

Introduction to GPU programming (part 3)

By Kevin Fotso

A photograph of a person's hand pointing their index finger towards a computer monitor. The monitor displays a dark-themed code editor with white and yellow text. The visible code is a Python script related to Blender's operator classes, specifically for mirror modifiers. It includes logic for selecting objects based on their names and handles different mirror operations (X, Y, Z) and axis settings (use_x, use_y, use_z).

```
mirror_mod = modifier_obj
# set mirror object to mirror
mirror_mod.mirror_object = mirror_obj

if operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
elif operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

# selection at the end - add
modifier_obj.select= 1
modifier_obj.select=1
context.scene.objects.active = modifier_obj
print("Selected" + str(modifier_obj))
modifier_obj.select = 0
bpy.context.selected_objects = []
data.objects[one.name].select = 1
print("please select exactly one object")

# - OPERATOR CLASSES -
# --- MIRROR OPERATOR ---
def execute(self, context):
    # X mirror to the selected object.mirror_mirror_x
    # or X
    if context:
        if context.active_object is not None:
```



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

Recap

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.

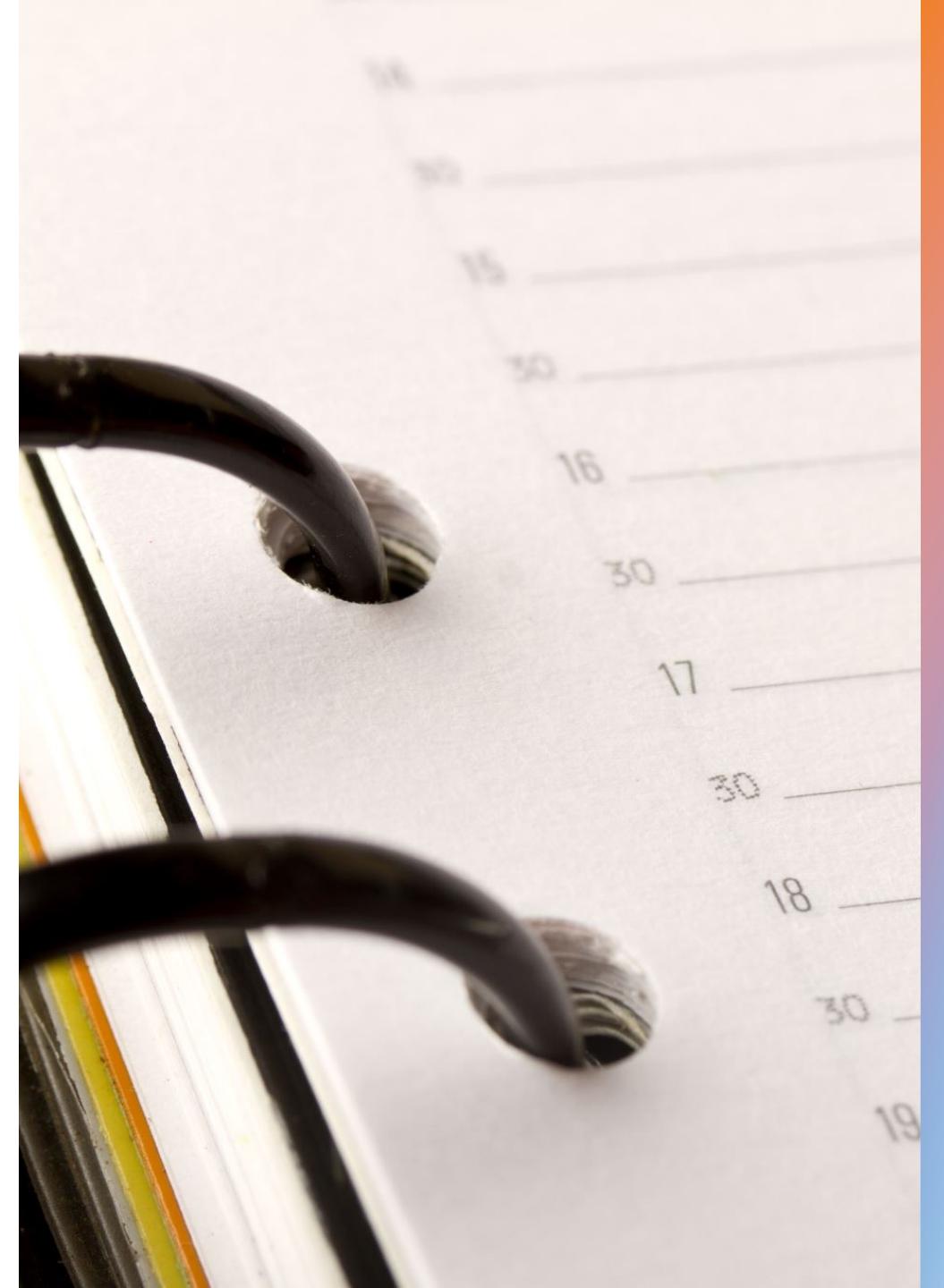
We learned about cupy, an open source library providing GPU accelerated computing with Python.

