GStream: A General-Purpose Data Streaming Framework on GPU Clusters*

Yongpeng Zhang, Frank Mueller
Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, Email:mueller@cs.ncsu.edu

***Abstract*—Emerging accelerating architectures, such as GPUs, have proved successful in providing significant performance gains to various application domains. However, their viability to operate on general streaming data is still ambiguous. In this paper, we propose GStream, a general-purpose, scalable data streaming framework on GPUs. The contributions of GStream are as follows: (1) We provide powerful, yet concise language abstractions suitable to describe conventional algorithms as streaming problems. (2) We project these abstractions onto GPUs to fully exploit their inherent massive data-parallelism. (3) We demonstrate the viability of streaming on accelerators. Experiments show that the proposed framework provides flexibility, programmability and performance gains for various benchmarks from a collection of domains, including but not limited to data streaming, data parallel problems and numerical codes.**

I. Introduction

Stream processing has established itself as an important application area that is driving the consumer side of computing today. While traditionally used in video encoding/decoding scenarios, other application areas, such as data analysis and computationally intensive tasks are also discovering the benefits of the streaming paradigm. High computational demands by streaming have been met by general-purpose architectures via multicores. But with no end in sight for these inflating demands, conventional architectures are struggling to keep up. We already see significant increases in power and resource management costs particularly for homogeneous general-purpose multicores. Heterogeneous architectures with accelerators, such as GPUs, offer a viable alternative to meet the demand in computing as they deliver not only higher cost and power efficiency but also higher performance and scalability.

These performance potentials of GPUs originate from architectural design and programming strategies in favor of massive data parallelism. Today's latest generation of GPUs features hundreds of stream processing units capable of supporting much more data parallelism than a CPU does. The NVIDIA GPU programming model CUDA encourages users to create light-weight software threads at the scale of tens of thousands, which is orders of magnitude larger than the maximal hardware concurrency inside the GPU. This over-subscription of software threads relative to the hardware parallelism allows latency hiding mechanisms to be realized that mitigate the effects of the memory wall [1].

* This work was supported in part by DOE grant DE-FG02-08ER25837, NSF grants 0237570, 0937908 and 0958311.

Existing streaming models, such as Lucid [2], LUSTRE [3] and SIGNAL [4] (see [5] for a survey), strive to provide a comprehensive streaming abstraction on one end of the spectrum. They are designed to be architecture-independent and focus on generality. This comes at a price as efficient execution under different computing platforms becomes an afterthought. On the other end, various compiler techniques and runtime systems were developed to map streaming abstractions to specific hardware, *e.g.*, StreamIt [6], Brook [7], Cg [8] and Auto-pipe [9].

In this work, we consider a language extension and runtime system approach to map streaming abstractions to GPU *clusters*. Efficient utilization of resources in a GPU cluster is an essential prerequisite for its adoption in streaming domain, especially for large scale, data-intense applications. However, programming the state-of-the-art GPUs is not as flexible as programming CPU clusters. More specifically, we experience two challenges to achieve both programmability and performance:

(1) Deep (multi-level) memory hierarchies in a typical loosely-coupled GPU cluster connected via network interface pose a challenge. It takes multiple hops to transfer data in one GPU to another: first between the GPU device and host memory, then over distributed memory spaces onto a different node. The obligation to manage memory and data transfers exerts a burden to programmers, especially in a system where data flows are complicated. Recent work to mitigate this problem ([10], [11]) still falls short as it exposes programmers to the underlying communication topology.

(2) Performance objectives between GPU parallelization and stream specifications tend to conflict with one another. On one hand, the GPU architecture is optimized for throughput making it applicable for latency-tolerant applications. On the other hand, many stream systems consider response time as the key performance metric. A delicate trade-off is necessary to incorporate GPUs as accelerators for such stream systems to meet requirements of these metrics.

Our GStream framework provides language and run-time support as a first-order design objective to map streaming abstractions onto GPU clusters. It addresses the aforementioned challenges in two complementary ways:

(1) GStream provides a unified memory transfer interface in the context of streaming data flows. No matter where source and destination of streaming data reside, the run-time system automatically performs the necessary memory copies or initiates message passing to guarantee data coherence.

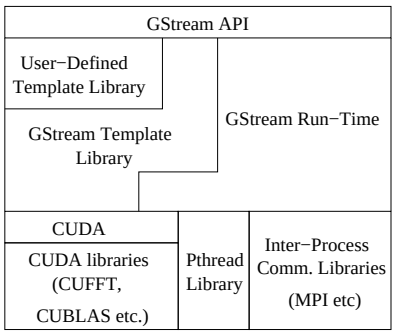


Figure 1. GStream Software Stack

As a result, users no longer need to write explicit MPI messaging or CUDA memory copy directives. This feature greatly reduces development time.

(2) GStream provides an elastic data API (stream push/pop) to dynamically adapt the batch size for each GPU kernel. This is based on the observation that many streaming steps allow re-sizable input size. By applying this technique at runtime we are able to handle streaming systems that have dynamically fluctuating data-flow characteristics without sacrificing response time requirements.

Our GStream framework can be easily integrated with third-party CUDA libraries. This is motivated by the trend to provide GPU kernels that can accelerate hot-spots even with complex dependencies and synchronizations. Past GPUs thrived on naive data parallelism without synchronization between fine-grained data operations. Compilers can exploit such embarrassingly parallel algorithms by detecting idempotent operations where output is a strict function of prior input flow without referencing any immediately preceding output. In modern GPUs, synchronization is natively provided at a fine-grained level. Consequently, numerous algorithms, including but not limit to linear algebra functions [12], FFT [13] and physical dynamics [14], can be significantly accelerated by CUDA abstractions and libraries. Nonetheless, today's most efficient CUDA implementations are still hand-written codes. Being able to reuse these capabilities is considered crucial both from the performance and productivity points of view.

