The Alea Reactive Dataflow System for GPU Parallelization *

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

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

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

Abstract

The Alea reactive dataflow system represents a general, efficient, and memory-safe model for homogeneous programming of heterogeneous platforms. Programmers can describe computations as asynchronous dataflow graphs built from generic prefabricated or custom operations. The system is based on the .NET runtime system and allows to seamlessly target both CPU and GPU executing operations on either platform including multi-GPU scheduling. Language embedded GPU kernels are cross-compiled from .NET IL to GPU code. The dataflow runtime system takes care of efficient lock-free data management including garbage collection and performs just-in-time optimization of the dataflow graph.

1. Introduction

GPUs are designed for massive parallelization promising tremendous performance. It is however very challenging for programmers to make use of the available processing power due to the *single instruction multiple threads* (SIMT) architecture and due to various architecture intrinsics directly passed to programmers[1]. The standards such as CUDA and OpenCL, as well as most other frameworks, require the formulation of the algorithm in this model. The process of developing efficient GPU kernels normally takes longer and results in code that is harder to understand than its corresponding sequential version. For these reasons, performance-critical applications are the only ones justifying the extra complexity. Our goal is to substantially simplify GPU programming in order to lower this cost and, thereby, to extend the range of applications that can benefit from superior GPU performance.

The use of dataflow models to express calculations that run concurrently on heterogeneous hardware is gaining more importance – a statement also backed by Google's very recent release of TensorFlow [11]. We understand a dataflow as an asynchronous reactive process in which data is propagated through a graph of operations along the connections triggering the processing of the data at each operation. This programming model is general and equally suitable for both fine-grained and coarse-grained operations; fine-grained use is only limited by the overhead of the runtime system. By providing a library of generic parameterizable dataflow operation implementations for both GPU and CPU, programmers can readily write a wide range of applications on this abstraction level. In addition, there is the possibility to implement custom operations for very specific problems, offering the same possibilities as CUDA C [19] or other cross-platform frameworks in managed runtimes [1, 2, 4, 5, 10, 33, 46]. Moreover, custom operations can easily wrap other GPU libraries. In the future, the model could also be applied to other technologies, e.g., to FPGAs or to heterogeneous distributed systems.

Alea reactive dataflow provides classes to model dataflows accompanied by a runtime system that takes care of the efficient execution. Alea reactive dataflow implements this programming model based on the .NET framework. The evaluation of the system shows that the overhead for dataflow orchestration and memory management is very low. The performance of the generated kernels is in the same range as the corresponding CUDA C version.

Alea reactive dataflow exceeds similar systems [3, 6, 11, 12, 22, 28, 34, 35, 40, 42–44] in its versatility and convenience (except for distributed evaluation). It allows cyclic dataflows, supports flexible dataflow synchronization, supports custom operations, generics and lambdas, i.e., it is fully extensible, provides automatic and minimal data movement, performs garbage collection on the GPU, avoids unnecessary GPU synchronization and supports seamless heterogeneous computation on the CPU platform and multiple GPUs. In addition, it offers the possibility of just-in-time optimization of the operation graph for GPU execution. Both the construction of operation graphs and custom operation implementations can be elegantly programmed in a .NET language, e.g., C#. The sum of these features facilitate its ease of use and assist in producing high-performance solutions. The related work section provides a more systematic discussion.

We introduced the Alea reactive dataflow programming model on a conceptual level in [7]. This paper augments and complements that paper with (1) a more comprehensive description of the programming model including execution semantics, (2) the description of the runtime system design including optimizations, and (3) the report on the experimental evaluation of the system. The remainder of this paper is structured as follows: Section 2 elaborates on the programming model. Section 3 describes the runtime system. Section 4 describes how to extend the system. Section 5 presents an experimental evaluation of the system with artificial tests and a realistic application. Section 6 discusses related works, while section 7 draws a conclusion.

* 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.

[1] The programmer needs to explicitly optimize the use of on-chip memory, warp-thread-divergence and access alignment to GPU global memory (coalescing) to achieve highly performing code. He also needs to explicitly move data from host to GPU device and vice versa [13].

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Figure 1. Graphic representation of operations

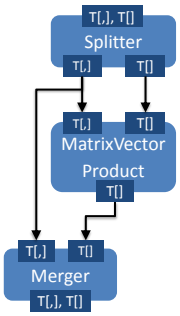

Figure 2. Operation graph for a Markov chain calculation step

2. Programming Model

2.1 Operations and Graph

An *operation* implements a function, of one or more parameters and one or more results. Operations can feature function implementations for CPU and/or GPU. An operation interacts via input ports representing the parameters of the function and output ports representing the results of function applications. Each port defines the type of the input it consumes or produces, ports can be parametrized with generic types. Figure 1 depicts operations illustrated with rounded boxes, input ports with square boxes at the top border and output ports at the bottom border. The ports are annotated with their data type. The operation `Splitter` for example, takes a tuple as input and produces both components at separate output ports; it is defined as follows[2].

```
class Splitter<T1, T2> : Operation
{
  InputPort<Tuple<T1, T2>> Input
    { get; private set; }
  OutputPort<T1> First { get; private set; }
  OutputPort<T2> Second { get; private set; }
}
```

The `Map` operation is defined analogously; in addition, it takes a lambda-function as argument in the constructor that is applied to the input. Operations can be connected together to form a directed graph. A particular instance of a graph is created by instantiating operations and by connecting output ports to input ports of matching type. The operations together with the topology determine the graph's composed functionality. There are no restrictions on supported topologies: single output ports can be connected to multiple input ports, multiple output ports can be connected to a single input port; input as well as output ports can also remain unconnected. Graphs can contain cycles. Figure 2 illustrates one Markov chain calculation step. The `Merger` is the inverse of the `Splitter` operation and `MatrixVectorProduct` calculates the matrix-vector product of the inputs. The graph for the iteration phase can be constructed as follows.

```
  var splitter =
    new Splitter<float[,], float[]>();
  var mvp = new MatrixVectorProduct<float>();
  var merger = new Merger<float[,], float[]>();

  splitter.First.ConnectTo(mvp.Left);
  splitter.First.ConnectTo(merger.First);
  splitter.Second.ConnectTo(mvp.Right);
  mvp.Output.ConnectTo(merger.Second);
```

[2] Code is denoted in the C# language; non-private visibility modifiers are omitted.

