

PARALLEL AND GPU PROGRAMMING IN PYTHON

PRACE Training Course

Ben Czaja, Mohsen Safari, Clicia Dos Santos Pinto
HPC advisors, SURF



SURF

© Annemiek van de

Amsterdam Science Park



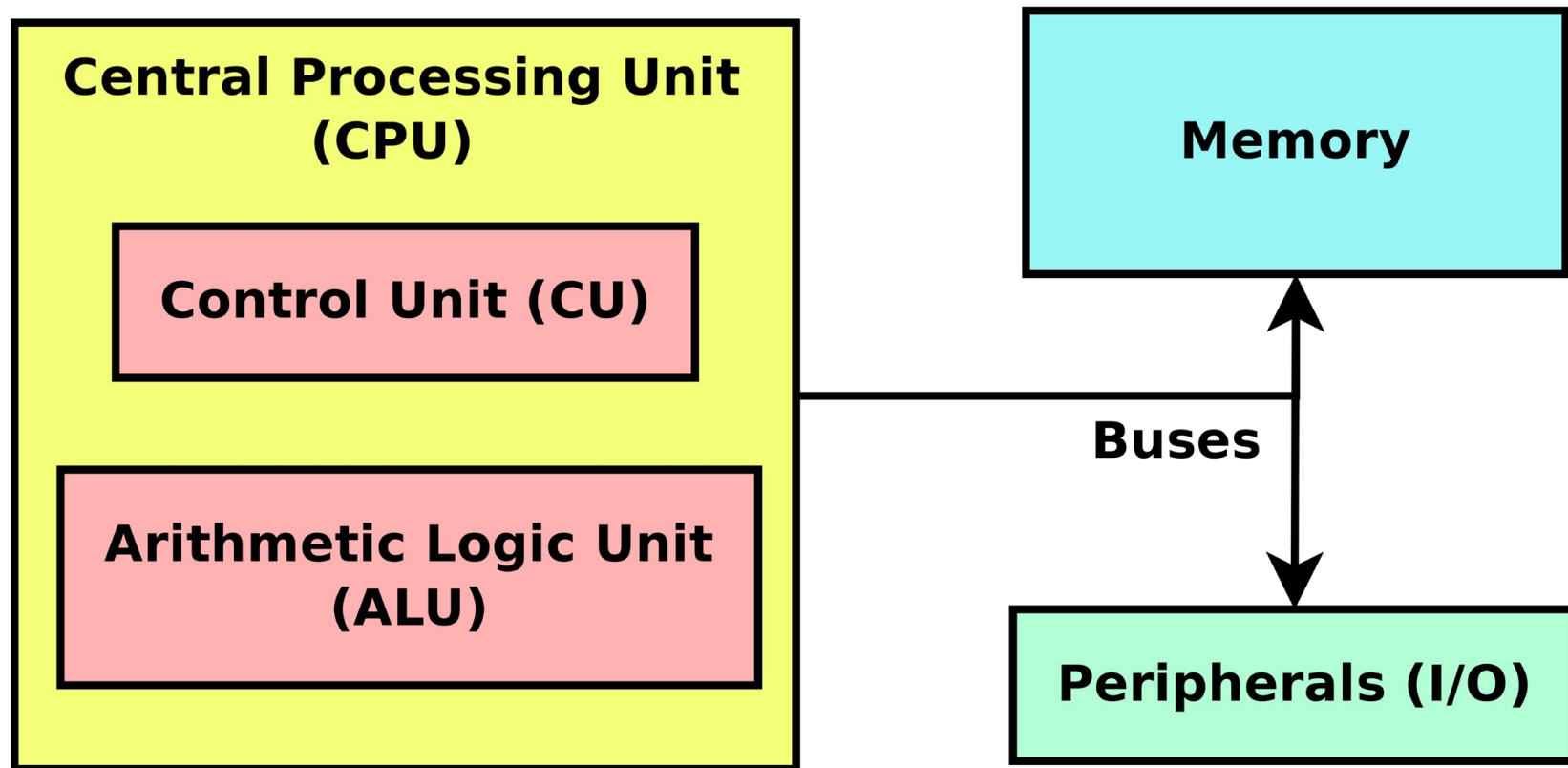
Outline

- 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

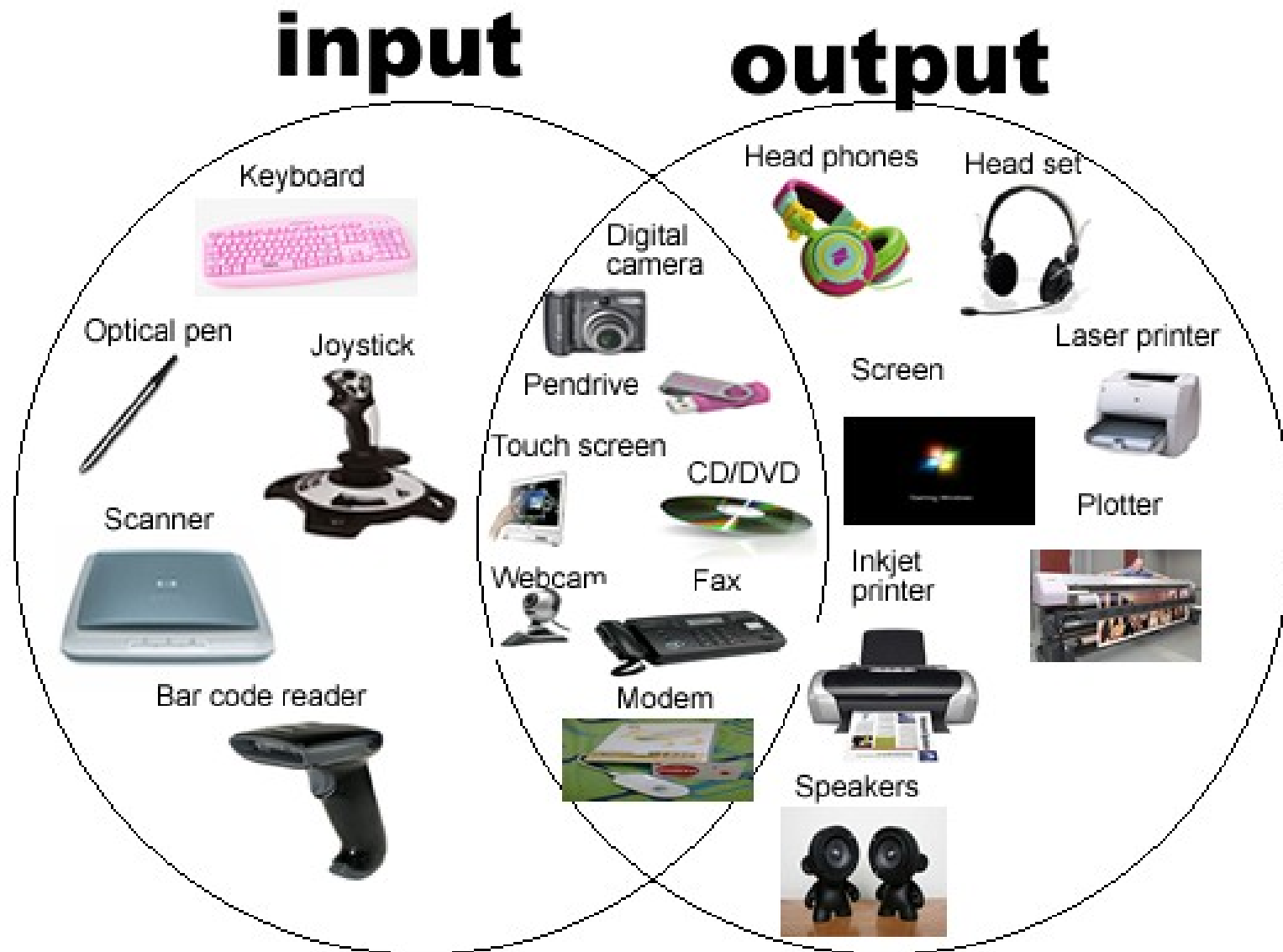
Hardware Accelerator (e.g., GPUs)

