

GPU Programming Workshop

Session 1: Introduction to OpenACC

*Special Technical Projects
&
Client Support Group*
April 6, 2021

Session 1 Overview

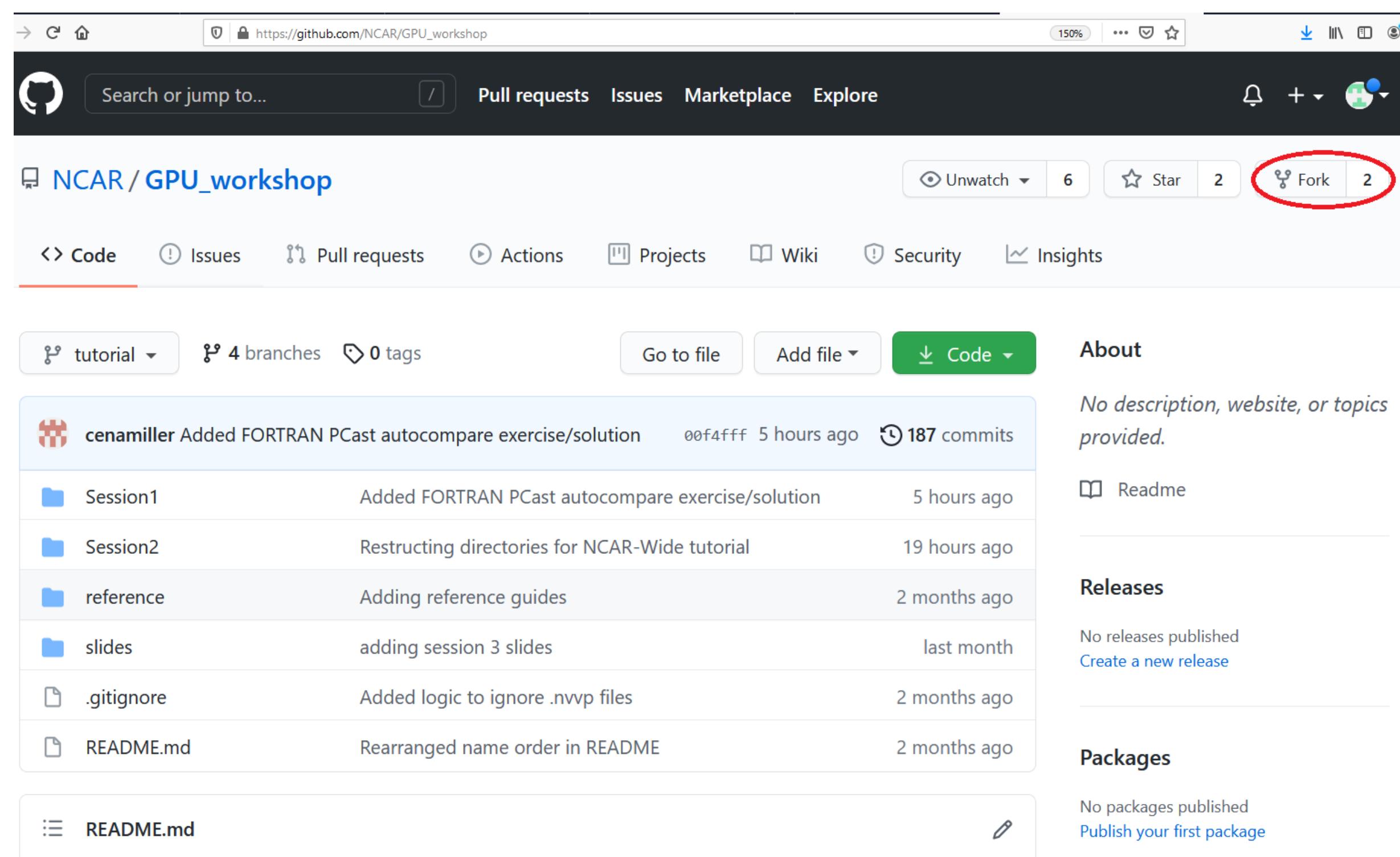
- Introduction to GPU architecture, key concepts, and terminologies
- OpenACC programming model
- Connecting to Rescale & Accessing Code
- Breakout Room 1
 - Coding Practice - Matrix Multiplication
 - Coding Practice - Matrix Multiplication with Tile Clause
- Profiling - NVprof and PCAST
- Breakout Room 2
 - Coding Practice - NVprof
 - Coding Practice - PCAST

The GitHub Repo

The code samples and versions of the slides are available at the link below:

[https://github.com/NCAR/GPU workshop](https://github.com/NCAR/GPU_workshop)

Before we begin, you should open this repository in a browser. In order to save your progress in this workshop, you should click the "Fork" button in the top right to have your own version of this repository. When the session wraps up today you should use **git add**, **git commit**, and **git push** to update your fork of the repository.

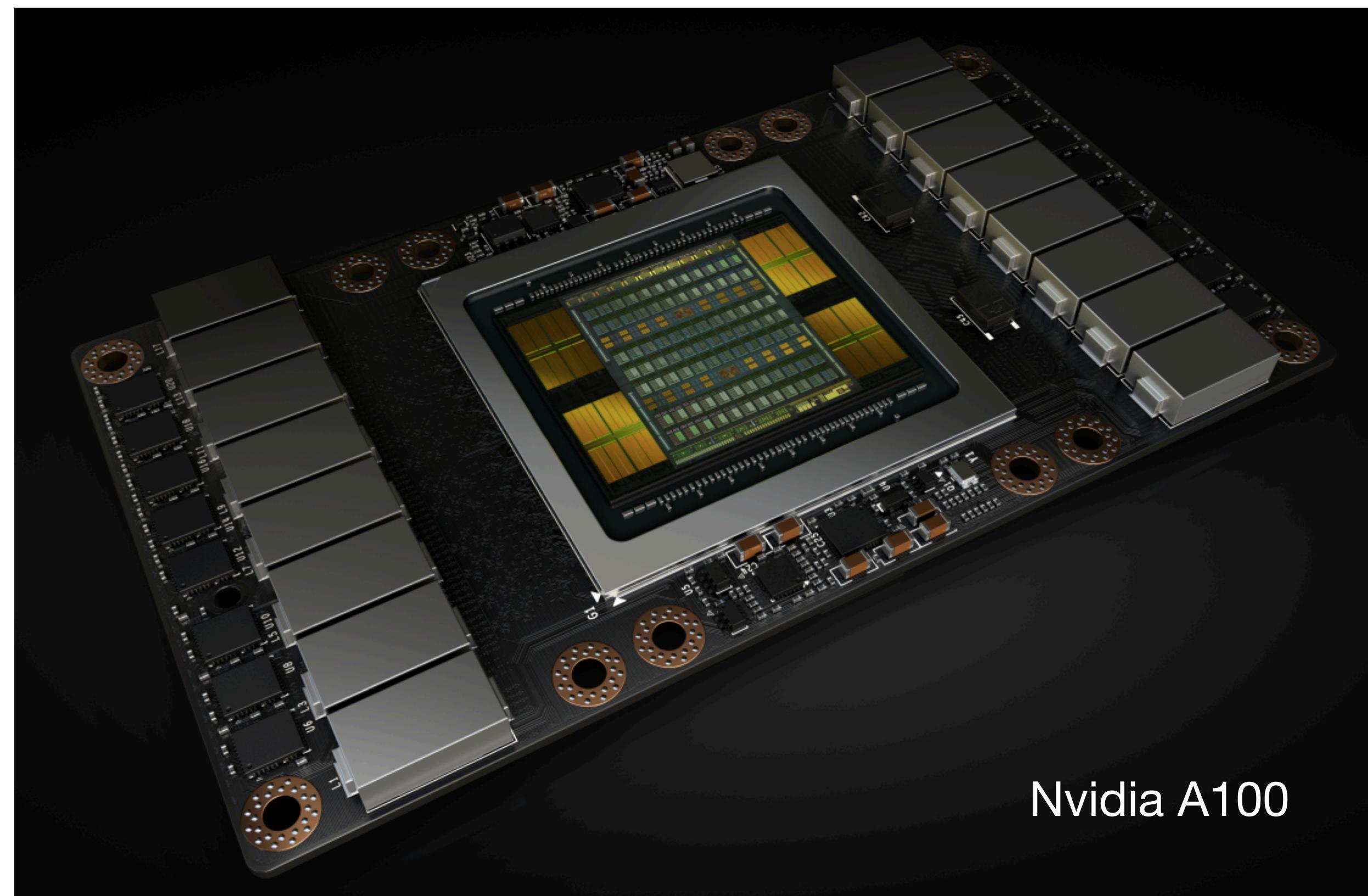


Introduction and GPU Architecture



Why GPUs?

