Introduction to GPU programming (part 3)

By Kevin Fotso

Recap

Last time we learned about numba, a compiler (JIT) that translates python code into fast machine code.

Recap

Last time we learned about numba, a compiler (JIT) that translates python code into fast machine code.

We saw basic examples of GPU computation with numba.vectorize and numba.cuda

Recap

Last time we learned about numba, a compiler (JIT) that translates python code into fast machine code.

We saw basic examples of GPU computation with numba.vectorize and numba.cuda

We learned about cupy, an open source library providing GPU accelerated computing with Python.

Recap

Last time we learned about numba, a compiler (JIT) that translates python code into fast machine code.

We saw basic examples of GPU computation with numba.vectorize and numba.cuda

We learned about cupy, an open source library providing GPU accelerated computing with Python.

We saw very basic examples of matrix operation & cupy function

Today

- We will do a short recap

Today

- We will do a short recap
- Then we will learn about the PI example

Today

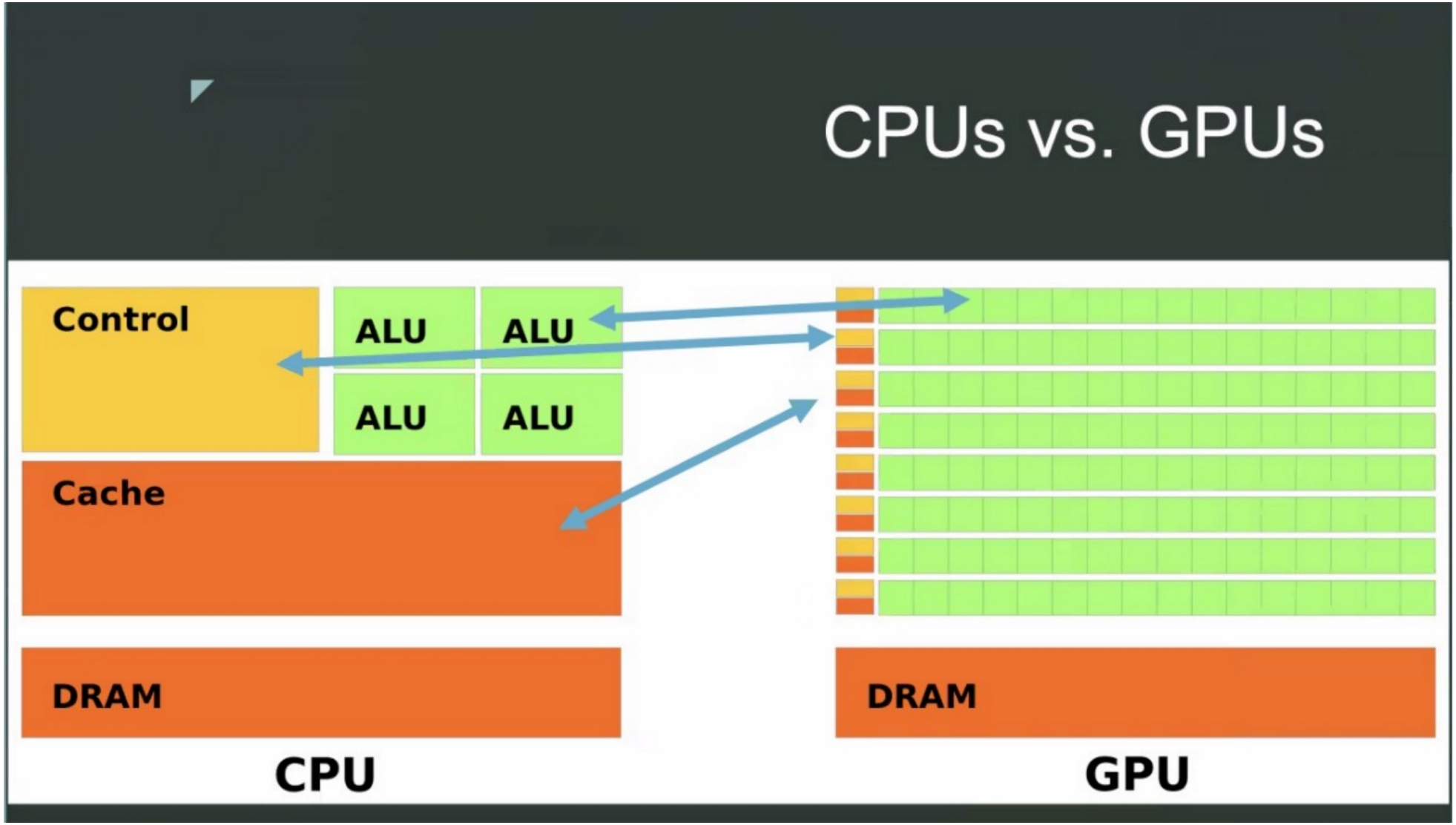
- We will do a short recap
- Then we will learn about the PI example
- Then we will convert it to GPU and compare the time duration

Today

- We will do a short recap
- Then we will learn about the PI example
- Then we will convert it to GPU and compare the time duration
- Finally, we will learn about profiling with nsys profiling

I - RECAP

CPU vs GPU comparison



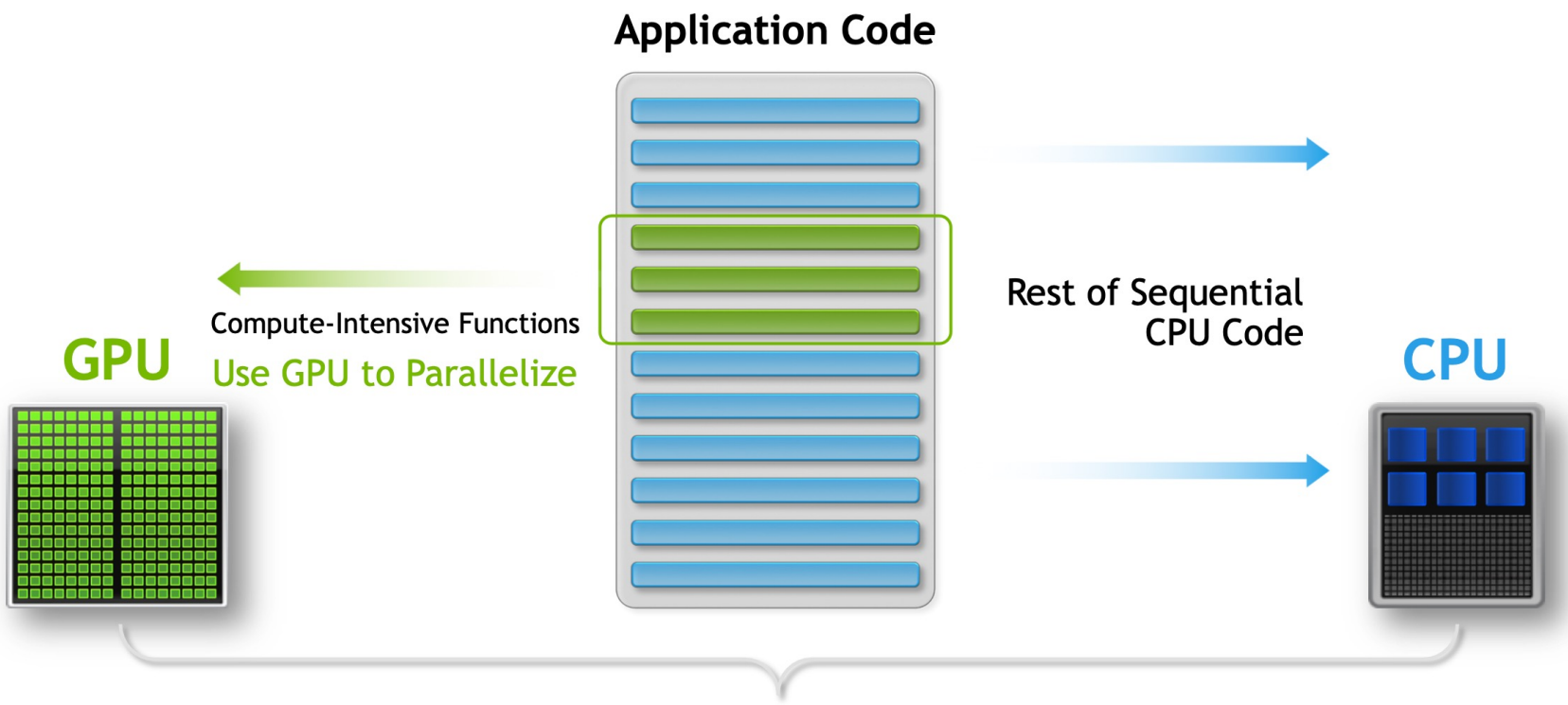
Source: Jayce Dowell Presentation on GPU University of New Mexico, Fall 2021

CPU vs GPU programming model

GPU: Matrix operations. (e.g. FFT)

CPU:

OS, security etc …

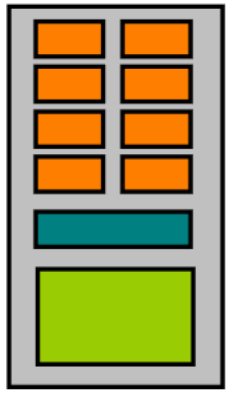


Source: https://www.olcf.ornl.gov/cuda-training-series/ ; CUDA C++

NVIDIA A100 specs

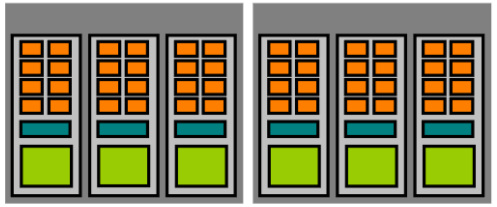
Scalar Processor

cuda core. **64 per SM**



Multiprocessor

streaming multiprocessor
: **108 SMs**

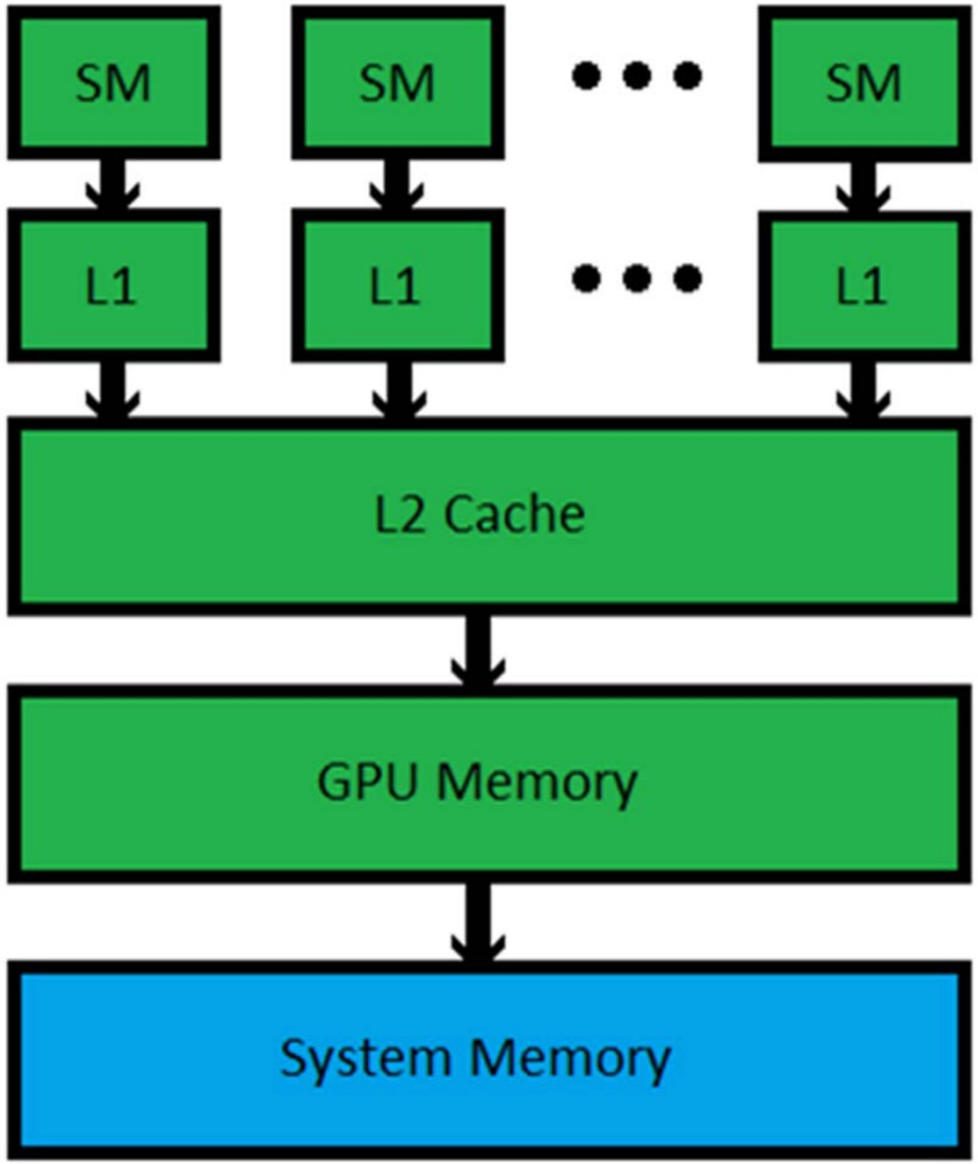


Device

108 SMs per device

Source: https://www.olcf.ornl.gov/wp-content/uploads/2019/12/03-CUDA-Fundamental-Optimization-Part-1.pdf

