RaftLib: A C++ Template Library for High Performance Stream Parallel Processing

Jonathan C. Beard
Peng Li
Roger D. Chamberlain

Jonathan C. Beard, Peng Li and Roger D. Chamberlain. “RaftLib: A C++ Template Library for High Performance Stream Parallel Processing,” in *Proc. of 6th Int’l Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM), February 2015, pp. 96-105.

Dept. of Computer Science and Engineering
Washington University in St. Louis

RaftLib: A C++ Template Library for High Performance Stream Parallel Processing

Jonathan C. Beard, Peng Li and Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington University in St. Louis
{jbeard,pengli,roger}@wustl.edu

ABSTRACT

Stream processing or data-flow programming is a compute paradigm that has been around for decades in many forms yet has failed garner the same attention as other mainstream languages and libraries (e.g., C++ or OpenMP [15]). Stream processing has great promise: the ability to safely exploit extreme levels of parallelism. There have been many implementations, both libraries and full languages. The full languages implicitly assume that the streaming paradigm cannot be fully exploited in legacy languages, while library approaches are often preferred for being integrable with the vast expanse of legacy code that exists in the wild. Libraries, however are often criticized for yielding to the shape of their respective languages. RaftLib aims to fully exploit the stream processing paradigm, enabling a full spectrum of streaming graph optimizations while providing a platform for the exploration of integrability with legacy C/C++ code. RaftLib is built as a C++ template library, enabling end users to utilize the robust C++ standard library along with RaftLib's pipeline parallel framework. RaftLib supports dynamic queue optimization, automatic parallelization, and real-time low overhead performance monitoring.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: [distributed systems]; D.1.3 [**Concurrent Programming**]: [distributed programming, parallel programming]

1. INTRODUCTION & BACKGROUND

Decries touting the end of frequency scaling and the inevitability of our multi-core future are frequently found in current literature [20]. Equally prescient are the numerous papers with potential solutions to programming multi-core architectures [6, 33, 41, 52]. One of the more promising programming modalities to date is a very old one: stream processing [18, 36] (the term "stream processing" is also used by some to refer to online data processing [13, 21]).

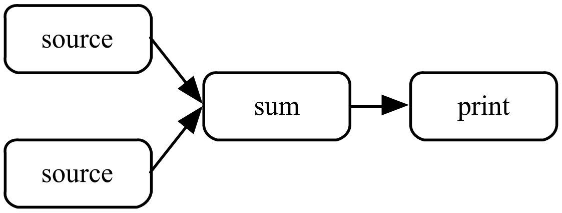

Figure 1: Simple streaming application example with four compute kernels of three distinct types. From left to right: the two source kernels each provide a number stream, the "sum" kernel adds pairs of numbers and the last kernel prints the result. Each kernel acts independently, sharing data via communications streams depicted as arrows.

Stream processing is a compute paradigm that views an application as a set of compute kernels (also sometimes termed "filters" [49]) connected by communication links that deliver data streams. Each compute kernel is typically a sequentially executing unit. Each stream is a first-in, first-out (FIFO) queue whose exact allocation and construction is dependent upon the link type. Figure 1 is an example of a simple streaming `sum` application, which takes in two streams of numbers, adds each pair, and then writes the result to an outbound data stream.

A salient feature of streaming processing is the compartmentalization of state within each compute kernel [1], which simplifies parallelization logic for the run-time [19] as well as the programming API (compared to standard parallelization methods [2]). Therefore, stream processing has two immediate advantages: 1) it enables a programmer to think sequentially about individual pieces of a program while composing a larger program that can be executed in parallel, 2) a streaming run-time can reason about each kernel individually while optimizing globally [38]. Moreover, stream processing has the fortunate side effect of encouraging developers to compartmentalize and separate programs into logical partitions.

In addition to simpler logic, stream processing also enables easier heterogeneous and distributed computation. A compute kernel could have individual implementations that target an FPGA and a multi-core running within the same application, known as "heterogeneous computation" or "hybrid computing" [14]. As long as the inputs and outputs are matching, the application will run identically regard-

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA

ACM 978-1-4503-3404-4/15/02 ..$15.00.
http://dx.doi.org/10.1145/2712386.2712400.

less of which resource a kernel is scheduled to execute on. Brook [12], Auto-Pipe [22], GPU-Chariot [23], and Scala-Pipe [51] are examples of such systems. Stream processing also naturally lends itself to distributed (network) processing, where network links simply become part of the stream.

Despite the promising features of stream processing, there are hurdles that affect programmers' decision to use the paradigm. One hurdle to adoption is communication cost. For the example of Figure 1, the overhead of moving data elements between compute kernels could easily overwhelm the benefits of parallel computation. Clearly, a courser-grained streaming decomposition of an application is more realistic (we will continue to use this example, however, for illustration purposes). A second hurdle is simply the bulk of legacy code and the requirement on the part of most streaming frameworks that applications be re-authored or substantially modified to conform [37].

This paper introduces RaftLib, a C++ template and framework enabling safe and fast stream processing. By leveraging the power of C++ templates, RaftLib can be incorporated with a few function calls and the linking of one additional library. No compiler other than a C++ compiler is needed, so it is completely self contained. By using the native C++ environment, the C++ standard library functionality is available. As to communication between nodes, RaftLib seamlessly integrates TCP/IP networks, and the parallelized execution on multiple distributed compute nodes is transparent to the programmer. RaftLib reduces the communication cost in a multitude of ways, drawing on research from past works [8, 9, 29, 30] and enabling avenues of research that would otherwise be irrelevant in conventional programming models.

