GPGPU Programming with OpenACC

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

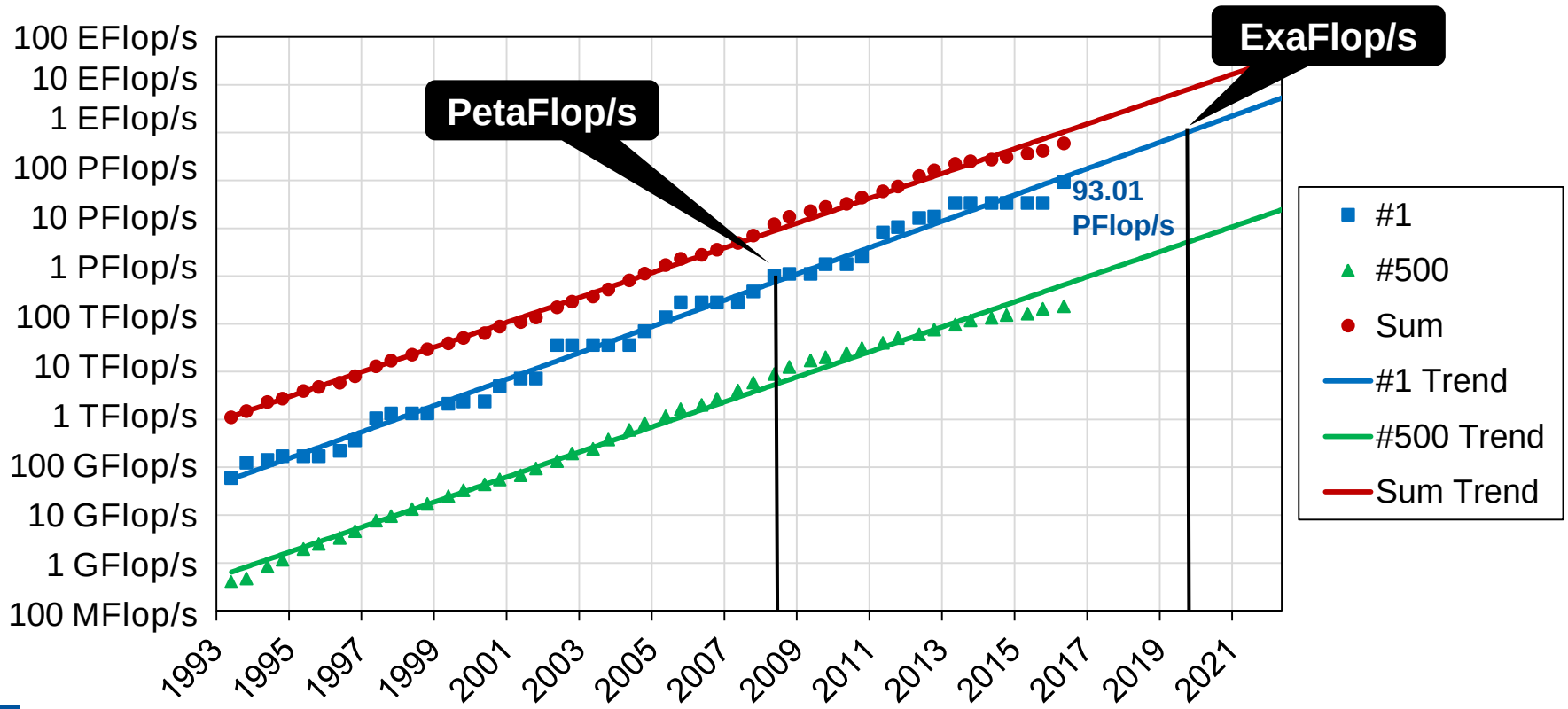
PPCES, March 24th 2017

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Motivation

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

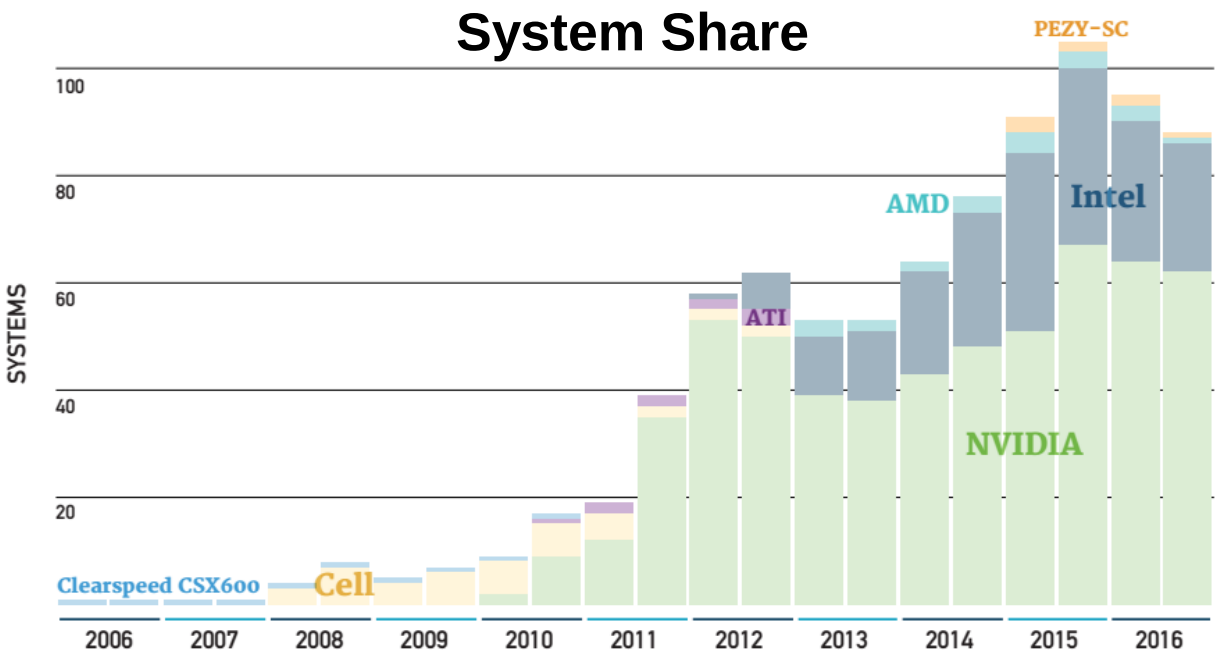


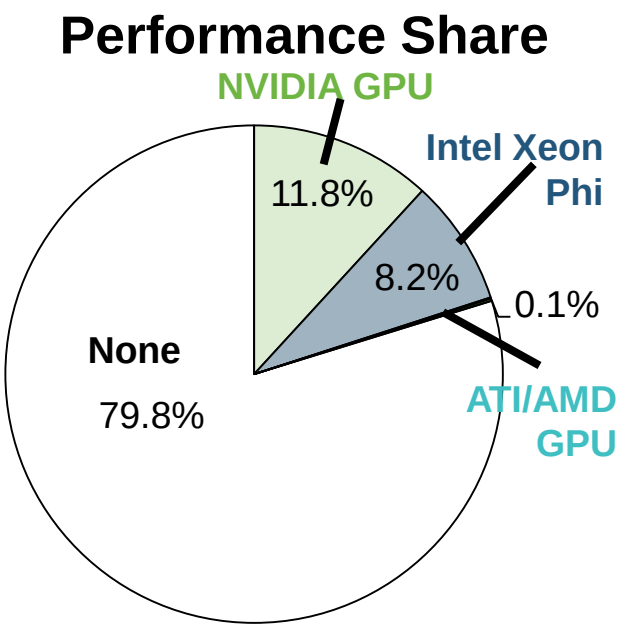
Source:Top500 06/2016

Motivation

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

Here, NVIDIA GPUs are the focus.





Source: Top500, 11/2016

Motivation – Top500 (11/2016)

- **50% of the first 10 Top500 systems contain accelerators**

Rank	Name	Site	Rmax (GFlop/s)	Power (kW)	MFlops/ Watt	Accelerator/ Co-Processor
1	Sunway TaihuLight	National Supercomputing Center in Wuxi (China)	93,014,600	15,371	6051.30	
2	Tianhe-2 (MilkyWay-2)	National Super Computer Center in Guangzhou (China)	33,862,700	17,808	1901.54	**Intel Xeon Phi 31S1P**
3	Titan	DOE/SC/Oak Ridge National Laboratory (USA)	17,590,000	8,209	2142.77	**NVIDIA K20x**
4	Sequoia	DOE/NNSA/LLNL (USA)	17,173,224	7,890	2176.58	None
5	Cori	DOE/SC/LBNL/NERSC (USA)	14,017,700	3,939	3558.70	**Intel Xeon Phi 7250**
6	Oakforest-PACS	Joint Center for Advanced High Performance Computing (Japan)	13,554,600	2,719	4985.14	**Intel Xeon Phi 7250**
7	K computer	RIKEN Advanced Institute for Computational Science (Japan)	10,510,000	12,660	830.17	None
8	Piz Daint	Swiss National Supercomputing Centre (Switzerland)	9,779,000	1,312	7453.51	**NVIDIA P100**
9	Mira	DOE/SC/Argonne National Laboratory (USA)	8,586,612	3,945	2176.58	None
10	Trinity	DOE/NNSA/LANL/SNL (USA)	8,100,900	4,233	1913.75	None

Motivation – Green500 (11/2016)

Rank	MFlops/ Watt	Site	Accelerator/ Co-Processor	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-SC**	150.0
4	6051.3	National Supercomputing Center in Wuxi	-	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	2908.6	Scientific research institution	–	162

Comparison to 2x Intel Sandy Bridge@ 2GHz

- HPL: ~260 W
- Peak performance: 256 GFlop/s

→ 985 MFlops/Watt

Source: Green500, 11/2016

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)

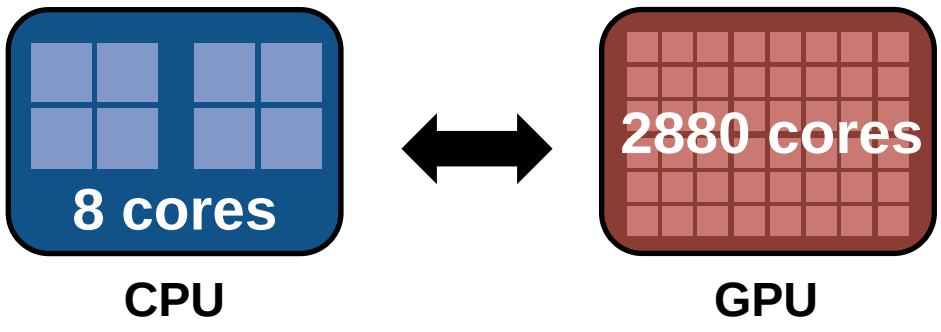
  Researchers take notice of performance growth of GPUs: Tasks must be cast into native graphics operations

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

  Support of programming language C, synchronization,…

  → “General purpose”

Comparison CPU ⇔ GPU

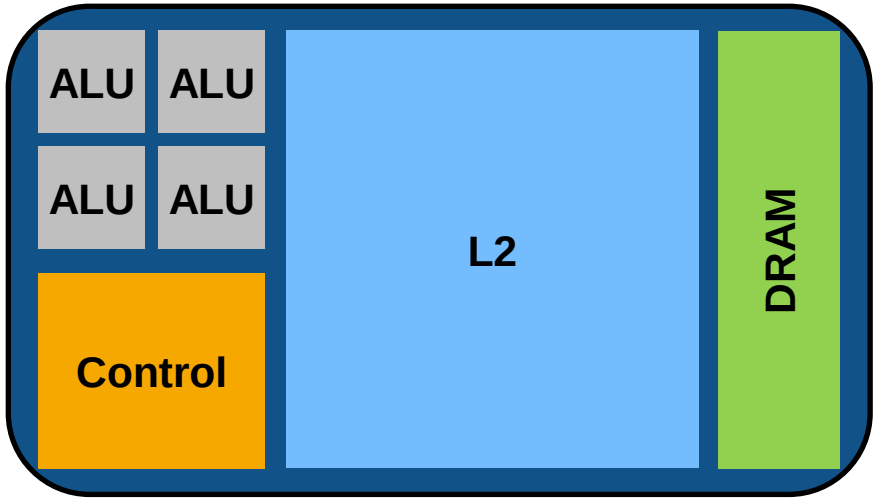


- **GPU-Threads**
  - → Thousands (“few” on CPU)
  - → Light-weight, little creation overhead
  - → Fast switching

- **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

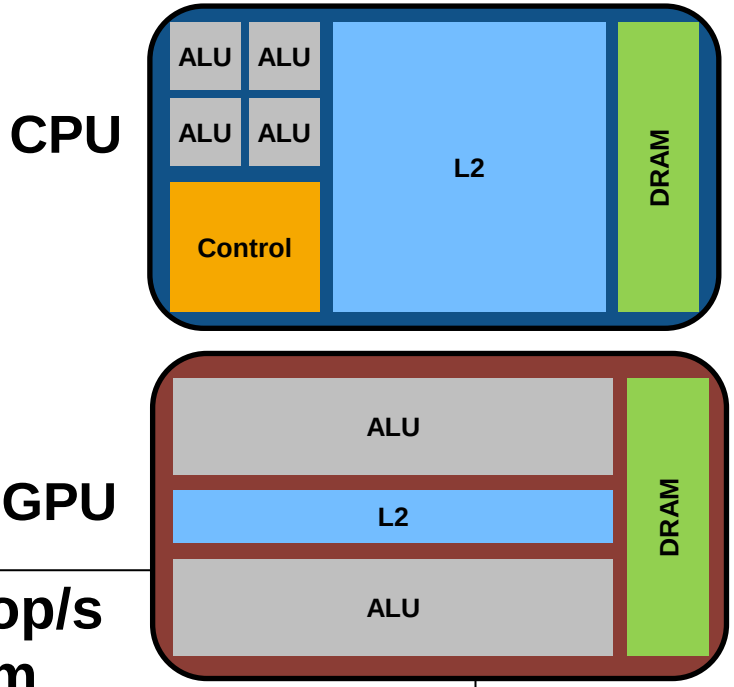
→ Optimized for data-parallel throughput
→ Architecture tolerant of memory latency
→ More transistors dedicated to computation

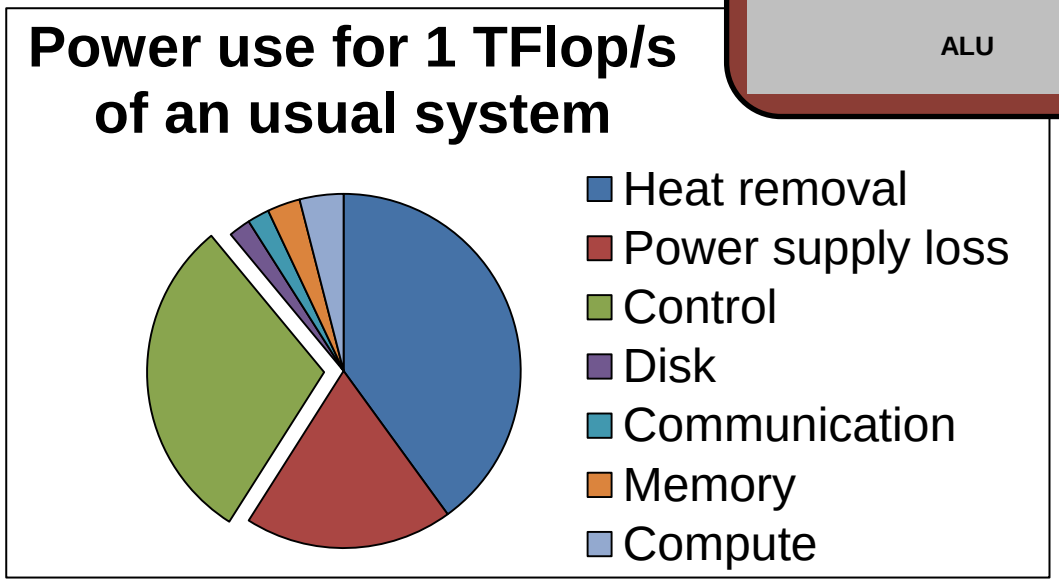
Why can accelerators deliver good performance watt ratio?

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





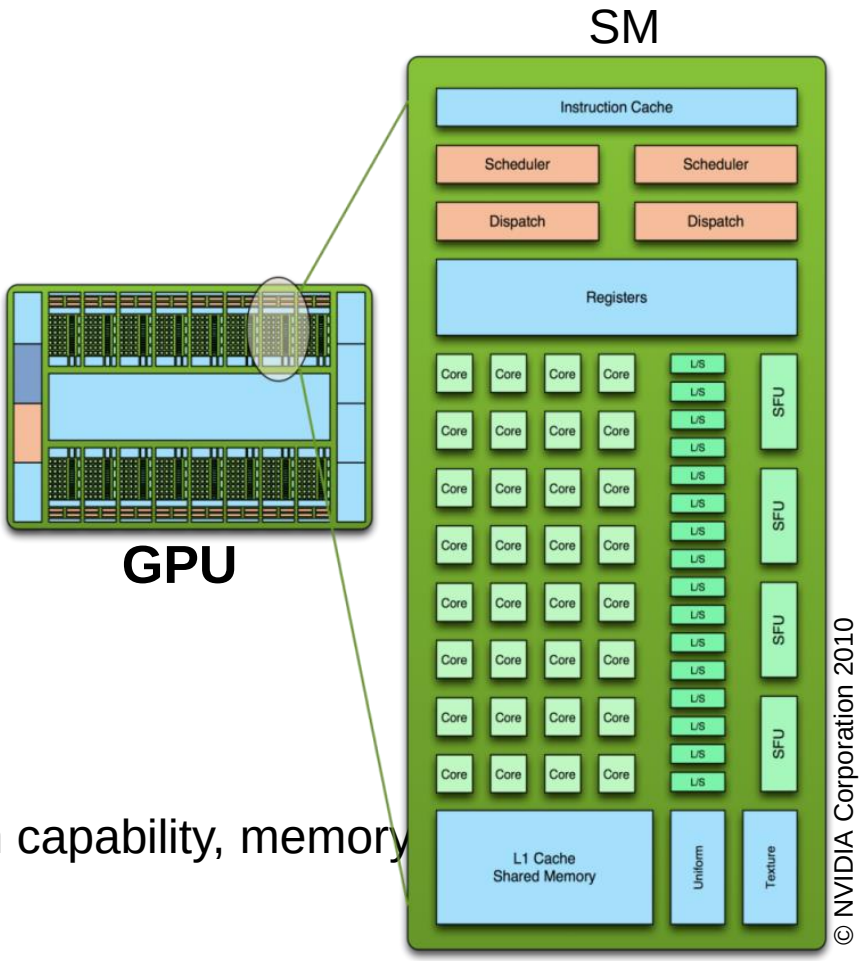
Source: Andrey Semin (Intel): HPC systems energy efficiency optimization thru hardware-software co-design on Intel technologies, EnaHPC 2011; & S. Borkar, J. Gustafson

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

GPU architecture: Fermi

- **3 billion transistors**
- **14-16 streaming multiprocessors (SM)**
  - → Each comprises 32 cores
- **448-512 cores**
  - → i.a. Floating point & integer unit
- **Memory hierarchy**
- **Peak performance (Tesla M2070)**
  - → SP: 1.03 TFlops
  - → DP: 515 GFlops
- **ECC support**
- **Compute capability: 2.0**
  - → Defines features, e.g. double precision capability, memory

