GPGPU Programming with OpenACC

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

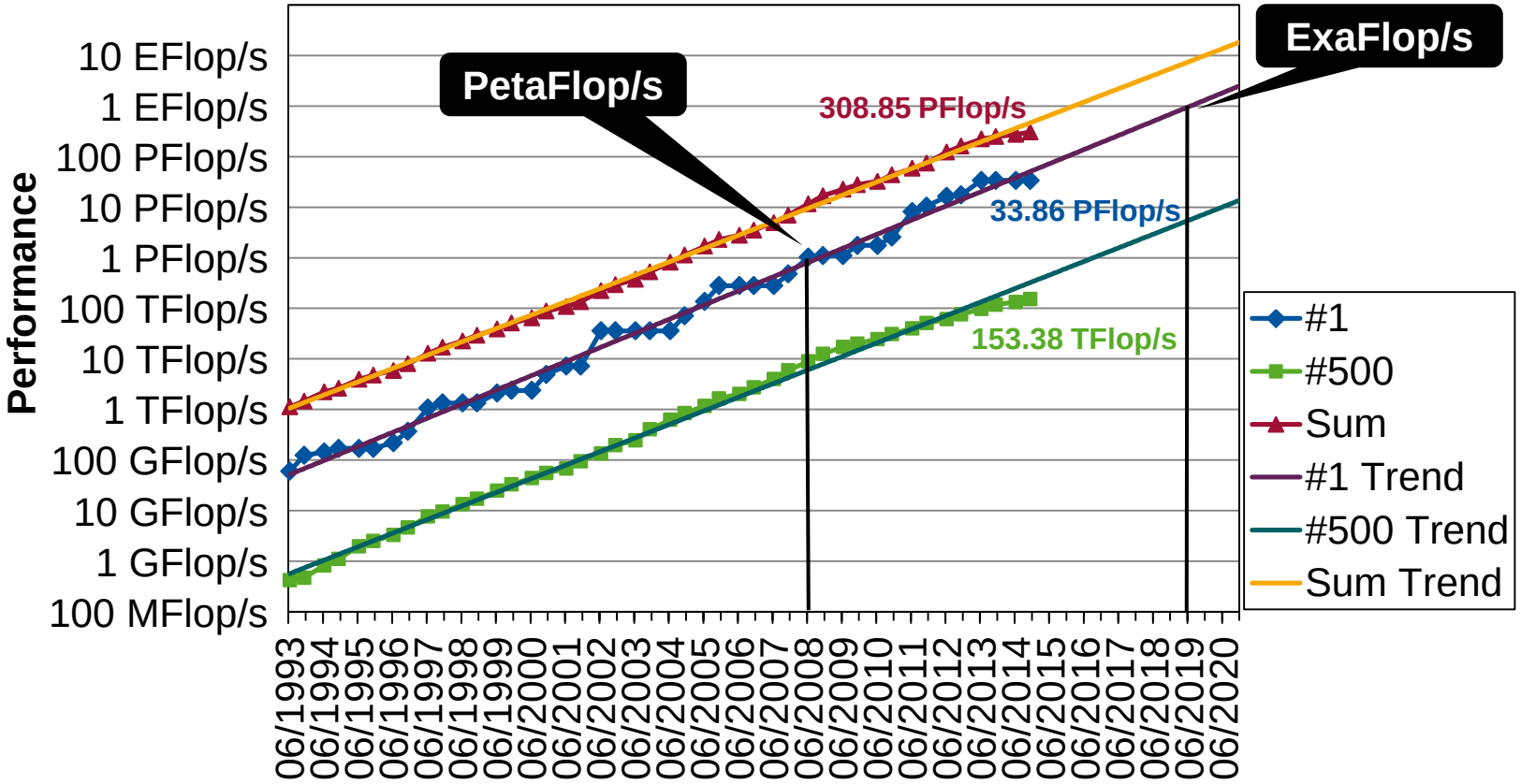
PPCES, March 20th 2015

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions 
  - → Data Management 
- **OpenACC Advanced**
  - → Heterogeneous Computing 
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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)

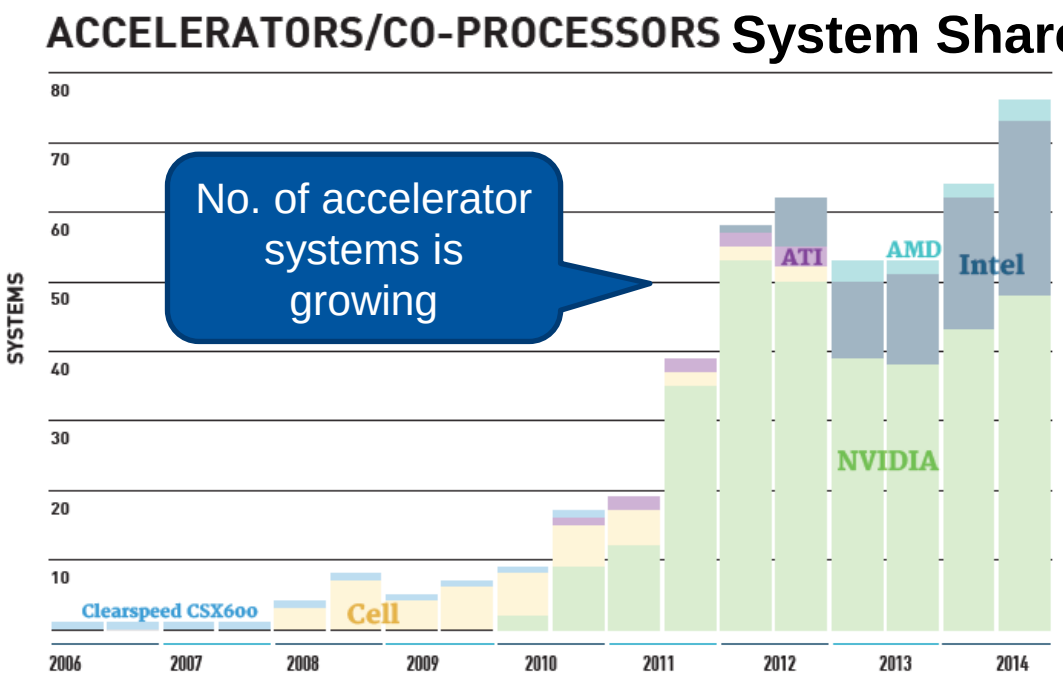


Source: Top500, 11/2014

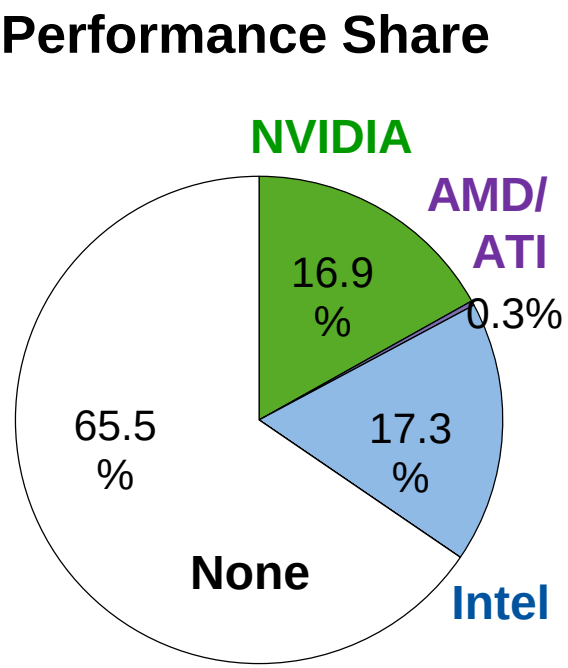
Motivation

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

Here, NVIDIA GPUs are the focus.



Source: Top500, 11/2014



Source: Top500, 6/2014

Motivation – Top500 (11/2014)

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

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

Motivation – Green500 (11/2014)

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

performance per watt

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

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

→ 985 MFLOPS/W

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

Overview

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

- **History – a very brief overview**

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

  Fixed-function 3D graphics pipeline

  Graphics APIs like OpenGL, DirectX popular

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

  (adjustments in OpenGL, DirectX)

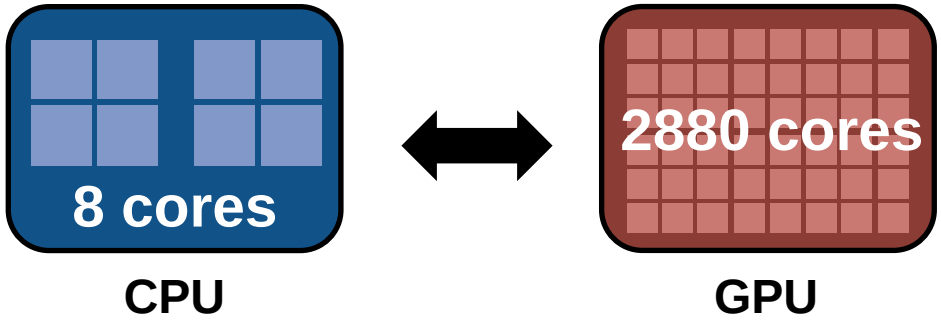
  Researchers take notice of performance growth of GPUs: Tasks must be cast into native graphics operations

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

  Support of programming language C, synchronization,…

  → “General purpose”

Comparison CPU ⇔ GPU

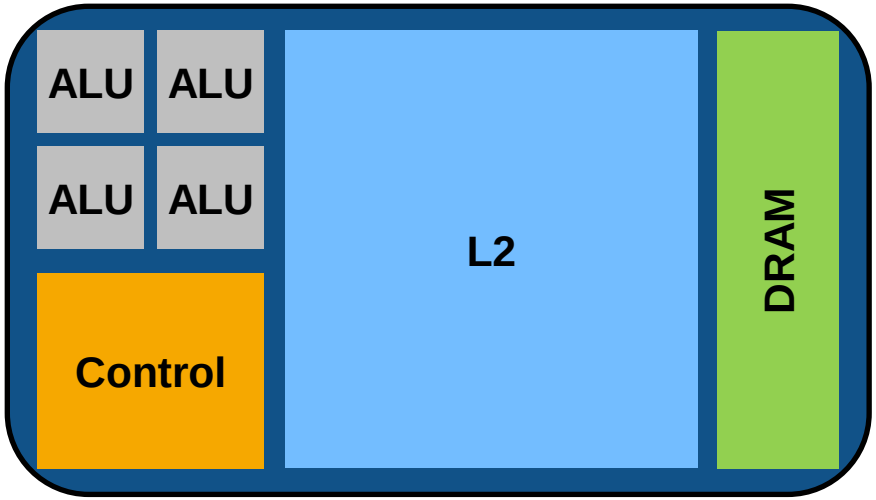


- **GPU-Threads**
  - → Thousands (“few” on CPU)
  - → Light-weight, little creation overhead
  - → Fast switching

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

Comparison CPU ⇔ GPU

- **Different design**



CPU

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

GPU

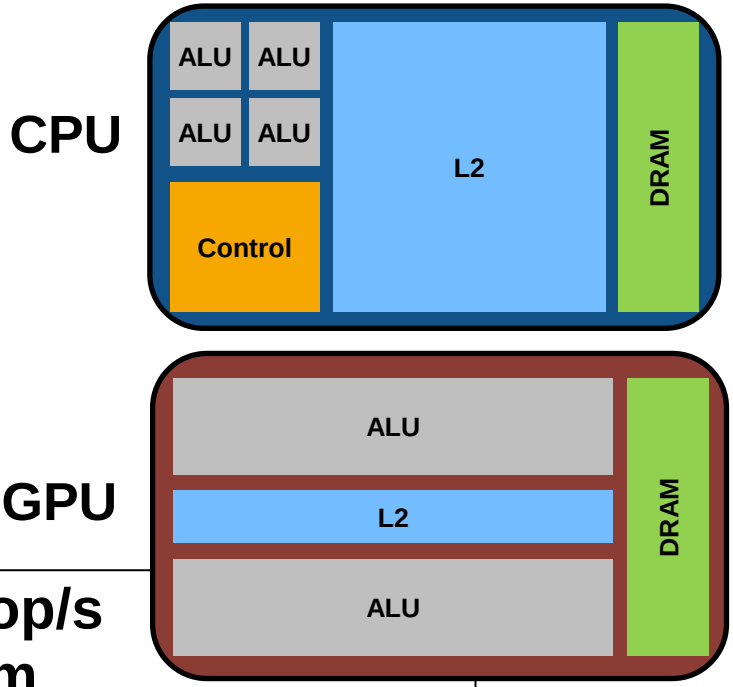
- → Optimized for data-parallel throughput
- → Architecture tolerant of memory latency
- → More transistors dedicated to computation

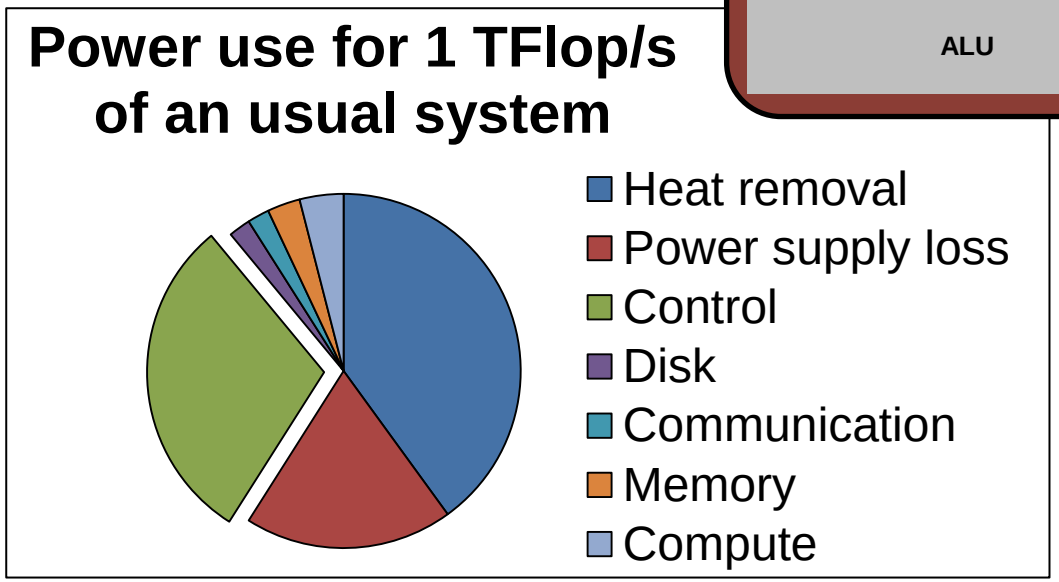
Why can accelerators deliver good performance watt ratio?

