2011 IEEE International Parallel & Distributed Processing Symposium

Data Parallel Programming Model for Many-Core Architectures

Yongpeng Zhang
North Carolina State University
yzhang25@ncsu.edu

***Abstract*—Emerging accelerating architectures, such as GPUs, have proved successful in providing significant performance gains to various application domains. This is done by exploiting data parallelism in existing algorithms. However, programming in a data-parallel fashion imposes extra burdens to programmers, who are used to writing sequential programs. New programming models and frameworks are needed to reach a balance between programmability, portability and performance. We start from stream processing domain and 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, numerical codes and text search. This work lays a foundation to our future work to develop more general data parallel programming models for many-core architectures.**

I. Introduction

The amount of data that needs to be processed continues to grow in the era of computing with no foreseeing end. In contrast, processor frequencies have reached their limits due to power constraints. Traditional instruction level parallelism can no longer provide worthy performance benefits. Instead, higher degree of parallelism has to be extracted, both in the algorithmic or implementation level, to fully utilize emerging multi and many-core architectures.

The exploitation of data parallelism has proved highly important in emerging accelerating architectures, such as GPUs, in various application domains. In fact, the high computational throughput of GPUs originates from the inherent 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 effective latency hiding mechanisms to be realized that mitigate the effects of the memory wall [21]. Such GPU hardware unfolds its full potential over general-purpose CPUs when data parallelism can be exploited [13].

However, this great performance benefit comes with the penalty of sacrificing programmability. Despite the claim that CUDA only adds a few extensions to C programming language, writing highly efficient CUDA programs requires deep understanding of the underlying architecture. It often takes arduous work to optimize the code to achieve noticeable speedups over the correspondent CPU code. Moreover, it is often the case that some optimization strategies may only work for certain generation of architectures. As hardware evolves, the same optimization procedure needs to go through again to find the best configuration, which becomes tedious and error-prone.

Therefore, it is desirable to develop new programming models to reach an acceptable balance between programmability, portability and performance. Those models need to accomodate the increasing number of cores per chip in current industrial trend. Even though it is ideal to reuse existing algorithms designed for sequential programs and rely on smart compilers to recognize more parallelism, the success of this approach has only be found in limited domains, where prior knowledge of the application characteristics is assumed. We believe that the legacy code must be revised manually in such a way that enough knowledge of parallelism can be deduced by tools.

II. GStream

We start from looking at possibilities in a specific area, namely the stream processing domain. 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 domains, such as data analysis and computationally intensive tasks are also discovering the benefits of the underlying streaming paradigm. We have already developed GStream, a general-purpose, scalable data streaming framework for GPUs. The contribution of this work are the following:

- Our streaming abstraction expresses data parallelism more naturally than task-oriented parallelism.
- The abstraction is extremely concise and intuitive. It is supported by template language abstractions of C++, which helps adoption of our approach (in contrast to inventing yet another language).

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.378

- The data-parallel approach reduces data dependencies and increases the potential for efficiently providing massive parallelism, particularly when mapped onto GPUs.
- This is the first work to exploit streaming applications on *clusters* of GPUs, to the best of our knowledge.
- The validity of the abstraction reaches well beyond streaming as illustrated by sample implementations for various domains, including data streaming, data parallel problems, numerical codes and text search.

Two key concepts in GStream are *filters* and *channels*. Filters encapsulate computing kernels that can be accelerated by GPUs. The main body of a filter is generalized into a three-step pattern in the following:

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

All of the start(), finish() and kernel() functions can be overloaded by the programmer with user-defined behaviors. It is inside the kernel() function that the user can choose to run GPU-accelerated CUDA kernels.

