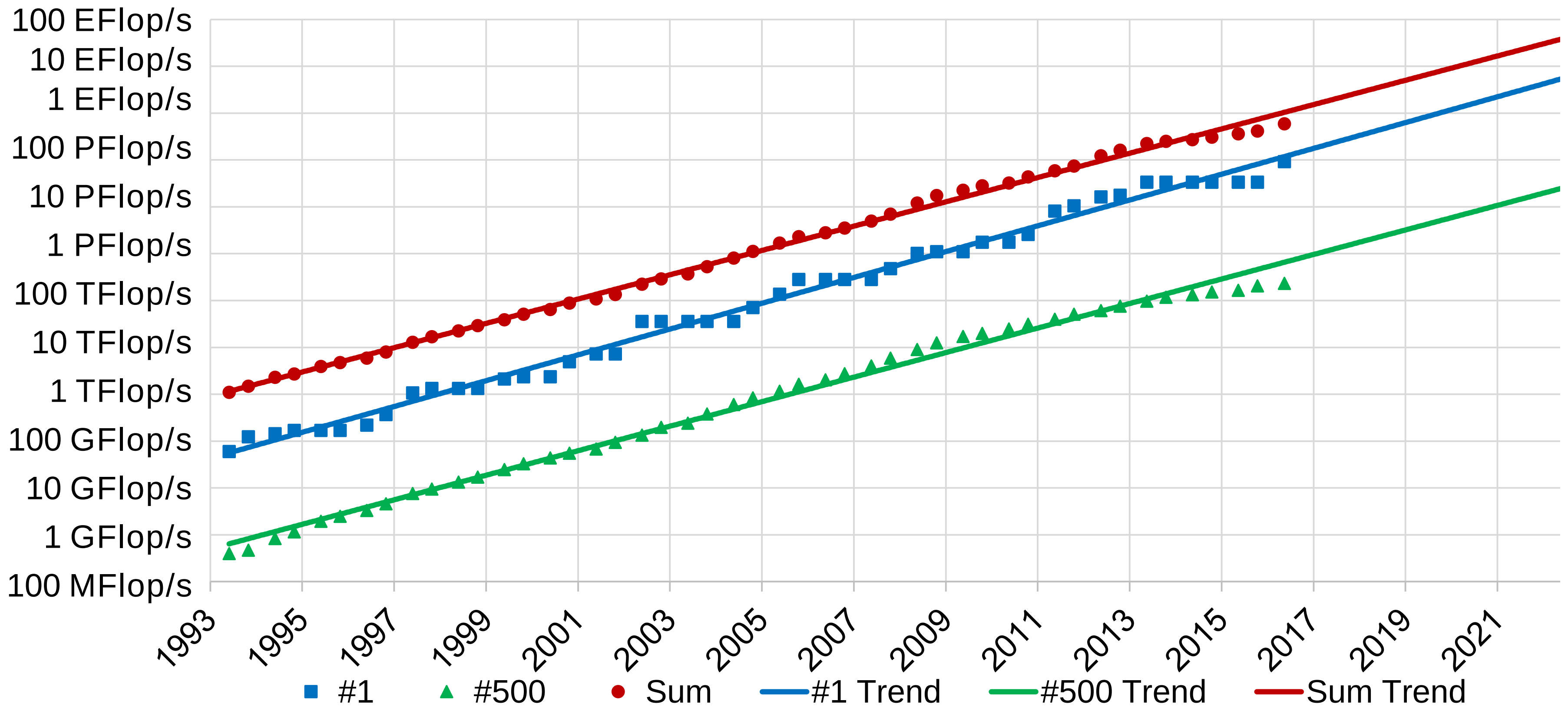
Outline

Motivation

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



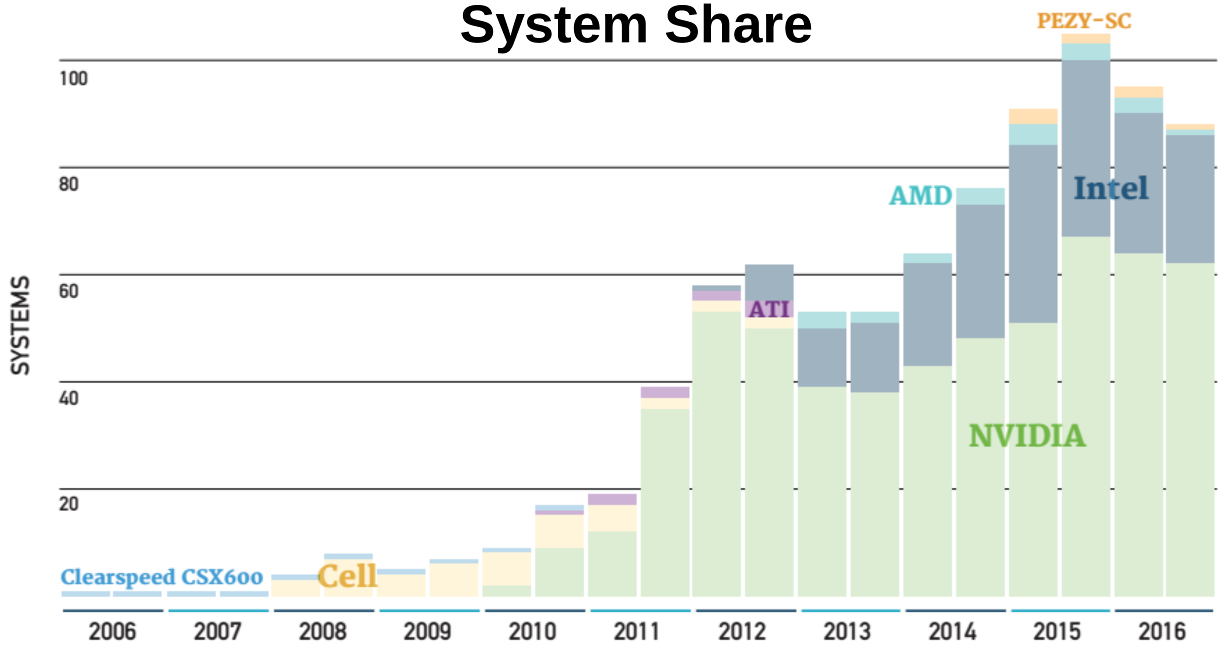
Source:Top500 06/2016

Motivation

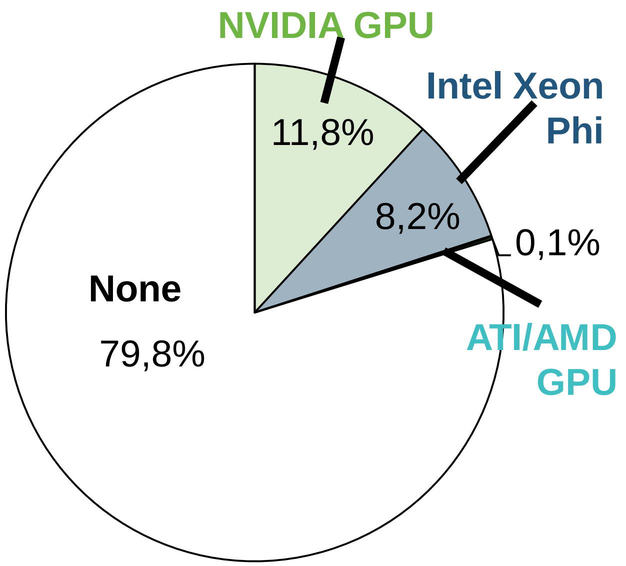
Today, NVIDIA GPUs are the focus.

- **Accelerators/ co-processors**
  - → GPGPUs (e.g. NVIDIA, AMD)
  - → Intel Many Integrated Core (MIC) Arch. (Intel Xeon Phi)
  - → FPGAs (e.g. Convey), …
- **Heterogeneous Arch.**
  - → Combination of commodity processors & accelerators

System Share



Performance Share



Source: Top500, 11/2016

Motivation – Green500 (11/2016)

Rank	MFLOPS/W	Site*	Proc/ Accelerator	Total Power (kW)
1	9462.1	NVIDIA Corporation	NVIDIA Tesla P100	349.5
2	7453.5	Swiss National Supercomputing Centre (CSCS)	NVIDIA Tesla P100	1312
3	6673.8	Advanced Center for Computing and Communication, RIKEN	PEZY-SCnp	150.0
4	6051.3	National Supercomputing Center in Wuxi	[heterogeneous]	15371
5	5806.3	Fujitsu Technology Solutions GmbH	Intel Xeon Phi 7210	77
6	4985.7	Joint Center for Advanced High Performance Computing	Intel Xeon Phi 7250	2718.7
7	4688.0	DOE/SC/Argonne National Laboratory	Intel Xeon Phi 7230	1087
8	4112.1	Stanford Research Computing Center	Nvidia K80	190
9	4086.8	Academic Center for Computing and Media Studies (ACCMS), Kyoto University	Intel Xeon Phi 7250	748.1
10	3836.6	Thomas Jefferson National Accelerator Facility	Intel Xeon Phi 7230	111
33	2903.6	Scientific research institution	–	162

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

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

→ 985 MFLOPS/W

performance per watt

Source: Green500, 11/2016
* Performance data obtained from publicly available sources including TOP500