Table 1. Standard Operation Catalog

Category	Operations
Controlflow	Merger, Predicate, Splitter, Approximator, Turnout
Data Manipulation	Map, MapCyclic, MapValue, MapColToMatrix, MatrixMap, MatrixColumnExtract, MatrixSumColumns, MatrixTranspose, Reduce, ReduceBy, Scan
Mathematical / Statistical	CartesianProduct, Convolution, MatrixProduct, MatrixVectorProduct, ScalarProduct, Random
Convenience / Performance	Average, MatrixSum

The framework provides a catalog of prefabricated vector-parallel or control-flow operations summarized in table 1. All operations are implemented for both CPU and GPU. Many operations are generic, i.e. only provide a partial implementation skeleton to be completed by a delegate/lambda at construction time. The lambda is then applied to the data in parallel, e.g., `Map` applies a side-effect-free function to each element of a single- or multi-dimensional array. The library contains different versions of `Map` for one to three inputs. This enables relatively high expressiveness with a small set of operations.

2.2 Streams and Dataflow

An operation graph can be executed as a dataflow. The execution is triggered by sending data into operation input ports. When an operation has received all required data, it executes by applying the operation's function to the supplied data, producing the resulting data at its output ports. Not all operations require data at all input ports; operations can define per execution from which ports they require input, e.g., require only one input at any port, one input at a particular port and use input from other ports if available or ignore other ports, and so on.

At runtime, two connected operation ports define a stream of data starting at the output and ending at the connected input port. A stream is an indefinite sequence of data of the ports' type with items arriving in arbitrary intervals. If multiple output ports are connected to a single input port, the streams are merged with an undetermined order. If a single output port is connected to multiple input ports, each item is propagated to all associated streams. Streams connected to a cycle lead to iterative and potentially infinitely running dataflows if not checked by appropriate control operations. Data arriving at unconnected output ports is discarded.

Operations themselves do not produce any side effects, but their output ports can pass results to previously registered reception delegates. The following code illustrates this registration for the Markov chain example. Data passed to reception delegates can be interpreted as results of the entire dataflow calculation.

```
merger.Output.OnReceive(result => ...);
```

Graphs must be fully constructed including the registration of delegates before they can be used. This prevents concurrency issues

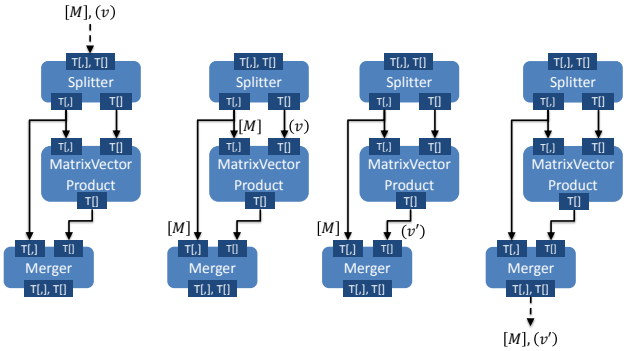

Figure 3. The dataflow for the Markov chain calculation step (from left to right)

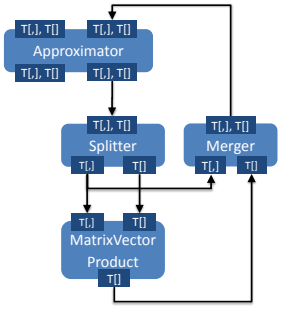

Figure 4. The complete Markov chain calculation graph

with graphs changing during execution. Also, data may only be sent into unconnected input ports or internal ports connected to control-flow operations to leave room for structural optimization of the graph. The following code illustrates how to invoke the Markov calculation step.

```
splitter.Input.Send(
  new Tuple<float[,], float[]>(M, v));
```

Figure 3 illustrates the evaluation process of the Markov chain dataflow. A Markov chain is described by a state transition probability matrix M and a vector v containing the current probability of each state. Matrices M and vector v arrive at the `splitter` operation that propagates M and v separately. When M and v arrive at the `mvp` operation, it calculates the product and yields v'. Since the `merger` requires all inputs, M is pending until v' arrives a the `merger` operation that finally produces the resulting tuple containing the matrix M and v', the new probability of each state. The tuple is passed to the reception delegate that can, e.g., display the new state probabilities to the user.

The `Approximator` operation can be used to approximate a fixed-point of an iteratively calculated value until a defined threshold (defined at construction of the operation) is reached. It can be used to complete the Markov chain shown in figure 4 in order to approximate the steady state probabilities. The steady state calculation is triggered by sending M and the initial v into the left input port of the `Approximator` operation resulting in a repeated execution of the Markov chain step until the threshold is reached and the `Approximator` operation routes the data out of the cycle.

The description of the evaluation process illustrates that it is data driven and data is propagated through the graph as soon as possible, only pending at operations until they have all required input. The process continues as long as there are operations that can execute. Of course, graphs must be designed such that the dataflows terminate with all data being processed and without pending data. The sending (production) and reception (consumption) of data is asynchronous on dataflow as well as on operation level, a so-called reactive execution.

It is supported to concurrently provision dataflows with multiple input sets, resulting in multiple sets of associated values being pushed through the graph. These sets must remain strictly separated during the entire process to ensure correct execution. Moreover, it must be possible to relate the output sets to the corresponding input sets. The runtime system ensures that the sequential order of data along any path through the graph is preserved. In order to leverage this to the level of entire dataflows, the programmer has the responsibility to select the topology such that the separation of the sets remains intact when joining paths (as is is the case in the Markov chain example).

Data passed between operations must be immutable. All prefabricated operation implementations do not modify input data but instead produce new output data; all custom operations are required to do this as well. The evaluation engine can therefore freely optimize data propagation by using copying or referencing.

The execution on heterogeneous hardware remains transparent. The operations' function can be implemented for the CPU and/or GPU platform. Each operation can thereby define the strategy for the selection of the platform at runtime based on the availability of data on the two platforms. The runtime system automatically performs all necessary data transfers to ensure data availability.

Memory management on CPU (host side) is performed by the .NET runtime system; however, this does not apply to data residing in GPU memory. The dataflow runtime system takes care of disposing unreachable memory and kernels on the GPU, providing a uniform model to the programmer in this regard.

3. Runtime System Implementation

3.1 Dataflow Execution

The architecture of the runtime system is driven by the reactive execution concept, the needs of the data propagation and the GPU garbage collection.

The runtime system builds up a data stream infrastructure for each dataflow holding all the data that will be propagated within it. Streams are realized by FIFO queues associated with each operation input port. The data propagation structure can be used to achieve the reactive execution. Each time data is enqueued at an input port, the readiness of the owning operation is checked. In case it is ready, data is dequeued from all used streams and the execution platform is determined. The actual execution of the operations' implementation is delegated to .NET TPL [21] in case of the CPU, and to the *GPU scheduler* in case of GPU platform described in the next section.