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

We saw very basic examples of matrix operation & cupy function

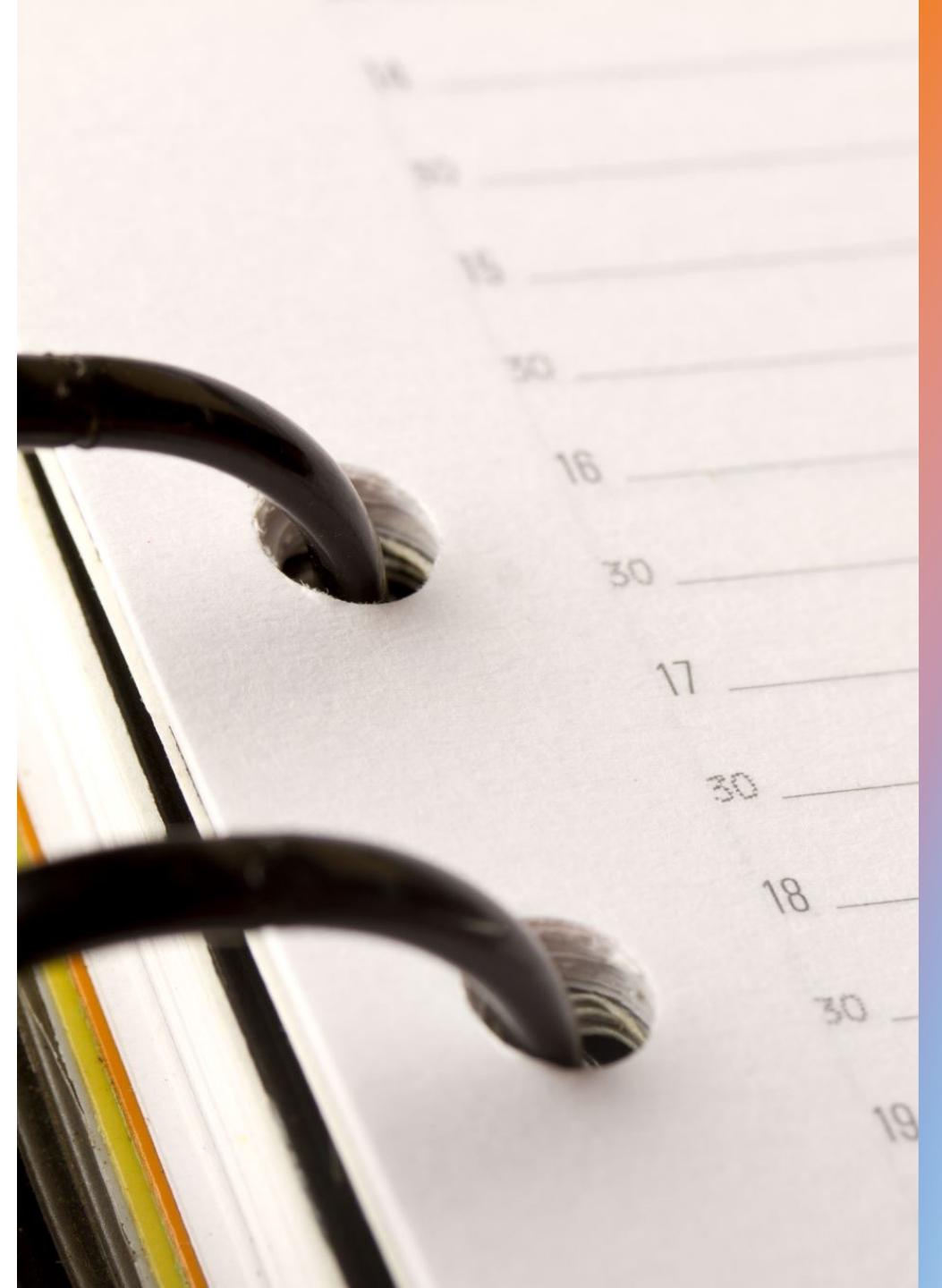
Today

- 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



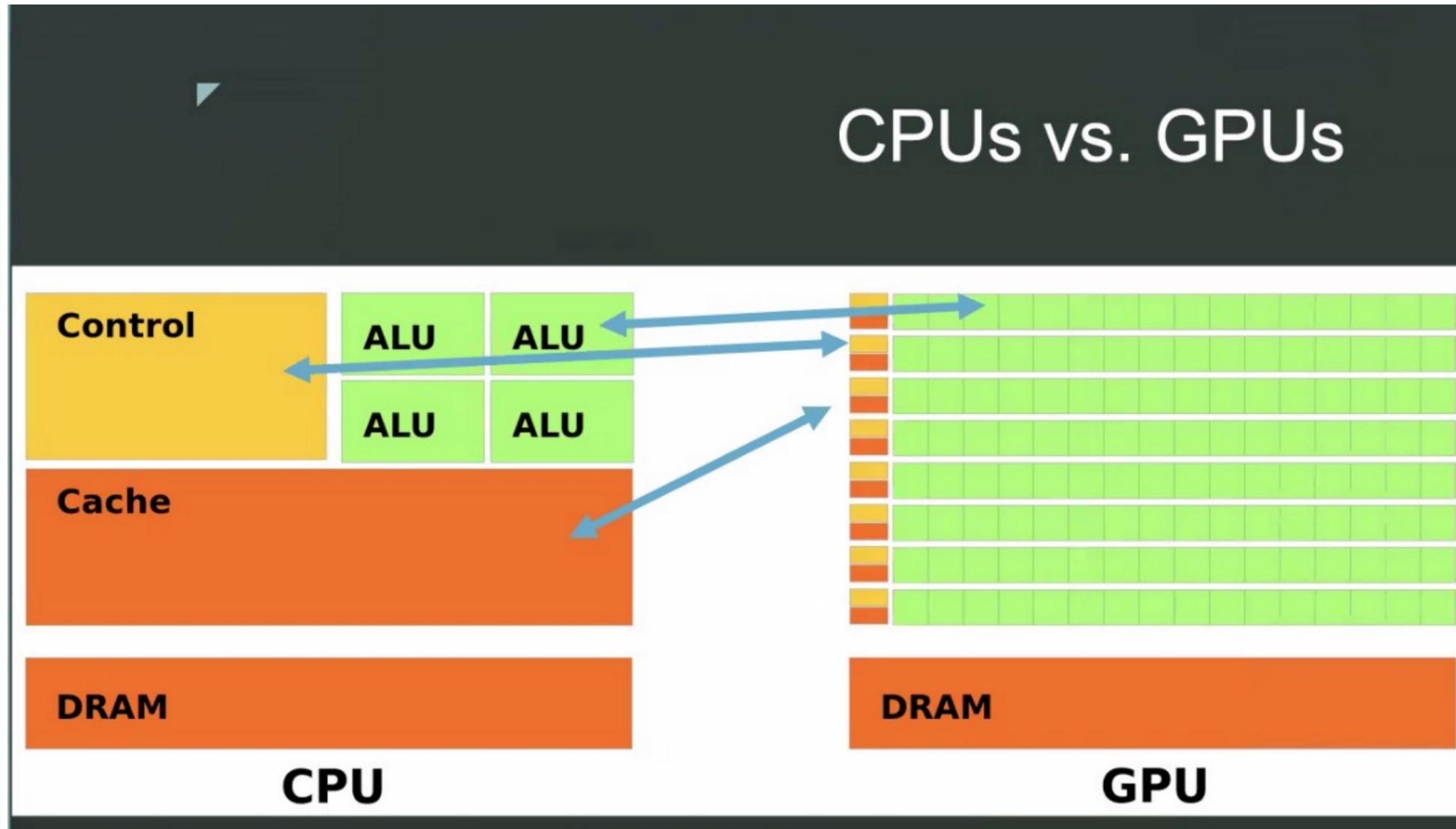
Today

- 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

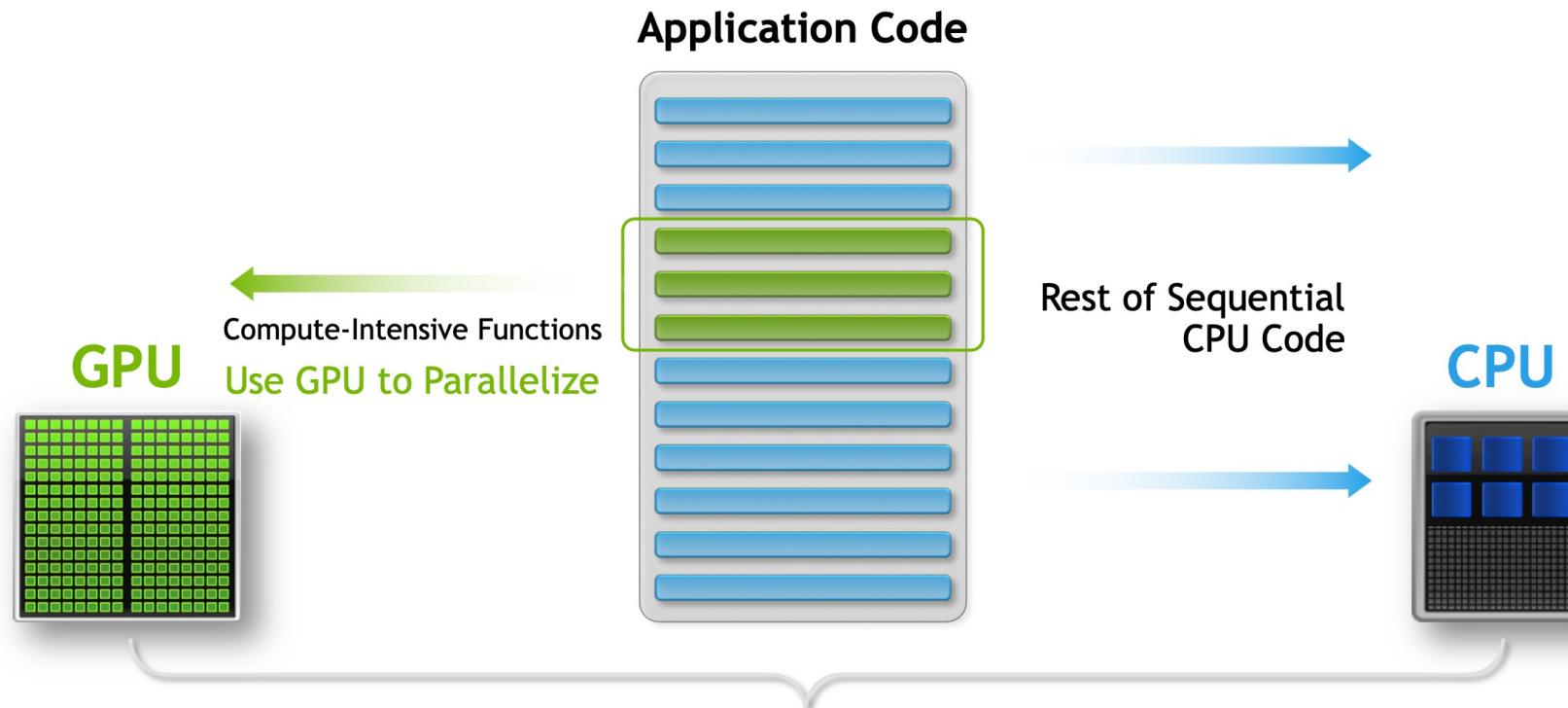


CPU vs GPU programming model

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

PORTING TO CUDA

CPU:
OS, security etc ...



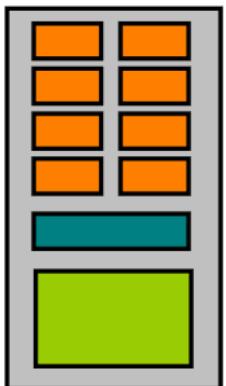
NVIDIA A100 specs



Scalar
Processor

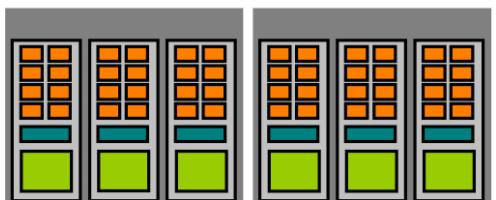


cuda core. **64 per SM**



streaming multiprocessor
: 108 SMs

Multiprocessor

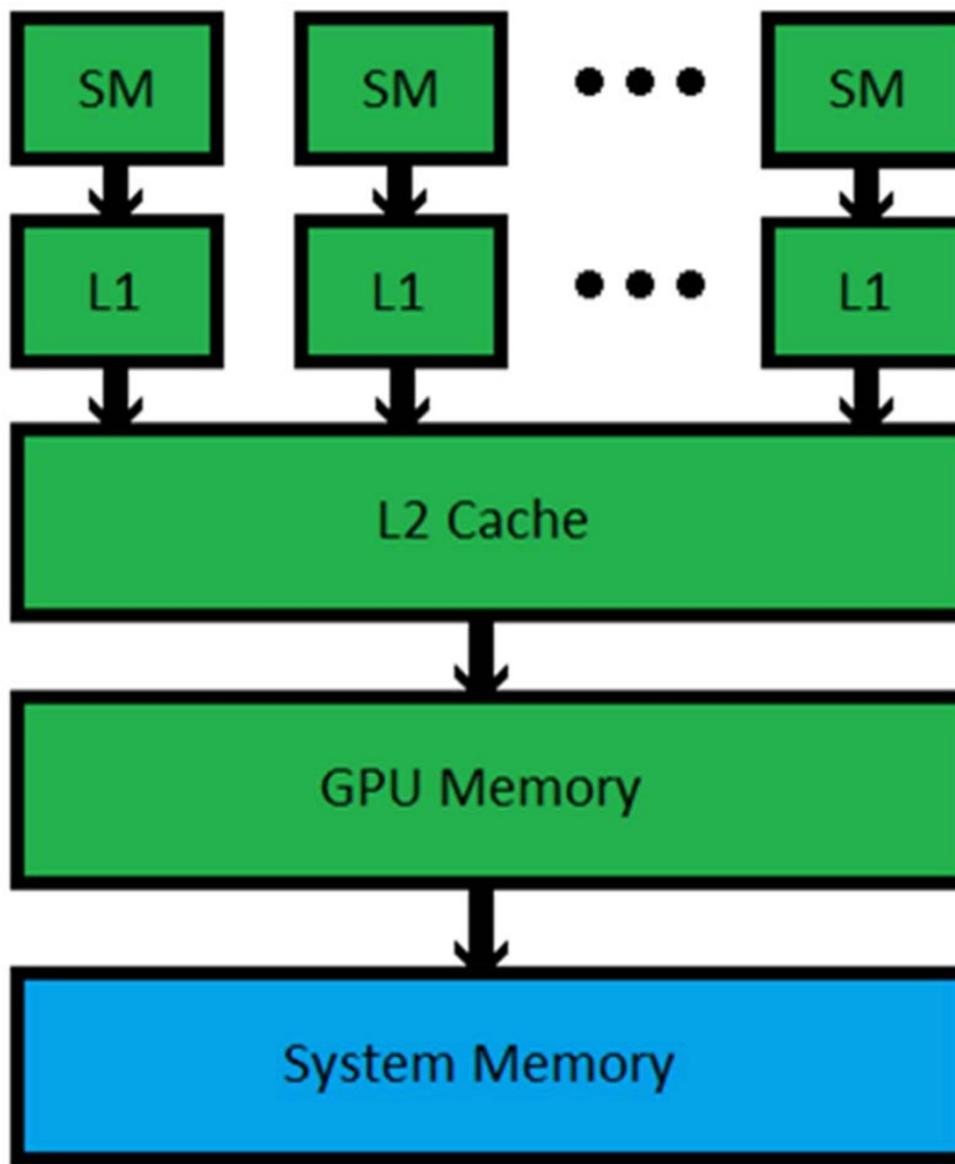


Device



108 SMs per device

NVIDIA GPU memory hierarchy



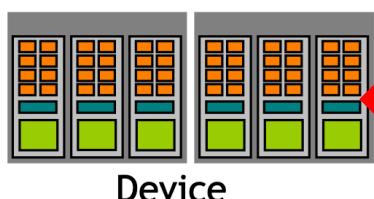
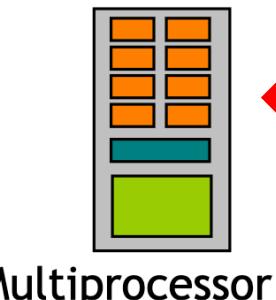
- Registers and local memory (local memory is slower)
- Shared memory is accessible by all threads in the SM
- Global memory is accessible by all blocks and the CPU

GPU software programming

Software



Hardware



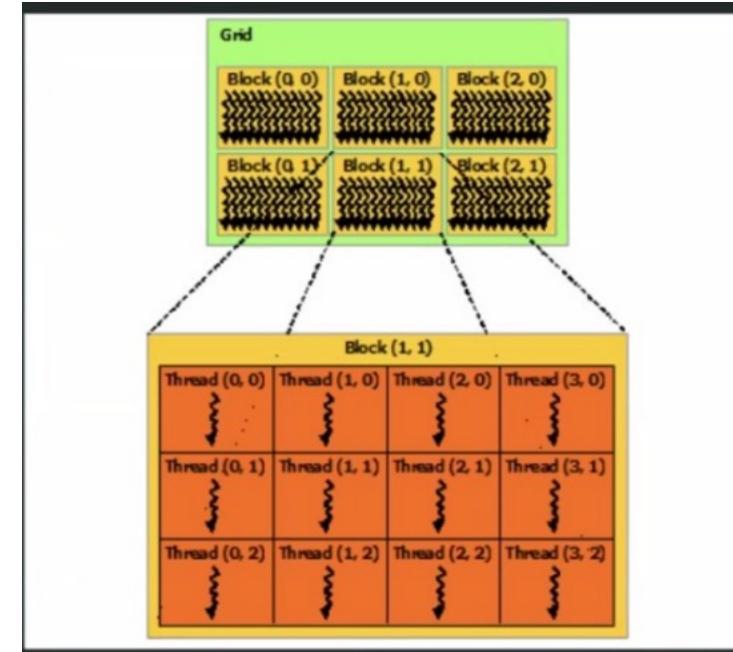
Threads are executed by scalar processors