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

A computer is



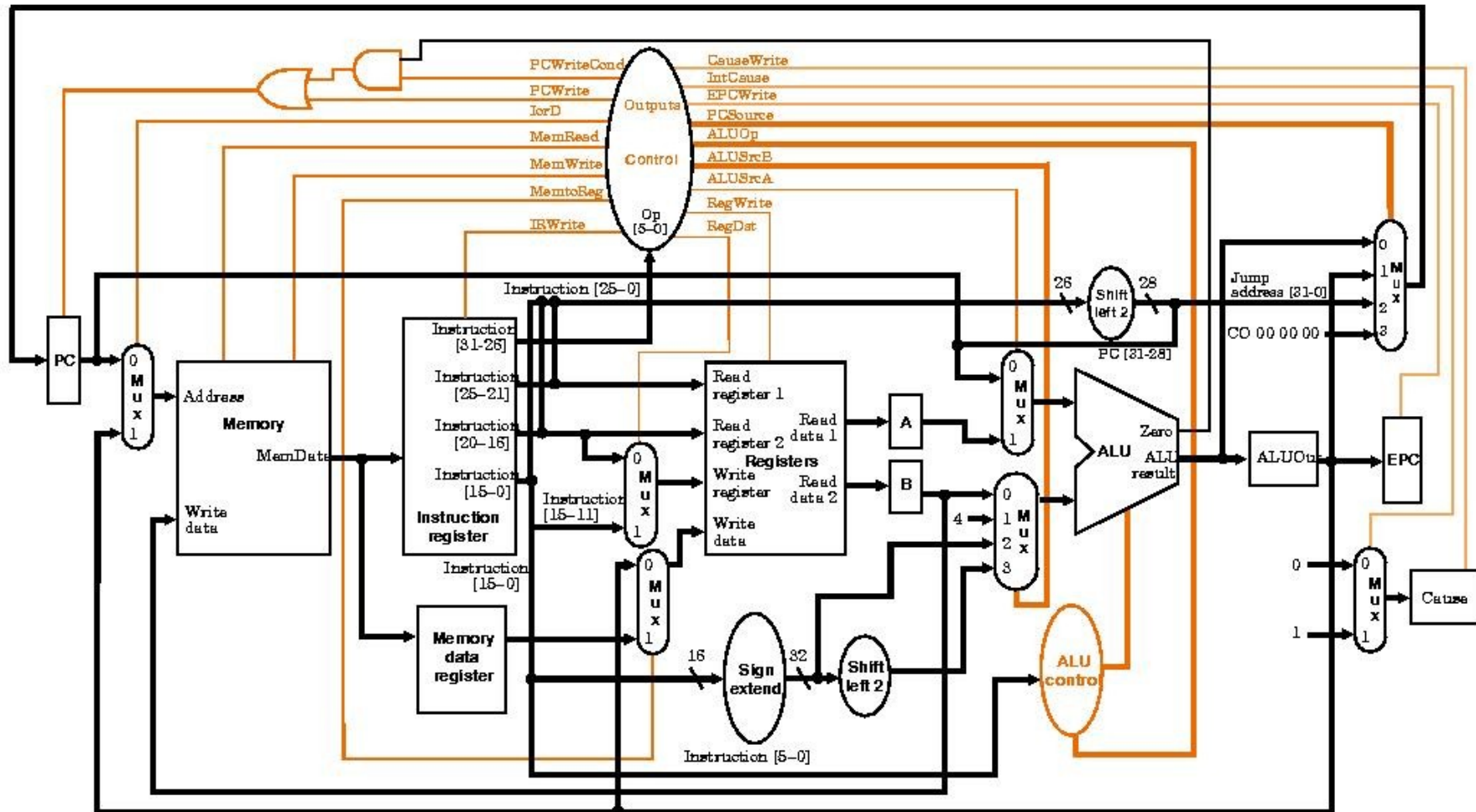
Peripherals



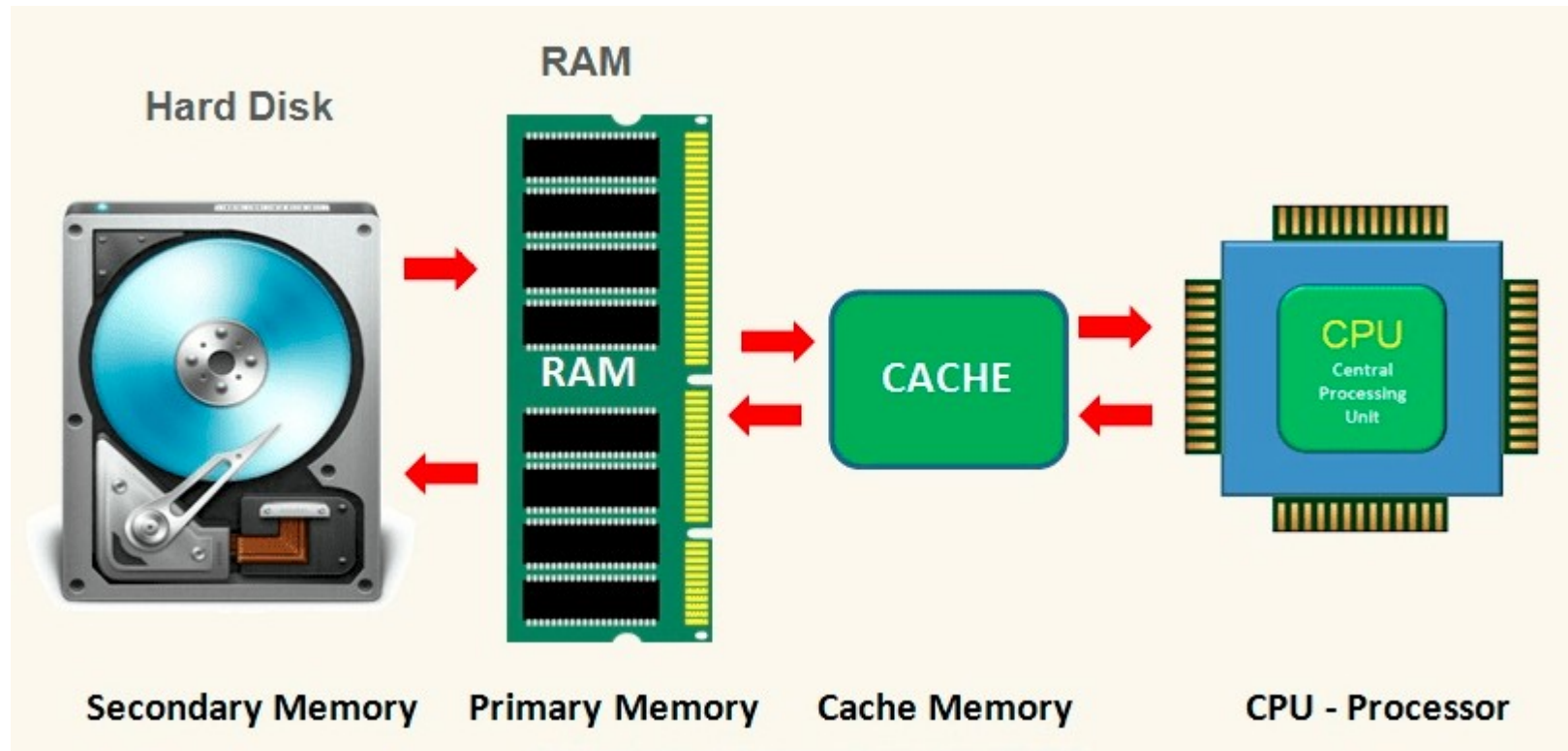
Main Goals

- General-Purpose
- Low latency

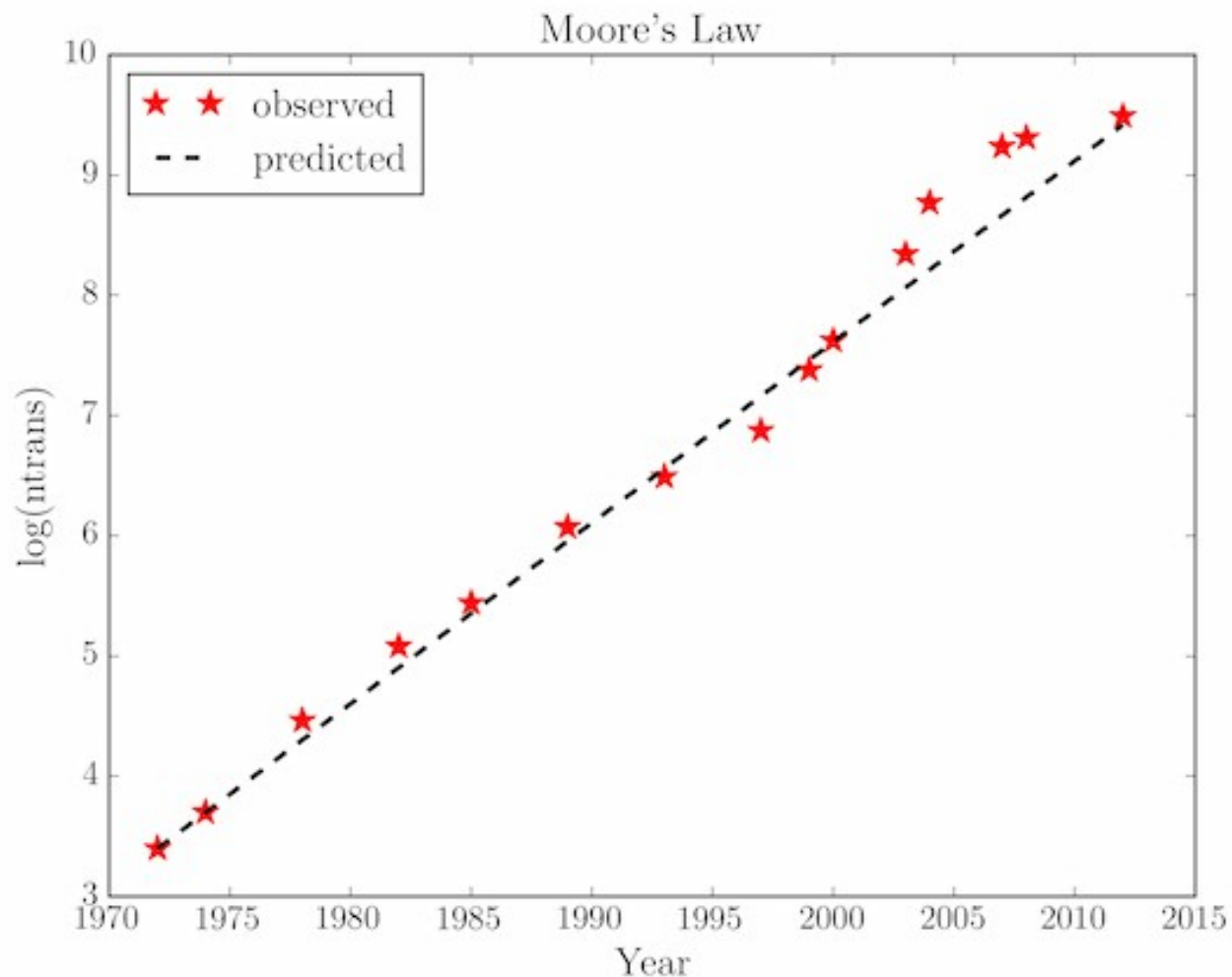
Complicated CPUs



Memory Hierarchy



Moor's Law

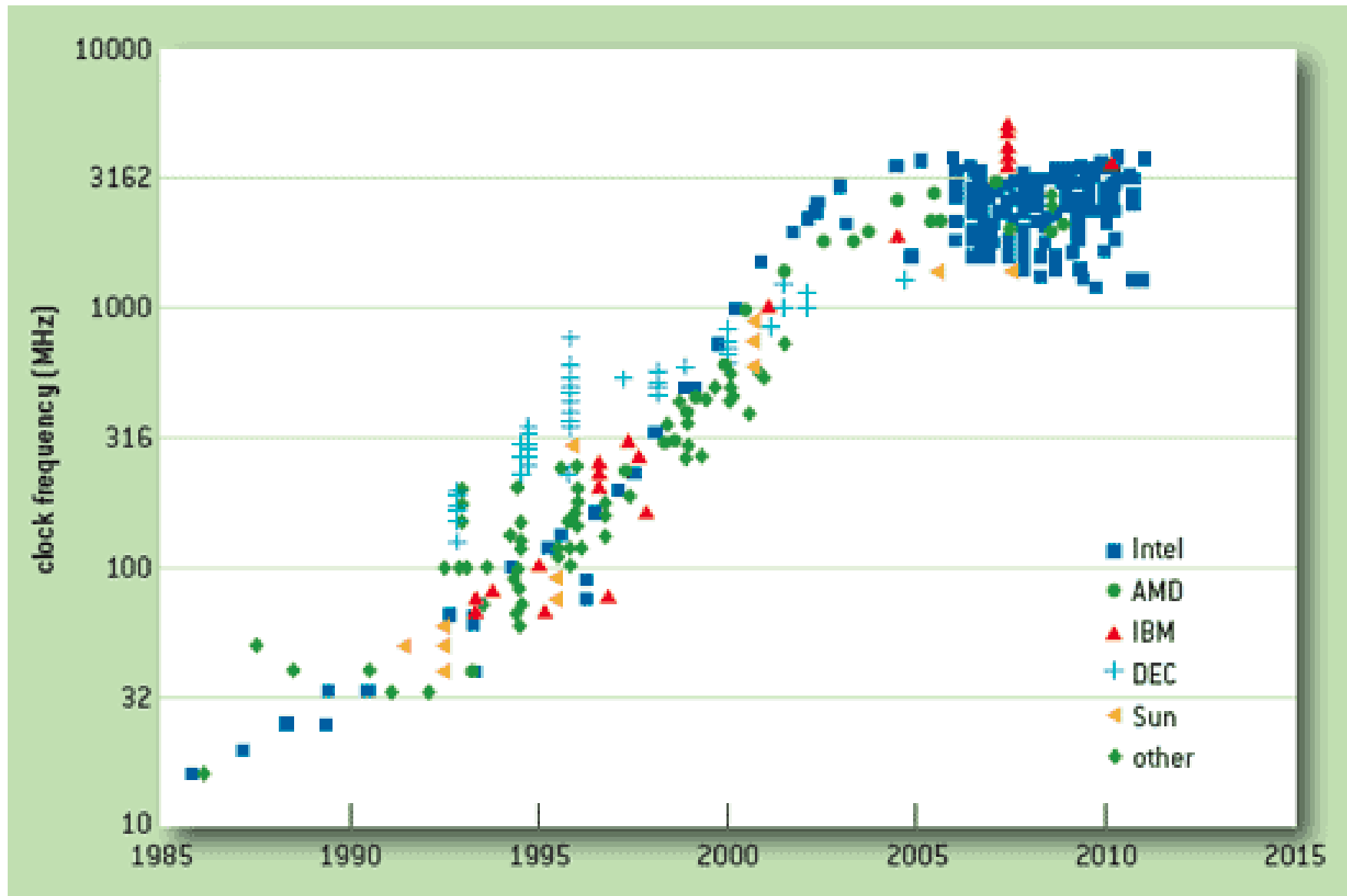


Number of transistors on a CPU chip will double every 18 months.