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

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

→ No control logic





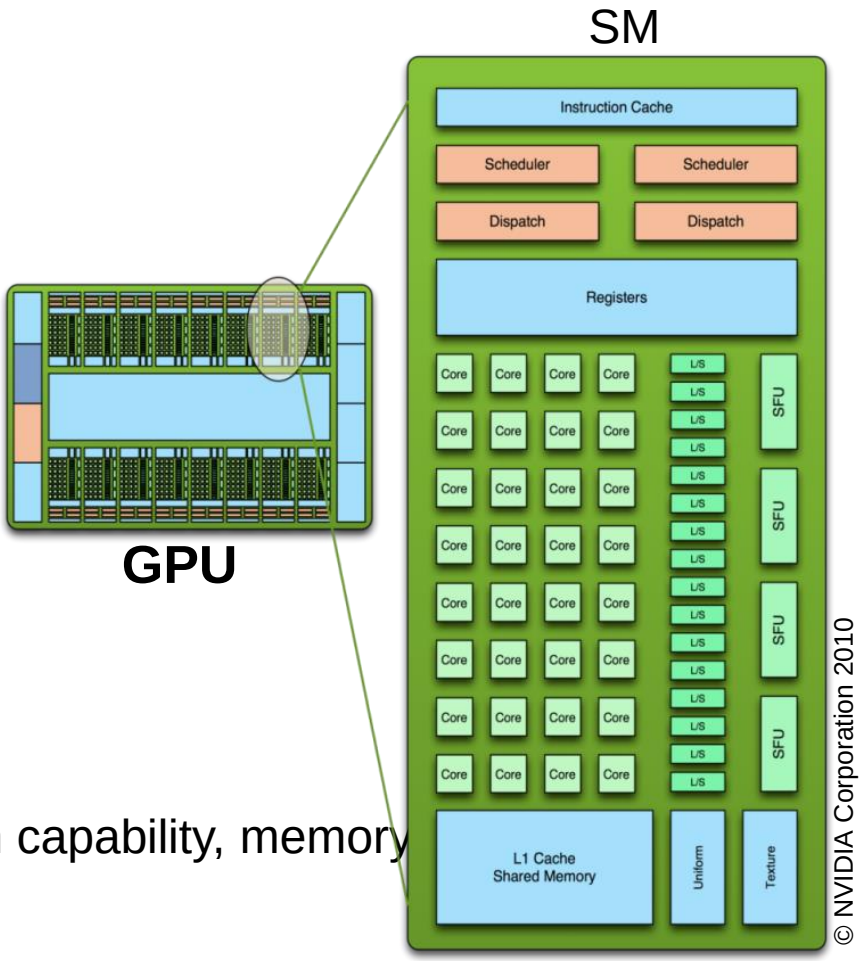
Source: Andrey Semin (Intel): HPC systems energy efficiency optimization thru hardware-software co-design on Intel technologies, EnaHPC 2011; & S. Borkar, J. Gustafson

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Heterogeneous Computing
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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

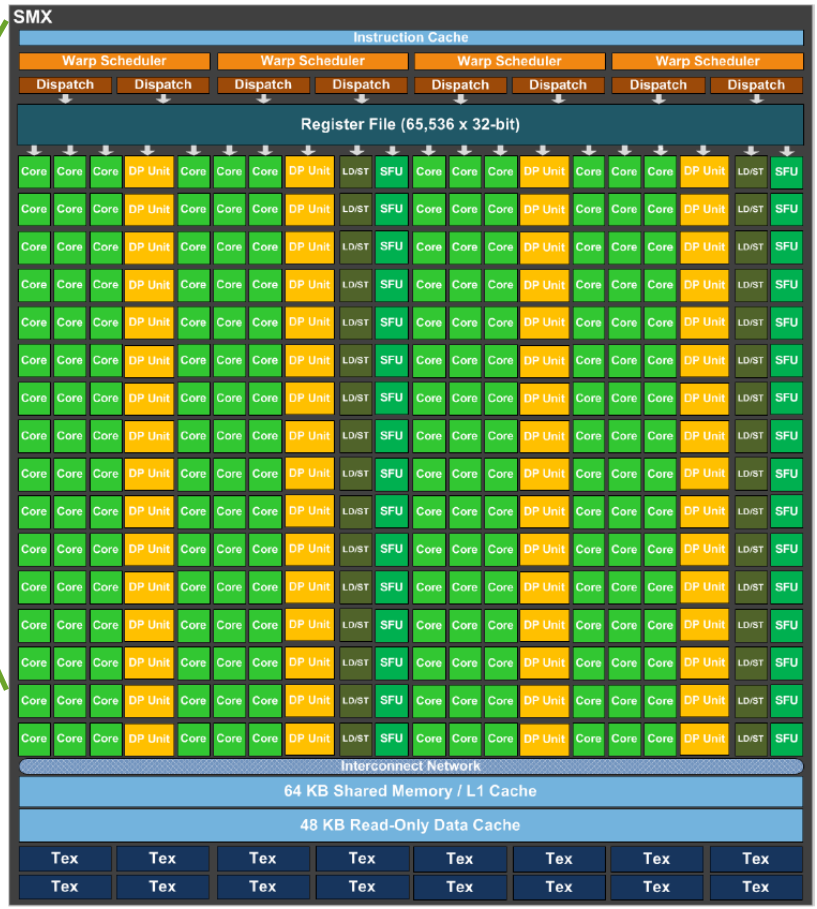


GPU architecture: Kepler

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

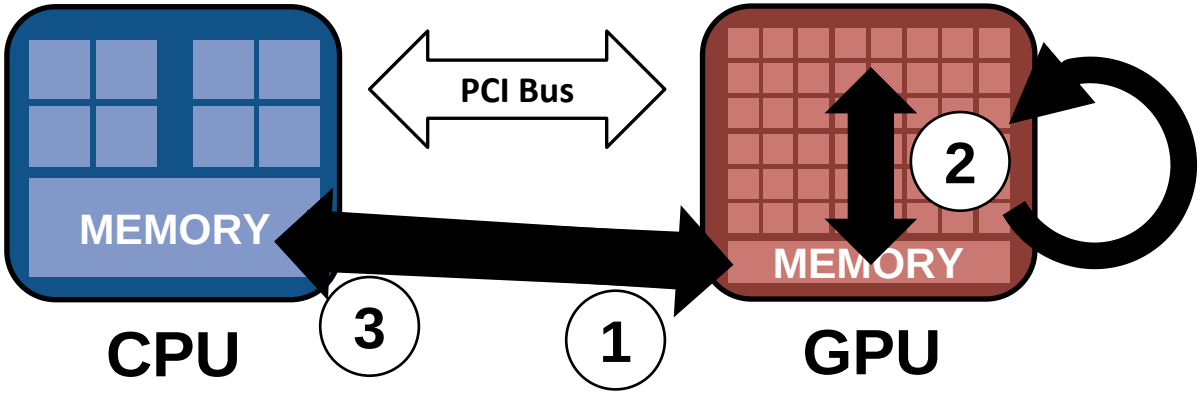
GPU

SMX



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

Comparison CPU ⇔ GPU



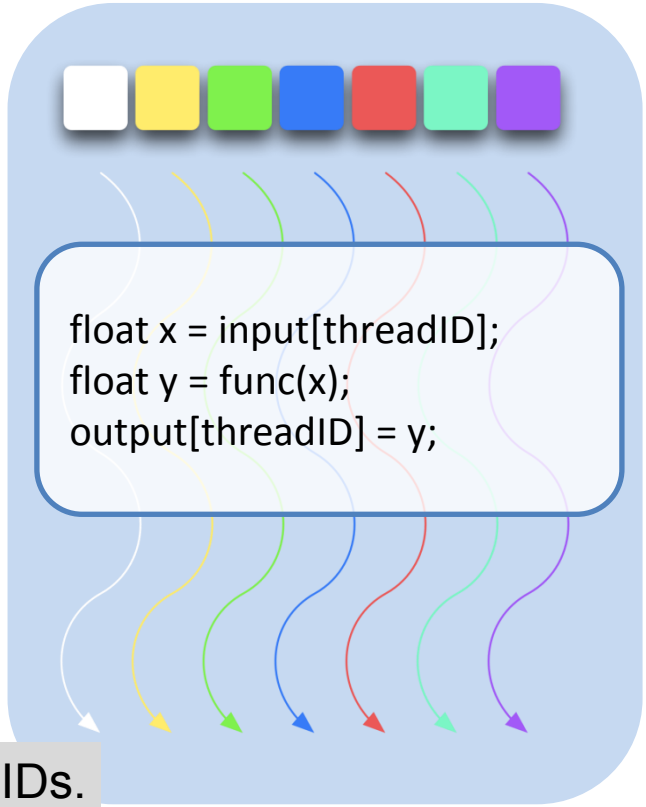
- **Weak memory model**
  - → Host + device memory = separate entities
  - → No coherence between host + device
    - → Data transfers needed
- **Host-directed execution model**
  - → Copy input data from CPU mem. to device mem.
  - → Execute the device program
  - → Copy results from device mem. to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

like the Xeon Phi Offload model

Programming model

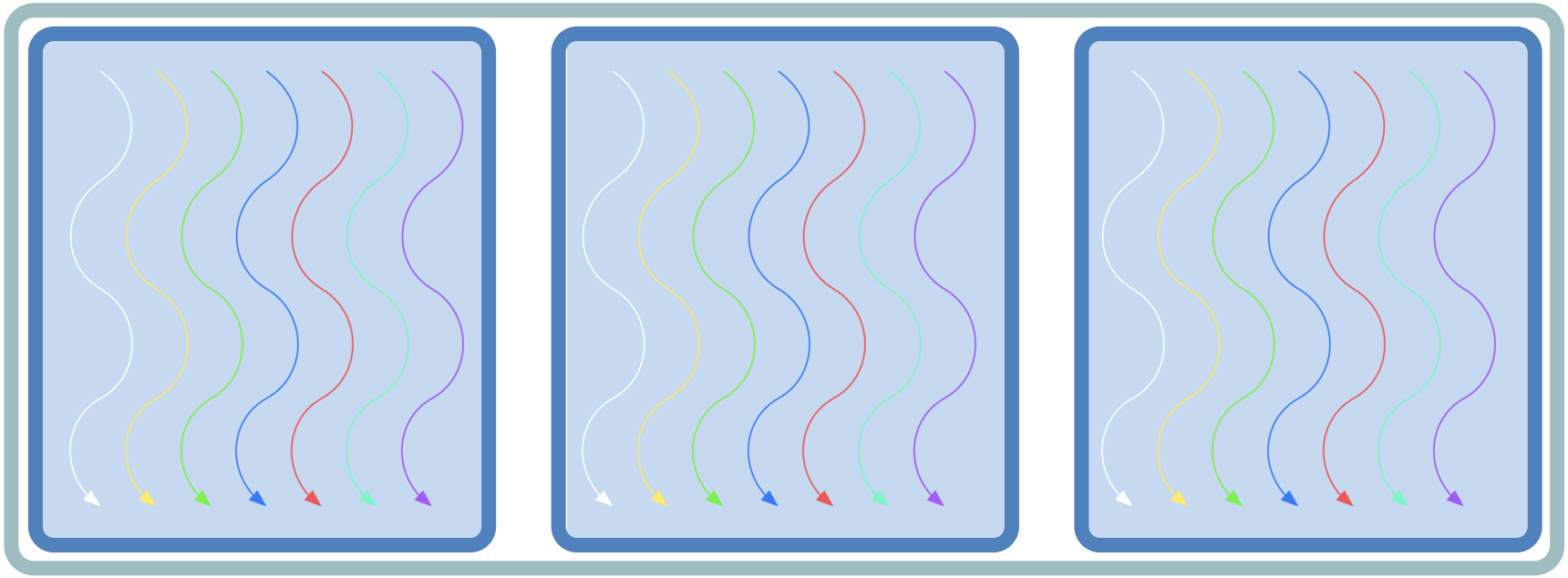
- **Definitions**
  - → Host: CPU, executes functions
  - → Device: usually GPU, executes kernels