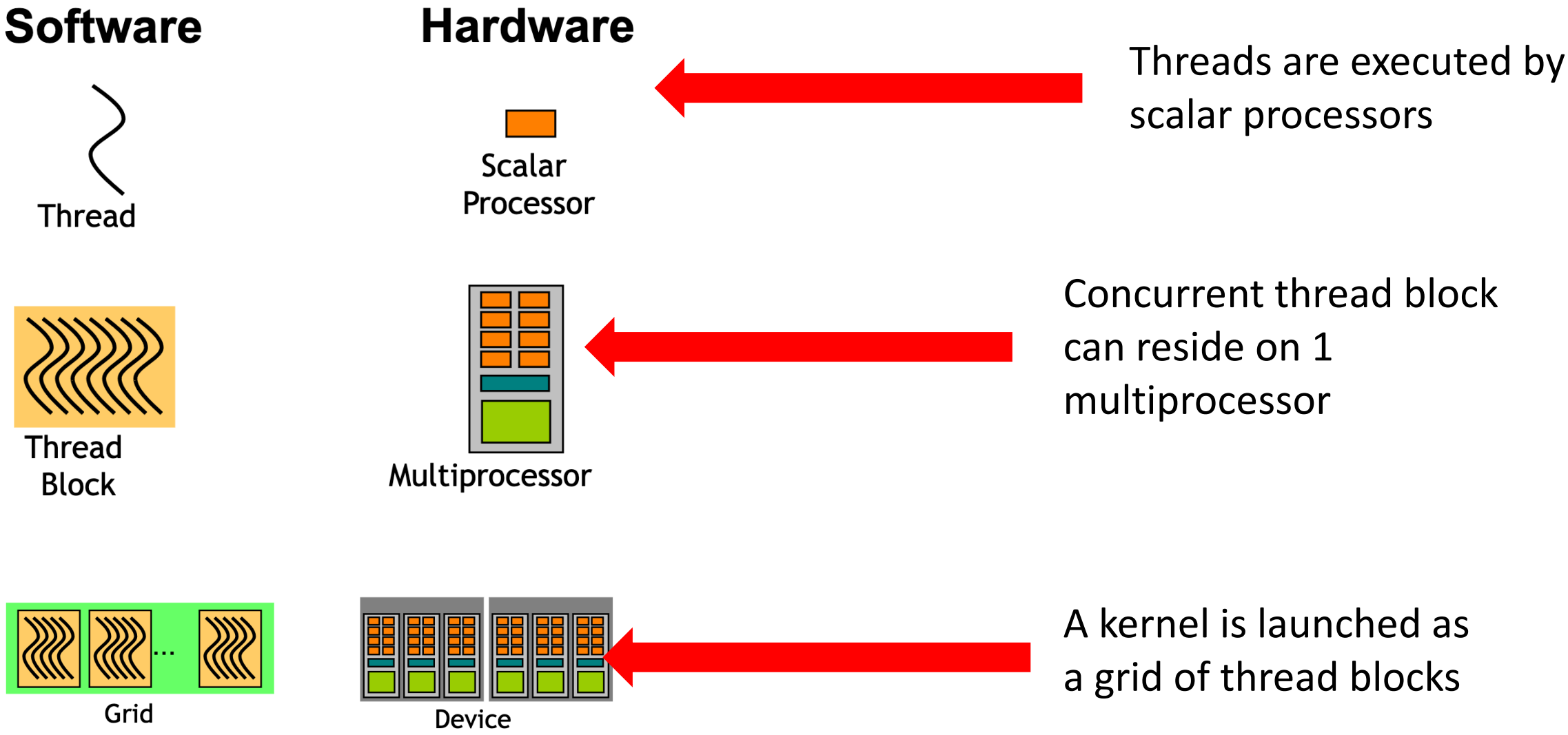
NVIDIA GPU memory hierarchy



- Registers and local memory (local memory is slower)
- Shared memory is accessible by all threads in the SM
- Global memory is accessible by all blocks and the CPU

Source: http://meseec.ce.rit.edu/551-projects/spring2015/3-2.pdf; https://people.duke.edu/~ccc14/sta-663/CUDAPython.html

GPU software programming



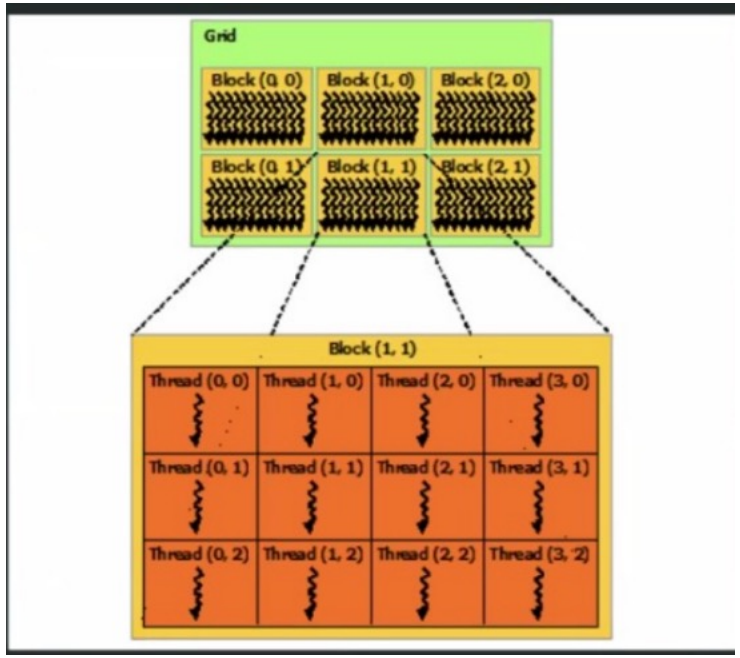
Source: https://www.olcf.ornl.gov/wp-content/uploads/2019/12/03-CUDA-Fundamental-Optimization-Part-1.pdf

CUDA terminology

* A kernel is the operation you want to run on your data which will be shipped to the GPU

* Kernels are launched in grid of blocks

* Grids and blocks can be 1D, 2D or 3D

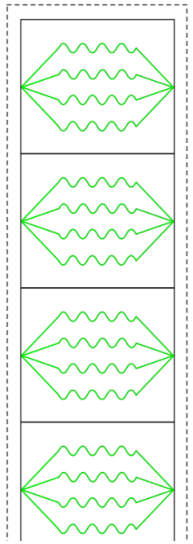


Source: Jayce Dowell Presentation on GPU University of New Mexico, Fall 2021

Illustration

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
                    4
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < N) c[id] = a[id] + b[id];
}
```



4

`blockDim`

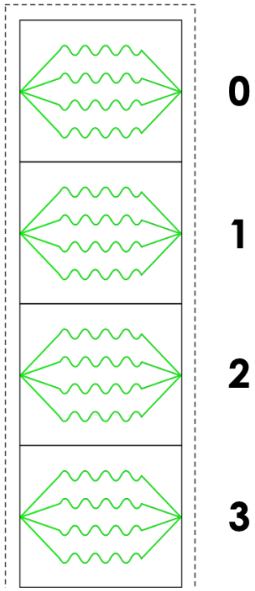
Gives the number of threads within each block (x-dimension for 1D).

- E.g., 4 threads per block

Illustration

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
                    4           0-3
  int id = blockDim.x * blockIdx.x + threadIdx.x;

  if (id < N) c[id] = a[id] + b[id];
}
```



`blockIdx`

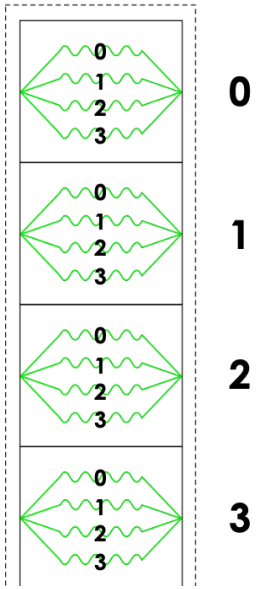
Specifies the block index of the thread (within the grid of blocks).

- I.e., which block the thread is in

Illustration

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)

{
                    4           0-3           0-3
  int id = blockDim.x * blockIdx.x + threadIdx.x;

  if (id < N) c[id] = a[id] + b[id];

}
```



`threadIdx`

Specifies a thread's local ID within a thread block.

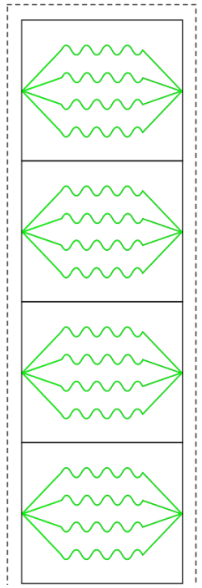
Illustration

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < N) c[id] = a[id] + b[id];
}
```

```
int id = blockDim.x * blockIdx.x + threadIdx.x;
```

This defines a unique thread ID among all threads in a grid.

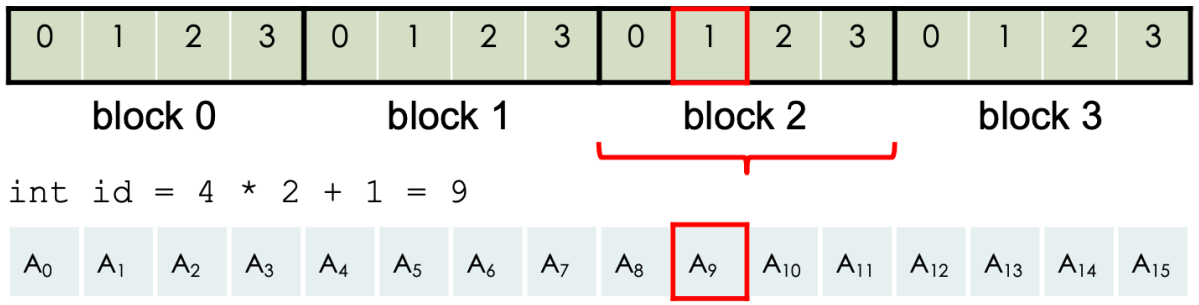


Illustration

GPU kernel

```
__global__ void vector_addition(double *a, double *b, double *c)

{
                    4              2              1
   int id = blockDim.x * blockIdx.x + threadIdx.x;

   if (id < N) c[id] = a[id] + b[id];

}
```

For example, with `blockIdx.x = 2` and `threadIdx.x = 1`...



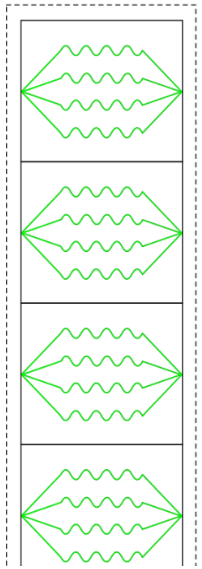
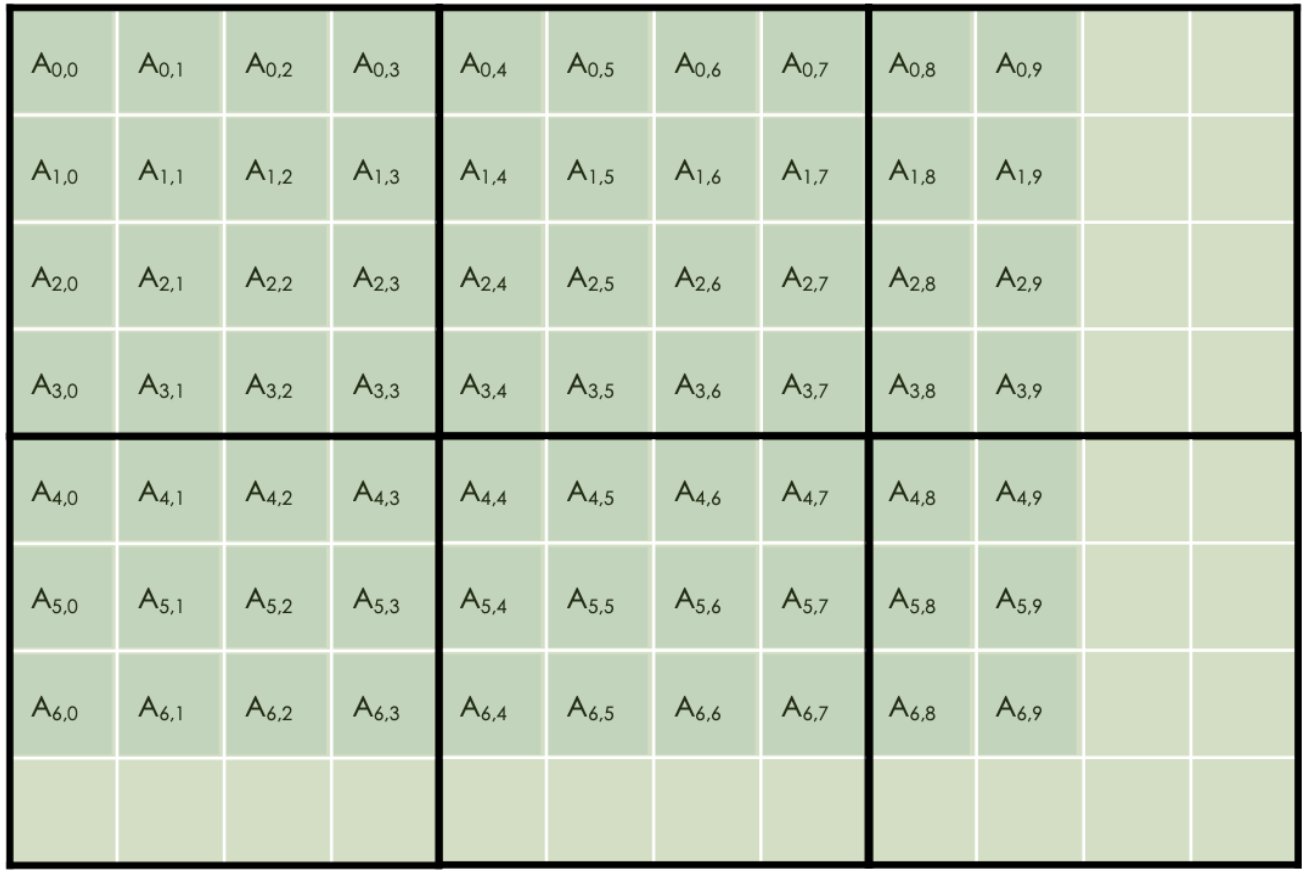


Illustration (Matrix addition)

