JCudaMP: OpenMP/Java on CUDA

Georg Dotzler, Ronald Veldema, Michael Klemm

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Programming Systems Group • Martensstraße 3 • 91058 Erlangen

Motivation

„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 M**ulti-**P**rocessing
    - 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
    ```

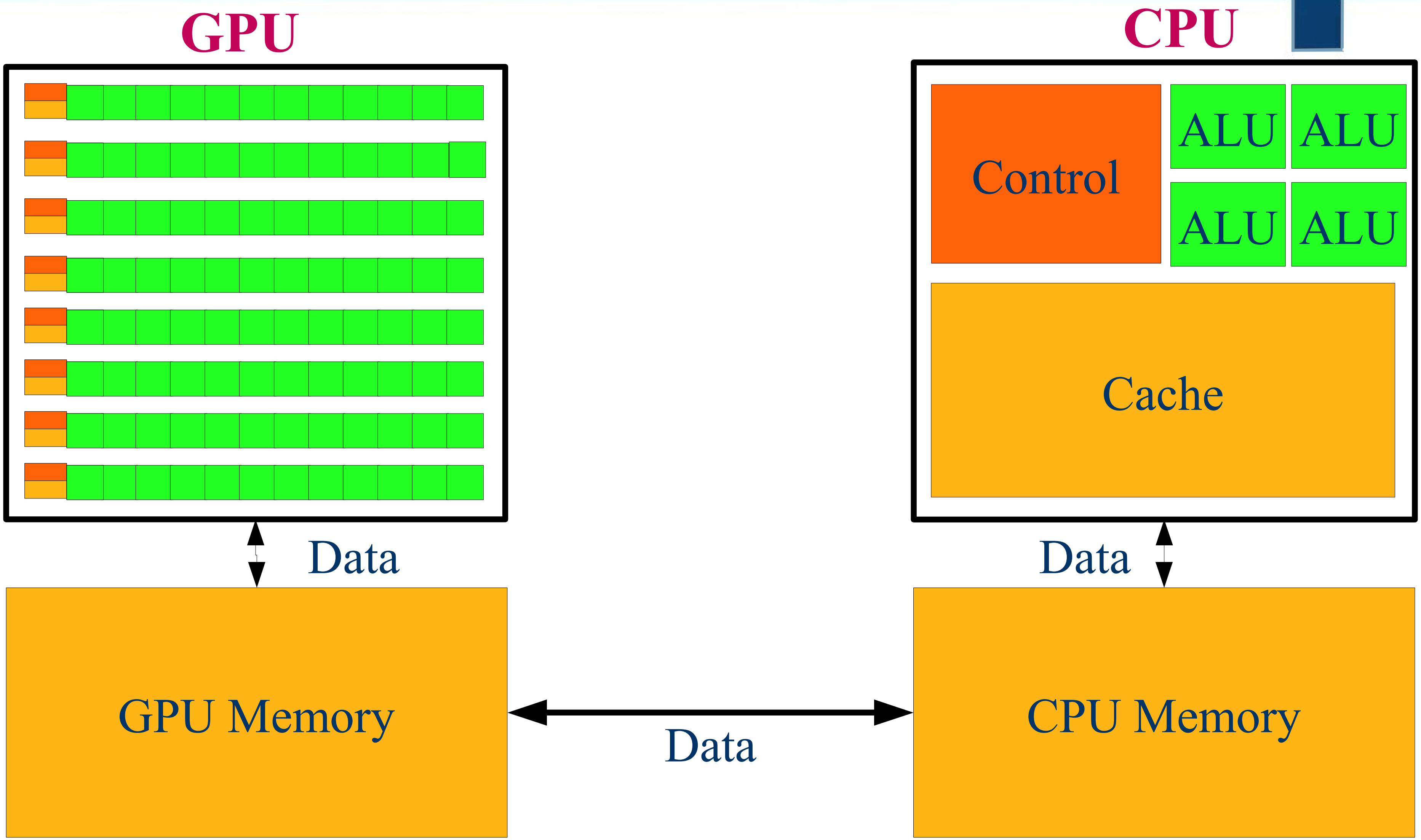
Introduction - CUDA

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

- Example:

```