2. RELATED WORK

There are many streaming languages and run-time systems designed to assist the development of streaming applications. Brook [12] is a language designed for streaming processing on GPUs. StreamIt [49] is a streaming language and compiler based on the synchronous dataflow model. Storm [46] and Samza [44] are open-source streaming platforms that are focused on message-based data processing. Those languages and frameworks have their "niche" applications. They are not designed for general-purpose programming and usually have steep learning curves. RaftLib's advantage over them is that a C++ template is easy to adopt and has more general usage. ScalaPipe [51] and StreamJIT [11] are two similar language extensions for streaming processing with a Scala frontend and a Java frontend, respectively. Other C++ parallelization template libraries include Threading Building Blocks [43] and Concurrency Collections [28], which are both Intel products. RaftLib differs from the last two in that it aims to provide a fully integrated and dynamically optimized heterogeneous stream parallel processing platform. A recent study of programming language adoption [37] asserts that factors such as lack of legacy code integration, library support, and language familiarity can influence the adoption rate of new languages. According to the monthly TIOBE index [50], the most popular language family by far is C/C++ (anecdotally this is true for scientific computing in general). The hurdles to adoption for a C++ template library are significantly lower than those of a new language (required of many other stream processing systems). To the best of our knowledge, RaftLib is the first C++ library directly targeting streaming processing.

There has been considerable work investigating the efficient execution of streaming applications, both on traditional multi-cores and on heterogeneous compute platforms, from the early work on dedicated data flow engines [19], to the synchronous data flow model [32]. Thies et al. [48] describe a number of performance impacting factors for applications described using StreamIt, and Optimus [26] realizes StreamIt applications on FPGA platforms. Streams-C [24] is an earlier streaming language that targets FPGAs for execution. Lancaster et al. [29, 30] have built low-impact performance monitors for streaming computations, and Padmanabhan et al. [38, 39, 40] have shown how to efficiently search the design space given a model that connects tuning parameters to application performance. RaftLib intends to leverage the above work as it seeks to efficiently execute streaming applications.

3. DESIGN CONSIDERATIONS

Several properties of streaming applications that must be exploited or overcome by streaming systems have been noted by others. The stream access pattern is often that of a sliding window [48], which should be accommodated efficiently. RaftLib accommodates this through a `peek_range` function. Streaming systems, both long running and otherwise often must deal with behavior that differs from the steady state [8, 34, 48]. Non-steady state behavior is often also observed with data-dependent behavior, resulting in very dynamic I/O rates (behavior also observed in [48]). This dynamic behavior, either at startup or elsewhere during execution, makes the analysis and optimization of streaming systems a slightly more difficult prospect, however it is not insurmountable and we will demonstrate empirically how RaftLib handles dynamic rates through a text search application. For example, text search has the property that while the input volume is often fixed, the downstream data volume varies dramatically with algorithms, which heuristically skip, as does the output (pattern matches). Kernel developers, as should be the case, focus on producing the most efficient algorithm possible for a given kernel. Despite this, kernels can become bottlenecks within the streaming system. Raft dynamically monitors the system to eliminate the bottlenecks where possible. We will show how well this works empirically and talk about future work that will expand the real-time modeling and monitoring of these systems.

At one time it was thought that end users were probably best at resource assignment [17], whereas automated algorithms were often better at partitioning an application into compute kernels (synonymous to the hardware-software co-design problem discussed in [5]). Anecdotally we've found that the opposite is often true. Programmers are typically very good at choosing algorithms to implement within kernels, however they have either too little or too much information to consider when deciding where to place a computation and how to allocate memory for communications links. The placement of each kernel changes not only the throughput but also the latency of the overall application. In addition, it is often possible to replicate kernels (executing them in parallel) without altering the application semantics [35]. RaftLib exploits this ability to blend pipeline and data parallelism as well.

Streaming systems can be modeled as queueing networks [8, 31]. Each stream within the system is a queue. The sizes of the queues within the system can have a notable effect on application performance. Too small of queues results in bottlenecks where otherwise none would exist, too big of queues increases the burden on the system (often increasing computation time through more page-ins, cache over-runs, etc.). The effects of queue sizing are shown empirically in Figure 4. RaftLib takes scheduling of compute kernels, allocation of queues, and resource mapping out of the user's hands. Once a compute mapping is defined, the run-time (through heuristics, machine learning, and/or mathematical modeling) attempts to keep the application performing optimally.

4. RAFTLIB DESCRIPTION

Writing parallel code traditionally has been the domain of experts. The complexity of traditional parallel code decreases productivity which can increase development costs [25]. The streaming compute paradigm generally, and RaftLib specifically, enables the programmer to compose sequential code and execute not only in parallel but distributed parallel (networked nodes) using the same code.

RaftLib has a number of useful innovations as both a research platform and a programmer productivity tool. As a research platform, it is first and foremost easily extensible; modularized so that individual aspects can be explored without a full system re-write. It enables multiple modes of exploration: 1) how to effectively integrate pipeline parallelism with standard threaded and/or sequential code, 2) how to reduce monitoring overhead, 3) how best to algorithmically map compute kernels to resources, 4) how to model streaming applications quickly so that results are relevant during execution. It is also fully open source and publicly accessible [42]. As a productivity tool it is easily integrable with legacy C++ code. It allows a programmer to parallelize code in both task and pipelined fashions.

