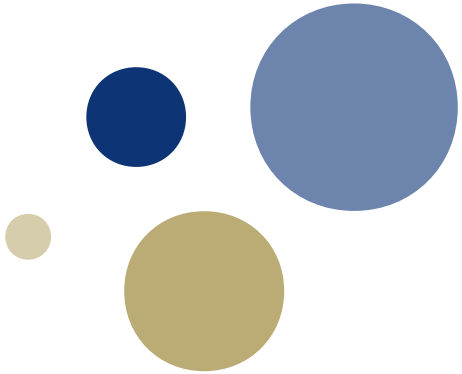
NTNU

Kunnskap for en bedre verden



OpenMP Part 3. GPU

John Floan, HPC group, IT avd.
(john.floan@ntnu.no)

www.hpc.ntnu.no
www.openmp.org
www.nvidia.com

OpenMP 4.5:
All exercises are for GPU accelerators.
(Note that; in the other OpenMP courses we using OpenMP 3.x)

There are two type of accelerators today:

- Intel Xeon Phi (MIC) processors with around 60 cores.
  (See more https://en.wikipedia.org/wiki/Xeon_Phi)

- GPU accelerators (100s to 1000s of cores).
  (See more http://www.nvidia.com/object/tesla-p100.html)

Accellerator is a device connected to PCI bus or NVLink Server and with lot of cores.

In this course we shall only work with Nvidia Accelerator Tesla P100

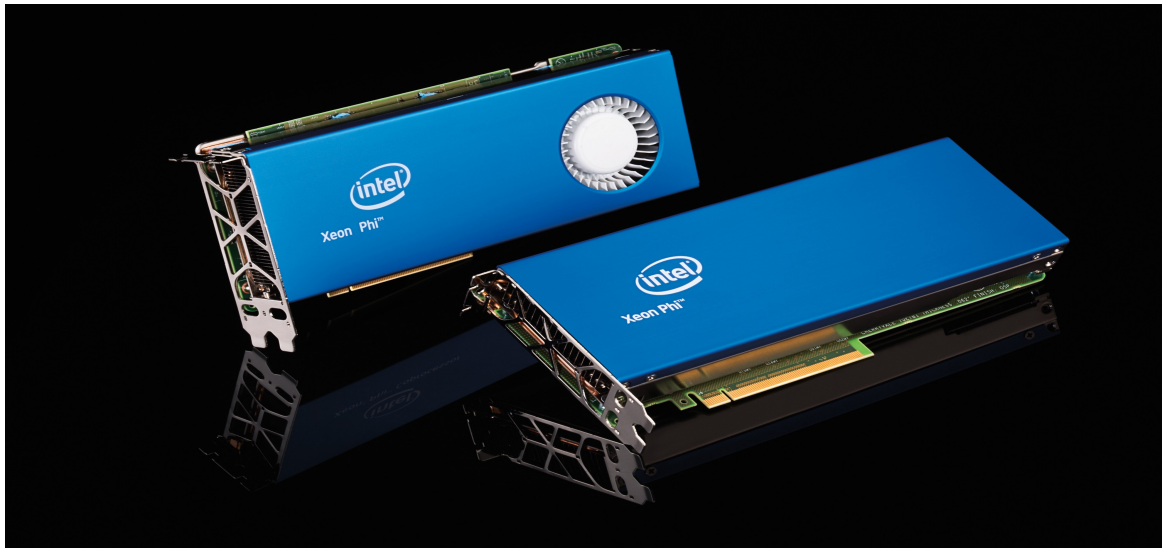
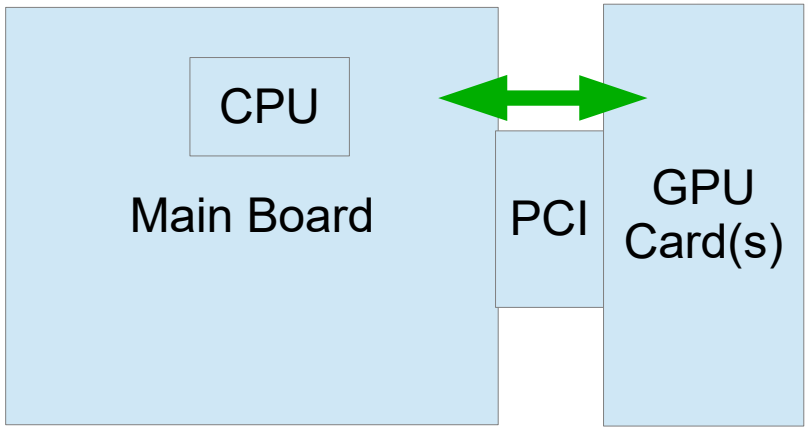


Figure 1. Intel Xeon Phi Multi Core device for PCI slot

Figure 2. Nvidia Tesla P100 device for PCI slot

GPU is connected to main board via PCI slot. Data and code have to be sent and received via PCI bus.



