Introduction to GPGPUs

using CUDA

Sandra Wienke, M.Sc.
wienke@itc.rwth-aachen.de
IT Center, RWTH Aachen University

May 28th 2015

Links

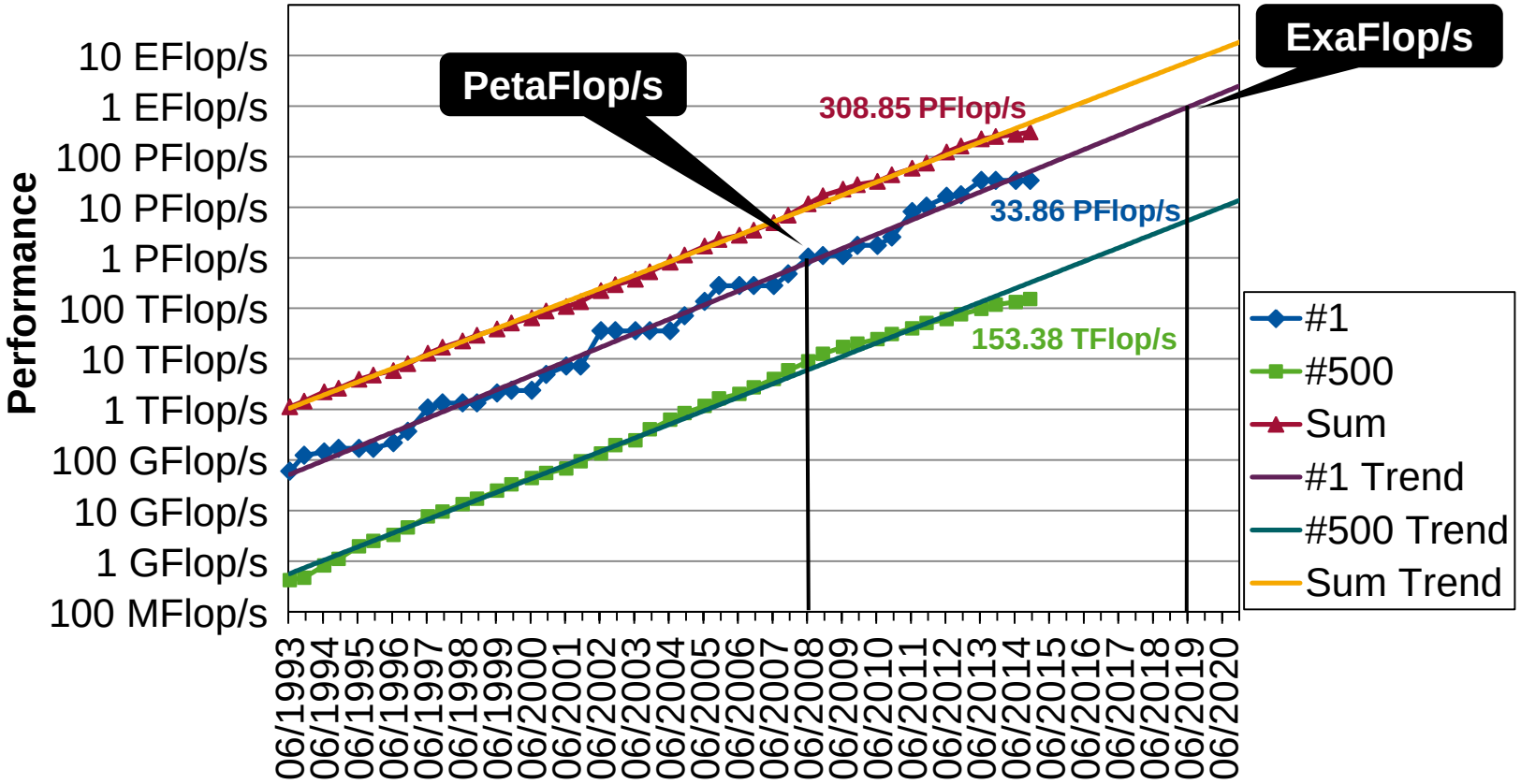
- **PPCES Workshop:** http://www.itc.rwth-aachen.de/ppces
  - → Slides and broadcasts on OpenMP, MPI and OpenACC
- **GPU Technology Conference (GTC) On-Demand** http://www.gputechconf.com/gtcnew/on-demand-GTC.php
  - → Collection of presentations given in the context of different GPU conferences & workshops
- **Webinars**
  - → Videos on different GPU topics: https://developer.nvidia.com/gpu-computing-webinars
- **NVIDIA CUDA Zone (Toolkit, Profiler, SDK, documentation,…):** http://www.nvidia.com/object/cuda_home_new.html
  - → CUDA Programming Guide
- **OpenACC Home:** www.openacc.org/

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **CUDA Basics**
  - → Offloading Regions
  - → Data Management
- **GPUs @ IT Center**
- **CUDA Advanced**
  - → Programming, Memory & Execution Model
  - → Maximize Global Memory Throughput
  - → Launch Configuration

Motivation

- **Why to care about accelerators?**
  - → Towards exa-flop computing (performance gain, but power constraints)
  - → Accelerators provide good performance per watt ratio (first step)

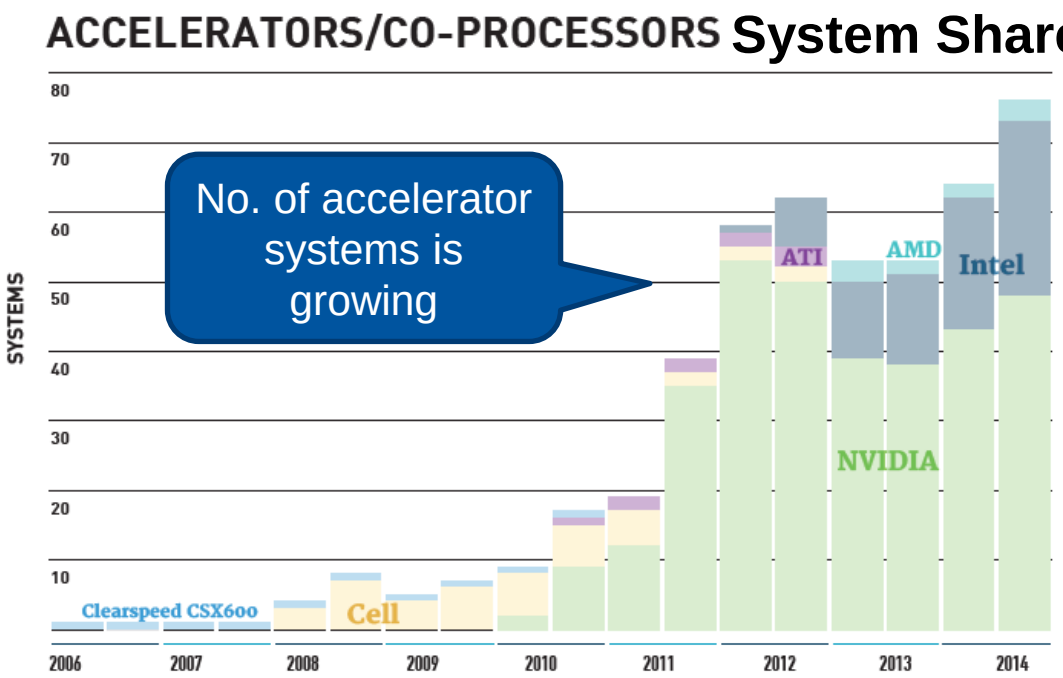


Source: Top500, 11/2014

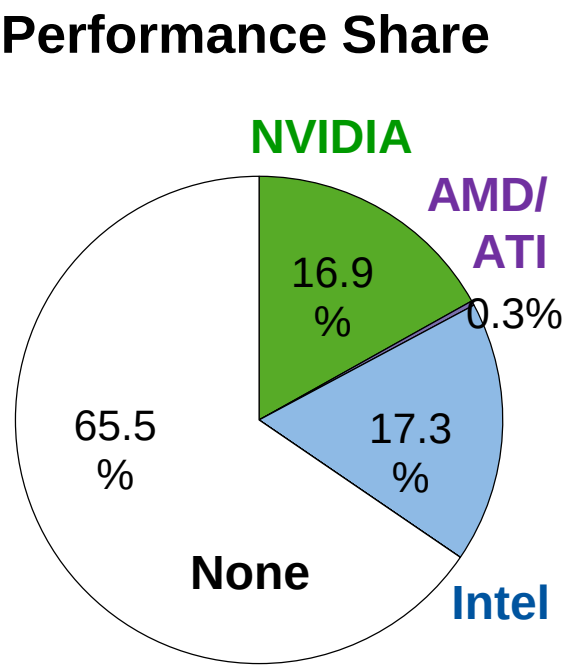
Motivation

- **Accelerators/ co-processors**
  - → GPGPUs (e.g. NVIDIA, AMD)
  - → Intel Many Integrated Core (MIC) Architecture (Intel Xeon Phi)
  - → FPGAs (e.g. Convey), DSPs (e.g. TI), …

Here, NVIDIA GPUs are the focus.



Source: Top500, 11/2014



Source: Top500, 6/2014

Motivation – Top500 (11/2014)

- **One half of the first 10 Top500 systems contains accelerators**

Rank	Name	Site	Rmax	Power (kW)	Mflops/ Watt	Accelerator/ Co-Processor
1	Tianhe-2 (MilkyWay-2)	National Super Computer Center in Guangzhou	33,862,700	17,808	1901.54	Intel Xeon Phi 31S1P
2	Titan	DOE/SC/Oak Ridge National Laboratory	17,590,000	8,209	2142.77	NVIDIA K20x
3	Sequoia	DOE/NNSA/LLNL	17,173,224	7,890	2176.58	None
4		RIKEN Advanced Institute for Computational Science (AICS)	10,510,000	12,660	830.18	None
5	Mira	DOE/SC/Argonne National Laboratory	8,586,612	3,945	2176.58	None
6	Piz Daint	Swiss National Supercomputing Centre (CSCS)	6,271,000	2,325	2697.2	NVIDIA K20x
7	Stampede	Texas Advanced Computing Center/Univ. of Texas	5,168,110	4,510	1145.92	Intel Xeon Phi SE10P
8	JUQUEEN	Forschungszentrum Juelich (FZJ)	5,008,857	2,301	2176.82	None
9	Vulcan	DOE/NNSA/LLNL	4,293,306	1,972	2177.13	None
10		Government	3,577,000	1,499	2386.42	Nvidia K40

Motivation – Green500 (11/2014)