We introduce RaftLib via the following example application. The `sum` kernel from Figure 1 is an example of a kernel written in a sequential manner (code shown in Figure 2). It is authored by extending a base class: `raft::kernel`. Each kernel communicates with the outside world through communications "ports." The base kernel object defines `input` and `output` port user accessible objects. These are inherited by sub-classes of `raft::kernel`. Port container objects can contain any type of port. Each port itself is essentially a FIFO queue. The constructor function of the `sum` kernel adds the ports. In this example, two input ports are added of type `A` & `B` as well as an output port of type `C`. Each port gets a unique name which is used by the run-time and the user. The real work of the kernel is performed in the `run()` function which is called by the scheduler. The code within this section can be thought of as a "main" function of the kernel. Input and output ports can access data via a multitude of intuitive methods from within the `run()` function. Accessing a port is safe, free from data race and other issues that often plague traditional parallel code [7]. Figure 3 shows the full application topology from Figure 1 assembled in code. Assembling the topology can be thought of as connecting a series of actors, which is also a sequential process. Each call to the link function connects the specified ports from the source and destination kernels. The function call returns a struct with references to the linked source and destination kernels for re-use by the programmer if needed. The run-time itself brings the parallel power to these sequential actors.

```
template< typename A,
          typename B,
          typename C > class sum :
            public raft::kernel
{
public:
   sum() : raft::kernel()
   {
      input.addPort<  A >( "input_a" );
      input.addPort<  B >( "input_b" );
      output.addPort< C >( "sum" );
   }

   virtual raft::kstatus run()
   {
      auto a( input[ "input_a" ].pop_s<   A >() );
      auto b( input[ "input_b" ].pop_s<   B >() );
      auto c( output[ "sum" ].allocate_s< C >() );
      (*c) = (C)((*a) + (*b));
      return( raft::proceed );
   }
};
```

Figure 2: A simple example of a `sum` kernel which takes two numbers in via `input_a` and `input_b`, adds them, and outputs them via the `sum` stream. With the `pop_s` and `allocate_s` functions, objects are returned which automatically pop and send items, respectively.

Once the kernel "actors" are assembled into a full application, the run-time starts to work parallelizing the application with the `exe()` function call. This feat is efficiently performed by mapping kernels to appropriate resources, sizing buffers, selecting the appropriate algorithm when more than one exists, scheduling kernels for execution, and tuning any remaining performance impacting run-time parameters.

Scheduling, mapping, and queueing behavior are each important to efficient, high-performance execution. RaftLib is intended to facilitate empirical investigation within each of these areas. RaftLib implements a simple but effective scheduler that is straightforward to substitute with new algorithms. Similarly, the modular mapping algorithms used in RaftLib can easily be altered for comparative study. Each communication link between compute kernels exhibit queueing behavior. RaftLib serves as a platform for optimizing the queueing network, not only statically but dynamically. RaftLib supports continuous optimization of a host of run-time settable parameters.

There are many factors that have led to the design of RaftLib. Chief amongst them is the desire to have a fully open source framework to explore how best to integrate stream processing with legacy code. Secondly it serves as an experimental platform for investigating optimized deployment and optimization of stream processing systems. In the following sections we discuss why we need the features included in the library, the science and engineering behind them and some examples of how those features are executed by the user. This will be followed by benchmark-

```
const std::size_t count( 100000 );
auto linked_kernels(
   map.link( kernel::make<
               generate< std::int64_t > >( count ),
             kernel::make<
                 sum< std::int64_t,
                      std::int64_t,
                      std::int64_t > >(),
               "input_a" ) );
map.link(
   kernel::make< generate< std::int64_t > >( count ),
   &( linked_kernels.dst ),
   "input_b" );
map.link(
   &( linked_kernels.dst ),
   kernel::make< print< std::int64_t ,'\n'> >() );
map.exe();
```

Figure 3: Example of a streaming application map for a "sum" application (topology given in Figure 1). Two random number generators are instantiated, each of which sends a stream of numbers to the `sum` kernel which sends the sum to a `print` kernel. The call to `link()` returns a struct (`linked_kernels`) with references to the kernels used within the `link()` function call (`linked_kernels.src` and `linked_kernels.dst` respectively) so that they may be referenced in subsequent link calls.

ing a text searching application against other leading parallel text search applications.

4.1 RaftLib as a Research Platform

As a research platform, RaftLib is designed to enable the investigation of a number of questions that impact the performance of streaming applications. We will address a number of these questions in the paragraphs below, with a focus not on the answer to the research question, but instead on how RaftLib facilitates the investigation.

We start with the ability to blend pipeline parallelism with data parallelism. Some applications require data to be processed in order, others are okay with data that is processed out of order, yet others can process the data out of order and re-order at some later time. RaftLib accommodates all of the above paradigms. Streams that can be processed out of order are ideal candidates for the run-time to automatically parallelize. Li et al. [35] describe algorithms for replicating kernels in a pipelined environment, both for homogeneous compute resources and for heterogeneous compute resources.

Automatic parallelization of candidate kernels is accomplished by analyzing the graph for segments that can be replicated preserving the application's semantics (indicated by the user at link type with template parameters). There are default split and reduce adapters that are inserted where needed. Custom split reduce objects can be created by the user by extending the default split / reduce objects. Split data distribution can be done in many ways, and the run-time attempts to select the best amongst round-robin and least-utilized strategies (queue utilization used to direct data flow to less utilized servers). As with all of the specific mechanisms that we will discuss, each of these approaches is designed to be easily swapped out for alternatives, enabling empirical comparative study between approaches.

Given an application topology to execute (possibly including some replicated kernels), the kernels need to be assigned to specific compute resources and scheduled for execution. We refer to the assignment of kernels to compute resources as the mapping problem, and the initial mapping algorithm provided with RaftLib is a simple one (similar to a spanning tree) that attempts to place the fewest number of "streams" over high latency connections (i.e., across physical compute cores or TCP links). It begins with a priority queue with the highest latency link getting the highest priority, finds the partition with the minimal number of links crossing it then proceeds to partition based on the next highest latency link for these two partitions. If no difference in latency exists (which can be the case if only a single socket core is used) then computation is shared evenly amongst the cores. No claim is made to optimality for this simple algorithm, however it is fast.

