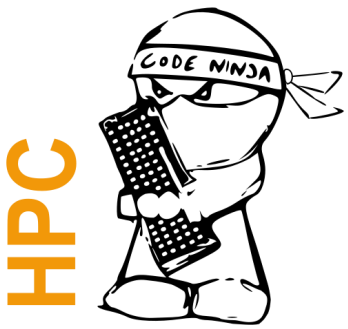
Introduction to Parallel Programming (w/ JAVA)

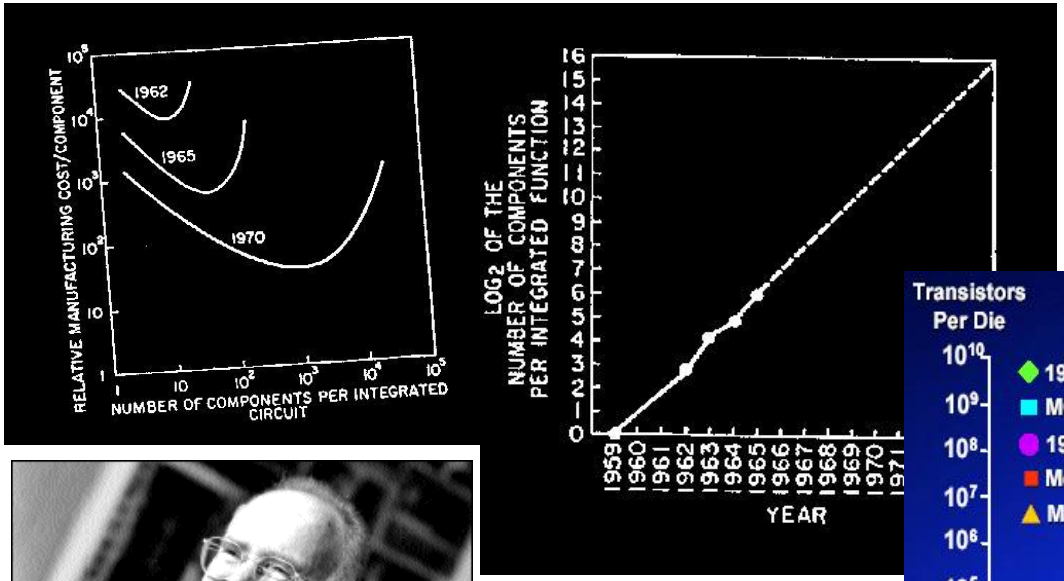
Dec. 21st, 2015

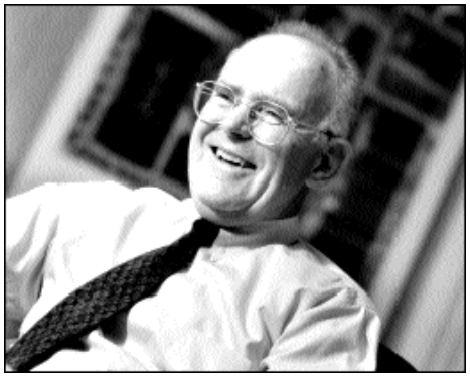


Christian Terboven
IT Center, RWTH Aachen University

terboven@itc.rwth-aachen.de

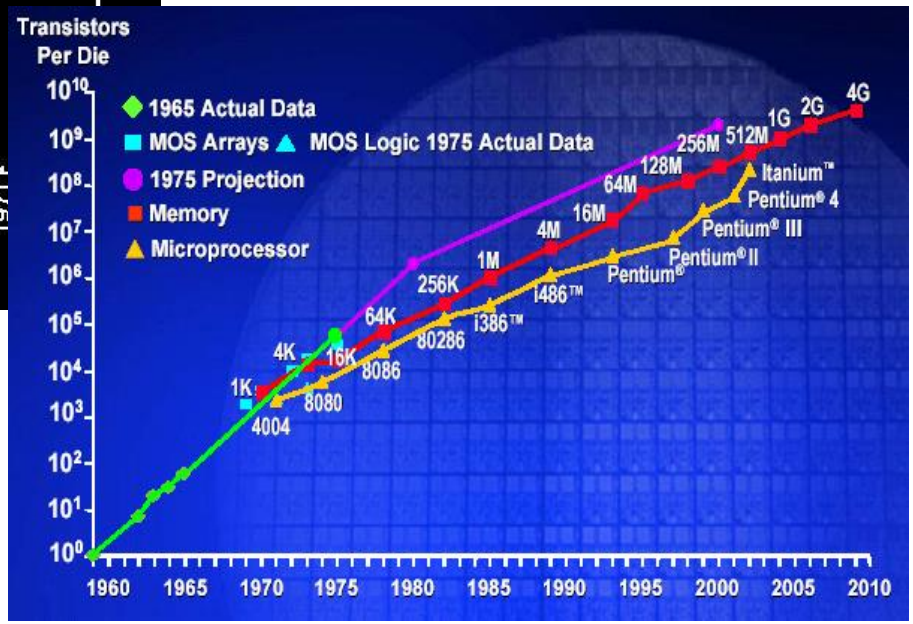
Moore‘s Law





“Cramming More Components onto Integrated Circuits”

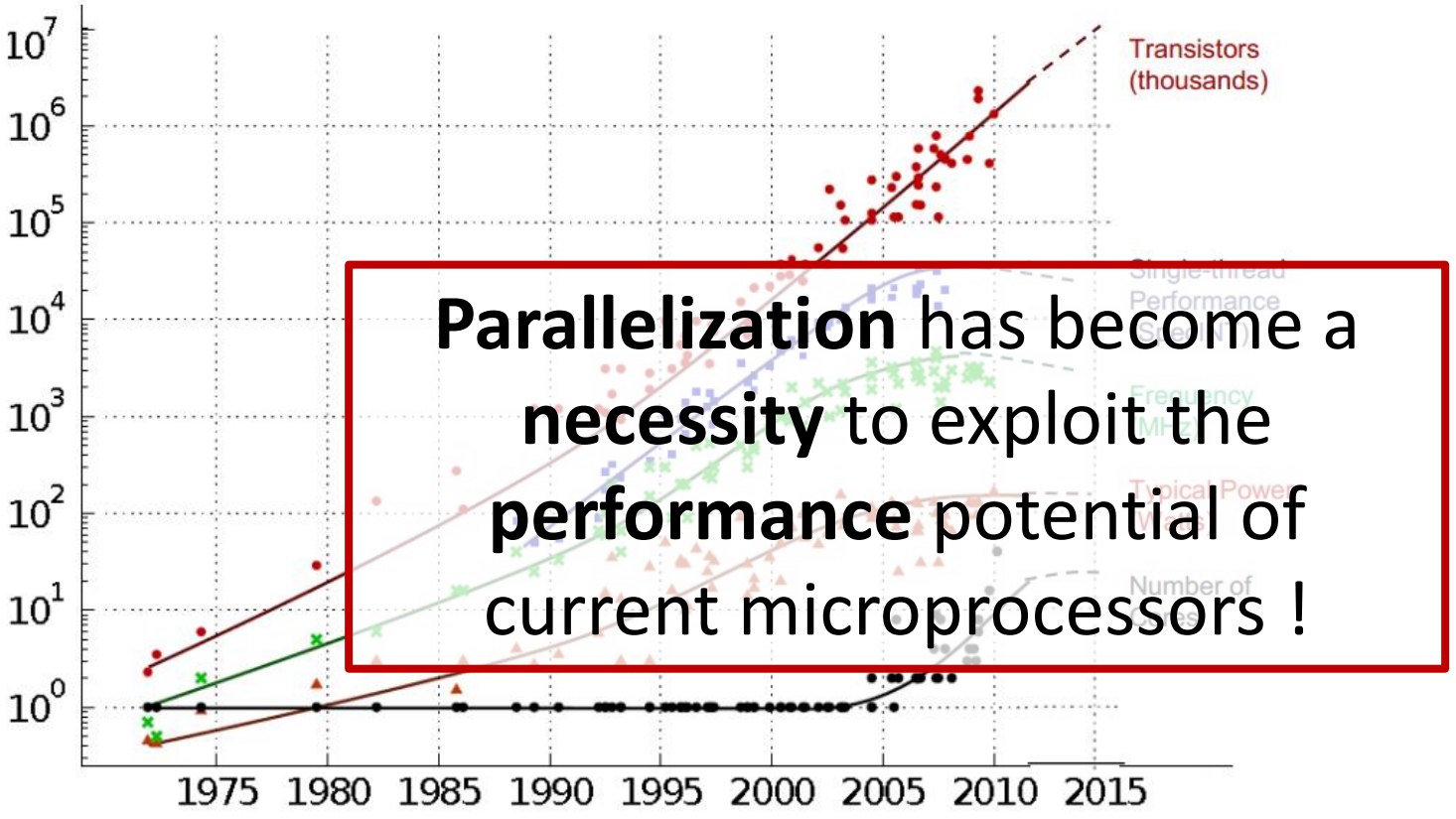
Gordon Moore, Electronics, 1965
(ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf)



- **# of transistors / cost-effective integrated circuit double every N months (12 <= N <= 24)**

There is no free lunch anymore

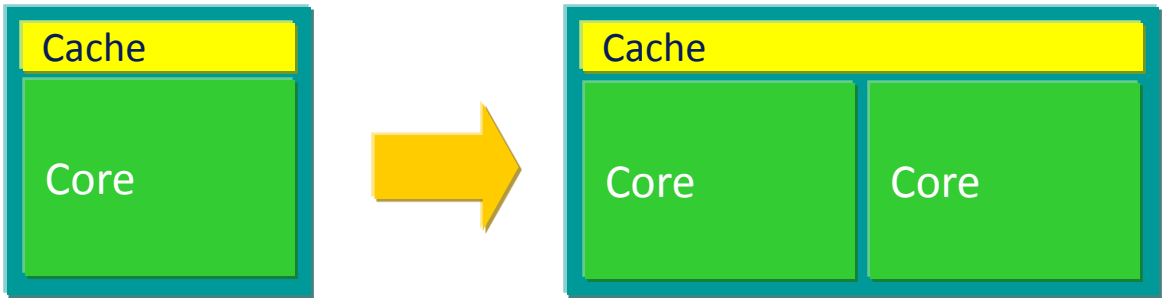
- **The number of transistors on a chip is still increasing, but no longer the clock speed! Instead, we see many cores per chip.**



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Multi-Core Processor Design

- **Rule of thumb: Reduction of 1% voltage and 1% frequency reduces the power consumption by 3% and the performance by 0.66%.**



Voltage	= 1	Voltage	= -15%
Freq	= 1	Freq	= -15%
Area	= 1	Area	= 2
Power	= 1	Power	= ~1
Perf	= 1	Perf	= ~1.8

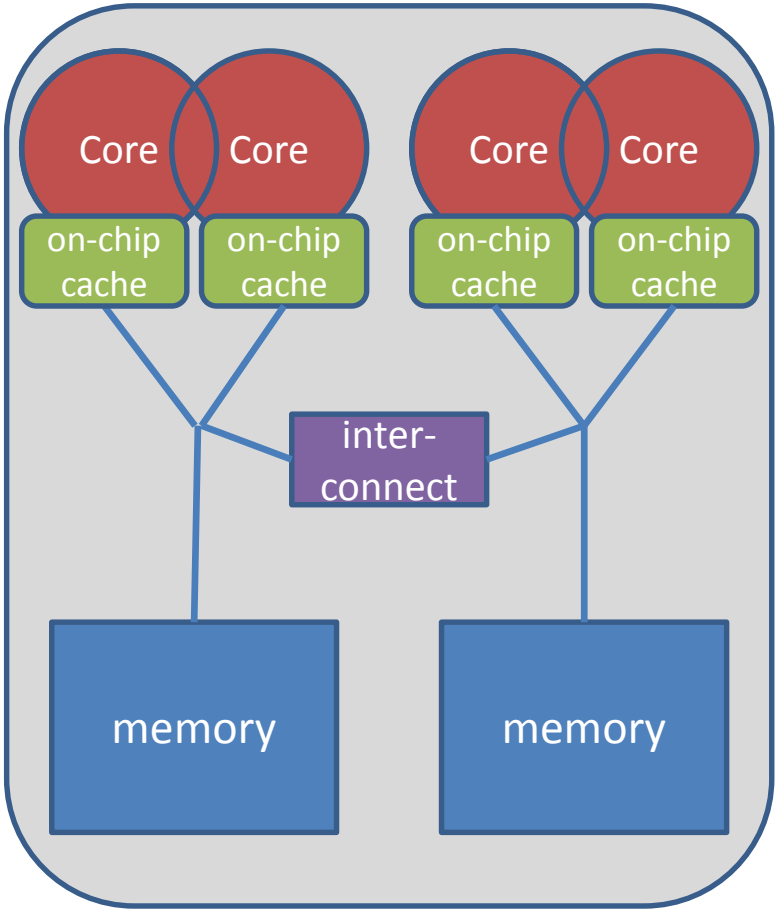
(Based on slides from Shekhar Borkar, Intel Corp.)

