

---

# Massively Parallel Computing with Cuda

**Hendrik Lensch**  
**Robert Strzodka**

# Today

---

- **Administration**
- **Motivation**
  - Why parallelism?
  - Parallel Architectures
- **First Programming Example**
  - Cuda “Hello World”

---

# Administrative Issues

# A Truly Parallel Course

---

- **Held in parallel in Ulm and Saarbruecken**
- **Ulm: Wahlpflichtvorlesung in Informatik + Medieninfo.**
- **Lectures in English**
- **Time and Location**
  - Mo, 10-12, MPI - room 024, Ulm – kiz videoconference room
- **ECTS:**
  - 6 credit points
- **Web-Page**
  - <http://www.uni-ulm.de/in/mi/lehre/2008ws/massively-parallel-computing-with-cuda.html>
  - <http://www.mpi-inf.mpg.de/departments/d4/teaching/ws2008-2009/ParCo08>
  - Schedule, Slides as PDF
  - Literature, Assignments, other Information
- **Sign up for course and e-mail list on web page !**

# People

---

- **Lecturer**

- Robert Strzodka

- Max-Planck-Institut für Informatik, Saarbruecken, Campus E1.4, Room 227, phone: 0681/9325-427

- E-mail: [strzodka at mpi-inf.mpg.de](mailto:strzodka@mpi-inf.mpg.de)

- Hendrik Lensch

- currently MPI, Room 228, phone: 0681/9325-428

- in Ulm: Room O27/338

- E-mail: [lensch at mpi-inf.mpg.de](mailto:lensch@mpi-inf.mpg.de)

- **Teaching Assistants**

- Hendrik Becker – Saarbrücken

- Room 226, phone: 0681/9325-426

- E-mail: [hbecker at mpi-inf.mpg.de](mailto:hbecker@mpi-inf.mpg.de)

- Holger Dammertz – Ulm

- Room O27/3302, phone: 3124

- E-mail: [holger dot dammertz at uni-ulm.de](mailto:holger_dot_dammertz@uni-ulm.de)

# Exercises

---

- **Wed, 10-12, MPI - Room 019, Ulm – Kiz videoconference room**
  - discuss assignment sheets
  - practical issues
- **Weekly assignment sheets**
  - practical exercises only
  - six/seven in total
  - prerequisite to be admitted to student project!
- **You need to register:**
  - online for the e-mail list ← general questions
  - for the grading before the student project starts

# Student Project

---

- **from beginning of December until the end of term**
  - write project proposal
  - we will give topic suggestions from current research
  - present proposal – mid semester
  - implement project
  - present solution – last lecture
  - write a project summary – Feb/Mar 09
  
- quality of project solution (difficulty + realization) will determine your final grade

# Course Syllabus (1)

---

- **Introduction**
  - Block-thread model
  - Cuda 101
- **Memory**
  - Latency, bandwidth, hierarchy
  - MatVec, MatMat multiplication
  - Reduction
- **Branching**
  - Number of operations vs. number of processors
  - Reduction2
  - Prefix sums
- **Sorting**
  - Bitonic merge, radix sort, hybrids
  - Synchronization, shared and global memory
  - Atomic ops

# Course Syllabus (2)

---

- **Data Structures**
  - Tensor grids, trees, hashes, point clouds
  - Hierarchical, adaptive, unstructured
- **Searching**
  - lists, grids, trees, hashes, kNN
- **Student Proposals**
- **Numerics on GPUs(1)**
  - parallel linear algebra
    - dense LA: direct solvers, Gauss elimination, Cholesky
- **Numerics on GPUs(2)**
  - sparse LA: iterative solvers, CG, MG
  - precision, accuracy
  - adaptive grids, error estimators
- **N-body problems**
  - gravitation, electrostatic, molecular

# Course Syllabus (3)

---

- **Complexity and profiling**
  - compute/memory bounds
  - execution speed of instructions
  - speed of memory access patterns, streams, prefetching, on-demand
  - estimated theoretical limits
  - profiling techniques
- **System integration**
  - MultiGPU computers
  - GPU clusters
  - coupling of different parallelism types
  - Legacy SW integration
- **Parallel Ray Tracing**
- **Student Project Presentation**

# Overview

---

- **You will get a first glimpse of Cuda.**
- **You will hopefully be enabled to write your own Cuda-Software.**
- **You will observe how much faster 100-200 processors can do the job.**
- **You will be disappointed by how slow your current programs are.**
- **We cannot teach all possible optimizations. There are typically 20 different ways.**

# Resources

---

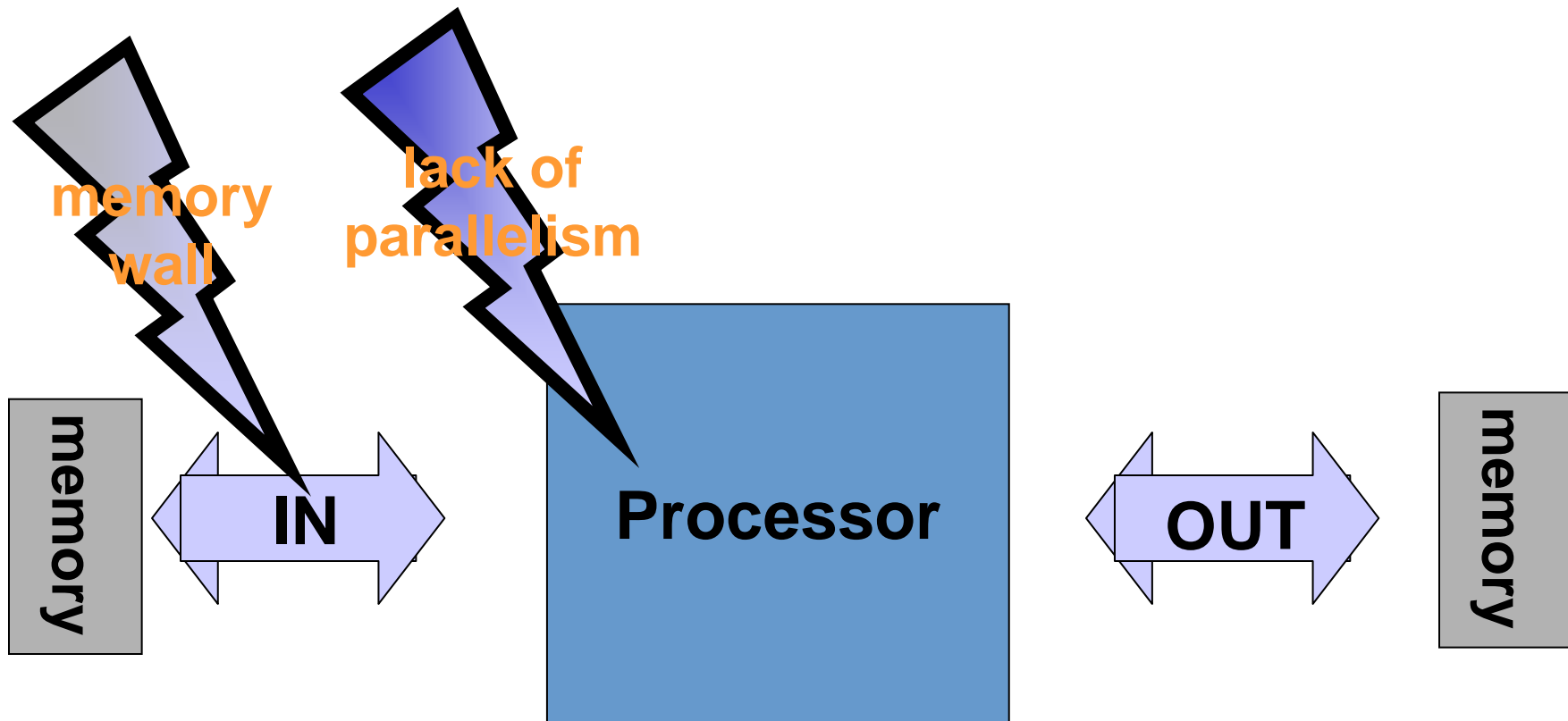
- **Where to find Cuda and the documentation?**
  - [http://www.nvidia.com/cuda\\_home.html](http://www.nvidia.com/cuda_home.html)
- **Take a look at the**
  - Cuda 2.0 programming guide
  - Cuda 2.0 SDK
- **Lecture on parallel programming on the GPU by David Kirk and Wen-mei W. Hwu (most of the following slides are copied from this course)**
  - <http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html>
- **Books:**
  - **Patterns for Parallel Programming** by [Timothy G. Mattson](#); [Beverly A. Sanders](#); [Bernie L. Massingill](#) (online!)
  - **Practical PRAM Programming** by [Jörg Keller](#), [Christoph W. Kessler](#), [Jesper L. Träff](#)

