PARALLEL AND GPU PROGRAMMING IN PYTHON

PRACE Training Course

Ben Czaja, Sagar Dolas, Mohsen Safari
HPC advisors, SURF

Outline

- Using GPUs in Python
- CUDA Programming Model
- CUDA Execution Model
- Examples:
  - Vector (1D array) Addition
  - Matrix (2D array) Addition
  - Matrix Multiplication
- Optimization Tips

Accessing to GPUs in Python

PyTorch

PyCUDA Numba CuPy TensorFlow

Low-level High-level

PyCUDA gives you easy, Pythonic access to NVIDIA's CUDA parallel computation API.
https://documen.tician.de/pycuda/

CUDA Programming Model

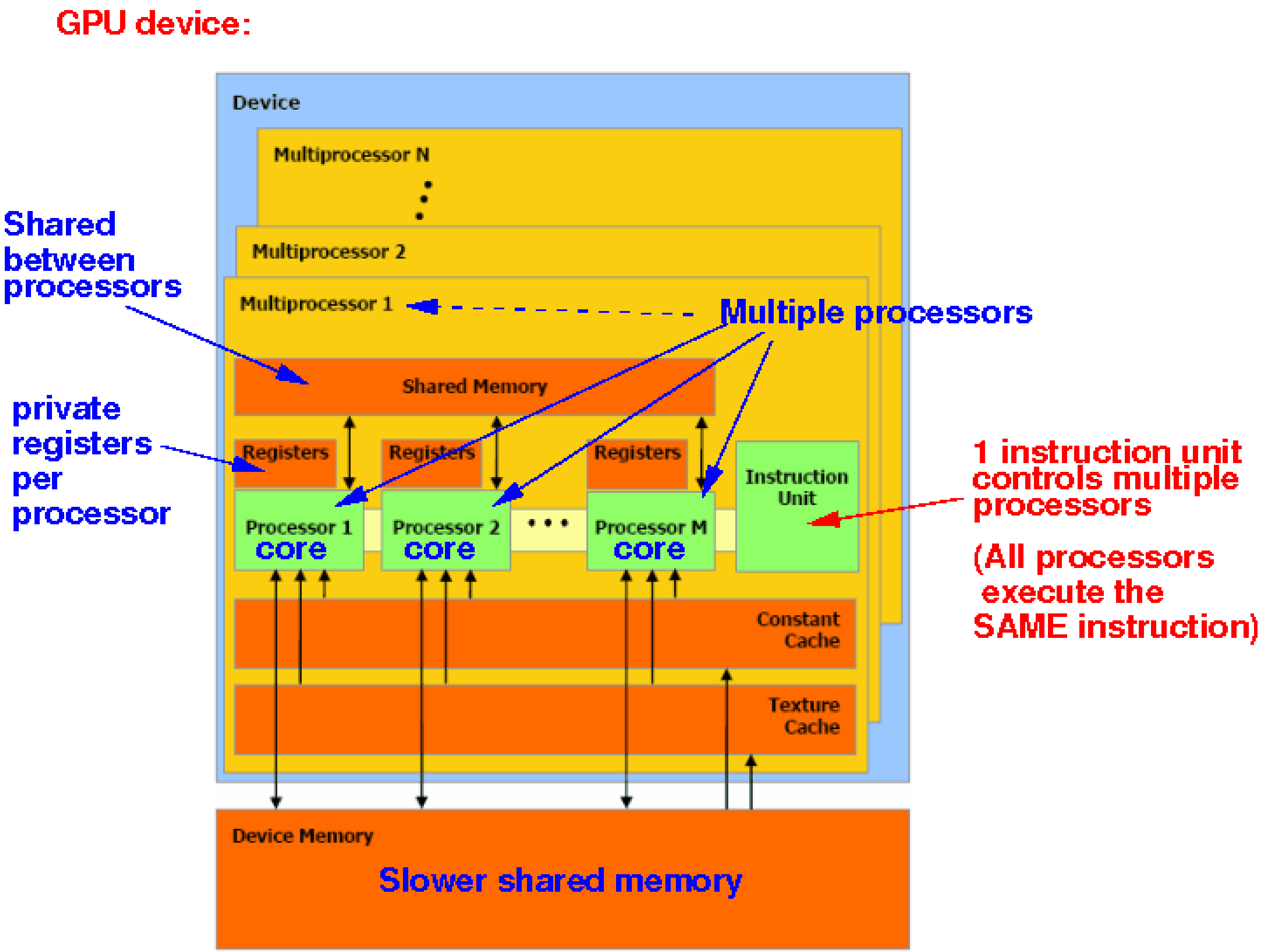
- Introduced by NVIDIA in 2006, Compute Unified Device Architecture
- General purpose programming model that leverages the parallel compute engine in NVIDIA GPUs
- An extension of C language
- CUDA programs are CPU-GPU programs:
  - CPU part is called *host*
  - GPU part is called *kernel*

CUDA Programming Model

To execute any CUDA program, there are three main steps:

- Copy the input data from host memory to device memory, also known as host-to-device transfer
- Call the kernel from host and execute the GPU program
- Copy the results from device memory to host memory, also called device-to-host transfer

