PARALLEL AND GPU PROGRAMMING IN PYTHON

Mohsen Safari

HPC Advisors, SURF

January 23, 2025

Amsterdam Science Park

Outline

- GPUs as hardware accelerator
- PyCUDA programming
  - CUDA programming and execution model
- Examples:
  - Vector (1D array) addition
  - Matrix (2D array) addition
  - Matrix multiplication
  - Reduction
- Optimization tips
- Two bugs in GPU programming

Resources

- The slides and source code of the examples can be found at:
  - https://github.com/sara-nl/Parallel-and-GPU-programming-in-Python/tree/main/Day2

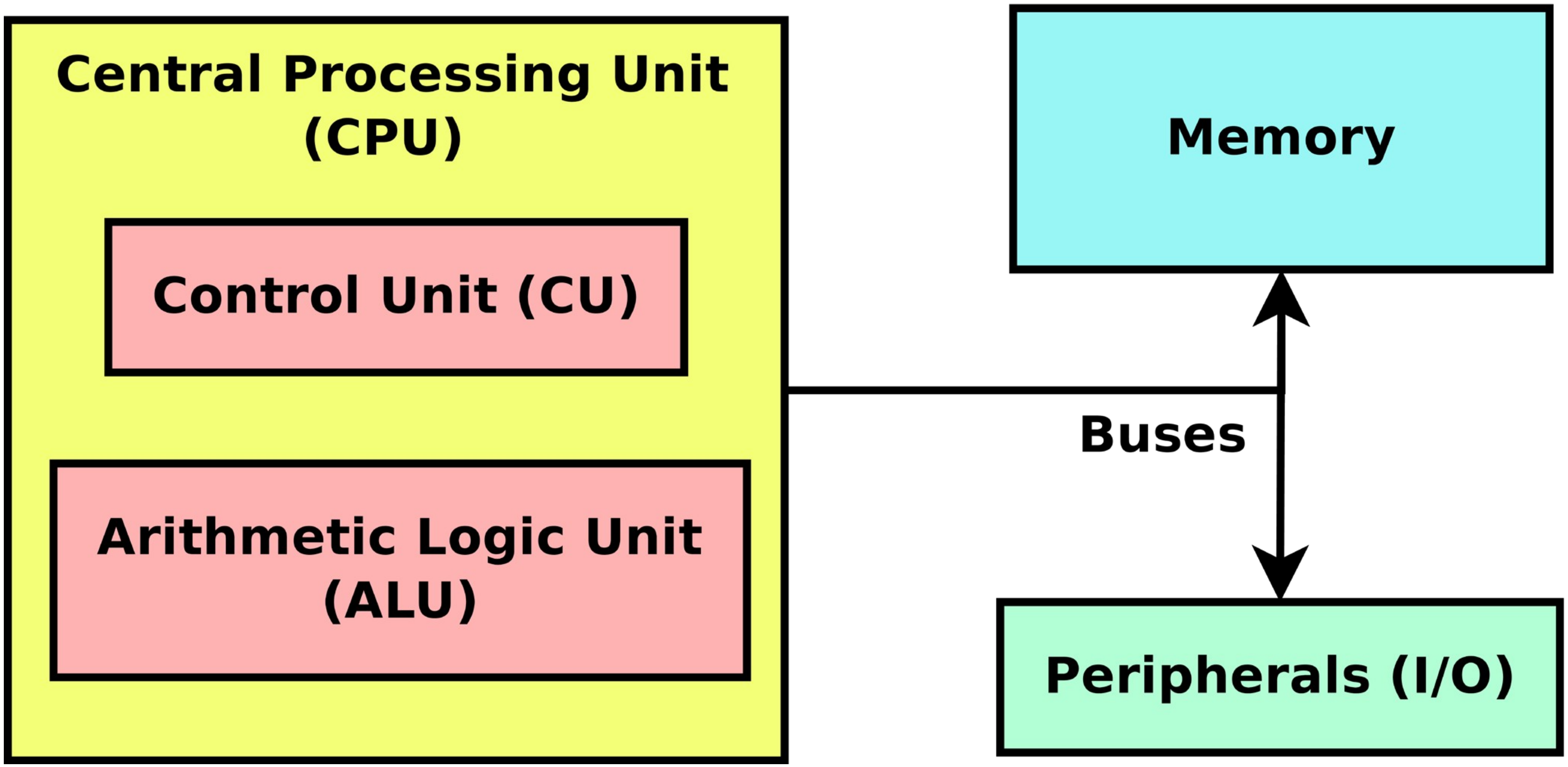
Jupyter Notebook

- To reset your PASSWORD:
  - https://portal.cua.surf.nl
- The Jupyter-hub for the GPU part of the course:
  - https://jupyter.snellius.surf.nl/jhssrf019

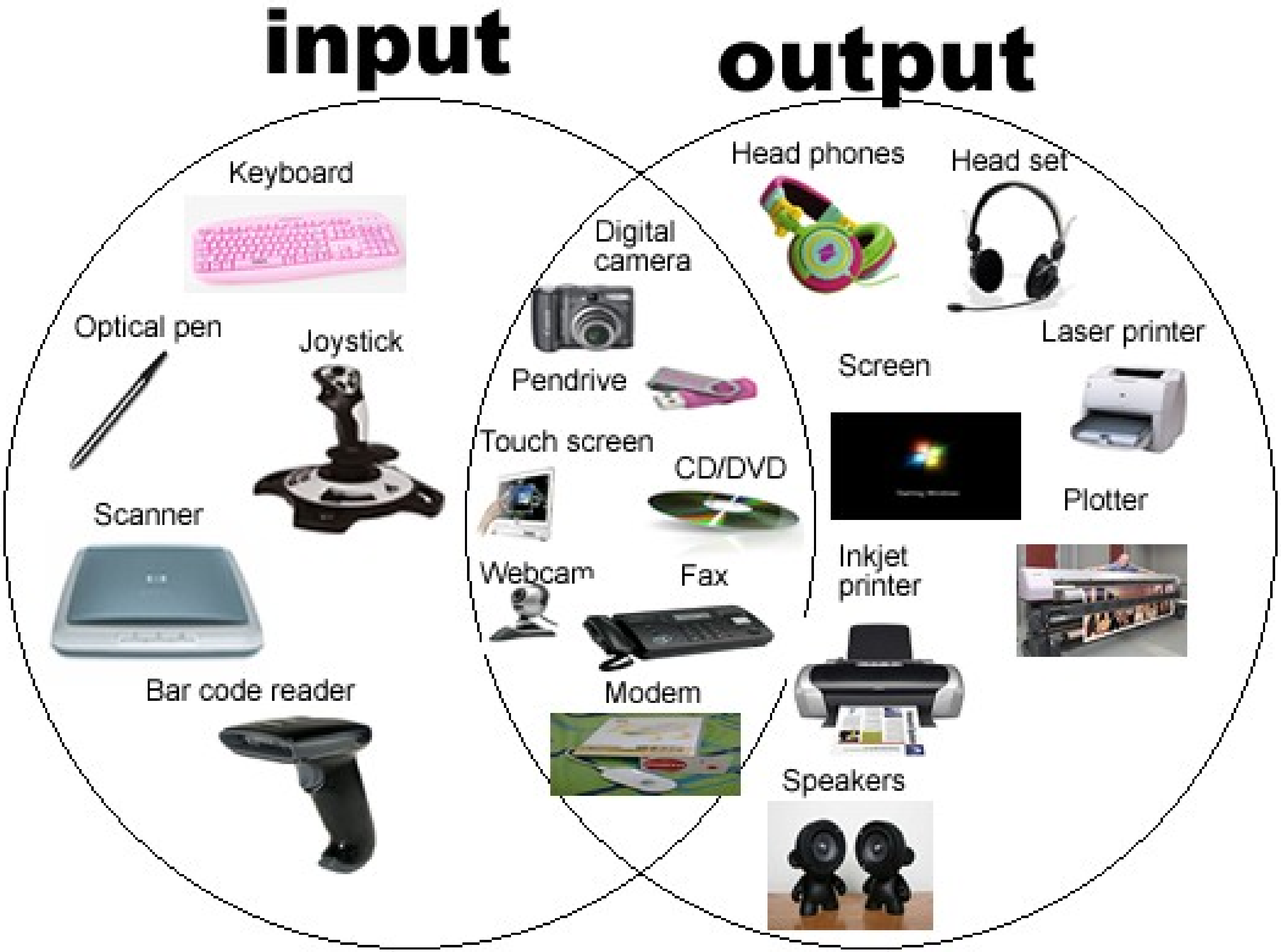
Hardware Accelerator (e.g., GPUs)

- What is it?
- Why do we need it?
- How to benefit it?
- ...