Multi-Core Multi-Socket Compute Node Design

- **Set of processors is organized inside a locality domain with a locally connected memory.**

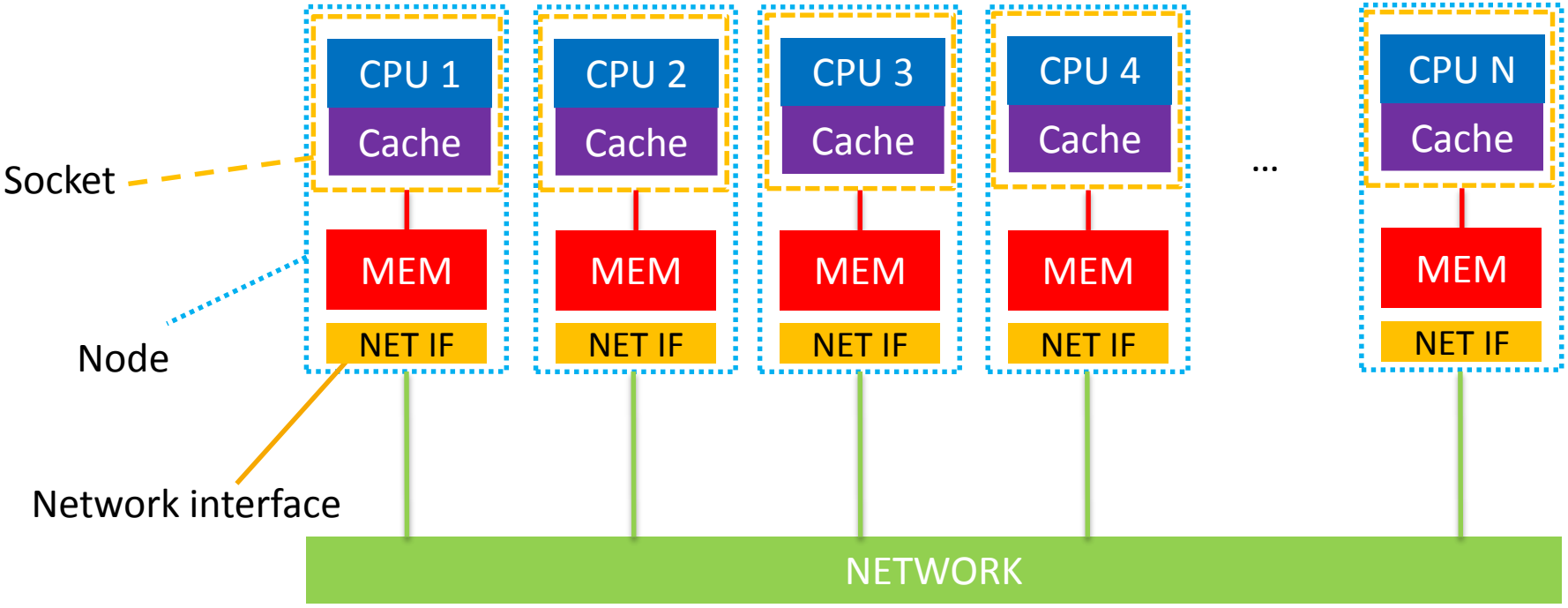
  → The memory of all locality domains is accessible over a shared virtual address space.

  → Other locality domains are access over a interconnect, the local domain can be accessed very efficiently without resorting to a network of any kind



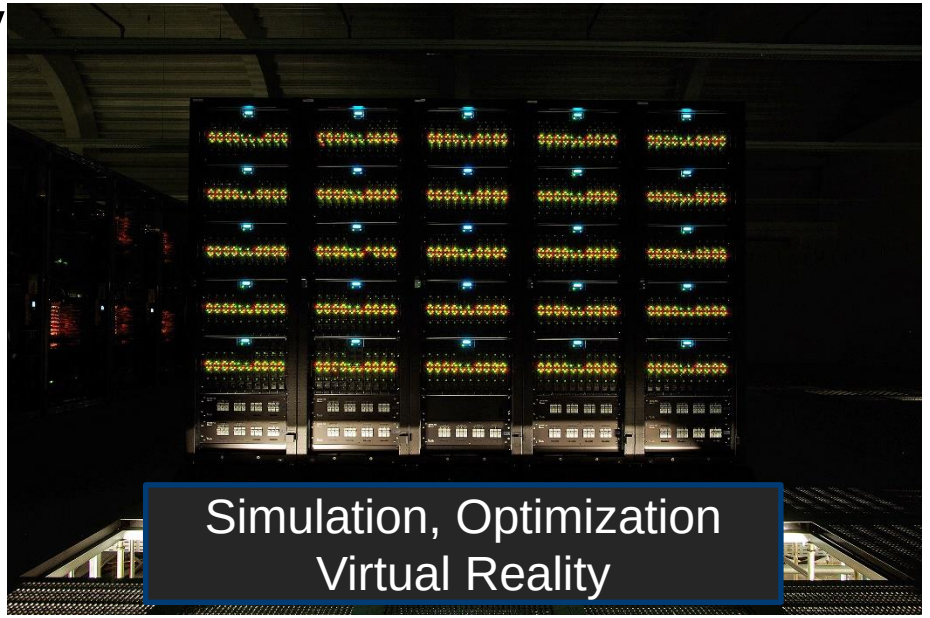
Multi-Core Multi-Socket Compute Node Cluster Design

- **System where memory is distributed among “nodes”**
  - → No other node than the local one has direct access to the local memory



What is High Performance Computing (HPC)?

- **From Wikipedia:**
  “A supercomputer is a computer at the frontline of current processing capacity, particularly speed of calculation.”



- **Historically there were two principles of science: Theory and Experiment. Computational Science extends them as a third**

Theory
Models, Differential Equations, linear equation systems

Experiments
Observation and prototypes
empirical studies/sciences

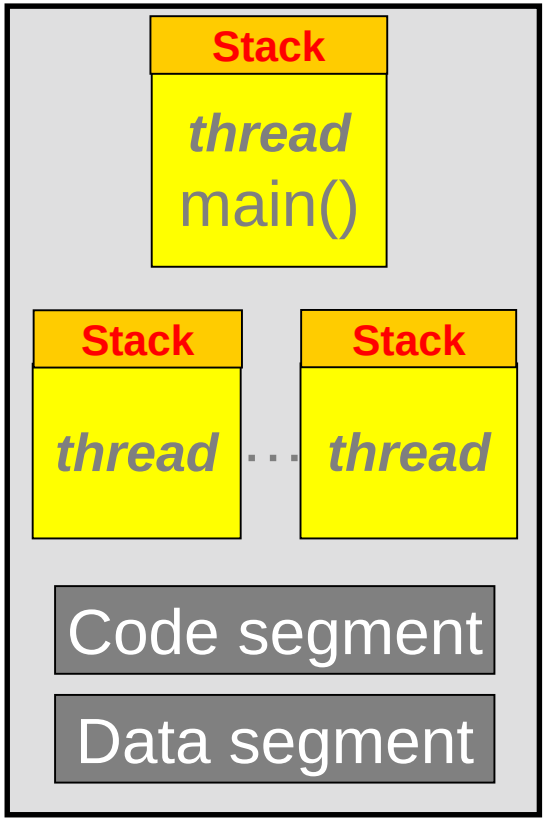
Agenda

- I hope you are motivated by now ☺
- **Basic Concepts of Threading**
- **Matrix Multiplication: from Serial to Multi-Core**
- **Amdahl‘s Law and Efficiency**
- **Matrix Multiplication Reviewed**
- **Basic GPGPU Concepts**
- **Matrix Multiplication: from Serial to Many-Core**
- **Summary**
- **Christmas Exercise**

Basic Concepts of Threading

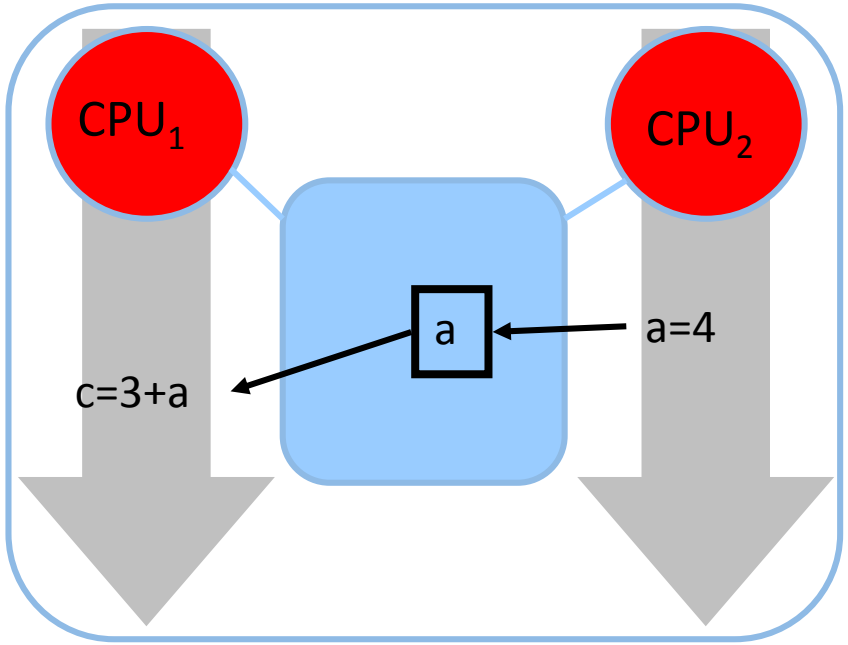
Processes vs. Threads

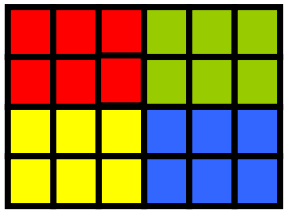
- Applications have to create a team of threads to run on multiple cores simultaneously
- Threads share the global (shared) data of the program, typically the data on the heap
- Every thread has its own stack, which may contain private data only visible to the thread
- Operating systems and/or programming languages offer facilities to creat and manage threads in the application



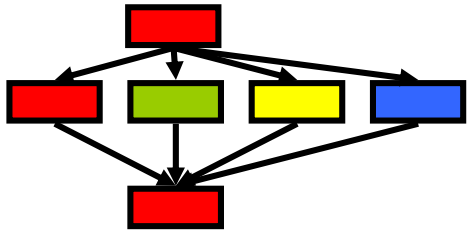
Shared Memory Parallelization

- **Memory can be accessed by several threads running on different cores in a multi-socket multi-core system:**





Decompose data into distinct chunks to be processed independently (data parallelism)



Look for tasks that can be executed simultaneously (task parallelism)