Individual kernels are implemented in the library as independent execution units (i.e., a thread, process, FPGA accelerator, etc.). The initial scheduling algorithm for threads and processes is simply the default thread-level scheduler provided by the underlying operating system. FPGA accelerator cards and GPGPU(s) utilize their own schedulers. We anticipate incorporating recent work [3] which supports cache-aware scheduling of pipelined computations. RaftLib, of course, allows the substitution of any scheduler desired.

As illustrated in Figure 4, the allocated size of each queue of a streaming application can have a significant impact on performance (the figure is for a matrix multiply application, performance based on overall execution time). One would assume perhaps that simply selecting a very large buffer might be the best choice, however as shown the upper confidence interval begins to increase after about eight megabytes. Queueing models are often the fastest way to estimate an approximate queue size, however service rates and their distributions must be determined, which is hard to do during execution. In general, two options are available for determining how large of a buffer to allocate: branch and bound search or analytic modeling. Branch and bound searching has the advantage of being extremely simple, and eventually finds some reasonable condition. If the queue is destined to be of infinite size, a simple engineering solution presents itself in the form of a buffer cap. Model based solutions are also often straightforward to calculate, assuming the conditions are right for considering each queue individually (e.g., the queueing network is of product form).

While treating compute kernels as a "black" box, queue sizing approaches must accommodate program correctness. If a kernel asks to receive five items and the buffer size is only allocated for two, the program cannot continue. RaftLib deals with this by detecting this condition with a monitoring thread, updated every $\delta \leftarrow 10\ \mu s$. When conditions dictate that the FIFO needs to be resized, it is done using lock-free exclusion and only under certain conditions (to maximize resizing efficiency). The resizing operation is most efficiently accomplished when the read position is less than the write pointer (i.e., the queue or ring-buffer is in a non-wrapped position). There are multiple conditions that could trigger a resize and they differ depending on the end of the queue under consideration. On the side writing to the queue, if the write process is blocked for a time period of $3 \times \delta$ then the

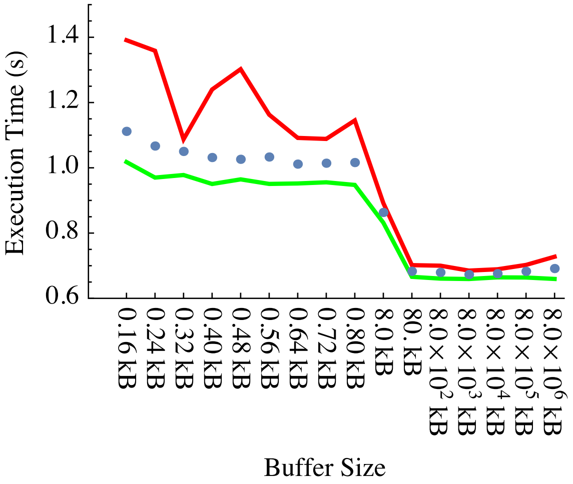

Figure 4: Queue sizes for a matrix multiply application, shown for an individual queue (all queues sized equally). The dots indicate the mean of each observation (each observation is a summary of 1k executions). The red and green lines indicate the 95^{th} and 5^{th} percentiles respectively. The execution time increases slowly with buffer sizes $\geq$ 8 MB, as well as becoming far more varied.

queue is resized. On the read side, if the reading compute kernel requests more items than the queue has available then the queue is tagged for resizing. Temporary storage is provided on the receiver end to ensure the request is fulfilled as well as ensure that conditions for fast resizing are met.

Considering the application as a whole for optimization is also possible for RaftLib. Prior works by Beard and Chamberlain [8] demonstrate the use of flow models to estimate the overall throughput of an application. This procedure however requires estimates of the output distribution for each edge within the streaming application. The flow-model approximation procedure can be combined with well known optimization techniques such as simulated annealing or analytic decomposition [38, 39, 40] to continually optimize long-running high throughput streaming applications.

The "share-nothing" mantra of stream processing might introduce extra overhead, however it enables fairly easy programming of massively parallel systems. Each compute kernel can be easily duplicated on the same system, on different hardware across network links or even on GPGPU systems. As a research vehicle, RaftLib enables studies that explore how the communication and resource placement can be optimized. As a productivity tool, we are more interested in how few lines of code it takes to produce a result. Mentioned but not described has been the distributed nature of RaftLib. The capability to use TCP connections for many systems is clunky at best. With RaftLib there is no difference between a distributed and a non-distributed program from the perspective of the developer. A separate system called "oar" is a mesh of network clients that continually feed system information to each other. This information is provided to RaftLib in order to continuously optimize and monitor Raft kernels executing on multiple systems. The "oar" system also provides a means to remotely compile and execute kernels so that a user can have a simple compile and forget experience. Future work will see the full and complete integration of both TCP links and GPGPU kernels.

Performance monitoring is critical to the exploration of algorithms. It is also central to the automated optimization and modeling that is part of RaftLib. As such the user has access to monitor useful things such as queue size, current kernel configuration as they are updated by the run-time. In addition to these, more exciting statistics such as mean queue occupancy, service rate, throughput, queue occupancy histograms are available. The data collection process itself is optimized to reduce overhead and has been the subject of much research [29, 30]. Future work in visualization could determine the best way to display this information to the user in order to improve their ability to act upon it.

4.2 Authoring Streaming Applications

Next we consider the authoring of application. We'll show some code segments to see how little code is needed to write a parallel algorithm and how familiar it can be to C++ programmers.