A computer is



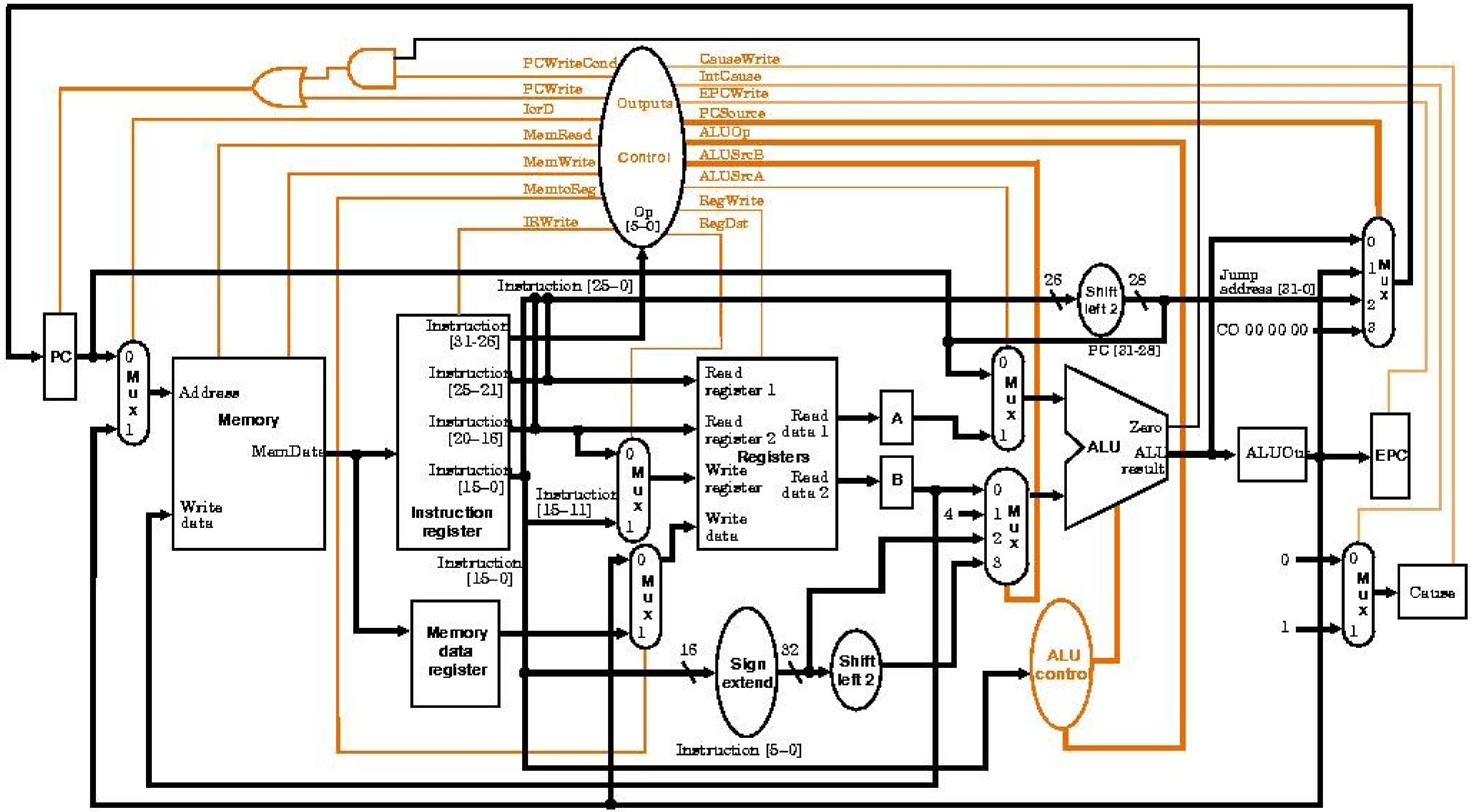
Peripherals



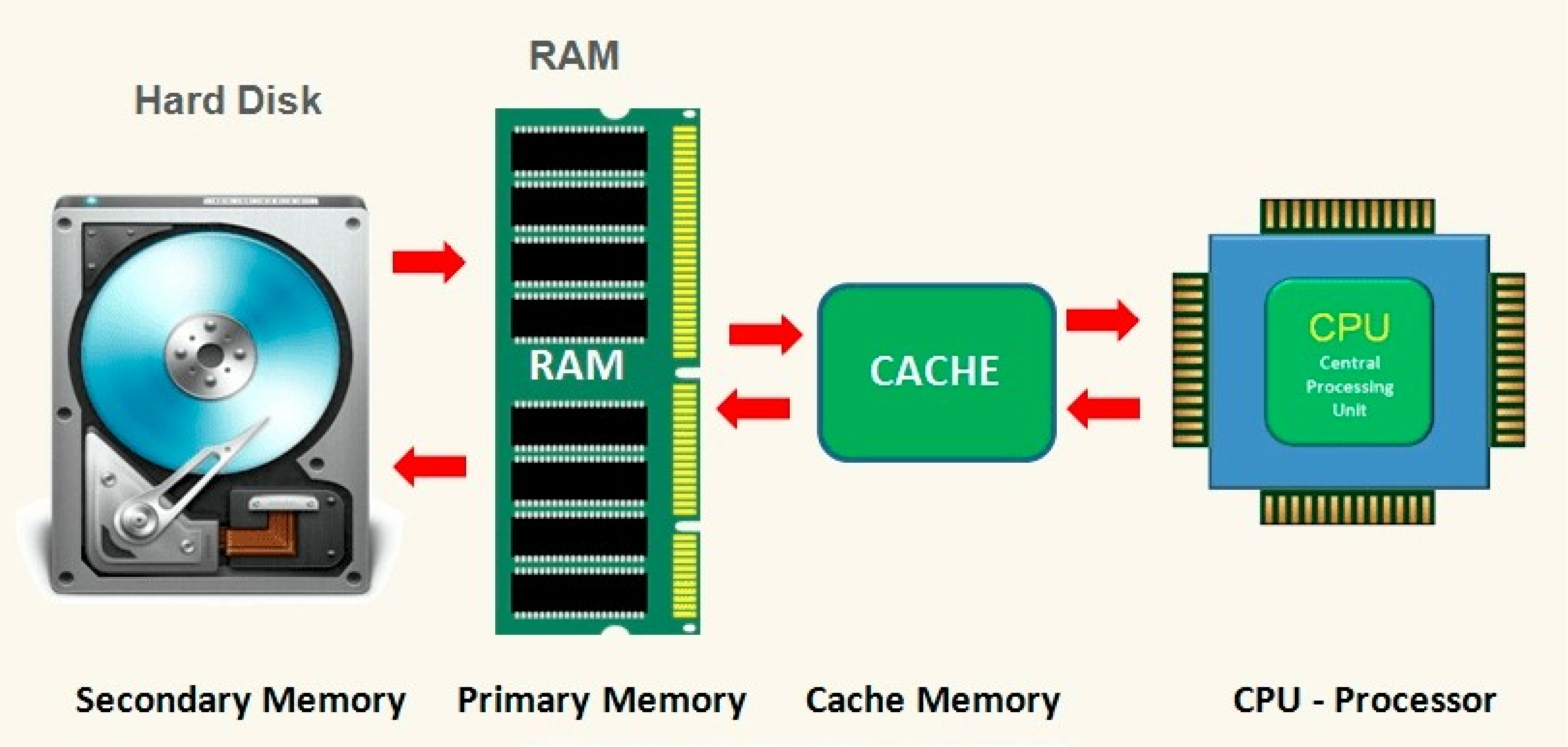
Main Gloals

- General-Purpose
- Low latency

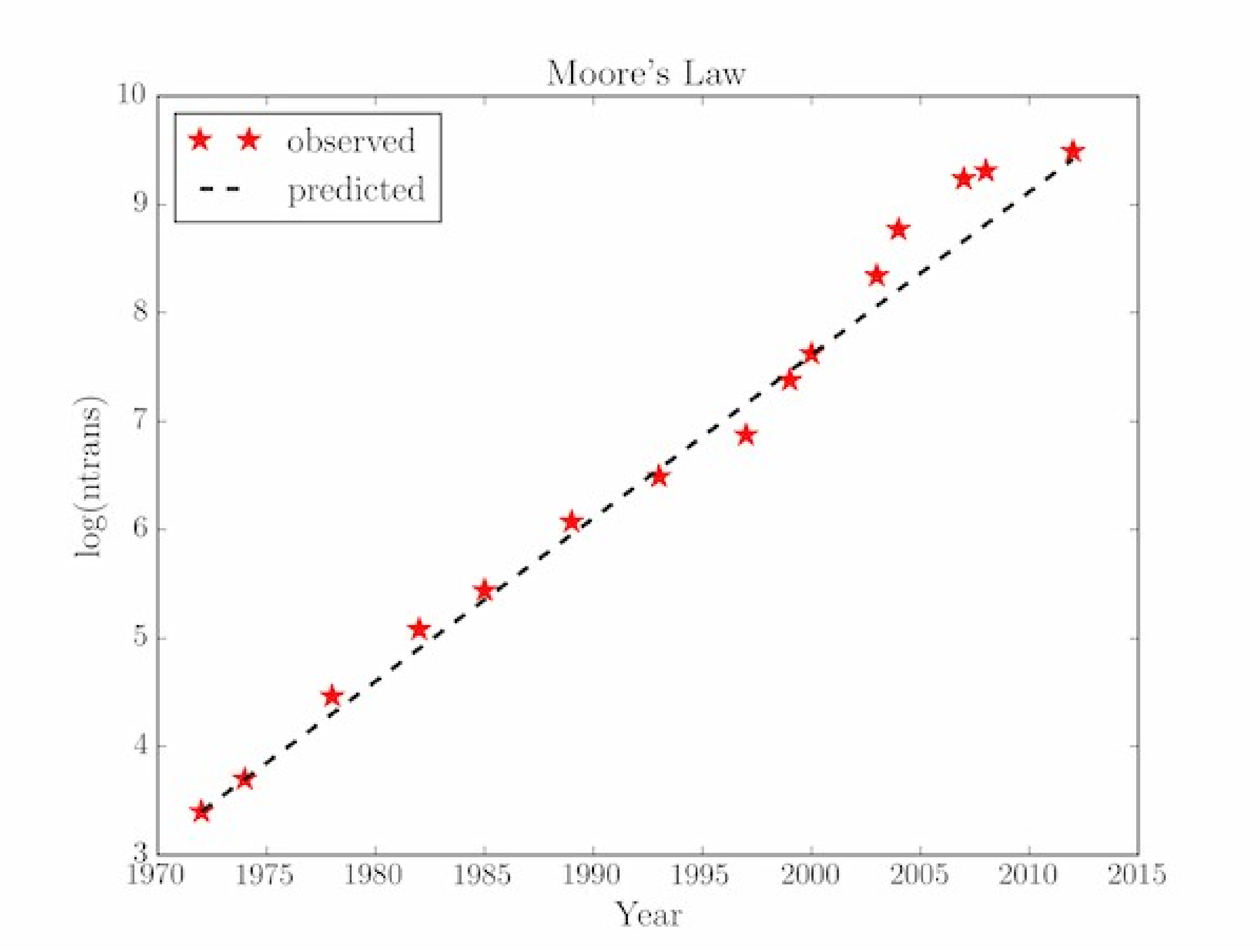
Complicated CPUs



Memory Hierarchy



Moore’s Law



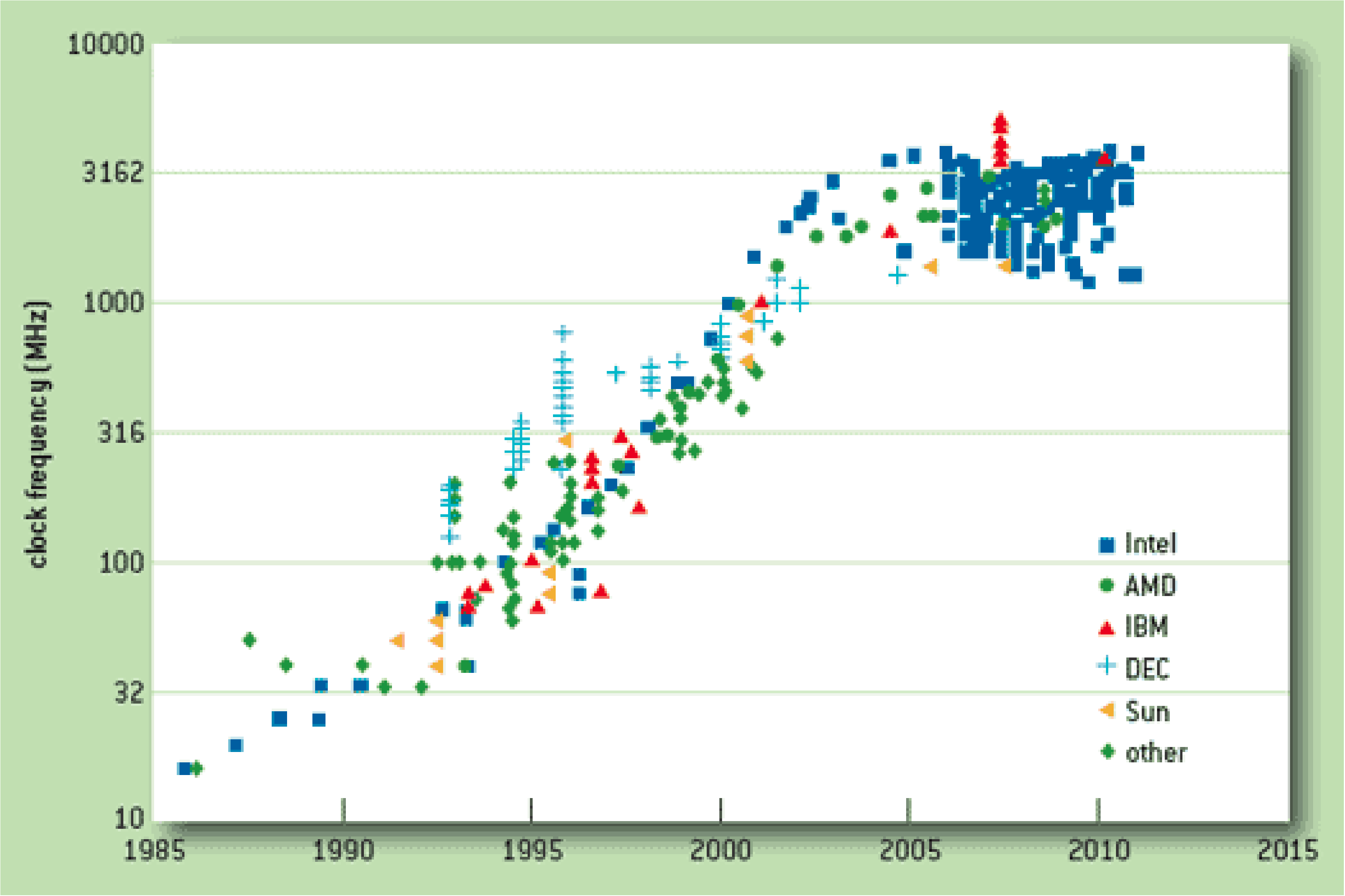
Number of transistors on a CPU chip will double every 18 months

Dennard Scaling Law

- As transistors become smaller, their power density stays constant

- As a result of Moore’s and Dennard’s law:
  - **CPU manufacturers can raise clock frequency without significantly increasing overall circuit power consumption**