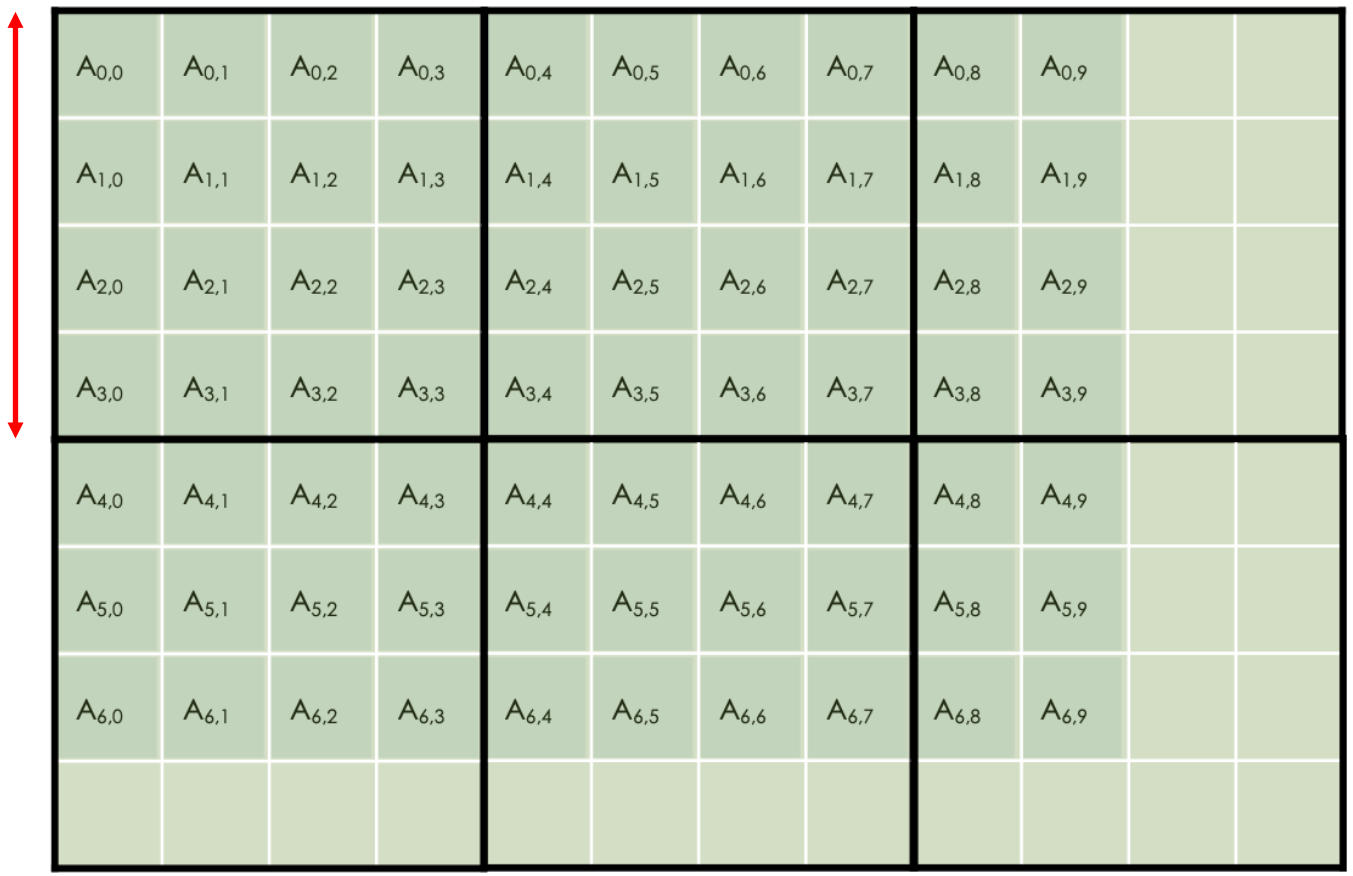


Total size of Matrix is:

M = 7 rows

N = 10 columns

Illustration (Matrix addition)



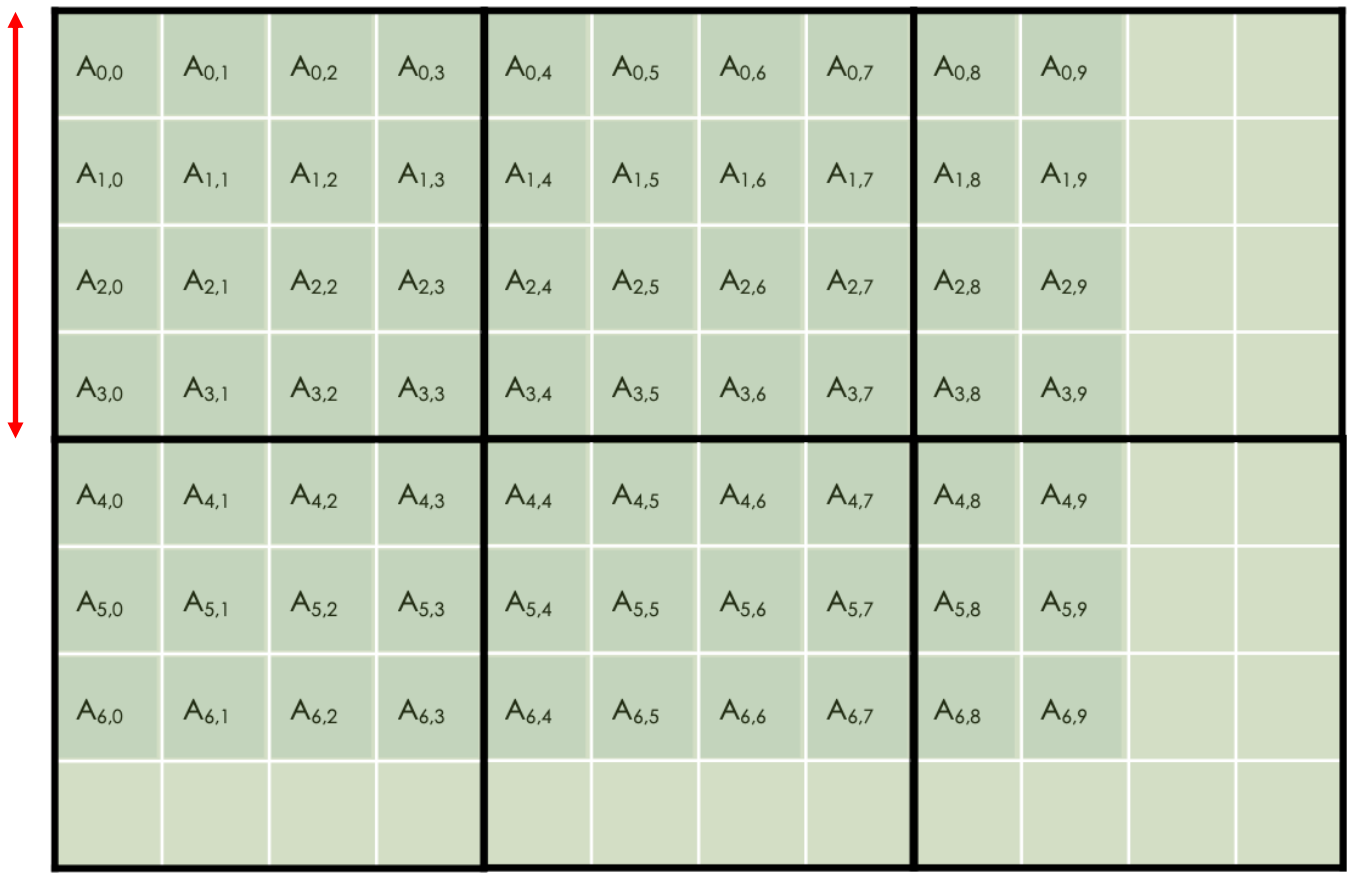
Total size of Matrix is:

M = 7 rows

N = 10 columns

But we want 4X4 block of thread !!

Illustration (Matrix addition)



Total size of Matrix is:

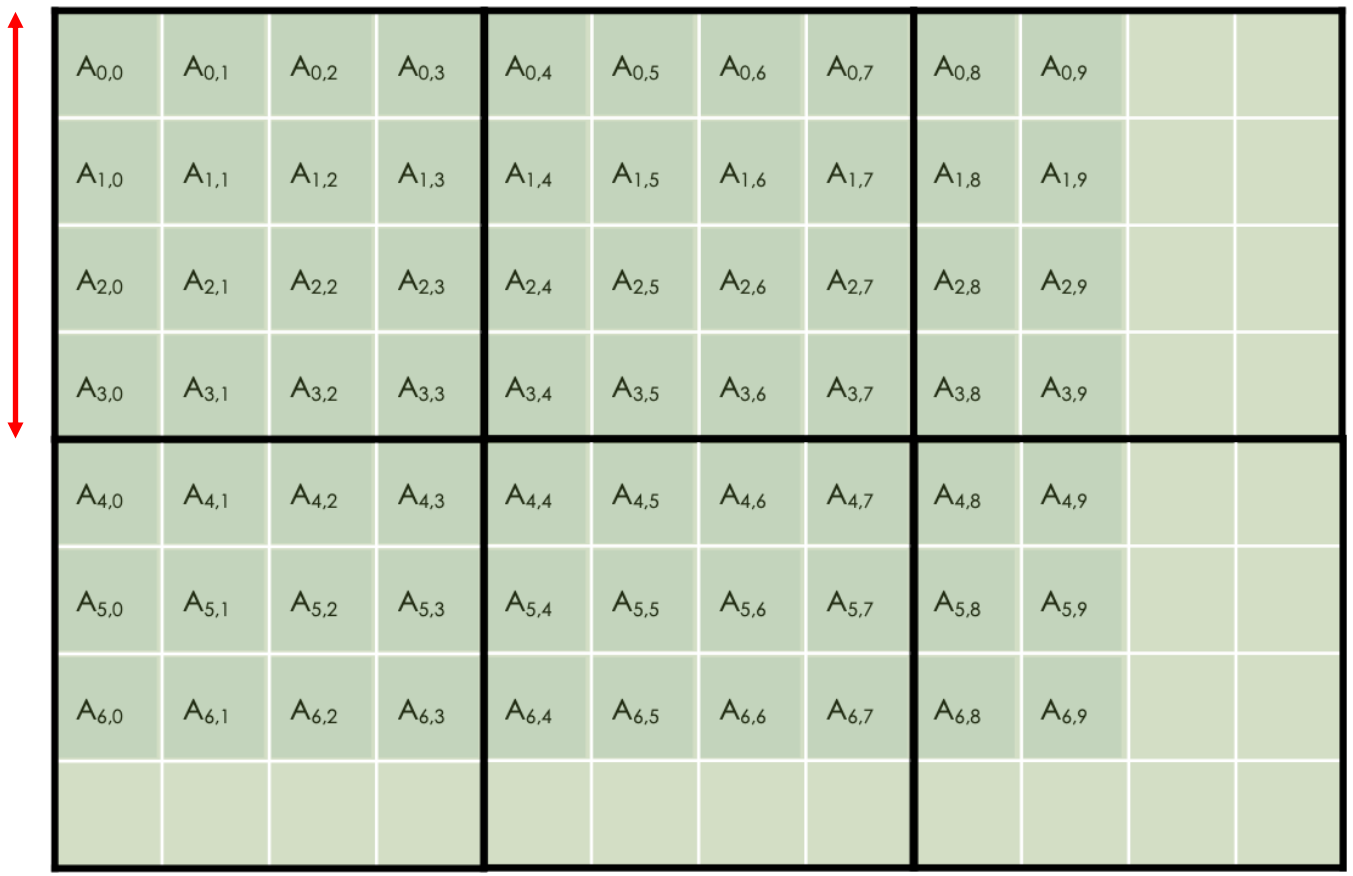
M = 7 rows

N = 10 columns

But we want 4X4 block of thread !!

```
dim3 threads_per_block( 4, 4, 1 );
```

Illustration (Matrix addition)



Total size of Matrix is:

M = 7 rows

N = 10 columns

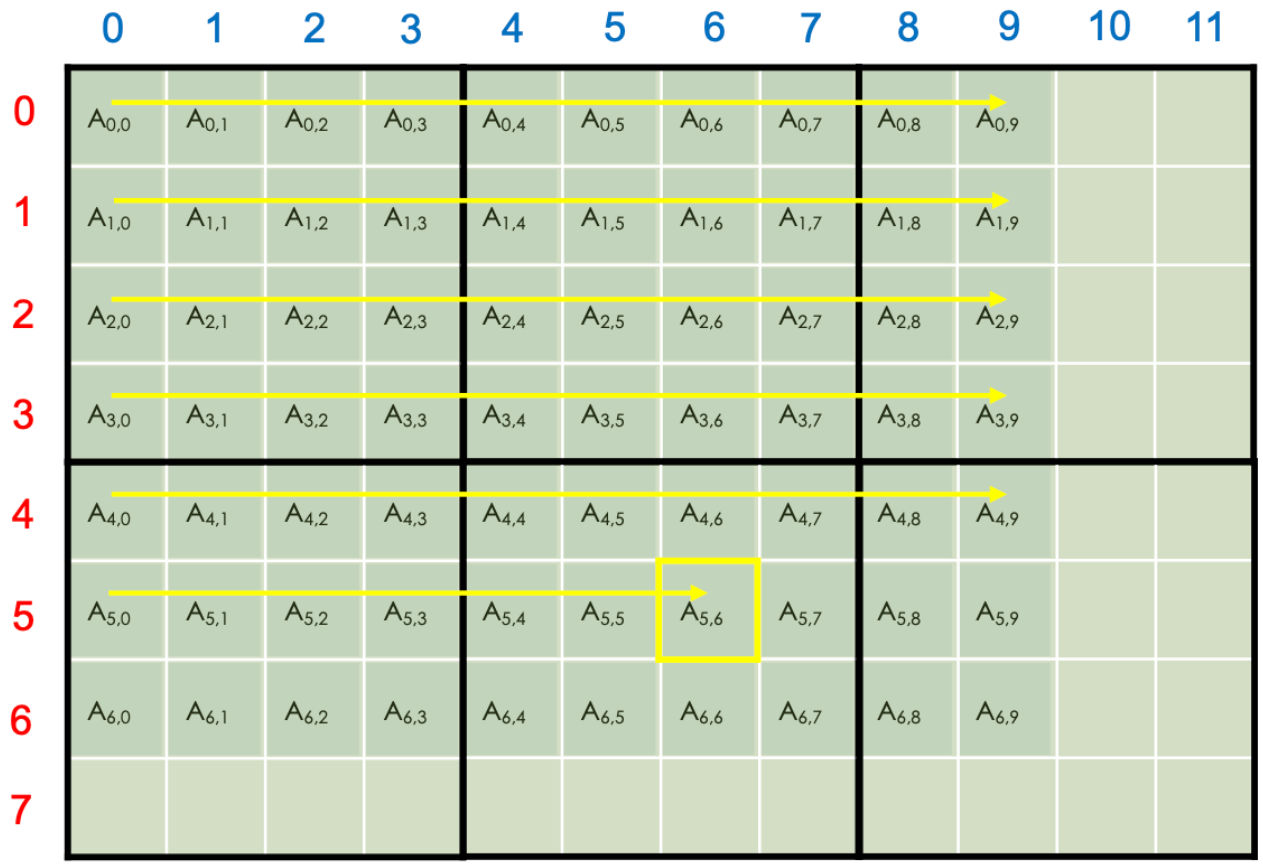
But we want 4X4 block of thread size !!

