Dr. Karl Fürlinger
Lehrstuhl für Kommunikationssysteme und
Systemprogrammierung

Josef Weidendorfer, LRZ

GPU Hardware and OpenMP on Heterogeneous Architectures

Top 10 of the Top500 List (Nov. 2022)

Rank	System	Rmax (TFlop/s)	
1	**Frontier** - AMD EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, DOE/SC/ORNL, United States	1,102,000	**AMD MI250X**
2	**Fugaku** - Fujitsu ARM A64FX 48C 2.2GHz, Tofu interconnect D, RIKEN Center for Computational Science Japan	442,010	
3	**LUMI** - AMD EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, EuroHPC/CSC, Finland	309,100	**AMD MI250X**
4	**Leonardo** - Xeon 32C 2.6GHz, NVIDIA A100, Quad-rail NVIDIA HDR100 Infiniband, EuroHPC/CINECA, Italy	174,700	**Nvidia Ampere**
5	**Summit** - IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, EDR Infiniband, DOE/SC/Oak Ridge National Laboratory United States	148,600	**Nvidia Volta**
6	**Sierra** - IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, EDR Infiniband, DOE/NNSA/LLNL United States	94,640	**Nvidia Volta**
7	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz NRCPC National Supercomputing Center in Wuxi China	93,014	
8	**Perlmutter** - HPE Cray, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, LBNL United States	70,870	**Nvidia Ampere**
9	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, HDR Infiniband, Nvidia Corporation United States	63,460	**Nvidia Ampere**
10	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center, China	61,444	

Accelerators in the Top 500 List

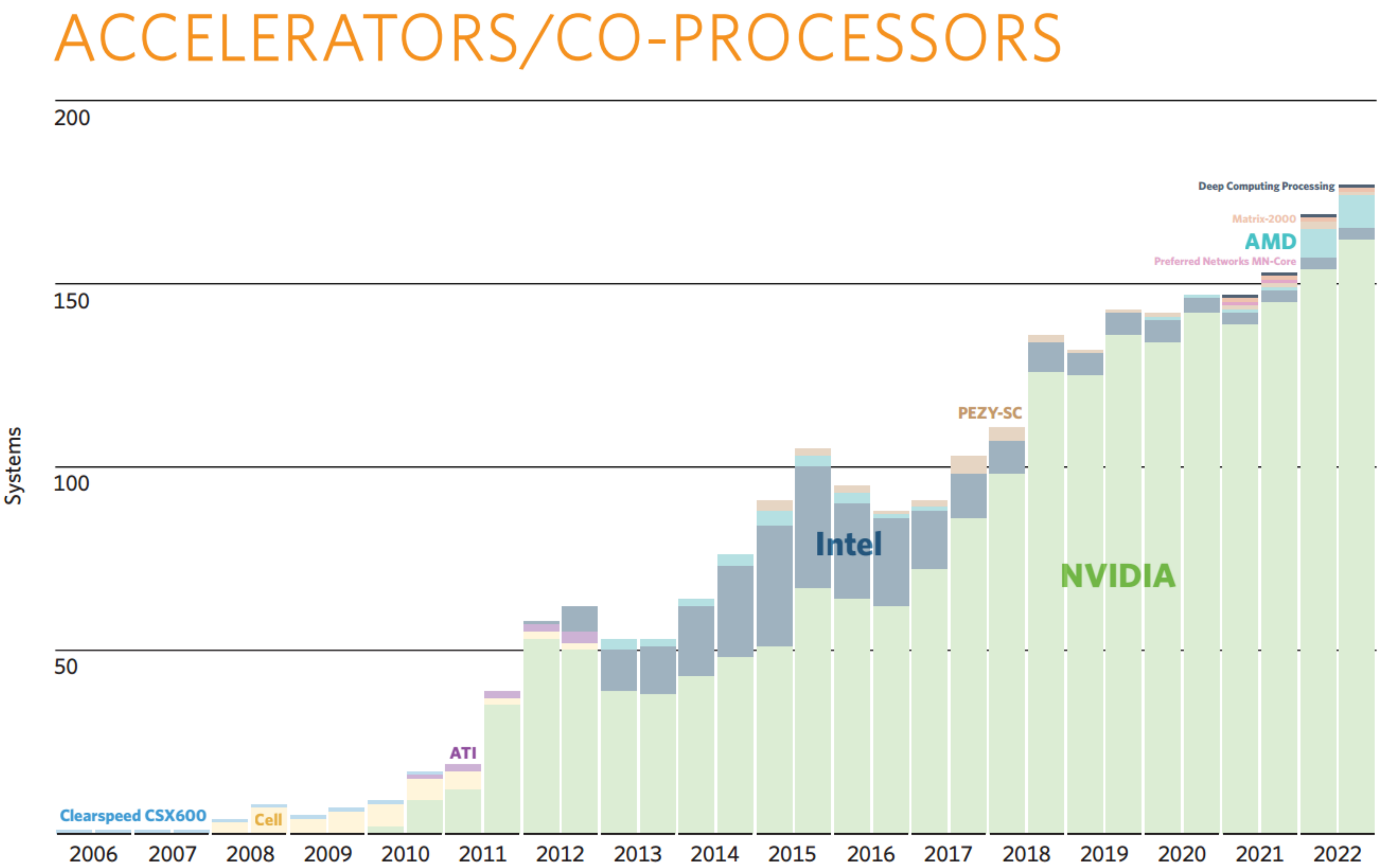
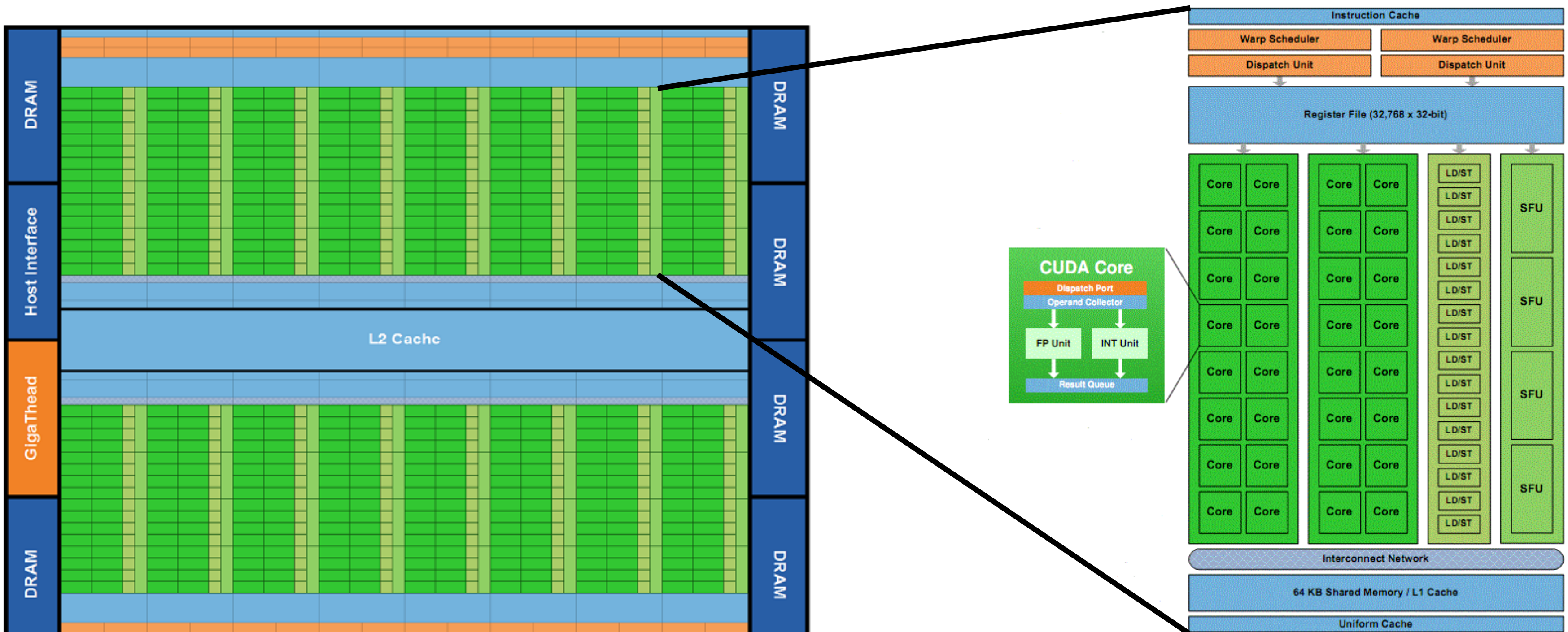