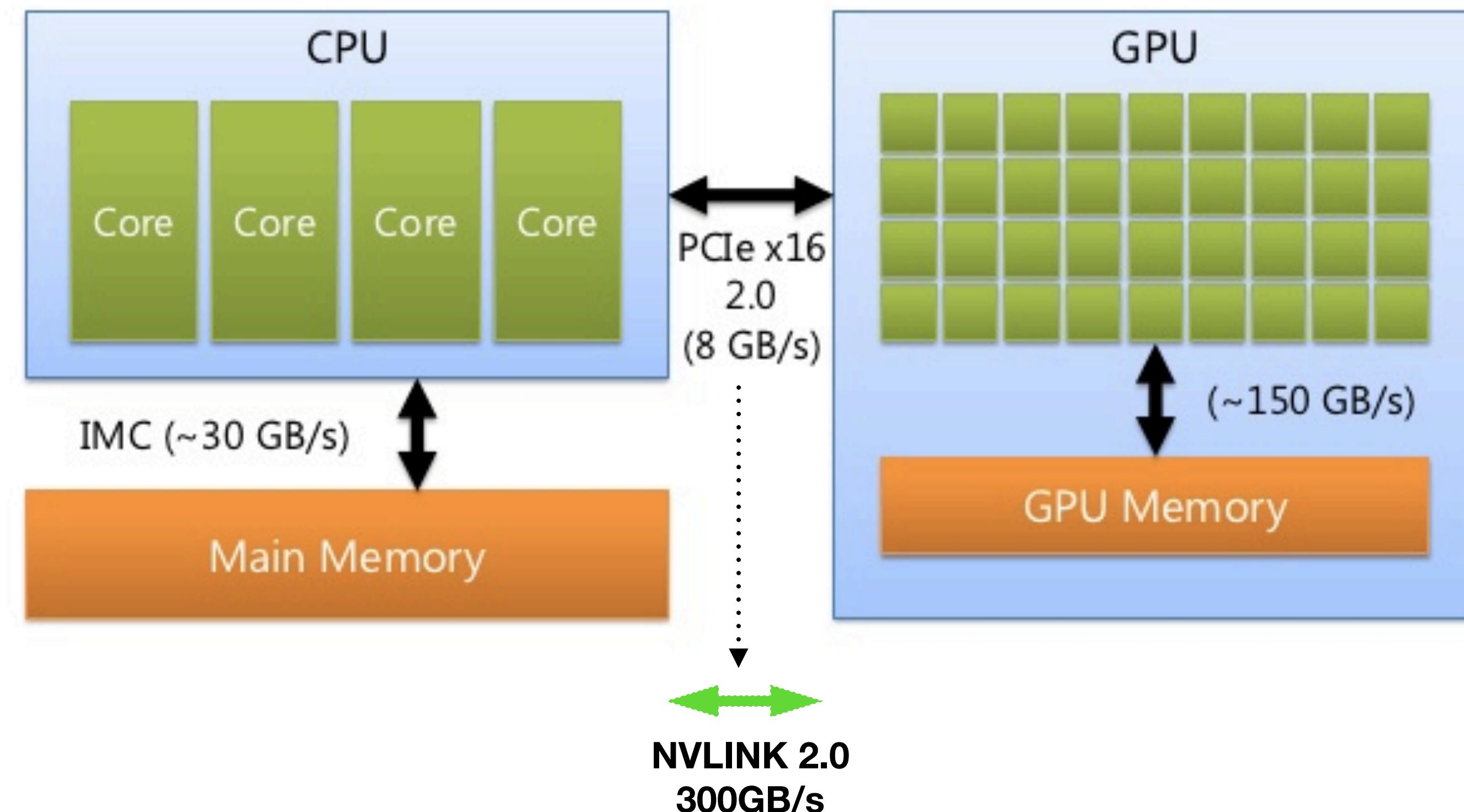
- Classic CPU single-threaded performance reached its limit
- GPUs naturally can handle massively parallel compute loads at high bandwidth
- GPUs is a **throughput-oriented**, many-core architecture, with thousands of cores bundled into dozens of SIMD multiprocessors
- Operational latencies for executions are higher on the GPU than on the CPU because of clock difference
- Operational latencies must be covered by thread-level and instruction-level parallelism
- GPUs have their own RAM, which can be accessed by the host



CPU-GPU Relationship

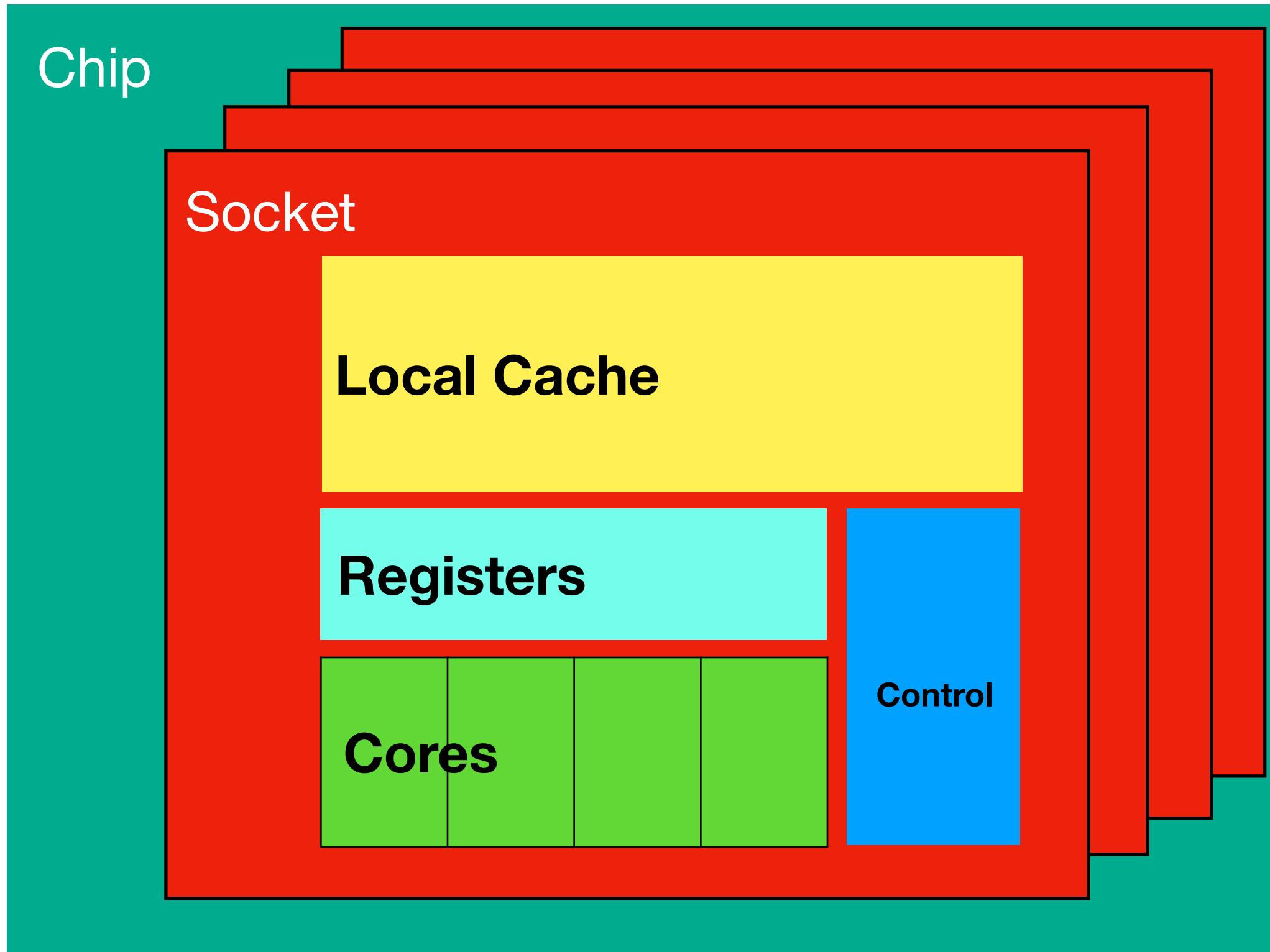
- Transfer between the chip and the GPU is conducted over PCI express bus or NvLink (Power systems)
- The third version of PCI-3
 - Delivers 32GB/s per lane for 'x' number of lanes
- The more lanes, the faster the transfer

