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

Maximilian Knespel

Betreuer: Dipl.-Inf. Nico Hoffmann

GPU-Beschleuniger zum Hochleistungsrechnen

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

- \+ bessere Leistungsaufnahme pro GFlops
- – nur für bestimmte Anwendungen geeignet
- – **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)

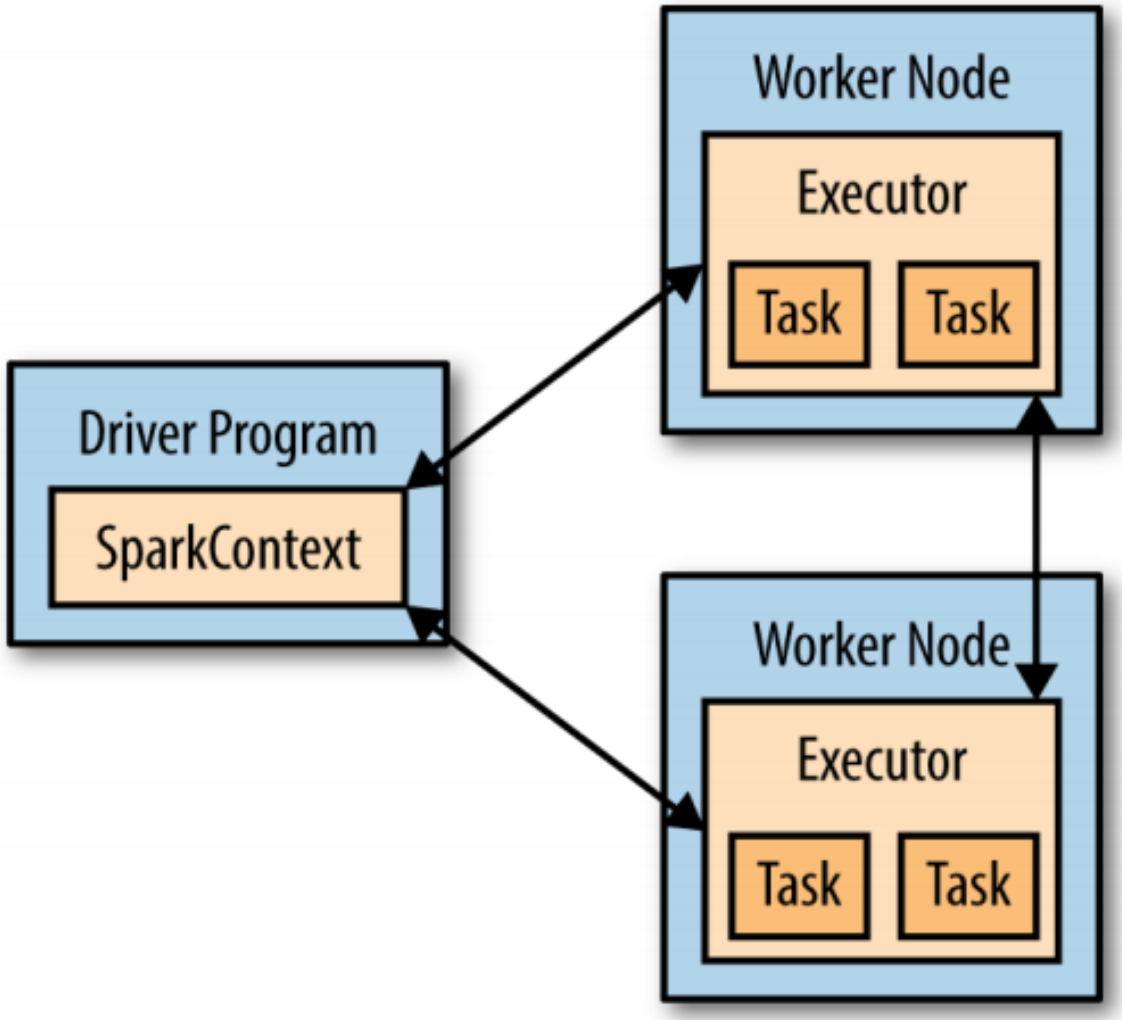
- zu bearbeitende Daten liegen in RDD-Objekten
- Resilience: opake Neuberechnung der Teildaten bei Absturz eines Knotens möglich
- Methoden zur Verarbeitung der Daten

```
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 auf Taurus über Slurm starten

