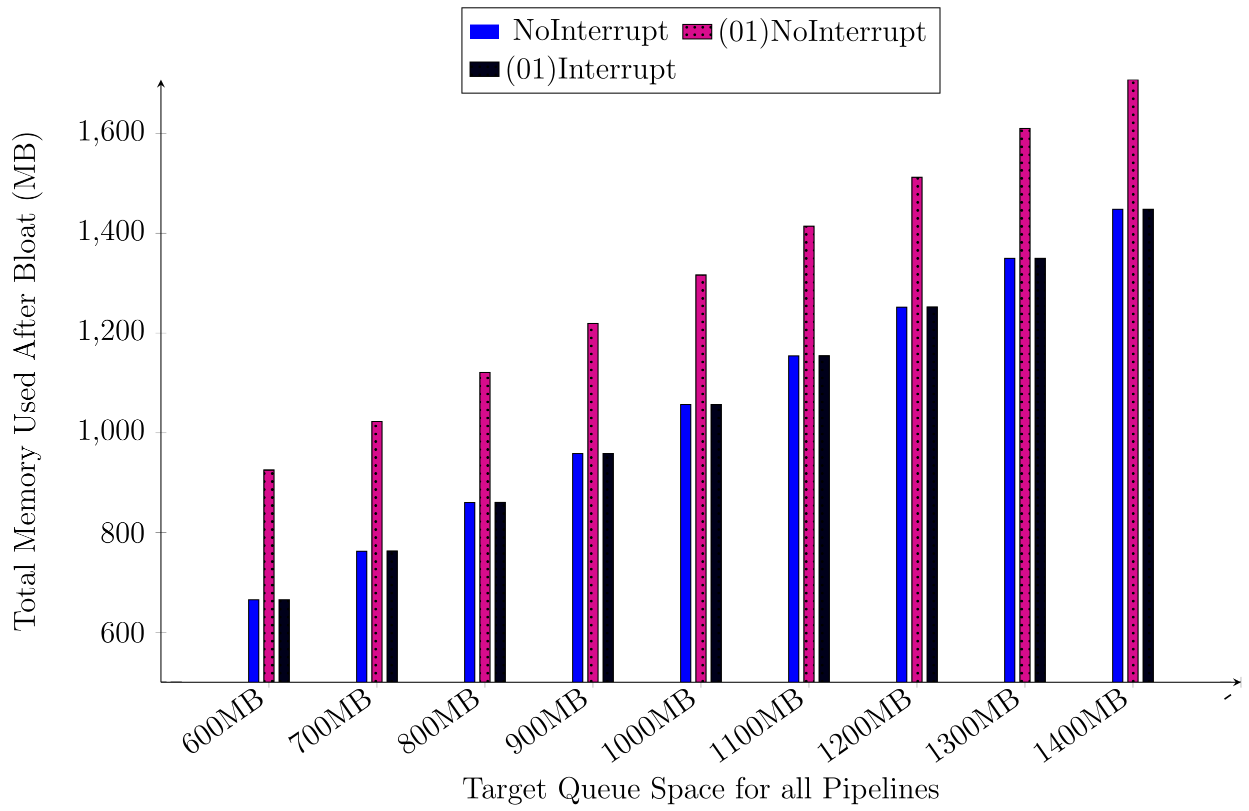


Figure 4.7: Total actual memory used for each target allocation in N-Queens before and after merging nodes 0 and 1.

4.7 Conclusion and Future Work

Irregular streaming dataflow applications have great potential for wide-SIMD parallelization, but this potential can only be realized 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, Prokopec and Liu [28] use 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.

Chapter 5

Iterative Streaming Application Optimization

This chapter describes work in progress that will eventually become part of a peer-reviewed publication.

5.1 Introduction

In previous chapters, we focused solely on the throughput of irregular dataflow streaming applications on the GPU. Applications in those previous chapters fit all operations and memory within a single GPU kernel call, so the communication overhead and data transfer time between the CPU and GPU was incurred only once over the lifetime of the application. We now consider applications where a single GPU kernel call is insufficient for all the required processing.

A class of applications, *branching search*, solves a problem instance by traversing a tree of sub-problems, each of which arises from a possible partial solution to the original problem. The tree structure arises because of multiple choices for how to begin solving a sub-problem, each of which lead to a distinct, smaller sub-problem. The total number of possible sub-problems grows by a multiplicative factor at each level of the tree. A parallel implementation can process many sub-problems at once, provided it has sufficient space to store the results of such processing on the GPU.

When the goal of branching search is to find a single best solution to the problem, we may use *branch-and-bound optimization* to reduce the number of sub-problems to be solved.

Branch-and-bound uses an estimate of the best result obtainable starting from a given sub-problem (the bound) to *prune* sub-problems that provably cannot produce a better result than the best solution found so far (also called the *incumbent* solution). Branch-and-bound is particularly useful when finding optimal solutions to NP-optimization problems such as 01-Knapsack, Job Shop Scheduling, and Traveling Salesman. However, from the perspective of parallelism, pruning creates irregularity in the dataflow of a branching search application – the number of new sub-problems produced by expanding a given sub-problem (i.e., its gain) now depends on the quality of the incumbent solution, which improves at an unpredictable rate over the course of the search.

A natural implementation of branching search accumulates solutions at the output of the GPU pipeline and then copies sufficiently good solutions back to the host. The number of solutions that actually accumulate at the output depends dynamically on how effective pruning is, but in the worst case (if pruning is ineffective), the number of outputs may be very large. Planning for this worst case means allocating enough space on the GPU to store all the outputs that *could* arise from the pipeline, which may be infeasible for even moderately sized problems given the multiplicative expansion at each step of branching search. To limit the amount of GPU space required, we split the search into smaller parts, each of which traverses only a limited number of levels in the search tree, or iterations, and so has bounded output size per input. Each of these parts becomes a separate GPU kernel call, for which results must be returned to the CPU and queued for further work. This design necessitates multiple kernel calls to solve the full problem and adds a new potential throughput bottleneck — *CPU-GPU communication overhead*.