RWTH Environment

- **GPU cluster**
  - → 28 nodes with each

    2 NVIDIA Quadro 6000 GPUs (Fermi)

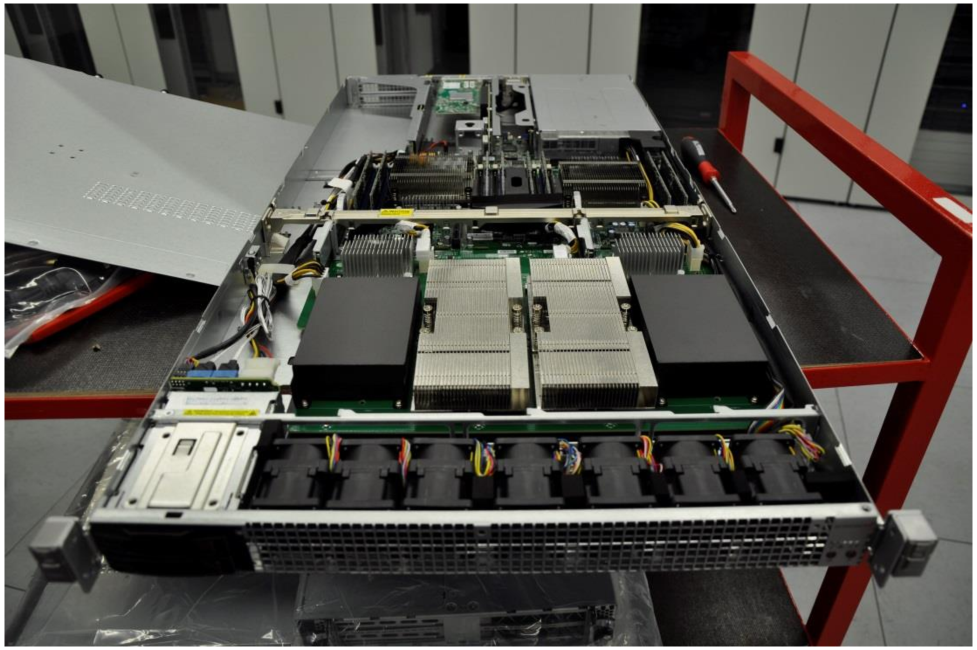
    → VR + HPC
  - → 2 nodes with each

    2 NVIDIA K20X GPUs (Kepler)
  - → 10 nodes with each

    2 NVIDIA P100 SXM2 (Pascal)

    with 16GB

aixCAVE, VR, RWTH Aachen, since June 2012



Outline

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

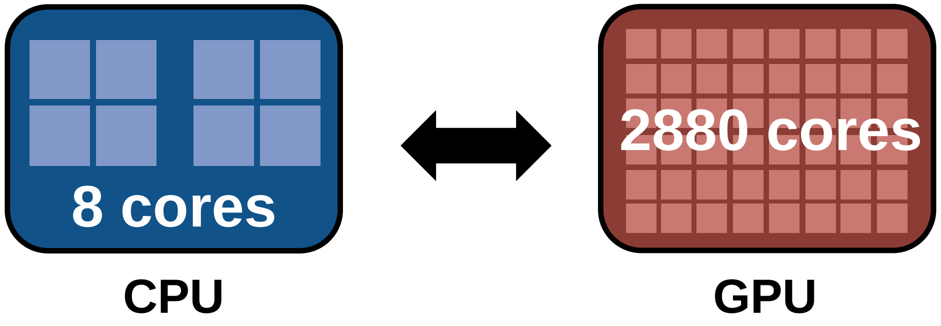
    Graphics APIs like OpenGL, DirectX popular
  - → Since 2001: Programmable pixel and vertex shader in graphics pipeline (adjustments in OpenGL, DirectX)

    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"

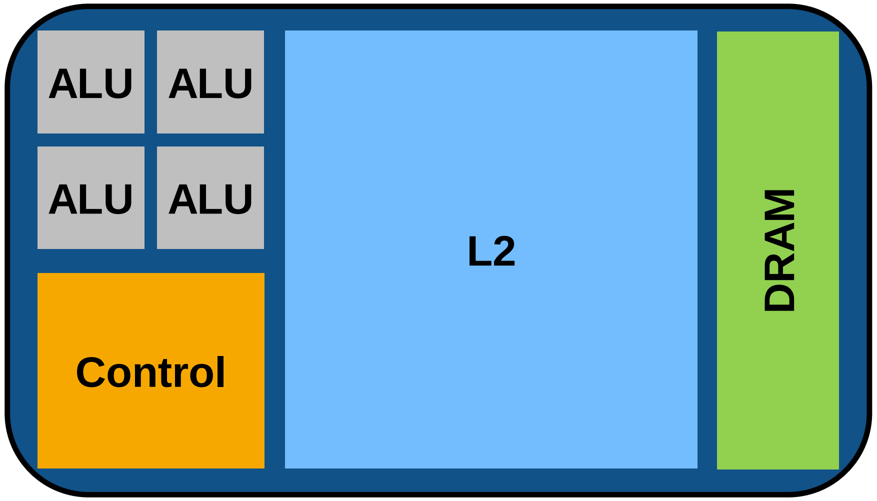
Comparison CPU ⇔ GPU

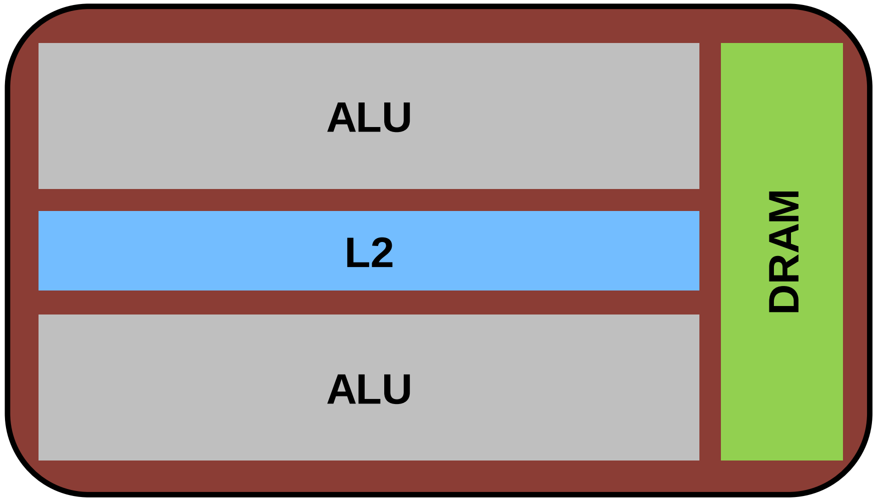


- **GPU-Threads**
  - → Thousands (“few” on CPU)
  - → Light-weight, little creation overhead
  - → Fast switching
- **Massively Parallel Processors**
- **Manycore Architecture**

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

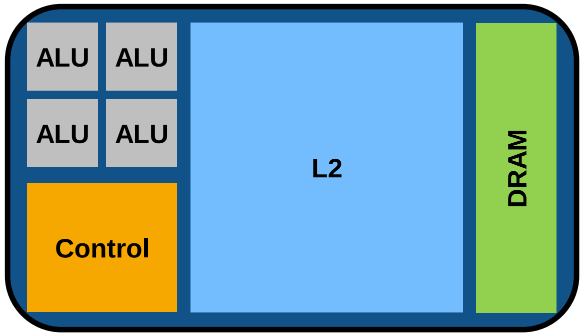
Why can accelerators deliver good performance watt ratio?

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

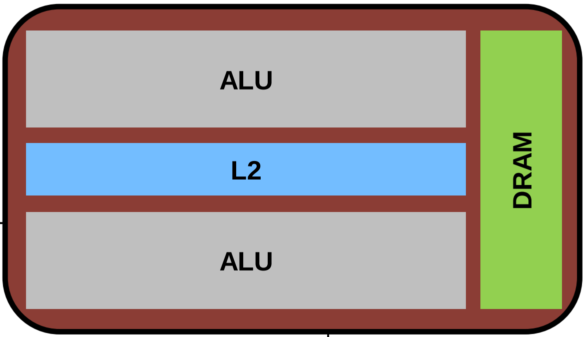
$$P \sim V^2 \cdot f$$

→ No control logic

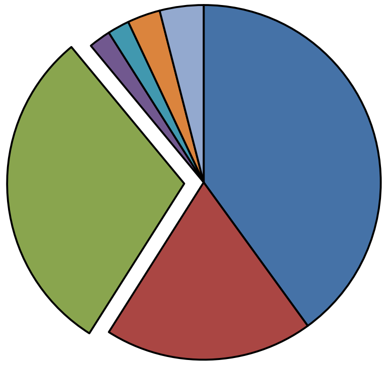
CPU



GPU



Power use for 1 TFlop/s of an usual system



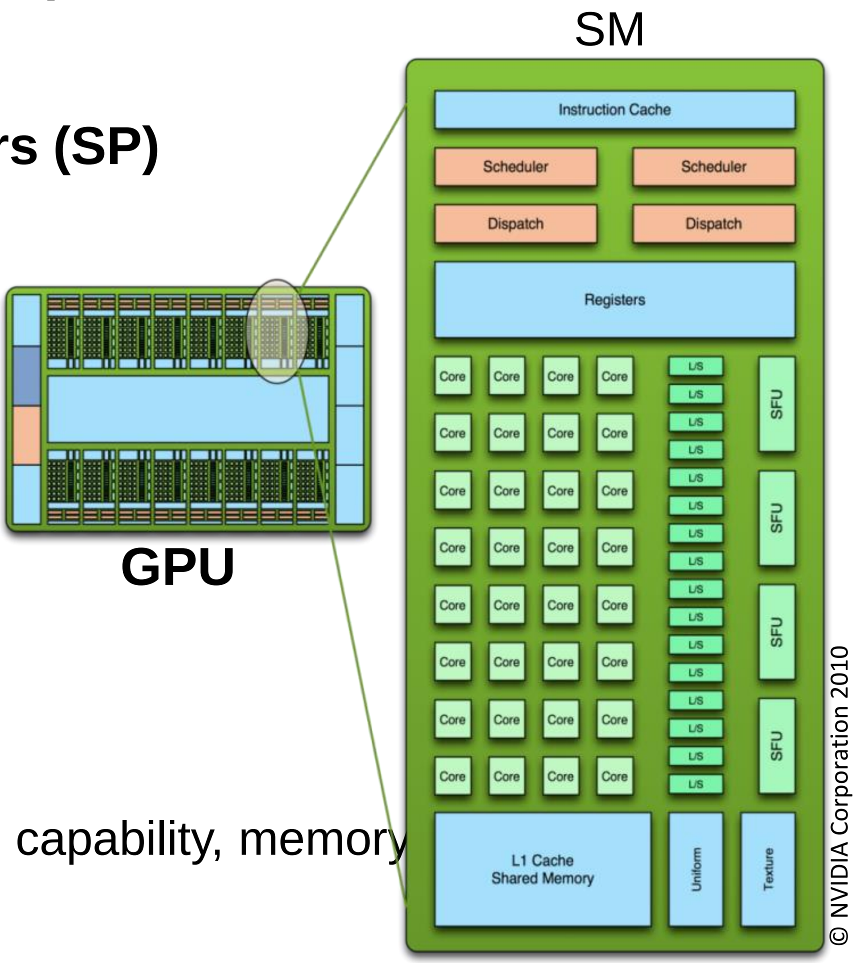
- 🟦 Heat removal
- 🟥 Power supply loss
- 🟩 Control
- 🟪 Disk
- 🟦 Communication
- 🟧 Memory
- 🟦 Compute

