Leveraging the GPU on Spark

Tobias Polzer, Friedrich-Alexander University Erlangen-Nuremberg

Josef Adersberger, QAware GmbH

May 17, 2017

Contents

Motivation

- Initial motivation: Time series analysis in Chronix

Motivation

- Initial motivation: Time series analysis in Chronix
- Accelerating operations with high arithmetic intensity is "easy":
  - copy from Spark to accelerated native application
  - compute…
  - copy back results

Motivation

- What if intermediate results need to be exchanged?
  e.g. in outlier detection

Motivation

- What if intermediate results need to be exchanged?
  e.g. in outlier detection
- More generally: accelerate operations with low arithmetic intensity
- typically CPU $\leftrightarrow$ GPU slow, GPU RAM fast

Motivation

- What if intermediate results need to be exchanged?
  e.g. in outlier detection
- More generally: accelerate operations with low arithmetic intensity
- typically CPU $\leftrightarrow$ GPU slow, GPU RAM fast
- Can we just keep the data on the GPU all the time?

GPU $\leftrightarrow$ Java

- Project Sumatra aimed for deep integration into Hotspot. Didn't happen (project is "currently inactive").

GPU $\leftrightarrow$ Java

- Project Sumatra aimed for deep integration into Hotspot. Didn't happen (project is "currently inactive").
- OpenCL and CUDA are native APIs, interfacing via JNI possible but tedious
- There has yet to emerge a standard way of GPU acceleration for Java

GPU $\leftrightarrow$ Java

- Project Sumatra aimed for deep integration into Hotspot. Didn't happen (project is "currently inactive").
- OpenCL and CUDA are native APIs, interfacing via JNI possible but tedious
- There has yet to emerge a standard way of GPU acceleration for Java
- Many publications, but few publish code

Transpilers

There are two serious transpilers publicly available:

- Rootbeer (Java→CUDA)

Transpilers

There are two serious transpilers publicly available:

- Rootbeer (Java→CUDA)
- Aparapi (Java→OpenCL)

Transpilers

There are two serious transpilers publicly available:

- Rootbeer (Java→CUDA)
- Aparapi (Java→OpenCL)

Both could use some love...

jocl/jcuda

Near 1:1 wrappers around OpenCL/CUDA

- Very flexible in usage

jocl/jcuda

Near 1:1 wrappers around OpenCL/CUDA

- Very flexible in usage
- Direct OpenCL usage makes runtime code generation easy.

jocl/jcuda

Near 1:1 wrappers around OpenCL/CUDA

- Very flexible in usage
- Direct OpenCL usage makes runtime code generation easy.
- Buffer management with exceptions but without proper destructors is awkward.

jocl/jcuda

Near 1:1 wrappers around OpenCL/CUDA

- Very flexible in usage
- Direct OpenCL usage makes runtime code generation easy.
- Buffer management with exceptions but without proper destructors is awkward.

Currently the only reasonable choices.

CUDA vs. OpenCL

CUDA

- has a mature ecosystem
- needs separate compilation
- works only on Nvidia GPUs

OpenCL

- “works” on lots of devices (CPUs, GPUs, FPGAs, etc)
- supports JIT compilation of kernels (from C)
- most implementations are fragile/quirky

GPU $\leftrightarrow$ Spark

- Project Tungsten (theoretically)

GPU $\leftrightarrow$ Spark

- Project Tungsten (theoretically)
- IBM GPUEnabler (Tungsten prototype?)
  - looks promising

GPU $\leftrightarrow$ Spark

- Project Tungsten (theoretically)
- IBM GPUEnabler (Tungsten prototype?)
  - looks promising
  - but mostly undocumented
  - uses internal Spark APIs
  - had randomly failing tests
  - their example code is faster on the CPU

CLRDD

```
CLRDD[T](val wrapped: RDD[CLPartition[T]]) extends RDD[T]
```

- One `CLPartition` yields one context and an iterator of binary chunks
  - The context provides asynchronous methods on chunks

CLRDD

```
CLRDD[T](val wrapped: RDD[CLPartition[T]]) extends RDD[T]
```

- One `CLPartition` yields one context and an iterator of binary chunks
  - The context provides asynchronous methods on chunks
- Provides GPU functions on the RDD
- The user can choose caching on the GPU at runtime
- If data is not cached on the GPU, it is streamed as needed