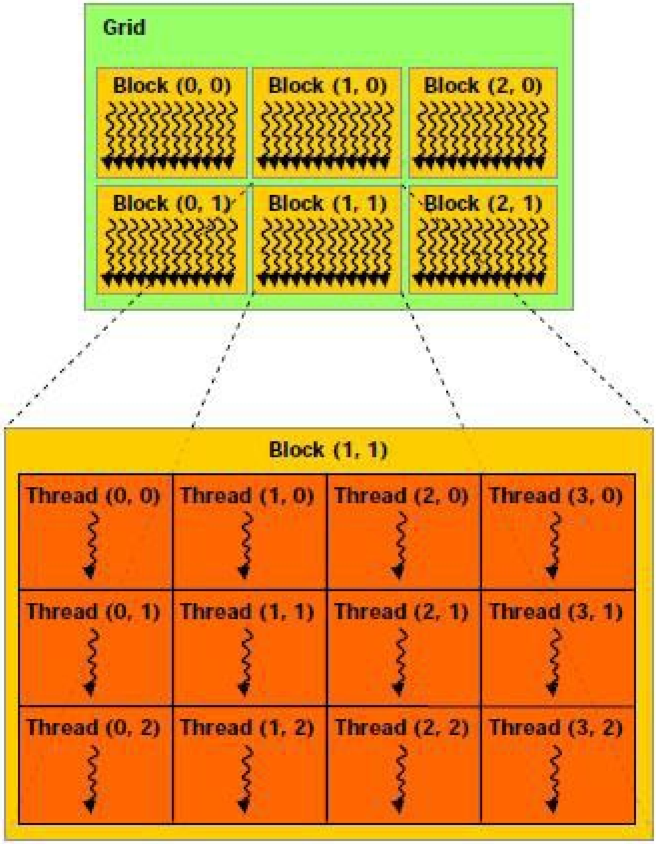
NVIDIA GPU Hardware



CUDA Programming Model

- Threads are organized into two hierarchical levels:
  - Threads are grouped into *blocks*
  - Blocks are grouped into *grids*