Image Source: http://top500.org/lists/2022/11/

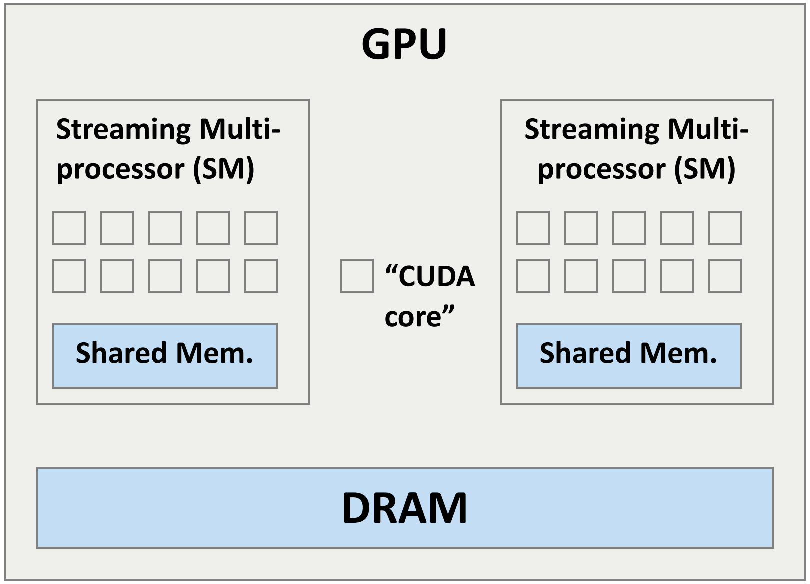
NVidia GPU Hardware (Fermi Generation – Old!)