RaftLib views each compute kernel as a black-box at the level of a port interface. Once ports are defined, the only observability that the run-time has is the interaction of the algorithm implementation inside the compute kernel with those ports. A new compute kernel is defined by extending `raft::kernel` as in Figure 2. Kernels have access to add named ports, with which the kernel can access data from incoming or write to outgoing data "streams." Programmers building a kernel have a multitude of options to access data on the kernel. The example in Figure 2 shows the simplest method (`pop_s`) to get data from the input stream, which is a return object which gives the user a reference to the head of the incoming queue (for variables `a` & `b`). A reference to the output queue is given by the `allocate_s` function. When each of these variables exits the calling scope, they are popped from the incoming queues and pushed to the outgoing queue respectively. The return objects from the `allocate_s` and `pop_s` calls also have associated signals accessible through the `sig` variable. There are multiple calls to perform push and pop style operations, each embodies some type of copy semantic (either zero copy or single copy), all provide a means to send or receive synchronous signals. Synchronized signaling is implemented so that downstream kernels will receive the signal at the same time the corresponding data element is received (useful for things like end of file signals). Asynchronous signaling (i.e., immediately available to downstream kernels) is also available. Future implementations will utilize the asynchronous signaling pathway for global exception handling.

Arranging compute kernels into an application is one of the core functionalities of a stream processing system. RaftLib has an imperative mode of kernel connection via the `link` function. The `link` function call has the effect of assigning one output port of a given compute kernel to the input port of another compute kernel. A `map` object is defined in the `raft namespace` of which the `link` function is a member. Figure 3 shows our simple example application which takes two random number generating kernels, adds pairs of the random numbers using the `sum` kernel and prints them.

When the user runs the `exe()` function of `map` object, the graph is first checked to ensure it is fully connected, then type checking is performed across each link. Before a link allocation type is selected (POSIX shared memory,

```
/** data source container **/
std::vector< std::uint32_t > v;
int i( 0 );
/** fill container **/
auto func( [&](){ return( i++ ); } );
while( i < 1000 ){ v.push_back( func() ); }
/** receiver container **/
std::vector< std::uint32_t > o;
/** read from one kernel and write to another **/
map.link(
   kernel::make< read_each< std::uint32_t > >
   (  v.begin(),
      v.end() ),
   kernel::make< write_each< std::uint32_t > >
   ( std::back_inserter( o ) ) );
/** data is now copied to 'o' **/
```

Figure 5: Syntax for reading and writing to C++ standard library containers from raft::kernel objects. The `read_each` and `write_each` kernels are reading and writing on independent threads.

heap allocated memory or TCP link), and each kernel is mapped to a resource. This could be pinning the thread or heavyweight process to a compute core, mapping the kernel to another compute node over a distributed system or even potentially a GPGPU. Once the link allocation types are selected, the run-time selects the narrowest convertible type for each link type and casts the types at each endpoint. Future versions will incorporate link data compression as well, further improving the cache-able data. Once memory is allocated for each link, a thread continuously monitors all the queues within the system and reallocates them as needed (either larger or smaller) to improve performance.

Streaming applications are often ideally suited for long running, data intense applications such as big data processing or real-time data analytics. The conditions for these applications often change during the execution of a single run. Algorithms use different optimizations based on differing inputs (i.e., sparse matrix vs. dense matrix multiply). Changing conditions can often benefit from additional resources or differing types of algorithms within the application to eliminate bottlenecks as they emerge. RaftLib gives the user the ability to specify synonymous kernel groupings that the run-time can swap out to optimize the computation. These can be kernels that are implemented for multiple hardware types, or can be differing algorithms. For instance, a version of the UNIX utility `grep` could be implemented with multiple search algorithms. Some of these algorithms require differing pieces, however they can all be expressed as a "search" kernel.

Integration with legacy C++ code is one of our goals. As such, it is imperative that RaftLib work seamlessly with the C++ standard library functions. Figure 5 shows how a C++ container can be used directly as an input queue to a streaming graph, in parallel if out of order processing is allowed. Just as easily, a single value could be read in. Output integration is simple as well, standard library containers maybe be output queues, or a reduction to a single output value is possible.

Copying of data is often an issue as well within stream processing systems. RaftLib provides a `for_each` kernel (Figure 6), which has behavior distinct from the `write_each` and `read_each` kernels. The `for_each` takes a pointer value and uses its memory space directly as a queue for downstream compute kernels. This is essentially a zero copy and enabling behavior from a "streaming" application similar to that of an OpenMP [15] parallelized loop. Unlike the C++ standard library `for_each`, the RaftLib version provides an index to indicate position within the array for the start position. This enables the compute kernel reading the array to calculate the position within it. When this kernel is executed, it appears as a kernel only momentarily, essentially providing a data source for the downstream compute kernels to read.

```
int *arr = { 0, ..., N };
int val = 0;
auto &kernels(
map.link( kernel::make< for_each< int > >( arr,
                                           arr_length ),
          kernel::make< some_kernel< int > >() ) );
map.link( &( kernels.dst ),
   kernel::make<
      reduce< int,
              func /* reduct function */ >
               >( val ) );
/** val now has the result **/
```

Figure 6: Example of the `for_each` kernel, which is similar to the C++ standard library `for_each` function. The data from the given array is divided amongst the output queues using zero copy, minimizing data extraneous data movement.

Code verbosity is often an issue. Readily available in C++ is the declaration of a class or a template, when often what is wanted is the ability to pass a simple function and have it executed by the called function. Newer languages and C++11 have met this demand with lambda functions. RaftLib brings lambda compute kernels, which give the user the ability to declare a fully functional, independent kernel while freeing him/her from the cruft that would normally accompany such a declaration. Figure 7 demonstrates the syntax for a single output random number generator. The closure type of the lambda operator also allows for usage of the `static` keyword to maintain state within the function [16]. These kernels can be duplicated and distributed, however they do induce one complication if the user decides to capture external values by reference instead of by value, undefined behavior may result if the kernel is duplicated; especially across a TCP link (an issue we will resolve in subsequent versions of RaftLib).