Accelerator vs CPU

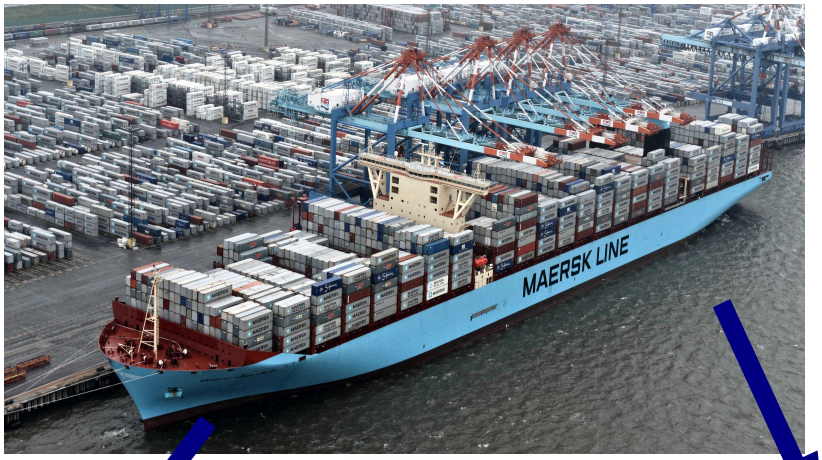
CPU has
-Small number of cores (2-20),
-High clock frequency
-Large cache memory.
-Low memory bandwidth
-Many registers (like AVX etc)

Accelerator GPU has:
-Many cores (or SMs; streaming multiprocessors) (>1000)
-High memory bandwidth,
-Low clock freqeuency
-Small shared cache memory.
-Have to use same code for all cores in teams.

An accelerator have to be off-loaded from the CPU to GPU, and move data and code between main board and GPU device via PCI.

I compare CPU and Accelerator as to unload a container ship and drive the cargo to a destination: Use few trucks or 1000s of mopeds.

Container ship represent lot of data (GB)



GPU

1000s of slow mopeds (small cache)

VS

CPU

Few trucks (large cache)

-CPU: Each truck can drive independently and in different route.
-GPU: All mopeds have to drive in teams of 32 in same route

CPU / GPU specifications (Epic2)

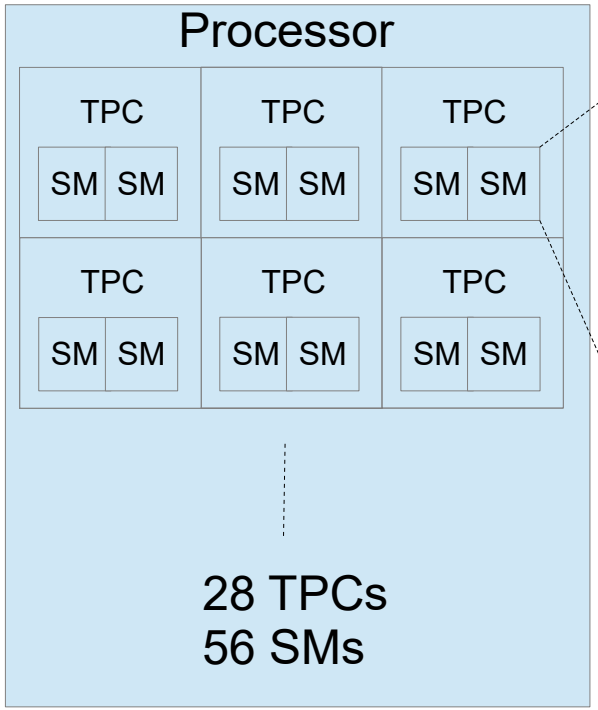
	CPU	GPU
Cores	24 cores	
FP32 (single precision)	-	3584 cores
FP64 (double precision)	-	1792 cores
Clock frequency	2.2 GHz	1126 MHz
Memory size (max)	1.54 TB	16 GB
Memory bandwidth	68 GB/s	732 GB/s
Power	135W	300W

Epic2:
-Intel CPU Broadwell Xeon E5 (E5-2650V4)
-Nvidia GPU Tesla P100 (PCI)

More about Nvidia Tesla P100 Card with GP100 Processor :

For more details about Tesla P100.
https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