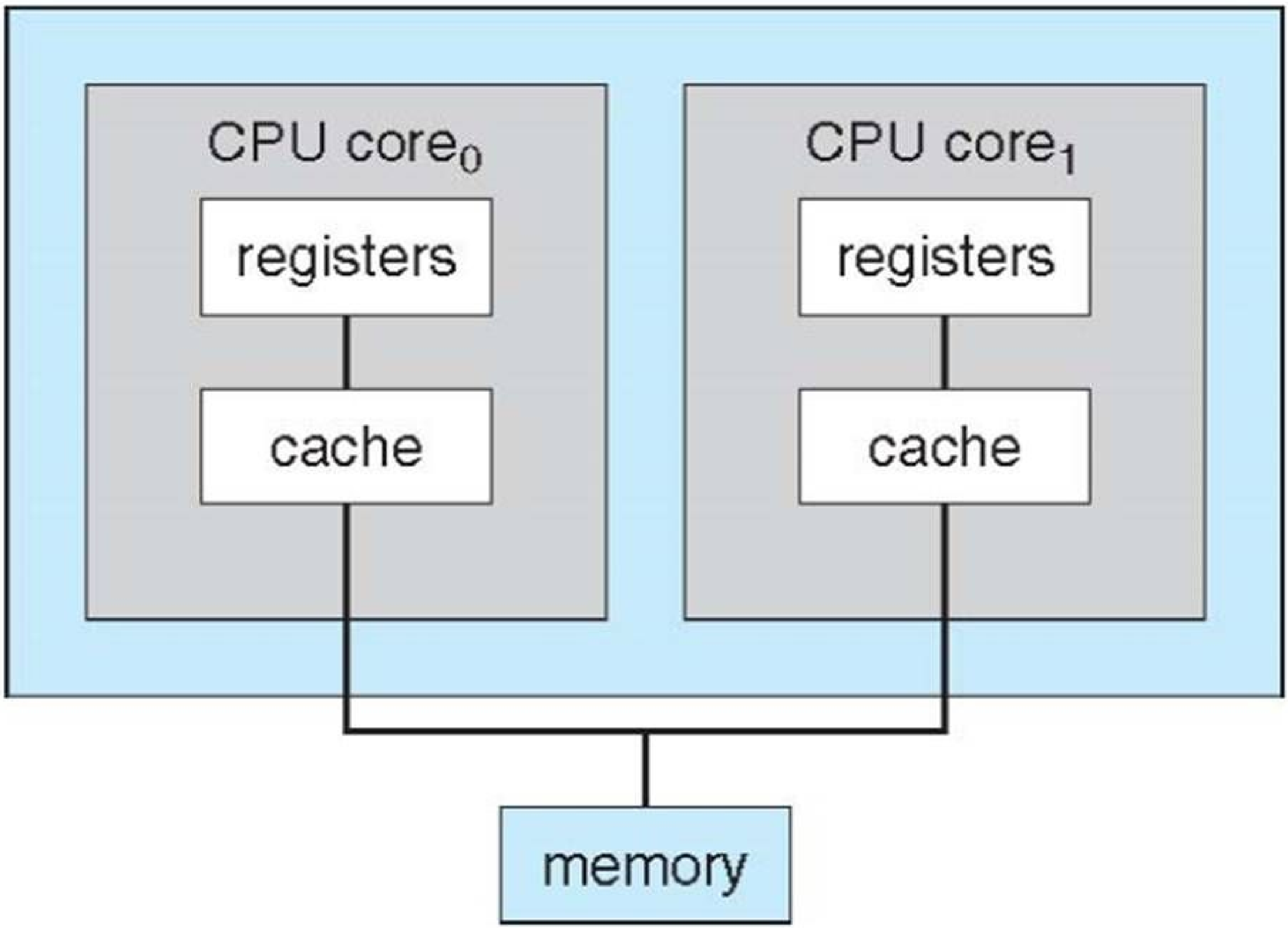
Clock Frequency



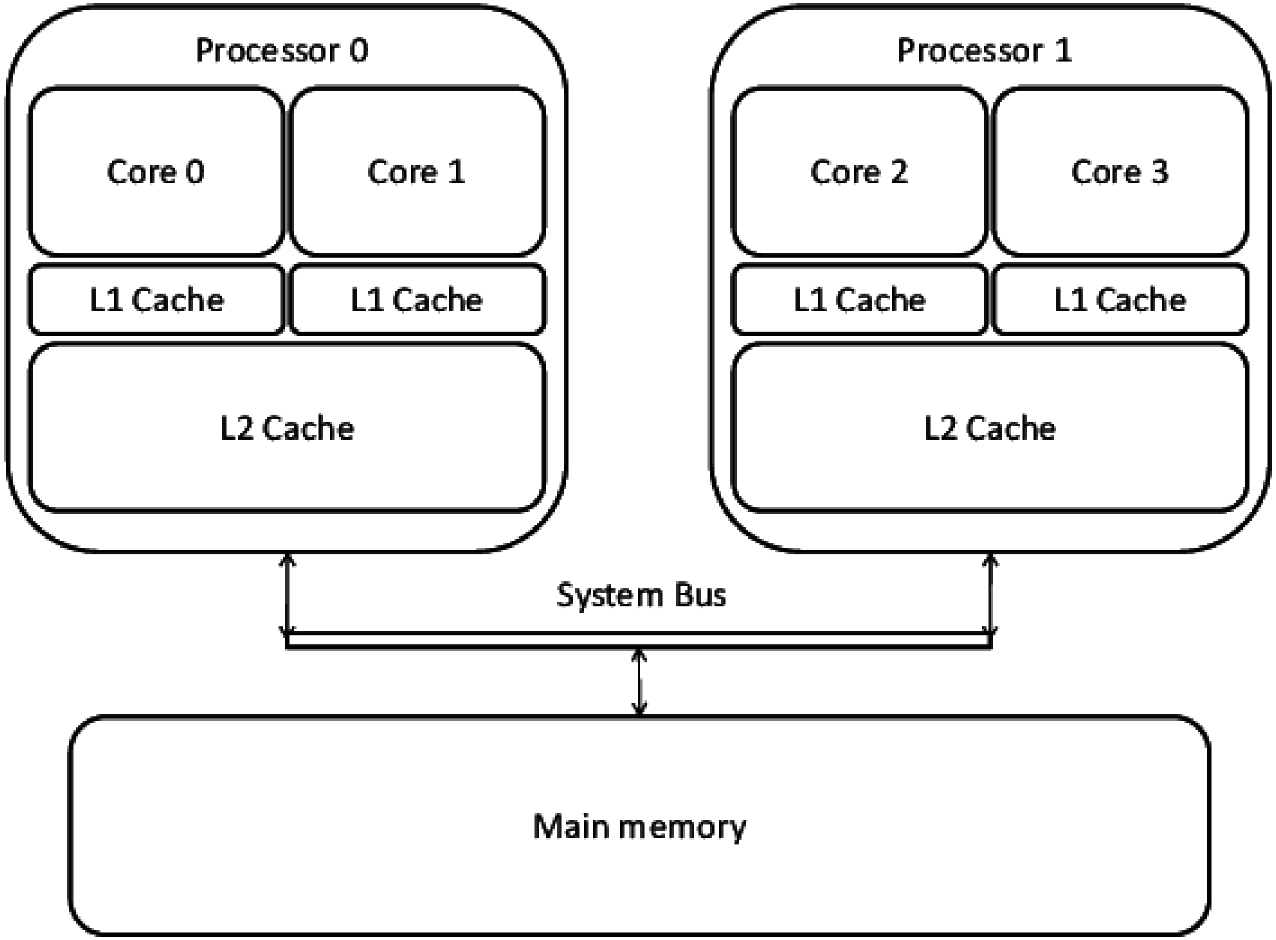
End of Moore’s/Dennard’s Law

- The prediction had been true for a long time
- We observe that #transistors does not increase in the scale of Moore’s law
- We reach the end of Dennard scaling law

Multi-core CPUs



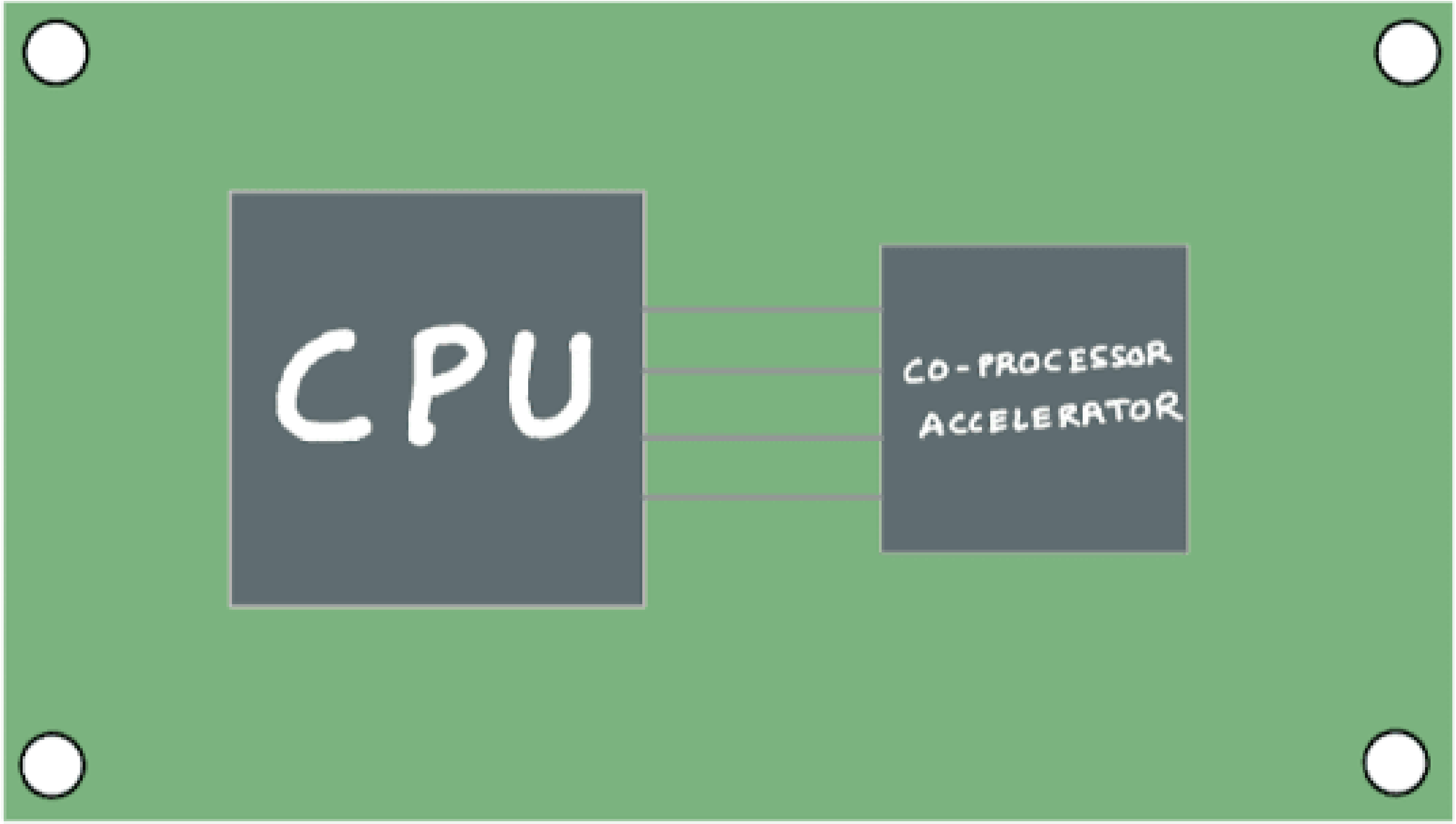
Multi-processors



New requirements

- Big data
- New applications:
  - Massively parallel
  - Certain operations
- Faster computation

Accelerator



Accelerators/Co-processors

- Graphics Processing Units (GPUs)
- Field Programmable Gate Arrays (FPGAs)
- Tensor Processing Units (TPUs)
- ...

Simplere many cores

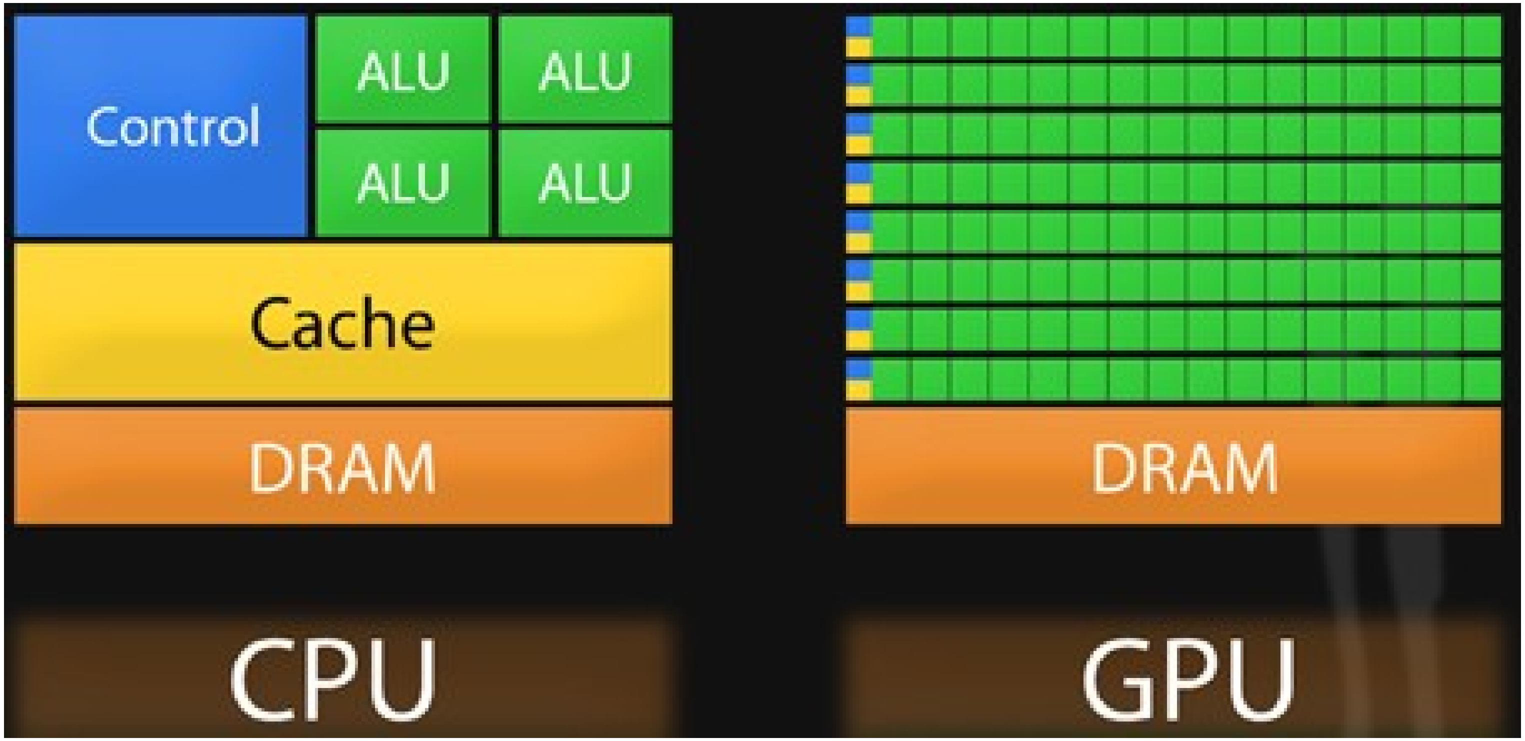
- Simpler cores (i.e., simplified ALUs and CUs)
- Replicate many of them
- As a co-processor

GPUs

- Initially invented for image rendering purposes

- Gradually evolved to be used as General Purpose GPU

GPUs vs CPUs



Two Metrics

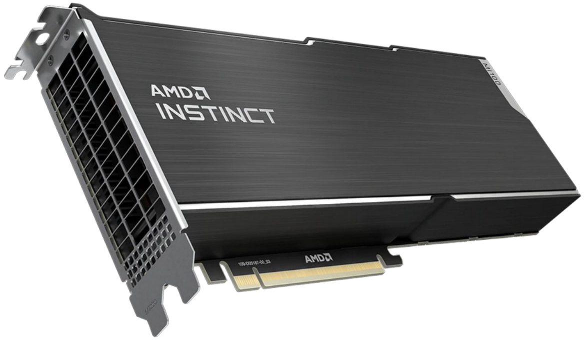
- Latency: the time it takes an instruction to be processed

- Throughput: the number of instructions that can be processed in a certain amount of time

Two Metrics

- CPUs are latency-optimized processors
- GPUs are throughput-optimized (co-)processors

GPU Manufacturers



Supercomputers

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
5	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461