Storage

- all useful operations on `CLRDD[T]` require a typeclass instance `CLType[T]`
- minimal definition includes OpenCL type, mapping to/from ByteBuffer storage
- optionally: OpenCL arithmetics
- macro generated instances for all primitve vector/tuple types

Operations

Operations are represented as composable case classes that can generate a kernel source:

```
case class MapReduceKernel[A, B](
  f: MapKernel[A,B],
  reduceBody: String,
  identity: String,
  cpu: Boolean,
  implicit val clA: CLType[A],
  implicit val clB: CLType[B]
) extends CLProgramSource {
  def generateSource(supply: Iterator[String])
        : Array[String] = ...
  ...
}
```

Functions on the GPU

High level functions that are implemented:

- One to one map functions (inplace/copying):
  ```
  crdd.map[Byte]("return x%2;")
  ```

Functions on the GPU

High level functions that are implemented:

- One to one map functions (inplace/copying):

```
crdd.map[Byte]("return x%2;")
```

- Simple reduction:

```
def sum(implicit num: Numeric[T]) : T = {
  val clT = implicitly[CLType[T]]
  reduce(MapReduceKernel(
    MapKernel.identity[T], // first map
    "return x+y;", // then reduce
    clT.zeroName, // string zero
    useCPU, // algorithm selection
    clT, clT // explicit typeclasses
  ), num.zero, ((x: T, y: T) => num.plus(x,y)))
}
```

Functions on the GPU

- Many to one sliding window map

```
def movingAverage(width: Int)(implicit clT: CLType[T])
//polymorphic return type, e.g.CLRDD[(Double,Double)]
  : CLRDD[clT.doubleCLInstance.elemType] = {
  val clRes = clT.doubleCLInstance
  sliding[clT.doubleCLInstance.elemType](
    width, 1, // width, stride
    s"""${clRes.clName} res = ${clRes.zeroName};
    for(int i=0; i<$width; ++i)
      res += convert_${clRes.clName}(GET(i));
      return res/$width;"""
  )//just scala things...
  (clT.doubleCLInstance.selfInstance,
    clT.doubleCLInstance.elemClassTag)
}
```

Benchmarking Setup

Workstation

- Spark local mode
- Intel i7-3770: 4 cores, 8 threads, ~20GiB/s
- Radeon HD 7950, ~200GiB/s

Benchmarking Setup

Workstation

- Spark local mode
- Intel i7-3770: 4 cores, 8 threads, ~20GiB/s
- Radeon HD 7950, ~200GiB/s

Cluster

- Spark standalone cluster mode
- 4 nodes, 40Gbit/s Infiniband interconnect
- two Xeon 2660v2: 20 cores, 40 threads, ~100GiB/s
- two K20m, ~400GiB/s

Benchmarks

- All benchmarks operate on `RDD[Double]`s.
- AMD's OpenCL implementation for the CPUs

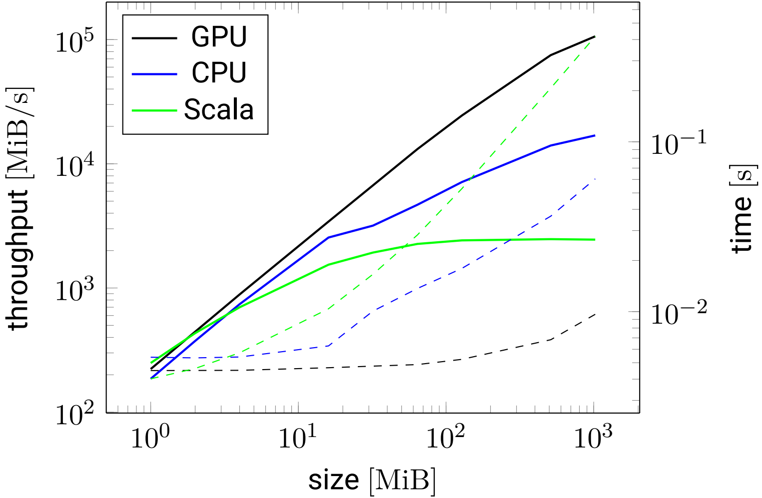
Benchmarks

- All benchmarks operate on `RDD[Double]`s.
- AMD's OpenCL implementation for the CPUs
- all data cached in RAM/graphics RAM before benchmarking

