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

Maximilian Knespel

Betreuer: Dipl.-Inf. Nico Hoffmann

GPU-Beschleuniger zum Hochleistungsrechnen

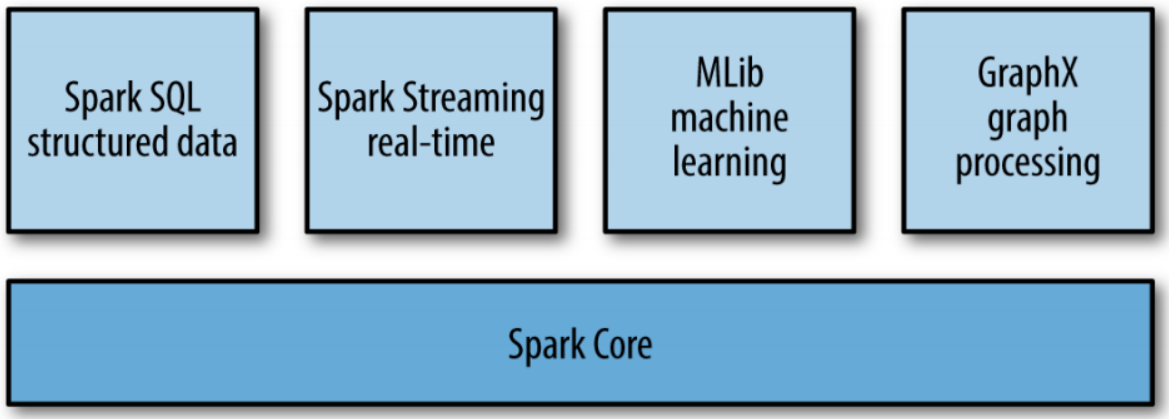
General Purpose Graphical Processing Units (GPGPU) im Vergleich zu Prozessoren:

- \+ bessere Leistungsaufnahme pro GFlops
- – nur für bestimmte Anwendungen geeignet
- – **zusätzliche Komplexität beim Programmieren**

Spark Übersicht

Alternative zu 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)

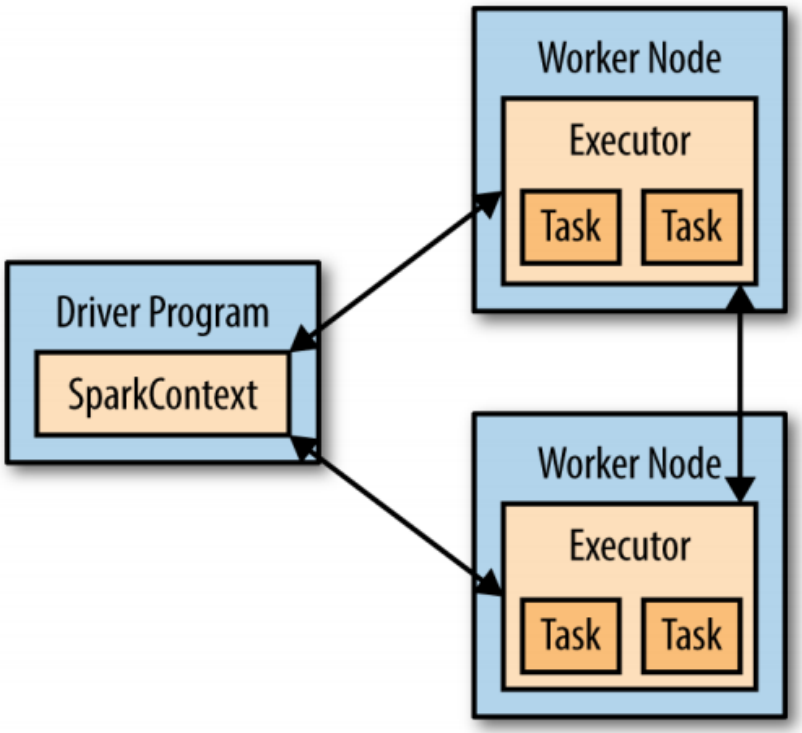
- zu bearbeitende Daten liegen in RDD-Objekten
- Resilience: opake Neuberechnung der Teildaten bei Absturz eines Knotens möglich
- Methoden zur Verarbeitung der Daten

```
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



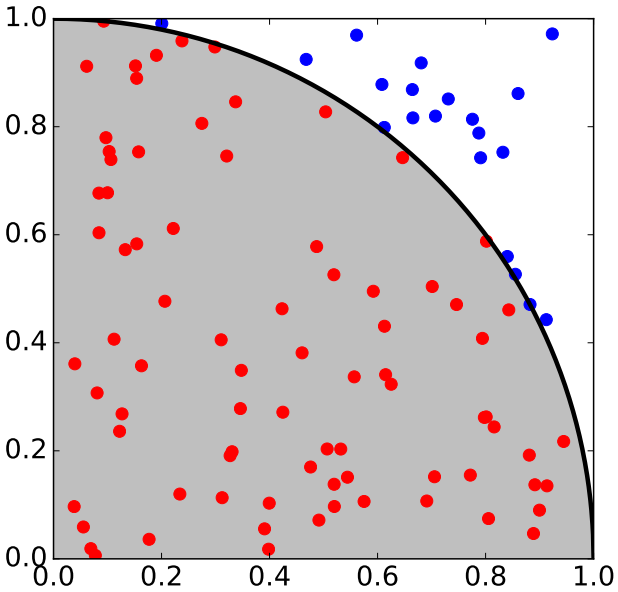
Master:

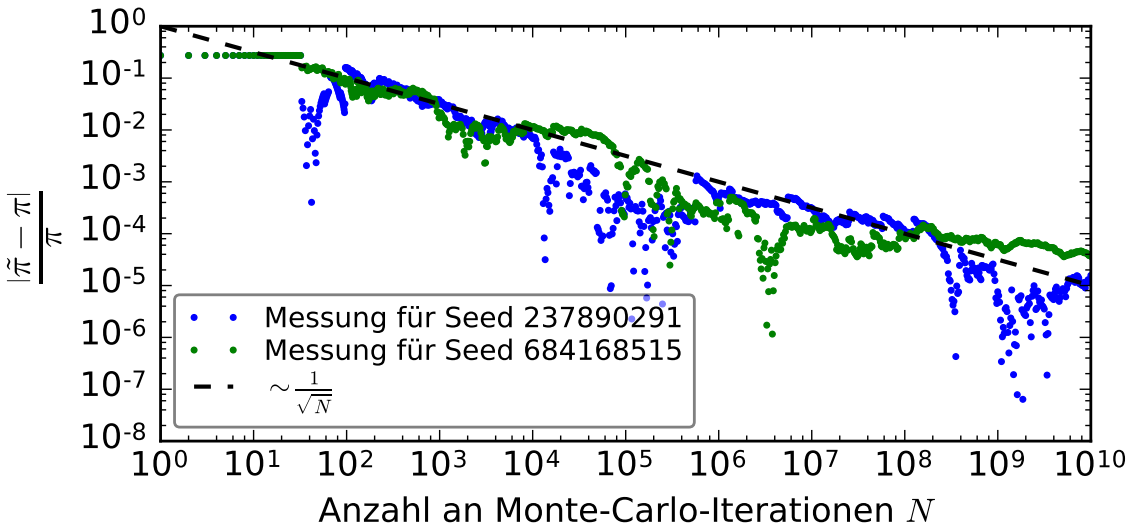
```
"$SPARK_ROOT/bin/spark-class"
    org.apache.spark.deploy.
    master.Master --ip $(
    hostname) --port 7077 --
    webui-port 8080
```

Worker:

```
"$SPARK_ROOT/bin/spark-class"
    org.apache.spark.deploy.
    worker.Worker \
    spark://$(scontrol show
    hostname $SLURM_NODELIST |
    head -n 1):7077
```

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
```

Rootbeer

- ≈ 30k Codezeilen hauptsächlich Java und ein wenig C
- leider kaum Aktivität auf Github seit Juni 2015
- Unterstützte 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 );
```

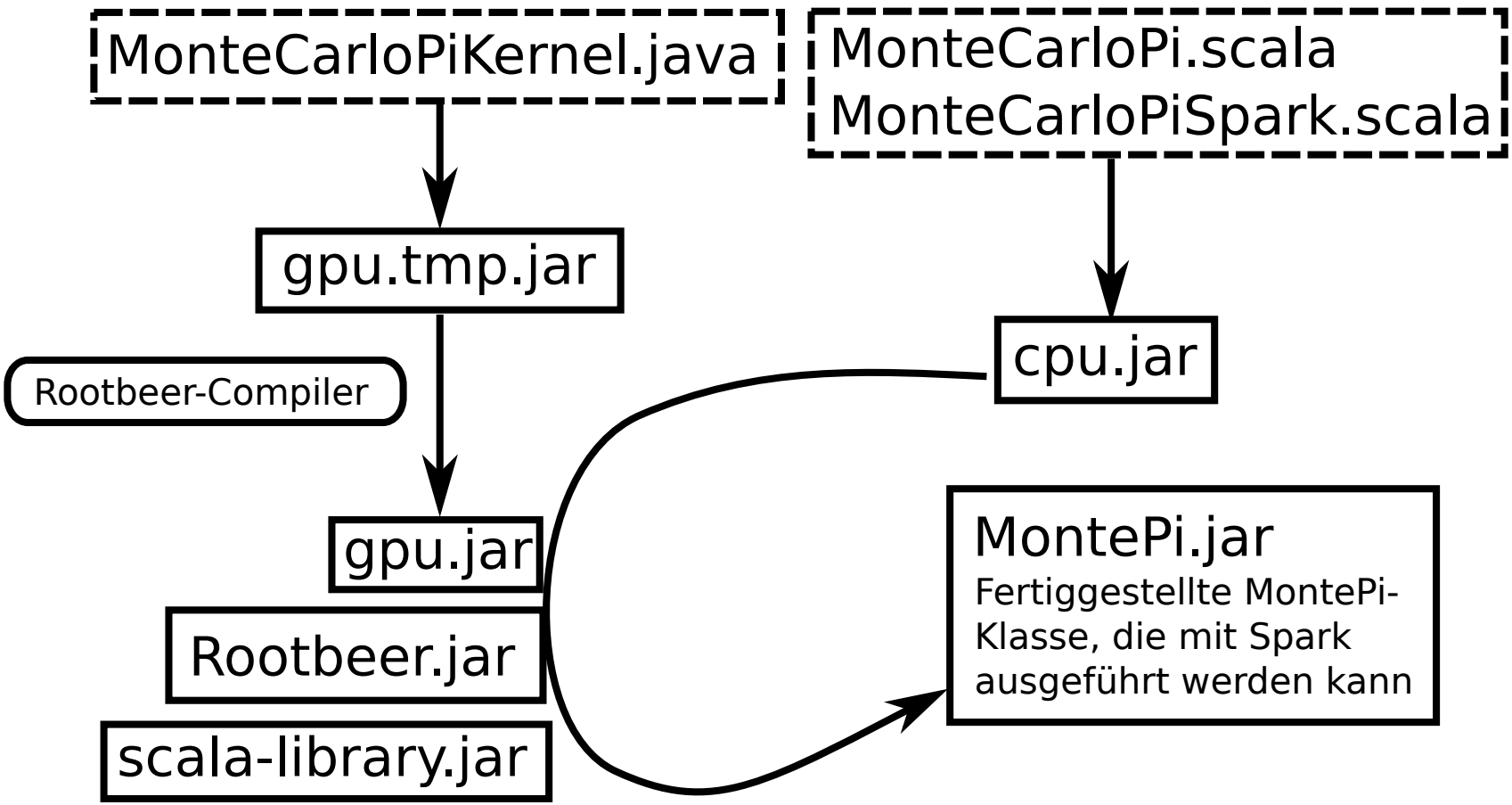
Monte-Carlo-Pi mit Rootbeer Teil 2