Rank	Power(KW)	Mflops/Watt	Site	Accelerator
1	57,153	5271,8142	GSI Helmholtz Center	AMD FirePro S9150
2	37,83282752	4945,625592	High Energy Accelerator Research C	PEZY-SC
3	35,39	4447,584063	GSIC Center, Tokyo Institute of Tech	NVIDIA K20x
4	44,54	3962,73013	Cray Inc.	Nvidia K40m
5	52,62	3631,864623	Cambridge University	NVIDIA K20
6	54,6	3543,315018	Financial Institution	NVIDIA K20x
7	78,77	3517,83674	Center for Computational Sciences, U	NVIDIA K20x
8	44,4	3459,459459	SURFsara	Nvidia K40m
9	1753,66	3185,908329	Swiss National Supercomputing Cen	NVIDIA K20x
10	81,41	3131,06498	ROMEO HPC Center - Champagne-A	NVIDIA K20x
11	86,20	3019,686547	Commonwealth Scientific and Industri	Nvidia K20m
12	927,86	2951,941233	GSIC Center, Tokyo Institute of Tech	NVIDIA K20x
13	66,25	2629,418868	Financial Institution	NVIDIA K20x
14	66,25	2629,418868	Financial Institution	NVIDIA K20x
15	66,25	2629,418868	Financial Institution	NVIDIA K20x
16	66,25	2629,418868	Financial Institution	NVIDIA K20x
17	269,94	2629,102764	Max-Planck-Gesellschaft MPI/IPP	NVIDIA K20x
18	1227	2598,207009	Exploration & Production - Eni S.p.A	NVIDIA K20x
19	87	2540,022989	Tulane University	Intel Xeon Phi 7120P
20	101,93	2495,124105	Mississippi State University	Intel Xeon Phi 5110P
21	71	2401,408451	St. Petersburg Polytechnic University	Intel Xeon Phi 5120D
22	1498,9	2386,416706	Government	Nvidia K40
23	179,15	2351,10	King Abdulaziz City for Science and	AMD FirePro S10000

performance per watt

Comparison to server with 2x Intel Sandy Bridge@ 2GHz

- HPL: ~260 W
- Peak Performance: 256 GFLOPS

→ 985 MFLOPS/W

Source: Green500, 11/2014
* Performance data obtained from publicly available sources including TOP500

Overview

- **GPGPUs = General Purpose Graphics Processing Units**

- **History – a very brief overview**

  → ‘80s - ‘90s: Development is mainly driven by games

  Fixed-function 3D graphics pipeline

  Graphics APIs like OpenGL, DirectX popular

  → Since 2001: Programmable pixel and vertex shader in graphics pipeline

  (adjustments in OpenGL, DirectX)

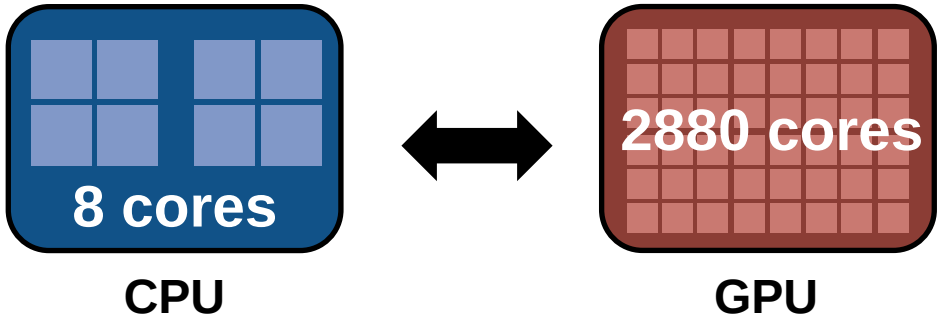
  Researchers take notice of performance growth of GPUs: Tasks must be cast into native graphics operations

  → Since 2006: Vertex/pixel shader are replaced by a single processor unit

  Support of programming language C, synchronization,…

  → “General purpose”

Comparison CPU ⇔ GPU

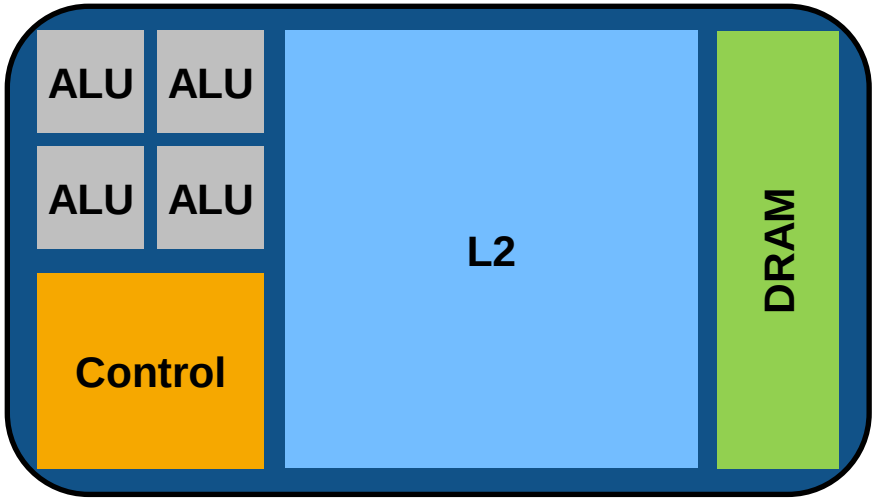


- **GPU-Threads**
  - → Thousands (“few” on CPU)
  - → Light-weight, little creation overhead
  - → Fast switching

- **Massively Parallel Processors**
- **Manycore Architecture**

Comparison CPU ⇔ GPU

- **Different design**



CPU

- → Optimized for low latencies
- → Huge caches
- → Control logic for out-of-order and speculative execution

GPU

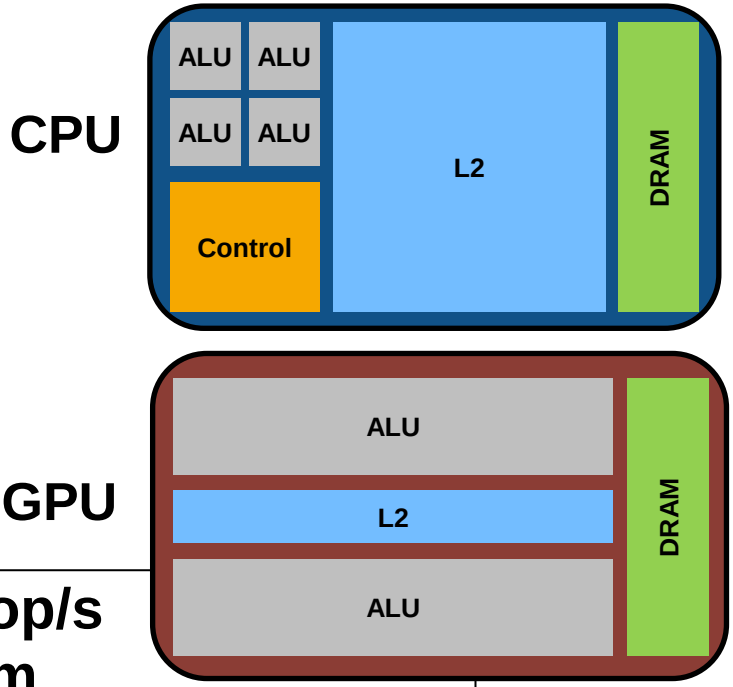
- → Optimized for data-parallel throughput
- → Architecture tolerant of memory latency
- → More transistors dedicated to computation

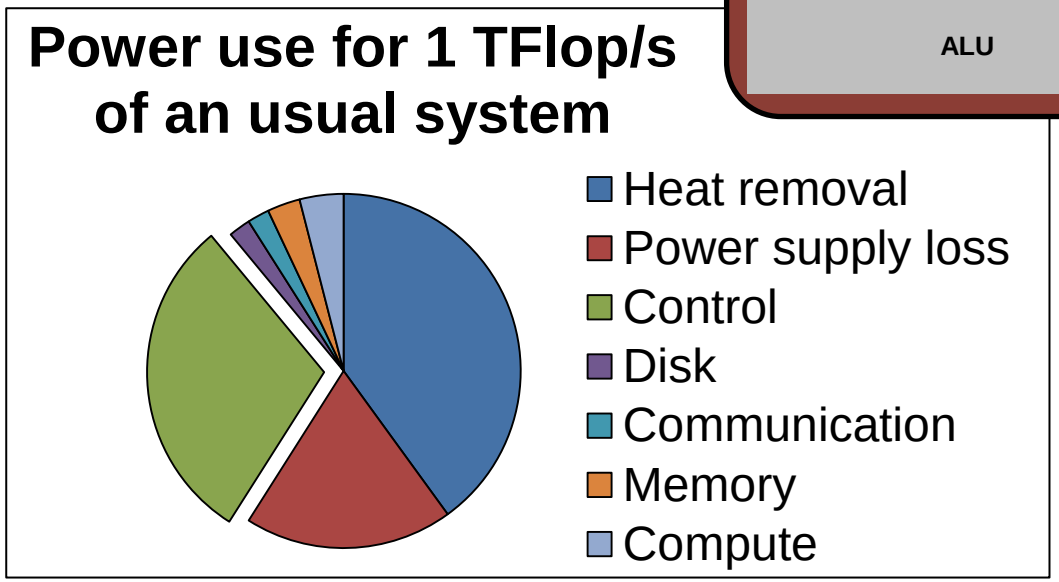
Why can accelerators deliver good performance watt ratio?

1. **High (peak) performance**
   - → More transistors for computation
     - → No control logic
     - → Small caches
2. **Low power consumption**
   - → Many low frequency cores

$$P \sim V^2 \cdot f$$

→ No control logic





Source: Andrey Semin (Intel): HPC systems energy efficiency optimization thru hardware-software co-design on Intel technologies, EnaHPC 2011; & S. Borkar, J. Gustafson

