# Radically Simplified GPU Programming in F# and .NET

Luc Bläser
Institute for Software, HSR Rapperswil

Oskar Knobel, Philipp Kramer
Joint Project with QuantAlea
Daniel Egloff, Xiang Zhang, Dany Fabian

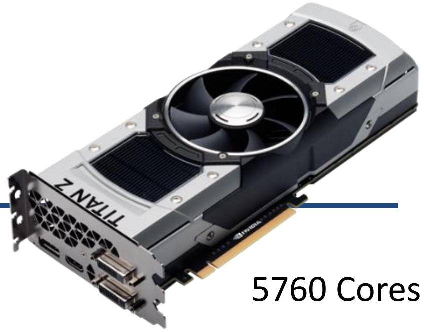
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT
FÜR
SOFTWARE

QuantAlea

F# User Meeting
3 Sep 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

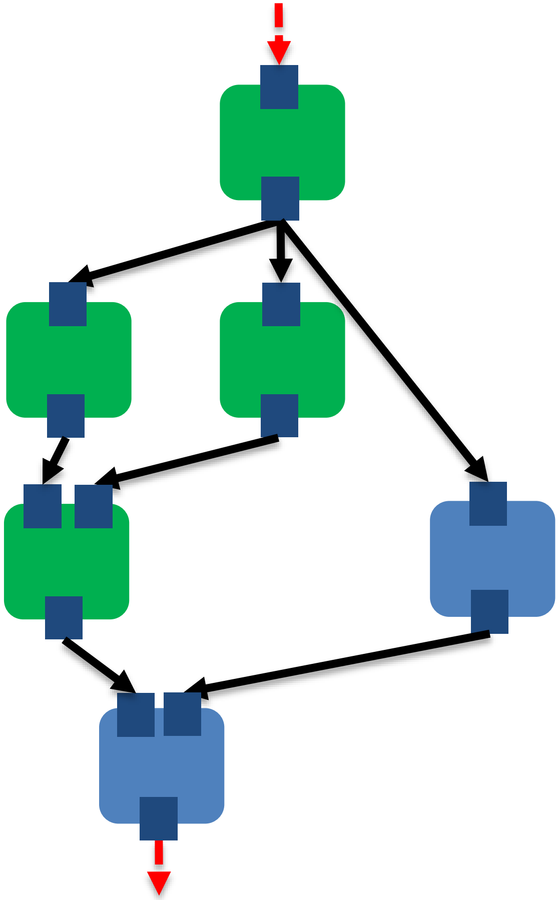
# Our Goal

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

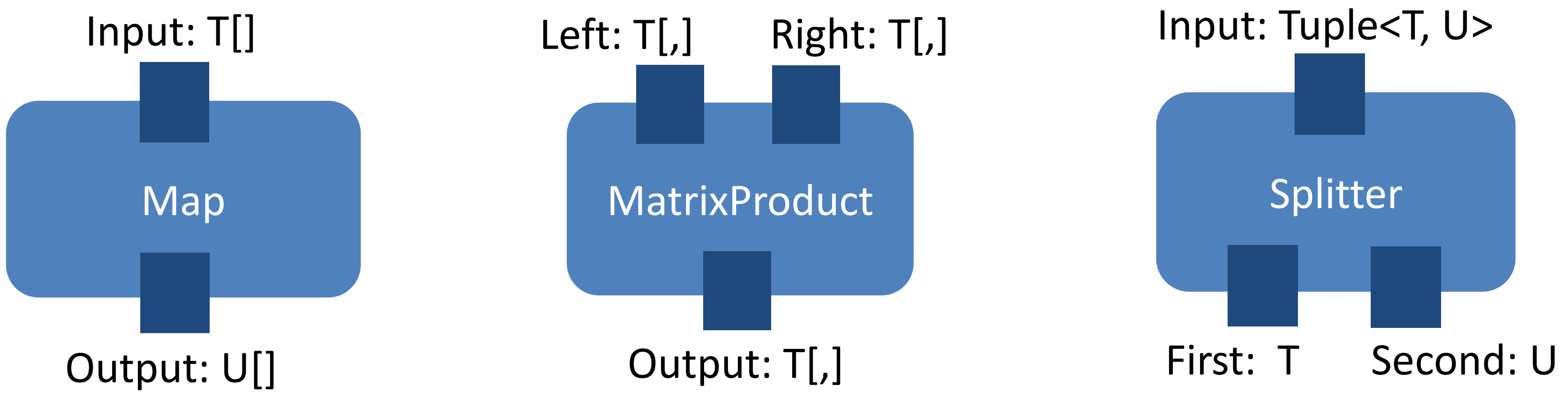


# The Descriptive Power

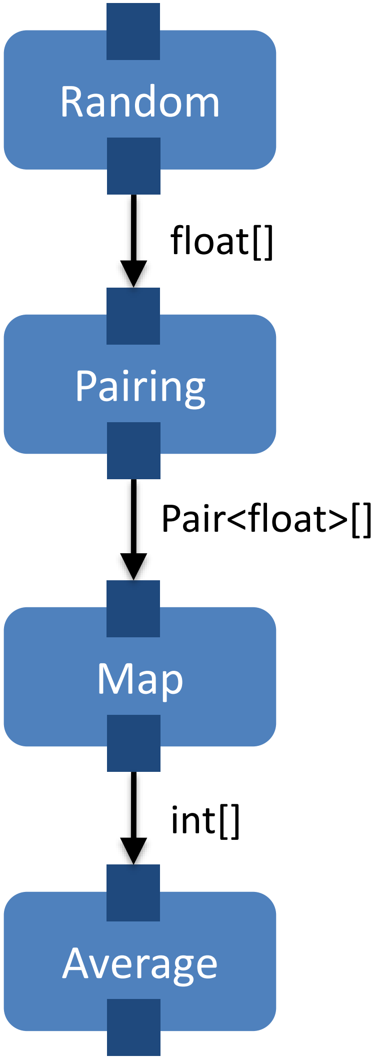
- Program is purely descriptive
  - What, not how
- Efficient execution behind the scenes