This is where we design the number of blocks in grid.
3 blocks in x-dim and 2 blocks in y-dim

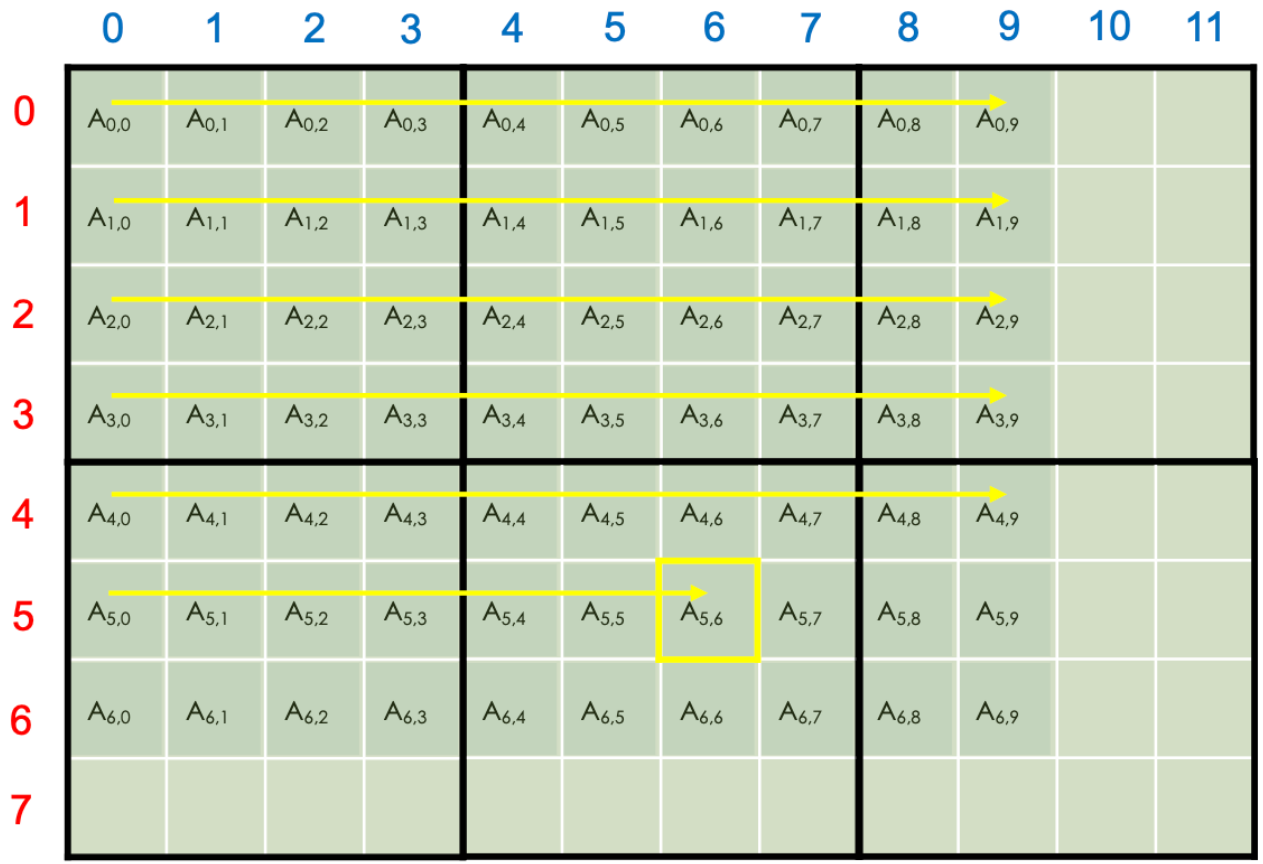
```
dim3 threads_per_block( 4, 4, 1 );

dim3 blocks_in_grid( ceil( float(N) / threads_per_block.x ),
                     ceil( float(M) / threads_per_block.y ) , 1 );
```

ceil(10/4) = ceil(2.5) = 3
ceil(7/4) = ceil(1.75) = 2

Let’s find the threadID



Let's find the threadID




```
__global__ void matrix_addition(double *a, double *b, double *c){
        int column = blockDim.x * blockIdx.x + threadIdx.x;
        int row    = blockDim.y * blockIdx.y + threadIdx.y;
        if (row < M && column < N){
            int thread_id = row * N + column;
             c[thread_id] = a[thread_id] + b[thread_id];
        }
```

(0 – 11)

(0 – 7)

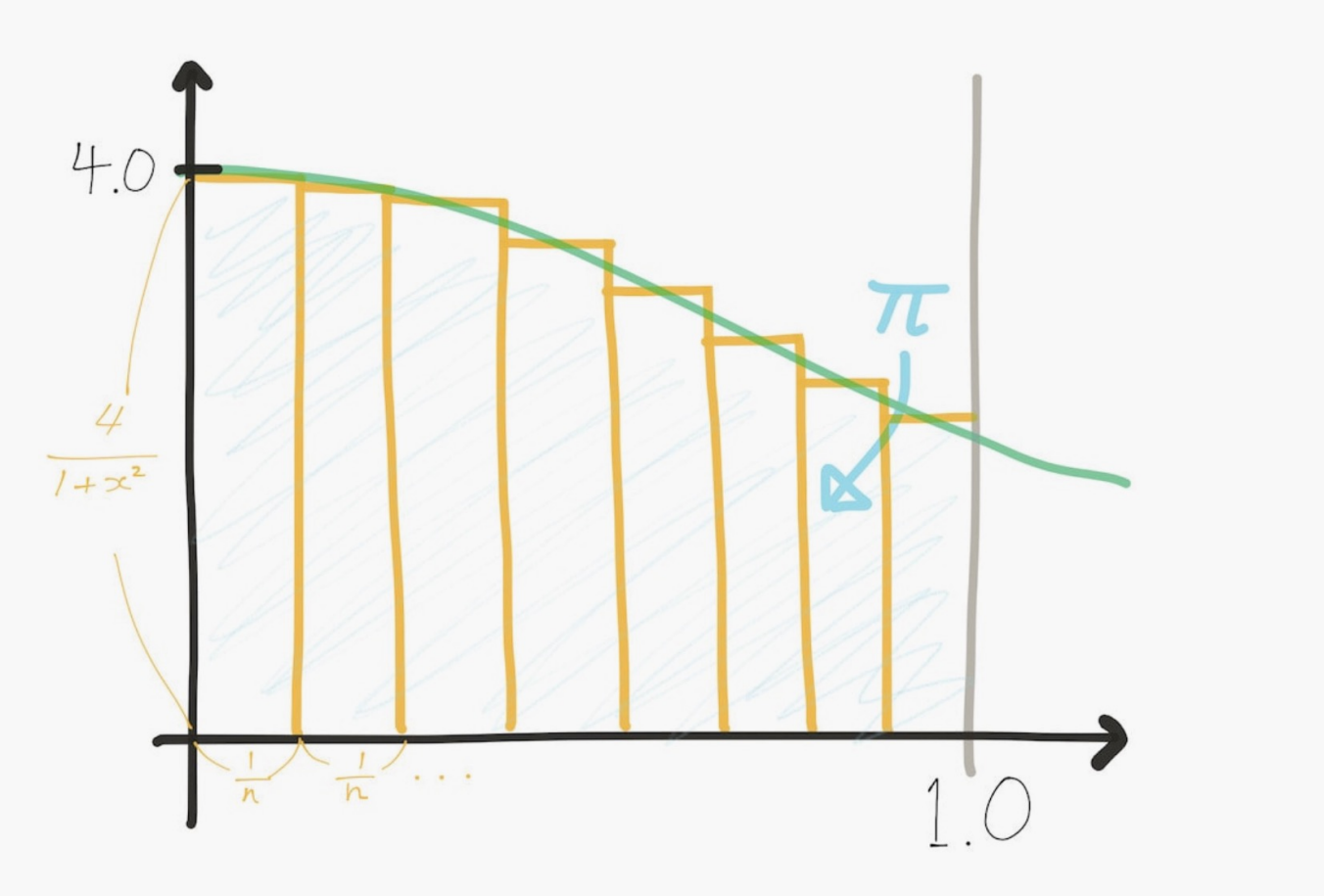
(0 – 69)

Ex: What element of the array does the highlighted thread correspond to?

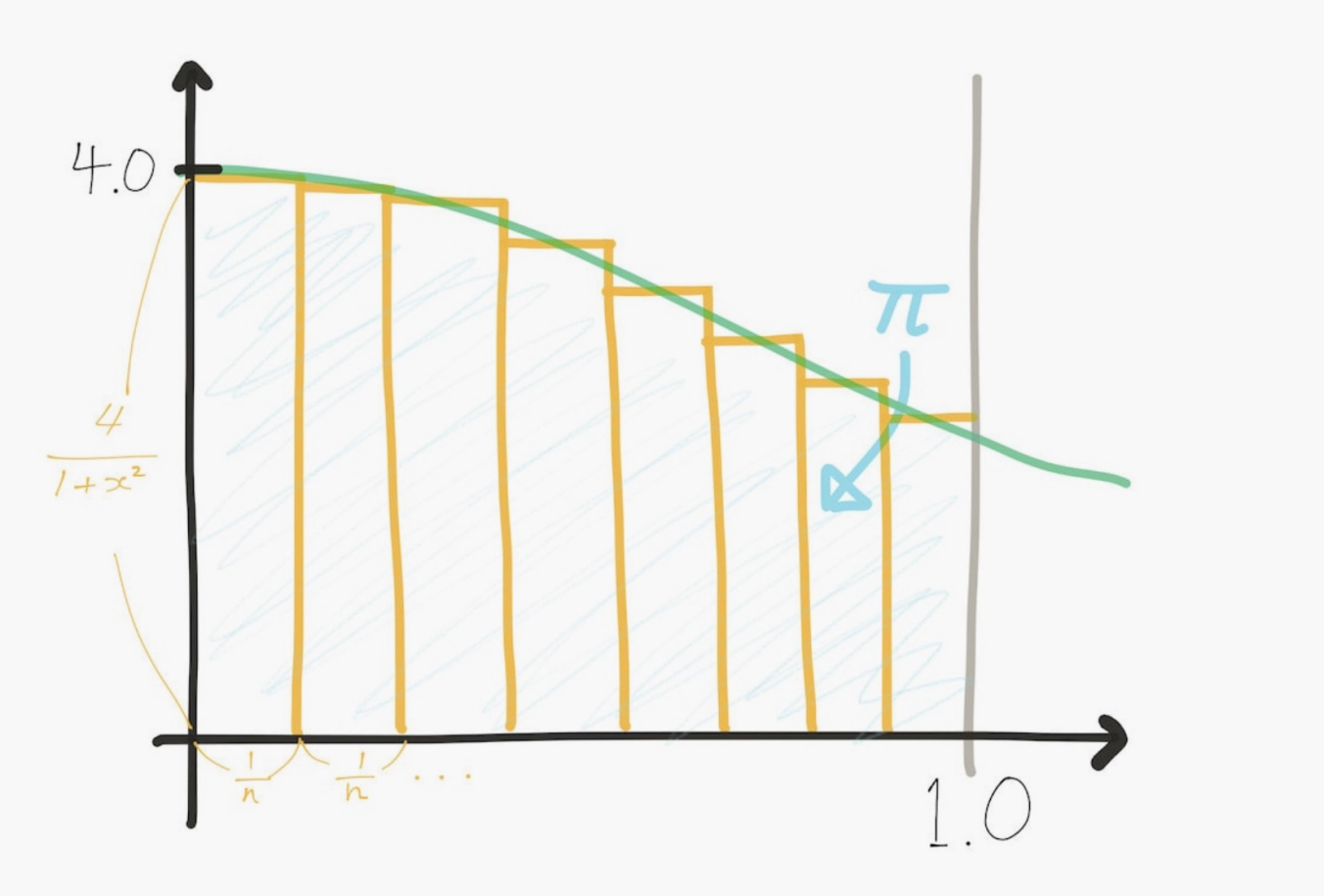
```
thread_id = row * N  + column
          = 5   * 10 + 6      = 56
```

II – The PI example

How to estimate PI?

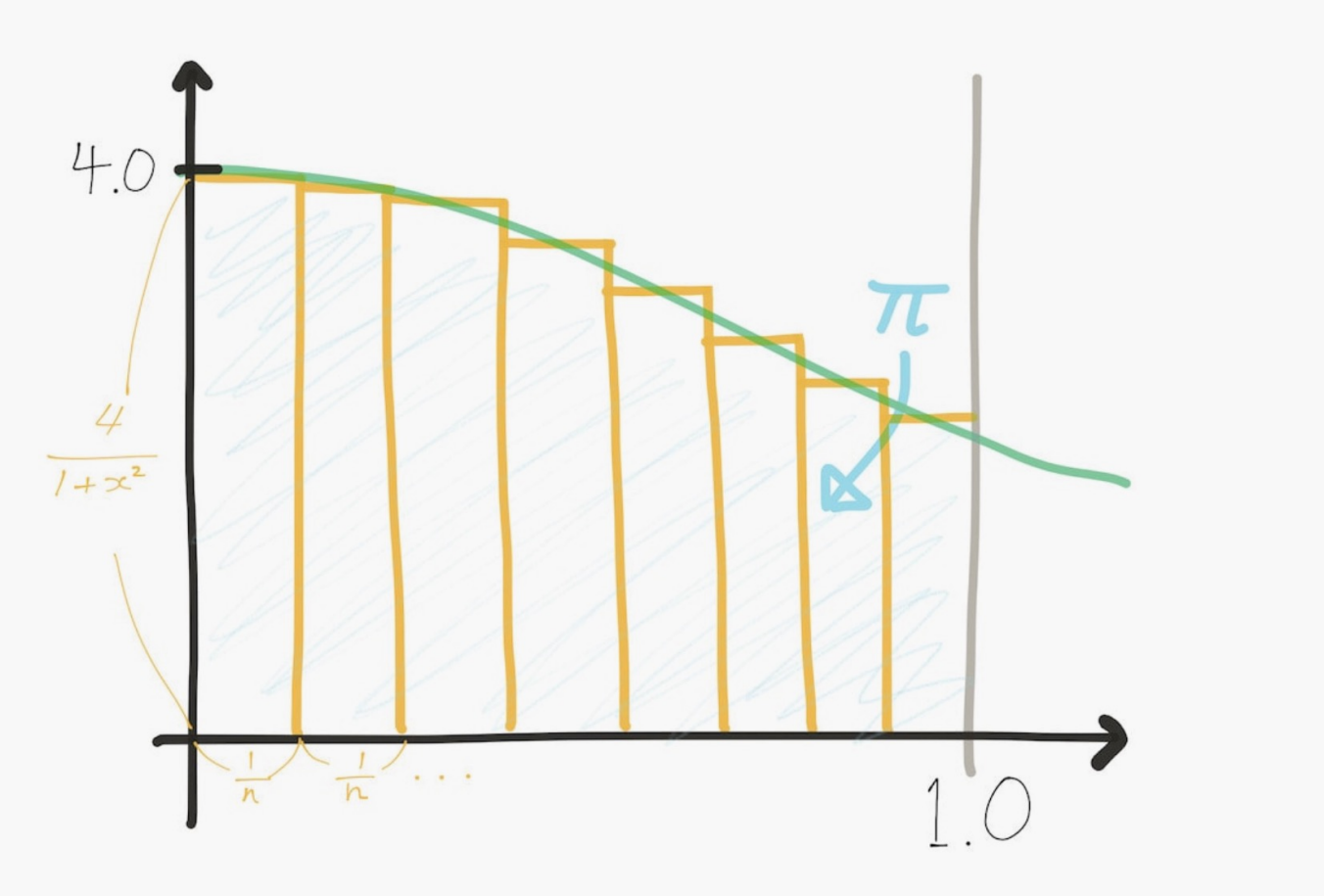


How to estimate PI?



