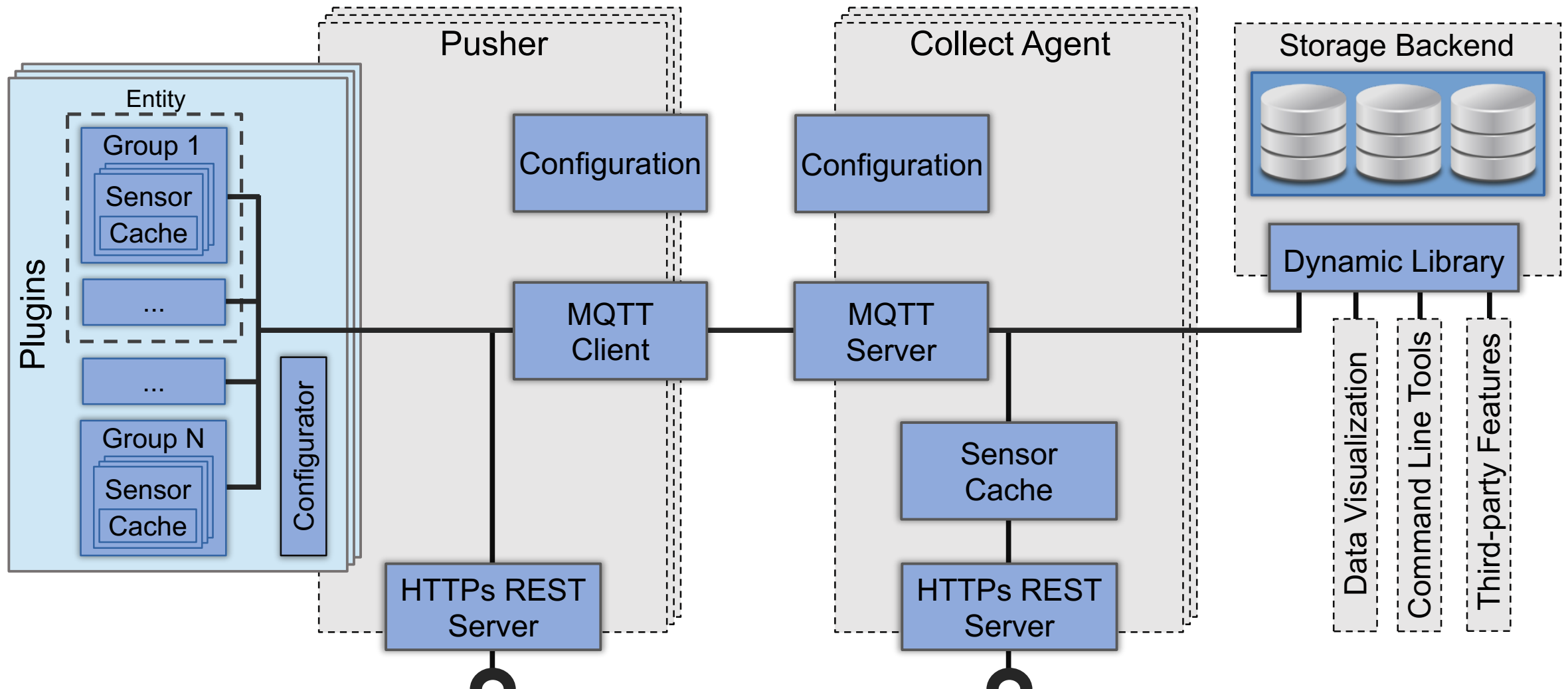


DCDB 101

BEAST Lab Course | 06/05/2021 | Michael Ott

DCDB Architecture



Environment



```
lu78mod2@beast-gw:~> source /opt/dcdb/dcdb.bash
```

dcdbconfig sensor list



```
lu78mod2@beast-gw:~> dcdbconfig sensor list  
dcdbconfig 0.4.272-g4611155 (libdcdb 0.4.272-g4611155)
```

```
/beast/beast-gw/power_ac  
/beast/cs/power_ac  
/beast/cs1-8/power_ac  
/beast/ibswitch/power_ac  
/beast/ipmiswitch/power_ac  
/beast/pdu1/power1  
/beast/pdu1/power10  
/beast/pdu1/power11  
/beast/pdu1/power12  
/beast/pdu1/power2  
/beast/pdu1/power3  
/beast/pdu1/power4
```

dcdbconfig sensor show



```
lu78mod2@beast-gw:~> dcdbconfig sensor show /beast/rome1/power_ac  
dcdbconfig 0.4.272-g4611155 (libdcdb 0.4.272-g4611155)
```

Details for virtual sensor /beast/rome1/power_ac:

Expression: /beast/pdu1/power5+/beast/pdu2/power5

vSensorId: /beast/rome1/power_ac

T-Zero: 2021-04-15T14:39:17.211384000 (1618497557211384000)

Unit: W

Scaling factor: 1

Operations:

Interval: 300000000000

TTL: 0

Sensor Properties:

dcdbquery (1/2)



```
lu78mod2@beast-gw:~> dcdbquery /beast/rome1/power_ac now-5m now  
dcdbquery 0.4.272-g4611155 (libdcdb 0.4.272-g4611155)
```

```
sensor(/beast/pdu1/power5) sensor(/beast/pdu2/power5) add  
Sensor,Time,Value (W)  
/beast/rome1/power_ac,2021-05-06T15:16:47.211384000,332  
/beast/rome1/power_ac,2021-05-06T15:17:17.211384000,334  
/beast/rome1/power_ac,2021-05-06T15:17:47.211384000,335  
/beast/rome1/power_ac,2021-05-06T15:18:17.211384000,338  
/beast/rome1/power_ac,2021-05-06T15:18:47.211384000,341  
/beast/rome1/power_ac,2021-05-06T15:19:17.211384000,344  
/beast/rome1/power_ac,2021-05-06T15:19:47.211384000,346  
/beast/rome1/power_ac,2021-05-06T15:20:17.211384000,345  
/beast/rome1/power_ac,2021-05-06T15:20:47.211384000,344
```

dcdbquery (2/2)



```
lu78mod2@beast-gw:~> dcdbquery -r /beast/rome1/power_ac "2021-05-05 02:30" "2021-05-05 02:35"
```

```
dcdbquery 0.4.272-g4611155 (libdcdb 0.4.272-g4611155)
```

```
sensor(/beast/pdu1/power5) sensor(/beast/pdu2/power5) add  
Sensor,Time,Value (W)
```

```
/beast/rome1/power_ac,1620174587211384000,334
```

```
/beast/rome1/power_ac,1620174617211384000,334
```

```
/beast/rome1/power_ac,1620174647211384000,334
```

```
/beast/rome1/power_ac,1620174677211384000,333
```

```
/beast/rome1/power_ac,1620174707211384000,335
```

```
/beast/rome1/power_ac,1620174737211384000,336
```

```
/beast/rome1/power_ac,1620174767211384000,335
```

```
/beast/rome1/power_ac,1620174797211384000,334
```

```
/beast/rome1/power_ac,1620174827211384000,334
```



LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Dr. Karl Furlinger

Lehrstuhl für Kommunikationssysteme und
Systemprogrammierung

GPU Hardware and OpenMP on Heterogeneous Architectures



Top 10 of the Top500 List (Nov. 2020)

Rank	System	Rmax (TFlop/s)	
1	Fugaku - Fujitsu ARM A64FX 48C 2.2GHz, Tofu interconnect D, RIKEN Center for Computational Science Japan	442,010.00	
2	Summit - IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, EDR Infiniband, DOE/SC/Oak Ridge National Laboratory United States	148,600.00	Nvidia Volta
3	Sierra - IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, EDR Infiniband, DOE/NNSA/LLNL United States	94,640.00	Nvidia Volta
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz NRCPC National Supercomputing Center in Wuxi China	93,014.60	
5	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, HDR Infiniband, Nvidia Corporation United States	63,460.00	Nvidia Ampere
6	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	61,444.50	
7	JUWELS Booster Module - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR Forschungsz. Juelich (FZJ) Germany	44,120.00	Nvidia Ampere
8	HPC5 - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	35,450.00	Nvidia Volta
9	Frontera - Xeon Platinum 8280 28C 2.7GHz, HDR InfiniBand, Dell EMC TACC/Univ. of Texas United States	23,516.40	
10	Dammam-7 - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	22,400.00	Nvidia Volta