Concurrent thread block can reside on 1 multiprocessor

A kernel is launched as a grid of thread blocks

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



Illustration

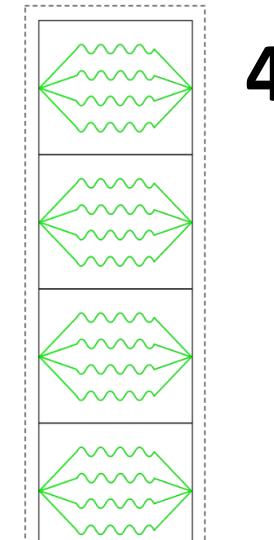
GPU kernel

```
__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];
}
```

blockDim

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

- E.g., 4 threads per block



Illustration

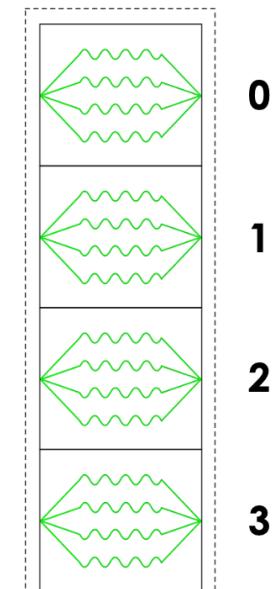
GPU kernel

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



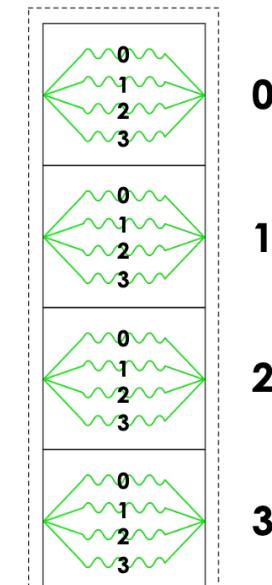
Illustration

GPU kernel

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

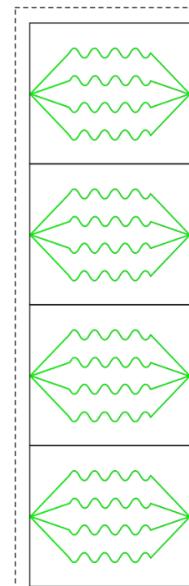


Illustration

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)
{
    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`...

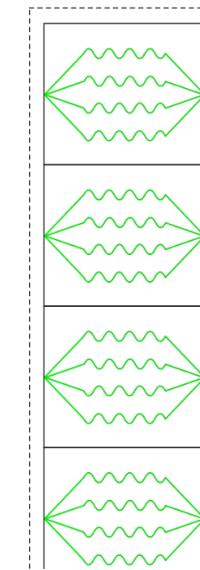
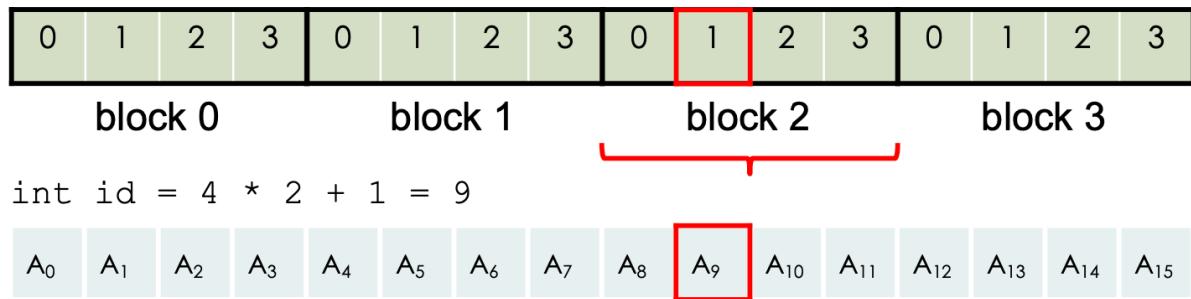


Illustration (Matrix addition)

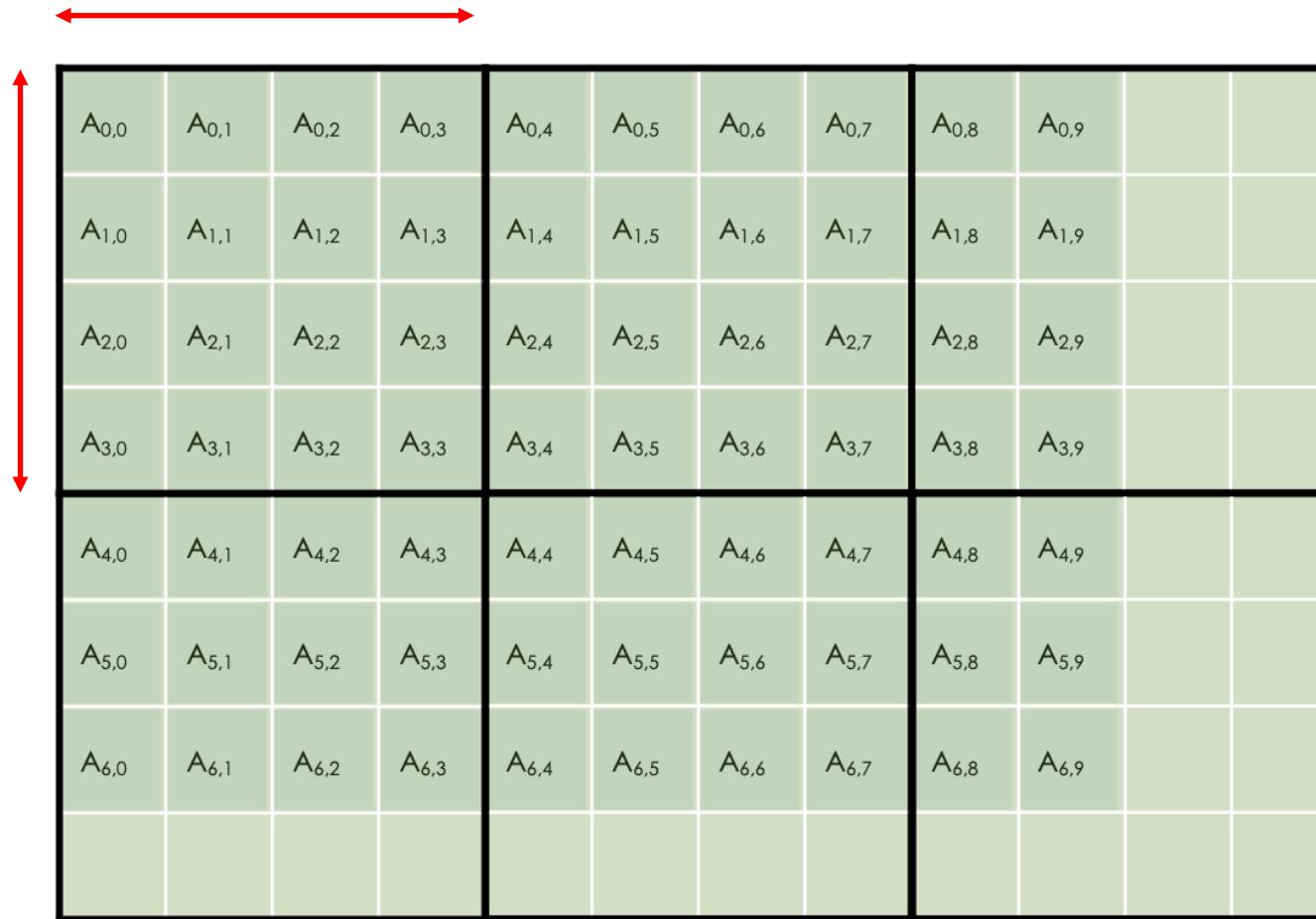
A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}	
A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}	
A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}	
A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}	
A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}	
A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}	
A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}	

Total size of Matrix is:

M = 7 rows

N = 10 columns

Illustration (Matrix addition)



A diagram illustrating a matrix A with 7 rows and 10 columns. The matrix is represented by a grid of 70 cells, each containing an element labeled $A_{i,j}$ where i is the row index and j is the column index. The rows are indexed from 0 to 6, and the columns are indexed from 0 to 9. The matrix is divided into two vertical sections by a thick black vertical line, creating two separate blocks of 4 columns each. The first block contains columns 0 through 3, and the second block contains columns 4 through 9. A red double-headed arrow is positioned above the matrix, spanning the width of the 10 columns, to indicate the total size of the matrix.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$	$A_{0,6}$	$A_{0,7}$	$A_{0,8}$	$A_{0,9}$	
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$	$A_{1,6}$	$A_{1,7}$	$A_{1,8}$	$A_{1,9}$	
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$	$A_{2,6}$	$A_{2,7}$	$A_{2,8}$	$A_{2,9}$	
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$	$A_{3,6}$	$A_{3,7}$	$A_{3,8}$	$A_{3,9}$	
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$	$A_{4,6}$	$A_{4,7}$	$A_{4,8}$	$A_{4,9}$	
$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$	$A_{5,6}$	$A_{5,7}$	$A_{5,8}$	$A_{5,9}$	
$A_{6,0}$	$A_{6,1}$	$A_{6,2}$	$A_{6,3}$	$A_{6,4}$	$A_{6,5}$	$A_{6,6}$	$A_{6,7}$	$A_{6,8}$	$A_{6,9}$	

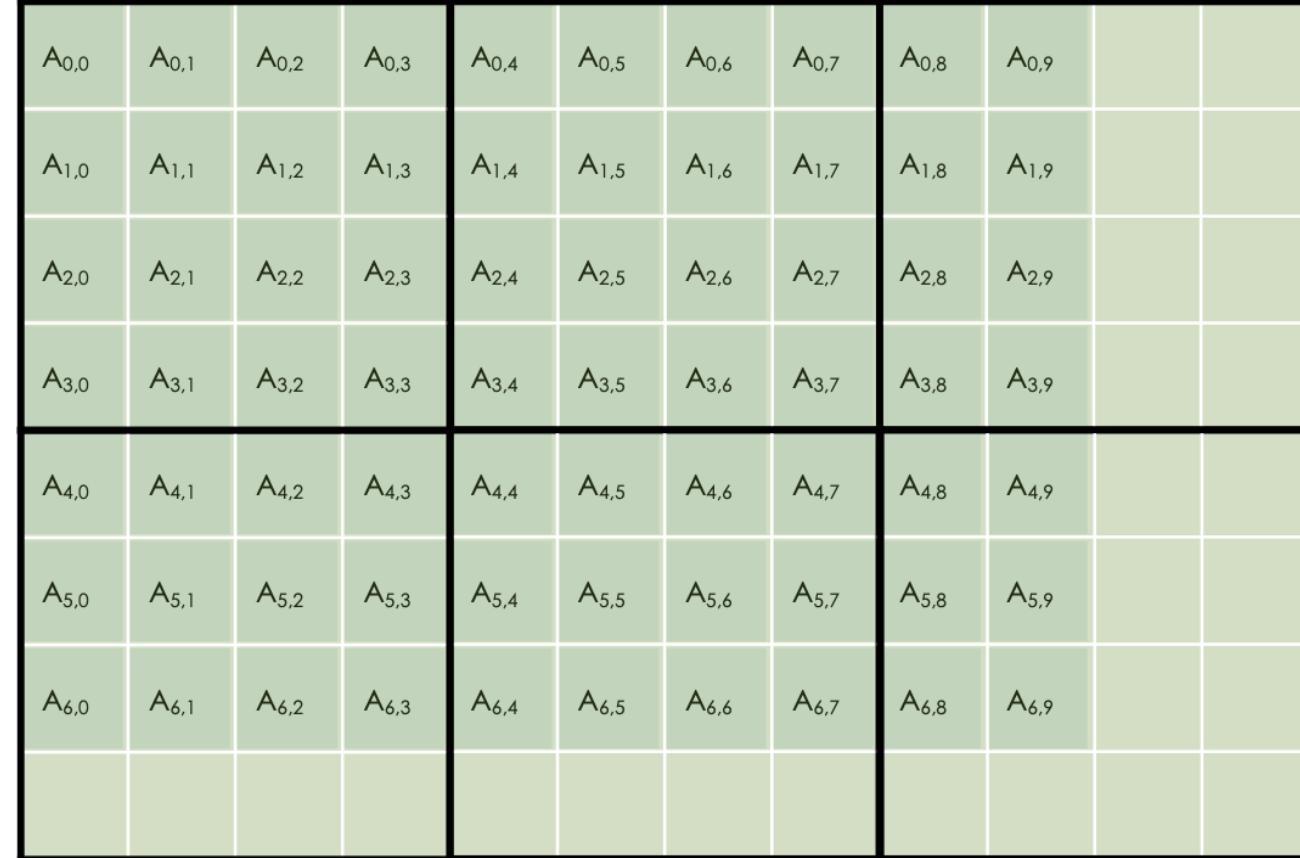
Total size of Matrix is:

$M = 7$ rows

$N = 10$ columns

But we want 4X4 block of thread !!

Illustration (Matrix addition)



A diagram showing a 7x10 matrix A. The matrix is represented by a grid of 7 rows and 10 columns. Each cell contains an element labeled A_{i,j}, where i is the row index (0 to 6) and j is the column index (0 to 9). The matrix is divided into two vertical sections by a thick black vertical line. The left section contains elements A_{0,0} through A_{3,3}, and the right section contains elements A_{4,0} through A_{6,9}. A red double-headed arrow is positioned above the matrix, spanning the width of the right section (from column 4 to 10), indicating the total width of the matrix.

A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}	
A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}	
A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}	
A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}	
A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}	
A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}	
A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}	

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)

A diagram of a 7x10 matrix A. The matrix is represented by a grid of 7 rows and 10 columns. Each cell contains a label A_{i,j}, where i is the row index (0 to 6) and j is the column index (0 to 9). The matrix is divided into two vertical sections by a thick black vertical line. The left section contains columns 0 to 3, and the right section contains columns 4 to 9. Red arrows point from the top-left corner to the bottom-right corner, spanning both the width and height of the matrix.

A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}	
A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}	
A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}	
A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}	
A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}	
A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}	
A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}	

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 );
```

$$\text{ceil}(10/4) = \text{ceil}(2.5) = 3$$

$$\text{ceil}(7/4) = \text{ceil}(1.75) = 2$$

Let's find the threadID

	0	1	2	3	4	5	6	7	8	9	10	11
0	A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}		
1	A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}		
2	A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}		
3	A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}		
4	A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}		
5	A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}		
6	A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}		
7												

Let's find the threadID

	0	1	2	3	4	5	6	7	8	9	10	11
0	A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{0,5}	A _{0,6}	A _{0,7}	A _{0,8}	A _{0,9}		
1	A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{1,5}	A _{1,6}	A _{1,7}	A _{1,8}	A _{1,9}		
2	A _{2,0}	A _{2,1}	A _{2,2}	A _{2,3}	A _{2,4}	A _{2,5}	A _{2,6}	A _{2,7}	A _{2,8}	A _{2,9}		
3	A _{3,0}	A _{3,1}	A _{3,2}	A _{3,3}	A _{3,4}	A _{3,5}	A _{3,6}	A _{3,7}	A _{3,8}	A _{3,9}		
4	A _{4,0}	A _{4,1}	A _{4,2}	A _{4,3}	A _{4,4}	A _{4,5}	A _{4,6}	A _{4,7}	A _{4,8}	A _{4,9}		
5	A _{5,0}	A _{5,1}	A _{5,2}	A _{5,3}	A _{5,4}	A _{5,5}	A _{5,6}	A _{5,7}	A _{5,8}	A _{5,9}		
6	A _{6,0}	A _{6,1}	A _{6,2}	A _{6,3}	A _{6,4}	A _{6,5}	A _{6,6}	A _{6,7}	A _{6,8}	A _{6,9}		
7												

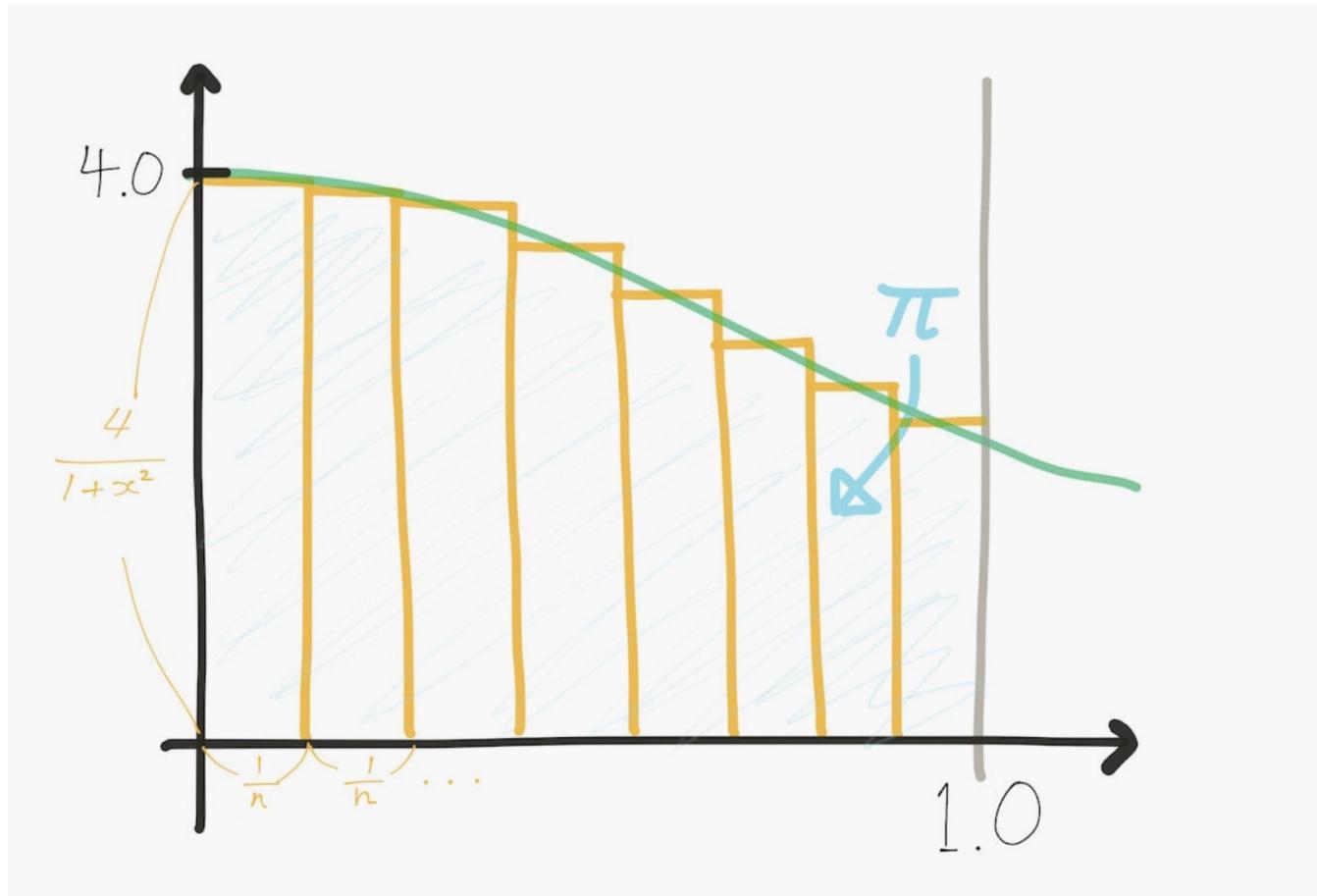
```
__global__ void matrix_addition(double *a, double *b, double *c) {
    int column = blockDim.x * blockIdx.x + threadIdx.x;      (0 - 11)
    int row    = blockDim.y * blockIdx.y + threadIdx.y;      (0 - 7)
    if (row < M && column < N) {
        int thread_id = row * N + column;                    (0 - 69)
        c[thread_id] = a[thread_id] + b[thread_id];
```

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

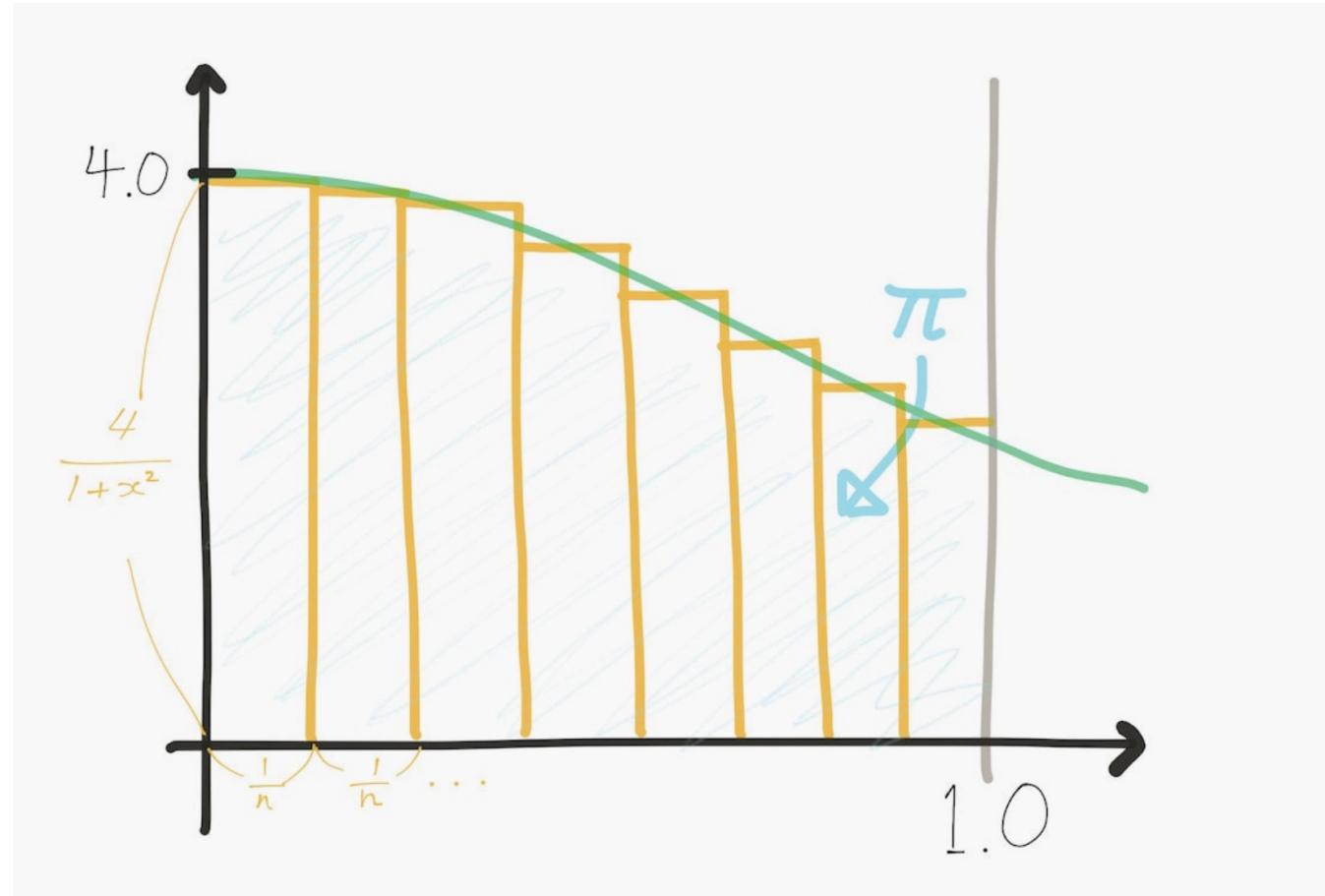
$$\begin{aligned} \text{thread_id} &= \text{row} * N + \text{column} \\ &= 5 * 10 + 6 = 56 \end{aligned}$$

III – The PI example

How to estimate PI?

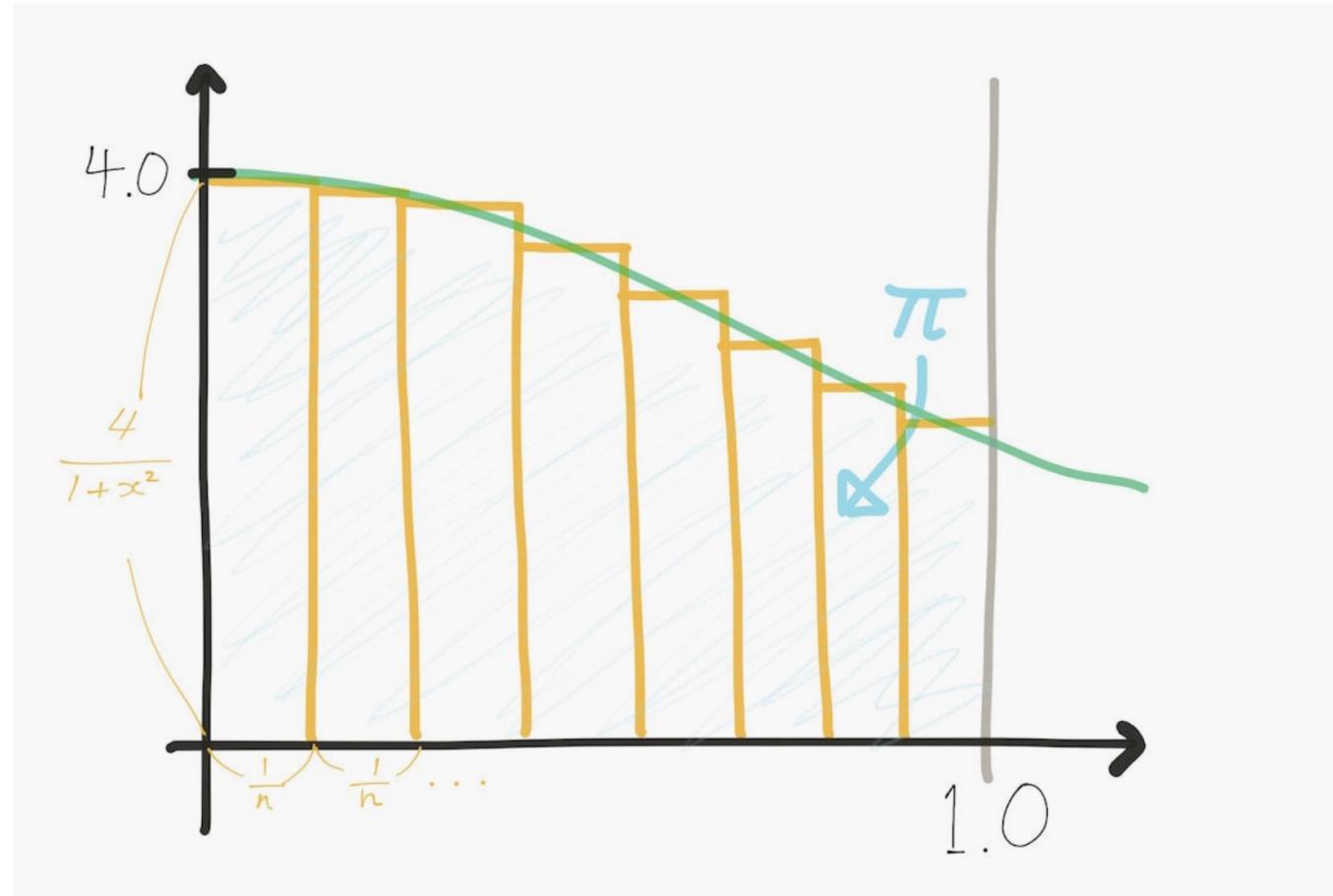


How to estimate PI?



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

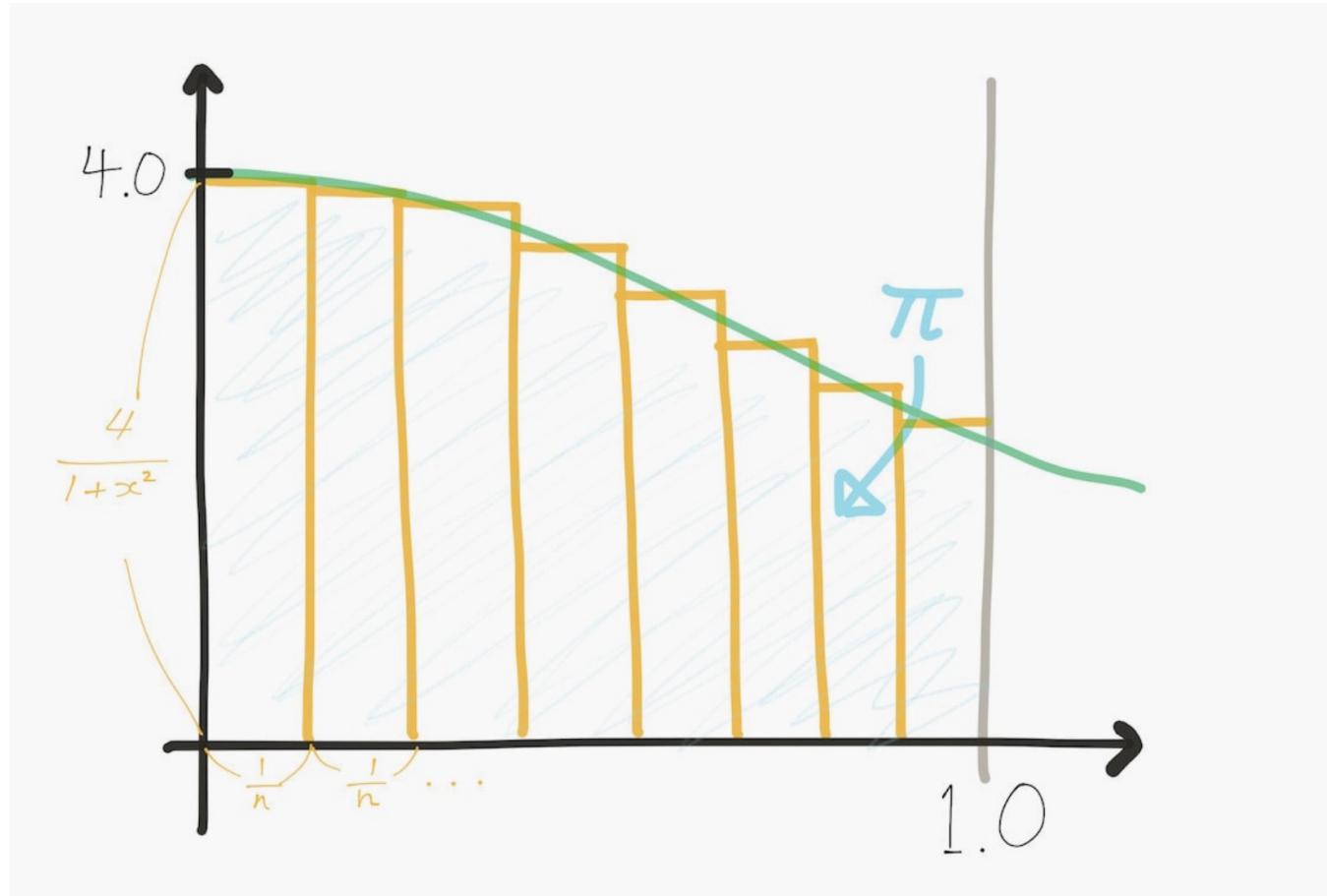
How to estimate PI?



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

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

How to estimate PI?



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

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

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

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

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)
```

Riemann sum of each rectangles (steps)

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

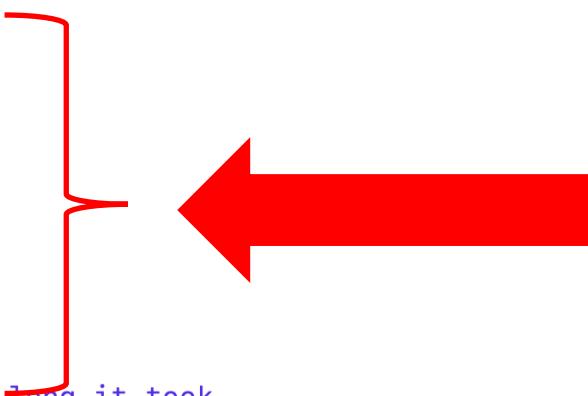
if len(sys.argv)!=2:
    print("Usage: ", sys.argv[0], "<number of steps>")
    sys.exit(1)

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)
```



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=0.000000833332957022) (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 threadIdx
within 1 block, the
blockId, the block size
and the grid size

Computation of pi on the GPU

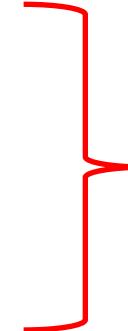
```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):