```
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

- Rootbeer funktioniert nicht mit Java 8
- Automatische Kernel-Konfiguration hatte einen Bug, der standardmäßig immer so viele Kernel startete wie gleichzeitig auf einem Shared Multiprozessor laufen können.
- Rootbeer hat standardmäßig versucht den gesamten freien Speicher zu alloziieren
- Rootbeer war nicht thread-safe

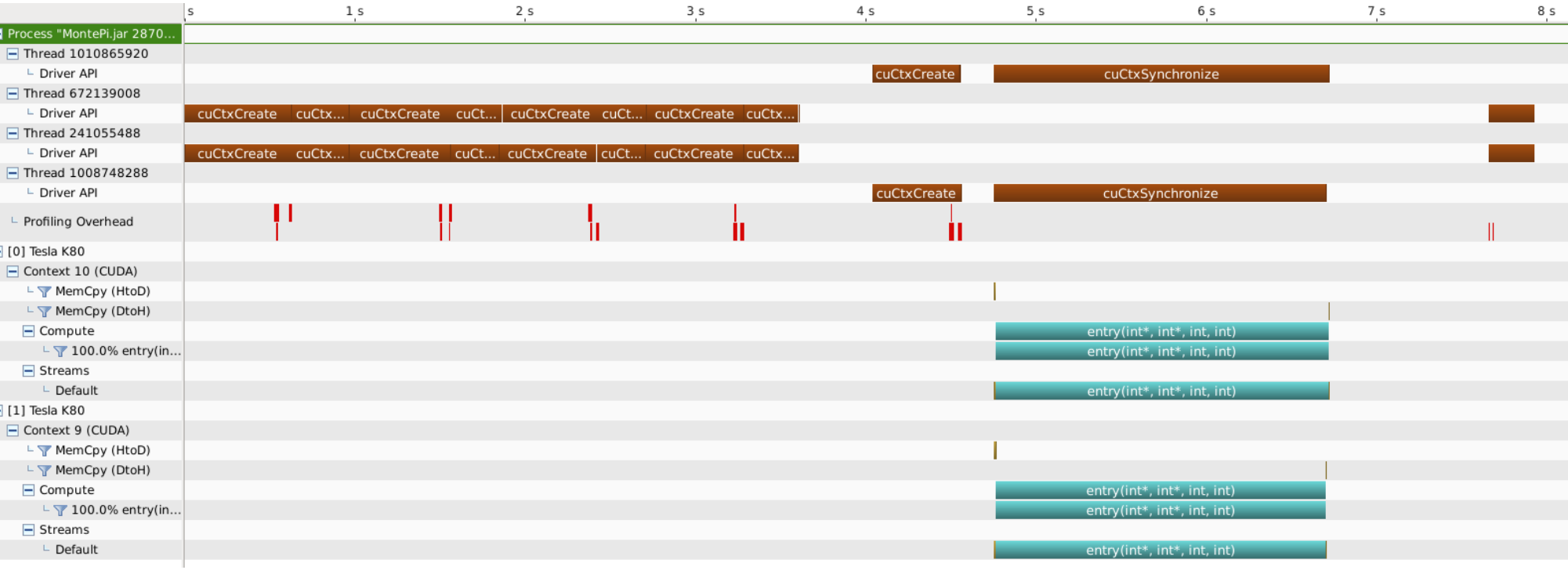
⇒ Fork auf `https://github.com/mxmlnkn/rootbeer1`

Testsystem: Taurus Insel 2

Knoten	44
Prozessor	2x Intel(R) Xeon(R) CPU E5-2450 (8 Kerne) @ 2.10GHz, 2x 268.8 GSPFLOPS
GPU	2x NVIDIA Tesla K20x je 2688 CUDA-Kerne, 3935 GSPFLOPS
Arbeitsspeicher	32 GiB
Knoten	64
Prozessor	2x Intel(R) Xeon(R) CPU E5-2680 v3 (12 Kerne) @ 2.50GHz, 2x 537.6 GSPFLOPS
GPU	4x NVIDIA Tesla K80 je 4992 CUDA-Kerne, 5591 GSPFLOPS
Arbeitsspeicher	64 GiB

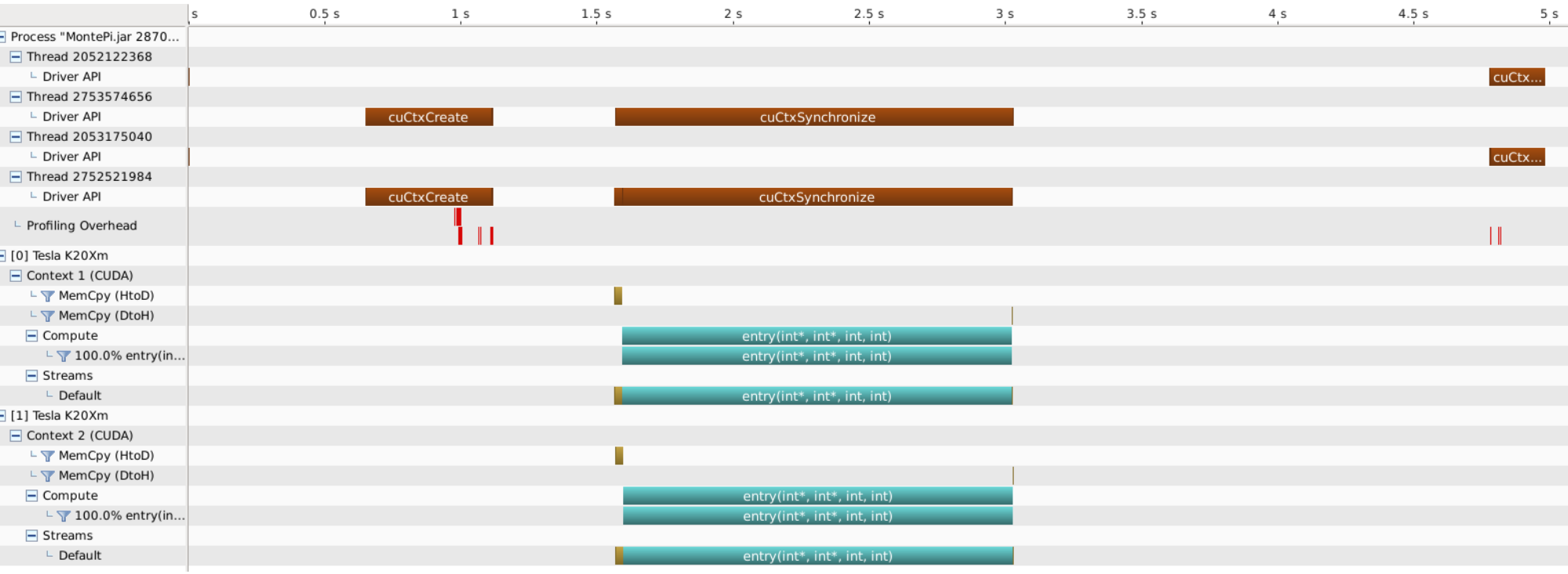
Profiling mit NVIDIA nvvp möglich:

```
nvprof --analysis-metrics --metrics all -o $HOME/
    scaromare/MontePi/profilingDataMultiGpuScala.nvp%
    p /sw/global/tools/java/jdk1.7.0_25/bin/java -jar
     $HOME/scaromare/MontePi/singleNode/multiGpu/
    scala/MontePi.jar $((2*14351234510)) 2
```