```
1 sbatch --output="$SPARK_LOGS/%j.out" --error="
      $SPARK_LOGS/%j.err" $HOME/scaromare/
      start_spark_slurm.sh --time=04:00:00 --nodes=$((
      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 ...
```

start_spark_slurm.sh:

```
1 export SPARK_WORKER_CORES=$SLURM_CPUS_PER_TASK
2 if [ $SLURM_PROCID -eq 0 ]; then # if master start driver
3   "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.master.
      Master --ip $(hostname) --port $SPARK_MASTER_PORT --webui-
      port $SPARK_MASTER_WEBUI_PORT
4 else
5   "$SPARK_ROOT/bin/spark-class" org.apache.spark.deploy.worker.
      Worker $MASTER_NODE
6 fi
```

Spark 1.5.2

Spark Master at spark://taurusi6456:7077

URL: spark://taurusi6456:7077
REST URL: spark://taurusi6456:6066 *(cluster mode)*
Alive Workers: 1
Cores in use: 1 Total, 0 Used
Memory in use: 61.7 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160707020155-172.24.46.161-58039	172.24.46.161:58039	ALIVE	1 (0 Used)	61.7 GB (0.0 B Used)

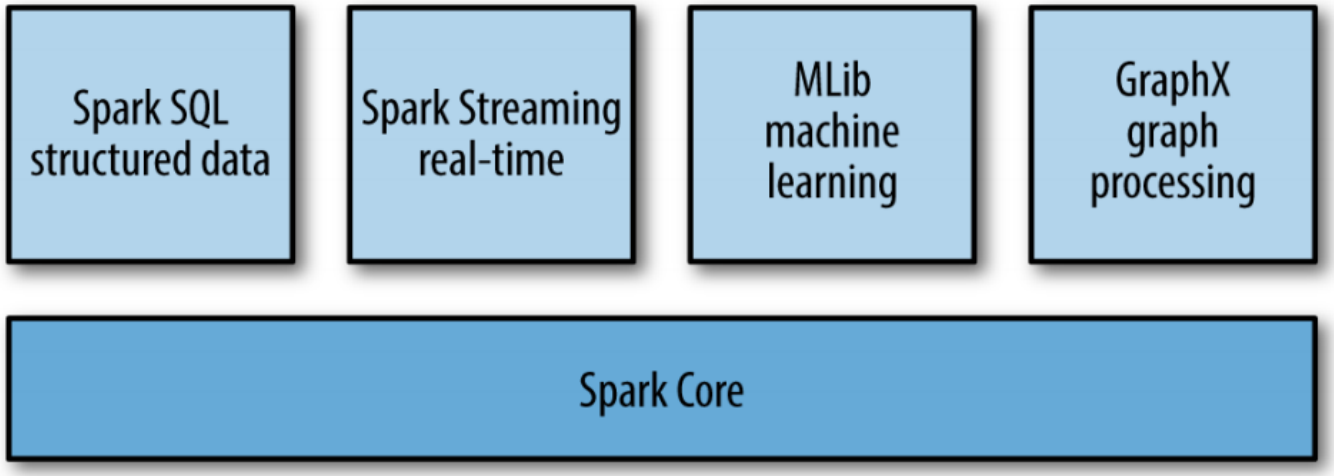
Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

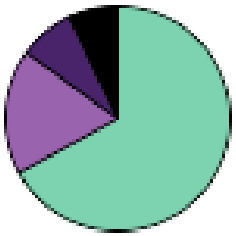
Spark Anwendungen



Spark auf Github

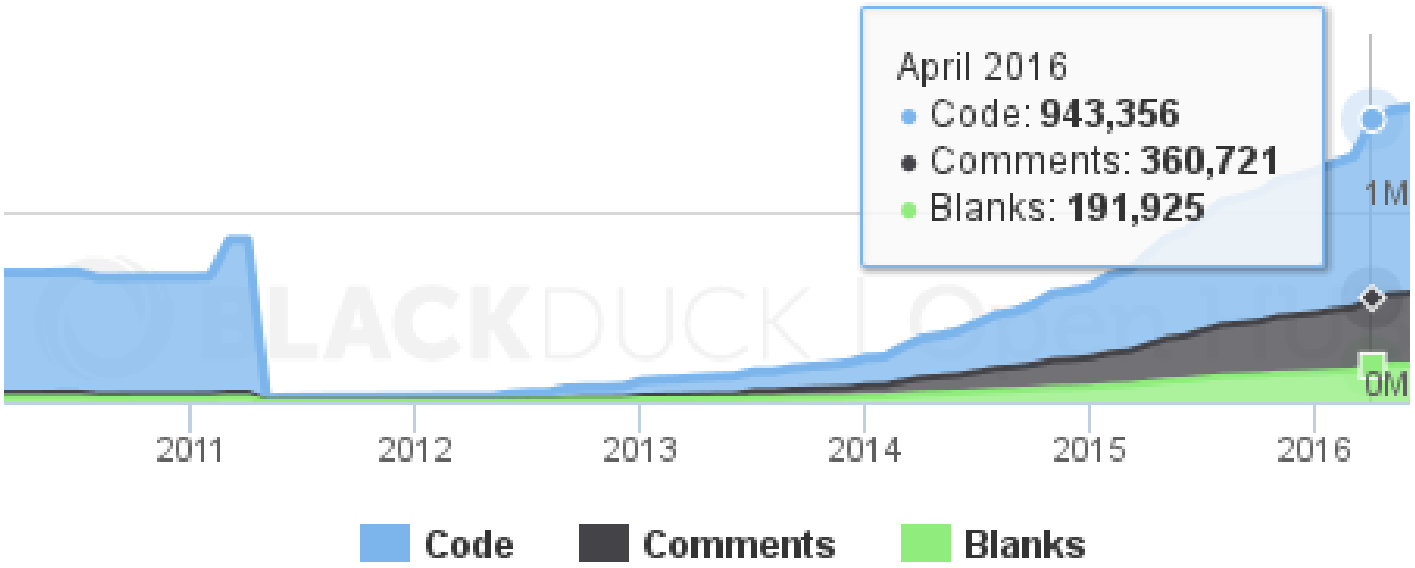
`github.com/apache/spark.gitmaster`

Languages



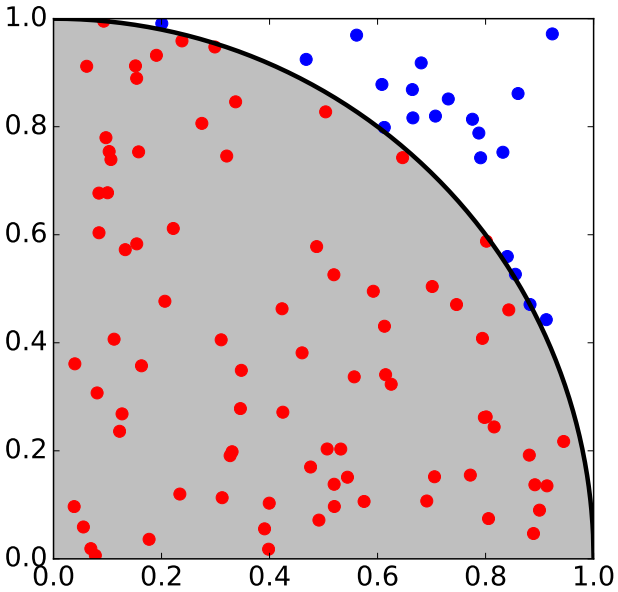
Scala	67%	Java	18%
Python	8%	11 Other	7%

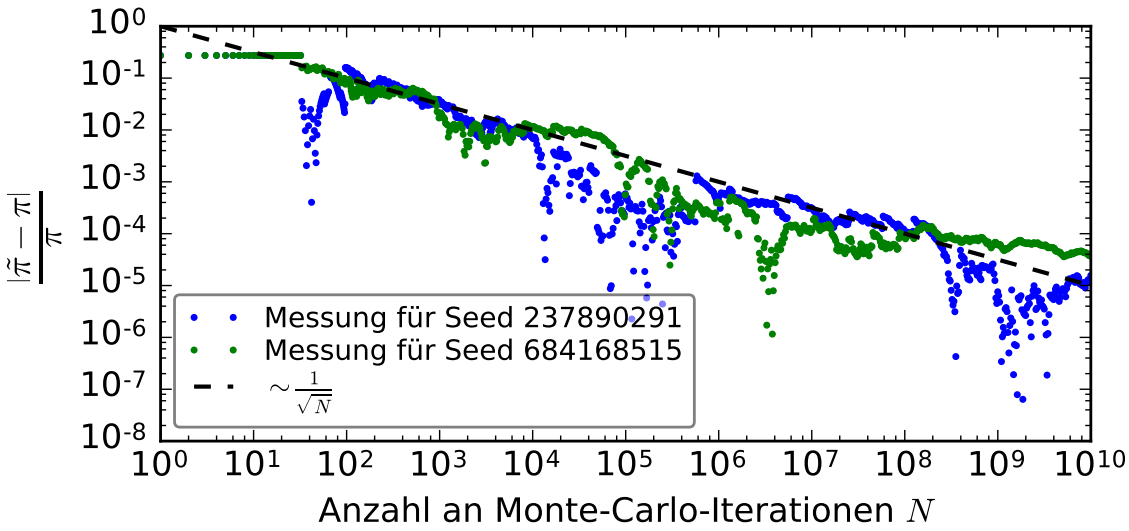
Lines of Code



`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

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

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

Programmausgabe:

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

GPU-Programmierung mit Java/Scala

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

Rootbeer

- ≈ 30k Codezeilen hauptsächlich Java und ein wenig C
- leider kaum Aktivität auf Github seit Juni 2015
- 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. Der Anwender implementiert das Kernel-Interface, insbesondere `gpuMethod`
2. Lese alle Felder aus benötigten Objekten in ein Java Byte Array
3. Sende Byte Array an GPU
4. Konvertiere Java-Bytecode mit Soot nach Jimple
5. Generiere Getter und Setter für alle Zugriffe
6. Java-Methoden werden in simple Device-Funktionen umgewandelt
7. Kompiliere den generierten CUDA-Code mit nvcc
8. Packe die modifizierte Klassen und den kompilierten nativen Code zu einer jar
9. Bei der Ausführung entpacke CUDA-Binaries in temporäres Verzeichnis und führe sie aus

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 );
```

