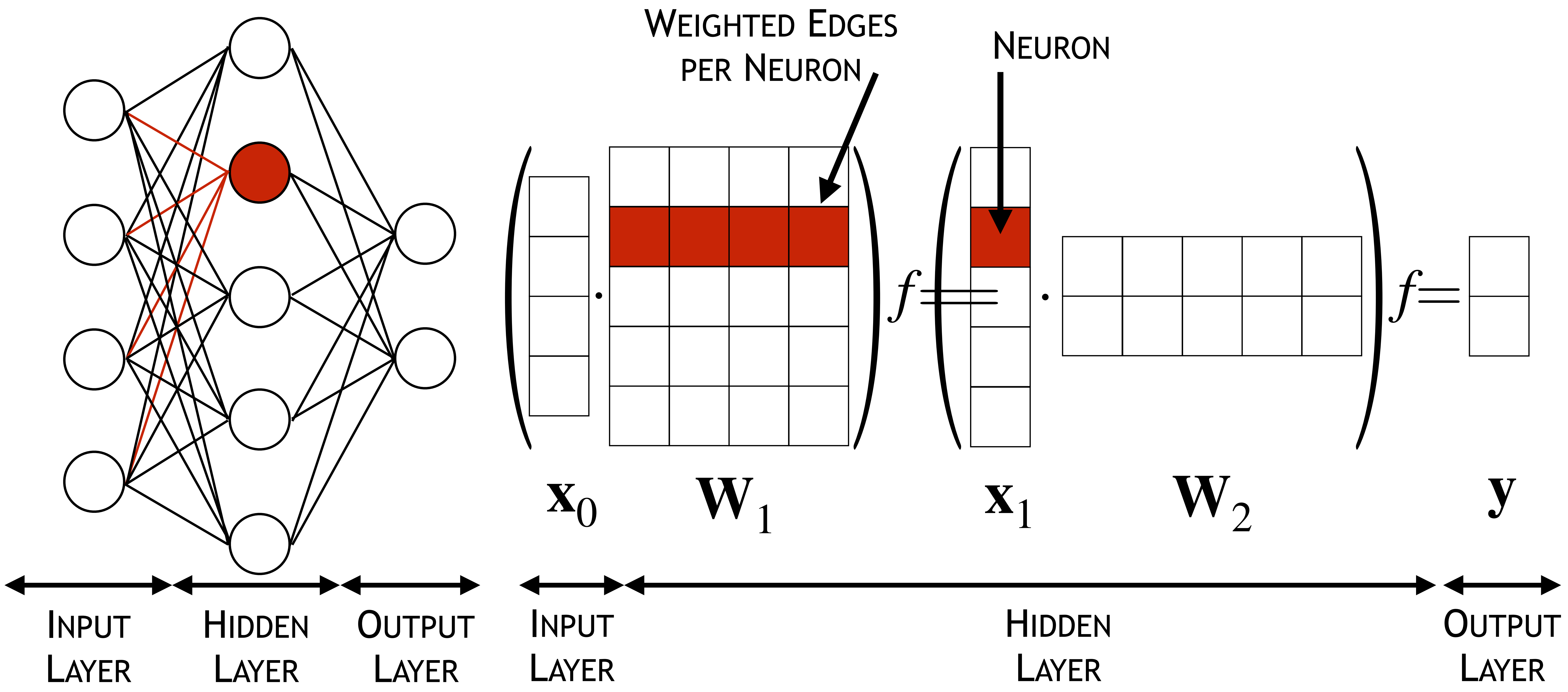
ANFÄNGERPRAKTIKUM
NEURAL NETWORKS FROM SCRATCH

GPUS AND NEURAL NETWORKS

Hendrik Borras, Franz Kevin Stehle
hendrik.borras@ziti.uni-heidelberg.de, kevin.stehle@ziti.uni-heidelberg.de
HAWAII Group, Institute of Computer Engineering
Heidelberg University

REMINDER: NEURAL NETWORKS ARE MASSIVE MATRIX MULTIPLY CONSTRUCTS



How do we execute this quickly and thus make it scalable?