- Blocks and grids can be 1D, 2D and 3D

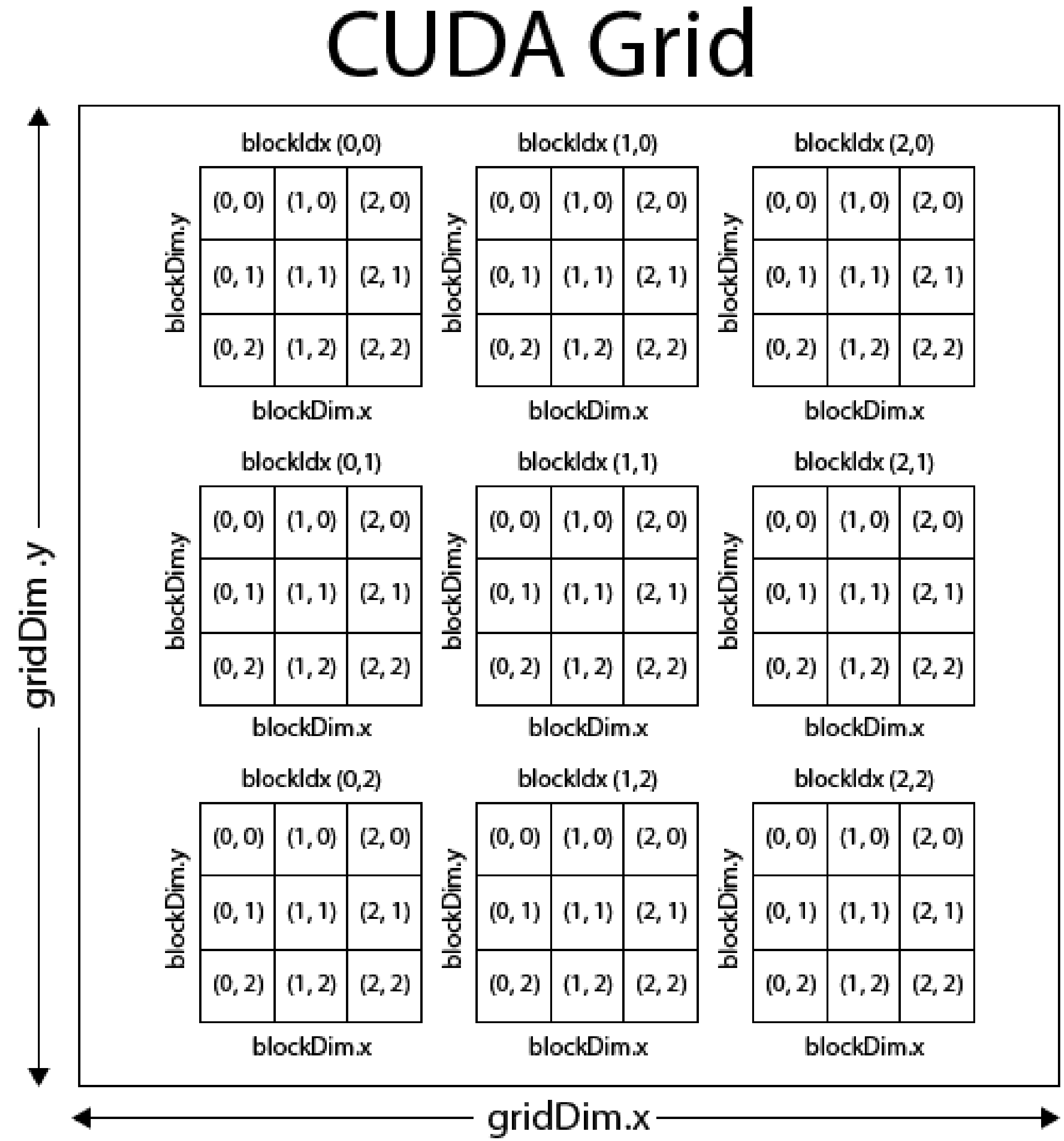


CUDA Programming Model

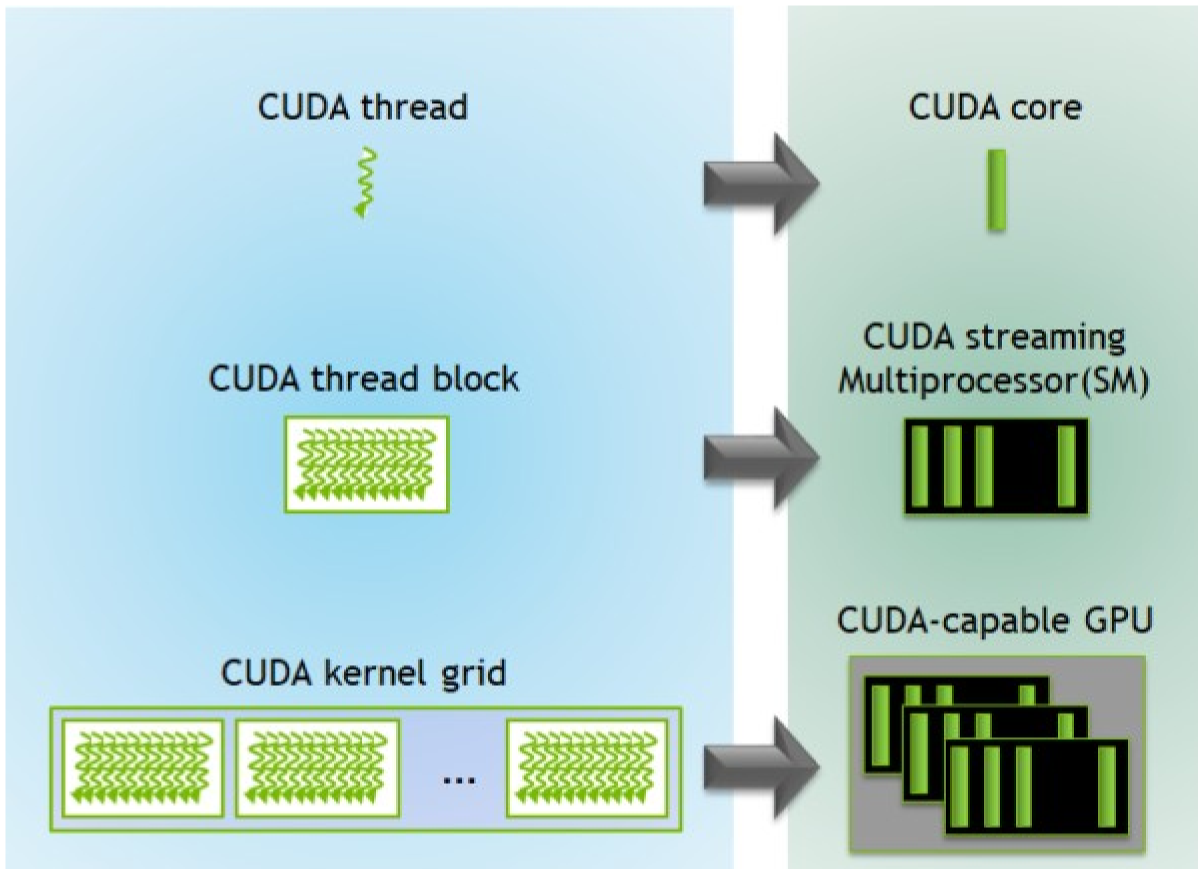
- Built-in functions:
  - Dimension:
    - gridDim.x, gridDim.y, gridDim.z
    - blockDim.x, blockDim.y, blockDim.z