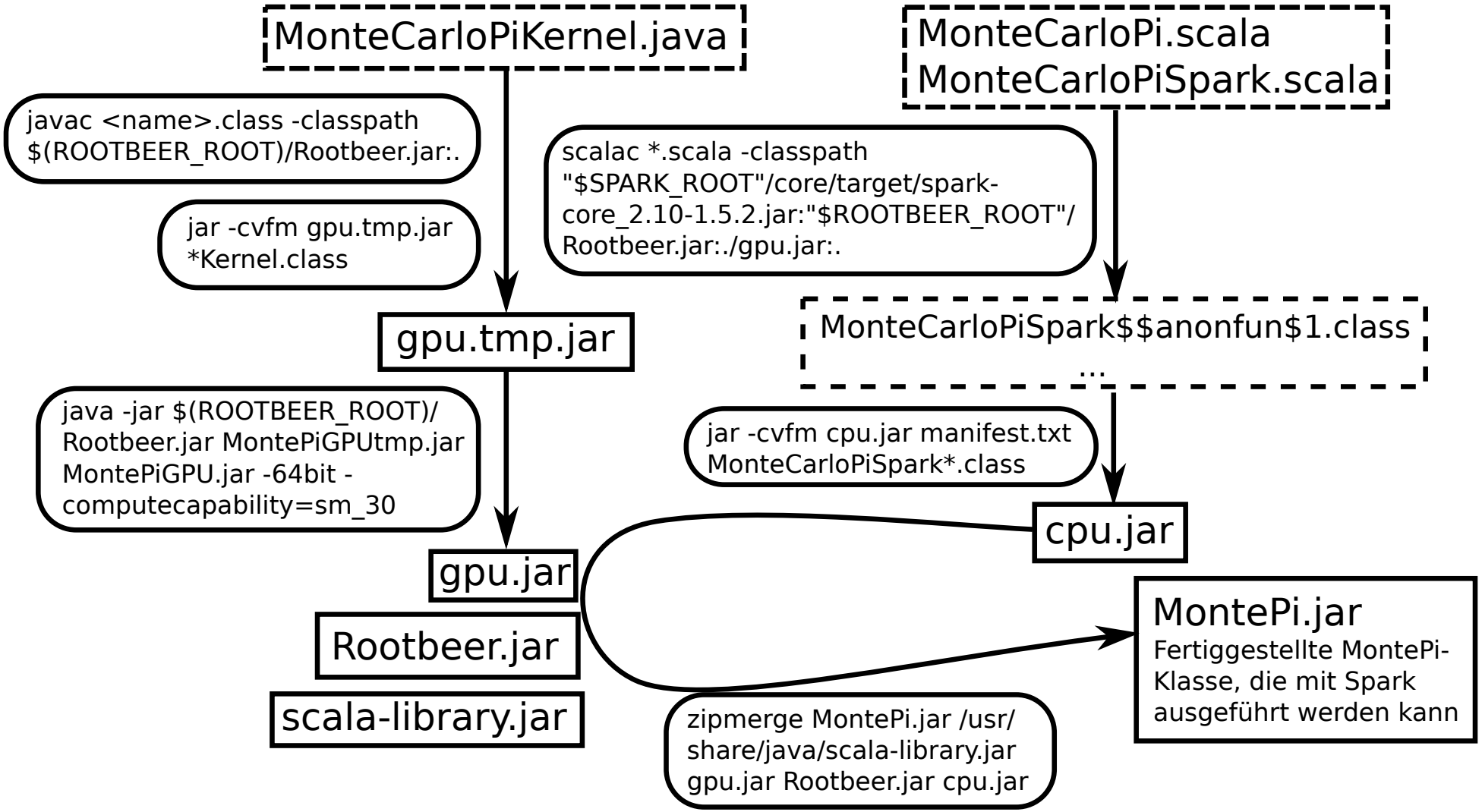
Monte-Carlo-Pi mit Rootbeer Teil 2

```
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 runWaitEvent.take()
```