Dennard Scaling Law

- As transistors become smaller, their power density stays constant
- As a result of Moor's and Dennard's law:
 - **CPU manufacturers can raise clock frequency without significantly increasing overall circuit power consumption**

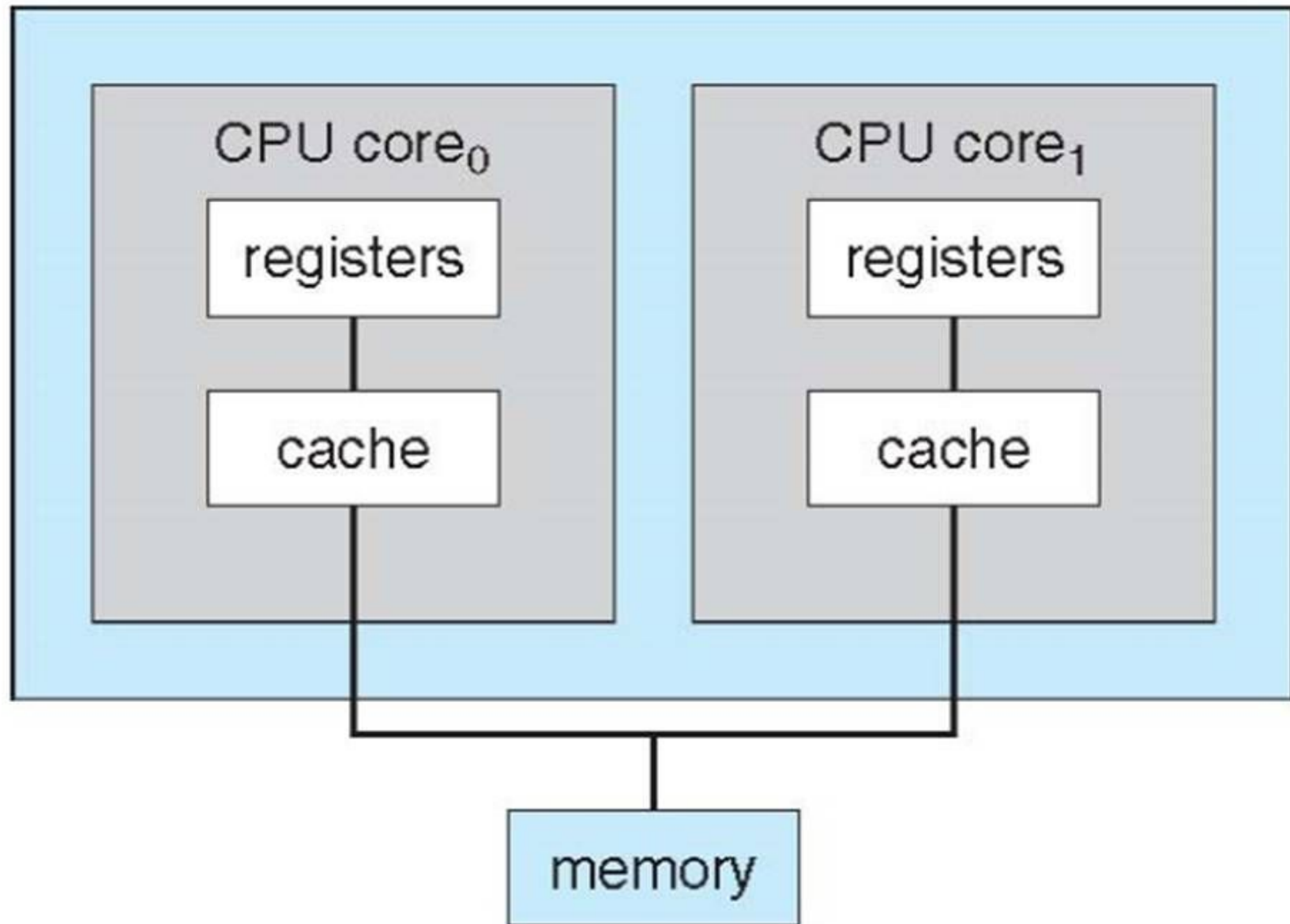
Clock Frequency



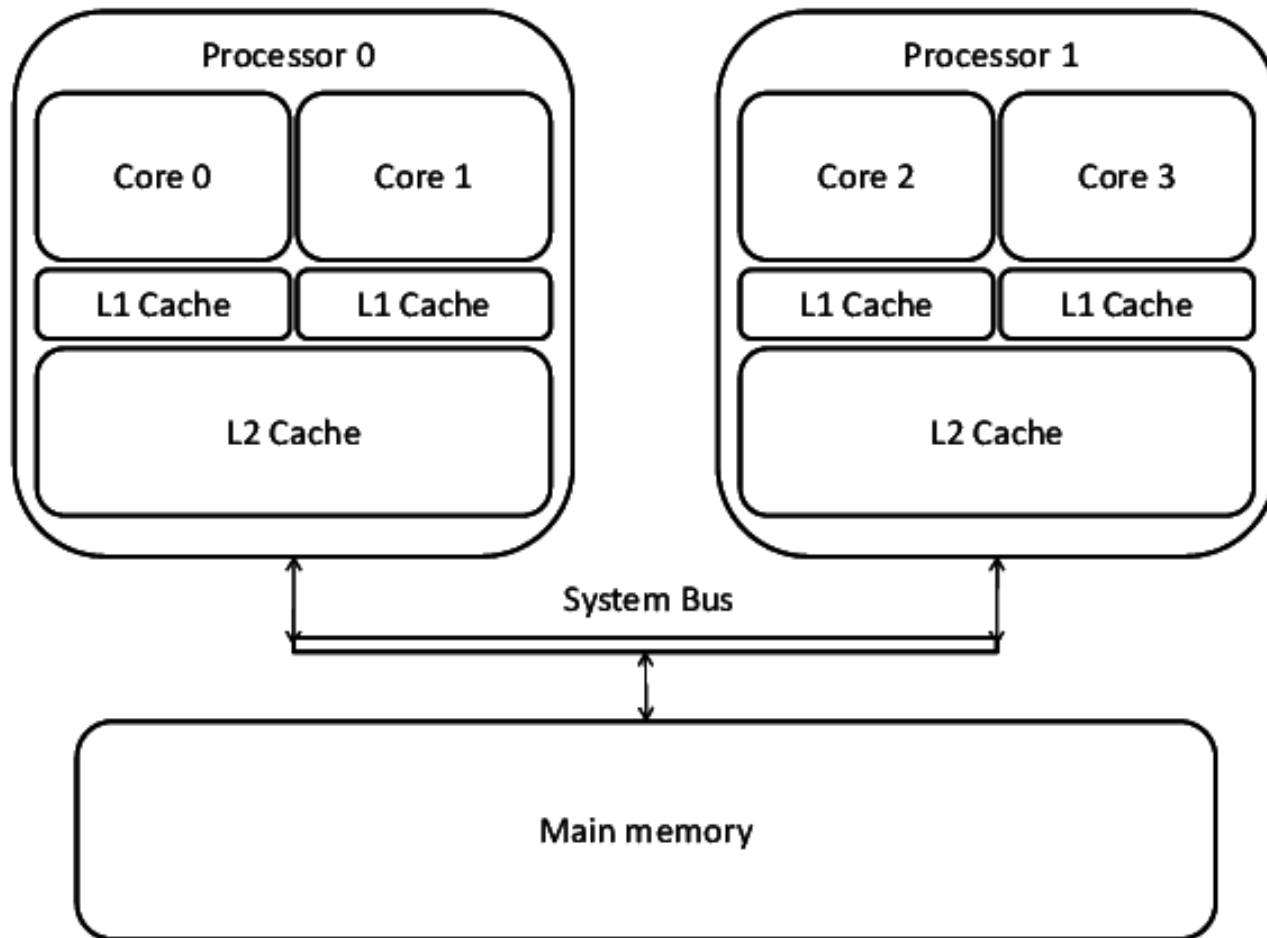
End of Moor's/Dennard's Law

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

Multi-core CPUs



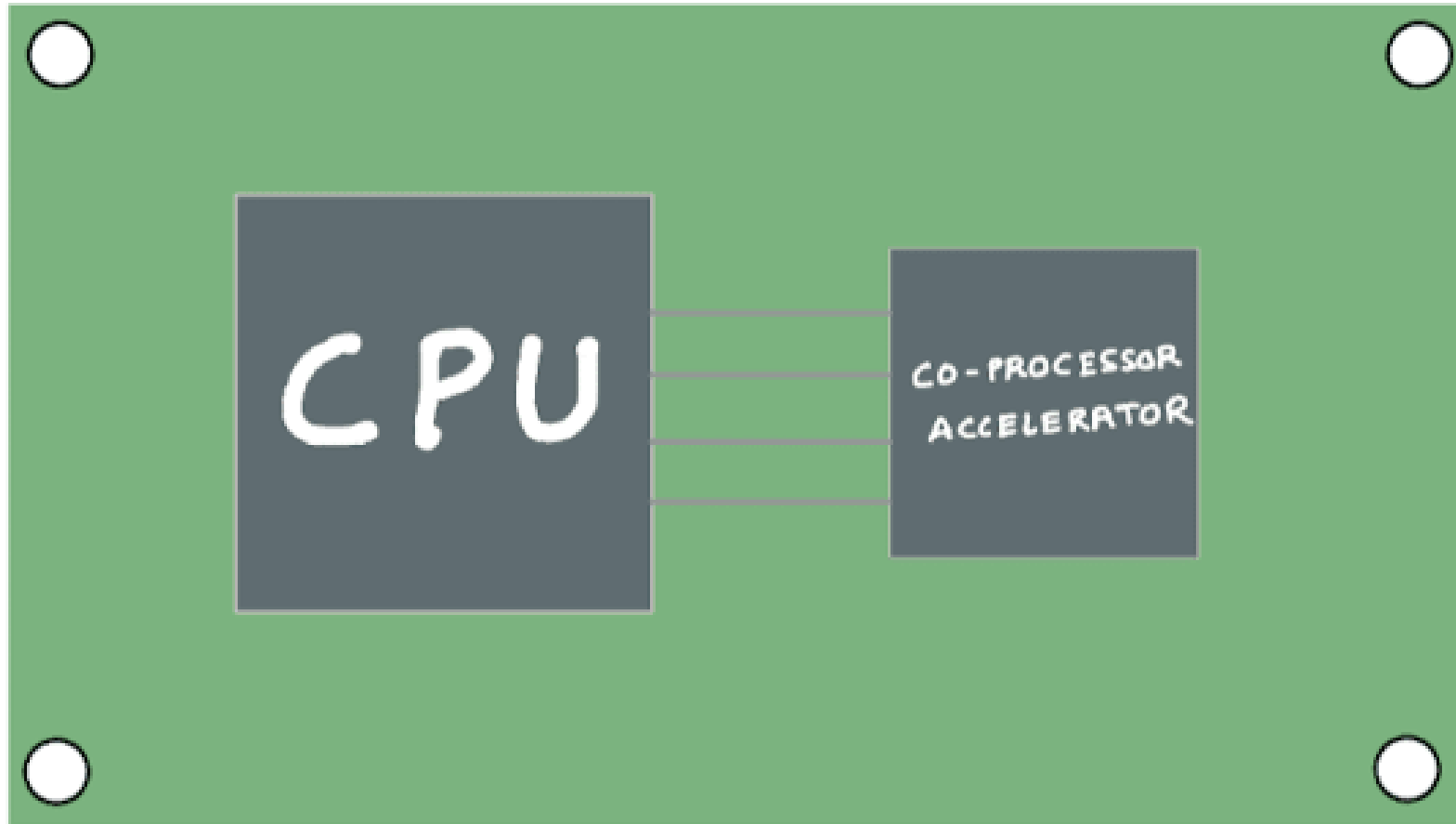
Multi-processors



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)
- ...

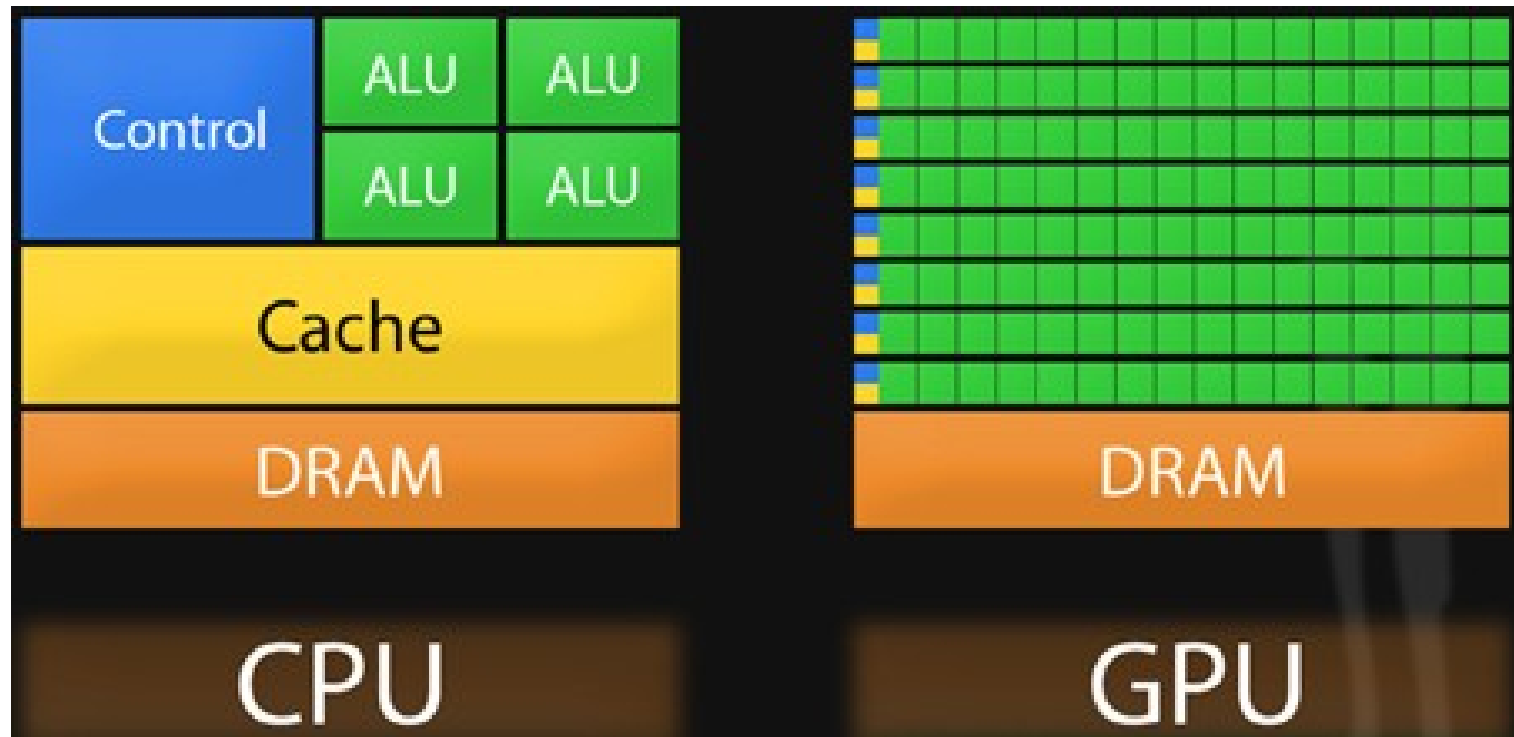
Simplere many cores

- 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	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	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
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438