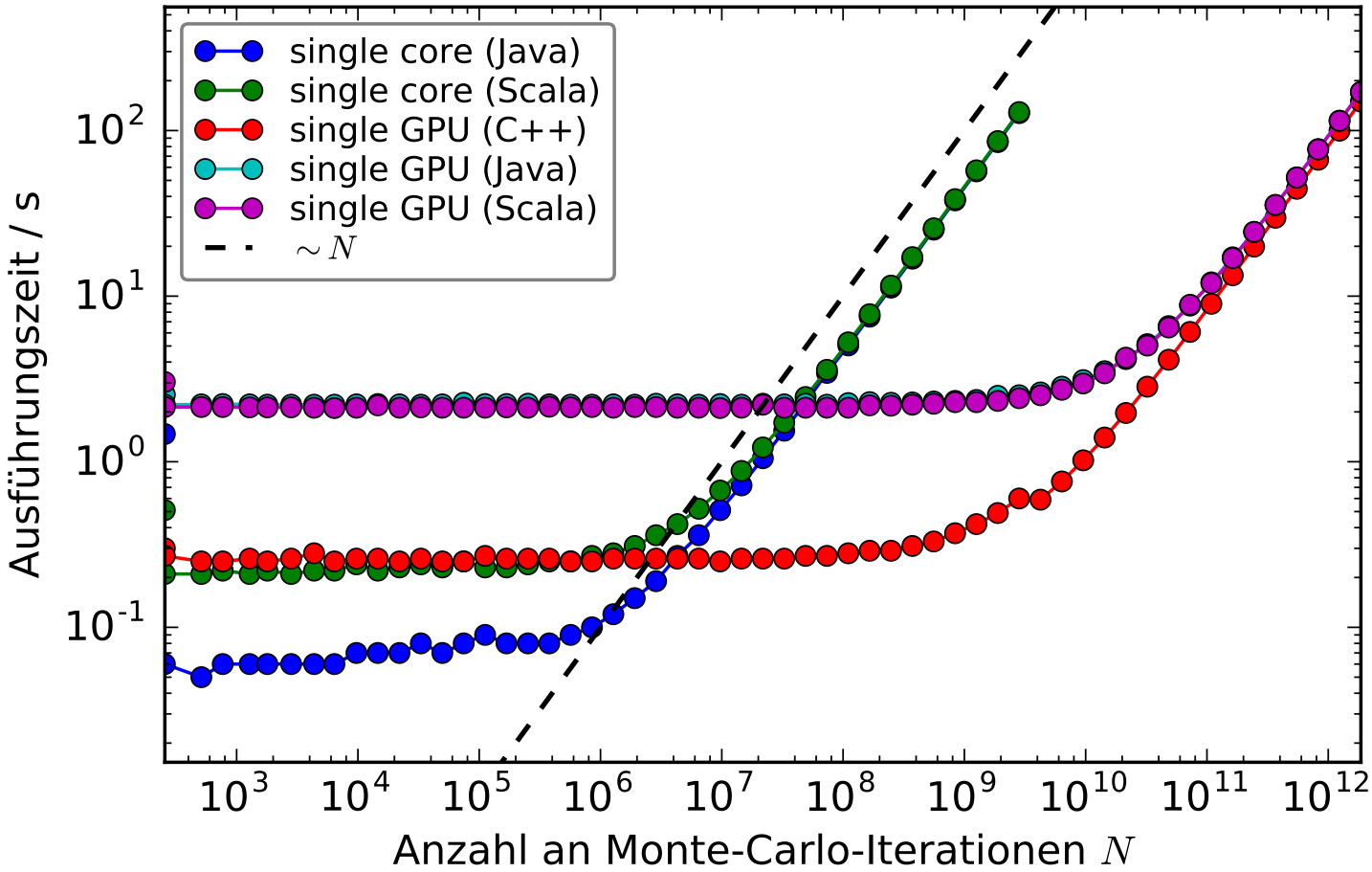
Profiling mit NVIDIA nvvp möglich:

```
nvprof --analysis-metrics --metrics all -o $HOME/
    scaromare/MontePi/profilingDataMultiGpuScala.nvp%
    p /sw/global/tools/java/jdk1.7.0_25/bin/java -jar
     $HOME/scaromare/MontePi/singleNode/multiGpu/
    scala/MontePi.jar $((2*14351234510)) 2
```