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

Outline

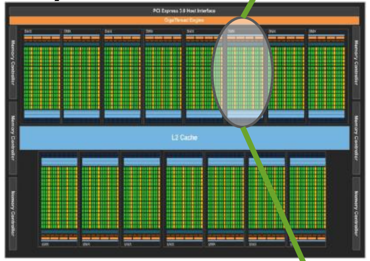
GPU architecture: Fermi

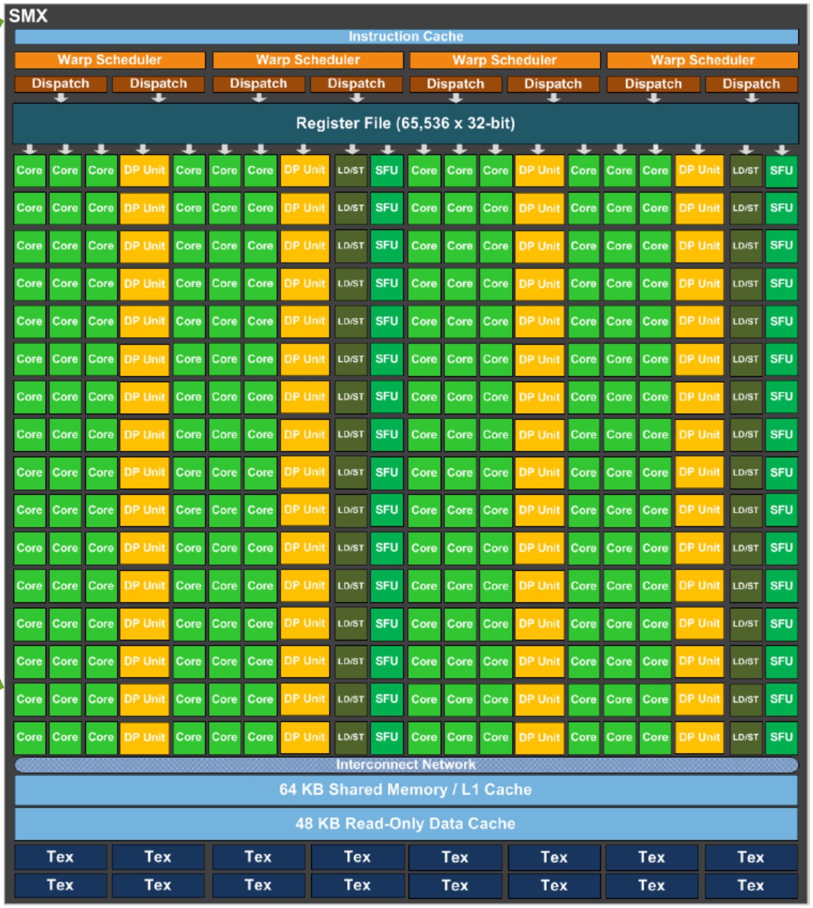
- 3 billion transistors
- 14-16 streaming multiprocessors (SM)
  - → Each comprises 32 (SP) cores
- 448-512 cores/ streaming processors (SP)
  - → i.a. Floating point & integer unit
- Memory hierarchy
- Peak performance
  - → SP: 1.03 TFlops
  - → DP: 515 GFlops
- ECC support
- Compute capability: 2.0
  - → Defines features, e.g. double precision capability, memory



GPU architecture: Kepler

- 7.1 billion transistors
- 13-15 streaming multiprocessors extreme (SMX)
  - → Each comprises 192 (SP) 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

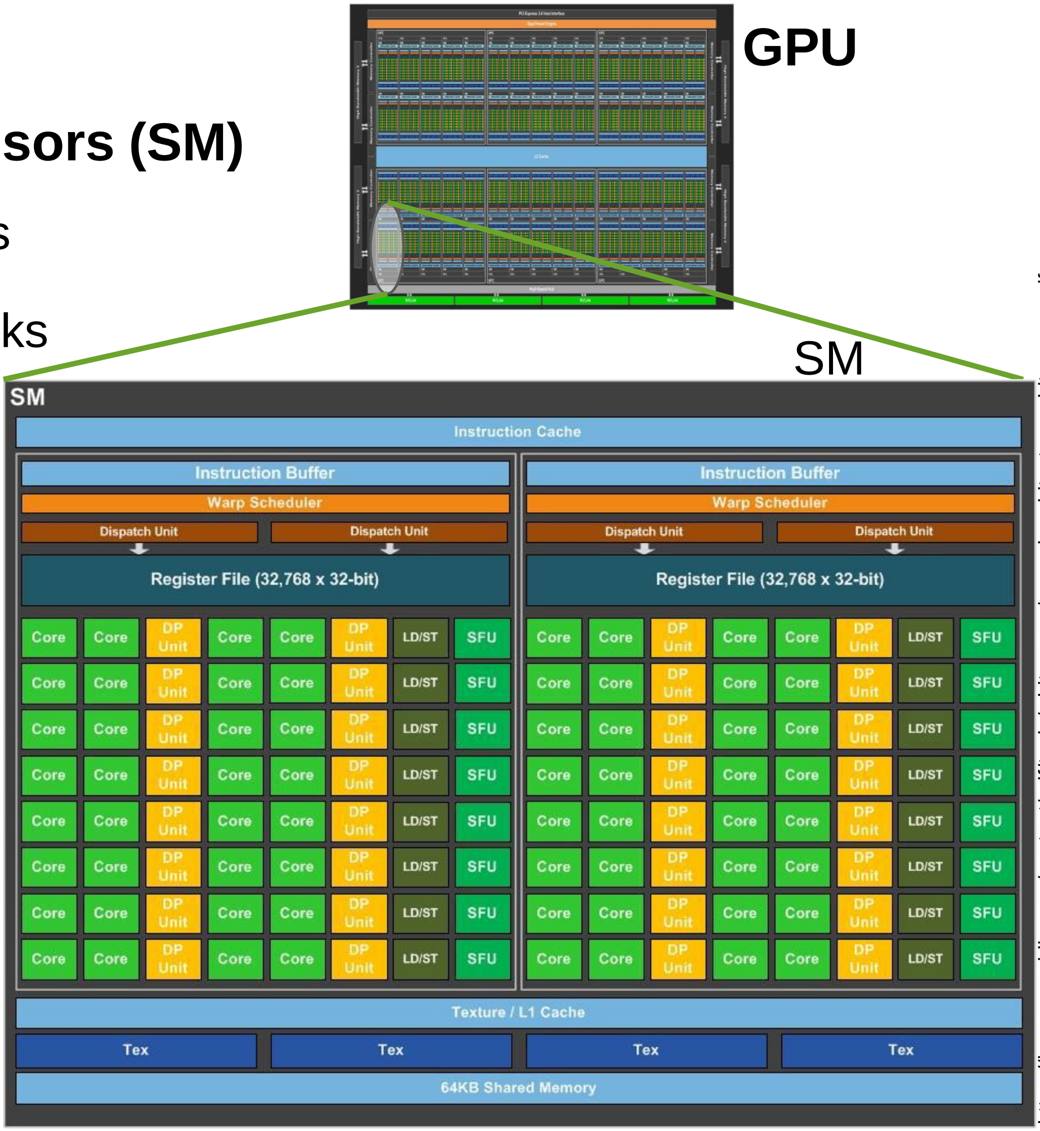

GPU

SMX

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

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

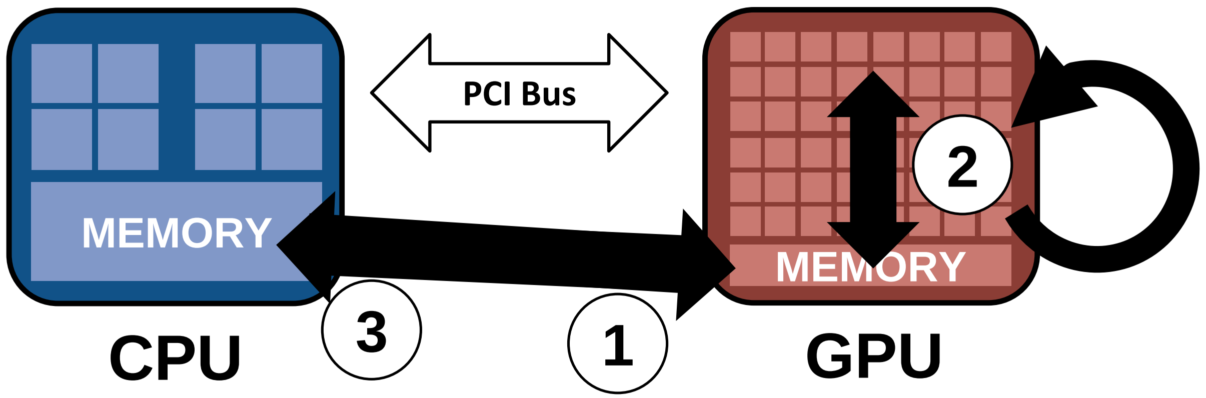
GPU architecture: Pascal

- **15.3 billion transistors**
- **56-60 streaming multiprocessors (SM)**
  - → Each comprises 64 (SP) cores
  - → Divided into 2 processing blocks
- **3584-3840 cores**
- **Memory hierarchy**
- **Peak performance (P100)**
  - → SP: 9.52 Tflops
  - → DP: 4.76 Tflops
- **ECC support, NVLink**
- **Compute capability: 6.0**
  - → E.g. atomicAdd() on 64-bit float



https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

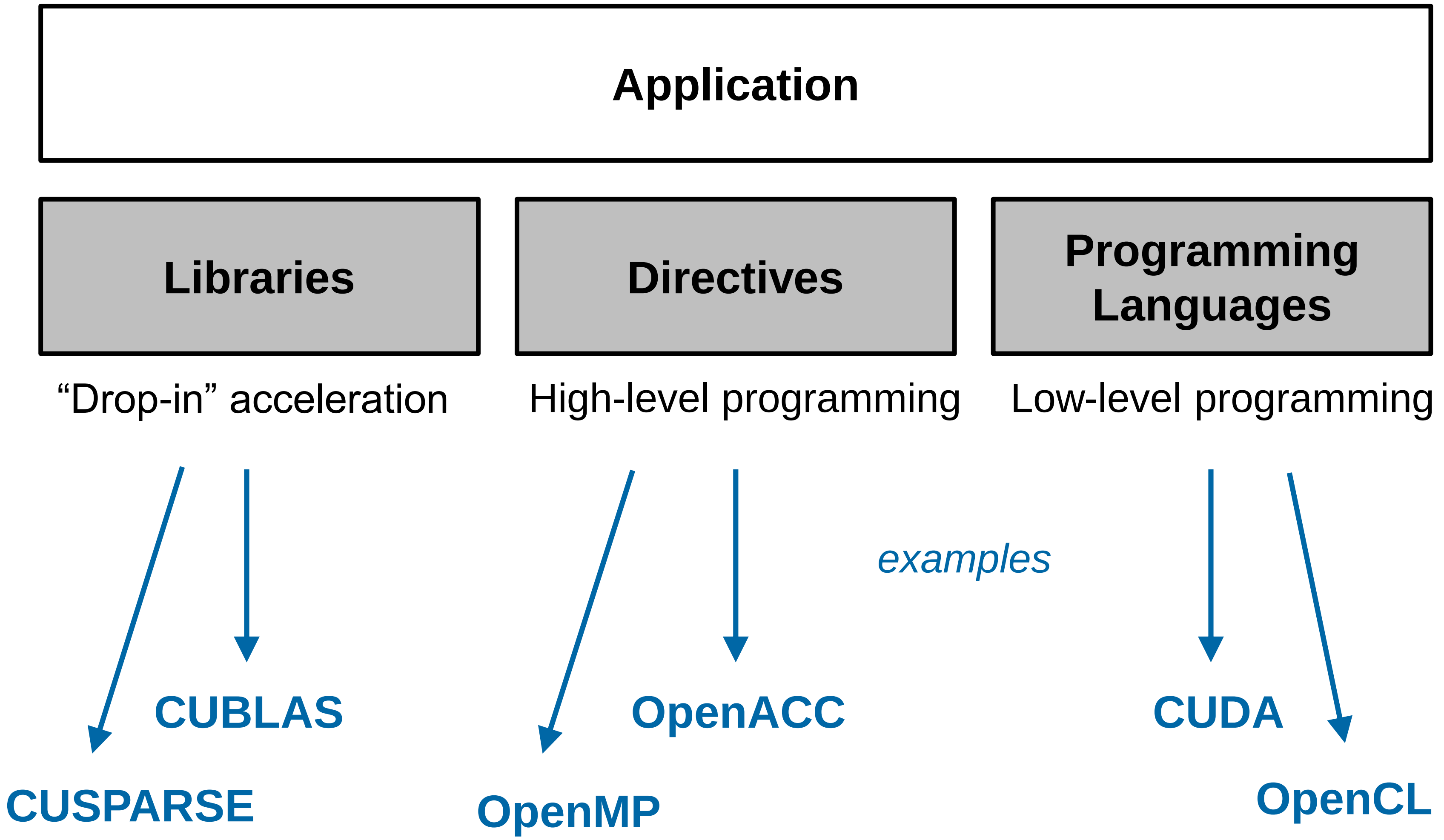
Comparison CPU ⇔ GPU



- **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.