In this chapter, we explore how to maximize the throughput of a branching search application using branch-and-bound optimizations. We explore how input size per kernel call and branching depth per kernel call affects average wall-clock time. We empirically evaluate varying these two parameters in an implementation of the Traveling Salesman Problem and provide insight on how these findings could be used for other branching search applications.

5.2 Related Work

Similar to the MERCATOR framework [9] used in this work, which was designed for irregular dataflow streaming applications, Mokhtari and Stumm create a framework to handle similar tasks [23]. Their work focuses on "Big Data" operations such as data transformations, filtering, aggregation, etc. To handle dataflow irregularity, they use the CPU to partition and assign problems to the GPU, rather than maintaining queues in an uberkernel like MERCATOR. The goal of their work is to optimize CPU-GPU communication and optimize GPU memory accesses. Here, we focus on the former, but in the context of sequential kernel calls with non-overlapping host commands (e.g., data partitioning and CPU-GPU data transfers are done sequentially after a kernel call). We look at how large to make a kernel on the GPU in terms of CPU-GPU communication and CPU overheads. We do not examine GPU memory accesses, as this functionality is handled by MERCATOR.

Parallel branch-and-bound has been implemented on CPU-GPU systems before, such as that by Boukedjar et al. [4] for the classic 01-Knapsack problem. Both the branching and bounding occur on the GPU, much like this work. However, the branching and bounding are limited to a single sub-problem per kernel call, and each operation has a separate kernel. We coalesce both branching and bounding into a single computational stage and have multiple computational stages per kernel, which reduces the CPU-GPU communication overhead. Their work focuses on comparing their CPU-GPU implementation versus a CPU only implementation, whereas our work focuses on minimizing overall application time via GPU kernel configurations.

The work by Chakroun and Melab [7] noted that branch-and-bound algorithms have dataflow irregularity, which makes these applications harder to implement efficiently on GPUs. They solve the irregularity by breaking the problem into *regular* sized chunks which are determined by a heuristic of how much data can fit on the GPU at once. The GPU kernel performs the branching, bounding, and pruning of solutions, each in separate kernels, but leaves the solutions on the device between kernels. This is similar to what MERCATOR does, although in MERCATOR one does not have to return control back to the host to call the separate operations. Instead, we conditionally branch within each computational stage of the kernel based on the bound and only update the incumbent solution once we return to the CPU.

An irregular dataflow streaming application similar to the branch-and-bound optimizations here is DNA sequence alignment using a suffix trie, found in previous work of our lab by Cole et al. [10]. Their objective is to align a given set of short DNA sequences (reads) against all valid starting positions in a large DNA database with no more than k differences between the segments. They have a similar issue of a very large search space frontier, since there are millions of reads and the database is typically billions of characters. The traversal of inputs is also similarly tree structured, with specific alignment traversals being pruned once there are more than k differences.

5.3 Application Model

In this section, we describe our abstract application model used to implement irregular dataflow streaming applications. As discussed in Chapter 1, we express our applications as a pipeline of compute nodes $n_1, n_2, ...$ with edges between compute nodes, as shown in Figure 1.3. Each compute node n_i takes an input vector of v data items at a time and produces some data-dependent number of outputs, what we call output gain. Each edge has a finite sized queue which stores outputs from the upstream node for later consumption by the downstream node. The queue sizes are fixed during the application's execution because dynamic memory allocation for our target platform is expensive.

Our execution platform as explained in Chapter 1, the GPU, is designed as a set of wide-SIMD processors, with each processor running multiple asynchronous non-communicating contexts, or GPU blocks. The application pipeline is therefore replicated multiple times across a single GPU, with the application input evenly partitioned and each with an independent node scheduler. The scheduler uses the AFIE protocol independently within each pipeline replica to decide which node should run next.

Once the application has finished running on its current set of inputs, the outputs are returned to the host (the CPU) for further processing. This is a natural synchronization point between pipeline replicas, since communication between pipeline replicas is not easy and expansionary effects within the pipeline can make the potential output size too large to fit into GPU memory, requiring the pipeline replicas to offload results anyways.

5.4 Iterative Expansionary Applications

Iterative expansionary applications implement an iterative operation that is performed a finite number of times to expand the application's input. Each iteration processes the result(s) of the prior iteration, producing one or (typically) more outputs for each of its inputs. For branching search, an iteration involves expanding a search space by one step, that is, taking a sub-problem (corresponding to a partial solution of the original input problem) and expanding it by all feasible next choices, producing new sub-problems. For example, an iteration of N-Queens takes a partial solution in the form of a chess board with some number of queens placed and attempts to place one more queen in all feasible positions of the next empty row, generating perhaps many new partial solutions. N-Queens may be expansionary, but the rate of expansion in other problems, such as the TSP task described below, is much larger and varies dynamically depending on the value of the incumbent solution.

A major issue of iterative expansionary branching search is that the search frontier, and eventually the number of feasible outputs, can grow exponentially with the number of iterations. While traditional multi-core CPUs have large enough memory pools to hold the entire search output, memory-limited devices such as GPUs do not have this luxury, and so we cannot implement the entire search as a single GPU pipeline. Even if one wanted to implement the problem as a single GPU pipeline, certain parts of the search tree may take significantly longer to traverse than others; blocks that finish first are left idle while the others run, since MERCATOR cannot re-distribute work from one block to another. Additionally, if we want to bias the traversal to reach leaves and so produce new, better incumbent solutions (and increase pruning) faster, we would need to modify MERCATOR's AFIE scheduler in a non-trivial way, potentially violating the scheduler's high occupancy guarantees and incurring runtime overhead on every node firing.