The operation implementation is supplied with access to the dequeued data, which it can, but must not consume. The runtime system ensures that all required data for a particular operation execution is present on the selected platform. For this purpose, each piece of data is encapsulated and can transparently reside in the host and/or GPU memory. The framework applies lazy copying of the data to minimize work. In case of the CPU platform, the implementation performs the actual calculation; in case of the GPU platform, it generates a sequence of GPU kernel launches performing the calculation. During this processing, the implementation can produce new data for output ports that is subsequently propagated to all connected streams.

The runtime system performs garbage collection on the GPU platform. GPU garbage collection can be safely performed by a reference counting scheme in case of dataflows. Reference count-

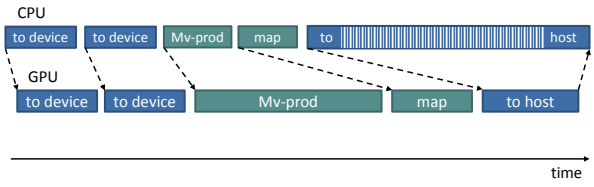

Figure 5. Qualitative timing and dependency diagram for an exemplary command sequence with two data transfers and two kernels

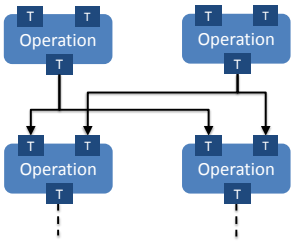

Figure 6. Worst-case scenario for data affine scheduling on operation level

ing is applied to all data used in the dataflow. The engine counts the usages for each device memory block, i.e. the number of operations where it is currently in use plus the number of streams in which it is buffered. When the counter becomes zero, the block is automatically freed on the GPU. In contrast to ordinary reference counting implementations, the mechanism is sound since the usage counting is acyclic: blocks do not carry references to other blocks, but only operations and streams may reference blocks. The potentially cyclic dataflows themselves are managed by the .NET garbage collector that *can* dispose cyclically referencing objects. Since the runtime system has the right to move data from host to device memory, the reference counting scheme has to be applied irrespective of the current location of the data.

The data stream infrastructure is associated with instances of the operation graph exclusively via weak memory references in order not to block the garbage collection of the graph and to enable the garbage collection of the data propagation infrastructure which is distinct from the data within the streams managed with the reference counting scheme.

The programming model requires that the sequential order of data along any path through the graph is preserved. The runtime system ensures this by using thread-safe and lock-free queues for the streams, and by not allowing the concurrent execution of the same operation instance using atomic counters. This implies that data sets belonging to different calculations cannot overtake each other which might be beneficial in some cases. Our design avoids this complexity having the advantage of simpler and, with that, faster scheduling decisions.

3.2 GPU Scheduling

Since the GPU does not support time-slicing as the CPU does, GPU commands have to be serialized. Also, there is a significant delay in the communication to GPU devices. Each GPU has a command pipeline that supports hiding this delay. The pipeline can be filled with kernel launch and data transfer commands. The GPU offers additional device synchronization commands; calling these methods exposes the communication latency. The GPU scheduler is designed not use these commands, but instead performing all kernel calls in a dataflow consistent order. As an exception in case of device-to-device transfers, data dependencies have to be ensured by explicit synchronization.

Consider an exemplary dataflow consisting of a matrix-vector-product followed by a map operation with the qualitative execution timing shown in figure 5. The runtime launches two initial host-to-device data transfers, followed by two kernel launches and a final device-to-host data transfer to run this dataflow. The execution of each command goes through the phases invocation, pending and execution. The pending and execution phase are happening on the GPU device. The asynchronous behavior on GPU command level transcends to the dataflow operation level. So, each operation scheduled on the GPU platform goes through the same phases as the individual commands. Thus, single operations or subgraphs can be completely scheduled, but not have started execution. The GPU command pipeline effectively enables ahead of time scheduling of operations and with that hiding the overhead of the dataflow runtime. This is only limited by operations executing on CPU requiring input to be present in the host memory and inter-device data transfers.

3.3 Multi-GPU Support

There is a recent trend towards using multiple GPU devices. Naturally, the runtime system should be able to distribute the execution of the GPU kernels to these devices. GPU devices require the data to be present locally, the resulting data transfers between GPUs do however take a relatively long time to complete and limit ahead of time scheduling. They should therefore be minimized, which can be done by scheduling dependent kernels on the same GPU device. The GPU scheduler manages the pool of available GPU devices and can be configured to use one of the two following pragmatic strategies manage this pool.

The first strategy schedules entire dataflows to GPUs which means that all operations of a particular dataflow will always execute on the same GPU device. This has the advantage that there are no inter-device data transfers happening. Ideally, each dataflow is executing exclusively on its GPU device, but if there are more dataflows that GPUs, execution of single operations of different dataflows are interleaved on a common GPU device. This strategy has the disadvantage that a program consisting of a single dataflow graph cannot make use of multiple GPU devices; or similarly, in a scenario with multiple dataflows, there can be significant load imbalance. Nevertheless, there is a large class of applications processing many independent problem instances at a large granularity allowing for parallel execution by instantiating multiple copies of the dataflow and taking care of concurrent provisioning.

The offered alternative is data affine scheduling on operation level, which schedules operations per execution on the GPUs in an ”on-line” approach. The GPU is selected such that the amount of data that must be transferred for the operation’s arguments is minimized. In case of equal transfer cost, the GPU is selected with the smallest expected waiting time. The advantage of this method is that the system automatically uses all available GPUs and can perform adaptive load balancing. This strategy can distribute different sets of associated values to different GPUs. However, it can also, in the worst case, lead to performance degradation compared to single GPU execution: Consider the repeated dataflow shown in figure 6. Operations on the left will always be executed on one GPU and operations the right hand side always on another GPU. This can happen if the outputs of the operations at the top have the same size resulting in the second priority scheduling criteria to be applied: as soon as a kernel is scheduled on one GPU, it has a larger expected waiting than the other GPU, which causes the next kernel to be scheduled on the other GPU. This leads to a repeated ping-pong of data transfers that might take more time than merely executing the kernels on a single GPU.

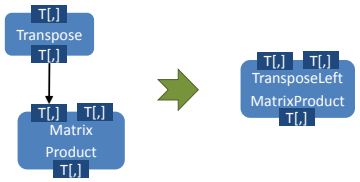

Figure 7. Fusion of a matrix-transpose and -product graph into a single combined operation

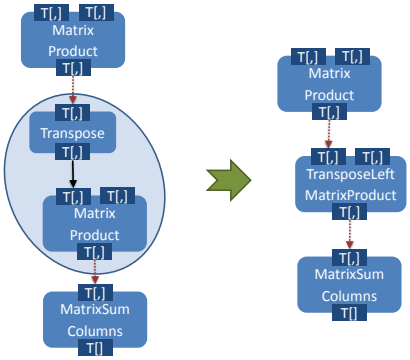

Figure 8. Application of the fusion to an exemplary graph. The connections drawn as red dotted arrows need to be cut open in the left graph and reconnected to the fused operation in the right graph.