GPU COMPUTING

GPU BACKGROUND

Primary use in gaming

- Each console has a (powerful) GPU
- Meantime photorealistic

NVIDIA

Graphics: big, multi-dimensional floating-point operations in parallel

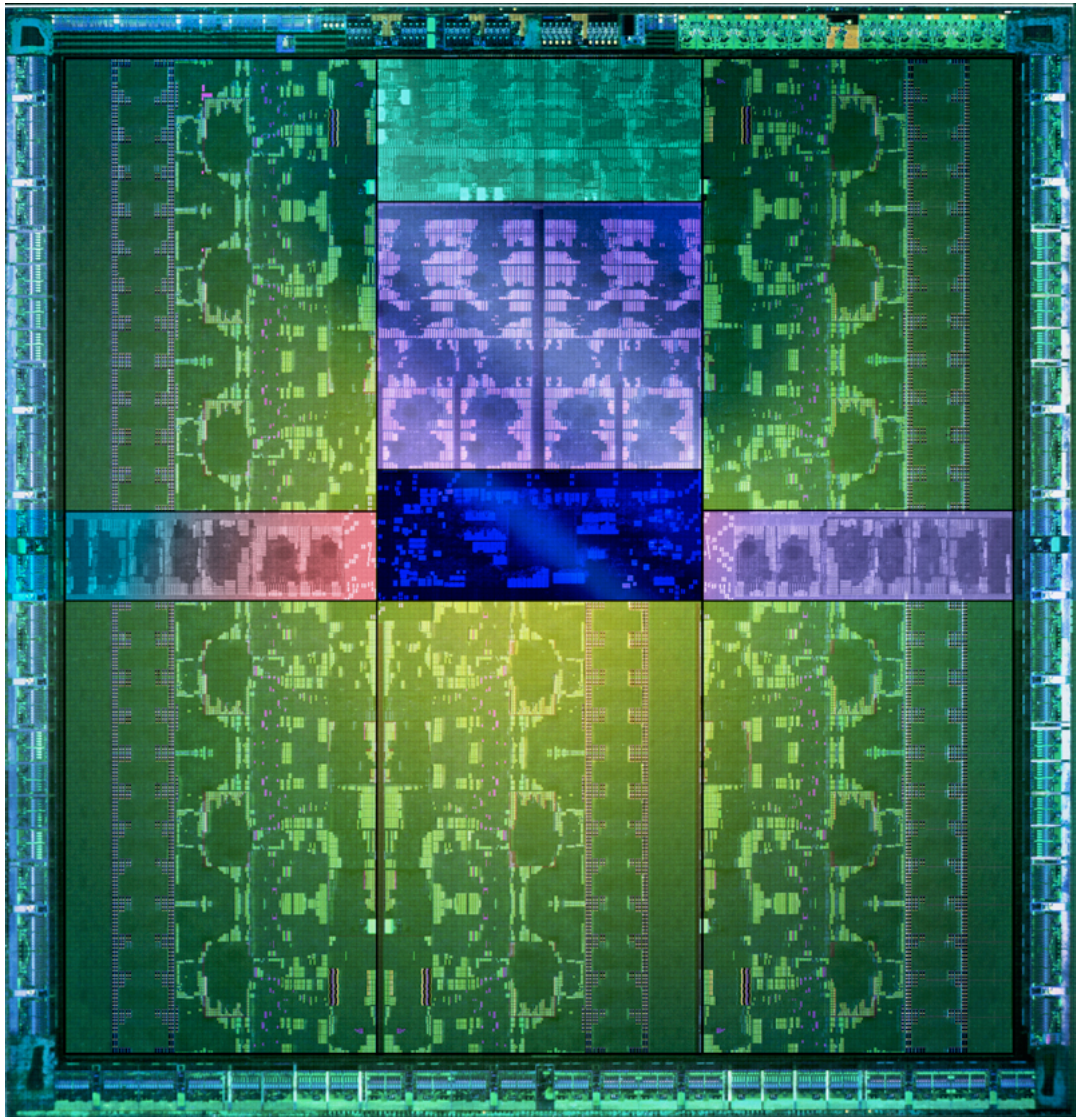
- Programmable

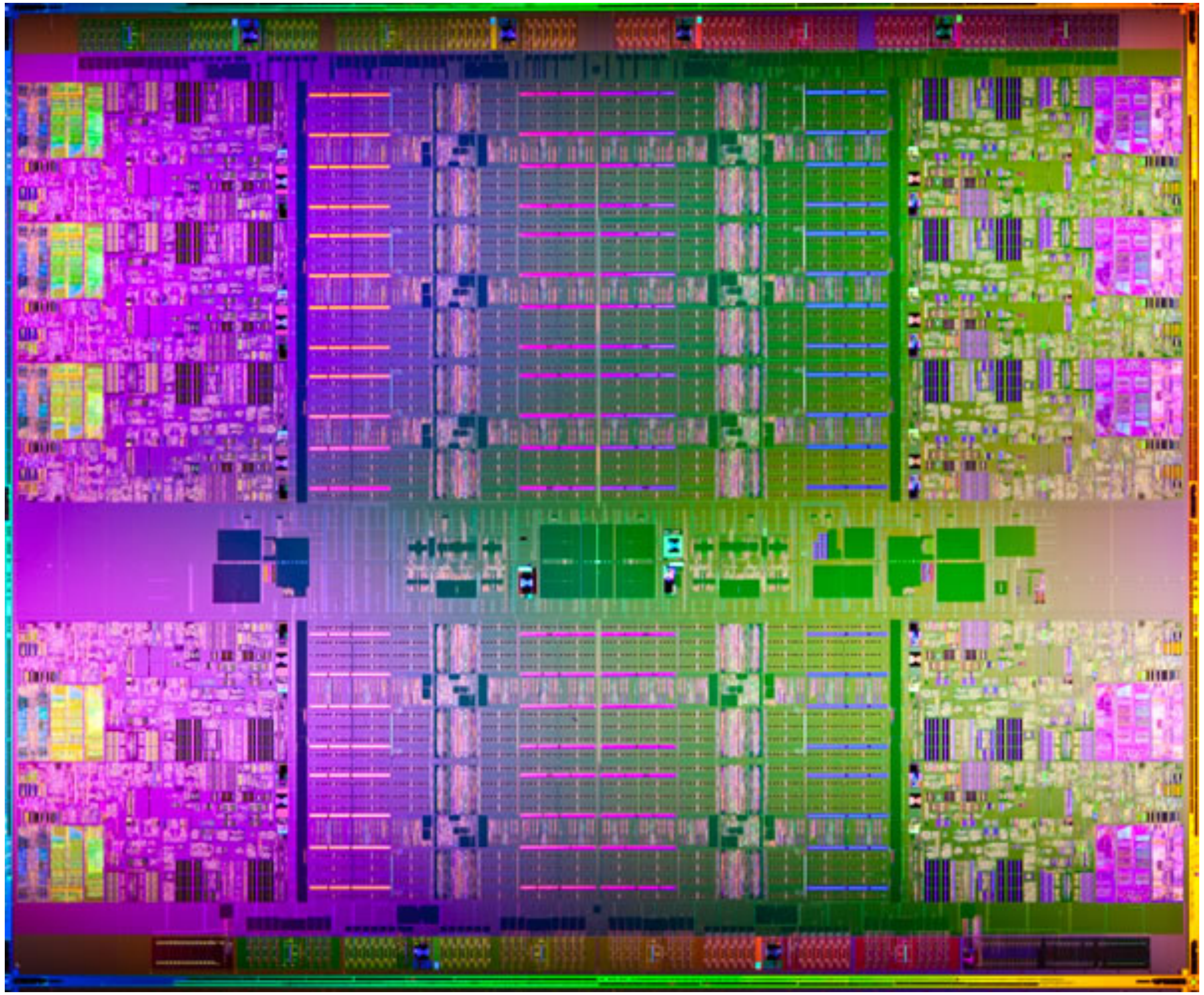
Since ~2007 used for general-purpose computing