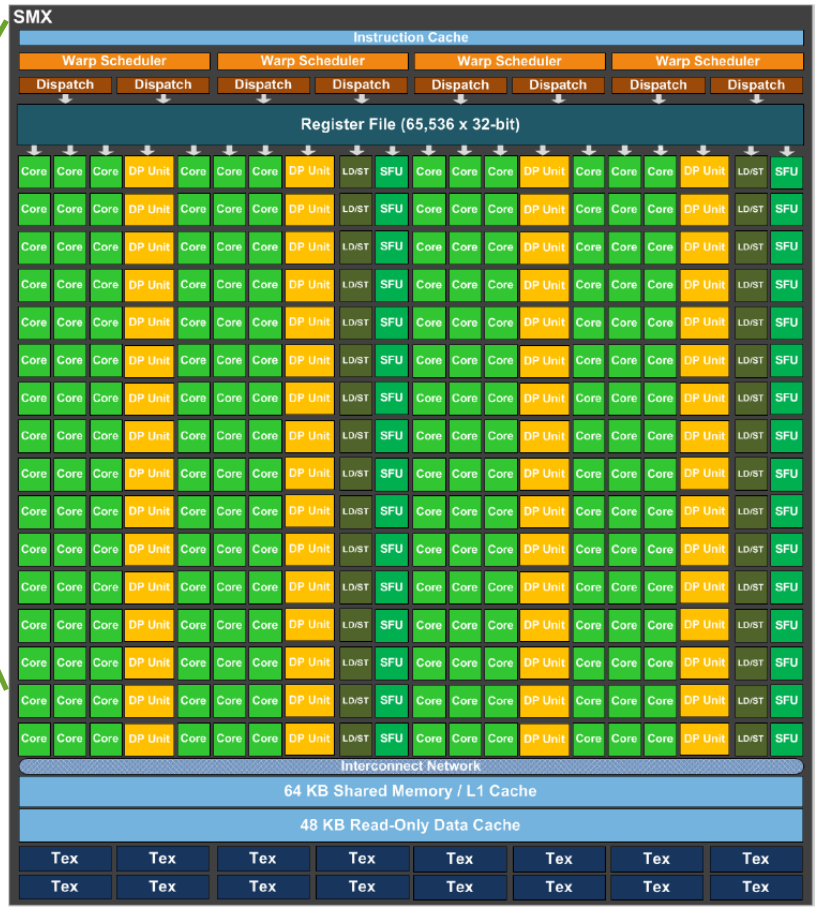


GPU architecture: Kepler

- **7.1 billion transistors**
- **13-15 streaming multiprocessors**
  - → Each comprises 192 cores
- **2496-2880 cores**
- **Memory hierarchy**

GPU

SM

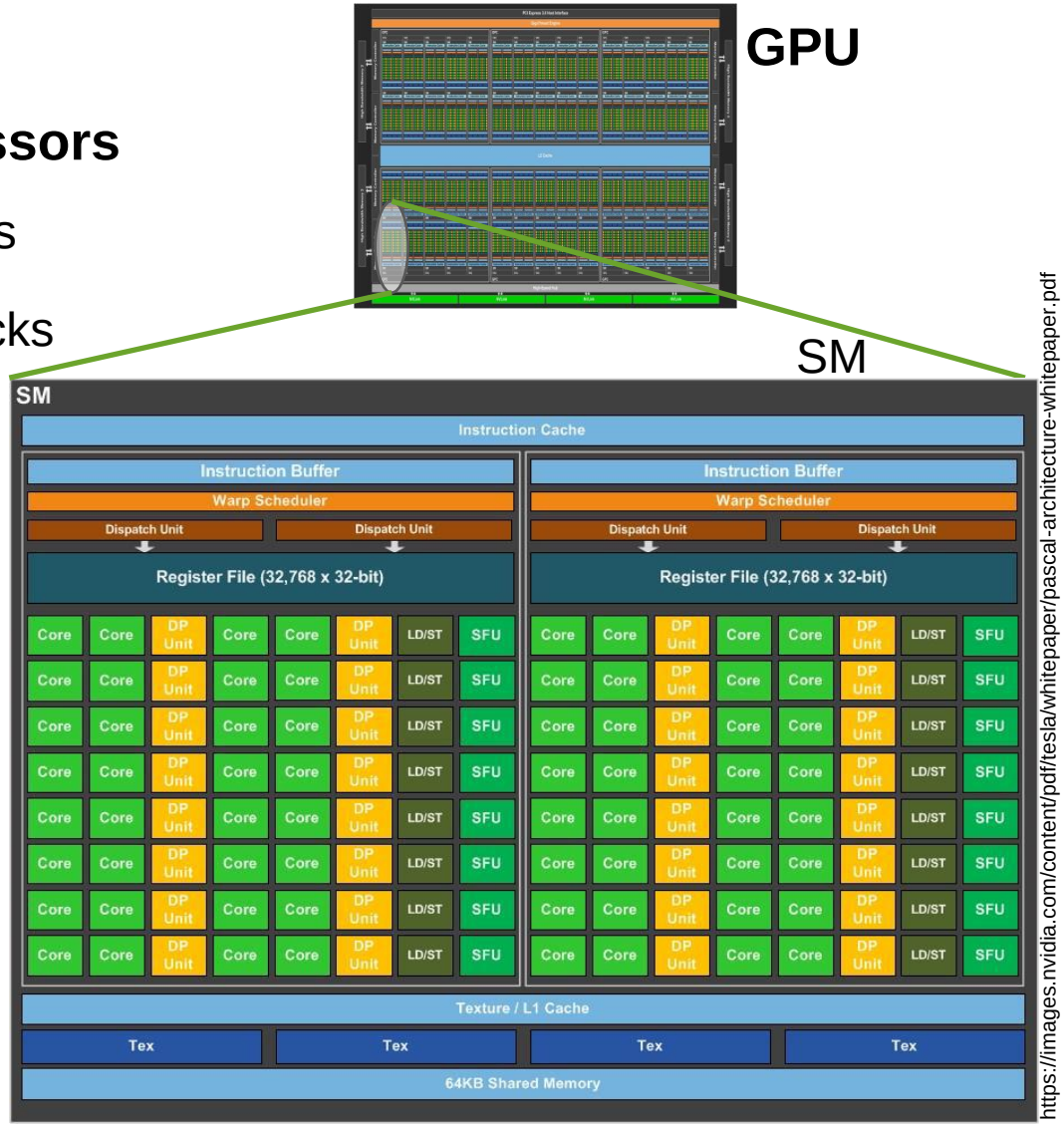


http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

- **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

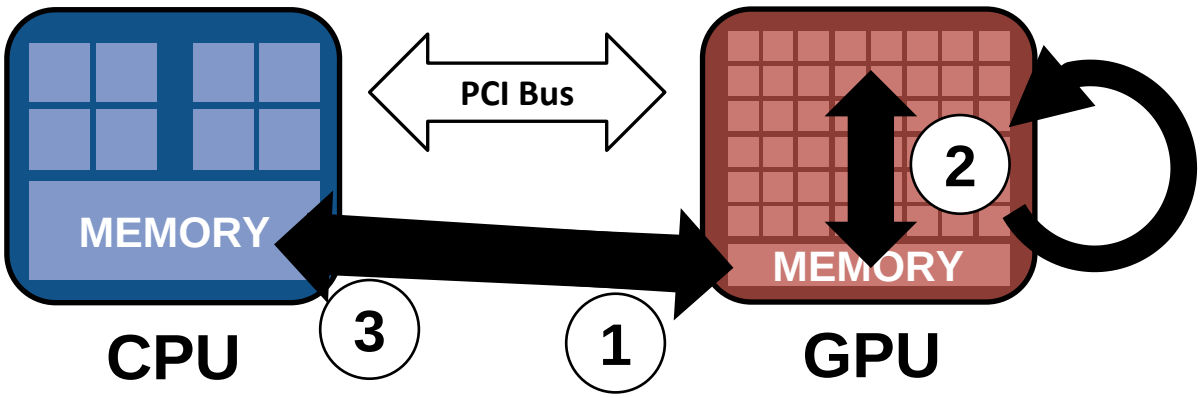
GPU architecture: Pascal

- **15.3 billion transistors**
- **56-60 streaming multiprocessors**
  - → 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



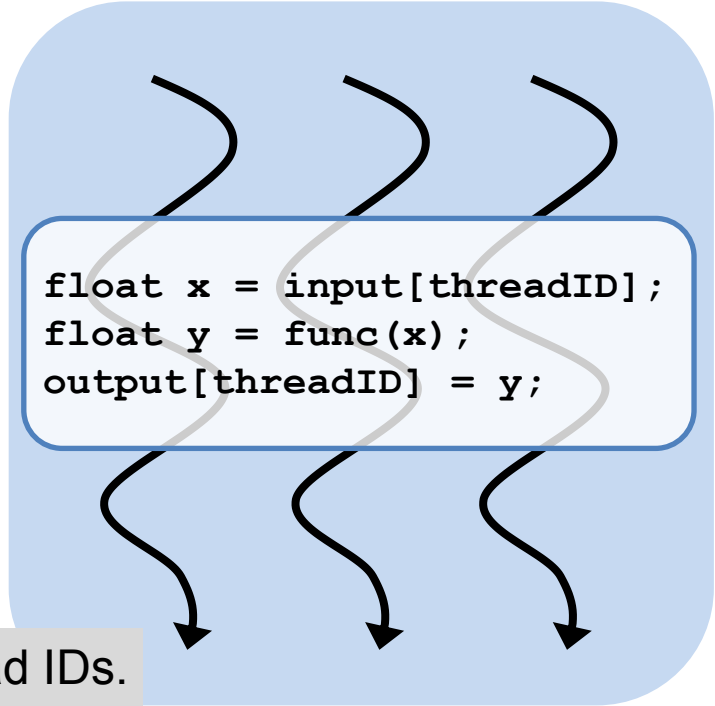
- **Weak memory model**
  - → Host + device memory = separate entities
  - → No coherence between host + device
    - → Data transfers needed
- **Host-directed execution model**
  - → Copy input data from CPU mem. to device mem.
  - → Execute the device program
  - → Copy results from device mem. to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

Programming model

We use CUDA terminology here.

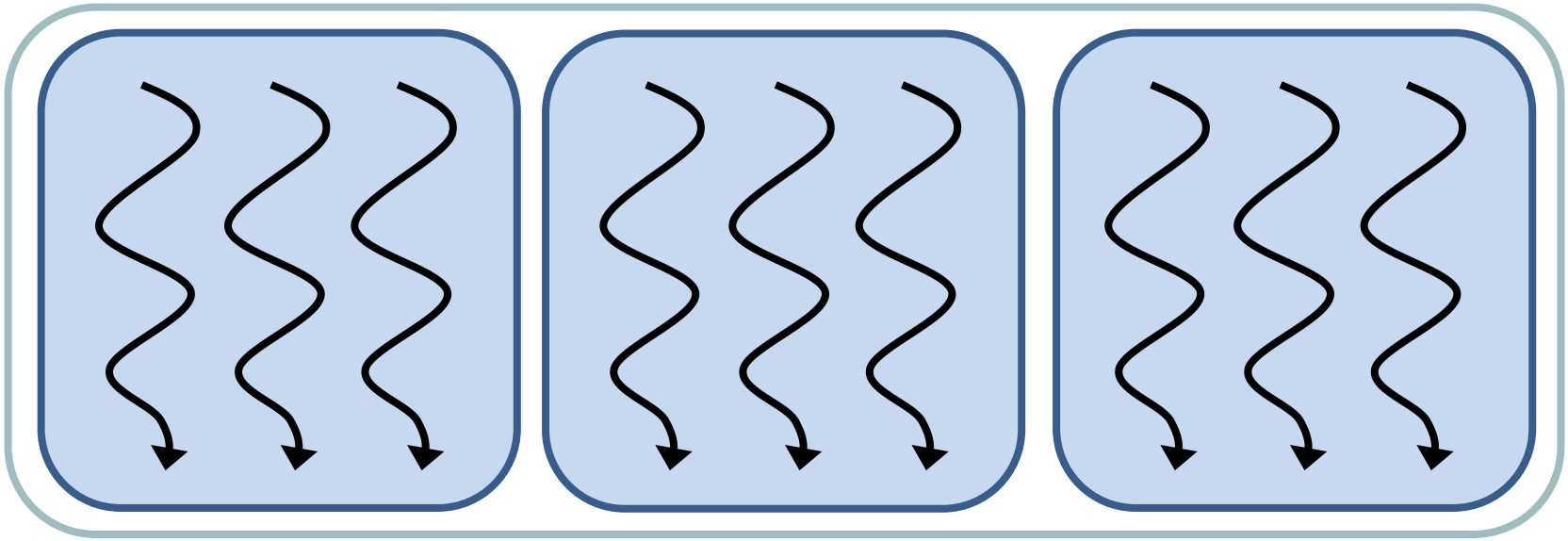
- **Definitions**
  - → Host: CPU, executes functions
  - → Device: usually GPU, executes kernels

