JCudaMP: OpenMP/Java on CUDA

Georg Dotzler, Ronald Veldema, Michael Klemm
„Write once, run anywhere“
- Java Slogan created by Sun Microsystems
Motivation

- Keeping that promise in mind:
  - How to use general purpose GPUs (gpGPU) ...
  - How to recognize parallel regions ...

- ... to speed-up Java programs without ...
  - ... grave changes to existing source code?
  - ... risking performance loss on some target systems?
Motivation

- Keeping that promise in mind:
  - How to use general purpose GPUs (gpGPU) ...
  - How to recognize parallel regions ...

- ... to speed-up Java programs without ...
  - ... grave changes to existing source code?
  - ... risking performance loss on some target systems?

- Matrix multiplication 4096 x 4096 floats
  - In Java: 855 seconds
Motivation

- Keeping that promise in mind:
  - How to use general purpose GPUs (gpGPU) ...
  - How to recognize parallel regions ...

- ... to speed-up Java programs without ...
  - ... grave changes to existing source code?
  - ... risking performance loss on some target systems?

- Matrix multiplication 4096 x 4096 floats
  - In Java: 855 seconds
  - In C: 736 seconds
Motivation

- Keeping that promise in mind:
  - How to use general purpose GPUs (gpGPU) ...
  - How to recognize parallel regions ...

- ... to speed-up Java programs without ...
  - ... grave changes to existing source code?
  - ... risking performance loss on some target systems?

- Matrix multiplication 4096 x 4096 floats
  - In Java: 855 seconds
  - In C: 736 seconds
  - In CUDA: 13 seconds
Overview

- Introduction
  - OpenMP
  - JaMP
  - CUDA

- Issues in Using GPUs from Java
  - Transparent Use of CUDA
  - Memory Management
  - Limited GPU Memory

- Performance Measurements
- Summary
Introduction - OpenMP

- **Open Multi-Processing**
  - Standardized specification for parallel programming

- API for multi-platform shared-memory parallel programming in C/C++ and Fortran

- Consists of
  - compiler directives (#pragma omp parallel for, ...)
  - runtime routines (omp_set_num_threads(), ...)
  - environment variables (OMP_NUM_THREADS, ...)
Introduction - JaMP

- Adaptation of OpenMP for Java
- Compiler directives are implemented as comments
  - `//#omp parallel for`
  - `for (int i = 0; i < SIZE; i ++)`
  - `work();`
- Implemented with a modified Eclipse Java Compiler
  - Translates JaMP code to Java bytecode
- Runtime routines and environment variables are provided by a Java package
Introduction - CUDA

- **Compute Unified Device Architecture**
  - NVIDIA GPU architecture for parallel computations
  - Programmable in C (with NVIDIA extensions)

- **Example:**
  ```c
  __global__ void kernel( float* a) {
    work();
  }
  int main(int argc, char** argv) {
    float* gpu_a = cudaMalloc (size_a);
    kernel<<<number_of_Threads>>> (gpu_a);
    cudaMemcpy (gpu_a, a, size_a);
  }
  ```
CUDA Architecture

GPU

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
</tr>
</tbody>
</table>

GPU Memory

Data

CPU

Cache

ALU

ALU

ALU

ALU

Control

Data

CPU Memory

Data
Issue:
- Identify suitable regions for parallel execution on GPUs
- Different hardware available on the target platforms
- Developers are often not aware of the target platform

Conclusion:
- Postpone the execution strategy until run-time

Unfortunately:
- Loss of developer knowledge about suitable parallel regions
- No GPU support in the current Java JVMs
Transparent Use of CUDA II

Solution:

- Use of JaMP
  - Annotations defined in comments
  - Suitable parallel regions are marked with `//omp parallel for`
  - Compilation in regular Java compilers possible

- Use of additional bytecode section
  - Information is ignored by a standard classloader
  - Contains information about ...
    - the hardware requirements (long, double, float division....)
  - the JaMP annotations
Transparent Use of CUDA III

JaMP Source Code \(\rightarrow\) JCudaMP Compiler \(\rightarrow\) Bytecode with JaMP Attribute

compile-time

run-time

Standard Classloader \(\rightarrow\) Execution with CPU Threads

JCudaMP Classloader \(\rightarrow\) Execution with GPU Threads
 JCuda Classloader at run-time:

- Bytecode with JaMP Attribute
- Requirements fulfilled?
- Yes: Generate and load CUDA Shared Library
- Loading modified Bytecode
- Execution with GPU Threads
- No: Execution with CPU Threads
- JCuda Classloader at run-time:

Transparent Use of CUDA IV
Issue:
- CUDA needs arrays and objects on the graphics card
- Transfer rates
  - In 1 KB blocks: 93.0 MB/s
  - In 1 MB blocks: 5319.1 MB/s
  - In 100 MB blocks: 5623.3 MB/s

Conclusion:
- Make the transferred memory blocks as large as possible

Unfortunately:
- Java uses arrays of arrays
- Array on average < 4KB
- Object on average < 128B
Example:

```java
float[] a = new float[12];
float[][] b = new float[3][];
b[0] = new float[3];
b[1] = new float[4];
b[2] = new float[2];

//#omp parallel for
for (int i = 0; i < SIZE; i++)
    work(a, b);
```
GPU Memory Management III