3.4 Operation Fusion

Most GPU kernels follow the general pattern *load data from GPU global memory, process and store it*. If the process step is not computationally intensive per loaded and stored data item, kernels of this type are memory-bound. Since loading and storing of data items is inevitable, the only way to improve performance is to do more in the processing step. Applied to the dataflow model, this requires combining multiple operations into a single operation such that the loading and storing is performed once for the combined calculation. One way to achieve this, is to provide a library offering a large number of operations for composed calculations such as, e.g., cuBLAS [18]. This is not ideal with respect to good design because of too many special case operations, i.e. lack of cohesion. A better approach is to perform this optimization behind the scenes and provide the programmer a small set of powerful operations.

Since the graph must be fully constructed before it is used, the runtime system is free to modify and optimize all operations and connections in-between ports. This can be performed just-in-time at the first sending of data into a particular graph.

Combining operations implies combining CUDA kernels known as kernel fusion. Research indicates that analyzing and fusing kernels in full generality is at least a NP-complete problem [45]. Therefore, Alea reactive dataflow pragmatically enables the operation library to perform the fusion of its operations. An operation library can define any number of *fusers* that enable the runtime system to fuse a particular operation graph; an exemplary *matrix-transposed-product* fuser is shown in figure 7 and an exemplary application in figure 8. The runtime system needs to perform the following tasks in order to enable this: (1) discover the fusers provided by the library, (2) discover when a graph is used as dataflow for the first time, (3) analyze each new graph and find all fusable subgraphs, (4) create the fused operations and (5) replace the subgraphs with the new operations by rewiring.

4. Extension of the Operation Catalog

The operation catalog can be extended seamlessly. Prefabricated operations only use features that are also available for building custom operations; however, the GPU implementation of operations requires advanced knowledge of the GPU architecture as mentioned in section 1.

4.1 Operation Implementation

Both, the prefabricated operations and custom operations, are implemented based on the same classes and interfaces. The operation and port framework classes are purely declarative and do not contain any runtime system logic.

The following code shows the implementation of the `MatrixVectorProduct` operation class. Each operation class needs to derive from the abstract `Operation` base class. Each port is defined as a property of the new class having types based on the two framework classes `InputPort<>` and `InputPort<>`. The constructor must initialize the port properties and set the `Implementations` property of the `Operation` base class providing the implementation of the operation for the different supported platforms.

```
class MatrixVectorProduct<T> : Operation
{
  // declare the operations featured ports
  InputPort<T[,]> Left { get; private set; }
  InputPort<T[]> Right { get; private set; }
  OutputPort<T[]> Output { get; private set; }

  internal MatrixVectorProduct()
  {
    // initialize ports
    Left = new InputPort<T[,]>(this);
    Right = new InputPort<T[]>(this);
    Output = new OutputPort<T[]>(this);

    // define supported implementations
    Implementations = new[] {
      new CudaMatrixVectorProductImpl<T>(),
      new CpuMatrixVectorProductImpl<T>()
    };
  }
}
```

There are two base framework classes, one for CPU and one for GPU, to implement the platform-mapping of an operation. For simplicity we continue the example with the `Map` operation. The key piece is the `Execute` method that receives a `script` object created by the dataflow framework functioning as interface to the dataflow, i.e. allowing to consume and produce data by indicating the respective operation port. Implementations can only consume at most one piece of data from each input port per invocation and can assume that it will be immediately available.

```
class CudaMapImplementation<TInput, TOutput>
  : CudaImplementation<Map<TInput, TOutput>>
{
  ...

  override void PlanExecution(
    Map<TInput, TOutput> operation,
    CudaScript script)
  {
    TInput[] input;
    input = script.Consume(operation.Input);
    var output = new TOutput[input.Length];
    script.Launch(new LaunchParam(...),
      Map, input, output);
    script.Produce(operation.Output, output);
  }
}
```

The execution plan is built with limited information about the data to be processed, i.e. scalar values and the sizes of input arrays, to make decisions for optimal kernel launch configurations. This is because the execution plan and the launch configuration are built on the host, but the data resides on the device. It is thus not allowed to read from or write to arrays. Obviously, to run efficiently on GPUs, operations need to implement a massively vector-parallel calculation per input, e.g., `Map` transforms an array of elements.

The GPU implementation also includes one or several kernels written as .NET methods that are seamlessly integrated into .NET resembling the standard CUDA C model. On a syntactic level, normal .NET types, in particular arrays, can be used as arrays instead of low-level pointers. A special property of our model constitutes the ability to used generics and invoke .NET delegates inside kernels. The following code illustrates a simple CUDA kernel implemented in C# applying the `_map` delegate being an instance field to an input array and storing the result in an output array.

```
void Map(int[] input, int[] output)
{
  var start = blockIdx.x*blockDim.x+threadIdx.x;
  var stride = gridDim.x*blockDim.x;
  for (var i = start; i < output.Length;
       i += stride) {
    output[i] = _map(input[i]);
  }
}
```

4.2 GPU Cross Compilation

We engage automatic kernel cross compilation from .NET CIL (Common Intermediate Language) to the target CUDA PTX (portable executable). Apart from kernels, cross compilation needs to include all methods/lambdas that the kernel may directly or indirectly call. Non-recursive methods are in-lined by our compiler. We impose certain restrictions on the translatable GPU code, i.e., exceptions, object references, object creation, IO code, unmanaged code cannot be cross-compiled and yield an error. Kernels however support all primitive types, struct-types, as well as access to closure variables of lambdas and to static variables.

By default, cross compilation is triggered by the runtime system on the first launch of a kernel. Alternatively, programmers may opt-in for ahead-of-execution cross compilation. Of course, ahead-of-execution entails certain restrictions, in particular that the callable methods/lambdas can be statically inferred; otherwise, a fall-back to runtime translation occurs. For performance improvements, we cache generated target code and only recompile it when the host code has changed (detected by a hash value of the binaries).

Architecturally, we realized the translation from CIL to CUDA by the help of the LLVM [39] compiler framework and its specific CUDA backend (NVVM [30]). During cross compilation, debug information (source code locations and variable name mappings) can be included into the generated PTX code.