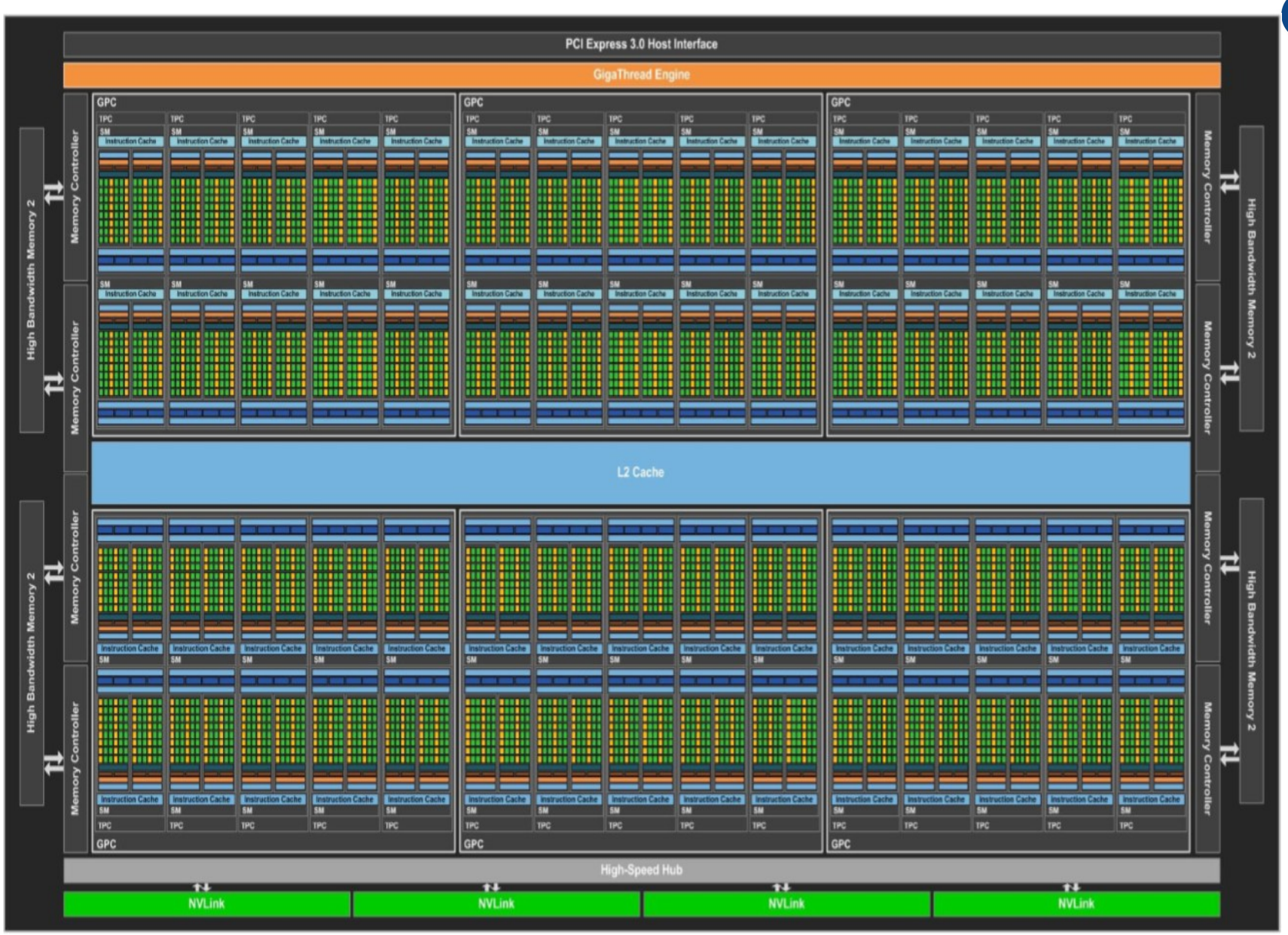
GPC: Graphical Processing Cluster
TPC: Texture Processor Cluster
SM: Streaming Multiprocessor
WARP: Warp is set of 32 threads within a thread block that such that all threads execute the same instruction.



SM: Streaming Multiprocessor

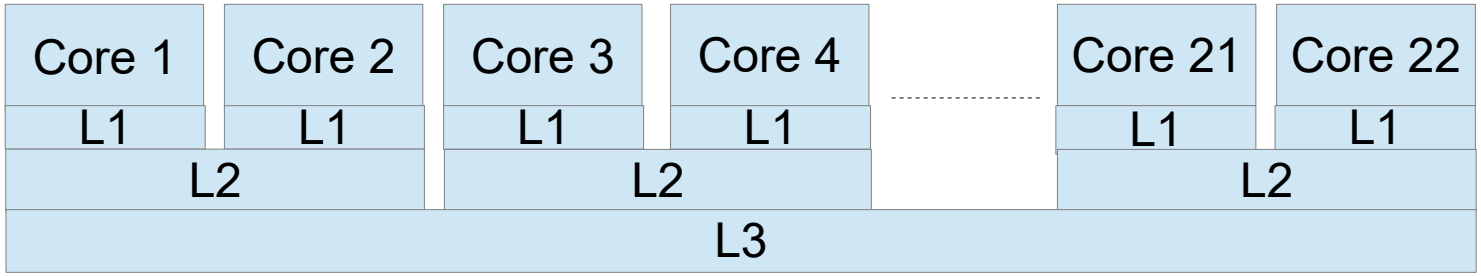
SP Units: 2 X 32 cores, DP Units: 32 cores
SP: Single precision floating point, DP: Double precision