- **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



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

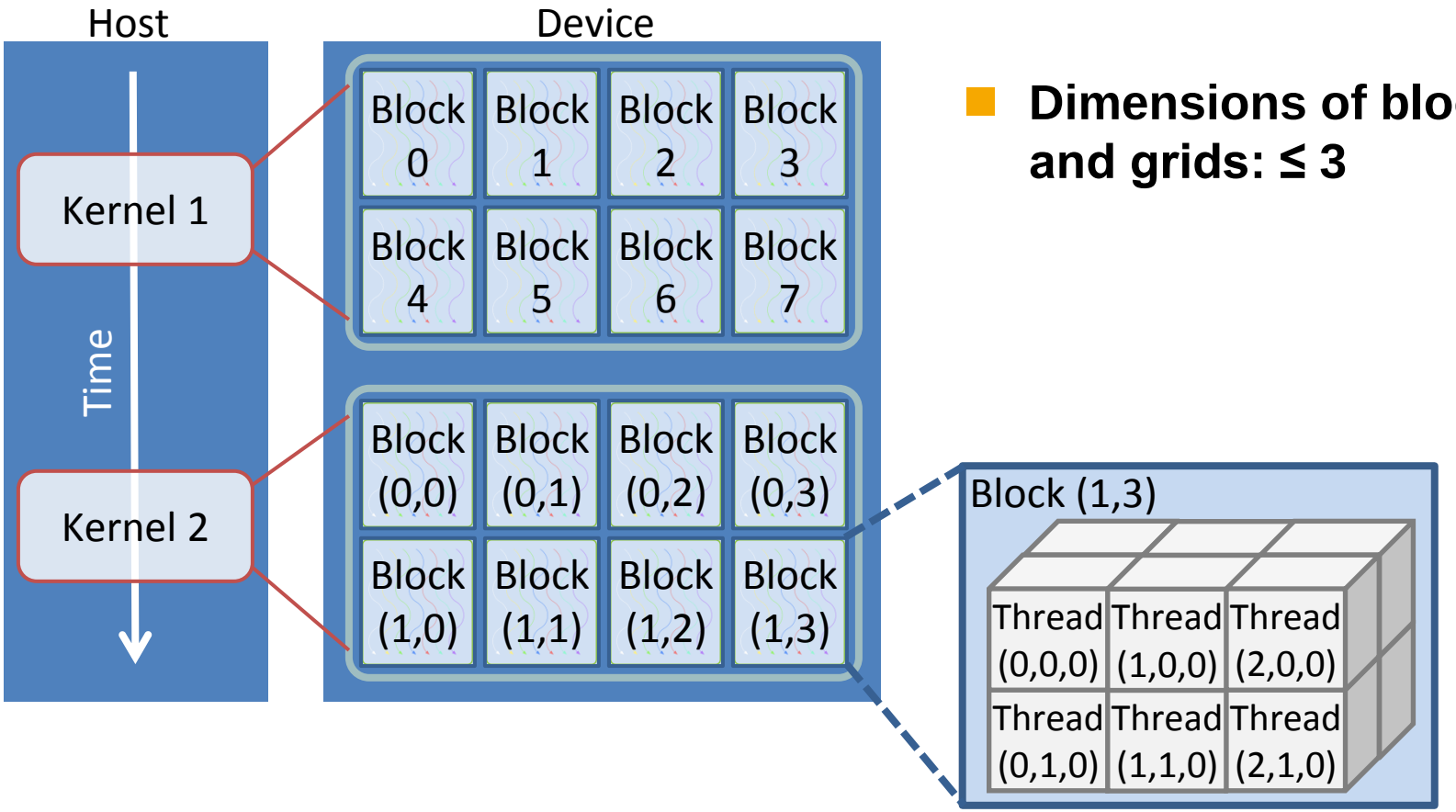
Programming model



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

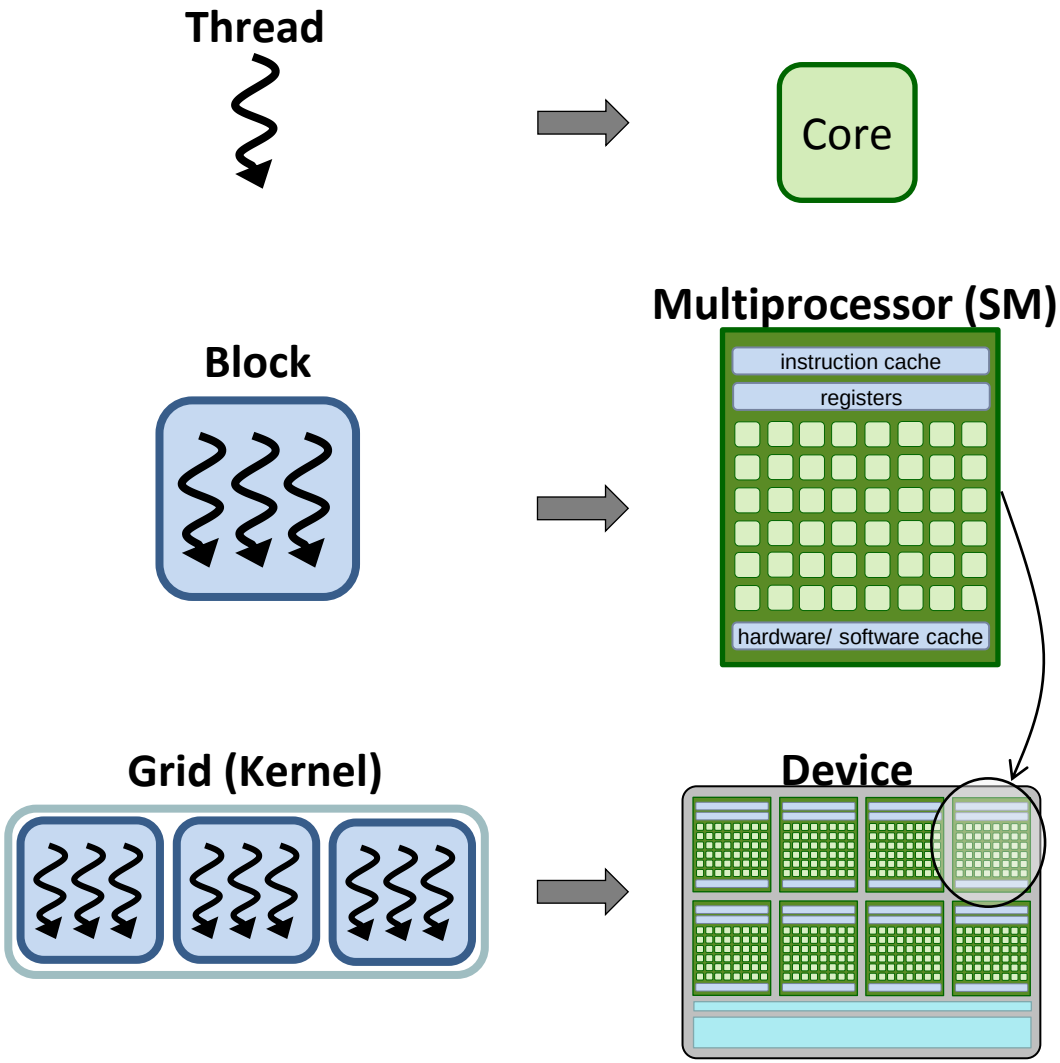
Programming model

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



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

Mapping to Execution Model



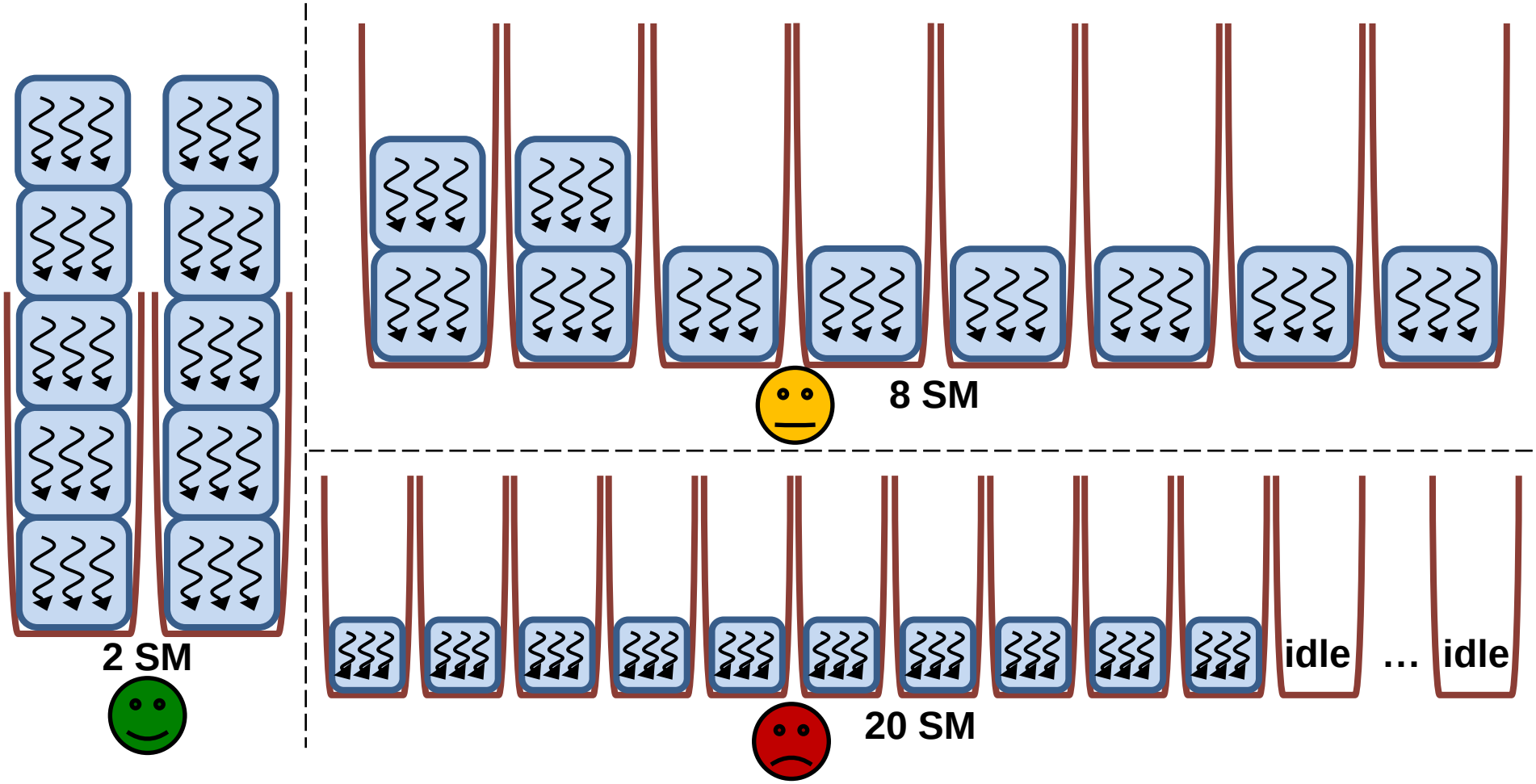
- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources
- Each kernel is executed on a device

Why blocks?

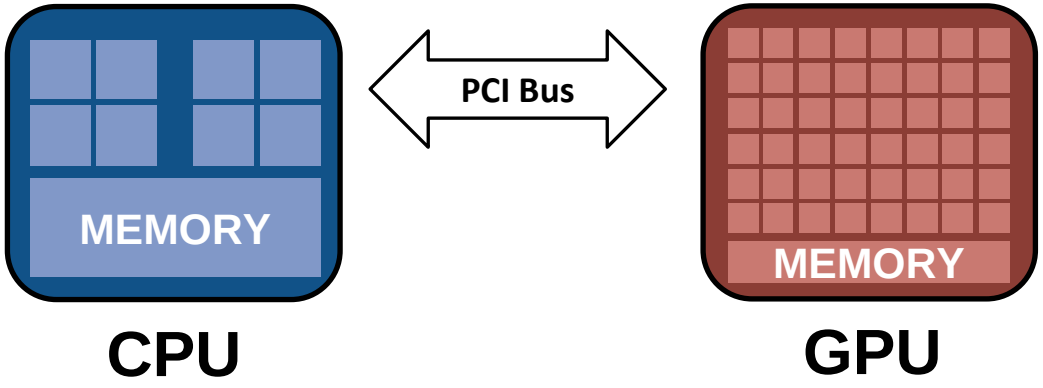
- **Cooperation of threads within a block possible**
  - → Synchronization
  - → Share data/ results using shared memory
- **Scalability**
  - → Fast communication between n threads is not feasible when n large
    - → No global synchronization on GPU possible (only by completing one kernel and starting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

Why blocks?

- **Assume: 10 thread blocks, 1 thread block per SM**



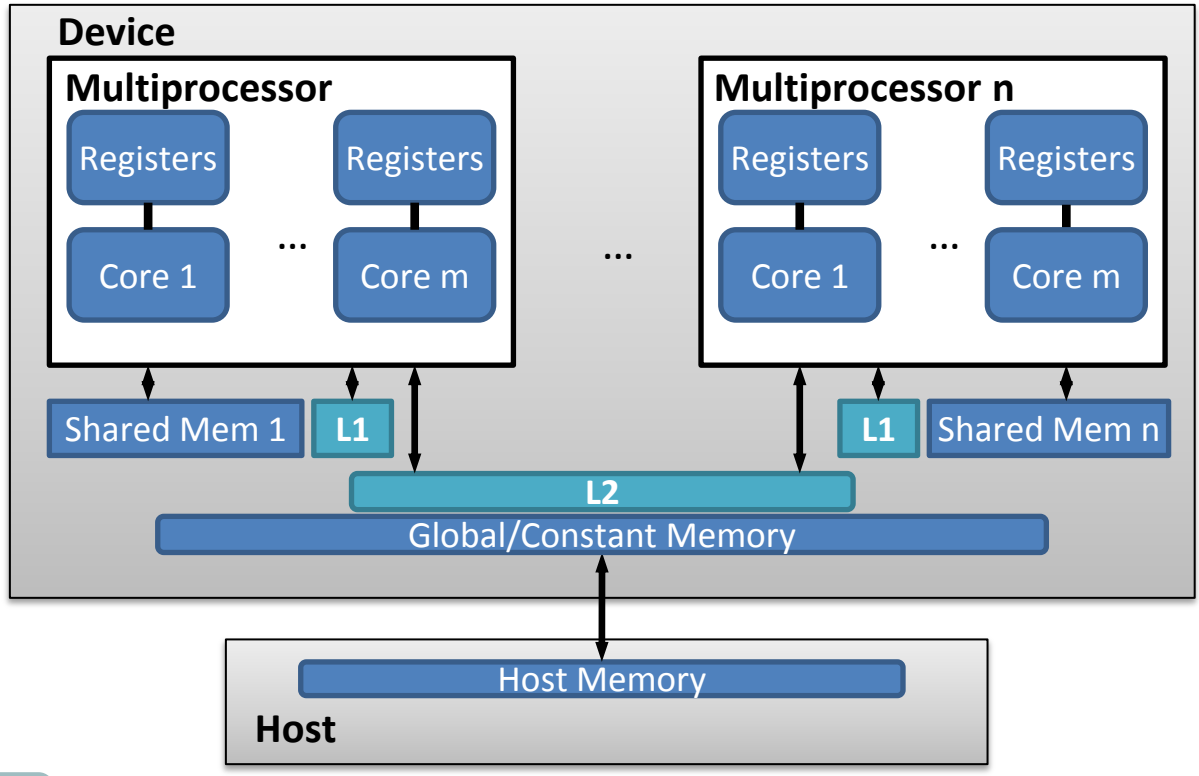
Memory model



- **Host + device memory = separate entities**
  - → But, host and device may share memory
- **No coherence between host + device**
  - → Data synchronization/transfer

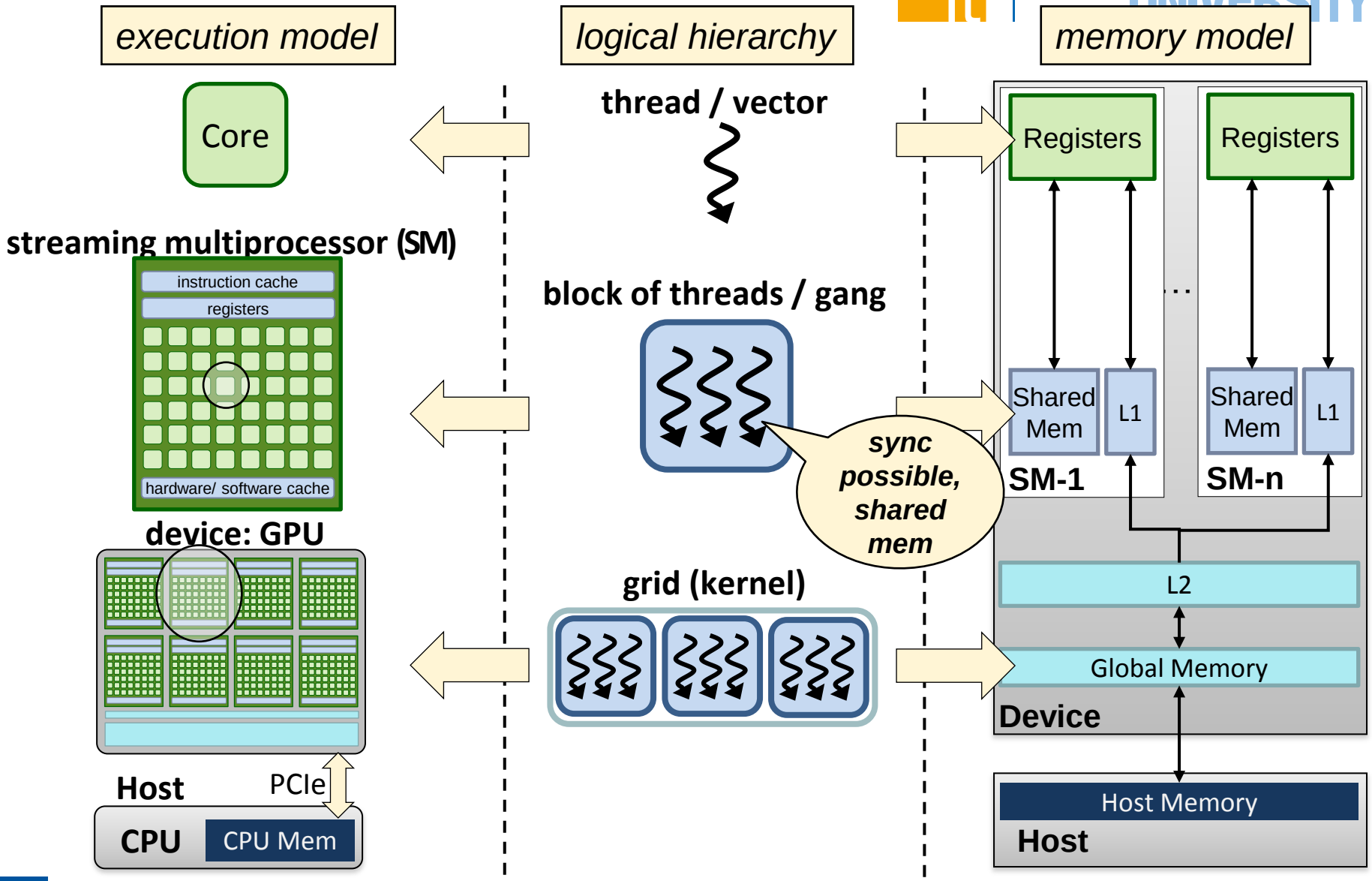
Memory hierarchy

- **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**
  - → *Global* memory
    - several GB; off-chip



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

Summary

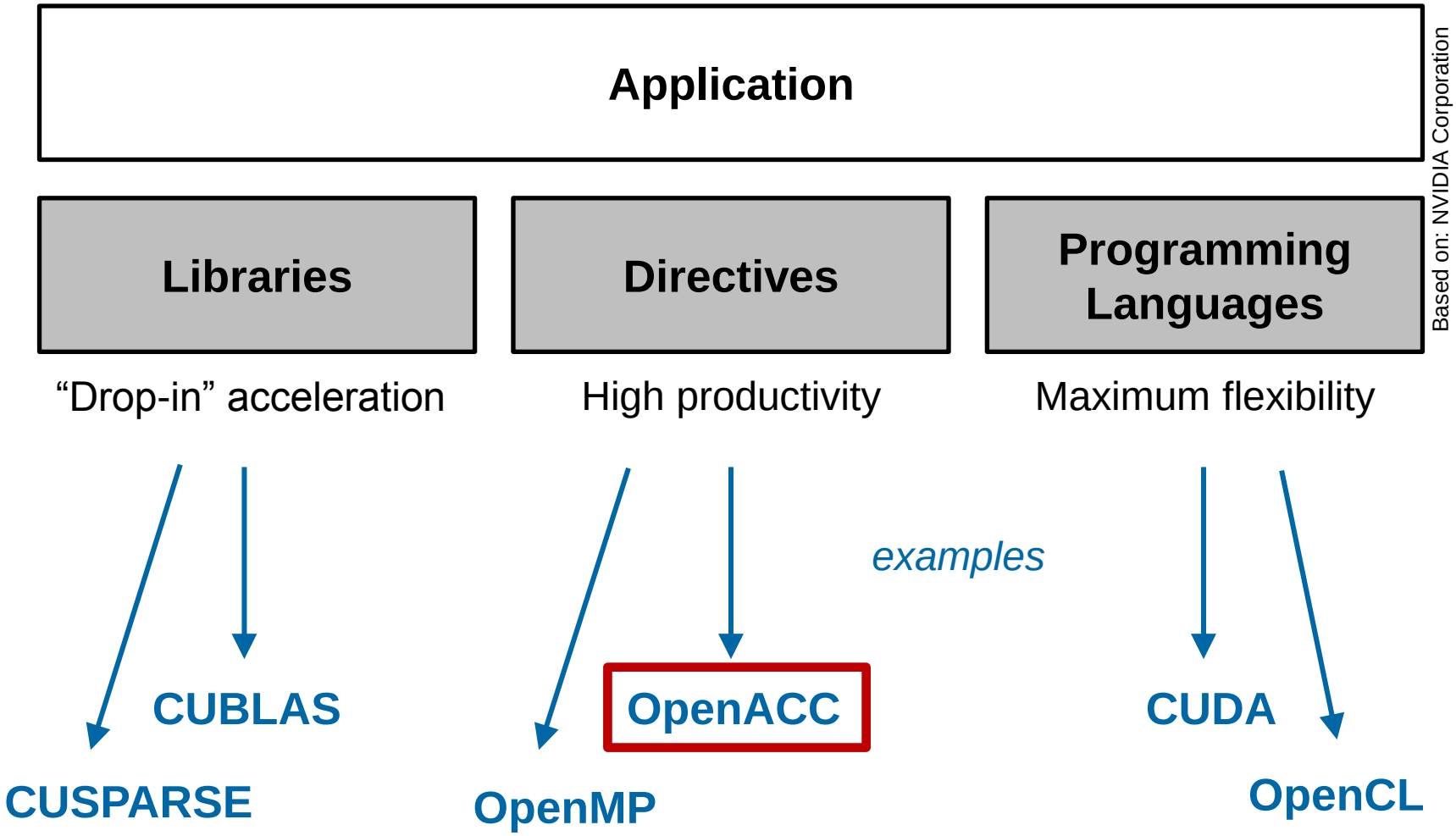


Vector, worker, gang mapping is compiler dependent.

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

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
- **…**

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 [illegible] // [illegible] memory
 cudaM[illegible]
 cudaM[illegible]
```

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[illegible].x);
saxpy_parallel<<<blocksPerGrid,
  threadsPerBlock>>>(n, 2.0, d_x, d_y);
```

2. Launch kernel

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

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

3. Transfer data to CPU + free data on GPU

Motivation

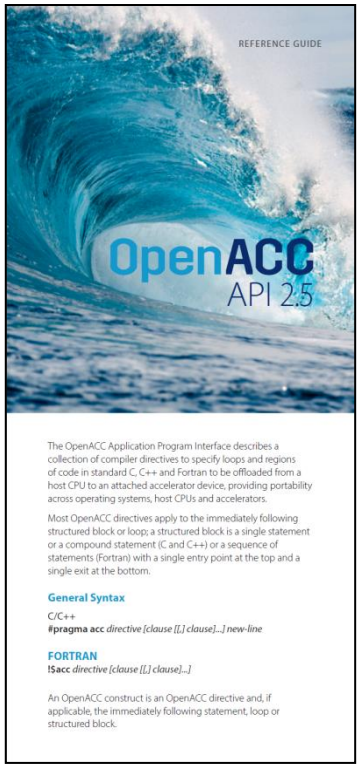
- **Nowadays: GPU APIs (like CUDA, OpenCL) often used**
  - → May be difficult to program (as/but more flexibility)
  - → Verbose/ may complicate software design
- **Directive-based programming model delegates responsibility for low-level GPU programming tasks to compiler**
  - → Data movement
  - →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

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

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

Overview *OpenACC*

- **Open industry standard**
  - → Portability
- **Introduced by CAPS, Cray, NVIDIA, PGI (Nov. 2011)**
- **Today's members and supporter organizations:**
  - → Industry: Allinea, AMD, Cray, NVIDIA, RogueWave,…
  - → Universities/Labs: TU Dresden, EPCC, CSCS, Georgia Tech, Indiana Uni, ORNL, SNL, Supercomputing Center Wuxi,…
- **Support**
  - → C,C++ and Fortran
  - → NVIDIA GPUs, AMD GPUs, x86 Multicore
- **Timeline**
  - → Nov'11: Specification 1.0
  - → Jun'13: Specification 2.0
  - → Nov'15: Specification 2.5



Source: www.openacc.org

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

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,cc20,7.5 -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
- → **`7.5`** Optional. Uses CUDA Toolkit 7.5 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 *Development Tips*).