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



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

Programming model

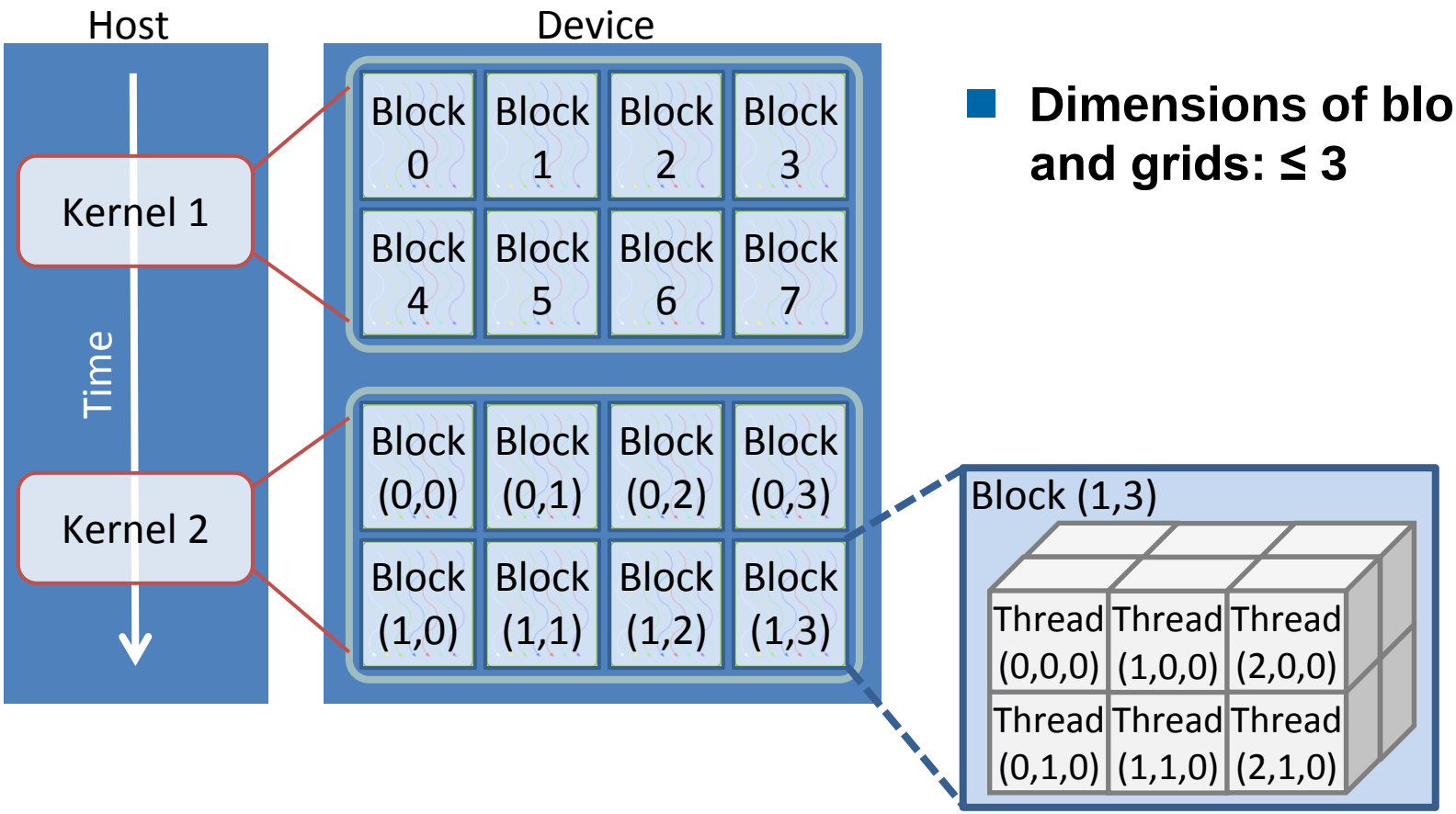


Based on: NVIDIA Corporation 2010

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

Programming model

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



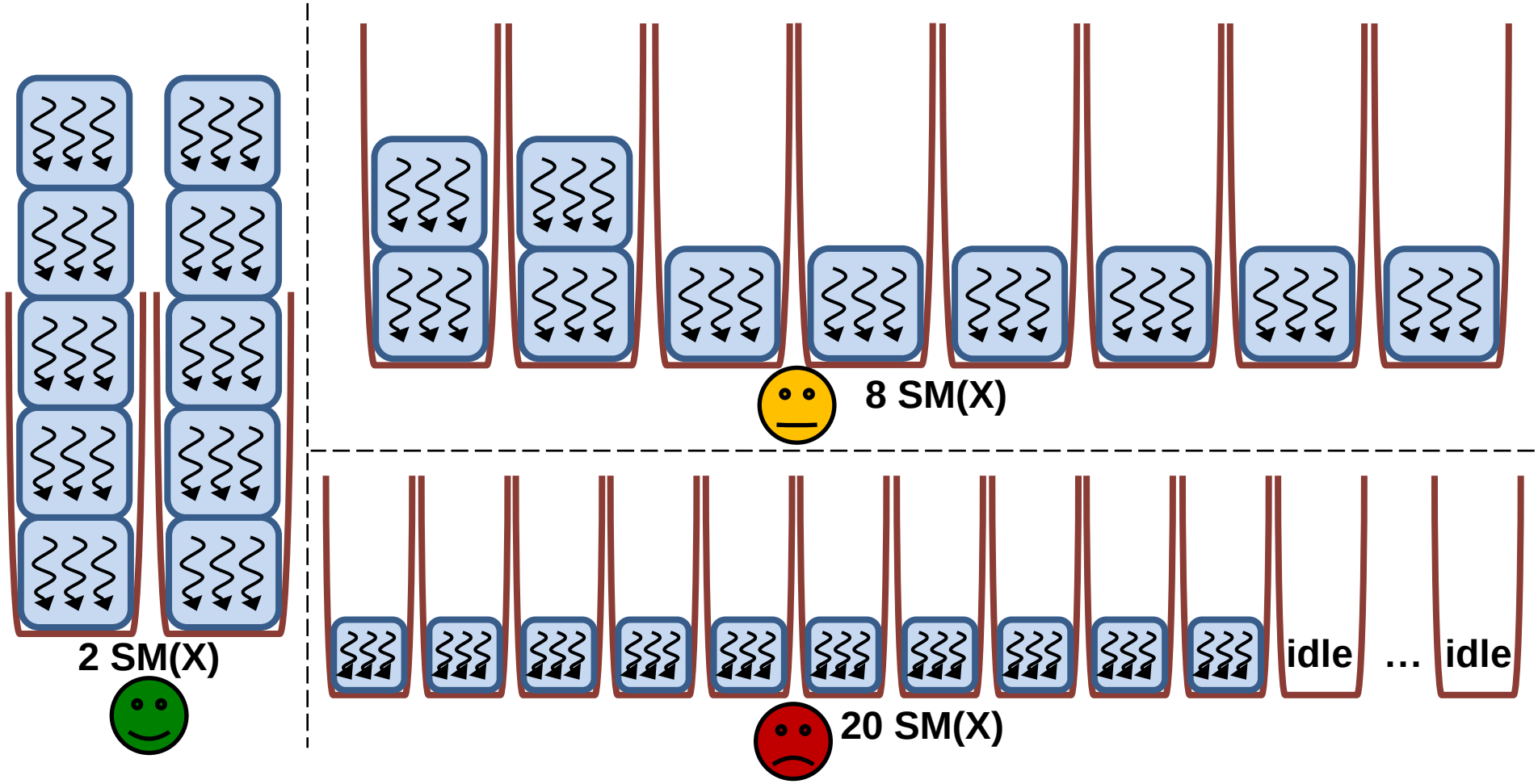
- **Dimensions of blocks and grids: ≤ 3**

Programming model

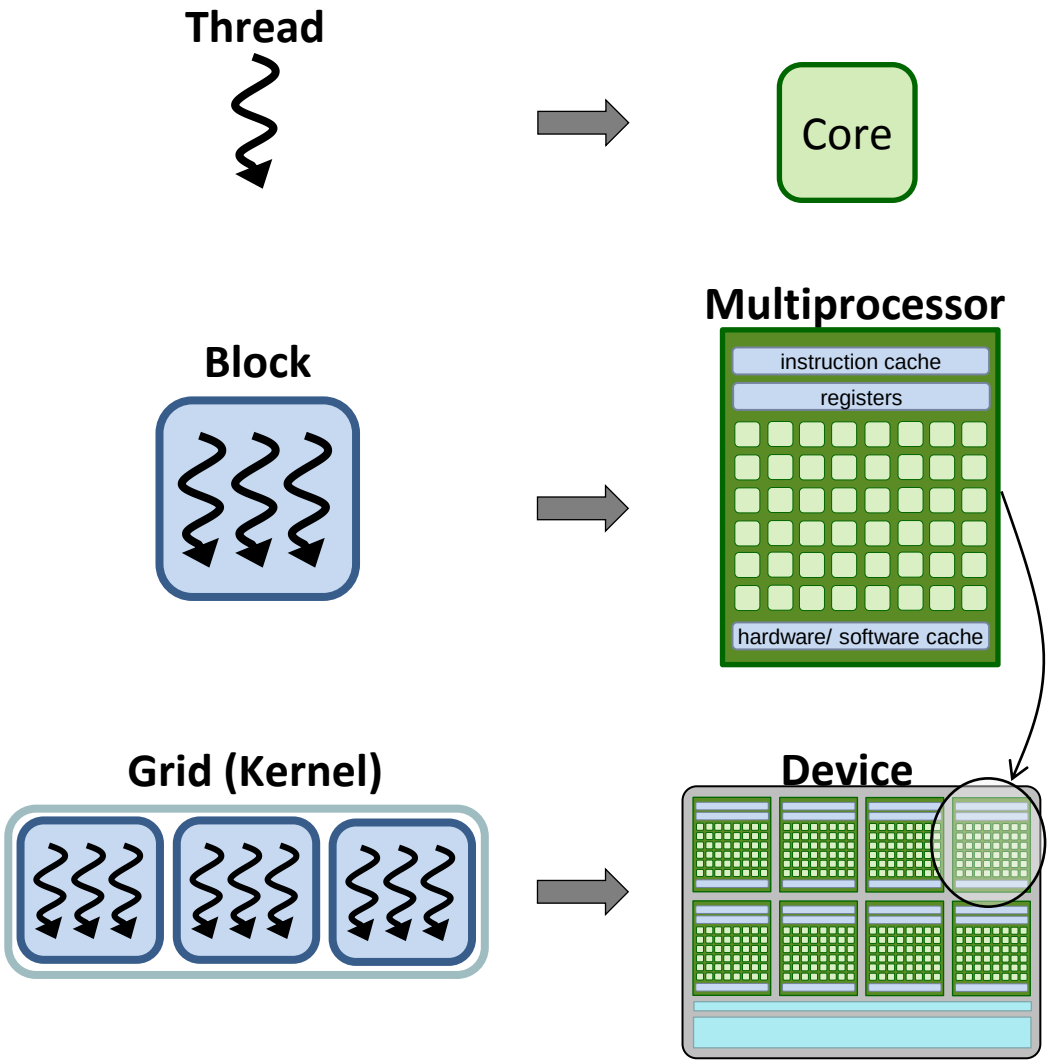
- **Why blocks?**
  - → Cooperation of threads within a block possible
    - →Synchronization
    - →Share data/ results using shared memory
  - → Scalability
    - →Fast communication between n threads is not feasible when n large
      - →No global synchronization on GPU possible (only by completing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