We instead segment the iterative application's pipeline into a series of *GPU steps*, each of which performs only some of the required iterations. The output of a GPU step is copied back to the CPU to free up the GPU's memory, and another GPU step is launched starting with (some of) the sub-problems returned from previous steps. The CPU is a natural synchronization point for the application, removing the need for any GPU block-block communication and allowing us to not violate our GPU scheduler runtime guarantees, although we pay the cost of CPU-GPU data transfers for the results of intermediate steps.

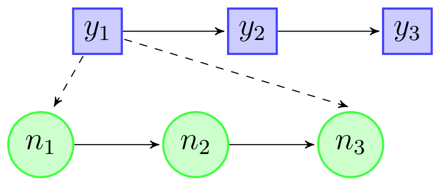


Figure 5.1: A 3-step CPU/GPU topology for an iterative expansionary application performing 9 iterations on its input. Each step y_i contains the same 3-node pipeline n_1, n_2, n_3, which is run as a GPU kernel on the device. Step y_1's output feeds into step y_2's input, and step y_2's output feeds into step y_3's input.

Figure 5.1 shows a simple iterative expansionary application implemented on the GPU using our application model. It is designed as a sequence of u steps $y_1, y_2, ..., y_u$, where each step performs a pipeline of z iterations $n_1, n_2, ..., n_z$ on the GPU, totaling $z \cdot u$ iterations. Just as the pipeline within one step is composed of multiple nodes, as in Figure 1.3 from Chapter 1, each step has an input queue which is managed on the CPU. Outputs from steps feed into subsequent steps of the pipeline, so y_1's output feeds the input queue for y_2.

The pipeline of computations run by a GPU step is the same for all steps and can be implemented as a common GPU kernel. Each node within that pipeline performs the same action, which is a single iteration. The pipeline for a single step functions the same as and with the same restrictions as the GPU pipelines described previously in Chapter 1.

The CPU repeatedly runs steps until there are no inputs remaining for any step. The host determines which step to run next based on the number of inputs queued for each step. When a step has enough input, the CPU executes it on the GPU. Steps are run sequentially, each occupying the entire GPU, analogously to the way MERCATOR executes nodes sequentially within a single GPU pipeline replica.

Adding Branch-and-Bound Optimization. A class of branching search applications of particular interest is *branch-and-bound* optimization. Here, a *bound* is an estimate of the *best possible final solution value* obtainable by solving a given sub-problem. When the value of this bound for a sub-problem is worse than that of the current best solution found so far, the sub-problem is pruned, eliminating the work that would otherwise be needed to solve it. Branch-and-bound techniques are used to exactly solve instances of NP-optimization problems such as 01-Knapsack, Job Shop Scheduling, and Traveling Salesman.

We add branch-and-bound behavior to our CPU/GPU design for branching search as follows. During the search, the CPU detects when a GPU step at the bottom of the search tree has produced a new, better solution and makes this result the incumbent solution for future GPU steps. The GPU computes a bound for each new sub-problem generated by any node within any step and prunes the sub-problem immediately if its bound is worse than the incumbent solution. We note that steps other than the one that corresponds to the bottom of the search tree cannot update the incumbent solution.

5.4.1 Traveling Salesman

The Traveling Salesman Problem (TSP) is a classic problem in computer science and logistics. The task is to find the shortest route for a salesman to visit all of a given set of n cities and return to the starting point, with each city being visited exactly once. Because finding the best TSP tour for a given set of cities is an NP-optimization problem, no polynomial-time algorithm is known to find an exact solution, though polynomial-time, constant-factor approximations exist for metric TSP. If one requires an exact solution, one may use a branch-and-bound optimization approach.

Branching search over a TSP instance with n cities traverses a tree of height n. At tree level k, the search considers sub-problems consisting of a partial tour that specifies the first k cities visited in order. In each level's iteration, all of the remaining $n - k$ cities are considered for the next city in the tour, producing $n - k$ new sub-problems with $k + 1$ cities visited. Each new sub-problem then calculates a lower bound on the value of the best possible completion of its partial tour. If the bound is greater than the incumbent solution, we know that this sub-problem will never produce a better solution, and so it is pruned from the search.

In this work, the lower bound computed at each branching step is double the sum of the current partial tour and the two minimum distance connections of each unvisited city. There is a subtle difference in how the first and last cities' contributions to the bound are calculated that is described in [13].

A simple example TSP search tree is presented in Figure 5.2. Here, we show how a 4-city TSP problem is expanded and the subsequent sub-problems are explored in a sequential implementation. We begin with the first node which has a tour containing only our starting

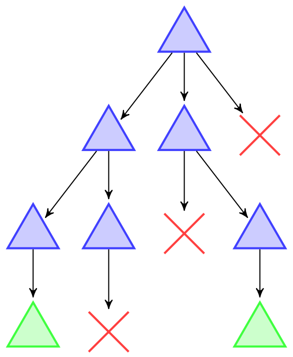


Figure 5.2: A sample sequential exploration of sub-problems for a 4-city TSP. Blue triangles represent an explored sub-problem, green triangles are sub-problems that result in a better global solution, and red x's are sub-problems that were pruned since their bound was worse than the incumbent solution. Search space traversal visits the left-most sub-problem first, adding the next available lowest-index city to the partial tour. If a partial tour after adding a city has a worse bound than the current best solution, that sub-problem is not examined. When a leaf sub-problem is reached, if the solution is better than the current best, the incumbent solution is updated for future use.

city. We then try to add the next lowest city index to the tour (left-most city in the search tree) until we either reach a leaf node or can no longer process that path. In this case, we reach a leaf node which has a better solution than that of the current best, and so use that solution to compare to subsequent sub-problems. We continue processing the left-most sub-problem in the tree, pruning when a sub-problem has a worse bound than the current solution and updating the solution if necessary when a leaf sub-problem is reached.

When translating this computation to SIMD, we notice that every sub-problem is independent, allowing us to explore multiple sub-problems at the same time. In our implementation, we split the search tree into *steps* as shown earlier in Section 5.4, where each step does some number of iterations on the search tree. For example, we could split the search tree in Figure 5.2 into 2 steps, each of pipeline length 2.