Parallel Programming in Theory and Practice

- **Parallelism has to be exploited by the programmer…**

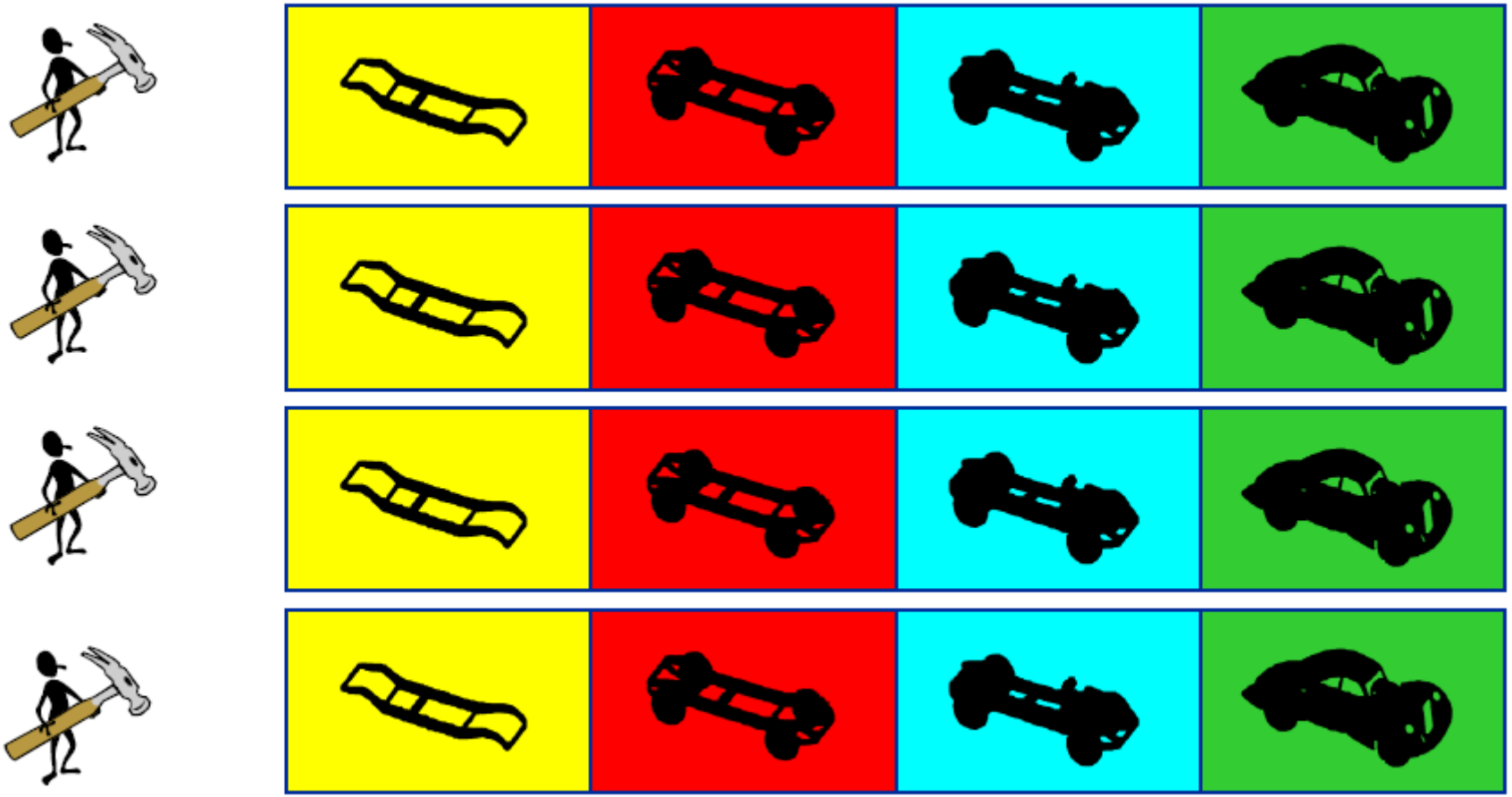
Theory

Practice

Example of parallel work

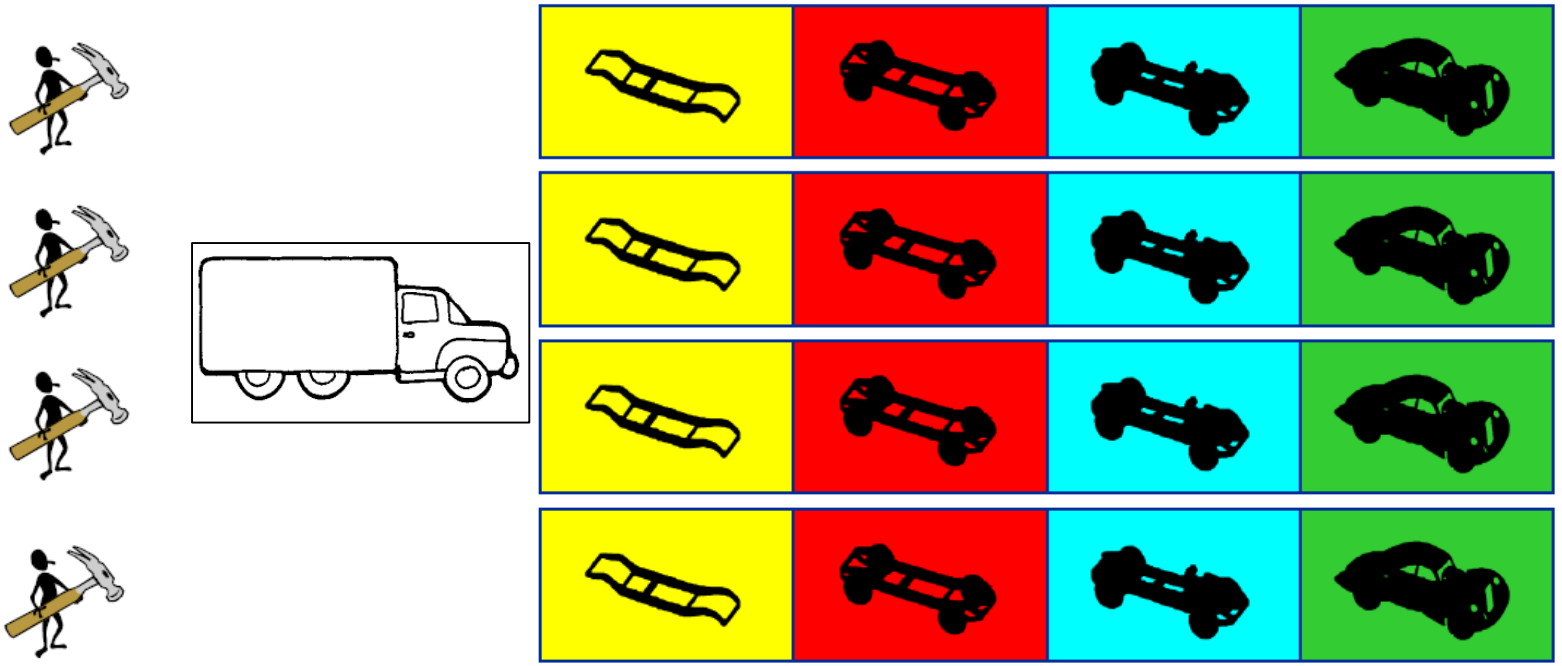
- **Example: 4 cars are produced in parallel**

Prof. Dr. G. Wellein, Dr. G. Hager,
Uni Erlangen-Nürnberg



Limits of scalability

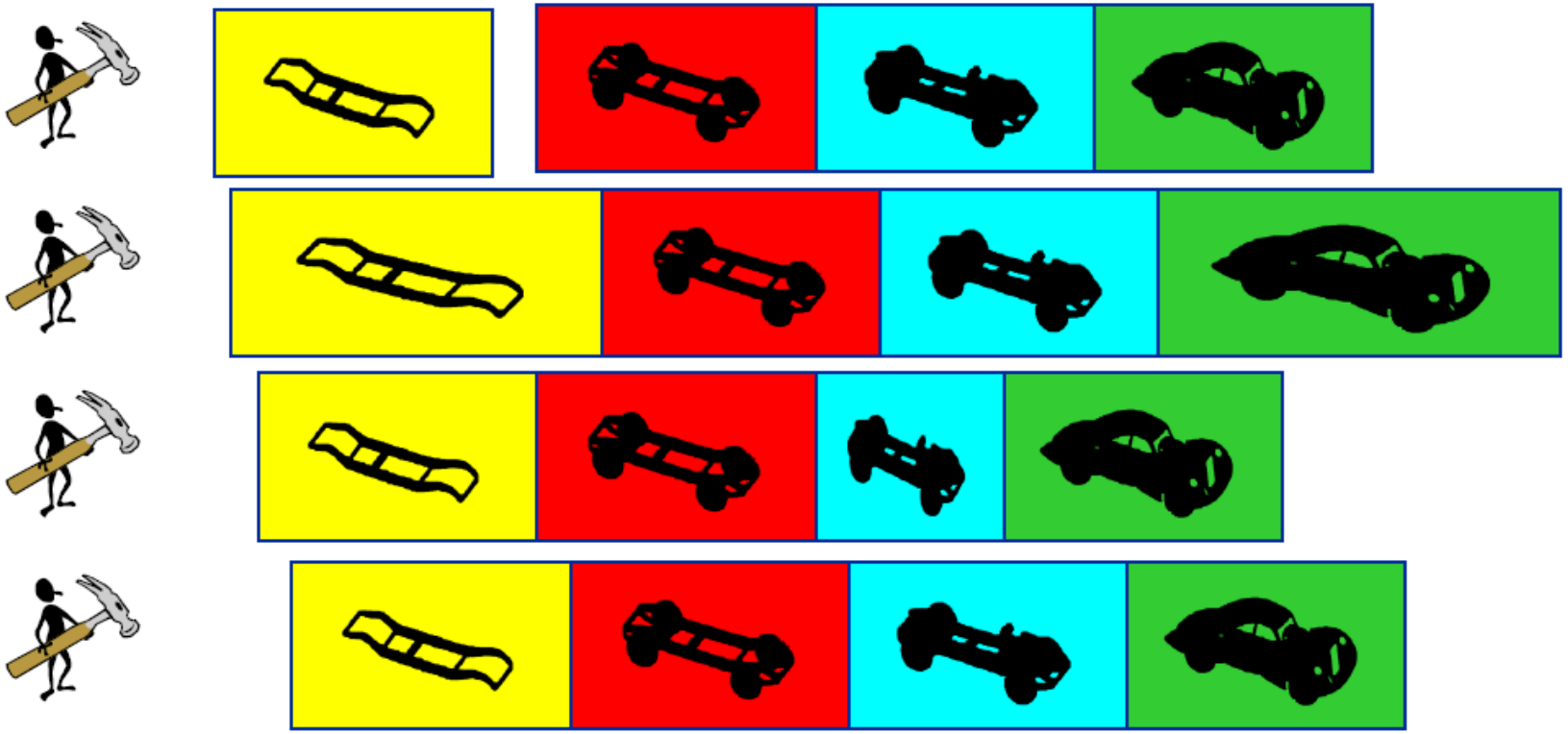
- **Parts of the manufacturing process can not be parallelized**
  - → Example: Delivery of components (all workers have to wait)