Scalability

- **Assume: 10 thread blocks**



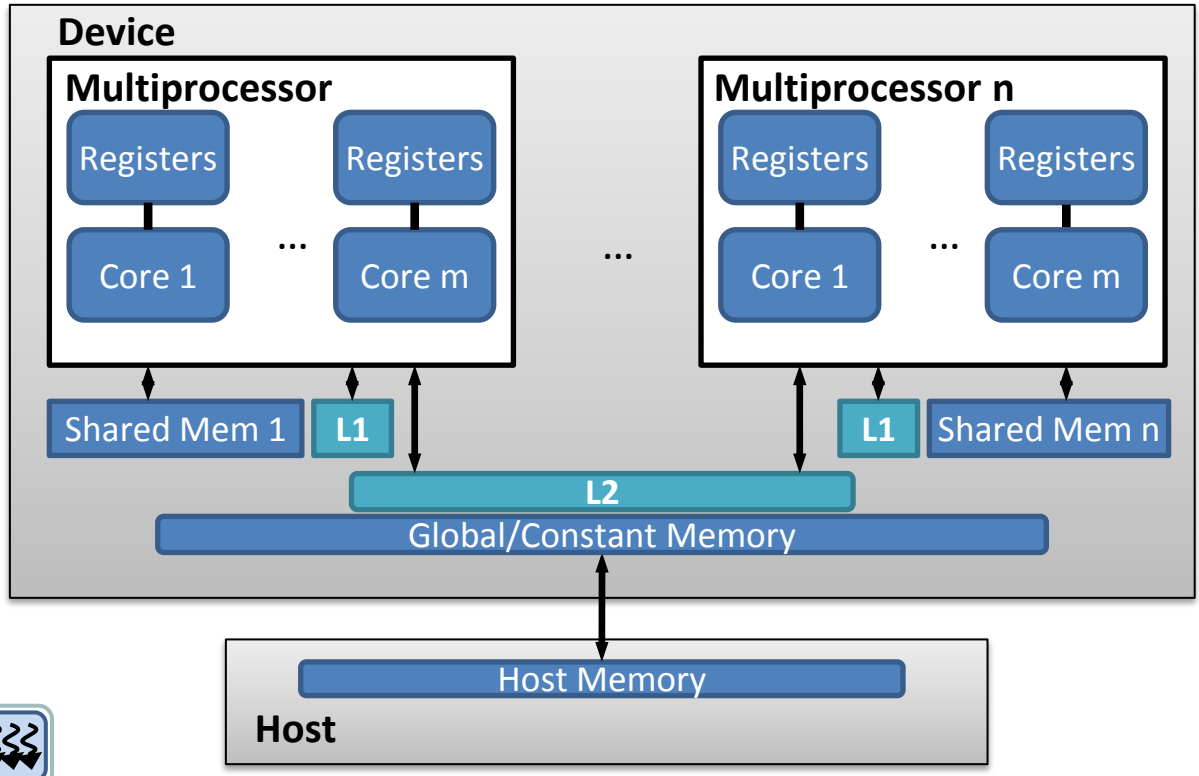
Mapping to Execution Model



- 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

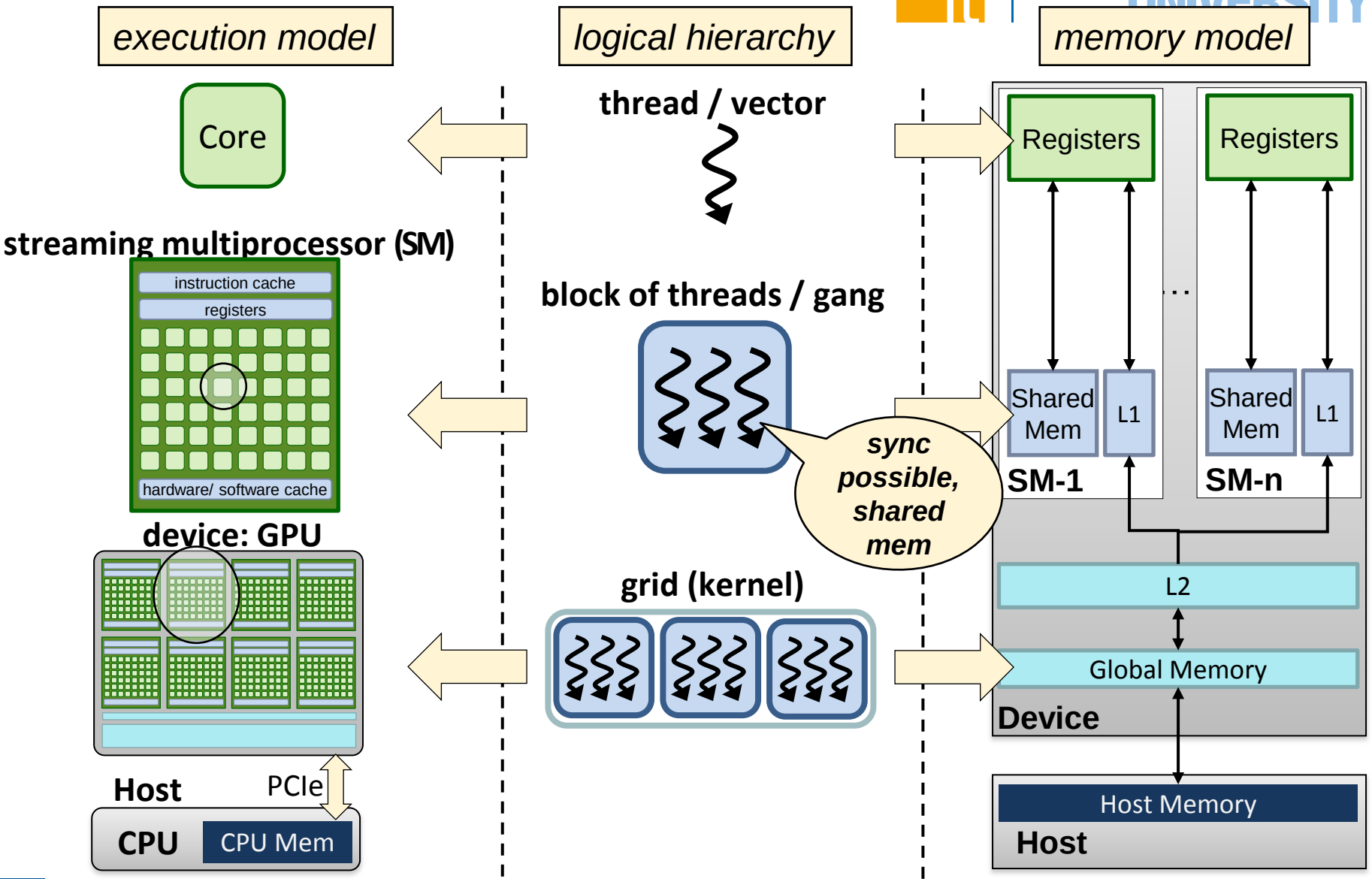
Memory model

- **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

Summary

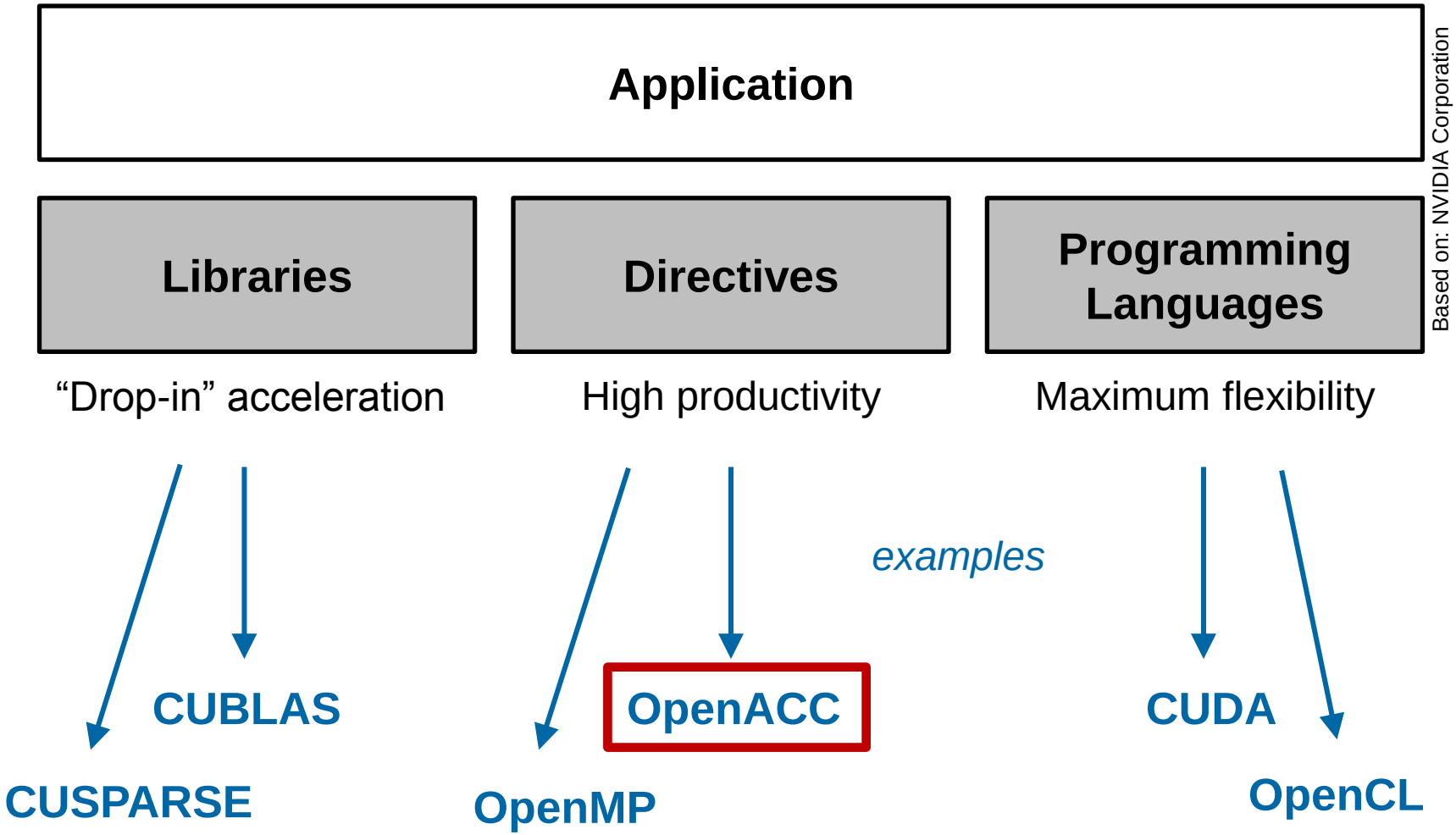


Vector, worker, gang mapping is compiler dependent.

Agenda

- Introduction to GPGPUs
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- OpenACC Advanced
  - → Heterogeneous Computing
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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 *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);
saxpy_parallel<<<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

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 4.0 device constructs**

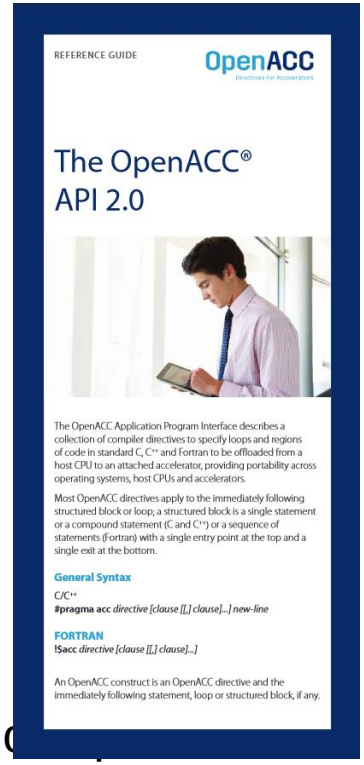
OpenMP 4.0 device constructs (for the Intel Xeon Phi) were covered on Thursday. For GPUs, implementations are coming soon. Thus, we just cover it as outlook and emphasize similarities and differences to OpenACC. Next year, we will probably teach OpenMP for accelerators.

Overview *OpenACC*

- **Open industry standard**
  - → Portability

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

- **Introduced by CAPS, Cray, NVIDIA, PGI (Nov. 2011)**
- **Support**
  - → C,C++ and Fortran
  - → NVIDIA GPUs, AMD GPUs & Intel MIC (now/near future)
- **Timeline**
  - → Nov'11: Specification 1.0
  - → Q1/Q2'12: Cray, PGI & CAPS compiler understands parts o
  - → Nov'12: Proposed additions for OpenACC 2.0, more supporters
  - → Jun'13: Specification 2.0
  - → Q4'13/Q1'14: Cray, PGI compiler understand OpenACC 2.0
  - → Q4'14: Proposals for complex data management & tools interface



Source: www.openacc.org

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,cc20,6.5 -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
- → **6.5** Optional. Uses CUDA Toolkit 6.5 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
  - → Heterogeneous Computing
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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];
  }
 // Modify y on host, do not touch x on host



  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];
  }
 // Modify y on host, do not touch


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

"magic"

sync

Kernel 1

Kernel 2

PGI Compiler Feedback

```
main:
29,Generating copy(y[0:n])
   Generating copyin(x[0:n])
   Accelerator kernel generated
   31, #pragma acc loop gang,vector(256)
29,Generating NVIDIA code
35,Generating copy(y[0:n])
   Generating or_copyin(x[0:n])
   Accelerator kernel generated
   37, #pragma acc loop gang,vector(256)
35,Generating NVIDIA code
```

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



sync
#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;
}
```

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”

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: size]

[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

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)`

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**
  - → Distributes work into thread blocks ……
  - → Distributes work into warps ……
  - → Distributes work into threads within warp/ thread block ……
  - → Executes loop sequentially on the device……
  - → Collapse *n* tightly nested loops……
  - → Says independent loop iterations(kernels loop only)….
  - → Reduction with *op*……
  - → Private copy for each loop iteration……

C/C++, Fortran

Loop schedule:
```
gang
worker
vector
seq
```
```
collapse(n)
independent
reduction(op:list)
private(list)
```

OpenACC 2.0 also: `auto, tile, device_type`

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`

Procedure Calls

- **Procedure calls within parallel region**
  - → `routine` directive: compile for accelerator + host
  - → Clauses denote level of parallelism

```
#pragma acc routine seq
float saxpy(float a, float x, float y) {
  return (a*x + y);
}

// [..]
#pragma acc parallel
#pragma acc gang vector
for (int i=0; i<n; ++i) {
 //call on the device
  y[i] = saxpy(a,x[i],y[i]);
}
```

- **With OpenACC 1.0 (!) routines must be inlined (no routine support)**

Directives: Procedure Calls

- **Routine construct**
  - → Tells compiler to compile the given procedure for an accelerator & host
  - → Either use *name* or directly put construct at procedure call

C/C++

```
#pragma acc routine clause-list
#pragma acc routine (name) clause-list
```

Fortran

```
!$acc routine clause-list
!$acc routine (name) clause-list
```

- **A clause must be specified**
  - → (May) contain/call gang parallelism, executed in gang-redundant mode…….. `gang`
  - → (May) contain/call worker parallelism (no gang), executed in worker-single mode `worker`
  - → (May) contain/call vector parallelism (no worker/gang), executed in vector-single mode `vector`
  - → Does not contain/call gang, worker, vector parallelism…….. `seq`
  - → Specifies name for compiling/calling (as if specified in language being compiled) `bind(name)`
  - → Specifies name for compiling/calling (string is used for name)…….... `bind(string)`
  - → No compilation for the host, must called from within acc compute regions…… `nohost`
  - → Device-specific clause tuning…… `dtype`

C/C++, Fortran
`gang`
`worker`
`vector`
`seq`
`bind(name)`
`bind(string)`
`nohost`
`dtype`

Summary

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

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)`
  - → 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)

Hands-On

- **Introduction to RWTH GPU Cluster environment**

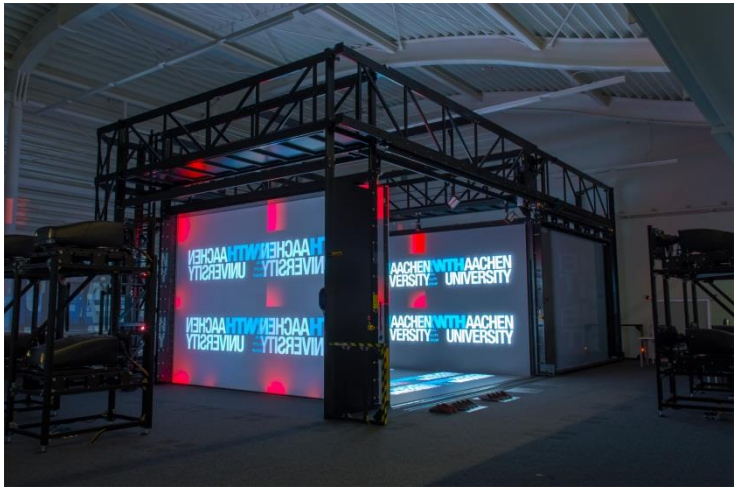
- **Write your first OpenACC program (Jacobi solver)**
  - → Use **`parallel`** and **`loop`** directives
  - → Follow TODOs in **`GPU/exercises/task1`**

The RWTH GPU Cluster

Foto: C. Iwainsky

GPU Cluster: 57 Nvidia Quadro 6000

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


aixCAVE, VR, RWTH Aachen, since June 2012

Hardware & Software Stack

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

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

Lab: Login to GPU machines

These nodes are available in interactive mode only for the workshop!

- **User name (see badge): `hpclab<XY>`**
- **Password (see *Access to Lab Machine*)**
- **GPU node (see paper snippet): `linuxgpus<AB>`**
- **Jump from frontend node to GPU node: `ssh -Y linuxgpus<AB>`**

- **Remark: GPUs are set to “exclusive mode” (per process)**
  - → Only one person can access GPU
  - → If occupied, e.g. message “all CUDA-capable devices are busy or unavailable”

- **Set `export CUDA_VISIBLE_DEVICES=<no>` to use certain GPU!**
  - → See your printout for the correct number (either 0 or 1)

See what is running: nvidia-smi