Accelerators/Co-processors

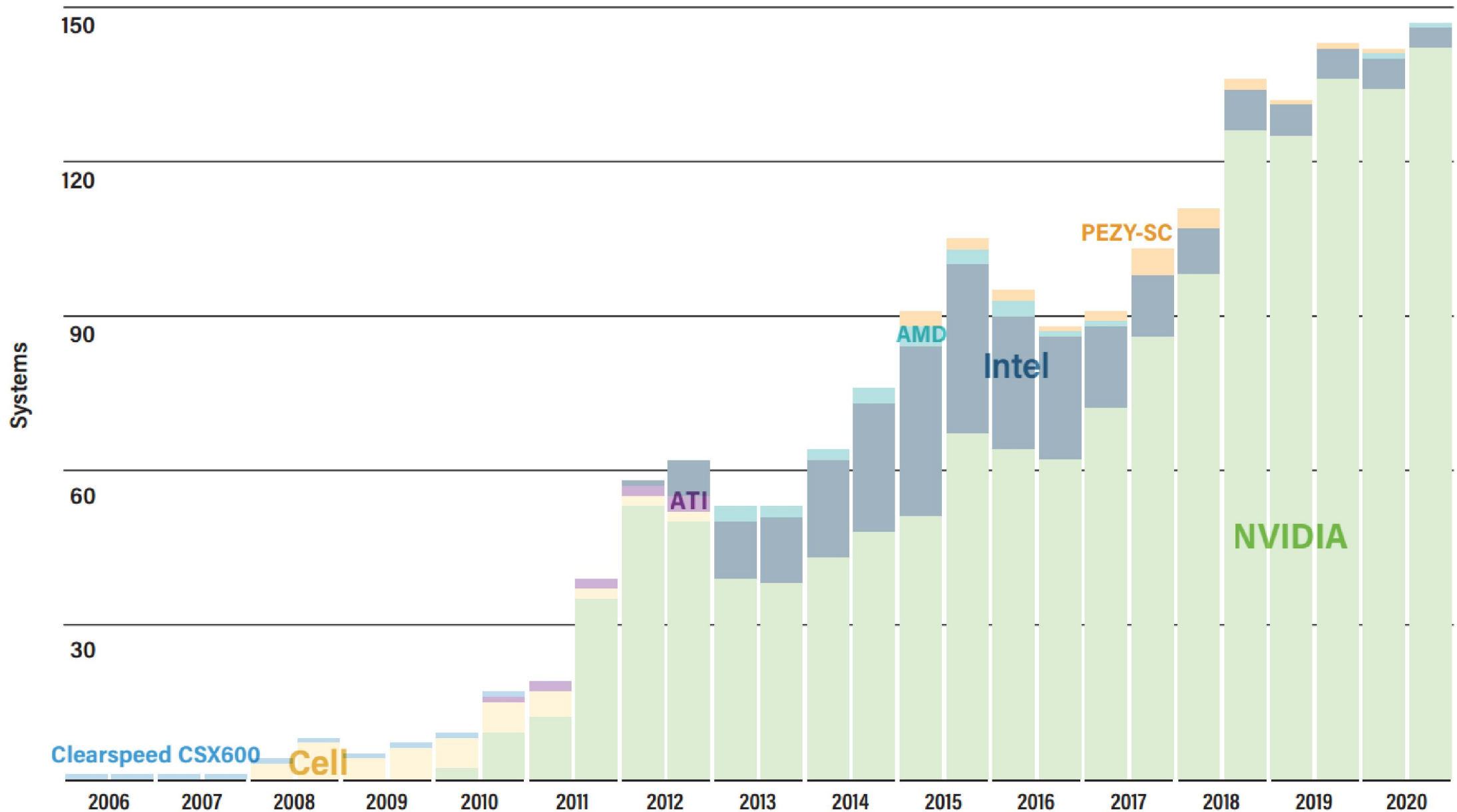
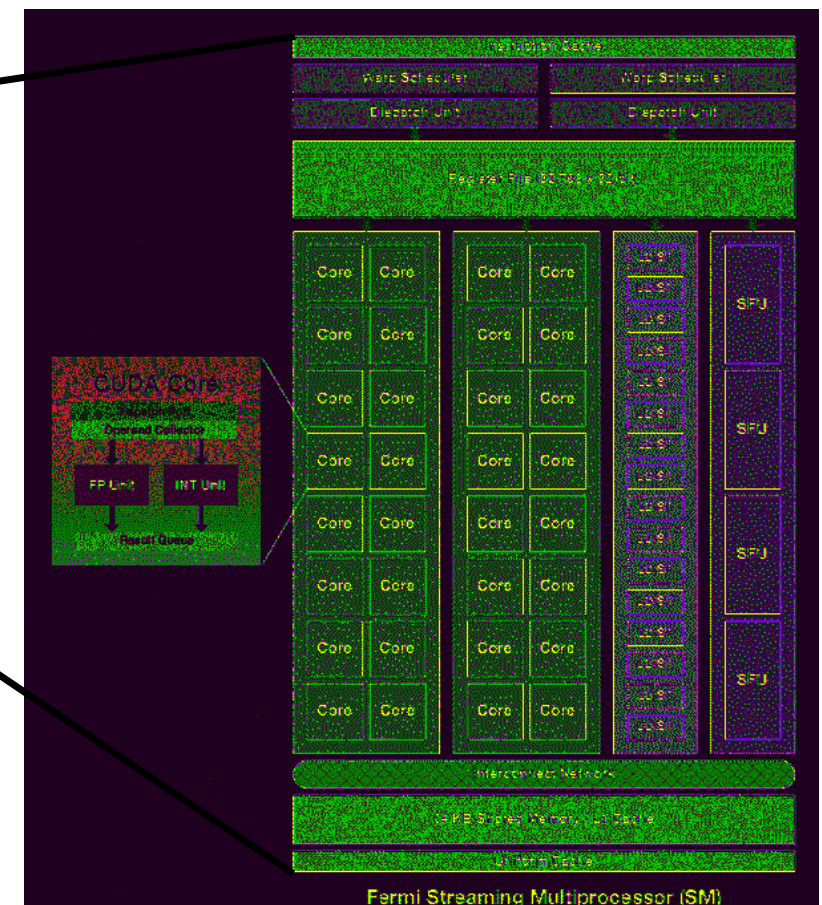
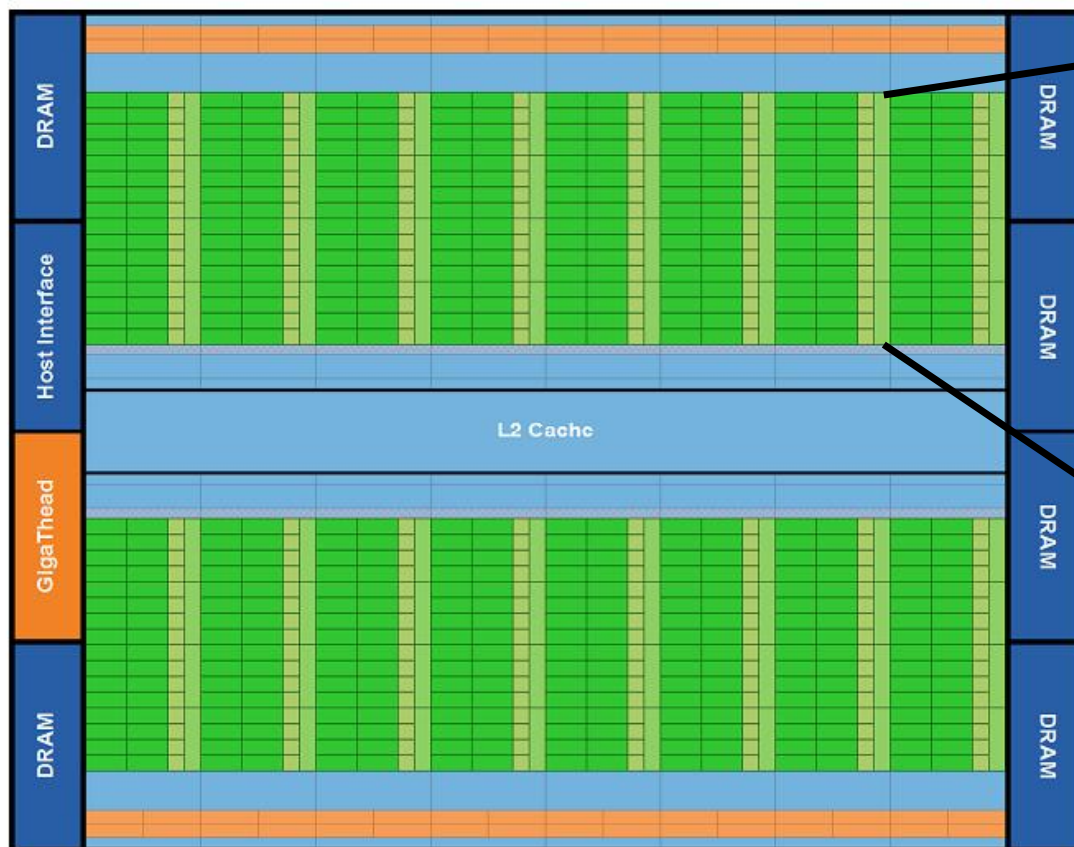