Benchmarks

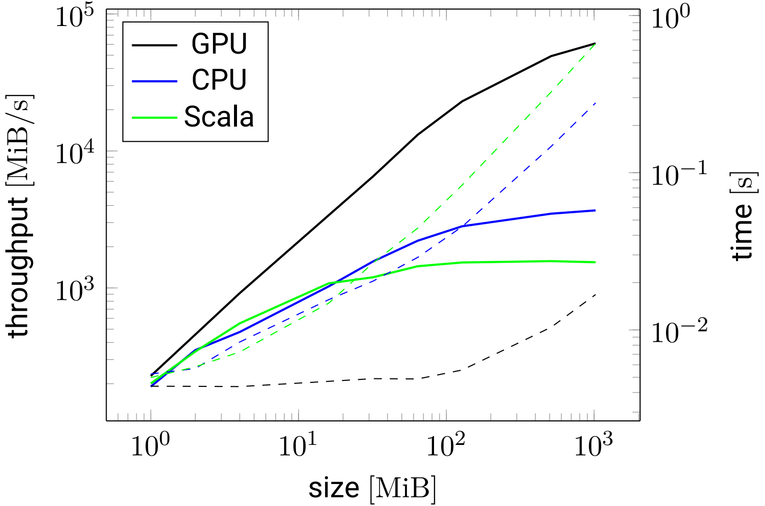
- All benchmarks operate on `RDD[Double]`s.
- AMD's OpenCL implementation for the CPUs
- all data cached in RAM/graphics RAM before benchmarking
- solid lines show throughput
- dashed lines show time to process one RDD