5. Experimental Evaluation

We implemented a number of artificial tests and realistic sample applications to validate the programming model and the performance of the entire system. We used the hardware as specified in table 2 for all evaluation purposes; the E5 does not use hyper-threading.

5.1 Performance of Micro Tests

We run micro tests to evaluate specific performance aspects of the system. The first series of tests compares the wall time[3] for a number of dataflows running with our kernel-cross-compilation approach on a single GPU (DFG1) to C++ applications with the same kernels written in Nvidia's CUDA C, to .NET implementations running entirely on all 4 CPU cores (DFC4) and to itself using two GPUs (DFG2). The GPU kernels are modestly optimized not exceeding 50 lines of code. Each benchmark is composed of two basic operations implemented as single kernels and two simple dataflows. The size of the inputs is 1,000,000 for one-dimensional and 1,000 x 1,000 for two-dimensional arrays. Our test setup is designed to not expose the extra overhead caused by .NET's managed heap slowing down host-to-device data transfers by almost a factor of 2. The underlying assumption is that the benchmark dataflows are part of a larger dataflow that is off-loaded to the GPU.

[3] The measurement denotes the average wall time over 100 runs including the time of a single initial host to device transfer and a single back transfer for the GPU versions.

Table 2. Hardware used for performance evaluation

	GeForce GTX TITAN Black	Intel Xeon E5-2609
Number of (multi-)processors	15	4
Number of threads	2880	4
Internal clock rate	0.98 GHz	2.4 GHz
Memory clock rate	3.5 GHz	3.2 GHz
Level 2 cache	1.5 MB	1 MB
Level 3 cache	-	10 MB

Table 3. Micro benchmark results for arrays of `floats` (absolute time of DFG1 and speedup relative to other setups). The measured times for the data type `int` are practically the same; `double` arrays take around 30% to 50% longer for both frameworks.

Benchmark	DFG1	$\frac{\text{CUDA C}}{\text{DFG1}}$	$\frac{\text{DFC4}}{\text{DFG1}}$	$\frac{\text{DFG2}}{\text{DFG1}}$
Average	1.69 ms	1.12	11.6	0.56
Convolution	5.91 ms	1.03	37.0	0.51
Multiply Add	30.4 ms	0.81	400	0.51
Monte Carlo π	6.11 ms	1.06	5.3	0.57
Arith. Average	-	1.01	-	0.54

Table 3 shows the results of these comparisons. The comparison to CUDA C suggests that simpler kernels perform slightly faster while more involved kernels perform slightly slower, but overall in the same range as implementations in CUDA C. The comparison to pure CPU execution (DFC4) shows that GPUs can indeed outperform CPUs. If the task is compute-intensive and large enough to saturate the GPU and hide the memory latency, the speedup is immense – as it is the case for the matrix multiplication (factor 400).

The DFG2/DFG1 benchmark assesses the effectiveness of the multi-GPU scheduling. For the DFG2 setup, we apply the GPU assignment on dataflow graph level with one graph created per GPU and concurrent provisioning, i.e. sending data in parallel into the two dataflows. The input is split up such that the same amount of numbers is calculated in total. An average speedup of 0.54 in the table of course equals a 1.85 speedup of the dual over the single GPU setup. It can be observed that fast running kernels or dataflows composed thereof exhibit slight suboptimal scaling; this is due to the circumstance that there is only half as much data to be processed per dataflow exposing some serial overhead of the runtime system and the test setup.

To evaluate the overhead of the dataflow scheduling, we created an artificial dataflow building a binary tree consisting of 2'000 trivial operations propagating data from the leaves to the root. Running the test in pure CPU mode shows that the orchestration of the evaluation process and the reference counting used for garbage collection consume around 5 microseconds per operation; the vast majority of efficiently running GPU kernels take much more time

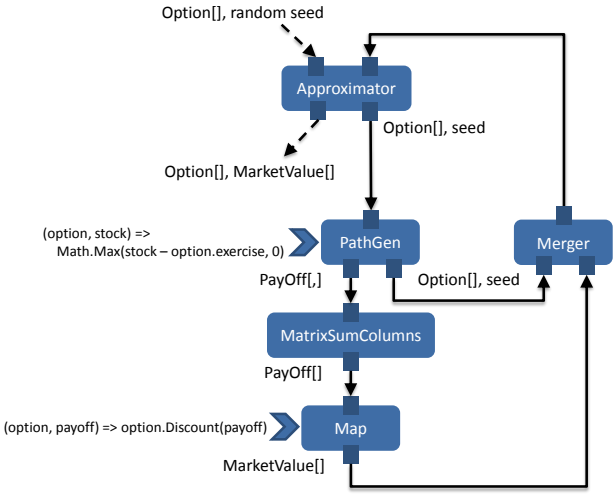

Figure 9. Dataflow of the Monte Carlo option pricing sample application

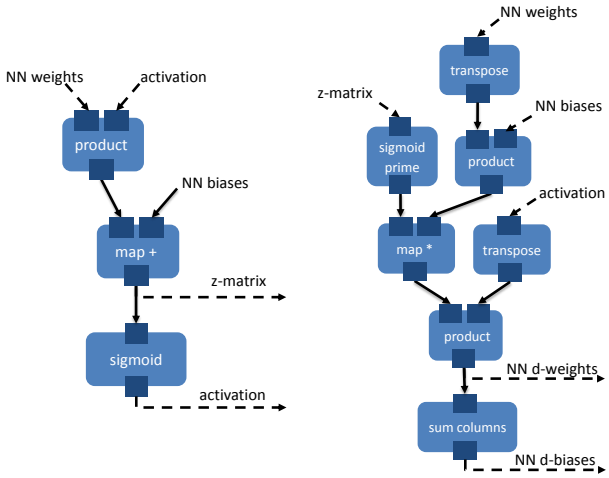

Figure 10. Dataflow of the forward (on the left) and the backward propagation (on the right) phase used for training neural networks (the + and * signs annotated on the map operations denote the lambda function)

to execute. This micro test shows that the data management and scheduling overhead of the dataflow runtime is negligible for GPU applications and that the system can also be used for more fine-grained CPU applications.

Overall, the performance evaluation shows that the dataflow and the CUDA .NET runtime system incurs a small overhead over direct implementation based on the CUDA C model.