Overall, GStream is a general-purpose, scalable run-time framework allowing application to be expressed as streaming problems and executing them efficiently on GPUs. GStream executes single program, multiple data (SPMD) codes on a cluster of accelerated machines. Here, GStream provides transparent streaming data transmissions and automatic memory synchronization while offering users full control to utilize computational resources of both CPUs and GPUs.

An overview of the GStream software stack is shown in Fig 1. GStream combines software abstractions with concrete implementations targeted at different levels of parallelism: CUDA and CUDA-derived libraries for data-parallelism; POSIX thread abstraction for task parallelism in shared-memory; and inter-processing communication libraries for data sharing across distributed-memory machines. The later two components are completely concealed by the GStream run-time system. They can be replaced by any other libraries that provide similar functionality without affecting the application code base. For instance, we utilize the message-passing functionality of MPI for inter-node communication. Similar implementations can be built on top of other inter-node communication libraries (e.g., TCP sockets). GStream's run-time system integrates library components and completely hides the thread management and data movement from the user.

The contributions of this paper are the following: (1) We propose a novel streaming abstraction dedicated for GPU clusters. (2) The streaming data-flow abstraction is made extremely concise, intuitive and can be supported by existing language abstractions (instead of inventing yet another language). It hides from user the complexity of memory transfers between different address spaces. (3) The validity of the abstraction reaches well beyond streaming illustrated via sample implementations for various domains, including data streaming, data parallel problems and numerical codes.

The rest of the paper is organized as follows. The system model and design goals are stated in Section II. In Section III, we describe the GStream API and its usage in detail. We present our system design in Section IV, experimental results in Section V, related work in Section VI and summarize the work in Section VII.

II. Design Goals and System Model

A. Design Goals

The focus of this work is to provide a general-purpose streaming framework dedicated to GPU architectures. We aim at satisfying several design goals:
(1) Scalability: The targeted platform is a cluster of machines accelerated by GPUs. There is no restriction on the size of the cluster.
(2) Transparency: Both the task scheduling and the GPU/host memory management for streaming data should be handled by the run-time system without any user intervention.
(3) Extendability: The library should be made extendable to meet customized needs while providing basic functionality.
(4) Programmability:The language syntax should be concise and type checking should be done at the compiler time.
(5) Flexibility: The computation cores can be chosen to freely execute on either CPU or GPU platforms. This allows fast prototyping with full debugging support on CPUs first.
(6) Re-usability: The cost of developing high-performance code on GPUs is higher than on general purpose microprocessors. Being able to reuse existing libraries will be a significant benefit.

B. System Model

Streaming systems are better understood when their internal data flow is characterized and analyzed. While efforts to specifically target certain architectures have led to different semantic abstractions of streaming, two fundamental components are common to typical streaming systems: data processing units and data links that connect them. In GStream, we refer to these as *filters* and *channels*, respectively.

Filters consume zero, one or multiple streams of data types and similarly produce any number of streams of identical or dissimilar data types. Filters without input or without output are referred as *source filters* or *sink filters*, respectively. In GStreams, source and sink filters are not differentiated from any other filters.

Channels exist whenever there is a data flow between filters, *e.g.*, one filter's input stream originates from another filter's output stream. From the filter's point of view, channels are effectively unbounded queues. Two types of channels are differentiated in GStream: point-to-point (p2p) channels and group channels. P2p channels are uni-directional and used for ad-hoc data transmissions. Each p2p channel has a predecessor filter and a successor filter. In contrast, group channels have well-defined group behavior and are used for inter-node data transmissions. GStream currently supports broadcast, reduce and all-to-all group channels. Both group and P2P of channels are strongly typed, connected to filters via *ports* and are associated with unique port IDs on filters. The operation of data on channels are realized via a simple push/pop interface (see Section III-B).

With the definitions of filters and channels, we can build customized streaming applications in a multi-node environment. For example, we can construct a standard pipelined system consisting of just P2P channels (Fig. 2(a)). Furthermore, backward channels are supported to realize feedback systems. Another dimension is given by arrays of filters (Fig. 2(b)), where each array element resides on a different node. The communication between filter arrays can be facilitated by group channels but P2P channels are supported as well.

The hybrid model of channels allows users to combine flexibility with productivity. On one hand, the P2P channel abstraction makes it possible to build any kind of stream graph. On the other hand, group channels prevent the programmer from having to build well-defined and widely-used communication patterns from scratch, which is a tedious and error-prone task.

III. GStream Overview

GStream is a C++ template library for data parallel, data streaming applications based on the streaming abstraction described in the previous section. Using C++ has two major advantages: (a) It seamlessly integrates with existing frameworks including CUDA and MPI; (b) The template metaprogramming feature in C++ provides an ideal technique to

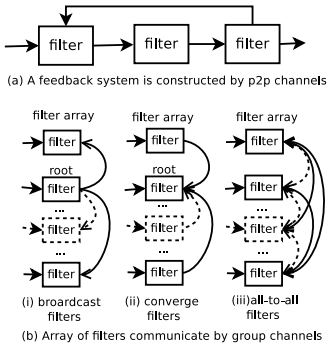


Figure 2. System Model

make the library reusable and expandable. In the following, we will (1) present the key characteristics of filters and channels in GStream; (2) show the principle GStream API and (3) illustrate the steps to write a program (as streaming specifications) in GStream.

