Alea Reactive Dataflow
GPU Parallelization Made Simple

Luc Bläser
IFS Institute for Software, HSR Rapperswil

D. Egloff, O. Knobel, P. Kramer, X. Zhang, D. Fabian
(Joint HSR and QuantAlea Zurich)

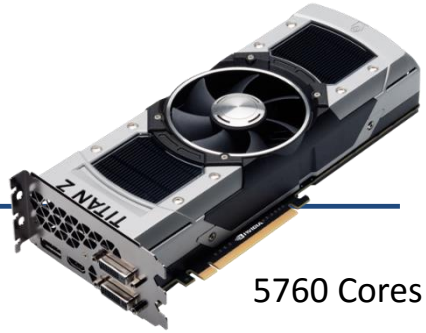
HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT FÜR SOFTWARE

QuantAlea

REBLS'14
21 Oct 2014

GPU Programming Today



5760 Cores

- Massive parallel power
  - Very specific pattern: vector-parallelism
- High obstacles
  - Particular algorithms needed
  - Machine-centric programming models
  - Poor language and runtime integration
- Good excuses against it - unfortunately
  - Too difficult, costly, error-prone, marginal benefit

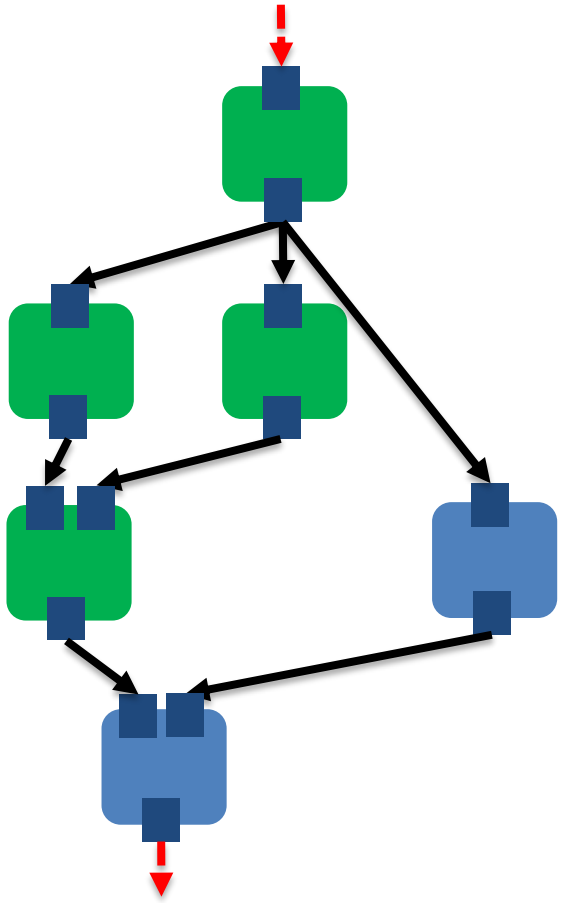
Our Goal

GPU parallel programming for (almost) everyone

- Radical simplification
  - No GPU experience required
  - Fast development
  - High performance comes automatically
  - Guaranteed memory safety
- Broad community
  - .NET in general: C#, F#, VB etc.
  - Based on Alea cuBase F# runtime

Alea Dataflow Programming Model

- Dataflow
  - Graph of operations
  - Data propagated through graph
- Reactive
  - Feed input in arbitrary intervals
  - Listen for asynchronous output