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Directives (in examples): SAXPY serial (host)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE





  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }



  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

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

Directives (in examples): SAXPY
acc kernels

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE


#pragma acc kernels
{
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

                                  – synchronization –

  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

Kernel 1

Kernel 2

"magic"
compiler analyzes code & tries to parallelize/ optimize it

Directives (in examples): SAXPY
acc kernels

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
```

PGI Compiler Feedback

```
main:
48, Generating implicit copyin(x[:10240])
    Generating implicit copy(y[:10240])
50, Loop carried dependence of y-> prevents parallelization
    Complex loop carried dependence of x-> prevents parallelization
    Loop carried backward dependence of y-> prevents vectorization
    Accelerator kernel generated
    50, #pragma acc loop seq
50, Accelerator scalar kernel generated
60, Loop carried dependence of y-> prevents parallelization
    Complex loop carried dependence of x-> prevents parallelization
    Loop carried backward dependence of y-> prevents vectorization
    Accelerator kernel generated
    60, #pragma acc loop seq
60, Accelerator scalar kernel generated
```

Kernel 1

Kernel 2

Problem: pointer aliasing

Directives (in examples): SAXPY
acc kernels

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float * restrict x = (float*) malloc(n * sizeof(float));
  float * restrict y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
```

PGI Compiler Feedback

```
main:
48, Generating implicit copyin(x[:10240])
    Generating implicit copy(y[:10240])
50, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
    50, #pragma acc loop gang, vector(128) /* blockIdx.x
        threadIdx.x */
60, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
    60, #pragma acc loop gang, vector(128) /* blockIdx.x
        threadIdx.x */
```

Kernel 1

Kernel 2

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

Directives (in examples): SAXPY
acc kernels

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float * restrict x = (float*) malloc(n * sizeof(float));
  float * restrict y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE



#pragma acc kernels
{
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
}// Modify y on host, do not touch x on host
#pragma acc kernels
{
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE

#pragma acc parallel
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
 // Modify y on host, do not touch x on host
#pragma acc parallel
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

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

no *restrict* needed

parallel region + worksharing
Programmer asserts that parallelization is safe

Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
```

PGI Compiler Feedback

```
main:
50, Generating implicit copyin(x[:10240])
    Generating implicit copy(y[:10240])
    Accelerator kernel generated
    Generating Tesla code
52, #pragma acc loop gang, vector(128) /* blockIdx.x
    threadIdx.x */
60, Generating implicit copyin(x[:10240])
    Generating implicit copy(y[:10240])
    Accelerator kernel generated
    Generating Tesla code
62, #pragma acc loop gang, vector(128) /* blockIdx.x
    threadIdx.x */
```

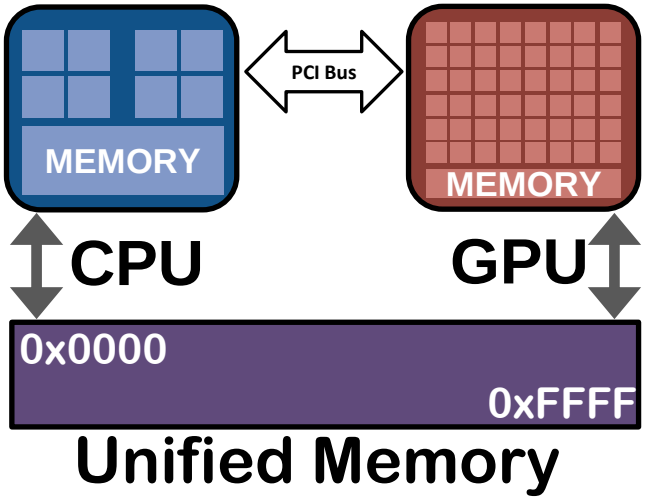
Kernel 1

Kernel 2

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

Managed Memory

- **Next step: Verify that results are correct**
  - → E.g., compare to host version
  - → Don't bother with data movement for now
- → **Managed memory**
  - → Based on Unified Memory
  - → Automatically migrates data to/ from GPU (coherent on kernel launch and sync)



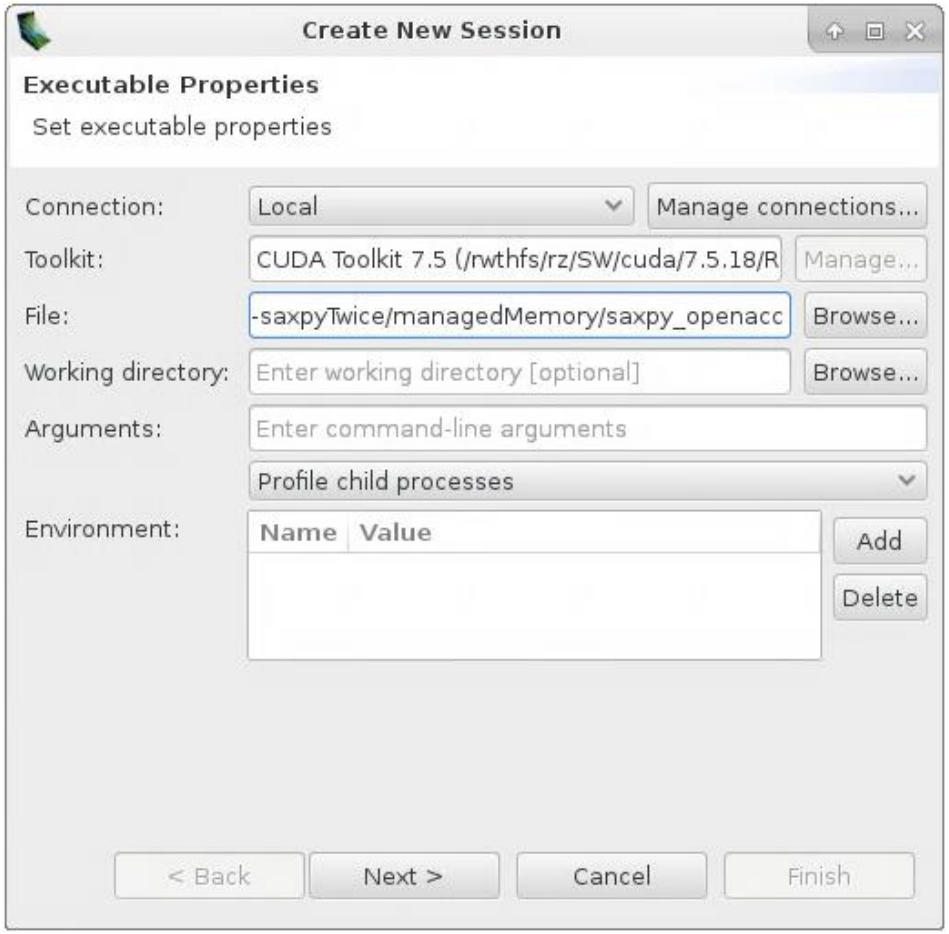
- **Usage**
  - → Enable managed memory: `pgcc -ta=nvidia:managed <src>`
  - → To avoid fallback to zero-copy on multi GPU systems (if not all GPUs are peer-to-peer compatible): `CUDA_MANAGED_FORCE_DEVICE_ALLOC=1`

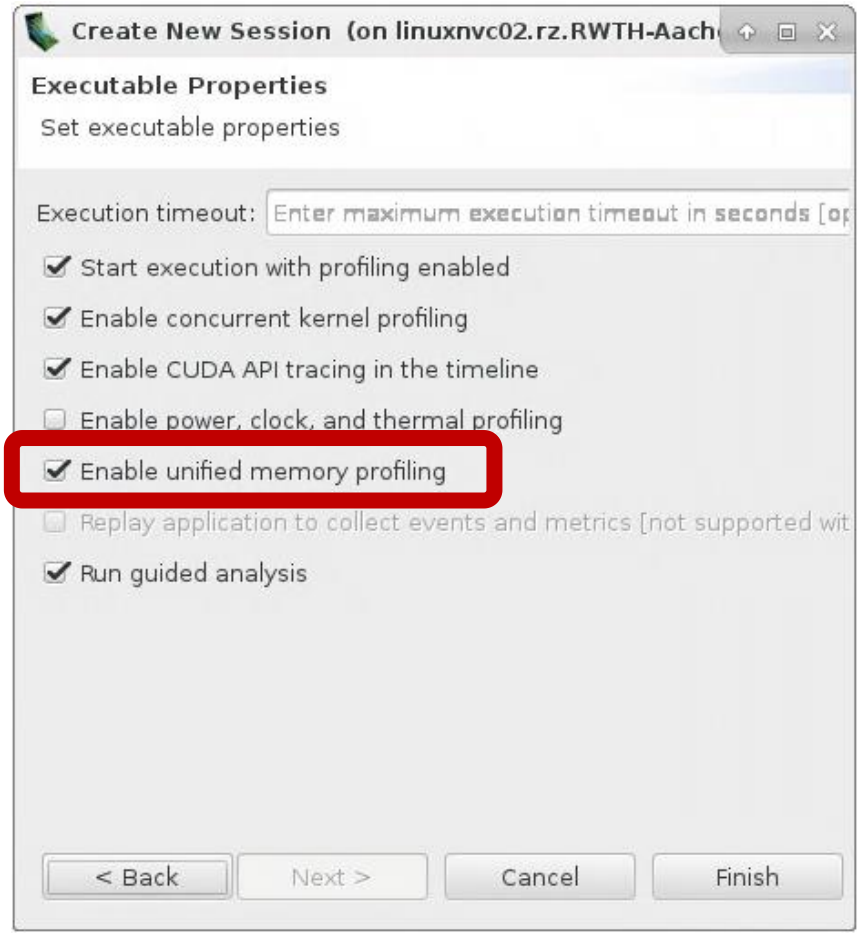
GPUs with CC >= 6.0 have even more possibilities to move data

NVIDIA Profiler

- **GUI: >$ `nvvp`**
- **Command line interface: >$ `nvprof`**

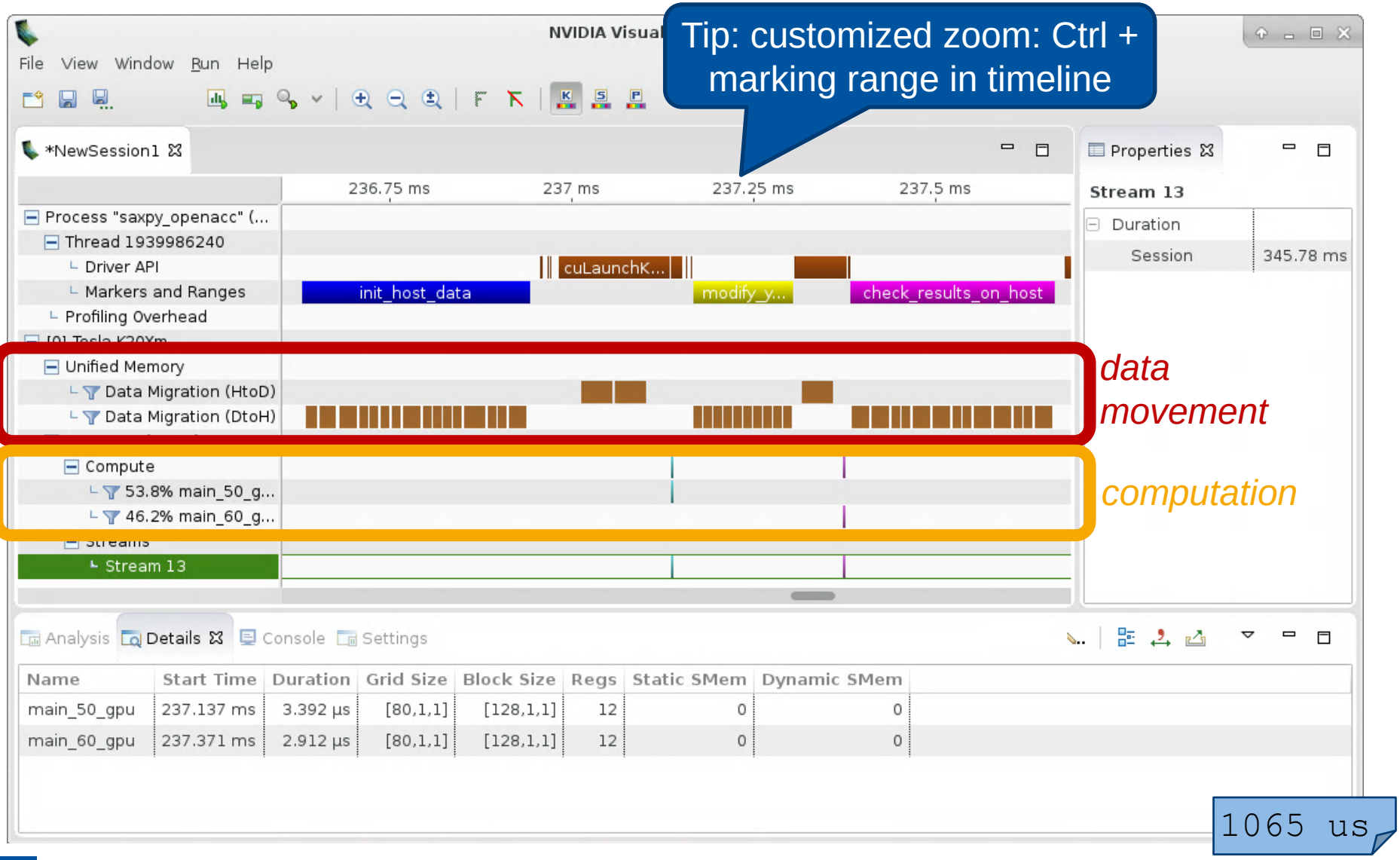
Where is data moved?





Included in CUDA Toolkit

NVIDIA Profiler



Tip: customized zoom: Ctrl + marking range in timeline

data movement

computation

Markers and Ranges were created using NVTX. More details: "CUDA Pro Tip: Generate Custom Application Profile Timelines with NVTX", https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/

Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE

#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
 // Modify y on host, do not touch x on host
#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

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

Also be used with **kernels**

Possible clauses: **copyin**, **copy**, **copyout**, **create**, **present**

Disable managed memory
→ otherwise data clauses will be ignored

Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
```

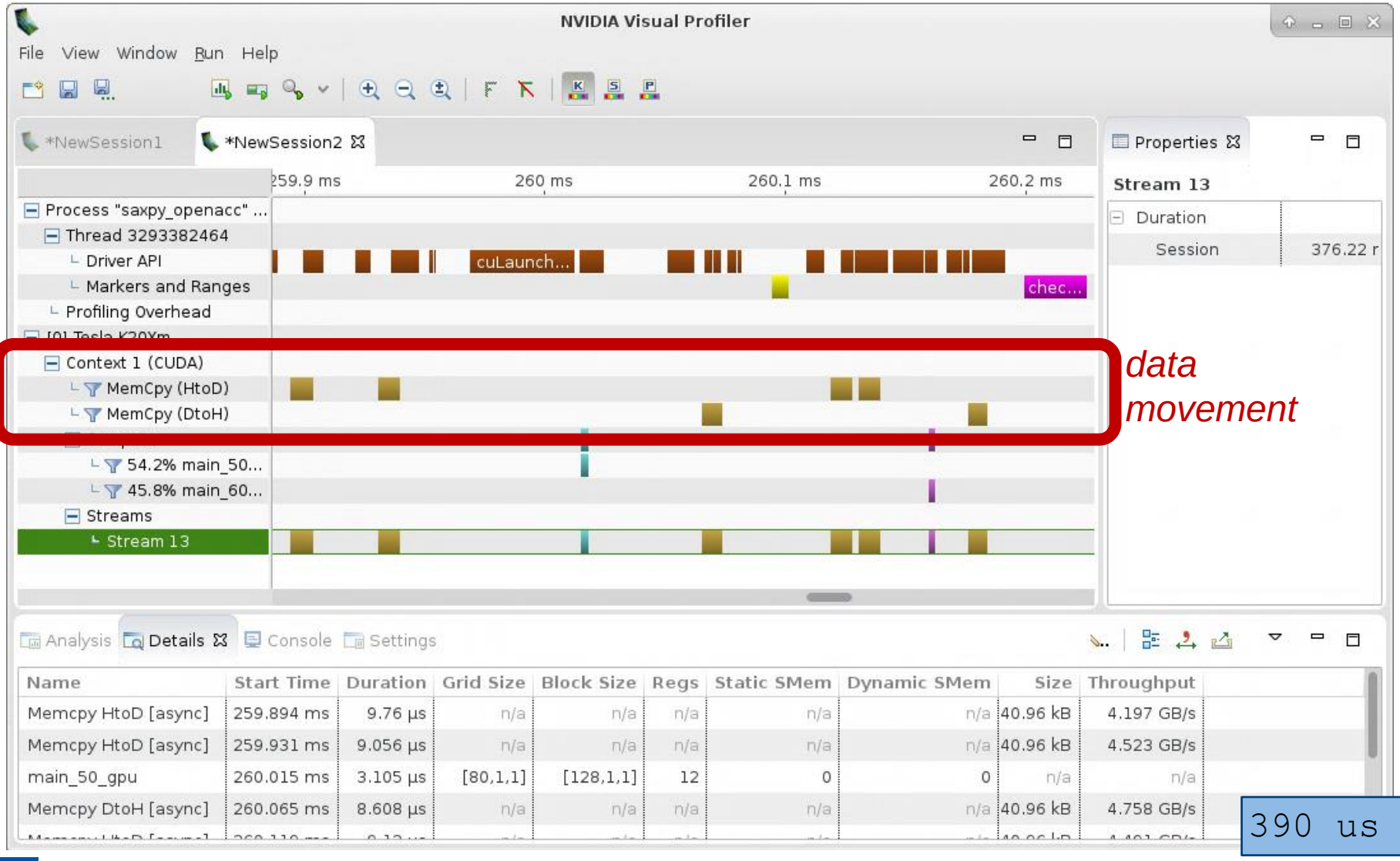
PGI Compiler Feedback

```
main:
50, Generating copyin(x[:n])
    Generating copy(y[:n])
    Accelerator kernel generated
    Generating Tesla code
    52, #pragma acc loop gang, vector(128) /* blockIdx.x
        threadIdx.x */
60, Generating copyin(x[:n])
    Generating copy(y[:n])
    Accelerator kernel generated
    Generating Tesla code
    62, #pragma acc loop gang, vector(128) /* blockIdx.x
        threadIdx.x */
```

Explicit data movement

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

NVIDIA Profiler



Array Shaping

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

Since OpenACC 2.5:
```
copy = present_or_copy = pcopy
copyin = present_or_copyin = pcopyin
copyout = present_or_copyout = pcopyout
```

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

[lower bound: size]

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

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

[lower bound: upper bound]

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

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	`num_gangs(int-expr)`
→ Define number of workers within gang	`num_workers(int-expr)`
→ Define length for vector operations	`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)`

Also: `wait, device_type, default(none|present)`

background 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` (similar to `num_gangs` etc with `parallel`)

- **Loop clauses**
  - → 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)`

C/C++, Fortran

Loop schedule: `gang`, `worker`, `vector`, `seq`

OpenACC 2.0 also: `auto, tile, device_type`

background 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.

Summary

- **Offloading work**
  - → `kernels` – „magic“
  - → `parallel` region & worksharing construct (`loop`)
- **Easy reductions by `reduction` clause**
- **Moving data with data clauses**
  - → Array shaping
  - → `copy`, `copyin`, `copyout`
- **Compiler feedback**
  - → What is done implicitly?
  - → How are explicit directives understood?