A. GStream Abstraction and Convention

GStream makes several assumptions to abstract a streaming system. The basic computation unit is a filter. Filters can run independently from each other once their input data is available on an input channel. The main body of a filter is generalized into a three-step pattern (see Fig. 3). The start() and finish() functions are executed once at

```
void Filter::run() {
    start();
    while (!isDone())
        kernel();
    finish();
}
```

Figure 3. Filter Specification Pattern

the beginning and the end of the filter life cycle. They are used to execute chores such as parameter initialization and internal resource allocation/deallocation. They can also be used to allocate/deallocate scratch space, which is encapsulated in the filter itself. The central activity of a filter body is the kernel() function. Inside the kernel() function, a filter typically executes as follows: It waits for tuples from input ports, processes data and generates output tuples to output ports. A sequence of these steps is called a batch process. Batches continue to execute until input data is exhausted (or run forever if inputs are infinite streams). One of the differences between GStream and other streaming abstractions, such as StreamIt [15], is how the parallelism is defined as filters. In StreamIt, the user needs to define the behavior of a filter on the most fine-grained unit. In contrast, the *filter parallelism* in GStream is defined as a range of tuples a filter can process in one batch. This design caters to dynamic scenarios where the batch size can change at run-time. Such variance is controlled by the user through two APIs: getMinDegree(portId) and getMaxDegree(portId) define the legal range of the number of input tuples per port. The user is then required to provide a general routine

that can successfully handle input tuples in this range. On the output side, the number of tuples to be generated is determined by the size of the input tuples a filter receives in a batch. Making fine-grained filter behavior transparent and flexible through massive parallelism is precisely what distinguishes GStream from other streaming abstractions.

P2P channels and group channels are exposed to the user at different levels. P2P channels are explicitly constructed by the pipe operator | of the filter class. Group channels do not require any user intervention. They are associated with predefined special filter arrays. Their construction is handled internally and transparently by GStream. The advantages of using the pipe operator to express P2P channels are its conciseness and intuitive notation.

```
(1) f|g|h; // simple filter pipeline
(2) h|f;   // extend (1) with feedback
(3) for (i=0;i<M;i++) a[i]|b[i];//filter array
(4) for (i=0;i<M;i+=2) {
      a[i]|c[i/2];a[i+1]|c[i/2]; } // merge
```

In example (1), a pipeline of filters can be expressed in just one line of code by interlacing filters and pipes. Specification of a feedback path requires (2) just one extra line of code. Arrays of filters (3) can also be linearly combined or by flow splits or merges (4).

B. GStream APIs

The list of filters is maintained internally by the GStream run-time. Different filters in GStream are defined via concrete classes derived from the same base class (see Fig. 5) with basically three predefined virtual functions: void start(), finish() and kernel(). It is not necessary to override the start() and finish() functions if their bodies are empty. Similarly, getMinDegree(portId) and getMaxDegree(portId) have default values (1 and 4096, respectively) that can be overridden by the user to specify a different range.

Streaming data is owned and managed by channels through a simple channel interface. We found that a simple data push and pop API suffices to express data processing. Pop extracts streaming data from input channels. Conversely, push APIs injects data on output channels. To stream data out of an input channel, pop() is first called to obtain the buffer pointer. The call is blocked if the channel does not contain the number of tuples in the range defined by the minDegree and maxDegree on this port (unless end-of-stream is reached where the stream is flushed unconditionally). Upon returning, the run-time system supplies the current maximal number of tuples that satisfies the given range. As soon as the user has consumed the input, pop_finalize() can be invoked to inform the runtime that the channel can safely release the input. Similar two-phase operations apply to the output channel. This two-phase API requires a strict pairing of API calls by the user, which results in a number of benefits:

- Unnecessary memory copy operations are avoided. For instance, in the push API, reserve() is first called to obtain the current memory pointer of a channel. Once the data is ready, reserve_finalize() is called to signal the availability of the data. In contrast, if only reserve_finalized() were provided, the user would need to allocate memory explicitly.
- The size of reserved/popped data during the first step is not necessarily the same as the finalized size in the second step (but always greater or equal). This addresses the case when peek size differs from pop size.
- Irregular data types or types with unknown prior size (*e.g.*, IP packets) can be handled by casting channel types to unit type characters. When the actual tuple size is unknown during the first step, the consumption size can be adjusted in the second step.

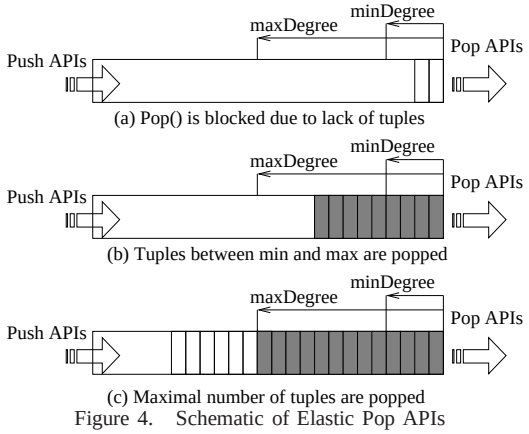
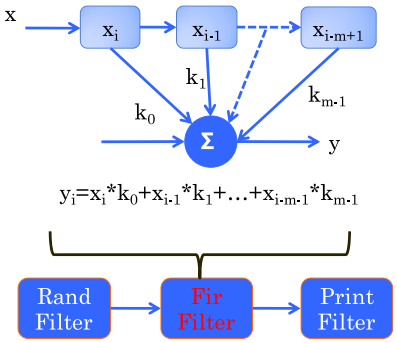


Figure 4. Schematic of Elastic Pop APIs

An illustration of the elastic pop API is given in Fig. 4. For each channel, the runtime system maintains a buffer viewed as an unbounded queue of tuples. When the downstream filter calls a pop() with an acceptable range, the runtime system checks the availability of data on the channel. If the number of tuples is less than the minimal range, the call is blocked (case (a)). If the number is greater than or equal to the minimal threshold, the call returns indicating the actual number of elements (capped by the maximal threshold). The system maintains the integrity of the popped data until the pop_finalize() is called. Only thereafter can the buffer be reused by the channel to store new pushed data.