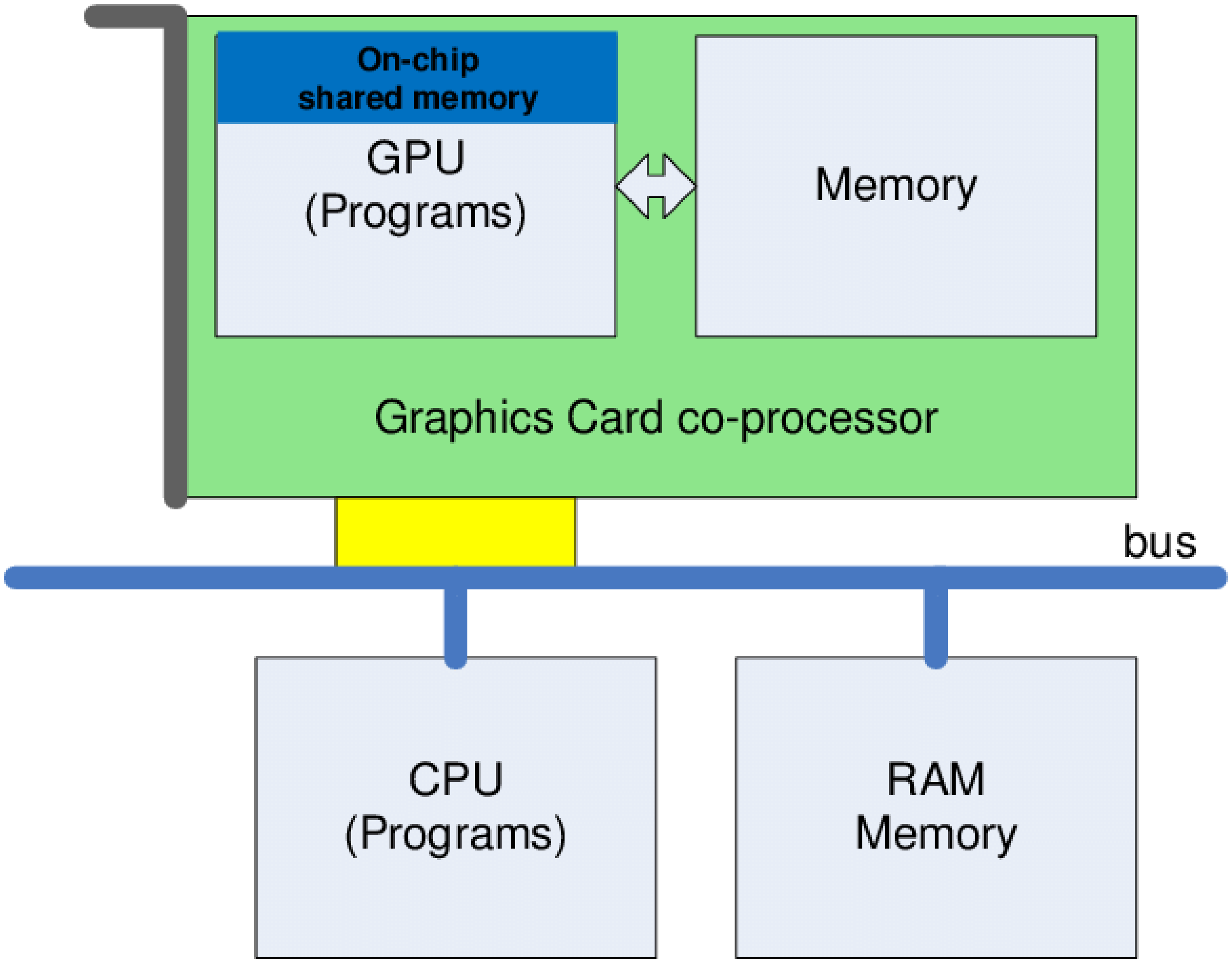
Supercomputers

6	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
7	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	2,121,600	434.90	574.84	7,124
8	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
9	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494
10	**Tuolumne** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	1,161,216	208.10	288.88	3,387

Snellius Supercomputer

- 72 GPU nodes with 4 A100 NVIDIA GPUs; 288 A100 GPUs
- 88 GPU nodes with 4 H100 NVIDIA GPUs; 352 H100 GPUs
- In total 640 GPUs

GPU CPU Connectivity



GPU Usability

- How to Use GPUs?

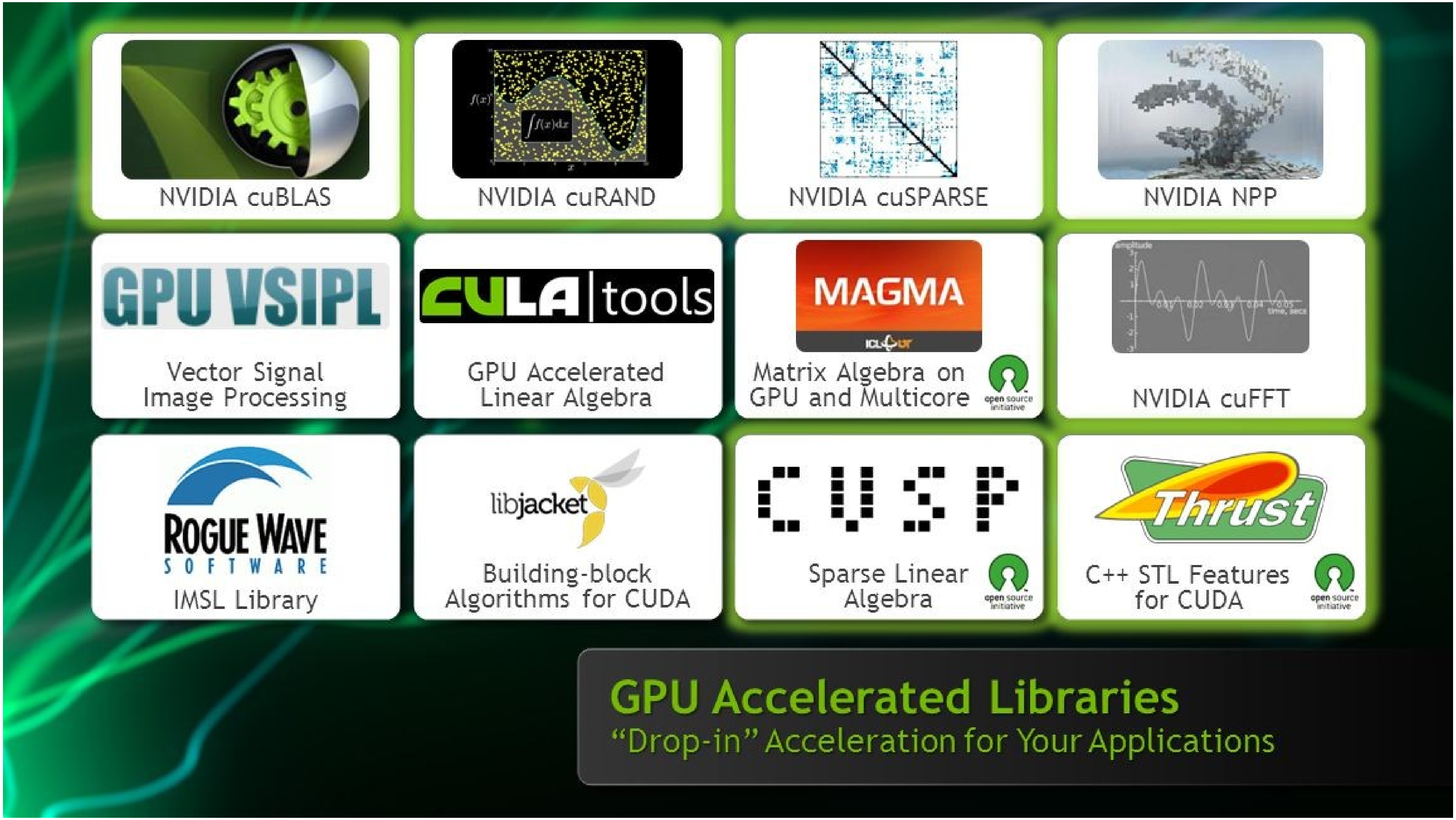
GPU Usability

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

GPU Libraries



GPU Usability

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

OpenACC/OpenMP

- OpenACC stands for Open Accelerators
- OpenMP stands for Open Multi-Processing
- Directive-based APIs
- Simple compiler hints to parallelize the code

GPU Usability

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran, OpenACC, OpenMP4.5
C	CUDA C, OpenCL, OpenACC, OpenMP4.5
C++	CUDA C++, Thrust, OpenCL, OpenACC/OpenMP4.5
Python	PyCUDA/PyOpenCL, Numba, …

Core GPU Programming

- Nvidia GPUs:
  - CUDA, OpenCL, HIP
- AMD GPUs:
  - OpenCL, HIP
- Intel GPUs:
  - OpenCL

Accessing to GPUs in Python

PyTorch

PyCUDA Numba CuPy TensorFlow

Low-level High-level

PyTorch/TensorFlow

- They are powerful and mature deep learning libraries
- They benefit from GPUs without knowing GPU programming knowledge
- They are open sources
- They are taught in machine learning courses

CuPy vs NumPy

NumPy

```
import numpy as np
X_cpu = np.zeros((10,))
W_cpu = np.zeros((10, 5))
y_cpu = np.dot(x_cpu, W_cpu)
```

CuPy

```
import cupy as cp
x_gpu = cp.zeros((10,))
W_gpu = cp.zeros((10, 5))
y_gpu = cp.dot(x_gpu, W_gpu)
```

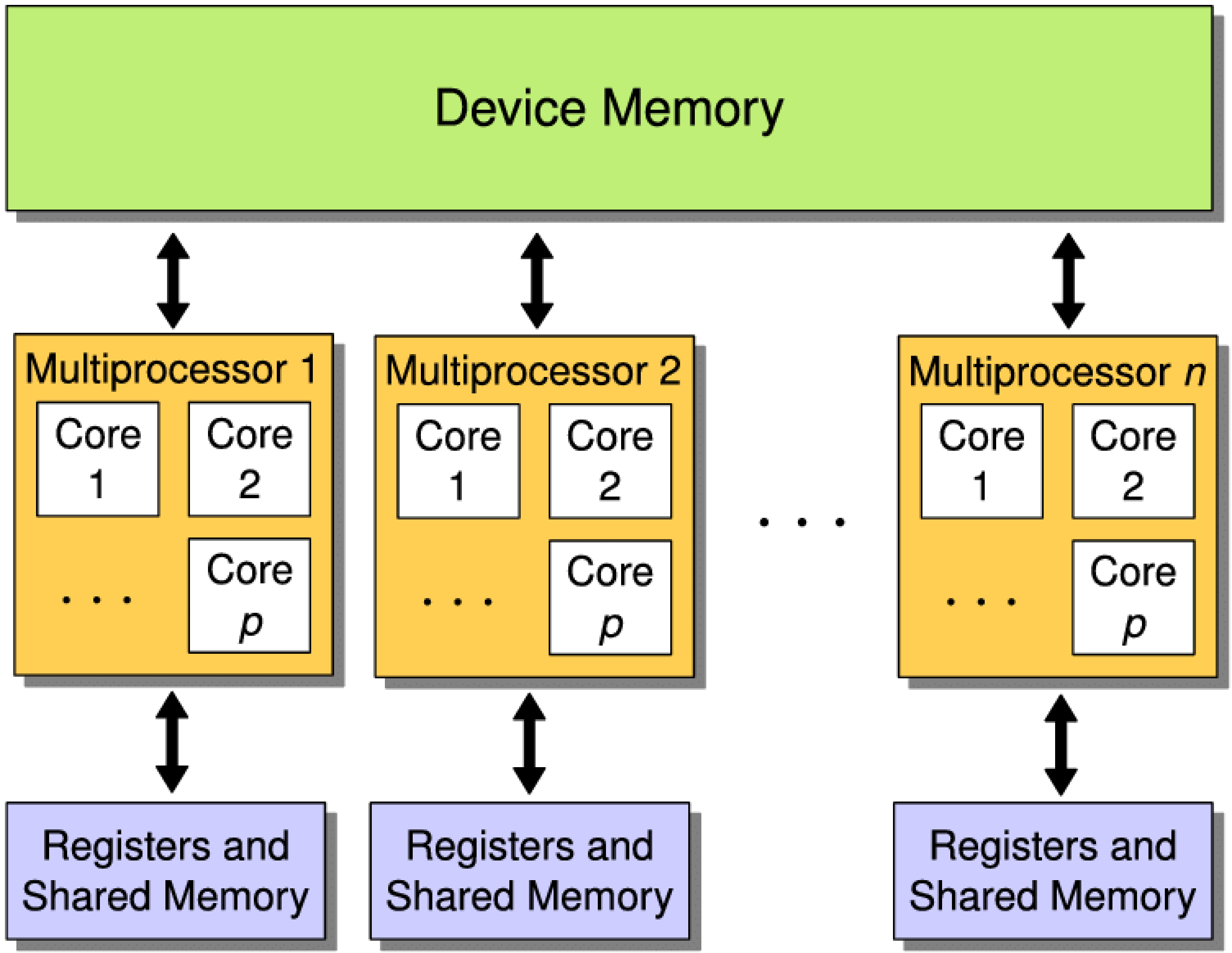
Numba

- It is an open-source Just-In Time (JIT) compiler that translates a subset of Python and Numpy into GPU machine code
- It uses a collection of decorators that can be applied to your functions to instruct Numba to compile them
- For more information: https://numba.pydata.org/

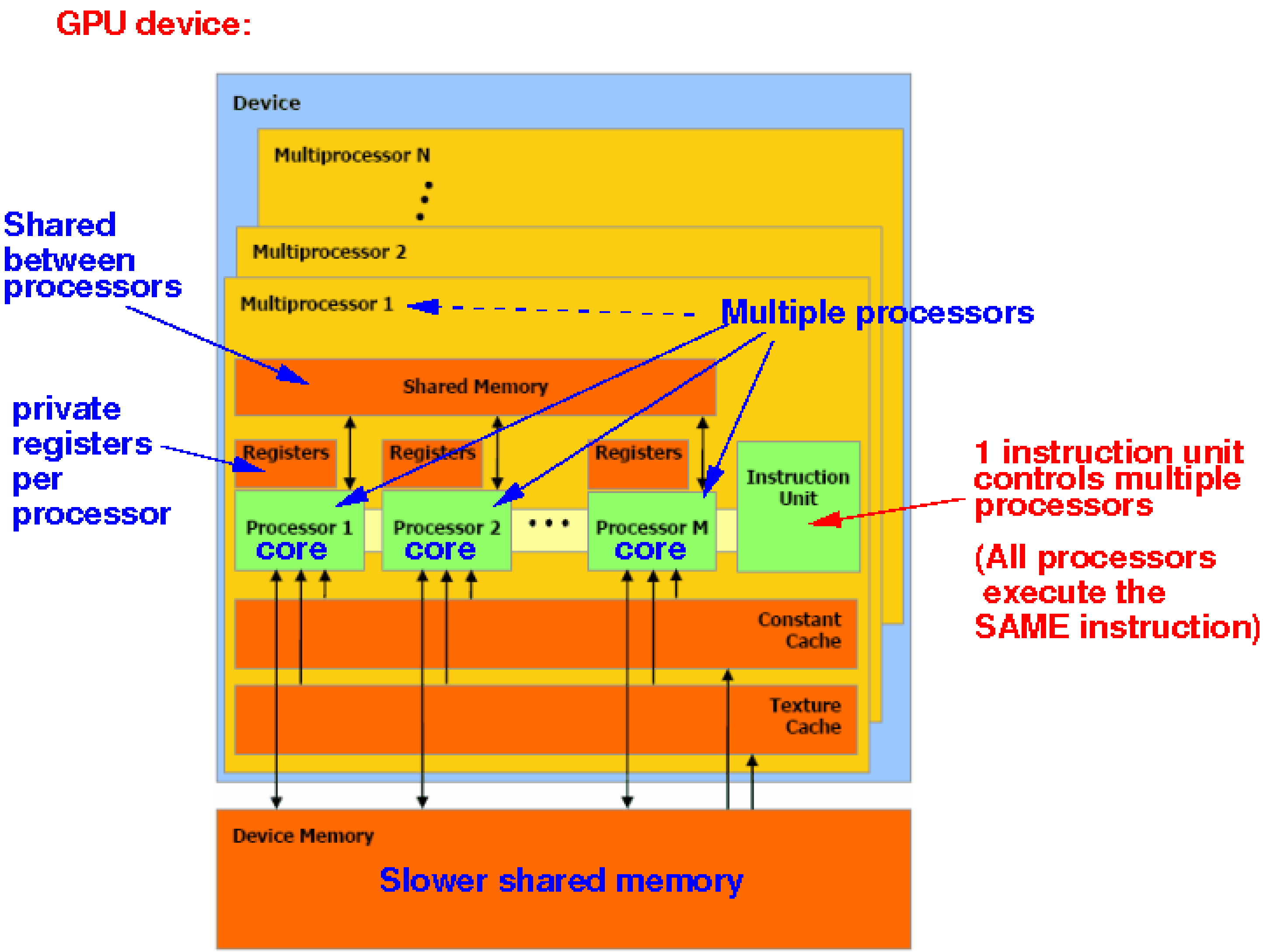
PyCUDA

- It gives you easy, Pythonic access to NVIDIA's CUDA parallel computation API
- There is more flexibility to write custom CUDA kernels
- For more information: https://documen.tician.de/pycuda/

