Master "Computational Science and Engineering" Belegarbeit: Verteilte GPGPU-Berechnungen mit Spark

Maximilian Knespel

Betreuer: Dipl.-Inf. Nico Hoffmann

GPU-Beschleuniger zum Hochleistungsrechnen

Platz 3 Titan XK7 in der Top 500 (Juni 2016):

- 18 688 AMD Opteron 6274 (16 Kerne) zu je 140 TFLOPS(DP)
- 18 688 Nvidia Tesla K20X zu je 1.3 TFLOPS(DP)

General Purpose Graphical Processing Units (GPGPU) im Vergleich zu Prozessoren:

\+ bessere Leistungsaufnahme pro GFlops

– nur für bestimmte Anwendungen geeignet

\+ hohe Speicherbandbreiten

– hohe Speicherlatenzen

\+ massiv parallel

– **zusätzliche Komplexität beim Programmieren**

MapReduce Programmiermodell

"MapReduce: Simplified Data Processing on Large Clusters" (2004) von Jeffrey Dean und Sanjay Ghemawat (Google Inc.)

- Für Cluster aus tausenden von kommerziellen PCs entwickelt
- MapReduce bezeichnet ein Programmiermodell und die dazugehörige Implementation
- Schlüssel/Wert-Paare auf die eine Abbildung und nachfolgend eine Reduktion aller Werte eines Schlüssels ausgeführt wird

$$\begin{aligned} \text{Map} &: (k, v) &&\mapsto [(l_1, x_1), \ldots, (l_{r_k}, x_{r_k})] \\ \text{Reduce} &: (l, [y_1, \ldots, y_{s_l}]) &&\mapsto [w_1, \ldots, w_{m_l}] \end{aligned}$$

- Programme in diesem funktionalen Stil werten automatisch von MapReduce parallelisiert

Spark Übersicht

Nachfolger von Hadoop MapReduce. Vorteile:

+ Abarbeitung im Arbeitsspeicher möglich
+ iterative Algorithmen schneller als Hadoop
+ mehr und komplexere Methoden
+ interaktive Konsole
+ Unterstützung für: Scala, Java, Python, ...
+ lokales Dateisystem, HDFS, AmazonS3, ... nutzbar

Resilient Distributed Datasets (RDDs)

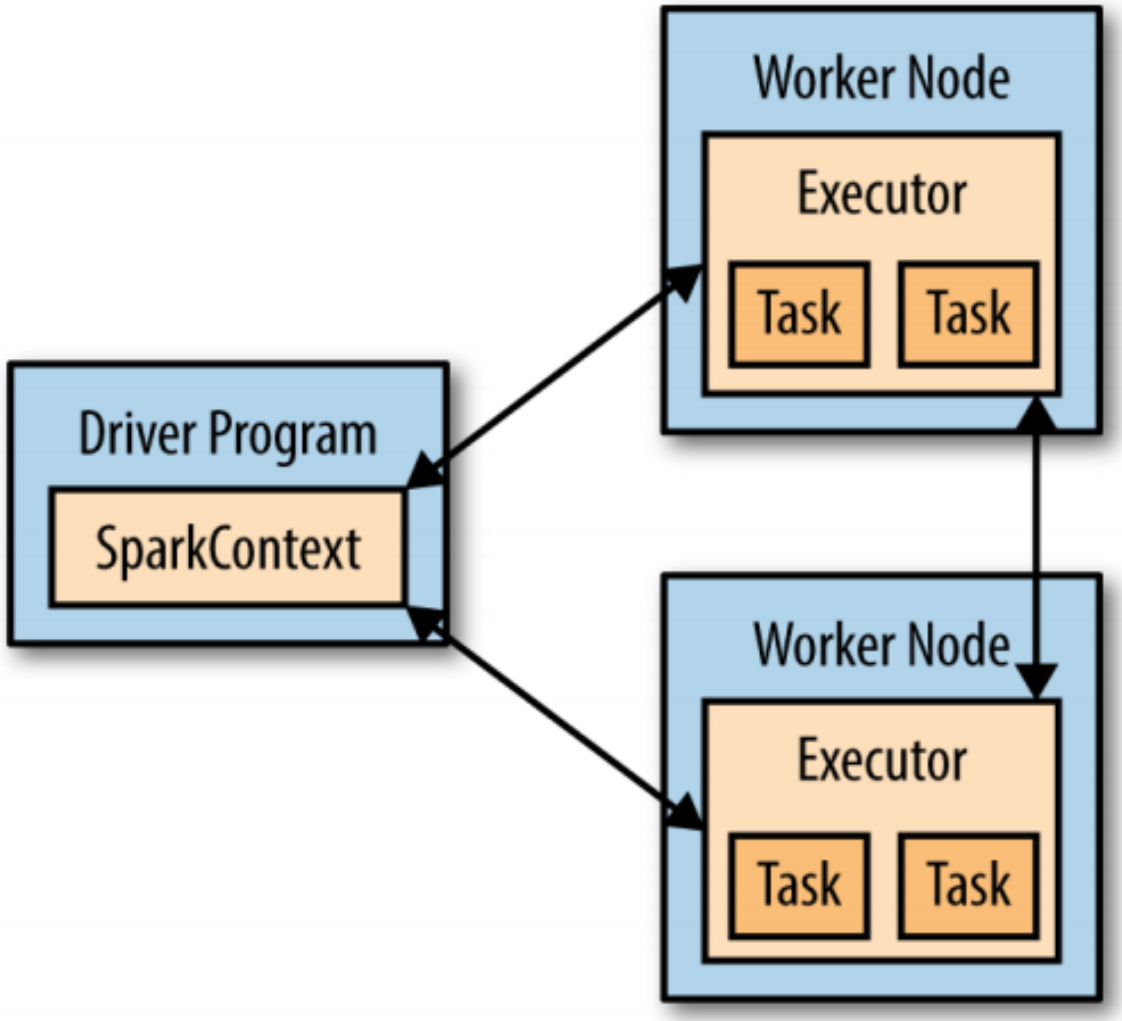
- zu bearbeitende Daten liegen in RDD-Objekten
- Resilience: opake Neuberechnung der Teildaten bei Absturz eines Knotens möglich
- Methoden zur Verarbeitung der Daten

```
def filter(f: (T) ⇒ Boolean): RDD[T]
```