Java Heap

JCudaMP CPU Memory Area

JCudaMP GPU Memory Area

float b[][]

Header
b[0] b[1] b[2]

b[0][0] b[0][1] b[0][2]

b[1][0] b[1][1] b[1][2] b[1][3]

float a[]

Header
GPU Memory Management IV

Java Heap

float b[][]

Header

b[0] b[1] b[2]

b[0][0] b[0][1] b[0][2]

b[1][0] b[1][1] b[1][2] b[1][3]

float a[]

Header


JCudaMP CPU Memory Area

float a[]

float b[][]

JCudaMP GPU Memory Area
GPU Memory Management

Java Heap

float b[][]

Header

b[0] b[1] b[2]

b[0][0] b[0][1] b[0][2]

b[1][0] b[1][1] b[1][2] b[1][3]

float a[]

Header


JCudaMP CPU Memory Area

float a[]

float b[][]

JCudaMP GPU Memory Area
GPU Memory Management VII

Java Heap

float b[][]
Header
b[0] b[1] b[2]

b[0][0] b[0][1] b[0][2]

b[1][0] b[1][1] b[1][2] b[1][3]

float a[]
Header


JCudaMP CPU Memory Area

float a[]
GPU Memory Management VIII

```
float b[][
Header b[0] b[1] b[2]
```

```
float a[]
```

```
JCudaMP CPU Memory Area
```

```
float a[]
```

```
Java Heap
```
GPU Memory Management IX

**Java Heap**

`float b[][]`

`Header`:

```
<table>
<thead>
<tr>
<th>b[0]</th>
<th>b[1]</th>
<th>b[2]</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>b[0][0]</th>
<th>b[0][1]</th>
<th>b[0][2]</th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>b[1][0]</th>
<th>b[1][1]</th>
<th>b[1][2]</th>
<th>b[1][3]</th>
</tr>
</thead>
</table>
```

`float a[]`

`Header`:

```
|------|------|------|------|------|------|------|------|------|------|-------|-------|
```

**JCudaMP CPU Memory Area**

`float a[]`

```
|------|-----|-------|------|------|------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
```

```
|------|-----|-------|------|------|------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
```
Memory Management – Recurring Transfers

- **Issue:**
  - Often parallel regions are called more than once
  - OpenMP requires that data changes on shared memory are visible after a parallel region
  - A naive and conservative translation causes a copy in and copy out operation for each execution
  - Unnecessarily consumes time

- **Conclusion:**
  - Data should be kept on the GPU for performance

- **Unfortunately:**
  - Hard to decide how long the data should remain on the GPU
Example of Recurring Transfers I

```java
int[][] arrayB = new int[4096][42];
...
while (quality < 5) {
    //Copy data to the graphics card
    //#omp parallel for
    for (int i = 0; i < size; i++) {
        arrayB[i][j] = ...
    }
    //Copy data to main memory
    quality = calculateQuality(arrayB);
}

public int calculateQuality(int[][] array){
    return array[0][0];
}
```
Example of Recurring Transfers II

```java
int[][] arrayB = new int[4096][42];
...
while (quality < 5) {
    //Copy data to the graphics card
    //#omp parallel for
    for (int i = 0; i < size; i++) {
        arrayB[i][j] = ...
    }
    //Copy data to main memory
    quality = calculateQuality(arrayB);
}

public int calculateQuality(int[][] array){
    return array[0][0];
}
```

A naive implementation copies arrayB to the GPU and back for each loop iteration.
int[][] arrayB = new int[4096][42];

... 

while (quality < 5) {
    //Copy data to the graphics card
    for (int i = 0; i < size; i++) {
        arrayB[i][j] = ...
    }

    //Copy data to main memory
    quality = calculateQuality(arrayB);
}

public int calculateQuality(int[][] array) {
    return array[0][0];
}
Memory Management - Array Packages I

- Implementation of an array package
  - In CUDA a standard array access is used (e.g. a[i][j] bzw. a[i * length + j])
  - In Java get/set methods are used

- Array data remains on the graphics card

- Allows different handling of rectangular arrays

- The Java part holds a small cache for data elements

- Use of `//omp managed` annotations to avoid ...
  - ... large code changes (no get/set methods)
  - ... array packages in regular Java compilers
Example:

```
IntArrayPackage ArrayB = new IntArrayPackage(4096,42);
..
while (quality < 0.5) {
    //Copy to the graphics card (arrayA, arrayB)
    //#omp parallel for
    for (int i = 0; i < size; i++) {
        int tmp = arrayA.get(8) + arrayB.get(4, i);
        arrayB.set(i, 4, tmp);
    }
    //Copy to main memory (arrayA, arrayB)
    quality = calculateQuality(arrayA, arrayB.get(0,0));
}
```
Memory Management - Array Packages III