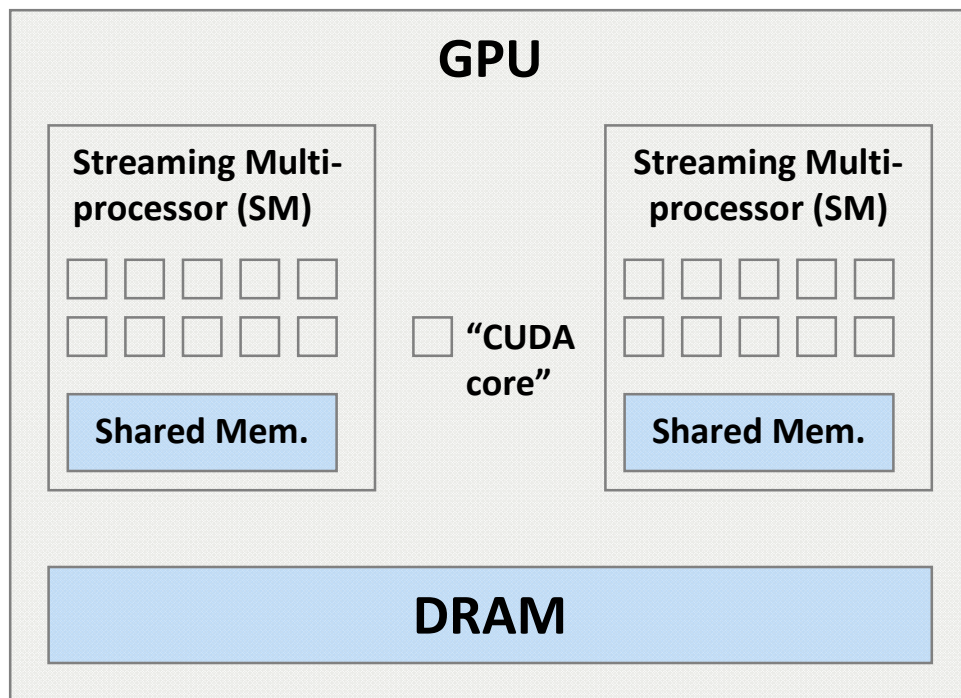
Image Source: <http://top500.org/lists/2020/06/>

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



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)