NVIDIA GPU Hardware



NVIDIA GPU Hardware



Flynn’s classical taxonomy

		Instruction stream	
		Single	Multiple
Data stream	Single	SISD	MISD
	Multiple	SIMD	MIMD

CUDA Programming Model

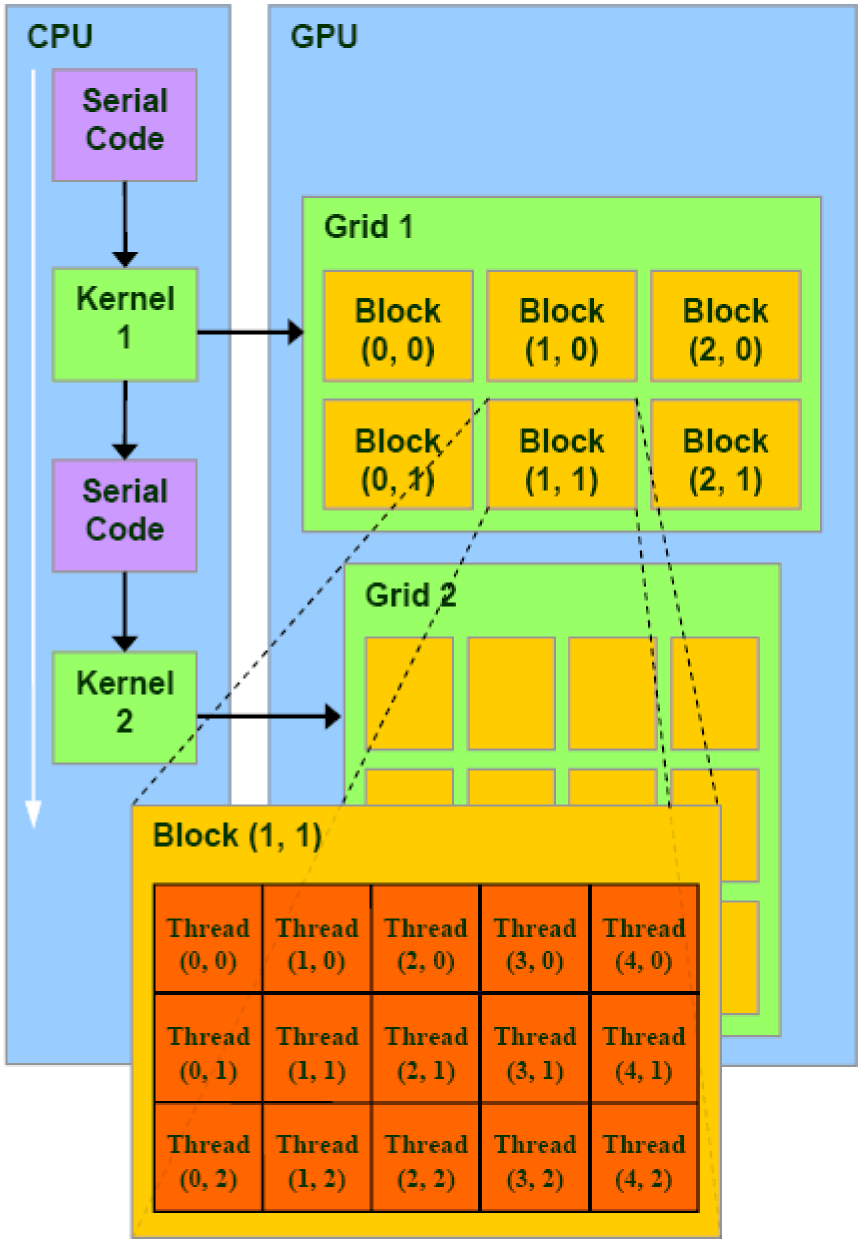
- Introduced by NVIDIA in 2006, Compute Unified Device Architecture
- General purpose programming model that leverages the parallel compute engine in NVIDIA GPUs
- An extension of C language
- CUDA programs are CPU-GPU programs:
  - CPU part is called *host*
  - GPU part is called *kernel*

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

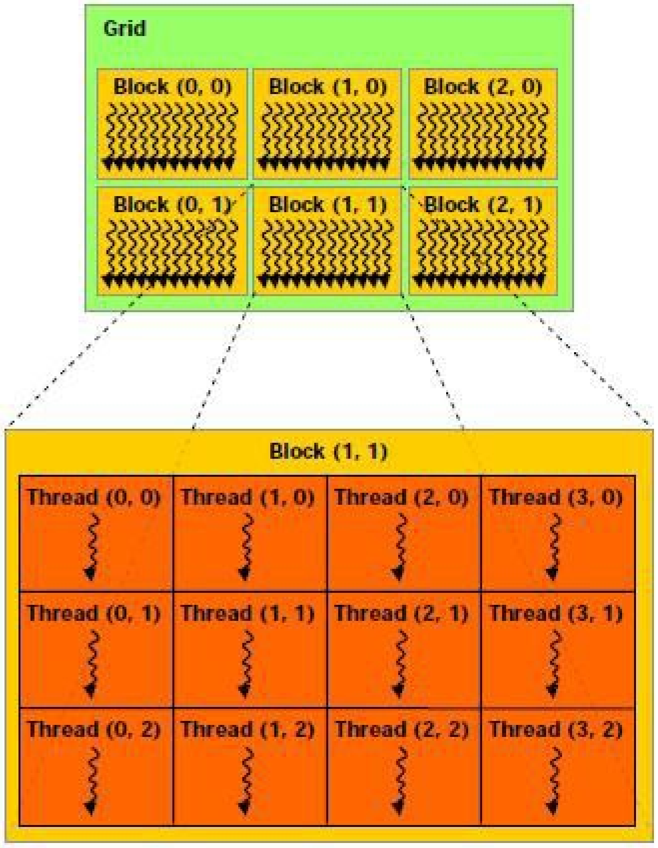
CUDA Programming Model



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

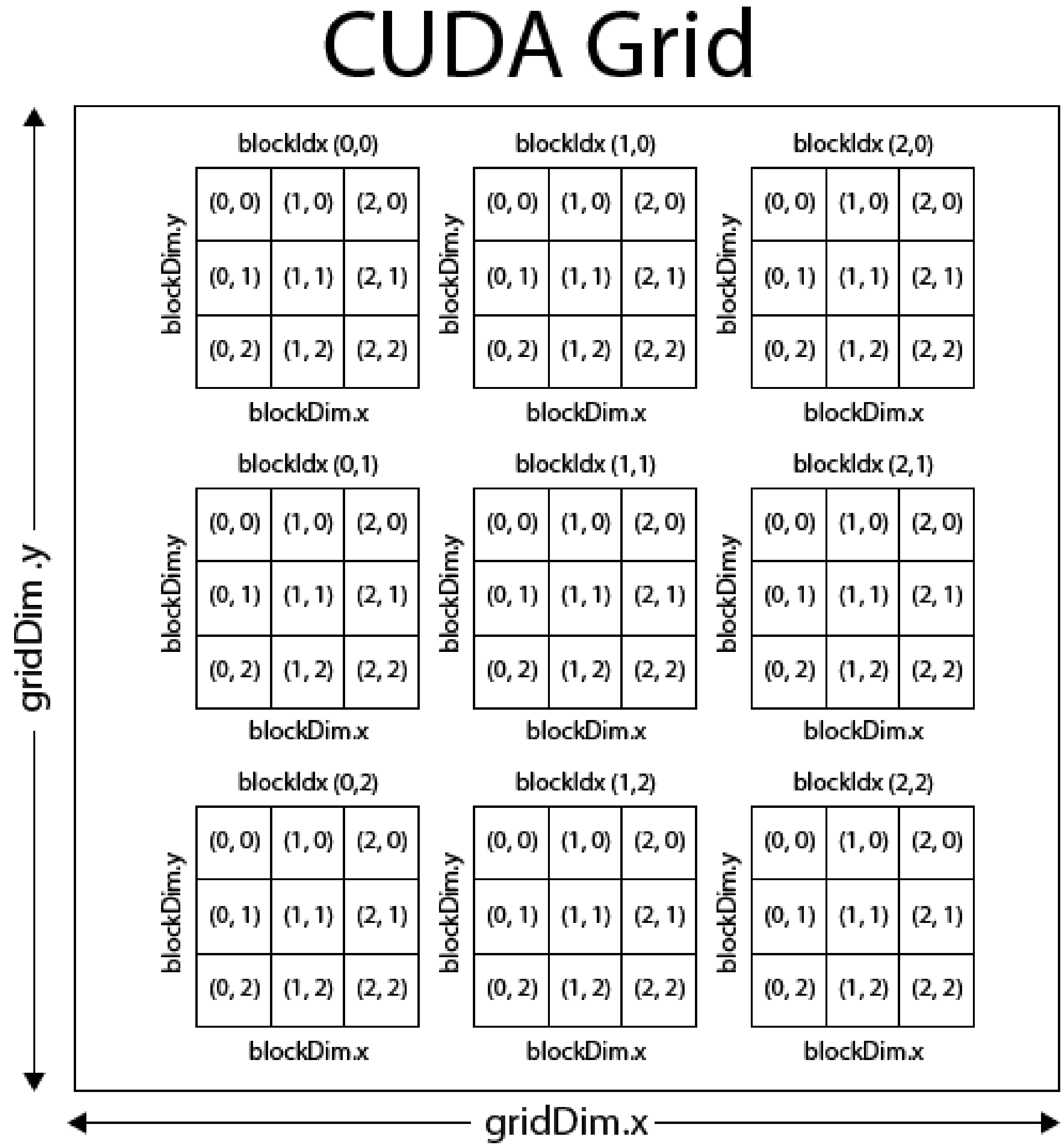


CUDA Programming Model

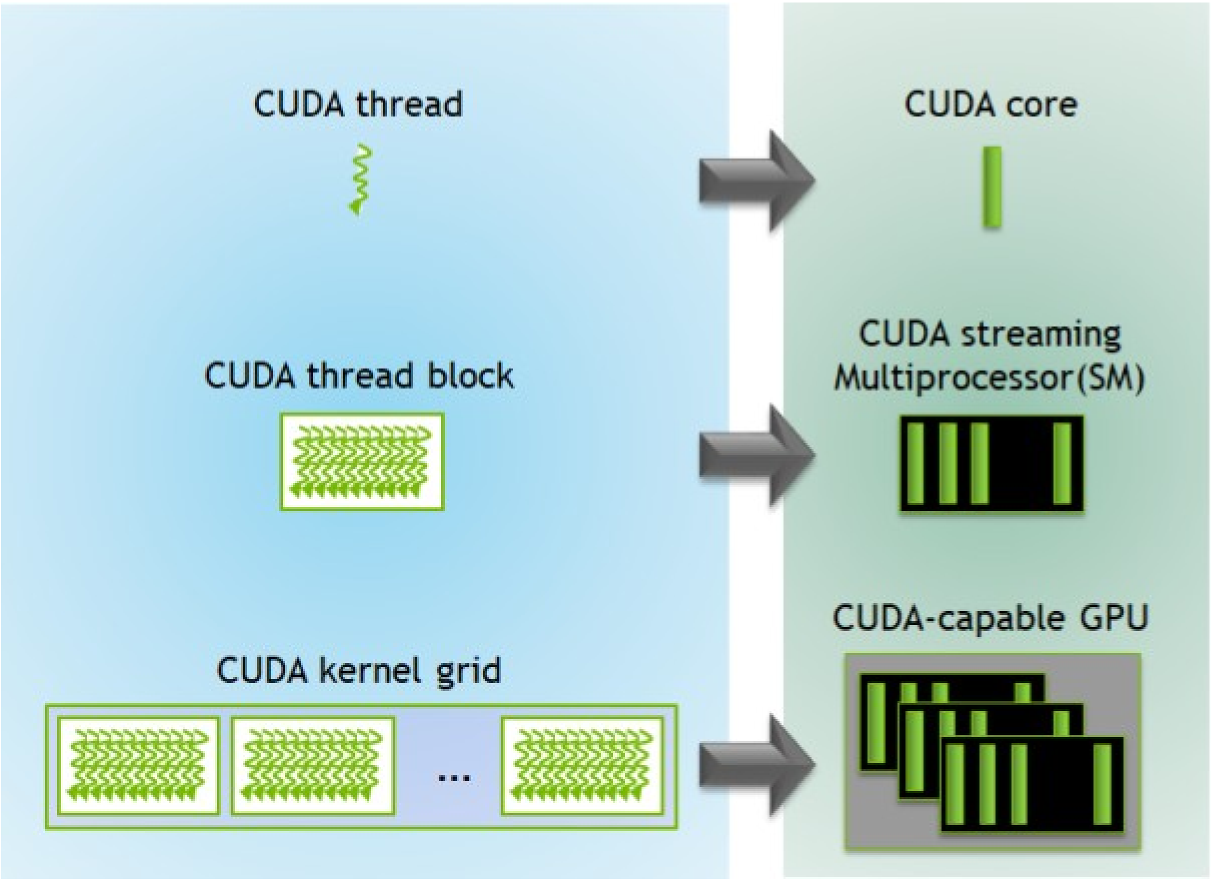
- Built-in functions:
  - Dimension:
    - gridDim.x, gridDim.y, gridDim.z
    - blockDim.x, blockDim.y, blockDim.z
  - Index:
    - blockIdx.x, blockIdx.y, blockIdx.z
    - threadIdx.x, threadIdx.y, threadIdx.z