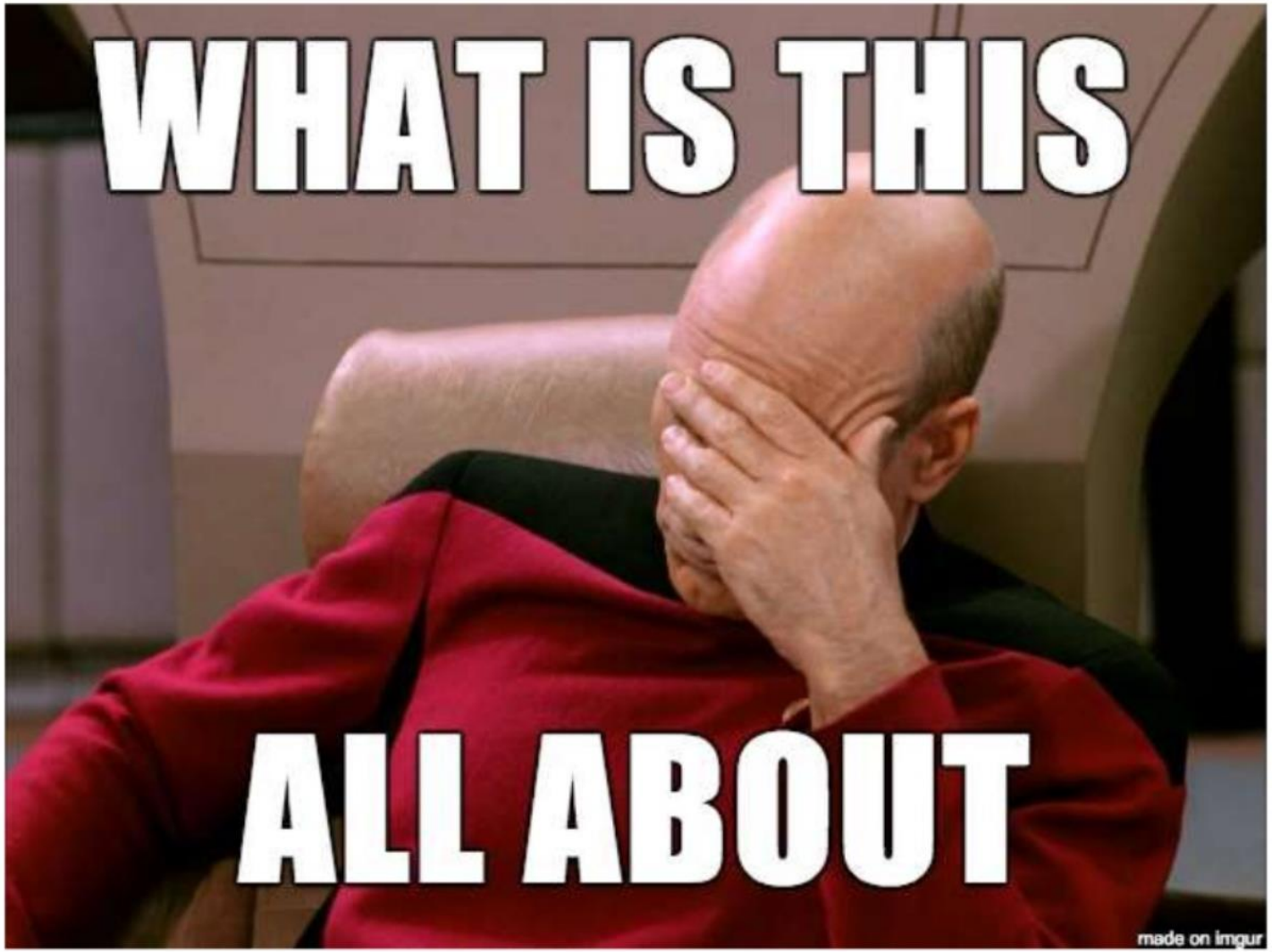
Limits of scalability (cont.)

- **Individual steps may take more or less time**

  → Load imbalances lead to unused resources



How are you doing?



Matrix Multiplication: from Serial to Multi-Core

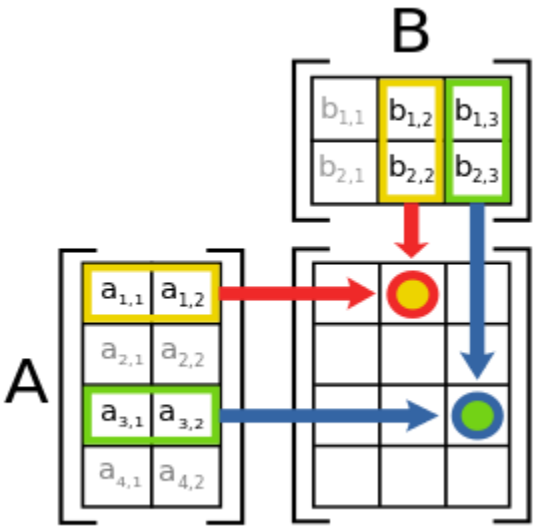
Illustration of Matrix Multiplication

- **Simple thing: C = A times B, with naive implementation in O(n^3)**

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix}}_{4\times 2 \text{ matrix}} \underbrace{\begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix}}_{2\times 3 \text{ matrix}} = \underbrace{\begin{bmatrix} \cdot & x_{12} & x_{13} \\ \cdot & \cdot & \cdot \\ \cdot & x_{32} & x_{33} \\ \cdot & \cdot & \cdot \end{bmatrix}}_{4\times 3 \text{ matrix}}$$

→ results in the following computations

$$x_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$x_{13} = a_{11}b_{13} + a_{12}b_{23}$$
$$x_{32} = a_{31}b_{12} + a_{32}b_{22}$$
$$x_{33} = a_{31}b_{13} + a_{32}b_{23}$$



source: Wikipedia

- **Independent computations exploitable for parallelization:**

→ rows of the matrix C

Illustration of Matrix Multiplication (cont.)

- **Class Matrix.java:**

```
final public class Matrix {
    private final int M;              // number of rows and columns
    private final double[][] data;    // M-by-M array

    // create M-by-N matrix of 0's
    public Matrix(int dim) {
        this.M = dim;
        data = new double[M][M];
    }

    [...]

} // end of class Matrix
```

Illustration of Matrix Multiplication (cont.)

- **Class Matrix.java, Matrix Multiplication implementation:**

```
// return C = A * B
   public Matrix times(Matrix B) {
       Matrix A = this;

       if (A.M != B.M) throw
          new RuntimeException("Illegal matrix dimensions.");

       Matrix C = new Matrix(A.M);

       for (int i = 0; i < M; i++)
           for (int j = 0; j < M; j++)
               for (int k = 0; k < M; k++)
                   C.data[i][j] +=
                   (A.data[i][k] * B.data[k][j]);

       return C;
    }
```

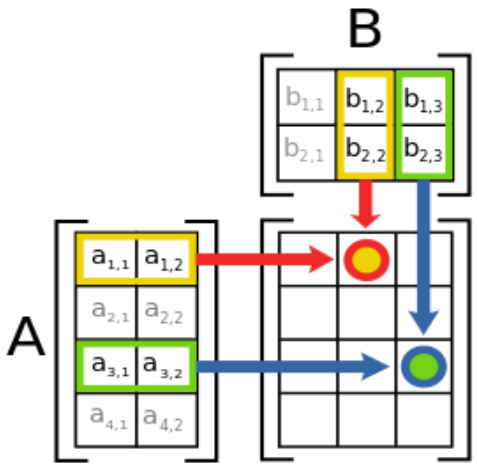
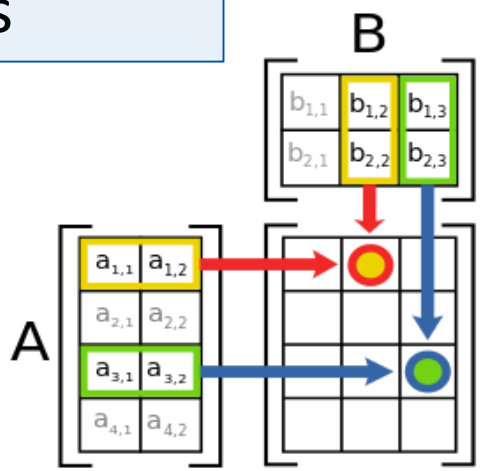


Illustration of Matrix Multiplication (cont.)

- **Class Matrix.java, Matrix Multiplication implementation:**

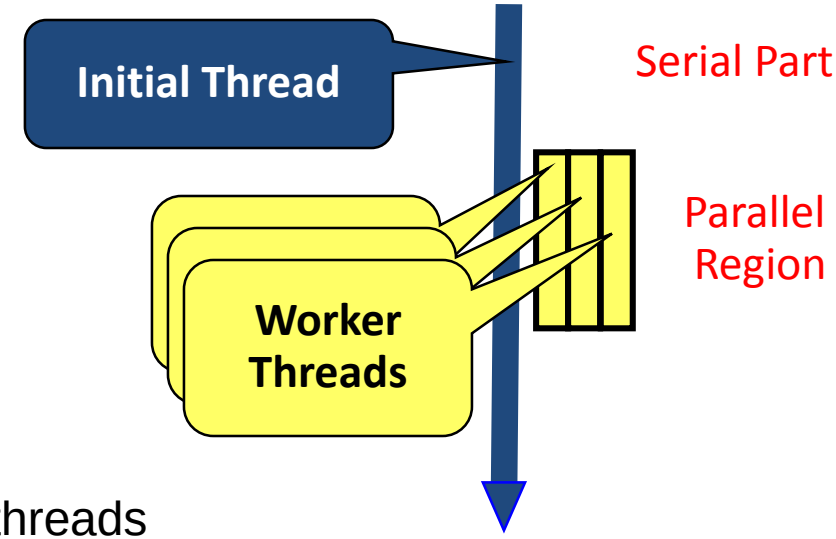
```
// return C = A * B
   public Matrix times(Matrix B) {
       Matrix A = this;

       if (A.M != B.M) throw
          new RuntimeException("Illegal matrix dimensions.");

       Mat

       for (int i = 0; i < M; i++)
           for (int j = 0; j < M; j++)
               for (int k = 0; k < M; k++)
                   C.data[i][j] +=
                   (A.data[i][k] * B.data[k][j]);

       return C;
   }
```

Independent for every i:
parallelize this loop over the threads



Thread-level Parallelization

1. **Determine the number of threads to be used**
   - → by querying the number of cores,
   - → or by user input
2. **Compute iteration chunks for every individual thread**
   - → Rows per Chunk = M / number-of-threads
3. **Create a team of threads and start the threads**
   - → Java class Thread encapsulates all thread mgmt. tasks
   - → Provide suitable function: matrix multiplication on given chunk
   - → Start thread via start() method
4. **Wait for all threads to complete their chunk**
   - → For each thread call join() method



Thread-level Parallelization (cont)