- CUDA

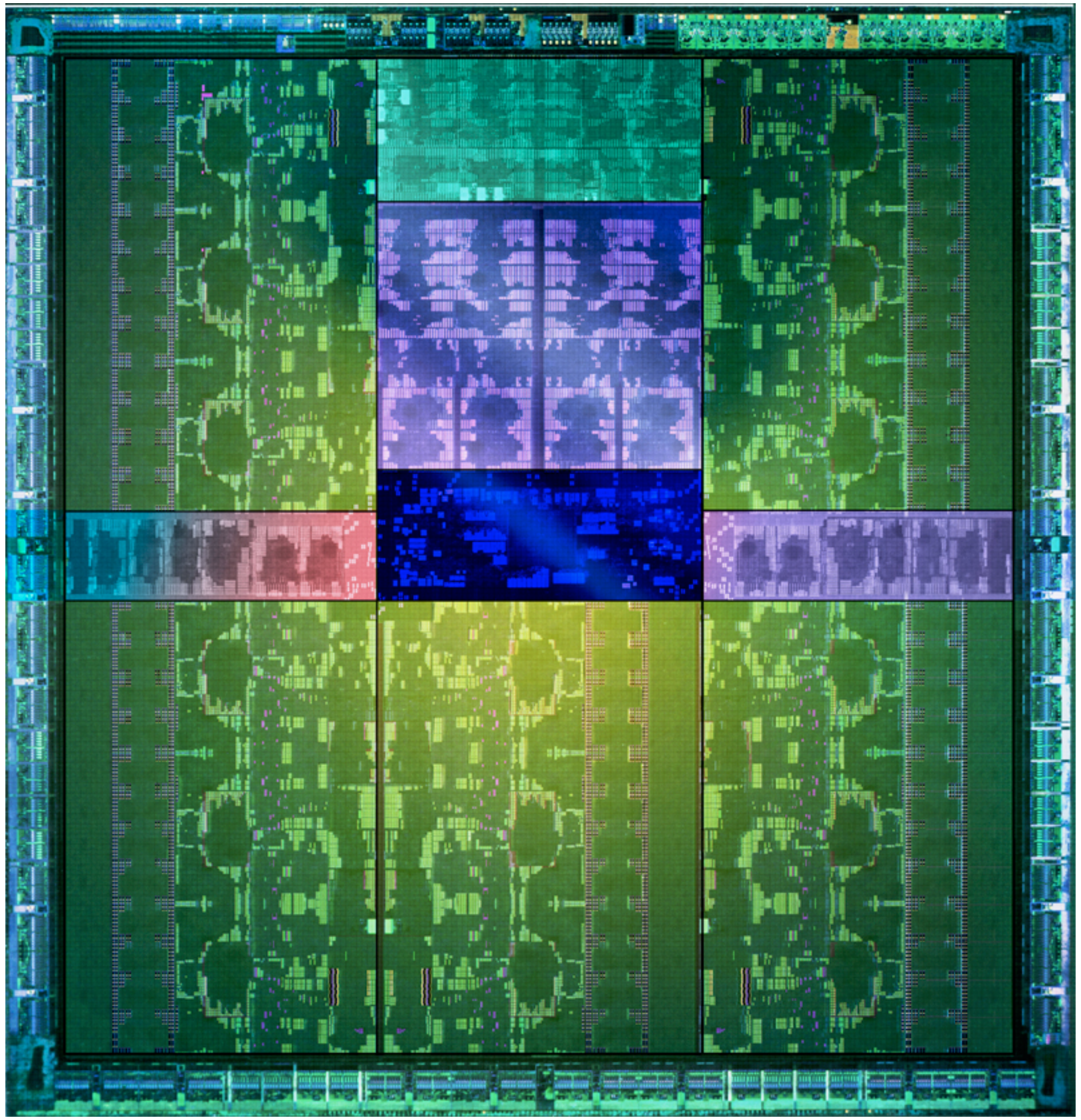
https://80.lv/articles/ue4arch-interview-about-realiscit-environments-in-ue4/

DIE SHOTS - CPU OR GPU?