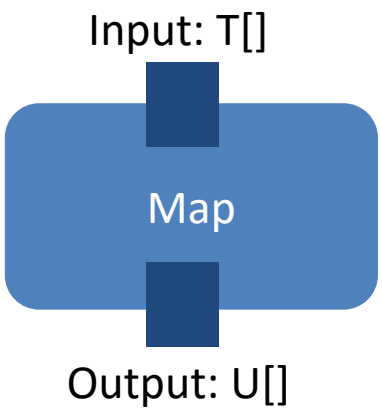


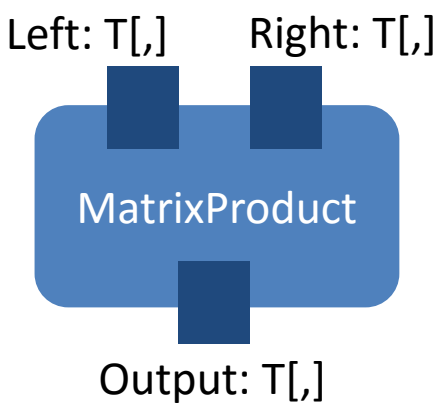
The Descriptive Power

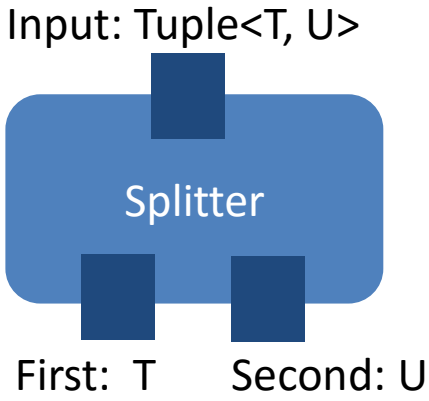
- Program is purely descriptive
  - What, not how
- Efficient execution behind the scenes
  - Vector-parallel operations
  - Stream operations on GPU
  - Minimize memory copying
  - Hybrid multi-platform scheduling
  - Tune degree of parallelization
  - …

Operation

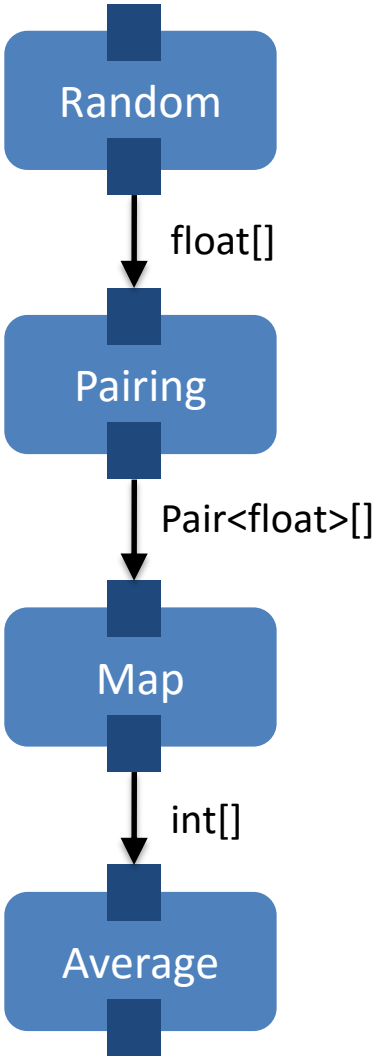
- Unit of calculation (typically vector-parallel)
- Input and output ports
- Port = stream of typed data
- Consumes input, produces output