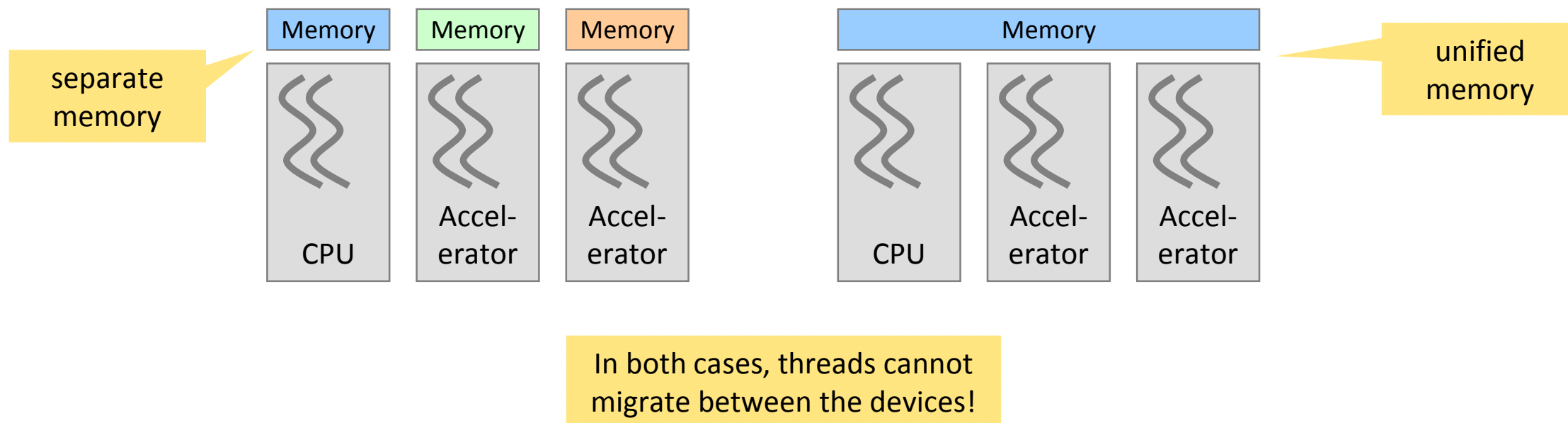


Image Source: CAAQA (6th Edition)

■ 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

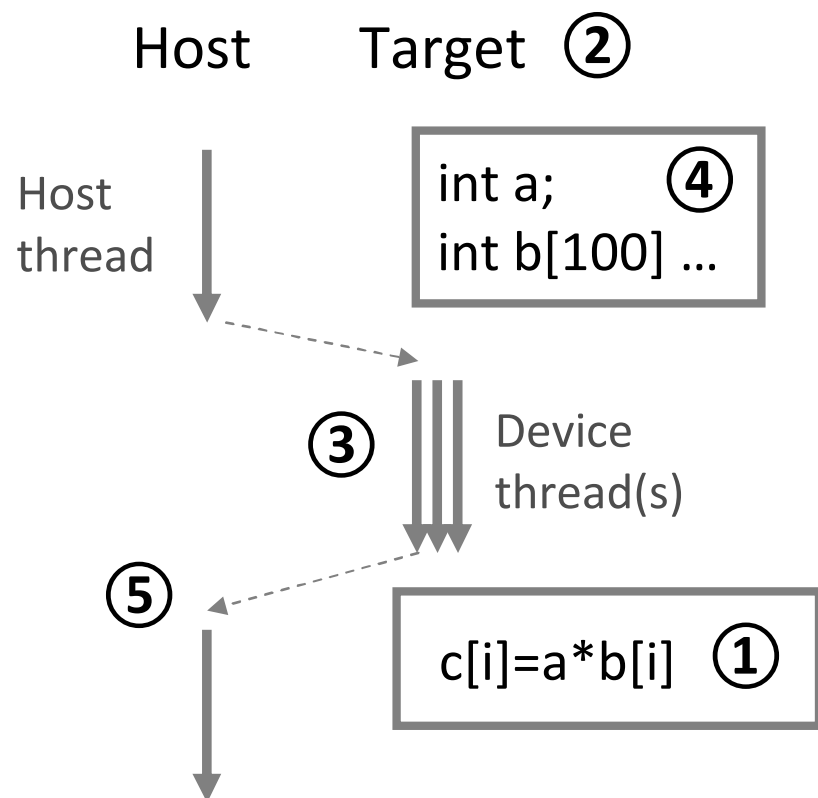


- Benefits of special purpose processors
 - Can execute some types of computation quicker and more energy efficient
- Examples:
 - GPUs: good at massive data parallel operations
 - DSPs: good at signal processing applications
 - FPGAs: good at data-flow type operations
- Challenges for programming
 - 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

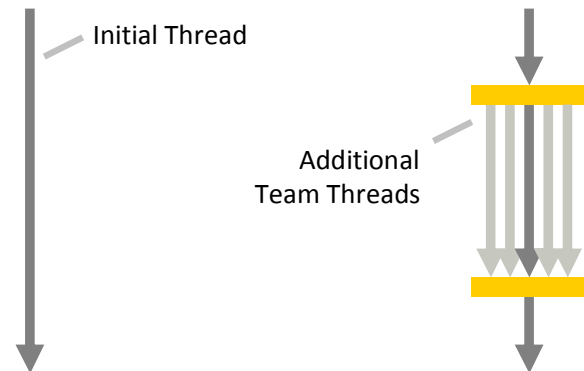
■ To offload code and specify the execution configuration