__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



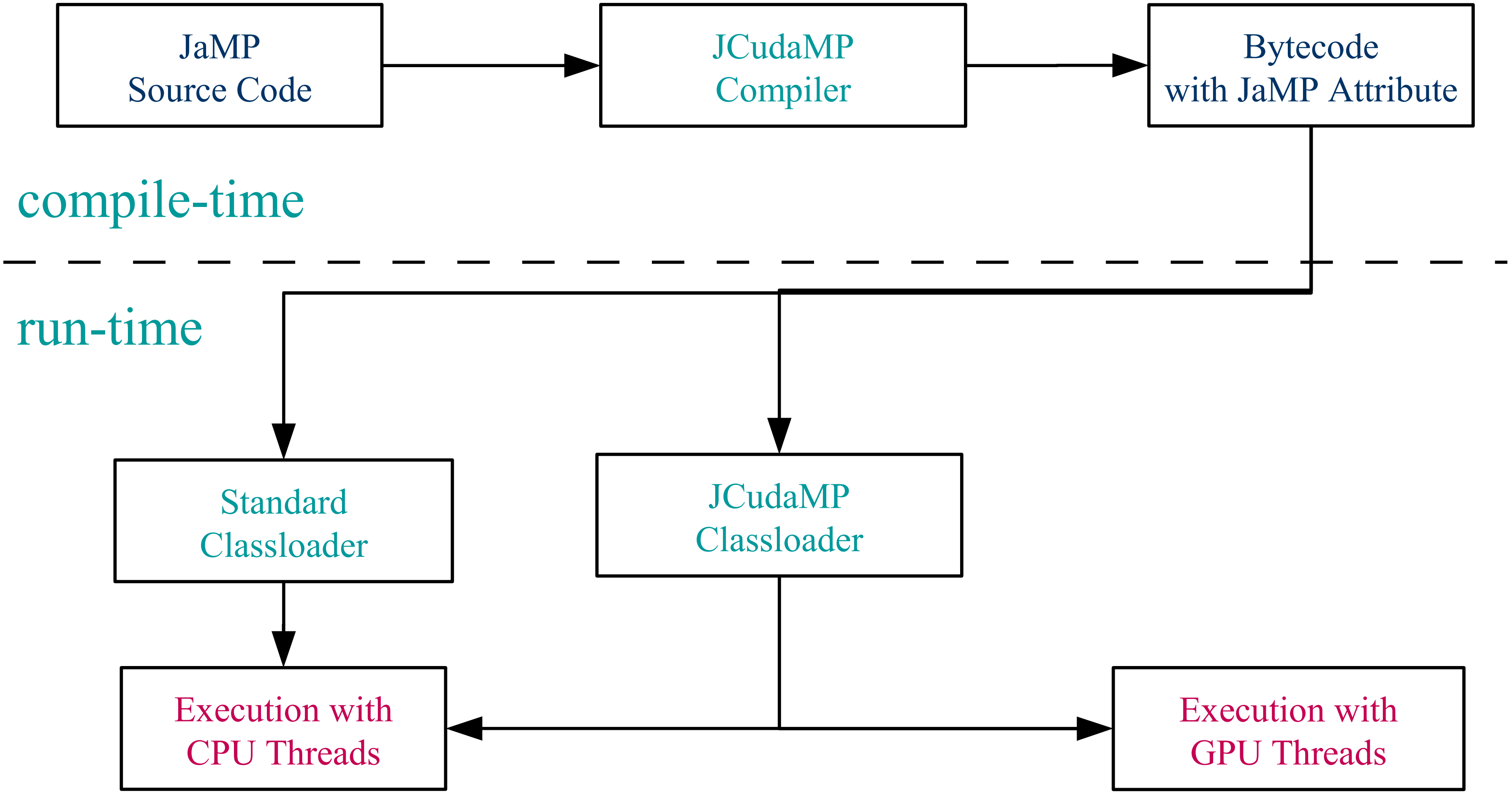
Transparent Use of CUDA I

- 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

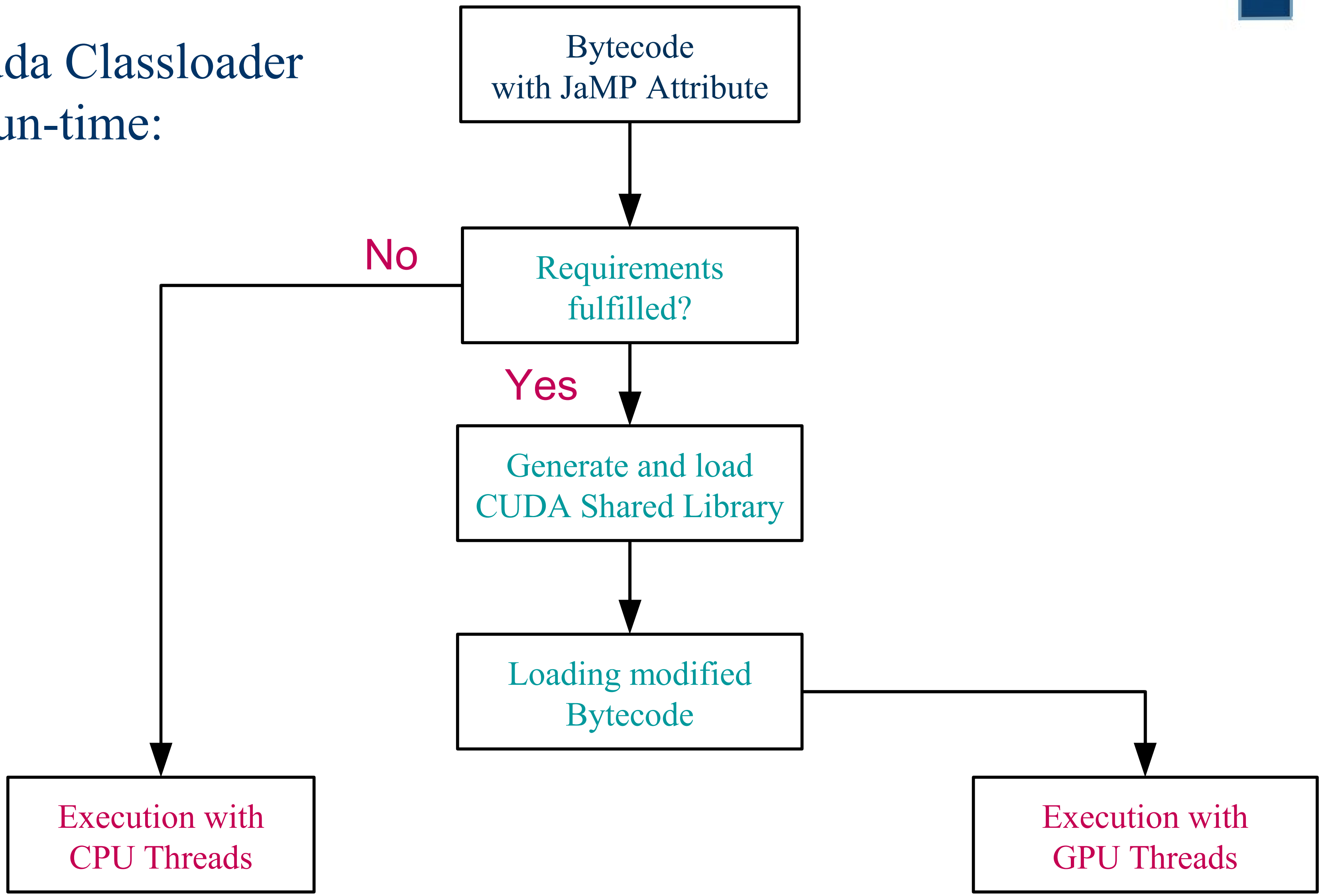
- Solution:
  - Use of JaMP
    - Annotations defined in comments
    - Suitable parallel regions are marked with //#omp parallel for
    - Compilation in regular Java compilers possible
  - Use of additional bytecode section
    - Information is ignored by a standard classloader
    - Contains information about ...