  - Vector-parallel operations
  - Stream operations on GPU
  - Minimize memory copying
  - Hybrid multi-platform scheduling
  - Tune degree of parallelization
  - …

# Operation

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



# Graph

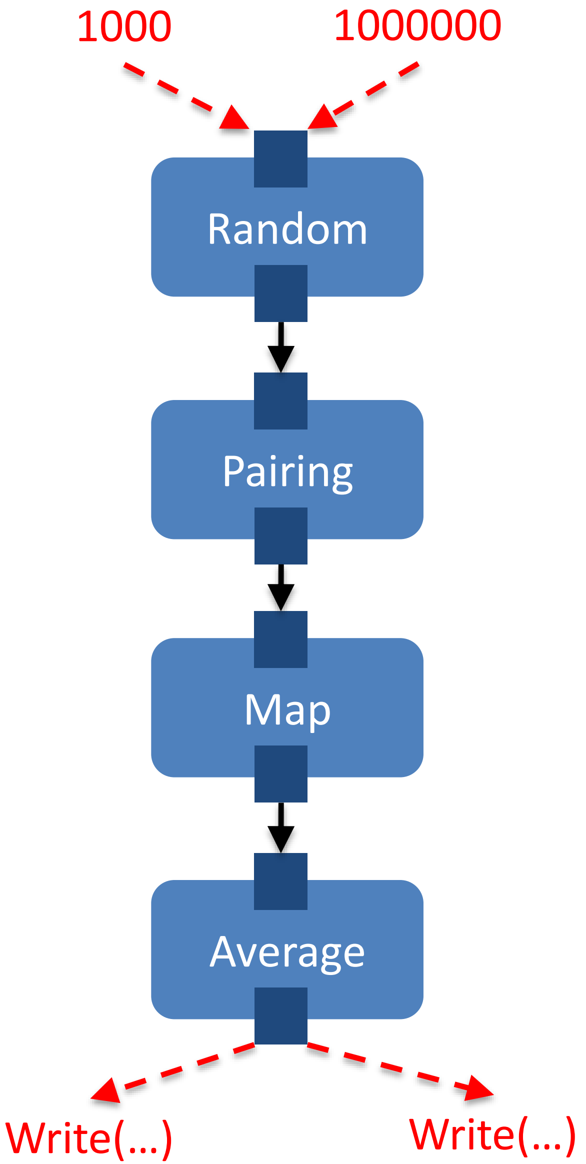


```
let randoms = Random<single>(0.f, 1.f)
let coordinates = Pairing<single>()
let inUnitCircle = Map<Pair<single>, single>
   (fun p -> if p.Left * p.Left +
                p.Right * p.Right <= 1.f
             then 1.f else 0.f)
let average = new Average<single>()

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

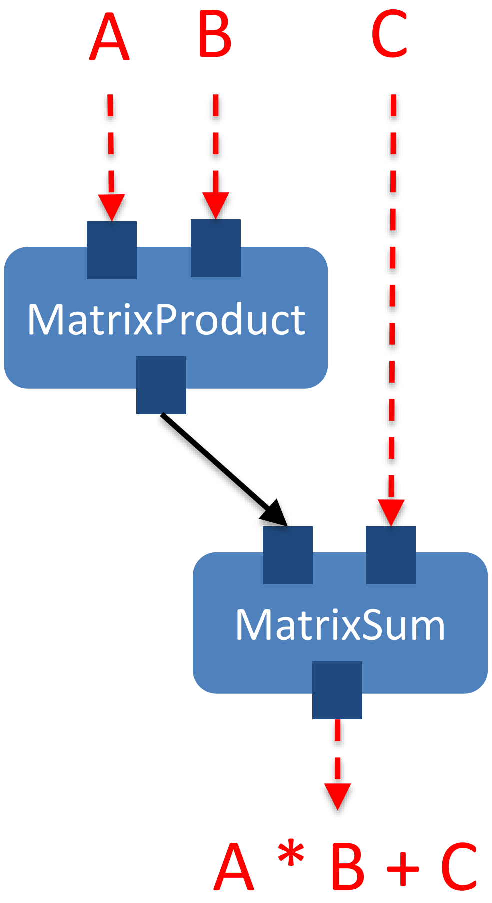
```
average.Output.OnReceive(fun x ->
          Console.WriteLine(4.f * x))