Outline

GPGPU programming



GPGPU programming

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

Overview *CUDA*

= **Compute Unified Device Architecture**

- **CUDA C/C++ from NVIDIA**
  - → Based on industry standard C/C++ (extensions & restrictions)
  - → Driver API (low level), Runtime API (higher level)
- **Brief timeline**
  - → Nov’06: Introduction of CUDA, G80 GPU architecture
  - → Jun’07: CUDA Toolkit 1.0
  - → Jun’08: GT200 GPU architecture
  - → March’10: Fermi GPU architecture
  - → Nov’12: Kepler K20(X) GPU architecture
  - → July’16: Pascal GPU architecture

CUDA Fortran available by PGI.

CUDA Toolkit
Developer toolkit

Already installed in RWTH Cluster environment:

```
module load cuda
```

CUDA - Compiling

- **Download + install CUDA Toolkit**

  (cf. “Links” section)

- **Compiling**

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

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

  → `-arch=sm_20`: Set compute capability 2.0

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

Low-level languages vs. directives

- **Nowadays: GPU APIs (like CUDA, OpenCL) often used**
  - → May be difficult to program (as/but more flexibility)
  - → Verbose/ may complicate software design
- **Directive-based programming model delegates responsibility for low-level GPU programming tasks to compiler**
  - → Data movement
  - → Kernel execution
  - → "Awareness" of particular GPU type
  - → …

$\left.\begin{array}{l}\text{Data movement}\\\text{Kernel execution}\\\text{“Awareness” of particular GPU type}\end{array}\right\}$

- → Many tasks can be done by compiler/ runtime
- → User-directed programming

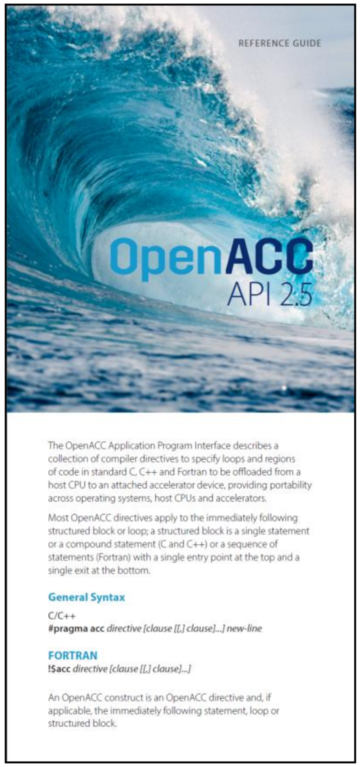
→ **OpenACC or OpenMP 4.x device constructs**

Overview *OpenACC*

Here, PGI's OpenACC compiler is used for examples. Details are compiler dependent.

- **Open industry standard**
  - → Portability
- **Introduced by CAPS, Cray, NVIDIA, PGI (Nov. 2011)**
- **Support**
  - → C,C++ and Fortran
  - → NVIDIA GPUs, AMD GPUs, x86 Multicore, Intel MIC (now/near future)
- **Timeline**

→ Nov'11:	Specification 1.0
→ Jun'13:	Specification 2.0
→ Nov'14:	Proposal for complex data management
→ Nov'15:	Specification 2.5



The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator device, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C and C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

General Syntax

C/C++
#pragma acc *directive [clause [[,] clause]...] new-line*

FORTRAN
!$acc *directive [clause [[,] clause]...]*

An OpenACC construct is an OpenACC directive and, if applicable, the immediately following statement, loop or structured block.

Source: www.openacc.org

OpenACC - Directives

- **Syntax**

C

```
#pragma acc directive-name [clauses]
```

Fortran

```
!$acc directive-name [clauses]
```

- **Iterative development process**

➔ **Compiler feedback helpful**

→ Whether an accelerator kernel could be generated

→ Which loop schedule is used

→ Where/which data is copied

OpenACC - Compiling (PGI)

```
pgcc -acc -ta=nvidia,cc20,7.0 -Minfo=accel saxpy.c
```

→ `pgcc`	C PGI compiler (`pgf90` for Fortran)
→ `-acc`	Tells compiler to recognize OpenACC directives
→ `-ta=nvidia`	Specifies the target architecture → here: NVIDIA GPUs
→ `cc20`	Optional. Specifies target compute capability 2.0
→ `7.0`	Optional. Uses CUDA Toolkit 7.0 for code generation
→ `-Minfo=accel`	Optional. Compiler feedback for accelerator code

The PGI tool `pgaccelinfo` prints the minimal needed compiler flags for the usage of OpenACC on the current hardware (see *Develop-ment Tips*).

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 kernels loop gang vector(128)
    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

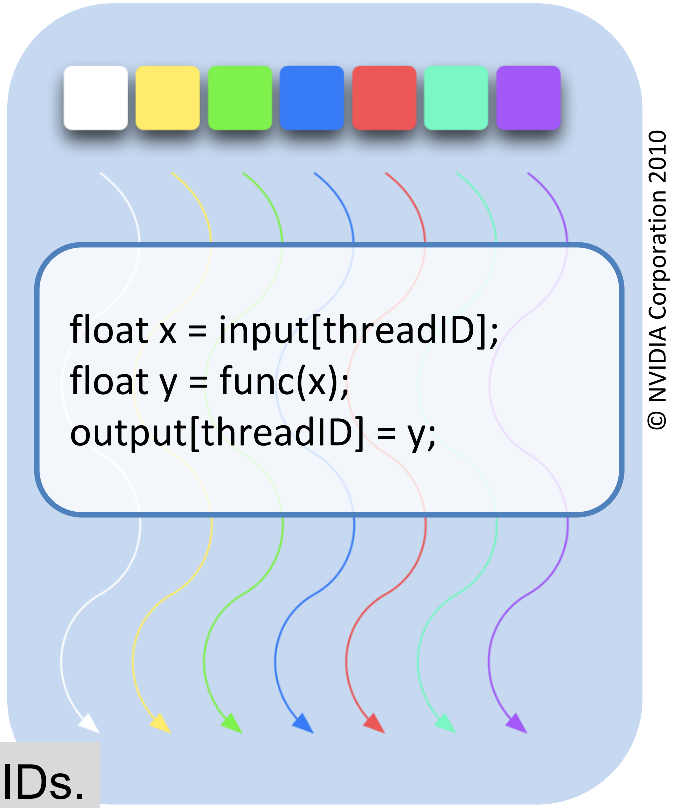
```
__global__ void saxpyCUDA(int n, float a,
   float *x, float *y) {
    int i = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (i < n){
        y[i] = a*x[i] + y[i];
    }
}

int main(int argc, char* argv[]) {
 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);
saxpyCUDA<<<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;
}
```