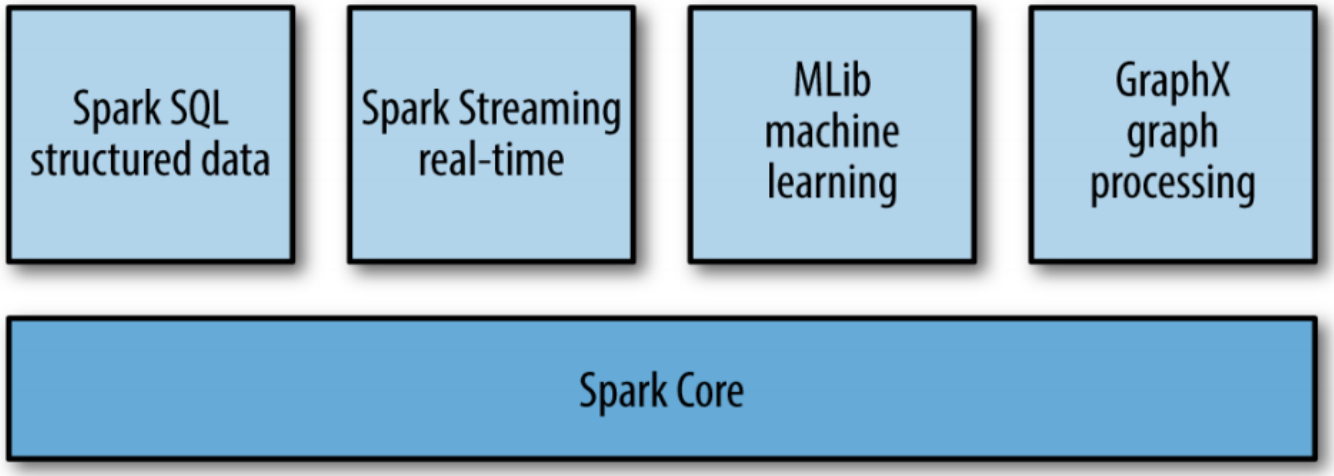
Return a new RDD containing only the elements that satisfy a predicate.

- ist aufgeteilt in Partitionen, welche auf verschiedenen Knoten berechnet werden können