5.2 Application Cases

We built two relatively complex applications to evaluate the dataflow model. The first case originates from the financial domain and applies the dataflow model to calculate the price of financial options. Monte Carlo simulation is one of the problem solving strategies that are very suitable to exploit data parallel hardware architectures and a well-known and general approach to option pricing. Option pricing with Monte Carlo simulation incrementally approaches the mathematically exact price by generating random stock price paths and applying the payoff formula at defined points in time and discounting the payoffs back to the valuation date. Figure 9 shows the dataflow used for the option pricing application. The `PathGen` and the `Map` operation are parametrized with the annotated lambdas. The dataflow is cyclic reflecting the incremental approximation until the present value of the option is considered steady. `PathGen` is a custom operation performing the actual generation of the stochastic price paths. In addition to calculating thousands of price paths at a time, the dataflow also simultaneously calculates a number of options with the same expiration date for the same underlying to increase parallelism inside all operations. Moreover, the dataflow supports the concurrent calculation of multiple sets of options by ensuring that all relevant information for one set is kept together for each new iteration round.

The second sample application implements a machine learning engine based on a fully connected neural network. Learning is an iterative process training the network with a newly composed set of data (epoch) until a defined recognition rate has been achieved. We used the MNIST dataset [20] for the classification of hand-written digits.

The engine applies the dataflow programming model on two levels, on the coarse-grained level to formulate the entire problem solution and on a fine-grained level for the performance critical parts being the training and evaluation phase of the neural network. The implementation confines the neural network to GPU device

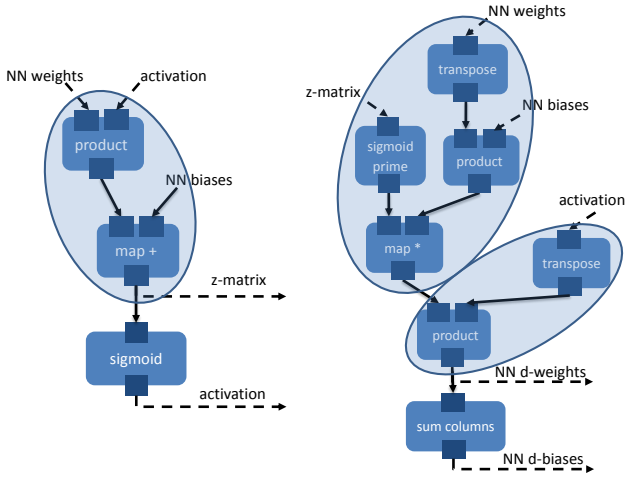

Figure 11. Fusing operations of the forward (on the left) and the backward propagation dataflows (on the right)

memory during the entire process to avoid the prohibitive penalty of repeatedly copying it from host to device and vice versa. Consequently, all operations working with the neural network run on the GPU platform and the others entirely on the host.

Figure 10 shows the forward and the backward propagation dataflows for one layer of the neural network used in the training and evaluation phase (with some variation). All operations are standard operations contained in the set of pre-fabricated operations. Again, multiple digit images are trained simultaneously for more data parallelism allowing the formulation of the critical product operations as favorable matrix-matrix instead of matrix-vector products. In addition, we implemented all the applicable fusers for the two dataflows to achieve the best possible speedup with our framework and to assess the effectiveness of this optimization technique. Figure 11 shows how the graph is optimized denoting operations

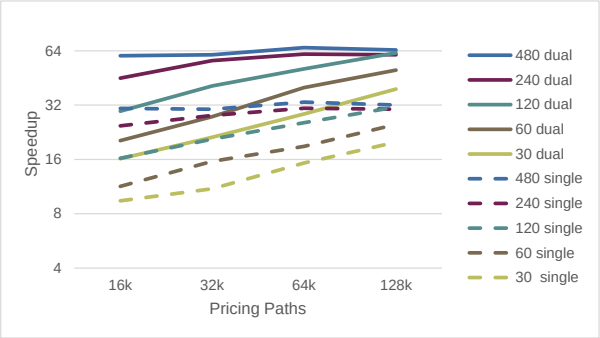

Figure 12. Speedup of single- and dual-GPU over a multi-threaded 4-core CPU execution for the option pricing case with the number of simultaneously calculated pricing paths

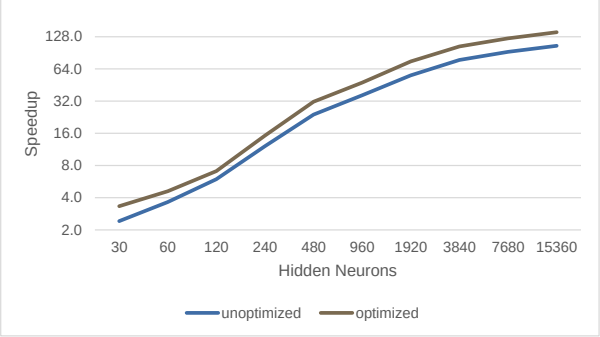

Figure 13. Unoptimized and optimized speedup of the single-GPU execution over a multi-threaded 4-core CPU execution for the machine learning case with increasing numbers of hidden neurons

that are fused into single operations implementing the composed functionality encircled with ellipses.

Both application cases have been implemented purely in C#. The option pricing case is around 300 lines of code including a custom GPU operation and the machine learning case is around 800 lines of productive code including 5 custom CPU operations for the high-level dataflow.

5.3 Performance of the Application Cases

The performance evaluation of the applications determines the speedup the GPU can achieve over pure multi-threaded CPU execution, assessing the suitability of the cases for GPU acceleration and verifying that Alea reactive dataflow scales up to real application cases.

Figure 12 shows the achieved speedup of the Monte Carlo option pricing application for different calculation parameters (option maturities in days) and setups (single and dual GPU execution). The speedup relative to pure CPU execution mainly comes from the data-parallel stochastic stock price path generation. The diagram shows that the dataflow executing on a single GPU can already achieve a decent speedup for medium to large pricing parameters. The diagram also shows that the scheduler scales well to dual GPU execution.

Figure 13 shows the achieved speedup of the training phase of the machine learning application. The problem size has been increased artificially by increasing the number of hidden neurons.[4] The speedup mainly comes from the matrix-products. The overhead of copying the images of the handwritten digits from host to device memory in combination with the inefficiency of GPU kernels for small datasets results in a low speedup for small neural networks. The comparison of the optimized versus the unoptimized dataflow shows a speedup of around 30% largely independent of the problem size.

Both graphs show that the problem needs to have a certain size and with that enough data parallelism to allow the GPU to achieve a significant speedup. When comparing the option pricing to the machine learning speedup, it can be observed that the machine learning case is dominated by the matrix multiplication achieving an excellent speedup for large enough matrix sizes; the Monte Carlo simulation conversely is fully data parallel, but the individual GPU CUDA threads work is more complex and with that still, but less suitable for the GPU architecture.

The two cases demonstrate that our system as a whole works reliably with respect to, e.g., garbage collection or multi-threading and can achieve good speedups for realistic applications.