Programming model

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

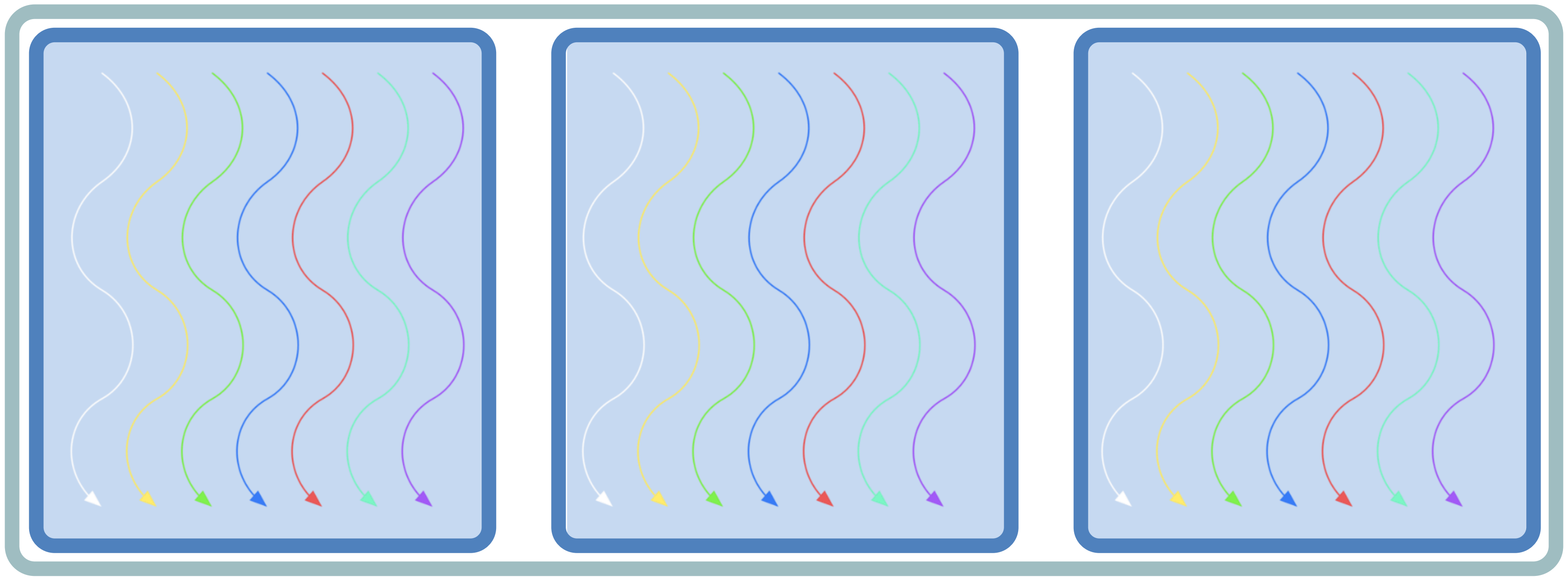
- **Parallel portion of application executed on**
  - → Kernel is executed as array of threads
  - → All threads execute the same code
  - → Threads are identified by IDs
    - → Select input/output data
    - → Control decisions



```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

With OpenACC, you don't have to bother with thread IDs.

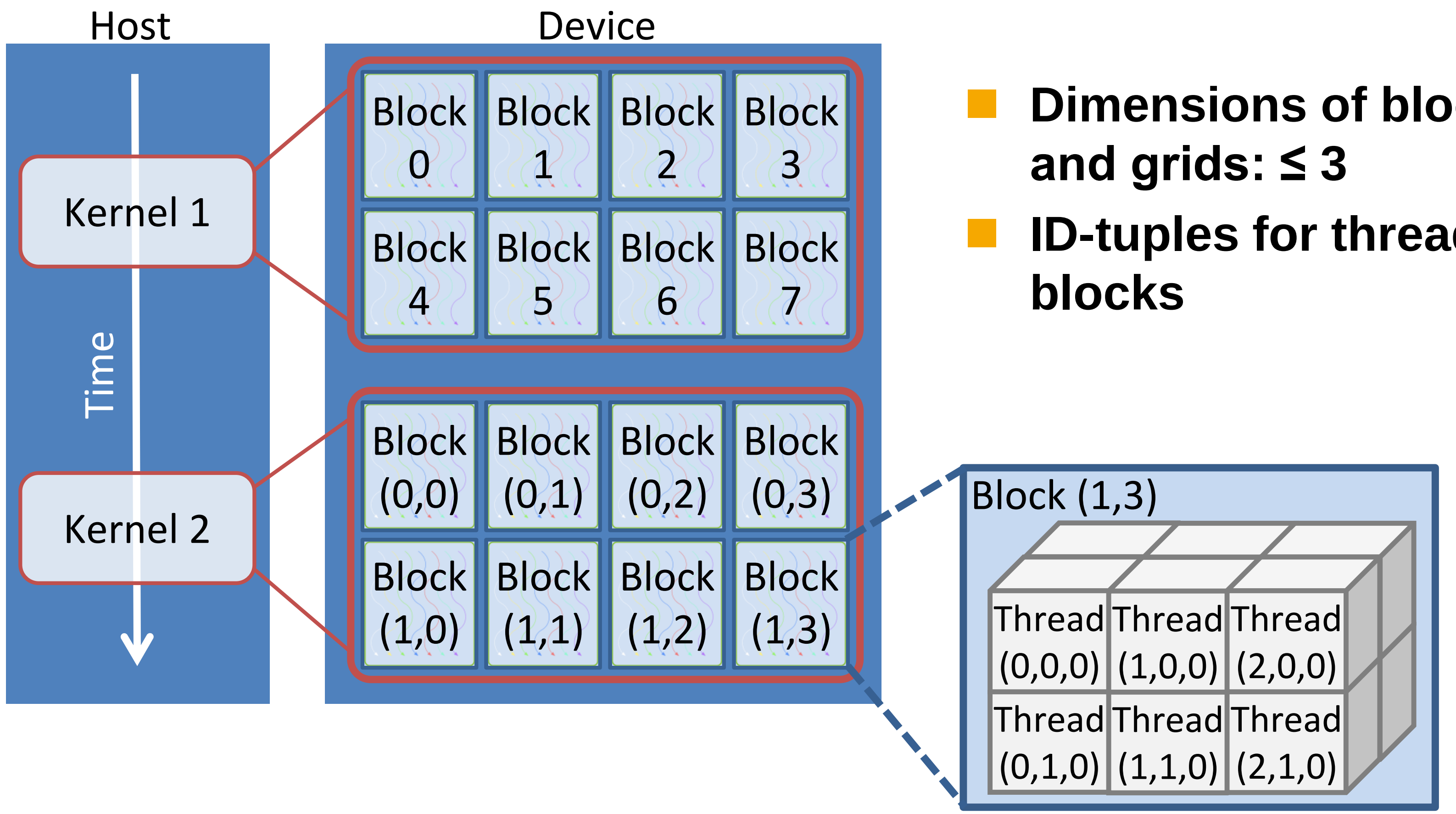
Programming model



- 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
- ID-tuples for threads and blocks

Example SAXPY: Kernel usage

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);
  [..]
}
```

OpenACC

```
void saxpyOpenACC (int n, float a, float
    *x, float *y)
{
#pragma acc kernels or #pragma acc parallel
#pragma acc loop gang vector(128)
    for (int i=0; i < n; ++i) {
      y[i] = a*x[i] + y[i];
    }
}

int main(int argc, char* argv[])
{
  [..]
  // Invoke parallel SAXPY kernel


  saxpyOpenACC(n,2.0,d_x,d_y);


  [..]
}
```

optional for kernels

CUDA – Kernel function & launch

- **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)`

background CUDA

OpenACC directives: Offload regions

- **Offload region**
  - → Region maps to a CUDA kernel function

C/C++
```
#pragma acc parallel [clauses]
```
Fortran
```
!$acc parallel [clauses]

!$acc end parallel
```

→ User responsible for finding parallelism (loops)

→ **`acc loop`** needed for work-sharing

→ No automatic sync between several loops

C/C++
```
#pragma acc kernels[clauses]
```
Fortran
```
!$acc kernels[clauses]

!$acc end kernels
```

→ Compiler responsible for finding parallelism (loops)

→ **`acc loop`** directive only for tuning needed

→ Automatic sync between loops within kernels region

background OpenACC

OpenACC directives: Offload regions

- **Clauses for compute constructs (`parallel`, `kernels`)**

	C/C++, Fortran
→If *condition* true, acc version is executed	`if(condition)`
→Executes async, see Tuning slides	`async [(int-expr)]`
→Define number of parallel gangs(parallel only)	`num_gangs(int-expr)`
→Define number of workers within gang(parallel only)	`num_workers(int-expr)`
→Define length for vector operations(parallel only)	`vector_length(int-expr)`
→Reduction with *op* at end of region(parallel only)	`reduction(op:list)`
→H2D-copy at region start + D2H at region end	`copy(list)`
→Only H2D-copy at region start	`copyin(list)`
→Only D2H-copy at region end	`copyout(list)`
→Allocates data on device, no copy to/from host	`create(list)`
→Data is already on device	`present(list)`
→Test whether data on device. If not, transfer	`present_or_*(list)`
→See Tuning slides	`deviceptr(list)`
→Copy of each *list*-item for each parallel gang(parallel only)	`private(list)`
→As `private` + copy initialization from host(parallel only)	`firstprivate(list)`

OpenACC 2.0 also: `wait, device_type, default(none)`

background OpenACC

OpenACC directives: Loops

- **Share work of loops**
  - → Loop work gets distributed among threads on GPU (in certain schedule)

C/C++

```
#pragma acc loop[clauses]
```

Fortran

```
!$acc loop[clauses]
```

kernels `loop` defines loop schedule by int-expr in `gang`, `worker` or `vector` (instead of `num_gangs` etc with `parallel`)

- **Loop clauses**

	C/C++, Fortran
→ Distributes work into thread blocks	`gang`
→ Distributes work into warps	`worker`
→ Distributes work into threads within warp/ thread block	`vector`
→ Executes loop sequentially on the device......	`seq`
→ Collapse *n* tightly nested loops....................	`collapse(n)`
→ Says independent loop iterations(kernels loop only)....	`independent`
→ Reduction with *op*....................................	`reduction(op:list)`
→ Private copy for each loop iteration...............	`private(list)`

$\left.\begin{matrix}\texttt{gang}\\ \texttt{worker}\\ \texttt{vector}\\ \texttt{seq}\end{matrix}\right\}$ Loop schedule

OpenACC 2.0 also: `auto, tile, device_type`

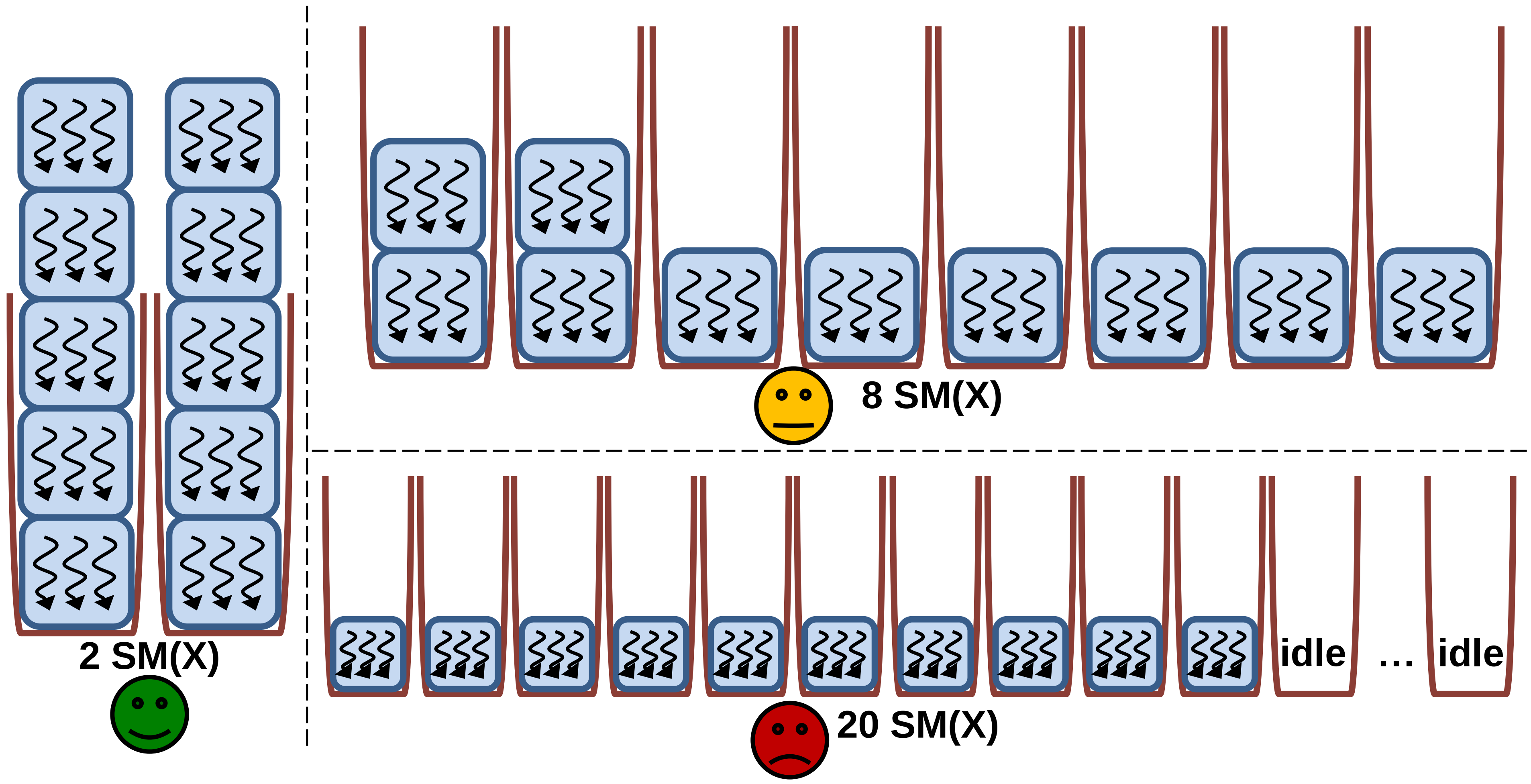
background OpenACC

Programming model

- **Why blocks?**
  - → Cooperation of threads within a block possible
    - → Synchronization (barrier)
    - → 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