Spark Funktionsweise

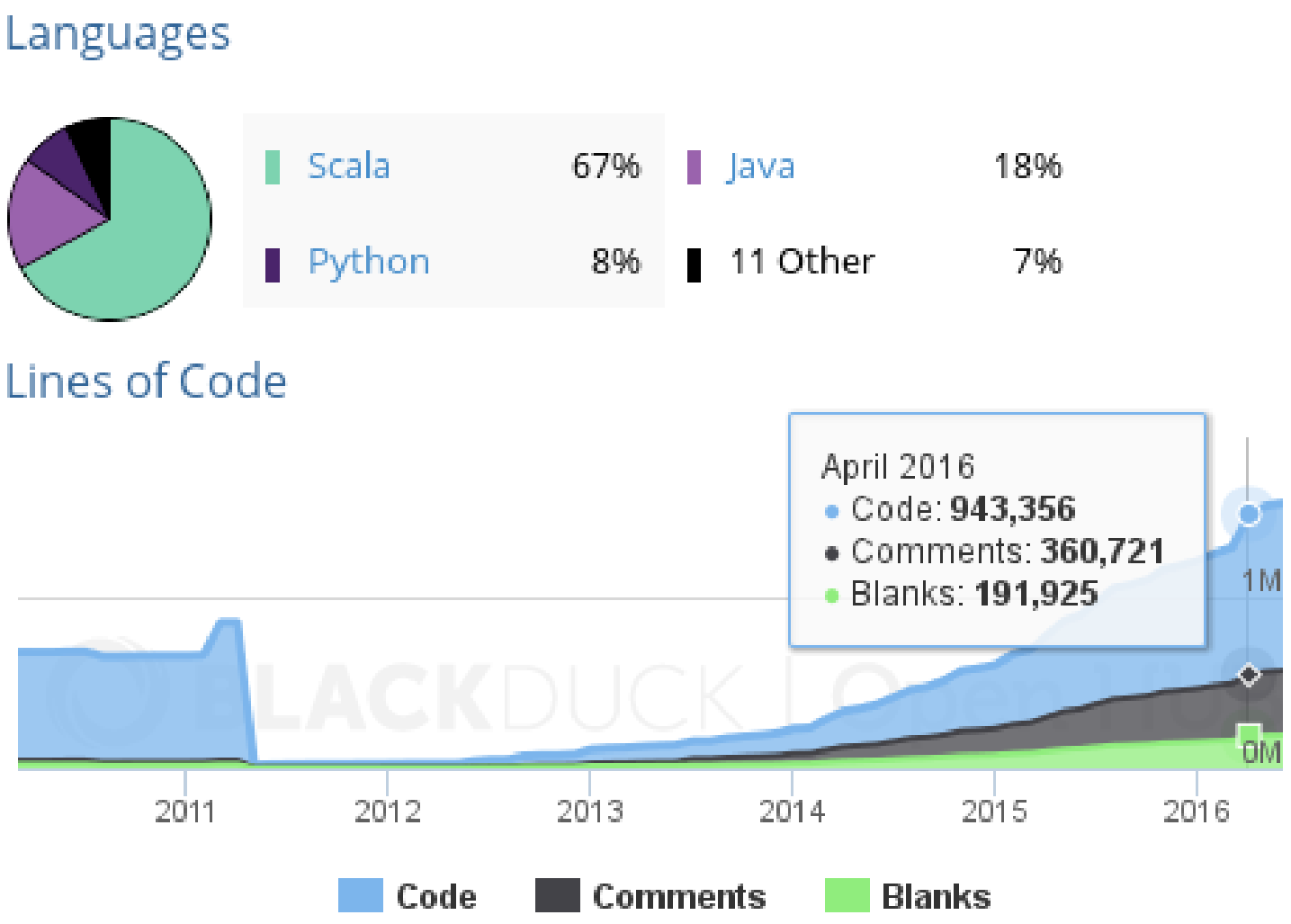


Spark Anwendungen



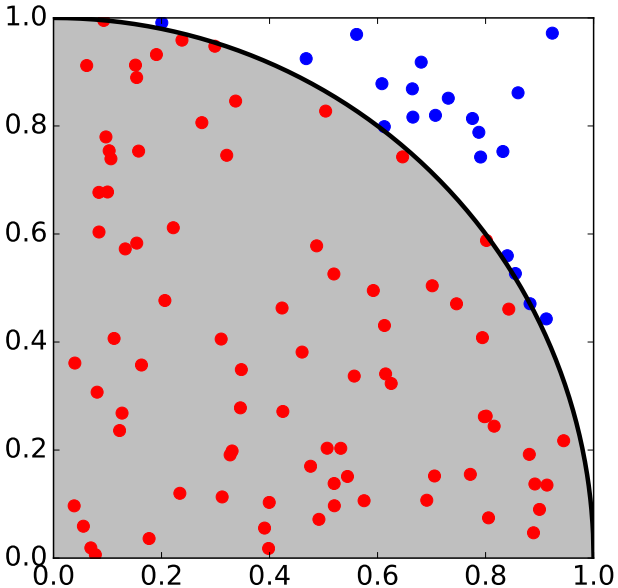
Spark auf Github

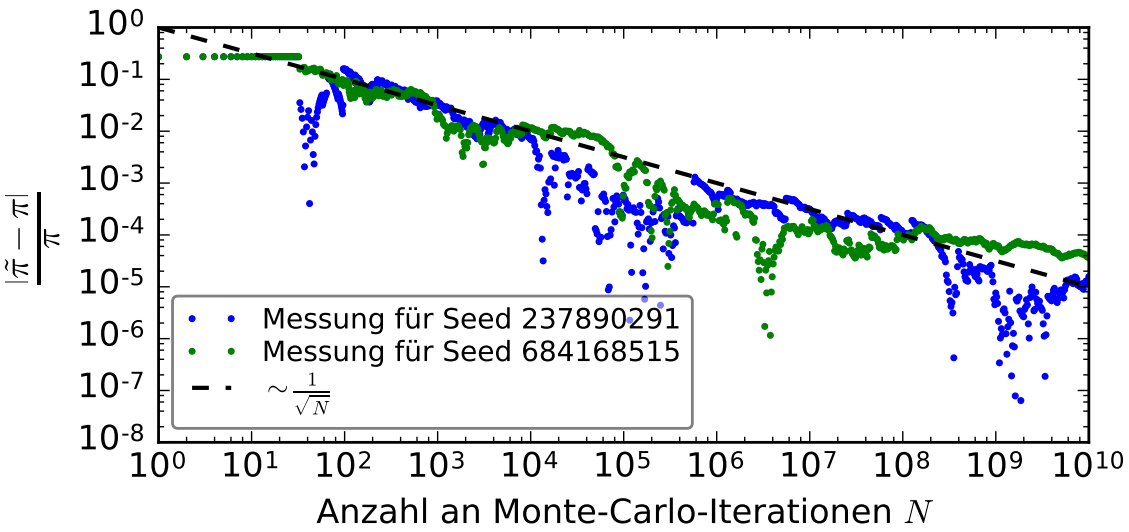
`github.com/apache/spark.gitmaster`



https://www.openhub.net/p/apache-spark

Monte-Carlo-Integration





Implementation Monte-Carlo-Pi-Berechnung

```
1  def calcQuarterPi (
2      nIterations : Long,
3      rSeed       : Long
4  ) : Double = {
5    var seed      = rSeed
6    var nHits     = 0L
7    val randMax   = 0x7FFFFFFF
8    val randMagic = 950706376L
9    var i = 0; while ( i < nIterations ) {
10     seed  = ( ( randMagic * seed ) % randMax ).toInt
11     val x = seed
12     seed  = ( ( randMagic * seed ) % randMax ).toInt
13     val y = seed
14     if ( 1L*x*x + 1L*y*y < 1L*randMax*randMax )
15         nHits += 1
16     i += 1
17   }
18   return nHits.toDouble / nIterations
19 }
```

Spark Monte-Carlo-Pi-Beispiel

```
1 "$SPARK_ROOT"/bin/spark-shell --master local[*]
```

```
1  val sparkConf    = new SparkConf().setAppName("Pi")
2  var sc           = new SparkContext( sparkConf )
3  val seed0        = 875414591
4  val nPartitions  = 100
5  val nIterations  = 1e8.toLong
6  val quarterPis   = sc.parallelize(1 to nPartitions).
7    map( iRank => ( {
8      val seed  = ( seed0 + ( iRank.toDouble /
       nPartitions * Integer.MAX_VALUE ).toLong ) %
       Integer.MAX_VALUE
9      calcQuarterPi( nIterations, seed ) ).cache
10 quarterPis.take(4).foreach( x => print( x + " " ) )
11 println( "pi = " + 4*quarterPis.reduce(_+_)/
       nPartitions )
```

Programmausgabe:

```
1 0.7852178 0.7852558 0.7855507 0.78554
2 pi = 3.1415955324
```

GPU-Programmierung mit Java/Scala

