Alea Reactive Dataflow: GPU Parallelization Made Simple

Luc Bläser
University of Applied Sciences
Rapperswil
Institute for Software
lblaeser@hsr.ch

Daniel Egloff
QuantAlea Inc. Zurich
daniel.egloff@quantalea.net

Oskar Knobel
University of Applied Sciences
Rapperswil
Institute for Software
oknobel@hsr.ch

Philipp Kramer
University of Applied Sciences
Rapperswil
Institute for Software
pkramer@hsr.ch

Xiang Zhang
QuantAlea Inc. Zurich
xiang.zhang@quantalea.ch

Daniel Fabian
InCube Group Inc. Zurich
daniel.fabian@incubegroup.com

Abstract

Making effective use of the GPU parallel power requires relatively complex and tedious work: Understandably, most programmers spare the efforts. The Alea reactive dataflow programming model now aims to substantially lower this threshold by simplifying GPU parallelization quite radically. Programs are described as data that is asynchronously propagated through a graph of operations, each typically predestined for vector parallelization. Programmers do no longer need to write GPU-specific code but instead leave the GPU-parallelization to the runtime system. Due to the declarative and reactive paradigm, operations can be easily scheduled as parallel streams on a GPU with minimum memory copying overheads.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages

Keywords GPU; parallelization; reactive; dataflow

1. Introduction

For many programmers, the threshold for engaging GPU parallelization is too high. In order to make adequate use of the many cores of a GPU, several obstacles need to be taken: (1) Algorithms need to be tailored for vector parallelization since the cores are de facto per-element views of vector-parallel instructions. (2) The parallel implementation is based on a rather low-level machine-centric programming models such as CUDA [1], OpenCL [2], or other alternatives [3, 4]. (3) Integrating the typical C technology stack into a managed environment, such as .NET, necessitates extra workarounds. Therefore, GPU parallelization is unfortunately often perceived as too difficult, too costly and offering only a marginal benefit.

Several cross-platform frameworks support GPU parallel programming in managed runtimes, such as in Java [6, 7] or .NET [5, 8, 9]. The programming abstractions, however, essentially remain at the same low level of CUDA or OpenCL. More elegant integrations have been proposed but they usually lack generality, e.g. a LINQ-integration [10, 11] only supports a limited set of query operations. Dataflow programming models for GPU are more general: Xcelerit [12], PTask [13], and FastFlow [14] follow this approach but have the drawback that programmers typically have to implement custom nodes since the model has no or only fixed predefined operations. This is where more low-level and tedious programming is again involved. We also believe that substantial benefits can be gained if dataflow would become more reactive, i.e. fully asynchronous and ready to process sequences of inputs. For a more detailed analysis, see the discussion of related work in Section 4.

Our goal is to radically simplify GPU parallelization while still retaining expressiveness and efficiency. For this reason, we have developed *Alea reactive dataflow*, a programming model based on .NET. A computation is described as data propagated through a directed graph of operations. The propagation is asynchronous, reactive and push-based, while operations are typically vector-parallelizable and generic. Programmers can easily define computations without writing GPU code.

The runtime system takes care of the efficient parallelization on GPUs, by streaming operations, configuring launches and minimizing copying between CPU and GPU memory. The runtime system as well as the implementation of operations guarantee memory safety. Although we currently focus on GPUs, the model could be equally applied to general heterogeneous distributed parallelization.

The remainder of this paper is structured as follows. Section 2 introduces the programming model. Section 3 briefly outlines the current runtime system. Section 4 discusses related works. Section 5 finally draws a conclusion.

2. Programming Model

Alea reactive dataflow programs are defined by connecting operations to form a directed graph. Computations are triggered by feeding input to operations. This implies a chain of reactions: Operations execute asynchronously whenever sufficient input is present and thereby produce output passed to subsequent operations. To obtain results, output can be observed from any operation. In the

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

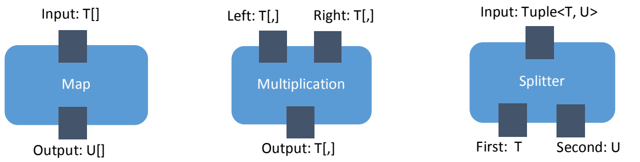


Figure 1. Three operations with input and output ports.

following subsections, we explain the elementary concepts of the programming model, accompanied by two exemplary application cases.