Nvidia GP100 GPU has 60 SMs. GP100 Accelerator has 56 SMs



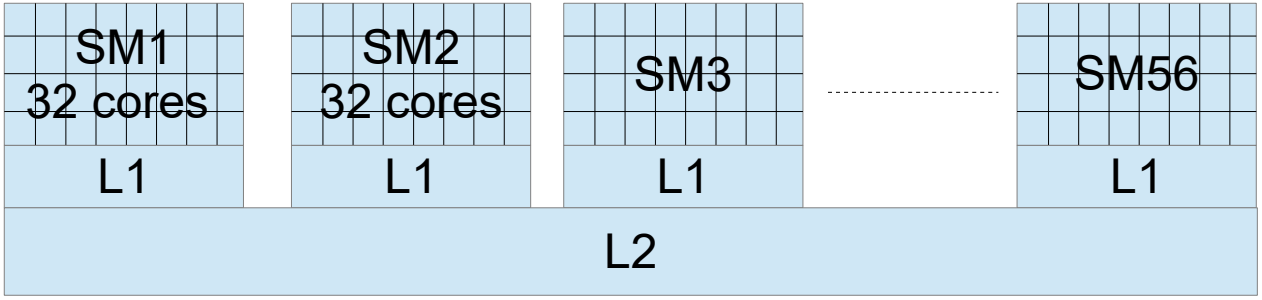
CPU vs GPU cores.

CPU (intel)



One L1 cashe (level 1) each core. Each core is independent of others.

GPU (P100)



Each SM (streaming multiprocessors), of 32 cores, have one L1 cashe.
Each SM have to use same program (cores are dependent of 32 other).

TUTORIALS 1. DEVICE TARGET AND TEAMS

The GPU and MIC devices are connected to CPU main board via PCI slot.

You have to move (offload) the instruction counter from the CPU to GPU.

The directive for offloading GPU:

```
#pragma omp target    //Default GPU 1
```

Specify device (GPU 1 or GPU 2)

```
#pragma omp target device (0)  // GPU 1
#pragma omp target device (1)  // GPU 2
```

The directive for offloading to MIC (Intel Xeon Phi)

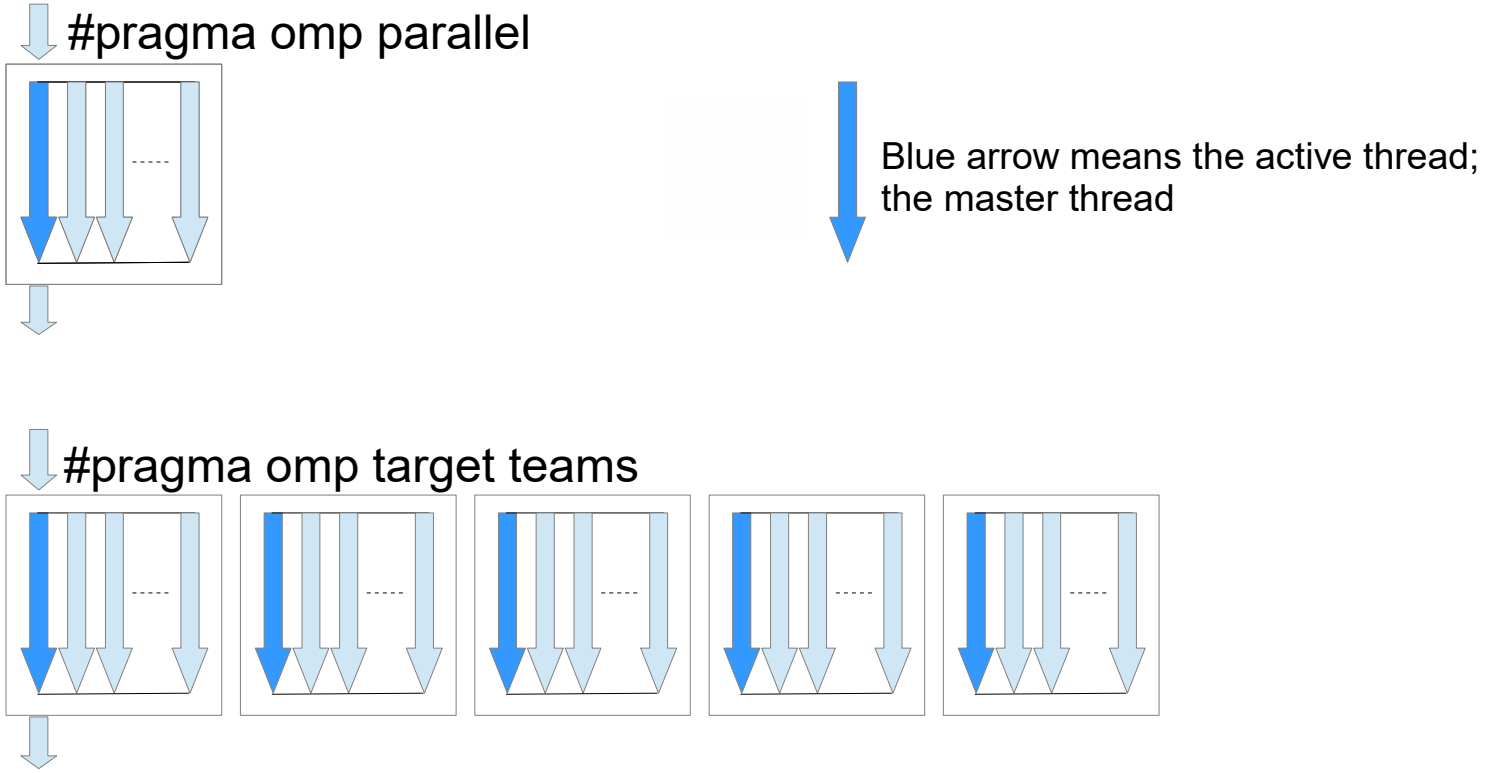
```
#pragma offload target (mic)
#pragma offload target (mic:0) //MIC in PCI slot 0
#pragma offload target (mic:1) //MIC in PCI slot 1
```