- OpenCL-Schnittsellen: JogAmp JOCL, JOCL, JavaCL
- OpenCL für Scala: ScalaCL, Firepile
- jCUDA
- Aparapi
- Rootbeer

Rootbeer

- 1000x starred auf github
- ≈ 30k Codezeilen hauptsächlich Java und ein wenig C
- leider kaum Aktivität auf Github seit Juni 2015
- "Rootbeer is pre-production beta. If Rootbeer works for you, please let me know."
- Unterstütze Java Features:
  - Arrays jeden Types und Dimension
  - beliebige (auch zyklische) Objektgraphen
  - innere / geschachtelte Klassen
  - dynamische Speicherallokation
  - Exceptions
  - ...
- Nicht unterstützt:
  - native Methoden
  - garbage collection
  - Reflections

Rootbeer Funktionsweise

1. Lese alle Felder aus benötigten Objekten in ein Java Byte Array
2. Sende Byte Array an GPU
3. Konvertiere Java-Bytecode mit Soot nach Jimple
4. Generiere Getter und Setter für alle Zugriffe
5. Java-Methoden werden in simple Device-Funktionen umgewandelt
6. Kompiliere den generierten CUDA-Code mit nvcc
7. Packe die modifizierte Klassen und den kompilierten nativen Code zu einer jar

Monte-Carlo-Pi mit Rootbeer Teil 1

```
1  import org.trifort.rootbeer.runtime.Kernel;
2  public class MonteCarloPiKernel implements Kernel {
3    private long[] mnHits;
4    private long   mnDiceRolls;
5    private long   mRandomSeed;
6    public MonteCarloPiKernel(
7      long[] rnHits,
8      long rnDiceRolls
9      long rRandomSeed,
10   ) {
11     mnHits      = rnHits;
12     mnDiceRolls = rnDiceRolls;
13     mRandomSeed = rRandomSeed;
14   }
15   public void gpuMethod() {
16     ...
```

Kernelaufruf in CUDA:

```
1  kernelMonteCarloPi<<<nBlocks,nThreadsPerBlock>>>(
       dpnInside, nIterationsPerThread, seed );
```