- Estimate Pi as a Riemann sum of the rectangles under the curve.

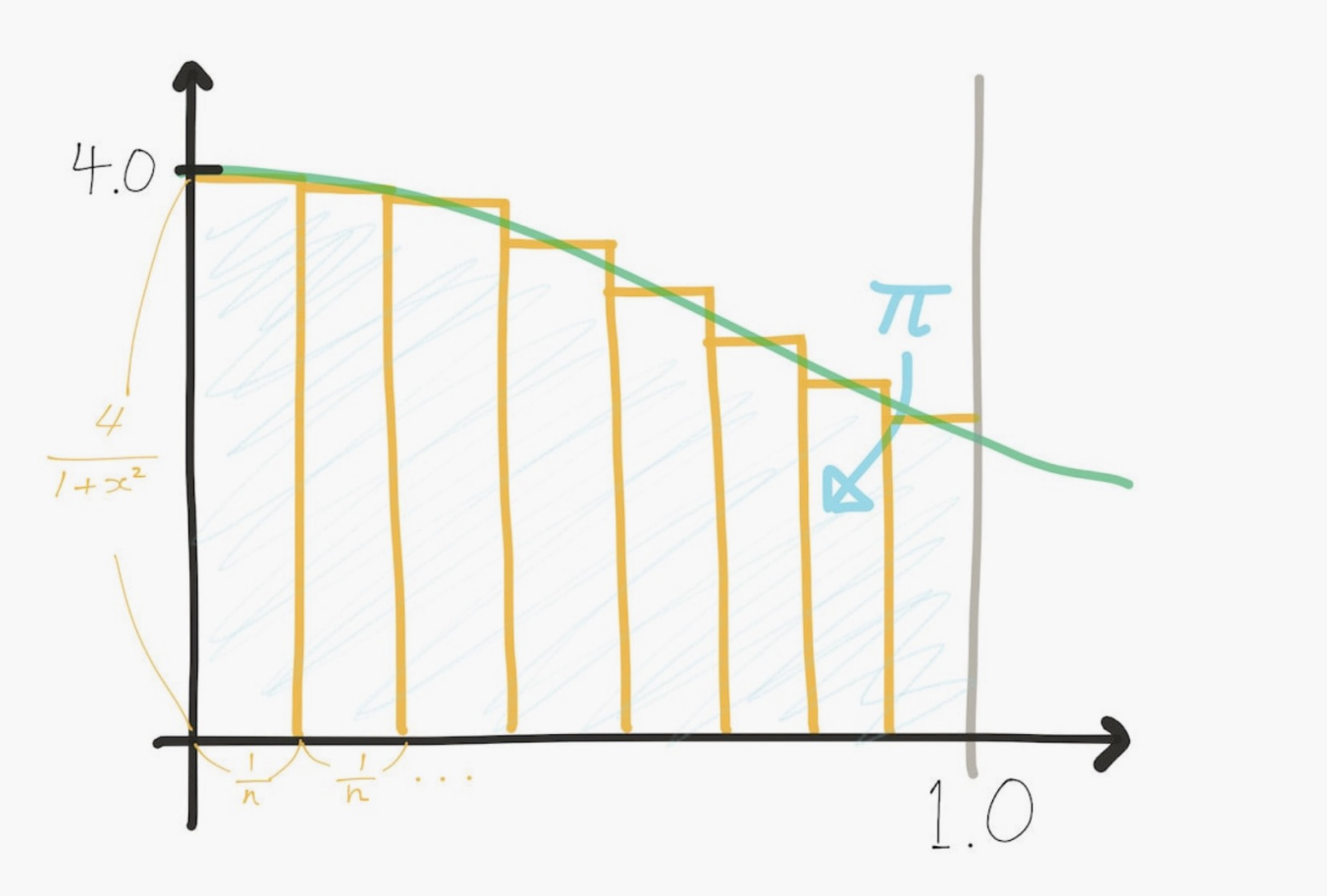
How to estimate PI?



- Estimate Pi as a Riemann sum of the rectangles under the curve.

$$\int_0^1 \frac{4}{1+x^2}\mathrm{d}x = 4\tan^{-1}(x) = \pi$$

How to estimate PI?



- Estimate Pi as a Riemann sum of the rectangles under the curve.

$$\int_0^1 \frac{4}{1+x^2}\mathrm{d}x = 4\tan^{-1}(x) = \pi$$

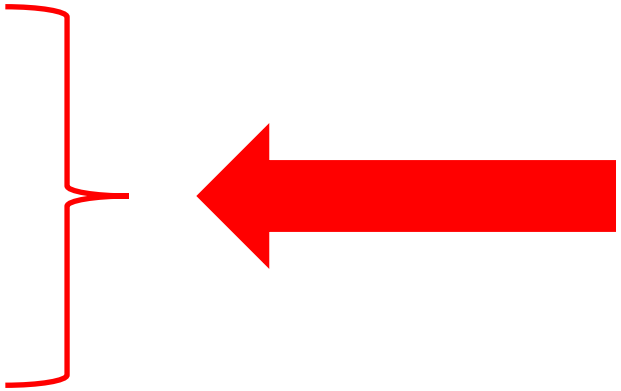
$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{1}{n} \frac{4}{1+x_i^2}$$

where x_i is the midpoint of

$$x_i = \frac{1}{n} \cdot i + \frac{1}{2n} = \frac{2i+1}{2n},$$

Computation of pi on cpu

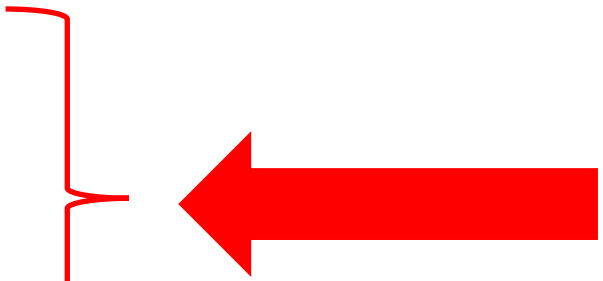
```python
import time
import sys
import numpy as np


# Here we calculate pi by assessing the area under the curve
# Then we check the value against what is calculated in numpy
# source: https://github.com/UNM-CARC/QuickBytes/blob/master/workshop_slides/CS_Math_471.pdf

def Pi(num_steps):
    step = 1.0 / num_steps
    sum = 0

    for i in range(num_steps):
        x = (i+0.5) * step
        sum = sum + 4.0 / (1.0 + x * x)

    pi = step * sum


    return pi

if len(sys.argv)!=2:
    print("Usage: ", sys.argv[0], "<number of steps>")
    sys.exit(1)

num_steps = int(sys.argv[1],10)

# Call the function to calculate pi
start = time.time() #Start
pi = Pi(num_steps)
end = time.time() # End

# Print estimation, the difference from the numpy val and how long it took.
print("Pi = %.20f, (Diff=%.20f) (calculated in %f secs with %d steps)" %(pi, pi-np.pi, end-start, num_steps))

sys.exit(0)
```



Riemann sum of each rectangles (steps)

Computation of pi on cpu

```python
import time
import sys
import numpy as np


# Here we calculate pi by assessing the area under the curve
# Then we check the value against what is calculated in numpy
# source: https://github.com/UNM-CARC/QuickBytes/blob/master/workshop_slides/CS_Math_471.pdf

def Pi(num_steps):
    step = 1.0 / num_steps
    sum = 0

    for i in range(num_steps):
        x = (i+0.5) * step
        sum = sum + 4.0 / (1.0 + x * x)

    pi = step * sum


    return pi

if len(sys.argv)!=2:
    print("Usage: ", sys.argv[0], "<number of steps>")
    sys.exit(1)

num_steps = int(sys.argv[1],10)

# Call the function to calculate pi
start = time.time() #Start
pi = Pi(num_steps)
end = time.time() # End

# Print estimation, the difference from the numpy val and how long it took.
print("Pi = %.20f, (Diff=%.20f) (calculated in %f secs with %d steps)" %(pi, pi-np.pi, end-start, num_steps))

sys.exit(0)
```



Compute Pi

Computation of pi on cpu

```
import time
import sys
import numpy as np


# Here we calculate pi by assessing the area under the curve
# Then we check the value against what is calculated in numpy
# source: https://github.com/UNM-CARC/QuickBytes/blob/master/workshop_slides/CS_Math_471.pdf

def Pi(num_steps):
    step = 1.0 / num_steps
    sum = 0

    for i in range(num_steps):
        x = (i+0.5) * step
        sum = sum + 4.0 / (1.0 + x * x)

    pi = step * sum


    return pi

if len(sys.argv)!=2:
    print("Usage: ", sys.argv[0], "<number of steps>")
    sys.exit(1)

num_steps = int(sys.argv[1],10)

# Call the function to calculate pi
start = time.time() #Start
pi = Pi(num_steps)
end = time.time() # End

# Print estimation, the difference from the numpy val and how long it took.
print("Pi = %.20f, (Diff=%.20f) (calculated in %f secs with %d steps)" %(pi, pi-np.pi, end-start, num_steps))

sys.exit(0)
```

Compare Pi with the actual value.

III– The PI example on GPU

Computation on cpu

```
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ python calculate_pi.py 1000
Pi = 3.14159273692312268622, (Diff=0.00000008333332957022) (calculated in 0.000104 secs with 1000 steps)
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$
```

A few thousands steps perform fairly well on the CPU

Computation of pi on the GPU

```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):

    tx =  cuda.threadIdx.x # Unique thread ID within 1 block
    ty = cuda.blockIdx.x # Unique block ID
    block_size = cuda.blockDim.x # Number of threads per block
    grid_size = cuda.gridDim.x # Size of the grid

    # We define the grid size:
    startX = cuda.grid(1) # Starting point on the Grid
    gridX = cuda.gridDim.x * cuda.blockDim.x # stride in x
    stride = cuda.gridsize(1)

    # Or we could have written the following
    #startX = tx + ty * block_size
    #stride = block_size * grid_size

    for i in range(startX,  step_vec.shape[0], stride):
        step_vec[i] = (step_vec[i]+0.5) * step_size[0]
        sum_vec[i] +=  4.0 / (1.0 + step_vec[i] * step_vec[i])
```

Getting the threadId within 1 block, the blockId, the block size and the grid size

Computation of pi on the GPU

```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):

    tx =  cuda.threadIdx.x # Unique thread ID within 1 block
    ty = cuda.blockIdx.x # Unique block ID
    block_size = cuda.blockDim.x # Number of threads per block
    grid_size = cuda.gridDim.x # Size of the grid

    # We define the grid size:
    startX = cuda.grid(1) # Starting point on the Grid
    gridX = cuda.gridDim.x * cuda.blockDim.x # stride in x
    stride = cuda.gridsize(1)

    # Or we could have written the following
    #startX = tx + ty * block_size
    #stride = block_size * grid_size

    for i in range(startX,  step_vec.shape[0], stride):
        step_vec[i] = (step_vec[i]+0.5) * step_size[0]
        sum_vec[i] +=  4.0 / (1.0 + step_vec[i] * step_vec[i])
```

We get the global threadID with startX.
By using a loop with stride we ensure that the grid size is independent of the input size.

Computation of pi on the GPU

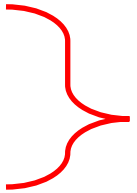
```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):

    tx =  cuda.threadIdx.x # Unique thread ID within 1 block
    ty = cuda.blockIdx.x # Unique block ID
    block_size = cuda.blockDim.x # Number of threads per block
    grid_size = cuda.gridDim.x # Size of the grid

    # We define the grid size:
    startX = cuda.grid(1) # Starting point on the Grid
    gridX = cuda.gridDim.x * cuda.blockDim.x # stride in x
    stride = cuda.gridsize(1)

    # Or we could have written the following
    #startX = tx + ty * block_size
    #stride = block_size * grid_size

    for i in range(startX,  step_vec.shape[0], stride):
        step_vec[i] = (step_vec[i]+0.5) * step_size[0]
        sum_vec[i] +=  4.0 / (1.0 + step_vec[i] * step_vec[i])
```

Also, striding will ensure that threads with consecutive indices. are accessing consecutive memory locations as much as possible.

Source: https://github.com/ContinuumIO/gtc2017-numba/blob/master/4%20-%20Writing%20CUDA%20Kernels.ipynb

Computation of pi on the GPU

```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):

    tx =  cuda.threadIdx.x # Unique thread ID within 1 block
    ty = cuda.blockIdx.x # Unique block ID
    block_size = cuda.blockDim.x # Number of threads per block
    grid_size = cuda.gridDim.x # Size of the grid

    # We define the grid size:
    startX = cuda.grid(1) # Starting point on the Grid
    gridX = cuda.gridDim.x * cuda.blockDim.x # stride in x
    stride = cuda.gridsize(1)

    # Or we could have written the following
    #startX = tx + ty * block_size
    #stride = block_size * grid_size

    for i in range(startX,  step_vec.shape[0], stride):
        step_vec[i] = (step_vec[i]+0.5) * step_size[0]
        sum_vec[i] +=  4.0 / (1.0 + step_vec[i] * step_vec[i])
```



Note that we could have calculated startX and stride this way.

Computation of pi on the GPU

```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):

    tx =  cuda.threadIdx.x # Unique thread ID within 1 block
    ty = cuda.blockIdx.x # Unique block ID
    block_size = cuda.blockDim.x # Number of threads per block
    grid_size = cuda.gridDim.x # Size of the grid

    # We define the grid size:
    startX = cuda.grid(1) # Starting point on the Grid
    gridX = cuda.gridDim.x * cuda.blockDim.x # stride in x
    stride = cuda.gridsize(1)

    # Or we could have written the following
    #startX = tx + ty * block_size
    #stride = block_size * grid_size

    for i in range(startX,  step_vec.shape[0], stride):
        step_vec[i] = (step_vec[i]+0.5) * step_size[0]
        sum_vec[i] +=  4.0 / (1.0 + step_vec[i] * step_vec[i])
```

Sum vector which contains the elements of the Riemann sum.