  - Index:
    - blockIdx.x, blockIdx.y, blockIdx.z
    - threadIdx.x, threadIdx.y, threadIdx.z

CUDA Programming Model

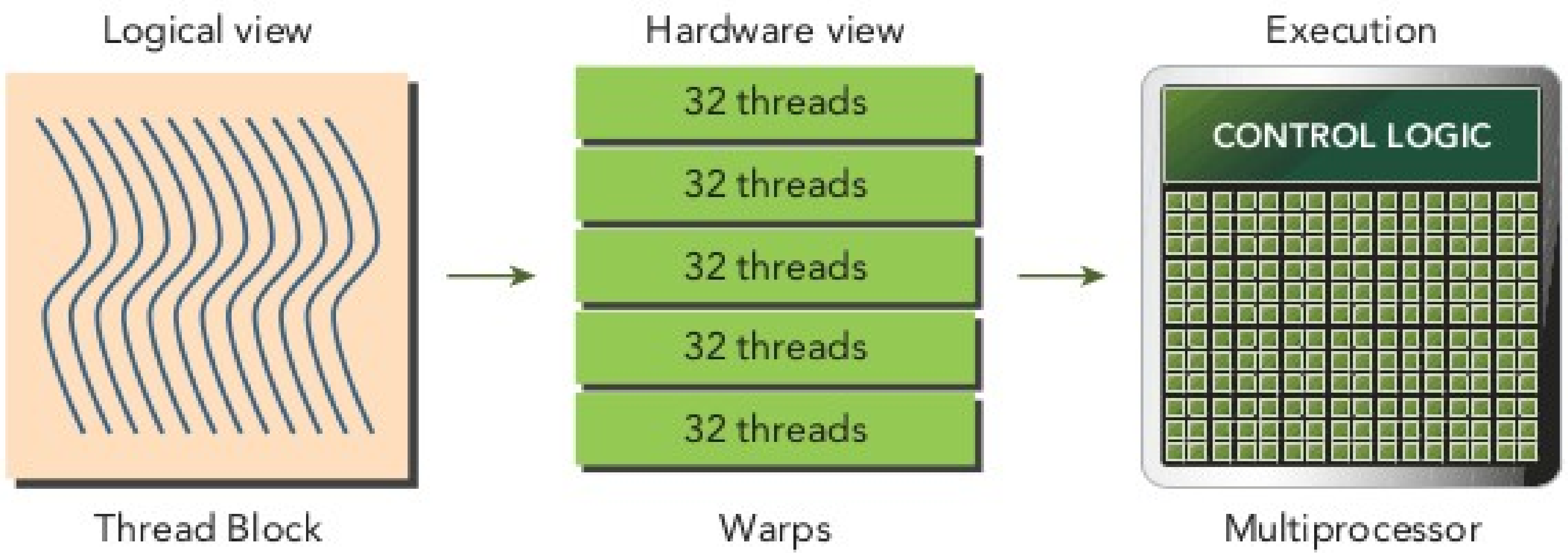
- gridDim.x = 3
- gridDim.y = 3
- blockDim.x = 3
- blockDim.y = 3