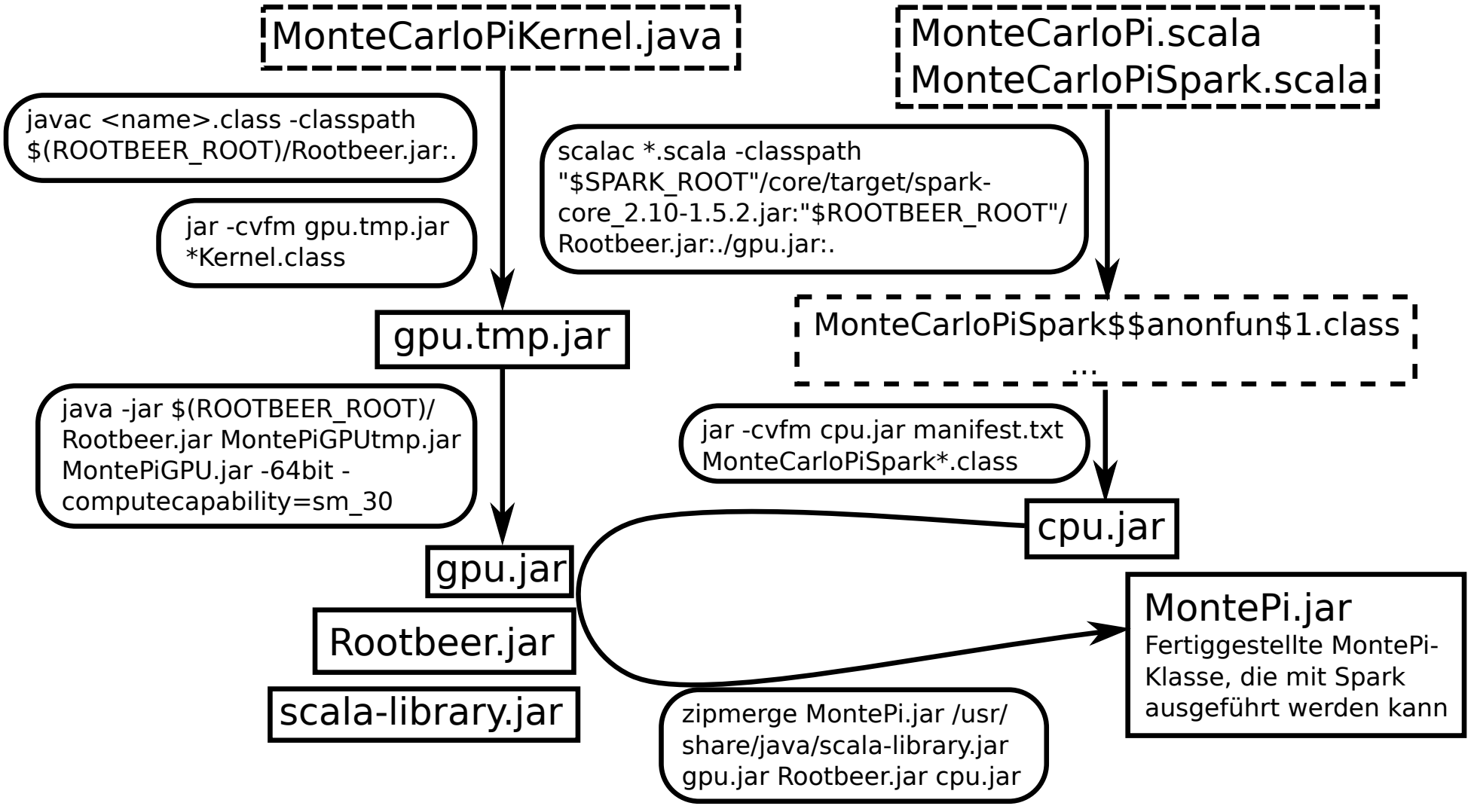
Monte-Carlo-Pi mit Rootbeer Teil 2

```
1   ...
2   public void gpuMethod() {
3     final int  randMax    = 0x7FFFFFFF;
4     final long randMagic  = 950706376L;
5     int dSeed             = (int) mRandomSeed;
6     final int dnDiceRolls = (int) mnDiceRolls;
7     long nHits            = 0L;
8     for ( int i = 0; i < dnDiceRolls; ++i ) {
9       dSeed   = (int)( (randMagic*dSeed) % randMax );
10      float x = (float) dSeed / randMax;
11      dSeed   = (int)( (randMagic*dSeed) % randMax );
12      float y = (float) dSeed / randMax;
13      if ( x*x + y*y < 1.0 )
14        nHits += 1;
15    }
16    mnHits[ RootbeerGpu.getThreadId() ] = nHits;
17  }
```

Monte-Carlo-Pi mit Rootbeer Teil 3

```
1  var mRootbeerContext  = new Rootbeer()
2  val mAvailableDevices = mRootbeerContext.getDevices()
3  val work = lnWorkPerKernel.zipWithIndex.map( x => {
       new MonteCarloPiKernel( lnHits(iGpu),
       lnIterations(iGpu), seed, nIterations ) } )
4  val context = mAvailableDevices.get( riDeviceToUse ).
       createContext( nBytesMemoryNeeded )
5  val thread_config = new ThreadConfig( threadsPerBlock
       , 1, 1, nBlocks, 1, work.size );
6  context.setThreadConfig( thread_config )
7  context.setKernel( work.get(0) )
8  context.setUsingHandles( true )
9  context.buildState()
10 val runWaitEvent = context.runAsync( work )
11 context.run( work );
```