- GPU consists of several Streaming Multiprocessors (SMs)
  - Up to 16 on Fermi cards
- Streaming Multiprocessor
  - Consists of a number of CUDA cores or Thread Processors



Fermi Streaming Multiprocessor (SM)

Hierarchical Organization (Nvidia Terminology)



- Threads on different SMs can not (cheaply) communicate and synchronize
- **Coalesced** memory access is important
  - Thread i accesses Array[i]
- Hierarchical HW is also reflected in the programming model

- CUDA
  - **Grid** consists of several thread blocks
  - **Thread block** consists of several CUDA threads
  - **CUDA threads** (scalar execution contexts) are managed in groups of 32, called **warps**

- Thread blocks are independent units of execution, can be scheduled in any order

Threads, Warps, Thread Blocks, Grids

- CUDA example

```
__global__
void product(int n, double *A, double *B, double *C)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) A[i] = B[i]*C[i];
}

//n=8192, 512 threads per block => 16 thread blocks
product<<<16, 512>>>(8192, A, B, C);
```

- **Hierarchical** organization
  - A thread block consists of a number of threads
  - A grid consists of a number of thread blocks
  - **SIMD thread** = **warp** in CUDA terminology
  - 32 CUDA threads per warp (a constant, determined by current HW)

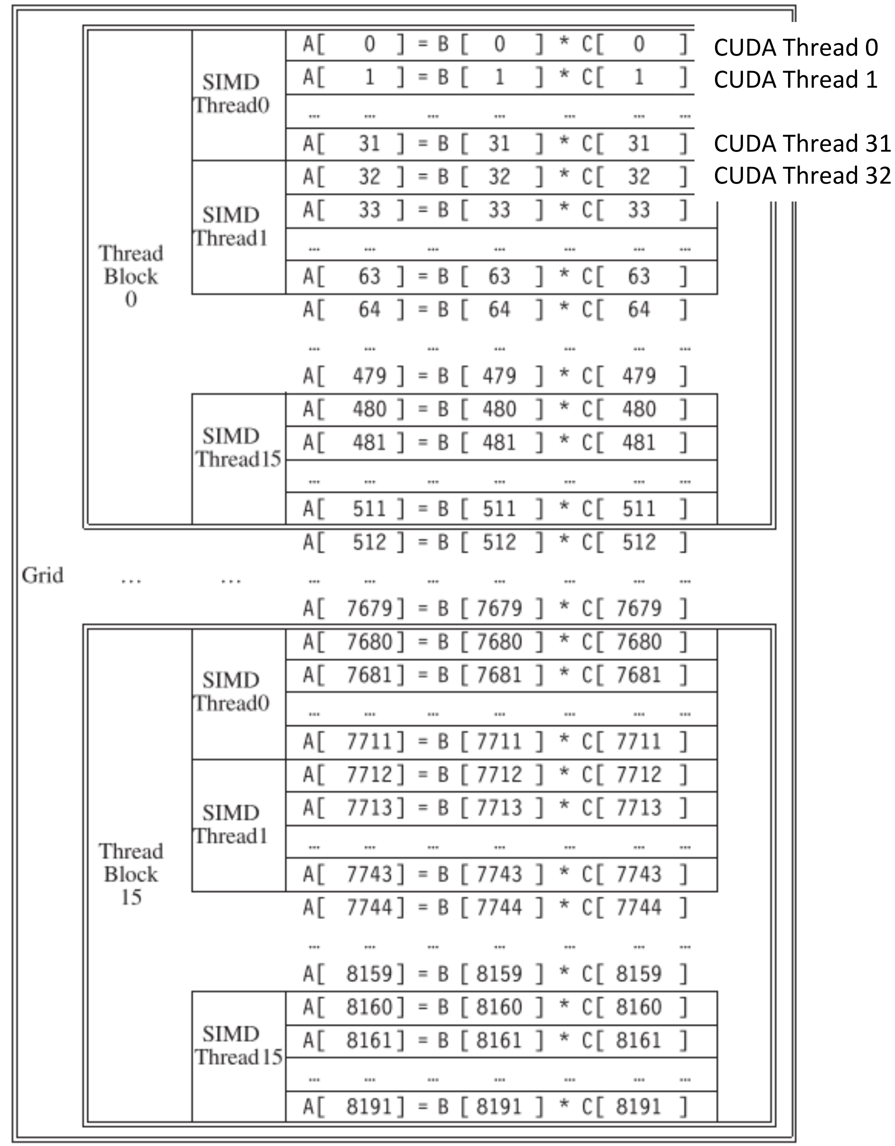


Image Source: CAAQA (6th Edition)

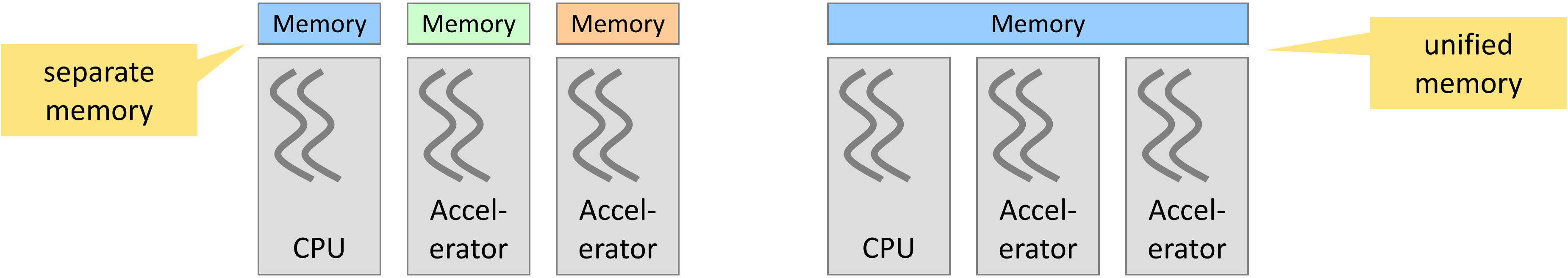
- OpenMP on Heterogeneous Architectures

Heterogeneous Architectures (1)