- **Class Matrix.java, threaded Matrix Multiplication implementation:**

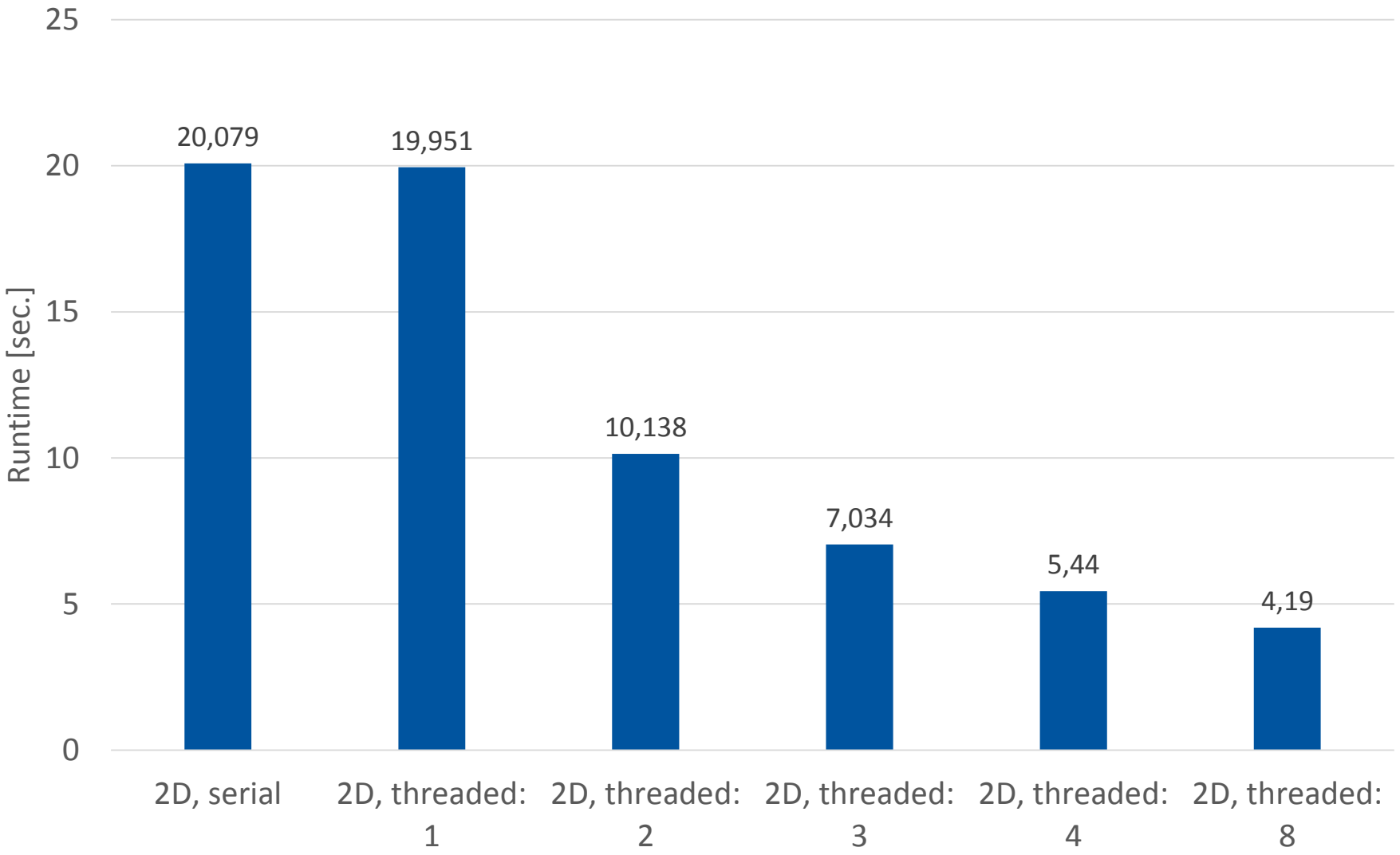
```
[...]
Thread threads[] = new Thread[num_threads];

    // start num_threads threads with their individual tasks
    for (int i = 0; i < num_threads; i++) {
       // compute chunk
       int rowsPerChunk = M / num_threads;
       int sRow = i * rowsPerChunk; int eRow = [...];
       // initialize task, create thread, start thread
       MultiplicationAsExecutor task = new MultiplicationAsExecutor
          (sRow, eRow, C.data, A.data, B.data, M);
       threads[i] = new Thread(task);
       threads[i].start();
    }

    // wait for all threads to finish
    for (int i = 0; i < num_threads; i++) {
        threads[i].join();
    }
[...]
```

Thread-level Parallelization (cont)

- **Class MultiplicationAsExecutor.java:**

```
public class MultiplicationAsExecutor implements Runnable {
    [...]
    // initialization by storing local chunk
    public MultiplicationAsExecutor(int sRow, int eRow, double[][] dC,
       double[][] dA, double[][] dB, int dim) {
        this.startRow = sRow;   this.endRow = eRow;
        this.c = dC;   this.a = dA;   this.b = dB;   this.dim = dim;
    }
    // perform the actual computation
    public void run() {
        for (int i = startRow; i < endRow; i++)
            for (int j = 0; j < dim; j++)
                for (int k = 0; k < dim; k++)
                    c[i][j] += (a[i][k] * b[k][j]);
    }
    // execute immediately
    public void execute(Runnable r) {
        r.run();
    }
}
```

Performance Evaluation



Amdahl‘s Law and Efficiency

Parallelization Overhead

- **Overhead introduced by the parallelization:**
  - → Time to start / end / manage threads
  - → Time to send / exchange data
  - → Time spent in synchronization of threads / processes
- **With parallelization:**
  - → The total CPU time increases,
  - → The Wall time decreases,
  - → The System time stays the same.
- **Efficient parallelization is about minimizing the overhead introduced by the parallelization itself!**

Speedup and Efficiency

- **Time using 1 CPU: T(1)**
- **Time using p CPUs: T(p)**

- **Speedup S: S(p)=T(1)/T(p)**

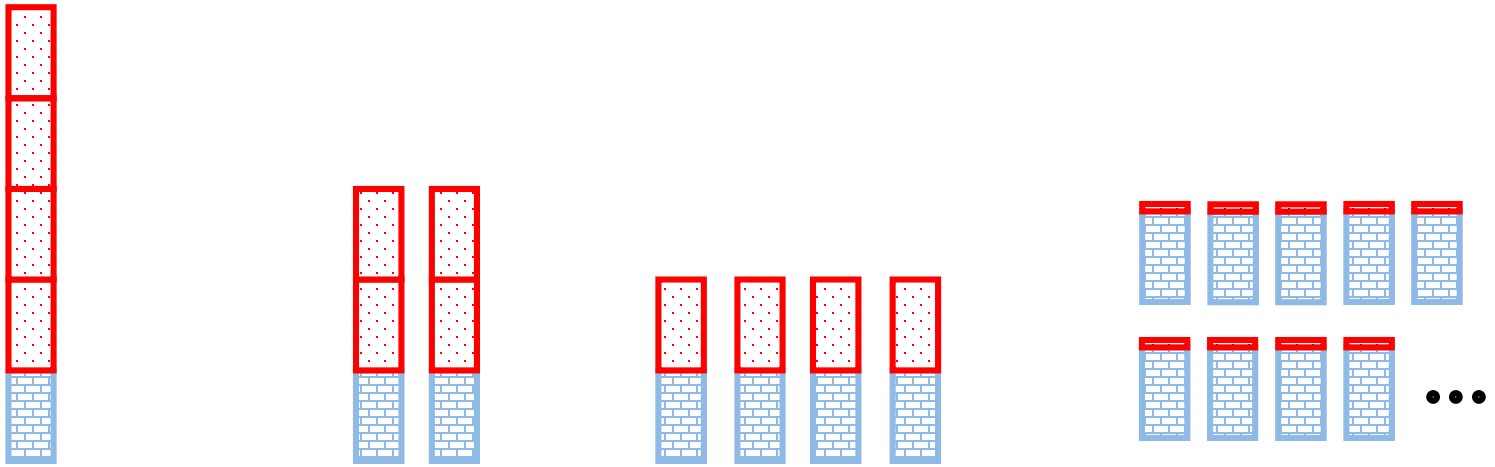
  → Measures how much faster the parallel computation is!

- **Efficiency E: E(p)=S(p)/p**

- **Ideal case: T(p)=T(1)/p → S(p)=p E(p)=1.0**

Amdahl‘s Law Illustrated

- **If 80% (measured in program runtime) of your work can be parallelized and „just“ 20% are still running sequential, then your speedup will be:**



1 processor:
time: 100%
speedup: 1

2 processors:
time: 60%
speedup: 1.7

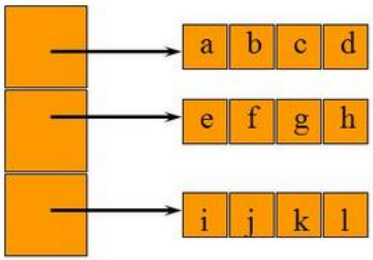
4 processors:
time: 40%
speedup: 2.5

∞ processors:
time: 20%
speedup: 5

Matrix Multiplication Reviewed

Performance Considerations

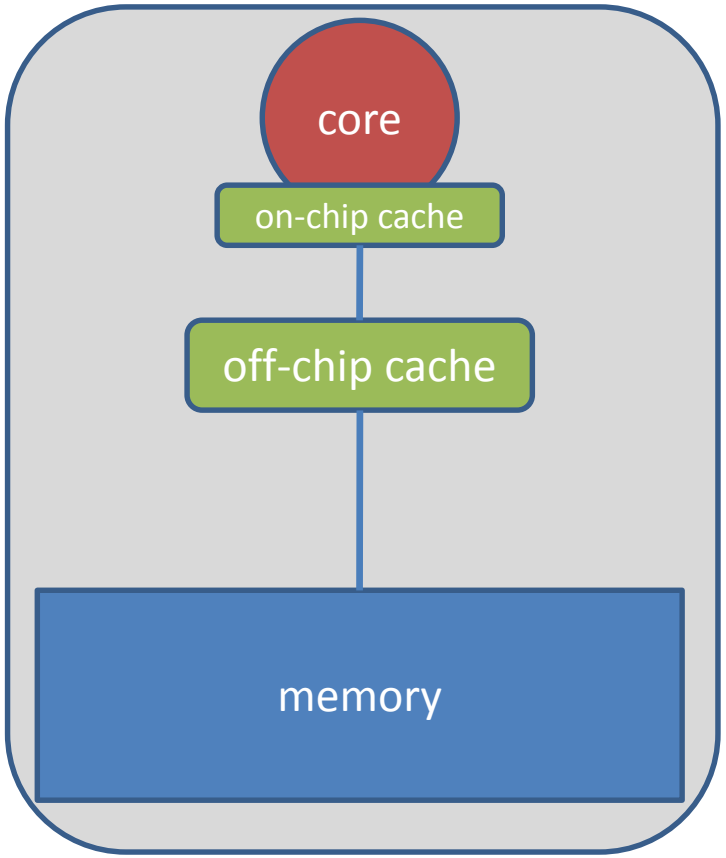
- **The Issue: 2D arrays in JAVA result in bad performance**



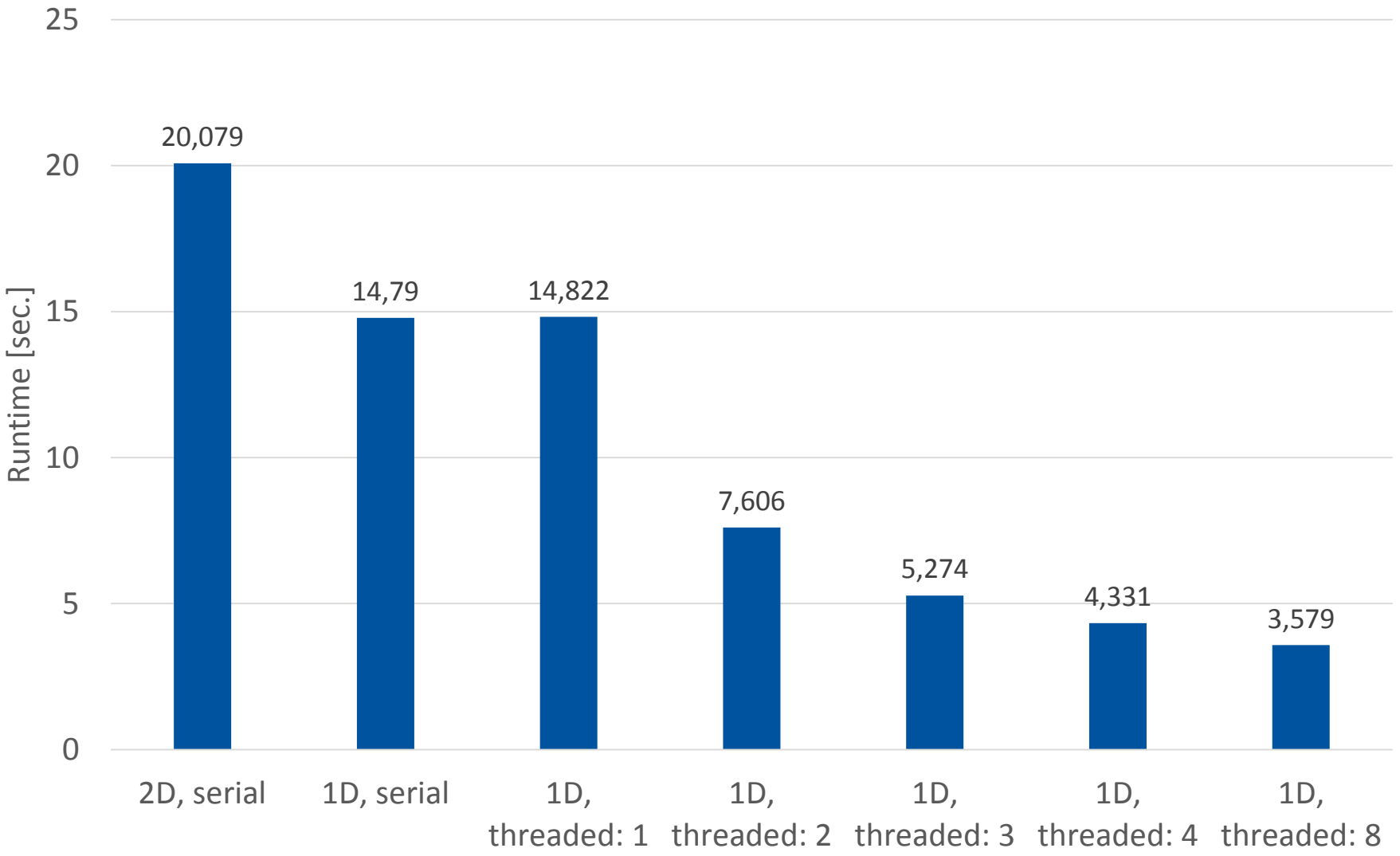
- **Better: 1D array with index fct.**

→ Caches only work well for consecutive memoy accesses!