C. Case Study – A Finite Impulse Response (FIR) Filter

An example of a simple FIR filter expressed in GStream is shown in Fig. 6. A FIR filter is a specific aggregate filter with a sliding window of order m (see Fig. 6(a)). We create a pipeline of three filters: a random number generator, the FIR filter and a print filter. In the main program (Fig. 6(b)), we demonstrate how the three filters are initialized and added to the stream system (lines 3 to 11). The P2P channel connection is expressed concisely as a single line (line 14). Both the random number generator and print filters are provided by the template library. The FIR filter definition is shown in Fig. 6(c).



(a) Fir Filter and GStream Structure

```
1  int main()
2  {
3    StreamSystem ss;
4    RandFilter<float> rf;
5    FirFilter<float, 100> firf;
6    PrintFilter<float> pf;
7
8    /* add filters to system */
9    ss.addFilter(&rf);
10   ss.addFilter(&firf);
11   ss.addFilter(&pf);
12
13   /* construct p2p channels */
14   rf | firf | pf;
15
16   /* ready to run */
17   ss.run();
18   return 0;
19 }
```

(b) Main Program

```
1  template<typename T, int m>
2  class FirFilter:public Filter<typelist1<T>, typelist1<T>>
3  {
4   public:
5    virtual void start() {
6      ... /* setup coefficient array k[m] */
7    }
8    virtual int getMinDegree(int) {
9      return m; // overwrite the default return value 1
10   }
11   virtual void kernel()
12   {
13     StreamChannelBuffer<T> input;
14     StreamChannelBuffer<T> output;
15     /* pop inputs of size from m to getMaxParallel(0) */
16     int batch = inputPort[0]→pop(&input, getMinDegree(0),
           getMaxDegree(0));
17     if (batch != −1){
18       /* reserve output buffer */
19       outputPort[0]→reserve(&output, batch − m + 1);
20       for (int i = 0; i != m; i++) {
21         ... /* the kernel calculation, omitted */
22       }
23       /* output data ready, finalize the reservation */
24       outputPort[0]→reserve_finalize();
25       /* only pop (batch − m + 1) from the input port */
26       inputPort[0]→pop_finalize(batch − m + 1);
27     } else { // returning −1 indicates the end of stream
28       setDone();
29   } }
30  private:
31    T k[m];
32 };
```

(c) Fir Filter Class Definition

Figure 6. Fir Filter Example

StreamSystem API:
void addFilter(FilterBase *filter);
void run();
Major Filter Functions:
void kernel()*;
void start()+; (empty by default)
void finish()+; (empty by default)
int getMinDegree(int portId)+; (return 1 by default)
int getMaxDegree(int portId)+; (return 4096 by default)
void assignToNode(int nodeId); /* for multi-node case */
void setToUseGpu(); /* set to use GPU or CPU */
*: must override. + has default behaviors
Channel Push API:
void reserve(StreamChannelBuffer &buffer, int size);
void reserve_finalize(int size);
Channel Pop API:
int pop(StreamChannelBuffer &buffer, int min, int max);
void pop_finalize(int size);
void waitForAny();

Figure 5. GStream API

Lines 5 to 7 override the start() function to set up the coefficient array. Method getMinDegree() needs to be overridden (lines 8 to 10) because it takes at least m input tuples to generate the first output tuple, where m is the degree of the FIR filter. Line 11 to 31 depict the execution of the main body of the FIR kernel function. It keeps popping data from its input port (lines 16 to 17). The returned size (int batch in line 16) always falls in the provided range of [getMinDegreel(0) ... getMaxDegree(0)]. The GStream runtime system guarantees continuous storage for the data in memory. Once the input size is known, we can use the information to reserve a buffer on the output channel (line 19). After the computation (lines 20 to 22) is completed, the output is pushed to the output port (line 26). To add GPU support, all we need to do is to replace the CPU code from lines 20 to 22 with a GPU kernel call. Any other code sections remain unchanged.

IV. Design and Implementation

We have implemented the GStream library using C++ programming language features with extensive use of template-based generic programming techniques [16]. GStream is deployed on a cluster of nodes, each equipped with a GPU.

With a template tool for manipulating collections of types (a typelist template of the Loki library [17]), we design the filter class to realize the filter abstraction in GStream. The base filter class (*Filter<inputTypeList, outputTypeList>*) is an abstract template class. It contains two templates as the

filter's input and output type lists. Filters are mapped to different threads and executed independently of each other.

The template design ensures the objectives of high programmability and extendability (see Section II-A). It provides enough flexibility for users to customize a filter's behavior. The derived filter is required to define its own kernel() function as a pure virtual function. The void start() and finish() functions can be optionally overridden if internal state needs to be initialized or resources need to be allocated/deallocated. The library currently contains several pre-defined filters such as a random number generator, a printing filter and a hash/map filter. Users may add new filters by deriving new classes from the base filter class.

The design of a kernel() function gives the user wide flexibility, but it usually adheres to the following pattern:

```
out_channel->pop();
in_channel->reserve();
kernel_calculation();
out_channel->pop_finalize();
in_channel->reserve_finalize();
```

The kernel_calculation() is the core computation that can be implemented by mapping the kernel to either a GPU (via CUDA) or a CPU. It can also be replaced by library calls, including numerical GPU libraries such as CULA, which meets the reusability objective of Section II-A.

Every port in a filter is associated with a data type, and the data type needs to be incorporated in the filter class' typelist template. This ensures strong type-checking at compile time. This limits filters in that they cannot have an arbitrarily large number of fan-in/out ports. We address this problem by (a) supporting (a) group channels (with only one port id, even if there are multiple data links); (b) creating intermediate filters in a tree structure; and (c) combining data types for multiple ports into one complex data type.