    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-Writing%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
```

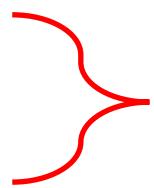
Calculate
the sum
vector

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 = 3.14159273692312268622, (Diff=0.0000008333332957022) (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 = 3.14159297943115234375, (Diff=0.0000023841857910156) (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=-0.00003617358979290941) (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=0.0000000000004574119) (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=-0.000000000006217249) (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=-0.000000000000266454) (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

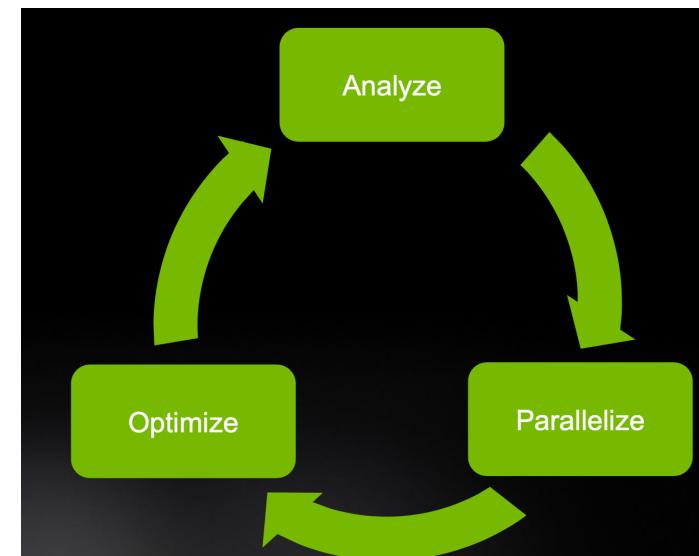
- Analyze code: for assessing the blocks that need optimization

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



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-%20CPU%20and%20GPU%20Performance%20Analysis%20with%20Nsight%20Evo.pdf>

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)

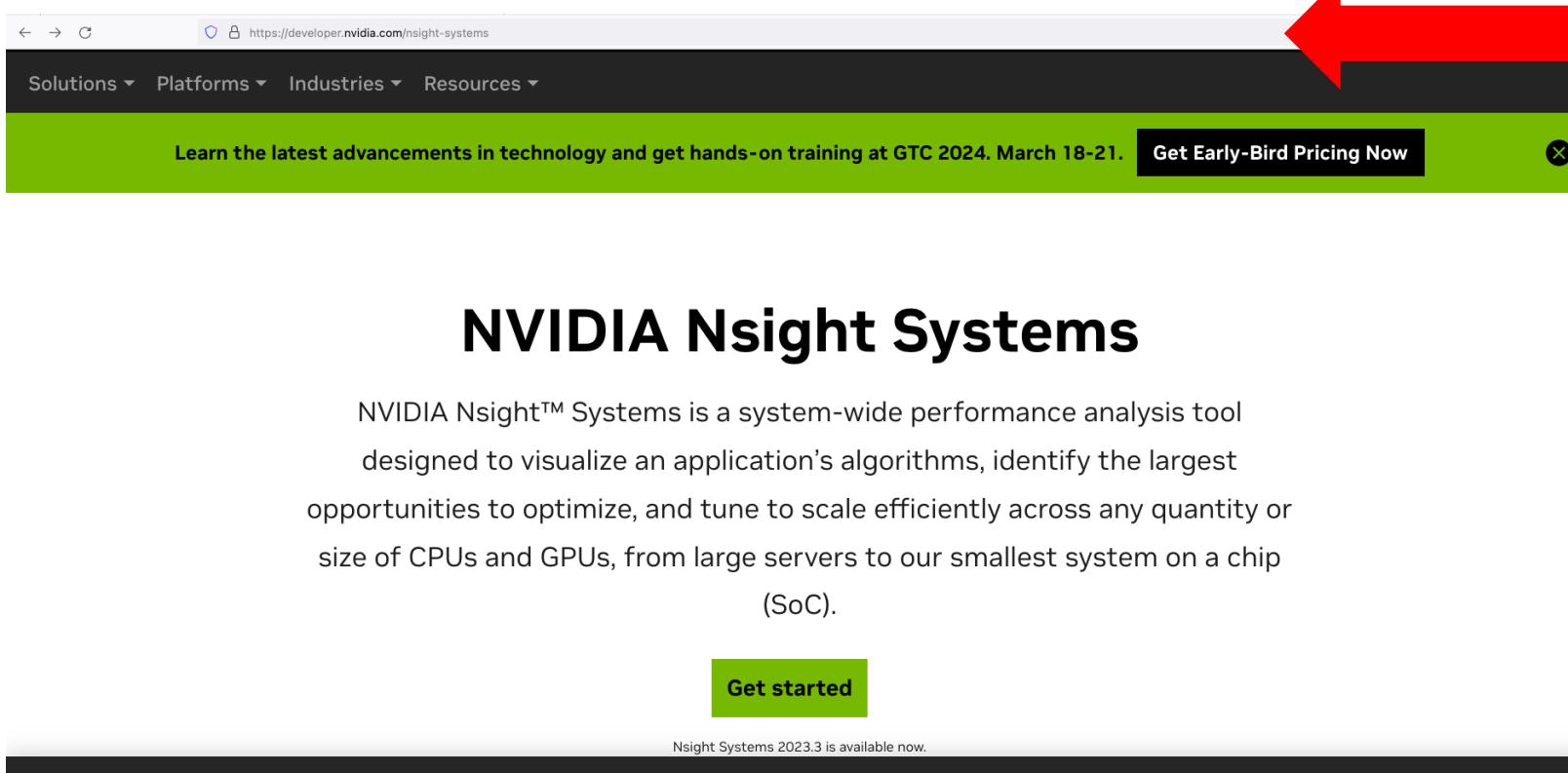
```
# We test it with 4 GPUs
echo "Testing it with 4 GPUs"
nsys profile --trace=cuda,nvtx,osrt,cudnn,cublas \
    --capture-range=cudaProfilerApi \
    --gpu-metrics-device=all \
    --output=horovod_profile \
    horovodrun -np 4 -H $node1:2,$node2:2
horovod_housing_tf.py
```



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



A screenshot of a web browser displaying the NVIDIA developer website at <https://developer.nvidia.com/nsight-systems>. The page title is "NVIDIA Nsight Systems". The main content area describes the tool as a system-wide performance analysis tool designed to visualize algorithms, identify optimization opportunities, and tune efficiently across various hardware. A prominent green "Get started" button is visible at the bottom. A red arrow points from the top right towards the browser's address bar.