Development Tips

- **Iterative development process (basic) + compiler feedback**
  1. Offload compute-intensive kernel(s) to GPU (`kernels`, `parallel loop`)
  2. Enable managed memory (if data movement is not working out of the box) for automatic data transfers (`managed` compiler flag)
  3. Run your code and verify that your results are correct (execution might be very slow)
  4. Use profiler to figure out where data was (automatically) moved
  5. Implement data movement manually correspondingly (`copy`, `copyin`, `copyout`, `data`, `enter data`, `exit data`)
     → Disable managed memory (if enabled, all data clauses will be ignored)
  6. Optimize data movement step by step (moving data copying out → soon)
     → You can nest data clauses: inner data clauses will not trigger data movement if data is present on GPU

Development Tips

- **Favorable for parallelization: (nested) loops**
  - → Large loop counts to compensate (at least) data movement overhead
  - → Independent loop iterations
- **Think about data availability on host/ GPU**
  - → Use data regions to avoid unnecessary data transfers
  - → Specify data array shaping (may adjust for alignment)
- **Verifying execution on GPU**
  - → See PGI compiler feedback. Term "`Accelerator kernel generated`" needed.
  - → See information on runtime (more details in the hands-on session):

    `export ACC_NOTIFY=3`

Development Tips

- **Conditional compilation for OpenACC**
  - → Macro `_OPENACC`
- **Using pointers (C/C++)**
  - → Problem: compiler can not determine whether loop iterations that access pointers are independent → no parallelization possible
  - → Solution (if independence is actually there)
    - → Use restrict keyword: `float *restrict ptr;`
    - → Use compiler flag: `-ansi-alias` (PGI)
    - → Use OpenACC loop clause: `independent`
  - → Prefer array notation over pointer arithmetic: `array[2]` instead of `*(array+2)`

Development Tips

- **Runtime measurements**
  - → GPU needs time to initialize (up to a couple of seconds, depending on GPU)
    - → Might get included into runtime measurements & influence short runtimes
  - → Exclude GPU initialization from measurements by calling

    `acc_init(acc_device_nvidia)`, `#pragma acc init(acc_device_nvidia)`
  - → Simple profile (that also shows the init time) by
    - → Compiler flag: `-ta=nvidia,`**`time`**
    - → Environment variable: **`PGI_ACC_TIME=1`**
- **Setup information**
  - → Information on GPU type, hardware details and configuration
  - → NVIDIA: Download CUDA SDK, run `deviceQuery`
  - → PGI: `pgaccelinfo` (also specifies needed compiler flags for OpenACC)

RWTH Environment – GPUs

- **NVIDIA Fermi**
  - → 28 nodes with each

    2 NVIDIA Quadro 6000 GPUs

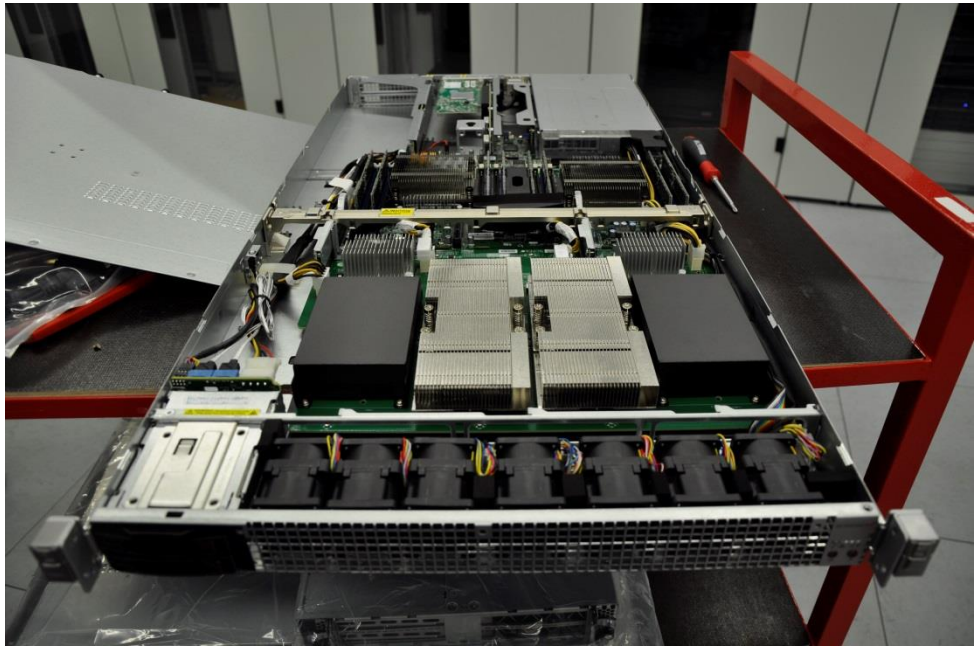
    → VR + HPC
- **NVIDIA Kepler**
  - → 2 nodes with each

    2 NVIDIA K20X GPUs
- **NVIDIA Pascal**
  - → 10 nodes with each

    2 NVIDIA P100 SXM2

    with 16GB
  - → NVLink between 2 GPUs

aixCAVE, VR, RWTH Aachen, since June 2012



server with two NVIDIA Pacal P100 GPUs, 2016

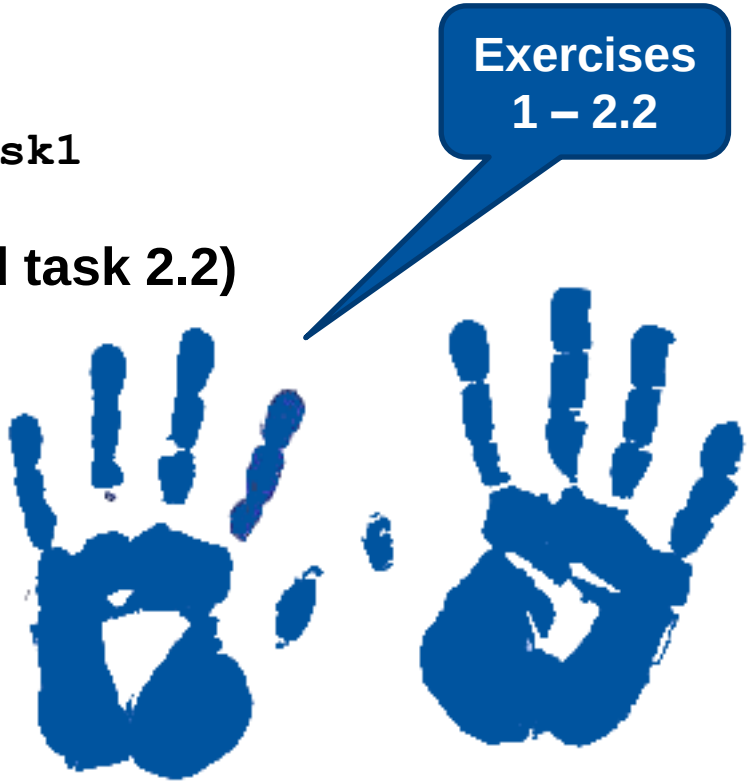
Hands-On

- **Setting up your GPU cluster environment**

- **Write your first OpenACC program (Jacobi solver)**
  - → Use `parallel` and `loop` directives
  - → Follow TODOs in `GPU/exercises/task1`

- **<u>Optional</u> (only if you have completed task 2.2)**
  - → Read slide 106 (pinned memory)
  - → Apply this to your program from task 2.2
  - → Investigate performance diffs

- **<u>Optional</u>**
  - → Work on task 2.9 (roofline model)



Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.19
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	12.36
	OpenACC-Data	
	OpenACC-Collapse	
	OpenACC-Hetero *(1 GPU + 12 OMP threads)*	
	OpenACC-MultiGPU *(2 GPUs + 12 OMP threads)*	

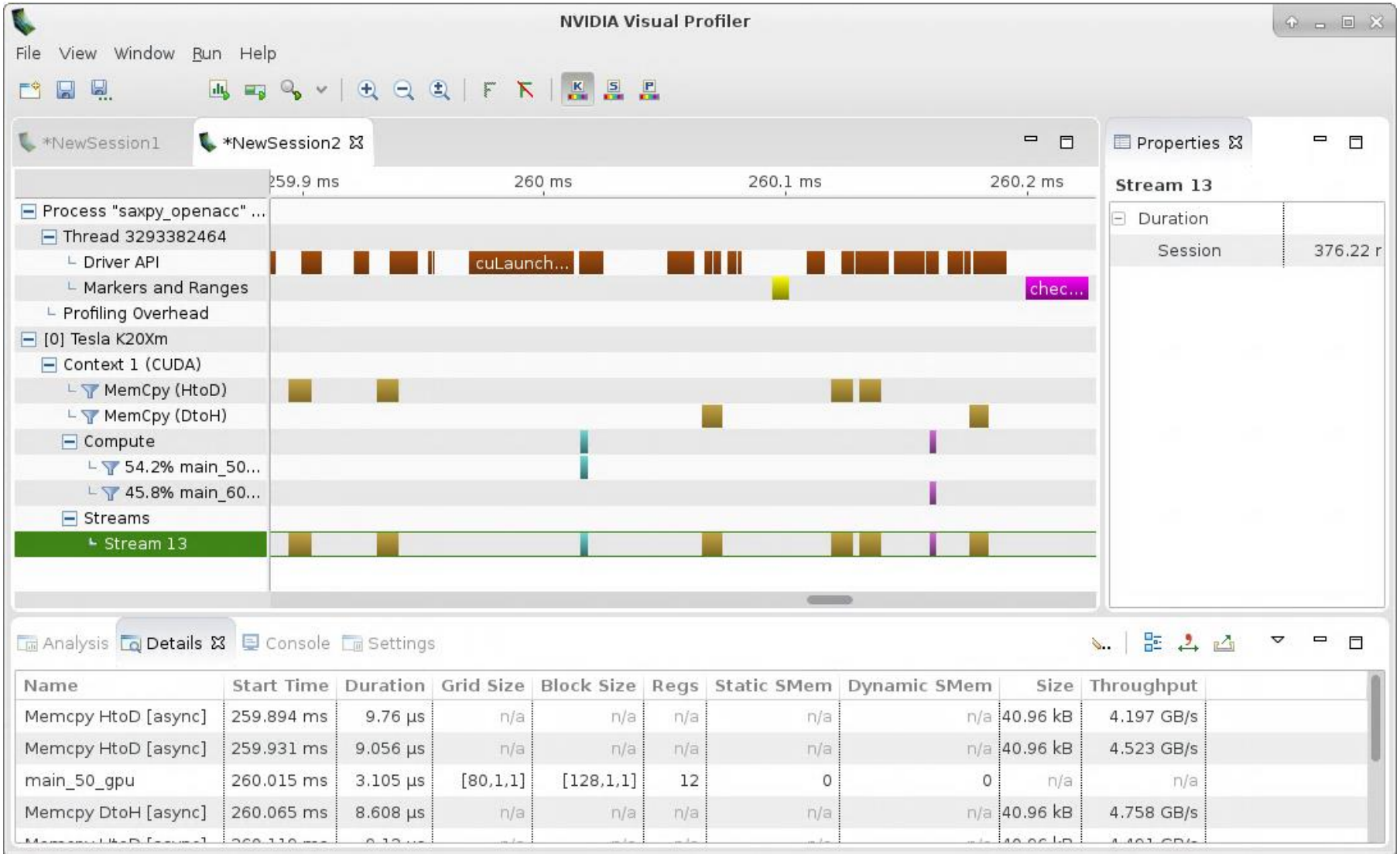
Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

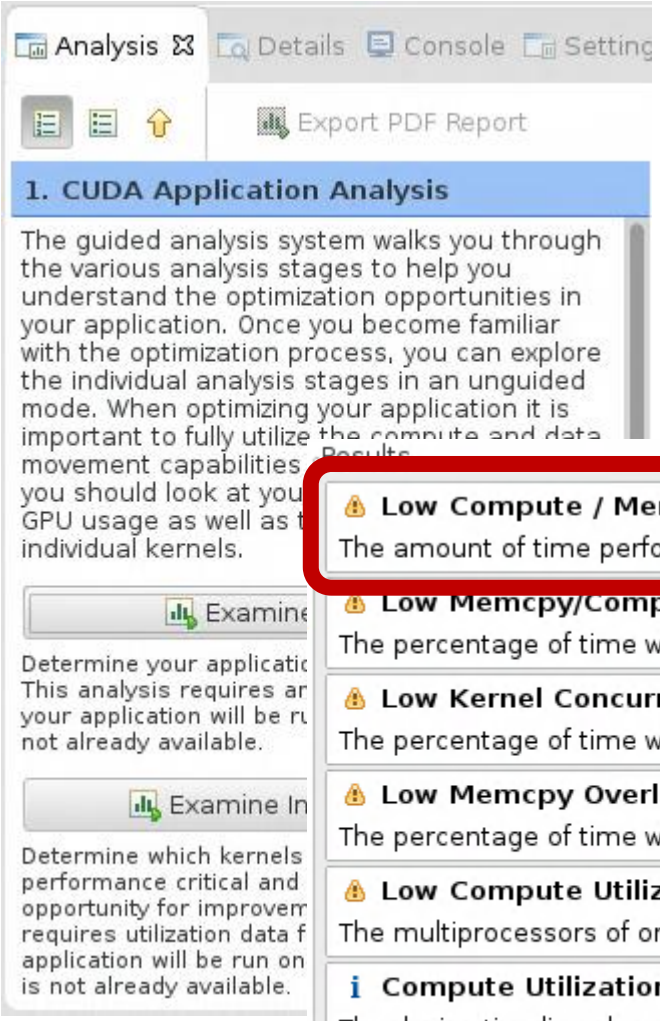
Recap: Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE



#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
 // Modify y on host, do not touch x on host
#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

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

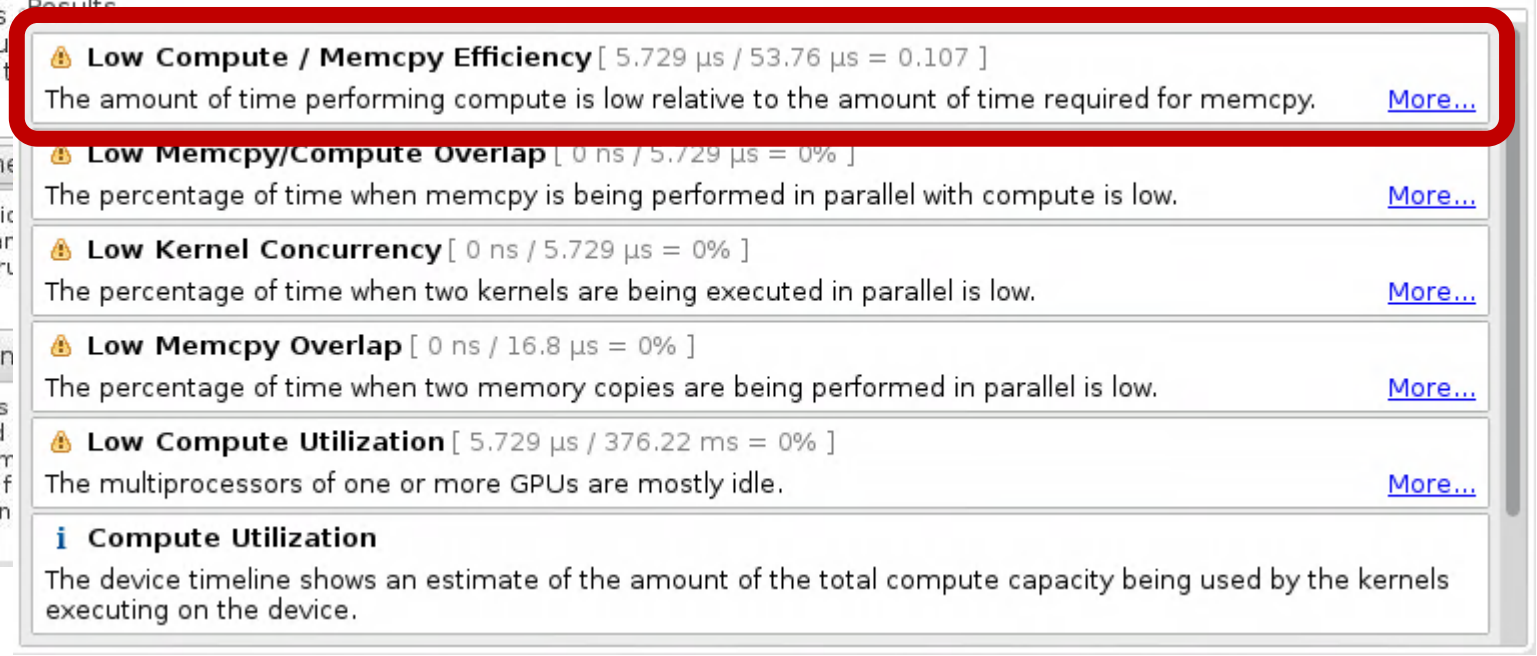
Recap: NVIDIA Profiler



NVIDIA Profiler



- **Guided analysis**
  - → GPU usage
  - → Single kernel analysis
- **Includes tips for optimization and reading material**



Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE



#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
 // Modify y on host, do not touch x on host
#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

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

Directives (in examples): SAXPY
acc data

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc parallel copy(y[0:n]) present(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
 // Modify y on host, do not touch x on host
#pragma acc parallel copy(y[0:n]) present(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

x remains on the GPU until the end of the data region.

Directives (in examples): SAXPY
acc data

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc parallel c
#pragma acc loop
  for (int i = 0; i < n; -
        y[i] = a*x[i] + y
  }
 // Modify y on host, do r
#pragma acc parallel c
#pragma acc loop
  for (int i = 0; i < n; -
        y[i] = b*x[i] + y
  }
}
  free(x); free(y); return
}
```

x remains on the GPU until the end of the data region.

PGI Compiler Feedback

```
main:
48, Generating copyin(x[:n])
50, Generating copy(y[:n])
    Generating present(x[:n])
    Accelerator kernel generated
    Generating Tesla code
    52, #pragma acc loop gang, vector(128)
60, Generating copy(y[:n])
    Generating present(x[:n])
    Accelerator kernel generated
    Generating Tesla code
    62, #pragma acc loop gang, vector(128)
```

380 us

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
  - → If *cond* true, move data to accelerator……
  - → H2D-copy at region start + D2H at region end ……
  - → Only H2D-copy at region start ……
  - → Only D2H-copy at region end ……
  - → Allocates data on device, no copy to/from host ……
  - → Data is already on device ……
  - → Test whether data on device. If not, transfer……
  - → See Tuning slides……

C/C++, Fortran

```
if(cond)
copy(list)
copyin(list)
copyout(list)
create(list)
present(list)
present_or_*(list)
deviceptr(list)
```


background OpenACC

Data Movement

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

Since OpenACC 2.5:
```
copy = present_or_copy = pcopy
copyin = present_or_copyin = pcopyin
copyout = present_or_copyout = pcopyout
```

- **`copy` (also `copyin`, `copyout`)**
  - → Checks whether data on device, if not: copies data
- **`create`**
  - → Does not copy any data, just creates it on the device
- **`deviceptr`**
  - → See section “Interoperability with CUDA & GPU Libraries”

Directives (in examples)
acc update

```
#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]
```

  - → Data movement can be conditional or asynchronous

- **Update clauses**
  - → *list* variables are copied from acc to host……
  - → *list* variables are copied from host to acc……..
  - → If *cond* true, move data to accelerator……
  - → Issues no error when data is not present on device…
  - → Executes async, see Tuning slides……
  - → Data movement waits on actions in queue to finish….
  - → Set certain device type for update……

C/C++, Fortran
`host|self(list)`
`device(list)`
`if(cond)`
`if_present`
`async[(int-expr)]`
`wait[(int-expr)]`
`device_type(list)`

background OpenACC

Unstructured Data Lifetime

- **Structured data lifetime**
  - → Applied to a region

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

- **Unstructured data lifetime**
  - → **`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 copyin(this)
  #pragma acc enter data create(v[0:n])
 }
 ~Matrix() {
  #pragma acc exit data delete(v[0:n], this)
  delete[] v;
 }
 private:
  double* v;
}
```

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)
```

- **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)
finalize
```

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

background OpenACC

Summary

- **Optimizing code with explicit data transfers**
  - → Structured: `data` region
  - → Unstructured: `enter data`, `exit data`
  - → Within data environment: `update`
  - → Clauses: `copy`, `copyin`, `copyout`,…
- **Compiler and Runtime feedback**
  - → Where is the data moved?
  - → How much time is spent with data movement?
- **Using NVIDIA Visual Profiler with OpenACC**

Hands-On

- **Data motion optimization of your first OpenACC program**
  - → Use structured **`data`** regions to minimize data movement across the PCI bus
  - → Use **`parallel`** and **`loop`** directives on all loops that read or write the data within the data regions
  - → Follow TODOs in **`GPU/exercises/task2`**
- **<u>Optional</u> (only if you have completed task 2.4)**
  - → Work on task 2.8 (eliminating data swapping)
- **<u>Optional</u>**
  - → If not already done, work on task 2.9 (roofline model)

Exercises 2.3 – 2.4

Lab Results

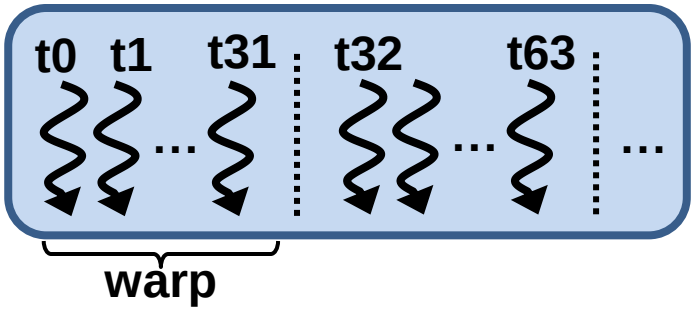
Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.19
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	12.36
	OpenACC-Data	1.21
	OpenACC-Collapse	
	OpenACC-Hetero *(1 GPU + 12 OMP threads)*	
	OpenACC-MultiGPU *(2 GPUs + 12 OMP threads)*	