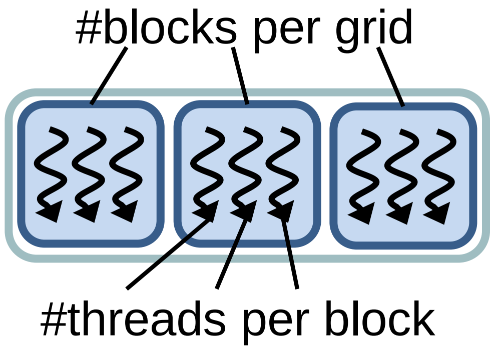
- Assume: 10 thread blocks



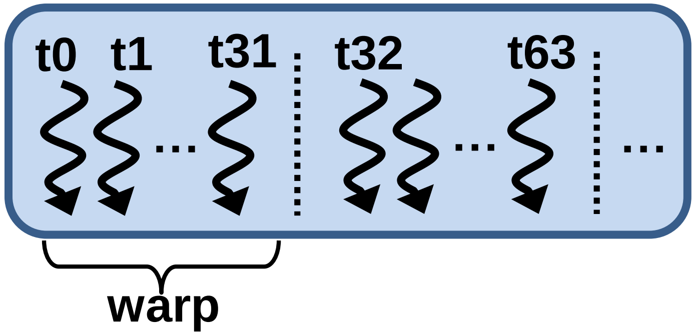
Outline

Execution model

- **Host-directed execution model**
  - → Main program runs on host
  - → Certain code regions run on device
- **Launch configuration**
  - → blocks per grid, threads per block

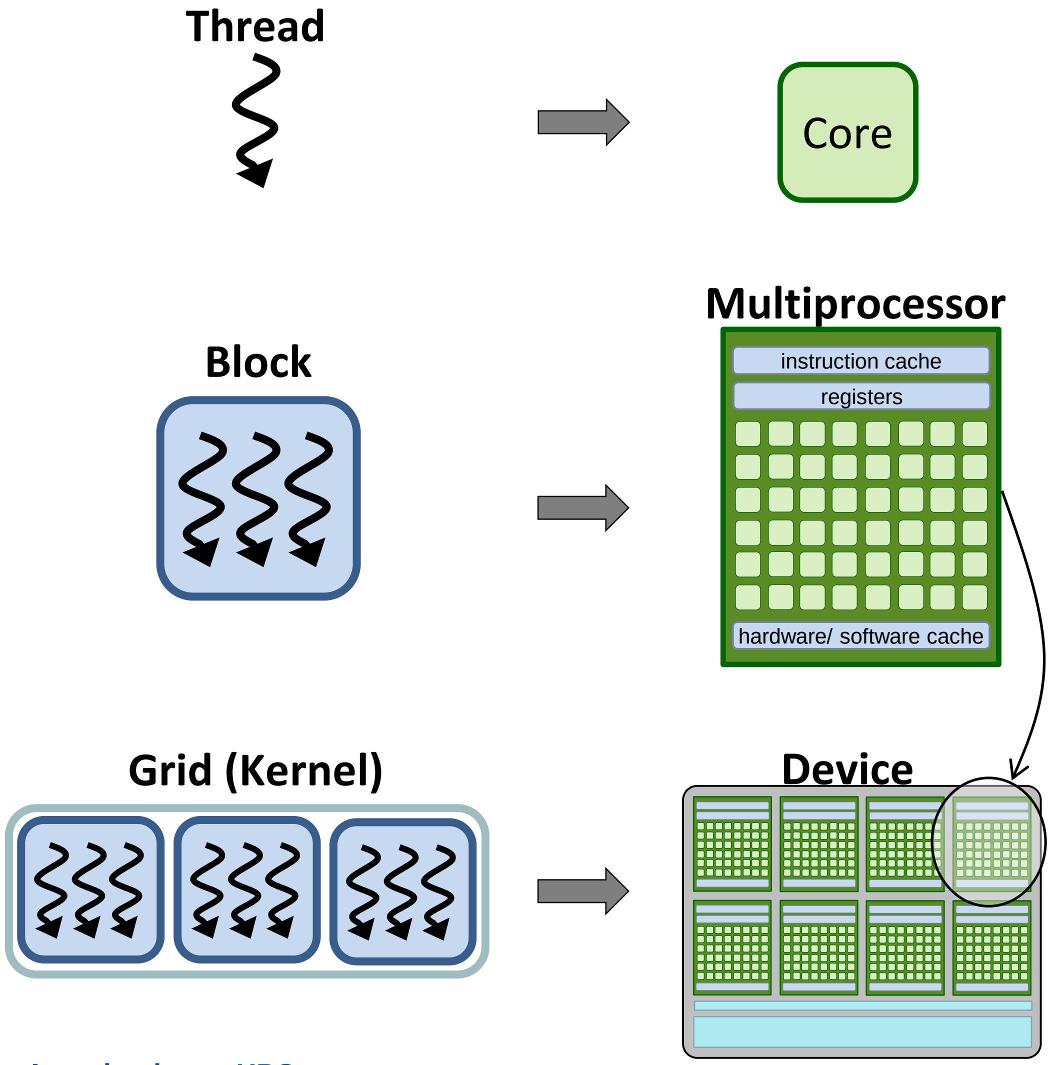


- **Warps**
  - → Threads execute as groups of 32
  - → Threads in warp share same program counter



➔ **Single instruction multiple threads (SIMT)**

Mapping to Execution Model



→ Each thread is executed by a core

→ Each block is executed on a SM(X)

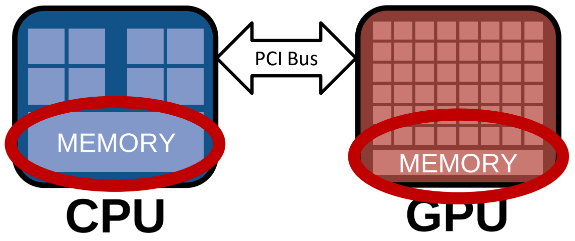
→ Several concurrent blocks can reside on a SM(X) depending on shared resources

→ Each kernel is executed on a device

Outline

Memory model

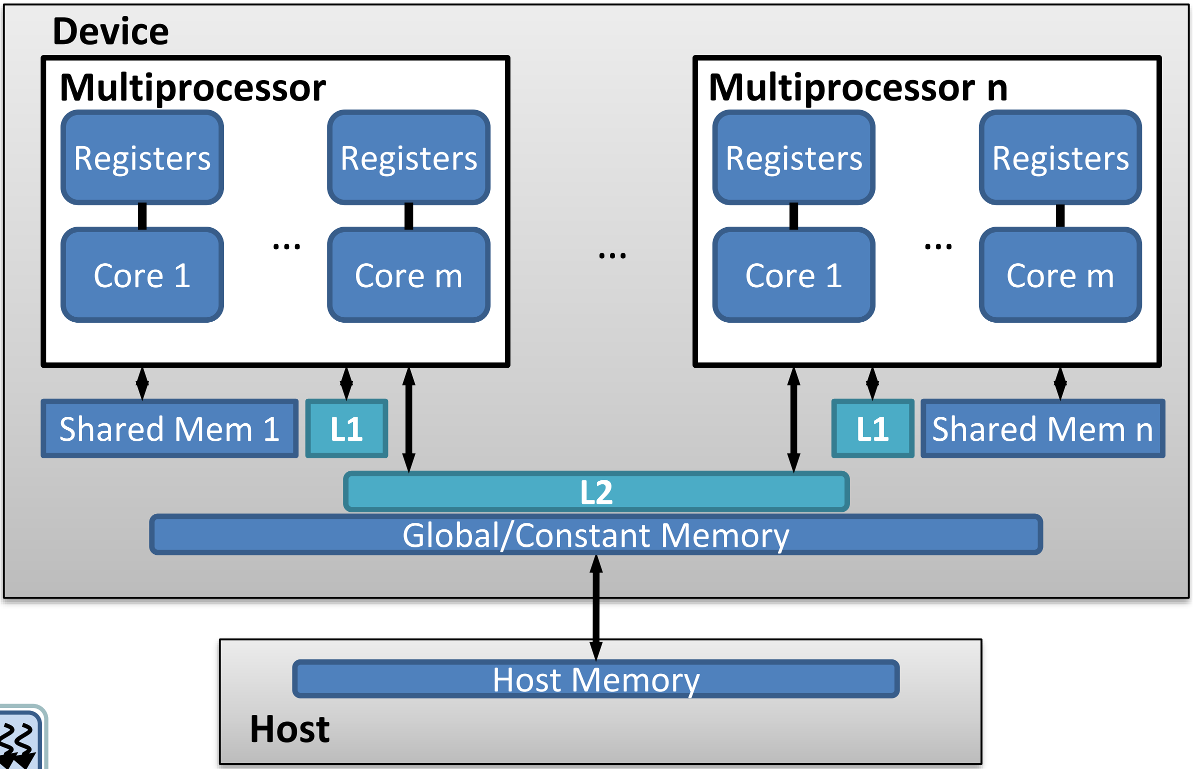
- **Host + device memory = separate entities**
- **No coherence between host + device**
  - → Data synchronization/transfer



- **Host**
  - → (De-)Allocates device memory (global, constant, texture)
  - → Triggers data transfer
- **Device**
  - → Works on device memory (hierarchy)

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

Example SAXPY: Memory

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;
}
```

OpenACC

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

  #pragma acc data copyin(x[0:n])
                   copy(y[0:n])
  {
    //Ir
  }

  free(h_x); free(h_y); return 0;
}
```

```
#pragma acc kernels
      copyin(x[0:n]) copy(y[0:n])
#pragma acc loop gang vector(128)
    for (int i=0; i < n; ++i) {
      y[i] = a*x[i] + y[i];
    }
```

may be optional

CUDA's new *Unified Memory* feature can manage data copies on its own; best performance cannot be expected.