Teams

#pragma omp target teams

Creates a league of thread teams where the master thread of each team executes the region

Example: OpenMP with omp parallel for all cores and teams.



Data scoping for Accelerators.

Inside Accelerator (OpenMP host) teams, we have this data clauses shared, private, firstprivate and reduction.

Default scoping for teams:

Arrays:	shared
Loop variables:	private, firstprivate
Variables:	firstprivate (4.5)

Data scoping for Accelerators continue.....

Shared variables/arrays between CPU and GPU are more complicated.
We have to think about moving data between main bord and the accelerator.

Map data to GPU:

Read only: map(to:x,y,...)
Data will be copied to accelerator at start

Write only: map(from:x,y,...)
Data will be copy from accelerator at end.

Read-write: map(tofrom:x,y,...)
Data will be copied to/from accelerator at start/end

Scratch: map(alloc:x,y,...)
Data will NOT be copied. Data must be initialized inside the accelerator.

Array sections

Array section is specified using : notation in map

Fortran:

start:end

C:

start:length

Example

Fortran

.... map(to:A(1:10) means: from element 1 to element 10

C:

.... map(to:A[0:10]) means: from element 0 to element 9

Example:

Standard c code:

```
int i;
int a=1;

a=a+1;
```

Targeting and teams:

```
int i;
int a=0;


//Target GPU and copy to/from device with a
#pragma omp target teams map(tofrom:a)
{
 a=1;
}
```

Fortran:

```
Integer::i
Integer::a

a=0

!$omp target teams map(tofrom:a)
 a=1;
!$omp end target teams
```

Some runtime routins:

int omp_is_initial_device()

Returns true (1) if the current task is executing on the host device; otherwise, it returns false (0).
You get this info from main board/CPU

int omp_get_num_devices()

Returns the number of target devices
You get this info from main board/CPU.

int omp_get_num_teams()

Returns the number of teams in the current teams region, or 1 if called from outside of a teams region.
Need to inside GPU to check this.

Job script:

```
#!/bin/bash
#
#SBATCH -J array1          # Sensible name for the job
#SBATCH -N 1               # Allocate 1 nodes for the job
#SBATCH -gres=gpu:2        # Select 2 GPUs
#SBATCH -t 00:10:00        # Upper time limit for the job
#SBATCH -p EPIC2           # Select Epic2 nodes

#Loads module
module purge
module load GCC/8.1.0-CUDA-9.1.85
./array1
```

Compiling:

```
gcc -O2 -fopenmp -DOPENMP -foffload=nvptx-none -Wall -o array1 array1.c
```

For debuging:

Update:
It is also possible to update variable from GPU to CPU with:
Example (update cpu with variable a and print it out)

```
#pragma omp target update from(a)
printf("show a %d\n",a);
```

Exercise 1. Test for checking if GPUs are connected and get number of thread teams.