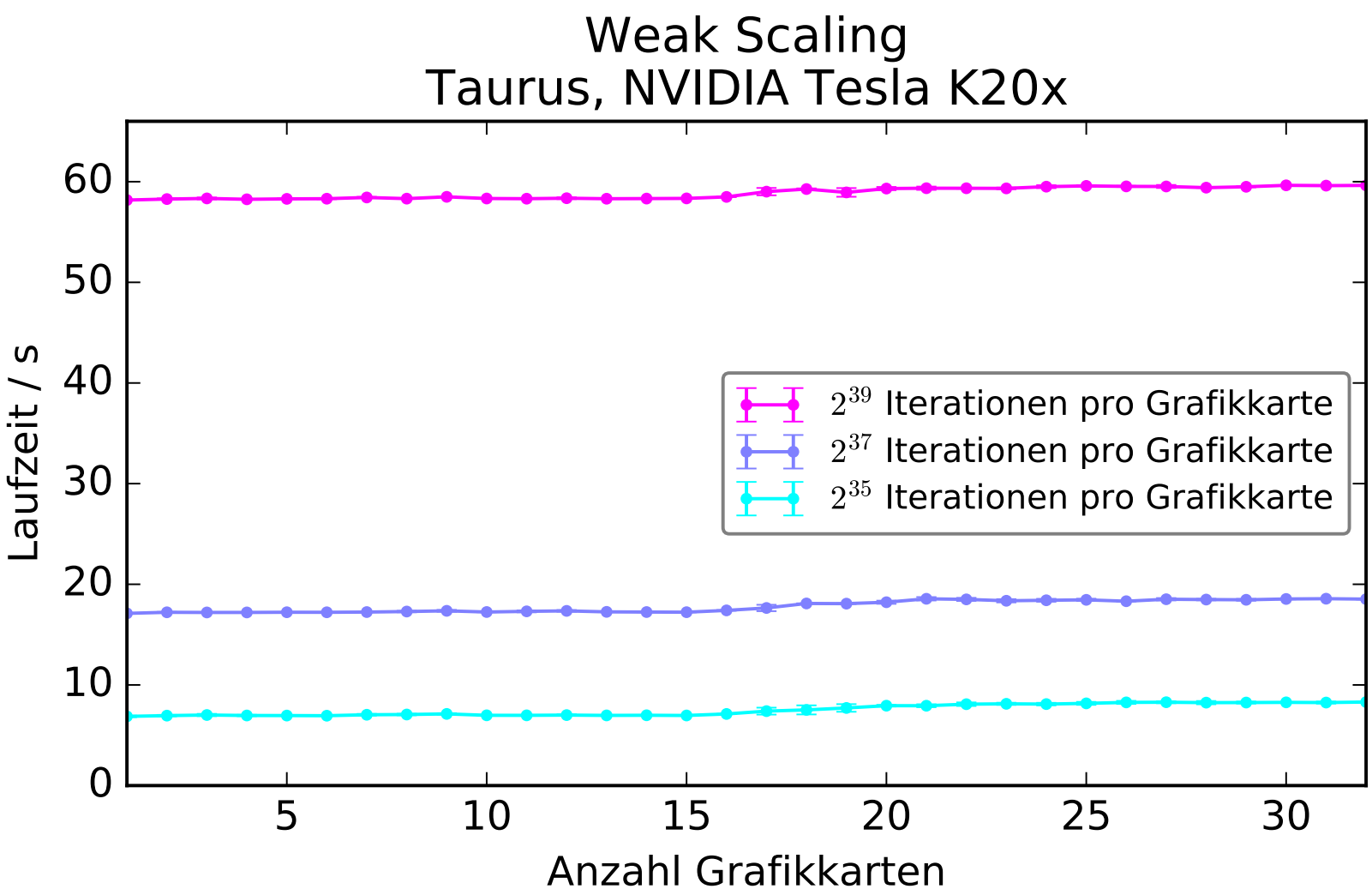


Leistungsanalyse: 1 CPU-Kern / GPU auf Taurus gpu2

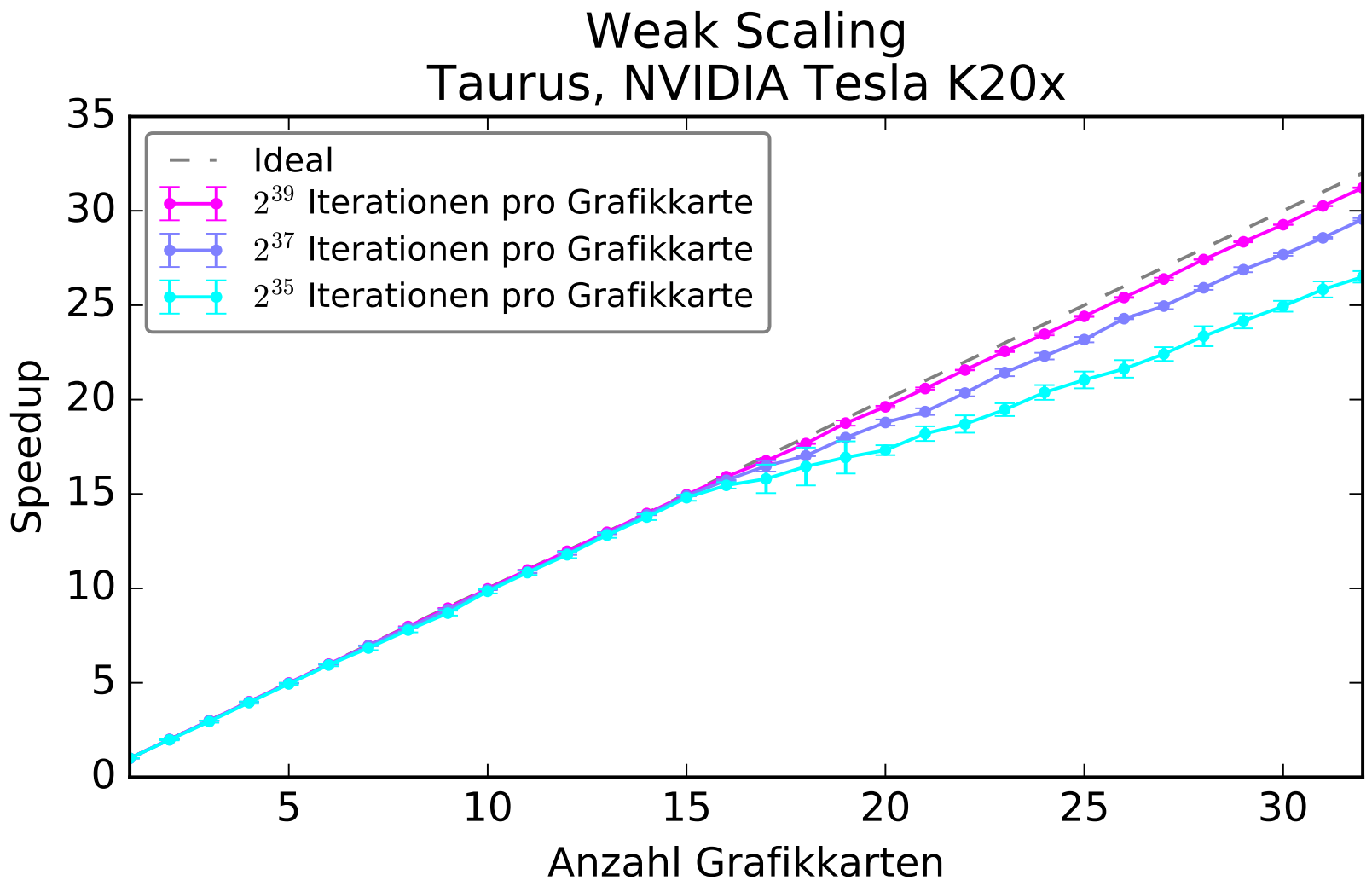
```
salloc -p gpu2-interactive --nodes=1 --ntasks-per-
    node=1 --cpus-per-task=1 --gres=gpu:1 --time
    =02:00:00
```