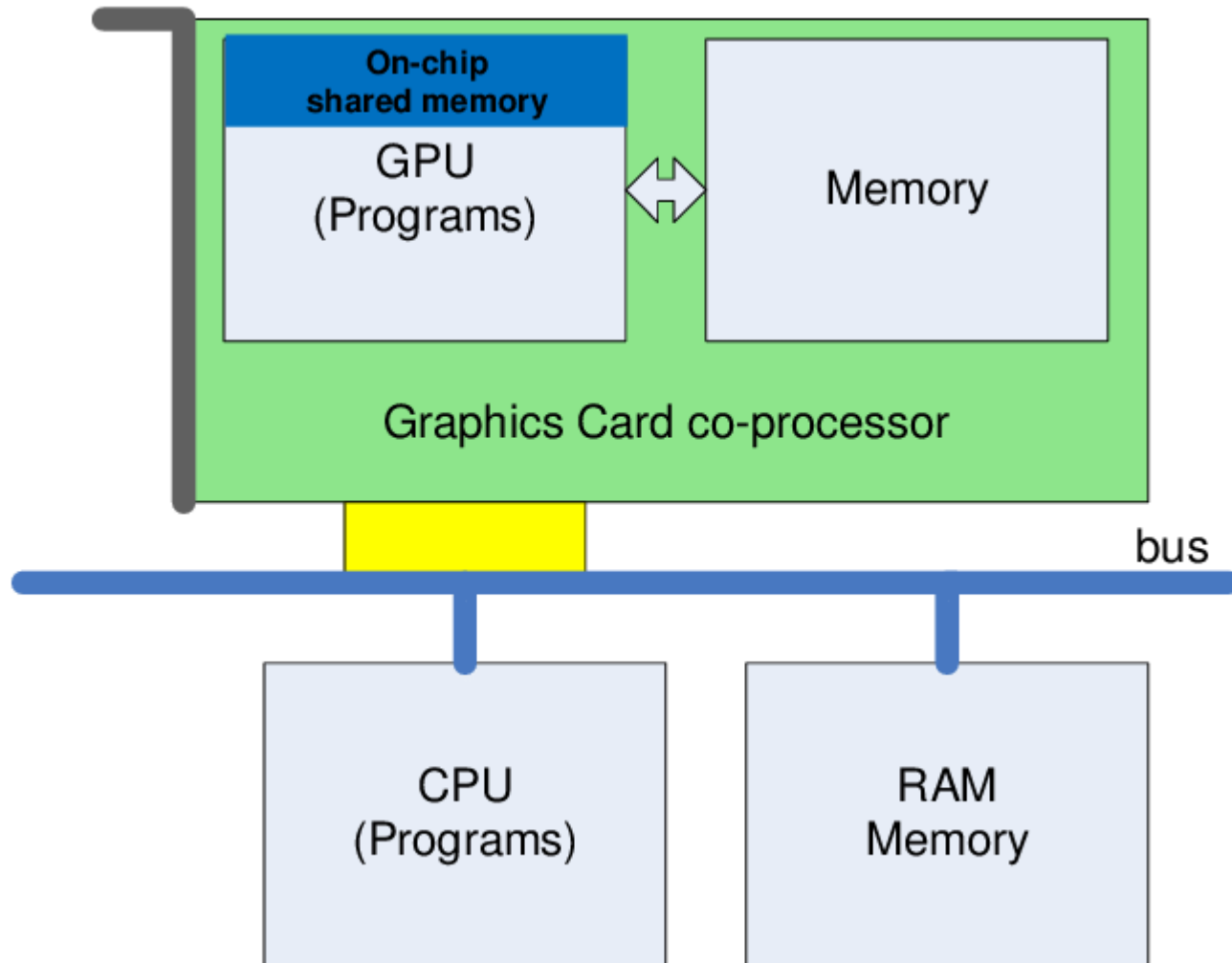
Supercomputers

6	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371
7	Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC United States	761,856	70.87	93.75	2,589
8	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63.46	79.22	2,646
9	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61.44	100.68	18,482
10	Adastra - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Grand Equipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Suprieur (GENCI-CINES) France	319,072	46.10	61.61	921

Snellius Supercomputer

- 36 GPU nodes
- Each node has 4 A100 NVIDIA GPU devices
- In total 144 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

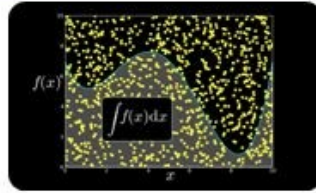
Easily Accelerate
Applications

Maximum
Flexibility

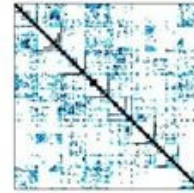
GPU Libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP

GPU VSIPL

Vector Signal
Image Processing

CULA | tools

GPU Accelerated
Linear Algebra



Matrix Algebra on
GPU and Multicore



NVIDIA cuFFT



IMSL Library



Building-block
Algorithms for CUDA

CUSP

Sparse Linear
Algebra



C++ STL Features
for CUDA



GPU Accelerated Libraries

"Drop-in" Acceleration for Your Applications

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

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

Programming
Languages

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

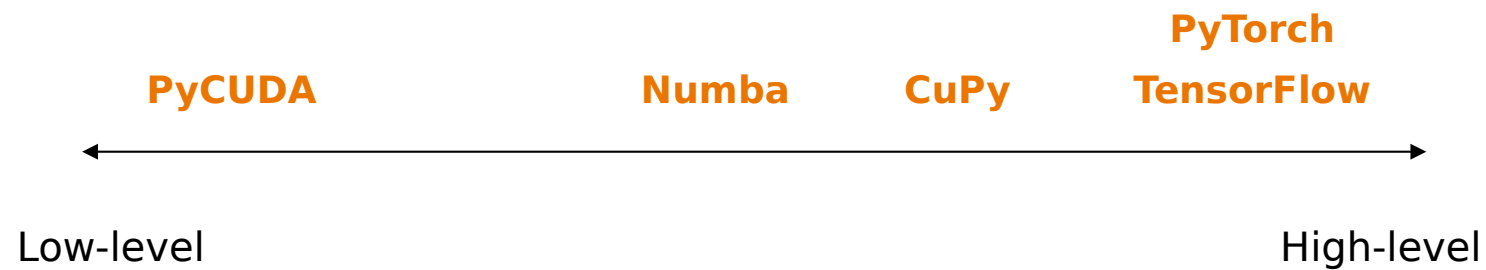
Python ▶

PyCUDA/PyOpenCL, Numba, ...