To meet our objective of flexibility (see Section II-A), a method setToUseGpu() in filter is provided to indicate that the computation routine should be accelerated by a GPU. By default, streaming data of such a filter resides within the GPU address space. This call acts as a hint to GStream to automatically perform necessary DMAs. Filters are assigned to a particular node by calling the assignToNode() member function. If two concatenated filters are assigned to different nodes, a pair of asynchronous MPI send/recv calls are setup to realize the channel pop/push interface. Each channel is associated with a data type, which matches one of the types in the filter class' input/output typelist according to the channel's port id in the filter. Internally, a channel has two buffers, one each for CPU and GPU. Depending on the receiver's filters property, the run-time system automatically synchronizes the memory.

Fig 7 depicts the overall system design for a GPU cluster. The resulting executable is an SPMD program. All filters assigned locally are instantiated by a CPU thread on a node. The GPU is time-shared among all filter threads: a global command FIFO queue is maintained for the GPU. All GPU-related operations issued by filters, including GPU memory allocation, DMA memory copy and kernel executions, are pushed to the queue. We thereby realize the transparency objective (see Section II-A) as scheduling, memory management and memory movement are automated.

A dedicated GPU thread serves the FIFO queue when the queue is non-empty. For nodes awaiting stream data from a different node, an upstream thread is created to listen to the network messages from other nodes. Once data is being received, the thread will push the data to the corresponding local filters. All MPI calls are asynchronous to avoid the deadlocks (e.g., due to blocking MPI send/receive orderings triggered by filter dependencies). GStream currently does not manipulate the execution order of the GPU FIFO queue. The mapping of filters to physical nodes is performed manually by the user through the filter::assignToNode(int nodeId) API. Since the underlying data transfer is made completely transparent to the user, users can experiment with different layouts via rapid prototyping to determine the best configuration. One of our future work is to automate the process by assigning nodes with high bandwidth streams to the same node.

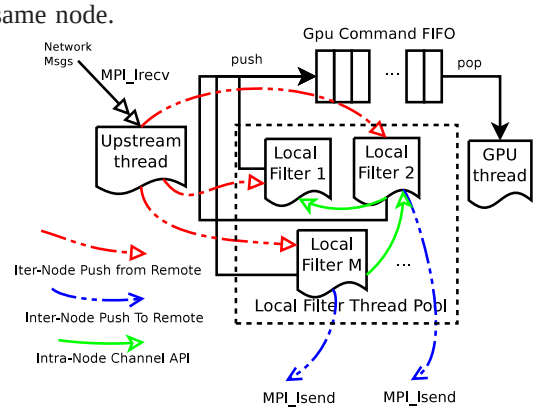


Figure 7. System Overview

V. Experimental Results

We performed experiments on a cluster where we utilized up to 32 nodes equipped with GPUs and connected by QDR Infiniband (36 Gbps). Each node consists of two AMD Opteron 6128 sockets (16 cores per node) and an NVIDIA Tesla C2050 graphic card. GStream is compiled with the CUDA 3.2 compiler combined with OpenMPI for MPI-style communication method.

A. Streaming Micro Benchmarks

We have implemented several representative streaming and non-streaming, iterative benchmarks using GStream, namely FIR filter, matrix multiply (MM) and FFT. These benchmarks require no more than a few filters, including a pre-defined random floating-point generator filter and a terminal output filter. For each benchmark, we provide four implementations: (a) A native C/C++ program running on CPUs without considering any streaming behavior (but still

uses third-party libraries); (b) a multi-threaded C/C++ program using the GStream library *without* GPU support; (c) a native CUDA implementation without considering streaming behavior and (d) GStream *with* GPU support.

The filter construction of the three benchmarks is shown in Fig. 8(a)(b)(c). GStream makes it straightforward to run filter arrays on multiple nodes to increase the throughput. We were able to run 32 copies of the filters on 32 nodes. The speedups of all implementations running on 32 nodes are shown in Fig. 9, with implementation (a) chosen as the baseline. The performance ratio of (b) over (a) indicates the overhead of the GStream run-time system, which is negligible as the ratios for all benchmarks are very close to one. In the following, we discuss each micro-benchmark in detail, including the detailed application parameters and third-party libraries we have used.

FIR has been introduced as a code example previously. We set the order of the FIR filter to 100, indicating an aggregate filter with a sliding window of size 100. A hand-coded FIR GPU kernel is developed in this benchmark. GStream using a GPU achieves a speedup of about a factor of 6 over the vanilla C version on a CPU.

For the matrix multiply (MM) benchmark, we measure the time to calculate a sequence of multiplies on square matrices of dimension 1024x1024. Both (c) and (d) integrates the CUBLAS library [12], an efficient implementation of BLAS for CUDA. To access the vanilla C version on a CPU, the Template Numerical Toolkit (TNT) is used [18].

Similarly for FFT, both the original C program and the CPU version of the GStream implementation use FFTW, a widely used and highly efficient FFT library. CUFFT [13], a FFT CUDA library shipped along with the CUDA SDK, is used in GPU evaluations. In this test case, the performance of a 2D (512x512) single-point complex FFT is compared.

Of these three benchmarks, the GPU version of GStream offers 3 to 27 times speedup over the corresponding C version. The CPU version of GStream outperforms the C program for FIR and FFT in spite of the synchronization overhead. This is because filters in GStream are executed in multiple threads. The random number generator filters in these two benchmarks execution is overlapped with FIR filter. This parallelism can compensate for the overhead of the library. The ratios of (b) over (a) and (d) over (c) show that GStream imposes little overhead to the overall system.