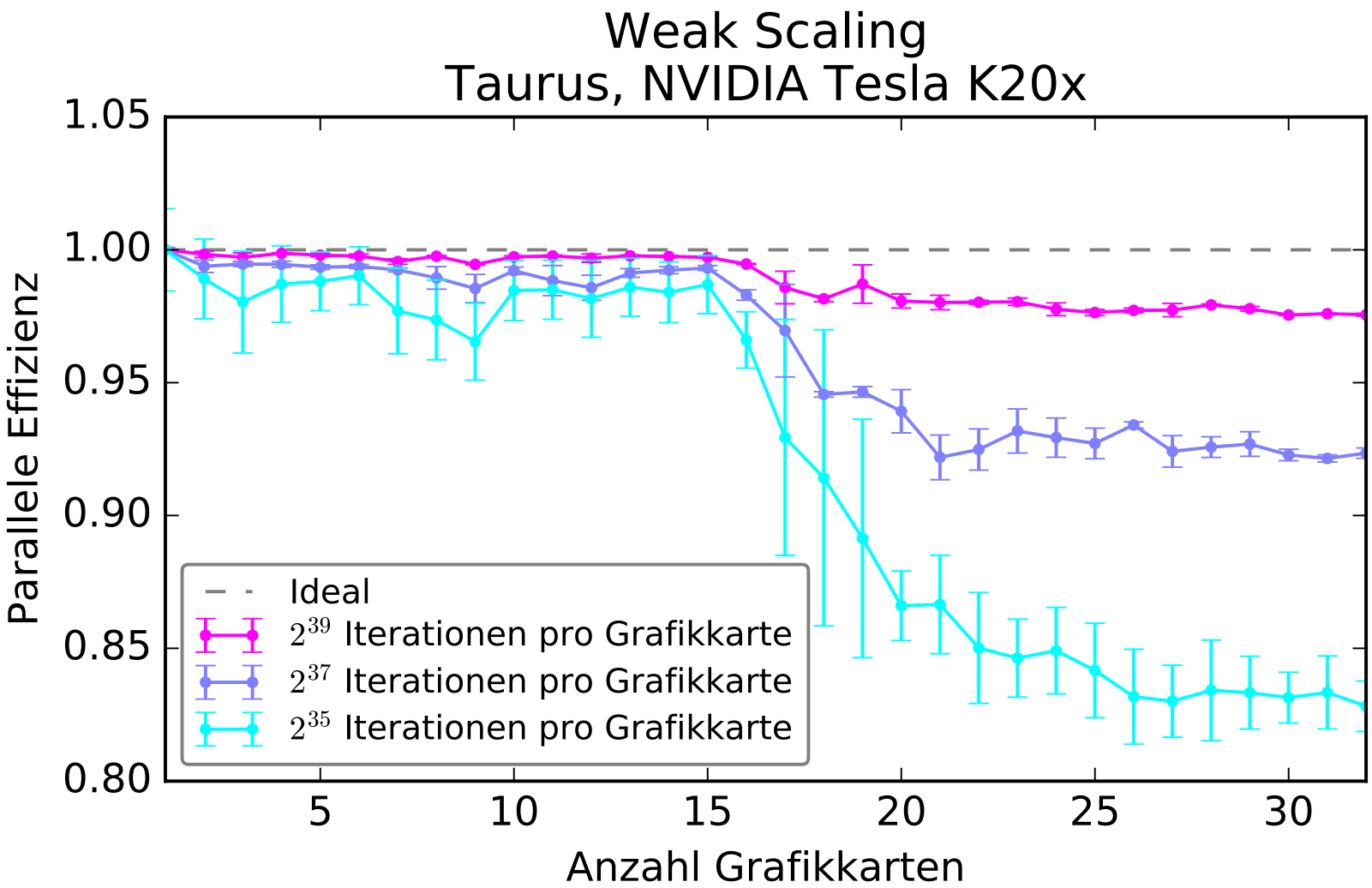
Weak-Scaling Benchmark Spark mit Rootbeer