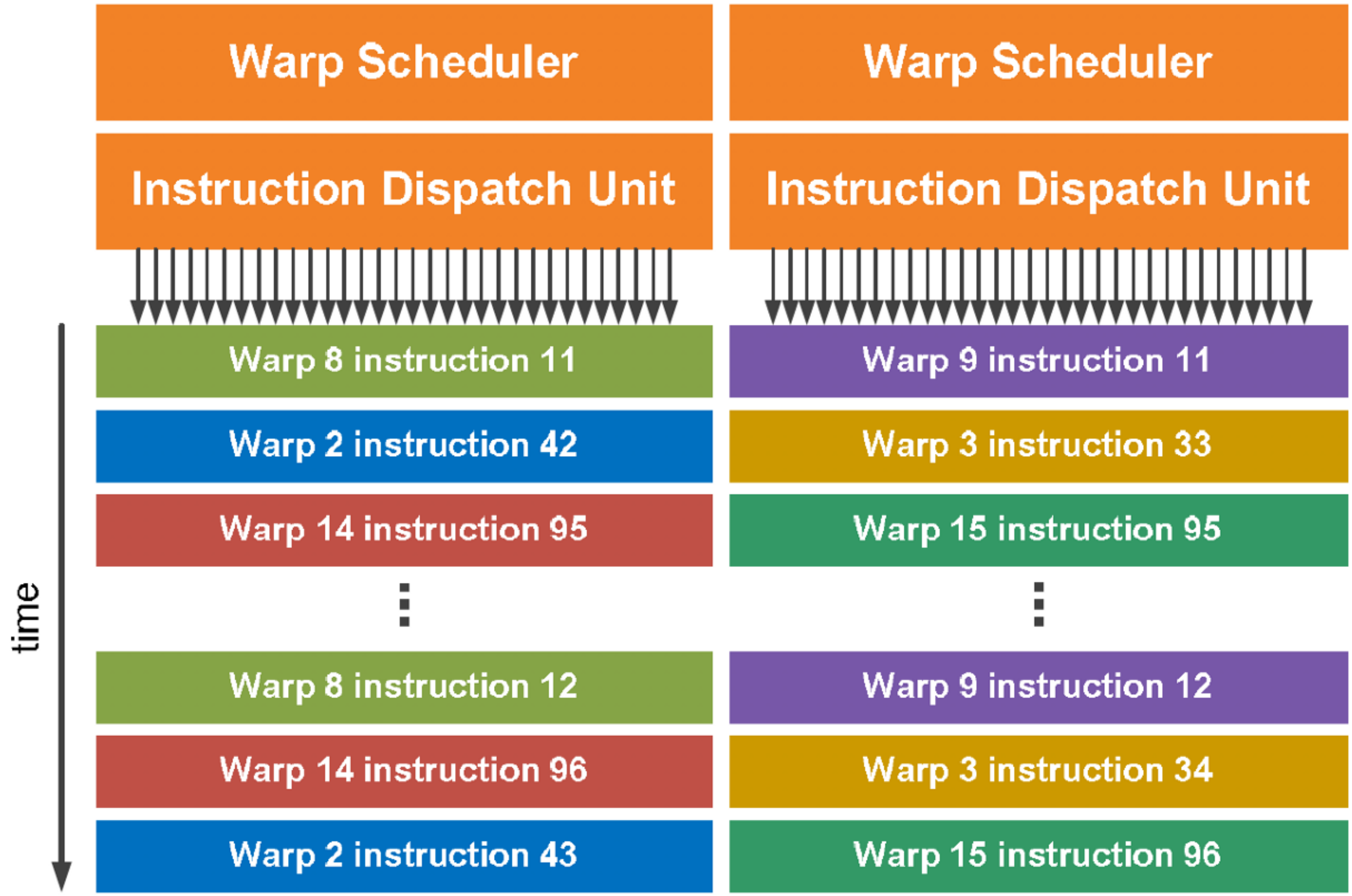
CUDA Execution Model



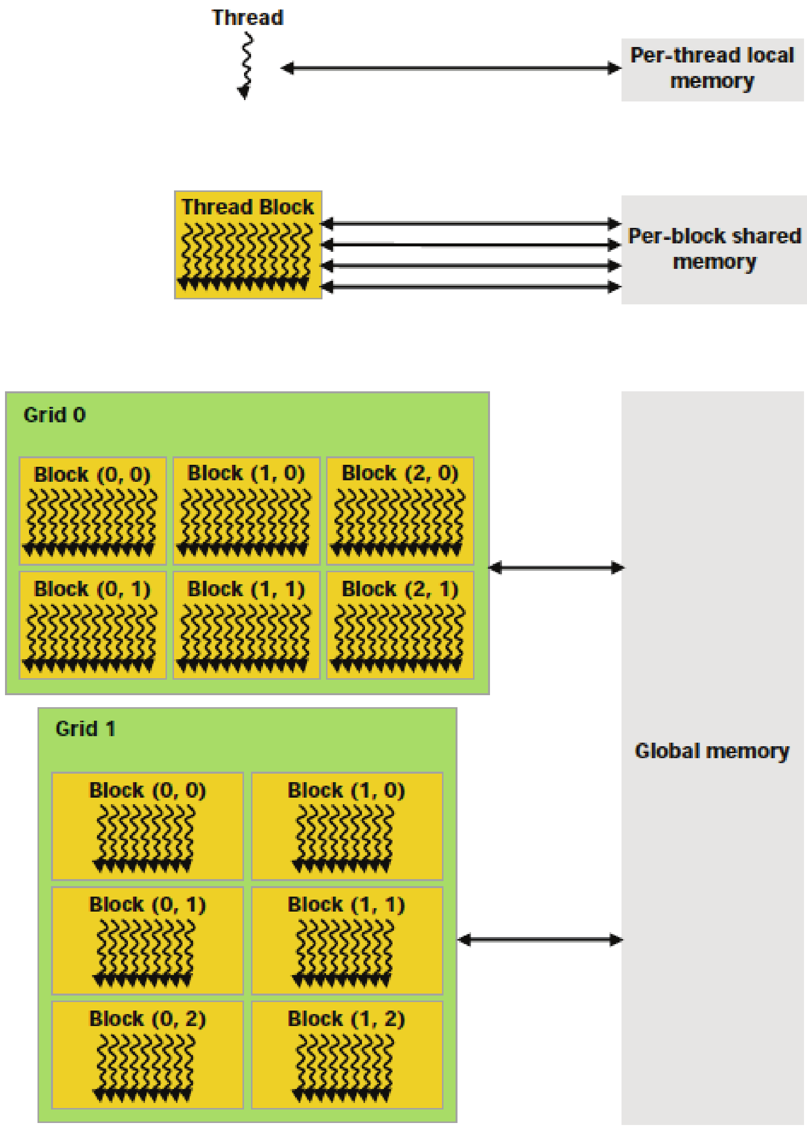
CUDA Execution Model



CUDA Execution Model



CUDA Execution Model



Synchronization in CUDA

- There is a mechanism to synchronize all threads in a block:
    - Built-in function __syncthreads()
- There is no mechanism to synchronize all threads across all blocks

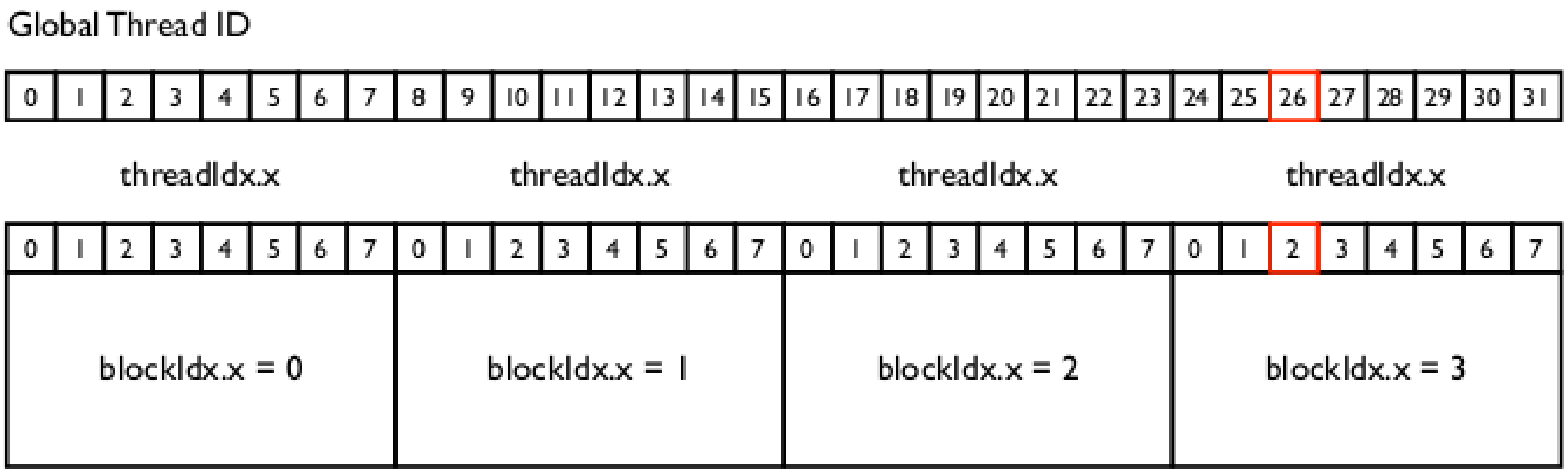
GPU Node

- 4 NVIDIA Titan RTX GPUs per node
  - Multiprocessors: 72
  - Steaming cores: 4608
  - Global memory: 24 GB
- One node is shared among 16 people
- One GPU is shared among 4 people
- Note that you have around 5 GB memory:
  - Matrix (35,000 * 35,000) = 35,000*35,000*4 ≈ 5 GB
  -

First Example:

Parallel Vector (1D array) Addition in PyCUDA

Calculate Global Index (1D grid, 1D block)



- Global Thread ID: blockIdx.x * blockDim.x + threadIdx.x
- For global thread ID 26:
  - blockIdx.x = 3
  - blockDim.x = 8
  - threadIdx.x = 2
  - Global thread ID = 3 * 8 + 2 = 26