5. BENCHMARKING

Text search is used in a variety of applications. We will focus on the exact string matching problem which has been studied extensively. The stalwart of string matching applications (both exact and inexact) is the GNU version of the `grep` utility. It has been developed and optimized for 20+ years resulting in excellent single threaded exact string matching performance (∼ 1.2 GB/s) on our test machine (see Table 1). To parallelize GNU `grep`, the GNU Parallel [47] utility is used to spread computation across one

```
map.link(
   /** instantiate lambda kernel as source **/
   kernel::make< lambdak< std::uint32_t > >( 0, 1, []
      ( Port &input,
        Port &output )
      {
         auto out(
         output[ "0" ].allocate_s< std::uint32_t >() );
         (*out) = rand();
      } /** end lambda kernel **/ ) /** end make **/,
      /** instantiate print kernel as destination **/
   kernel::make< print< float, ‘\n’ > >() );
```

Figure 7: Syntax for lambda kernel. The user specifies port types as template parameters to the kernel, in this example `std::uint32_t`. If a single type is provided as a template parameter, then all ports for this lambda kernel are assumed to have this type. If more than one template parameter is used, then the number of types must match the number of ports given by the first and second function parameters (input and output port count, respectively). The number of input ports is zero and the number of output ports is one for this example. Ports are named sequentially starting with zero. The third parameter is a lambda function which is called repeatedly by the runtime.

through 16 cores. Two differing text search algorithms will be tested and parallelized with RaftLib. One will utilize the Aho-Corasick [4] string matching algorithm which is quite good for multiple string patterns. The other will use the Boyer-Moore-Horspool algorithm [27] which is often much faster for single pattern matching. The realized application topology for both string matching algorithms implemented with RaftLib are conceptually similar to Figure 8, however the file read exists as an independent kernel only momentarily as a notional data source since the run-time utilizes zero copy, and the file is directly read into the in-bound queues of each `match` kernel.

Figure 9 shows code necessary to generate the application topology used to express both string matching algorithms using RaftLib. Not shown is the code to handle arguments, setup, etc. Note that there is no special action required to parallelize the algorithm. The `filereader` kernel takes the file name, it distributes the data from the file to each string matching kernel. The programmer can express the algorithm without having to worry about parallelizing it. The programmer simply focuses on the sequential algorithm. Traditional approaches to parallelization require the programmer to have knowledge of locks, synchronization, and often cache protocols to safely express a parallel algorithm. Even more exciting is that when using RaftLib, the same code can be run on multi-cores in a distributed network without the programmer having to do anything differently.

For comparison we contrast the performance of our implementations of Aho-Corasick and Boyer-Moore-Horspool against the GNU `grep` utility and a text matching application implemented using the Boyer-Moore algorithm implemented in Scala running on the popular Apache Spark framework. We'll use a single hardware platform with multiple cores and a Linux operating system (see Table 1).

```
auto kern_start(
   map.link< raft::out >(
      kernel::make<
         filereader >( file,
                       offset ),
      kernel::make<
         search< ahocorasick /** or boyermoore **/ >
                       >( search_term ) ) );

   map.link< raft::out >(
     &(kern_start.dst),
     kernel::make<
        write_each< match_t > >(
           std::back_inserter( total_hits ) ) );
```

Figure 9: Implementation of the string matching application topology using RaftLib. The actual search kernel is instantiated by making a `search` kernel. The exact algorithm is chosen by specifying the desired algorithm as a template parameter to select the correct template specialization.

We use version 2.20 of the GNU `grep` utility. In order to parallelize GNU `grep`, the GNU Parallel [47] application is used (version 2014.10.22), with the default settings. RaftLib (and all other applications/benchmarks used) is compiled using GNU GCC 4.8 with compiler flags "-Ofast." When parallelizing all algorithms, the maximum parallelism is capped to the number of cores available on the target machine. A RAM disk is used to store the text corpus to ensure that disk IO is not a limiting factor. The corpus to search is sourced from the post history of a popular programming site [45] which is ∼ 40 GB in size. The file is cut to 30 GB before searching. This cut is simply to afford the string matching algorithms the luxury of having physical memory equal to the entire corpus if required (although in practice none of the applications required near this amount). All timing is performed using the GNU `time` utility (version 1.7) except the Spark application, which uses its own timing utility.

Table 1: Summary of Benchmarking Hardware.

Processor	Cores	RAM	OS Version
Intel Xeon E5-2650	16	62 GB	Linux 2.6.32

Figure 10 shows the throughput (in GB/s) for all of the tested string matching applications, varying the utilized cores from one through 16. The performance of the GNU `grep` utility when single threaded is quite impressive. It handily beats all the other algorithms for single core performance (when not using GNU Parallel, as shown in the figure). Perfectly parallelized (assuming linear speedup) the GNU `grep` application could be capable of ∼ 16 GB/s. When parallelized with GNU Parallel however, that is not the case.

The performance of Apache Spark when given multiple cores is quite good. The speed-up is almost linear from a single core though 16 cores. The Aho-Corasick string matching algorithm using RaftLib performs almost as well, topping out at ∼ 1.5 GB/s to Apache Spark's ∼ 2.8 GB/s. RaftLib has the ability to quickly swap out algorithms during execution, this was disabled for this benchmark so we could more easily compare specific algorithms. Manually changing

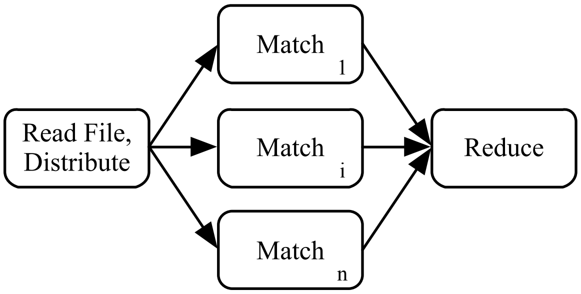

Figure 8: String matching stream topology for both Boyer-Moore-Horpool and Aho-Corasick algorithms. The first compute kernel (at left) reads the file and distributes the data. The second kernel labeled `Match` uses one or the other algorithms to find string matches within the streaming corpus. The matches are then streamed to the last kernel (at right) which combines them into a single data structure.

the algorithm RaftLib used to Boyer-Moore-Horspool, the performance improved drastically. The speed-up from one through 10 cores is now linear, with the 30 GB file searched in ~ 4.1 s which gives it close to 8 GB/s throughput.