    - the hardware requirements (long, double, float division....)
    - the JaMP annotations

Transparent Use of CUDA III



Transparent Use of CUDA IV

JCuda Classloader at run-time:



GPU Memory Management I

- 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

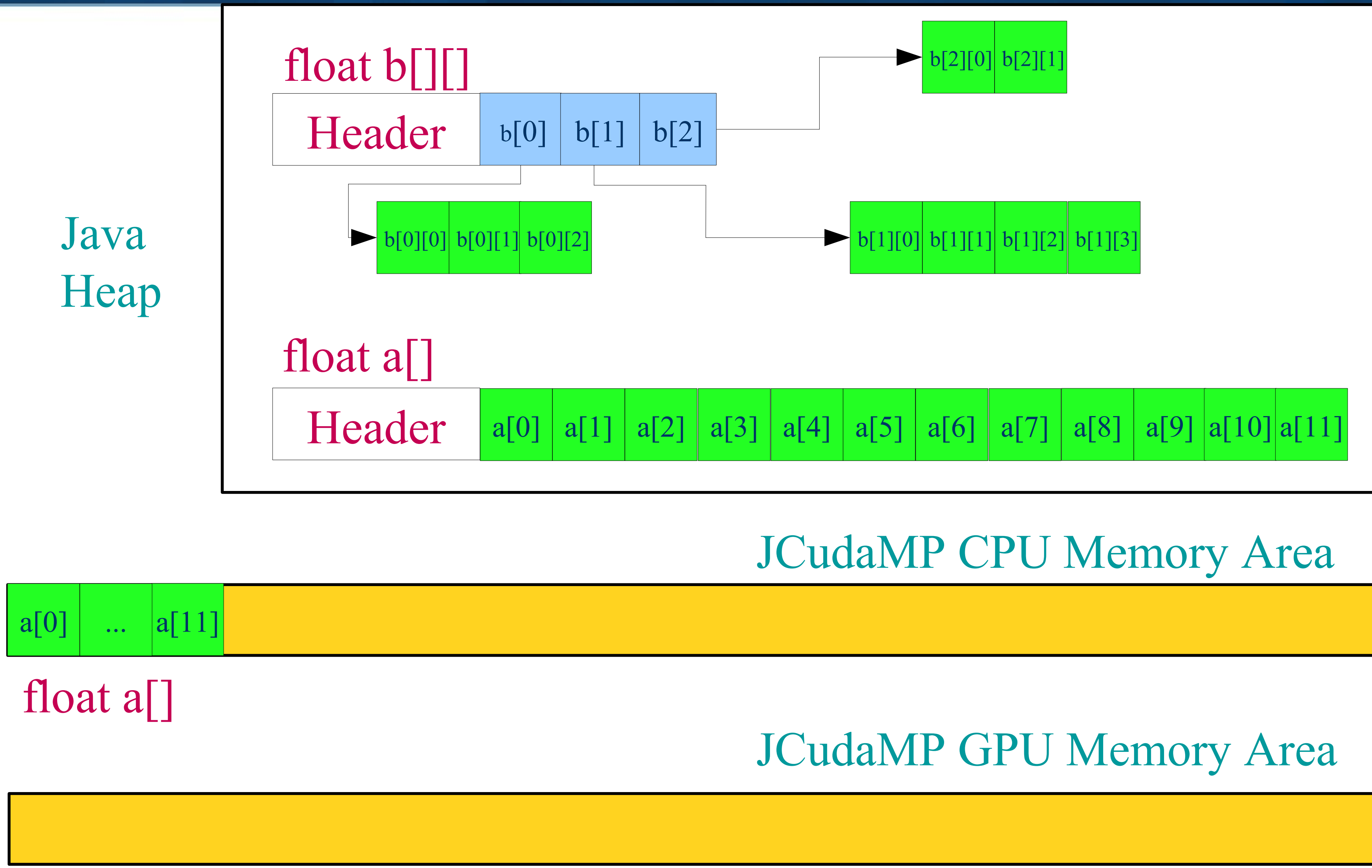
GPU Memory Management II

- Example:

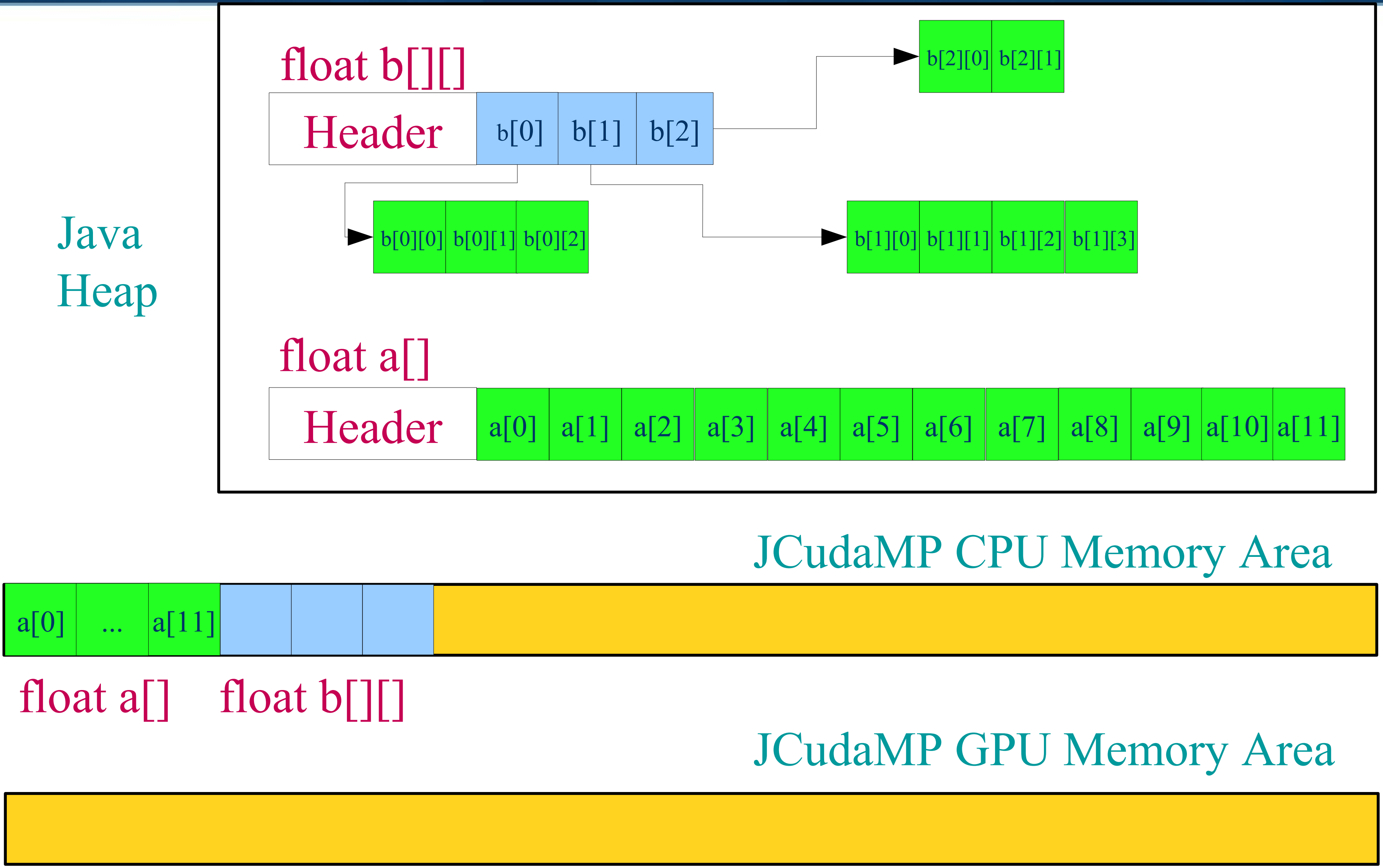
```
float[] a = new float[12];
float[][] b = new float[3][];
b[0] = new float[3];
b[1] = new float[4];
b[2] = new float[2];