5.5 Design Choices for Branch-and-Bound

When implementing branch-and-bound optimizations, there are numerous design choices that can be made which affect application throughput. Optimizations can be made to both the structure of the application framework and the bounding operation. We focus on the former for this work. In this section, we list a few of the design considerations for maximizing application throughput given our application model.

First, when running a GPU application, it is important to maintain high *occupancy*, or the number of processors doing active work to have a high application throughput. To achieve this, one must ensure that processors are not starved of inputs to work on. Thus, before sending a problem instance to the GPU, we have the CPU calculate the first few iterations of the branching search so that even the first GPU step has plenty of work.

One interesting difference between branch-and-bound optimization problems when compared to our previously studied N-Queens problem is that the rate at which better incumbent solutions are found affects how many sub-problems are explored. Because it is unpredictable and input-dependent when a better solution will be found during tree traversal, modeling the impact of solution discovery on runtime is challenging. Moreover, optimizations that could otherwise improve GPU performance may delay discovering a better solution and hence may actually result in more overall work performed.

We developed the following heuristic approach to balance GPU occupancy and solution discovery rate. When choosing which GPU step to run next, we select the *last (closest to a leaf) step* whose available input is large enough to ensure reasonable GPU occupancy. If no step has enough input to achieve high occupancy, we try to expand the set of active sub-problems rapidly by running the earliest (closest to root) step with any available input.

When a new, better incumbent solution is found, we examine the bounds associated with any sub-problems queued on the CPU and prune away all sub-problems that can no longer produce a global optimum before beginning the next GPU step. Pruning after discovering a new incumbent solution happens on the CPU to avoid copying sub-problems to the GPU that will immediately be pruned; however, CPU-side pruning is a sequential operation, unlike the parallel pruning performed on the GPU.

While the above design choices proved qualitatively important to overall application performance, the following empirical study focuses on tuning quantitative parameters of the implementation. In particular, we examine the effects of two application parameters on overall throughput: the number of sub-problems provided as input to each GPU step, and the length of each GPU step's pipeline.

5.5.1 Input Size

The input size of each step, which is the threshold of number of inputs required before the CPU will run a step, determines two key factors important to application throughput: **(1)** how many *potential* outputs a step will create and **(2)** how many sub-problems a step can examine at one time.

For the first factor, the number of potential outputs limits the maximum number of inputs a single step can handle. If too many inputs are given to a step, and that step overall has a large gain, the step may not have anywhere to write its outputs, since there is limited memory for output storage on the GPU. Individual nodes within a step may also exhibit large gains, but the storage requirements for these nodes' downstream queues can be mitigated by using interruptible nodes as described in Chapter 4. Presently, there is no comparable mechanism in MERCATOR to interrupt a GPU step to copy partial results back to the CPU; hence, we require the maximum potential output space for a step to be allocated on the GPU. The longer the pipeline for a step (i.e., the more branching operations possible within one step), the larger the required output space becomes; ultimately, this concern limits the maximum pipeline length that can be supported for a GPU step.

The impact of input size on parallelism follows from GPU occupancy. With lower occupancy, the GPU has less work to do every time it is called, thus increasing the proportion of total execution time spent with some threads idle or in overhead rather than in productive branching and bounding computations. Higher occupancy mitigates these issues and hence increases application throughput.

For our target application, finding a better incumbent solution in a branch-and-bound optimization may be more beneficial than processing more sub-problems at a time. Decreasing the input size for each step reduces the number of sub-problems that are expanded and explores deeper sub-problems first, or a *depth-first* focused approach. The idea is to find a better incumbent solution faster, thus pruning inputs in earlier steps before they are expanded. This comes at the cost of potential parallelism in the system, since less data is available to be worked on at any given time. Should a better incumbent solution not be found quickly, the overhead of working on smaller chunks of input can outweigh the benefits of finding the better incumbent solutions. Contrarily, a *breadth-first* focused search takes large amounts of input data to leverage the parallelism in the input data. In this case, more sub-problems are examined, but if a better incumbent solution is more difficult to find in the input, then leveraging the greater parallelism will yield better throughput.

The choice of input size to a GPU step trades off between GPU occupancy, required output space, and getting better incumbent solutions faster. We note that each step can have a distinct choice of input size, and that later steps (in TSP corresponding to sub-problems with more cities having already been fixed and hence fewer possible branching choices) produce fewer outputs in the worst case for a given input size. We examine the impact of *scaling* the input size of each step to account for its maximum output size in the investigation that follows.

5.5.2 GPU Step Size

The number of pipeline stages per step (i.e., the number of branching search iterations performed in the step) determines the pipeline length of the MERCATOR application. Increasing the step size requires more intermediate queue space for the application and more output buffer space on the GPU but reduces the number of times that the CPU and GPU must coordinate to transfer data, as well as the number of times the CPU must decide which step to run next. Moreover, because our implementation updates the globally best solution found so far only between steps, a larger step size means the best solution is updated less frequently and so may decrease the amount of pruning performed.

Determining the step size is highly application- and input-dependent. However, performance of a subset of representative application runs could be a useful indicator of good step size configurations. We examine how step size affects application performance by comparing the wall-clock times of implementations with different step sizes.

5.6 Results

To test how certain optimizations affect branching search applications, we developed a TSP implementation. All tests were conducted on an NVIDIA A40 GPU with 48GB of GDDR6 RAM using 100 blocks with 128 threads. The CPU used was an Intel Xeon Gold 6248R. Tests were run under Linux using CUDA 11.7.

While our target GPU supports execution with many more than 100 concurrent blocks, each block requires enough output space on the GPU to receive its results. As discussed previously, limits on available GPU memory restrict the total number of inputs processed at once, so we limited the number of blocks to ensure that each block could run with reasonably high occupancy given our overall limit on total number of inputs per GPU step.