- `#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

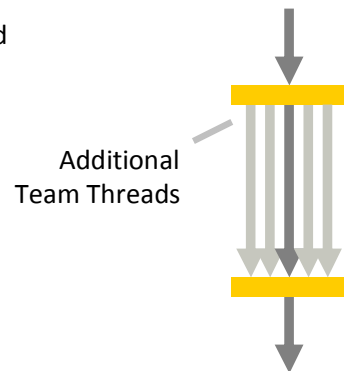
■ To specify the data environment

- `#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

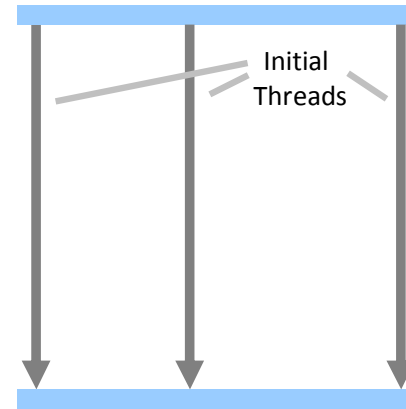
Execution Configuration on the Target Device



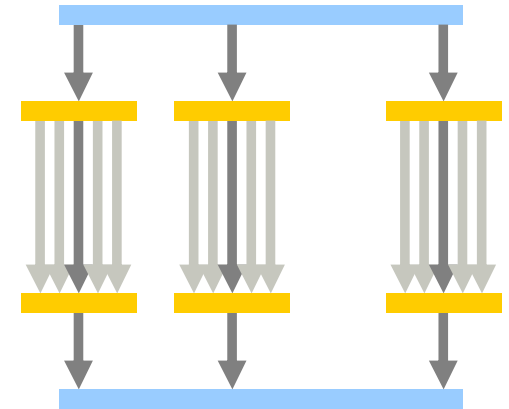
```
#pragma omp target
{
}
```



```
#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();
}
```

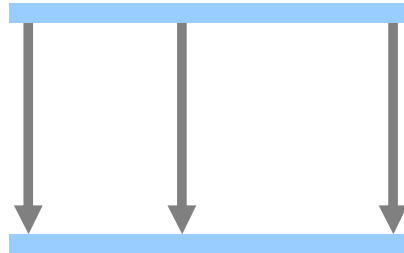


Thread [3/8] in team [0/1]
Thread [0/8] in team [0/1]
Thread [1/8] in team [0/1]
Thread [6/8] in team [0/1]
Thread [7/8] in team [0/1]
Thread [2/8] in team [0/1]
Thread [4/8] in team [0/1]
Thread [5/8] in team [0/1]

8 threads x 1 team =
8 threads in total

Execution Configuration Example (2)

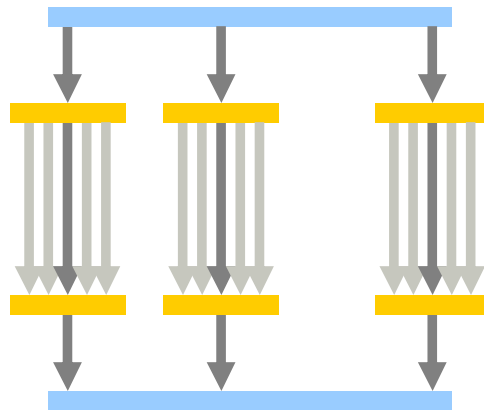
```
#pragma omp target
#pragma omp teams
{
    print();
}
```