Core GPU Programming

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

Accessing to GPUs in Python



PyTorch/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



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



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

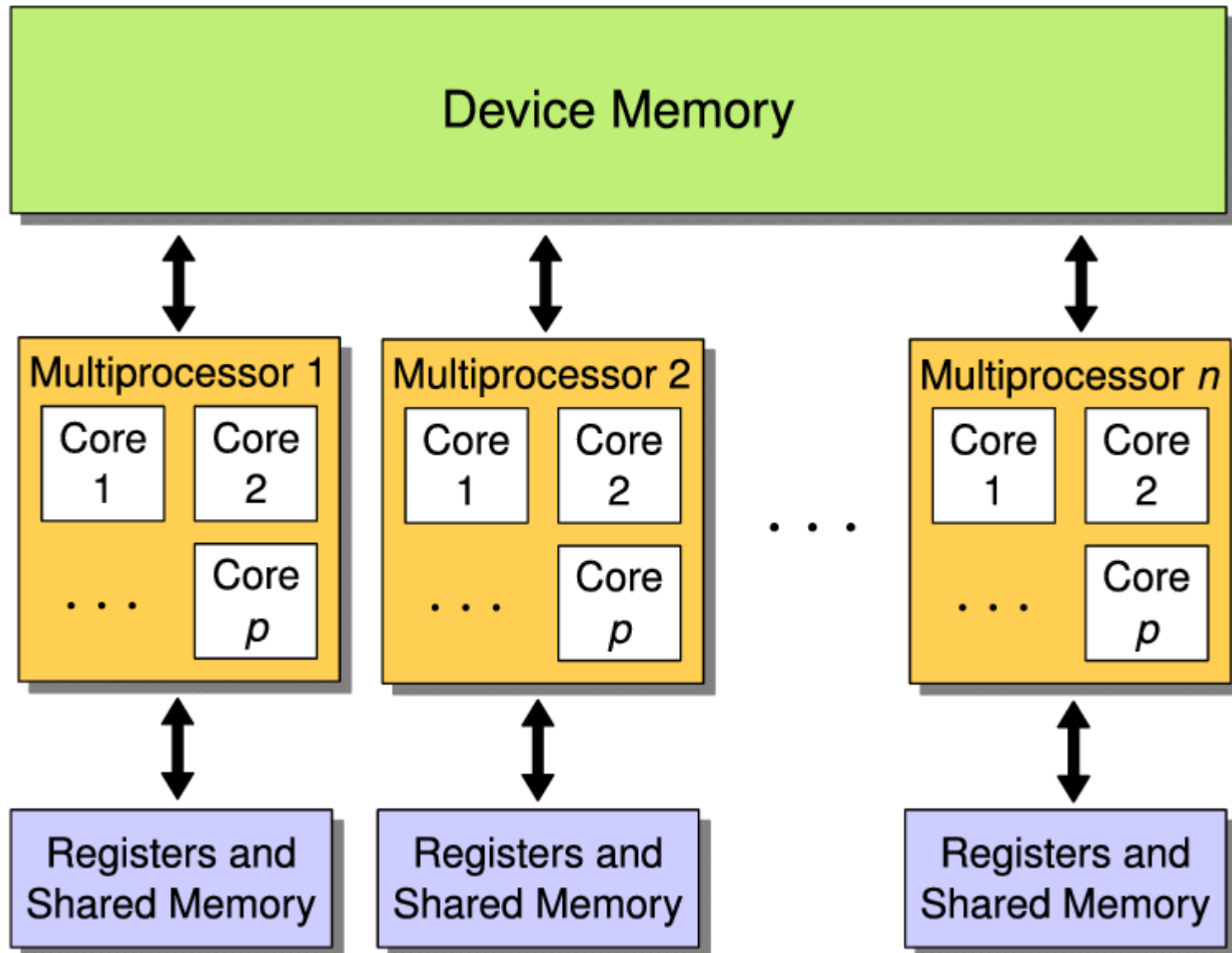
Numba

- 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

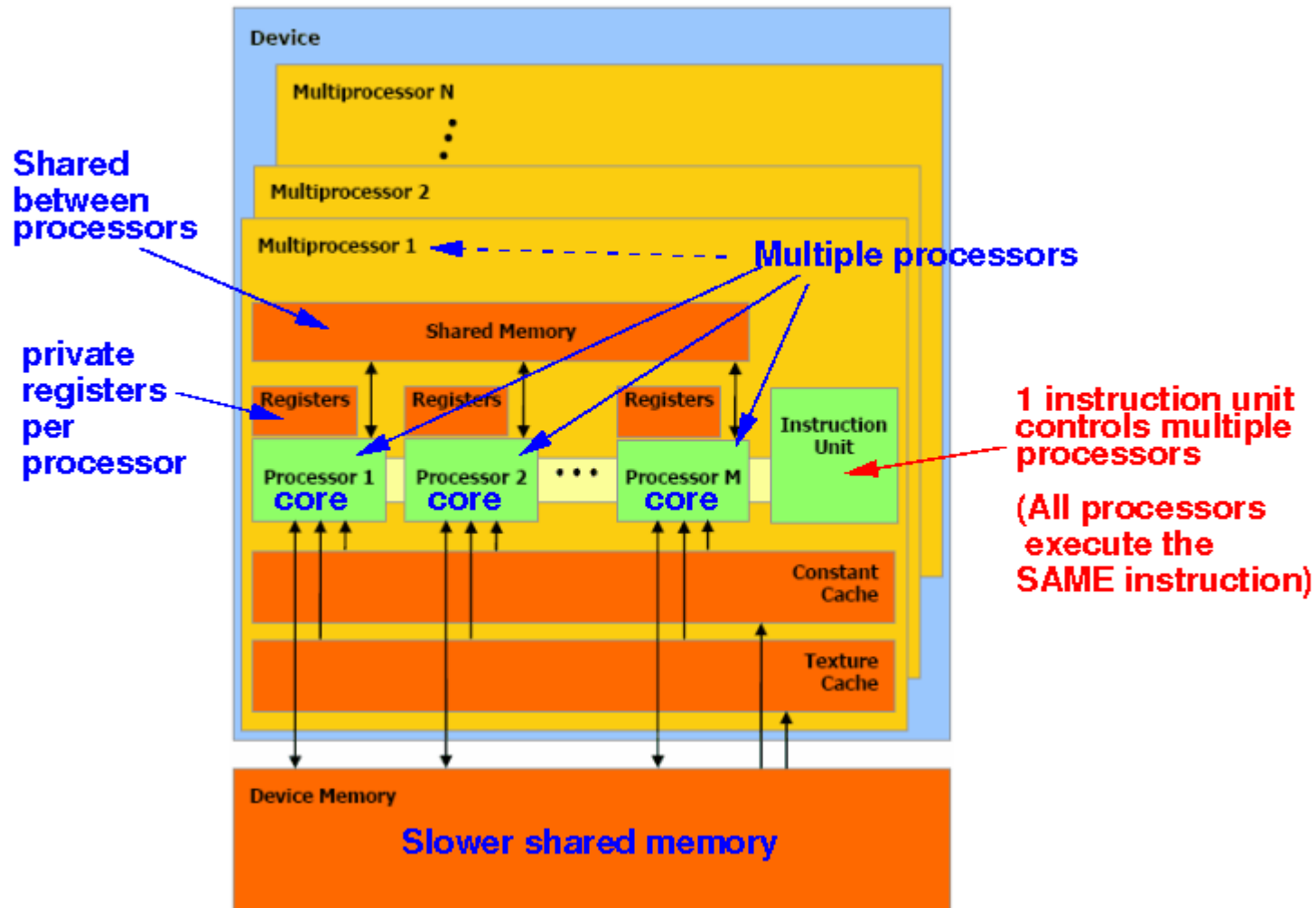
- It gives you easy, Pythonic access to NVIDIA's CUDA parallel computation API.
- There is more flexibility to write custom CUDA kernels
- For more information: <https://documen.tician.de/pycuda/>

NVIDIA GPU Hardware



NVIDIA GPU Hardware

GPU device:



Flynn's classical taxonomy

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

CUDA Programming Model

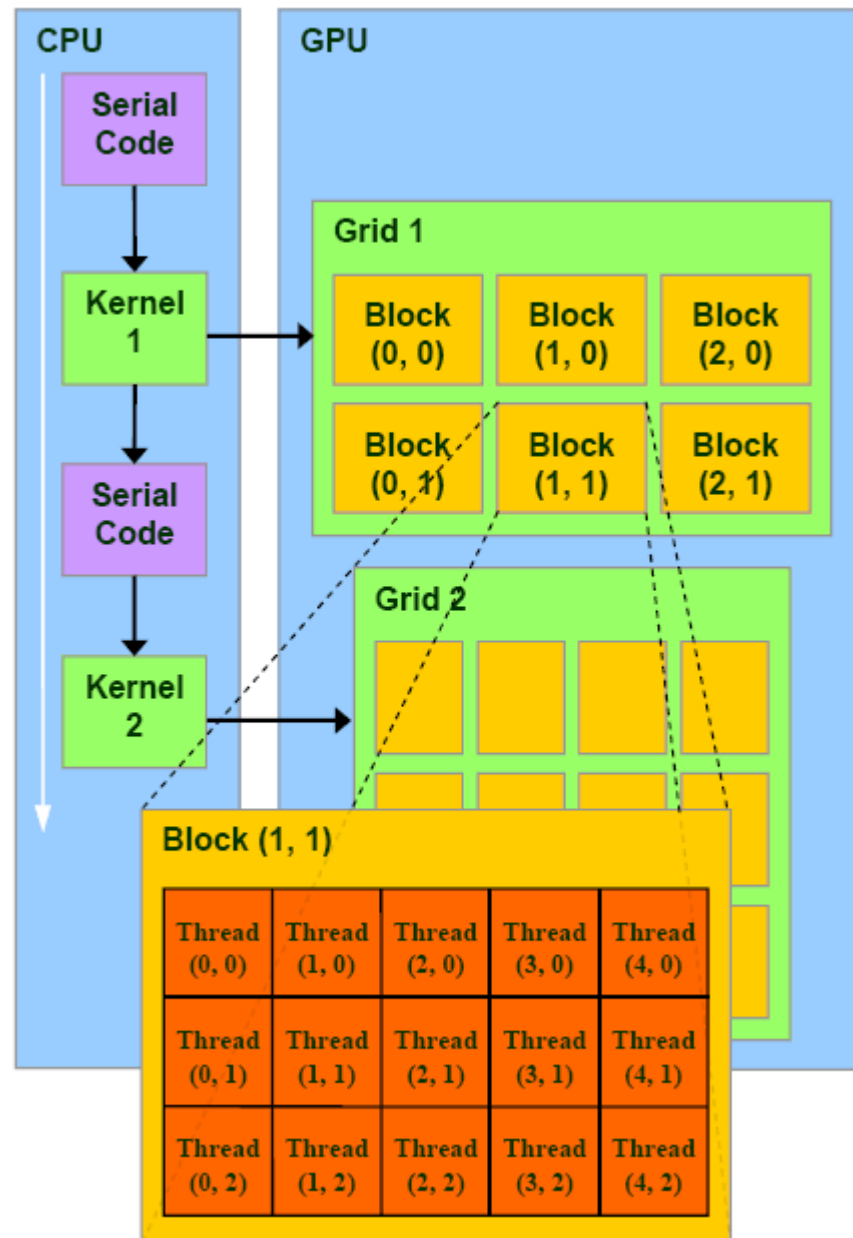
- Introduced by NVIDIA in 2006, Compute Unified Device Architecture
- General purpose programming model that leverages the parallel compute engine in NVIDIA GPUs
- An extension of C language
- CUDA programs are CPU-GPU programs:
 - CPU part is called *host*
 - GPU part is called *kernel*

CUDA Programming Model

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

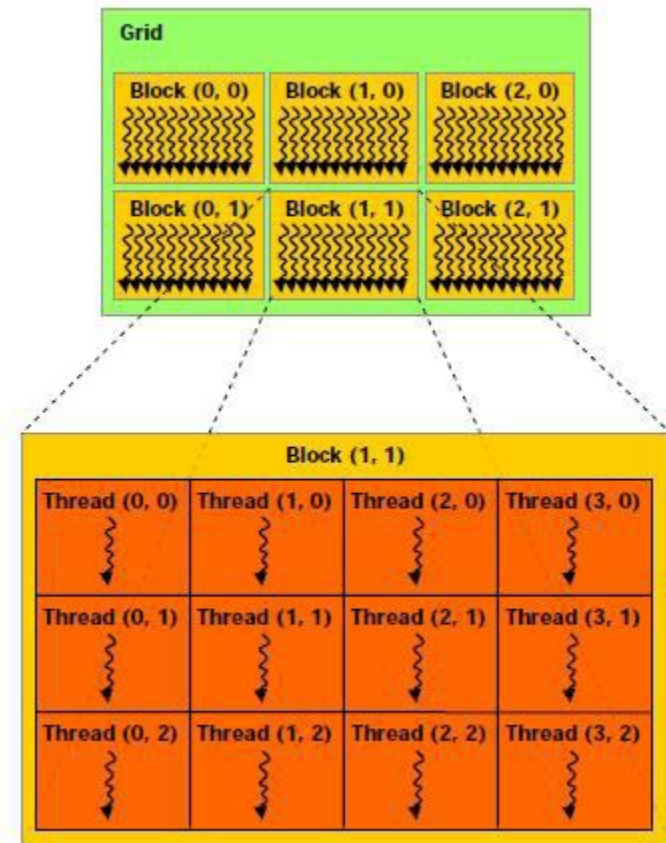
- Copy the input data from host memory to device memory, also known as host-to-device transfer
- Call the kernel from host and execute the GPU program
- Copy the results from device memory to host memory, also called device-to-host transfer

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

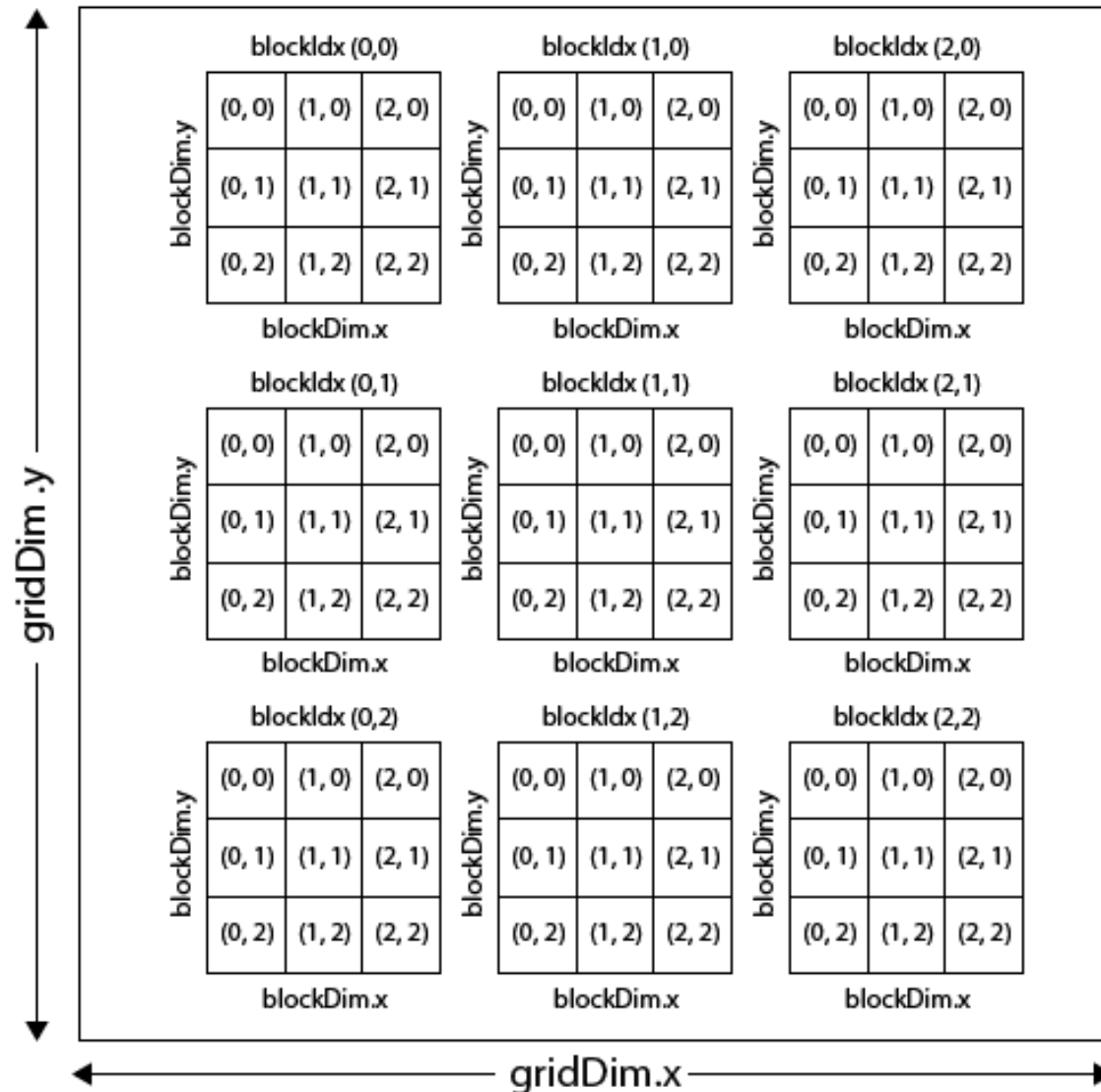
- Built-in functions:

- 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`

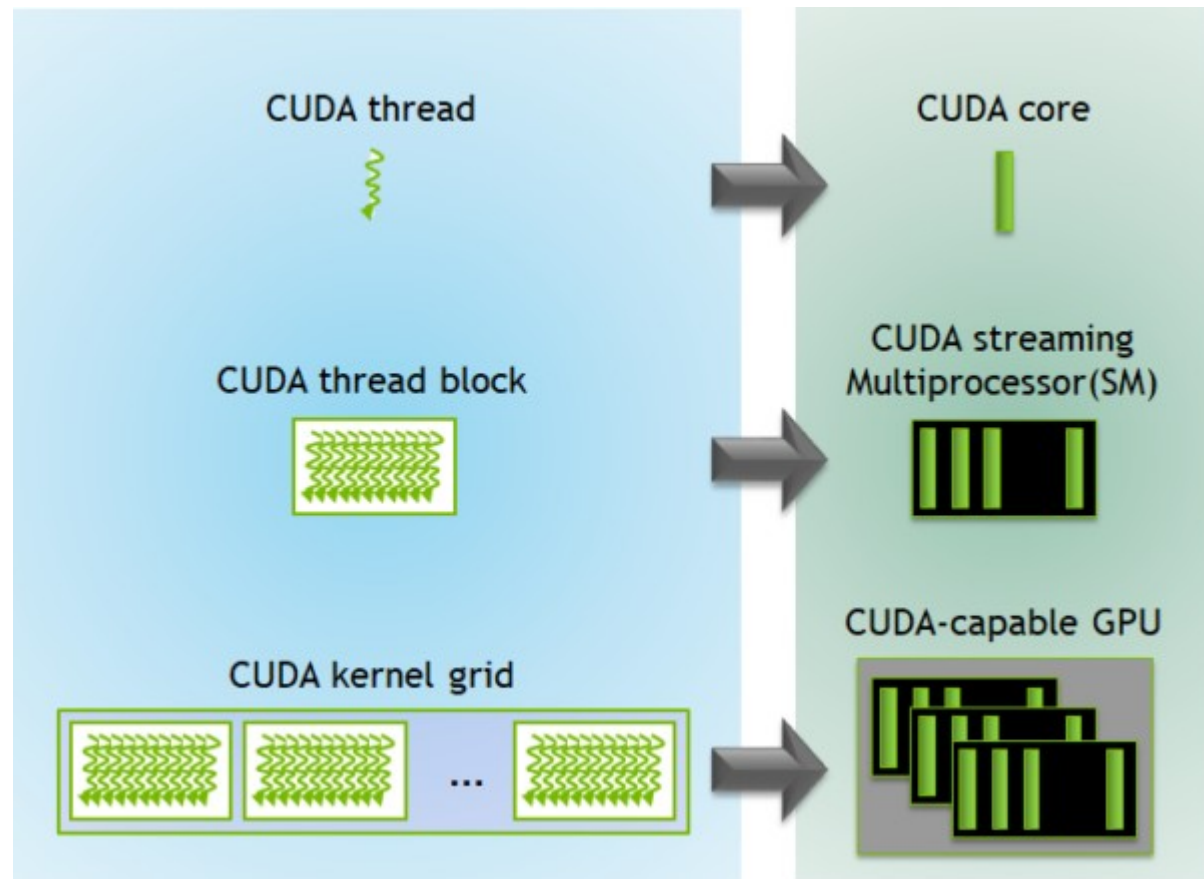
CUDA Programming Model

CUDA Grid

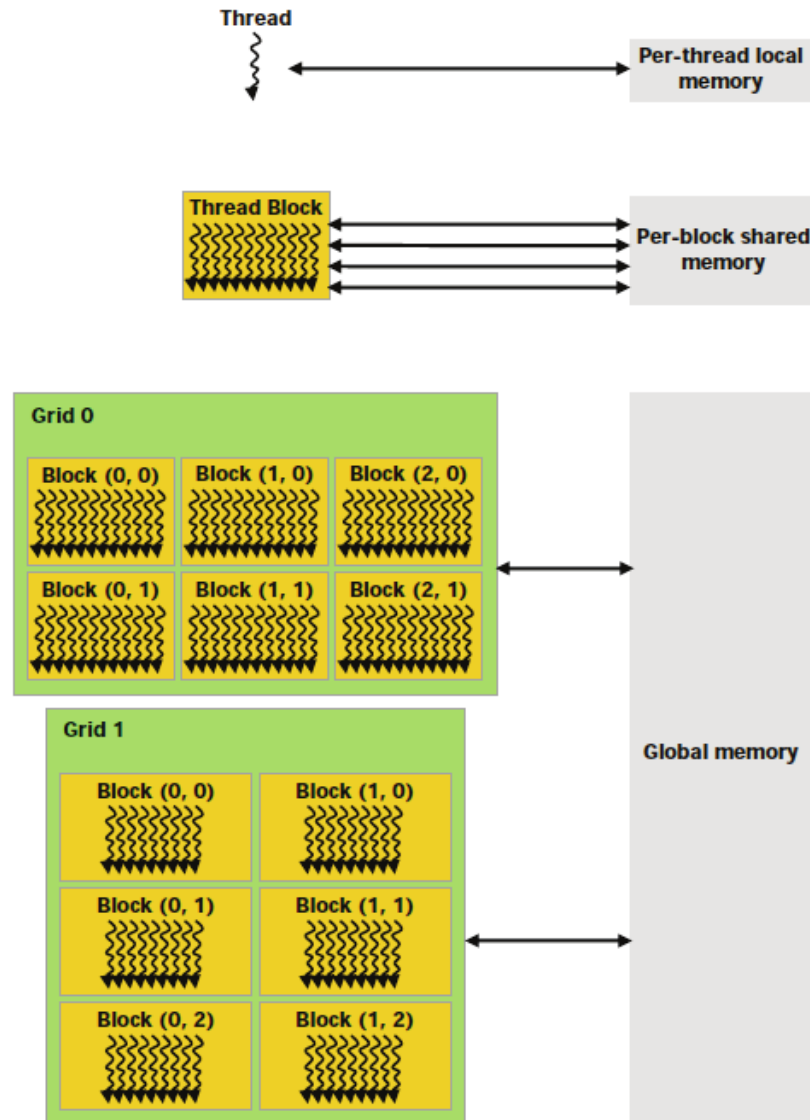
- gridDim.x = 3
- gridDim.y = 3
- blockDim.x = 3
- blockDim.y = 3



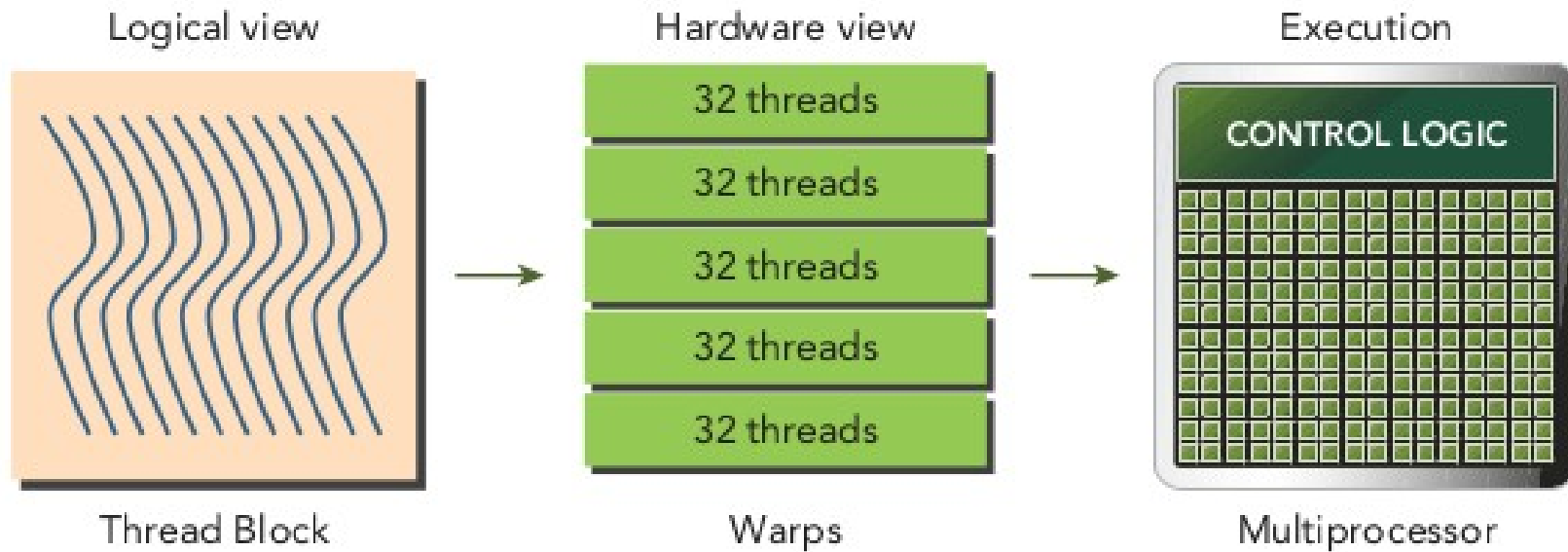
CUDA Execution Model



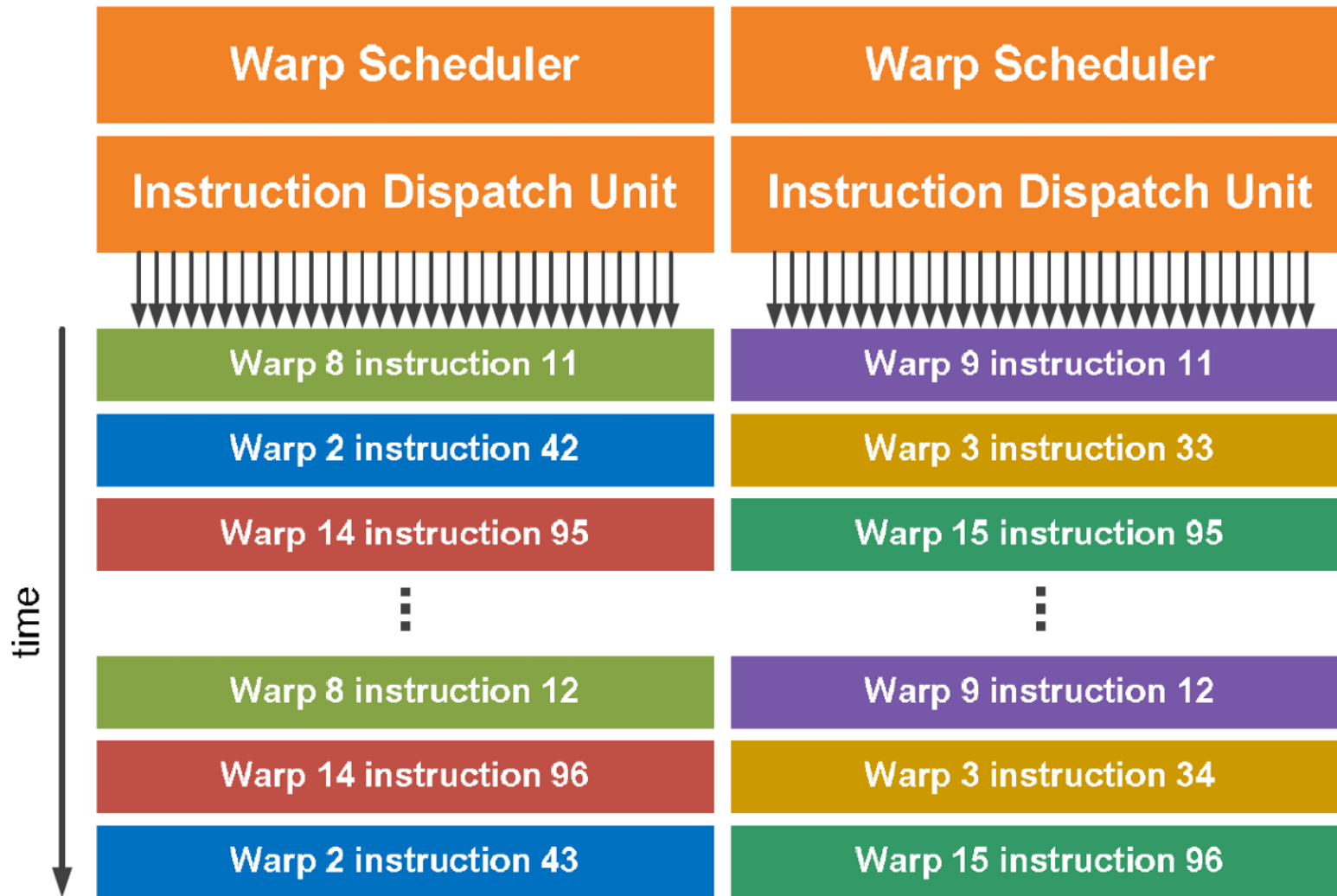
CUDA Execution Model



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 Titan RTX GPUs per node
 - Multiprocessors: 72
 - Streaming 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 $(35,000 * 35,000) = 35,000 * 35,000 * 4 \approx 5 \text{ GB}$