We generated 90 different TSP instances with 30 cities. The length of each edge between cities was chosen uniformly at random in the range of $[100, 1000]$ (hence, the instances are not necessarily metric). Branching search was always initiated from city 0. To produce a reasonable number of sub-problems for the first GPU step, we performed the first 4 iterations of the search on the CPU, so that each sub-problem given to the first GPU step fixed the first 5 cities in the tour. All individual runs are binned into the histogram in Figure 5.3 based on their CPU wall-clock time. With all the tests, the results are highly input-dependent, and ordering of which sub-problems are executed can drastically change application running times.

5.6.1 Input Size

As described above, we expect increased input size to broadly benefit overall execution time, subject to the limits of how much output space we have on the GPU. We ran our TSP application with a pipeline length of four nodes per step and compared using a fixed input size of 30×128 inputs for all steps versus growing the input size for each successive step.

For our problem size of 30 cities, the fixed size chosen is "safe" (i.e., does not exceed the available output space) for the first GPU step, which has the largest potential overall gain. We determined that, because of the reduced maximum gains of later steps, it was safe to increase the input size of the kth step ($k > 1$) to be $5k$ times that of the first step. Hence, for our implementation with varying input sizes, the second step had an input size $5\times$ that of the first step, the third had an input size $10\times$, and so forth.

As shown in Figure 5.3, the scaled input implementations (3NodePerStepScale and 4Node-PerStepScale) had more runs with faster CPU wall-clock times than their non-scaled input counterparts (3NodePerStepNoScale and 4NodePerStepNoScale). The standard deviation of the fastest running TSP instance for each implementation is 0.24 to 3.71 seconds, and

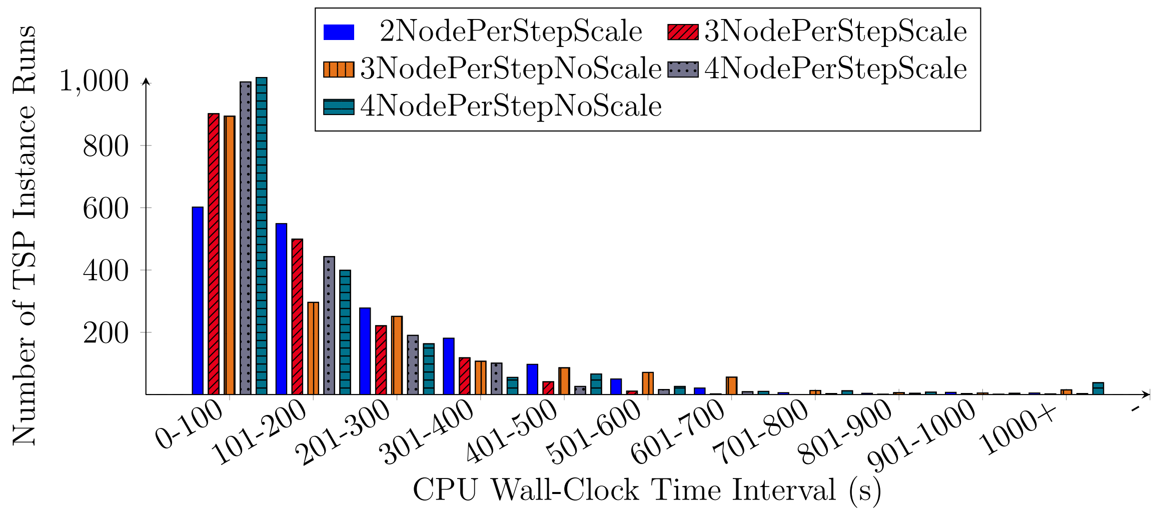


Figure 5.3: Histogram of wall-clock times observed over 90 different TSP instances, each tested 20 times. More runs in lower bin values (i.e., shorter running instances) indicates better performance. The NoScale implementations generally have more runs in the longer timed bins than their respective Scaled counterparts. Increasing the number of nodes per step generally increases the number of shorter timed runs.

205 to 675 seconds for the slowest running TSP instance. The variability in running times for shorter running problems was negligible while longer running problems had much higher variability.

An observation that can be gleaned from Figure 5.3 is that the input scaled implementations have more fast TSP instance runs than their non-scaled counterparts, except for 4NodePerStepScale in the $0 - 100$ second bin. Problems that finish quickly (i.e., take 300 seconds or less), or "easy" problem instances, are more likely to occur when scaling the inputs rather than not. We see that although the number of fastest instances' runs decreases slightly when scaling the input size threshold, the number of "easy" problem instances' runs is a greater portion of the total runs. The easiest TSP problem instances require the smallest amount of sub-problem expansion, but only benefit marginally from not scaling their input threshold size on the shortest running problems with 4 nodes per step.

When looking at longer running instances, we see that the non-scaled implementations have worse tail end execution than their scaled implementations. For "hard" problem instances, or TSP instances that take more than 300 seconds to solve, the scaled input implementation has fewer runs. When problems become more difficult, more inputs are expanded, and the overhead of not having full vectors for every step in the non-scaled input implementation starts to outweigh its depth-first focused benefits. The "easy" TSP instances prune more inputs in the earlier steps since the non-scaled input size lends itself to a depth-focused rather than breadth-focused search, hence happening upon a better incumbent solution faster while examining less input. In contrast, the scaled input implementation provides many full vectors to the later steps and can not only prune more results on the GPU but also call the GPU fewer times, reducing CPU-GPU communication overhead.

5.6.2 Step Size

To determine if step size had a meaningful impact on application throughput, we tested step sizes ranging from 2 to 4, the latter being the maximum feasible step size given our limits on overall GPU memory. We used the input size scaling described in Section 5.6.1. Figure 5.3 also shows a histogram of the number of TSP instance runs which fell in a CPU wall-clock time interval over all TSP instances tested on each step size configuration.