Agenda

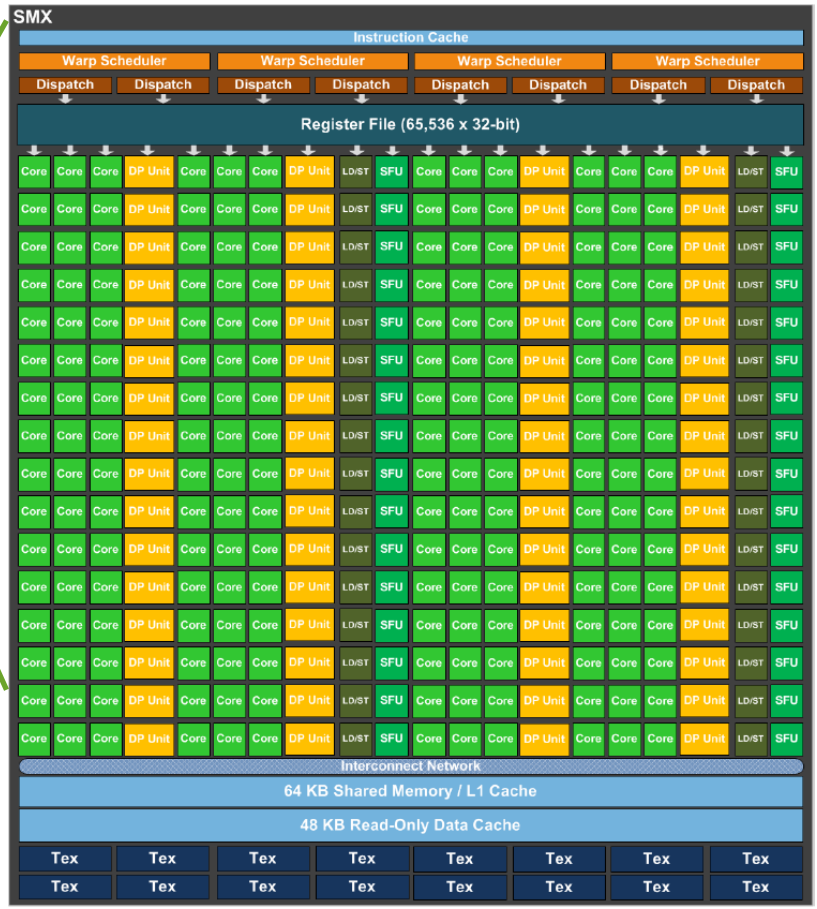
- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **CUDA Basics**
  - → Offloading Regions
  - → Data Management
- **GPUs @ IT Center**
- **CUDA Advanced**
  - → Programming, Memory & Execution Model
  - → Maximize Global Memory Throughput
  - → Launch Configuration

GPU architecture: Kepler

- **7.1 billion transistors**
- **13-15 streaming multiprocessors extreme (SMX)**
  - → Each comprises 192 cores
- **2496-2880 cores**
- **Memory hierarchy**
- **Peak performance (K20)**
  - → SP: 3.52 TFlops
  - → DP: 1.17 TFlops
- **ECC support**
- **Compute capability: 3.5**
  - → E.g. dynamic parallelism = possibility to launch dynamically new work from GPU

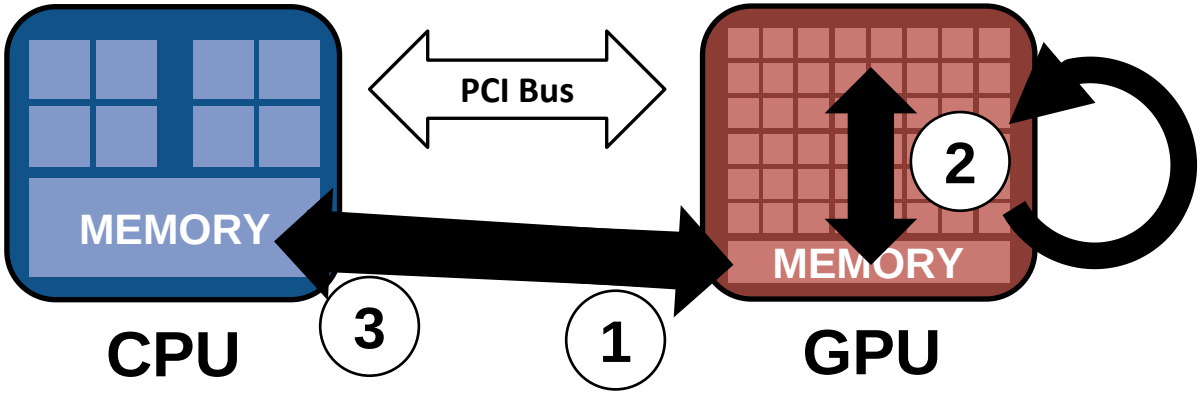
GPU

SMX



http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

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.

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **CUDA Basics**
  - → Offloading Regions
  - → Data Management
- **GPUs @ IT Center**
- **CUDA Advanced**
  - → Programming, Memory & Execution Model
  - → Maximize Global Memory Throughput
  - → Launch Configuration

Example SAXPY – CPU

```
void saxpyCPU(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));

  // Initialize x, y
  for(int i=0; i<n; ++i){
    x[i]=i;
    y[i]=5.0*i-1.0;
  }

  // Invoke serial SAXPY kernel
  saxpyCPU(n, a, x, y);

  free(x); free(y);
  return 0;
}
```

SAXPY = Single-precision real Alpha X Plus Y

$$\vec{y} = \alpha \cdot \vec{x} + \vec{y}$$

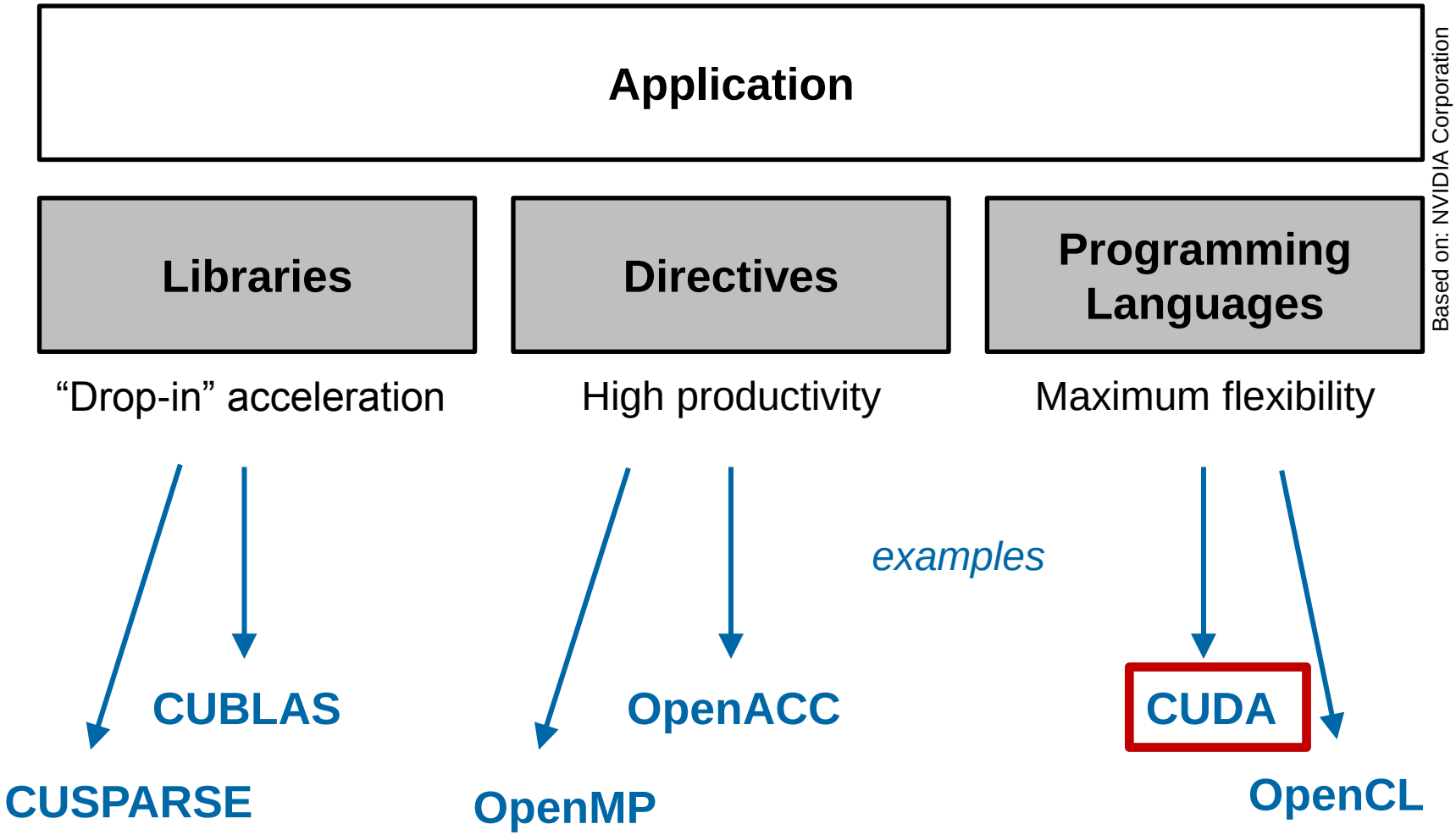
Example SAXPY – OpenACC

```
void saxpyOpenACC(int n, float a, float *x, float *y) {
#pragma acc parallel loop
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));

  // Initialize x, y
  for(int i=0; i<n; ++i){
    x[i]=i;
    y[i]=5.0*i-1.0;
  }

  // Invoke serial SAXPY kernel
  saxpyOpenACC(n, a, x, y);

  free(x); free(y);
  return 0;
}
```

Example SAXPY – CUDA

```
__global__ void saxpy_parallel(int n,
   float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (i < n){
        y[i] = a*x[i] + y[i];
    }
}

int main(int argc, char* argv[]) {
 int n = 10240; float a = 2.0f;
 float* h_x,*h_y; // Pointer to CPU memory
 h_x = (float*) malloc(n* sizeof(float));
 h_y = (float*) malloc(n* sizeof(float));
 // Initialize h_x, h_y
 for(int i=0; i<n; ++i){
   h_x[i]=i;
   h_y[i]=5.0*i-1.0;
 }
 float *d_x,*d_y; // Pointer to GPU memory
 cudaMalloc(&d_x, n*sizeof(float));
 cudaMalloc(&d_y, n*sizeof(float));

 cudaMemcpy(d_x, h_x, n * sizeof(float),
   cudaMemcpyHostToDevice);
 cudaMemcpy(d_y, h_y, n * sizeof(float),
   cudaMemcpyHostToDevice);

 // Invoke parallel SAXPY kernel
 dim3 threadsPerBlock(128);
 dim3 blocksPerGrid(n/threadsPerBlock.x);
 saxpy_parallel<<<blocksPerGrid,
   threadsPerBlock>>>(n, 2.0, d_x, d_y);

 cudaMemcpy(h_y, d_y, n * sizeof(float),
   cudaMemcpyDeviceToHost);

 cudaFree(d_x); cudaFree(d_y);
 free(h_x); free(h_y);
 return 0;
}
```

GPGPU Programming



GPGPU Programming

- **CUDA (Compute Unified Device Architecture)**
  - → C/C++ (NVIDIA): programming language, NVIDIA GPUs
  - → Fortran (PGI): NVIDIA‘s CUDA for Fortran, NVIDIA GPUs
- **OpenCL**
  - → C (Khronos Group): open standard, portable, CPU/GPU/…
- **OpenACC**
  - → C/C++, Fortran (PGI, Cray, CAPS, NVIDIA): Directive-based accelerator programming, industry standard published in Nov. 2011
- **OpenMP**
  - → C/C++, Fortran: Directive-based programming for hosts and accelerators, standard, portable, released in July 2013, implementations soon
- **…**

Kernel Launches

- **Setup GPU (e.g. driver, environment variables)**
- **Download + install CUDA Toolkit**