The data-parallelism inherently to filter kernels is facilitated by simple APIs to manipulate data in channels, which represent data links between any two filters:

Channel Push APIs:
void reserve(StreamChannelBuffer &buffer, int size);
void reserve_finalize(int size);
Channel Pop APIs:
int pop(StreamChannelBuffer &buffer, int min, int max);
void pop_finalize(int size);
void waitForAny();

GStream's abstraction of filters and the run-time system are specifically targeted at GPUs. The filter parallelism usually corresponds to the number of threads in GPU kernels and can be variable during run-time. The user can determine the filter's parallelism by the number of input tuples currently available on input channels. This dynamic greedy behavior provides a best-response-time-effort mechanism to achieve balance between GPU efficiency and response time. While GPU kernels still need to be hand-coded, they are confined to kernel() functions of the filter class. This enforces strong isolation of computation from underlying data movement, both for inter-node or intra-node data movement.

Applications are built by concatenating filters with channels and overloading kernel functions specified as a user-defined filter class. GStream provides powerful, yet concise language abstractions suitable to describe not only streaming problems but also conventional algorithms in a data-centric and data-parallel manner.

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

The software stack is shown in Figure 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. None of our concrete components supporting these abstractions are mandatory as long as substitute libraries provide the same functionality. For instance, we utilize the message-passing functionality of MPI for inter-node communication. The use of MPI is not required by GStream. It is an optional acceleration feature. Similar implementations can be built on top of other inter-node communication libraries. GStream's run-time system integrates library components and completely hides the thread management and data movement from the user.

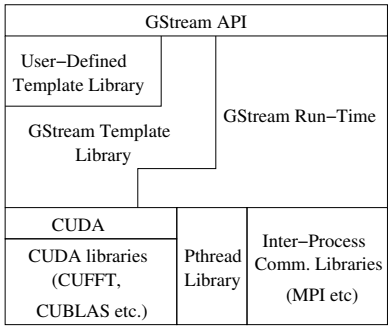

Figure 1. GStream Software Stack

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 variety of domains, including but not limited to data streaming, data parallel problems, numerical codes and text search. Some of the (non-)streaming benchmark results are shown in Fig. 2. For each benchmark, we provide four implementations: (a) A native single-threaded C/C++ program running on CPUs without considering any streaming behavior; (b) a multi-threaded C/C++ program using the GStream library *without* GPU support; (c) a multi-threaded program using GStream *with* GPU support and (d) a native CUDA implementation without considering streaming behavior. The five benchmarks are FIR (finite impulse filter), MM (matrix multiply), FFT (fast fourier transform), IS (integer sort in NAS benchmark) and

LAMMPS (a molecular dynamics simulator distributed by Sandia National Laboratories [14]).

GStream makes it straightforward to run filter arrays on multiple nodes to increase the throughput. We were able to run 16 copies of the filters on 16 nodes.

For the first three benchmarks, the GPU version of GStream offers 3 to 30 times speedup over the corresponding C version. 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. This parallelism can compensate for the overhead from the library. The ratios of (b) over (a) and (c) over (d) show that GStream imposes little overhead to the overall system.

We rewrote the IS (integer sort) benchmark of the NAS parallel benchmarks and converted it into a filter-based program. The GPU version is slightly faster than the original benchmark. This is because IS is a communication-dominant benchmark, which limits GPU benefits.

We have integrated GStream into LAMMPS. 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 Figure 2 shows the speedups of using the original GPU code and the GStream implementation vs. the CPU implementation on 16 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.

We provide a reference implementation for Linear Road Benchmark, a widely used real-time streaming benchmark for Data Stream Management Systems (DSMS), using GStream (Fig. 3). Our system can simulate up to 40 expressways per GPU within timing constraints in contrast to 2.5 expressways per CPU reported in previous literature ([1] [10]), which is a 16X improvement.

