Leveraging the GPU on Spark

Tobias Polzer, Friedrich-Alexander University Erlangen-Nuremberg
Josef Adersberger, QAware GmbH

May 17, 2017
Contents

Motivation

Challenges

Prototype Architecture

Benchmarks

Conclusions

The Way Forward
Leveraging the GPU on Spark

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 ↔ 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 ↔ GPU slow, GPU RAM fast
- Can we just keep the data on the GPU all the time?
GPU ↔ Java

- Project Sumatra aimed for deep integration into Hotspot. Didn’t happen (project is “currently inactive”).
GPU ↔ 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
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.
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.
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
Leveraging the GPU on Spark

Challenges

GPU ↔ Spark

- Project Tungsten (theoretically)
Leveraging the GPU on Spark

Challenges

GPU ↔ Spark

- Project Tungsten (theoretically)
- IBM GPUEnabler (Tungsten prototype?)
  - looks promising
Leveraging the GPU on Spark

Challenges

- 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

$\text{CLRDD}[T](\text{val wrapped: RDD}[\text{CLPartition}[T]])$ extends $\text{RDD}[T]$

- One $\text{CLPartition}$ 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
Leveraging the GPU on Spark

Prototype Architecture

Storage

- All useful operations on \( \text{CLRDD}[T] \) require a typeclass instance \( \text{CLType}[T] \)
- Minimal definition includes OpenCL type, mapping to/from ByteBuffer storage
- Optionally: OpenCL arithmetics
- Macro generated instances for all primitive vector/tuple types
Operations

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

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

- Simple reduction:
  ```scala
  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

```scala
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 RDDs of Double.
- 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

The diagram shows the throughput and time for different sizes of data processed by GPU, CPU, and Scala. The throughput is measured in MiB/s and the size is in MiB. The time is on a logarithmic scale.
Workstation moving Average

![Graph showing throughput vs size for GPU, CPU, and Scala](image-url)
Cluster sum

The diagram illustrates the throughput and time for different sizes of data processing tasks using GPU, CPU, and Scala. The y-axis represents throughput in MiB/s, and the x-axis represents size in MiB. The graph shows how the throughput and time vary with different data sizes for each of the computational methods. The GPU method consistently shows the highest throughput and the shortest time across all data sizes compared to CPU and Scala.
Cluster stats

![Graph showing throughput vs. size for GPU, CPU, and Scala](image)

- **Throughput** [MiB/s] vs. **Size** [MiB]
- **Time** [s] on the right axis

Lines represent:
- **GPU**
- **CPU**
- **Scala**
Cluster moving Average

![Graph showing throughput and time for different sizes and processes (GPU, CPU, Scala) with logarithmic scales for throughput and time.](image)
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?

Fully functioning prototype implementation at: https://github.com/TPolzer/spark-clrdd.
Code

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