Table 5.1: Total and Average CPU wall-clock time over all 90 different TSP instances, each tested 20 times.

Topology	Total CPU Wall-Clock Time (s)	Average CPU Wall-Clock Time (s)
2NodePerStepScale	347177.83	192.88
3NodePerStepScale	237249.64	131.81
3NodePerStepNoScale	343460.69	190.81
4NodePerStepScale	226236.70	125.69
4NodePerStepNoScale	289799.45	161.00

In Figure 5.3, we see from the number of TSP instance runs that a step size of 4 (4NodePerStepScale) has the most fastest running instances, a step size of 3 (3NodePerStepScale) has slightly fewer fastest running instances, and a step size of 2 (2NodePerStepScale) has many less fastest running instances. When looking at "harder" problems, or those that take more time for the application to complete, a step size of 4 is comparable to a step size of 3 and better than a step size of 2. Table 5.1 shows the total and average CPU wall-time over all the TSP instances' runs. We see that for total and average time, 4 nodes per step runs the fastest with 3 nodes per step not far behind.

Clearly, step size influences the running time of a TSP instance in our implementation. In terms of deciding which step size configuration to use, for this specific application a step size of 4 has the best performance in terms of total and average CPU wall-clock time and number of fastest running instances. Thus, considerations on memory usage are the next most important factor. Step size 3 has a lower minimum output buffer size but has slightly worse performance than a step size of 4. Although a step size of 2 requires less output buffer space than a 4 or 3 step implementation, the much higher overall running time does not justify its use. Testing larger step sizes was not feasible with the current implementation, as the minimum required space for the first step would become too large.

5.7 Conclusions and Future Work

Dataflow streaming applications are found in numerous fields and have great potential for parallelization. Wide-SIMD architectures, such as GPUs, are great targets for this parallelization because of the large amounts of independent data that need processing. However, realizing these applications becomes difficult in the presence of *dataflow irregularity*, for which we use queues to manage inputs. However, for iterative applications such as branching search, the data frontier can be so large that it cannot fit into GPU memory. Thus, we break down the application into *steps*, where data between steps is pooled on the host CPU and later consumed by the device GPU. Determining the best input size for each step is a tradeoff between the maximum number of frontier outputs that can fit in GPU memory and the occupancy within each step, which directly affects application throughput. Furthermore, deciding how many iterations to do on the GPU in each step also affects both the maximum frontier space required and the application throughput through the occupancy of the GPU kernel and when better incumbent solutions are reached.

We showed that input size scaling is important for "hard" problems, where better incumbent solutions may take longer to find and inputs are expanded more often. GPU occupancy plays a role in overall application throughput once inputs are expanded enough times. We also showed that selecting the right step size can be crucial to application throughput. Using a larger value of step size led to the best results, with a short step size incurring too much CPU-GPU overhead by comparison.

In future work, we would like to explore if a heuristic could be developed for determining high-throughput candidate implementations of iterative applications. This would involve easily gatherable runtime statistics using a representative dataset. Ideally, a single step size configuration and input size would be used for finding a result, but different step sizes may be beneficial at different depths in the tree, since the maximum gain per iteration tends to decrease as depth increases and set of remaining choices decreases.

Another idea for minimizing CPU-GPU communication overhead is to overlap running kernels with those doing data transfers. In other words, one can hide the data transfer latency between the CPU and GPU. Modern NVIDIA GPUs have multiple memory management engines which allow for a different GPU kernel to copy data to and from the CPU than the kernel actively using the GPU's processors. Additionally, the CPU and GPU can run concurrently with one another, which can further increase application parallelism.

Finally, we would like to explore GPU frontier offloading, where the GPU takes more inputs than the maximum gain that the step will produce. Here, the GPU notifies the CPU when the output buffer on the GPU has filled but the GPU still has inputs remaining. In response, the GPU's output buffer is copied to host memory, freeing up the buffer space so the GPU can continue to run. This approach would allow iterative applications to have much larger input sizes, especially at earlier steps where expansion factors are high, because the output buffer need not be sized large enough to hold all possible output at once. With larger input sizes, more GPU blocks could be utilized for higher application throughput. This would also make iterative application implementations feasible on lower-memory GPUs.

Chapter 6

Conclusions and Future Work

Through this dissertation, we explored various optimizations for irregular dataflow streaming applications. Chapter 1 motivated the work by providing example applications. We also discussed how to represent said applications on our target wide-SIMD architecture, the GPU, and disseminated their relevant optimization metrics. Chapter 2 explored how region-based state implementations affect application throughput, and when certain optimizations are worthwhile. Chapter 3 examined how best to statically assign queue space for irregular dataflow streaming applications given easily obtainable runtime statistics. In Chapter 4, we showed how minimum queue size constraints induced by our scheduling algorithm can reduce application throughput, and can be fixed by interrupting compute node execution. Finally, Chapter 5 implemented iterative applications, namely branching search, which explores how application design, particularly pipeline length and input sizes, affect application throughput. We conclude by providing thoughts on future work directions, both for optimizations presented in this work and for potential optimization targets.

6.1 Design Space Automation

Automating the process of profile gathering and profile-guided optimization for MERCATOR applications is important for future use of this work. The execution statistics supporting each of our optimizations are straightforward to collect. Currently, we 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.

When looking at throughput optimizations throughout this work, we utilized the *average gain* to provide a reasonable characterization of how the dataflow of our application would behave. Recall that the gain of an input is *data-dependent*. While the average gain may be a good general indication of dataflow behavior, it does not provide any insight as to the gains of different parts of the data stream. We know from some of our applications that the gain of the data stream changes over time and does not follow a monolithic average gain.

For example, in BLAST [1] we know that when we find a "hit" within the data stream, there are likely to be other "hits" following that input. One could use this information to change the queue sizes or the application topology to maximize the throughput for that section of the data stream. By developing a *probability-based evaluation* of the queue sizes and application topology given a snapshot of the execution, one could automatically modify the application design to accommodate variations in the data stream at runtime. This would require some speculation on future execution, but a reasonable estimate could be inferred by the current inputs and hints about the application from the developer.