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Latency hiding

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



- → **SIMT architecture**
  - → SIMT = single instruction, multiple threads
- **Hardware operation**
  - → Instructions are issued in order
  - → Thread execution stalls when one of the operands isn't ready
  - → GMEM latency: ~300 cycles (Kepler)
  - → Arithmetic latency: ~9 cycles (Kepler)

→ **Main concept: hide latencies**

→ **Switching threads** (need enough parallelism)

→ **Independent instructions** from same thread/ warp

Latency hiding

- **Hiding arithmetic latency**
  - → Need ~9 warps (288 threads) per SM
  - → Or, independent instructions from the same warp

```
x = a + b;//~9 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) —e.g.→
    - → Loads and stores from different threads (many threads)
    - → Larger word sizes can also help (“vectors”)

```
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

Occupancy

- **How much "parallelism"?**
- **Occupancy (per SM) =**

$$\frac{\text{active warps}}{\text{max. supported active warps}}$$

- **Occupancy limiters**
  - → E.g., unbalanced workloads
  - → Shared resources: registers, shared memory
  - → Device capabilities: thread blocks, warps

Technical Specifications	Compute Capability 2.x	3.5	6
Maximum x-dimension of a grid of thread blocks	65535	2^{31}-1	
Maximum x- or y-dimension of a block	1024		
Maximum number of threads per block	1024		
Maximum number of resident blocks per multiprocessor	8	16	32
Maximum number of resident warps per multiprocessor	48	64	
Maximum number of resident threads per multiprocessor	1536	2048	
Number of 32-bit registers per multiprocessor	32 K	64 K	64 K
Maximum number of 32-bit registers per thread block	32 K	64 K	64 K
Maximum number of 32-bit registers per thread	63	255	
Maximum amount of shared memory per multiprocessor	48 KB		64 KB
Maximum amount of shared memory per thread block	48 KB		

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

Occupancy

Examples for Fermi (cc2.0): max 1536 threads per SM

- **Example: Registers (32K per SM)**
  - → Kernel uses 64 registers per thread
  - → Active threads: 32K/64 = 512 threads
  - → 512/1536 = 0.33 occupancy

```
pgcc -ta=nvidia,maxregcount:<n>
```

- **Example: Shared memory (16K vs. 48K)**
  - → Kernel uses 32 Bytes of SMEM per thread
  - → Active threads: 16K/32 = 512 threads
  - → 512/1536 = 0.33 occupancy (occupancy of 1 for 48K)
- **Example: Block Size (8 active blocks per SM)**
  - → Choose appropriate launch configuration

Block Size	Active Threads	Occupancy
32	256	0.166
64	512	0.333
128	1024	0.666
192	1536	1
256	2048 (1536)	1

Occupancy

- **Low occupancy: indication for poor instruction issue efficiency**
  - → Not able to hide latencies
- **Needed occupancy depends on code**
  - → 100% is not necessarily needed for max performance
  - → Rule of thumb: aim for 66% occupancy
- **Hints**
  - → More independent work per thread → less occupancy is needed
  - → Memory-bound codes tend to need more occupancy
  - → E.g., several simultaneous kernel launches can increase utilization
- **NVIDIA Profiler gives information on occupancy**

Summary

- **GPU does not apply control logic**
  - → Waiting on arithmetic operations or memory accesses cause stalls
- **Latency hiding by switching threads**
  - → Needs enough parallelism
- **Occupancy as indicator for “enough parallelism”**
  - → Limiters could be: registers, shared memory, thread blocks

Hands-On

- **Investigation of occupancy**
  - → Use the NVIDIA Profiler's Guided Analysis to investigate performance limiters of the code
  - → **`Collapse`** nested loops
  - → Follow TODOs in **`GPU/exercises/task3`**

- **<u>Optional</u>**
  - → Read slides 97 – 105 (Launch Configuration & Loop Schedules)
  - → Apply different launch configurations and loop schedules
  - → Investigate performance differences in the NVIDIA Profiler

Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.19
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	12.36
	OpenACC-Data	1.21
	OpenACC-Collapse	1.16
	OpenACC-Hetero *(1 GPU + 12 OMP threads)*	
	OpenACC-MultiGPU *(2 GPUs + 12 OMP threads)*	

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Launch configuration

- **Recap: Threads per block**

Block Size	Active Threads	Occupancy
32	256	0.166
64	512	0.333
128	1024	0.666
192	1536	1
256	2048 (1536)	1

- **How many threads/ blocks to launch?**
  - → OpenACC runtime tries to find good values automatically
    - → Performance portability across different device types possible
  - → Own tuning may deliver better performance on a certain hardware
- **Possible mapping to CUDA terminology (GPUs)** compiler-dependent

block → gang

```
num_gangs(N)
```

warp → worker

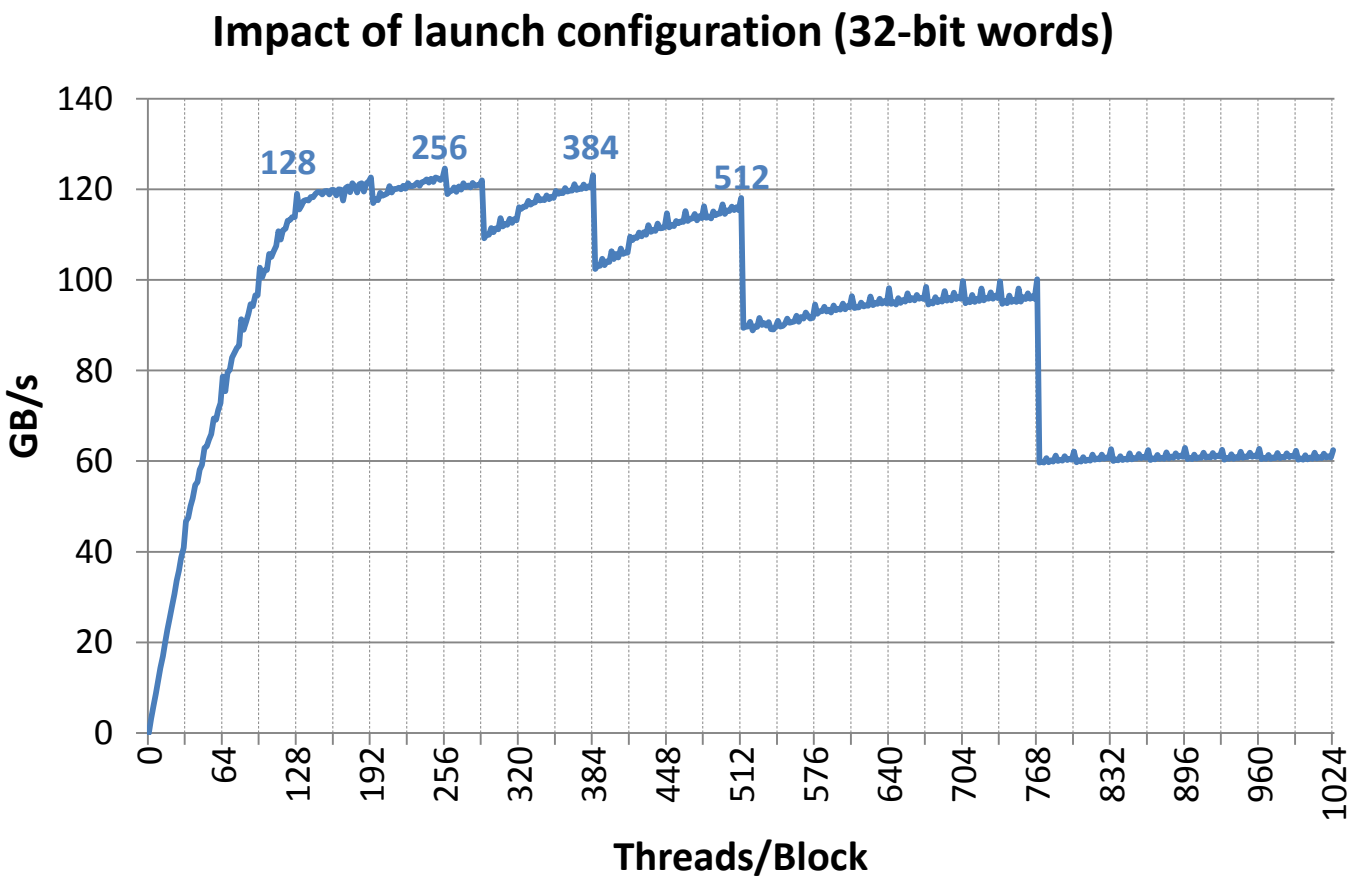
```
num_workers(N)
```

threads → vector

```
vector_length(N)
```

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

- **Vectors per gang: Multiple of warp size (32)**
  - → Threads are always spanned in warps onto resources
  - → Not used threads are marked as inactive
  - → Starting point: 128-256 vectors/gang
- **Blocks per grid heuristics**
  - → **`#blocks > #SMs`**
    - → MPs have at least one block to execute
  - → **`#blocks / #SMs > 2`**
    - → MP can concurrently execute up to 16 blocks
    - → Blocks that aren't waiting in barrier keep hardware busy
    - → Subject to resource availability (registers, smem)
  - → **`#blocks > 100`** 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!**

Loop schedule *(in examples)*

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc kernels copy(y[0:n]) present(x[0:n])
#pragma acc loop gang vector(256)
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
 // Modify y on host, do not touch x on host
#pragma acc kernels copy(y[0:n]) present(x
#pragma acc loop gang(64) vector
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

Without loop schedule

```
33, Loop is parallelizable
    Accelerator kernel generated
    33, #pragma acc loop gang,
    vector(128)
39, Loop is parallelizable
    Accelerator kernel generated
    39, #pragma acc loop gang,
    vector(128)
```

With loop schedule

```
33, Loop is parallelizable
    Accelerator kernel generated
    33, #pragma acc loop gang,
    vector(256)
39, Loop is parallelizable
    Accelerator kernel generated
    39, #pragma acc loop
    gang(64), vector(128)
```

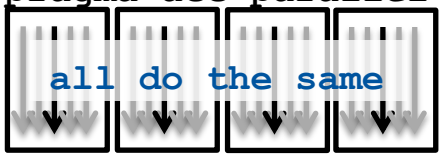
Loop schedule *(in examples)*

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc parallel copy(y[0:n]) present(x[0:n]) vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
 // Modify y on host, do not touch x on host
#pragma acc parallel copy(y[0:n]) present(x[0:n]) num_gangs(64)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

Can both be used with **`parallel`** or **`kernels`**:
`vector_length`: Specifies number of threads in thread block.
`num_gangs`: Specifies number of thread blocks in grid.

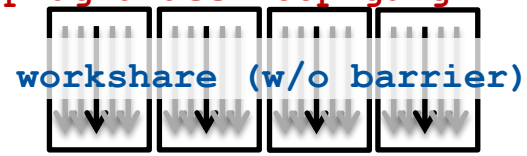
Loop schedule *(in examples)*

```
int main(int argc, const char* argv[]) {
  int n = 256; int blocks = 4; int bsize = 32;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars a, b & initialize x, y
  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{
#pragma acc parallel copy(y[0:n]) present(x[0:n]) \
                                  vector_length(bsize) num_gangs(blocks)
```

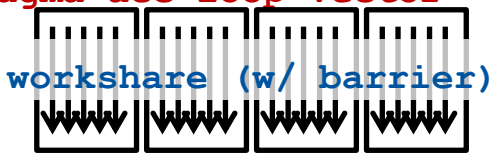


e.g. int tmp = 5;

```
#pragma acc loop gang
```

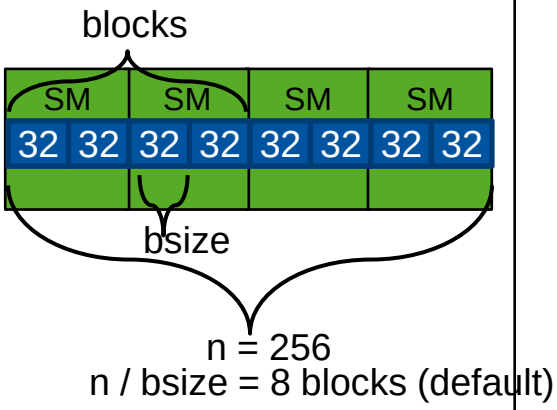



```
#pragma acc loop vector
```




```
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  } // Change y on host & do second SAXPY on device
}
  free(x); free(y); return 0; }
```

gang vector:
strip-mine iteration space



Note: Loop schedules are implementation dependent. This representation is one (common) example.

Loop schedule *(in examples)*

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
       // do something
  }
```

Distributes loop to **n** threads on GPU.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
     for (int j = 0; j < m; ++j){
          // do something
  }  }
```

Distributes outer loop to **n** threads on GPU.
Each thread executes inner loop sequentially.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

```
#pragma acc parallel vector_length(256) num_gangs(16)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
       // do something
  }
```

Distributes loop to threads on GPU (see above). If 16*256 < n, each thread gets multiple elements.
→256 threads per thread block
→16 blocks in grid

Loop schedule *(in examples)*

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang
  for (int i = 0; i < n; ++i){
#pragma acc loop vector
      for (int j = 0; j < m; ++j){
            // do something
  }  }
```

May maximize parallelism (alternative: **`collapse`**)

Distributes outer loop to GPU multiprocessors (block-wise). Distributes inner loop to threads within thread blocks.

→256 threads per thread block

→usually **`n`** blocks in grid

```
#pragma acc kernels
#pragma acc loop gang(100) vector(8)
  for (int i = 0; i < n; ++i){
#pragma acc loop gang(200) vector(32)
      for (int j = 0; j < m; ++j){
            // do something
  }  }
```

Multidimensional partioning with **`parallel`** construct only possible in v2.0.→ See **`tile`** clause

With nested loops, specification of multidimensional blocks and grids possible: use same resource for outer and inner loop.

→100 blocks in Y-direction (rows); 200 blocks in X-direction (columns)

→8 threads in Y-dimension of one block; 32 threads in X-dimension of one block

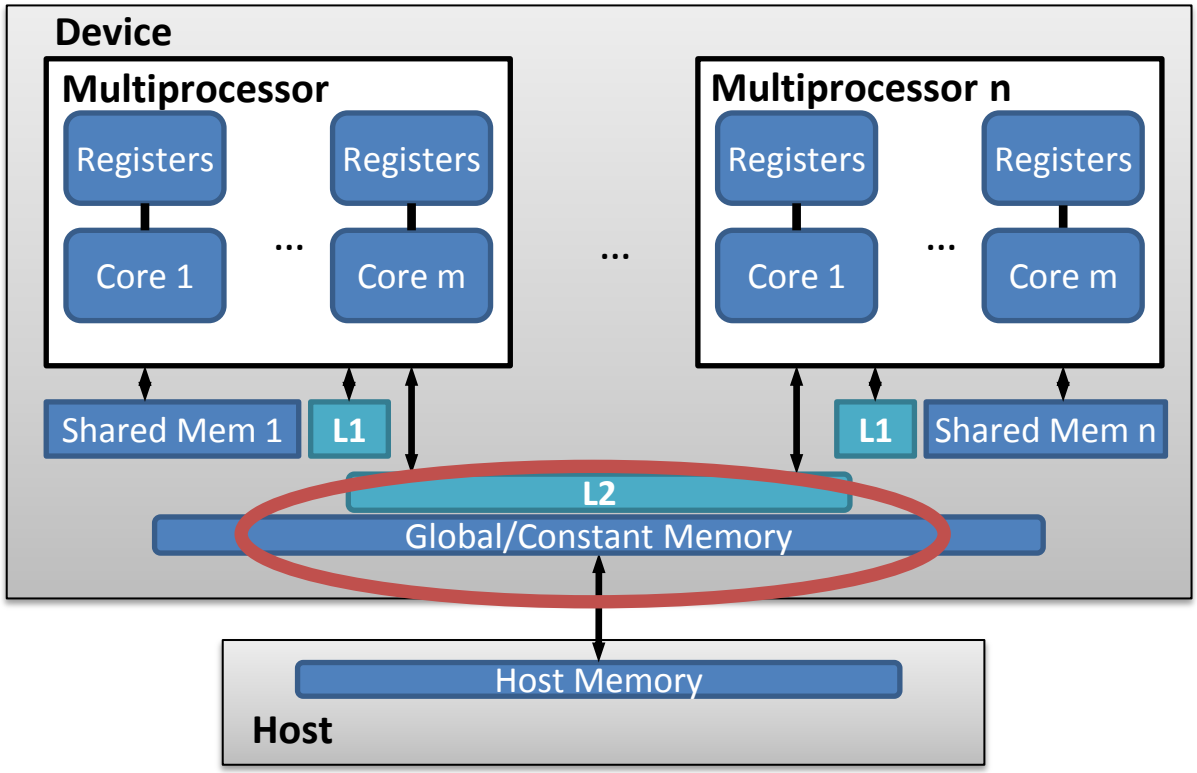
→Total: 8*32=256 threads per thread block; 100*200=20,000 blocks in grid

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

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)



Pinned Memory

- **CPU default: data allocations are pageable**
  - → Use of virtual memory: physical memory pages can be swapped out to disk
    - → When host needs page, it loads it back in from the disk
  - → No direct access to pageable memory by GPU possible
    1. CUDA driver allocates a temporary page-locked (“pinned”) host array
    2. Copies host data to pinned array (PCIe transfers occur only using DMA)
    3. Transfer data from pinned array to device
    4. Delete pinned memory after transfer completion
- **Pinned memory = page-locked memory**
  - → Prevents memory from being swapped out to disk
  - → Avoids cost of transferring between pageable and pinned host arrays
  - → Improves transfer speeds
  - → But: reduces amount of physical memory available to OS & other programs