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.
- Rootbeer hat standardmäßig versucht den gesamten freien Speicher zu alloziieren versuchte
- Zu kompilierende jar an Rootbeer >muss< auf .jar enden

Bemerkungen und Hinweise

- Profiling mit NVIDIA nvvp möglich. (Executable: java, Argumente: `-jar ./MontePi.jar`)
- `ERROR STATUS:716 : Error in cuCtxSynchronize` wenn man den Konfigurationsordner löscht oder woanders kompiliert als man ausführt
- Shared Libraries werden nach `~/.rootbeer/` entpackt, was aber auf Taurus shared ist → Bugfix: entpacke nach `~/.rootbeer/<hostname>/<pid>-<nanotimestamp>/`
- **Flüchtiger Fehler**

```
1 java.lang.ClassCastException: MonteCarloPiKernel
      cannot be cast to [J
2   at MonteCarloPiKernel.
      org_trifort_readFromHeapRefFields_MonteCarloPiKerne
      (Jasmin)
3   at MonteCarloPiKernelSerializer.doReadFromHeap(
      Jasmin)
4   at org.trifort.rootbeer.runtime.Serializer.
```

Testsystem: Taurus Insel 2 Phase 2

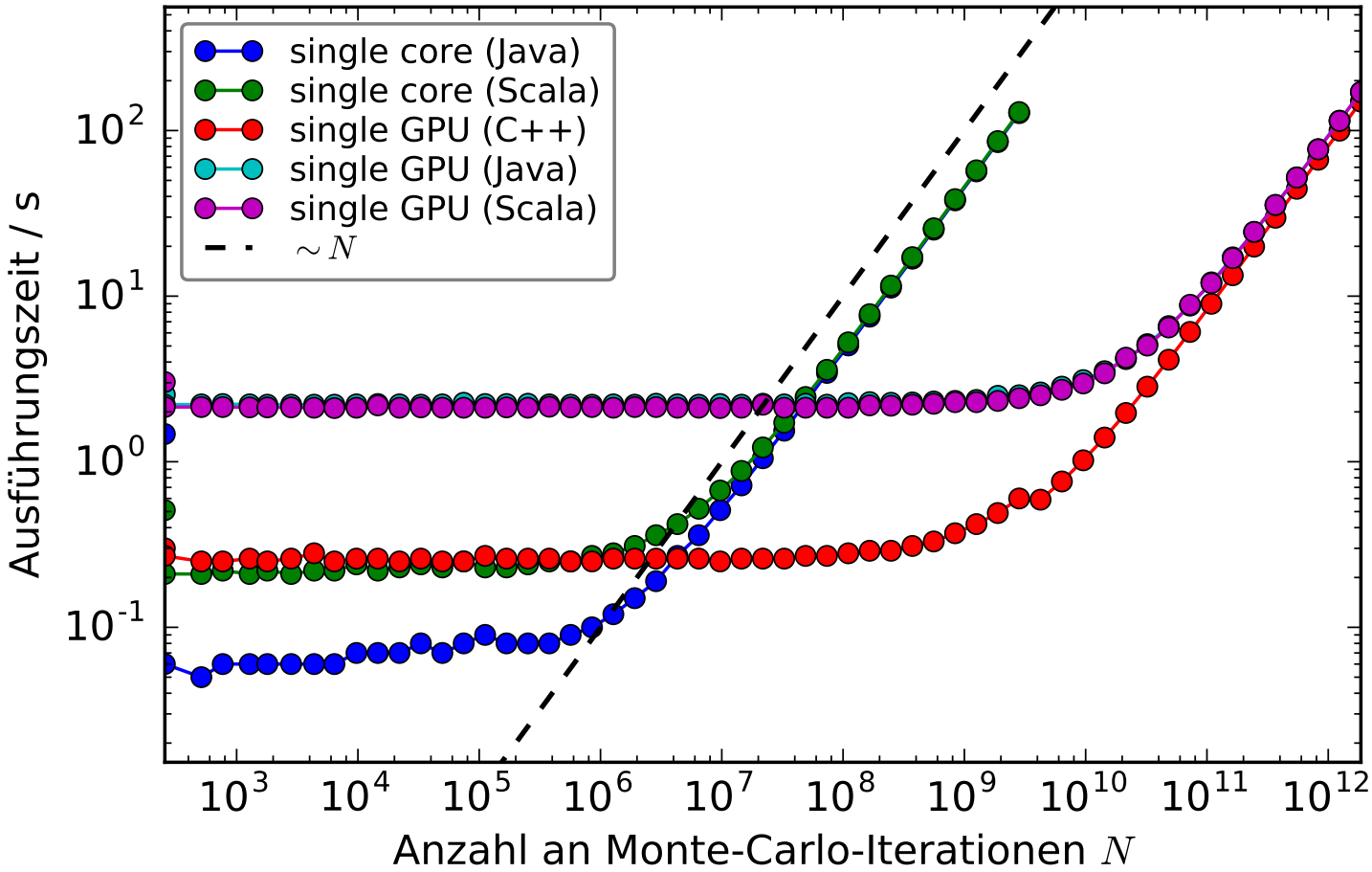
Knoten	64
Hostnamen	taurusi2[045-108]
Prozessor	2x Intel(R) Xeon(R) CPU E5-2680 v3 (12 Kerne) @ 2.50GHz, MultiThreading deaktiviert, AVX2 insbesondere FMA3, 2x 537.6 GSPFLOPS
GPU	4x NVIDIA Tesla K80
Arbeitsspeicher	64 GiB
Festplatte	128 GiB SSD

Tesla K80:

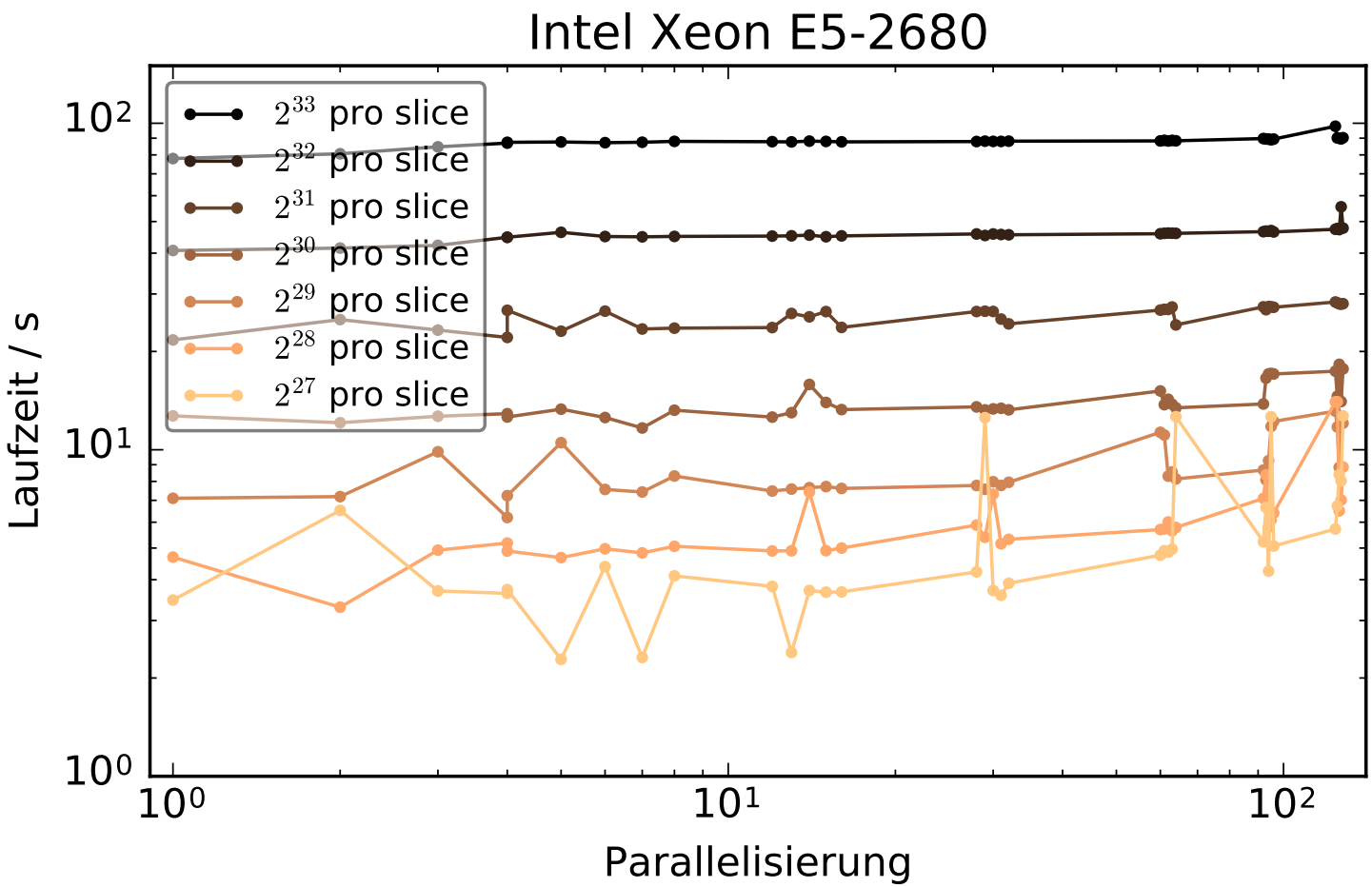
Chip	GK210
Takt	0.560 GHz
CUDA-Kerne	4992
Speicher Bandbreite	$2 \times 240\,\mathrm{GB\,s^{-1}}$
Theoretische Spitzenleistung	5591 GSPFLOPS