6.2 Thread to Work Mapping

When looking at how much work individual nodes perform for irregular dataflow streaming applications, each thread may not have many operations to perform. We discussed in Chapter 3 that merging compute nodes could have benefits to application throughput by giving threads more work when running a specific compute node and removing the overhead of intermediate data queues. We also see from some of our test applications that the overall application running times can be short, even with large data sets. Although node merging is a simple form of providing more work to individual threads, certain pipelines are of the same node type and could benefit from assigning more work to threads when a node is called.

In terms of our application model, one could add capability for threads to look at more than one data input at a time, effectively *mapping multiple inputs to a single thread*, which we will call a *multimap node*. When a multimap node has a sufficient number of inputs to run with full vectors, each thread will be assigned a vector of inputs that can be iterated over. This increases the amount of work available to the thread, allowing nodes with few operations to verify output queue space less often. In the context of node merging, a multimap node will not run the inputs of later nodes, but instead will run more inputs for a specific node, increasing application throughput while maintaining the same occupancy rate.

6.3 Kernel Runtime Communication Overlap

Most of the applications presented in this work either run in a single GPU kernel call or are designed as a "snapshot" of a single slice of a running application. In Chapter 5, we discussed iterative applications, which require multiple GPU kernel calls to complete. Current work does not look at how overlapping CPU-GPU communication with kernel running time for iterative applications affects overall application throughput.

Ideally, one should be able to find a reasonable range of input sizes that have similar kernel running time to that of CPU-GPU communication with any additional CPU computation. In the case of our branching search applications, some operations are performed on the CPU because those points are natural for synchronization between threads. If one could effectively overlap the CPU-GPU communication and CPU computations with the GPU kernels (i.e. have two GPU kernels swapping with each other on the GPU, one running and the other transferring data), overall running times could be greatly reduced.

6.4 Output Buffer Offloading

One issue with some of our applications is their *expansionary* nature, which means the output buffers on the GPU that are returned to the CPU must be very large. This is a similar issue to what was posed in Chapter 4, but instead involves an area outside of our current application model's scheduler and queueing system: CPU-GPU communications. Current work requires that the output buffer of an application be sized to house the most possible number of outputs based on the number of inputs given to the application and the maximum cumulative gain of the application pipeline. This limits how large the GPU input can be to the specific GPU's amount of memory. For many of the problems tested in this work, memory constraints limited how large of problems could be done, even if all the data could be processed in mere seconds. In contrast, a modern host system has a CPU which has access to a much larger memory pool than the GPU.

To remedy this, we propose *output buffer offloading*, where the GPU sends results in the output buffer back to the CPU before completing the kernel. This allows the GPU to continue processing a large input data set without having to restart the kernel multiple times with small chunks of the input. Much like in Chapter 4 of this work, where nodes within the application pipeline are interruptible, the *entire kernel* is now interruptible. Kernels could then allocate much smaller output buffer space, granting more space for input data for applications that are memory bound, or providing more space for intermediate data queues since larger queues are known to provide throughput advantages. Moreover, removing the limits on input size due to output space requirements could permit greater GPU occupancy for expansionary applications.

6.5 Output Overflow Arena

Current queues are designed as a single circular buffer which has pointers for determining the head and tail of the queue. Our scheduler (AFIE [27]) requires that we have at least a specific amount of queue space for ensuring we do not overflow our queues, as discussed in Chapter 5. Since the required queue space is relative to the output gain of a compute node, we implemented *interruptible nodes* to reduce the gain at any given moment of time to *one vector width*. This effectively reduced the required amount of queue space for any compute node's output queue to one vector width, rather than the vector width times the maximum output gain of the node. However, we showed that pausing execution requires *saving and restoring execution state*, which is an expensive operation and requires modifications to user code.

Another idea for removing minimum queue size requirements that does not require the user to re-write their compute node code is to have an *output overflow arena*. This arena would be a chunk of global memory that can be used if any compute node overflows its output queue. The idea is that compute nodes will continue to run until they reach some specified *full size* that is less than the required minimum size. Should the last vector(s) of input produce more outputs than the available output queue space, those outputs would be placed in the arena and gathered up later by the downstream queue. For the applications we explored that have problems associated with minimum queue size, the problem is a large discrepancy between the *maximum gain* and *average gain* for some or all of the compute nodes. Instead of having to check and see if the queue space will overflow if we commit another vector of results, like with interruptible nodes, one could let the buffer overflow and use background logic to retrieve the inputs from the arena as necessary. This method does not require the user to change their node code. However, it is unclear how resolving where data should be gathered from (the queue or the arena) would affect input gathering times. It is also unclear as to how one would manage the arena since it would be available to all blocks at the same time.

6.6 Other SIMD Processors

This work focused on the GPU as our target of wide-SIMD processors. However, we previously mentioned other processors, namely multicore CPUs and FPGAs, which also perform SIMD operations. These processors have different memory systems and instruction constraints than those of GPUs and thus have slightly different performance concerns when it comes to what optimizations are good for improving application throughput.

For example, multicore CPUs have *SIMD instructions* available to each processor, which allows a processor to run in SIMD like on GPUs but still have the flexibility of sequential instructions. The memory pool of these processors is much larger and cheaper to allocate dynamically than on GPUs, since modern processors have secondary storage in their memory hierarchy and are built for numerous, small changes to memory. In terms of the optimizations presented in this work, node merging would have the most applicability to multicore CPUs since when one compacts the output stream is what still determines the occupancy of SIMD operations within a node. Comparing a multicore CPU implementation that leverages the greater sequential flexibility of the processor to a GPU implementation with our optimizations would provide a better understanding as to when a GPU implementation is desired over a multicore-CPU one.