6. Related Work

Our model can be classified as a flexible and accomplished variation of a data driven static dataflow model allowing multiple pieces of data on arcs, non-strict evaluation and non-recursive but cyclic graphs. Our model is targeted at homogeneous programming of heterogeneous platforms, specifically CPU and GPU.

The recent related works can be classified by programming model into low-level and high-level focused models; the high-level models can be further split up into pure dataflow, imperative, hybrid-dataflow-imperative and functional as well as hybrid-functional-imperative programming models.

Low-level GPU parallel programming frameworks, such as CUDA [19] (with version 7 also supporting lambda functions) OpenCL [14] are applied to implement the vast majority of the frameworks. Several frameworks raise these imperative models into managed runtime systems providing seamless integration into the platform languages, for Java [33, 46] and for .NET [1, 2, 4, 5, 10].

Among the high-level programming models, pure dataflow models gain increasing relevance for programming heterogeneous parallel architectures, in particular for GPUs. PTask [34], Dandelion [35], Xcelerit [22], Hyperflow [44], FastFlow [3], FlowCL [43], and GpuLinq [28] all employ a dataflow abstraction to express GPU parallelization. The strength resides in the descriptiveness, leaving the degrees of freedom for the runtime system to schedule flexibly, minimize memory copying, and select among multiple implementations or tune configurations per operation. As demonstrated by TensorFlow, PTask, Dandelion, Xcerlerit and others, this approach also enables seamless generalization towards distributed parallelization on CPU/GPU clusters, a step we have not yet taken for our system.

The reactive character of our model is inspired by Rx.NET [17, 27] and the TPL dataflow [41], although these systems are only suited for CPUs, not for GPUs because of lacking integration.

While still considering data dependencies, StarPU [6], XKaapi [42], StarSs [40], and Harmony [12] remain more imperative than the aforementioned pure dataflow models. These models promote a notion of task parallelization, where tasks can be dispatched on GPUs. Due to the additional task dependencies, the runtime system cannot as freely optimize data management as in merely descriptive dataflows. Data dependencies need to be inferred in Harmony,

[4] This does not result in an improved recognition rate for this particular learning problem, which is however not relevant from a performance perspective.

and annotated in the other frameworks of this type. In TensorFlow, the use of control-dependencies is optional, but helpful for, e.g., for controlling peak memory usage. C++ AMP [16] and OpenACC [32] take this approach further by remaining purely in the C++ programming model. Whippletree [36] is an interesting and innovative combination of a hybrid high-level and low-level approach in the area of task based systems. It allows composing warp-, block- and device-level tasks into so called mega-kernels performing fine-grained scheduling of these tasks and is therefore better able to exploit sparse, scattered parallelism.

Google's very recently released TensorFlow [11] is aimed at large-scale (distributed) machine learning, but it can be considered as a general-purpose programming model. It features control-edges and stateful variable operations as optional elements. It can derive a gradient-version of a dataflow graph which is frequently needed in machine learning. TensorFlow features an elaborate cost model to support distributed scheduling, which is however, due to its greedy simulation heuristic, in principle not superior to on-the-fly scheduling in case of local execution.

In the area of functional and multi-paradigm programming languages Firepile [31] allows targeting the GPU in Scala and the Alea GPU development system [4] in .NET F#. Implementation of GPU kernels is seamlessly integrated into the respective languages but remains essentially imperatively formulated. Delite [8] is a framework for parallelization of DSLs that can use Scala ASTs as their base. GPU support is limited to a set of parallel execution patterns such as Map, Reduce, ZipWith, and Scan for which the Delite runtime generates optimized kernels and executes them in an optimized execution plan, a similar approach is taken by Rust [15] which has an interesting concept of unique pointers to avoid conservative duplication of data. Nikola [25], Accelerate [9] and Obsidian [38] allow targeting the GPU with array computations embedded in Haskell. These frameworks apply a number of advanced optimizations, including the composition of GPU kernels from the embedded domain specific languages.

In the following, we highlight the features of Alea reactive dataflow with respect to the most related works in the area of high-level dataflow and imperative models.

Our scheduler uses the lazy copying approach, as described in the PTask system, to minimize memory transfer between host and GPU devices. In addition, the runtime system performs garbage collection of blocks allocated on the GPU. This can only be performed by dataflow and task based systems keeping track of data dependencies.

Alea reactive dataflow supports cyclic graphs resulting in iterative computation. This permits us to solve complex application cases in one dataflow. TensorFlow, FastFlow, Hyperflow and Harmony also support feedback cycles.

We apply on-the-fly scheduling based on available data input for operations, similar as in PTask, Dandelion and Hyperflow, but without the possibility to define priorities. Alea reactive dataflow is more general than all other systems with respect to dataflow synchronization, allowing each operation to determine the set of inputs it requires for the next execution. To the best of our knowledge, there is no other system targeting GPUs, including imperative and functional approaches that supports this. Moreover, our script metaphor also permits ahead-of-time scheduling of sub-graphs in one GPU stream.

Another distinction point of our system is the genericity: Operations can be parameterized by generic types and lambda/functions. This means that operations do not carry fix implementation but their implementation is completed at instantiation time. This requires cross-compilation of host program code at runtime, in our case from .NET IL to GPU code. Most dataflow systems and GPU libraries, such as Unbound [29], do not have that possibility with the exception of Dandelion, FastFlow, GpuLinq and SkelCL [37]. In these frameworks, a reduce operation, for example, only offers a fixed set of aggregator functions; otherwise, new custom operations need to be implemented. Dandelion, GpuLinq and SkelCL are also limited to (directed acyclic) queries with a fixed operation repertoire [26]. In FastFlow and the Boost.Compute library [23], GPU-feasible functions for operations need to be wrapped in C macros due to the missing support for dynamic code reflection under C/C++. However, strarting with version 7, CUDA supports parameterizing kernels with lambdas, and as a result, the support for this is expected to grow.

Finally, in the area of optimization for dataflow execution, Helium [24] follows a compelling approach to kernel fusion of OpenCL programs. Helium intercepts OpenCL function calls to build a dependency graph of kernel launches and data transfers. Helium builds up the graph until the host program requires the results. At this point, the graph is analyzed and optimized. Optimizations include kernel fusion, task parallelization and code specialization. However, this elegant approach has drawbacks as well: The complexity of the kernels that can be fused is limited, e.g., a matrix transpose kernel cannot be fused automatically with a matrix multiply kernel. Also, delaying GPU command invocation exposes the host-to-device communication delay and even increases it because the graph analysis and kernel fusion is performed during this critical time window that is relevant especially for applications with relatively short running kernels. Alea reactive dataflow, in contrast, can be extended with kernel fusion methods that include know-how about the particular kernels to be fused and interferes much less with the host-to-device communication timing. To the best of our knowledge, there is no other system that applies this framework approach.