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



# Short Fluent Notation

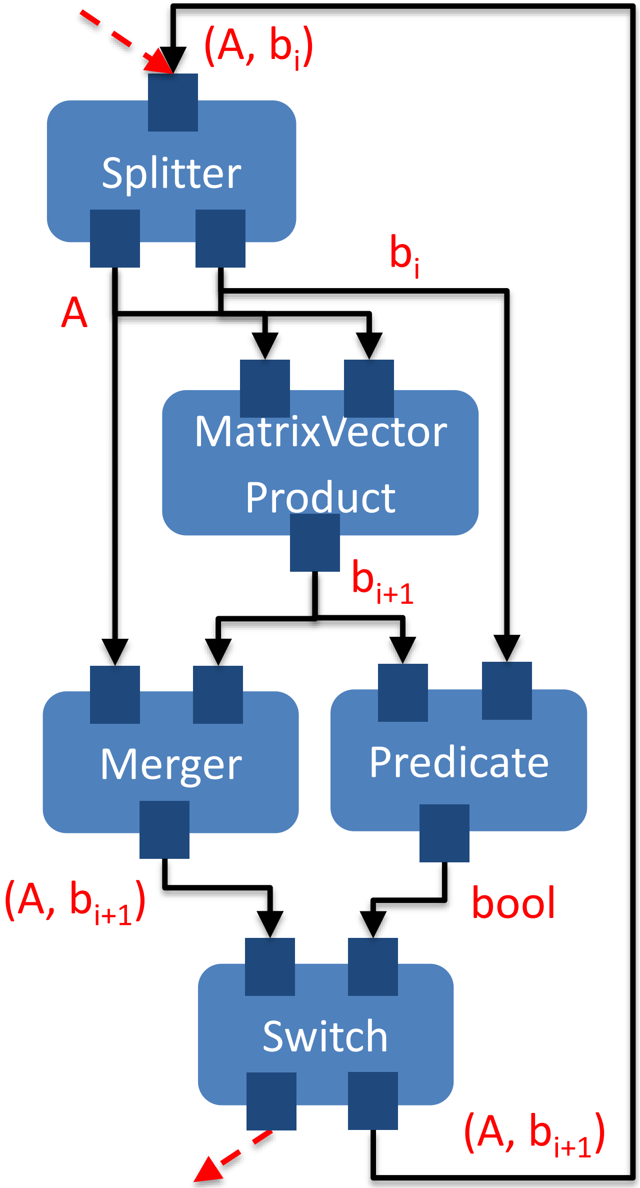
```
let randoms = new Random<single>(0.f, 1.f)
randoms
  .Pairing()
  .Map(fun p -> if p.Left * p.Left + p.Right * p.Right <= 1.f
                then 1.f else 0.f)
  .Average()
  .OnReceive(fun x -> Console.WriteLine(4.f * x))

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

# Algebraic Computation