ssh -X training.hpc.ntnu.no

module load GCC/8.1.0-CUDA-9.1.85

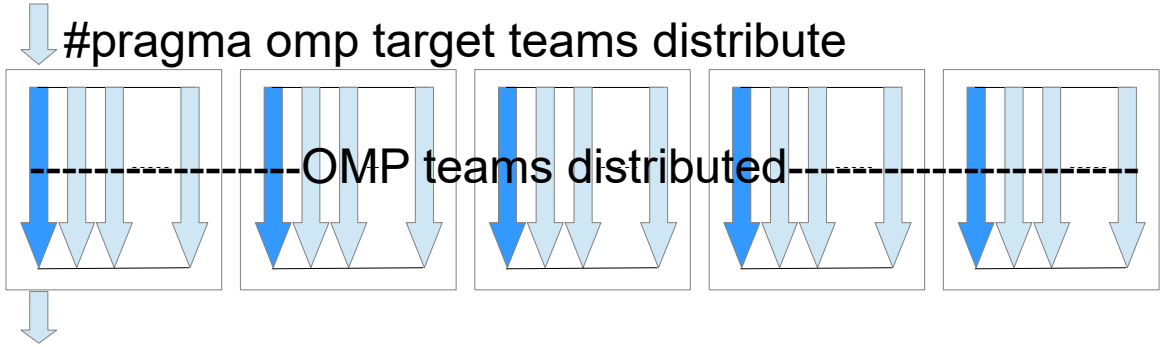
cd tutorials/OpenMP_GPU/

cd part1_target_teams
make
sbatch targeting_c.job
(_c for c code and _f for fortran)

Check the output.
cat slurm-xxxx.out

Change code in targeting.c to target the GPU and find number of teams (more then 1 team).

Tutorial 2. Distribute teams.



C: #pragma omp target teams distribute
Fortran: !$omp target teams distribute

Specifies loops which are executed by the thread teams

Directive for Teams Distributed has to be used together with for-loops.
-There are no parallelisme inside a thread teams.
-Iterations are distributed statically
-No garanti that the teams will execute simuntaneously
-Reduction of variables are also included: ... reduction(+:x)
Example

```
#pragma omp target teams distribute
for (i=1;i<n;i++)
      A[ i ] = i ;
```

Exercise 2. Teams distribute

a) Check running time for normal cpu code.
cd ../..
cd part2_team_dist/cpu
module purge
module load GCC
make
sbach array1_c.job (or array1_f.job)

Check the running time.

b) Parallelize the code for teams distribute
cd ..
cd gpu
Change array.c with team distribute
module purge
module load GCC/8.1.0-CUDA-9.1.85
make
sbatch array1_c.job (or array1_f.job)

Check the running time and compare with cpu job.

Profiling:
After you are finished and get correct result you can check the program with a profiler.
Add in the job script with the profiler nvprof

nvprof ./array1_c (or _f)

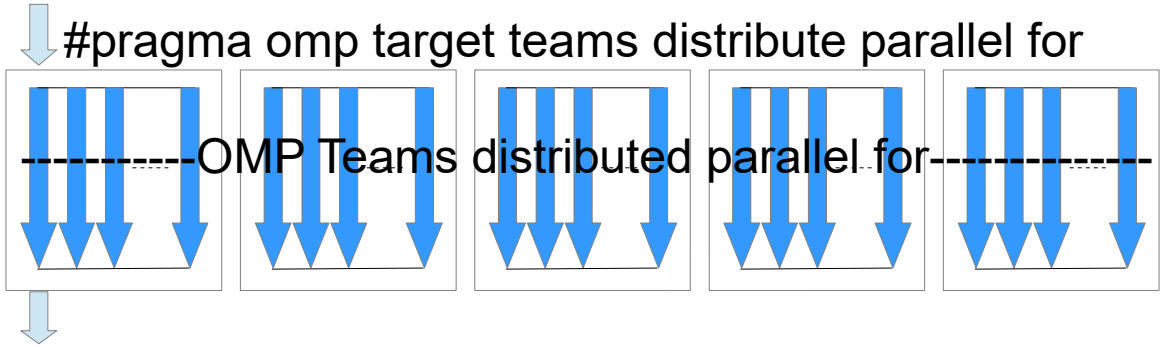
DtoH means: Device to Host (GPU to Main board (CPU))
HtoD means: Host to Device

Run your code again.

Tutorial 3. Parallelizing teams.

#pragma omp target teams distribute parallel for

These constructs specify a loop that can be executed in parallel by multiple threads that are members of multiple teams.




```
#pragma omp target teams distribute parallel for
for (i=0;i<n;i++)
        A[ i ] = B[ i ];
```

It is also allowed to do this

```
#pragma omp target teams distribute
for (i=0;i<n;i++)
{
        #pragma omp parallel for
        for (j=0;j<n;j++)
                A[ i ] += j * B[ j ];
}
```

The streaming processors in a SM can only run the same code and with different chunk of the data array, teamed in 32 CUDA Cores.

Example:

```
#pragma omp target teams distribute parallel for
for (i=0;i<32;i++)
        A[ i ] = B[ i ];
```

Array A and B is stored into level 1 cashe.

The code runs as this

- CUDA core 1 calculate B[1] with A[1]
- CUDA core 2 calculate B[2] with A[2]
- Etc

Fortran:

```
!$omp target teams distribute parallel do
```

Exercise 3: Optimize your code with teams distribute parallel for

Same place:
make
sbatch array1.job

Tutorial 3. Data region

Data region:
#pragma omp target data | Fortan: !$omp target data
Creates a device data environment for the extent of the region.