- **Compiling**

```
module load cuda                          # on our cluster
nvcc -arch=sm_20 saxpy.cu
```

→ `nvcc`: NVIDIA's compiler for C/C++ GPU code

→ `-arch=sm_20`: Set compute capability 2.0 (for Fermi GPUs)

→ Sets certain architecture features, e.g. enabling double precision floating point operations

Example SAXPY: Kernel Launches

CUDA

```
__global__ void saxpyCUDA(int n, float a, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n){
      y[i] = a*x[i] + y[i];
    }
}

int main(int argc, char* argv[])
{
  [..]
  // Invoke parallel SAXPY kernel
  dim3 threadsPerBlock(128);
  dim3 blocksPerGrid(n/threadsPerBlock.x);
  saxpyCUDA<<<blocksPerGrid, threadsPerBlock>>>(n,2.0,d_x,d_y);
  [..]
}
```

Kernel Launches

grey = background information
CUDA C

- **Kernel code**
  - → Function qualifiers: `__global__`, `__device__`, `__host__`
  - → Built-in variables: **`gridDim`**: contains dimensions of grid (type **`dim3`**)

    `blockDim` : contains dimensions of block (type **`dim3`**)

    `blockIdx` : contains block index within grid (type **`uint3`**)

    `threadIdx`: contains thread index within block (type **`uint3`**)
  - → Compute unique IDs, e.g. global 1D Idx:
    ```
    gIdx = blockIdx.x * blockDim.x + threadIdx.x
    ```
- **Kernel usage**
  - → Compiling with **`nvcc`** (creating PTX code)
  - Kernel arguments can be passed directly to the kernel
  - Kernel invocation with *execution configuration* (chevron syntax):
    ```
    func<<<dimGrid, dimBlock>>> (parameter)
    ```

Example SAXPY: Data Movement

CUDA

```
int main(int argc,char* argv[]){
  float* h_x,*h_y; // host pointer
  // Allocate and initialize h_x and h_y

  float *d_x,*d_y; // device pointer
  cudaMalloc(&d_x, n*sizeof(float));
  cudaMalloc(&d_y, n*sizeof(float));

  cudaMemcpy(d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice);

  // Invoke parallel SAXPY kernel

  cudaMemcpy(h_y, d_y, n * sizeof(float), cudaMemcpyDeviceToHost);

  cudaFree(d_x); cudaFree(d_y);
  free(h_x); free(h_y); return 0;
}
```

Data Movement

- **Variable type qualifiers**

  `__device__`, `__shared__`, `__constant__`

- **Memory management**

  `cudaMalloc(pointerToGPUMem, size)`

  `cudaFree(pointerToGPUMem)`

- **Memory transfer (synchronous)**

  `cudaMemcpy(dest, src, size, direction)`

```
direction:
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
cudaMemcpyDefault (with UVA)
```

Example SAXPY: all

- 3 steps for a basic program with CUDA

```
__global__ void saxpy_parallel(int n,
   float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (i < n){
        y[i] = a*x[i] + y[i];
}   }

int main(int argc, char* argv[]) {
 int n = 10240;
 float* h_x,*h_y; // Pointer to CPU memory
 // Allocate and initialize h_x and h_y
 float *d_x,*d_y; // Pointer to GPU memory
 cudaMalloc(&d_x, n*sizeof(float));
 cudaM
 cudaM                                  t),
  cuda
 cudaMemcpy(d_y, h_y, n * sizeof(float),
  cudaMemcpyHostToDevice);
// Invoke parallel SAXPY kernel
dim3 t                                  );
dim3 b                                  .x);
saxpy_parallel<<<blocksPerGrid,
  threadsPerBlock>>>(n, 2.0, d_x, d_y);
cudaMemcpy(h_y, d_y, n * sizeof(float),
  cudaMemcpyDeviceToHost);
cudaFr
cudaFr
free(h_x);
free(h_y);
return 0;
}
```

1. Allocate data on GPU + transfer data to GPU

2. Launch kernel

3. Transfer data to CPU + free data on GPU

Summary

- **GPUs may deliver high performance for certain applications**
  - → Huge number of cores
  - → Very simple architecture (no control logic, small caches)
- **GPU architecture**
  - → Many cores
  - → Distinct memory from CPU (PCIe)
  - → Control by CPU

  } “offloading”
- **GPU programming**
  - → CUDA for low-level (but highly-flexible) programming
  - → Directive-based approaches (standards) for high-level programming

Agenda

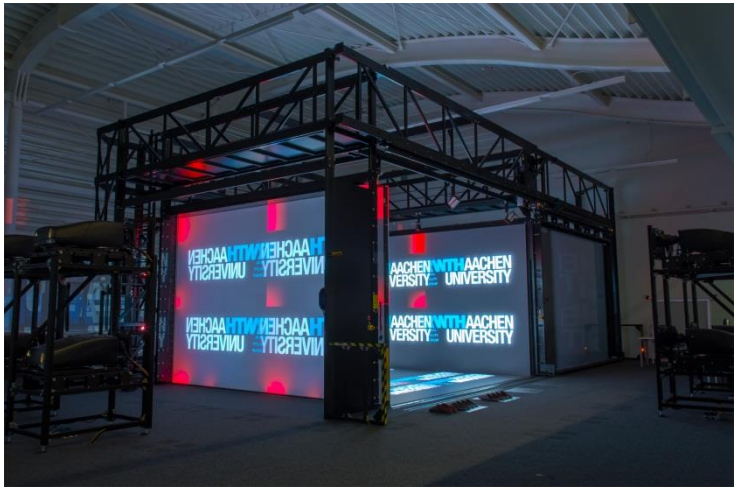
- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **CUDA Basics**
  - → Offloading Regions
  - → Data Management
- **GPUs @ IT Center**
- **CUDA Advanced**
  - → Programming, Memory & Execution Model
  - → Maximize Global Memory Throughput
  - → Launch Configuration

The RWTH GPU Cluster

Foto: C. Iwainsky

GPU Cluster: 57 Nvidia Quadro 6000

- **High utilization of resources**
  - → Daytime
    - →VR: new CAVE (49 GPUs)
    - →HPC: interactive software development (8 GPUs)
  - → Nighttime
    - →HPC: Processing of GPGPU compute jobs (55-57 GPUs)


aixCAVE, VR, RWTH Aachen, since June 2012

Hardware & Software Stack

- **24 rendering nodes**
  - → 2x NVIDIA Quadro 6000 GPUs (Fermi)
  - → 2x Intel Xeon X5650 EP @ 2.67GHz (Westmere, 12 cores)

- **GPU cluster software environment**
  - → CUDA Toolkit 6.5 — `module load cuda` → directory: `$CUDA_ROOT`
  - → PGI Compiler (OpenACC) — `module load pgi` (`module switch intel pgi`)
  - → TotalView (CUDA Debugging) — `module load totalview`

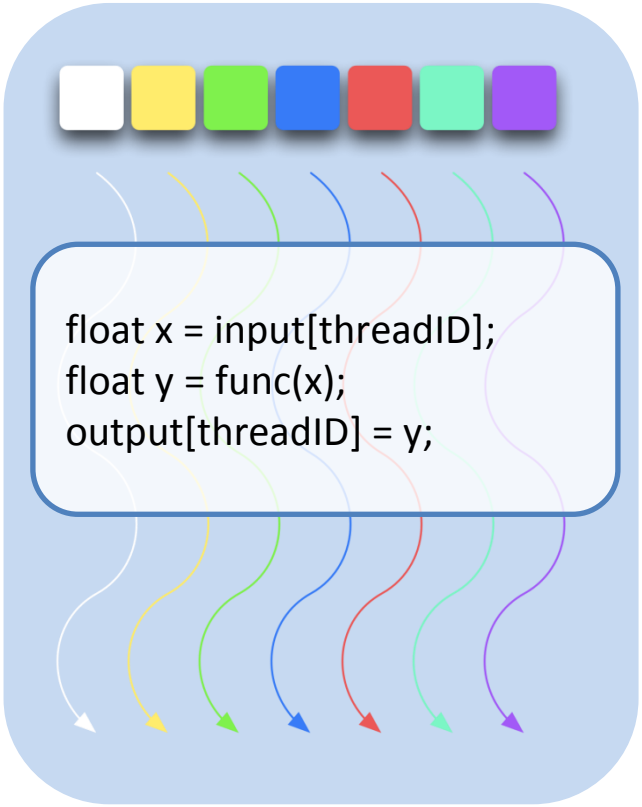
Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **CUDA Basics**
  - → Offloading Regions
  - → Data Management
- **GPUs @ IT Center**
- **CUDA Advanced**
  - → Programming, Memory & Execution Model
  - → Maximize Global Memory Throughput
  - → Launch Configuration