- **CPU is fast**
- **Caches:**
  - → Fast, but expensive, thus small [MBs]
- **Memory is slow**
  - → Slow, but cheap, thus large [GBs]



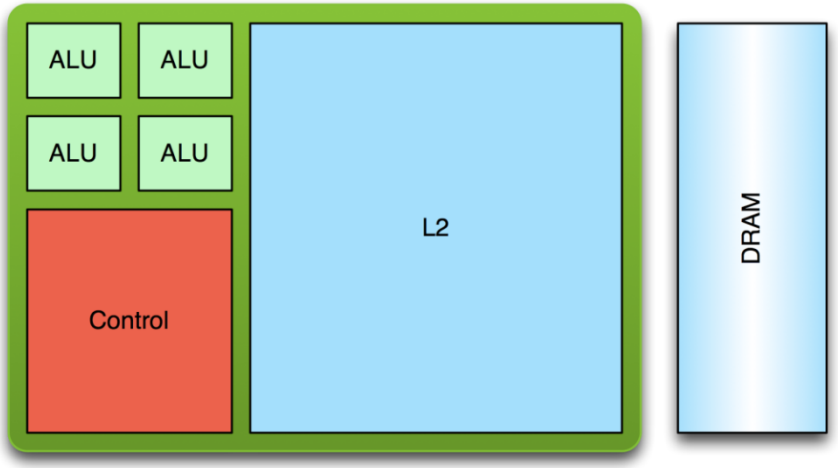
Performance Evaluation

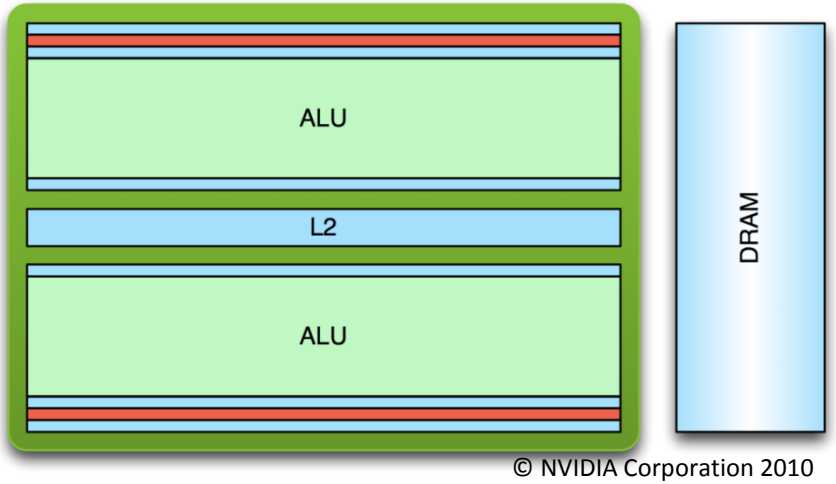


Basic GPGPU Concepts

Comparison CPU ⇔ GPU

- **Similar # of transistors but different design**





© NVIDIA Corporation 2010

CPU

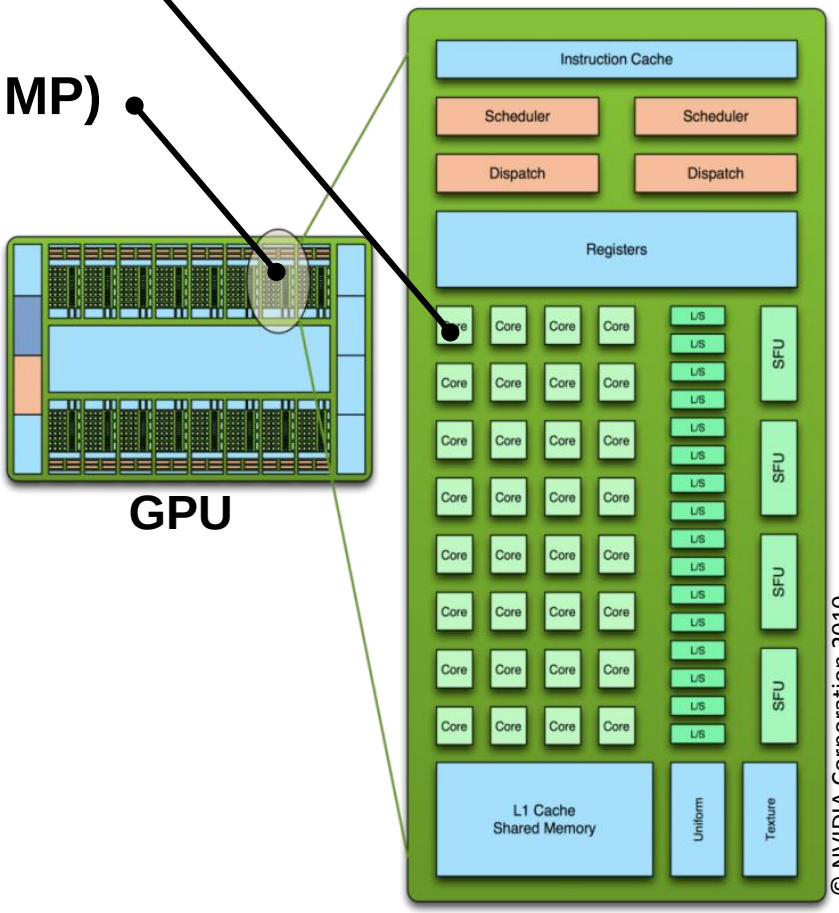
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

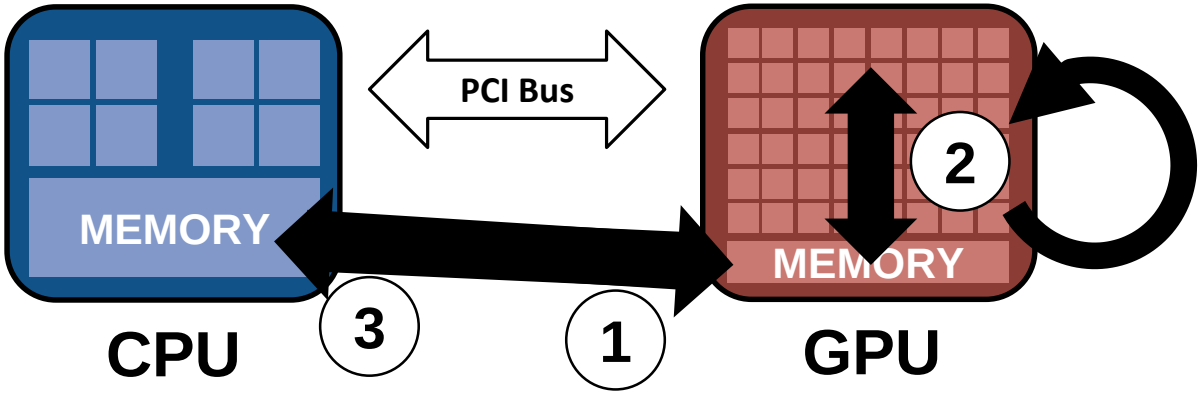
GPGPU architecture: NVIDIA‘s Fermi

- **3 billion transistors**
- **448 *Cores/ Streaming Processors* (SP)**
  - → E.g. floating point and integer unit
- **14 *Streaming Multiprocessors* (SM, MP)**
  - → 32 cores per MP
- **Memory hierarchy**
- **Processing flow**
  - → Copy data from host to device
  - → Execute kernel
  - → Copy data from device to host



© NVIDIA Corporation 2010

Comparison CPU ⇔ GPU

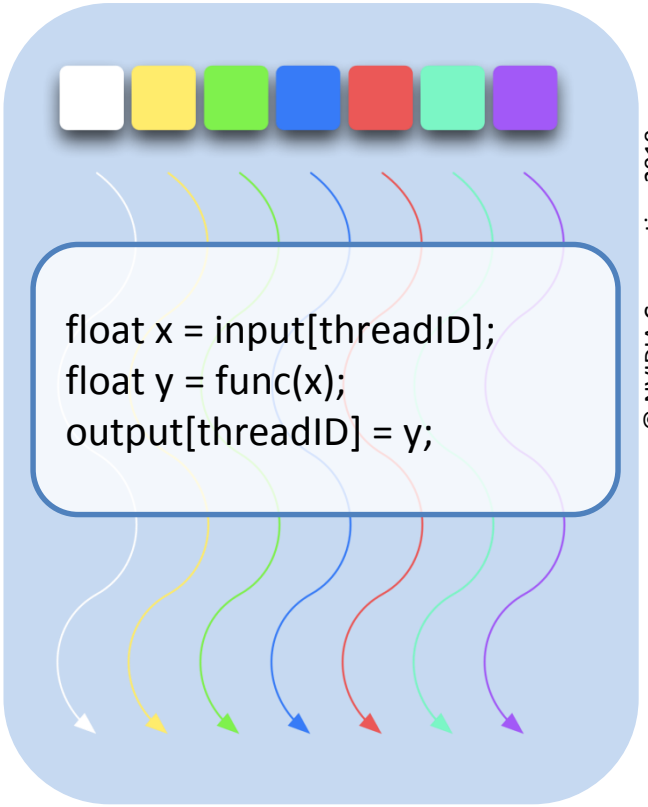


- **Weak memory model**
  - → Host + device memory = separate entities
  - → No coherence between host + device
    - →Data transfers needed
- **Host-directed execution model**
  - → Copy input data from CPU mem. to device mem.
  - → Execute the device program
  - → Copy results from device mem. to CPU mem.