Workstation sum

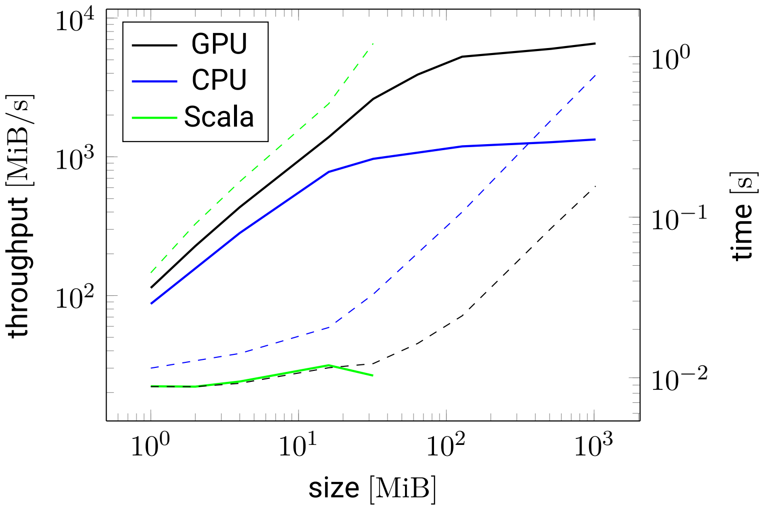


[1] "Scala" result with neither `rdd.sum()`, nor `rdd.reduce()`

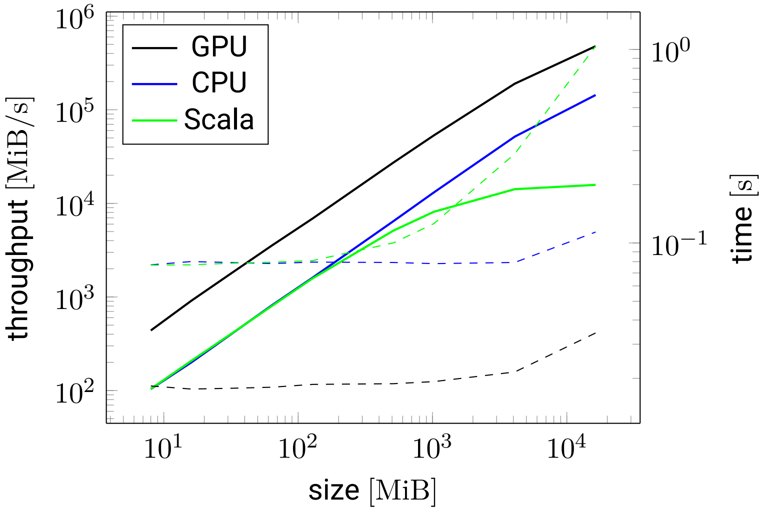
Workstation stats



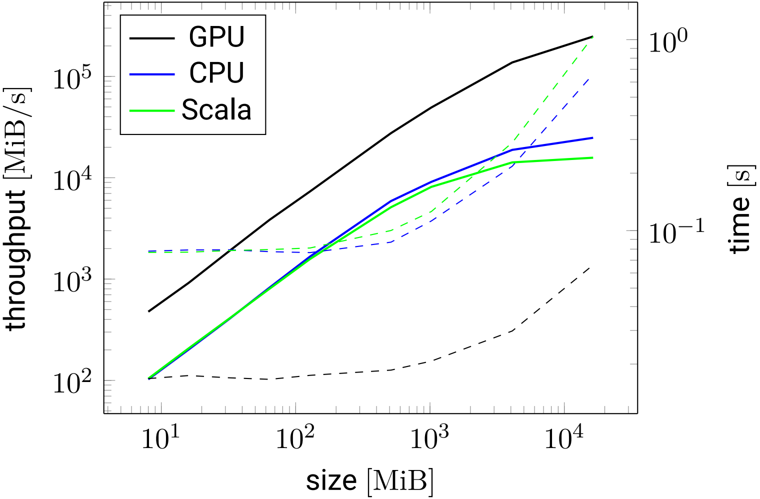
Workstation moving Average



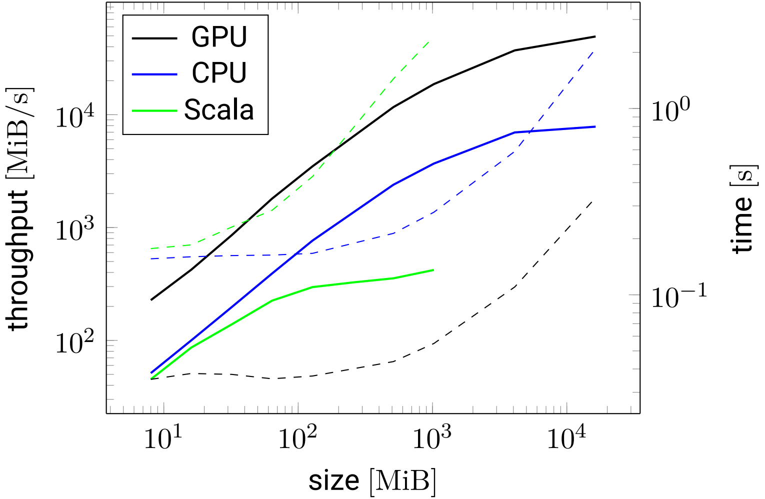
Cluster sum


GPU
CPU
Scala
throughput [MiB/s]
time [s]
size [MiB]
10^6
10^5
10^4
10^3
10^2
10^0
10^{-1}
10^1
10^2
10^3
10^4

Cluster stats



Cluster moving Average



Conclusions

- simple aggregations could be faster even without GPUs.

Conclusions

- simple aggregations could be faster even without GPUs.
- large speedups for big datasets in GPU memory

Conclusions

- simple aggregations could be faster even without GPUs.
- large speedups for big datasets in GPU memory
- implementation effort vs. plain Spark is a lot higher
  - fit data into GPU RAM
  - special GPU code?
  - debugging
  - deploying

The Way Forward

- Efficiently using GPUs (for arbitrary tasks) is a hard problem.

The Way Forward

- Efficiently using GPUs (for arbitrary tasks) is a hard problem.
- Builtins could benefit, especially with intelligent caching in GPU memory (typically scarce).

The Way Forward

- Efficiently using GPUs (for arbitrary tasks) is a hard problem.
- Builtins could benefit, especially with intelligent caching in GPU memory (typically scarce).
- Bytecode inspection for simple operations (see SPARK-14083)?
- Spark as a compiler?

Code

- Remember that complaint about not publishing code?

Code

- Remember that complaint about not publishing code?
- Fully functioning prototype implementation at: `https://github.com/TPolzer/spark-clrdd`.

Questions?