---

# Data Processing

# Data Processing in General

---

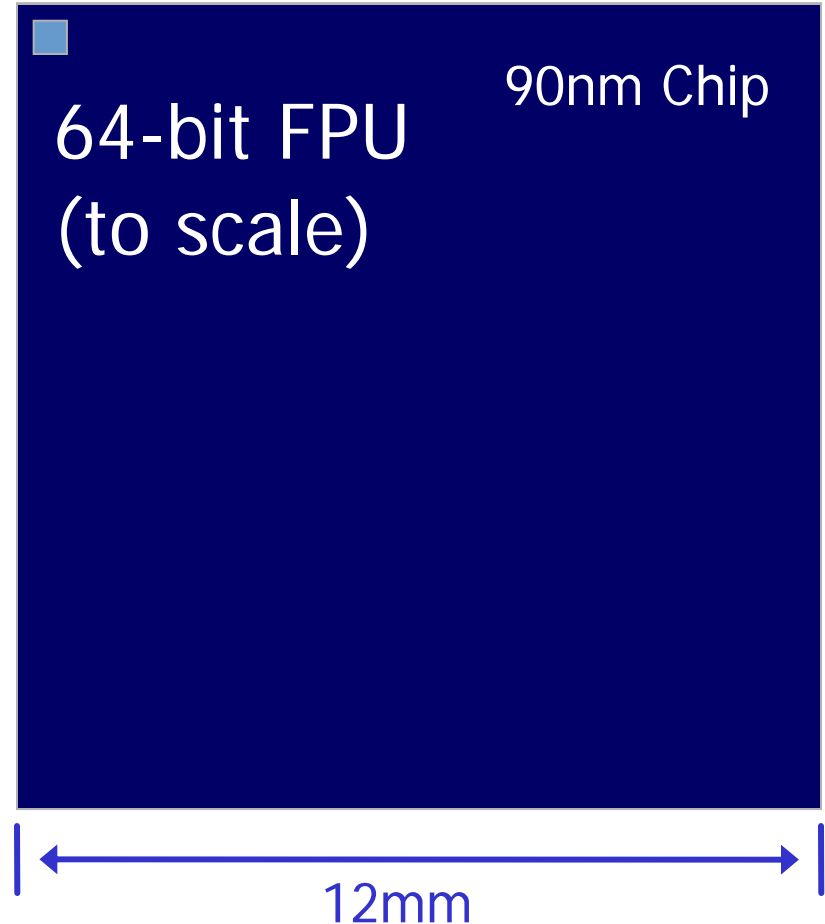


# Computation is Cheap, Bandwidth is Expensive

---

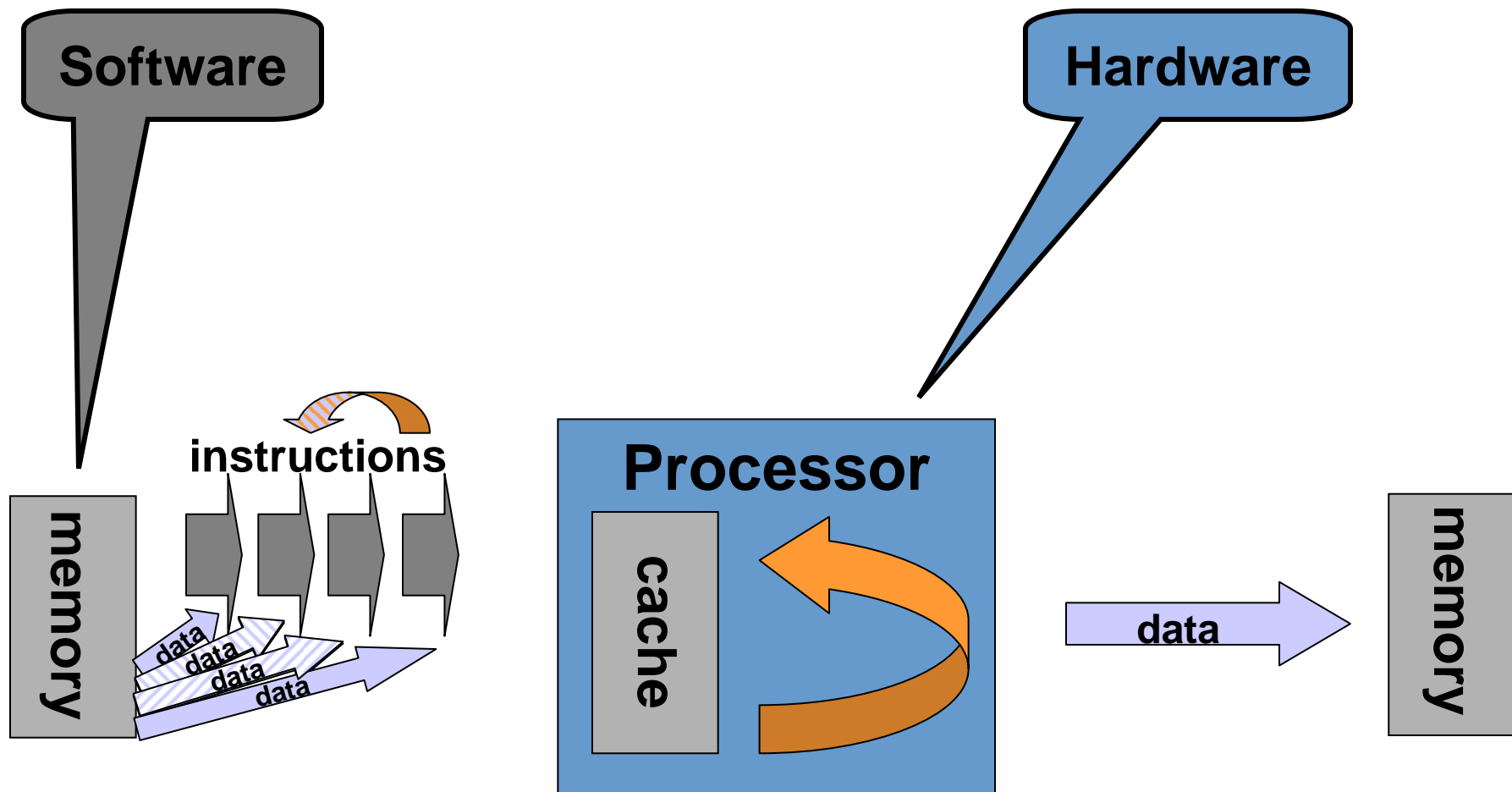
- Computation
  - Enough space for **100s of ALUs** per chip
  - **TFLOP** on single chip possible
- Bandwidth
  - **Number of pins** per chip huge cost factor
  - **Exponential gap** between memory DRAM speed and computation

→ | | ← 0.5mm

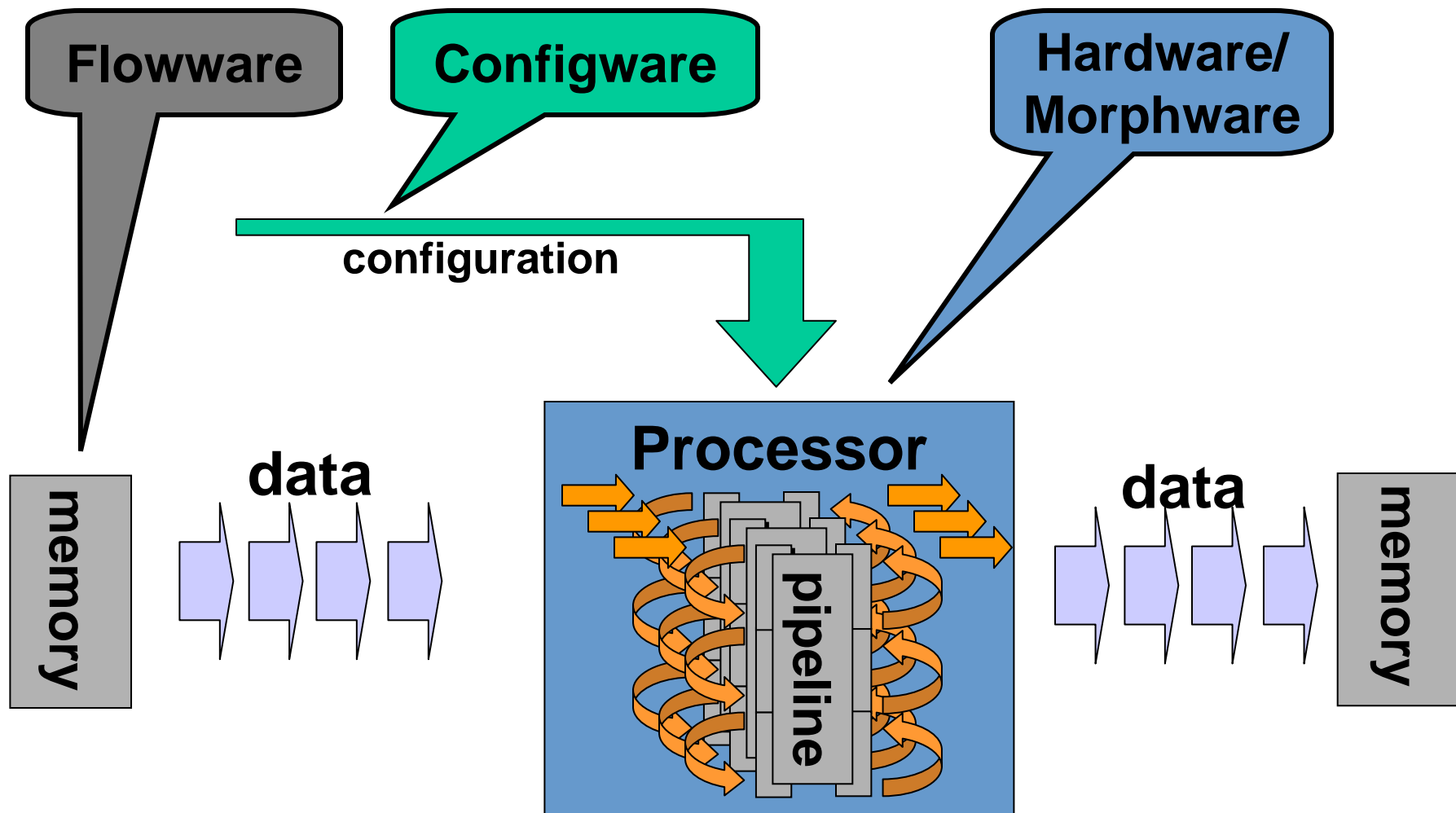


# Instruction-Stream-Based Processor

---



# Data-Stream-Based Processing

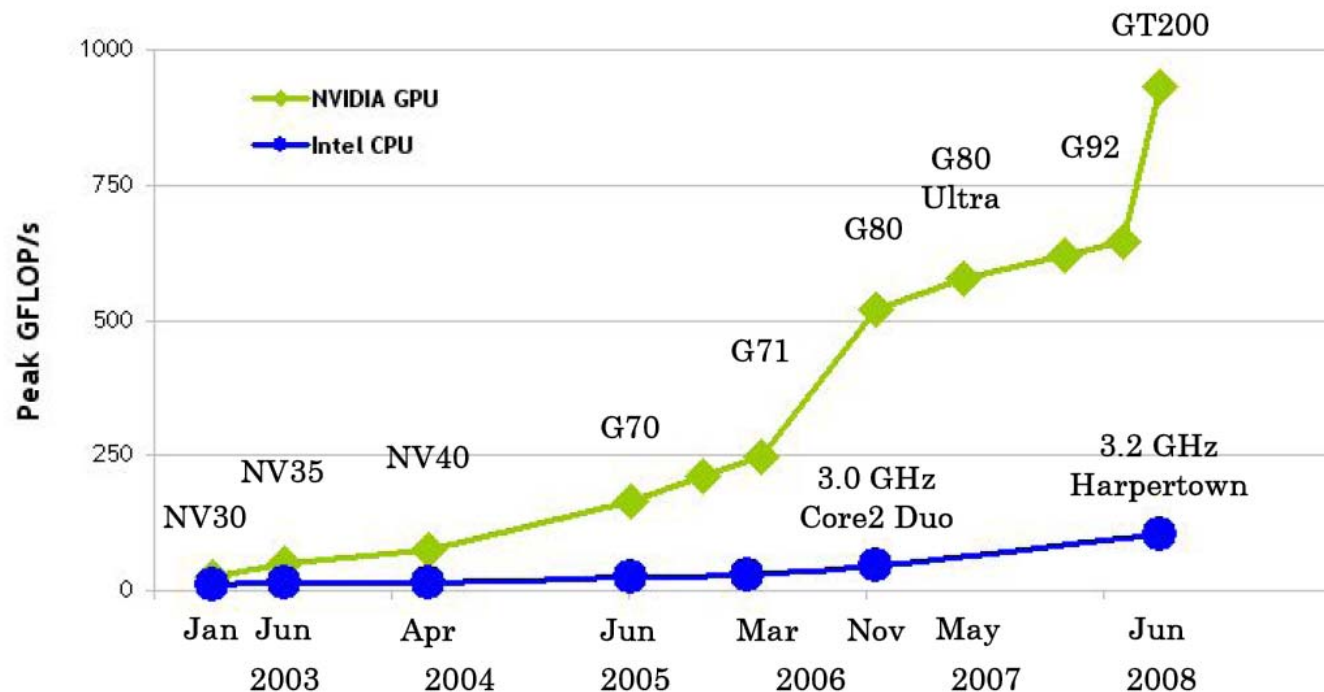


Nomenclature from

[Hartenstein. Data-stream-based computing: Models and architectural resources, MIDEM 2003]