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

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

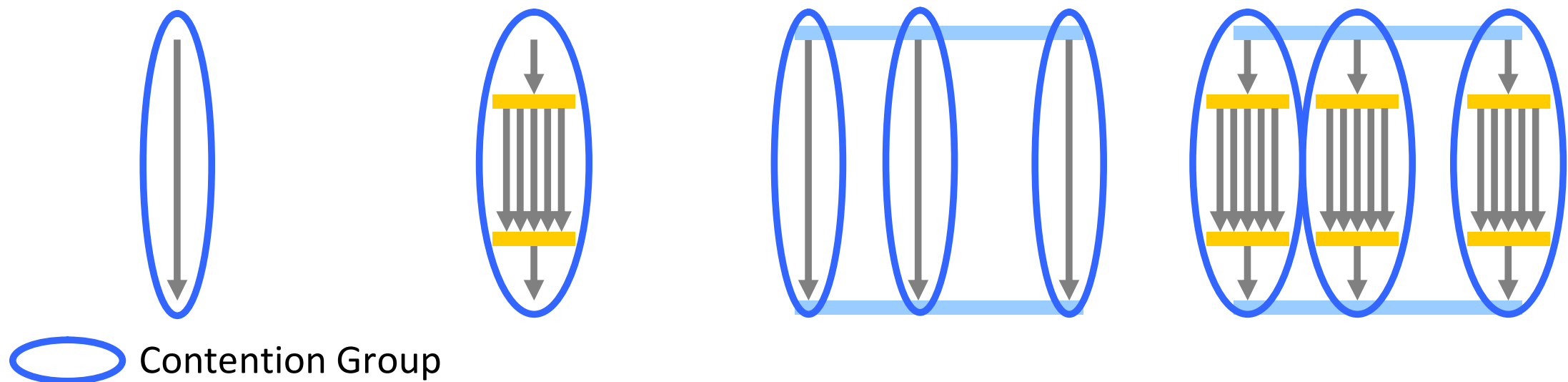


Thread [0/8] in team [9/15]
 Thread [6/8] in team [8/15]
 Thread [2/8] in team [8/15]
 Thread [3/8] in team [9/15]
 Thread [4/8] in team [9/15]
 Thread [2/8] in team [9/15]
 Thread [1/8] in team [4/15]
 Thread [7/8] in team [9/15]
 Thread [4/8] in team [8/15]
 Thread [3/8] in team [4/15]
 Thread [0/8]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

8 threads x 16 teams =
128 threads in total

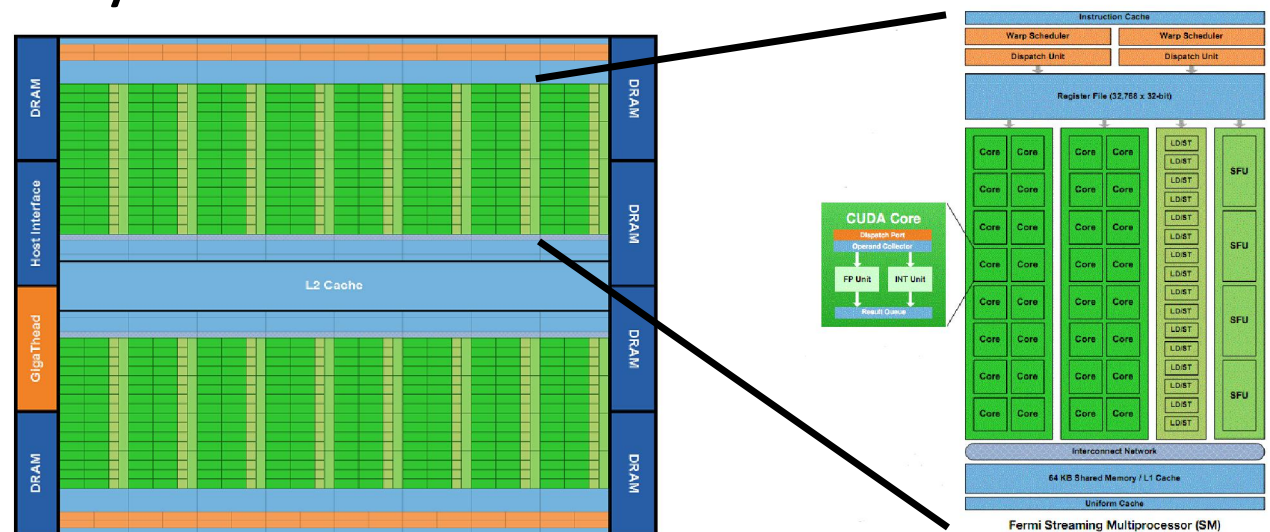
Initial Threads and Contention Groups

- 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 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)

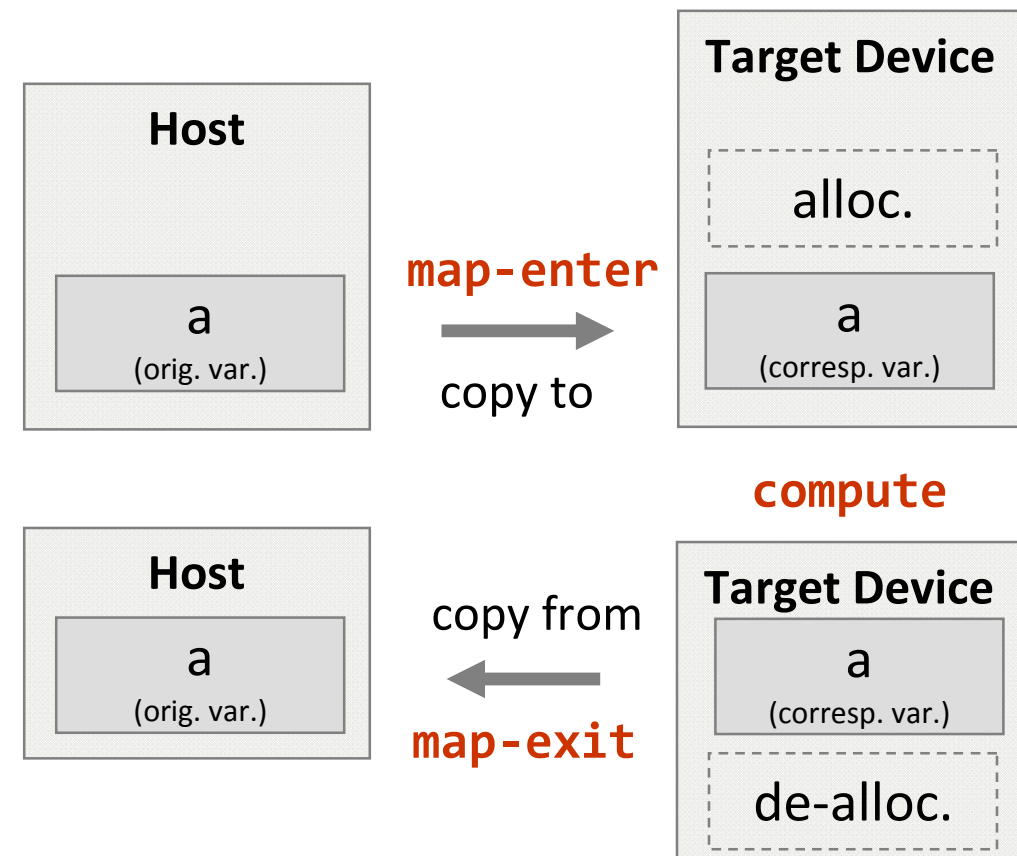
- The DDE is the set of variables present on the device
 - An **original variable** (on the host) is mapped to a **corresponding variable** (on the device)
 - Updates between original and corresponding variable happen **automatically** and can also be triggered **manually**
- Typical usage and default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