Solutions ▾ Platforms ▾ Industries ▾ Resources ▾

Learn the latest advancements in technology and get hands-on training at GTC 2024. March 18-21. [Get Early-Bird Pricing Now](#)

NVIDIA Nsight Systems

NVIDIA Nsight™ Systems is a system-wide performance analysis tool designed to visualize an application's algorithms, identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs, from large servers to our smallest system on a chip (SoC).

[Get started](#)

Nsight Systems 2023.3 is available now.

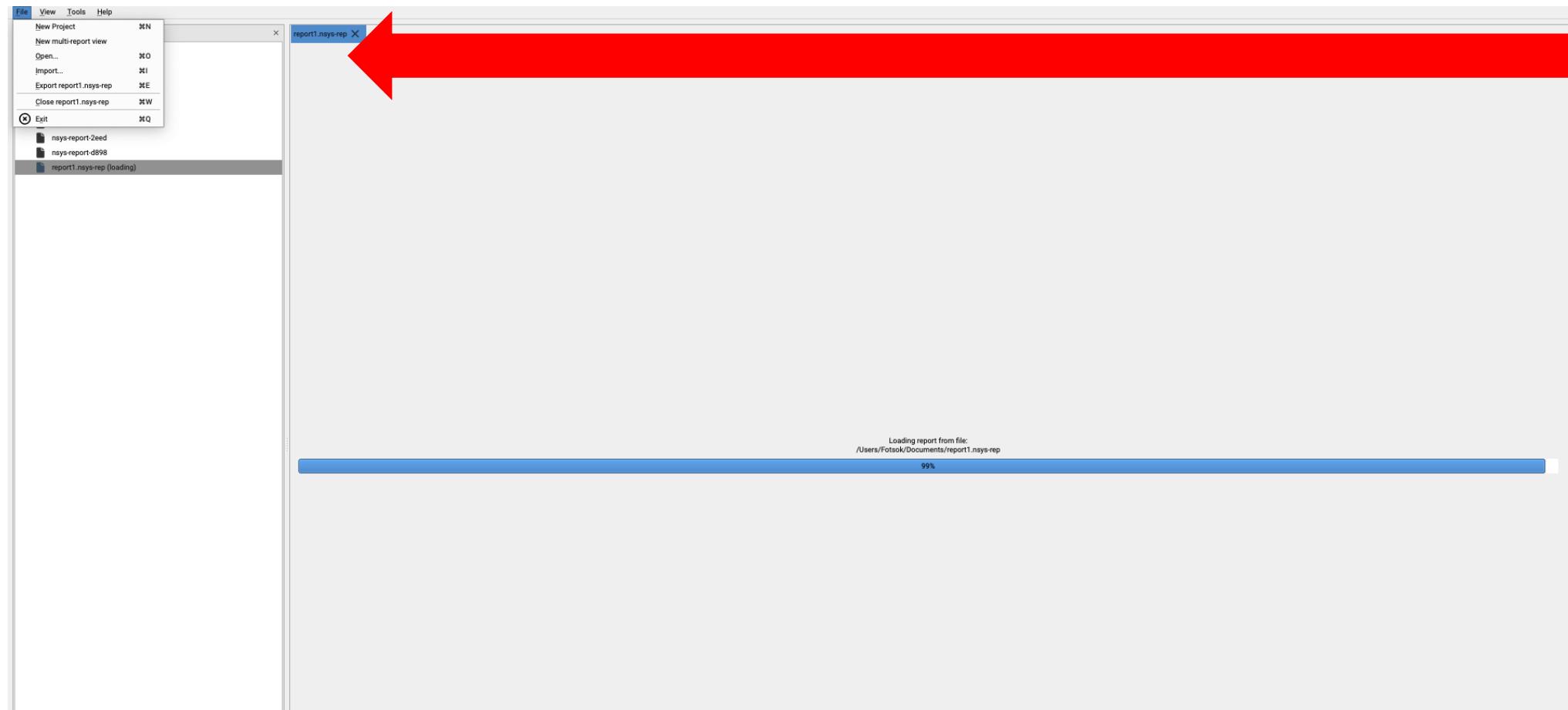
<https://developer.nvidia.com/nsight-systems>

NVIDIA Nsight (2)



Name	Date Modified	Size	Kind
Image Capture	Sep 16, 2023, 7:28 AM	3.2 MB	Application
iMovie	Feb 9, 2023, 1:41 AM	3.26 GB	Application
Keynote	Feb 9, 2023, 1:42 AM	710 MB	Application
Launchpad	Sep 16, 2023, 7:28 AM	710 KB	Application
Linaro Forge Client 23.0.1	Jul 25, 2023, 4:47 PM	121.5 MB	Application
Mail	Sep 16, 2023, 7:28 AM	27.5 MB	Application
Maps	Sep 16, 2023, 7:28 AM	71.1 MB	Application
MATLAB_R2021b	Apr 23, 2023, 8:19 PM	9.49 GB	Application
MATLAB_R2023a	Apr 21, 2023, 10:49 PM	6.62 GB	Application
Messages	Sep 16, 2023, 7:28 AM	5.7 MB	Application
Microsoft Defender	Oct 17, 2023, 4:14 AM	840.8 MB	Application
Microsoft Excel	Oct 10, 2023, 10:16 AM	2.06 GB	Application
Microsoft OneNote	Oct 10, 2023, 10:17 AM	1.14 GB	Application
Microsoft Outlook	Oct 18, 2023, 10:50 AM	2.18 GB	Application
Microsoft PowerPoint	Oct 13, 2023, 9:00 AM	1.8 GB	Application
Microsoft Teams classic	Oct 6, 2023, 4:49 PM	491.5 MB	Application
Microsoft Word	Oct 13, 2023, 9:22 AM	2.36 GB	Application
Mission Control	Sep 16, 2023, 7:28 AM	301 KB	Application
Music	Sep 16, 2023, 7:28 AM	103 MB	Application
News	Sep 16, 2023, 7:28 AM	10.3 MB	Application
Notes	Sep 16, 2023, 7:28 AM	31.6 MB	Application
Numbers	Feb 9, 2023, 1:42 AM	579.2 MB	Application
NVIDIA Nsight Systems	Nov 16, 2022, 11:14 PM	1.1 GB	Application
OneDrive	Today, 6:59 PM	1.18 GB	Application
Pages	Feb 9, 2023, 1:42 AM	642.6 MB	Application