```
let product = MatrixProduct<float>()
let sum = 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

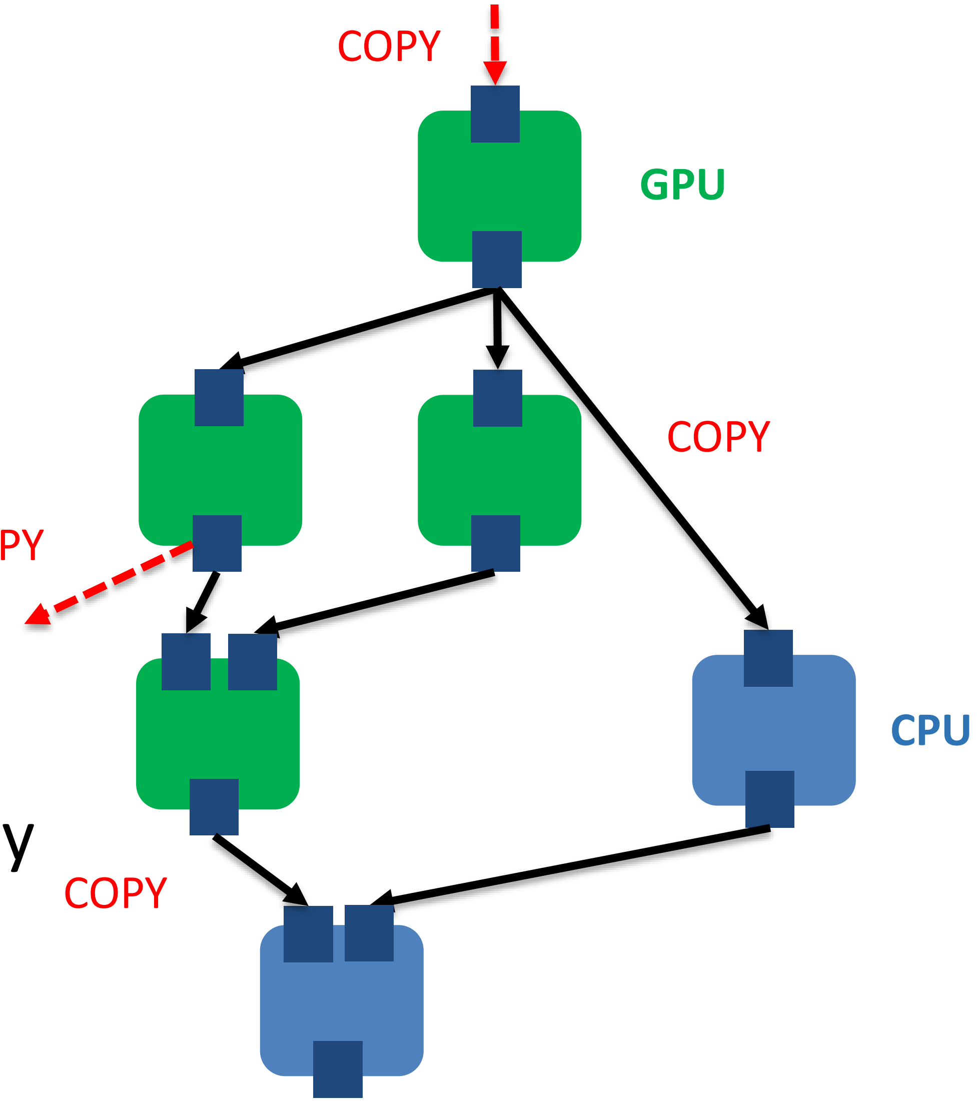


$$b_{i+1} = A \cdot b_i \text{ (until } b_{i+1} = b_i\text{)}$$

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

branch.True.OnReceive(Console.WriteLine)
source.Send((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, Predicate
  - Map, Reduce, 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 (margin 10-20%)

# 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

# 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