B. Scientific Benchmarks

We rewrote the IS (integer sort) benchmark of the NAS parallel benchmarks [19] and converted it into a filter-based program. The filter structure is depicted in Fig. 8(d). Input integer numbers are produced by the Random Number Generator. The Bucket Filter consumes these integers in large batches and calculates the bucket statistics of each batch. The Window Reduce Filter summarizes bucket information over batches until all inputs are processed. The final bucket statistics is fed to an Alltoallv Filter for post processing. Both Bucket and Window Reduce Filters can be mapped onto GPUs or CPUs. The GPU version is slightly faster than the original benchmark (see Fig. 9 for class D on $M = 32$ nodes). This is because IS is a communication-bounded benchmark, which limits GPU benefits.

GStream can be integrated with legacy codes by exposing APIs such as addFilter() and addChannel(). We have integrated GStream into LAMMPS, a molecular dynamics simulator distributed by Sandia National Laboratories [20]. LAMMPS is designed as a computing platform for simulating soft materials, solid-state materials and coarse-grained or mesoscopic systems. The original code runs on single processors or in parallel systems using MPI. More recently, accelerators, such as GPUs, have been deployed and are supported by LAMMPS as an effort to reduce the total computation time [14]. The simulation in LAMMPS is organized as a pipeline of computational steps making it a perfect candidate to apply our GStream concept.

In this case study, we replaced the LJ (Lennard-Jones) potential cutoff step with a customized filter in GStream and added channel manipulations in the LAMMPS source code to trigger its execution. The last set of bars in Fig. 9 shows the speedup of using the original GPU code and the GStream implementation vs. the CPU implementation on 32 nodes. Again, the overhead of adding the GStream library is negligible. We have only converted one hot-spot of the entire pipeline into GStream filters at this time. Complete transformation of all pipeline steps to GStream would result in a code base that is better organized and more expandable. In general, the ease of integration within legacy codes step-by-step for each kernel, such as demonstrated with LAMMPS, provides a graceful transition that facilitates the adoption of the GStream in other domains, such as complicated numerical codes.

C. Linear Road Benchmark

A widely-used real-time streaming benchmark is the Linear Road Benchmark [21], originally proposed to provide a scalable and fair benchmark for Stream Data Management Systems (SDMS). It simulates a toll system of motor vehicle expressways of a large metropolitan area. Expressways are divided into one-mile-long segments. The system needs to keep track of the number of vehicles, detect accidents in each segment and determine toll charges for each vehicle. In the meantime, queries such as vehicle balance, historical charges and travel time estimation need to be answered. In a three-hour simulation, the response time of each output is measured with regard to the following timing constraint: Outputs need to be produced within 30 seconds for travel time estimation queries and within 5 seconds for all other events, especially for toll notifications, which are on the critical path of the system. The performance metric of an

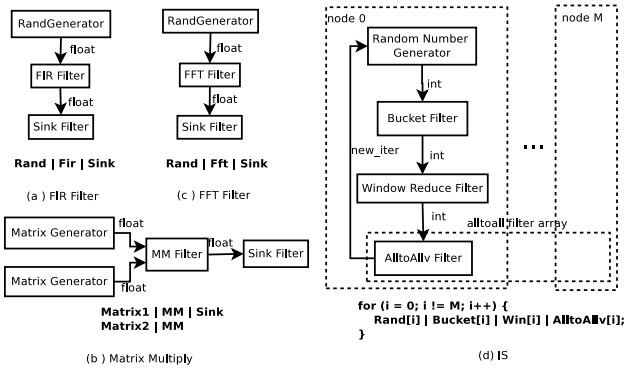


Figure 8. Filter Structure for Benchmarks

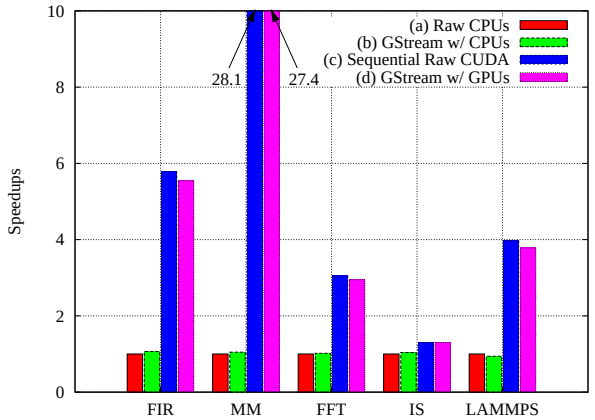
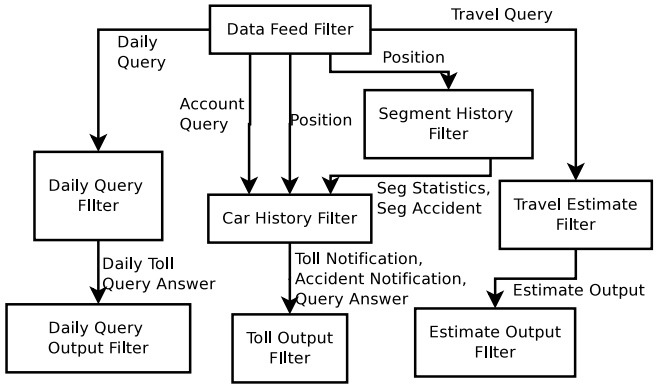
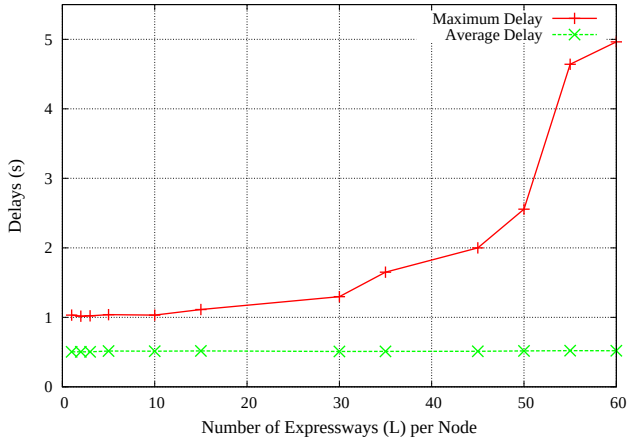


Figure 9. Speedup of Benchmarks on 32 CPU/GPU Nodes



(a) Linear Road Benchmark Filter Structure per Expressway



(b) Output Delays per Expressway

Figure 10. Linear Road Benchmark on 32 CPU/GPU Nodes

implementation is then given as the maximal number of expressways, L, without violating the timing constraint.