- Same example with managed annotation:

```java
// #omp managed
int[][] ArrayB = new int[4096][42];

.. while (quality < 0.5) {
    // Copy to the graphics card (arrayA, arrayB)
    // #omp parallel for
    for (int i = 0; i < size; i++) {
        int tmp = arrayA[8] + arrayB[4][i];
        arrayB[i][4] = tmp;
    }
    // Copy to main memory (arrayA, arrayB)
    quality = calculateQuality(arrayA, arrayB[0][0]);
}
```
Limited GPU Memory I

- **Issue:**
  - Data often fits barely into main memory (~16GB)
  - GPU memory is much smaller (~1GB)

- **Conclusion:**
  - Splitting of parallel regions and data to reduce memory requirements

- **Unfortunately:**
  - Exact data size and index values not available at compile-time
  - Index analysis can get very time consuming at run-time (requires alias analysis)
Limited GPU Memory II

Java Example:

```java
for (int x = start_x; x < size_x-1; x++) {
    for (int y = start_y; y < size_y-1; y++) {
        float value1 = src[x - 5, y + 2] + x;
        float value2 = src2[y - 1, x + 1];
        float value3 = src[x + 3, y];
        dst[x, y] = value1 + value2;
    }
}
```
Limited GPU Memory III

- **Java Example:**
  ```java
  //omp parallel for tile ( dst : { x : 0 , 0 ; y : 0 , 0 } ...  

  for ( int x = start_x; x < size_x−1; x++) {
    for ( int y = start_y; y < size_y−1; y++) {
      float value1 = src [ x − 5, y + 2 ] + x;
      float value2 = src2 [ y − 1, x + 1 ];
      float value3 = src [ x + 3 , y ];
      dst [ x , y ] = value1 + value2 ;
    }
  }
  ```
Limited GPU Memory IV

Java Example:

```java
//omp parallel for tile ( dst : { x : 0 , 0 ; y : 0 , 0 } ,
    src: { x :-5 , 3 ; y : 0 , 2} ...  

for ( int x = start_x; x < size_x-1; x++) {
    for ( int y = start_y; y < size_y-1; y++) {
        float value1 = src [ x − 5 , y + 2 ] + x;
        float value2 = src2 [ y − 1, x + 1 ] ;
        float value3 = src [ x + 3 , y ] ;
        dst [ x , y ] = value1 + value2 ;
    }
}  
```
Limited GPU Memory V

- Java Example:

```java
//#omp parallel for tile ( dst : { x : 0 , 0 ; y : 0 , 0 } ,
    src: { x :-5 , 3 ; y : 0 , 2} ,
    src2: { y :-1 , -1; x : 1 , 1})

for ( int x = start_x; x < size_x-1; x++) {
    for ( int y = start_y; y < size_y-1; y++) {
        float value1 = src [ x - 5, y + 2 ] + x;
        float value2 = src2 [ y - 1, x + 1 ] ;
        float value3 = src [ x + 3 , y ] ;
        dst [ x , y ] = value1 + value2 ;
    }
}
```
Limited GPU Memory VI

\[
tile(\text{dst}: \{ x: 0, 0; y: 0, 0 \}, \\
\quad \text{src}: \{ x: -5, 3; y: 0, 2 \}, \\
\quad \text{src2}: \{ y: -1, -1; x: 1, 1 \}) \\
\]

Block x: 3 – 5
Block y: 6 – 9

![Diagram showing limited GPU memory with tiles and blocks.](image-url)
Performance Test Environment

- **CPU:** Dual-Quadcore Intel Xeon L5420 (2.5 Ghz)
- **Memory:** 16 GB
- **GPU:** GeForce GTX 280 SC (1 GB Memory)
- **OS:** Linux, kernel 2.6.24
- **Compiler:** CUDA 2.2, GCC version 4.2.4
- **JVM:** Java Hotspot JIT in Version 1.6.0_13
Run-time comparison of the different program parts

- Compilation Time: 3.7 s
- Kernel Execution: 1.7 s
- Memory Management: 0.3 s
Performance Matrix Multiplication 4096 x 4096

- Java, 1 Thread: 855 s
- JaMP, 7 Threads: 254 s
- JCudaMP: 17 s
- JCudaMP Array Pkg: 15 s
- JCudaMP Array Pkg Rectangular: 6 s
- NVIDIA: 1 s
Convolution - Matrix 4096 x 4096, Kernel 16 x 16

- Java, 1 Thread: 43 s
- JaMP, 7 Threads: 20 s
- JCudaMP: 7 s
- JCudaMP Array Pkg: 4 s
- JCudaMP Array Pkg: 4 s
- NVIDIA: 1 s
Performance Matrix-Multiplication 7500 x 7500

- JaMP: 7 Threads, 1019 s
- JCudaMP + tiling limited to 500MB GPU memory: 130 s
- JCudaMP + tiling 1GB GPU memory: 99 s
Summary

- JCudaMP leads to a speedup of 80 or higher

- For short program run-times the compilation time is too high
  - JIT?
  - OpenCL?

- Use developer knowledge
  - The array package annotation avoids unnecessary copy operations and minimizes the copy overhead
  - The tiling annotation solves the limited GPU memory issue for most applications
Thank you for your attention.