# Why Massively Parallel Processors

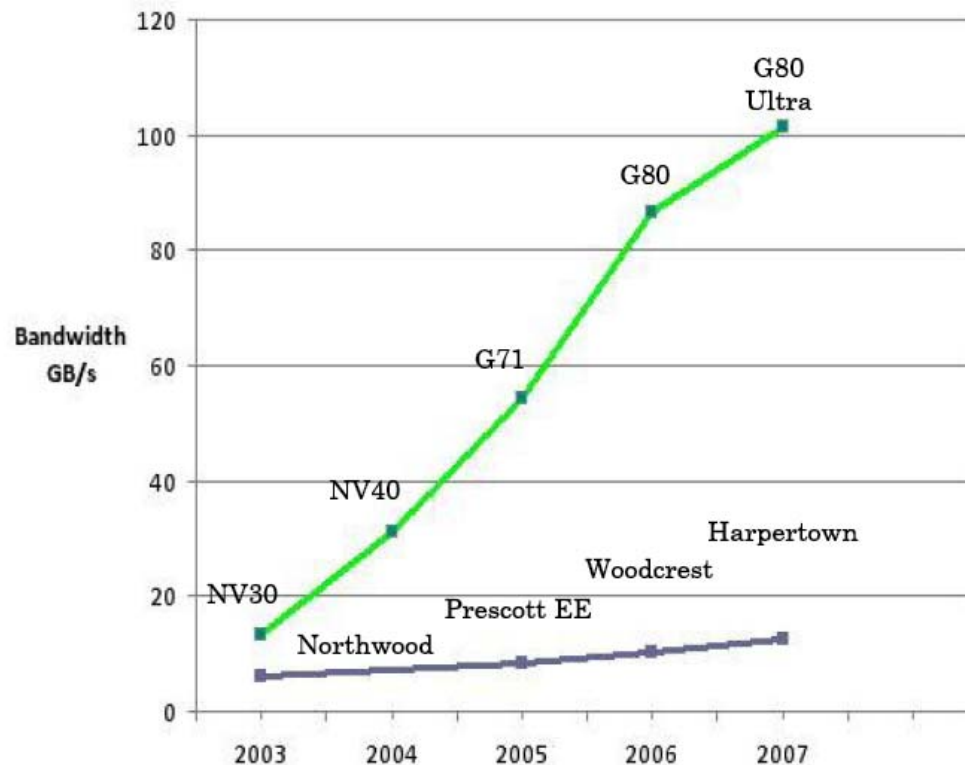
- **A quiet revolution and potential build-up**
  - Calculation: 800 GFLOPS vs. 80 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until two years ago, programmed through graphics API
- **GPU in every PC and workstation – massive volume and potential impact**



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

# Why Massively Parallel Processor

- **A quiet revolution and potential build-up**
  - Calculation: 800 GFLOPS vs. 80 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until two years ago, programmed through graphics API
- **GPU in every PC and workstation – massive volume and potential impact**



# Future Apps Reflect a Concurrent World

---

- **Exciting applications in future mass computing market have been traditionally considered “supercomputing applications”**
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products
  - These “Super-apps” represent and model physical, concurrent world
- **Various granularities of parallelism exist, but...**
  - programming model must not hinder parallel implementation
  - data delivery needs careful management

---

# Architecture Classification

# Flynn's Taxonomy [Flynn 72]

---

- **SISD - Single Instruction Single Data**
  - $(\text{inst}, \text{data}) \rightarrow \text{result}$
  - $a + b = c$
- **MISD - Multiple Instruction Single Data**
  - $((\text{inst}_k)_k, \text{data}) \rightarrow (\text{result}_k)_k$
  - $a (+, *, /, \%) b = (c_0, c_1, c_2, c_3)$
- **SIMD - Single Instruction Multiple Data**
  - $(\text{inst}, (\text{data}_k)_k) \rightarrow (\text{result}_k)_k$
  - $(a_0, a_1, a_2, a_3) + (b_0, b_1, b_2, b_3) = (c_0, c_1, c_2, c_3)$
- **MIMD - Multiple Instruction Multiple Data**
  - $((\text{inst}_k)_k, (\text{data}_k)_k) \rightarrow (\text{result}_k)_k$
  - $(a_0, a_1, a_2, a_3) (+, *, /, \%) (b_0, b_1, b_2, b_3) = (c_0, c_1, c_2, c_3)$

# Specializations SIMD

---

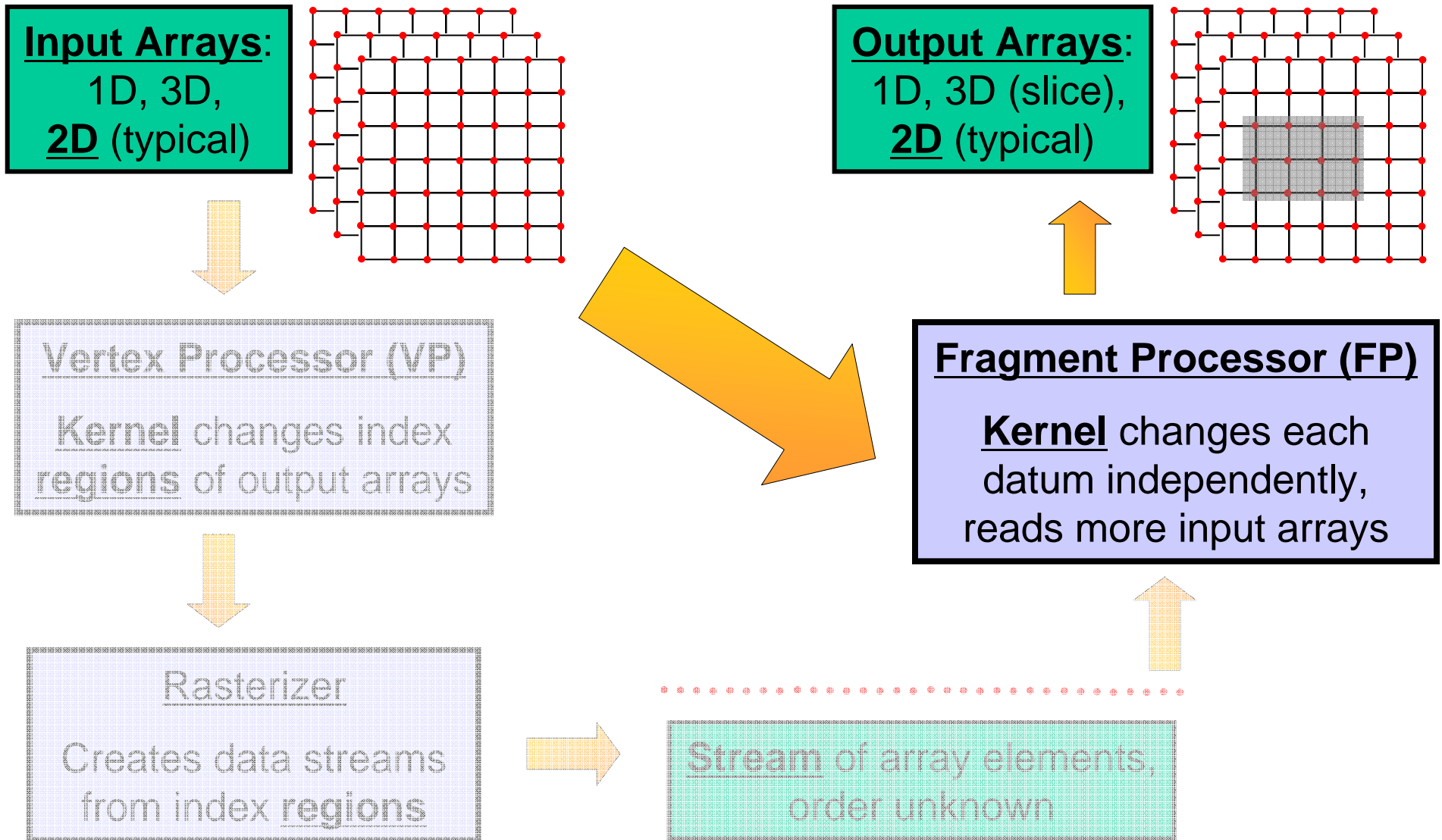
- **SIMD variant: Vector Processing**
  - Fixed HW and SW vector width
  - $(a_0, a_1, a_2, 0) + (b_0, b_1, b_2, 0) = (c_0, c_1, c_2, 0)$
  - Possible coarse configurability, e.g. double2 ops vs. float4 ops
- **SIMD variant: SIMT - Single Instruction Multiple Threads**
  - Fixed HW but flexible SW vector width
  - $(a_0, a_1, a_2, a_3, a_4) + (b_0, b_1, b_2, b_3, b_4) = (c_0, c_1, c_2, c_3, c_4)$
  - Interaction of components similar to MIMD, e.g. in shared memory
- **SIMD/MIMD variant: VLIW – Very Long Instruction Word**
  - Fixed HW vector width, one instruction with sub instructions
  - $((\text{inst.sub}_k)_k, (\text{data}_k)_k) \rightarrow (\text{result}_k)_k$
  - $(a_0, a_1, a_2, a_3) (+, *, /, \%) (b_0, b_1, b_2, b_3) = (c_0, c_1, c_2, c_3)$
  - Multiple instructions of single instruction type multiple data