- Heterogeneous: more than one type of compute resource

- Most often
  - One **general purpose processor** (aka. CPU, the "host")
  - One or more **special purpose processors** (aka. accelerators, "devices", OpenMP: "target devices")

- Memory can be **unified** or **separate**
  - Programming model must support the "worst case", i.e., separate memory



In both cases, threads cannot migrate between the devices!

Heterogeneous Architectures (2)

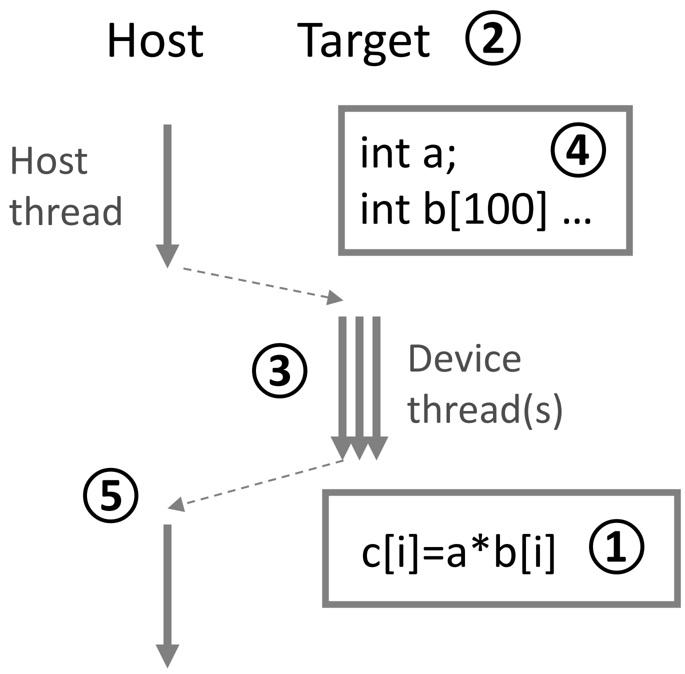
- Benefits of special purpose processors
  - Can execute some types of computation quicker and more energy efficient

- Examples:
  - GPUs: good at massive data parallel operations
  - DSPs: good at signal processing applications
  - FPGAs: good at data-flow type operations

- Challenges for programming
  - We don’t want to maintain separate versions of source code for different parts of the application (OpenMP+CUDA+OpenCL+…)
  - Ideally: maintain just one set of source files, specify which part to execute on the device, let compiler optimize for accelerators
  - The device might have a separate memory space, so we also need to explicitly specify the data environment on the device

OpenMP Offloading Basics

- OpenMP model is **host-centric**
  - Execution begins on the host
  - The execution of **target regions** is **offloaded** to target devices



- Things we need to specify:
  - ① **Which code** to execute on the target device
  - ② **Which target device** to use
  - ③ **Execution configuration** on the target device (how many teams, how many threads)
  - ④ The **device data environment** for the execution on the target device
  - ⑤ What happens on the **host side** in the meantime: wait, do other work, etc.

OpenMP Offloading Example (1)

```
int N=100;
double b[N], c[N];
double a = 1.2;

for( int i=0; i<N; i++ ) {
  b[i]=(double)(i);
}

#pragma omp target map(a,b,c)
{
  for(int i=0; i<N; i++) {
    c[i] = a * b[i];
  }
}

for( int i=0; i<N; i++ ) {
  printf("%f ", c[i]);
}
printf("\n");
```

Output:

```
0.000000 1.200000 2.400000 3.600000
... 118.800000
```

- The **target directive** specifies the code to execute on the device, the **target region**
  - In this example, only one device thread is used
  - Parallelism and worksharing constructs can be used in the target region
  - The default device is used for the execution, can be changed with the **device clause**
- In this case the host is **waiting** for the target region
  - **nowait clause** can be used to avoid the waiting

OpenMP Offloading Example (2)

```
int N=100;
double b[N], c[N];
double a = 1.2;

for( int i=0; i<N; i++ ) {
  b[i]=(double)(i);
}

#pragma omp target map(a,b,c)
{
  for(int i=0; i<N; i++) {
    c[i] = a * b[i];
  }
}

for( int i=0; i<N; i++ ) {
  printf("%f ", c[i]);
}
printf("\n");
```