2.1 Operations

An *operation* represents a self-contained unit of calculation that has a set of input ports and a set of output ports. A *port* denotes a stream of data of a defined type. The stream can be infinite with data arriving in arbitrary intervals. When data is present at a defined set or subset of the input ports, the operation consumes this data as input, performs a calculation to produce data as output for a set or subset output ports. Input is processed in the order as it arrives, triggering output in the corresponding order, i.e. later input cannot result in earlier output. However, data can arrive at each port at different time intervals; they are not mutually synchronized.

Figure 1 depicts operations, with input ports at the top border and output ports at the bottom border. Each port is specified with a name and the type of the data. An operation is an instance of a particular class, implementing the operation. The operation determines which input ports are required to trigger a calculation, e.g. `Multiplication` requires data at both input ports to trigger. Analogously, the operation also defines to which output ports data is passed, e.g. `Splitter` yields data at both output ports for each input.

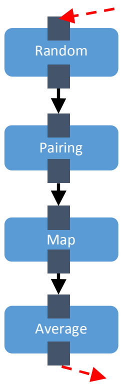
Operations can feature multiple implementations for different processor architectures, such as GPUs or CPUs, see Section 3. To be suited for GPUs, operations typically implement a massively vector-parallel (SIMD) calculation per input, e.g. `Map` transforms an array of elements. Many operations are generic, i.e. only provide partial implementation skeleton to be completed by a delegate/lambda at construction time, e.g. the element-wise map function delegate of the `Map` operation. This enables relatively high expressiveness despite a fixed set of prefabricated operation classes. Internally, operations can be stateless or stateful, i.e. work with or without a state that is stored between executions.

2.2 Graphs

Operations can be interconnected to form a *graph*. The output port of a preceding operation can be connected to one or multiple input ports of a succeeding operation, provided that the ports have the same type. Whenever data is passed to an output port, the data becomes available at all connected input ports. Multiple output ports may be also connect to the same input port, if the types match, using an arbitrary order to merge the data of multiple output ports into a common input port.

Figure 2 outlines a graph for a Monte Carlo Pi approximation. Figure 3 shows a graph for the iterative computation of the steady state in a Markov chain, based on the iterative formula $b_{i+1} = Ab_i$ until $b_{i+1} = b_i$. `Splitter` and `Merger` are used to synchronize A and b input, in the case of concurrent processing of multiple Markov chain inputs.

Passing is asynchronous, i.e. an operation can produce data to an output port, without awaiting the consumption of the data by any other connected operations. Operations adhere to the principle that passed data is immutable. Data can thus be passed by copying or by referencing. If an arbitrary merge order is inappropriate, dedicated operations may be used to join multiple data streams. Graphs can have cyclic connections, such as for iterative or continuous computations (e.g. Figure 3). Ports can also have no connections: they may be unused or serve for external sending or reception, as explained in the next subsection.



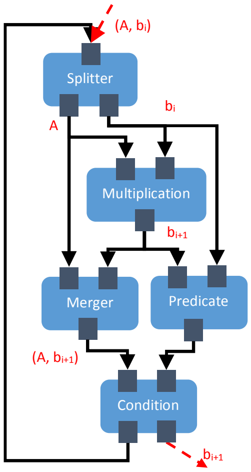
```
var random = new Random<float>();
var pairing = new Pairing<float>();
var map = new Map(p =>
   p.X * p.X + p.Y * p.Y <= 1
   ? 1 : 0);
var average = new Average<int>();

random.Output.ConnectTo(pairing.Input);
pairing.Output.ConnectTo(map.Input);
map.Output.ConnectTo(average.Input);

average.Output.OnReceive(x =>
           Console.WriteLine(x * 4));

random.Input.Send(1000);
random.Input.Send(1000000);
```

Figure 2. Monte Carlo Pi approximation dataflow graph.



```
var source = new Splitter
                     <Matrix, Vector>();
var mult = new Multiplication
                     <Matrix, Vector>();
var cmp = new Predicate<Vector>
        ((a, b) => Abs(a - b) > 1E-6);
var next = new Merger<Matrix, Vector>();
var cond = new Condition<Vector>();

source.First.ConnectTo(mult.First);
source.Second.ConnectTo(mult.Second);
source.First.ConnectTo(next.First);
mult.Output.ConnectTo(next.Second);
mult.Output.ConnectTo(cmp.First);
source.Second.ConnectTo(cmp.Second);
next.Output.ConnectTo(cond.PassThrough);
cmp.Output.ConnectTo(cond.Criterion);
cond.True.ConnectTo(source.Input);

cond.False.OnReceive(Console.WriteLine);
source.Send(new Tuple(A, b0));
```

Figure 3. Markov chain steady state as dataflow graph.