First Example:

Parallel Vector (1D array) Addition in
PyCUDA

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

Global Thread ID

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

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

PyCUDA Implementation

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

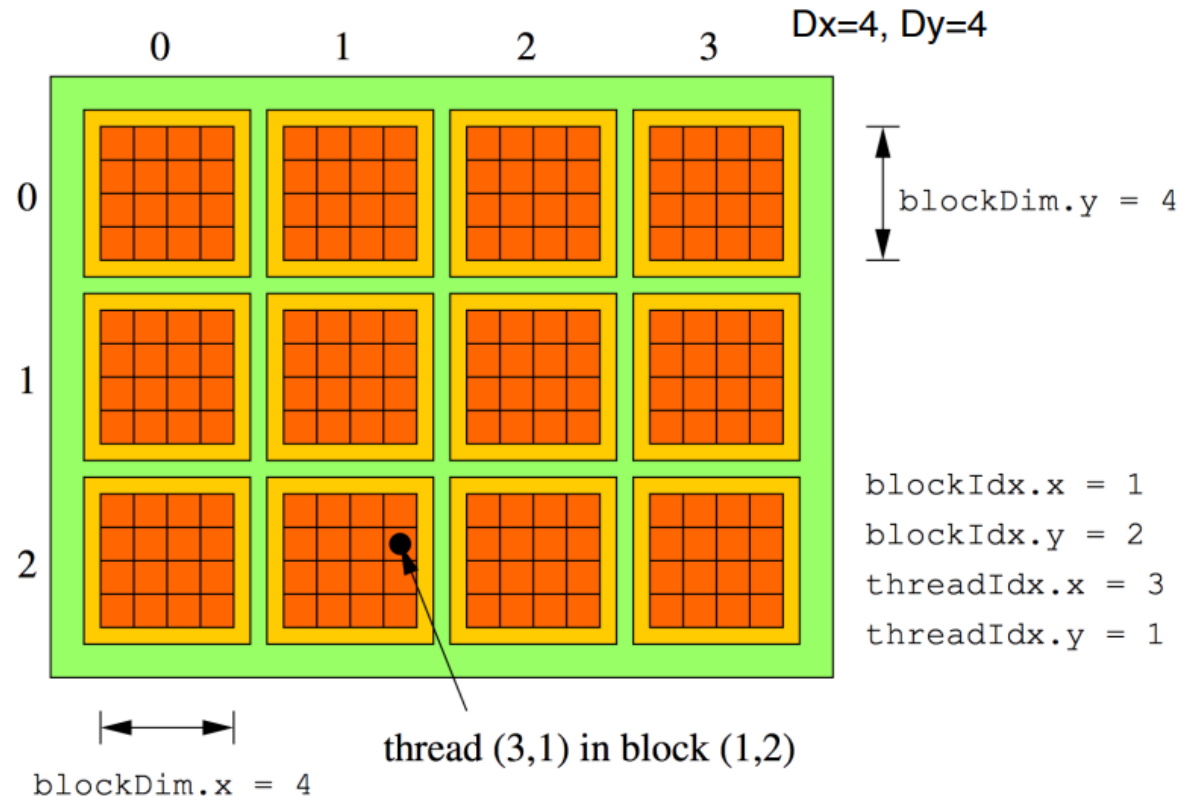
Automatic Data Transfer

- 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 = $1 * 3 + 2 = 5$

How we see a 2D array

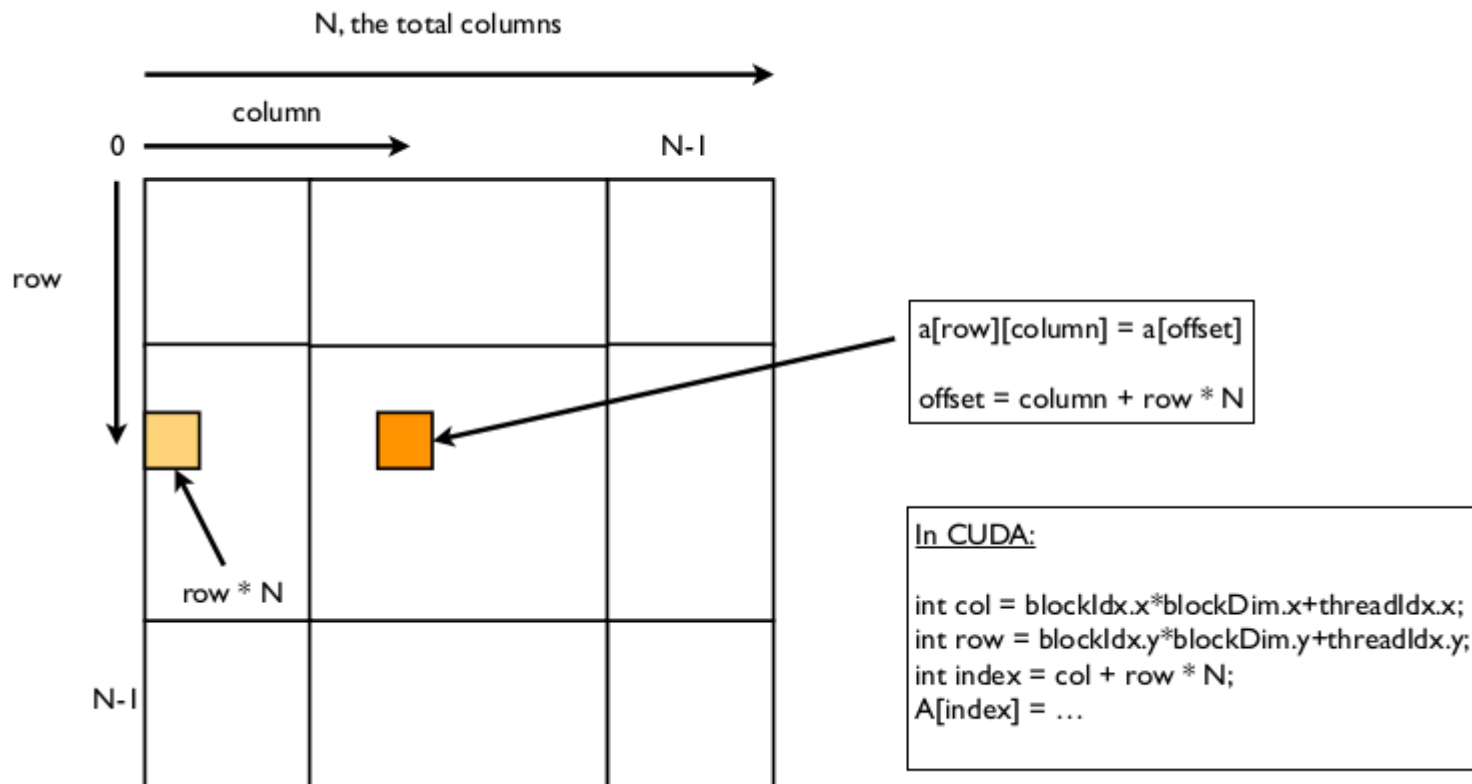
0	1	2
3	4	5
6	7	8



How it's stored in memory

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

Row-Major Flattening of a Matrix



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

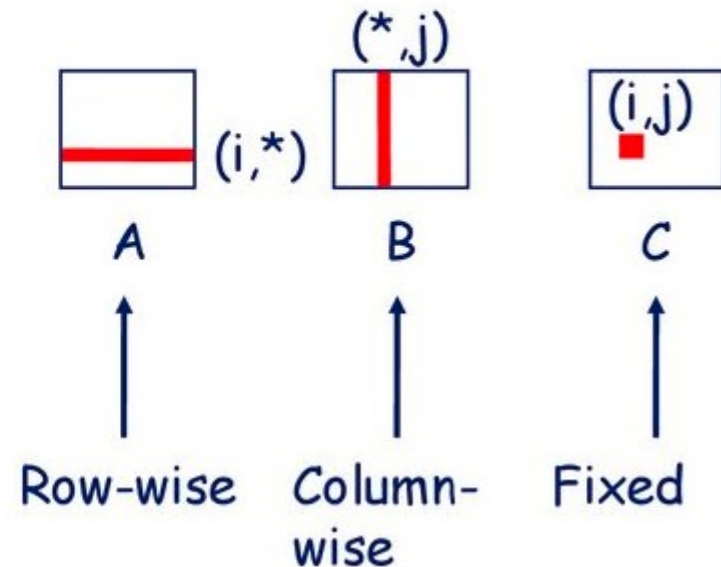
Third Example:

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:



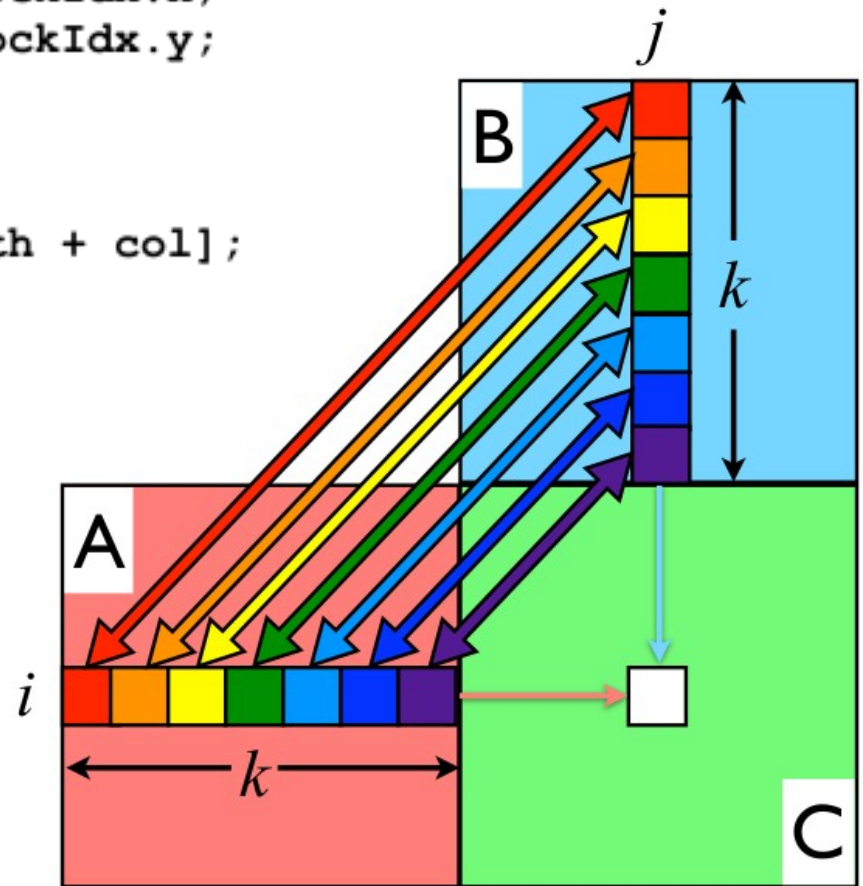
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}_A \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}_B = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}_C$$