```
$> nvidia-smi
Mon Oct 17 12:41:01 2011
+------------------------------------------------------+
| NVIDIA-SMI 2.285.05   Driver Version: 285.05.09      |
|-------------------------------+----------------------+----------------------+
| Nb.  Name                     | Bus Id        Disp.  | Volatile ECC SB / DB |
| Fan   Temp   Power Usage /Cap | Memory Usage         | GPU Util. Compute M. |
|===============================+======================+======================|
| 0.  Quadro 6000               | 0000:02:00.0  Off    |          0           0 |
|  30%   80 C  P0     Off /  Off |   4%  208MB / 5375MB |   99%      E. Process |
|-------------------------------+----------------------+----------------------|
| 1.  Quadro 6000               | 0000:85:00.0  On     |          0           0 |
|  36%   84 C  P8     Off /  Off |   0%   22MB / 5375MB |    0%      E. Process |
|-------------------------------+----------------------+----------------------|
| Compute processes:                                               GPU Memory |
|  GPU  PID     Process name                                       Usage      |
|======================================================================================|
|  0.  30234    nbody                                                196MB  |
+--------------------------------------------------------------------------------------+
```

nvidia-smi -q
Lists GPU details

GPU ID + type

process running on GPU

Jacobi Solver

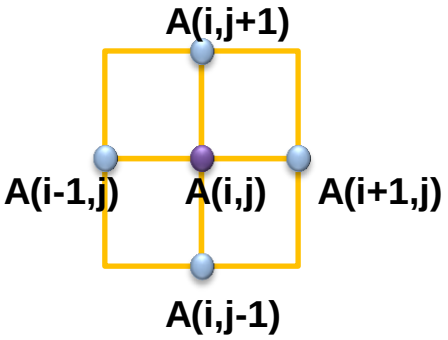
- **Solving Laplace equation (2D) by Jacobi iterative method**

$$\Delta A(x,y) = \nabla^2 A(x,y) = 0$$

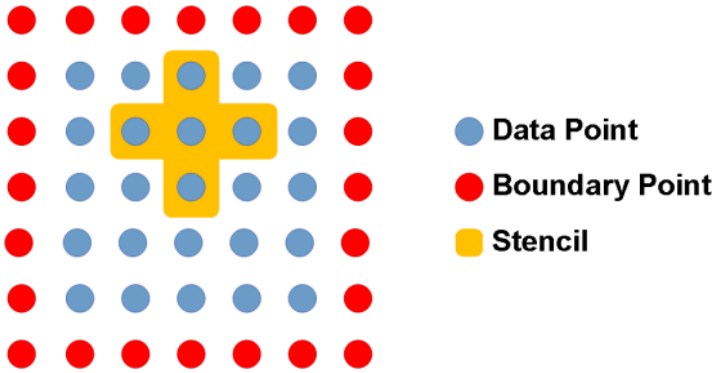
→ Start with guess of objective function A*(x,y)*

→ Use finite difference discretization (5-point-stencil)

$$A_{k+1}(i,j) = \frac{A_k(i-1,j)+A_k(i+1,j)+A_k(i,j-1)+A_k(i,j+1)}{4}$$



→ Iterate over inner elements of 2D grid



- **Repeat until**

→ Maximum of number of iterations reached

→ Computed approximation close to solution

→ Biggest change on any matrix element < tolerance value

Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.11
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	11.36
	OpenACC-Data	**What went wrong?**
	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
  - → Heterogeneous Computing
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Multiple GPUs]
- Comparison of OpenACC & OpenMP Device Constructs

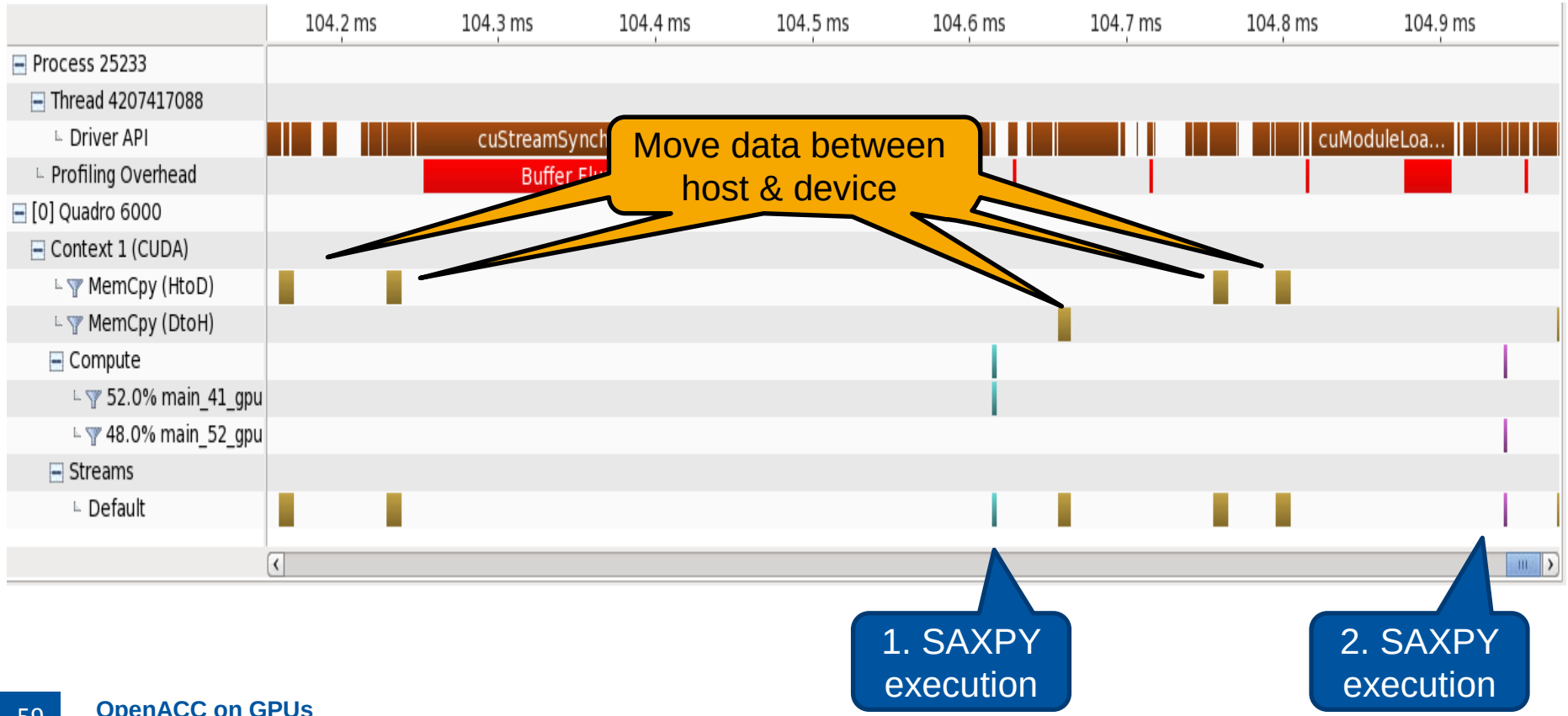
Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
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;
}
```

What went wrong?

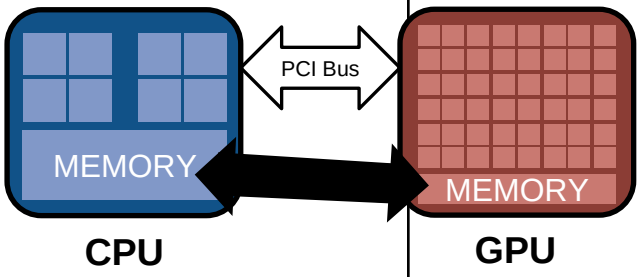
Analysis with the NVIDIA Visual Profiler

- Comes with the NVIDIA Toolkit
- Call: `nvvp`
- Also includes automatic application analysis (utilization,…)



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;
}
```



move x, y to device

move y to host

move x, y to device

move y to host

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

PGI Compiler Feedback

```
main:
27,Generating copyin(x[0:n])
29,Generating present_or_copyin(x[0:n])
   Generating copy(y[0:n])
   Accelerator kernel generated
   31, #pragma acc loop gang,vector(256)
29,Generating NVIDIA code
35,Generating copy(y[0:n])
   Generating present_or_copyin(x[0:n])
   Accelerator kernel generated
   37, #pragma acc loop gang,vector(256)
35,Generating NVIDIA code
```

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)
```

Data Movement

- **Data clauses can be used on `data`, `kernels` or `parallel`**
  - → `copy, copyin, copyout, present, present_or_copy, create, deviceptr`
- **`present_or_copy`, `pcopy`, (also `pcopyin`, `pcopyout`)**
  - → 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]
```

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

  - → 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……
  - → Executes async, see Tuning slides……

C/C++, Fortran
```
host(list)
device(list)
if(cond)
async[(int-expr)]
```

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
```

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

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

- **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
```

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

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

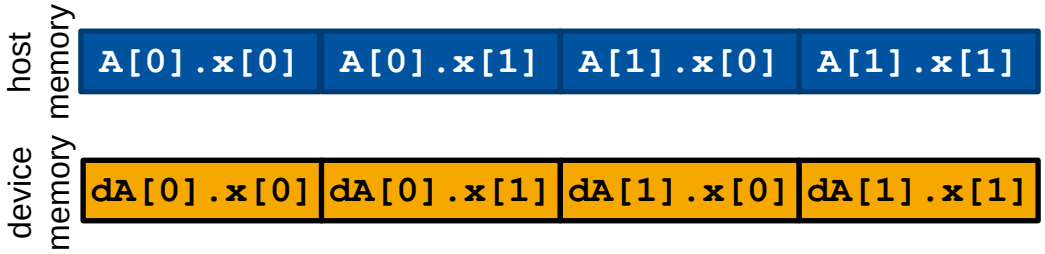
- **Clauses valid for both: `if`, `async`, `wait`**

Deep Copies

OpenACC proposal for complex data management since 11/2014.

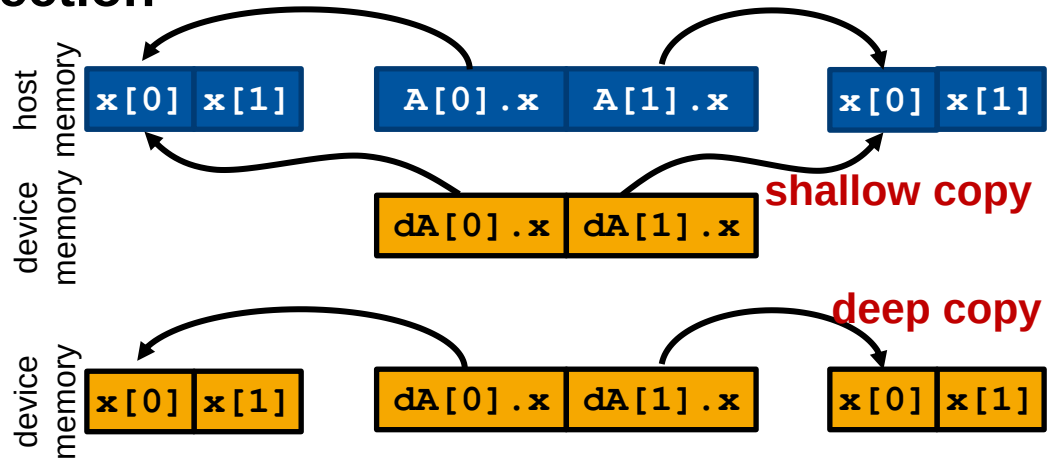
- **Support of a "flat" object model**
  - → Primitive types
  - → Composite types w/o allocatable/pointer members

```
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.

Directives: Data API

- **Data API (1/2)**
  - `d_void* acc_malloc(size_t)`: allocates device memory
  - `void acc_free(d_void*)`: frees device memory
  - `void* acc_copyin(h_void*,size_t)`: allocates device memory & copies data to device
  - `void* acc_present_or_copyin(h_void*,size_t)`: tests whether data is present on device, if not: allocates & copies data
  - `void* acc_create(h_void*,size_t)`: allocates device memory corresponding to host data
  - `void* acc_present_or_create(h_void*,size_t)`: tests whether data is present on device, if not: allocates device memory corresponding to host data
  - `void acc_copyout(h_void*,size_t)`:copies data to host & deallocates device memory
  - `void acc_delete(h_void*,size_t)`: deallocates device memory corresponding to host data

Directives: Data API

Data API (2/2)

- `void acc_update_device(h_void*,size_t)` and `acc_update_self`: updates device data copy corresponding to host data
- `void acc_memcpy_to_device(d_void* dest,h_void* src,size_t)`: copies data from host to device memory
- `void acc_memcpy_from_device(h_void* dest,d_void* src, size_t)`: copies data from device to host memory
- `int acc_is_present(h_void*,size_t)`: tests whether data is present on device
- `void acc_map_data(h_void*,d_void*,size_t)`: maps previously allocated data to the specific host data
- `void acc_unmap_data(h_void*)`: unmaps device data from the specific host data
- `d_void* acc_deviceptr(h_void*)`: returns device pointer associated with a specific host address
- `h_void* acc_hostptr(d_void*)`: returns host pointer associated with a specific device address

Summary

- **Optimizing code with explicit data transfers**
  - → Structured: `data` region
  - → Unstructured: `enter data`, `exit data`
  - → Within data environment: `update`
  - → Clauses: `copy`, `copyin`, `copyout`,…
- **Data API for deep copies (and more)**
- **Compiler and Runtime feedback**
  - → Where is the data moved?
  - → How much time is spend 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`**

Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.11
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	11.36
	OpenACC-Data	1.15
	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

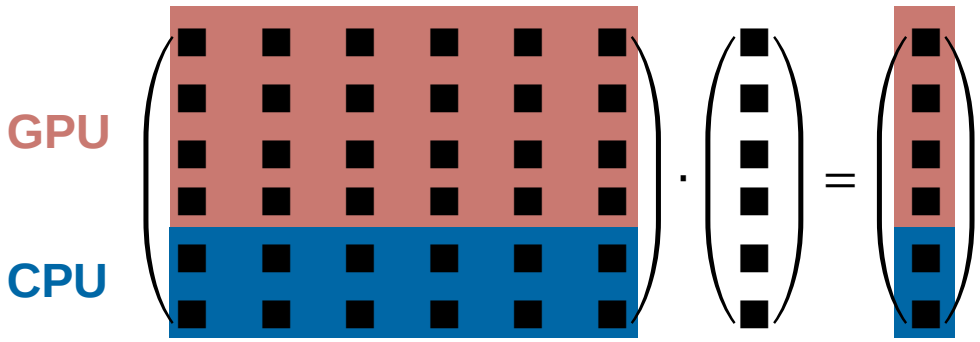
- **OpenACC Advanced**
  - → Heterogeneous Computing
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [Multiple GPUs]