CUDA Programming Model

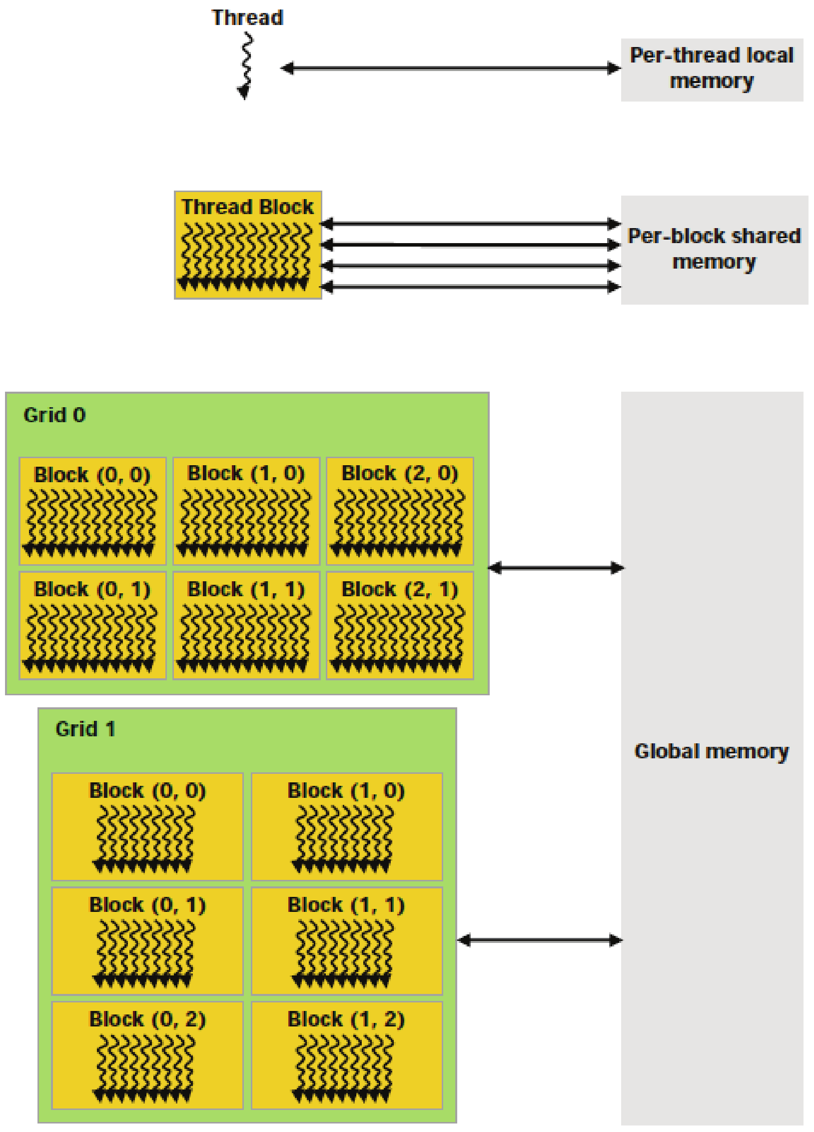
- gridDim.x = 3
- gridDim.y = 3
- blockDim.x = 3
- blockDim.y = 3



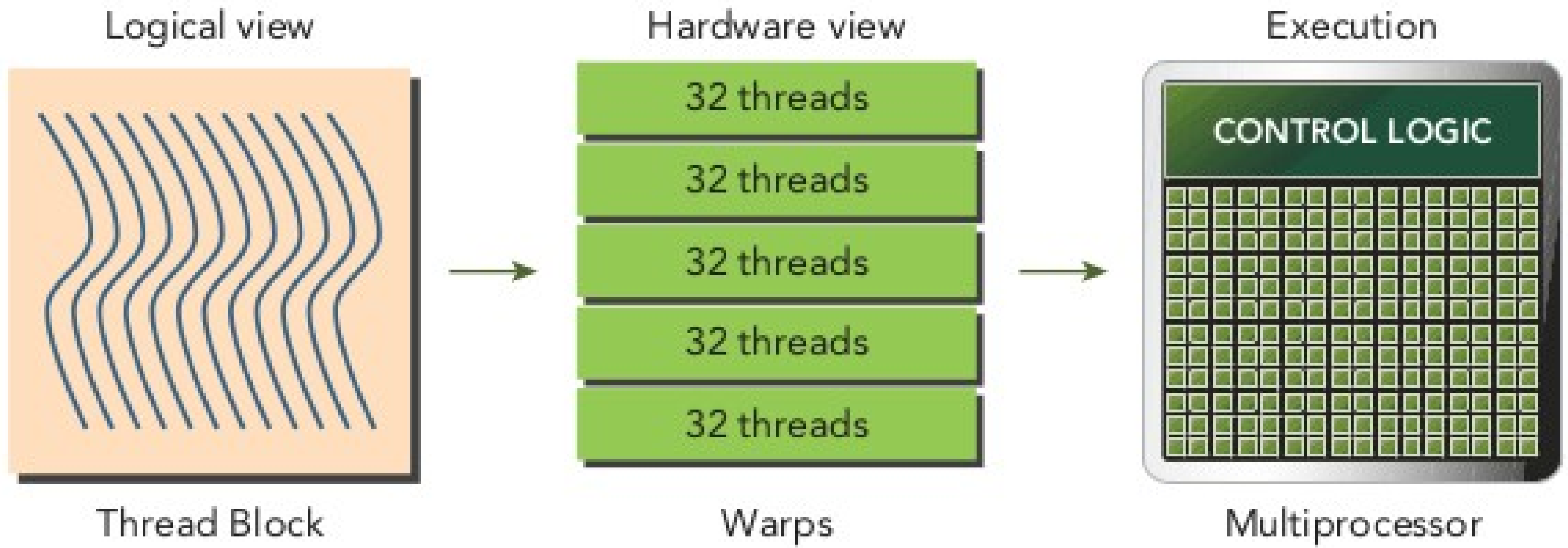
CUDA Execution Model



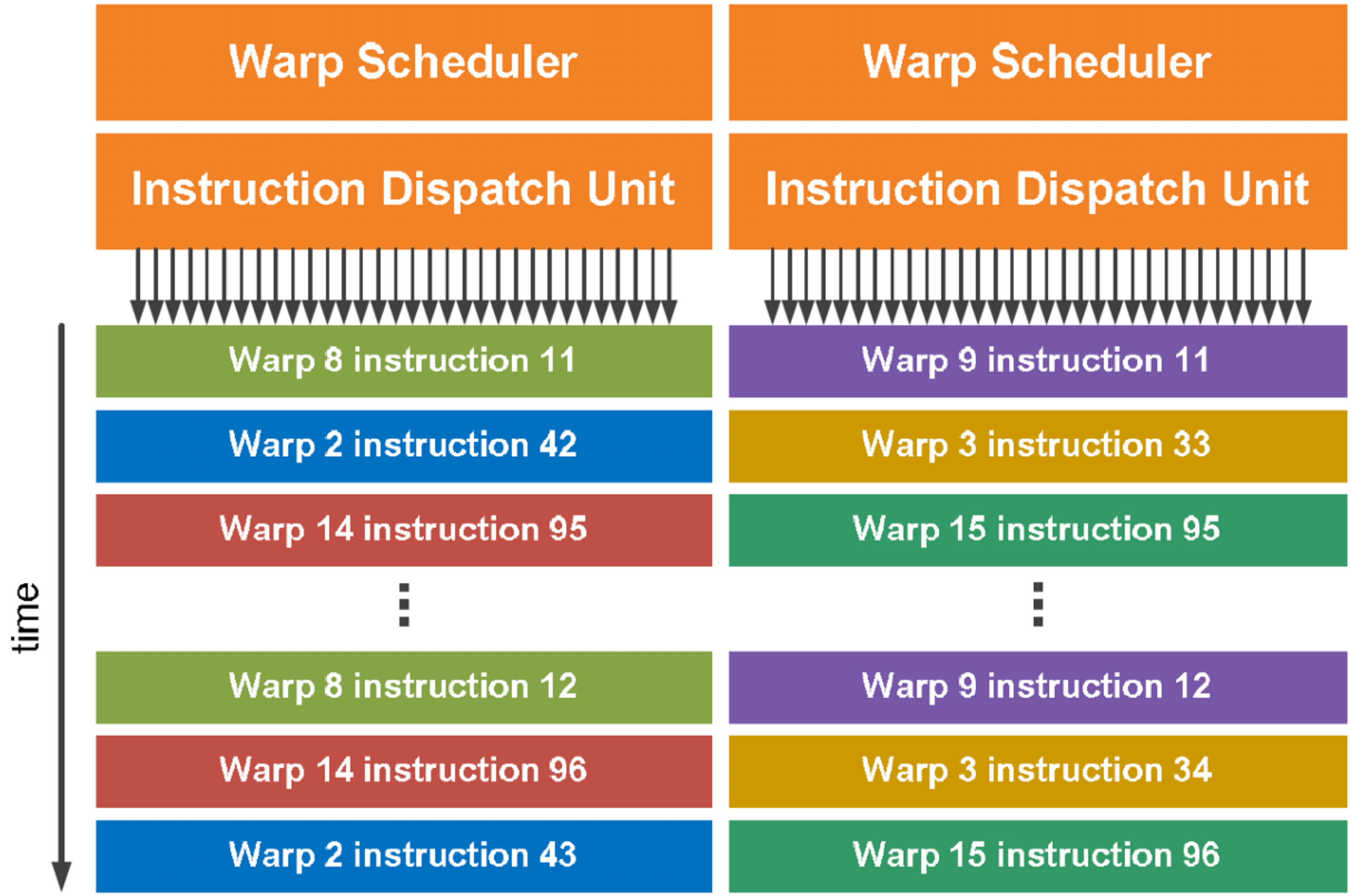
CUDA Execution Model



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
  - Decouple the kernel into two separate kernels

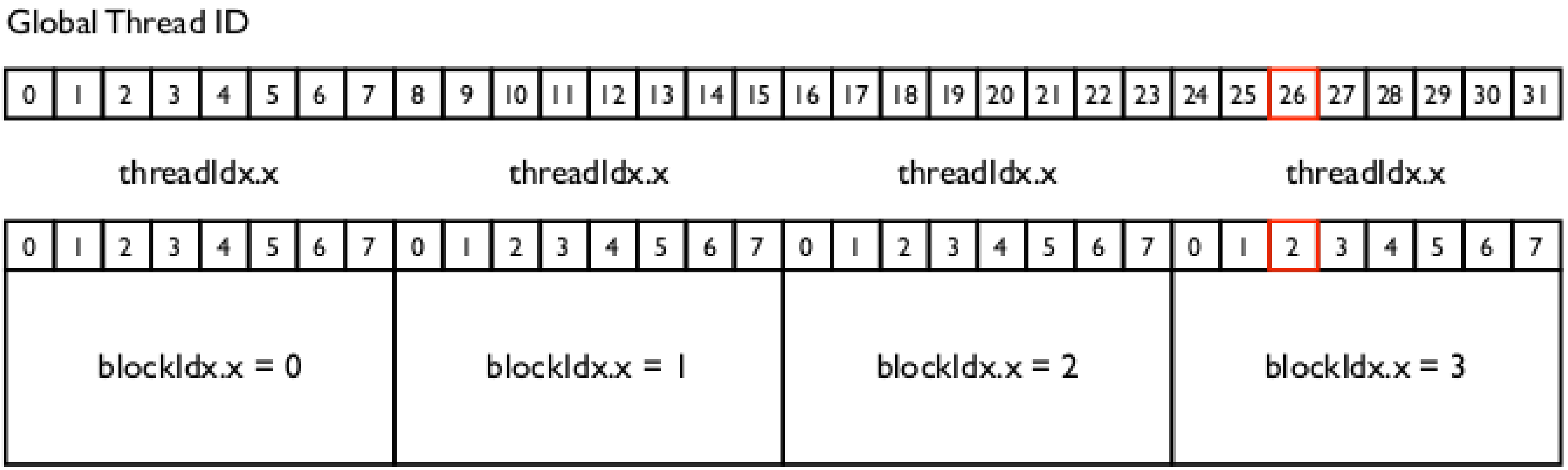
GPU Node

- 4 NVIDIA A100 GPUs per node
  - Multiprocessors: 108
  - Streaming cores: 6912
  - Tensor Cores: 432
  - Global memory: 40 GB
- MIG partitions: 1/7th of A100 GPUs
- One GPU is shared among 7 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

PyCUDA Implementation

- Implement vector addition in PyCUDA
- Compare its execution time to the sequential version