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

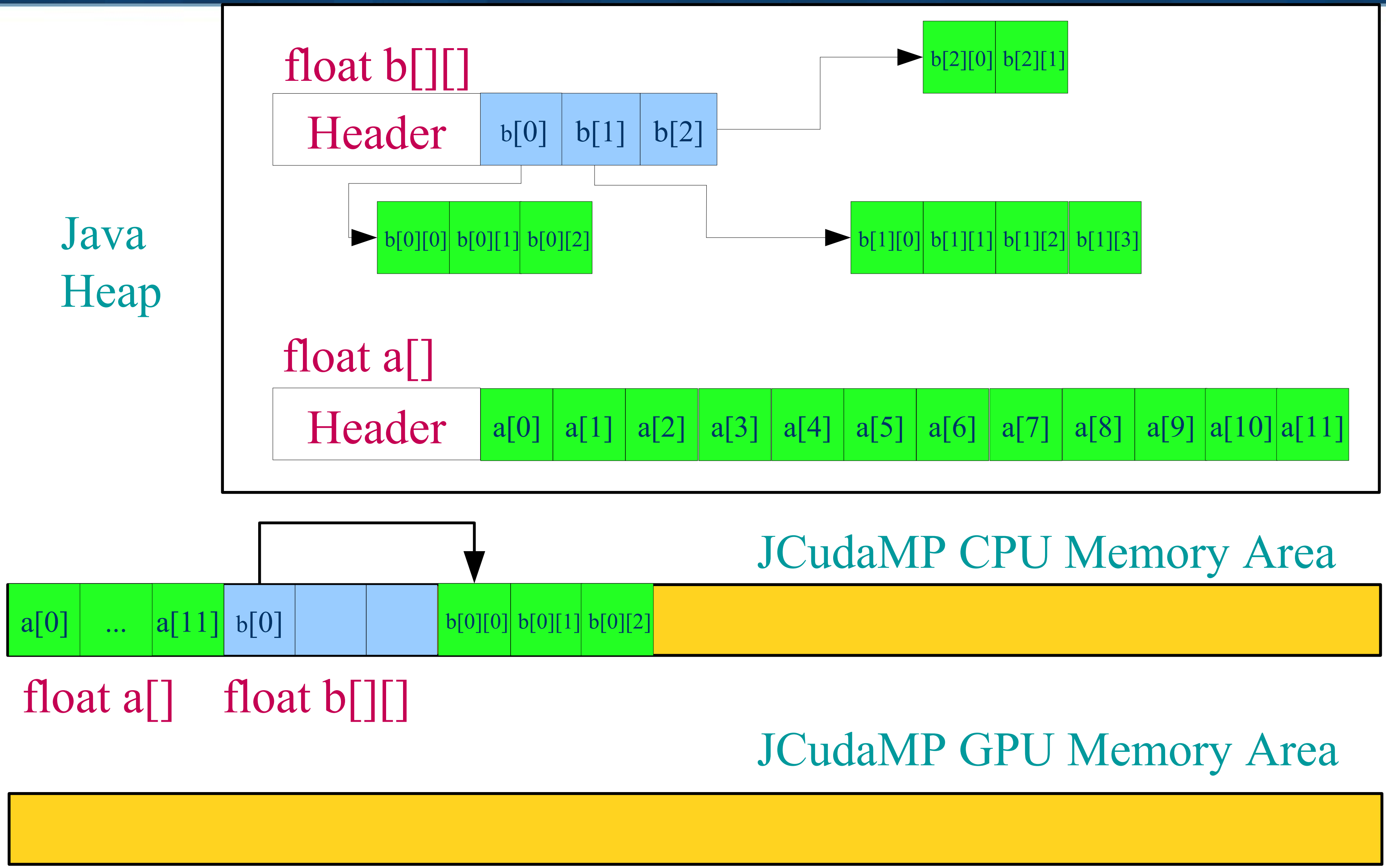
GPU Memory Management III



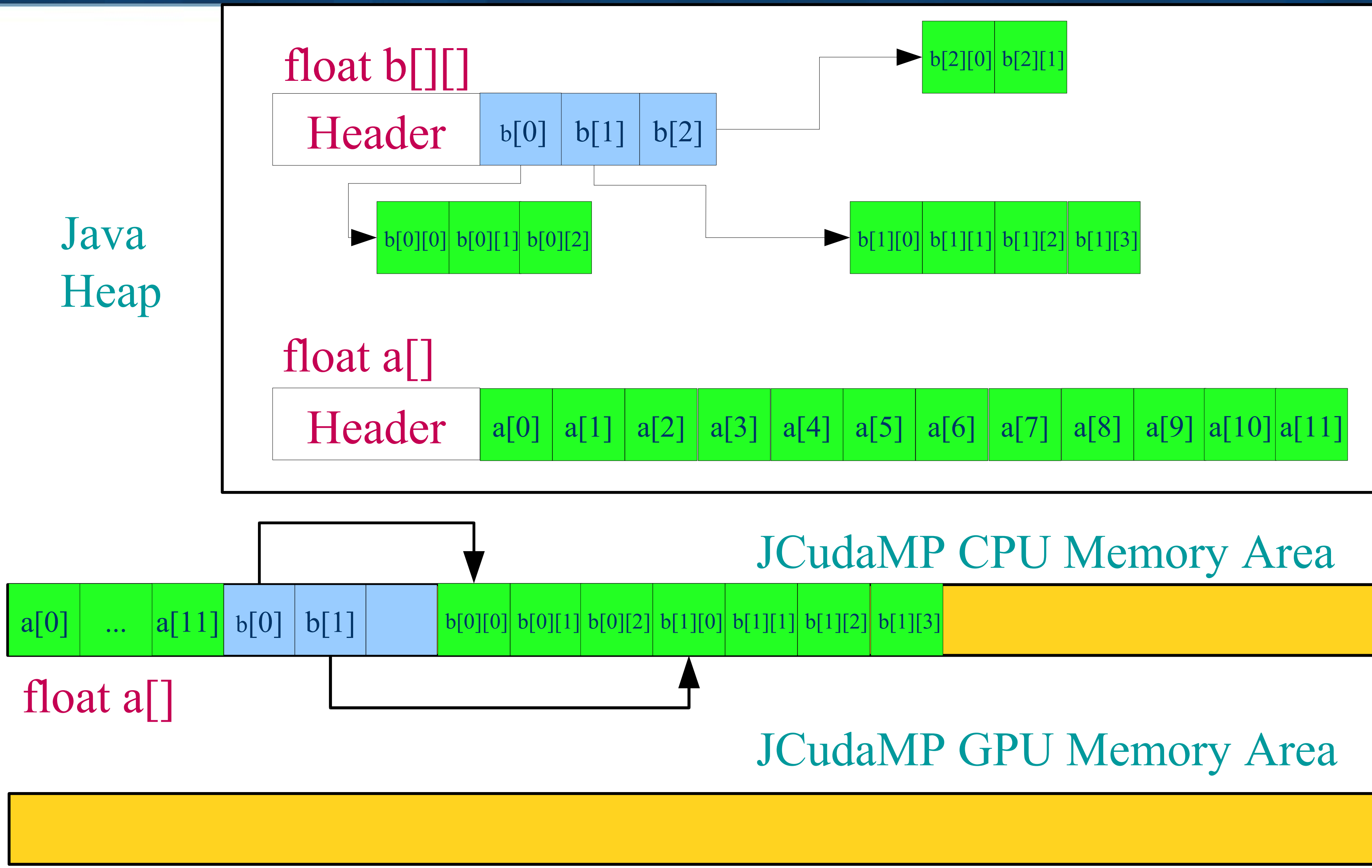
GPU Memory Management IV



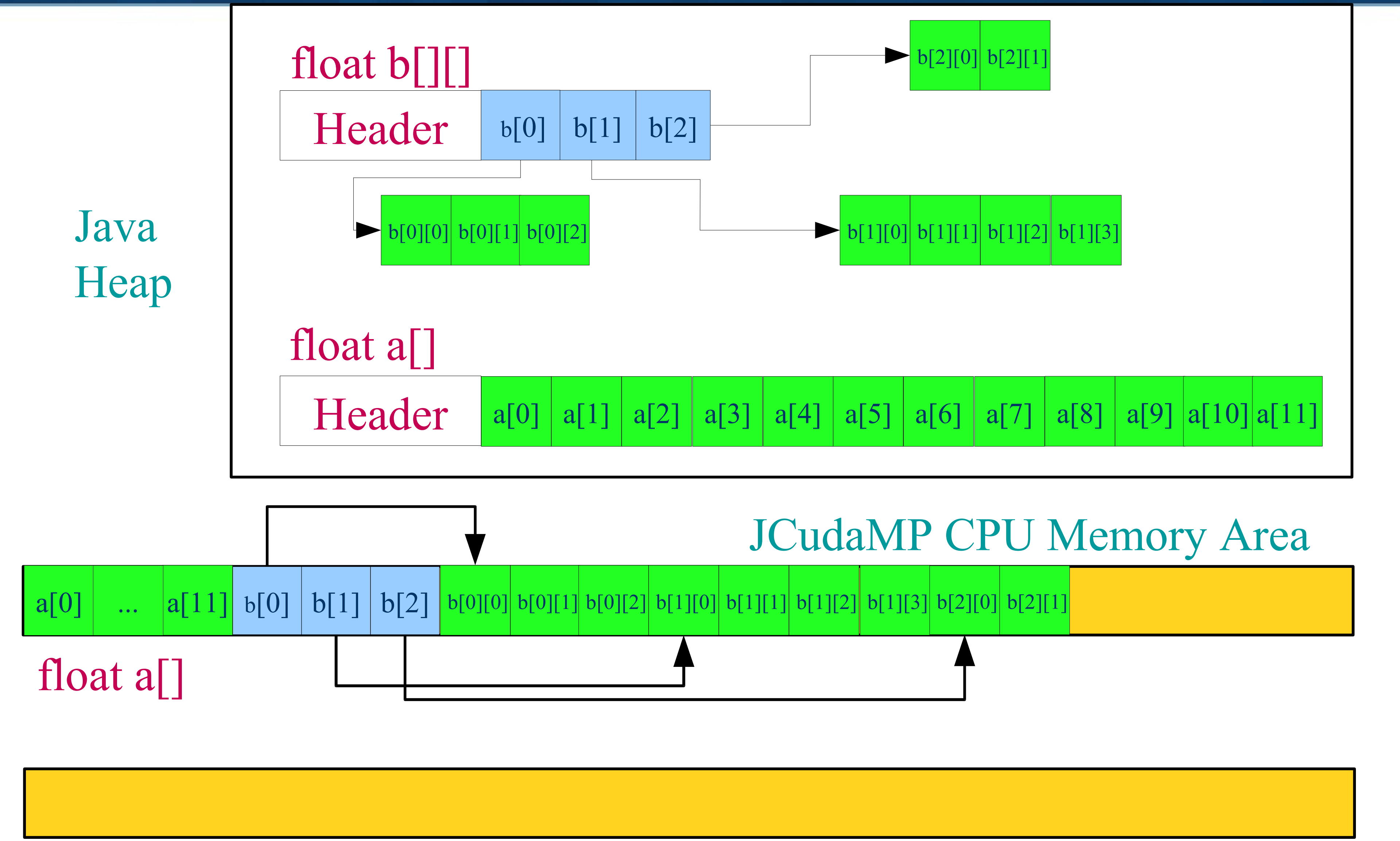
GPU Memory Management V



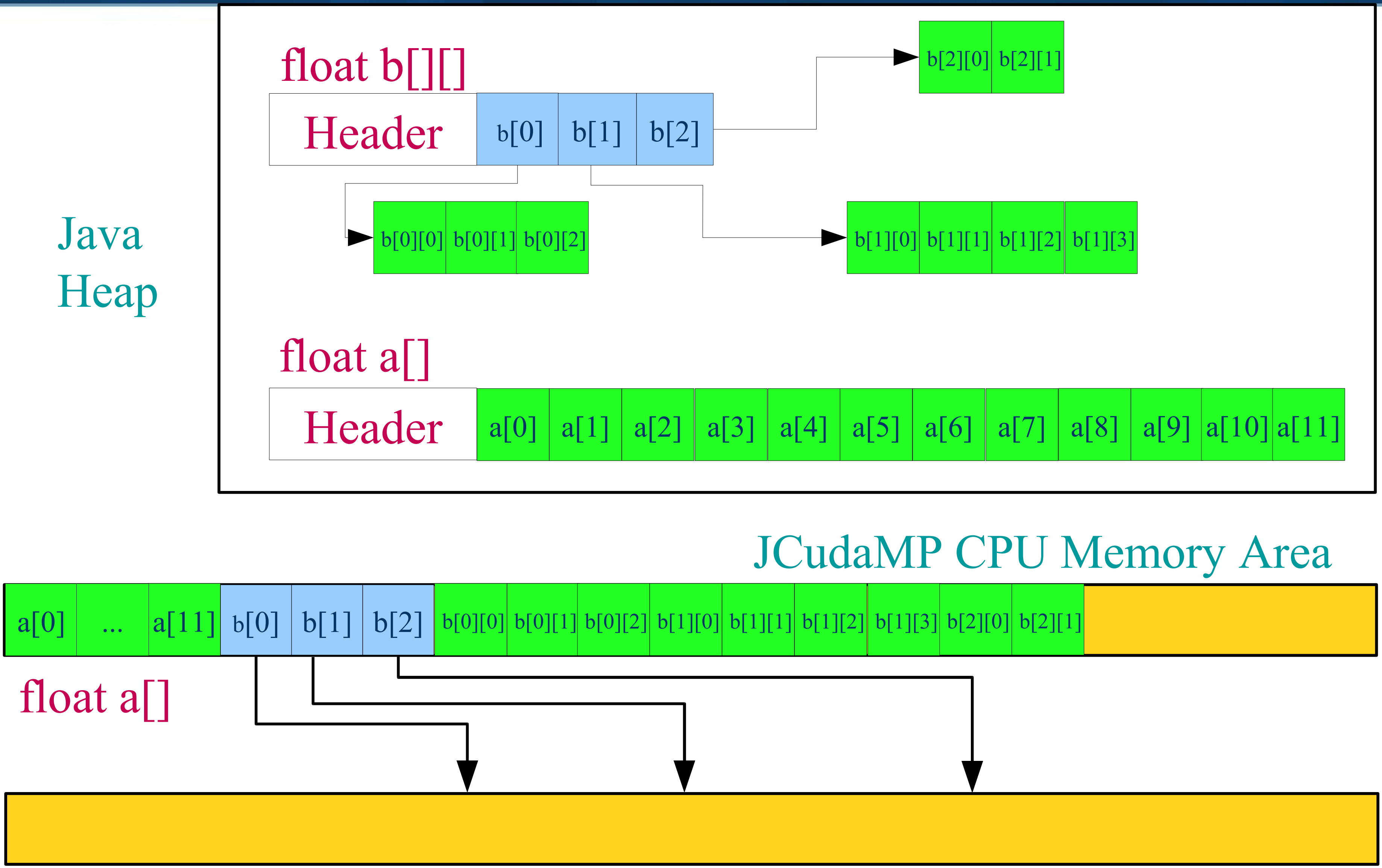
GPU Memory Management VI



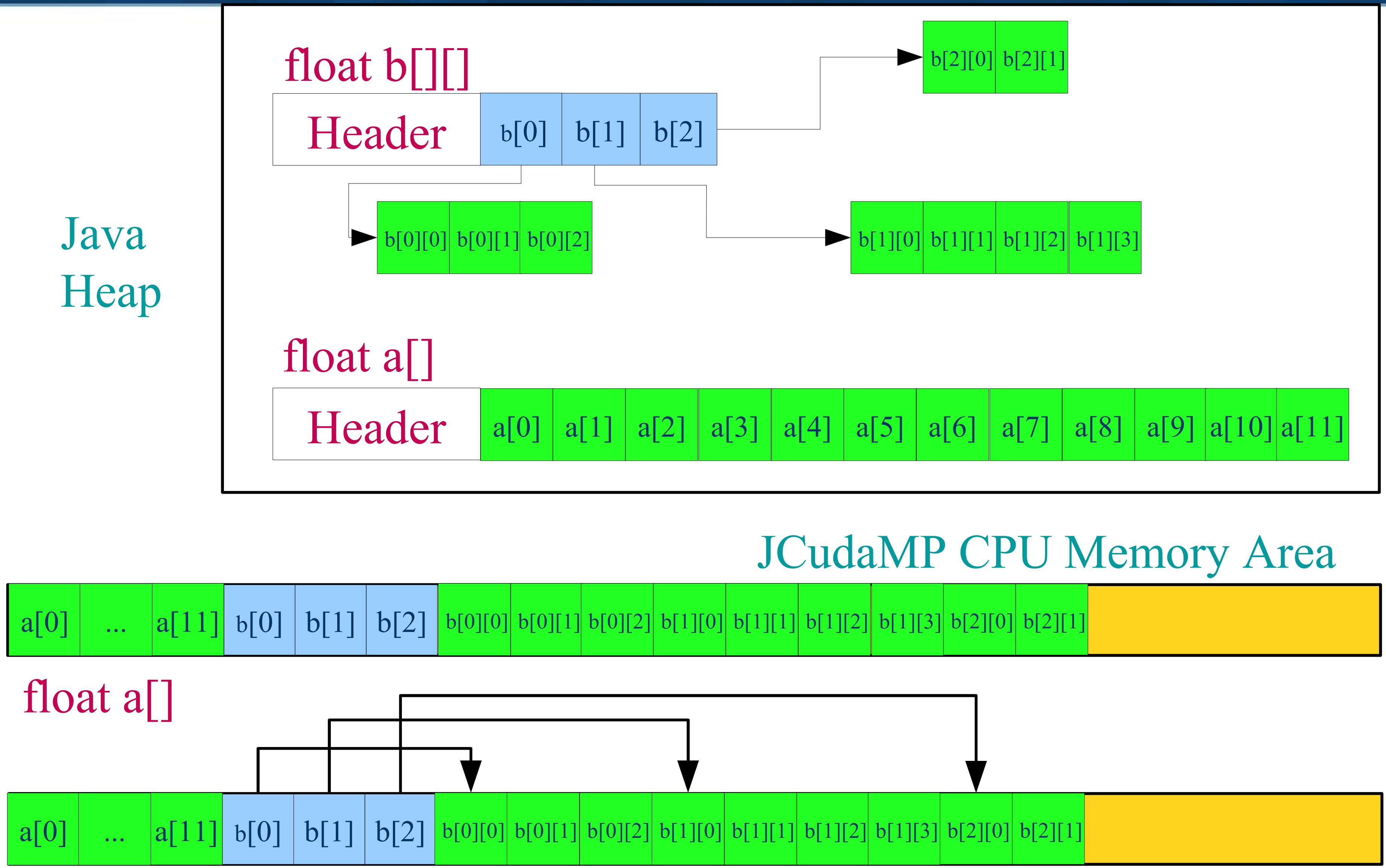
GPU Memory Management VII



GPU Memory Management VIII



GPU Memory Management IX



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

```