Example SAXPY – CUDA (all)

```
__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[]) {
 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
 cudaM
 cudaM
```

1. Allocate data on GPU + transfer data to CPU

```
 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 b                                          .x);
 saxpyCUDA<<<blocksPerGrid,
   threadsPerBlock>>>(n, 2.0, d_x, d_y);
```

2. Launch kernel

```
 cudaMemcpy(h_y, d_y, n * sizeof(float),
   cuda

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

3. Transfer data to CPU + free data on GPU

CUDA - Memory model

- **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)

background CUDA

OpenACC directives: Data regions & clauses

- **Data region**
  - → Decouples data movement from offload regions

C/C++
```
#pragma acc data [clauses]
```
Fortran
```
!$acc data [clauses]

!$acc end data
```

- **Data clauses**
  - → Triggers data movement of denoted arrays

	C/C++, Fortran
→ If *cond* true, move data to accelerator………………….	`if(cond)`
→ H2D-copy at region start + D2H at region end ……….	`copy(list)`
→ Only H2D-copy at region start …………………………..	`copyin(list)`
→ Only D2H-copy at region end …………………………….	`copyout(list)`
→ Allocates data on device, no copy to/from host ………	`create(list)`
→ Data is already on device ………………………………..	`present(list)`
→ Test whether data on device. If not, transfer………….	`present_or_*(list)`
→ See Tuning slides………………………………………..	`deviceptr(list)`

background OpenACC

Outline

OpenACC - Misc on Loops

- **Combined directives**

```
#pragma acc kernels loop
for (int i=0; i<n; ++i) { /*...*/}

#pragma acc parallel loop
for (int i=0; i<n; ++i) { /*...*/}
```

- **Reductions**

```
#pragma acc parallel
#pragma acc loop reduction(+:sum)
for (int i=0; i<n; ++i) {
  sum += i;
}
```

Also possible:
*, max, min,
&, |, ^, &&, ||

PGI compiler can often recognize reductions on its own. See compiler feedback, e.g.:
`Sum reduction generated for var`

CUDA Reduction (just as comparison)

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + tid;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
__syncthreads();
if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
__syncthreads(); }
if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
__syncthreads(); }
if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; }
__syncthreads(); }
if (tid < 32) {
if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Source: NVIDIA, “Optimizing Parallel Reduction in CUDA”

Efficient reduction implementations with CUDA are hard! See the NVIDIA SDK for hints.

OpenACC - Array Shaping

- **Data clauses can be used on `data`, `kernels` or `parallel`**
  - → `copy, copyin, copyout, present, present_or_copy, create, deviceptr`

- **Array shaping**
  - → Compiler sometimes cannot determine size of arrays
  - → Specify explicitly using data clauses and array “shape”

[lower bound: size]

C/C++

```
#pragma acc data copyin(a[0:length]) copyout(b[s/2:s/2])
```

Fortran

```
!$acc data copyin(a(0:length-1)) copyout(b(s/2:s))
```

[lower bound: upper bound]

OpenACC - update (in examples)

```
#pragma acc data copy(x[0:n])
{
  for (t=0; t<T; t++){
    // Modify x on device (e.g. in subroutine
    // w/o data copying)

    #pragma acc update host(x[0:n])

    // Modify x on host

    #pragma acc update device(x[0:n])
  }
}
```

Directives: Update

- **Update executable directive**
  - → Move data from GPU to host, or host to GPU
  - → Used to update existing data after it has changed in its corresponding copy

C/C++
```
#pragma acc update host|device [clauses]
```

Fortran
```
!$acc update host|device [clauses]
```

OpenACC 2.0: `update self` is preferred over `update host`. `wait` clause also possible.

  - → Data movement can be conditional or asynchronous

- **Update clauses**

	C/C++, Fortran
→ *list* variables are copied from acc to host………………	`host(list)`
→ *list* variables are copied from host to acc………………	`device(list)`
→ If *cond* true, move data to accelerator…………………	`if(cond)`
→ Executes async, see Tuning slides……………………	`async[(int-expr)]`

background OpenACC

OpenACC - Unstructured Data Lifetime

- **Structured data lifetime**
  - → Since OpenACC 1.0

```
#pragma acc data copyin(x[0:n]) \
                 create(y[0:n])
{
  // data lifetime
}
```

- **Unstructured data lifetime**
  - → Since OpenACC 2.0
  - → **`enter data`**: allocates (+ copies) data to device memory
  - → **`exit data`**: deallocates (+ copies) data from device memory

```
class Matrix {
 Matrix() {
  v = new double[n];
  #pragma acc enter data create(v[0:n])
 }
 ~Matrix() {
  #pragma acc exit data delete(v[0:n])
  delete[] v;
 }
 private:
  double* v;
}
```

Also possible: **`copyin`**

Also possible: **`copyout`**

Directives: Unstructured Data Lifetime

- **Enter data construct**
  - → Allocation (& copy) of scalars and (sub-)arrays in(to) device memory
  - → Remain there until end of program or corresponding exit data call

C/C++
```
#pragma acc enter data clause-list
```
Fortran
```
!$acc enter data clause-list
```

  - → Specific clauses for copy or just creation……….

C/C++, Fortran
```
copyin(list)
create(list)
present_or_copyin(list)
present_or_create(list)
```

- **Exit data construct**
  - → (Copies data to host memory &) deletes data from device memory

C/C++
```
#pragma acc exit data clause-list
```
Fortran
```
!$acc exit data clause-list
```

  - → Specific clauses for copy or deletion……………

C/C++, Fortran
```
copyout(list)
delete(list)
```

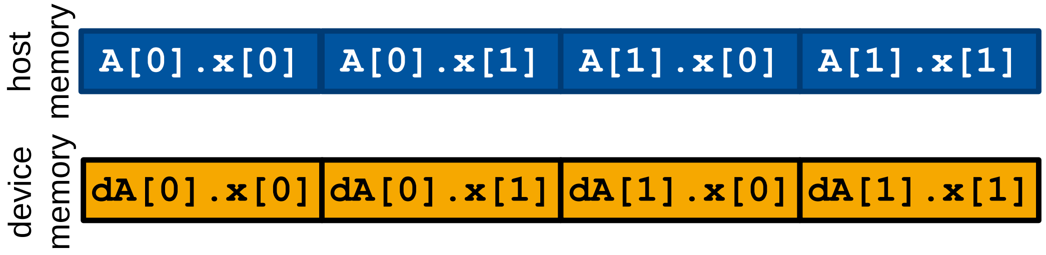
- **Clauses valid for both: `if`, `async`, `wait`**

background OpenACC

Deep Copies

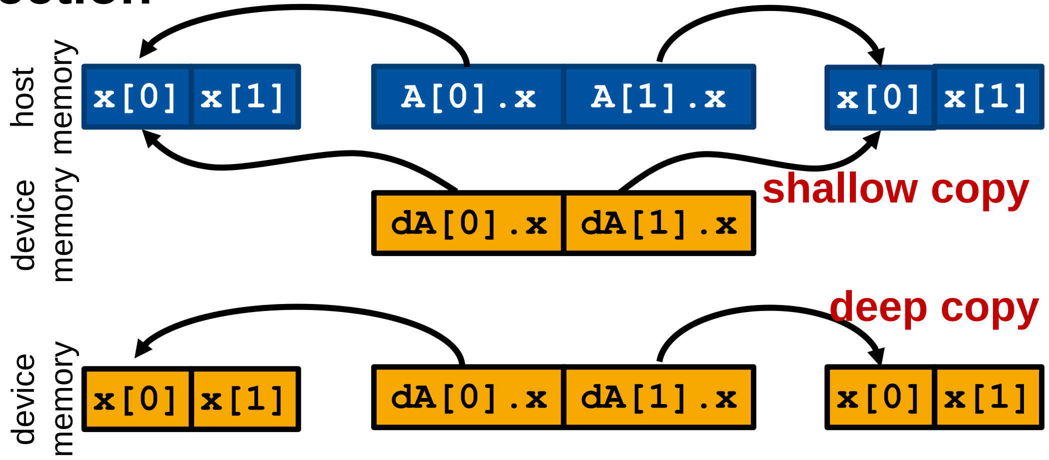
- **Support of a "flat" object model**
  - → Primitive types
  - → Composite types w/o allocatable/pointer members