- GPU: coprocessor with its own memory

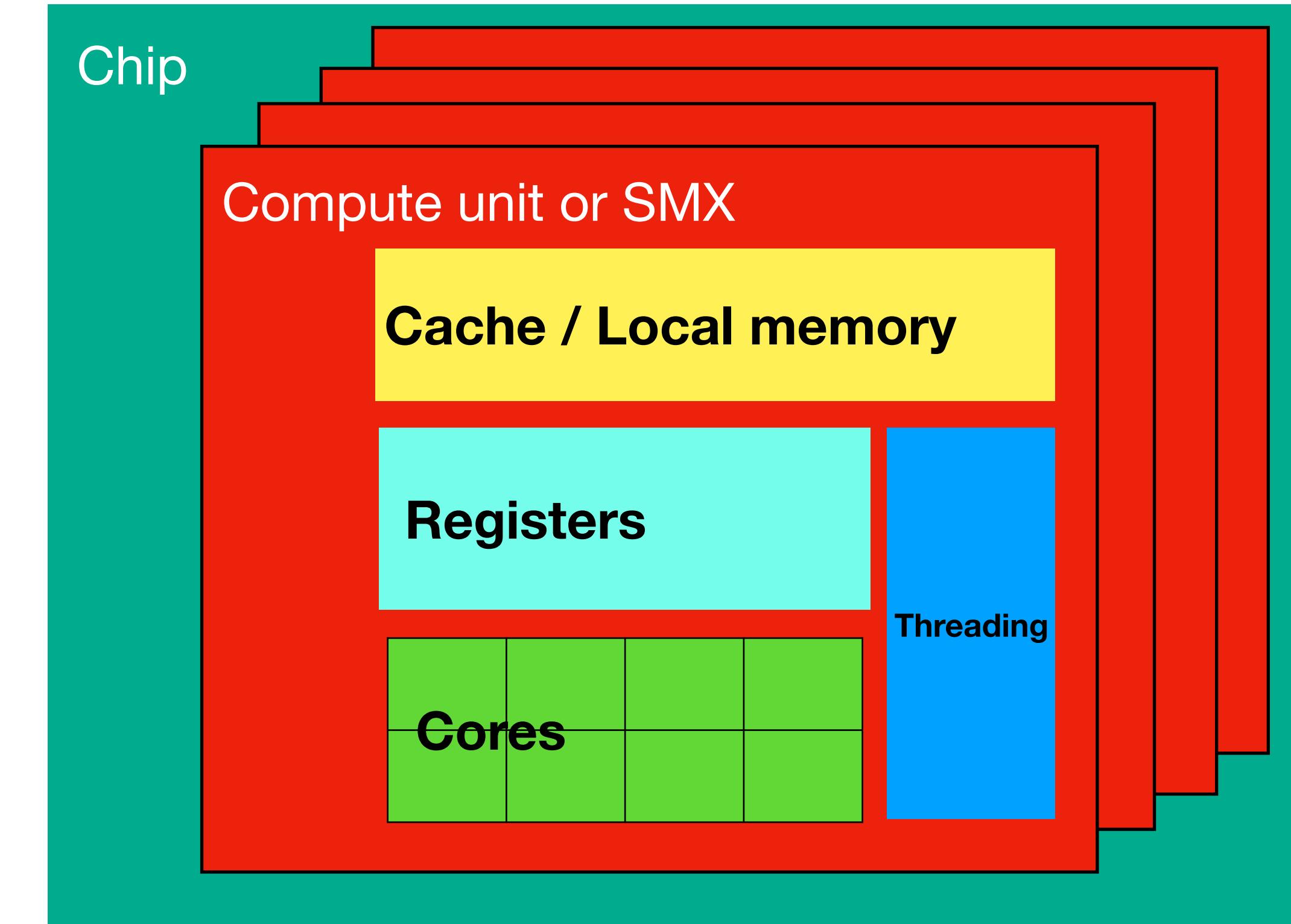


CPU and GPU are Designed Very Differently

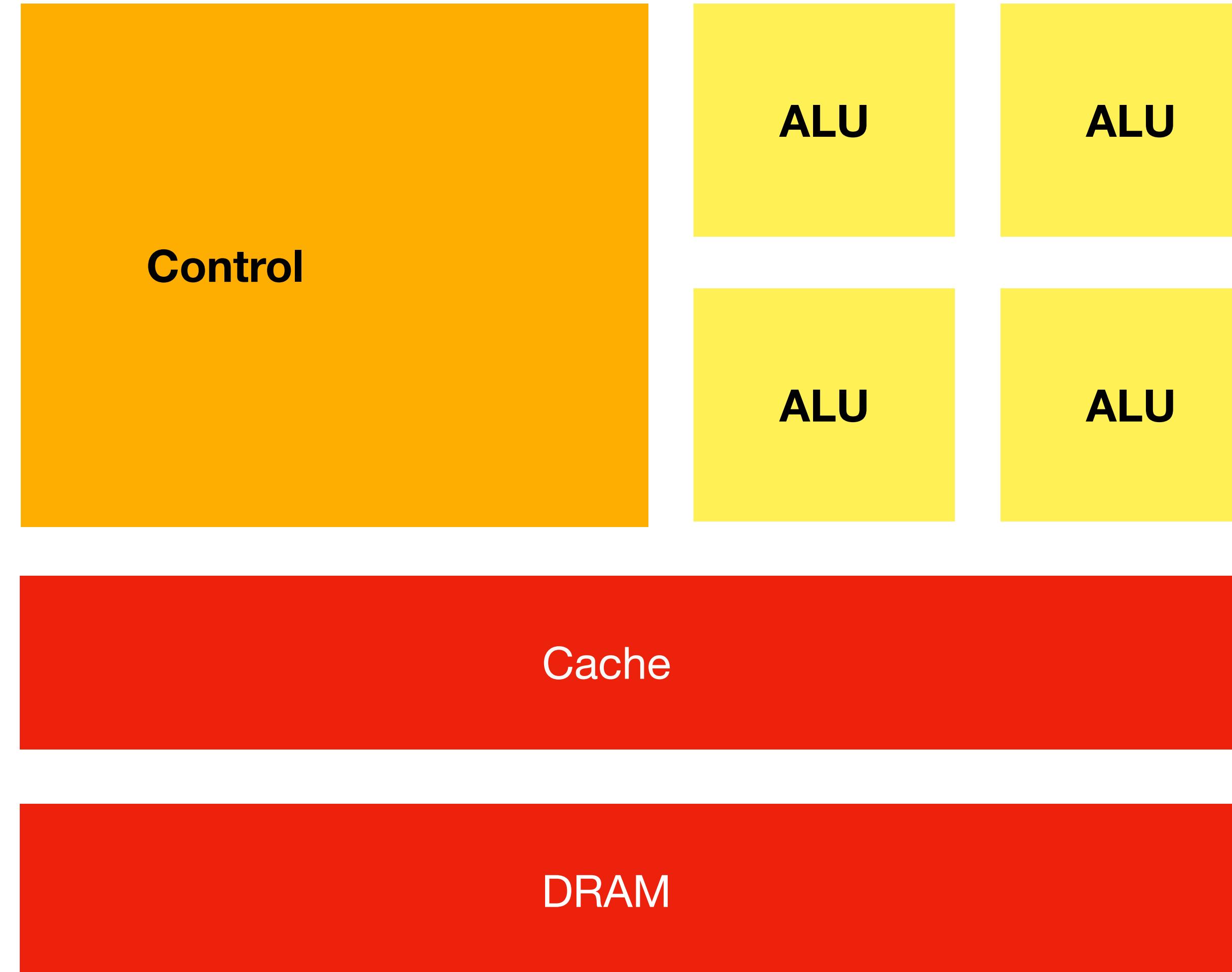
CPU
Latency Oriented Cores



GPU
Throughput Oriented Cores

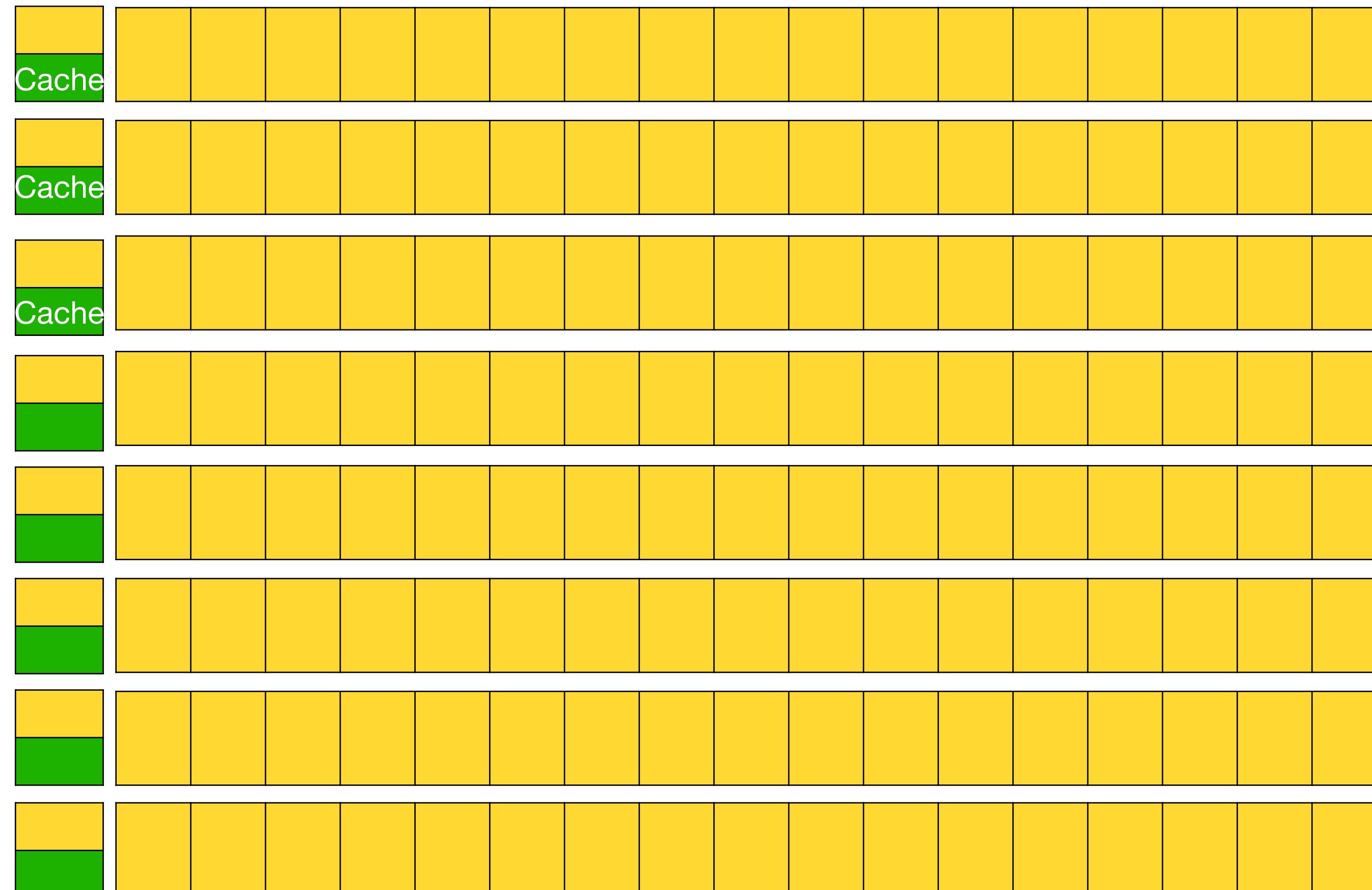


CPUs: Latency Oriented Design



- Powerful ALU
 - Great for serial operations
 - Reduced operation latency
- Large caches
 - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
 - Branch prediction for reduced branch latency
 - Data forwarding for reduced data latency (Data prefetch)