- 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`

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

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

// do work on host
```

Heterogenity

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

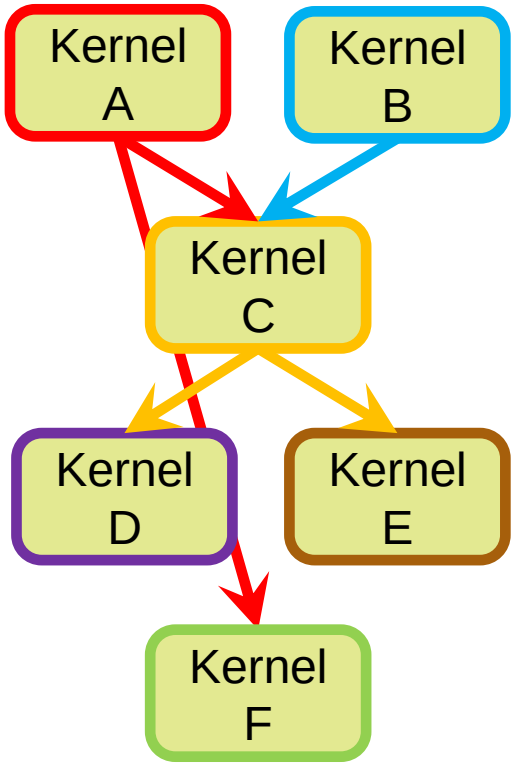
Streams

- **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**
  - → v1.0 (PGI 13.x) : Use int-expr “-1” in async clause or ACC_SYNCHRONOUS=1
  - → v2.0: 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)]
```

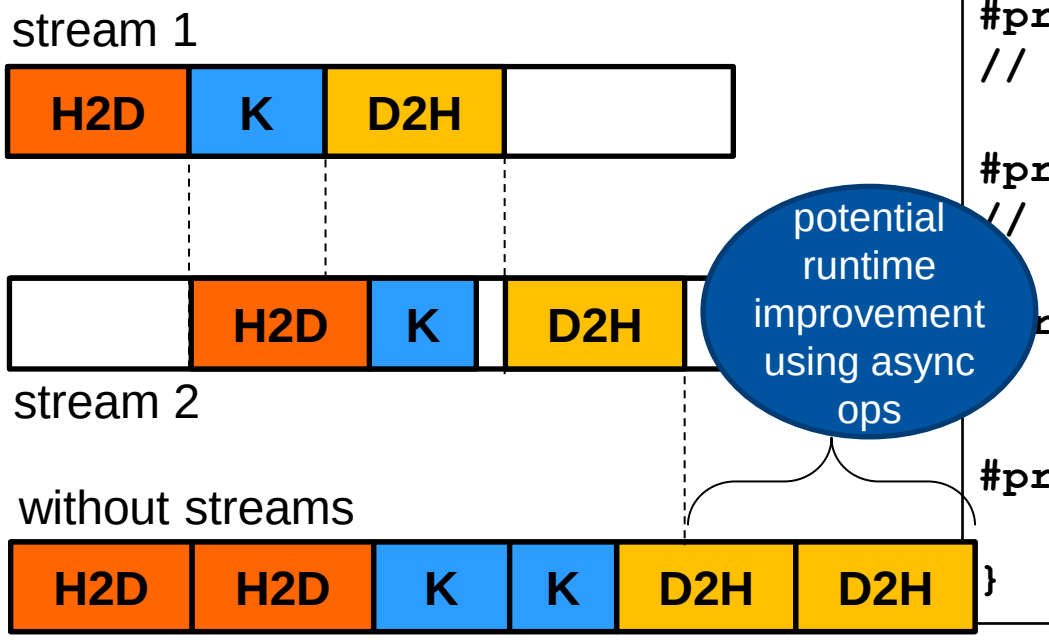
Fortran
```
!$acc wait[(scalar-int-expr)]
```

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

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 Fermi)




```
#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]) async(2)

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

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

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

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

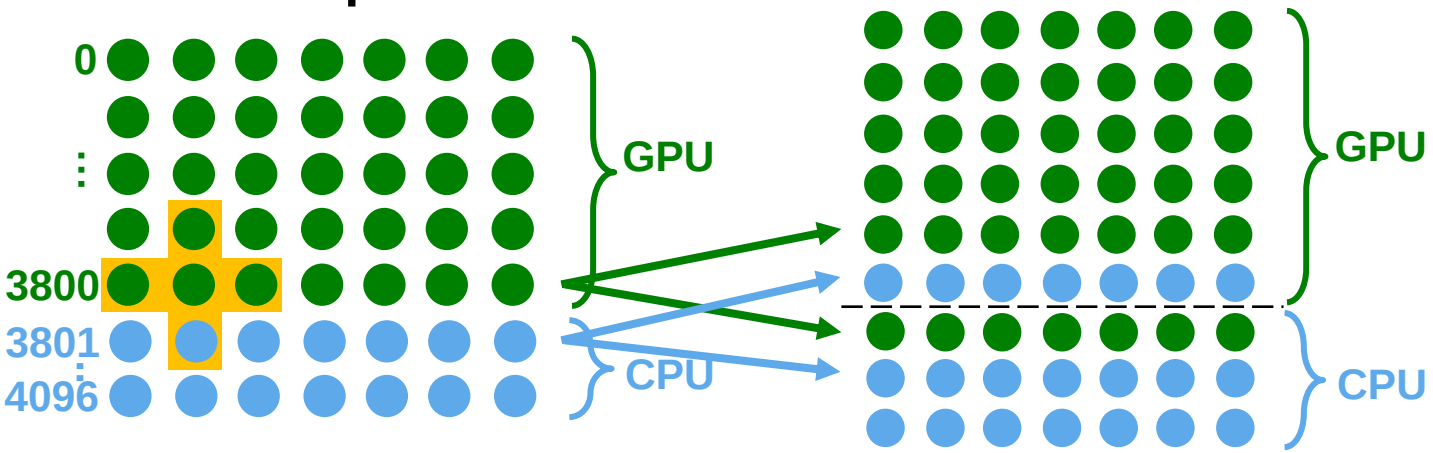
Summary

- **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**

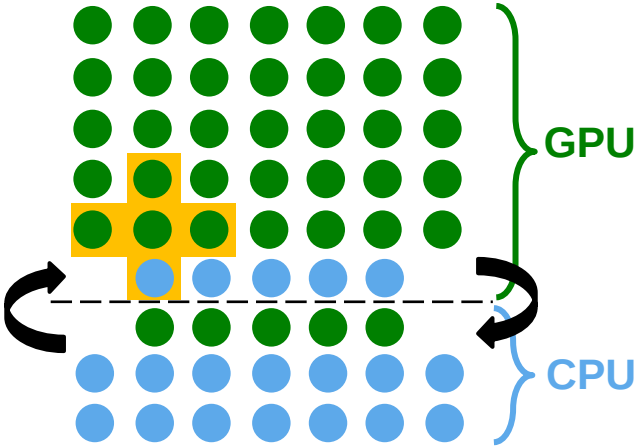
Hands-On

- **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/task3`**

- **Domain decomposition**



- **Halo exchange (each iteration)**



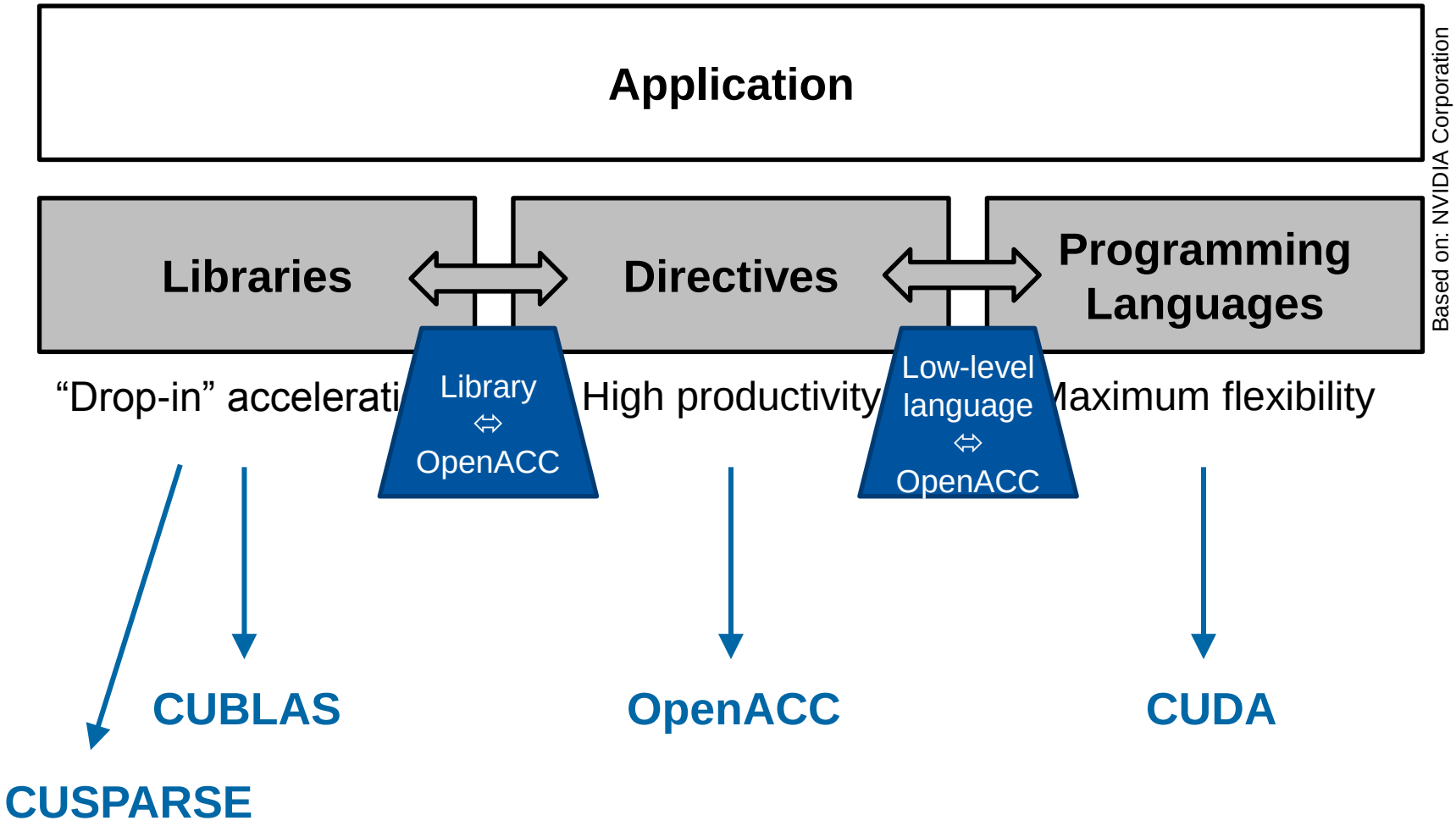
Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.11
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	11.36
	OpenACC-Data	1.15
	OpenACC-Hetero *(1 GPU + 12 OMP threads)*	0.82
	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**
  - → Heterogeneous Computing
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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

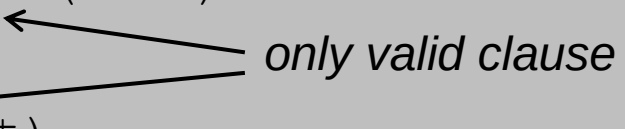
Directives: deviceptr & host_data

- **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)
```
Fortran
```
!$acc host_data use_device(list)

!$acc end host_data
```



Summary

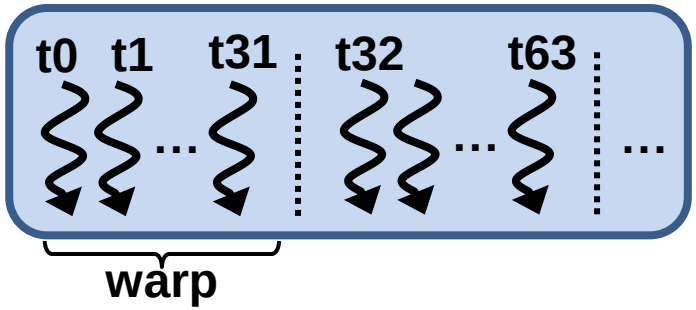
- **OpenACC is interoperable with CUDA and GPU Libraries**
  - → `host_data`: device address available on host
  - → `deviceptr`: device address available in OpenACC code

Agenda

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

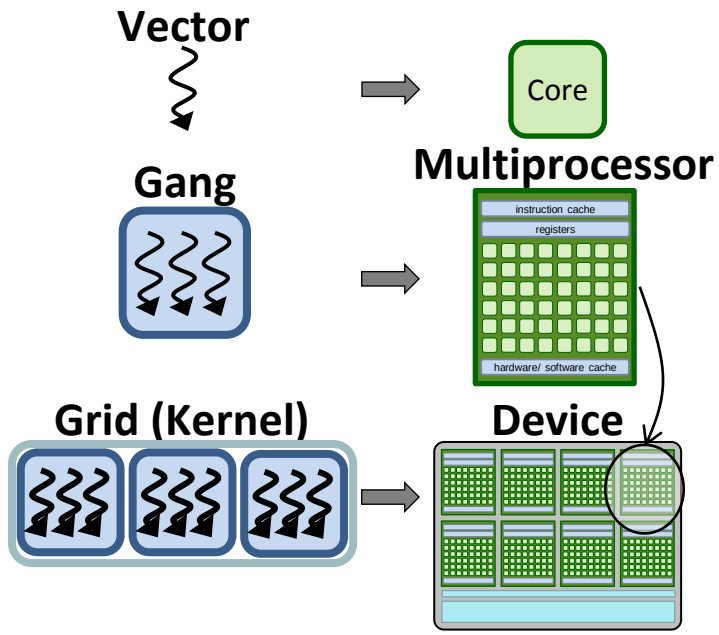
Execution Model

- **Threads execute as groups of 32 threads (warps)**
  - → Threads in warp share same program counter
- → **SIMT architecture**
  - → SIMT = single instruction, multiple threads