Output:

```
0.000000 1.200000 2.400000 3.600000
... 118.800000
```

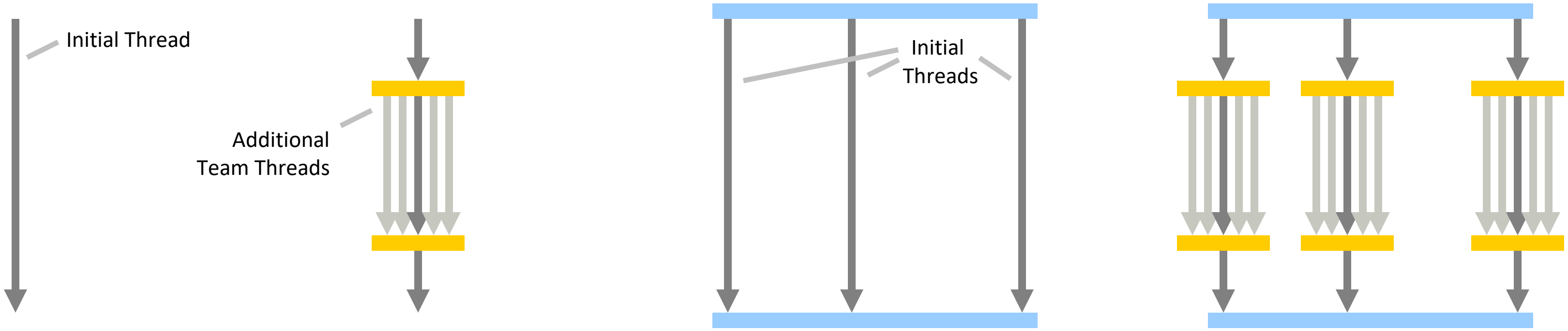
- The **device data environment** (DDE) contains all the variables accessed in a target region
  - i is **private** (by default)
  - N is **firstprivate** (by default)
  - a,b,c are **mapped** variables
  - There are default rules and ways to override them with clauses
- Mapped variables
  - "**mapped**" is a new data sharing type, available only for offloading
  - Generalization of "**shared**", to support both unified and separate memory
  - "**shared**" is **not available** at all for offloading

Overview of OpenMP Constructs for Offloading

- To offload code and specify the execution configuration
  - `#pragma omp` **`target`** — offload execution to the target device
  - `#pragma omp` **`teams`** — start a league of teams
  - `#pragma omp` **`parallel`** — start a team of threads
  - `#pragma omp` **`distribute`** — schedule loop iterations to teams
  - `#pragma omp` **`for`** — schedule loop iterations to threads
  - `#pragma omp` **`simd`** — schedule loop iterations to vector lanes

- To specify the data environment
  - `#pragma omp` **`target`**
  - `#pragma omp` **`target data`** — like target but no code is offloaded
  - `#pragma omp` **`target enter data`** — standalone version of target data
  - `#pragma omp` **`target exit data`** — standalone version of target data
  - `#pragma omp` **`target update`**
  - `#pragma omp` **`declare target`** — make vars and functs. available on the target

Execution Configuration on the Target Device




```
#pragma omp target
{
}
```

```
#pragma omp target
#pragma omp parallel
{
}
```

```
#pragma omp target
#pragma omp teams
{
}
```

```
#pragma omp target
#pragma omp teams
#pragma omp parallel
{
}
```

Note:

```
#pragma omp target
#pragma omp teams
```

Can be shortened to:

```
#pragma omp target teams
```

- The *teams* directive can only appear closely nested in a target directive and can not appear anywhere else!

Execution Configuration Example (1)

```
#pragma omp declare target
void print() {
  printf("Thread [%d/%d] in team [%d/%d]\n",
           omp_get_thread_num(), omp_get_num_threads(),
           omp_get_team_num(), omp_get_num_teams());
}
#pragma omp end declare target
```

“declare target” is used to make functions (and variables) available on the accelerator

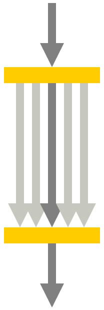
Returns the number of teams and the team-id

```
#pragma omp target
{
  print();
}
```

```
Thread [0/1] in team [0/1]
```

1 thread x 1 team =
1 thread in total

```
#pragma omp target
#pragma omp parallel
{
  print();
}
```



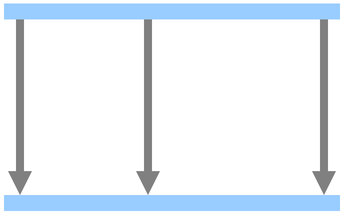
```
Thread [3/8] in team [0/1]
Thread [0/8] in team [0/1]
Thread [1/8] in team [0/1]
Thread [6/8] in team [0/1]
Thread [7/8] in team [0/1]
Thread [2/8] in team [0/1]
Thread [4/8] in team [0/1]
Thread [5/8] in team [0/1]
```

8 threads x 1 team =
8 threads in total

num_threads() clause
to set the number of threads
for the parallel region

Execution Configuration Example (2)

```
#pragma omp target
#pragma omp teams
{
  print();
}
```



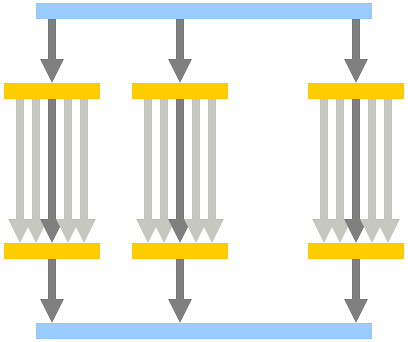
```
Thread [0/1] in team [0/15]
Thread [0/1] in team [3/15]
Thread [0/1] in team [11/15]
...
Thread [0/1] in team [13/15]
Thread [0/1] in team [6/15]
Thread [0/1] in team [14/15]
Thread [0/1] in team [7/15]
```