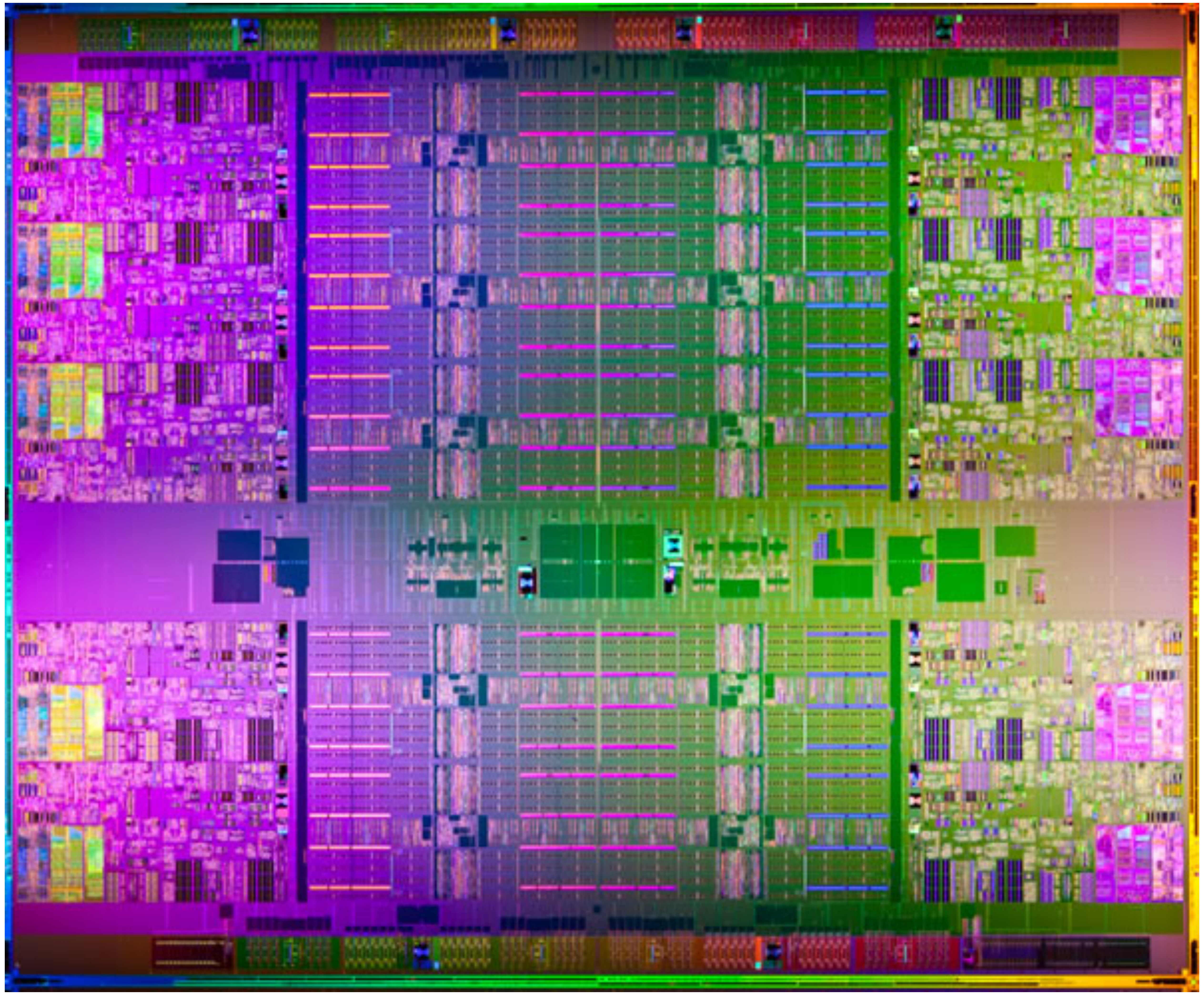




DIE SHOTS - CPU OR GPU?