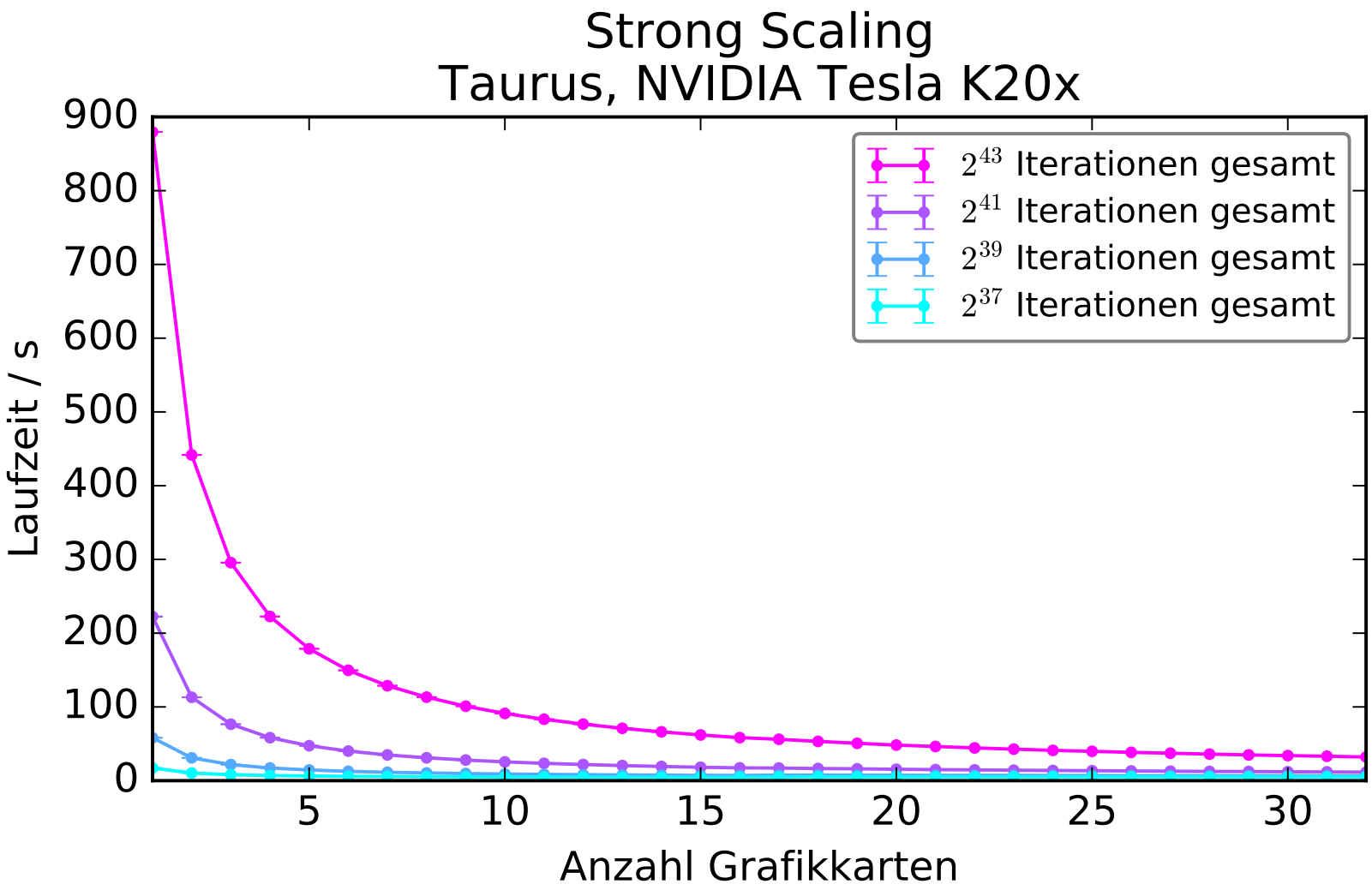
Weak-Scaling Benchmark Spark mit Rootbeer



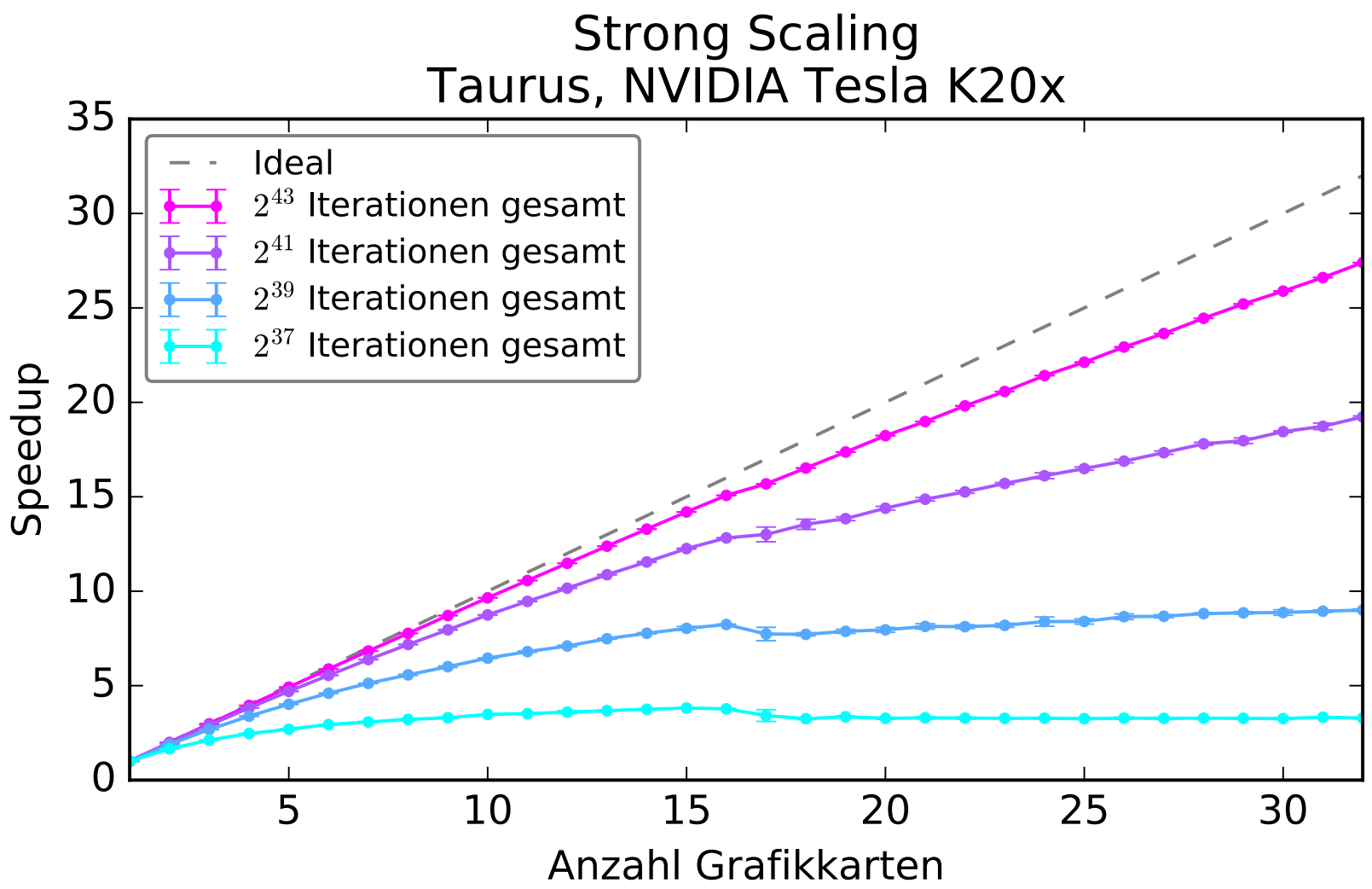
Weak-Scaling Benchmark Spark mit Rootbeer