Target data keep the data to the accelerator (GPU)
Example: Update A on GPU

```
#pragma omp target data map(tofrom:A)
{
#pragma omp target teams distribute parallel for
for (i=0;i<n;i++)
        A[i]=i;

....

}//End data region  (fortran: !$omp end target data)
```

Exercise 3.
Optimize the array1 code in part2_teams_dist/cpu
and add data region into your program

Exercise 4.
a) If you har finished with exercise 3 you can optimize
part3_team_dist
Compare running time for cpu and gpu.

b) Change arrays and tmp to type float (single precision)

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

[1] Peak TFLOPS rates are based on GPU Boost Clock

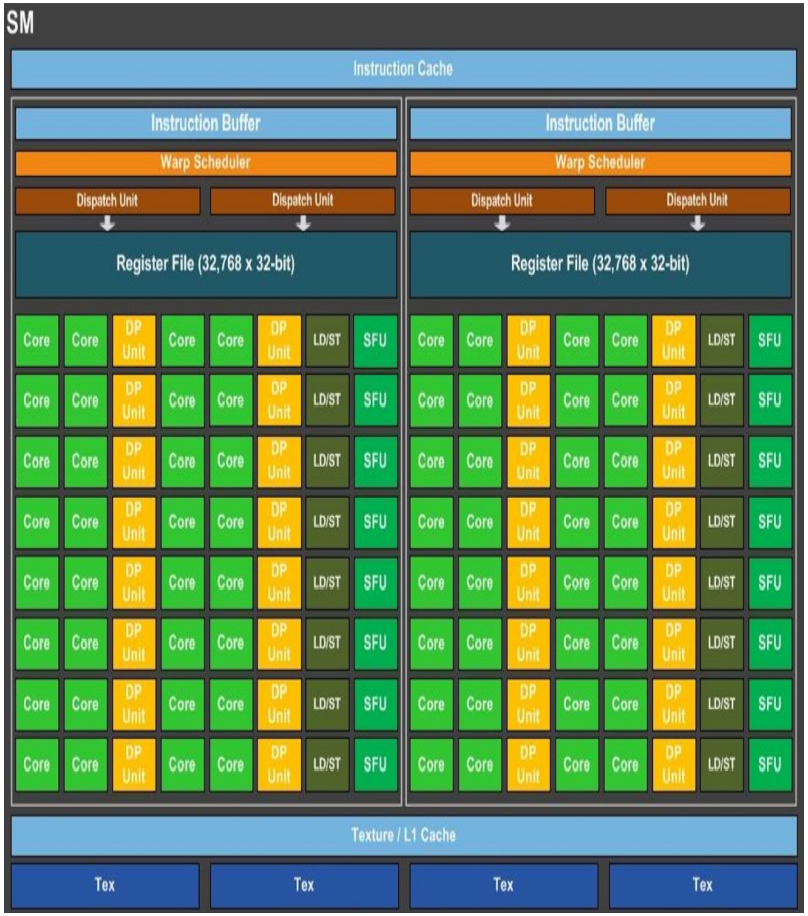
L1 cache 24kB

FP32: Single Precision CUDA cores, FP64: Double Precision CUDA cores
Texture Units are for 3D geometry (4 each SM)

P100 vs V100

New on V100 is Tensor Cores.
Streaming Multiprocessors (SM)

P100



V100

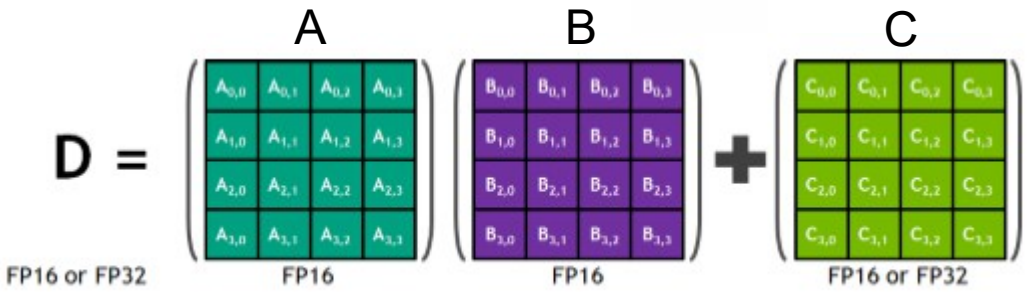
Tensor Cores.

New Tensor Cores are for deliver required performance to train large neural network.

Tesla V100 GPU contains 640 Tensor Cores.