Computation of pi on the GPU

```
step_vec = np.arange(int(sys.argv[1],10), dtype=np.float32)
sum_vec = np.zeros(int(sys.argv[1],10), dtype=np.float32)
pi = np.empty(1).astype(np.float32)
N = int(sys.argv[1],10)
step_size = np.ones(1, dtype=np.float32) / N
```

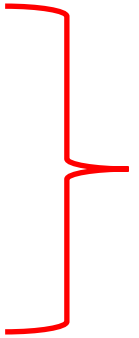
We create arrays that will be used during the computation

```
# We convert the variables to device
step_device = cuda.to_device(step_vec)
sum_device = cuda.to_device(sum_vec)
pi_device = cuda.to_device(pi)
step_size_device = cuda.to_device(step_size)
```

Computation of pi on the GPU

```
step_vec = np.arange(int(sys.argv[1],10), dtype=np.float32)
sum_vec = np.zeros(int(sys.argv[1],10), dtype=np.float32)
pi = np.empty(1).astype(np.float32)
N = int(sys.argv[1],10)
step_size = np.ones(1, dtype=np.float32) / N
```

```
# We convert the variables to device
step_device = cuda.to_device(step_vec)
sum_device = cuda.to_device(sum_vec)
pi_device = cuda.to_device(pi)
step_size_device = cuda.to_device(step_size)
```



We send those arrays to the GPU

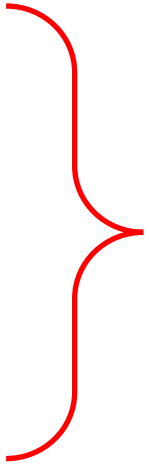
Computation of pi on the GPU

```
# We decide to use 256 blocks, each containing 128 threads
blocks_per_grid = 256
# With grid size of 32 I get
#"Warning: Grid size 32 will likely result in
#GPU under-utilization due to low occupancy."
thread_per_block = 128  # was 128

# Call the function to calculate pi
start = time.time() #Start
Pi[blocks_per_grid, thread_per_block](step_device, sum_device, step_size_device)

# I copy the devices to host and calculate the final pi version
sum_vec = sum_device.copy_to_host()
pi = step_size * sum_vec.sum()
end = time.time() # End
```

Number of blocks is a multiple of 32. We do not want to use a low number of blocks as the GPU will be underutilized.

Computation of pi on the GPU

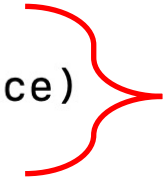
```
# We decide to use 256 blocks, each containing 128 threads
blocks_per_grid = 256
# With grid size of 32 I get
#"Warning: Grid size 32 will likely result in
#GPU under-utilization due to low occupancy."
thread_per_block = 128  # was 128

# Call the function to calculate pi
start = time.time() #Start
Pi[blocks_per_grid, thread_per_block](step_device, sum_device, step_size_device)

# I copy the devices to host and calculate the final pi version
sum_vec = sum_device.copy_to_host()
pi = step_size * sum_vec.sum()
end = time.time() # End
```



Number of threads per block can be higher than the number of cores per SM (64)

Computation of pi on the GPU

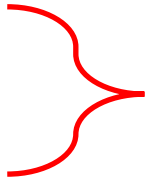
```
# We decide to use 256 blocks, each containing 128 threads
blocks_per_grid = 256
# With grid size of 32 I get
#"Warning: Grid size 32 will likely result in
#GPU under-utilization due to low occupancy."
thread_per_block = 128  # was 128

# Call the function to calculate pi
start = time.time() #Start
Pi[blocks_per_grid, thread_per_block](step_device, sum_device, step_size_device)

# I copy the devices to host and calculate the final pi version
sum_vec = sum_device.copy_to_host()
pi = step_size * sum_vec.sum()
end = time.time() # End
```



Calculate the sum vector

Computation of pi on the GPU

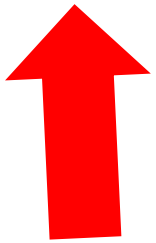
```
# We decide to use 256 blocks, each containing 128 threads
blocks_per_grid = 256
# With grid size of 32 I get
#"Warning: Grid size 32 will likely result in
#GPU under-utilization due to low occupancy."
thread_per_block = 128  # was 128

# Call the function to calculate pi
start = time.time() #Start
Pi[blocks_per_grid, thread_per_block](step_device, sum_device, step_size_device)

# I copy the devices to host and calculate the final pi version
sum_vec = sum_device.copy_to_host()
pi = step_size * sum_vec.sum()
end = time.time() # End
```



Sending back to the CPU, reduce and then calculate pi.

CPU vs GPU calculation comparison

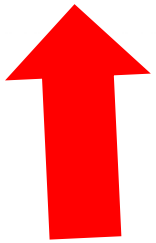
```
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ python calculate_pi.py 1000
Pi = 3.1415927369231226 8622, (Diff=0.00000008333332957022) (calculated in 0.000107 secs with 1000 steps)
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ python calculate_pi_gpu_opt_clean.py 1000
Pi = 3.1415929794311523 4375, (Diff=0.00000023841857910156) (calculated in 0.244134 secs with 1000 steps)
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$
```



The GPU is much slower than the CPU when using 1000 steps

CPU vs GPU calculation comparison

```
[kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ ml anaconda;conda activate numba_env
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ python  calculate_pi_gpu_opt_clean.py 2000000000
Pi = 3.14155648000000020659, (Diff=-0.000036173589792909 41) (calculated in 2.395604 secs with 2000000000 steps)
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ python  calculate_pi.py 2000000000
Pi = 3.14159265358983885719, (Diff=0.00000000000004574119) (calculated in 227.818585 secs with 2000000000 steps)
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$
```



The GPU is much faster than the CPU when using 2 billion steps!!!
Note that the discrepancy between the 2 pi results is due the GPU script not being on float64 precision.

CPU vs GPU calculation comparison

```
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ python calculate_pi.py 10000000
<class 'float'>
Pi = 3.14159265358973094351, (Diff=-0.00000000000006217249) (calculated in 1.015425 secs with 10000000 steps)
(numba_env) [kfotso@xsede.org@c3gpu-a9-u31-1 calculate_pi]$ python calculate_pi_gpu_opt_clean_float64.py 10000000
Pi = 3.14159265358979045146, (Diff=-0.00000000000000266454) (calculated in 0.284768 secs with 10000000 steps)
```

This is the pi outcome with both scripts being on float32 precision and 10000000 steps

IV- Intro to GPU profiling

3 steps

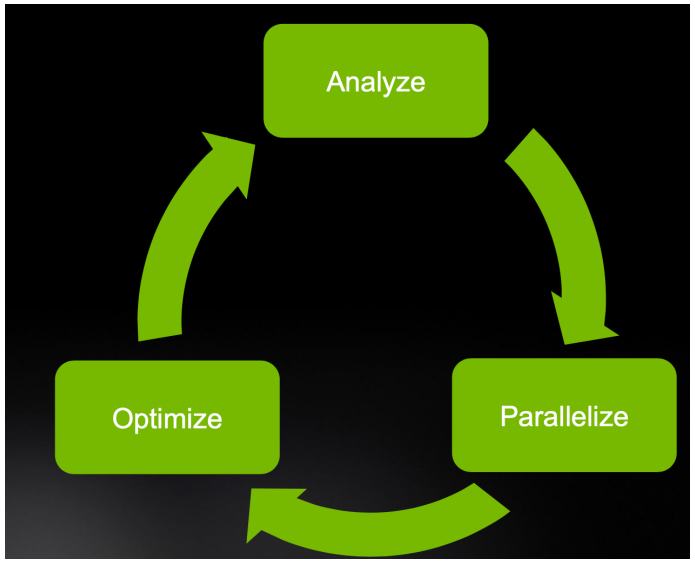
- Analyze code: for assessing the blocks that need optimization

3 steps

- Analyze code: for assessing the blocks that need optimization
- Then parallelize the further by rewriting the parts that take the most time

3 steps

- Analyze code: for assessing the blocks that need optimization
- Then parallelize the further by rewriting the parts that take the most time
- Optimize the code
- Repeat



Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

Benefits of nsights

- Allows to measure the bottlenecks of the pipeline to improve efficiency
- Allows to avoid work based on intuition or guess (is the pipeline CPU or GPU bound?)
- See how asynchronous the software is interacting.

Source: https://alcf.anl.gov/sites/default/files/2022-07/Nsight%20Systems%20-

2 things necessary to profile

- You need to have nsys on the hpc
  module load cuda/11.8

- When running your GPU pipeline run it with nsys and it will create an output

- Export that output profile to your local laptop where you downloaded the nsight app.

- You may profile for a 30s to 2 min to get an idea of the bottleneck.
  You do not need to profile for the entire time if you have a heavy run

nsys command

```
nsys profile --trace=cuda,nvtx,osrt,cudnn,cublas --backtrace=dwarf
--capture-range=cudaProfilerApi --gpu-metrics-devices=all -output=oft-profile-dwarf4
sh scripts/expt-singleGPU.sh --profile 50 --profile_start 5000 --profile_epoch 1
```

Nsight Systems CLI command
Select APIs to trace
Enable GPU memory use tracking (but there is extra overhead)
Collect thread call-stack sample backtraces via DWARF info - deeper but more expensive to collect
Trigger collection on cudaProfilerStart API in application, or consider timer-based options
GPU metrics sampling at default 10khz
Name the report file
Application command - plus arguments for when to start profiling

https://docs.nvidia.com/nsight-systems/UserGuide/index.html#cli-profile-command-switch-options

srun nsys profile … required on multi-node or multi-container
nsys profile mpirun … optional on single node to produce a single report

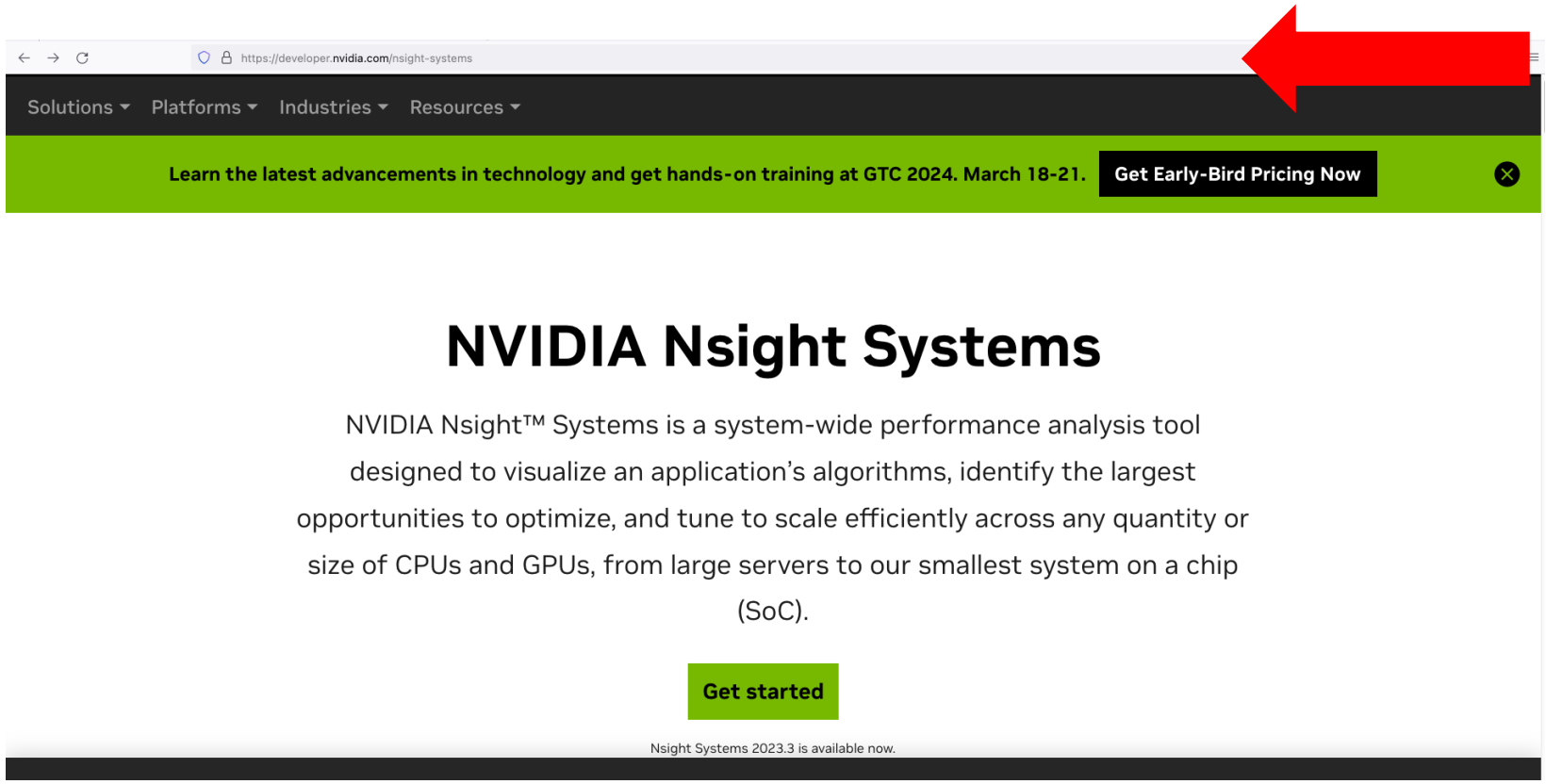
Nsys with Deep learning Horovod on Alpine (Slurm script)

```
# We test it with 4 GPUs
echo "Testing it with 4 GPUs"
nsys profile --trace=cuda,nvtx,osrt,cudnn,cublas \
        --capture-range=cudaProfilerApi \
        --gpu-metrics-device=all \
        --output=horovod_profile \
        horovodrun -np 4 -H $node1:2,$node2:2
        horovod_housing_tf.py
```