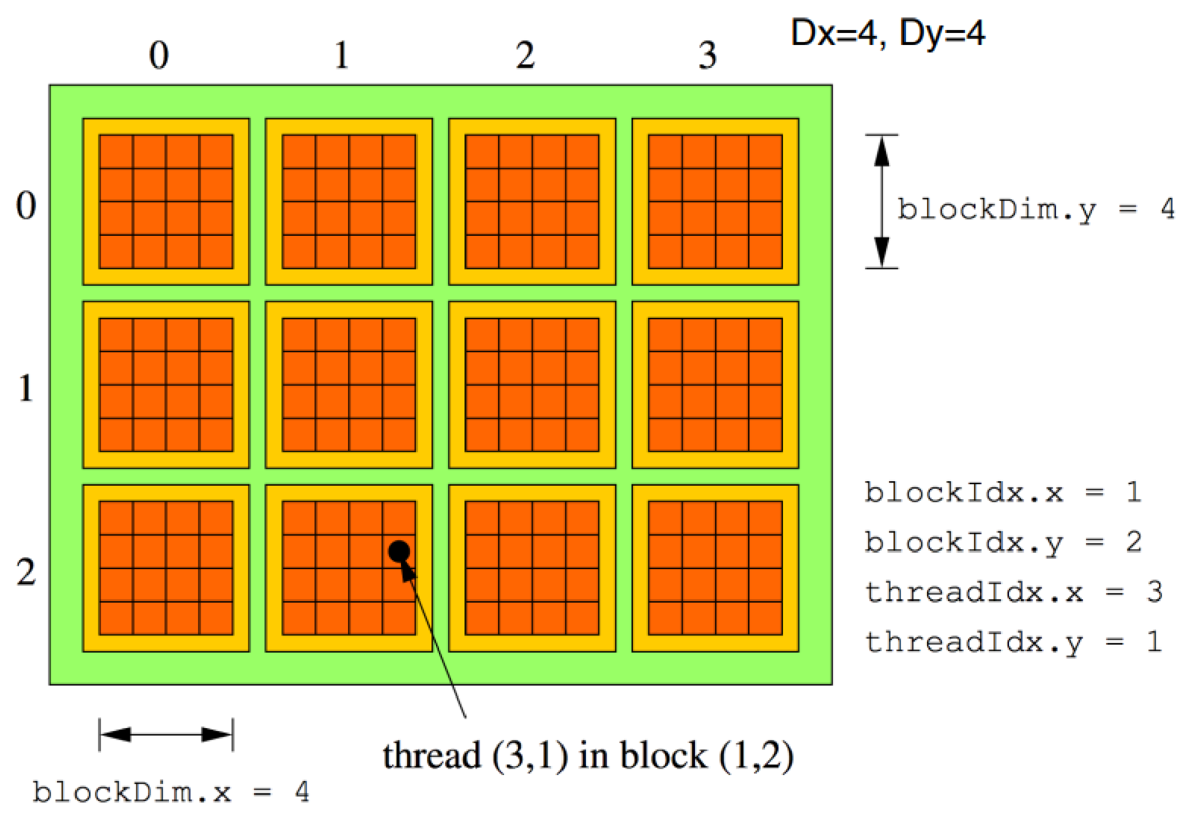
Automatic Data Transfer

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

Second Example:

Parallel Matrix (2D array) Addition in PyCUDA

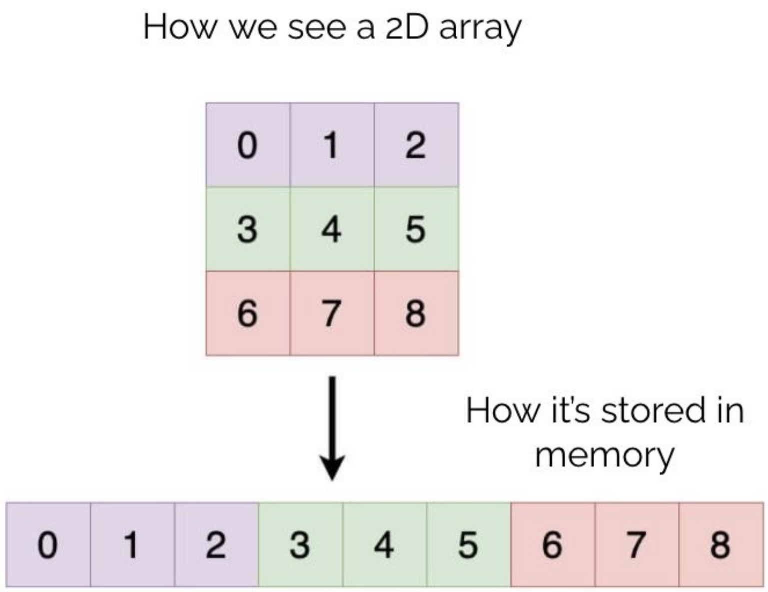
Calculate Global Index (2D grid, 2D block)



- Matrix 12*16

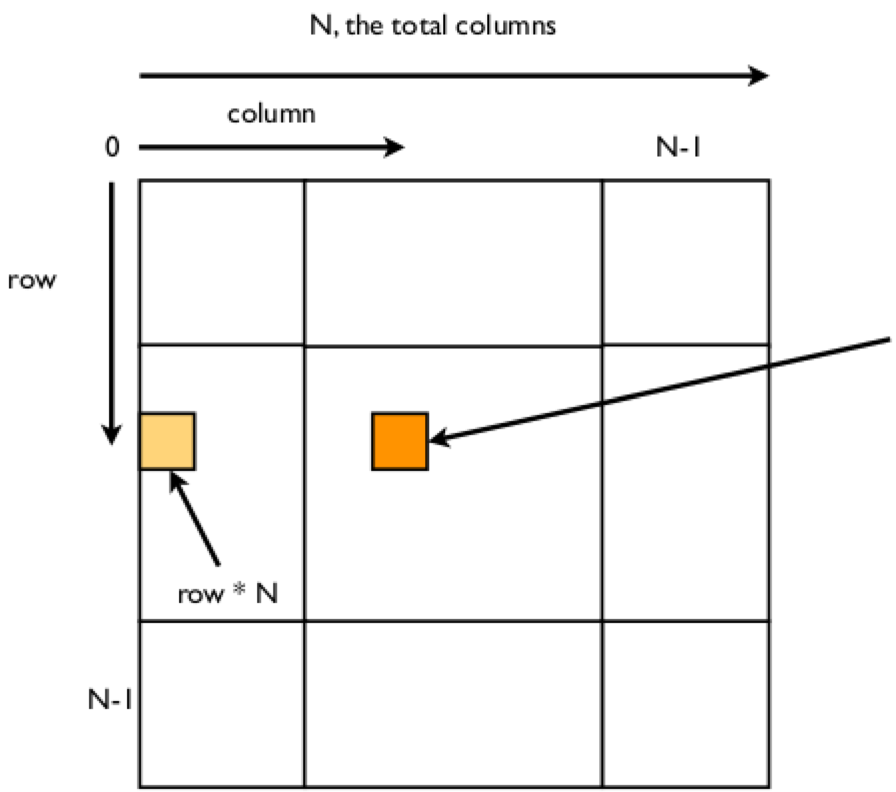
- Global Thread ID:
  - row = blockIdx.y * blockDim.y + threadIdx.y = 2 * 4 + 1 = 9
  - column = blockIdx.x * blockDim.x + threadIdx.x = 1 * 4 + 3 = 7

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] = …
```

PyCUDA Implementation

- Implement matrix addition in PyCUDA
- Compare its execution time to the sequential version

Exercise 1

- Try to transpose a matrix in parallel using PyCUDA
- Compare its execution time to the sequential version

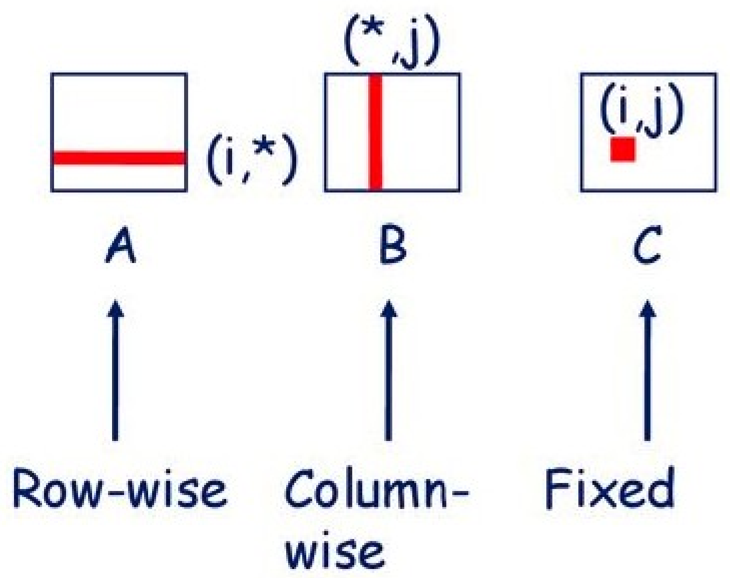
Third Example:

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;
  }
}
```

Inner loop:



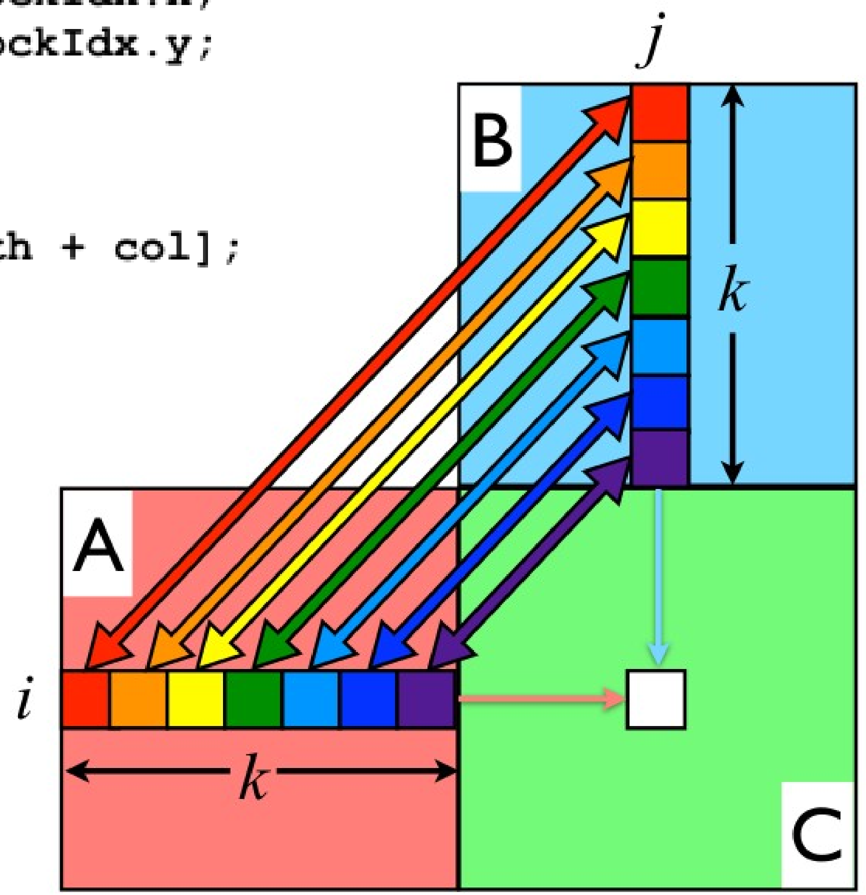
$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