# Specializations MIMD

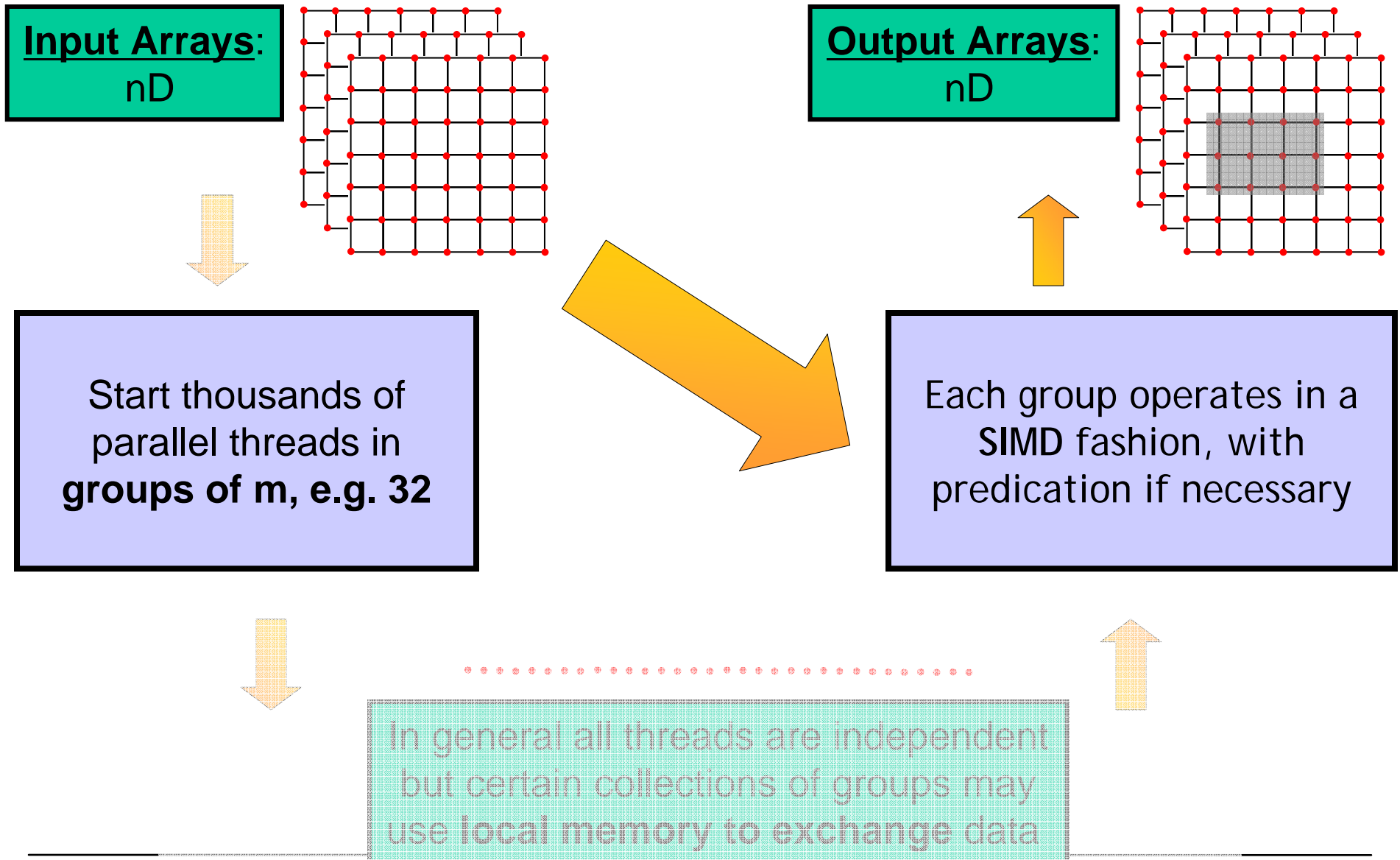
---

- **MIMD variant: SPMD – Single Program Multiple Data**
  - The program is the same but execution can be out of sync,
  - e.g. because of execution speed or data dependent code branches
  - The current program pointers may differ
  - Most common MIMD, easier reasoning about synchronization
- **MIMD variant: MPMD – Multiple Program Multiple Data**
  - Different instruction streams and program pointers.
  - Full flexibility, but difficult to develop and maintain
- **Synchronization / Interaction in MIMD**
  - Direct access to the registers or local memory of other units
  - Shared memory (local or global) communication
  - Message passing, e.g. the popular MPI library