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

```
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

Example of Recurring Transfers III

```
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

Only one element is required

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

Memory Management - Array Packages II

- 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:

```
//#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:

```
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:

```
//#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:

```
//#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:

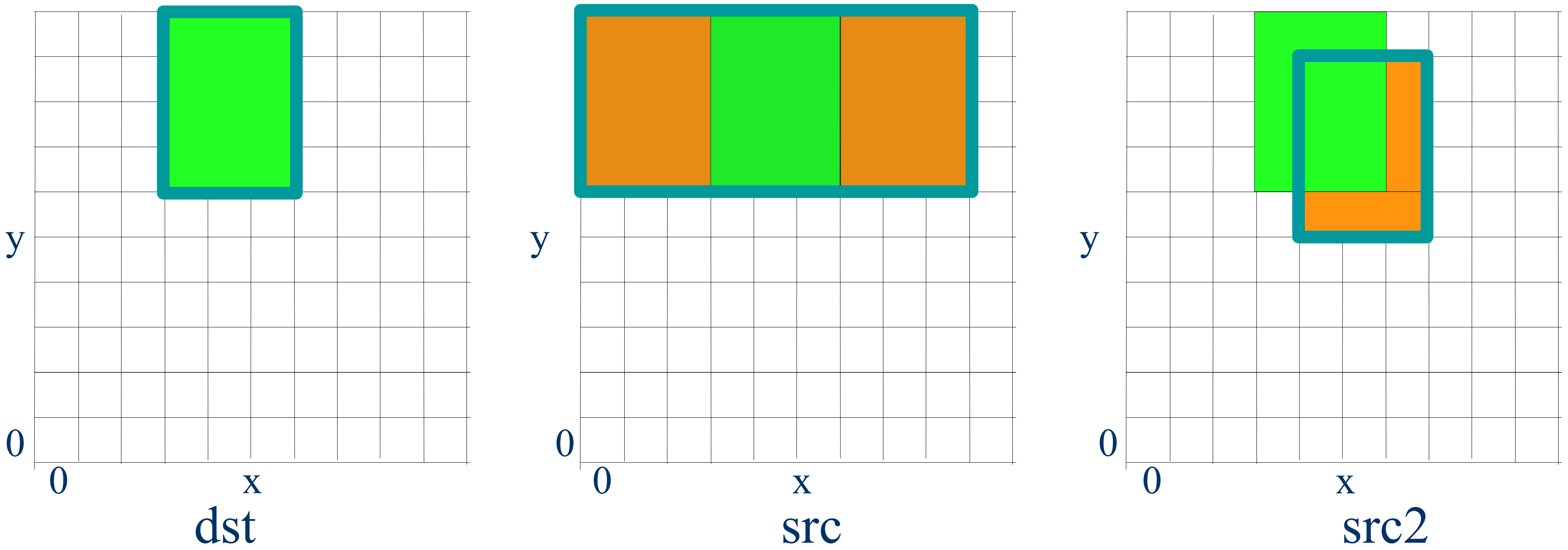
```
//#omp parallel for tile ( dst : { x : 0 , 0 ; y : 0 , 0 } ,
                                   src: { x :−5 , 3 ; y : 0 , 2} ,
                                   src2: { y :−1 , −1; x : 1 , 1})

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 (dst : { x : 0 , 0 ; y : 0 , 0 },
src: { x :−5 , 3 ; y : 0 , 2} ,
src2: { y :−1 , −1; x : 1 , 1})