- 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:

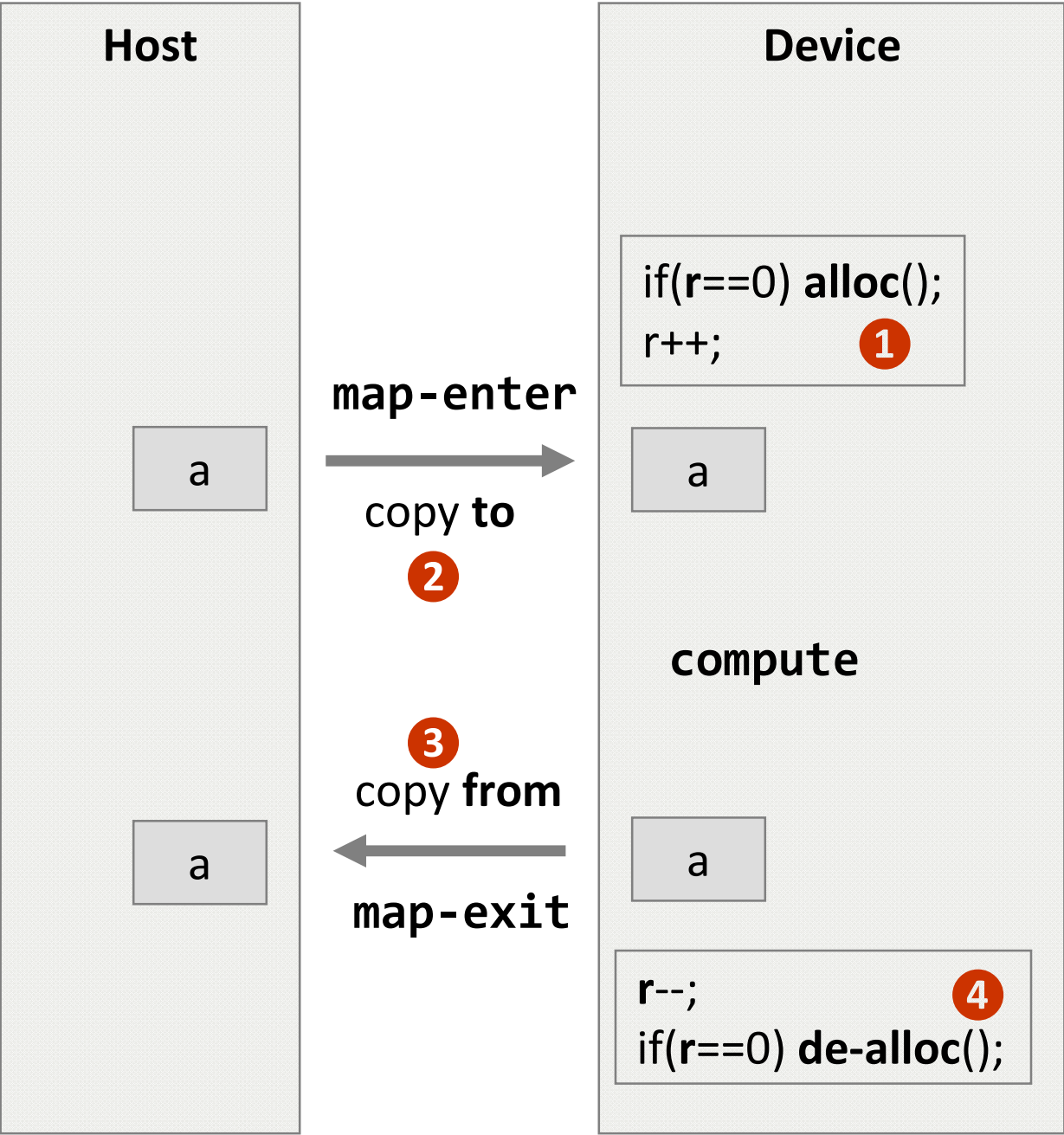
```
#include <stdlib.h>

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”