- **Possible mapping to CUDA terminology (GPUs)** *compiler dependent*
  - → gang = block
  - → worker = warp
  - → vector = threads
    - →Within block (if omitting worker)
    - →Within warp (if specifying worker)

- **Execution Model**



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)
```

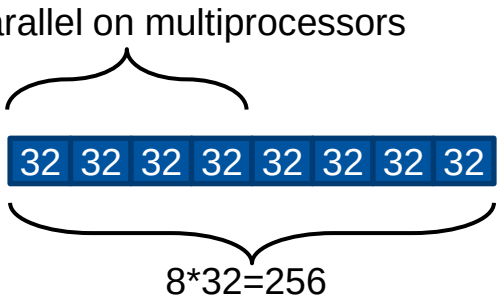
Loop schedule *(in examples)*

```
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;
}
```

`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]) \
                                  num_gangs(blocks) vector_length(bsize)
  all do the same      e.g. int tmp = 5;
#pragma acc loop gang
  workshare (w/o barrier)
#pragma acc loop vector
    workshare (w/ barrier)
  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; }
```



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
  }  }
```

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 portioning 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

Launch configuration

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

→ **Need enough threads to hide latency**

Launch configuration

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

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

Source: Volkov2010

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

e.g. →

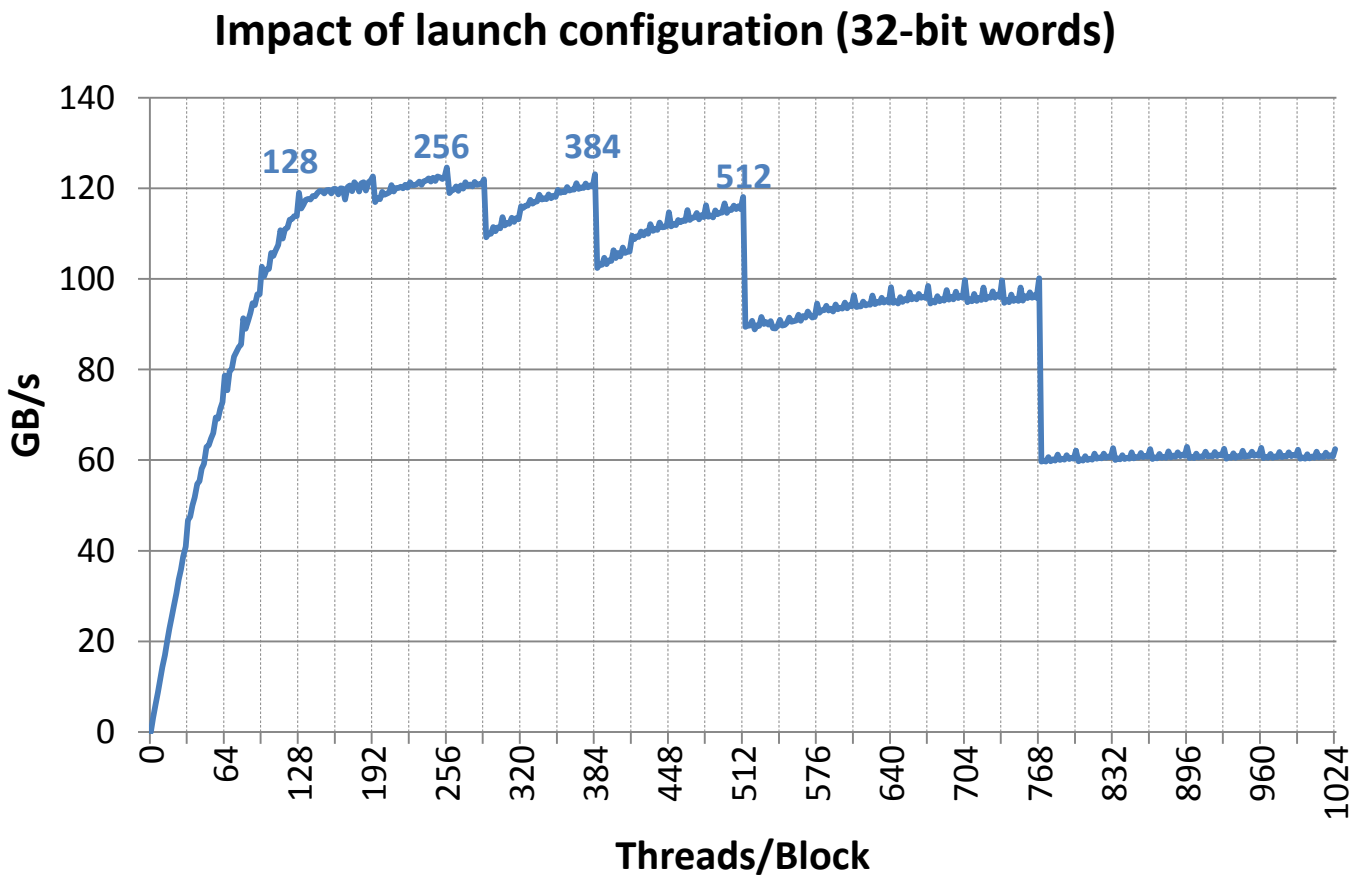
```
float a0= src[id];
// no latency stall
float a1= src[id+blockDim.x];
// stall
dst[id]= a0;
dst[id+blockDim.x]= a1;
// Note, threads don't stall
// on memory access
```

Source: Volkov2010

Launch configuration

- **Maximizing global memory throughput**

  → Example program: increment an array of ~67M elements

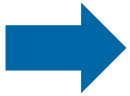


- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

Launch configuration

- **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 > #MPs`**
    - → MPs have at least one block to execute
  - → **`#blocks / #MPs > 2`**
    - → MP can concurrently execute up to 8 blocks
    - → Blocks that aren't waiting in barrier keep hardware busy
    - → Subject to resource availability (registers, smem)
  - → **`#blocks > 50`** to scale to future devices
  - → Most obvious: **`#blocks * #threads = #problem-size`**
    - → Multiple elements per thread may amortize setup costs of simple kernels:

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



Launch MANY threads to keep GPU busy!

Agenda

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

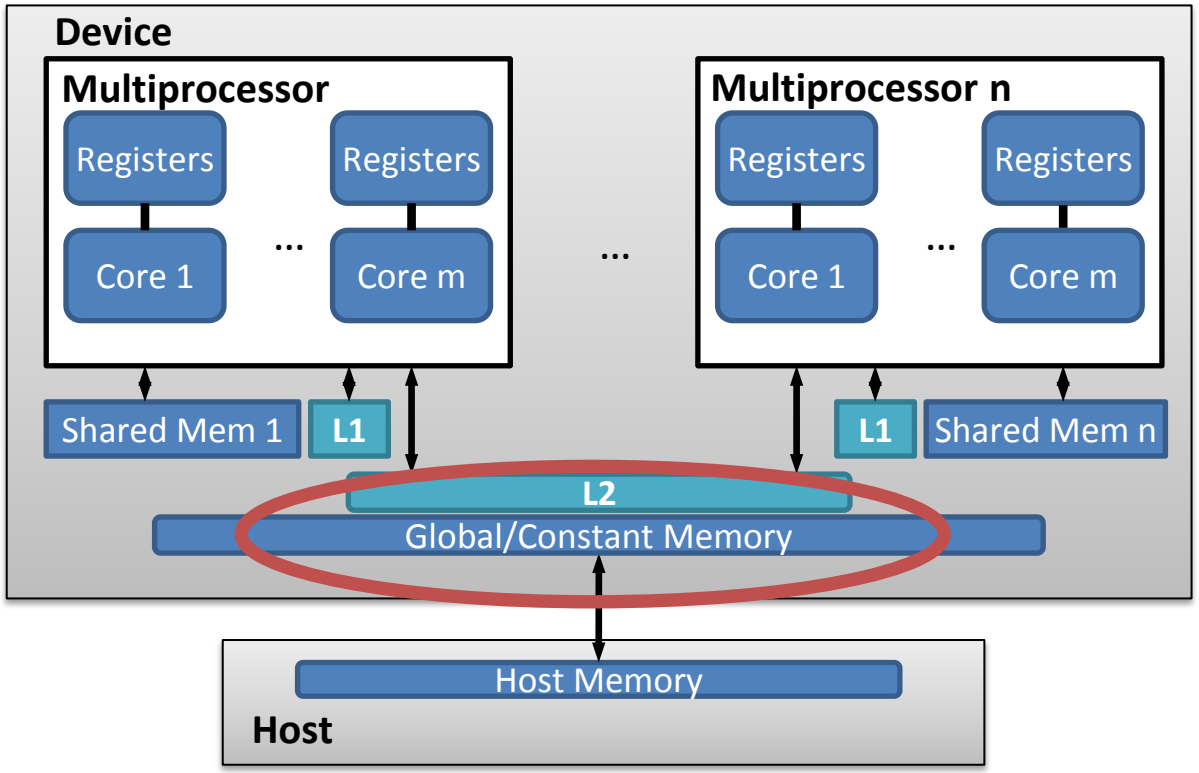
Background information

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

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

Recap memory hierarchy

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



gmem access

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

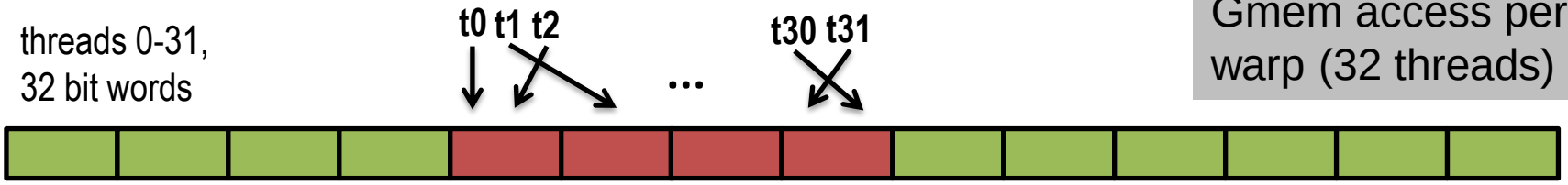
	Caching	Non-caching
Enabling by	default (Fermi)	Experimental compiler-option PGI: `-Mx,180,8`
Attempt to hit:	L1 → L2 → gmem	L2 → gmem (no L1: invalidate line if it's already in L1)
Load granularity	128-byte line	32-byte line

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

→ Threads in a warp provide memory addresses

→ Determine which lines/segments are needed

→ Request the needed lines/segments

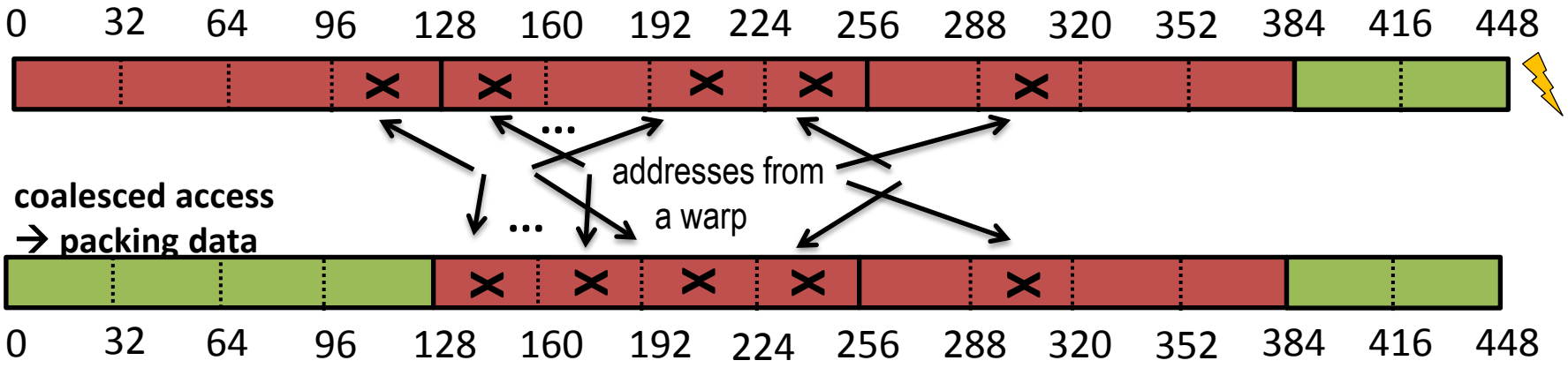


Gmem access per warp (32 threads)

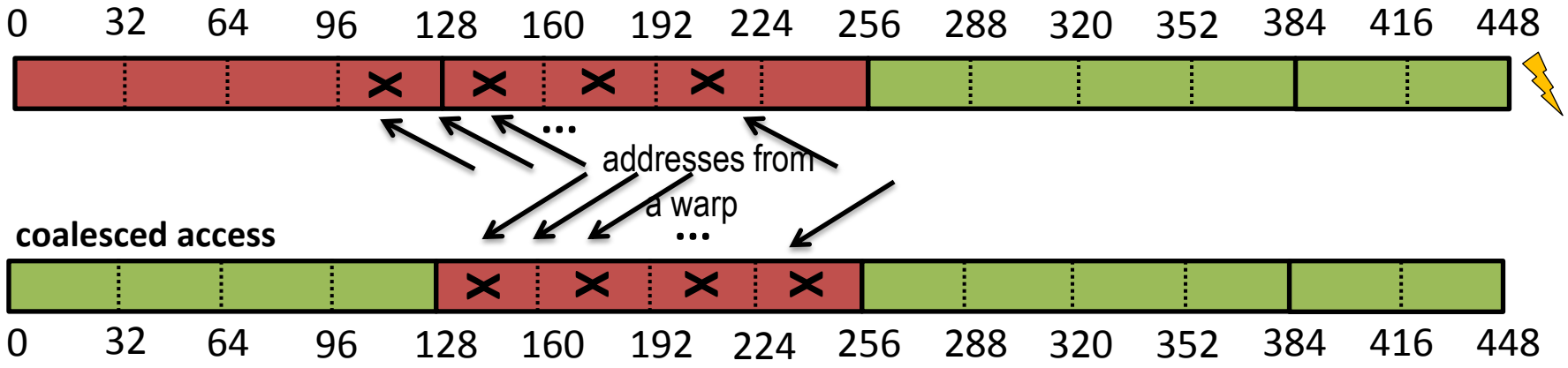
gmem throughput – coalescing

- **Range of accesses**

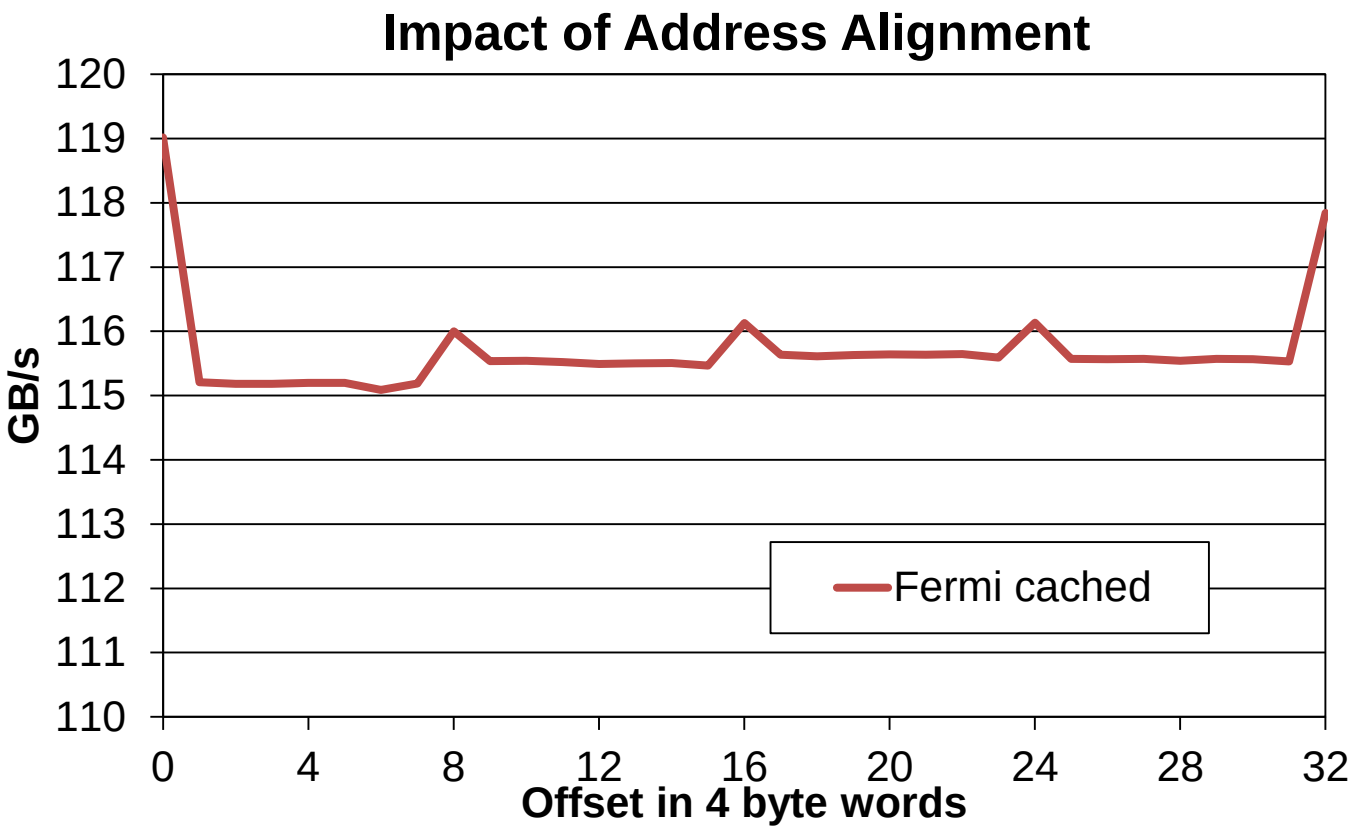
caching loads



- **Address alignment**



Impact of address alignment



Example program:

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

→ **Misaligned accesses can drop memory throughput**

Data layout matters (improve coalescing)

- **Example: AoS vs. SoA**

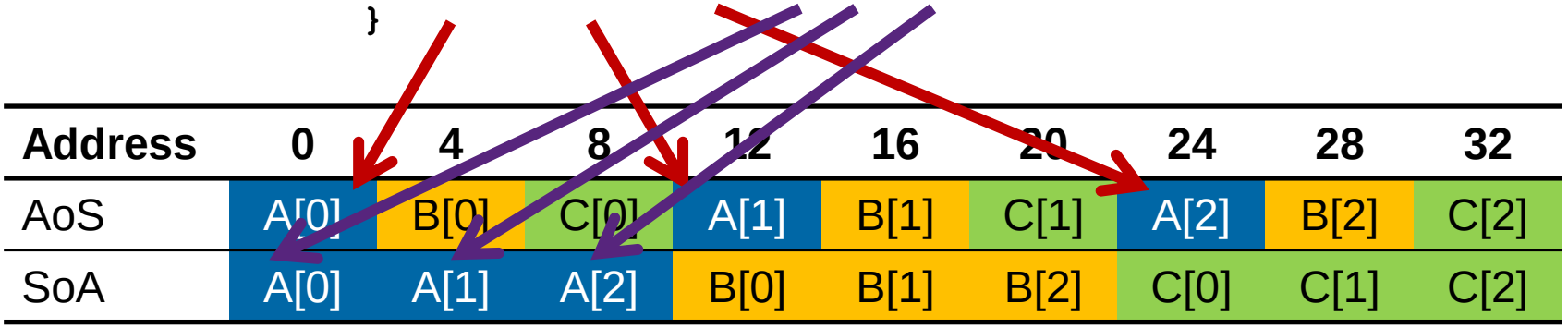
Array of Structures (AoS)

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

Structure of Arrays (SoA)

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