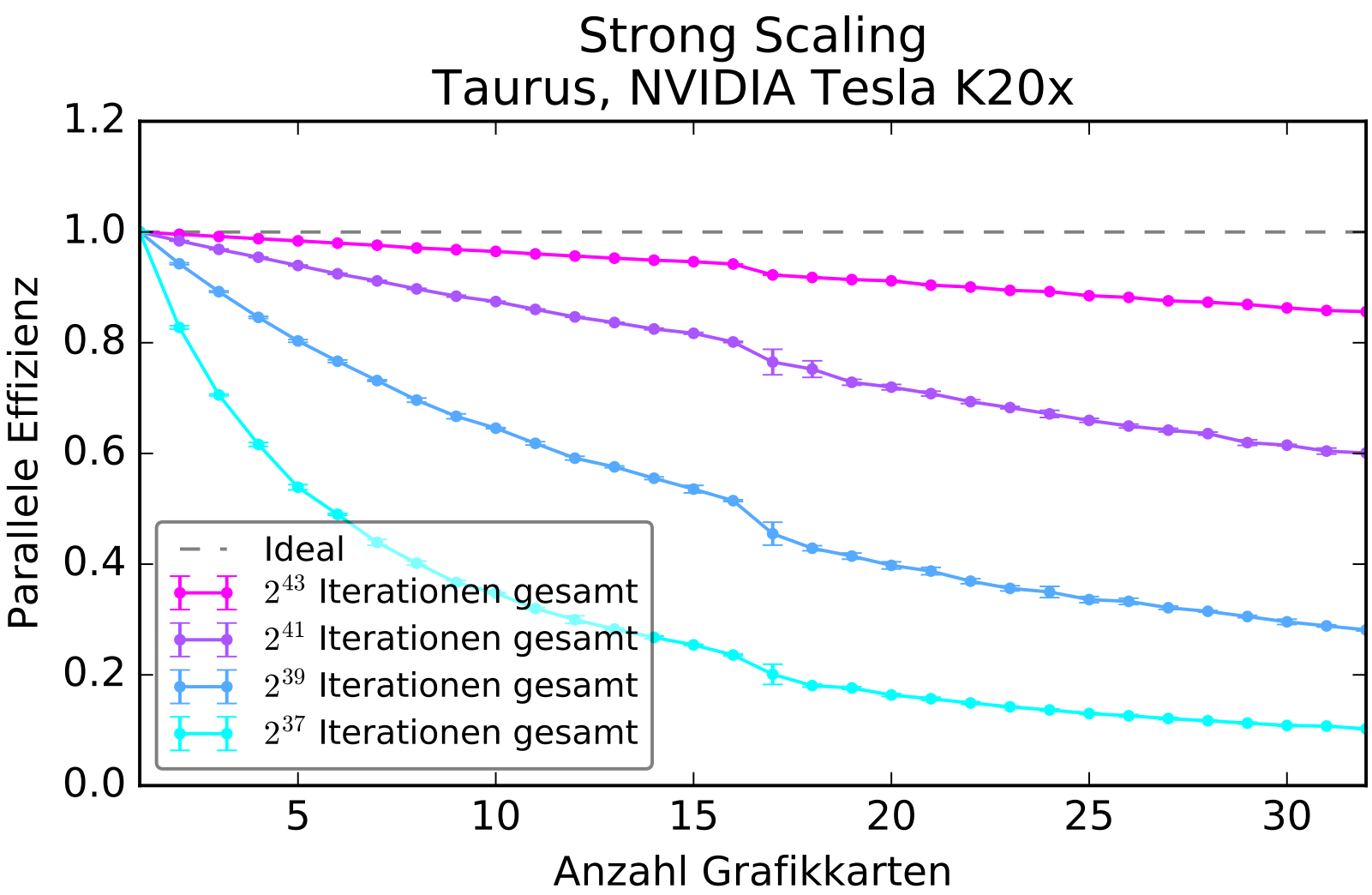
Strong-Scaling Benchmark Spark mit Rootbeer



Strong-Scaling Benchmark Spark mit Rootbeer



Strong-Scaling Benchmark Spark mit Rootbeer



Zusammenfassung

- Mehrere Bugfixes für Rootbeer geschrieben
- Kombination aus GPGPU mittels Rootbeer und Spark ausgetestet:
  `https://github.com/mxmlnkn/scaromare`

Ausblick:

- heterogene Berechnungen auf CPU + GPU (Problem Lastbalancierung)
- Erweiterung von Rootbeer, um Daten zwischen iterativen Map-Operationen im Speicher zu behalten
- festere Integration in den Spark-Scheduler zur Vermeidung von Stragglern