Overall the performance of the RaftLib Aho-Corasick string matching algorithm is quite comparable to the one implemented using the popular Apache Spark framework. The Boyer-Moore-Horspool however outperforms all the other algorithms tested. The change in performance when swapping algorithms indicates that the algorithm itself (Aho-Corasick) was the bottleneck. Once that bottleneck is removed we found that the memory system itself becomes the bottleneck. Future work with cache aware scheduling and pipeline prefetch could perhaps improve performance further by reducing memory latency. All in all the performance of RaftLib is quite good, comparable with one of the best current distributed processing frameworks (Apache Spark) and far better than the popular parallelizing utility GNU Parallel.

6. CONCLUSIONS & FUTURE WORK

RaftLib has many features that enable a user to integrate fast and safe streaming execution within legacy C++ code. It provides interfaces similar to those found in the C++ standard library, which we hope will enable users to pick up how to use the library at a faster pace. We've also shown new ways to describe compute kernels, such as the "lambda" kernels which eliminates much of the "boiler-plate" code necessary to describe a full C++ class or template. What we've also described is a framework for massively parallel execution that is simple to use. The same code that executes locally can execute distributively with the integration of the "oar" network framework. No programming changes are necessary. This differs greatly from many current open source distributed programming frameworks.

What we've done with the RaftLib framework is lay a foundation for future research. How best to integrate stream processing with sequential computation is still an open question. Pragma methods such as OpenMP for loop parallelization work well for parallelizing loops, however they're far from ideal as programmers must fully understand how to use the available options in order to get the most out of OpenMP. RaftLib promises similar levels of parallelism that are automatically optimized by the run-time. Towards this end instrumentation and dynamic optimization must be improved. Things like fast automatic model selection (e.g., Beard et al., [10]), resource to kernel matching and environmental adaptation must be researched and perfected in order for systems such as these to fully exploit the myriad of computational resources available today (multi-cores, vector processors, GPGPUs, etc.).

The RaftLib framework provides a platform for safe and fast parallel streaming execution within the C++ language. It serves as a productivity tool and a research vehicle for exploring integration and optimization issues. Despite the slow adoption rate of stream processing, we hope that the utilization of a widely used existing language (C++) serves as a catalyst to gain more than a niche user base.

7. ACKNOWLEDGMENTS

This work was supported by Exegy, Inc., and VelociData, Inc. Washington University in St. Louis and R. Chamberlain receive income based on a license of technology by the university to Exegy, Inc., and VelociData, Inc.

8. REFERENCES

[1] W. B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, 1982.

[2] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, and C.-W. Tseng. Requirements for data-parallel programming environments. Technical report, DTIC Document, 1994.

[3] K. Agrawal, J. Fineman, and J. Maglalang. Cache-conscious scheduling of streaming pipelines on parallel machines with private caches. In *Proc. of IEEE Int'l Conf. on High Performance Computing*, 2014.

[4] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[5] P. Arató, S. Juhász, Z. Á. Mann, A. Orbán, and D. Papp. Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing*, pages 197–202. IEEE, 2003.

[6] D. C. Arnold, H. Casanova, and J. Dongarra. Innovations of the NetSolve grid computing system.

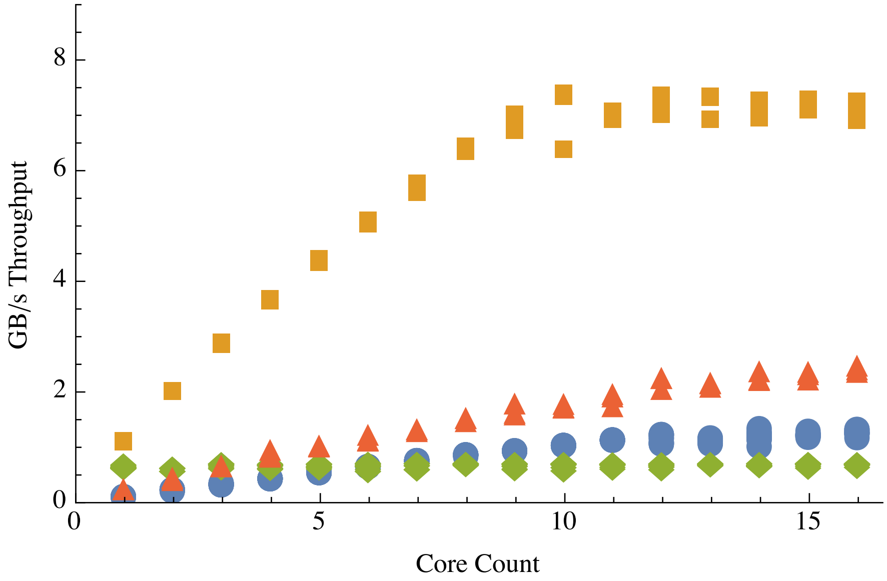

Figure 10: This figure shows the performance of each string matching application in GB/s by utilized cores. This is calculated using a 30 GB corpus searched on the hardware from Table 1. The green diamonds represent the GNU Parallel parallelized GNU grep. The red triangles represent Apache Spark. The blue circles and gold squares represent the Aho-Corasick and Boyer-Moore-Horspool text search algorithms, respectively, parallelized using RaftLib.