Kompilation

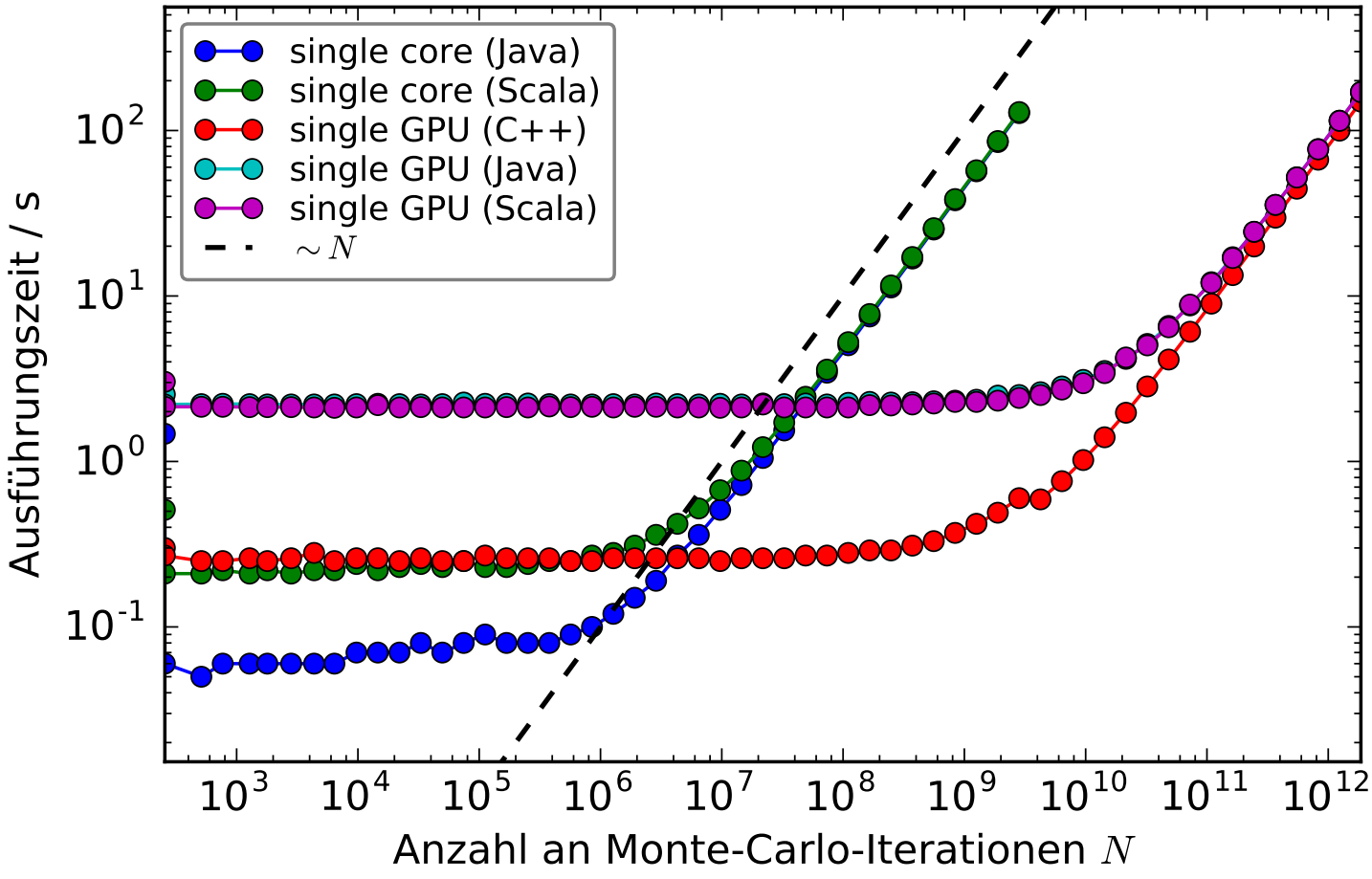


Bemerkungen und Hinweise

- Nur Java 6 ist von Rootbeer offiziell unterstützt, Java 7 geht aber großteils, nicht aber Java 8
- nur bis GCC 4.9 unterstützt
- die Option `-computecapability=sm_30` muss angegeben werden, da seit CUDA 7.0 die Standardarchitektur `compute_12` nicht mehr unterstützt wird (korrekt nun `sm_12`)
- Automatische Kernel-Konfiguration hat einen Bug, der standardmäßig immer so viele Kernel startet wie gleichzeitig auf einem Shared Multiprozessor laufen können.
- Bug wo Rootbeer hat standardmäßig versucht den gesamten freien Speicher minus den benötigten zu alloziieren versuchte
- die ausführbare jar für Spark darf kein Leerzeichen im Pfad enthalten:
- Zu kompilierende jar an Rootbeer >muss< auf .jar enden
- Profiling mit NVIDIA nvvp möglich. (Executable: java, Argumente: `-jar ./MontePi.jar`)

Leistungsanalyse: 1 CPU-Kern / GPU auf Taurus gpu2

```
salloc -p gpu2-interactive --nodes=1 --ntasks-per-
    node=1 --cpus-per-task=1 --gres=gpu:1 --time
    =02:00:00
```