Graph

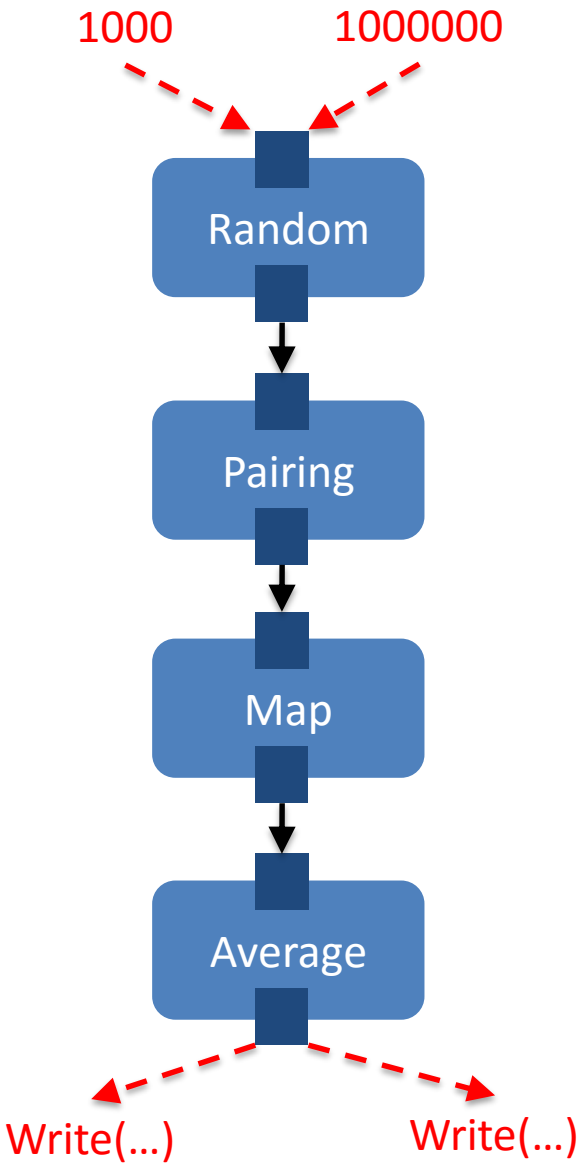



```
var randoms = new Random<float>(0, 1);
var coordinates = new Pairing<float>();
var inUnitCircle = new Map<Pair<float>, float>(
 p => p.Left * p.Left + p.Right * p.Right <= 1
      ? 1f : 0f
);
var average = new Average<float>();

randoms.Output.ConnectTo(coordinates.Input);
coordinates.Output.ConnectTo(inUnitCircle.Input);
inUnitCircle.Output.ConnectTo(average.Input);
```

Dataflow

- Send data to input port