Programming model

- **Definitions**
  - → Host: CPU, executes functions
  - → Device: usually GPU, executes kernels

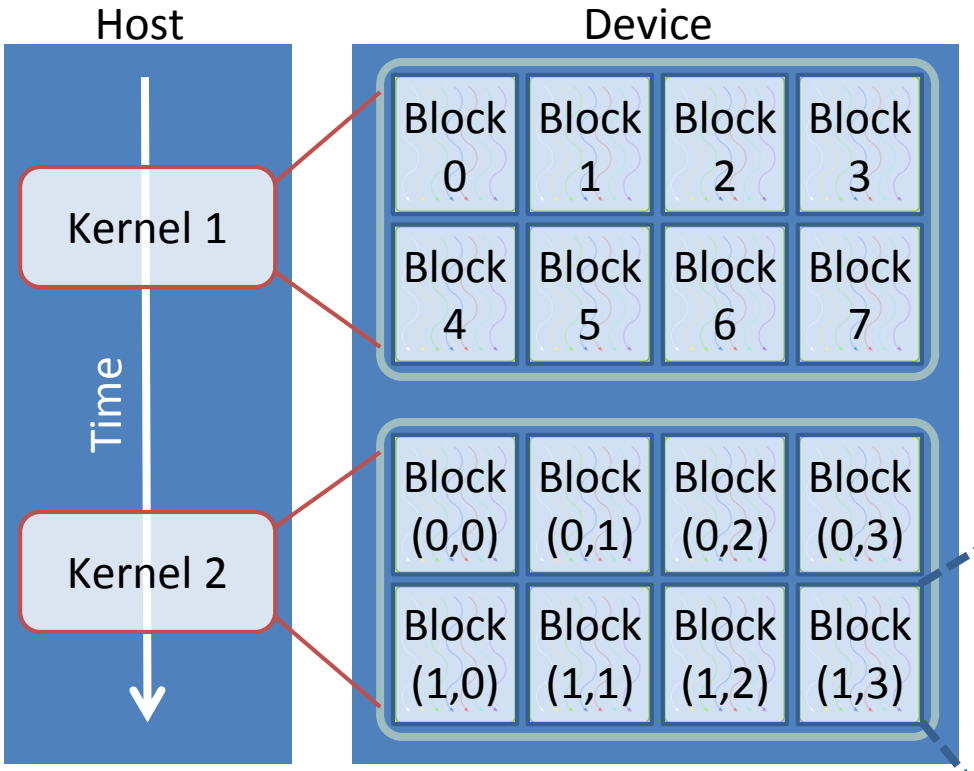
- **Parallel portion of application executed on device as kernel**
  - → Kernel is executed as array of threads
  - → All threads execute the same code
  - → Threads are identified by IDs
    - → Select input/output data
    - → Control decisions



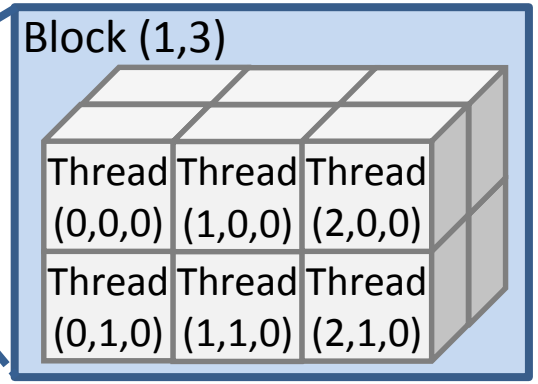
© NVIDIA Corporation 2010

Programming model (cont.)

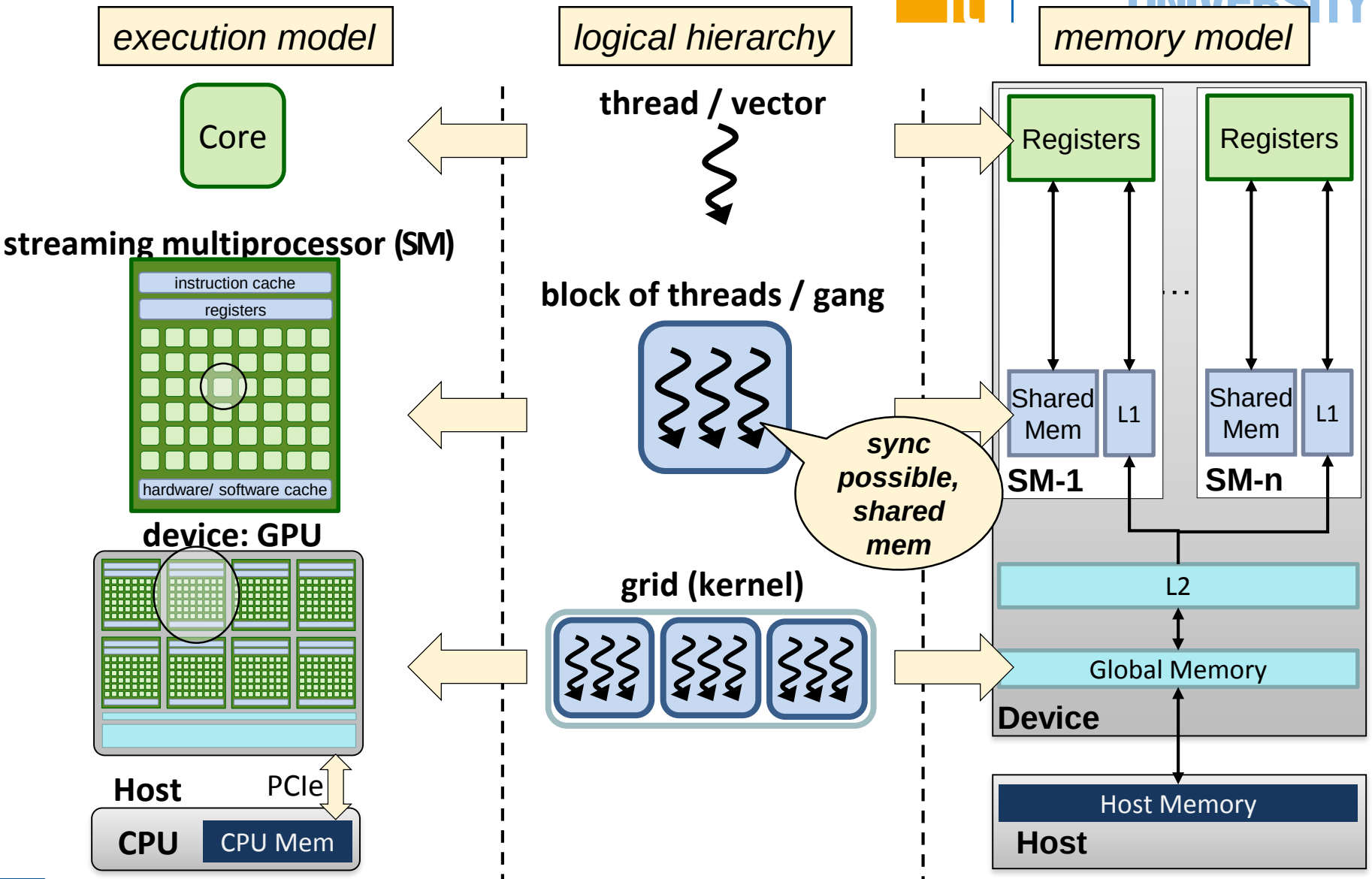
- **Threads are grouped into *blocks*, Blocks are grouped into a *grid*.**
- **Kernel is executed as a grid of blocks of threads**



- **Dimensions of blocks and grids: ≤ 3**



Putting it all together



Vector, worker, gang mapping is compiler dependent.

Matrix Multiplication: from Multi- to Many-Core

GPGPU Parallelization

1. **Create a CUDA-kernel well-suited for the GPGPU device**
   - → simple-enough and data-parallel code
2. **Setup JCuda and Compile your program accordingly**
   - → kernel code has to be compiled to .ptx file with NVIDIA‘s compiler
3. **Initialize JCuda environment**
   - → load driver library, initialize device
4. **Transfer data from host to device**
   - → all data necessary on the device
5. **Execute CUDA-kernel**
   - → launch kernel on device
6. **Transfer results from device to host**
   - → all data necessary after the kernel execution on the host

GPGPU Parallelization (cont.)

- **CUDA-kernel:**

  → C is often close enough to JAVA ☺

```
extern "C"
__global__ void matmult(int dim, double *c, double *a, double *b)
{
    int row = blockDim.y * blockIdx.y + threadIdx.y;
    int col = blockDim.x * blockIdx.x + threadIdx.x;

    if (row > dim || col > dim) return;

    double prod = 0;
    int kk;
    for (kk = 0; kk < dim; ++kk){
        prod += a[row * dim + kk] * b[kk * dim + col];
    }
    c[row*dim + col] = prod;
}
```

GPGPU Parallelization (cont.)

- **Class Matrix.java, CUDA Matrix Multiplication implementation:**

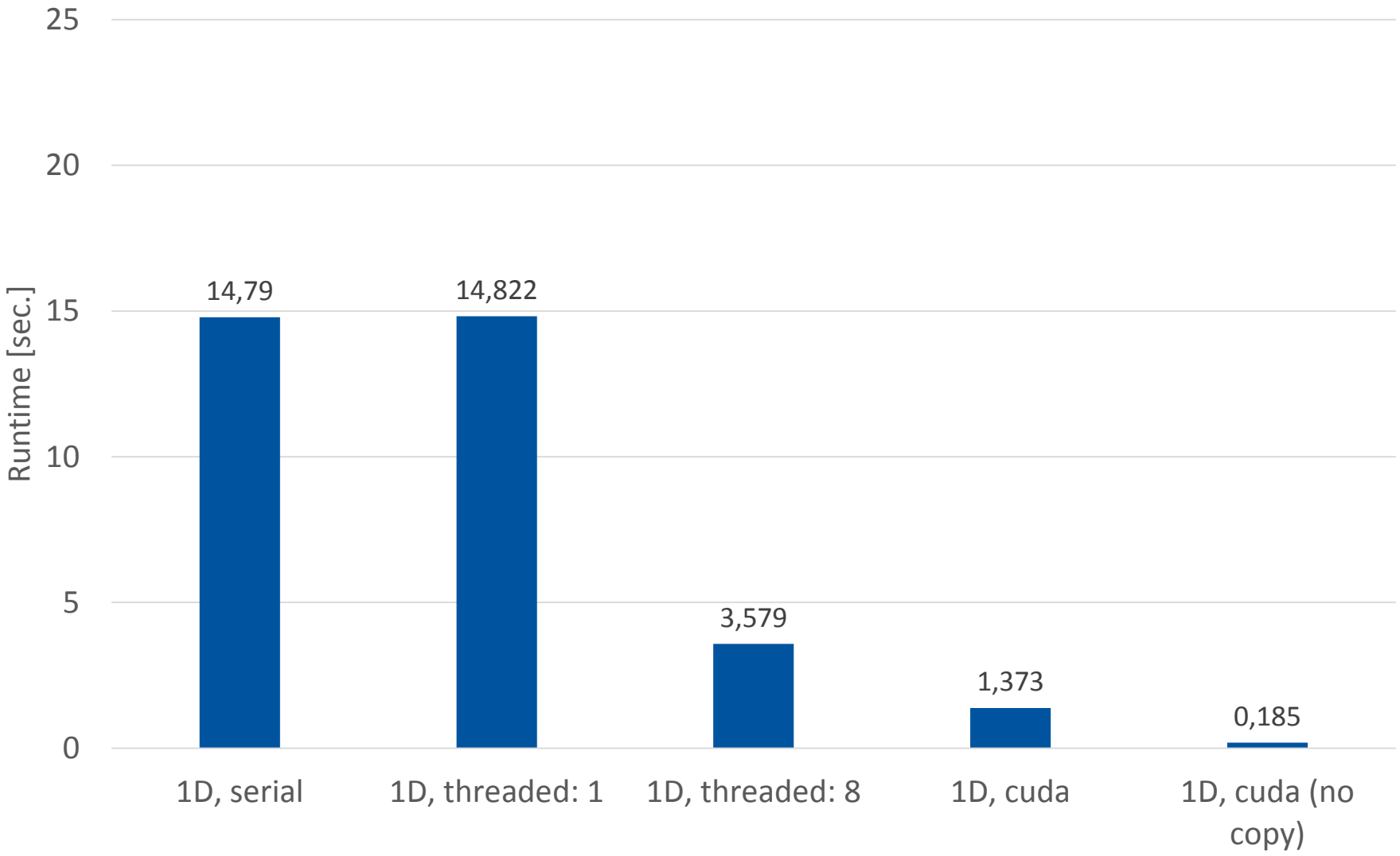
```
[...]

// allocate memory
int size = A.M * A.M * Sizeof.DOUBLE;
CUdeviceptr a_dev = new CUdeviceptr();
CUdeviceptr b_dev = new CUdeviceptr();
CUdeviceptr c_dev = new CUdeviceptr();
cudaMalloc(a_dev, size); cudaMalloc(b_dev, size);
cudaMalloc(c_dev, size);

// load code
CUmodule module = new CUmodule();
cuModuleLoad(module, "JCudaMatmulKernel.ptx");
CUfunction function = new CUfunction();
cuModuleGetFunction(function, module, "matmult");
```

GPGPU Parallelization (cont.)