GPUs: Throughput Oriented Design

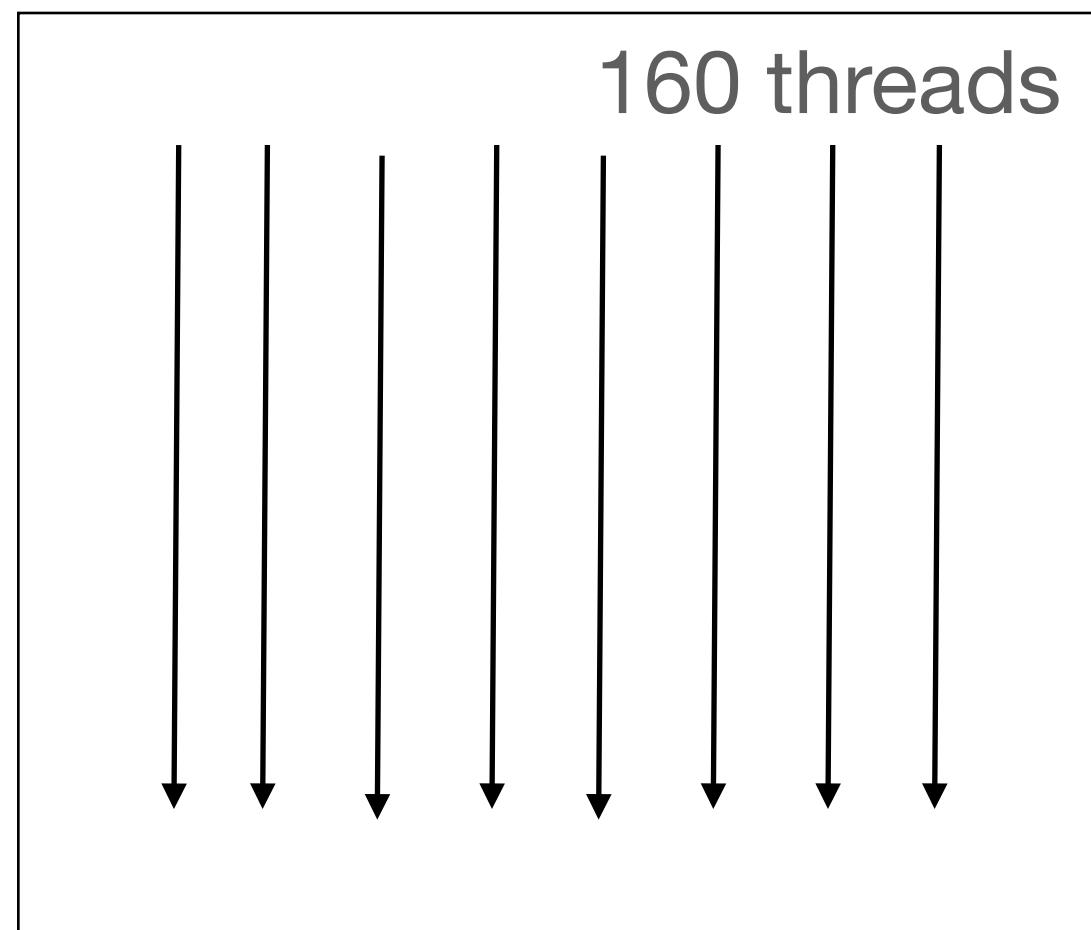


- Simple but a very large number of cores devoted to Floating Point operations.
- Small caches to boost memory throughput
- Simple control
 - No branch prediction
 - No data prefetching
- Require massive number of threads to hide operational latencies
 - Threading logic

Warps

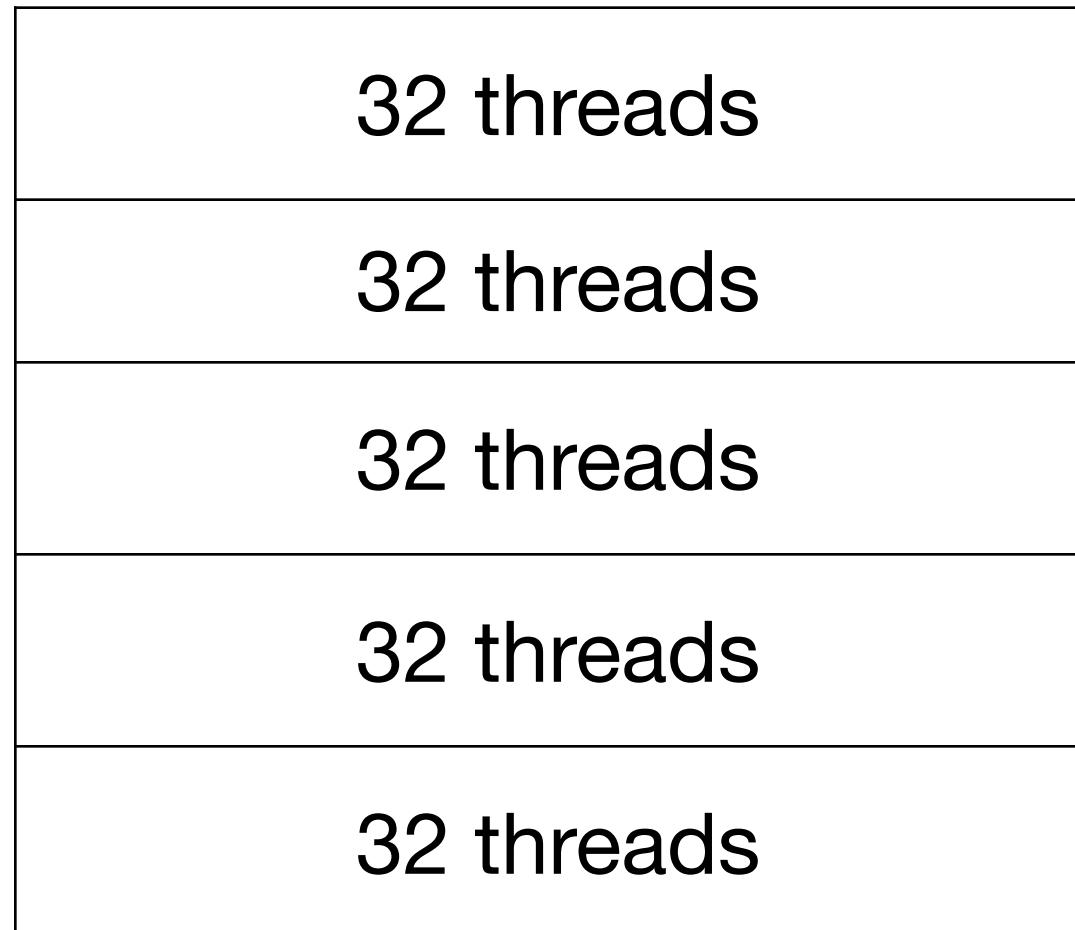
- A Warp is a collection of 32 threads
- GPU always executes all 32 threads in a Single Instruction Multiple Threads (SIMT) approach
- Due to SIMT all threads in a warp normally execute the same instruction in the same single clock cycle

Logical view



Thread block

Hardware view



Execution view

Warp scheduler			
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP

CPU vs GPU Threads & Register Usage per Thread

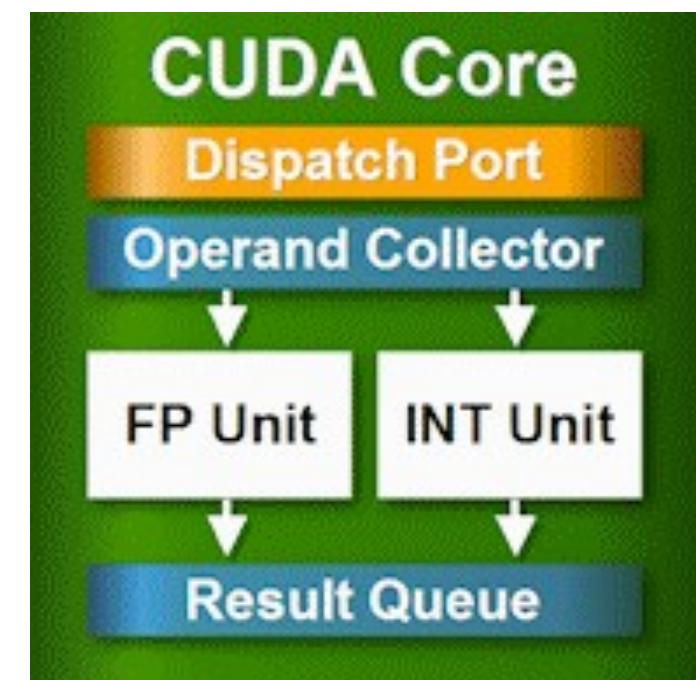
CPU threads	GPU threads
<ul style="list-style-type: none">• Heavy weight threads• Plenty of resources per thread• Limited number of threads can be launched because number of cores are limited• Data cached through L1, L2 and L3 cache	<ul style="list-style-type: none">• Light weight threads• Very limited resources per thread• A very high number of threads can be launched• Data cached through Shared memory and L2 cache• Requires special attention to number of registers allocated per thread