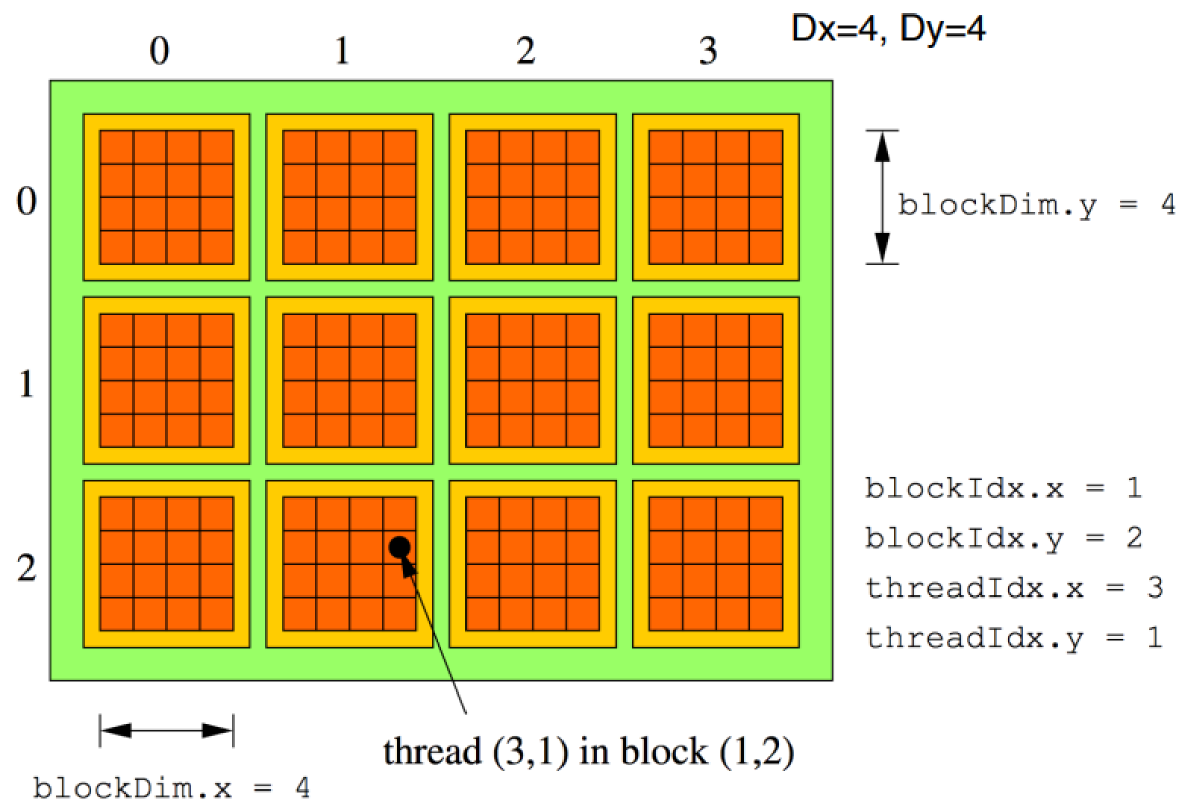
Automatic Data Transfer

- Automatic data transfer using PyCUDA driver:
  - In()
  - Out()
  - InOut()
- PyCUDA programs become simpler

Second Example:

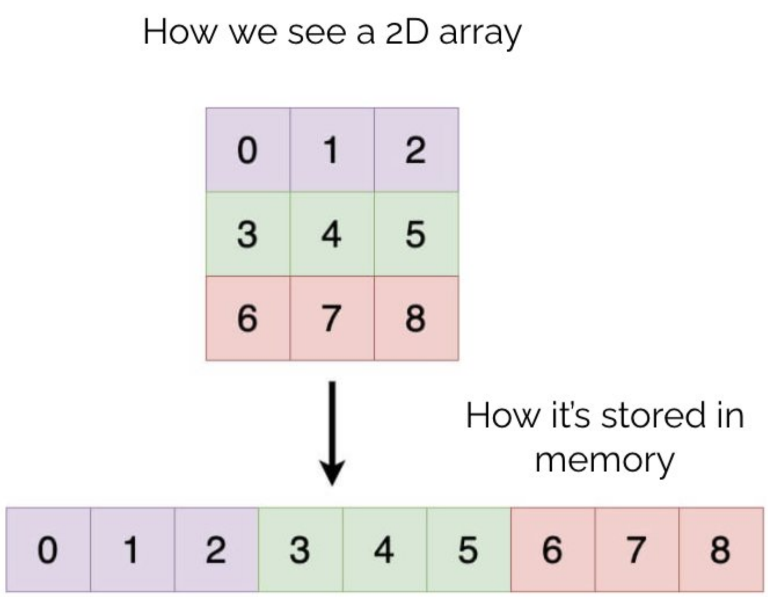
Parallel Matrix (2D array) Addition in PyCUDA

Calculate Global Index (2D grid, 2D block)



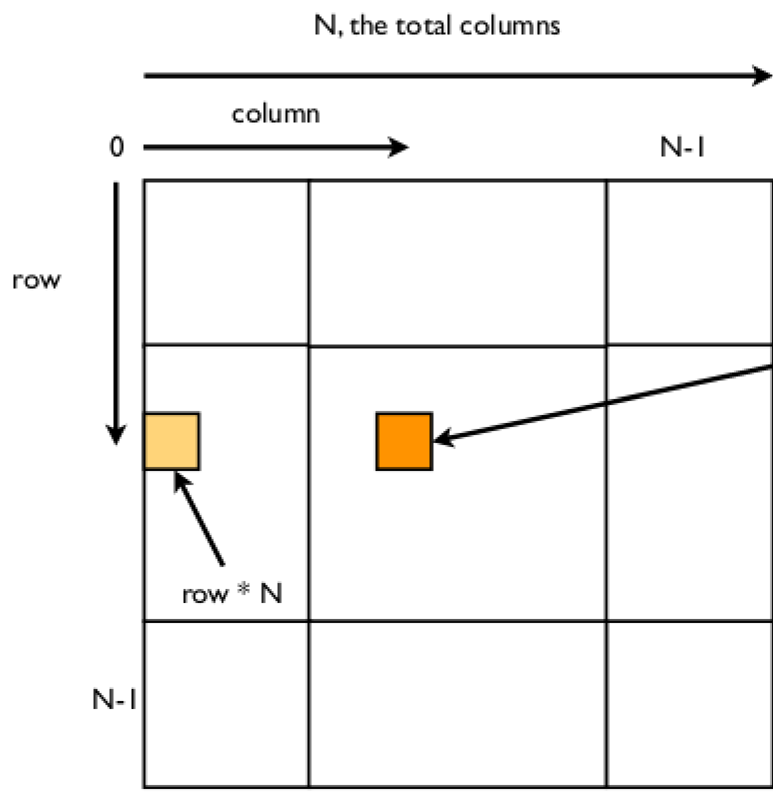
- Matrix 12*16
- Global Thread ID:
  - row = blockIdx.y * blockDim.y + threadIdx.y = 2 * 4 + 1 = 9
  - column = blockIdx.x * blockDim.x + threadIdx.x = 1 * 4 + 3 = 7