We have implemented the toll query system using the streaming abstraction of GStream, which allows us to focus specifically on customized filter design in the system. The filter graph per expressway is illustrated in Fig. 10(a). A data filter feeds the input tuples according to timestamps to mimic a real-world scenario. Inputs are filtered here to direct streams to different filters. Position reports are transferred to a segment history filter to generate segment statistics and detect potential accidents. The same position reports are also fed to a car history filter to determine if the car has entered a new segment. A channel connects the segment history filter with the car history filter. This channel is activated to transfer segment statistics (number of vehicles in the last minutes, accident flags) every minute to assist the car history filter in determining toll charges. Vehicle accounts are kept in the car history filter. Therefore, account queries pass through the car history filter, too. Other queries (daily/travel queries) are processed independently via a separate data flow.

Once the filters and their data dependencies are finalized, we can freely experiment with different filter mappings into our physical node space due to the transparency of data transmission provided by the GStream run-time system. The highest performance was obtained by assigning filters belonging to one expressway to the same physical node. "L-rating" defined as the maximal number of expressways supported by a system meeting response time constraints delimited by 5 seconds. The response time in one node at different number of expressways is shown in Fig. 10(b). In total, we get an L-rating of $60 \times 32 = 1920$ on 32 GPUs. Compared to previous work, both Aurora [22] (2003) and SPC [23] (2006) achieved L-ratings of 2.5 on a single machine. The most recent implementation in SCSQ [24] (2010) reported an L-rating of 64 on a dual quad-core.

D. 3D Stencil

GStream can improve the productivity to write programs that are not considered as traditional streaming applications. In this experiment, we implemented a 5-point 3D stencil Jacobi iteration on 32 GPU nodes to demonstrate that our scalability objective is being met (see Section II-A). Filters divide the stencil space along the Z axis. In each iteration, a filter needs to exchange its borders with two neighbor filters. We set each filter's stencil space to 512x512x512. Fig. 11 depicts that the wall-clock time remains constant under weak

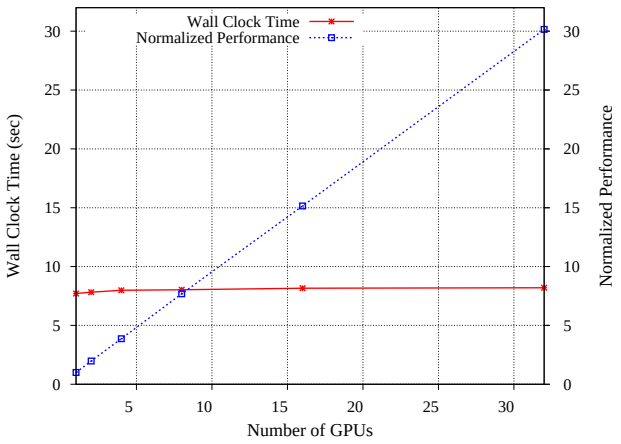


Figure 11. Weak Scaling in 3D Stencil on up to 32 GPUs

scaling (with proportional increase of both the number of nodes and the problem size). As a result, the performance normalized to a single GPU increases linearly. The reason GStream experiences perfect weak scaling is due to the fact that inter-node channels are implemented by asynchronous MPI calls that can overlap with the internal computations. Furthermore, programmers only need to focus on the development of stencil kernels on GPUs since communication is transparently handled by GStream.

VI. Related Work

Stream processing has been studies for a number of decades [5]. In the earlier years, the data flow semantic models and languages to support them were the primary focus. Several Data Stream Management Systems (DSMS), such as TelegraphCQ[25], Aurora [22], Medusa [26] and the STREAM project [27] [6], focused on continuous query processing, which is only one example of GStream's more general applicability and expressiveness.

Our concept of filters is loosely inspired by StreamIt [15], a platform-independent streaming language and compiler environment. Our runtime-centric dataflow approach is more general than their static analysis and transformation methodology. In fact, GStream could be used as part of the runtime system to extend StreamIt to GPU clusters. We further embrace a coarser-grained data parallelism than StreamIt, which results in performance beyond prior work [28]. A number of other filter-based frameworks have been designed [29], [30], [31], [32], [33]. Similarly, they encapsulate computations into filters, a central concept to express algorithms. But their designs are based on different objectives to fit a specific domain that they target. They also tend to target shared memory while we consider filters in a distributed memory environment across compute nodes in a cluster.

Brook [7] is a streaming language dedicated to GPUs. It does not support scheduling across kernels. It relies on a sequential language to trigger a streaming process. Udupa *et al.* [34] extended the ideas of StreamIt with a direct port to a single-node GPU platform. Filters are mapped to a sub-kernel level abstraction to realize transparent scheduling. GStream takes streaming to another level by combined support for *coarse-grained* data parallelism and filter arrays to target *multiple* GPUs. CUDA supports simple stream objects for command sequences that execute in order. While this concept matches simplistic pipelined computations, it fails to generalize to non-pipelined execution patterns and lacks support for expressing more complicated data dependencies that are widespread.