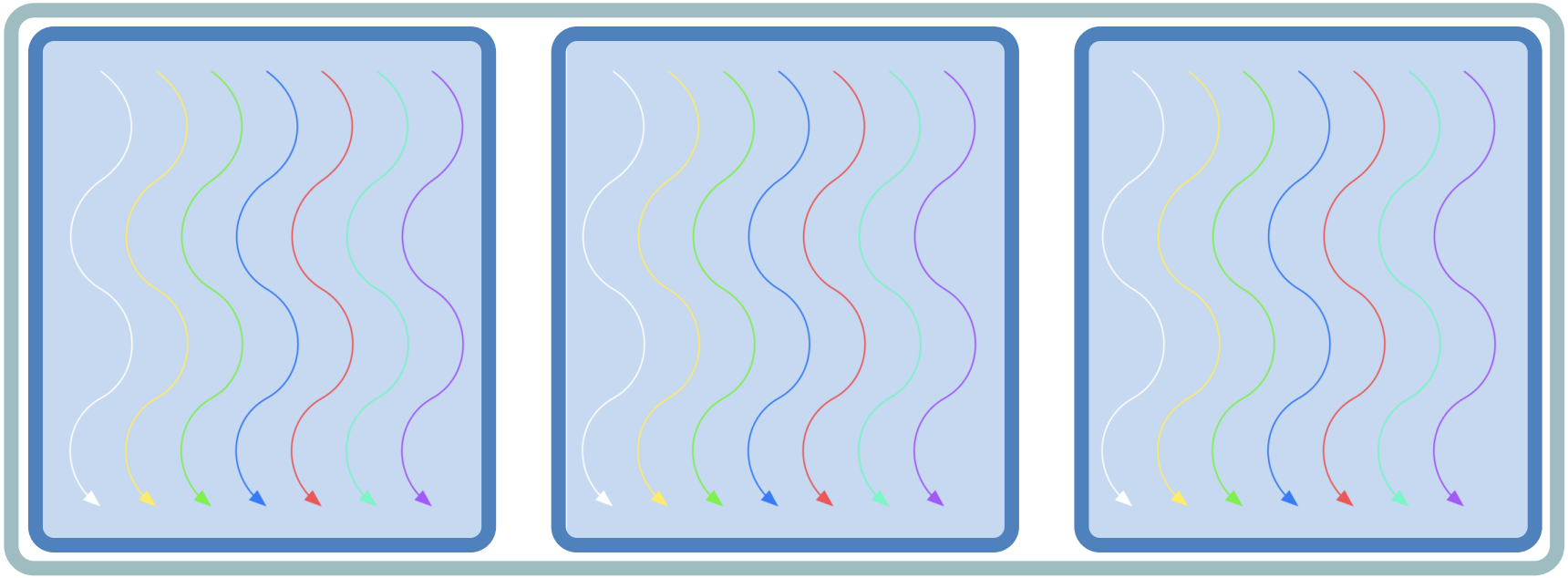
Programming model

- **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



Programming model

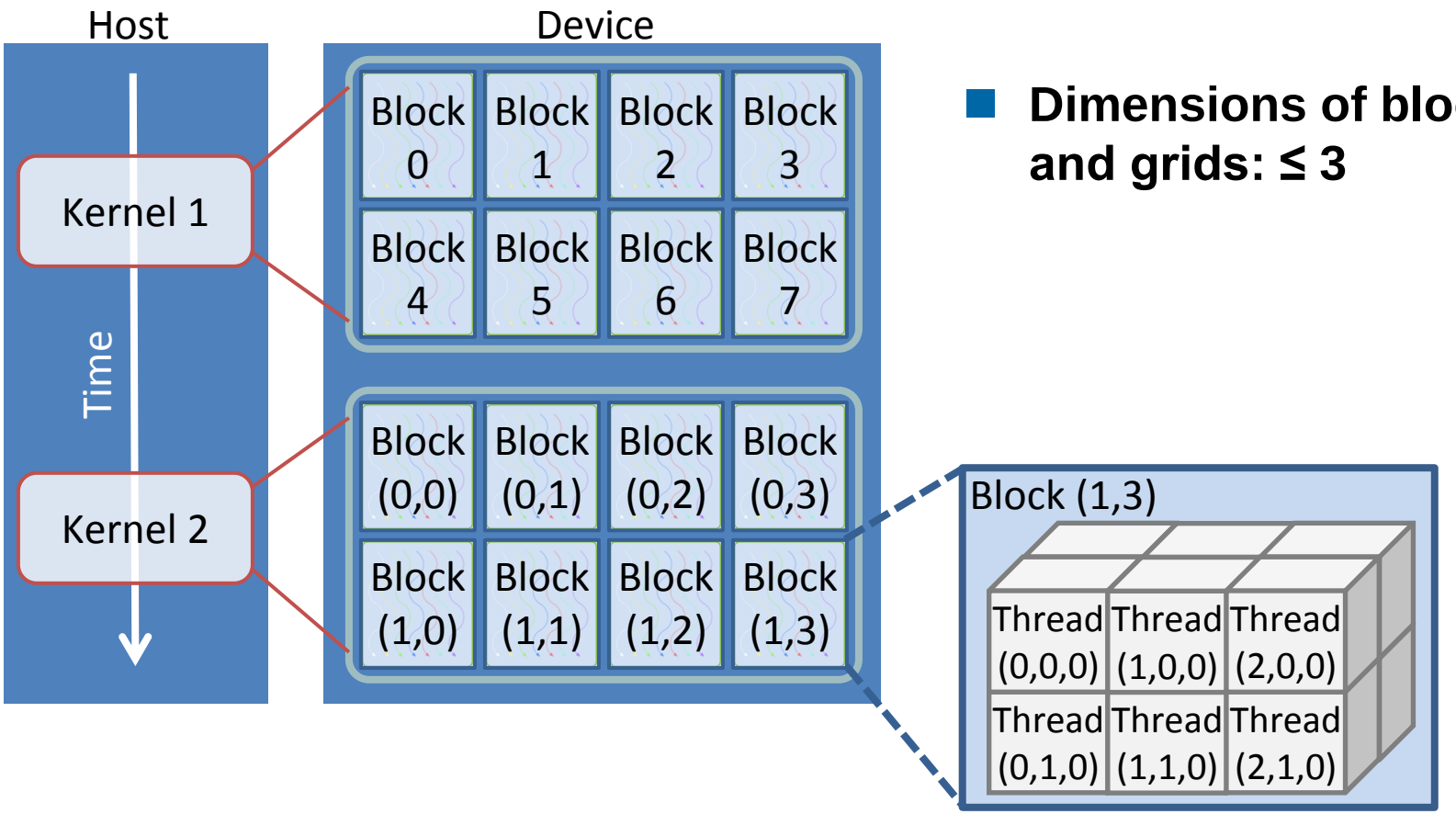


Based on: NVIDIA Corporation 2010

- Threads are grouped into *blocks*
- Blocks are grouped into a *grid*

Programming model

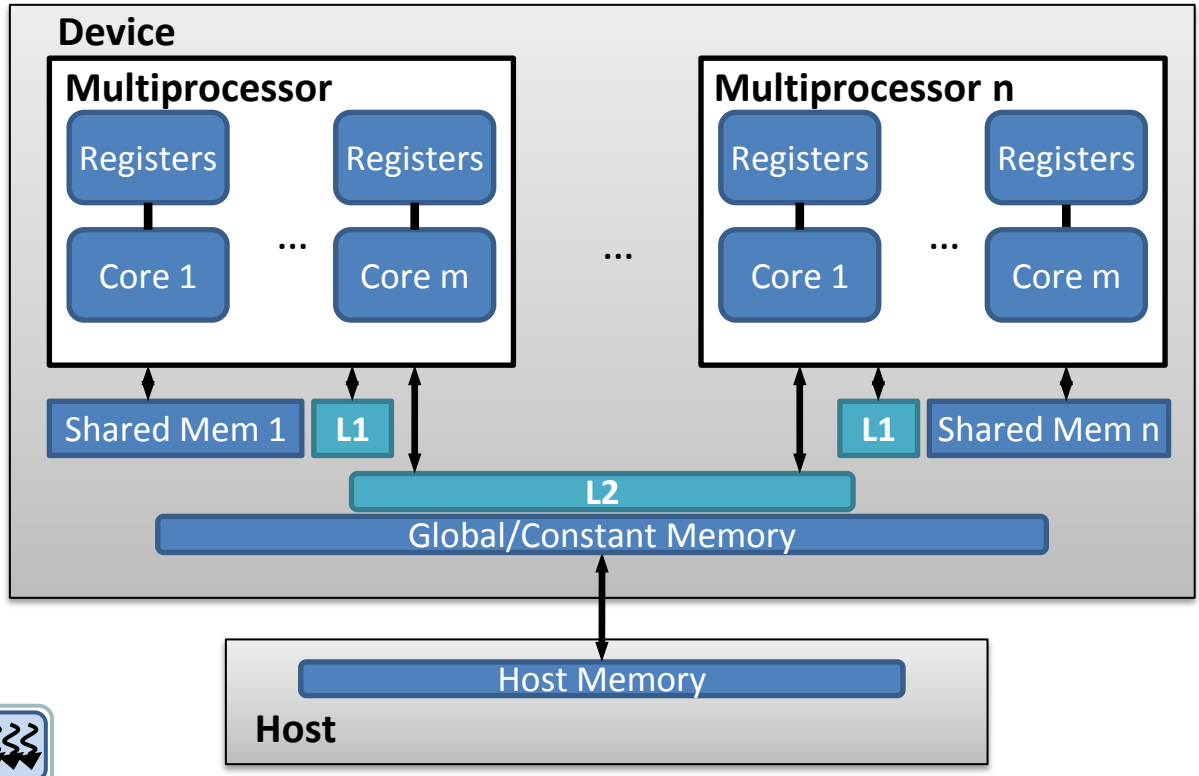
- **Kernel is executed as a grid of blocks of threads**



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

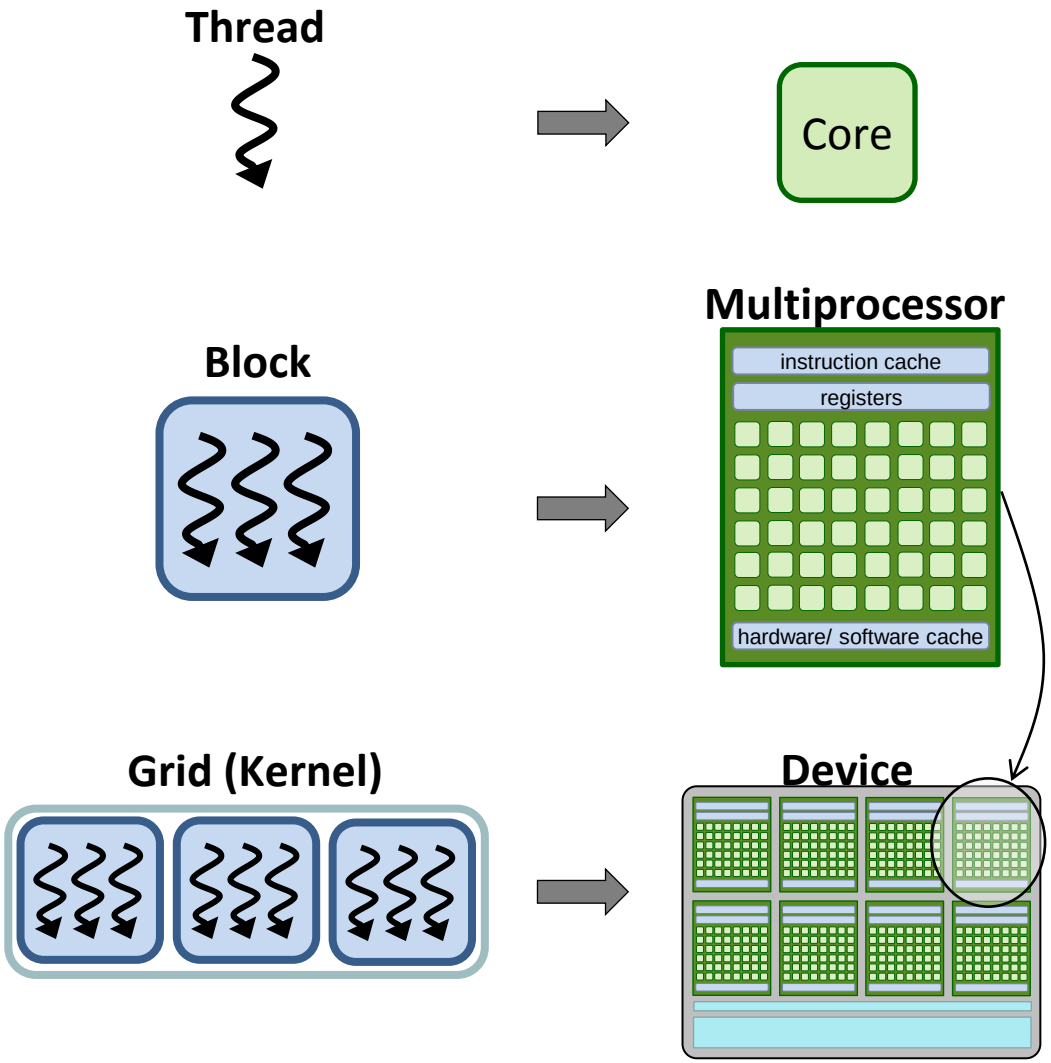
Memory model

- **Thread**
  - → *Registers*
    - **Fermi**: 63 per thread
    - **K20**: 255 per thread
  - → *Local* memory
- **Block**
  - → *Shared* memory
    - **Fermi**:64KB configurable,on-chip
    - 16KB shared +48KB L1 OR
    - 48KB shared +16KB L1
    - **K20**:64KB configurable,on-chip
    - 16KB shared +48KB L1 OR
    - 48KB shared +16KB L1 OR
    - 32KB shared +32KB L1
- **Grid/ application**
  - → *Constant* memory
    - read-only; off-chip; cached
  - → *Global* memory
    - several GB; off-chip
    - **Fermi/K20**: L2 cache