- Receive from output port
- All asynchronous

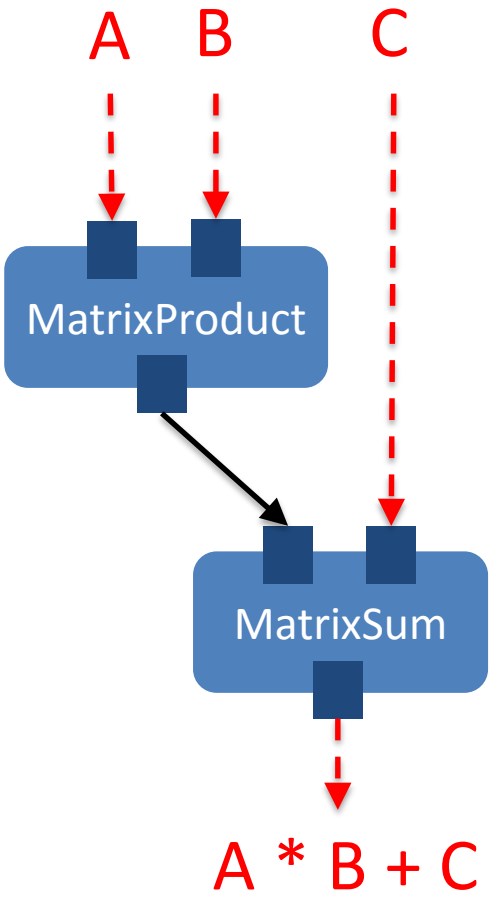
```
average.Output.OnReceive(x =>
       Console.WriteLine(4 * x));


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



Short Fluent Notation

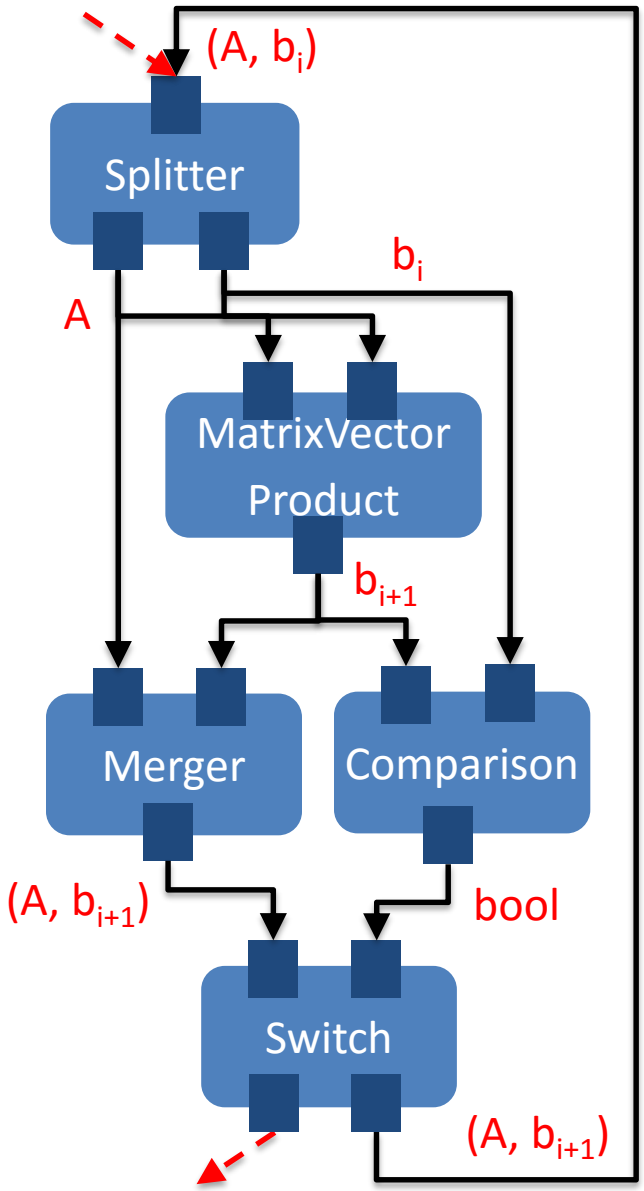
```
var randoms = new Random<float>(0, 1);
randoms
  .Pairing()
  .Map(p => p.Left * p.Left + p.Right * p.Right <= 1 ? 1f : 0f)
  .Average()
  .OnReceive(x => Console.WriteLine(4 * x));