Row-Major Flattening of a Matrix



- Matrix 3*3
- For each element (row, col):
  - New ID = row * (No of col) + col
- For instance element “5” in location (1, 2):
  - New ID = 1 * 3 + 2 = 5

Row-Major Flattening of a Matrix



a[row][column] = a[offset]

offset = column + row * N

<u>In CUDA:</u>

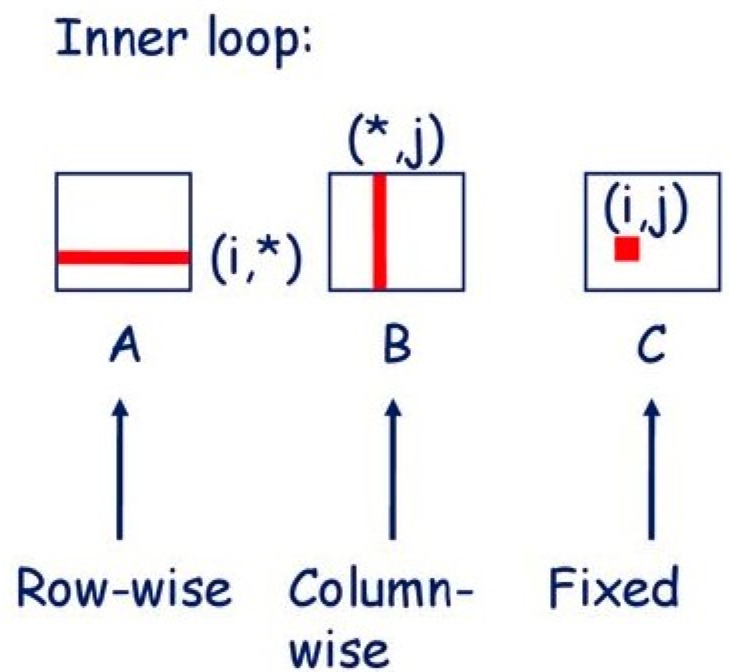
```
int col = blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int index = col + row * N;
A[index] = …
```

Third Example:

Parallel Matrix (2D array) Multiplication in PyCUDA

Sequential Matrix Multiplication

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



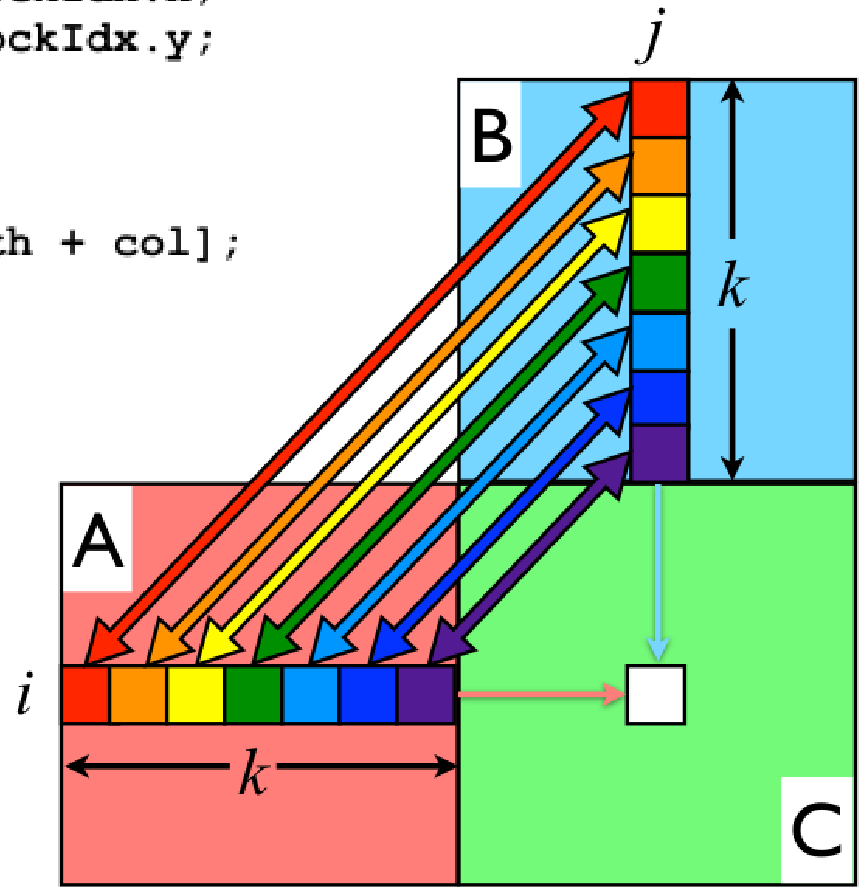
$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}_{C}$$

Parallel Matrix Multiplication

```
int k, sum = 0;

int col = threadIdx.x + blockDim.x * blockIdx.x;
int row = threadIdx.y + blockDim.y * blockIdx.y;

if(col < width && row < width) {
 for (k = 0; k < width; k++)
  sum += a[row * width + k] * b[k * width + col];
  c[row * width + col] = sum;
}
```