Block x: 3 – 5
Block y: 6 – 9

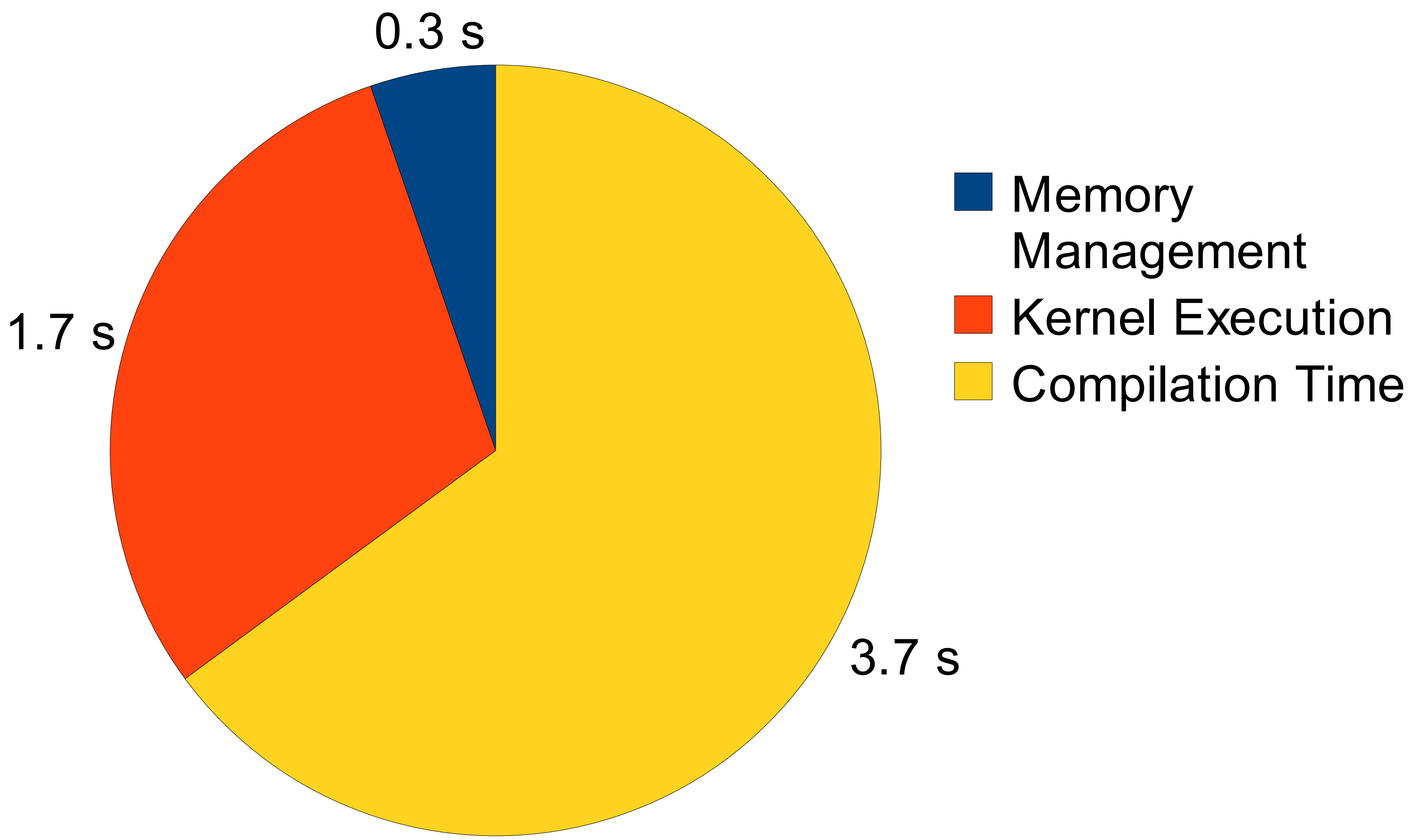


Performance Test Environment

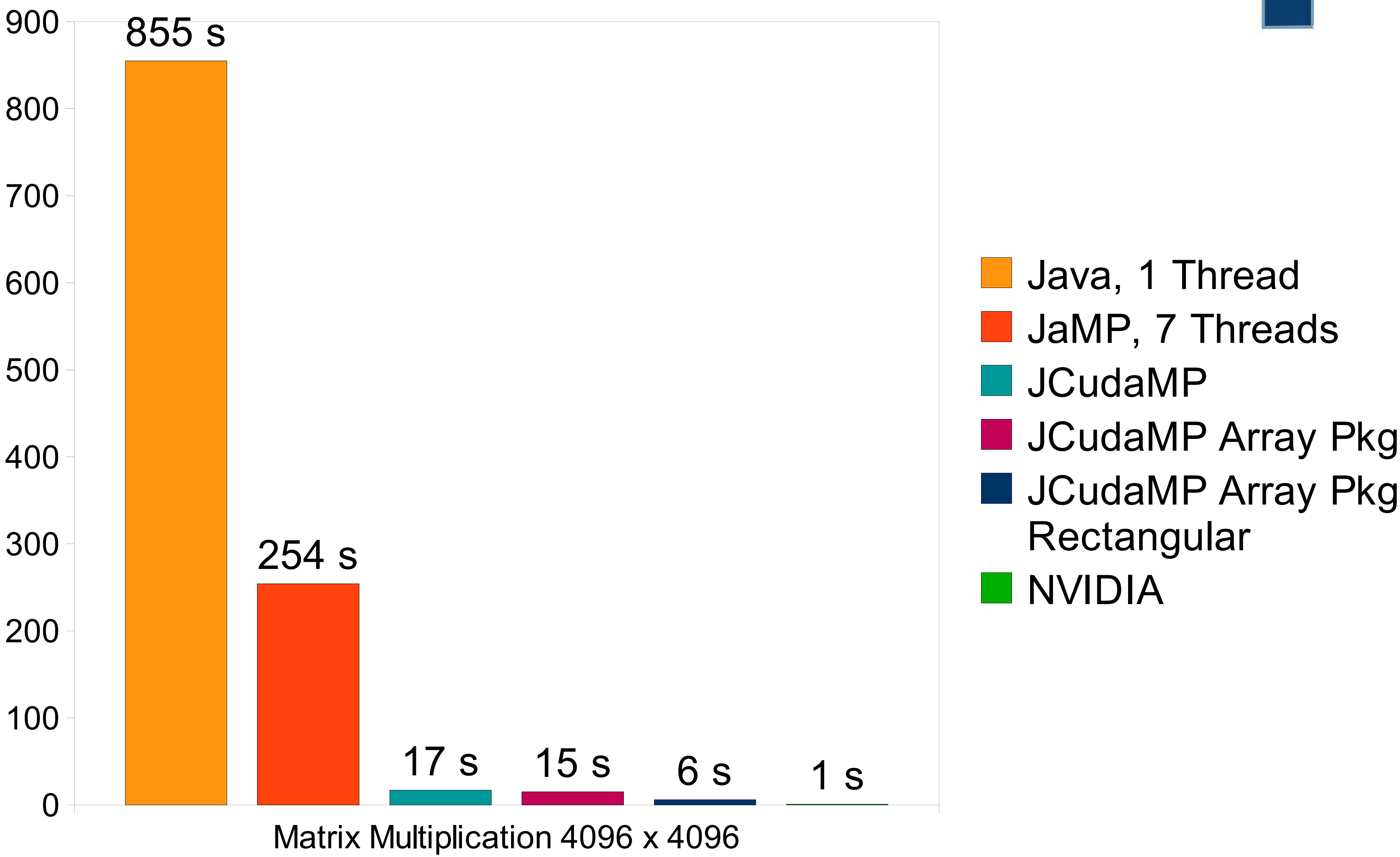
- CPU: Dual-Quadcore Intel Xeon L5420 (2.5 Ghz)
- Memory: 16 GB
- GPU: GeForce GTX 280 SC (1 GB Memory)
- OS: Linux, kernel 2.6.24
- Compiler: CUDA 2.2, GCC version 4.2.4
- JVM: Java Hotspot JIT in Version 1.6.0_13