2.3 Dataflow

A *dataflow* is the propagation of data through the graph. Data can be sent to any input port. Sending is asynchronous, i.e. does not block. Multiple data can also be sent at the same time to the same input port, in which case no order is postulated for the data. Conversely, data can also be received from any output port by registering delegates that are asynchronously invoked whenever output data is produced at that port. Figure 2 and 3 also demonstrate how data is sent to input ports and received from output ports (highlighted in red font).

The reception delegates are executed by arbitrary threads, i.e. multiple output data can be processed concurrently. If multiple delegates are registered for an output port, all are invoked in arbitrary order or possibly concurrently. Data streams require no explicit termination but represent a conceptually infinite sequence of data.

2.4 Short Notation

A shortcut fluent-style notation can be used for the graph and dataflow definition, see Figure 4. The selection of input and output

```
var random = new Random<float>();
random
  .Pairing()
  .Map(p => p.X * p.X + p.Y * p.Y <= 1 ? 1 : 0)
  .Average()
  .OnReceive(a => Console.WriteLine(a * 4));
random.Send(1000);
random.Send(1000000);
```

Figure 4. Short notation for the Monte Carlo Pi example.

port is thereby implicit if the operation has a single input or output port, respectively.

3. Runtime System

The dataflow runtime support is realized by two components: a scheduler and the internal implementations of operations.

Operations implement a function for determining when sufficient input is available to trigger the calculation. Moreover, an operation provides one or multiple mappings to defined processor architectures, such as CPU and GPU. The GPU mapping resembles the standard CUDA model [1], however type-safely integrated into .NET. As for generic operations, the concrete delegate .NET IL code is gathered and translated to CUDA code at runtime or at compile-time, and eventually fused into the operation's CUDA kernel. The GPU mapping of an operation additionally defines a script of specific malloc/launch commands as an execution plan to happen in the future. At the planning time, the script has only restricted information about the data to be processed, i.e. only knows scalar values and the sizes of input blocks to make decisions for optimal kernel launch configurations. The CPU mapping can be directly executed by the .NET TPL [15]. In contrast to GPU mappings, a CPU implementation can be stateful, i.e. carry state over calculation by defining their own CPU-side synchronization on that state.

Programmers can implement custom operations, by providing the mapping for CPU and/or CUDA. This certainly requires more expert knowledge in GPU parallelization. However, we aim to provide a good base functionality supplying a well-selected set of generic operations, such that users usually do not need to implement custom operations.

The scheduling is currently realized for hybrid CPU and single GPU execution. For an input, the scheduler collects the largest non-cyclic sub-graph of GPU-implemented operations to start these operation in one stream. Memory copying is only necessary and performed for transitions between CPU and GPU operations or when the host program sends data to or receives data from GPU operations. As transmitted data must be immutable, it can be shared or copied. Deallocation of GPU memory is automatically managed by the scheduler and not within the operation implementations. The scheduler disposes GPU memory blocks when no longer used by a running operation or contained in a data stream.

The dataflow system uses Alea cuBase [8] as the underlying engine for the CUDA runtime and compilation support within .NET.

4. Related Work

Our model is strongly inspired by Rx.NET [16, 17] and TPL dataflow [18]. These models are however not designed for GPUs, as the blocks are generally unsuited for vector parallelization. A further significant difference is that we support multiple input and output ports. This permits the design of arbitrary well-controlled mergers or splitters. In the TPL dataflow for example, splitting and merging can only be controlled to a limited degree, by filtering messages or using batch/join blocks with specific merge pattern. We also abandon the concept of explicit termination of a stream.

Several frameworks improve cross-platform GPU parallelization, e.g. for Java [6, 7] or .NET [5, 8, 9]. However, the majority essentially exposes the same low-level programming model. Programmers are still bothered by technical artefacts, such as writing SIMD-kernels, copying between CPU and GPU memory, wrapping code in special classes, dealing with launch configurations, thread block ids etc. Notable simplification are achieved by more abstract models, such as translating .NET LINQ expressions to GPU parallel code [10, 11]. However, the expressiveness of this approach is inherently limited by the fixed set of LINQ query functions, basically being projection, mapping, filtering, ordering, and grouping.