A high-level description of the Alea reactive dataflow programming model has been previously presented in [7] – however without implementation details and without experimental results.

7. Conclusions

The Alea reactive dataflow system establishes a high-level dataflow programming model for simple yet efficient GPU parallelization. It proves to be particularly powerful because of its asynchronous nature that supports cyclic and iterative computation as well as its genericity where operations can be parameterized by lambda-functions. Moreover, it offers a seamless and clean extension mechanism of the operation catalog.

We have built an efficient lock-free runtime system based on the .NET framework that takes care of orchestrating kernel launches and efficient data management, including GPU garbage collection. Its clean design allows the programmer to focus on the application problem and separate concerns. The performance evaluation confirms that the systems runs robustly and efficiently and can achieve high speedups on single and multiple GPUs with only a low performance overhead compared to direct imperative GPU programming.

References

[1] Cudafy.net. `http://cudafy.codeplex.com`. Accessed: 2014-08-25.

[2] Programming the graphics processors (gpu) in mc# language. `http://www.mcsharp.net/documentation/GPU_programming_with_MCSharp.pdf`. Accessed: 2015-07-29.

[3] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In *Euro-Par 2012: Parallel Processing Workshops*, pages 47–56. Springer, 2013.

[4] Alea. Alea cubase. `https://www.quantalea.net`. Accessed: 2014-08-25.

[5] Altimesh. Altimesh hibridizer. http://www.altimesh-hybridizer.com. Accessed: 2015-07-29.

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[7] Luc Bläser, Daniel Egloff, Oskar Knobel, Philipp Kramer, Xiang Zhang, and Daniel Fabian. Alea reactive dataflow: Gpu parallelization made simple. In *Proceedings of the companion publication of the 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity*, 2014.

[8] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *ACM Sigplan Notices*, volume 45, pages 835–847. ACM, 2010.

[9] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[10] G. Cocco. Fscl compiler. http://fscl.github.io/FSCL.Compiler. Accessed: 2014-08-25.

[11] Jeffrey Dean and Rajat Monga et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[12] Gregory F Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200. ACM, 2008.

[13] Nevena Ilieva-Litova Alan Gray and Anders Sjöström. Best practice mini-guide accelerated clusters. using general purpose gpus. http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-GPGPU.pdf, 2014. Accessed: 2015-07-28.

[14] Khronos Group. The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl. OpenCL 2.0. Accessed: 2014-08-25.

[15] Eric Holk, Milinda Pathirage, Anamika Chauhan, Andrew Lumsdaine, and Nicholas D Matsakis. Gpu programming in rust: Implementing high-level abstractions in a systems-level language. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 315–324. IEEE, 2013.

[16] Microsoft Inc. C++ accelerated massive parallelism (c++ amp). http://msdn.microsoft.com/en-us/library/hh265136.aspx. Accessed: 2014-08-25.

[17] Microsoft Inc. The reactive extensions (rx.net). http://msdn.microsoft.com/en-us/data/gg577609.aspx. Accessed: 2014-08-25.

[18] Nvidia Inc. cublas. https://developer.nvidia.com/cublas. Version 7.5. Accessed: 2015-10-26.

[19] Nvidia Inc. Cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide. Version 6.0. Accessed: 2014-08-25.

[20] Yann Lecun. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 2014-11-19.

[21] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Acm Sigplan Notices*, volume 44, pages 227–242. ACM, 2009.

[22] Jorg Lotze, Paul Sutton, and Hicham Lahlou. Many-core accelerated libor swaption portfolio pricing. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1185–1192. IEEE, 2012.

[23] K. Lutz. Boost compute. http://kylelutz.github.io/compute/. Accessed: 2015-04-08.

[24] Thibaut Lutz, Christian Fensch, and Murray Cole. Helium: a transparent inter-kernel optimizer for opencl. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 70–80. ACM, 2015.

[25] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[26] Erik Meijer. The world according to linq. *Queue*, 9(8):60, 2011.

[27] Erik Meijer. Your mouse is a database. *Queue*, 10(3):20, 2012.

[28] Nessos. Gpu linq. https://github.com/nessos/GpuLinq. Accessed: 2014-08-25.

[29] Nvidia. Cub (cuda unbound). http://nvlabs.github.io/cub/. Accessed: 2015-04-08.

[30] Nvidia. Cuda llvm compiler. https://developer.nvidia.com/cuda-llvm-compiler. Accessed: 2015-04-07.

[31] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for gpus in scala. In *ACM SIGPLAN Notices*, volume 47, pages 107–116. ACM, 2011.

[32] OpenAcc. The openacc application programming interface. http://www.openacc.org. Version 1.0, 2011. Accessed: 2014-08-25.

[33] Philip C Pratt-Szeliga, James W Fawcett, and Roy D Welch. Rootbeer: Seamlessly using gpus from java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 375–380. IEEE, 2012.

[34] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[35] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[36] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: task-based scheduling of dynamic workloads on the gpu. *ACM Transactions on Graphics (TOG)*, 33(6):228, 2014.

[37] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl-a portable skeleton library for high-level gpu programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182. IEEE, 2011.

[38] Joel Svensson. Obsidian: Gpu kernel programming in haskell. http://www.gpucomputing.net/sites/default/files/papers/5184/lic.pdf. Accessed: 2015-07-29.

[39] The LLVM Team and Chris Lattner. The llvm compiler infrastructure. http://www.llvm.org/. Accessed: 2014-07-05.

[40] Enric Tejedor, Montse Farreras, David Grove, Rosa M Badia, Gheorghe Almasi, and Jesus Labarta. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, 24(18):2421–2448, 2012.

[41] S. Toub. *Introduction to TPL Dataflow*. Microsoft Inc., April 2011.

[42] Danjean V., Gautier T., Laferri C., and Mentec F. L. Xkaapi. http://kaapi.gforge.inria.fr/XKaapi/XKaapi.html. Accessed: 2015-04-09.

[43] Sytse van Geldermalsen. Flowcl-declarative dataflow api for heterogenous platform computing. Master's thesis, University of Amsterdam, August 2013.

[44] Huy T Vo, Daniel K Osmari, Joao Comba, Peter Lindstrom, and Cláudio T Silva. Hyperflow: A heterogeneous dataflow architecture. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 1–10. The Eurographics Association, 2012.

[45] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *ACM SIGPLAN Notices*, volume 48, pages 57–68. ACM, 2013.

[46] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par 2009 Parallel Processing*, pages 887–899. Springer, 2009.