Spark auf Taurus über Slurm starten

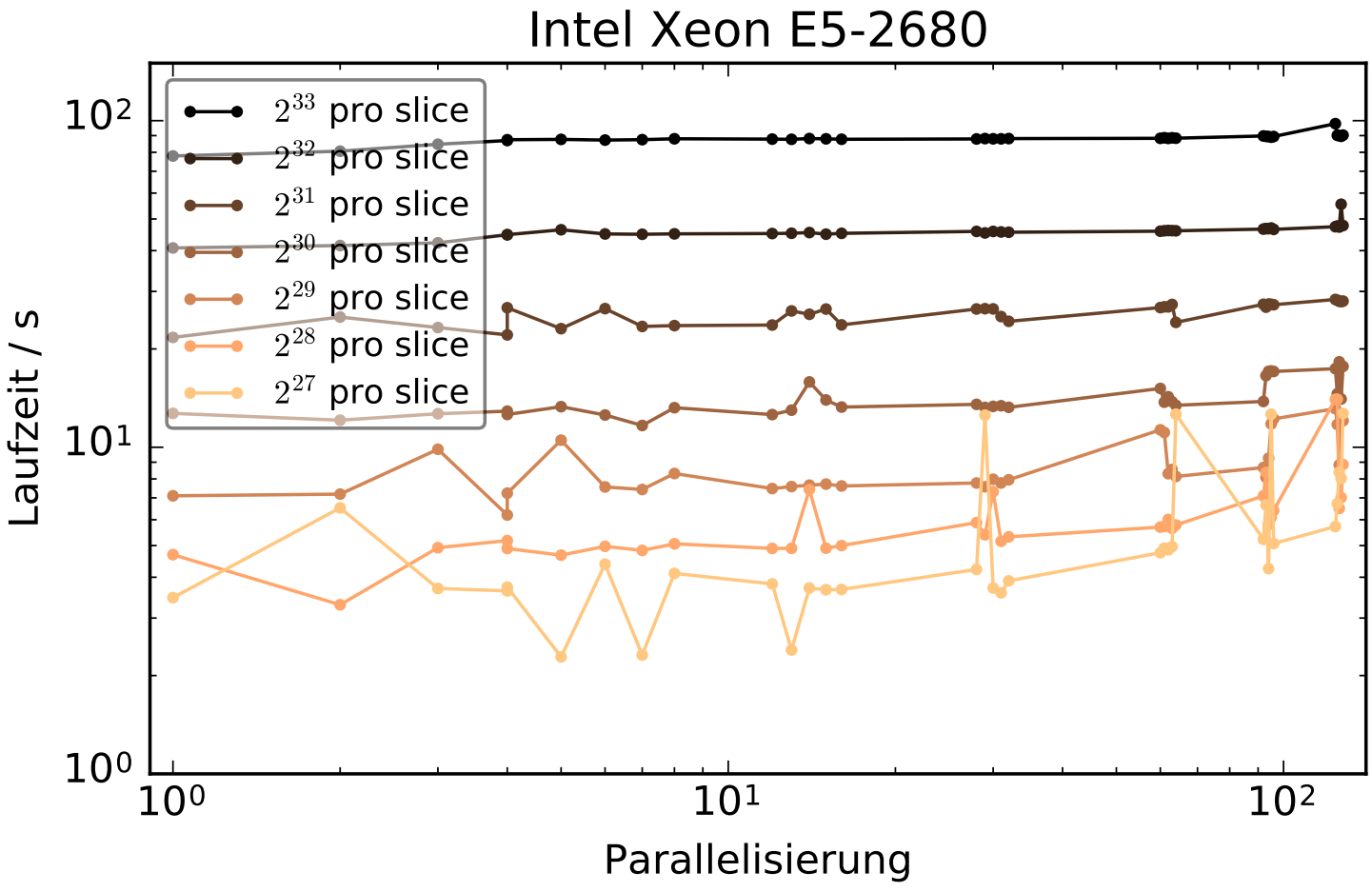
```
1 function startSpark() {
2   jobid=$(sbatch "$@" --output="$SPARK_LOGS/%j.out"
      --error="$SPARK_LOGS/%j.err" $HOME/scaromare/
      start_spark_slurm.sh)
3   # wait for job to start
4   export MASTER_ADDRESS=$(cat ~/spark/logs/${jobid}
      _spark_master)
5 }
```

```
1 startSpark --time=04:00:00 --nodes=$((nodes+1)) --
    partition=gpu2 --gres=gpu:$gpusPerNode --cpus-per
    -task=$coresPerNode
2 ~/spark-1.5.2-bin-hadoop2.6/bin/spark-submit --master
     $MASTER_ADDRESS ~/scaromare/MontePi/multiNode/
    multiGpu/scala/MontePi.jar $arguments
```