# The GPU is a Fast, Parallel Array Processor



# The GPU is a Fast, Highly Multi-Threaded Processor

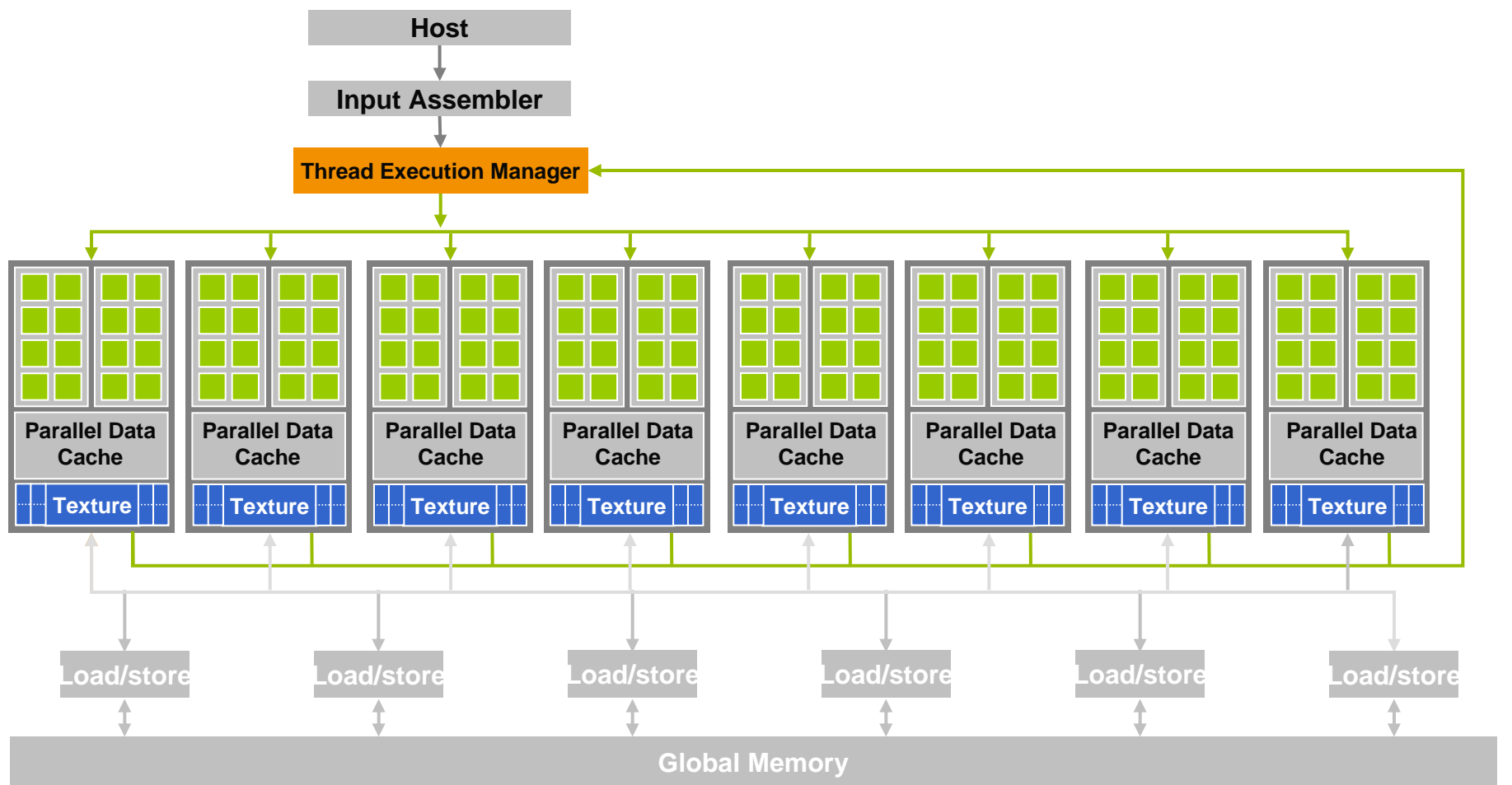


---

# CUDA Programming Model

# GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU

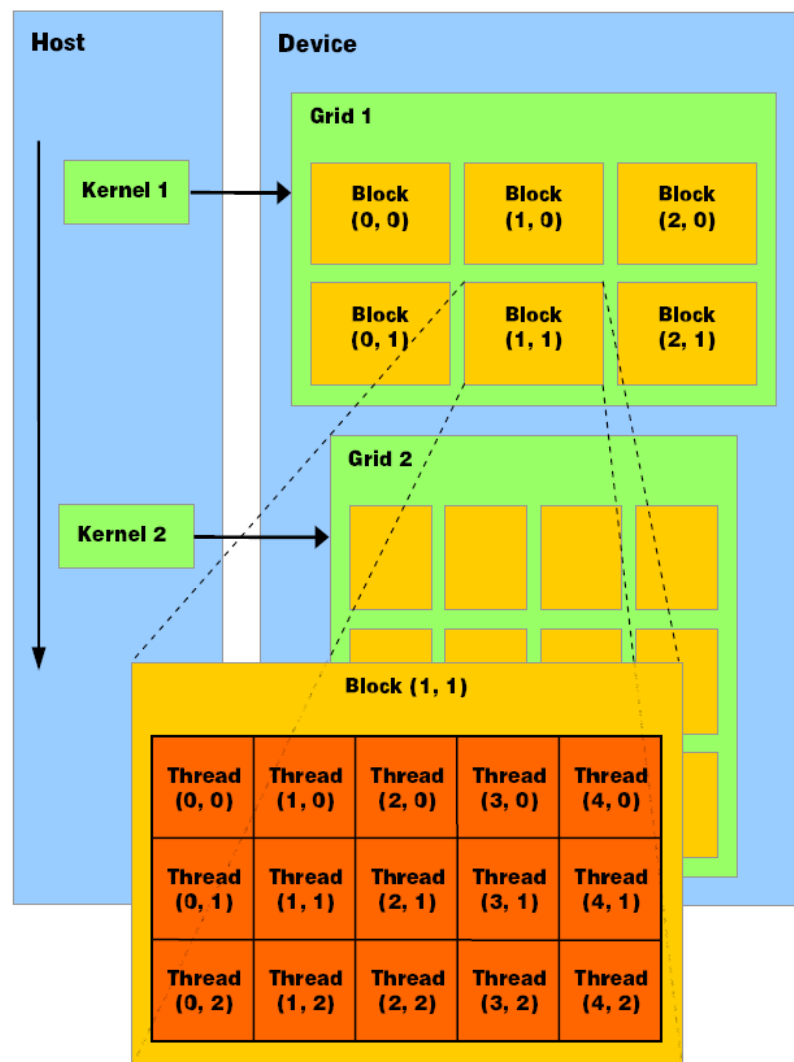


# Programming Model

- **Programming Model**

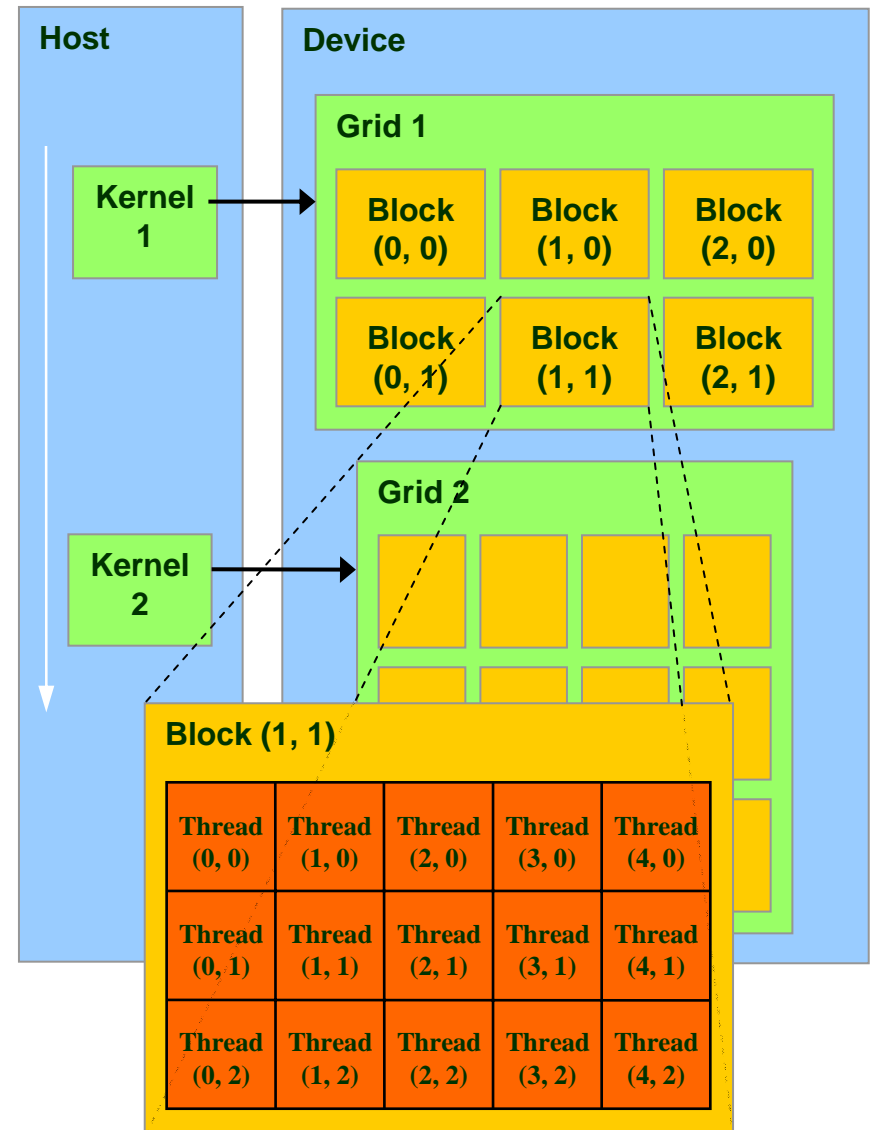
- The programmer writes a kernel (in C) for each task he or she wishes to perform
- The application splits the data to be processed into grids of thread blocks
- When a kernel is launched, each block is allocated to a single TP
- Threads of a given block are time sliced onto SPs contained within that block's TP

Many problems have natural grid structure, but decomposing data into threads can be difficult in general



# Thread Batching: Grids and Bloc

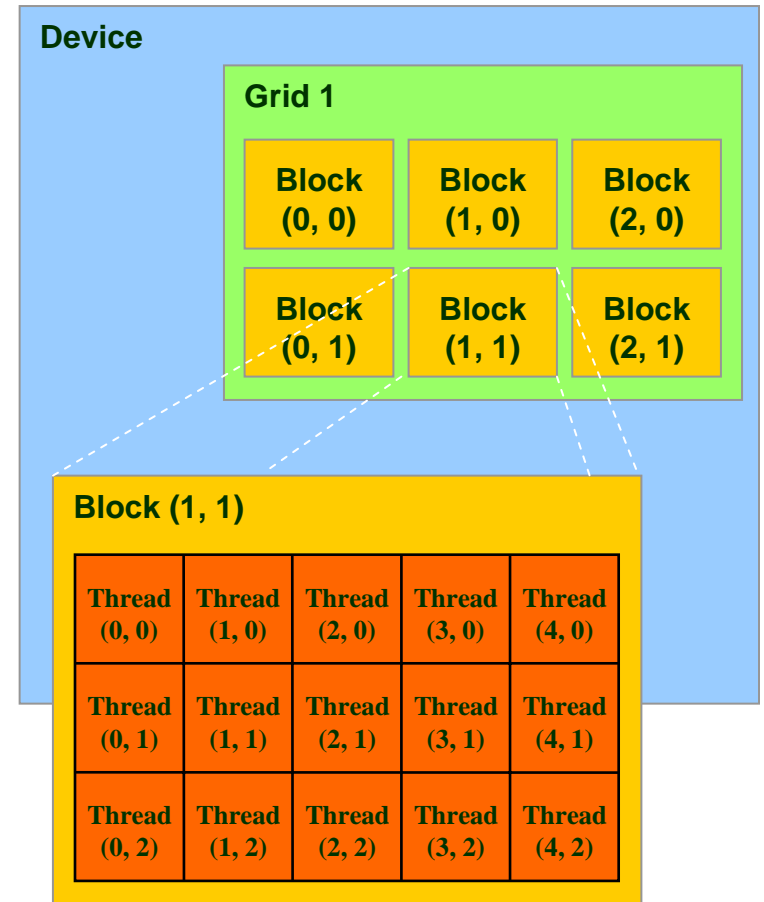
- A kernel is executed as a **grid of thread blocks**
  - All threads share data memory space
- A **thread block** is a batch of threads that can **cooperate** with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency **shared memory**
- **Two threads from two different blocks cannot cooperate**



Courtesy: NDVIA

# Block and Thread IDs

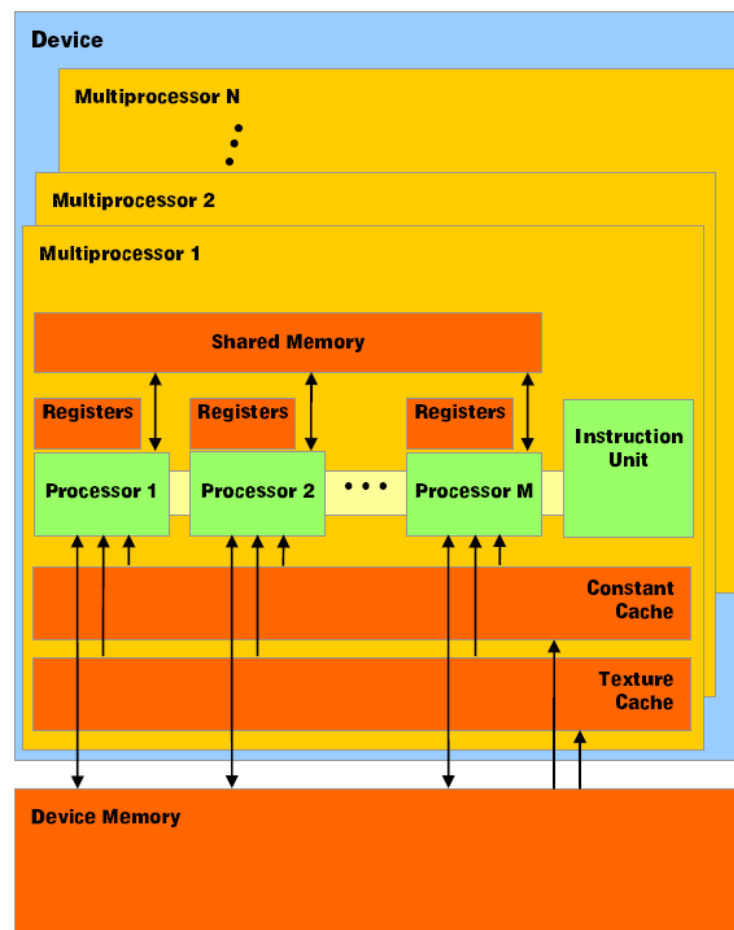
- **Threads and blocks have IDs**
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- **Simplifies memory addressing when processing multidimensional data**
  - Image processing
  - Solving PDEs on volumes
  - ...