- **Class Matrix.java, CUDA Matrix Multiplication implementation:**

```
// copy data
cuMemcpyHtoD(a_dev, Pointer.to(A.data), size);
cuMemcpyHtoD(b_dev, Pointer.to(B.data), size);

// launch kernel
Pointer parameters = Pointer.to(
    Pointer.to(new int[] { A.M }), Pointer.to(c_dev),
    Pointer.to(a_dev), Pointer.to(b_dev) );
final int threadsPerDim = 32;
int grids = (int) Math.ceil(((double) A.M) / threadsPerDim);
cuLaunchKernel(function, grids, grids, 1,
    threadsPerDim, threadsPerDim, 1, 0, null,
    parameters, null);
cuCtxSynchronize();

[...] // cleanup code omissed for brevity
```

Performance Evaluation



Performance: Critical Discussion

- **FLOPS: performance rate (no. floating-point operations per second)**
- **Matrix Multiplication Algorithm: n^2 + 2n complexity, here n = 1536**
  - → Result: 7247757.3 mega double precision floating-point operations performed
- **Host: Intel Xeon E5620**
  - → 4 Cores, with Hyper-Threading
  - → 2.4 GHz clock frequency
  - → SSE4.2: 4 floating-ypoint operations per cycle peak
  - → 4 * 2.4 * 4 = 38.4 GFLOPs peak performance
  - → Our multi-threaded code run at 2025 MFLOPS
    - →5.3 % efficiency

Performance: Critical Discussion (cont.)

- **GPU: NVIDIA Tesla C2050**
  - → 448 CUDA cores
  - → 1.15 GHz clock frequency
  - → 448 * 1.15 = 515 GFLOPS peak performance
  - → Our CUDA kernel runs at 5278 MFLOPS
    - →10.1 % efficiency
- **Note on the GPU: the data transfer is the most costly part**
  - → Kernel execution time incl. data transfer: 1.373 sec.
  - → Kernel execution time excl. data transfer: 0.185 sec.
- **The GPU would profit from**
  - → a larger problem size, or
  - → repeated executions of the kernel

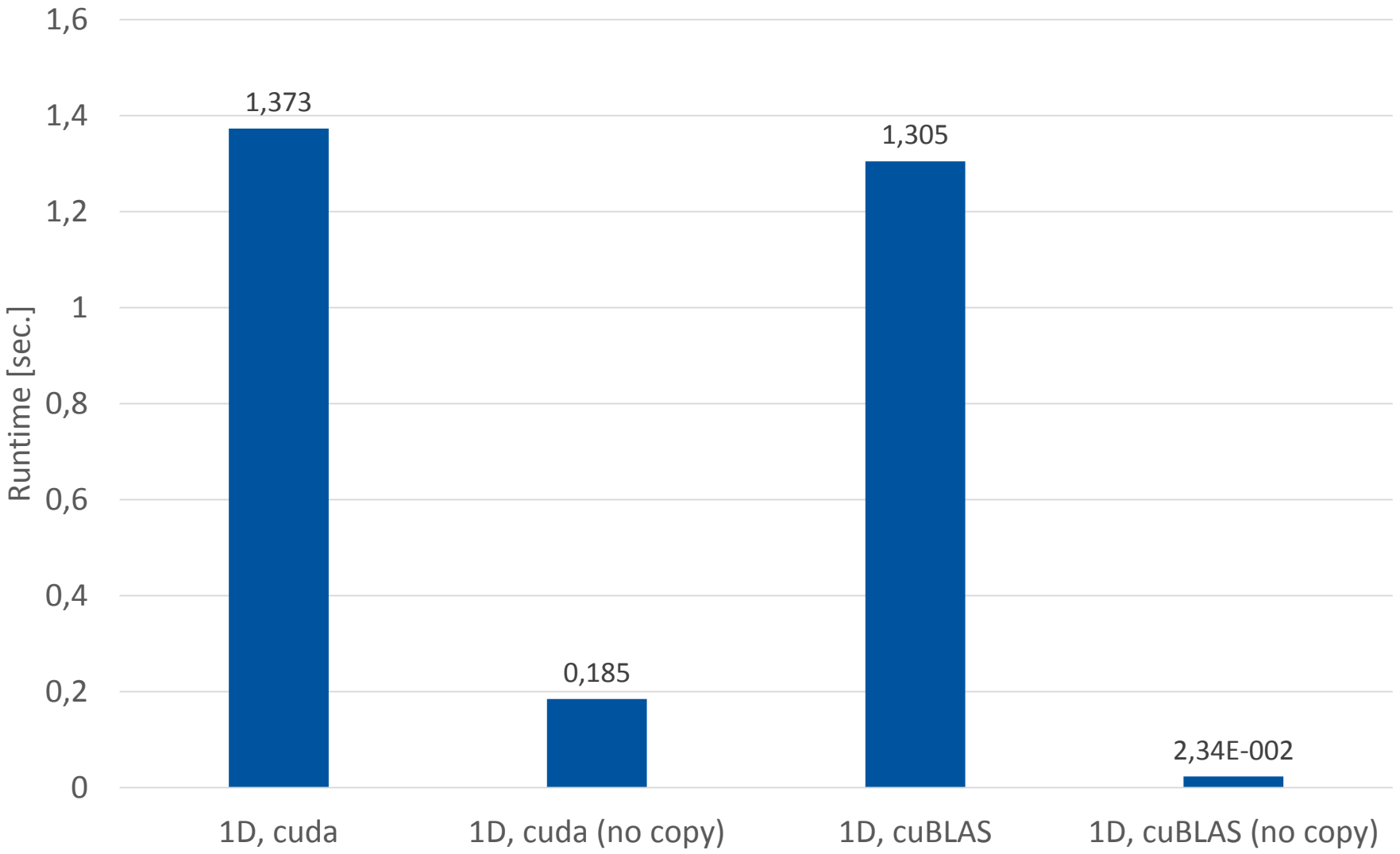
Performance: Critical Discussion (cont.)

- **Can we do better with the same algorithm? Yes!**
  - → Matrix Multiplication can profit from blocking, that is the reuse of data in the caches, for both the CPU and the GPU. And: Matrix Multiplication is a standard problem, there are libraries for that: BLAS (dgemm).

- **GPGPU Performance with cuBLAS**
  - → Kernel execution time incl. data transfer: 1.306 sec.
  - → Kernel execution time excl. data transfer: 0.0234 sec.

  => The CUDA kernel itself runs at 310 GFLOPS

Performance Evaluation



Summary

Summary

- **What did we learn?**
  - → Parallelization has become a necessity to exploit the performance potential of modern multi- and many-core architectures!
  - → Efficient programming is about optimizing memory access, efficient parallelization is about minimizing overhead.
  - → Shared Memory parallelization: work is distributed over threads on separate cores, threads share global data
  - → Heterogeneous architectures (here: GPGPUs): separate memories require explicit data management, well-suited problems can benefit from special architectures with large amount of parallelism.
- **Not covered today: Cache Blocking to achieve even better perf.**
- **Not covered today: Distributed Memory parallelization**

Lectures

- **SS 2016**
  - → Lecture: *Performance & correctness analysis of parallel programs*
  - → Software Lab: *Parallel Programming Models for Applications in the Area of High-Performance Computation (HPC)*
  - → Seminar: *Current Topics in High-Performance Computing (HPC)*

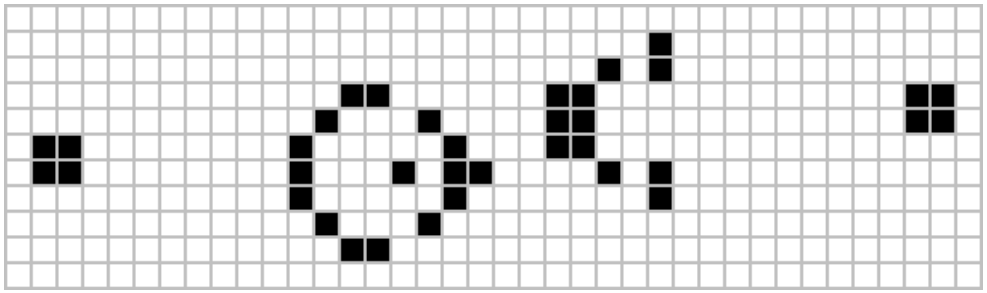
- **WS 2016/17**
  - → Lecture: *Introduction to High-Performance Computing*
  - → Seminar: *Current Topics in High-Performance Computing (HPC)*

www.hpc.rwth-aachen.de
contact@hpc.rwth-aachen.de

Christmas Exercise

Game of Life

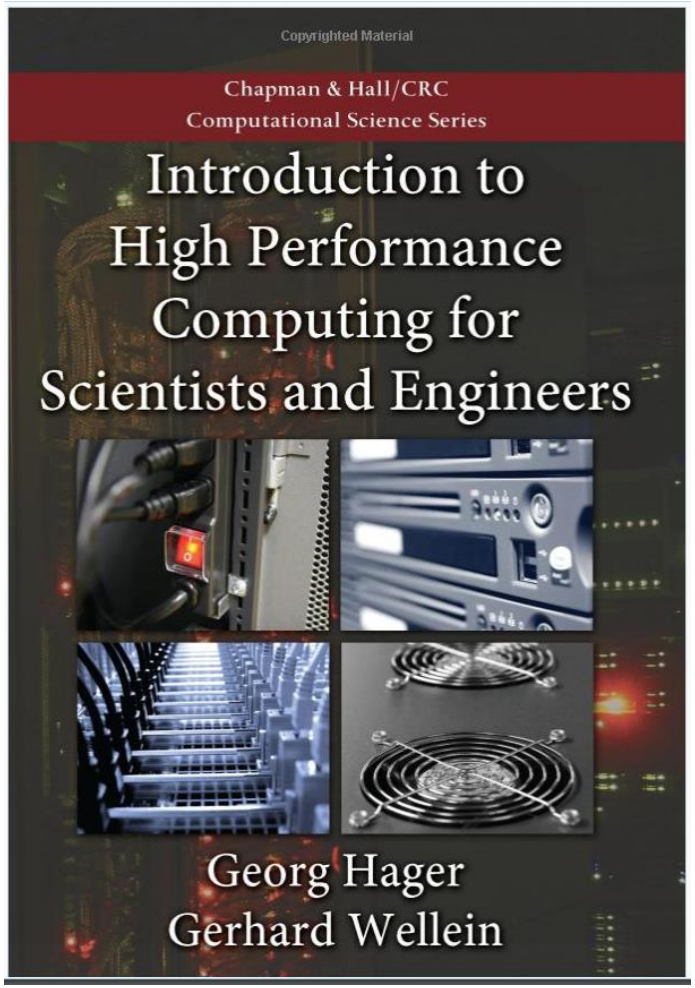
- **Game of Life: zero-player game ☺**
  - → Evolution is determined by initial state and game rules



- **Rules:**
  - → 2D orthogonal grid of cells
  - → every cell has only two possible states: alive (black) or dead (white)
  - → every cell interacts with its neighbours, and at each time step:
    - → any live cell with fewer than two live neighbours dies,
    - → any live cell with two or three live neighbours lives on,
    - → any live cell with more than three live neighbours dies,
    - → any dead cell with exactly three live neighbours becomes a live cell.

There is something to win!

- **A really nice book on HPC from really nice people:**



  - **One book for each in the group with**
    - → the highest performance in a multi-threaded solution
    - → the highest performance in a CUDA-parallel solution
    - → random drawing winner
  - **Requirement: fill out and hand in the questionnaire**
  - **Measurements will be done by us on linuxc8 (RWTH Compute Cluster)**

Umfrage zur Produktivität beim Entwickeln im Bereich HPC

- **Allgemeine Umfrage (einmalig gültig bis 20.1.2016)**
  - → Hilft unseren Forschungsaktivitäten!

Bitte nehmen Sie teil!

- **Am Ende der Weihnachtsaufgabe ausfüllen & auf Weihnachtsaufgabe beziehen!**
  - → Einflussfaktoren auf Programmieraufwand?
  - → Benötigter Programmieraufwand?
  - → Anzahl der programmierten Code-Zeilen?
  - → Erreichte Performance?

https://app.lamapoll.de/ProductivityHPCStudents/

- **Zum leichten und qualitativen Ausfüllen der Umfrage bitte während des Entwickelns obige Daten schon festhalten**

Questions?