randoms.Send(100);
randoms.Send(100000);
```

Algebraic Computation




```
var product = new MatrixProduct<float>();
var sum = new MatrixSum<float>();

product.Output.ConnectTo(sum.Left);

sum.Output.OnReceive(Console.WriteLine);

product.Left.Send(A);
product.Right.Send(B);
sum.Right.Send(C);
```

Iterative Computation

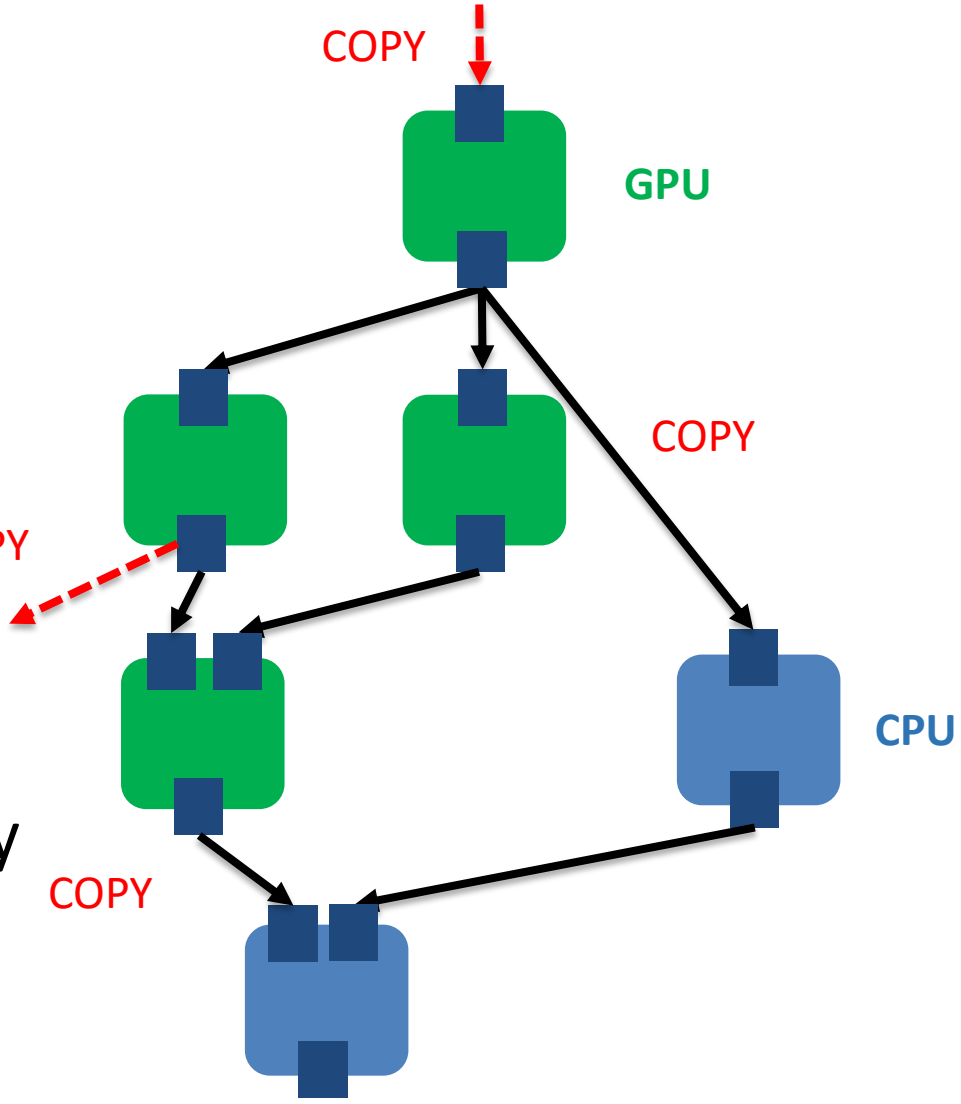


$$b_{i+1} = A \cdot b_i \text{ (until } b_{i+1} = b_i)$$

```
var source = new Splitter<double[,], double[]>();
var product =
  source.First.Multiply(source.Second);
var steady =
  product.Compare(source.Second, fun x y ->
                        Math.Abs(x - y) < 1E-6);
var next = source.First.Merge(product);
var branch = steady.Switch(next);
branch.False.ConnectTo(source.Input);

branch.True.OnReceive(Console.WriteLine)
source.Send(new Tuple<double[,], double>(A, b0));
```

Current Scheduler

- Operation implement GPU and/or CPU
- GPU operations combined to stream
- Memory copy only when needed
- Host scheduling with .NET TPL
- Automatic free memory management



Operation Catalogue

- Prefabricated generic operations
  - Switch, Merger, Splitter, Comparison
  - Map, Reduction, Average, Pairing
  - Random, MatrixProduct, MatrixSum, MatrixVectorProduct, VectorSum, ScalarProduct
  - More to come…
- Custom operations can be added
- Good performance
  - Nearly as fast as native C CUDA: overhead about 10%
    - Performance depends on operation implementation
    - Small overhead (cross-compilation, managed to unmanaged interop, scheduler)

Related Works

- Rx.NET / TPL Dataflow
  - Single input and output port
  - Not for GPU
- Xcelerit
  - Not reactive: single flow per graph
  - No generic operations with functors
- MSR PTasks / Dandelion
  - Synchronous receive, on C++, no generic operations
  - .NET LINQ integration (pull instead of push)
- Fastflow
  - Not reactive (sync run of the graph)
  - More low-level C++ tasks, no functors
- Nikola
  - Implicit dataflow described by functions
  - Limited set of operations

Conclusions

- Simple but powerful GPU parallelization in .NET
  - No low-level GPU artefacts
  - Fast and condensed problem formulation
  - Efficient and safe execution by the scheduler
- The descriptive paradigm is the key
  - Reactive makes it very general: cycles, infinite etc.
  - Practical suitability depends on operations
- Future directions
  - Advanced schedulers: multi GPUs, cluster, optimizations
  - Larger operation catalogue, optimized operations