Performance Matrix Multiplication 2048 x 2048

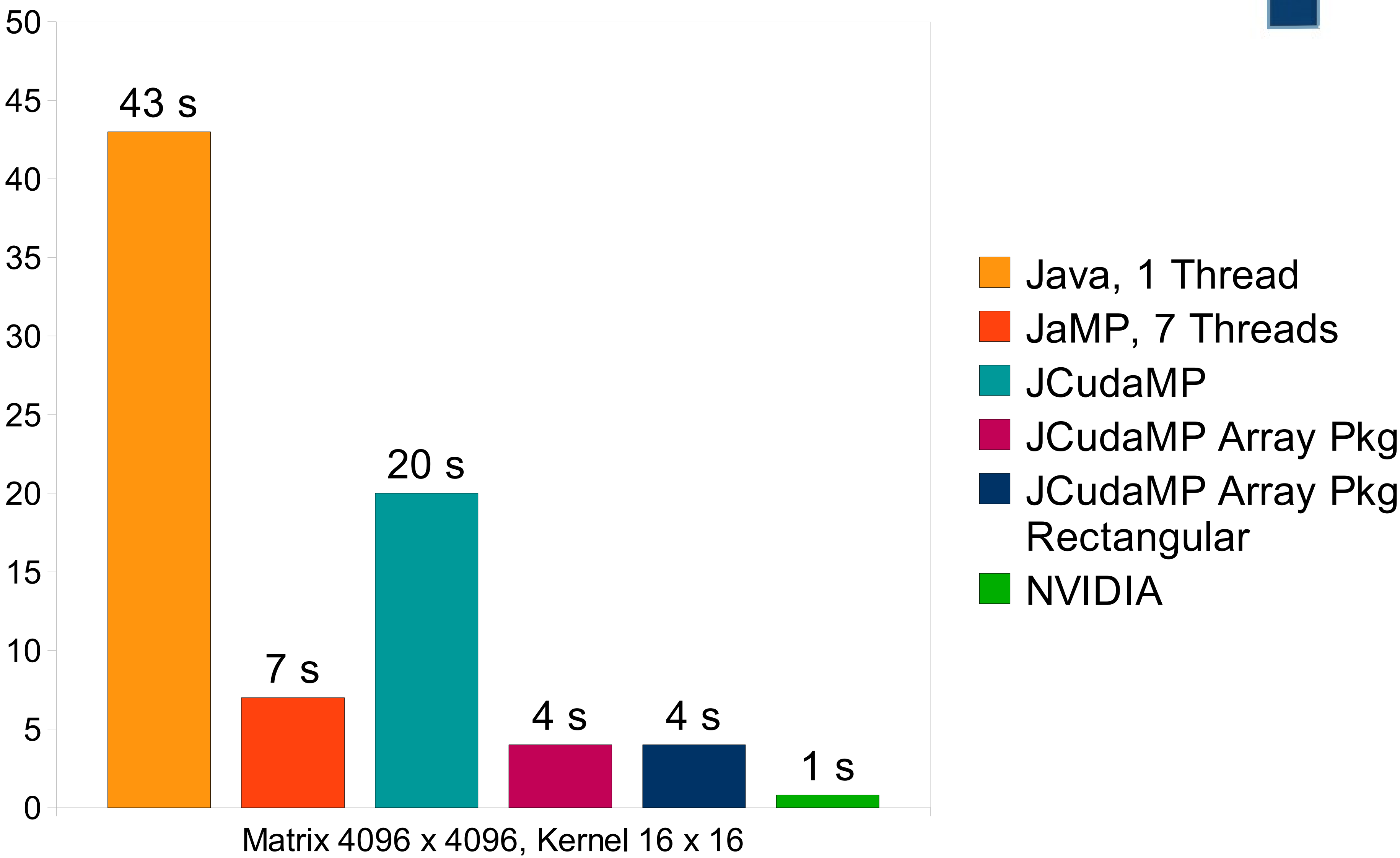
- Run-time comparison of the different program parts



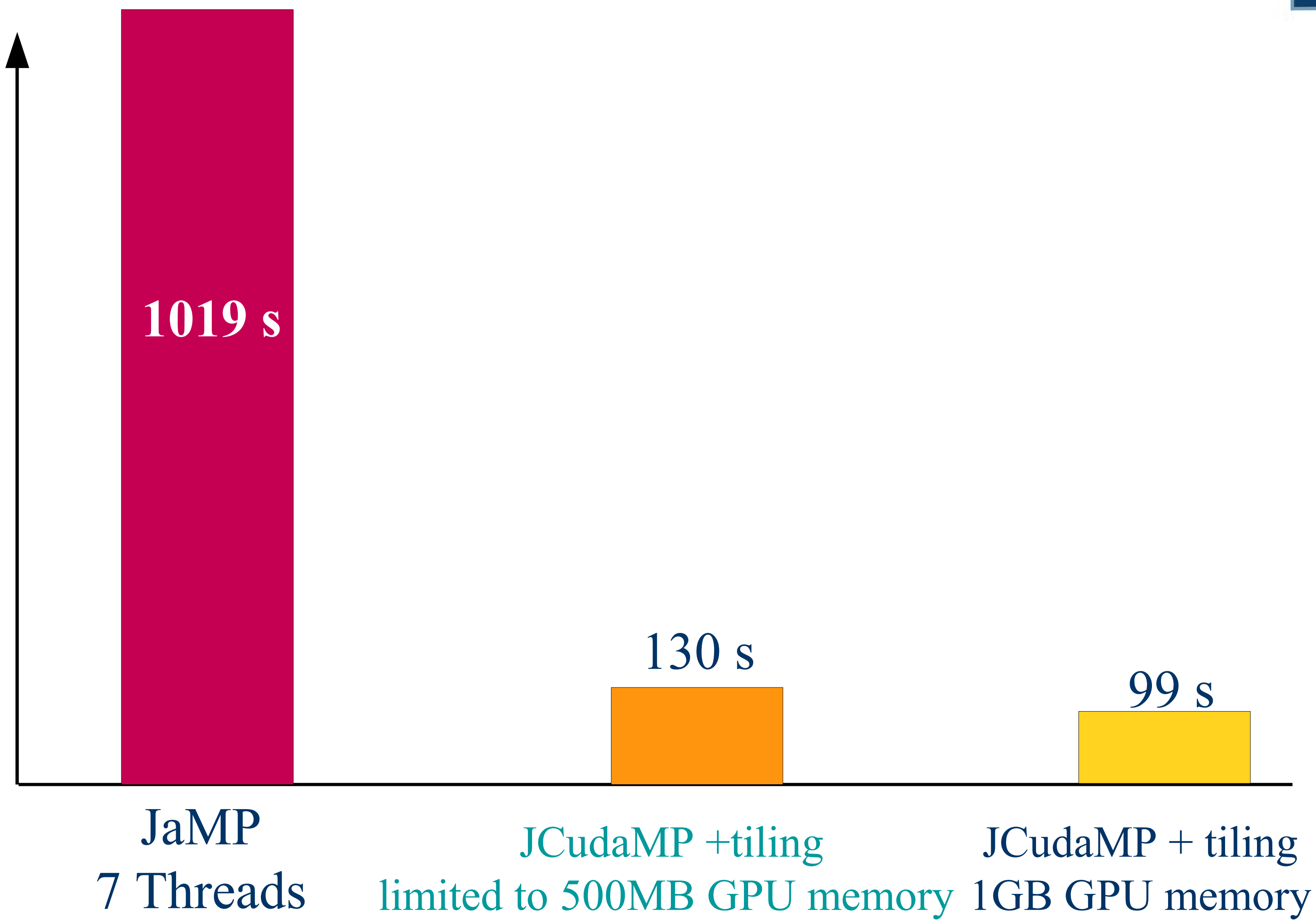
Performance Matrix Multiplication 4096 x 4096



Convolution - Matrix 4096 x 4096, Kernel 16 x 16



Performance Matrix-Multiplication 7500 x 7500



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.