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;
}
```

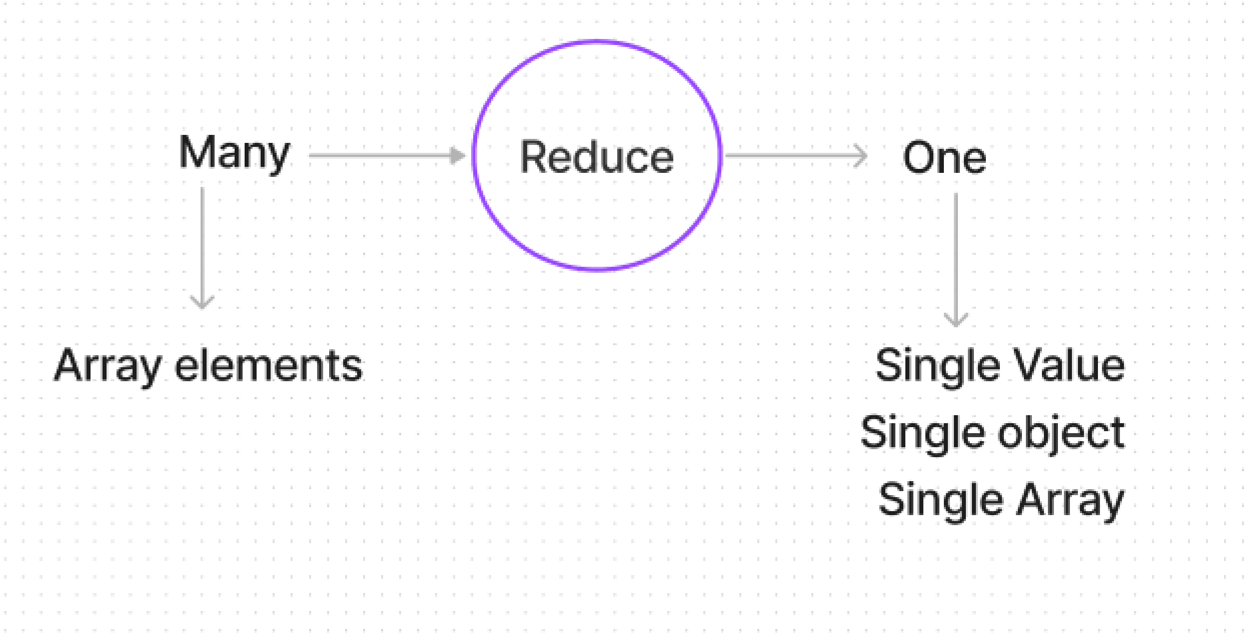


PyCUDA Implementation

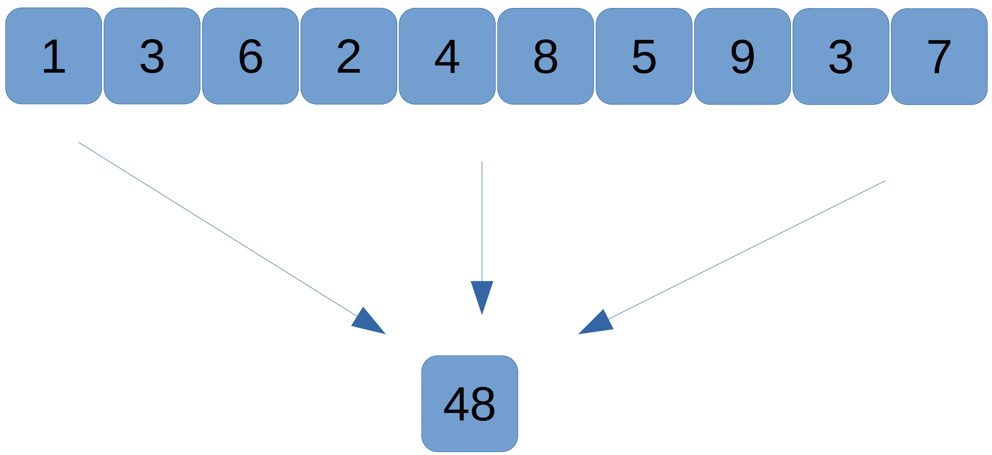
- Implement matrix multiplication in PyCUDA
- Compare its execution time to
  - Sequential CPU-based
  - Numpy.matmul()
  - @ operator

Fourth Example:
Reduction in PyCUDA

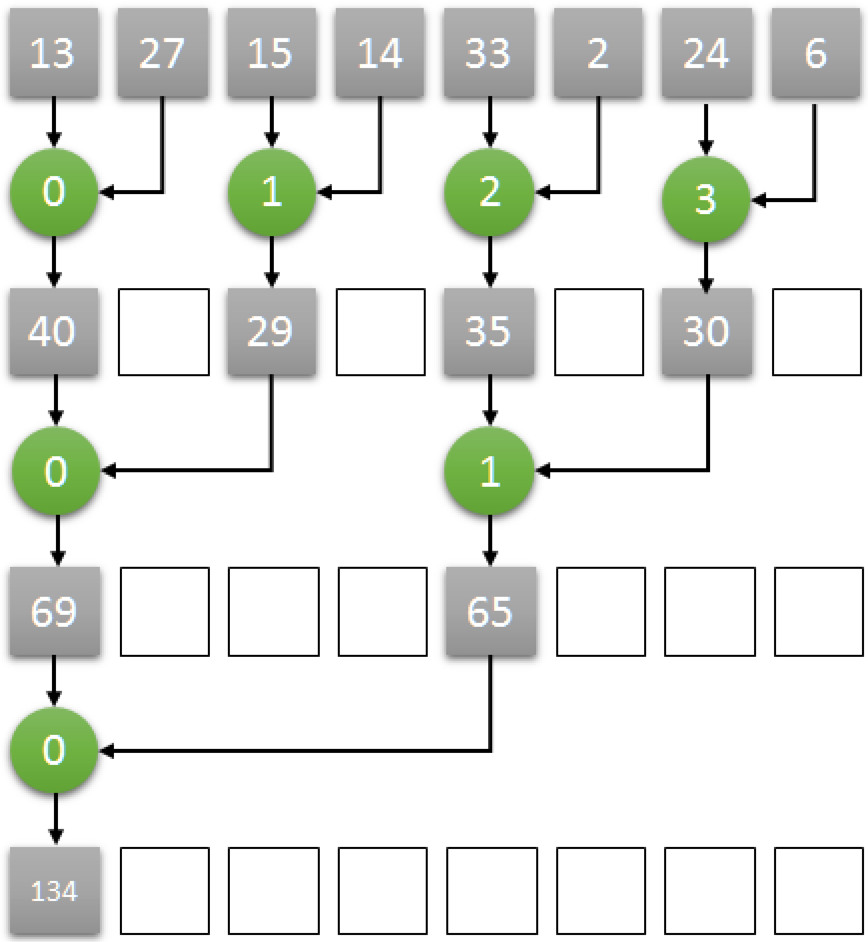
Reduction


Many
Reduce
One
Array elements
Single Value
Single object
Single Array

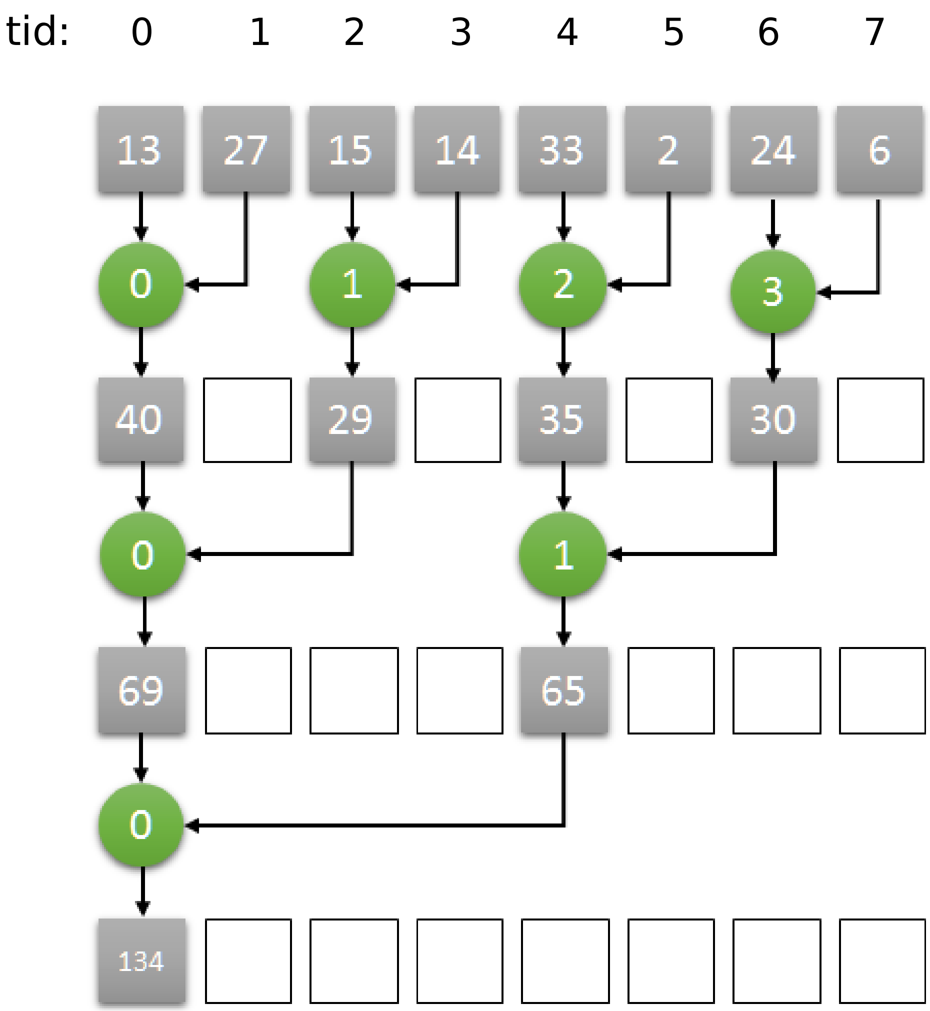
Reduction (addition)



Reduction (addition)



Reduction (addition)



level

1 (tid%2==0) ==> a[tid] += a[tid+1]

2 (tid%4==0) ==> a[tid] += a[tid+2]

3 (tid%8==0) ==> a[tid] += a[tid+4]

4 (tid%16==0) ==> a[tid] += a[tid+8]

(tid%(2^level)==0) ==> a[tid] += a[tid+2^(level-1)]

PyCUDA Implementation

- Implement reduction in PyCUDA using one thread block
- Compare its execution time to the sequential version and Python reduce operator

PyCUDA Implementation

- Extend it to use arbitrary size (i.e., multiple thread blocks)
- Compare its execution time to the sequential version and Python reduce operator

PyCUDA Implementation

- How to use shared memory in reduction?
- Compare its execution time to the sequential version and Python reduce operator

Exercise 2

- Reduce an array using other operators (subtraction, multiplication, etc.)

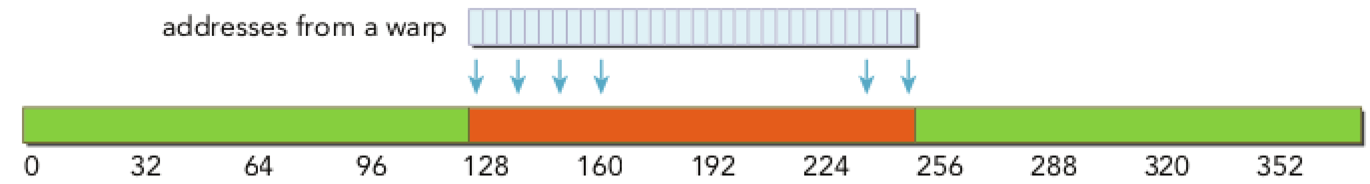
Optimization

There are different ways to optimize CUDA codes:

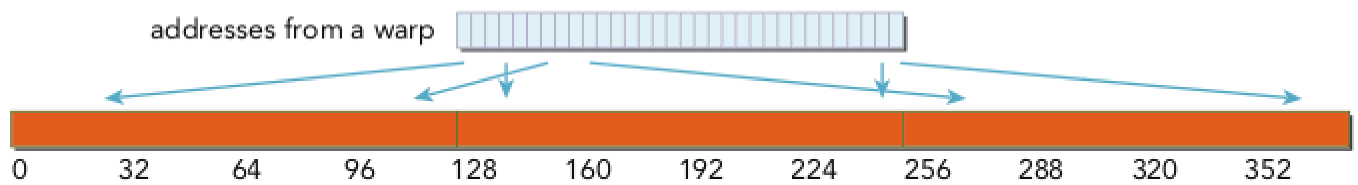
- Number of threads per block
- Workload per thread
- Total work per thread block
- Correct memory access and data locality
- ...

Tips for Optimization

- Global Memory Access:



Coalesced

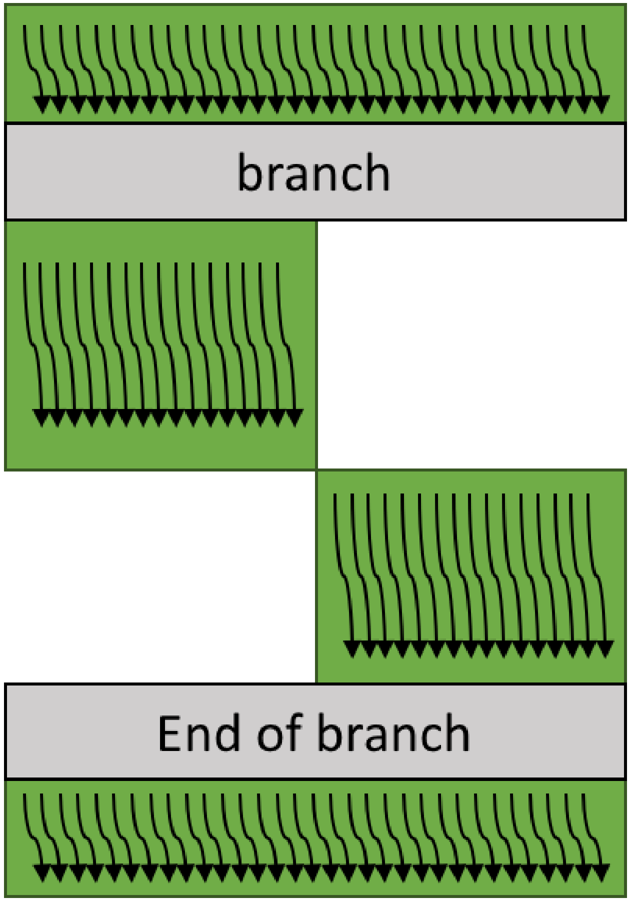


Non-coalesced

Tips for Optimization

- Avoid Warp Divergence:

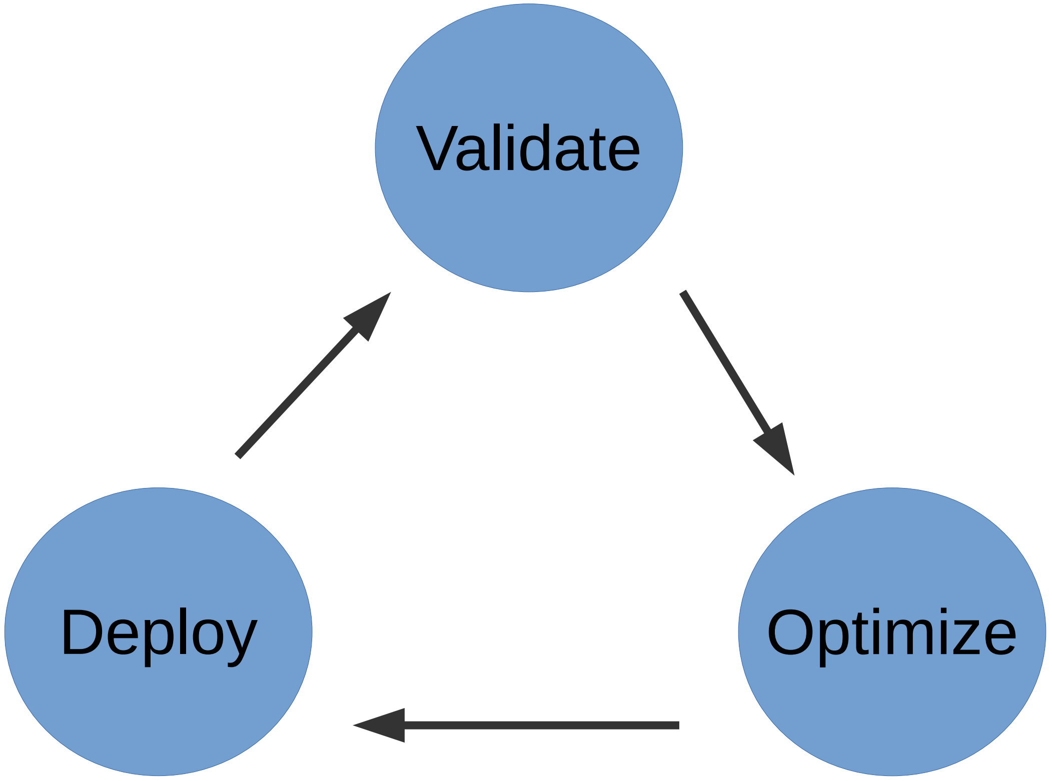
```
...
if ( threadIdx.x < 16 )
{
     ... A ...
}
else
{
     ... B ...
}
...
```



Tips for Optimization

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

GPU Development Cycle



Data Races

- A data race is a situation where two or more threads may access the same memory location simultaneously and at least one of them is a write
- It causes undefined behavior of programs

Data Race Example

```
__global__ void kernel(int *arr)
{
    arr += 1;
}
```

- One solution is to use built-in atomic operations in GPU programming languages

Data Race Example

```
__global__ void kernel(int *arr, int size)
{
    if (tid < size-1)
    {
        arr[tid] += arr[tid+1];
    }
}
```

- One solution is to use synchronization methods in GPU programming

Barrier Divergence

- A barrier divergence happens when threads from the same thread block diverge and hit different (syntactical) barriers

Barrier Divergence Example

```
__global__ void kernel(...){
    if (tid % 2  == 0){
        ....
        syncthreads();
        ....
    } else{
        ...
        syncthreads(); }
}
```

Questions

Thank you for participating! Any questions?