Software Abstraction



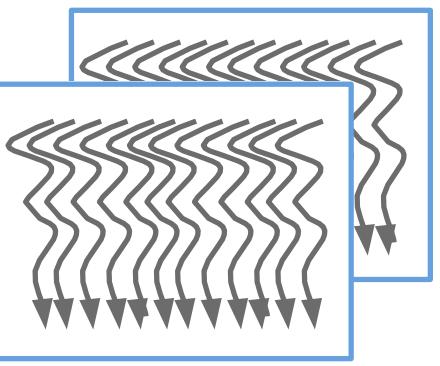
Software term

Kernel is executed by threads
processed by CUDA Core



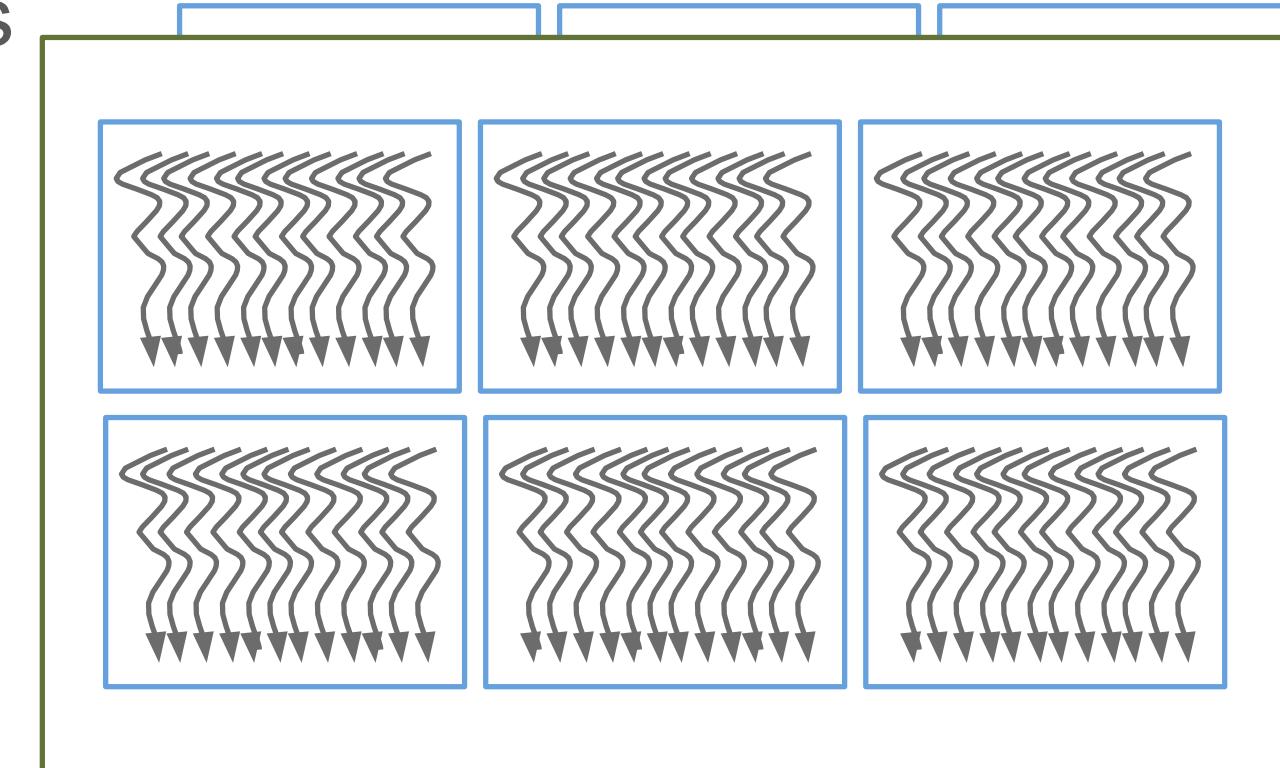
Hardware term

Vector (OpenACC)
Blocks



512–1024 threads / block

Grids

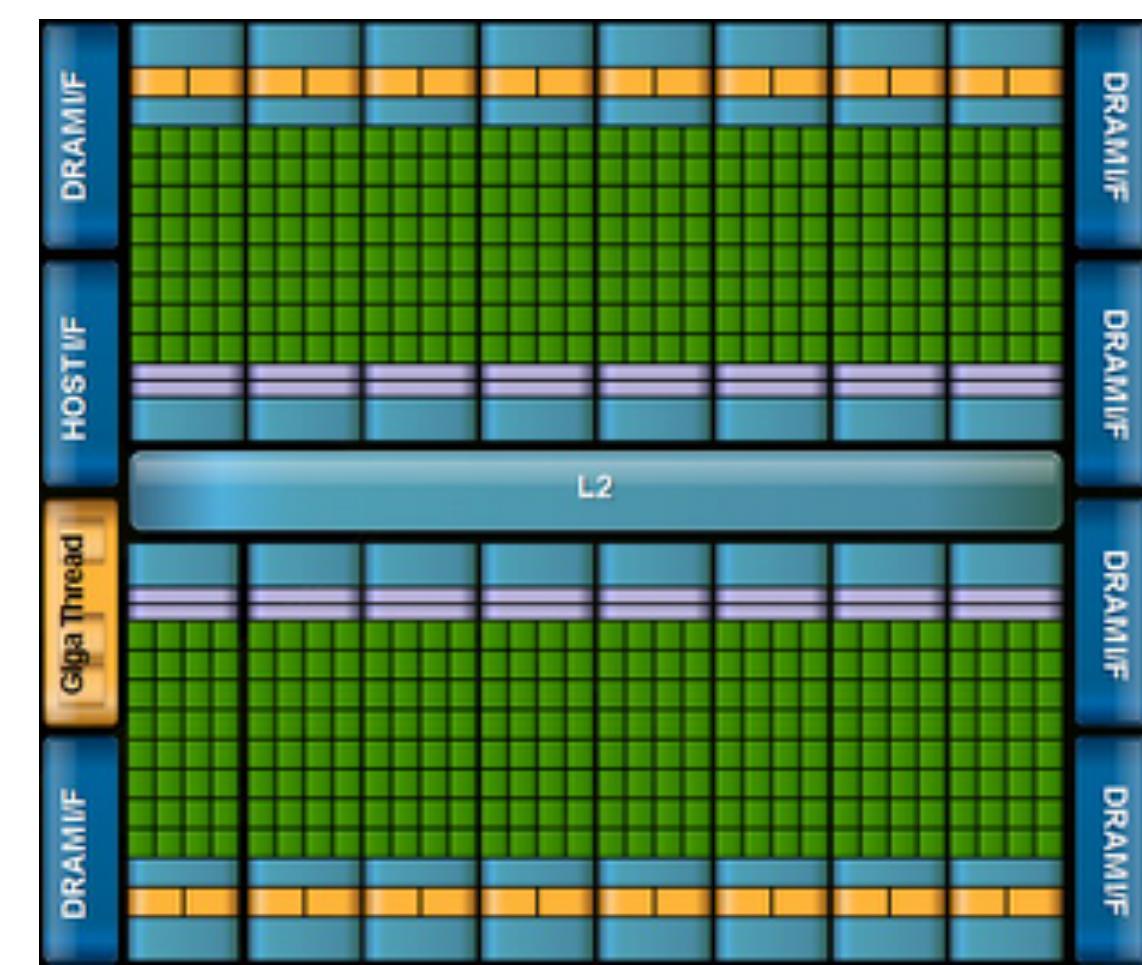


gangs (OpenACC)

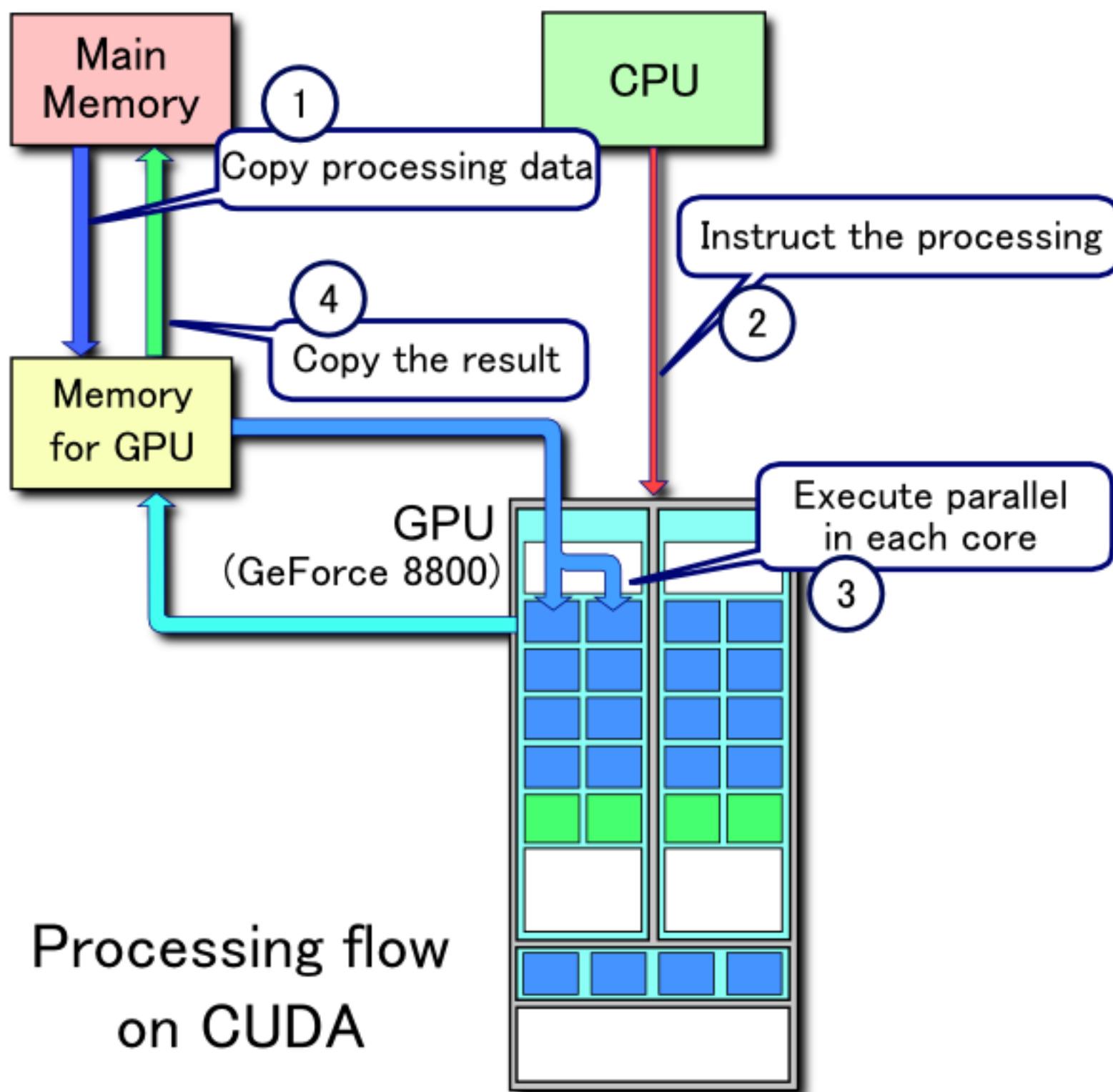
Maximum 8 blocks per SM
32 parallel threads are
executed at the same time in
a *WARP*



One grid per kernel with
multiple concurrent kernels



Developing a GPU Program



Example of a GPU process flow*

1. Copy data from main memory to GPU memory
2. CPU initiates threads on the GPU and launches the kernel on GPU
3. GPU executes parallel code
4. Copy the results from GPU memory to main memory

* This flow occurs in both OpenACC/CUDA.