```
pgcc -ta=nvidia:pinned saxpy.c
```

It’s just a compiler flag
→ Try it out!

gmem access

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

	Caching	Non-caching
Enabling by	default (Fermi)	Compiler options, e.g. PGI's loadcache
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

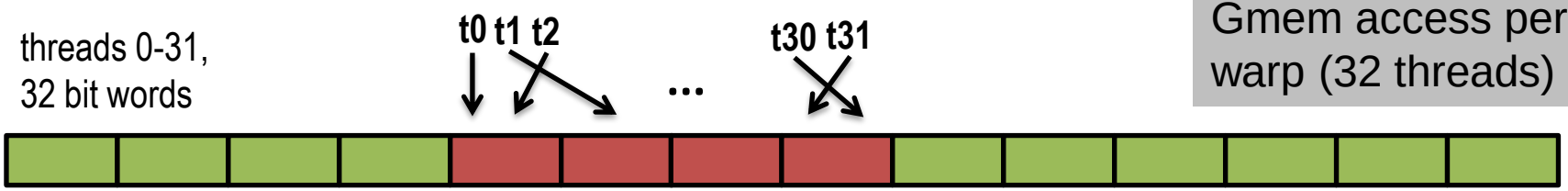
Kepler uses L1 cache only for thread-private data.

→ Threads in a warp provide memory addresses

→ Determine which lines/segments are needed

→ Request the needed lines/segments

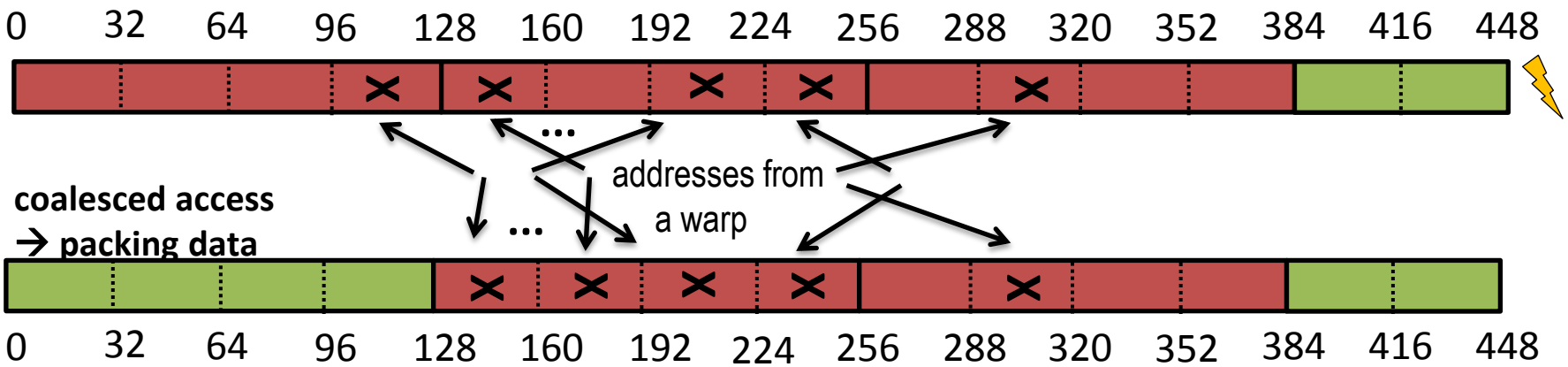
Gmem access per warp (32 threads)



gmem throughput – coalescing

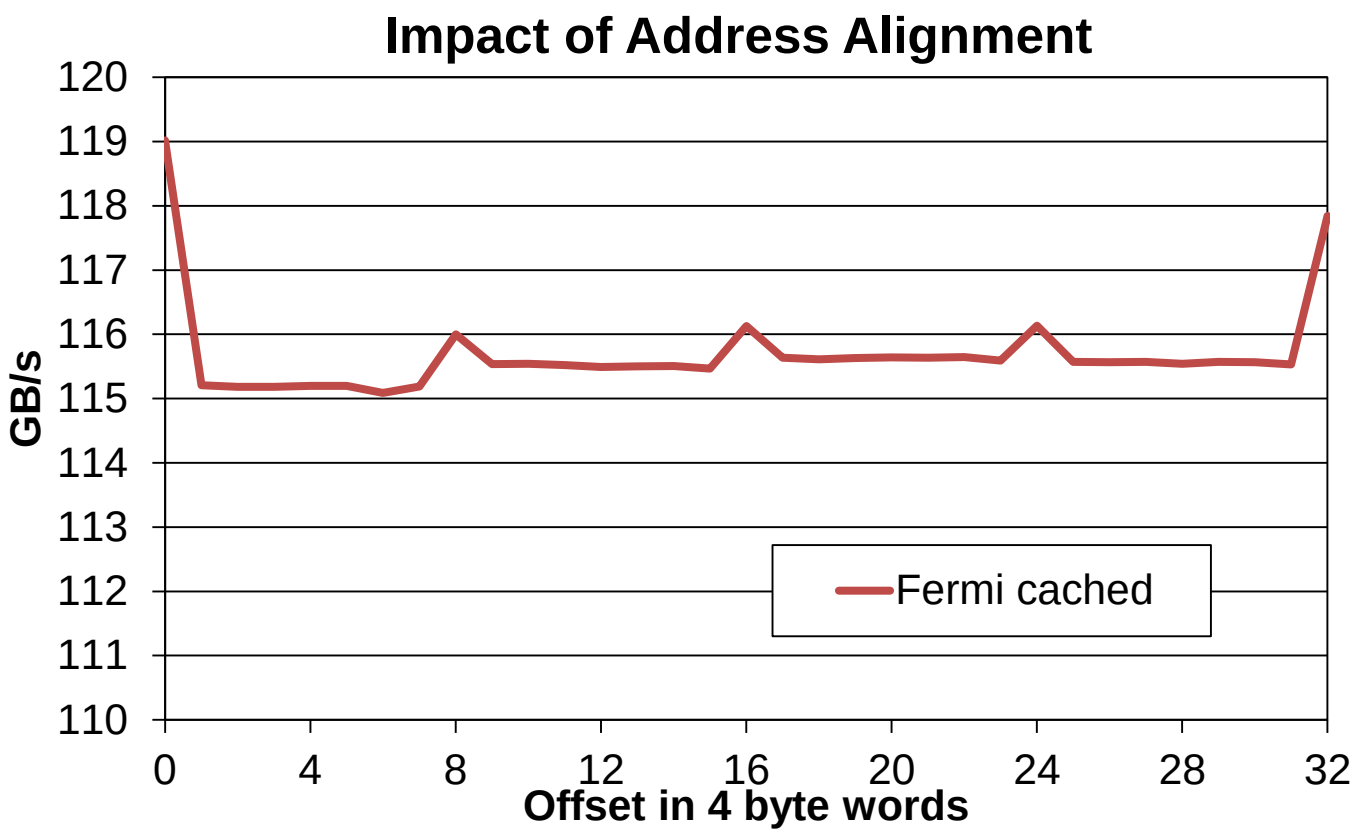
- **Range of accesses**

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[];
}
```

```
#pragma acc kernels
for (int i=0; i<n; i++) {
  … myData[i].a / myData.a[i] …
}
```

Address	0	4	8	12	16	20	24	28	32
AoS	A[0]	B[0]	C[0]	A[1]	B[1]	C[1]	A[2]	B[2]	C[2]
SoA	A[0]	A[1]	A[2]	B[0]	B[1]	B[2]	C[0]	C[1]	C[2]

Summary

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

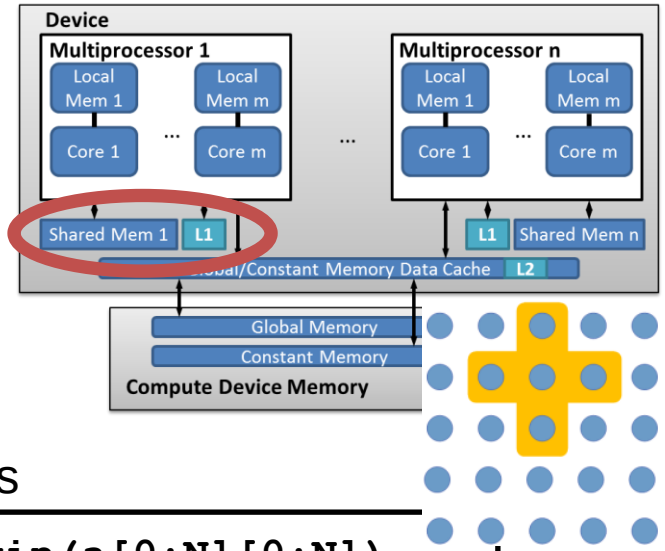
- **Have enough concurrent accesses to saturate the bus**
  - → Process several elements per thread
  - → Launch enough threads to cover access latency

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Shared memory

- **Smem per SM(X) (10s of KB)**
  - → Inter-thread communication within a block
  - → Need synchronization to avoid RAW / WAR / WAW hazards
- **Low-latency, high-throughput memory**
  - → Cache data in smem to reduce gmem accesses




```
#pragma acc kernels copyout(b[0:N][0:N]) copyin(a[0:N][0:N])
#pragma acc loop gang vector
  for (int i = 1; i < N-1; ++i){
#pragma acc loop gang vector
   for (int j = 1; j < N-1; ++j){
#pragma acc cache(a[i-1:3][j-1:3])
      b[i][j] = (a[i][j-1] + a[i][j+1] +
                 a[i-1][j] + a[i+1][j]) / 4;
  }    }
```

```
Accelerator kernel generated
65, #pragma acc loop gang, vector(2)
    Cached references to size
    [(y+2)x(x+2)] block of 'a'
67, #pragma acc loop gang, vector(128)
```

The `tile` clause can additionally strip-mine the data space.

Directives: Caching & Strip-mining

- **Cache construct**
  - → Prioritizes data for placement in the highest level of data cache on GPU

C/C++
```
#pragma acc cache(list)
```

Fortran
```
!$acc cache(list)
```

Sometimes the PGI compiler ignores the `cache` construct.
→ See compiler feedback

  - → Use at the beginning of the loop or in front of the loop

- **Tile clause**
  - → Strip-mines each loop in a loop nest according to the values in *expr-list*

C/C++
```
#pragma acc loop [<schedule>] tile(expr-list)
```

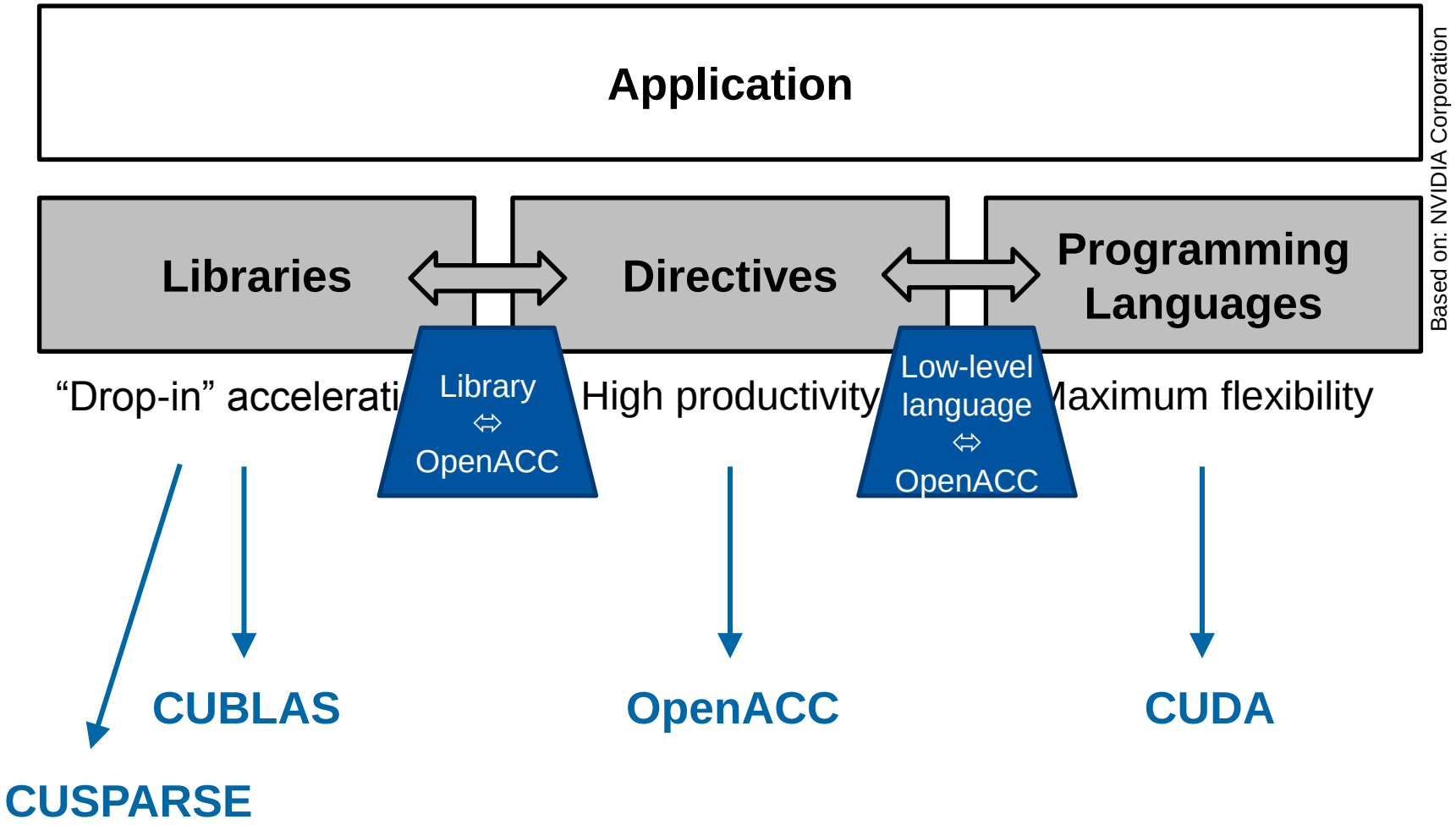
Fortran
```
!$acc loop [<schedule>] tile(expr-list)
```

1. entry applies to innermost loop

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Interoperability with Libraries and CUDA



Interoperability

- **CUDA and OpenACC both operate on device memory**
  - → Interoperability with (CUDA) libraries
  - → Interoperability with custom (CUDA) C/C++/Fortran code

→ **Make device address available on host or in OpenACC code**

```
#pragma acc data copy(x[0:n])
{
#pragma acc kernels loop
  for (int i=0; i<n; i++) {
    // work on x[i]
  }
#pragma acc host_data use_device(x)
  {
    func(x);
    // ..
} }
```

use device pointer x:
- for library
- for custom CUDA code

```
cudaMalloc(&x, sizeof(float)*n);
// use device pointer x:
// - for library
// - for custom CUDA code


#pragma acc data deviceptr(x)
{
#pragma acc kernels loop
  for (int i=0; i<n; i++) {
    // work on x[i]
} }
```

assume x is pointer to device memory

Directives: deviceptr & host_data

- **Deviceptr clause**
  - → Declares that pointers in *list* refer to device pointers that need not be allocated/ moved between host and device

C/C++

```
#pragma acc parallel|kernels|data deviceptr(list)
```

Fortran

```
!$acc data deviceptr(list)
```

- **Host_data construct**
  - → Makes the address of device data in *list* available on the host
  - → Vars in *list* must be present in device memory

C/C++

```
#pragma acc host_data use_device(list)
```

only valid clause

Fortran

```
!$acc host_data use_device(list)

!$acc end host_data
```


background OpenACC

Summary

- **OpenACC is interoperable with CUDA and GPU Libraries**
  - → **`host_data`**: device address available on host
  - → **`deviceptr`**: device address available in OpenACC code

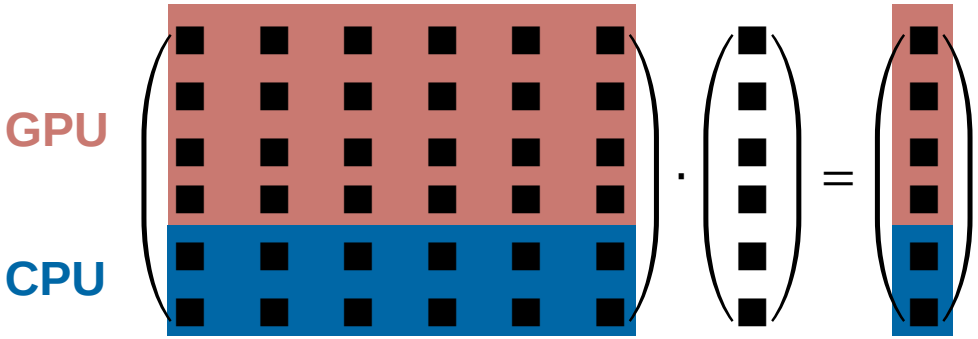
Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Heterogeneous Computing

- **Heterogeneous Computing**
  - → CPU & GPU are (fully) utilized
- **Challenge: load balancing**
- **Domain decomposition**
  - → If load is known beforehand, static decomposition
  - → Exchange data if needed (e.g. halos)

matrix vector multiplication



Asynchronous operations

- **Definition**
  - → Synchronous: Control does not return until accelerator action is complete
  - → Asynchronous: Control returns immediately
    - → Allows heterogeneous computing (CPU + GPU)
- **Default: synchronous operations**
- **Asynchronous operations with `async` clause**
  - → Kernel execution: `kernels`, `parallel`
  - → Data movement: `update`, `enter data`, `exit data`

```
#pragma acc kernels async
for (int i=0; i<n; i++) {…}

// do work on host
```

Heterogenity

The `wait` directive can also take an `async` clause.

Directives: async clause

Async clause

→ Executes **`parallel`** | **`kernels`** | **`update|enter data|exit data|wait`** operations asynchronously while host process continues with code

C/C++

```
#pragma acc <op-see-above> async[(scalar-int-expr)]
```

Fortran

```
!$acc <op-see-above> async[(scalar-int-expr)]
```

→ Integer argument (optional) can be seen as CUDA stream number

→ Integer argument can be used in a wait directive

OpenACC 2.0: int arg can be
`acc_async_sync` (sync execution)
`acc_async_noval` (same stream)

→ Async activities with same argument: executed in order

→ Async activities with diff. argument: executed in any order relative to each other

background OpenACC

Streams

- **Streams = CUDA concept**

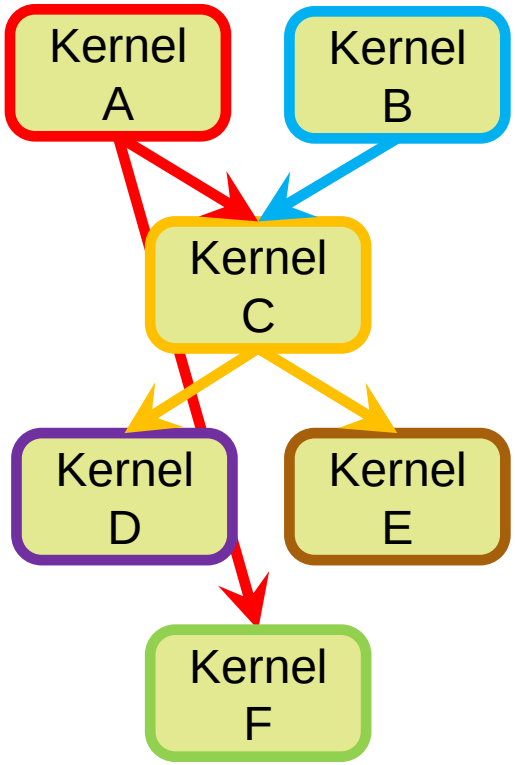
- **Stream = sequence of operations that execute in issue-order on GPU**
  - → Same int-expr in async clause: one stream
  - → Different streams/ int-expr: any order relative to each other

- **Tips for debugging - disable async**
  - → Environment variable `ACC_SYNCHRONOUS=1`
  - → Use pre-defined int-expr `acc_async_sync`

Asynchronous operations & wait

- **Synchronize async operations → wait directive**
  → Wait for completion of an asynchronous activity (all or certain stream)

```
#pragma acc parallel loop async(1)
// kernel A

#pragma acc parallel loop async(2)
// kernel B
#pragma acc wait(1,2) async(3)

#pragma acc parallel loop async(3)
// wait(1,2) // alternative to wait directive
// kernel C

#pragma acc parallel loop async(4) wait(3)
// kernel D

#pragma acc parallel loop async(5) wait(3)
// kernel E

#pragma acc wait(1)
// kernel F // on host
```



Courtesy by J.C. Beyer, Cray Inc.