Courtesy: NDVIA

# Programming Model: Memory Spaces

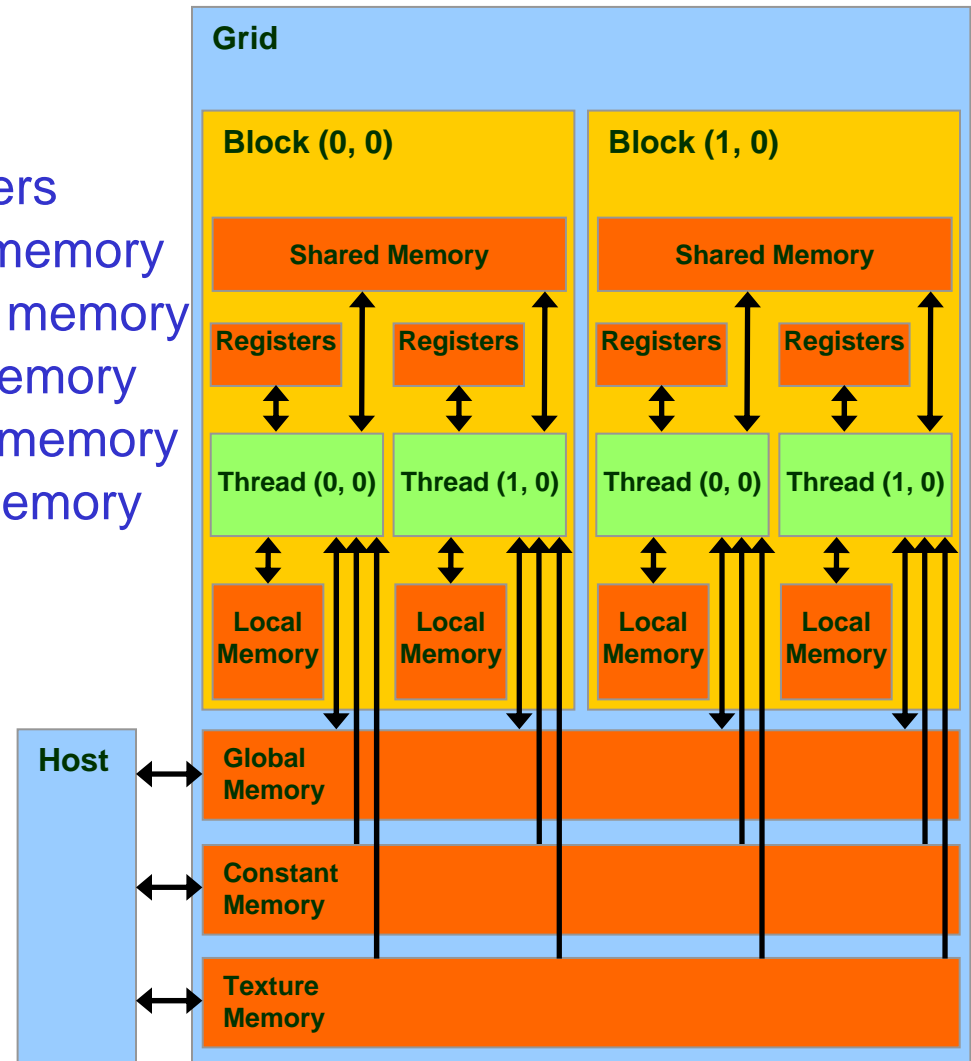
- **Global Memory**
  - Read-write per-grid
  - Hundreds of MBs
  - Very slow (600 clocks)
- **Texture Memory**
  - Read-only per-grid
  - Hundreds of MBs
  - Slow first access, but cached
  - Built-in filtering, clamping
- **Constant Memory**
- **Shared! Memory**
  - Read-write per-block
  - 16 KB per block
  - Very fast (4 clocks)
- **Registers**
  - Unique per thread



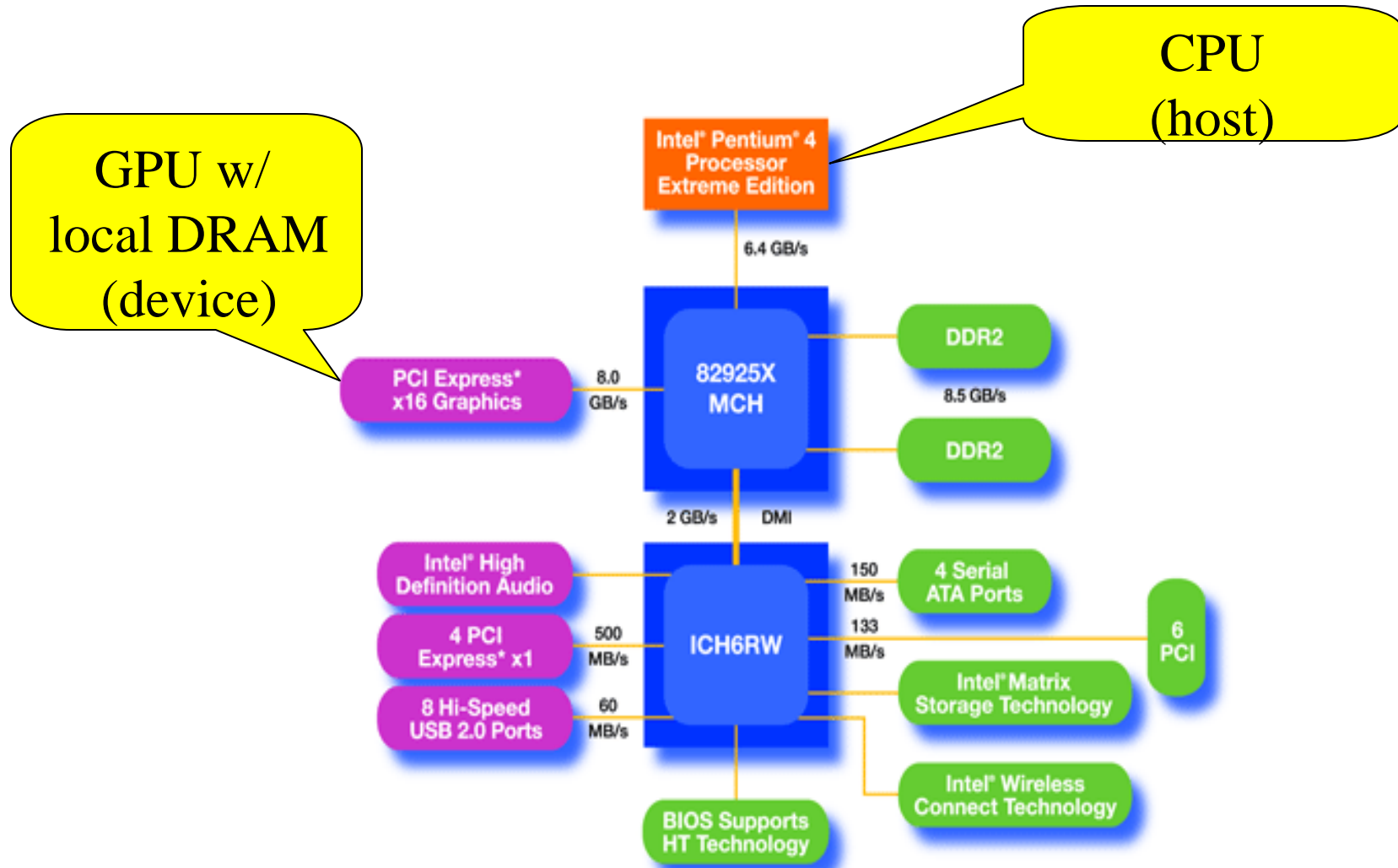
# Programming Model: Memory Spaces

- **Each thread can:**
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- **The host can read/write global, constant, and texture memory**



# An Example of Physical Reality Behind CUDA



# CUDA Programming Model: A Highly Multithreaded Coprocessor

---

- **The GPU is viewed as a compute device that:**
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (**device memory**)
  - Runs many **threads in parallel**
- **Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads**
- **Differences between GPU and CPU threads**
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

# Execution Model

---

- **Warps**
  - Each block is split into SIMD groups of threads called *warps*
  - Warps are swapped in and out via thread scheduling
  - Threads within a warp execute in lock step
  - Threads are assigned to warps consecutively by their thread ID
  - Issue order of warps and blocks is undefined, but there are synchronization primitives
  
- **Performance**
  - Branches are predicated
  - Divergence within a warp should be avoided if possible
  - Memory coherence extremely important
  - Always try to read/write in a coalesced manner

# CUDA

---



- **“Compute Unified Device Architecture”**
- **General purpose programming model**
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- **Targeted software stack**
  - Compute oriented drivers, language, and tools
- **Driver for loading computation programs into GPU**
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & read-back speeds
  - Explicit GPU memory management
- **Not another graphics API**



# Cuda

---

- **Compute Unified Device Architecture**
  - Unified hardware and software specification for parallel computation
  - Simple extensions to C language to allow code to run on the GPU
  - Developed by and for NVIDIA (introduced with the GeForce 8800 series)
  - Much easier to use than ATI's Close To Metal hardware interface
- **Benefits and Features**
  - Application controlled SIMD program structure
  - Fully general load/store to GPU memory
  - Totally untyped (not limited to texture storage)
  - No limits on branching, looping, etc.
  - Full integer and bit instructions
  - Supports pointers
  - Explicitly managed memory down to cache level
  - No graphics code (although interoperability with OpenGL/D3D is supported)

# What is the GPU Good at?

---

- **The GPU is good at**
  - data-parallel processing**
    - The same computation executed on many data elements in parallel – low control flow overhead
  - with high floating point arithmetic intensity**
    - Many calculations per memory access
    - (Currently also need high floating point to integer ratio)
- **High floating-point arithmetic intensity and many data elements mean that memory access latency can be hidden with calculations instead of big data caches – Still need to avoid bandwidth saturation!**

# Application Programming Interface

---

- **The API is an extension to the C/C++ programming language**
- **It consists of:**
  - Language extensions
    - To target portions of the code for execution on the device
    - Two stage compilation (e.g. nvcc + gcc)
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

# New Stuff

---

- **Function Quantifiers**

- `__device__` callable on the GPU from the GPU
- `__global__` callable on the GPU from the CPU
- `__host__` callable on the CPU from the CPU

- **Variable Quantifiers**

- `__device__` global memory on the GPU
- `__constant__` constant memory on the GPU
- `__shared__` shared per-block memory on the GPU

- ***Built-in Variables***

- `gridDim`, `blockDim` gives dimensions of grids and blocks in kernel
- `blockIdx`, `threadIdx` gives index of block and thread in kernel

- **Built-in Vector Types**

- `float2`, `float3`, `float4`, etc.

# Extended C

---

- Declspecs
    - **global, device, shared, local, constant**
  - Keywords
    - **threadIdx, blockIdx**
  - Intrinsic
    - **\_\_syncthreads**
  - Runtime API
    - **Memory, symbol, execution management**
  - Function launch
- ```
__device__ float filter[N];  
__global__ void convolve (float *image)  
{  
    __shared__ float region[M];  
    ...  
    region[threadIdx] = image[i];  
    __syncthreads()  
    ...  
    image[j] = result;  
}
```
- ```
// Allocate GPU memory  
void *myimage = cudaMalloc(bytes)
```
- ```
// 100 blocks, 10 threads per block  
convolve<<<100, 10>>> (myimage);
```

# CUDA Function Declarations

---

|                                            | Executed on the: | Only callable from the: |
|--------------------------------------------|------------------|-------------------------|
| <code>__device__ float DeviceFunc()</code> | <b>device</b>    | <b>device</b>           |
| <code>__global__ void KernelFunc()</code>  | <b>device</b>    | <b>host</b>             |
| <code>__host__ float HostFunc()</code>     | <b>host</b>      | <b>host</b>             |

- `__global__` defines a kernel function, must return **void**
- `__device__` and `__host__` can be used together

# CUDA Function Declarations (cont.)

---

- **\_\_device\_\_ functions cannot have their address taken**
- **For functions executed on the device:**
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

# Calling a Kernel Function – Thread Creation

---

- A kernel function must be called with an **execution configuration**:

```
__global__ void KernelFunc(...);  
dim3    DimGrid(100, 50);    // 5000 thread blocks  
dim3    DimBlock(4, 8, 8);   // 256 threads per block  
size_t  SharedMemBytes = 64; // 64 bytes of shared memory  
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);  
cudaThreadSynchronize();
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit sync needed for blocking
- Need to synchronize/ wait for the execution to end:  
`cudaThreadSynchronize()`

# Cuda Utility Functions

---

- Use them to hunt bugs more easily!
- It is so easy to start a kernel that does not do anything ;-(
- You better want to know if the device memory is really allocated, if the kernel is called or not.