<https://en.wikipedia.org/wiki/CUDA>

OpenACC



OpenACC Introduction

- OpenACC is a directives-based parallel programming model that is designed for performance and portability
- Enables easy conversion of sequential CPU based code to GPUs by adding pragmas
- General syntax:
 - C - #pragma acc [directive] [clauses]
 - FORTRAN - !\$acc [directive] [clauses]
- Multiple directives can be chained together in the same line and applied to the same blocks of code
- There is no need for explicit allocation/copying of data to and from the GPU in OpenACC



OpenACC Directives/Constructs

Construct	C code	Description
Kernels	#pragma acc kernels [clauses]	Surrounds loops to be executed on the GPU
Parallel	#pragma acc parallel [clauses]	Launches a number of gangs in parallel each with a number of workers, each with vector or SIMD operations
Loop	#pragma acc loop [clauses]	Applies to the immediately following loop or nested loops and describes the type of parallelism to execute these loops on the GPU
Data	#pragma acc data [clauses]	Defines a region of the program within which data is accessible by the GPU
Host data	#pragma acc host_data [clauses]	Makes the address of the device data available on the host
Cache	#pragma acc cache (list)	Added to the top of the loop. The elements in the list are cached in the software managed data cache
Update	#pragma acc update [clauses]	Copies data between the host memory and the data allocated on the GPU memory, and vice-versa
Clause - Collapse	#pragma acc loop collapse(n)	The collapse clause is used to specify how many tightly nested loops are associated with the loop clause.
Clause - Reduction	#pragma acc loop reduction(+:temp)	The reduction clause specifies a reduction operator and one or more vars.

OpenACC Introduction

- A vector is a group of threads that execute an operation on a chunk of data at once
- A worker executes one vector or SIMD operation. Workers in the same gang are able to share resources such as memory
- Gangs are a group of workers which in turn execute vectorized / SIMD operations

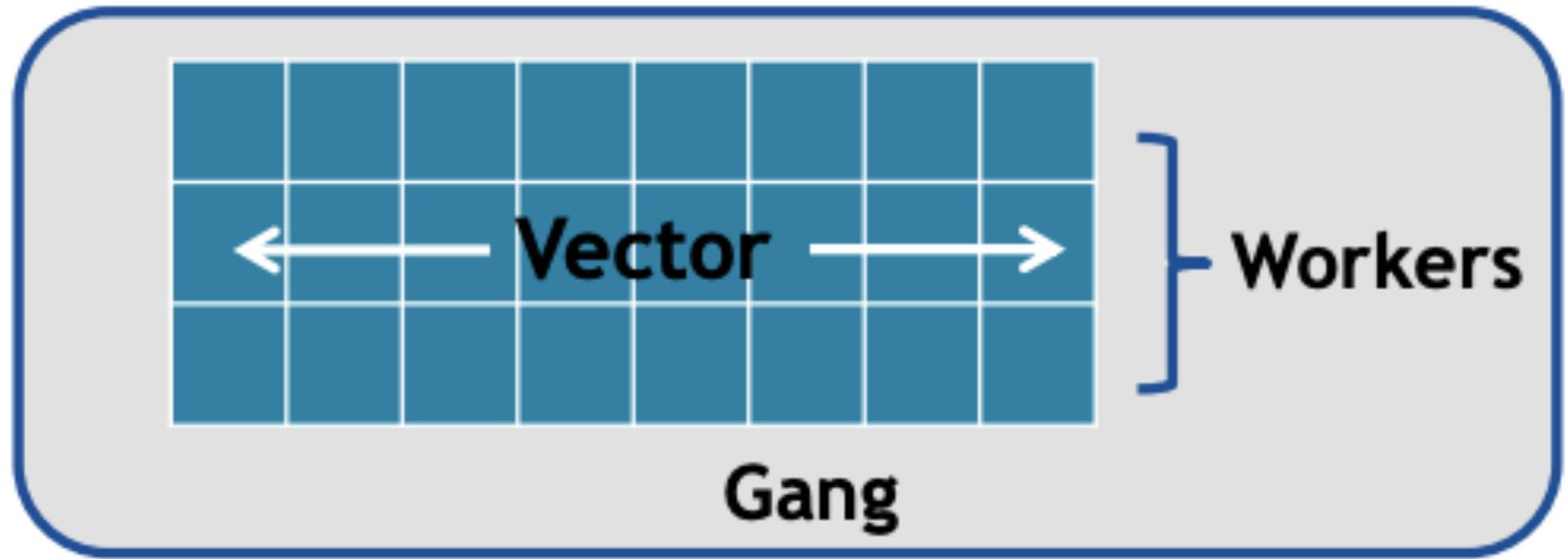


Image from <https://www.olcf.ornl.gov tutorial slides>

Kernels vs Parallel construct

- Kernels construct - The compiler is able to detect parts of the code that can be parallelized and parallelizes it. Sometimes it can parallelize the code better than the programmer.
- The downside is the compiler is conservative, and may not parallelize code that it deems unsafe

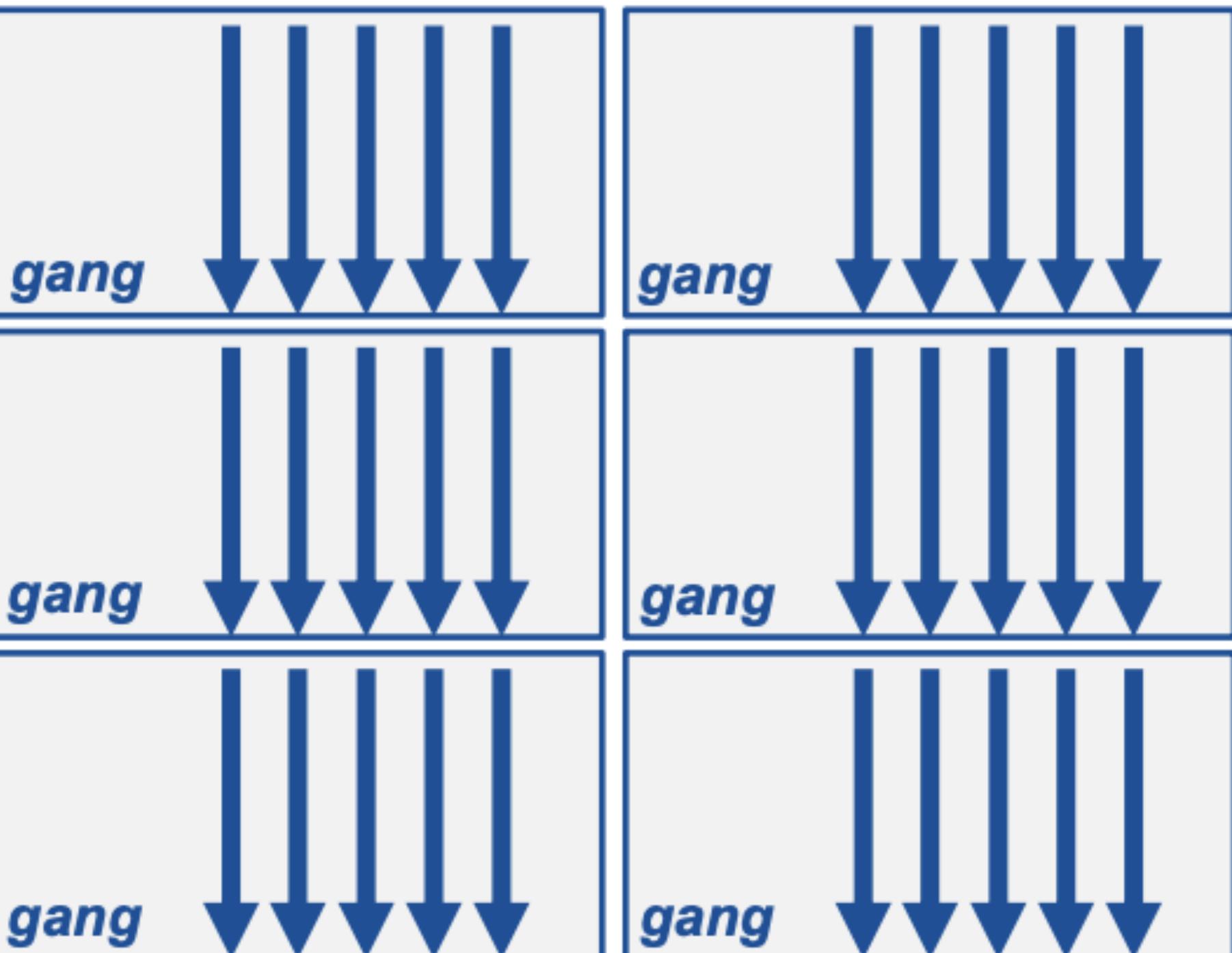
OpenACC Introduction

When you use "#pragma acc parallel", multiple gangs of workers are launched. The "#pragma acc loop" directive tells the compiler how to evenly split the work across the gangs.

C/C++

```
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; j < N; i++)
        a[i] = 0;
}
```

- parallel directive tells the compiler to launch one or more parallel gangs.
- without the loop directive each of the gangs will run the same loop redundantly which is not efficient.
- when the compiler encounters the loop directive it parallelizes the block of code immediately after it, and splits the work across each of the gangs equally.



OpenACC Introduction

Data construct

- `#pragma acc data [clauses]` defines a region below it within which the data is accessible by the GPU.