Concurrency and Computation: Practice and Experience, 14(13-15):1457–1479, 2002.

[7] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[8] J. C. Beard and R. D. Chamberlain. Analysis of a simple approach to modeling performance for streaming data applications. In *Proc. of IEEE Int'l Symp. on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 345–349, Aug. 2013.

[9] J. C. Beard and R. D. Chamberlain. Use of a Levy distribution for modeling best case execution time variation. In A. Horváth and K. Wolter, editors, *Computer Performance Engineering*, volume 8721 of *Lecture Notes in Computer Science*, pages 74–88. Springer International, 2014.

[10] J. C. Beard, C. Epstein, and R. D. Chamberlain. Automated reliability classification of queueing models for streaming computation using support vector machines. In *Proceedings of the 6th ACM/SPEC international conference on Performance engineering*, ICPE '15, New York, NY, USA, Jan. 2015. ACM. to be published.

[11] J. Bosboom, S. Rajadurai, W.-F. Wong, and S. Amarasinghe. StreamJIT: A commensal compiler for high-performance stream programming. In *Proc. of ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 177–195. ACM, 2014.

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

[13] A. Chakrabarti, G. Cormode, and A. McGregor. Robust lower bounds for communication and stream computation. In *Proc. of 40th ACM Symposium on Theory of Computing*, pages 641–650. ACM, 2008.

[14] R. D. Chamberlain, J. M. Lancaster, and R. K. Cytron. Visions for application development on hybrid computing systems. *Parallel Comput.*, 34(4-5):201–216, May 2008.

[15] R. Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[16] Working Draft, Standard for Programming Language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`. Accessed Ocbober 2014.

[17] G. De Michell and R. K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[18] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[19] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[20] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *38th International Symposium on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011.

[21] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: The value of space. In *Proc. of 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 745–754, Philadelphia, PA, USA, 2005. SIAM.

[22] M. Franklin, E. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-Pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[23] I. Fumihiko, S. Nakagawa, and K. Hagihara. GPU-Chariot: A programming framework for stream applications running on multi-GPU systems. *IEICE Transactions on Information and Systems*, 96(12):2604–2616, 2013.

[24] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 49–56, Apr. 2000.

[25] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Proc. of ACM/IEEE Supercomputing Conference*, pages 35–35. IEEE, 2005.

[26] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah. Optimus: efficient realization of streaming applications on FPGAs. In *Proc. of Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pages 41–50, 2008.

[27] R. N. Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[28] K. Knobe and C. Offner. Compiling to tstreams, a new model of parallel computation. Technical report, Technical report, 2005.

[29] J. M. Lancaster, E. F. B. Shands, J. D. Buhler, and R. D. Chamberlain. TimeTrial: A low-impact performance profiler for streaming data applications. In *Proc. IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, Sept. 2011.

[30] J. M. Lancaster, J. G. Wingbermuehle, J. C. Beard, and R. D. Chamberlain. Crossing boundaries in TimeTrial: Monitoring communications across architecturally diverse computing platforms. In *Proc. 9th IEEE/IFIP Int'l Conf. Embedded and Ubiquitous Computing*, Oct. 2011.

[31] S. S. Lavenberg. A perspective on queueing models of computer performance. *Performance Evaluation*, 10(1):53–76, 1989.

[32] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9), 1987.

[33] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[34] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *ACM Symp. on Parallelism in Algorithms and Architectures*, 2010.

[35] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain. Adding data parallelism to streaming pipelines for throughput optimization. In *Proc. of IEEE Int'l Conf. on High Performance Computing*, 2013.

[36] J. R. McGraw. Data-flow computing: the VAL language. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, 1982.

[37] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *Proc. of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 1–18. ACM, 2013.

[38] S. Padmanabhan, Y. Chen, and R. D. Chamberlain. Optimal design-space exploration of streaming applications. In *Proc. IEEE Int'l Conf. Application-specific Systems, Architectures and Processors*, Sept. 2011.

[39] S. Padmanabhan, Y. Chen, and R. D. Chamberlain. Convexity in non-convex optimizations of streaming applications. In *Proc. of 18th IEEE Int'l Conf. on Parallel and Distributed Systems*, pages 668–675, Dec. 2012.

[40] S. Padmanabhan, Y. Chen, and R. D. Chamberlain. Unchaining in design-space optimization of streaming applications. In *Proc. of Workshop on Data-Flow Execution Models for Extreme Scale Computing*, Sept. 2013.

[41] O. Pell and O. Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News*, 39(4):60–65, 2011.

[42] RaftLib. `http://www.raftlib.io`. Accessed November 2014.

[43] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ For Multi-core Processor Parallelism.* O'Reilly Media, Inc., 2007.

[44] Samza. `http://samza.incubator.apache.org`. Accessed November 2014.

[45] Stack Exchange Data Dump. `https://archive.org/download/stackexchange/stackoverflow.com-PostHistory.7z`. Accessed November 2014.

[46] Storm: Distributed and fault-tolerant realtime computation. `https://storm.apache.org`. Accessed November 2014.

[47] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[48] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376. ACM, 2010.

[49] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In R. Horspool, editor, *Proc. of Int'l Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. 2002.

[50] TIOBE Programming Community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Accessed Ocbober 2014.

[51] J. G. Wingbermuehle, R. D. Chamberlain, and R. K. Cytron. ScalaPipe: A streaming application generator. In *Proc. Symp. on Application Accelerators in High-Performance Computing*, July 2012.

[52] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.