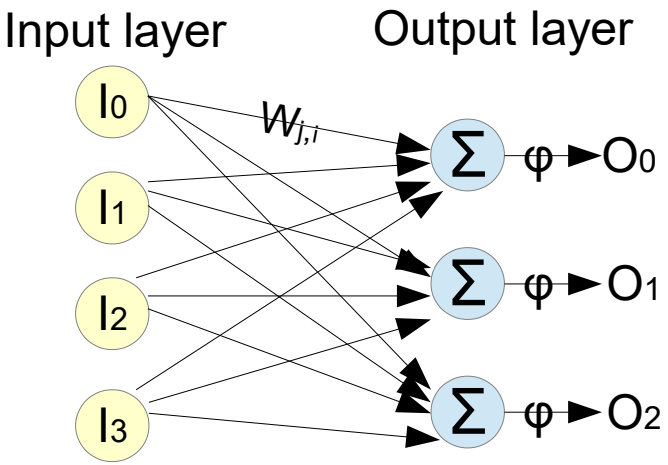
Each Tensor Core operate on a 4x4 matrix and performs the following operation:

D = AxB + C , where AxB is a matrix mulitplication



See more here:
https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Neural network:



$$0_j = \varphi_j \left(\sum_{i,j} W_{i,j}\, I_i\right)$$

where I is input, W is weight, φ is the activation function and O is output. Activation function (or transfer function) φ gives an output of 1 or 0 (or 1 or -1) if the treshold Θ is reached.

Threshold:

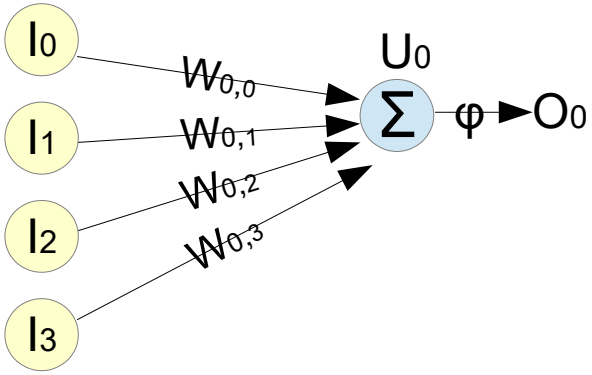
$$U_j = \sum W_{i,j}\, I_i \quad \text{and} \quad O_j = \varphi(U_j)$$

$$\text{Then the output } O_j = \begin{cases} 1 \text{ if } U_j \geq \theta \\ 0 \text{ if } U_j < \theta \end{cases}$$

This is same as matrix mulitplication:

$$U_j = \sum_{i,j} W_{i,j}\, I_{i,} \quad , \quad U = W \times I \text{ and } Oj = \varphi(Uj)$$

Example: Neural network with 4 input neurons and 1 output neuron and the threshold is 2



$$U_j = \sum W_{i,j}\, I_i$$

$$\begin{aligned} U_0 &= I_0 \cdot W_{0,0} + I_1 \cdot W_{0,1} + I_2 \cdot W_{0,2} + I_3 \cdot W_{0,3} \\ &= 0.7 \cdot 1 + 1 \cdot 0.5 + 0.1 \cdot 1 + 1 \cdot 1 \\ &= 0.7 + 0.5 + 0.1 + 1 \\ &= 2.3 \end{aligned}$$

Output:
$O_0 = 1$ because threshold (Θ) is 2

$$O_0 = \begin{cases} 1 \text{ if } U_0 \geq 2 \\ 0 \text{ if } U_0 < 2 \end{cases}$$

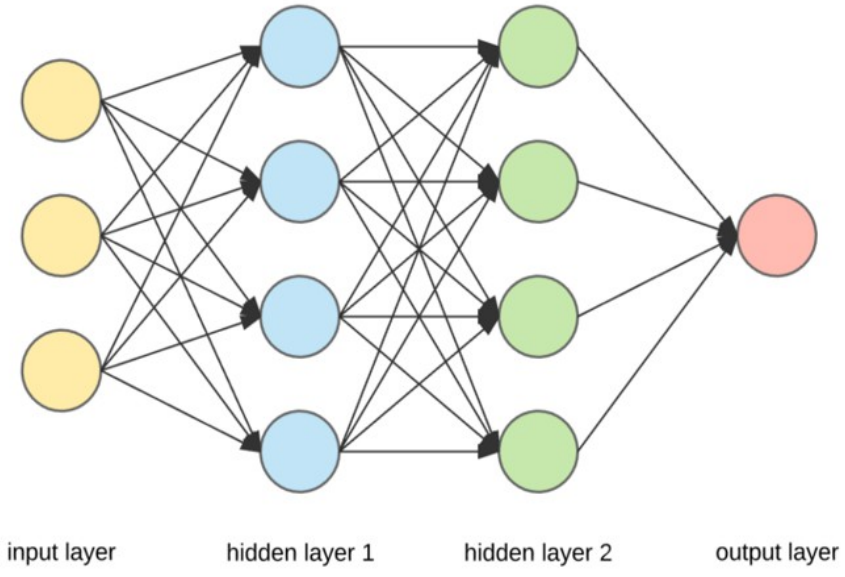


Figure 1

Artificial neural network