Clause	Description
copy(list)	1. Allocates the data in list on the GPU 2. Copies the data from the host to the GPU when entering the region 3. Copies the data from the GPU to the host when exiting the region
copyin(list)	Allocates the data in list on the GPU and copies the data from the host to the GPU when entering the region.
copyout(list)	Allocates the data in list on the GPU and copies the data from the GPU to the host when exiting the region.
create(list)	Allocates the data in list on the GPU, but does not copy data between the host and device.
present(list)	The data in the list must be already present on the GPU from some containing data region, which then can be found and used.

The information in this table and more about different directives and clauses at:
<https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf>

Breakout Session 1

- **Task 1: Connecting & Accessing the Code**
- **Task 2: Coding Practice - Matrix Multiplication**
- **Task 3: Coding Practice - Matrix Multiplication with Tile Clause**

General guidelines for screen sharing

https://docs.google.com/document/d/1U1jJ1W58SVDrXy7hJK9Yw7pAXQelWomKbU5g_u_nXHk/edit?usp=sharing

Task 1: Accessing the Workshop Instance

- Download the ssh key given to you by email yesterday
- Change file permissions of the private key

```
chmod 600 <path_to_private_key>
```

- Use ssh from a terminal window to connect to the instance IP address given to you
 - -i : Specifies identity file/private key
 - -p : Specifies the port number to use to connect

```
ssh -i <path_to_private_key> -p 22 <instance IP>
```

Find your instance IP and username here or ask your mentor later:

https://docs.google.com/spreadsheets/d/1m0W_u4XNAT2j6XjQr4hwElbft2OX_5xwE9zEuR91-3Q/edit?usp=sharing

Task 1: Using Container on Rescale

- Navigate to work/shared/<user>
- Fork the repo if you haven't already (from https://github.com/NCAR/GPU_workshop)
- Git clone **your forked** repository:
 - `git clone https://github.com/<GitHub Username>/GPU_workshop.git`
 - or (only if you are unable to fork or to use “git clone”) --
 - `wget "https://github.com/NCAR/GPU_workshop/archive/refs/tags/v1.0.tar.gz"`
- Start a shell using the Singularity container
 - `singularity shell --nv <path_to_cuda_inst.sif>`
- Once inside the container, try the command “nvidia-smi”. It should print information about your instance’s GPU.
- Navigate to <user>/GPU_workshop/Session1/

Task 1: File Structure

Example FORTRAN directory

```
|- exercise
  |- build.sh
  |- Makefile
  |- matrix_mult.f90
  |- submit.sh
|- solution
  |- build.sh
  |- Makefile
  |- matrix_mult.f90
  |- submit.sh
```

- Each lesson has a C/C++ or FORTRAN example code. Within those, there are two sub-directories:

- **Exercise** - the code that you will be working on.
- **Solution** - a *minimal* working code

To run the exercise:

1. “cd” into desired directory
2. Complete your porting task
3. Run “make” to build executable
4. ./example.exe <arg1> <arg2> ...

Example C/C++ directory

```
|- exercise
  |- build.sh
  |- common.cpp
  |- functions.cpp
  |- main.cpp
  |- Makefile
  |- matrix_mult.cc
  |- pch.h
  |- submit.sh
|- solution
  |- build.sh
  |- common.cpp
  |- functions.cpp
  |- main.cpp
  |- Makefile
  |- matrix_mult.cc
  |- pch.h
  |- submit.sh
```

Task 2: Matrix Multiplication with OpenACC

Task 2: Naive Matrix Multiplication with OpenACC

Parallelizing code with OpenACC involves adding directives to your sequential code. First of all, lets take a look at our CPU matrix multiplication code that runs sequentially.

```
// calculate Ax=B
float temp = 0.0;
for (int i = 0; i < rowsA; i++)
    for (int j = 0; j < colsB; j++)
    {
        temp = 0.0;
        for (int k = 0; k < rowsB; k++)
        {
            temp += A[i*rowsB+k] * B[k*colsB+j];
        }
        C[i*colsB+j] = temp;
    }
```

We will be adding OpenACC directives to this base version of our matrix multiplication code.

Task 2: Naive Matrix Multiplication with OpenACC

We will be adding OpenACC directives to the matrix multiplication function:

- Add a data directive that copies in/out the required variables.
- Add a directive to tell the compiler what region needs to be parallelized
- Add a clause that collapses the two tightly-nested for loops. Note: Add the reduction clause.
- Add a directive that vectorizes the inner most loop. Note: We will also be adding the reduction clause to this directive.

Task 2: Naive Matrix Multiplication with OpenACC - C/C++

We will be adding OpenACC directives to offload the matrix multiplication to the GPU:

- Add a **data directive** that copies in/out the required Matrix variables.
- Add a directive to mark the **parallel region** for the compiler
- Add a directive that **collapses** the two tightly-nested for loops. Note: Add the reduction clause.
- Add a directive that vectorizes the inner most loop. Note: We will also be adding the reduction clause to this directive.

Close the data region

C/C++
In matrix_mult.cc

```
float temp = 0.0;  
#pragma acc data copyout(C[0:rowsA*colsB]) \  
    copyin(A[0:rowsA*rowsB],B[0:rowsB*colsB])  
{  
#pragma acc parallel loop collapse(2) \  
    reduction(+:temp)  
    for (int i = 0; i < rowsA; i++)  
        for (int j = 0; j < colsB; j++)  
        {  
            temp = 0.0;  
#pragma acc loop reduction(+:temp)  
            for (int k = 0; k < rowsB; k++)  
            {  
                temp += A[i*rowsB+k] * B[k*colsB+j];  
            }  
            C[i*colsB+j] = temp;  
        }  
}
```

Task 2: Naive Matrix Multiplication with OpenACC - FORTRAN

We will be adding OpenACC directives to offload the matrix multiplication to the GPU:

- Add a **data directive** that copies in/out the required Matrix variables.
- Add a directive to mark the **parallel region** for the compiler
- Add a directive that **collapses** the two tightly-nested for loops. Note: Add the reduction clause.
- Add a directive that vectorizes the inner most loop. Note: We will also be adding the reduction clause to this directive
- Make sure to close parallel and data regions!

FORTRAN
In matrix_mult.f90, (line 90)

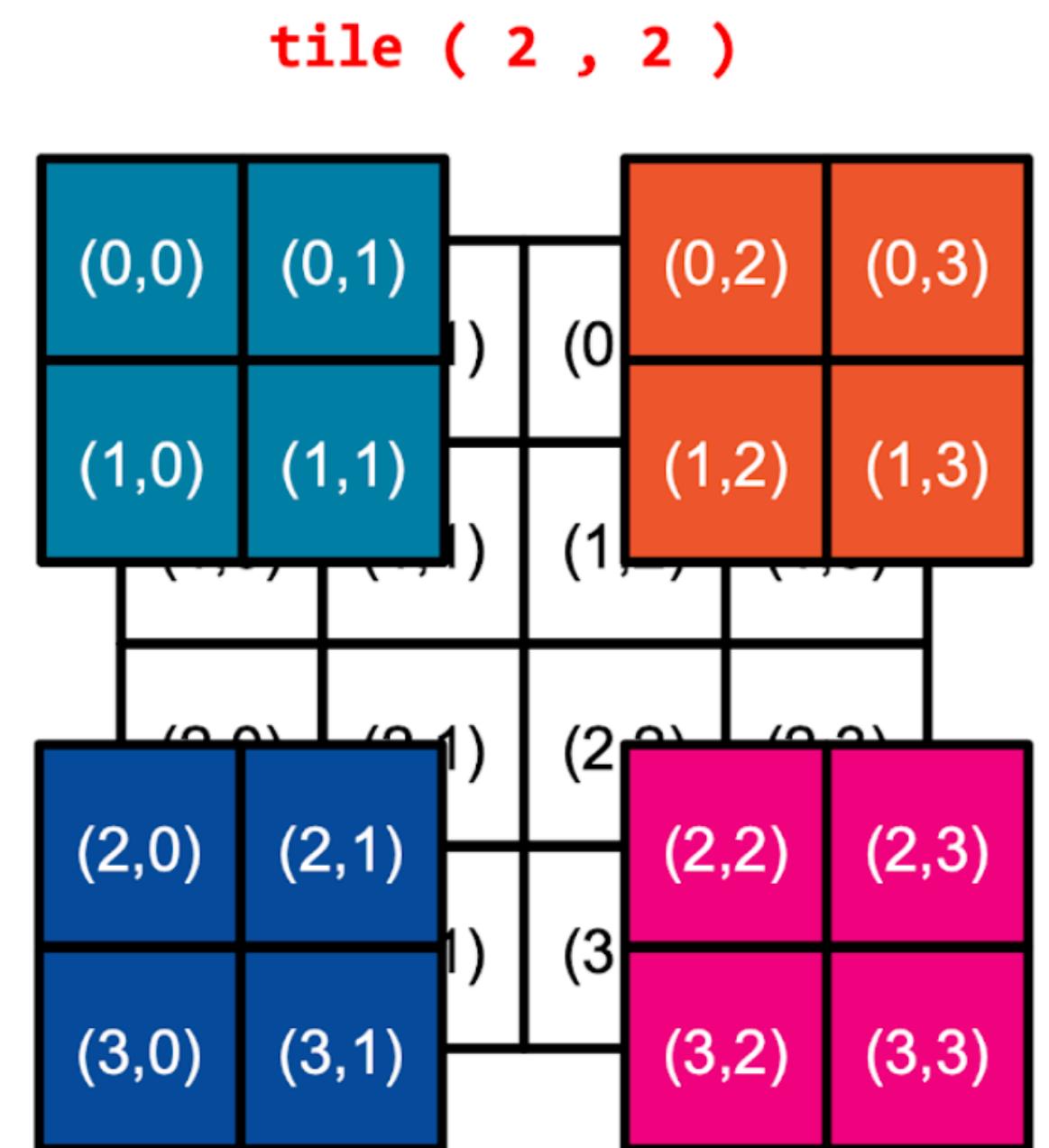
```
!$acc data copyin(a,b) copyout(c_gpu)
!$acc parallel loop collapse(2) reduction(: tmp)
    do j=1,colsB
        do i=1,rowsA
            tmp = 0.0
!$acc loop reduction(: tmp)
            do k=1,rowsB
                tmp = tmp + a(i,k) * b(k,j)
            enddo
            c_gpu(i,j) = tmp
        enddo
    enddo
!$acc end parallel
!$acc end data
```