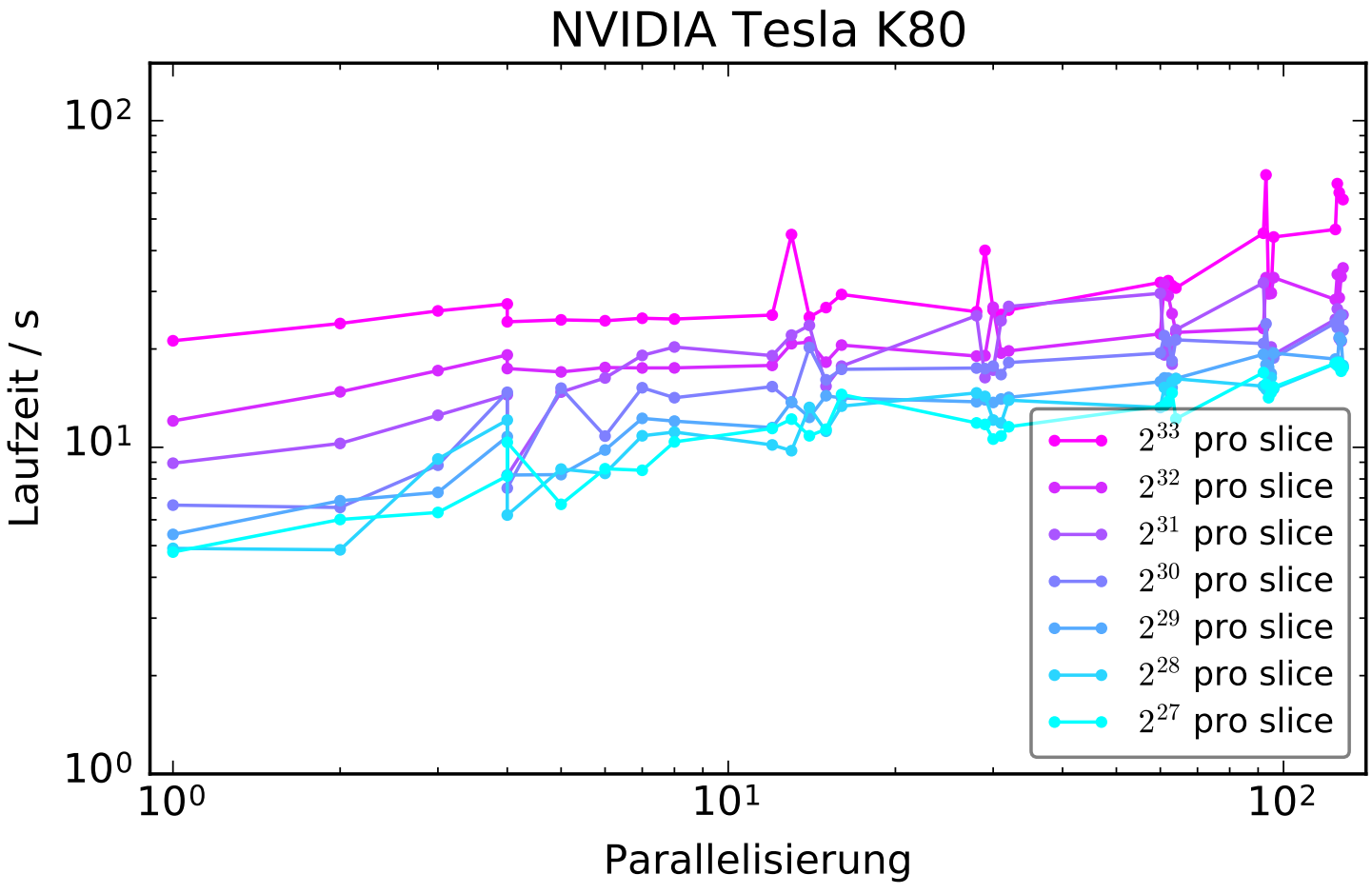
start_spark_slurm.sh

```
1  if [ "$1" != 'sran' ]; then
2    script=/scratch/$USER/${SLURM_JOBID}_$(basename "$0")
3    cp "$0" "$script"
4    export SPARK_DAEMON_MEMORY=$(( $SLURM_MEM_PER_CPU *
       $SLURM_CPUS_PER_TASK / 2 ))m
5    export SPARK_WORKER_CORES=$SLURM_CPUS_PER_TASK
6    srun "$script" 'sran' "$@"
7  else
8    module load scala/2.10.4 java/jdk1.7.0_25 cuda/7.0.28
9    if [ $SLURM_PROCID -eq 0 ]; then # if master start driver
10     . "$SPARK_ROOT/sbin/spark-config.sh"
11     . "$SPARK_ROOT/bin/load-spark-env.sh"
12     "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.
       master.Master --ip $(hostname) --port $SPARK_MASTER_PORT --
       webui-port $SPARK_MASTER_WEBUI_PORT
13   else
14     # convert "host20[39-40]" to "host2039"
15     MASTER_NODE=spark://$(scontrol show hostname
       $SLURM_NODELIST | head -n 1):7077
16     "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.
       worker.Worker $MASTER_NODE
17   fi
18 fi
```

Benchmark Spark auf CPU



Benchmark Spark mit Rootbeer (alte Ergebnisse)



Zusammenfassung

- Kombination aus GPGPU mittels Rootbeer und Spark ausgetestet
- Mehrere Bugfixes für Rootbeer geschrieben

Ausblick:

- Codereview von Rootbeer oder andere GPU-API ist nötig
- heterogene Berechnungen auf CPU + GPU
- Erweiterung von Rootbeer um neue Features wie NVIDIA NVLink
- Implementation direkt in Spark würde z.B. cache/persist auf GPUs erlauben, um Host-GPU-Transfers zu sparen