We use 2 GPUs on node 1 and 2 on node 2

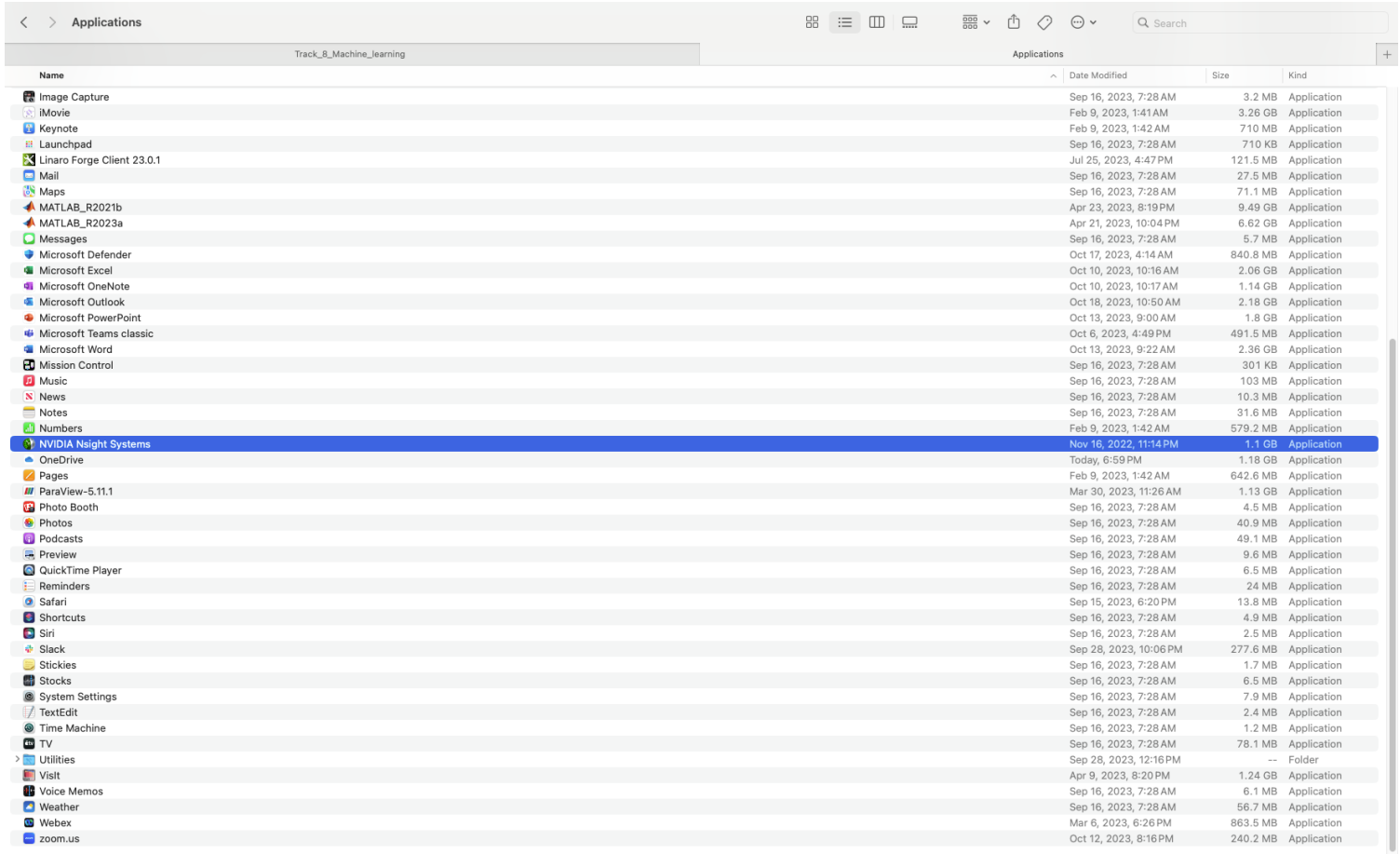
NVIDIA Nsight

- Download the NVIDIA Nsight system client tool to your local computer



https://developer.nvidia.com/nsight-systems

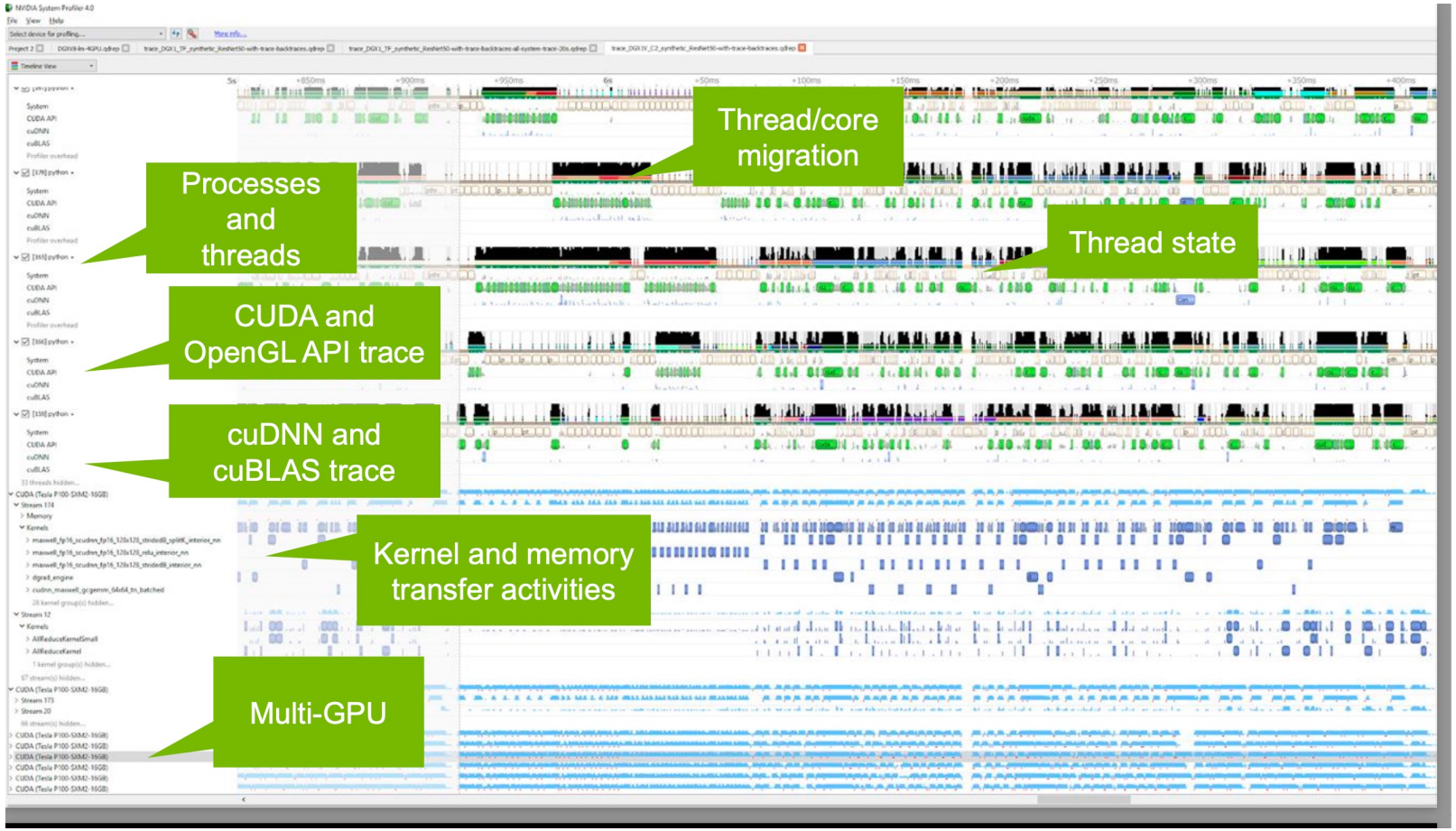
NVIDIA Nsight (2)



NVIDIA Nsight (3)

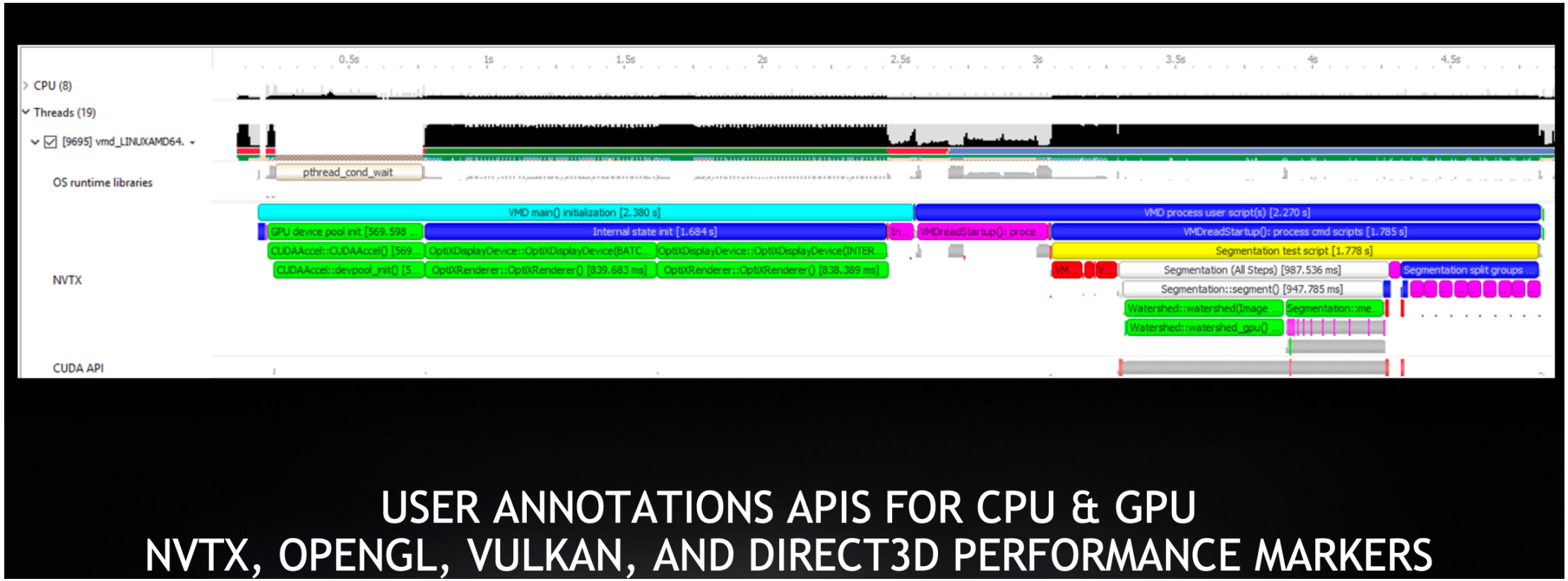
Click on open

What can be visualize with nsys



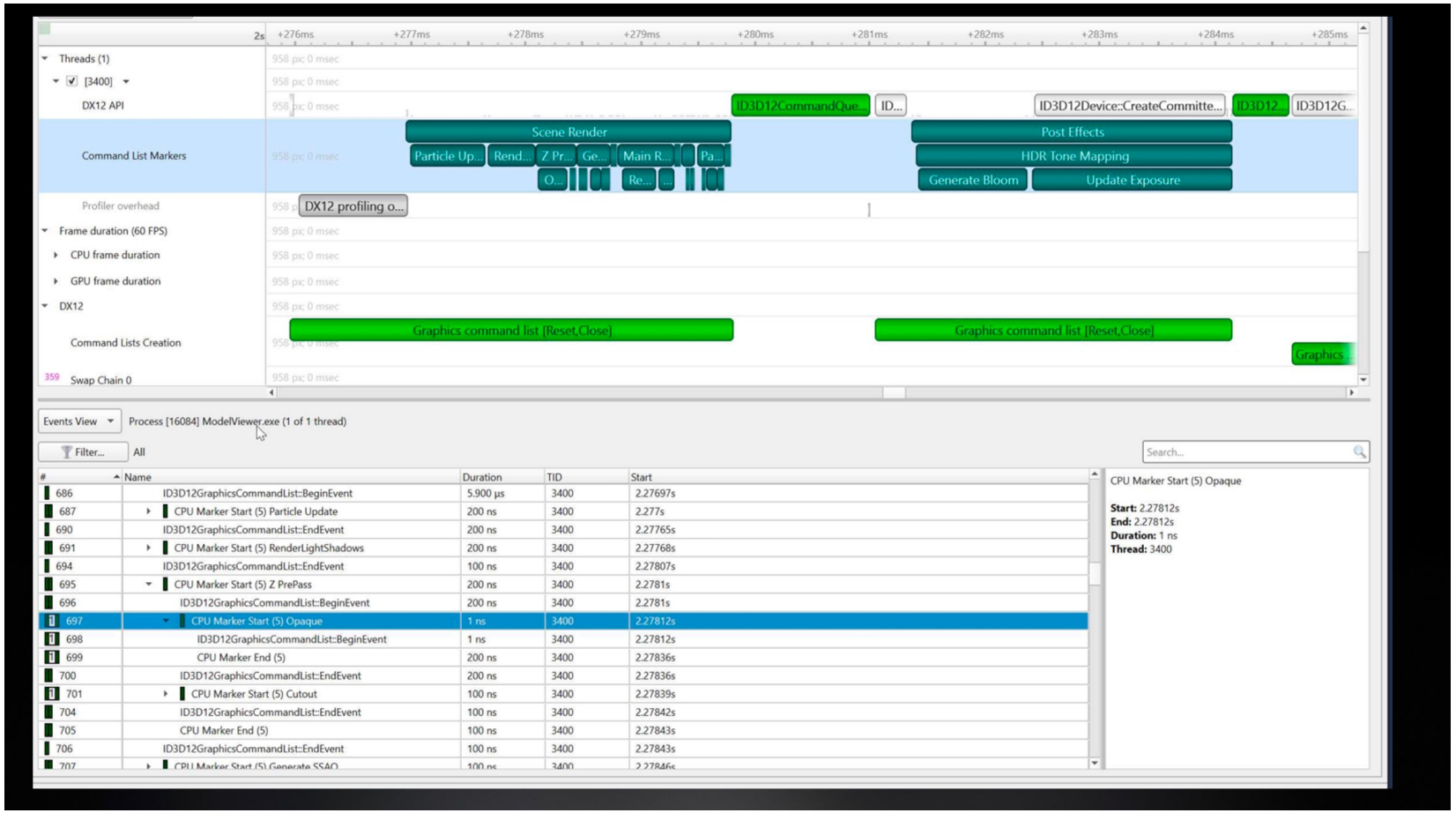
Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