Dataflow models allow the composition of parallel operations by minimizing memory transfers. Xcelerit [12], PTask [13], and FastFlow [14] are all based on this paradigm, to enable heterogeneous parallel computing in particular also for GPUs. These system still do not go as far as desired: A created graph essentially serves a single computation and/or synchronous invocation from the host side limits concurrency. In combination with a reactive concept, their practicability could be raised, i.e. by allowing the same graph to asynchronously process a conceptually infinite sequence of inputs, sent in arbitrary intervals. In contrast to the aforementioned systems, we also support generic operations, i.e. operations that implement a partial algorithm skeleton and are completed by a user-specific delegate/lambda/functor upon creation. This naturally requires cross-compilation of host code to the GPU platform.

5. Conclusions

The Alea reactive dataflow programming model enables simple but powerful GPU parallelization in .NET. Due to the descriptive paradigm, programmers are liberated from writing explicit low-level GPU code. This promotes fast and condensed program formulation, while the scheduler enables efficient and memory-safe execution behind the scenes. The reactive push-based paradigm makes the model particularly general, i.e. supports cycles, infinite stream of input delivered in arbitrary intervals. Naturally, the usefulness of the model stands and falls with the set of operations that is available. Generic operations provide a substantial step in this direction, such that programmers usually do not need to implement custom operations. Our work is still in progress: In the future, we plan to enhance the scheduler for the support of multiple GPUs and cluster distribution, as well as for further optimizations. Moreover, we aim to continuously extend the generic operation catalogue.

Addendum

This research is funded by the Swiss National Commission of Technology and Innovation (CTI), project number 16130.2. All trademarks, trade names etc. are the property of their respective owners.

References

[1] Nvidia Inc. *CUDA C Programming Guide*. Version 6.0, `http://docs.nvidia.com/cuda/cuda-c-programming-guide`, accessed 2014-08-25.

[2] Khronos Group. *The Open Standard for Parallel Programming of Heterogeneous Systems*. OpenCL 2.0, `https://www.khronos.org/opencl`, accessed 2014-08-25.

[3] Microsoft Inc. *C++ Accelerated Massive Parallelism (C++ AMP)*. `http://msdn.microsoft.com/en-us/library/hh265136.aspx`, accessed 2014-08-25.

[4] OpenACC. *The OpenACC Application Programming Interface*. Version 1.0, 2011, `http://www.openacc.org`, accessed 2014-08-25.

[5] *Cudafy.NET*. `http://cudafy.codeplex.com`, accessed 2014-08-25.

[6] P. C. Pratt-Szeliga, J. W. Fawcett, and Roy D. Welch. *Rootbeer: Seamlessly using GPUs from Java*. IEEE 9th International Conference

on High Performance Computing and Communication 2012 & IEEE 14th International Conference on on Embedded Software and Systems (HPCC-ICESS), 2012. IEEE, pp. 375-380, 2012.

[7] Y. Yonghong, M. Grossman, and V. Sarkar. *JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA*. Euro-Par 2009 Parallel Processing. Springer, 887-899, 2009.

[8] QuanAlea Inc. *Alea cuBase*. `https://www.quantalea.net`, accessed 2014-08-25.

[9] G. Cocco. *FSCL Compiler*. `http://fscl.github.io/FSCL.Compiler`, accessed 2014-08-25.

[10] Nessos, *GPU LINQ*. `https://github.com/nessos/GpuLinq`, accessed 2014-08-25.

[11] C. J. Rossbach, Y. Yu, J. Currey, J. P. Martin, and D. Fetterly. *Dandelion: A Compiler and Runtime for Heterogeneous Systems*. In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13). Nov. 2013.

[12] J. Lotze, P. D. Sutton, and H. Lahlou. *Many-Core Accelerated LIBOR Swaption Portfolio Pricing*. In Companion IEEE High Performance Computing, Networking, Storage and Analysis (SCC), 2012.

[13] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. *PTask: Operating System Abstractions to Manage GPUs as Compute Devices*. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11), Oct. 2011.

[14] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. *Targeting Distributed Systems in FastFlow*. In Euro-Par 2012: Parallel Processing Workshops (pp. 47-56). Springer, Jan. 2013.

[15] D. Leijen, W. Schulte, and S. Burckhardt. *The Design of a Task Parallel Library*. In Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'09), Oct. 2009.

[16] Microsoft Inc. *The Reactive Extensions (Rx.NET)*, `http://msdn.microsoft.com/en-us/data/gg577609.aspx`, accessed 2014-08-25.

[17] E. Meijer. *Your Mouse is a Database*. ACM Queue 10(3):20-34, March 2012.

[18] S. Toub. *Introduction to TPL Dataflow*. Microsoft Inc, Apr. 2011.