- **Caches**
  - → L1
    - **Fermi**: configurable 16/48KB
    - **K20**: configurable 16/32/ 48 KB
  - → L2
    - **Fermi**: 768KB
    - **K20**: 1536KB

Mapping to Execution Model



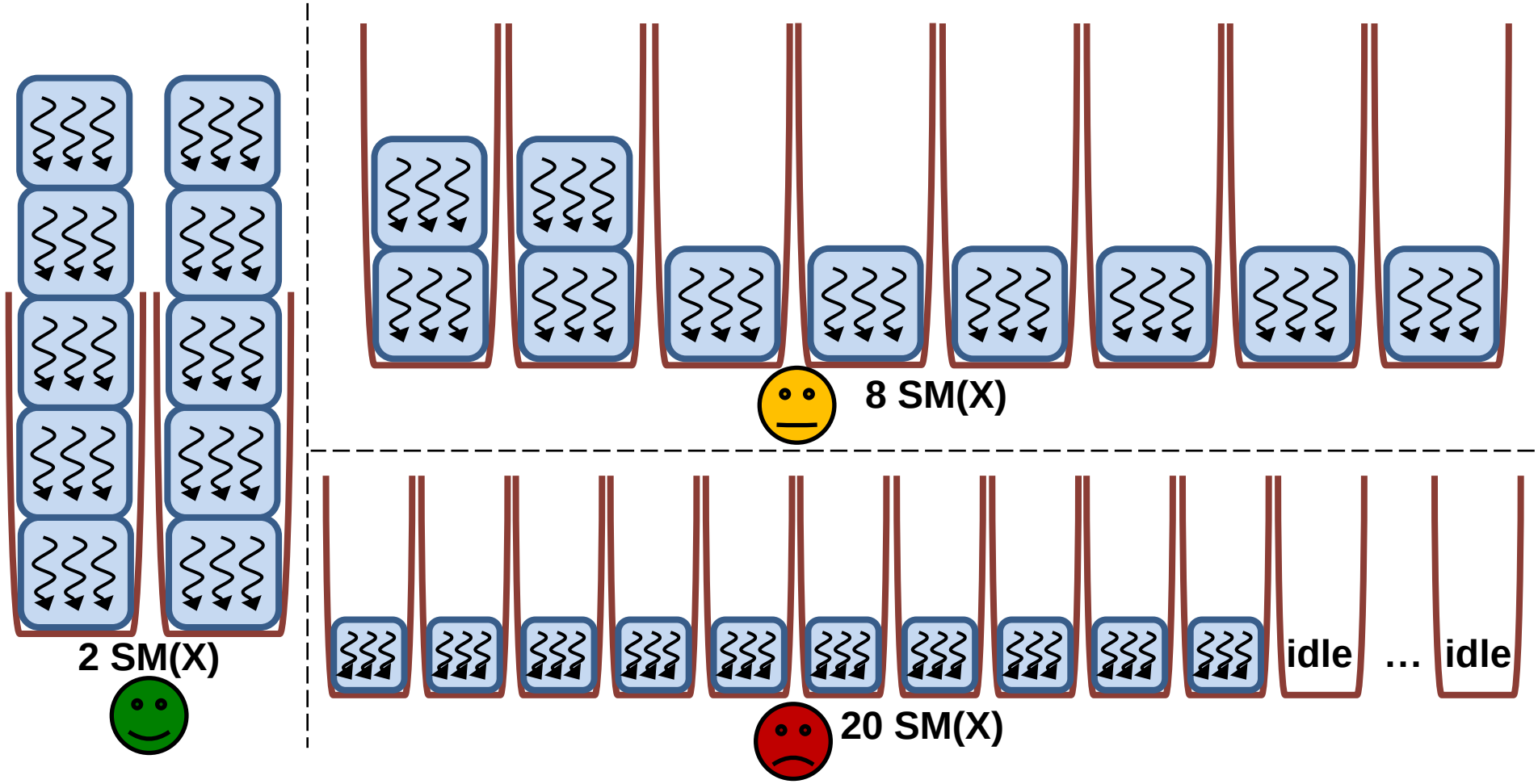
- Each thread is executed by a core
- Each block is executed on a SM(X)
- Several concurrent blocks can reside on a SM(X) depending on shared resources
- Each kernel is executed on a device

Programming model

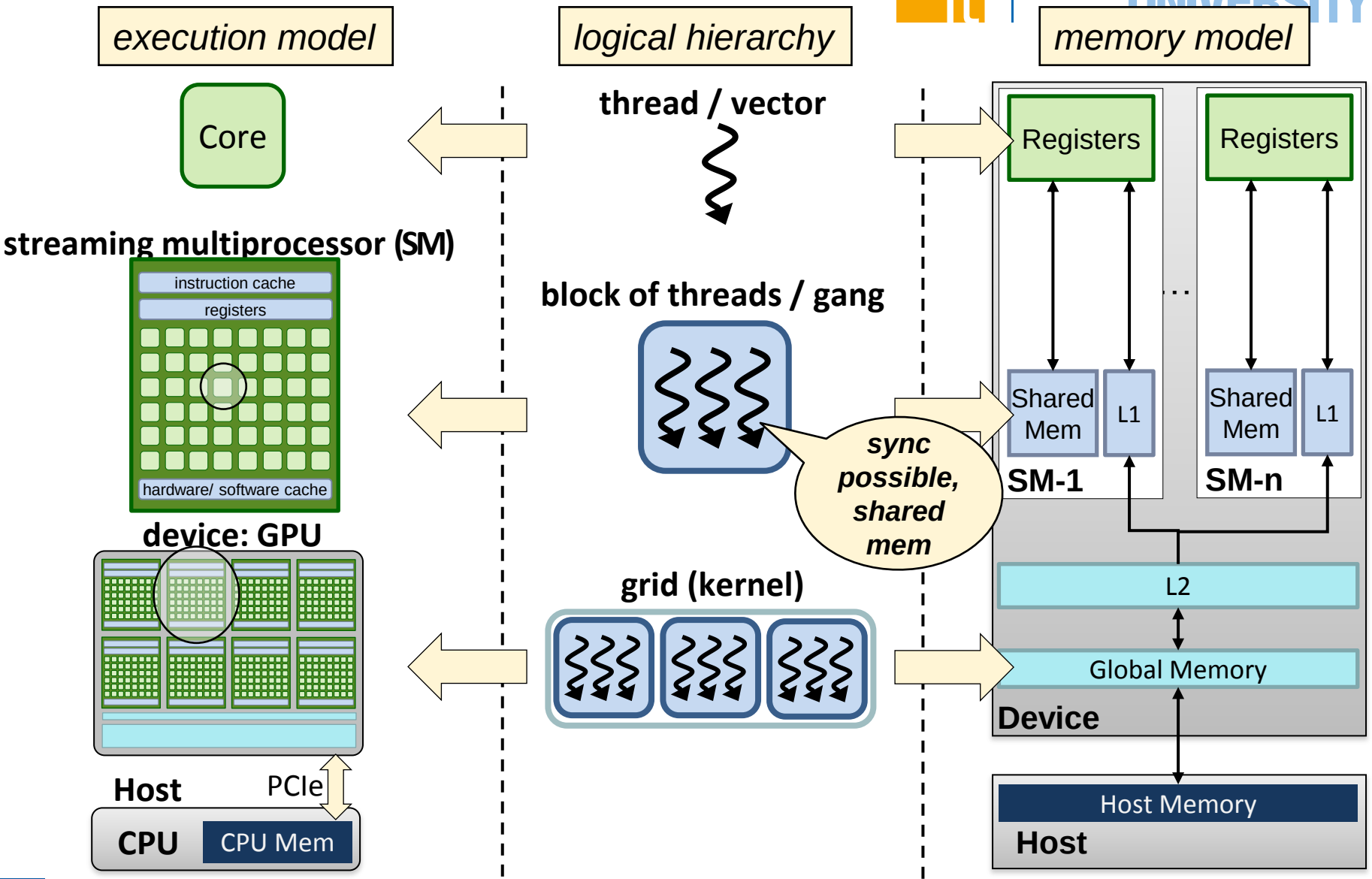
- **Why blocks?**
  - → Cooperation of threads within a block possible
    - →Synchronization
    - →Share data/ results using shared memory
  - → Scalability
    - →Fast communication between n threads is not feasible when n large
      - →No global synchronization on GPU possible (only by completing one kernel and starting another one from the CPU)
    - →But: blocks are executed independently
    - →Blocks can be distributed across arbitrary number of multiprocessors
      - →In any order, concurrently, sequentially

Scalability

- **Assume: 10 thread blocks**



Summary



Vector, worker, gang mapping is compiler dependent.