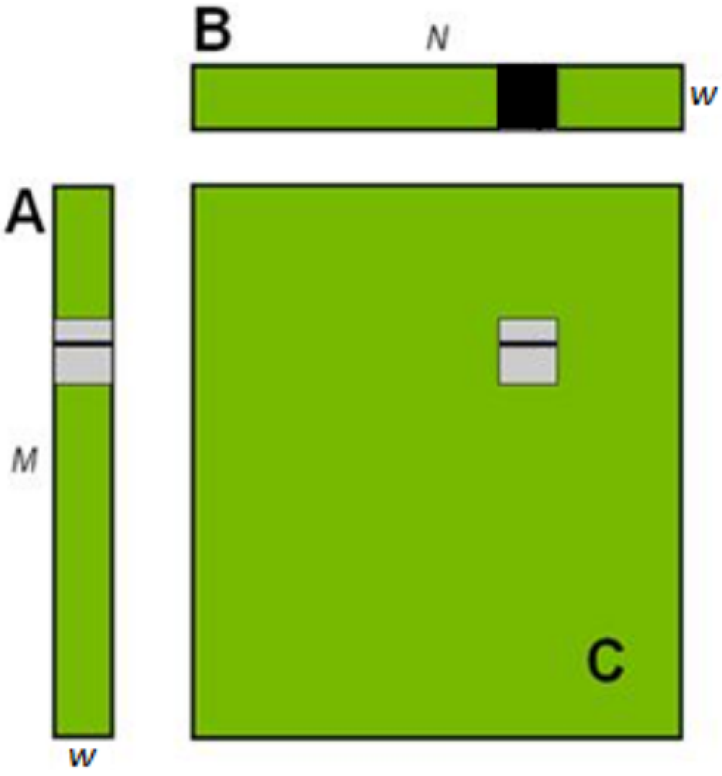


Time comparison

- Time comparison between
    - PyCUDA
    - Numpy.matmul()
    - @ operator

Further Optimization

- Matrix A: M*W
- Matrix B: W*N
- Matrix C: M*N
- Assume W = 32
- Assume blockDim.x = 32
- Assume blockDim.y = 32



Exercise 1

```
__shared__ int a_shared[32][32];
__shared__ int b_shared[32][32];
a_shared[threadIdx.y][threadIdx.x] = A[...];
b_shared[threadIdx.y][threadIdx.x] = B[...];
__syncthreads();
For (int k = 0; k < 32; k++)
    temp += a_shared[threadIdx.y][k] * b_shared[k][threadIdx.x];
C[id] = temp;
```

Exercise 2

- How to extend it to any size W?
- How is the performance now?

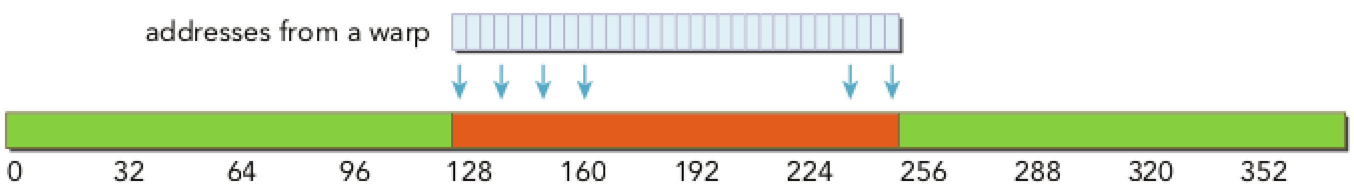
Optimization

There are different ways to optimize CUDA codes:

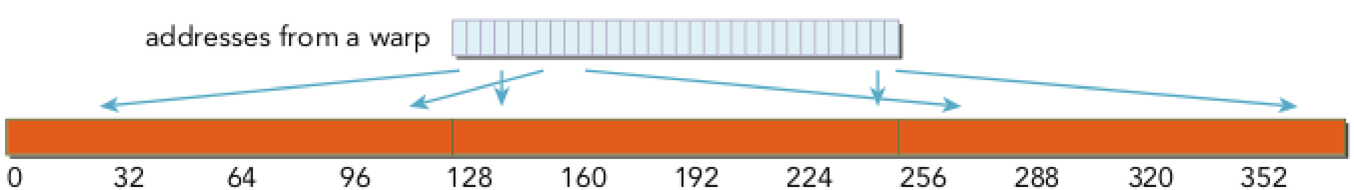
- Number of threads per block
- Workload per thread
- Total work per thread block
- Correct memory access and data locality
- …

Tips for Optimization

- Global Memory Access:



- Coalesced

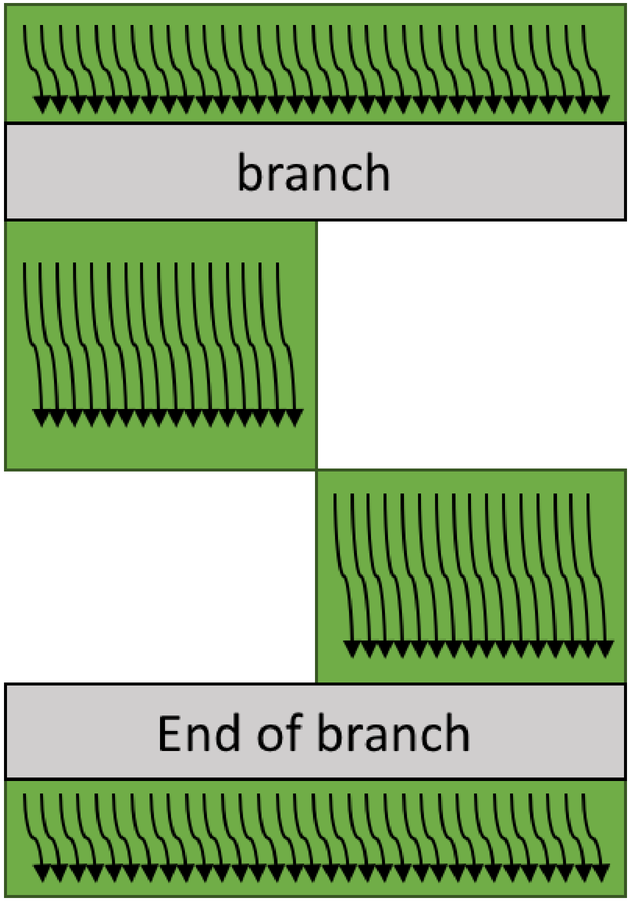


- Non-coalesced

Tips for Optimization

- Avoid Warp Divergence:

```
...
if ( threadIdx.x < 16 )
{
    ... A ...
}
else
{
    ... B ...
}
...
```



Tips for Optimization

- Use shared memory in two cases:
    - When threads in a block need to share data
    - When there are repeated accesses to one location in global memory
        - In this case, it is possible to use register as local memory to each thread

Questions

Thank you for participating! Any questions?