What can be visualize with nsys



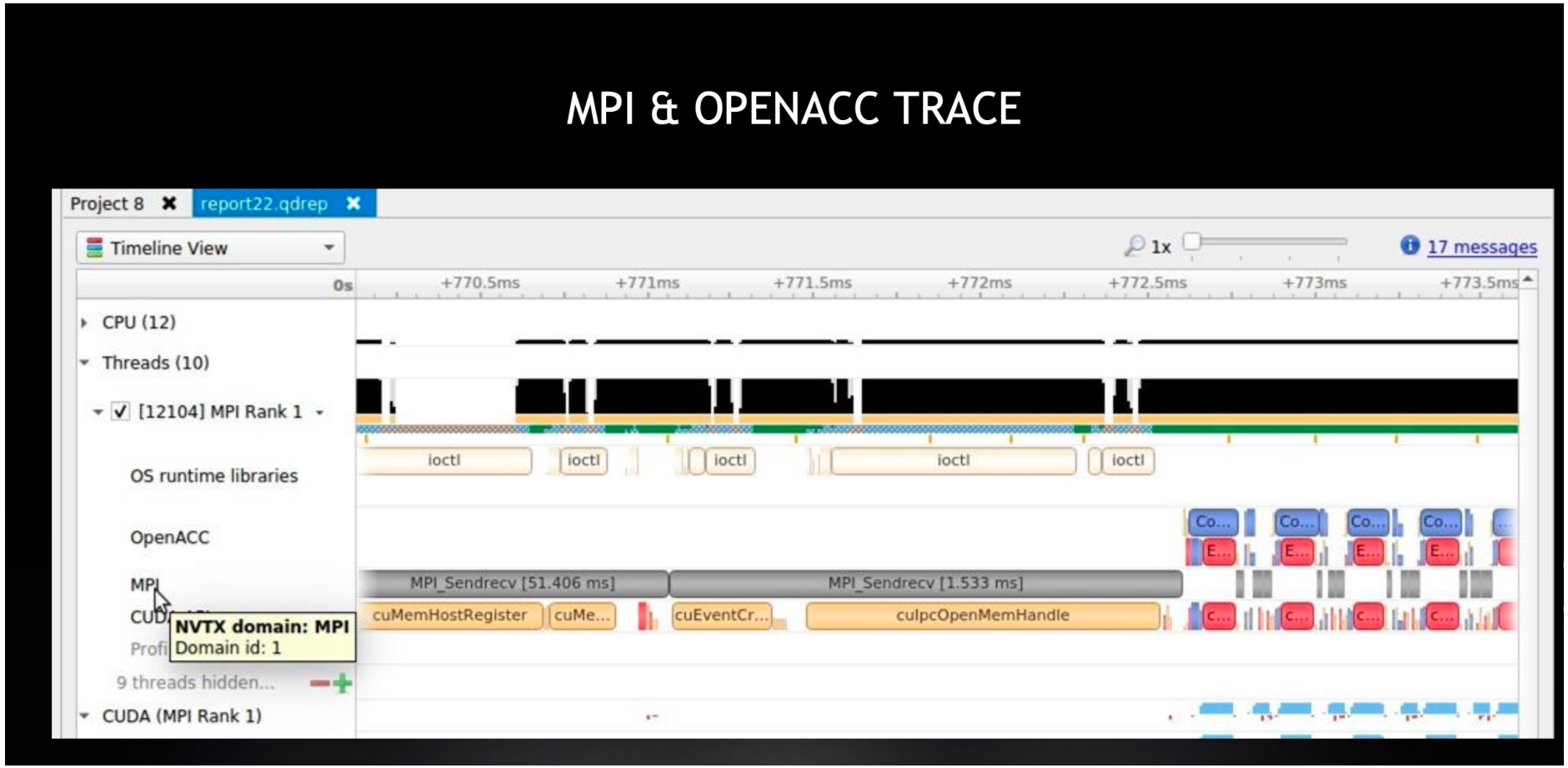
Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

What can be visualize with nsys



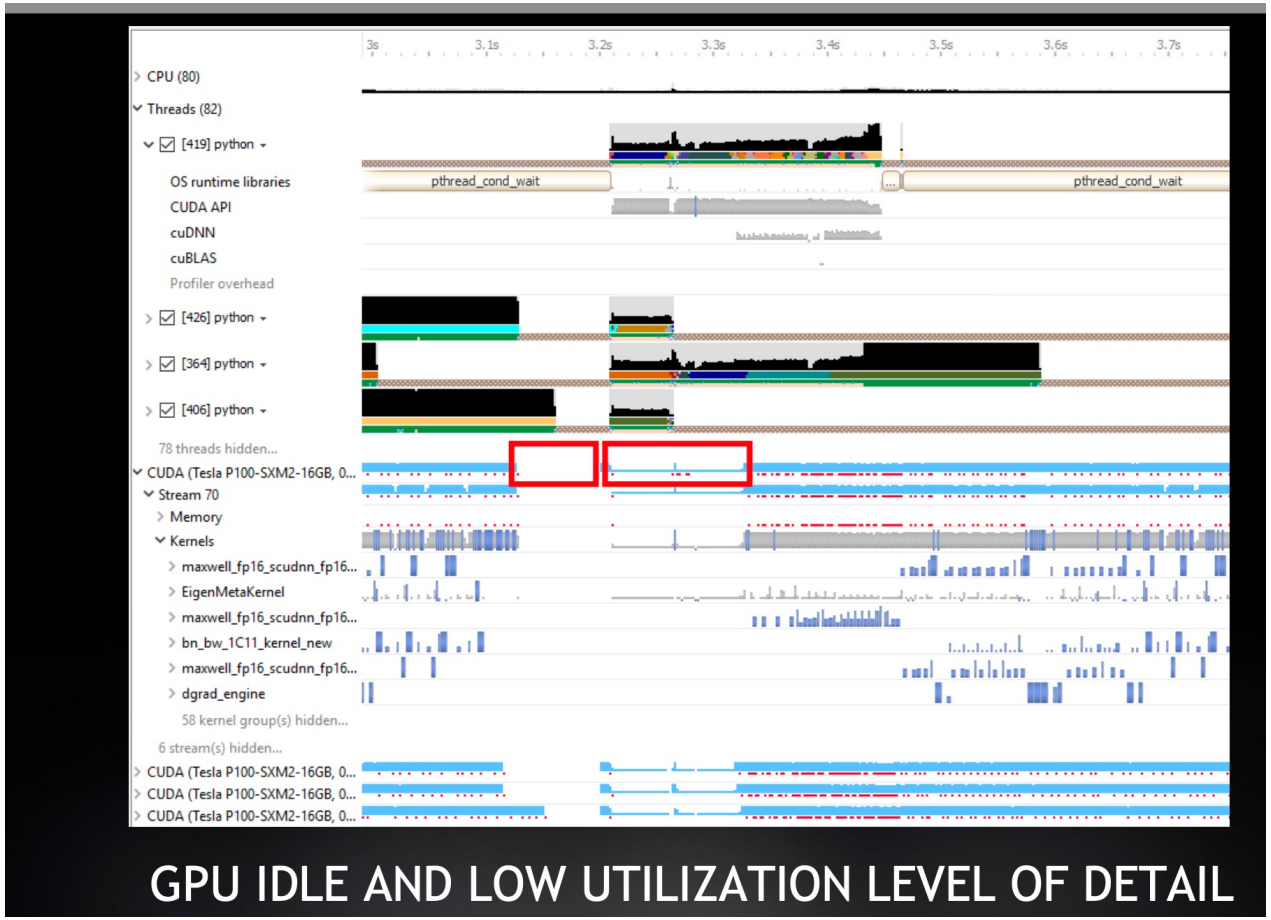
Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

What can be visualize with nsys



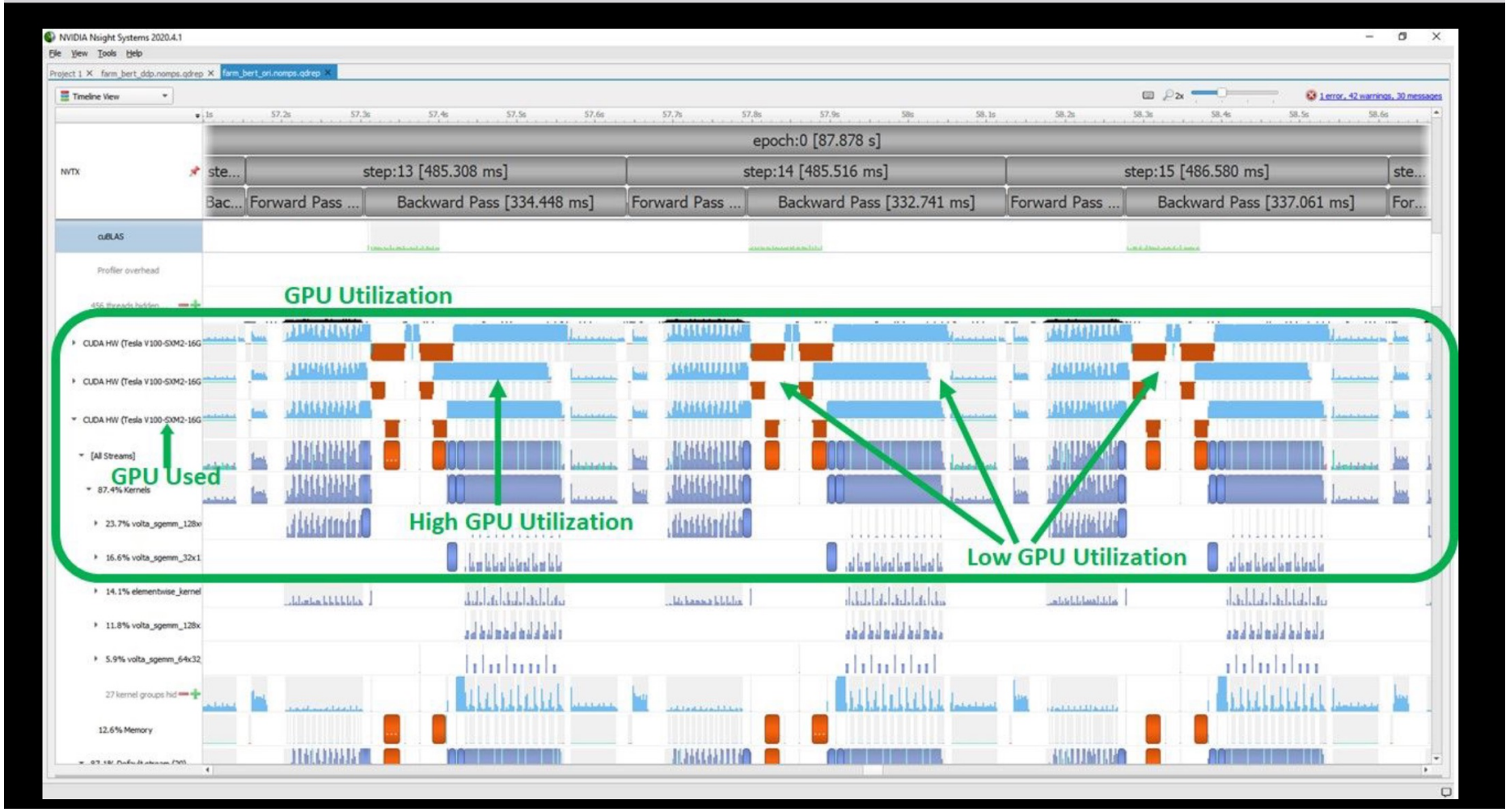
Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

What can be visualize with nsys



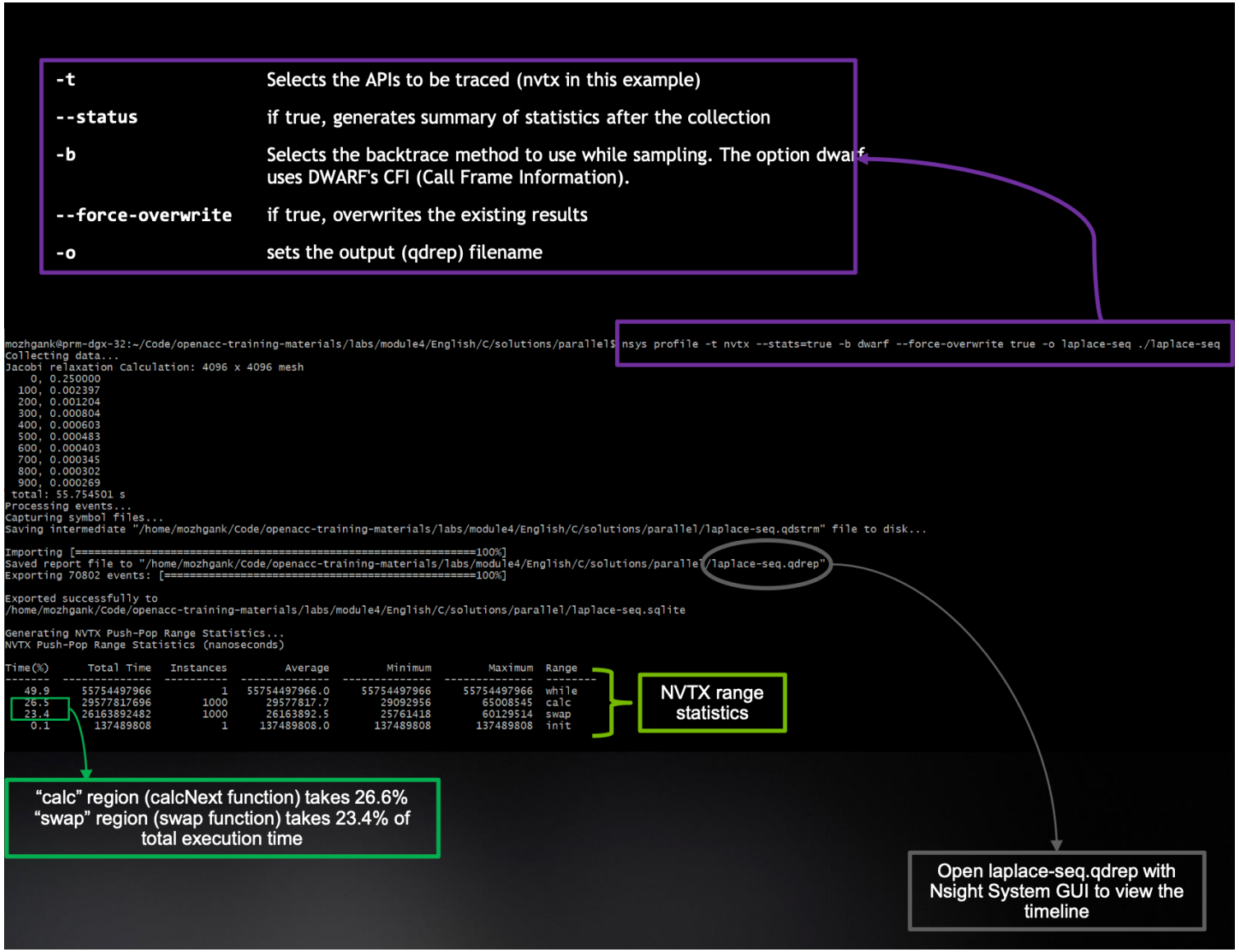
Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

High vs low GPU utilization



Source: https://alcf.anl.gov/sites/default/files/2022-07/Nsight%20Systems%20-

stats from algorithm



Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

Conclusion

- We were able to revisit key element of matrix computation on the GPU
- We briefly reviewed the architecture
- We worked on the Pi example and we compared it from the CPU to the GPU
- We introduced GPU profiling.