```
CUDA_SAFE_CALL( cudaMalloc( &devData, size ) );  
...  
kernel<<< 400234, 1024 >>>();  
CUT_CHECK_ERROR("kernel failed\n");  
CUDA_SAFE_CALL( cudaThreadSynchronize() );
```

---

# CUDA Example

# Simple Example: Gamma Correction

- **2000x3500 pixels**
- **for each color channel**
  - $c = \text{pow}(c, 0.8)$



# Simple Example: Gamma Correction

- **2000x3500 pixels**
- **for each color channel**
  - $c = \text{pow}(c, 0.8)$



# Simple Example: Gamma Correction

- **2000x3500 pixels**
- **for each color channel**
  - $c = \text{pow}(c, 0.8)$



compute with Cuda

# Simple Example: Gamma Correction

- **2000x3500 pixels**
- **for each color channel**
  - $c = \text{pow}(c, 0.8)$
- **it is close to instantaneous ;-)**



# C-Code

---

```
float gammaFcn(const float& _src, const float _gamma) {  
    return 255. * powf( _src / 255., _gamma);  
}
```

```
float* img;  
int w,h;  
float gamma = atof( argv[acount++] );  
// read in the image  
readPPM( argv[acount++], w, h, &img );  
  
// loop over all pixels and colors  
for (int i = 0; i < w*h* 3; ++i) {  
    img[i] = gammaFcn( img[i], gamma );  
}  
// write image  
writePPM( argv[acount++], w, h, (float*)img );  
delete[] img;
```

---

# Principle CUDA How-To

# Principle Mode of Cuda Programming

---

## 1. Upload data to GPU

- various different ways
- try to minimize this upload

## 2. Execute the kernel

- each thread typically produces one output element  
(the most easy way of thinking about parallelism)

## 3. Download and analyze the results

- one can distill results down to a single number on the GPU
- programming effort might be reduced if downloading the results of all threads, computing a final aggregation on the CPU

# Principle Structure of the Kernel

---

- 1. Perform some pointer calculations**
    - based on **threadIdx**, **blockIdx**, **gridDim**, **blockDim**
  - 2. Upload the required data into shared memory**
  - 3. Perform actual computation**
    - each thread computes one output value
    - store result in global memory
- **1. and 2. add approximate twice the lines of code to your task!**

# Allocation&Up/Download of Data

---

- **allocation (standard memory)**

```
float* devData;  
cudaMalloc( (void**)&devData, nElements * sizeof(float) );  
...  
cudaFree( devData );
```

- **upload/download**

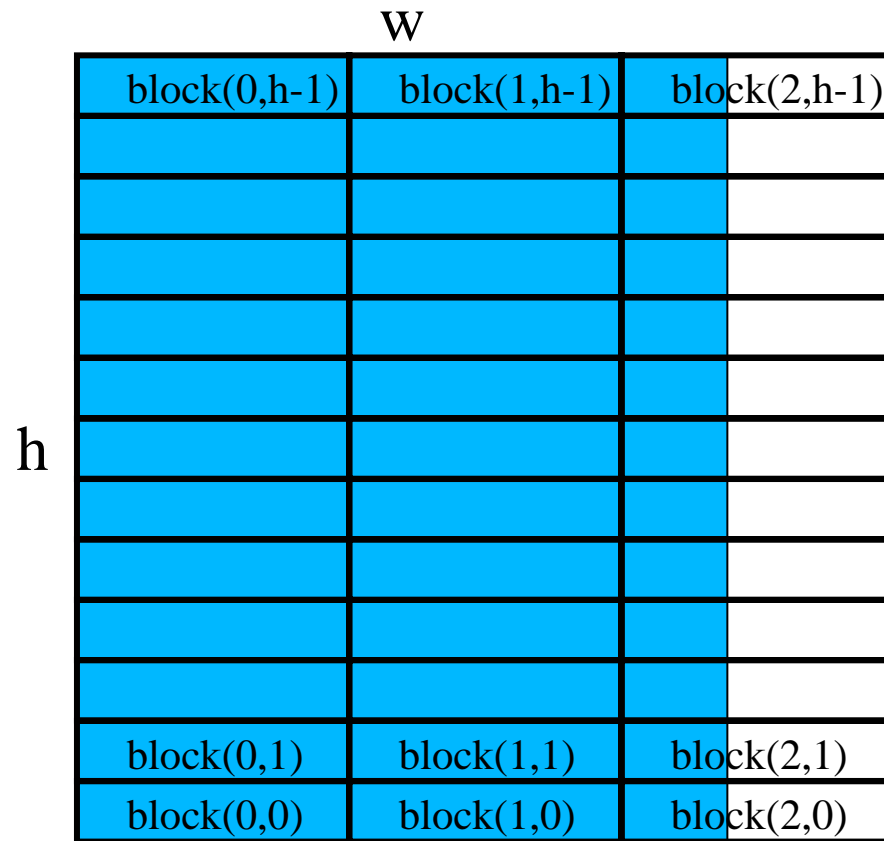
```
float* hostData = new float[imgsize];  
// upload  
cudaMemcpy( devData, hostData, size, cudaMemcpyHostToDevice );  
  