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **CUDA Basics**
  - → Offloading Regions
  - → Data Management
- **GPUs @ IT Center**
- **CUDA Advanced**
  - → Programming, Memory & Execution Model
  - → Maximize Global Memory Throughput
  - → Launch Configuration

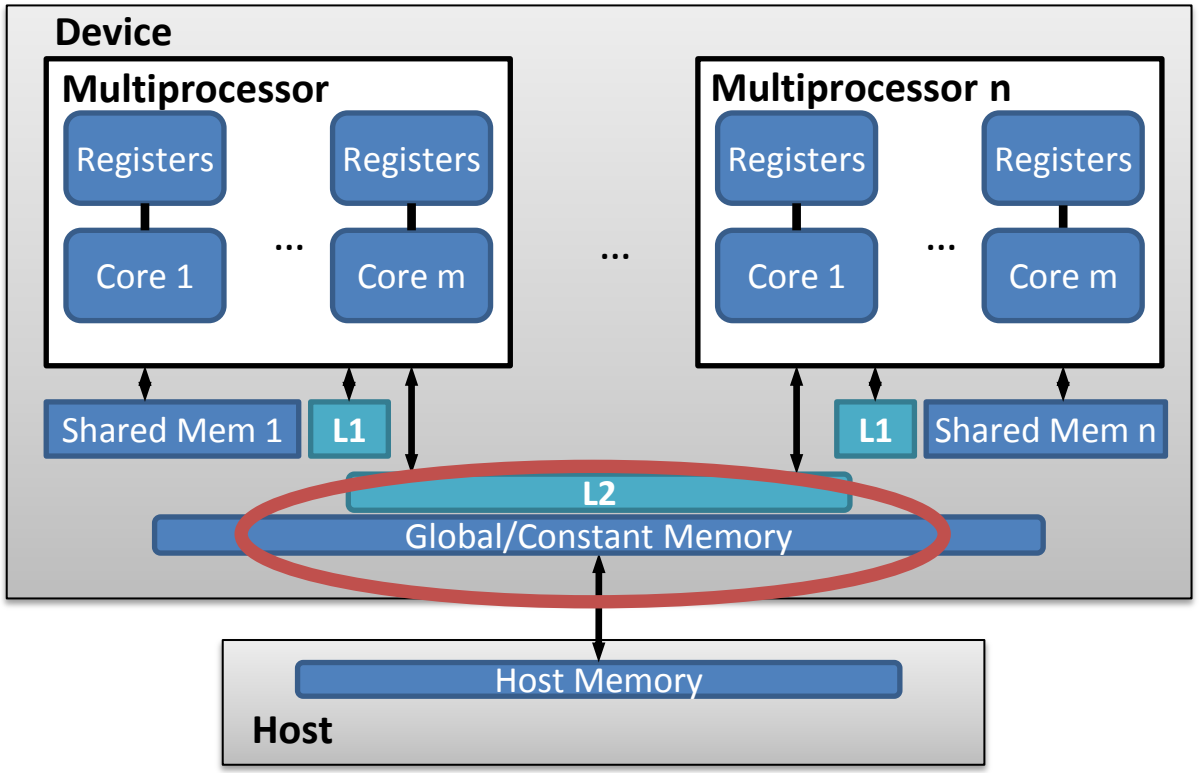
Background information

- **NVIDIA‘s compute capability (cc)**
  - → Describes architecture and features of GPU
  - → E.g. number of registers
  - → E.g. double precision computations (only since cc 1.3)
  - → E.g. management of memory accesses (per 16/32 threads)

- **Examples**
  - → GT200, Tesla C1060: cc 1.3
  - → Fermi C2050, Quadro 6000: cc 2.0
  - → Kepler K20: cc 3.5

Recap memory hierarchy

- **Local memory/ Registers**
- **Shared memory/ L1**
  - → Very low latency (~100x than gmem)
  - → Bandwidth (aggregate): 1+ TB/s (Fermi), 2.5 TB/s (Kepler)
- **L2**
- **Global memory**
  - → High latency (400-800 cycles)
  - → Bandwidth: 144 GB/s (Fermi), 250 GB/s (Kepler)



gmem access

- **Stores:** Invalidate L1, write-back for L2
- **Loads:**

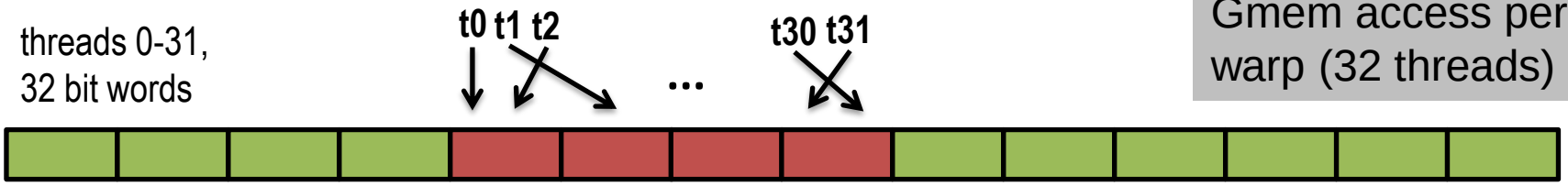
	Caching	Non-caching
Enabling by	default (Fermi)	Experimental compiler-option PGI: `-Mx,180,8`
Attempt to hit:	L1 → L2 → gmem	L2 → gmem (no L1: invalidate line if it's already in L1)
Load granularity	128-byte line	32-byte line

Kepler uses L1 cache only for thread-private data → L2 cache with 128-byte cache line.

→ Threads in a warp provide memory addresses

→ Determine which lines/segments are needed

→ Request the needed lines/segments

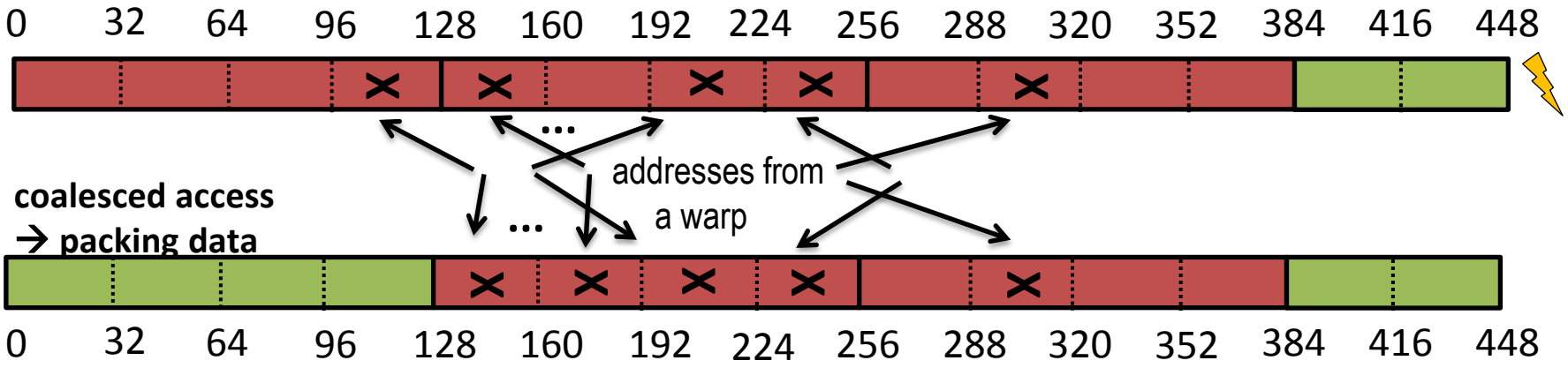


Gmem access per warp (32 threads)

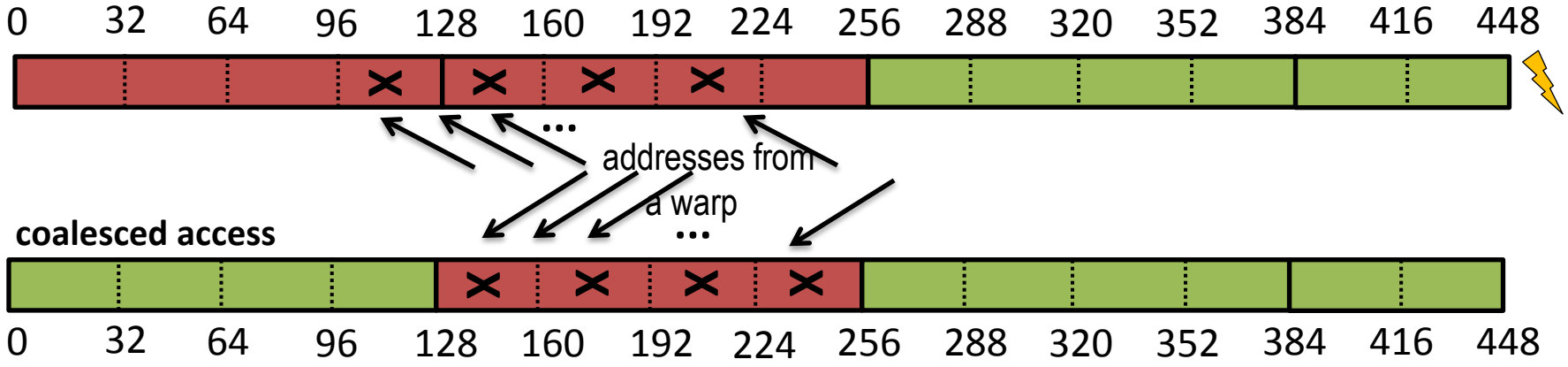
gmem throughput – coalescing

- **Range of accesses**

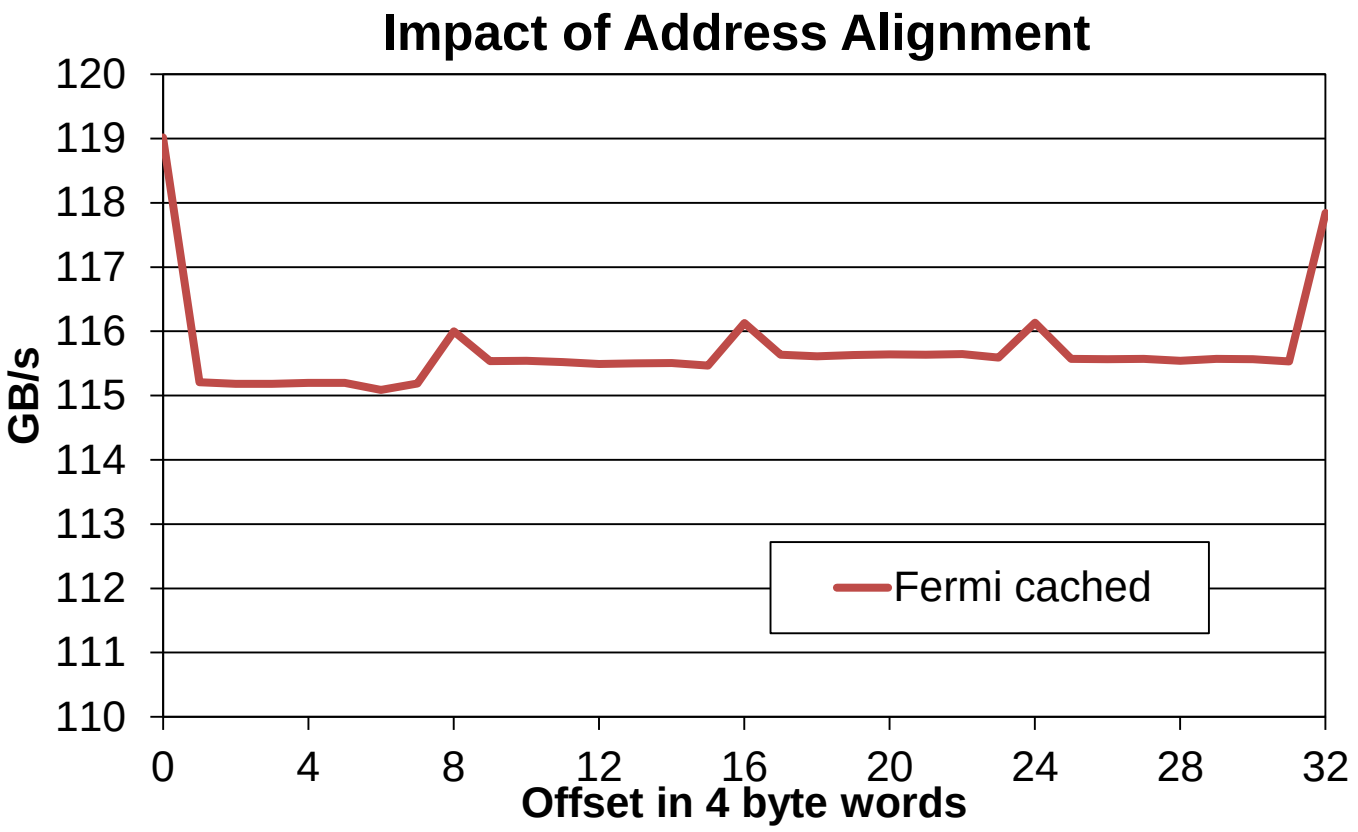
caching loads



- **Address alignment**



Impact of address alignment



Example program:

- Copy ~67 MB of floats
- NVIDIA Tesla C2050 (Fermi)
- ECC off
- 256 threads/block

→ **Misaligned accesses can drop memory throughput**

Data layout matters (improve coalescing)

- **Example: AoS vs. SoA**

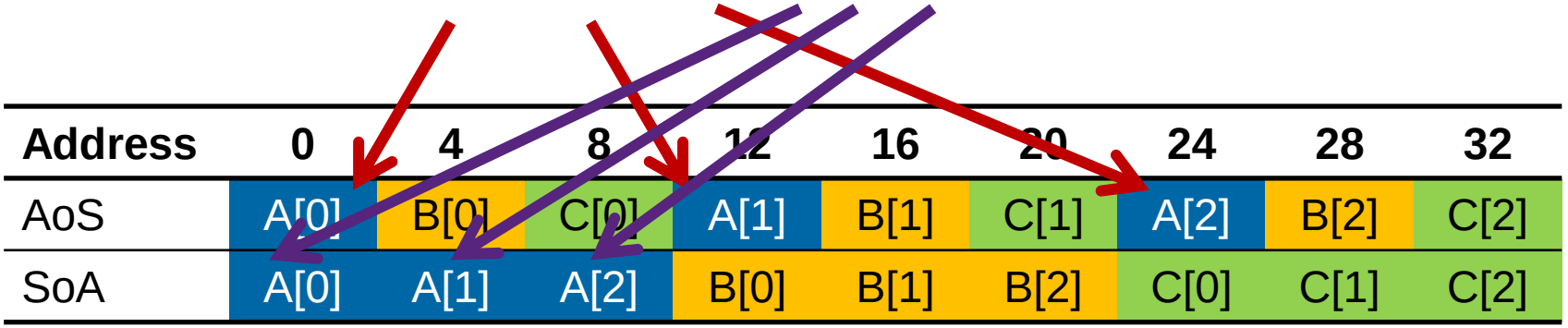
Array of Structures (AoS)

```
struct myStruct_t {
  float a;
  float b;
  int c;
}
myStruct_t myData[];
```

Structure of Arrays (SoA)

```
struct {
  float a[];
  float b[];
  int c[];
}
```

```
// CUDA kernel:
int i = blockIdx.x * blockDim.x + threadIdx.x;
… myData[i].a / myData.a[i] …
```



This slide is based on: NVIDIA Corporation

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **CUDA Basics**
  - → Offloading Regions
  - → Data Management
- **GPUs @ IT Center**
- **CUDA Advanced**
  - → Programming, Memory & Execution Model
  - → Maximize Global Memory Throughput
  - → Launch Configuration

Launch configuration

- **How many threads/ thread blocks to launch?**

```
myKernel<<<numBlocks, numThreads>>>(…)
```

- **Hardware operation**
  - → Instructions are issued in order
  - → Thread execution stalls when one of the operands isn't ready
  - → Latency is hidden by switching threads
    - →GMEM latency: 400-800 cycles
    - →Arithmetic latency: 18-22 cycles

→ **Need enough threads to hide latency**

Launch configuration

- **Hiding arithmetic latency**
  - → Need ~18 warps (576 threads) per Fermi MP
  - → Or, independent instructions from the same warp

```
x = a + b;//~18 cycles
y = a + c;//independent
          //can start any time
// stall
z = x + d;//dependent
          //must wait for compl.
```

Source: Volkov2010

- **Maximizing global memory throughput**
  - → Gmem throughput depends on the access pattern, and word size
  - → Need enough memory transactions in flight to saturate the bus
    - → Independent loads & stores from the same thread (mult. elements)
    - → Loads and stores from different threads (many threads)
    - → Larger word sizes can also help (“vectors”)

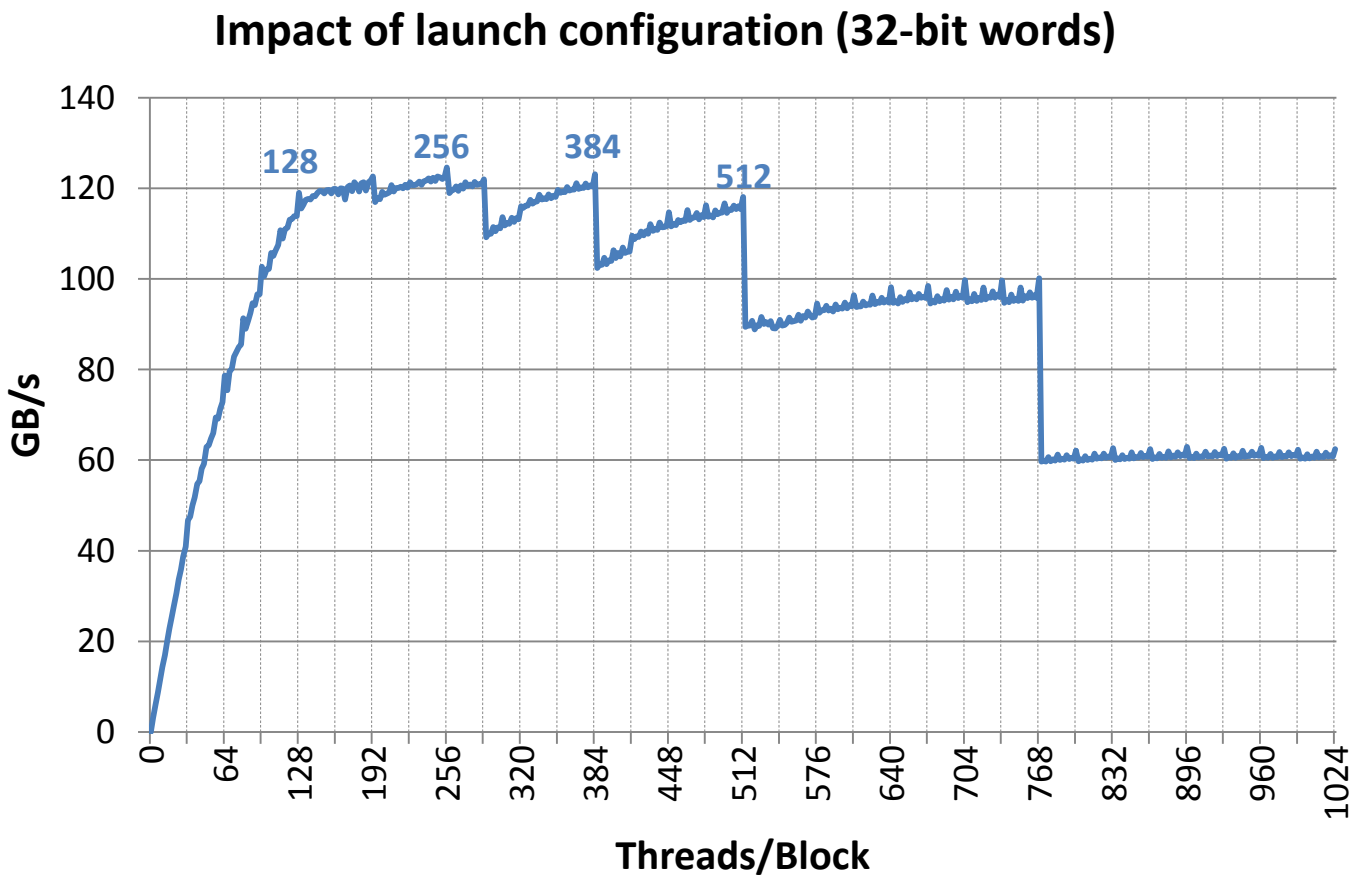
e.g. →

```
float a0= src[id];
// no latency stall
float a1= src[id+blockDim.x];
// stall
dst[id]= a0;
dst[id+blockDim.x]= a1;
// Note, threads don't stall
// on memory access
```

Source: Volkov2010

Launch configuration

- **Maximizing global memory throughput**
  - → Example program: increment an array of ~67M elements

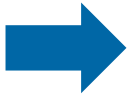


- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

Launch configuration

- **Threads per block: Multiple of warp size (32)**
  - → Threads are always spanned in warps onto resources
  - → Not used threads are marked as inactive
  - → Starting point: 128-256 threads/block
- **Blocks per grid heuristics**
  - → **`#blocks > #MPs`**
    - → MPs have at least one block to execute
  - → **`#blocks / #MPs > 2`**
    - → MP can concurrently execute up to 8 blocks
    - → Blocks that aren't waiting in barrier keep hardware busy
    - → Subject to resource availability (registers, smem)
  - → **`#blocks > 50`** to scale to future devices
  - → Most obvious: **`#blocks * #threads = #problem-size`**
    - → Multiple elements per thread may amortize setup costs of simple kernels:

      `#blocks * #threads < #problem-size`



Launch MANY threads to keep GPU busy!

Summary

- **Strive for perfect coalescing**
  - → Warp should access within contiguous region
  - → Align starting address (may require padding)

- **Have enough concurrent accesses to saturate the bus**
  - → Process several elements per thread
  - → Launch enough threads to cover access latency

Tuning

- **Avoid branch divergence in kernel**
- **Kernel launch configuration**
  - → Launch enough threads per MP to hide latency
  - → Launch enough thread blocks to load the GPU
- **Maximize global memory throughput**
  - → GPU has lots of bandwidth, use it effectively
  - → Coalescing, alignment, (non-)caching loads
- **Use shared memory when applicable** (over 1 TB/s bandwidth)
- **Think about data locality** (PCIe could be a bottleneck)
- **Minimize CPU/GPU idling by asynchronous operations**
- **Use analysis/ profiling when optimizing**