Task 3: Matrix Multiplication with Tile Clause

Task 3: Matrix Multiplication with Tile Clause

- `tile(x, y, z, ...)`
- Breaks multidimensional tightly nested loops into “tiles”
- Can increase data locality in some codes, exploit caching
- Execute multiple “tiles” simultaneously
- Product of dimensions must be less than or equal to maximum number of threads
- Experiment to find best options (some combinations can hurt performance)

```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
    for(int y = 0; y < 4; y++){
        array[x][y]++;
    }
}
```



Source: <https://www.olcf.ornl.gov/calendar/loop-optimizations-with-openacc/>

Task 3: Matrix Multiplication with Tile Clause

We will be adding OpenACC directives to the matrix multiplication function:

- Add a data directive that copies in/out the required variables.
- Add a directive to tell the compiler what region needs to be parallelized.
- Add a clause that creates the tiling for the **i** and **j** loops. Also add the reduction clause.
- Add a directive that sequentializes the inner most loop. Note: We will also be adding the reduction clause to this directive.

Go to
Breakout Session 1

Return to main room:
~15 mins

Breakout Session 2

- **Task 4: Profiling with NVprof**
- **Task 5: Error Checking with PCAST**
- **Task 6: Compiler OpenACC Profiling (NV_ACC_TIME)**

Task 4: Introduction to nvprof (NVIDIA Profiler)

nvprof is a command line profiler that provides information about your program execution along with timing information.

Below is an example of the nvprof output from our Matrix Multiplication code. This output is generated by adding the nvprof command while running our executable. Example: `nvprof --unified-memory-profiling off ./example.exe`

```
pgc++ -c -g -O3 -std=c++11 -Wall main.cpp
pgc++ -c -g -O3 -std=c++11 -Wall functions.cpp
pgc++ -c -g -O3 -std=c++11 -Wall common.cpp
nvcc -c -g -std=c++11 -O3 -I/glade/u/apps/dav/opt/cuda/11.0.3//include matrixMul.cu
pgc++ -o output.exe main.o functions.o common.o matrixMul.o -L/glade/u/apps/dav/opt/cuda/11.0.3//lib64 -lcudart
==149693== NVPROF is profiling process 149693, command: ./output.exe

Entire CPU process took 3.360000 seconds.... Starting Cuda Process.
Done. Matrix multiplication on GPU took 0.360000 seconds.
==149693== Profiling application: ./output.exe
==149693== Profiling result:

      Type  Time(%)     Time    Calls      Avg      Min      Max  Name
GPU activities:   43.82%  1.4974ms      2  748.68us  745.47us  751.90us  [CUDA memcpy HtoD]
                  38.48%  1.3149ms      1  1.3149ms  1.3149ms  1.3149ms  mmul(float*, float*, float*, int, int, int)
                  17.71%  605.18us      1  605.18us  605.18us  605.18us  [CUDA memcpy DtoH]
API calls:       97.41%  249.69ms      3  83.231ms  271.31us  249.02ms  cudaMalloc
                  1.15%  2.9541ms      3  984.69us  975.76us  1.0010ms  cudaMemcpy
                  0.53%  1.3542ms      1  1.3542ms  1.3542ms  1.3542ms  cudaDeviceSynchronize
                  0.29%  752.64us     101  7.4510us  130ns   333.63us  cuDeviceGetAttribute
                  0.29%  745.28us      3  248.43us  224.24us  289.07us  cudaFree
                  0.27%  696.19us      1  696.19us  696.19us  696.19us  cuDeviceTotalMem
                  0.04%  90.345us      1  90.345us  90.345us  90.345us  cuDeviceGetName
                  0.02%  44.011us      1  44.011us  44.011us  44.011us  cudaLaunchKernel
                  0.00%  7.4660us      1  7.4660us  7.4660us  7.4660us  cuDeviceGetPCIBusId
                  0.00%  3.3140us      4    828ns   190ns   2.3040us  cudaGetLastError
                  0.00%  2.0040us      3    668ns   223ns   1.1470us  cuDeviceGetCount
                  0.00%  1.5240us      2    762ns   511ns   1.0130us  cuDeviceGet
                  0.00%    234ns      1    234ns   234ns   234ns   cuDeviceGetUuid
```

Task 4: Introduction to nvprof (NVIDIA Profiler)

A more detailed output can be generated by using: `nvprof --unified-memory-profiling off --print-gpu-trace ./example.exe`

This command is most useful when running your code on two GPUs. It will display what GPU each part/Kernel of your code is running on. Below is an example of the output from running the command above.

```
rm *.o output.exe
pgc++ -c -g -O3 -std=c++11 -Wall main.cpp
pgc++ -c -g -O3 -std=c++11 -Wall functions.cpp
pgc++ -c -g -O3 -std=c++11 -Wall common.cpp
nvcc -c -g -std=c++11 -O3 -I/glade/u/apps/dav/opt/cuda/11.0.3//include matrixMul.cu
pgc++ -o output.exe main.o functions.o common.o matrixMul.o -L/glade/u/apps/dav/opt/cuda/11.0.3//lib64 -lcudart
==42032== NVPROF is profiling process 42032, command: ./output.exe

Entire CPU process took 3.260000 seconds.... Starting Cuda Process.
Done. Matrix multiplication on GPU took 0.270000 seconds.
==42032== Profiling application: ./output.exe
==42032== Profiling result:
      Start Duration          Grid Size        Block Size       Regs*       SSMem*       DSMem*        Size Throughput SrcMemType DstMemType           Device Context Stream
      Name
377.59ms 798.97us          -                  -          -          -          -   4.0000MB 4.8891GB/s  Pageable    Device  Tesla V100-SXM2      1      7
  [CUDA memcpy HtoD]
378.62ms 784.35us          -                  -          -          -          -   4.0000MB 4.9802GB/s  Pageable    Device  Tesla V100-SXM2      1      7
  [CUDA memcpy HtoD]
379.42ms 1.4475ms (32 32 1) (32 32 1)      32          0B          0B          -   4.0000MB 6.2491GB/s  Device  Pageable  Tesla V100-SXM2      1      7
  mmul(float*, float*, float*, int, int, int) [118]
380.88ms 625.09us          -                  -          -          -          -   4.0000MB 6.2491GB/s  Pageable  Tesla V100-SXM2      1      7
  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
```

Task 4: nvprof profiling Matrix Multiplication with tile

During the next breakout room we want you to get some experience with nvprof.

Within your matrix multiplication with tile exercise code folder, run a command like:

```
nvprof --unified-memory-profiling off --print-gpu-trace ./matmul.exe
```

Where is the majority of the time spent in this program? Discuss with your mentors.

Task 5: PGI Autocompare

- PCAST is used to check the validity of the GPU code
- When compiled with the **-ta=tesla:autocompare** flag, code in OpenACC compute regions will run redundantly on the CPU and GPU
- Simplest way to invoke PCAST is to use "**-ta=tesla:autocompare,<other options>**" compiler flag to enable the autocompare feature
- Set the environment variable "**export PCAST_COMPARE=summary,abs=6,verboseautocompare**" in the shell to get a summary of comparisons with absolute error tolerance up to 10^{-6} and verbose output
- Any data in a **copy**, **copyout**, or **update host** directive will be compared when it is copied off the device.

Source: <https://www.pgroup.com/resources/pcast.htm>

Task 6: Compiler OpenACC Profiling

- When not using nvprof, here are two environment variables you can set that will provide information about the OpenACC execution:
 - **export NV_ACC_TIME=1**
 - prints kernel timing data after program execution finishes
 - **export NV_ACC_NOTIFY= [1,2,4,8,16]** (bitmask)
 - 1 - launch
 - 2 - data upload/download
 - 4 - wait (explicit or implicit) for device
 - 8 - data/compute region
 - 16 - data createallocatedeletefree

Go to
Breakout Session 2

Return to main room:
By 2.25

Extra Coding Practice: Matrix Mult Scaling Curves

- Scenario: We'd like you to gather performance data to compare how the GPU and CPU Matrix Multiplication code performs as the size of the matrices increases. We know it's not fair to compare the times for a single GPU to a single-threaded CPU, so **we'd like try using OpenACC's Multicore CPU capabilities** to run the code on a fully-subscribed CPU node. (Documented here: <https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/index.html#using-openacc>)
- Then, for increasing sizes of square matrix multiplication, you can fill out this example table with performance data. Feel free to optimize the GPU and CPU code - the fastest times (with correct results) win!

Square Mat Dimension	Single GPU Execution Time	Single CPU Node Execution Time
256		
512		
1024		
2048		
4096		
8192		

References

1. <https://www.olcf.ornl.gov/wp-content/uploads/2020/04/OpenACC-Course-2020-Module-1.pdf>
2. <https://www.olcf.ornl.gov/wp-content/uploads/2020/02/OpenACC Course 2020 Module 2.pdf>
3. <https://www.olcf.ornl.gov/wp-content/uploads/2020/06/OpenACC Course 2020 Module 3 updated.pdf>
4. <https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf>
5. <https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/>
6. <https://docs.computecanada.ca/wiki/OpenACC Tutorial - Optimizing loops>

Thank you for your time!

GitHub Repository with slides:[https://github.com/NCAR/GPU workshop](https://github.com/NCAR/GPU_workshop)