// download  
cudaMemcpy( hostData, devData, size, cudaMemcpyDeviceToHost );
```

# Subdividing into Blocks&Threads

---

- **in this example**

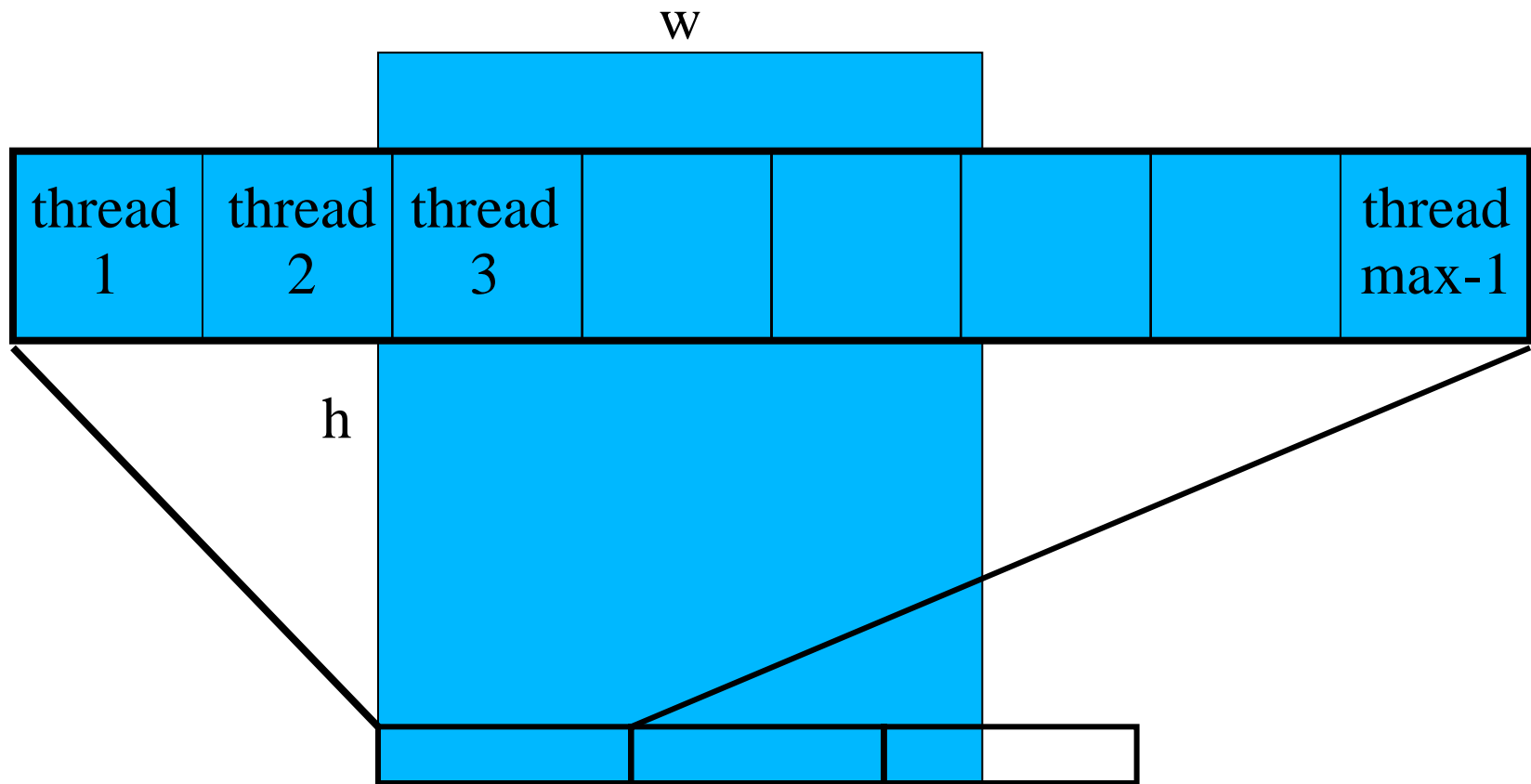
- 1D block of threads: `dim threads(MAX_THREADS)`
- 2D grid of blocks: `dim blocks( w / MAX_THREADS +1, h)`



# Subdividing into Blocks&Threads

- **in this example**

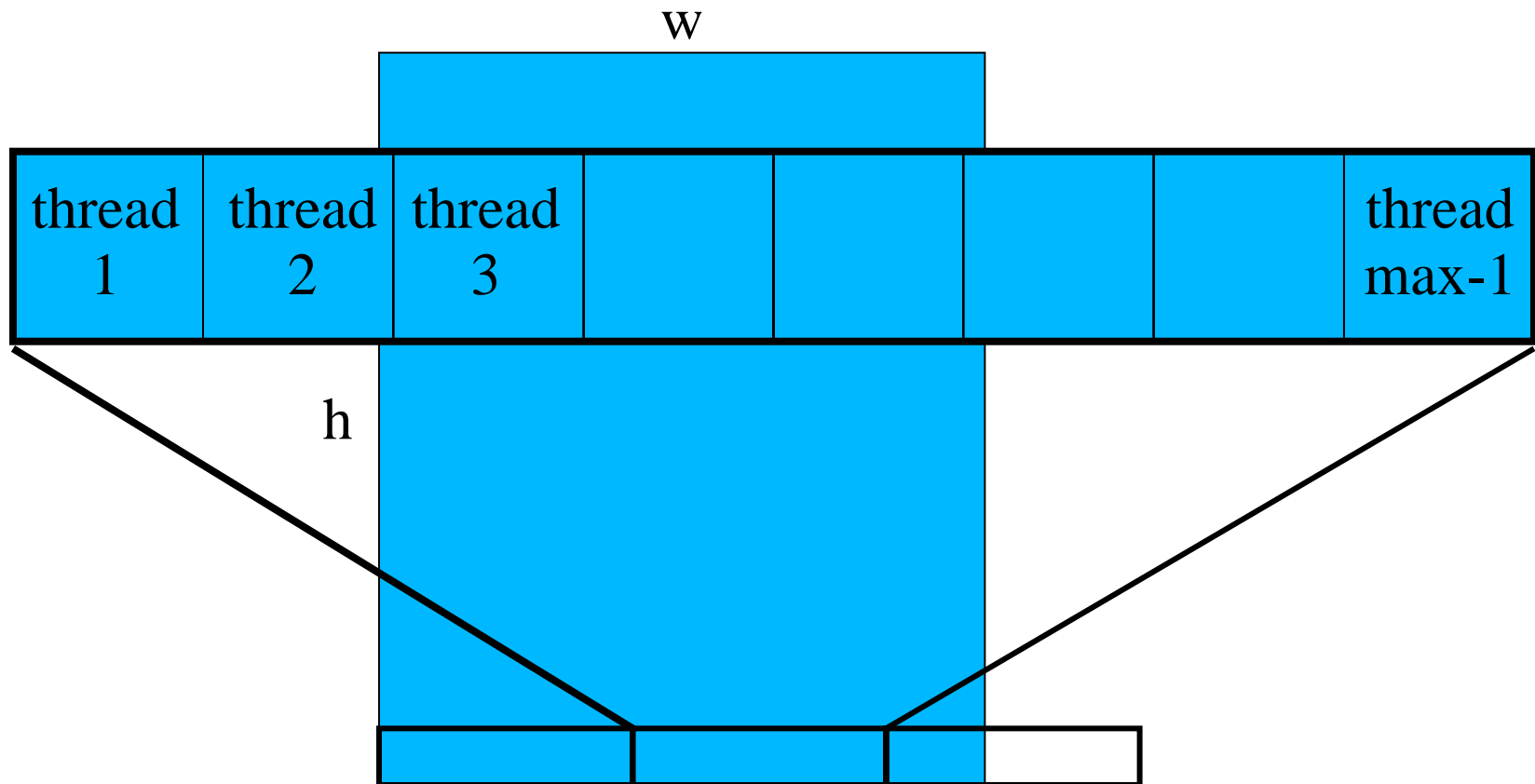
- 1D block of threads: `dim threads(MAX_THREADS)`
- 2D grid of blocks: `dim blocks( w / MAX_THREADS +1, h)`



# Subdividing into Blocks&Threads

- **in this example**

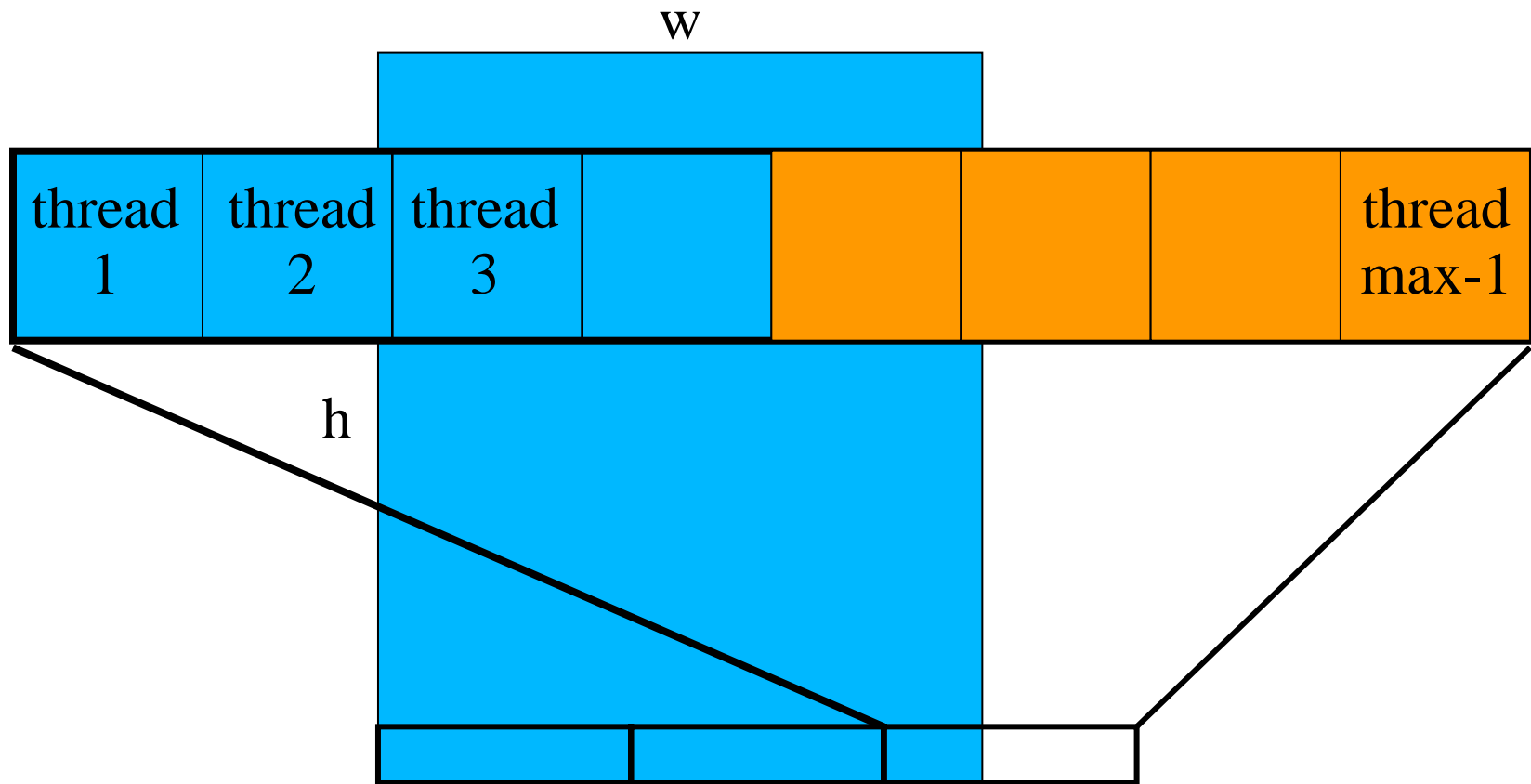
- 1D block of threads: `dim threads(MAX_THREADS)`
- 2D grid of blocks: `dim blocks( w / MAX_THREADS +1, h)`



# Subdividing into Blocks&Threads

- **in this example**

- 1D block of threads: `dim threads(MAX_THREADS)`
- 2D grid of blocks: `dim blocks( w / MAX_THREADS +1, h)`



# The Gamma Device Function

---

```
__device__ float gammaFcn(  
    const float& _src, const float _gamma ) {  
    return 255. * __powf( _src / 255., _gamma);  
}
```

- **can only be called on the device, i.e. from *within* a kernel function**

# The Kernel

---

```
__global__ void gammaKernel(  
    float *_dst, const float* _src, int _w, float _gamma )  
{  
    // index calculations  
    int x = blockIdx.x * MAX_THREADS + threadIdx.x;  
    int y = blockIdx.y;  
    int pos = y * _w + x;  
  
    // there could be more threads than pixels  
    // in the last block of each line  
    if (x < _w)  
        _dst[ pos ] = gammaFcn( _src[ pos ], _gamma);  
}
```

# Main

---

```
#include <cutil.h> // cuda utils
#include <vector_types.h> // definition of float123
...
float* img;
int w,h;
float gamma = atof( argv[acount++] );
readPPM( argv[acount++], w, h, &img );

float* gpuImg;
float* gpuResImg;

// allocate image on the device
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuImg, w*h*3 *
    sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuResImg, w*h* 3
    * sizeof(float) ) );
// copy the image to the device
CUDA_SAFE_CALL( cudaMemcpy( gpuImg, img, w*h*3 *
    sizeof(float), cudaMemcpyHostToDevice) );
...
```

# Main (2)

---

```
// calculate the block dimensions
dim3 threadBlock( MAX_THREADS, 1 );
dim3 blockGrid( (w*3)/MAX_THREADS +1 , h, 1);

// call the kernel
gammaKernel<<< blockGrid, threadBlock >>>( gpuResImg,
    gpuImg, w * 3, gamma);
CUT_CHECK_ERROR("gamma filter failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );

// download result
CUDA_SAFE_CALL( cudaMemcpy( img, gpuResImg, w*h*3 *
    sizeof(float), cudaMemcpyDeviceToHost) );

// free device memory
cudaFree( gpuResImg );
cudaFree( gpuImg );
writePPM( argv[acount++], w, h, (float*)img );
delete[] img;
```

# First steps

---

- **locate nvcc**
  - `nvcc -V`
  - release 2.0
- **locate cuda\_sdk**
- **make your own project folder**
- **compile with:** `nvcc -o gamma.o -c gamma.cu`
- **link with (if necessary):** `g++ -o gamma.o`

# Practical Exercises

---

- **read cmdline.txt**
- **compile with**
- **tmk -max**
  
- **solutions are provided as well**  
**(not necessarily optimal but working)**
  
- **have a look at**
- **`~cfuchs/new_proj/cuda_sdk_1.1/projects`**

# Practical Exercises

---

- **Where to run?**
- **in Ulm ssh to**
  - {wsl26,wsl27,wsl28}.informatik.uni-ulm.de (GTX 280)
  - look into /opt/cuda/
- **in SB**
  - we will create a login
  - you have to work locally on the machines {cuda00, cuda01, cuda02}.mpi-inf.mpg.de