1 thread x 16 teams = 16 threads in total

num_teams() clause
to set the number of teams

max_threads() clause
to set the maximum number of threads in a contention group

```
#pragma omp target
#pragma omp teams
#pragma omp parallel
{
  print();
}
```

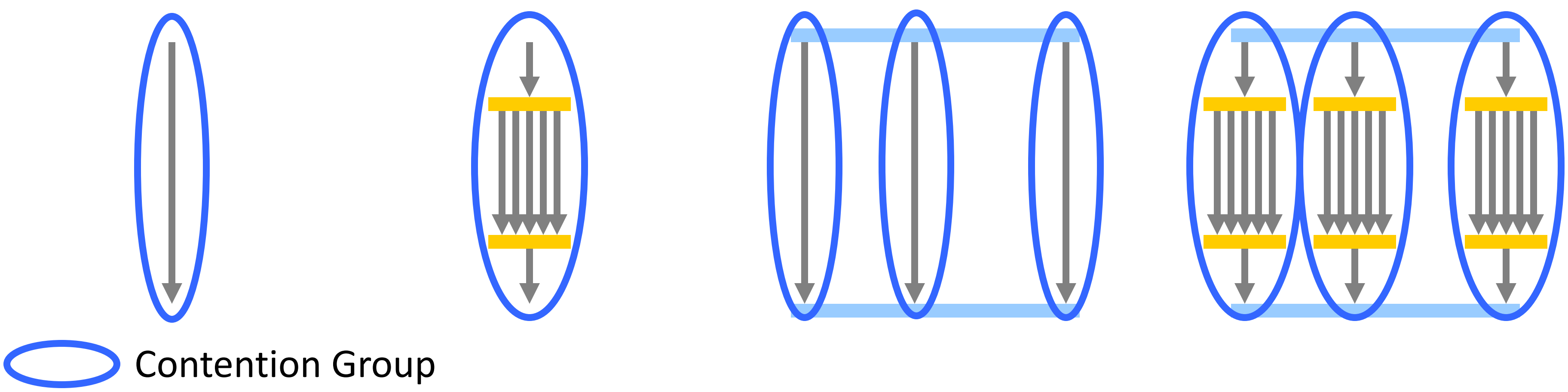


```
Thread [0/8] in team [9/15]
Thread [6/8] in team [8/15]
Thread [2/8] in team [8/15]
Thread [3/8] in team [9/15]
Thread [4/8] in team [9/15]
Thread [2/8] in team [9/15]
Thread [1/8] in team [4/15]
Thread [7/8] in team [9/15]
Thread [4/8] in team [8/15]
Thread [3/8] in team [4/15]
Thread [0/8] in team [8/15]
...
```

8 threads x 16 teams = 128 threads in total

Initial Threads and Contention Groups

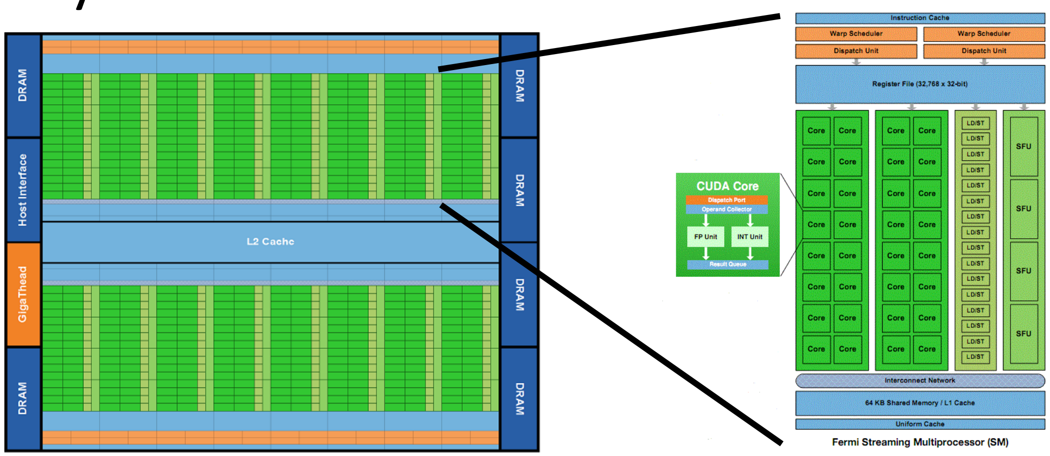
- The target directive **always** creates a **new initial thread**
  - This thread is either running on the accelerator (usually), or
  - running on the host (in the case of **host fallback**, e.g., when no target device is available)
- Contention groups
  - All descendants of an initial thread form a **contention group**
  - Two different initial threads are never in the same contention group
  - No synchronization is possible between contention groups
  - Communication is only possible by using **atomic variables**



Why Leagues of Teams and Contention Groups?

- Remember the hierarchical structure of GPUs:
  - The whole GPU consists of several…
  - **Streaming Multiprocessors** (SMs), which in turn consist of several…
  - **Streaming Processors** (SPs, or "CUDA Cores")

- Threads executing on the *same* SM have access to *shared memory* and can synchronize effectively
  - The whole GPU executes a league of teams
  - Each team is executed on a particular SM
  - The threads in a team are executed on the SPs



The Target Directive

- The target directive and clauses

```
#pragma omp target [clause ...]
  structured block

Clauses:

if ([target:] scalar-expression)
device (integer-expression)

map ([map-type-modifier[,]]
      map-type:] list)
private (list)
firstprivate (list)
is_device_ptr (list)
defaultmap(tofrom :scalar)

nowait
depend (dependence-type: list)
```

Control whether offloading actually happens or not, and which target device should execute the code.