FPGAS exhibit much greater similarity to GPUs in terms of their implementation constraints. There is the same concern over fixed queue space since die space on the FPGA is used for those queues and cannot be resized after placing and routing. Interrupting execution becomes simpler with the addition of *back-pressure signals*, which can tell nodes earlier in the pipeline to halt execution and wait for downstream nodes to process more of their inputs before resuming. An interesting next step would be to see how a GPU implementation with queue space redistribution both at compile time and then intermittently during runtime compares to performing the queue space redistribution at compile time for an FPGA implementation.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446, 1986.

[3] A. Benoit and Y. Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. *Algorithmica*, 57(4):689–724, Aug 2010.

[4] A. Boukedjar, M. E. Lalami, and D. El-Baz. Parallel branch and bound on a CPU-GPU system. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 392–398. IEEE, 2012.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, 2004.

[6] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain. DIBS: A data integration benchmark suite. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 25–28, New York, NY, USA, 2018. Association for Computing Machinery.

[7] I. Chakroun and N. Melab. Operator-level GPU-accelerated branch and bound algorithms. *Procedia Computer Science*, 18:280–289, 2013.

[8] R. Chamberlain, M. Franklin, E. Tyson, J. Buckley, et al. Auto-Pipe: streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, 2010.

[9] S. V. Cole and J. Buhler. MERCATOR: A GPGPU framework for irregular streaming applications. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 727–736, 2017.

[10] S. V. Cole, J. R. Gardner, and J. D. Buhler. WOODSTOCC: Extracting latent parallelism from a DNA sequence aligner on a GPU. In *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*, pages 197–204. IEEE, 2014.

[11] B. Consortium. B. volume and b. july: Bluetooth spec., 1999. https://www.bluetooth.com/specifications/specs/core-specification-5-3/, retrieved 2/7/2023.

[12] Cray Research, Inc. Cray-1 computer system hardware reference manual 2240004, rev C, 1977.

[13] N. Drakos. Optimal solution for TSP using branch and bound. https://gtl.csa.iisc.ac.in/dsa/node187.html, retrieved 2/7/2023.

[14] J. Eker, J. Janneck, E. A. Lee, J. Liu, J. Ludvig, S. Sachs, , and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE*, 91(1):127–44, 2003.

[15] M. Grossman, A. Simion Sbîrlea, Z. Budimlić, and V. Sarkar. CnC-CUDA: Declarative programming for GPUs. In K. Cooper, J. Mellor-Crummey, and V. Sarkar, editors, *Languages and Compilers for Parallel Computing*, pages 230–245. Springer Berlin, 2011.

[16] M. Harris. A brief history of GPGPU, 2014. https://www.cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf, retrieved 2/7/2023.

[17] S. Kato, K. Lakshmanan, R. Rajkumar, Y. Ishikawa, et al. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[18] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. 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)*, pages 1–10. IEEE, 2017.

[19] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75:1235–1245, 1987.

[20] P. Li, K. Agrawal, J. Buhler, and R. Chamberlain. Orchestrating safe streaming computations with precise control. In *4th Int'l Workshop on Extreme Scale Computing Application Enablement - Modeling and Tools*, pages 1017–22, 2014.

[21] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[22] Microsoft Docs. High-level shader language (HLSL), 2021. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl, retrieved 2/7/2023.

[23] R. Mokhtari and M. Stumm. BigKernel–high performance CPU-GPU communication pipelining for big data-style applications. In *2014 IEEE 28th Int'l Parallel and Distributed Processing Symposium*, pages 819–828. IEEE, 2014.

[24] M. Mouly and M.-B. Pautet. *The GSM system for mobile communications*. Telecom publishing, 1992.

[25] NVIDIA. CUDA toolkit documentation: Thread hierarchy, 2022. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#thread-hierarchy, retrieved 2/7/2023.

[26] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multi-tasking on a shared GPU. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015.

[27] T. Plano and 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, pages 1–9, New York, NY, USA, 2020. Association for Computing Machinery.

[28] A. Prokopec and F. Liu. Theory and practice of coroutines with snapshots. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[29] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, page 229–238, USA, 1999. USENIX Association.

[30] J. Subhlok and G. Vondran. Optimal latency–throughput tradeoffs for data parallel pipelines. *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 62–71, 1997.

[31] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In R. N. Horspool, editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[32] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amaransinghe. Teleport messaging for distributed stream programs. In *10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pages 224–35, 2005.

[33] S. Timcheck and J. Buhler. Streaming computations with region-based state on SIMD architectures. In *13th Int'l Wkshp. on Programmability and Architectures for Heterogeneous Multicores*, page 1, 2020.

[34] S. Timcheck and J. Buhler. Interruptible nodes: Reducing queueing costs in irregular streaming dataflow applications on wide-SIMD architectures. *International Journal of Parallel Programming*, pages 1–18, 2022.

[35] S. Timcheck and J. Buhler. Reducing queuing impact in streaming applications with irregular dataflow. *Parallel Computing*, 109:102863, 2022.

[36] S. W. Timcheck and J. D. Buhler. Reducing queueing impact in irregular data streaming applications. In *Proc. of IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 22–30, Nov. 2020.

[37] E. Tyson, J. Buckley, M. Franklin, and R. Chamberlain. Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design

system. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 595:474–479, 10 2008.

[38] P. Viola and M. Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[39] R. Vuduc and J. Choi. A brief history and introduction to GPGPU. In X. Shi, V. Kindratenko, and C. Yang, editors, *Modern Accelerator Technologies for Geographic Information Science*, pages 9–23. Springer US, Boston, MA, 2013.

[40] B. Wu, X. Liu, X. Zhou, and C. Jiang. FLEP: Enabling flexible and efficient preemption on GPUs. *ACM SIGPLAN Notices*, 52(4):483–496, 2017.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Conf. Networked Systems Design and Implementation*, page 2, 2012.

[42] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.