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;
}
```

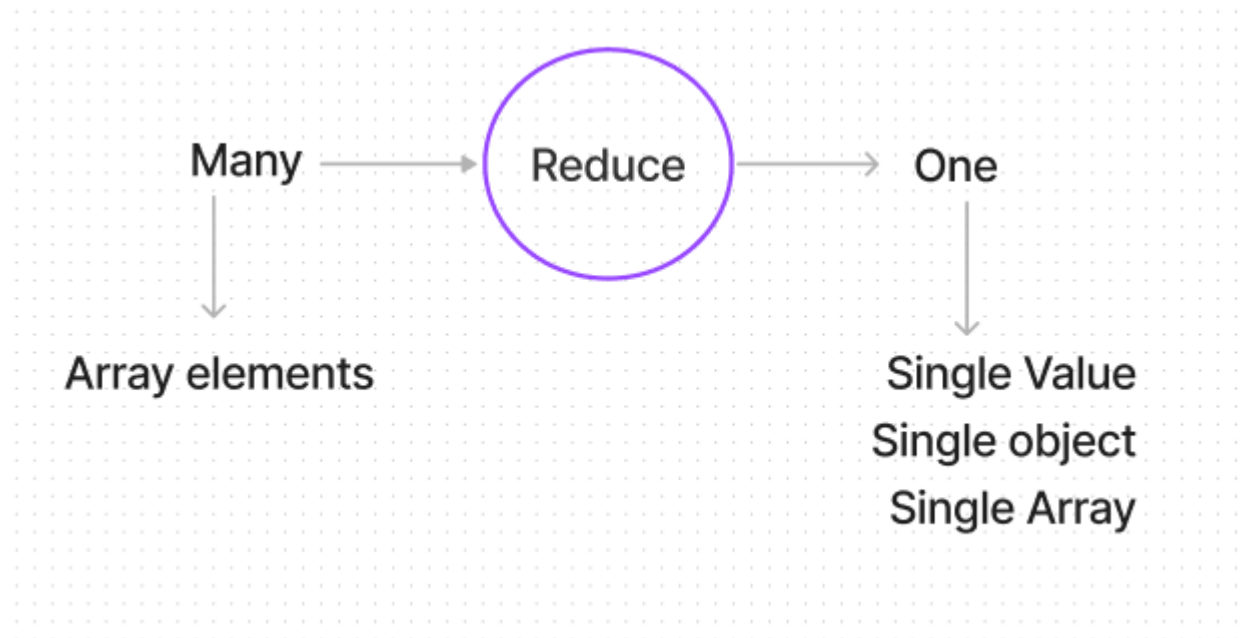


PyCUDA Implementation

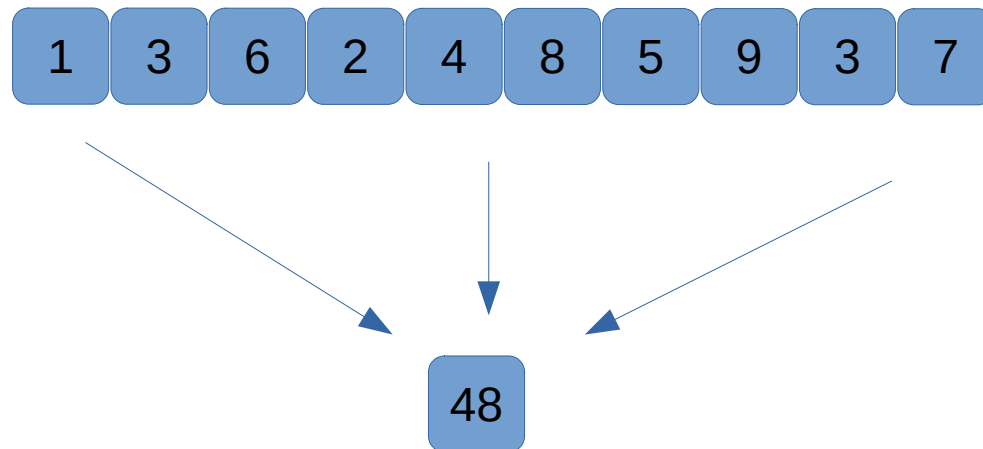
- Implement matrix multiplication in PyCUDA
- Compare its execution time to
 - Sequential CPU-based
 - Numpy.matmul()
 - @ operator

Fourth Example: Reduction in PyCUDA

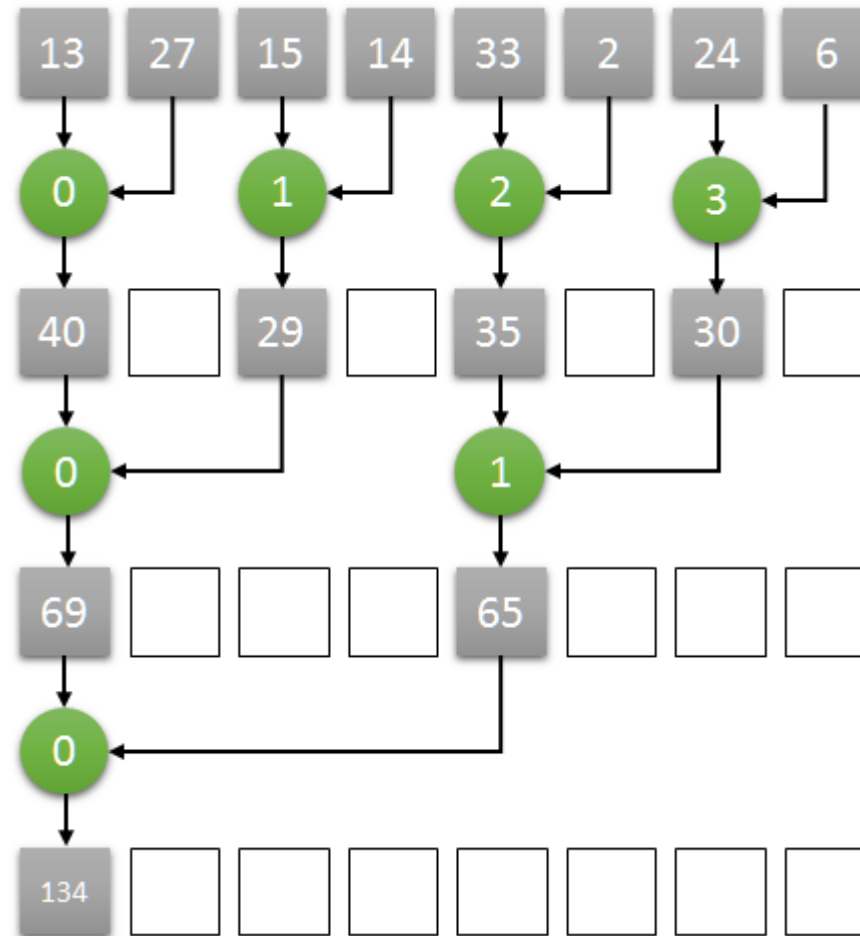
Reduction



Reduction (addition)



Reduction (addition)



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.)

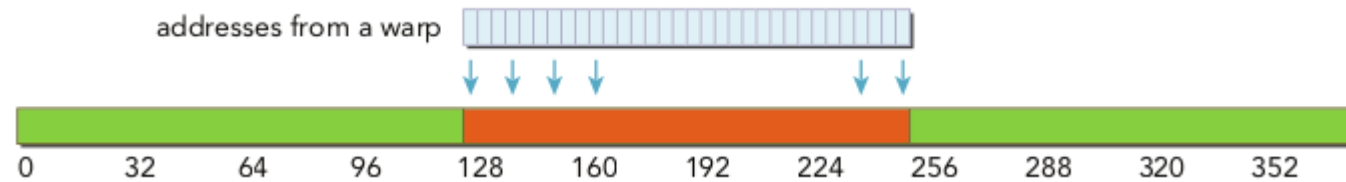
Optimization

There are different ways to optimize CUDA codes:

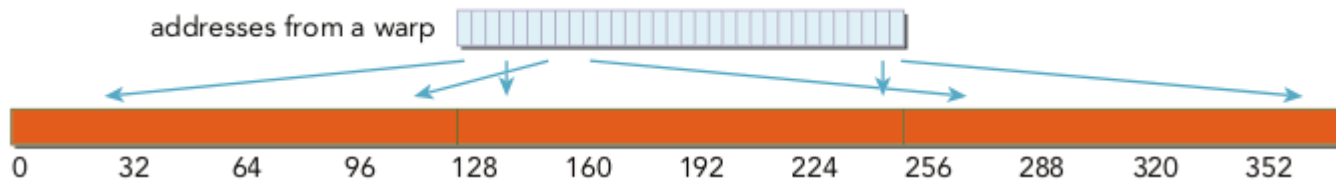
- Number of threads per block
- Workload per thread
- Total work per thread block
- Correct memory access and data locality
- ...

Tips for Optimization

- Global Memory Access:



Coalesced

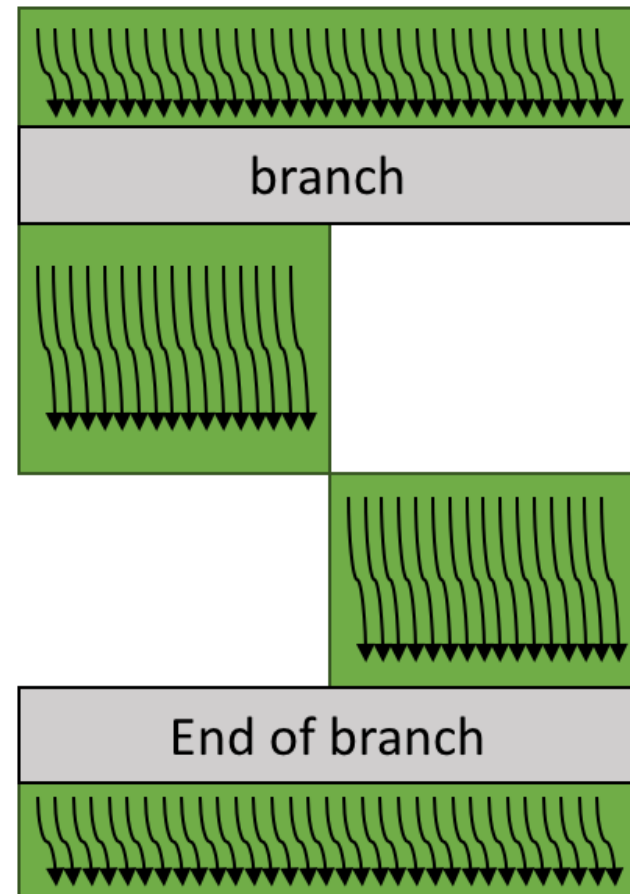


Non-coalesced

Tips for Optimization

- Avoid Warp Divergence:

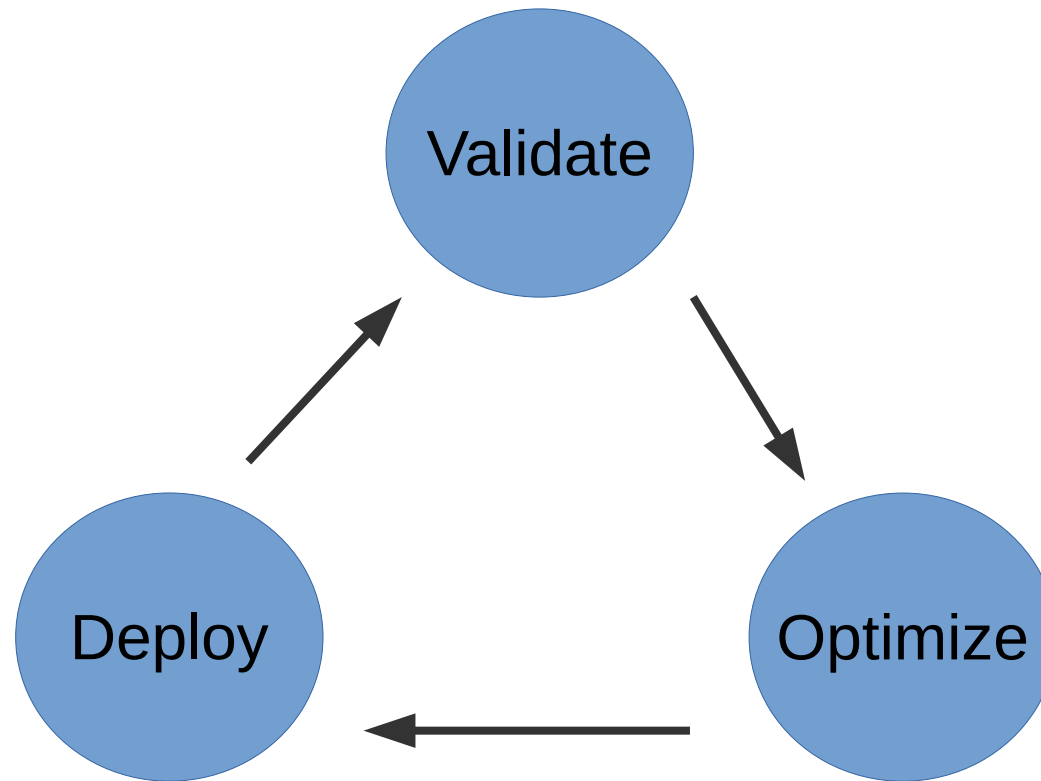
```
...  
if ( threadIdx.x < 16 )  
{  
    ... A ...  
}  
else  
{  
    ... B ...  
}  
...
```



Tips for Optimization

- Use shared memory in two cases:
 - When threads in a block need to share 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?