Control the device data environment (DDE).

Control the execution on the host side (waiting, using dependencies – similar to tasks).

The Device Data Environment (DDE)

- The DDE is the set of variables present on the device
  - An **original variable** (on the host) is mapped to a **corresponding variable** (on the device)
  - Updates between original and corresponding variable happen **automatically** and can also be triggered **manually**

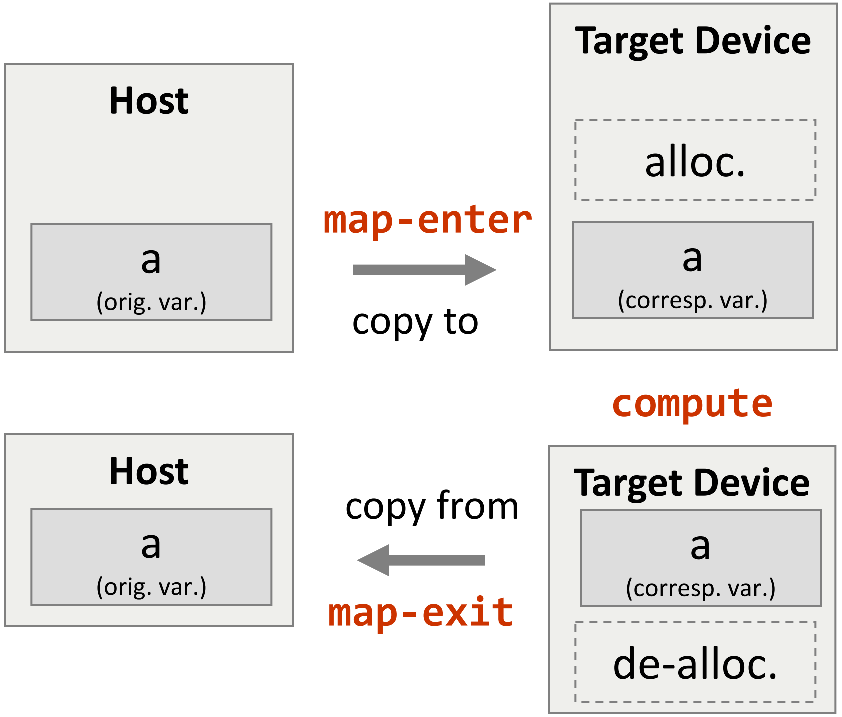
- Typical usage and default behavior:
  - 3 Phases: map-enter / compute / map-exit

```
int a[100]; // host memory

#pragma omp target map(a)
{     // (1) map-enter phase

  ... // (2) compute phase

}     // (3) map-exit phase
```



Controlling Allocation and Copy Operations

- Allocating memory and copying are expensive
  - Both can be avoided when not necessary

- For more flexible data management:
  - Need constructs to extend / manage the lifetime of data independently of offloaded region
  - Need to specify what copy operations should be taking place

- To manage lifetime of data on the device:
  - **target data** construct
  - **target enter data** construct
  - **target exit data** construct

- To manage copy operations:
  - **map types** in the map clause

Target Data Construct

- Similar to a target construct, but no code is executed on the target
  - The code following the construct is executed **on the host**

```
#pragma omp target data [clause...]
  structured block

Clauses:

if ([target:] scalar-expression)
device (integer-expression)

map ([map-type-modifier[,]]
      map-type:] list)
```

- Example

```
#pragma omp target data map(a)
{
  // a is already mapped
  #pragma omp target
  {
    foo(a);
  }
  // a is already mapped
  #pragma omp target
  {
    bar(a);
  }
}
```

Target Enter and Exit Data

- Similar to “target data” but more flexible
  - There is no associated structured block (they are standalone directives)
  - They can appear anywhere, no lexical nesting needed
  - The nowait and depend clauses can be used (e.g. for overlapping data transfers)

```
#pragma omp target enter data [clause...]
#pragma omp target exit  data [clause...]

Clauses:

if ([target:] scalar-expression)
device (integer-expression)

map ([map-type-modifier[,]]
      map-type:] list)

nowait
depend (dependence-type: list)
```

Target Enter and Exit Data Example

- These are similar to target data, but have no associated structured block

```
#pragma omp target enter data map(a)

// a is already mapped
#pragma omp target
{
  foo(a);
}

// a is already mapped
#pragma omp target
{
  bar(a);
}

#pragma omp target exit data map(a)
```

Map Types in the Map Clause

- The map clause allows a fine-grained specification what kind of transfers and allocations are to happen

```
map([map-type:] list)

where map-type is one of
 - to
 - from
 - tofrom
 - alloc
 - release
 - delete
```

- Example:

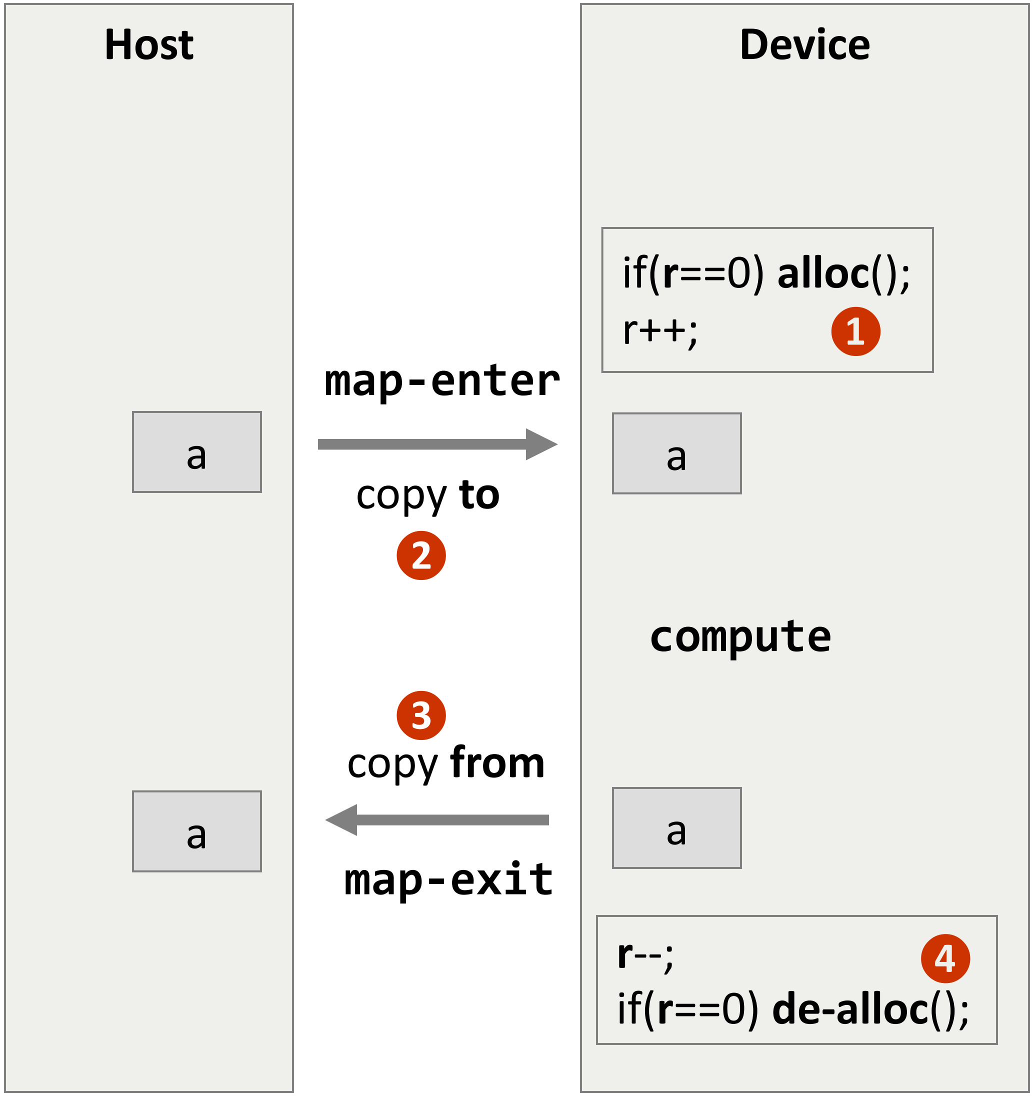
```
#include  <stdlib.h>

void func(float a[1024],
          float b[1024], int t[1024])
{
  #pragma omp target map(from:a) \
          map(to:b) map(alloc:t)
  {
    int i;

    for(i=0; i<1024; i++)
      t[i] = rand()%1024;

    for(i=0; i<1024; i++)
      a[i] = b[t[i]];
  }
}
```

Explanation of Map Types



r… reference count for variable a

Map Type	What happens?
alloc	1
release	4
delete	r := 1, then 4
to	1 2 4
from	1 3 4
tofrom	1 2 3 4

Saxpy Example - OpenMP

```
#pragma omp declare target
void saxpy(int beg, int end, float a,
           float *restrict x, float *restrict y)
{
#pragma omp parallel for simd
  for(int i = beg; i < end; ++i)
    y[i] = a*x[i] + y[i];
}
#pragma omp end declare target
```

"declare target" is used to make functions (and variables) available on the accelerator

```
#pragma omp target enter data map(to:x[0:n], y[0:n])

start = omp_get_wtime();
#pragma omp target
{
  for(int i=0; i<repeat; ++i) {
    saxpy(0, n, 1.2f, x, y);
  }
}
stop = omp_get_wtime();

#pragma omp target exit data map(from:x[0:n], y[0:n])
```

Coalesced Memory Access and OpenMP

- schedule(static,1)
  - simd directive may be an alternative

```
#pragma omp target teams distribute parallel for \
  reduction(max:error) collapse(2) schedule(static,1)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for \
  collapse(2) schedule(static,1)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Assign adjacent threads adjacent loop iterations.

OpenMP Offloading Summary

- OpenMP offers simpler approach to programming accelerators than CUDA, OpenCL
  - Host-centric model, regions of code are offloaded to target device
  - Execution configuration and data environment can be configured in detail

- Further reading:

- Ruud van der Pas, Eric Stotzer and Christian Terboven *Using OpenMP - The next Step*, Chapter 6, "Heterogeneous Architectures"