Directives: wait

- **Wait directive**
  - → Host thread waits for completion of an asynchronous activity
  - → If integer expression specified, waits for all async activities with the same value
  - → Otherwise, host waits until all async activities have completed

C/C++

```
#pragma acc wait[(scalar-int-expr)]
```

Fortran

```
!$acc wait[(scalar-int-expr)]
```

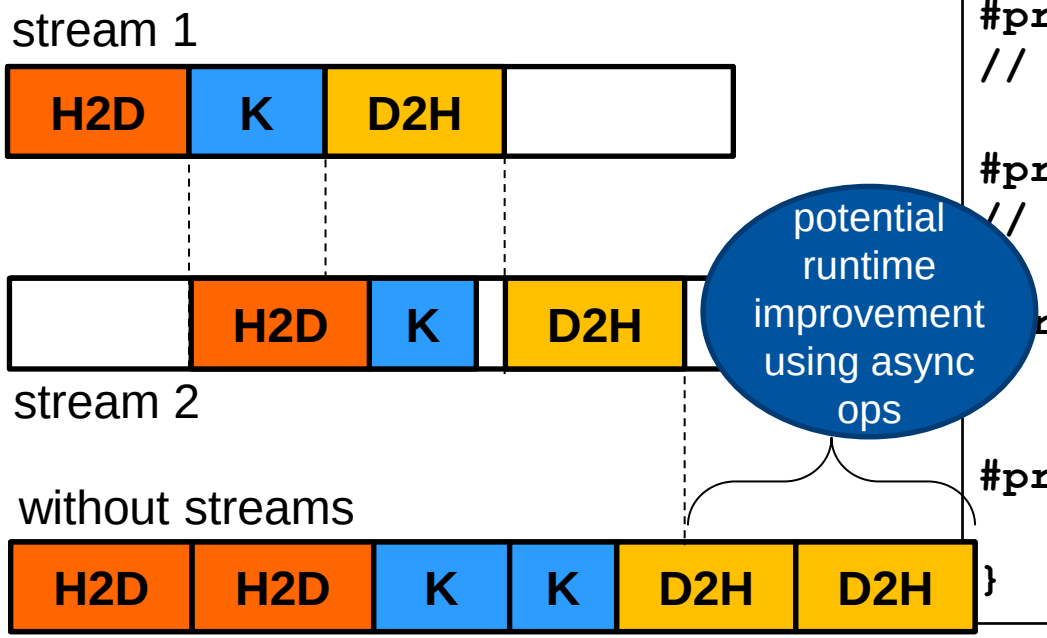
OpenACC 2.0: `wait` clause available (`parallel`, `kernels`, `update`, `enter data`, `exit data`)

- **Runtime routines**
  - → **`int acc_async_test(int)`**: tests for completion of all associated async activities; returns nonzero value/.true. if all have completed
  - → **`int acc_asnyc_test_all()`**: tests for completion of all async activities
  - → **`int acc_async_wait(int)`**: waits for completion of all associated async activites; routine will not return until the latest async activity has completed
  - → **`int acc_async_wait_all()`**: waits for completion of all asnyc activities

background OpenACC

Overlapping data transfers and computations

- **Other feature of streams**
- **Allows overlap of tasks on GPU**
  - → PCIe transfers in both directions
  - → Plus multiple kernels (up to 16 with Kepler)




```
#pragma acc data create(A[0:n])
{
#pragma acc update \
   device(A[0:n/2]) async(1)

#pragma acc update \
   device(A[n/2:n/2]) async(2)

#pragma acc parallel loop async(1)
// compute on A[0:n/2]

#pragma acc parallel loop async(2)
// compute on A[n/2:n/2]

ragma acc update \
   host(A[0:n/2])  async(1)

#pragma acc update \
   host(A[n/2:n/2]) async(2)
}
```

Summary

- **OpenACC default: synchronous operations**
  - → CPU thread waits until OpenACC kernel/ movement is completed
- **Asynchronous operations enable:**
  - → Heterogeneous computing (utilizing CPU + GPU)
    - → **`async`** clause returns control directly to CPU thread
  - → Overlapping of kernels (and data transfers)
    - → Specify streams by using int-expressions in **`async`** clause
- **Synchronization possible by `wait` directive**

Hands-On

- **Heterogeneous computing on CPU & GPU**
  - → Decomposition of the matrix (what is the best distribution?)
  - → Use **`async`** & **`wait`**
  - → Make sure reduction variables are copied back at certain point
  - → Use OpenMP on host
  - → Follow TODOs in **`GPU/exercises/task4`**

Exercise 2.6

Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.19
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	12.36
	OpenACC-Data	1.21
	OpenACC-Collapse	1.16
	OpenACC-Hetero *(1 GPU + 12 OMP threads)*	0.92
	OpenACC-MultiGPU *(2 GPUs + 12 OMP threads)*	

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Multiple GPUs

- **Several GPUs within one compute node**
  - → Can be used within or across processes/ threads
  - → E.g. assign one device to each CPU thread/ process

```
#include <openacc.h>

acc_set_device_num(0, acc_device_nvidia);
#pragma acc kernels async

acc_set_device_num(1, acc_device_nvidia);
#pragma acc kernels async
```

Device-specific tuning of clauses possible with **`device_type`** kernel clause.

Multiple GPUs – OpenMP + MPI

OpenMP

```
#include <openacc.h>
#include <omp.h>

#pragma omp parallel num_threads(12)
{
  int numdev = acc_get_num_devices(acc_device_nvidia);
  int devID = omp_get_thread_num() % numdev;
  acc_set_device_num(devID, acc_device_nvidia);
}
```

MPI

```
#include <openacc.h>
#include <mpi.h>

int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
int numdev = acc_get_num_devices(acc_device_nvidia);
int devID = myrank % numdev;
acc_set_device_num(devID, acc_device_nvidia);
```

Directives: Multiple Devices

- **Multi GPU (or device) setup**
  - → `int acc_get_num_devices(acc_device_t)`: returns number of device of type
  - → `void acc_set_device_num(int,acc_device_t)`: sets device number of type
  - → `int acc_get_device_num(acc_device_t)`: returns device number of type
  - → `void acc_set_device_type(acc_device_t)`: type for offload region
  - → `acc_device_t acc_get_device_type()`: returns device type for next offload region
- **Device-specific clause tuning: `device_type` or `dtype`**
  - → Takes accelerator architecture name or an asterisk: tuning for this device
  - → Clauses belong to dtype that follow it up to the end or another dtype
  - → Can applied for: `parallel`, `kernels`, `loop`, `update`, `routine`

C/C++

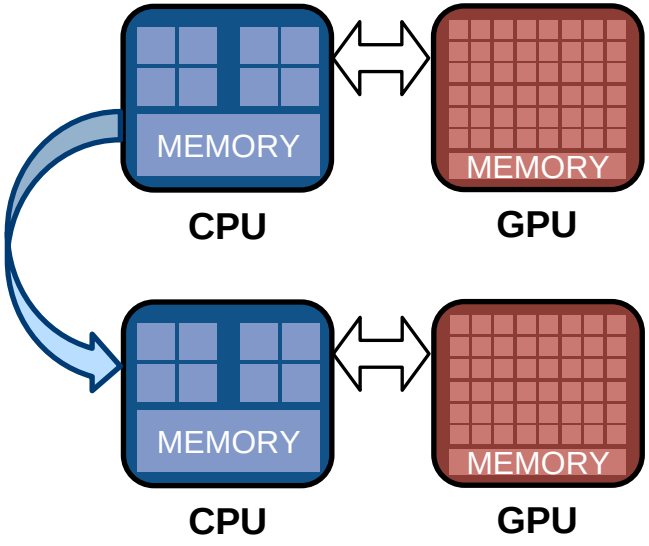
```
#pragma acc <op> device_type(device-type list) (clauses)
#pragma acc <op> device_type(*) (clauses)
```

Fortran

```
!$acc <op> device_type(device-type list) (clauses)
!$acc <op> device_type(*) (clauses)
```

Multiple GPUs – MPI

- **Multiple GPUs across nodes**
  - → Use MPI for communication
  - → Use update for a upt-to-date version on the before sending over



MPI rank 1

```
#pragma acc update host \
    ( s_buf[0:size] )

MPI_Send(s_buf,size,MPI_CHAR,
    n-1,tag,MPI_COMM_WORLD);
```

MPI rank n-1

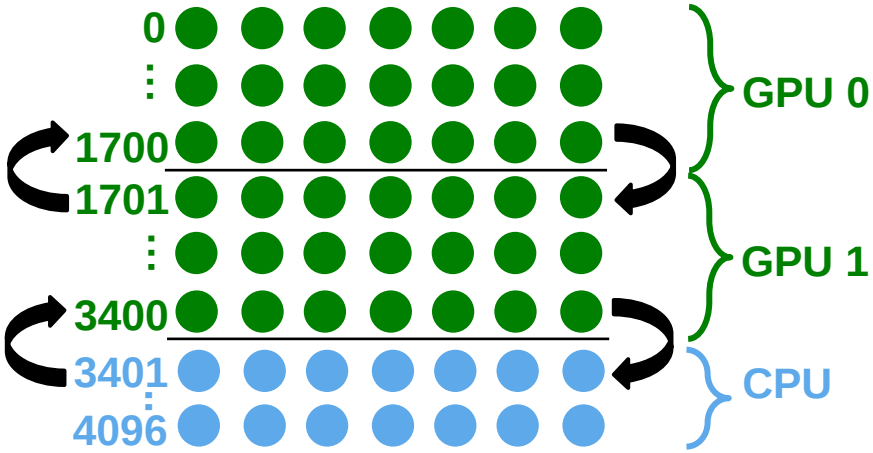
```
MPI_Recv(r_buf,size,MPI_CHAR,
    0,tag,MPI_COMM_WORLD,&stat);

#pragma acc update device \
    (r_buf[0:size] )
```

Summary

- **OpenACC can use several devices**
  - → Within one node: `acc_set_device_num` to handle GPU affinity
  - → Across nodes: use MPI

Hands-On

- **Extend the heterogeneous version to use 2 GPUs within one compute node (w/o MPI)**
  - → Decomposition of the matrix
  - → Use `acc_set_device_num` to differentiate between the GPUs
  - → Use `enter data` and `exit data` to move initial data to the GPUs
  - → Use `update` for CPU – GPU halo exchange
  - → Think about correct synchronization
  - → Follow TODOs in `GPU/exercises/task5`



Exercise 2.7

Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.19
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	12.36
	OpenACC-Data	1.21
	OpenACC-Collapse	1.16
	OpenACC-Hetero *(1 GPU + 12 OMP threads)*	0.92
	OpenACC-MultiGPU *(2 GPUs + 12 OMP threads)*	0.70

Summary on OpenACC

- **Easily enables running applications on GPUs**
  - → Parallel regions and loop constructs for offloading work
  - → Data region and clauses for data movement
- **Tuning possible**
  - → Asynchronous operations
  - → Interoperability with low-level kernels or CUDA libraries
  - → Loop schedules & launch configurations
  - → Caching & strip-mining
- **General GPU optimizations should be applied**
  - → Coalescing,…
- **Future? OpenACC or/and OpenMP?**

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Latency Hiding & Occupancy
  - → [Launch Configuration & Loop Schedules]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Interoperability with CUDA & GPU Libraries]
  - → Heterogeneous Computing
  - → [Multiple GPUs]
- **Comparison of OpenACC & OpenMP Device Constructs**

Introduction

- **OpenMP = de-facto standard for shared-memory parallelization**
  - → From 1997 until now
- **Since 2009: OpenMP Accelerator sub-committee**
  - → Sub group wanted faster development: OpenACC

    Idea: include lessons learnt into OpenMP standard
- **OpenACC**
  - → Initiated by Cray, CAPS, PGI, NVIDIA
  - → 2011: specification v1.0
  - → 2013: specification v2.0

http://www.OpenMP.org

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006. Main topics:

- Thread affinity
- Tasking
- Tool support
- Accelerator support

Some content of the following slides has been developed by James Beyer (Cray) and Eric Stotzer (TI), the leaders of the OpenMP for Accelerators subcommittee.

Overview on OpenACC & OpenMP Device Capabilities

Comparison of Constructs

OpenACC	OpenMP	Description of constructs/ clauses
parallel	target	offload work
parallel	teams, parallel	create in par. running threads
kernels		compiler finds parallelism
loop	distribute, do, for, simd	worksharing across parallel units
data	target data	manage data transfers (block)
enter data		unstructured data lifetime: to and from
exit data		the device
update	target update	data movement in data environment
host data		interoperability with CUDA/ libs
cache		advice to put objects to closer memory
tile		strip-mining of data
declare	declare target	declare global, static data
routine	declare target	for function calls

Overview on OpenACC & OpenMP Device Capabilities

Comparison of Constructs

OpenACC	OpenMP	Description of constructs/ clauses
async(int)	task depend	async. exec. w/ dependencies
wait	taskwait	sync of streams/ tasks
async wait		async. waiting
parallel in parallel	parallel in parallel/ team	nested parallelism
device_type		device-specific clause tuning
atomic	atomic	atomic operations
	sections	non-iterative worksharing
	critical	block executed by master
	master, single	block executed by one thread
	barrier	synchronization of threads

Overview on OpenACC & OpenMP Device Capabilities

Execution & Memory Model

OpenACC | OpenMP

Host-directed execution

Weak device memory model (no sync at gang/ team level)

Separate or shared memory

Concepts & Features

OpenACC	OpenMP
Error if data not present	Fix non-present data
Async calls (recursion difficult)	Task parallelism (tasks, sections)
Device-specific clause tuning	
Caching & tiling	
Nested parallelism	Nested parallelism limited
Unstructured data lifetimes	
Deep copies w/ data API	
Interoperability with CUDA	
Routine calls for the same level of parallelism	Routine calls from different contexts
	…more coming soon

We talk about the following specifications: OpenACC 2.0, OpenMP 4.0.

Sandra Wienke | IT Center, RWTH Aachen University

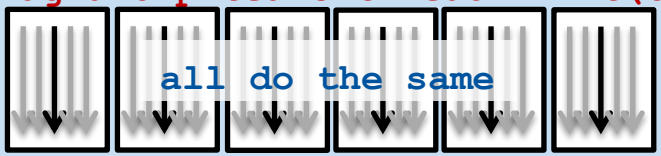
SAXPY (GPU)

OpenACC

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y
  // Run SAXPY
#pragma acc parallel \
 copy(y[0:n]) copyin(x[0.n])\
 vector_length(bsize)
#pragma acc loop gang
 for (int i = 0; i < n; i += bsize){
#pragma acc loop vector
   for (int j = i; j < i + bsize; j++) {
      y[j] = a*x[j] + y[j];
    }
  }
  free(x); free(y); return 0;
}
```

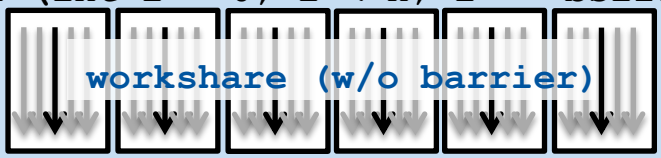
OpenMP

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y
  // Run SAXPY
#pragma omp target \
     map(tofrom:y[0:n]) map(to:x[0:n])
#pragma omp teams thread limit(bsize)
```

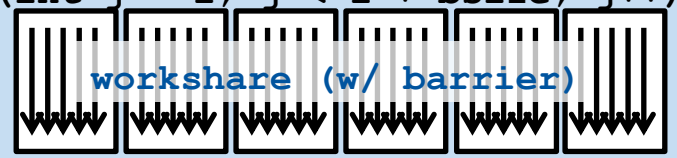


league = set of thread teams

```
#pragma omp distribute
  for (int i = 0; i < n; i += bsize){
```




```
#pragma omp parallel for
    for (int j = i; j < i + bsize; j++) {
```




```
        y[j] = a*x[j] + y[j];
  } }
  free(x); free(y); return 0;
}
```

SAXPY (GPU|MIC)

OpenACC

GPU

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y
  // Run SAXPY
#pragma acc parallel \
 loop gang vector
 for (int i = 0; i < n; i++){
     y[j] = a*x[j] + y[j];
  }
  free(x); free(y); return 0;
}
```

OpenMP

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y
  // Run SAXPY
#pragma omp target
#pragma omp teams distribute parallel for
  for (int i = 0; i < n; i++) {
        y[j] = a*x[j] + y[j];
  }
  free(x); free(y); return 0;
}
```

MIC

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y
  // Run SAXPY
#pragma acc parallel \
 loop gang vector
 for (int i = 0; i < n; i++){
     y[j] = a*x[j] + y[j];
  }
  free(x); free(y); return 0;
}
```

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n *
     sizeof(float));
  float *y = (float*) malloc(n *
     sizeof(float));
  // Define scalars n, a & init x, y
  // Run SAXPY
#pragma omp target
#pragma omp parallel for simd
  for (int i = 0; i < n; i++) {
        y[j] = a*x[j] + y[j];
  }
  free(x); free(y); return 0;
}
```

Subprograms Sharing GPU: Error vs. Fix-Up [Concept]

Courtesy by James Beyer, Cray Inc.

OpenACC

```
int double_scalar(int c){
 return (2*c);
}

void double_array(int* b, int n){
#pragma acc parallel present(b[:n])
#pragma acc loop
 for(int i=0; i<n; i++){
  b[i] = double_scalar(b[i]);
 }
}

int main(){
 int *a; // malloc with size n
#pragma acc data copyout(a[:n])
{                                if (test)
#pragma acc parallel loop
 for(int i=0; i<n; i++){
  a[i] = I;
 }
 double_array(a);
}
}
```

Missing data: Error
→ `present`: no data movement;
error if data not there

OpenMP

```
int double_scalar(int c){
 return (2*c);
}

void double_array(int* b, int n){
#pragma omp parallel map(tofrom:b[:n])
#pragma omp for
 for(int i=0; i<n; i++){
  b[i] = double_scalar(b[i]);
 }
}

int main(){
 int *a; // malloc with size n
#pragma omp target data map(from:a[:n])
{                                if (test)
#pragma omp target parallel for
 for(int i=0; i<n; i++){
  a[i] = I;
 }
 double_array(a);
}
}
```

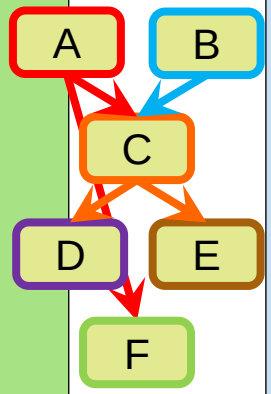
Missing data: Fix-Up
Implicit check for data;
move if not there

Heterogeneous Computing: async vs. tasks [Concept]

Courtesy by James Beyer, Cray Inc.

OpenACC

```
#pragma acc parallel loop async(1)
// kernel A

#pragma acc parallel loop async(2)
// kernel B
#pragma acc wait(1,2) async(3)

#pragma acc parallel loop async(3)
// wait(1,2) // or wait directive
// kernel C

#pragma acc parallel loop async(4) \
        wait(3)
// kernel D

#pragma acc parallel loop async(5) \
        wait(3)
// kernel E

#pragma acc wait(1)
// kernel F // on host
```



OpenMP

```
#pragma omp task depend(inout:a)
#pragma omp target parallel for
// kernel A

#pragma omp task depend(inout:b)
#pragma omp target parallel for
// kernel B

#pragma omp task depend(inout:c) \
                 depend(in:a,b)
#pragma omp target parallel for
// kernel C

#pragma omp task depend(inout:d) \
                 depend(in:c)
#pragma omp target parallel for
// kernel D

#pragma omp task depend(inout:e) \
                 depend(in:c)
#pragma omp target parallel for
// kernel E

#pragma omp task depend(in:a)
// kernel F // on host
```

OpenACC & OpenMP Support by Compilers

	Compiler	OpenACC	OpenMP
commercial	Cray	OpenACC 2.0 for NVIDIA GPUs	OpenMP 4.0 for NVIDIA GPUs
	IBM XL	-	OpenMP 4.5 for NVIDIA GPUs
	Intel	-	OpenMP 4.5 for Intel Xeon Phis
	PathScale	OpenACC 2.0 for NVIDIA GPUs and AMD FirePros	OpenMP 4.0 for NVIDIA GPUs and AMD FiroPros
	PGI	OpenACC 2.5 for NVIDIA GPUs and some AMD GPUs	- (no offloading support)
	Texas Instruments	-	OpenMP 4.0 for KeyStone II
free	GNU	OpenACC 2.0a for NVIDIA GPUs	OpenMP 4.5 for Intel Xeon Phis (from KNL on) and HSA
	Omni	OpenACC 1.0 for NVIDIA GPUs	- (no offloading support)
	Open Accelerator Research Compiler	OpenACC 1.0 for ?	- (no offloading support)
	OpenUH	OpenACC 2.0 for NVIDIA GPUs and HSA	- (no offloading support)

Information collected by Jonas Hahnfeld. Thanks! (as of December 2016)