ParaView-5.1.1	Mar 30, 2023, 11:26 AM	1.13 GB	Application
Photo Booth	Sep 16, 2023, 7:28 AM	4.5 MB	Application
Photos	Sep 16, 2023, 7:28 AM	40.9 MB	Application
Podcasts	Sep 16, 2023, 7:28 AM	49.1 MB	Application
Preview	Sep 16, 2023, 7:28 AM	9.6 MB	Application
QuickTime Player	Sep 16, 2023, 7:28 AM	6.5 MB	Application
Reminders	Sep 16, 2023, 7:28 AM	24 MB	Application
Safari	Sep 15, 2023, 6:20 PM	13.8 MB	Application
Shortcuts	Sep 16, 2023, 7:28 AM	4.9 MB	Application
Siri	Sep 16, 2023, 7:28 AM	2.5 MB	Application
Slack	Sep 28, 2023, 10:06 PM	277.6 MB	Application
Stickers	Sep 16, 2023, 7:28 AM	1.7 MB	Application
Stocks	Sep 16, 2023, 7:28 AM	6.5 MB	Application
System Settings	Sep 16, 2023, 7:28 AM	7.9 MB	Application
TextEdit	Sep 16, 2023, 7:28 AM	2.4 MB	Application
Tim Machine	Sep 16, 2023, 7:28 AM	1.2 MB	Application
TV	Sep 16, 2023, 7:28 AM	78.1 MB	Application
Utilities	Sep 28, 2023, 12:16 PM	--	Folder
Visit	Apr 9, 2023, 8:20 PM	1.24 GB	Application
Voice Memos	Sep 16, 2023, 7:28 AM	6.1 MB	Application
Weather	Sep 16, 2023, 7:28 AM	56.7 MB	Application
Webex	Mar 6, 2023, 6:26 PM	863.5 MB	Application
zoom.us	Oct 12, 2023, 8:16 PM	240.2 MB	Application

NVIDIA Nsight (3)

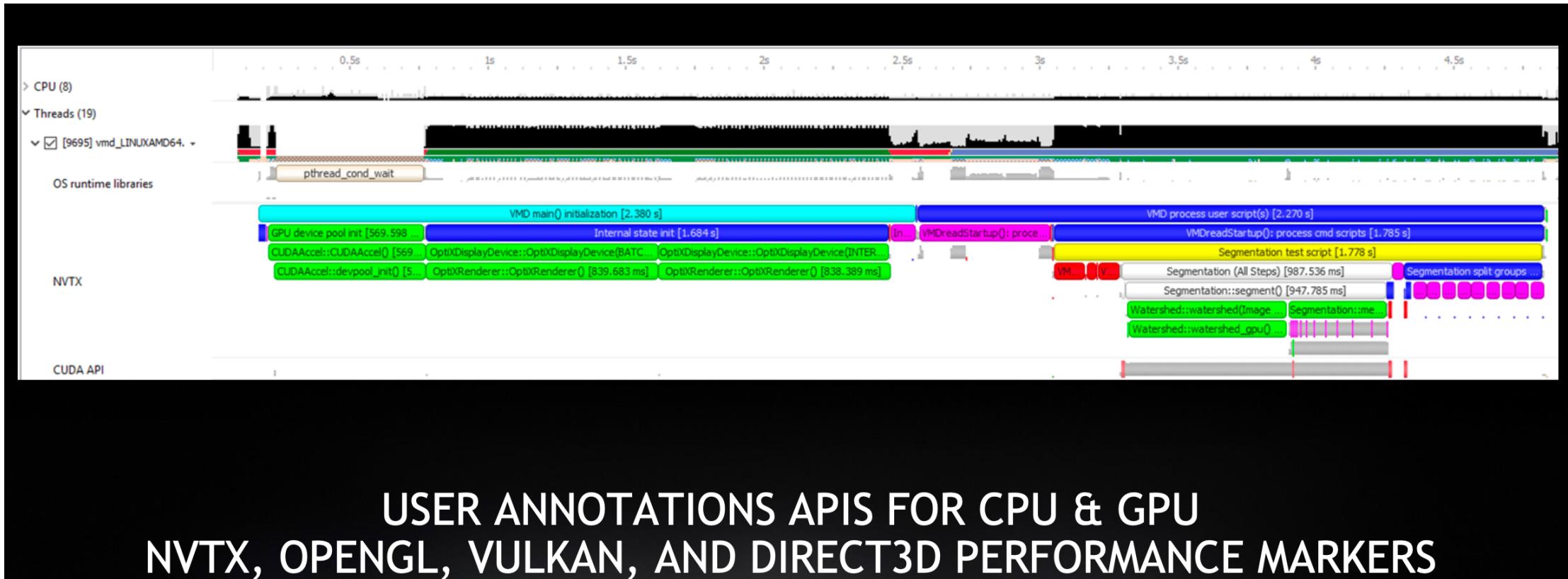


Click on open

What can be visualize with nsys

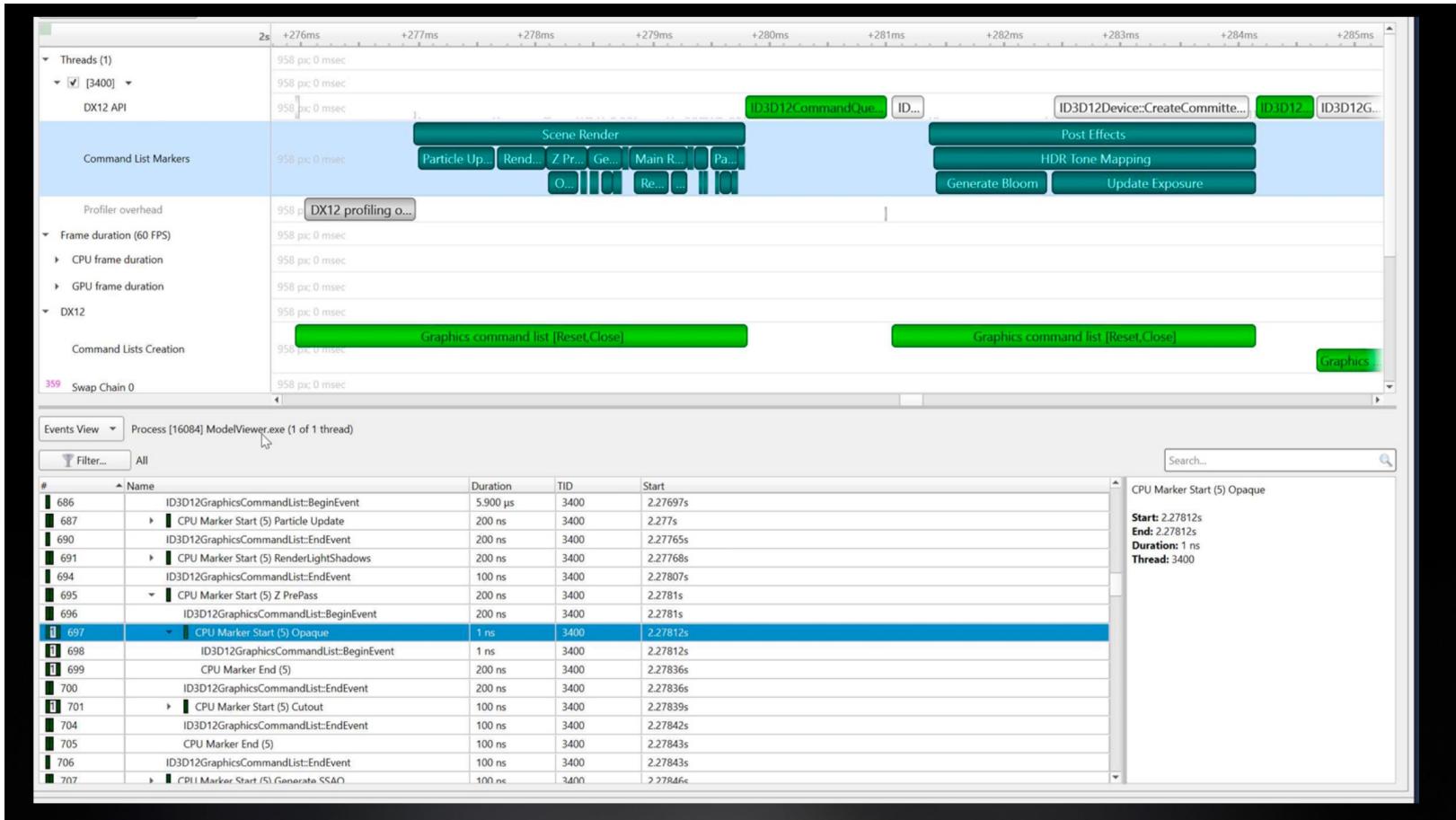


What can be visualize with nsys



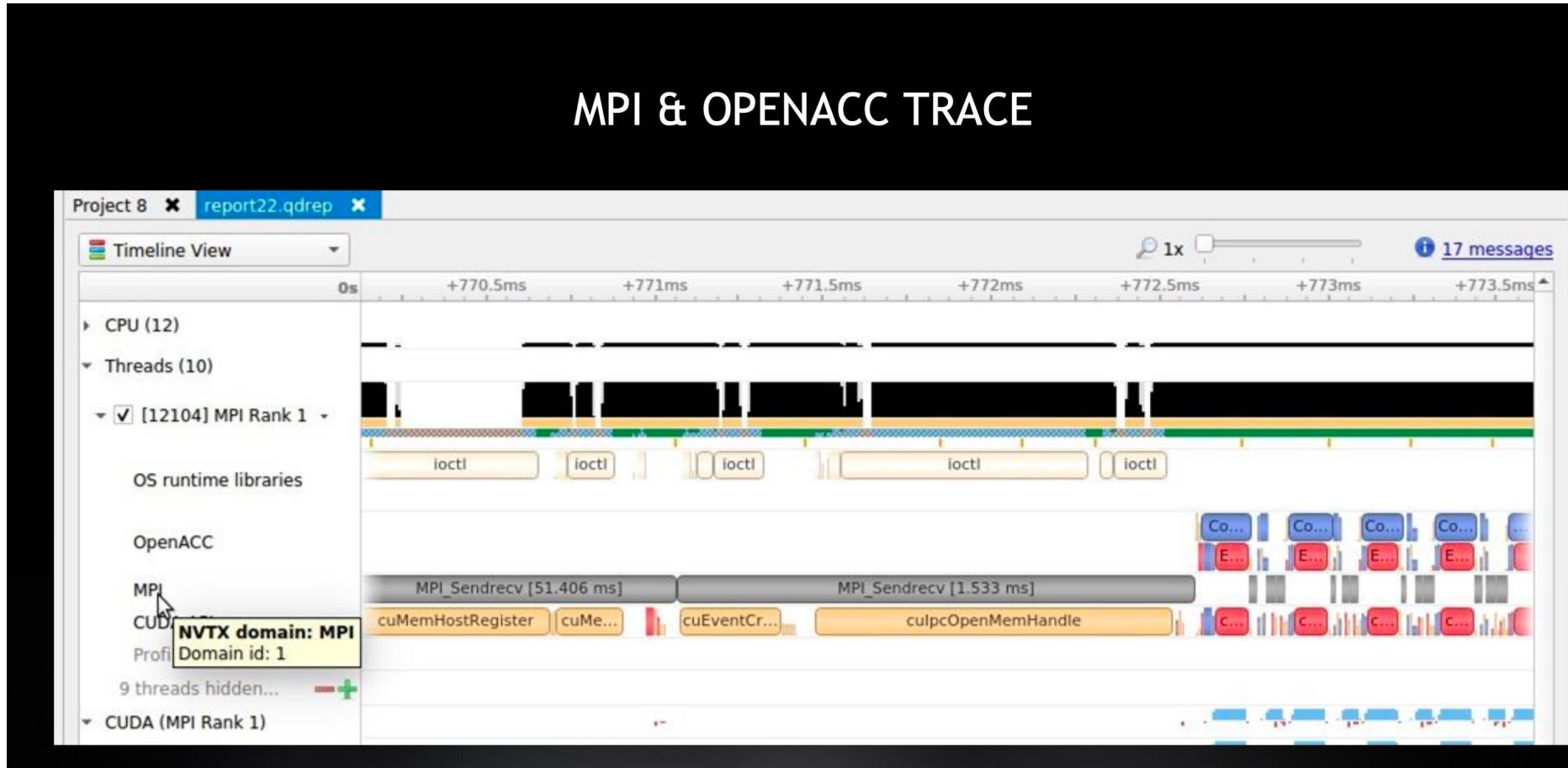
USER ANNOTATIONS APIs FOR CPU & GPU
NVTX, OPENGL, VULKAN, AND DIRECT3D PERFORMANCE MARKERS

What can be visualize with nsys

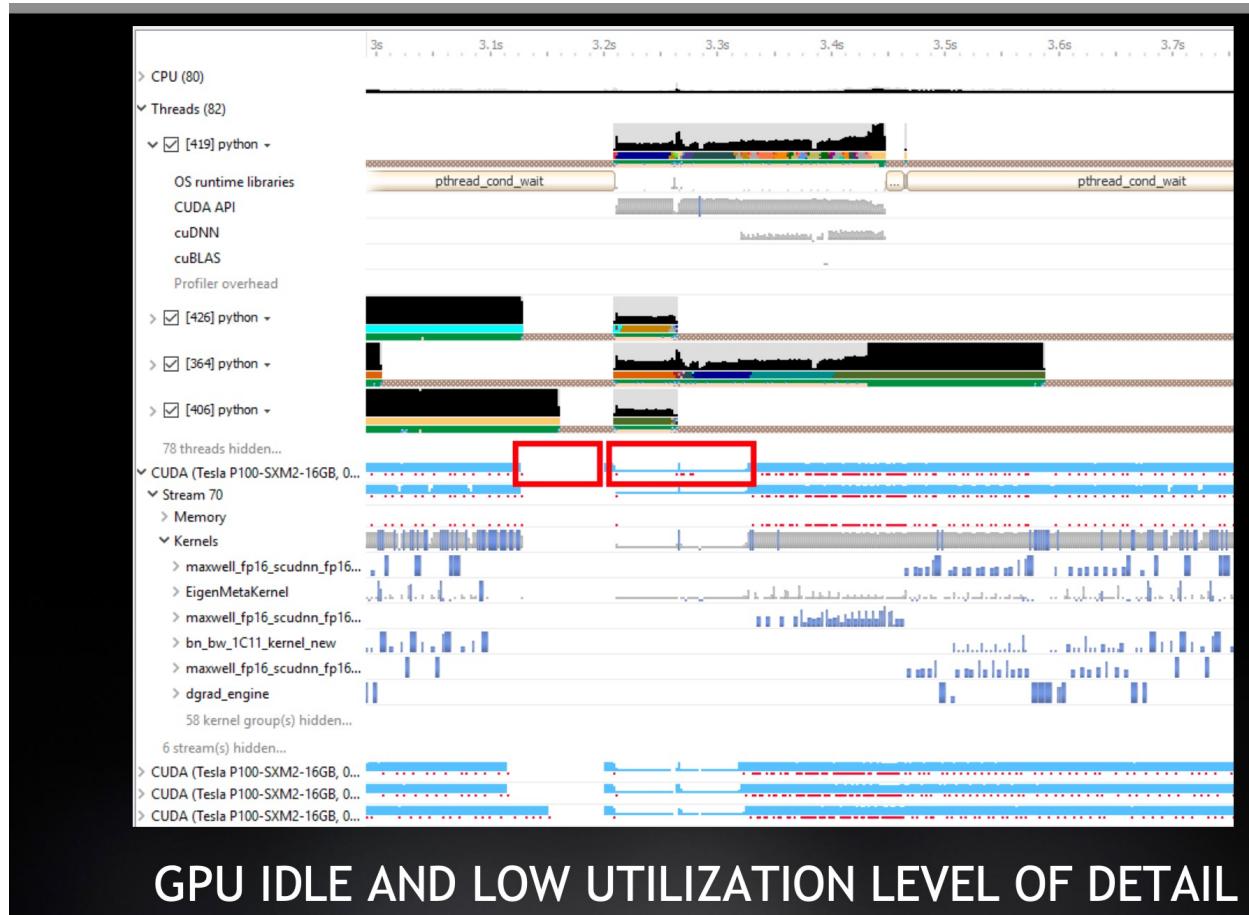


Source: Openhackaton NVIDIA GPU hackaton Princeton Spring 2023, Nsights

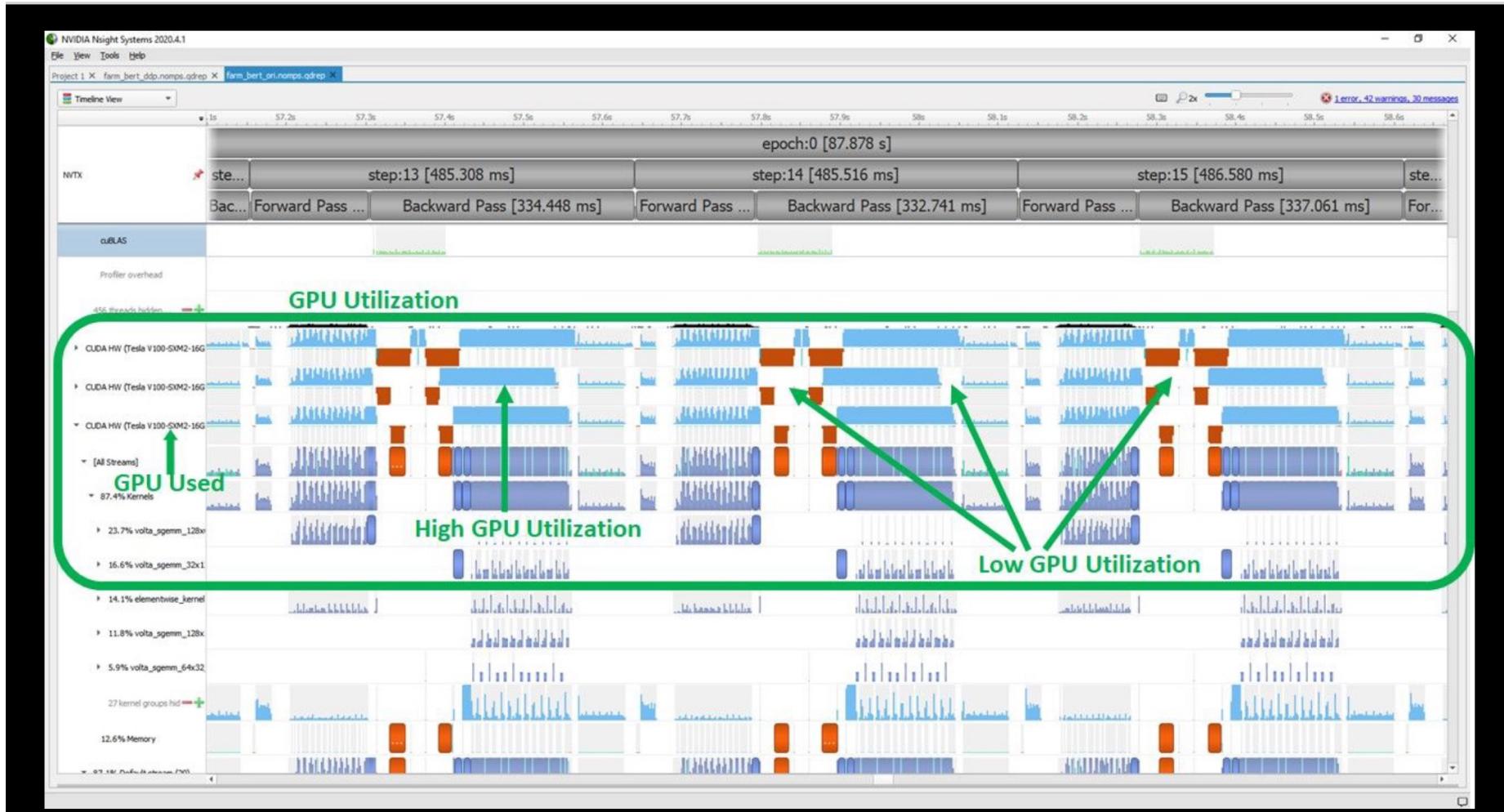
What can be visualize with nsys



What can be visualize with nsys



High vs low GPU utilization



Source: <https://alcf.anl.gov/sites/default/files/2022-07/Nsight%20Systems%20-2020.4.1%20GPU%20file%202024-07-2022%2011-10-20202020-06-22-16-16-16.pdf>

stats from algorithm

The screenshot shows a terminal window with the following content:

```
mozhgank@prm-dgx-32:~/Code/openacc-training-materials/labs/module4/English/c/solutions/parallel$ nsys profile -t nvtx --stats=true -b dwarf --force-overwrite true -o laplace-seq ./laplace-seq
Collecting data...
Jacobi relaxation Calculation: 4096 x 4096 mesh
 0, 0.250000
100, 0.002397
200, 0.001204
300, 0.000804
400, 0.000603
500, 0.000483
600, 0.000403
700, 0.000345
800, 0.000302
900, 0.000269
total: 55.754501 s
Processing events...
Capturing symbol files...
Saving intermediate "/home/mozhgank/code/openacc-training-materials/labs/module4/English/c/solutions/parallel/laplace-seq.qdstrm" file to disk...
Importing [=====100%]
Saved report file to "/home/mozhgank/Code/openacc-training-materials/labs/module4/English/c/solutions/parallel/laplace-seq.qdrep"
Exporting 70802 events: [=====100%]
Exported successfully to
/home/mozhgank/Code/openacc-training-materials/labs/module4/English/c/solutions/parallel/laplace-seq.sqlite
Generating NVTX Push-Pop Range Statistics...
NVTX Push-Pop Range Statistics (nanoseconds)
Time(%) Total Time Instances Average Minimum Maximum Range
49.9 55754497966 1 55754497966.0 55754497966 55754497966 while
26.5 29577817696 1000 29577817.7 29092956 65008545 calc
23.4 26163892482 1000 26163892.5 25761418 60129514 swap
0.1 137489808 1 137489808.0 137489808 137489808 init
```

Annotations in the terminal output:

- t Selects the APIs to be traced (nvtx in this example)
- status if true, generates summary of statistics after the collection
- b Selects the backtrace method to use while sampling. The option dwarf uses DWARF's CFI (Call Frame Information).
- force-overwrite if true, overwrites the existing results
- o sets the output (qdrep) filename

Annotations on the NVTX range statistics table:

- "calc" region (calcNext function) takes 26.6%
- "swap" region (swap function) takes 23.4% of total execution time
- NVTX range statistics

Annotations at the bottom right:

- Open laplace-seq.qdrep with Nsight System GUI to view the timeline

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.