```
#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)

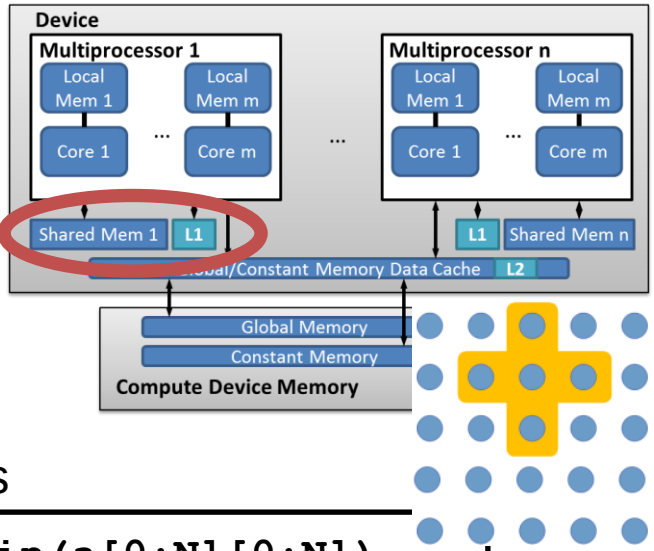
- **Have enough concurrent accesses to saturate the bus**
  - → Process several elements per thread
  - → Launch enough threads to cover access latency

Agenda

- Introduction to GPGPUs
  - → Motivation & Overview
  - → GPU Architecture
- OpenACC Basics
  - → Motivation & Overview
  - → Offload Regions
  - → Data Management
- **OpenACC Advanced**
  - → Heterogeneous Computing
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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. It is not yet supported by PGI.

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**
  - → Heterogeneous Computing 
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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++

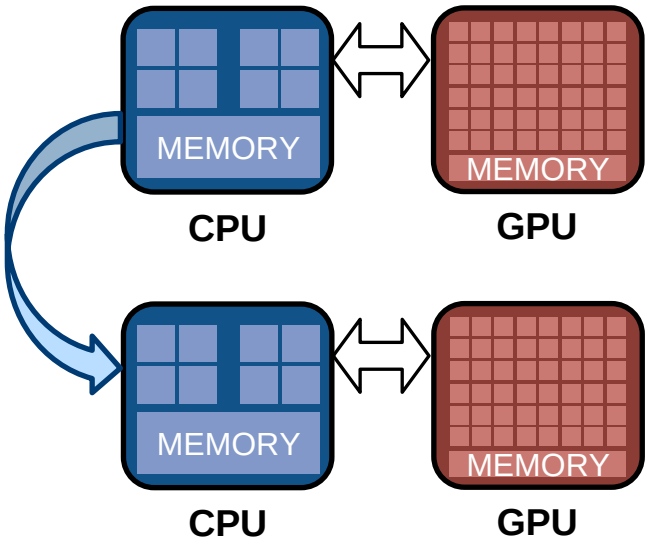
```
#pragma acc <op> device_type(device-type list) (clauses)
#pragma acc <op> device_type(*) (clauses)
```

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

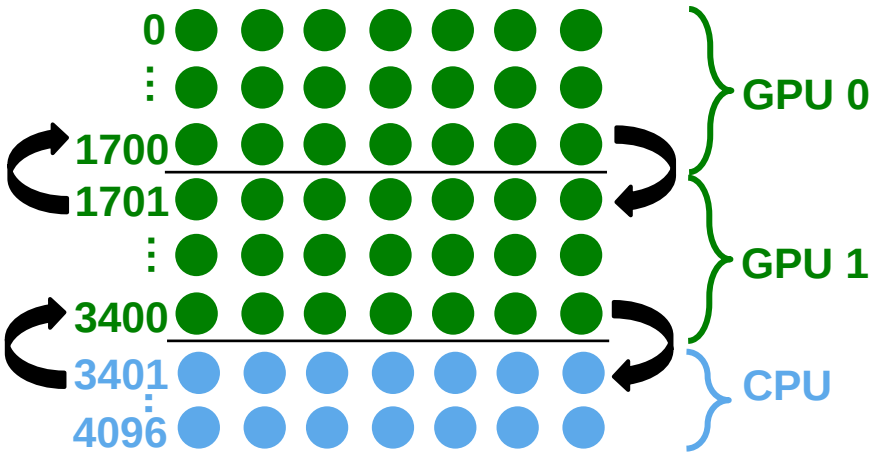
```
MPI_Recv(r_buf,size,MPI_CHAR,
    0,tag,MPI_COMM_WORLD,&stat);

#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/task4`**



Lab Results

Hardware	Version	Runtime [s]
2x Intel X5650 @ 2.67GHz (Intel Westmere, 12 cores)	OpenMP	2.11
NVIDIA Quadro 6000 (Fermi, cc 2.0)	OpenACC-Offload	11.36
	OpenACC-Data	1.15
	OpenACC-Hetero *(1 GPU + 12 OMP threads)*	0.82
	OpenACC-MultiGPU *(2 GPUs + 12 OMP threads)*	0.62

Agenda

- **Introduction to GPGPUs**
  - → Motivation & Overview
  - → GPU Architecture
- **OpenACC Basics**
  - → Motivation & Overview
  - → Offload Regions 
  - → Data Management
- **OpenACC Advanced**
  - → Heterogeneous Computing 
  - → Interoperability with CUDA & GPU Libraries
  - → [Loop Schedules & Launch Configuration]
  - → [Maximize Global Memory Throughput]
  - → [Caching & Tiling]
  - → [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 lernt into OpenMP standard
- **07/2013: OpenMP 4.0**
  - → Accelerator support
  - → SIMD,…

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.

OpenMP™

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

Overview on OpenACC & OpenMP Device Capabilities

- **Directive-based accelerator programming: C/C++, Fortran**
- **OpenACC**
  - → Initiated by Cray, CAPS, PGI, NVIDIA
  - → 2011: specification v1.0
  - → 2013: specification v2.0
- **OpenMP**
  - → Broad (shared-memory) standard
  - → 2013: specification v4.0

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
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

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

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.

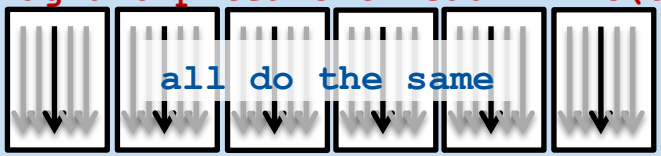
SAXPY (GPU)

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;
}
```

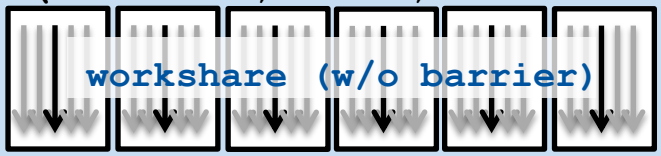
OpenMP

```
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)
```

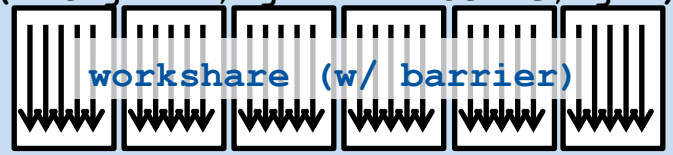


league = set of thread teams

```
#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)

RWTH AACHEN UNIVERSITY

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 \
 loop gang vector
 for (int i = 0; i < n; i++){
     y[j] = a*x[j] + y[j];
  }
  free(x); free(y); return 0;
}
```

GPU

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;
}
```

```
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;
}
```

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 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]

RWTH AACHEN UNIVERSITY

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);
}
}
```

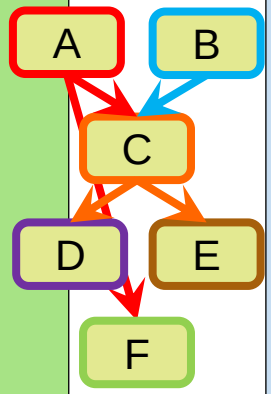
Missing data: Fix-Up
Implicit check for data;
move if not there

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
```

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?**

Thank you for the attention!