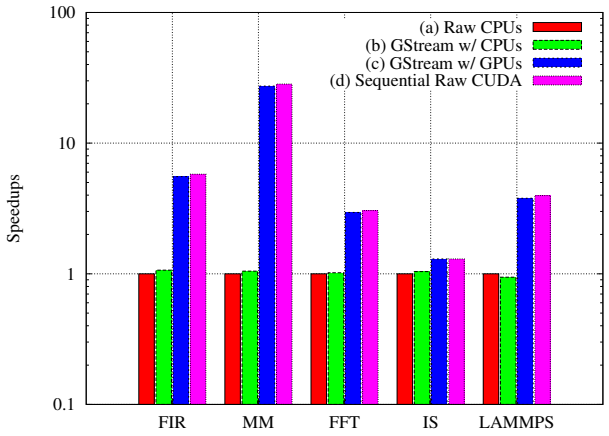

Figure 2. Benchmark Speedups

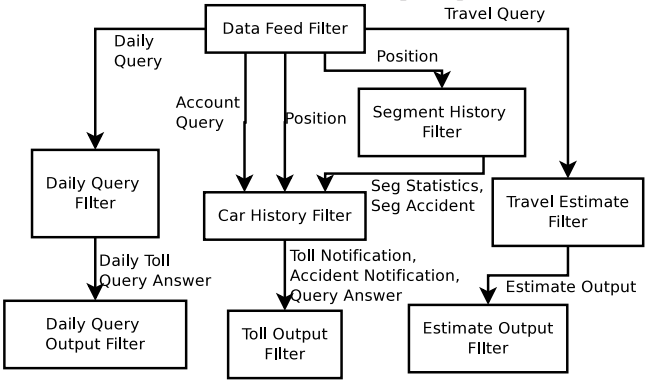

Figure 3. Linear Road Benchmark

III. Related Work

Stream processing has been studies for a number of decades [15]. 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[6], Aurora [1], Medusa [7] and the STREAM project [2] [11], 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 [19], a platform-independent streaming language and compiler environment. Their static analysis and transformation methodology is orthogonal to our runtime-centric approach. We further embrace a coarser-grained data parallelism that StreamIt, which results in performance beyond prior work [8]. A number of other filter-based frameworks have been designed [12], [3], [16], [18], [22]. 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 [5] 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.* [20] 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 se-

quences 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.

IV. Future Work

Our next step is to apply the lessons we learned so far to design a programming model suitable for general purpose applications. We believe that implicitly parallel programming model [9] will be a viable approach. Data-parallel languages seem to be good language candidates since they force programmers to design data-parallel algorithms from the very beginning. But current languages ([4]) lack enough information for merging consecutive data-parallel primitives. which is shown to be crucial for their efficient implementations on many-core architectures [17]. We also propose to add tuning parameters in algorithm expression for run-time systems to search the best configurations for various number of many-core architectures.

References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] 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. Technical Report 2004-20, Stanford InfoLab, 2004.

[3] Systems Michael Beynon and Michael Beynon etc. Datacutter: Middleware for filtering very large scientific datasets on archival storage. In *In IEEE Symposium on Mass Storage Systems*, pages 119–133. IEEE Computer Society Press, 2000.

[4] Guy Blelloch and Parallel Ram Model. Nesl: A nested data-parallel language. Technical report, 1990.

[5] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.

[6] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[7] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uur etintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[8] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[9] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, tephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 754–759, New York, NY, USA, 2007. ACM.

[10] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2006. ACM.

[11] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation ina data stream management system. Technical Report 2002-41, Stanford InfoLab, 2002.

[12] Anurag Acharya Mustafa, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation, 1998.

[13] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[14] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19, 1995.

[15] R. Stephens. A survey of stream processing, 1995.

[16] Michael Beynon Tahsin, Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Design of a framework for data-intensive wide-area applications. In *In Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000*, pages 116–130. IEEE Computer Society Press, 2000.

[17] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

[18] G. Teodoro, R. Sachetto, O. Sertel, M.N. 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*, pages 0–10, 2009.

[19] Bill Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *In International Conference on Compiler Construction*, pages 179–196, 2001.

[20] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[22] Jin Zhou and Brian Demsky. Bamboo: a data-centric, object-oriented approach to many-core software. *SIGPLAN Not.*, 45:388–399, June 2010.