```
struct {
 int x[2];  // static size 2
} *A;       // dynamic size 2
#pragma acc data copy(A[0:2])
```



- **Challenges with pointer indirection**
  - → Non-contiguous transfers
  - → Pointer translation

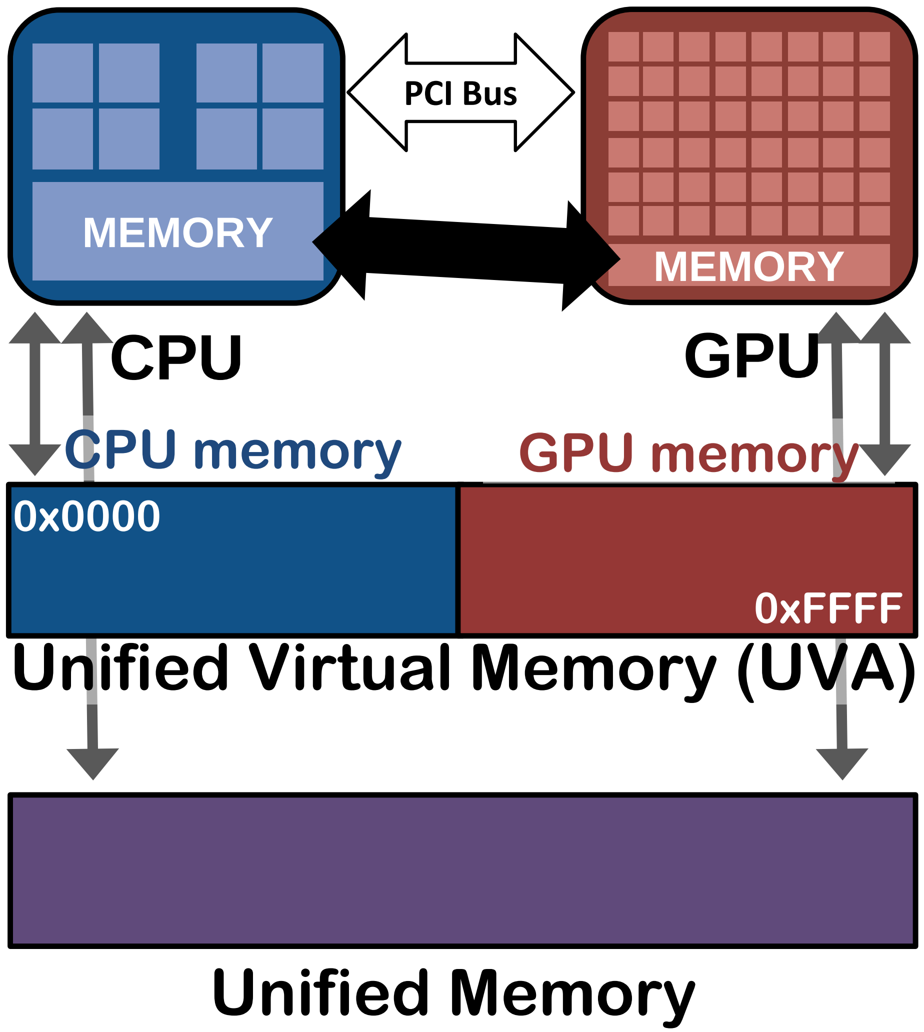
```
struct {
 int *x;    // dynamic size 2
} *A;       // dynamic size 2
#pragma acc data copy(A[0:2])
```



- **Manuel deep copy is possible with data API, but usually not practical**

Courtesy by J.C. Beyer, Cray Inc.

Managed Memory



- Explicit data transfers
- Single virtual address space for all CPU and GPU memory
- Did not get rid of the required explicit memory copying
- Depends on UVA
- “Managed memory” accessible by single pointer for CPU/GPU
- System automatically migrates corresponding data

Example SAXPY: Managed Memory

CUDA

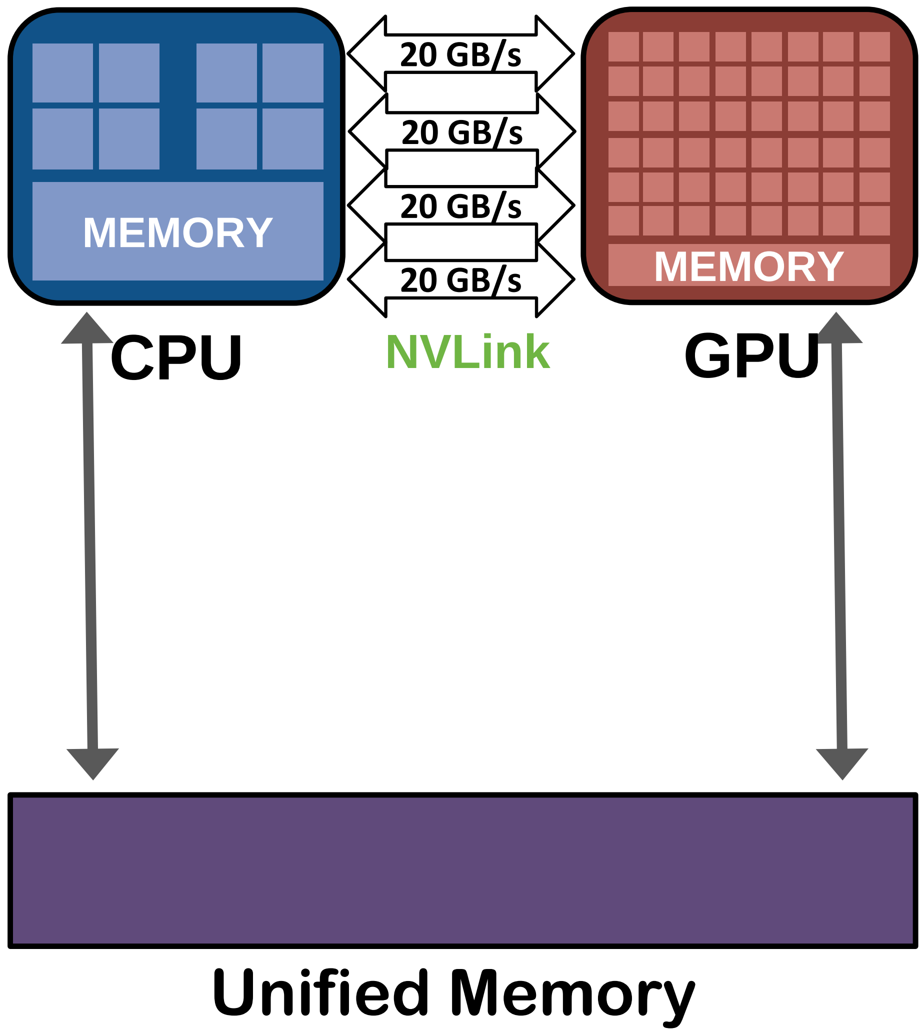
```
int main(int argc,char* argv[]){
  float* h_x,*h_y; // host pointer
  cudaMallocManaged(&h_x, n*sizeof(float));
  cudaMallocManaged(&h_y, n*sizeof(float));
  // Initialize h_x and h_y


  // Invoke parallel SAXPY kernel
  dim3 threadsPerBlock(128);
  dim3 blocksPerGrid(n/threadsPerBlock.x);
  saxpyCUDA<<<blocksPerGrid,
   threadsPerBlock>>>(n,2.0,h_x,h_y);


  cudaFree(h_x); cudaFree(h_y);
  return 0;
}
```

OpenACC

```
pgcc -acc -ta=tesla:managed saxypy.c
```

All data clauses will be ignored

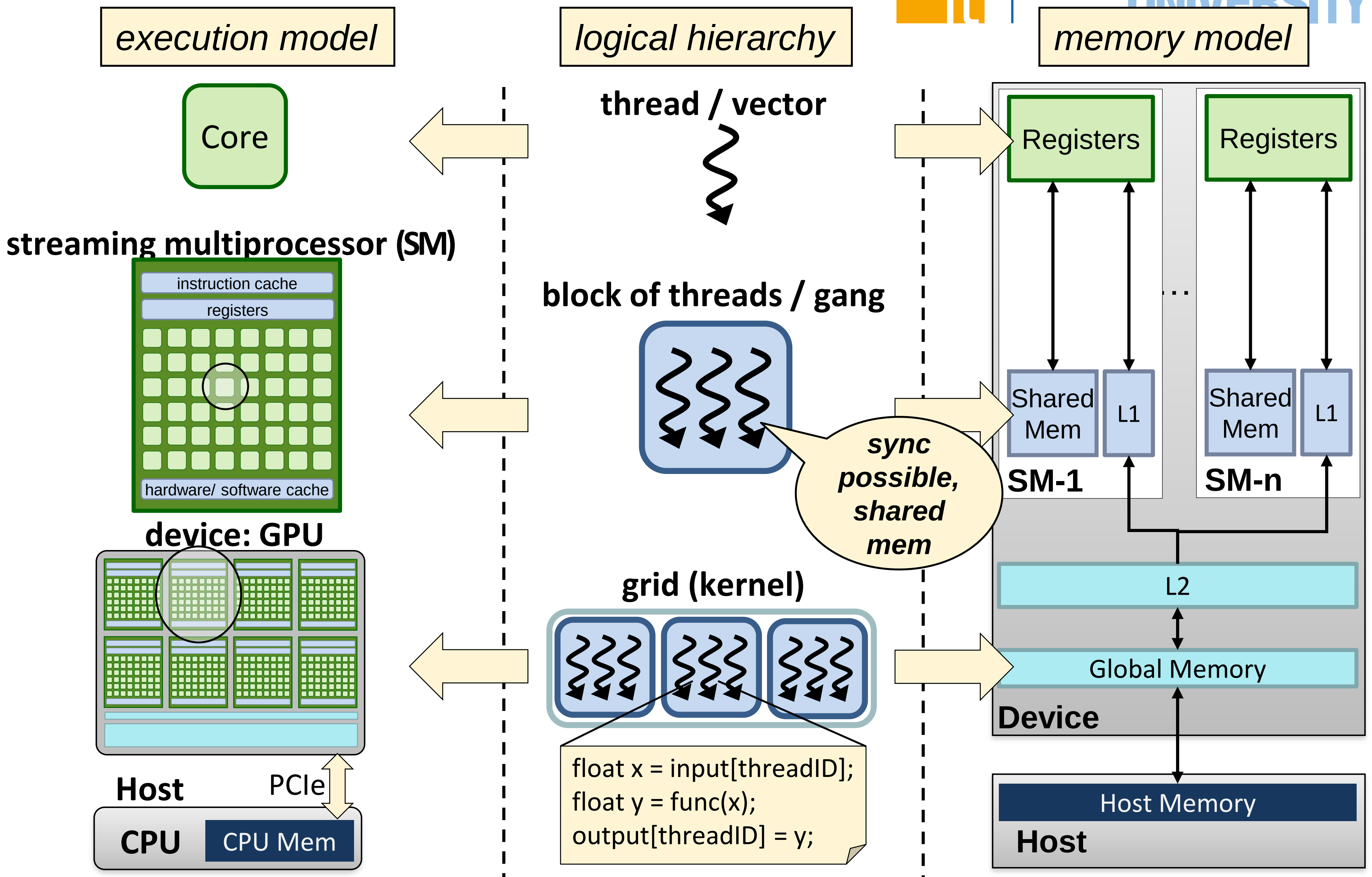
NVLink



- **Interconnect technology by NVIDIA (2016)**
  - → 20 GB/s per link
- **Use cases**
  - → CPU-GPU
  - → GPU-GPU

Outline

Summary



```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Vector, worker, gang mapping is compiler dependent.

What you have learnt

- **How does a GPU look like?**
  - → Why do GPUs deliver a good performance per Watt ratio?
  - → What is the difference to CPUs?
  - → How does the memory hierarchy look like?
  - → How can the logical programming hierarchy be mapped to the execution model?
- **Which models can be used to program a GPU?**
  - → How to handle offloading of regions?
  - → How to handle data management?
  - → What are the main differences?

Quiz (1/2)

- **What speedup would you expect when porting your code from a modern CPU node to the latest high-end GPU?**
  a) 0.5x
  b) 2x
  c) 10x
  d) 100x

Quiz (2/2)

- **Order the following interconnects by the bandwidth:**
  - → Network fabrics: InfiniBand EDR (4x)
  - → CPU/ GPU interconnects: PCIe 3.0 x16
  - → Memory: 2-sockets each with CPU RAM DDR3 (3 channels), GPU GDDR5 (NVIDIA K20x)

  Answers (low to high):

  a) InfiniBand, PCIe, CPU RAM, GPU DDR5

  b) PCIe, InfiniBand, CPU RAM, GPU DDR5

  c) CPU RAM, PCIe, InfiniBand, GPU DDR5

  d) PCIe, CPU RAM, GPU DDR5, InfiniBand

Quiz (2/2)

Interconnect	Bandwith
100 Gigabit Ethernet	100 Gbit/s
Infiniband EDR (4x or 12x)	100 Gbit/s or 290 Gbit/s
RAM DDR3 (3 channels) per socket Stream Benchmark (Intel Westmere, 2 sockets)	32 GB/s = 256 Gbit/s 40 GB/s = 320 Gbit/s
GPU GDDR5 (NVIDIA K20x) GPU Stream Benchmark	250 GB/s = 2000 Gbit/s 180 GB/s = 1440 Gbit/s
PCIe 3.0 x16 (GPU-CPU)	32 GB/s = 256 Gbit/s
NVIDIA NVLink (single link)	20 GB/s = 160 Gbit/s

- **Order t**
  - → Netw
  - → CPU
  - → Mem
    GPU

Answe

a) InfiniBand, PCIe, CPU RAM, GPU DDR5

b) PCIe, InfiniBand, CPU RAM, GPU DDR5

c) CPU RAM, PCIe, InfiniBand, GPU DDR5

d) PCIe, CPU RAM, GPU DDR5, InfiniBand