Leistungsanalyse: 1 CPU-Kern / GPU auf Taurus gpu2

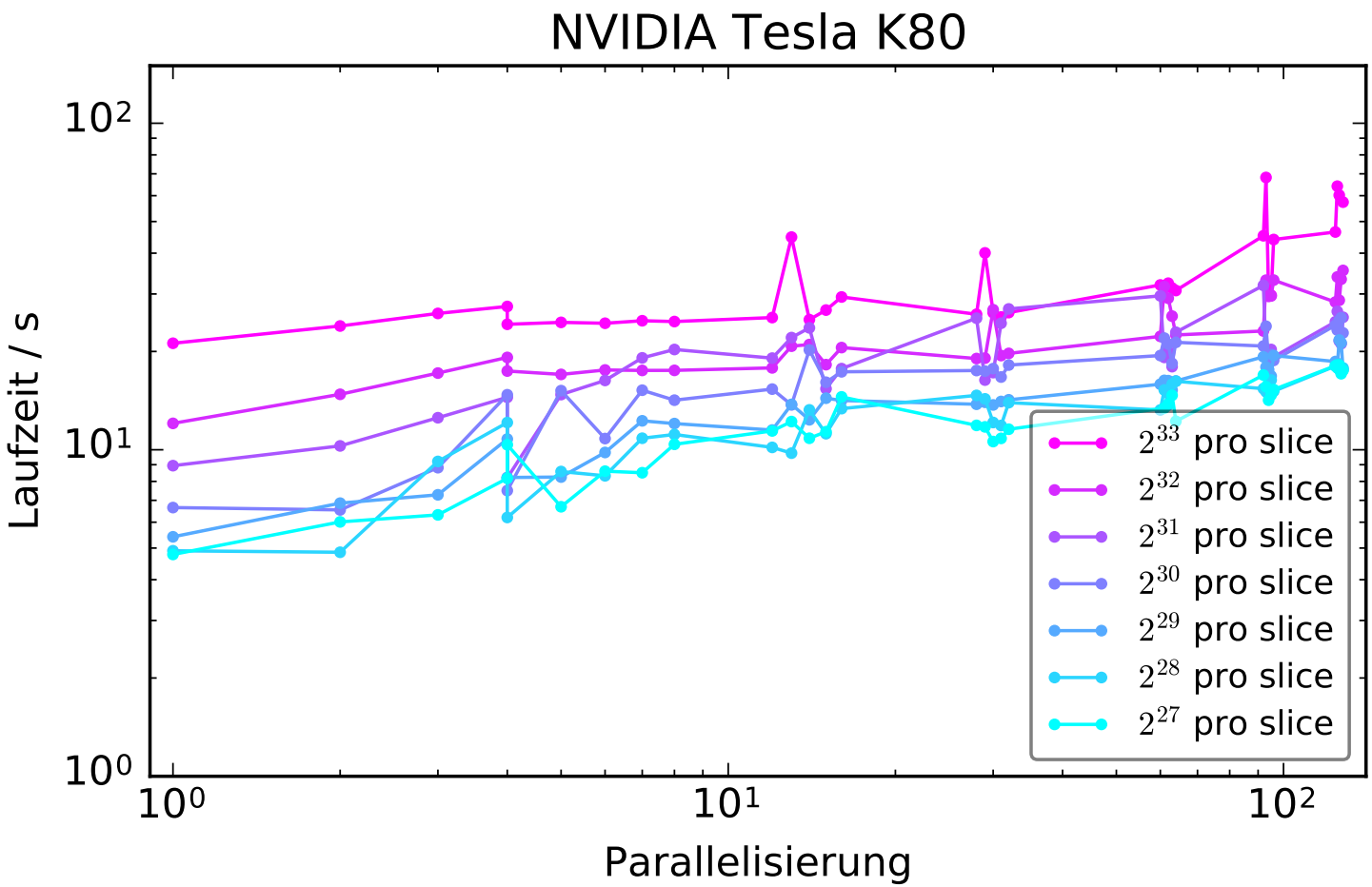
```
salloc -p gpu2-interactive --nodes=1 --ntasks-per-
    node=1 --cpus-per-task=1 --gres=gpu:1 --time
    =02:00:00
```



Benchmark Spark auf CPU



Benchmark Spark mit Rootbeer



Zusammenfassung

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

Ausblick:

- Codereview von Rootbeer oder Nutzung anderer GPU-API ist nötig
- heterogene Berechnungen auf CPU + GPU (Problem Lastbalancierung)
- Implementation direkt in Spark würde z.B. cache/persist auf GPUs erlauben, um Host-GPU-Transfers zu sparen