Recent years have witnessed many efforts to provide unified programming models or language support for accelerators including GPUs. StarSs [35] takes a pragma-based approach to express computational kernels as *tasks*. StarPU [36] uses *codelets* as an abstraction of a task that can be mapped to an accelerator. Both of them offer a certain degree of scalability but they are strictly constrained to the shared-memory paradigm. A new language called the X code is proposed in [9]. It contains a set of automated tools (AutoPipe) to aid in the design, evaluation and implementation of applications that can be executed on acyclic computational pipelines. It shares with GStream the philosophy that data flow should be expressed at a higher level to remove user interference. However, its scalability in larger clusters has not been shown, to the best of our knowledge.

VII. Conclusion

We have designed and implemented GStream, a general-purpose, scalable data streaming framework designed for clusters of GPUs. GStream is inspired by a lack of streaming abstraction dedicated to massively parallel architectures and their suitability to express data parallelism. We presented a novel and concise, yet powerful streaming abstraction amenable to GPUs. Communication patterns are expressed as point-to-point channels or as group channels. This abstraction ensures flexibility in runtime adaptation and fosters productivity during coding by letting programmers focus on the description of data organization and operations performed on the data without explicitly expressing task parallelism constraints. Programmability is realized through extensive use of template-based generic programming techniques in C++, which fosters portability and integration with an existing code base.

Overall, GStream's strength is in its ease of use and its applicability to a variety of domains not constrained to traditional streaming problems, as demonstrated by our experimental results. These aspects combined with efficient exploitation of GPU resources have the potential for a GStream-like paradigm to succeed.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.

[2] W. W. Wadge and E. A. Ashcroft, *LUCID, the dataflow programming language*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: a declarative language for real-time programming," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987, pp. 178–188.

[4] T. Gautier, P. Le Guernic, and L. Besnard, "Signal: A declarative language for synchronous programming of real-time systems," in *Conference on Functional Programming Languages and Computer architecture*, 1987, pp. 257–277.

[5] R. Stephens, "A survey of stream processing," *Acta Informatica 34*, pp. 491–541, 1995.

[6] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, resource management, and approximation ina data stream management system," Stanford InfoLab, Technical Report 2002-41, 2002.

[7] 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*, vol. 23, pp. 777–786, 2004.

[8] W. R. Mark, R. Steven, G. Kurt, A. Mark, and J. Kilgard, "Cg: A system for programming graphics hardware in a c-like language," *ACM Transactions on Graphics*, vol. 22, pp. 896–907, 2003.

[9] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. B. Shands, and N. Singla, "Auto-pipe: Streaming applications on architecturally diverse systems," *Computer*, vol. 43, pp. 42–49, 2010.

[10] O. S. Lawlor, "Message passing for gpgpu clusters: Cudampi." in *CLUSTER*. IEEE, 2009, pp. 1–8.

[11] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *IPDPS*, 2009, pp. 1–12.

[12] "CUDA CUBLAS library."

[13] "CUDA CUFFT library."

[14] G. Chen, G. Li, S. Pei, and B. Wu, "Gpgpu supported co-operative acceleration in molecular dynamics," *International Conference on Computer Supported Cooperative Work in Design*, vol. 0, pp. 113–118, 2009.

[15] B. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *In International Conference on Compiler Construction*, 2001, pp. 179–196.

[16] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[17] "http://loki-lib.sourceforge.net/."

[18] "http://math.nist.gov/tnt/."

[19] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "Nas parallel benchmark results," in *Supercomputing*, 1992, pp. 386–393.

[20] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995.

[21] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: A stream data management benchmark." in *VLDB*. Morgan Kaufmann, 2004, pp. 480–491.

[22] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[23] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the stream processing core," in *ACM SIGMOD International Conference on Management of Data*, 2006, pp. 431–442.

[24] E. Zeitler and T. Risch, "Scalable splitting of massive data streams," in *DASFAA, Proc. 15th Conf. on Database Systems for Advanced Application*, 2009.

[25] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing for an uncertain world." in *CIDR*, 2003.

[26] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. etintemel, Y. Xing, and S. Zdonik, "Scalable distributed stream processing," in *In CIDR*, 2003.

[27] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM: The stanford data stream management system," Stanford InfoLab, Technical Report 2004-20, 2004.

[28] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS*, 2006, pp. 151–162.

[29] A. A. Mustafa, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," 1998.

[30] S. M. Beynon, M. Beynon, R. Ferreira, T. Kurc, A. Sussman, J. Saltz, and J. H. Medical, "Datacutter: Middleware for filtering very large scientific datasets on archival storage," in *Symposium on Mass Storage Systems*, 2000, pp. 119–133.

[31] M. B. Tahsin, M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz, "Design of a framework for data-intensive wide-area applications," in *Heterogeneous Computing Workshop*, 2000, pp. 116–130.

[32] G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira, "Coordinating the use of gpu and cpu for improving performance of compute intensive applications," *IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, 2009.

[33] J. Zhou and B. Demsky, "Bamboo: a data-centric, object-oriented approach to many-core software," in *ACM SIGPLAN conference on Programming language design and implementation*, 2010, pp. 388–399.

[34] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on gpus," in *Symposium on Code Generation and Optimization*, 2009, pp. 200–209.

[35] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple gpus," in *Euro-Par Conference on Parallel Processing*, 2009, pp. 851–862.

[36] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par Conference on Parallel Processing*, 2009, pp. 863–874.

[37] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," Stanford InfoLab, Technical Report 2002-19, 2002.

[38] C. Consel, H. Hamdi, L. Reveillere, L. Singaravelu, H. Yu, and C. Pu, "Spidle: A DSL approach to specifying streaming applications," in *Int. Conf. on Generative Prog. and Component Engineering*, 2002, pp. 1–17.

[39] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer, "Auto-pipe and the x language: A pipeline design tool and description language," in *Intl Parallel and Distributed Processing Symp.*, 2006.