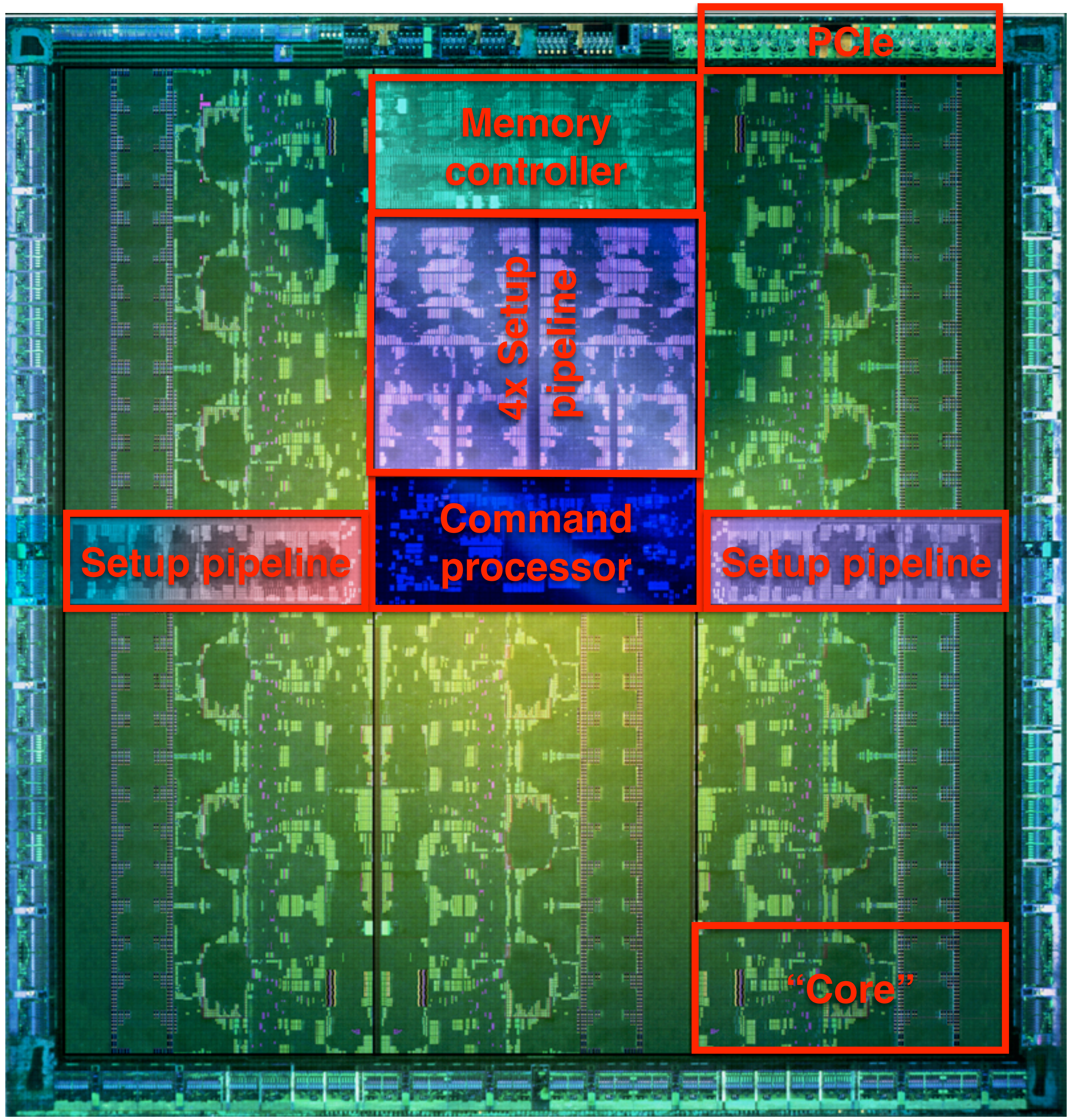


NVIDIA Kepler- GK110

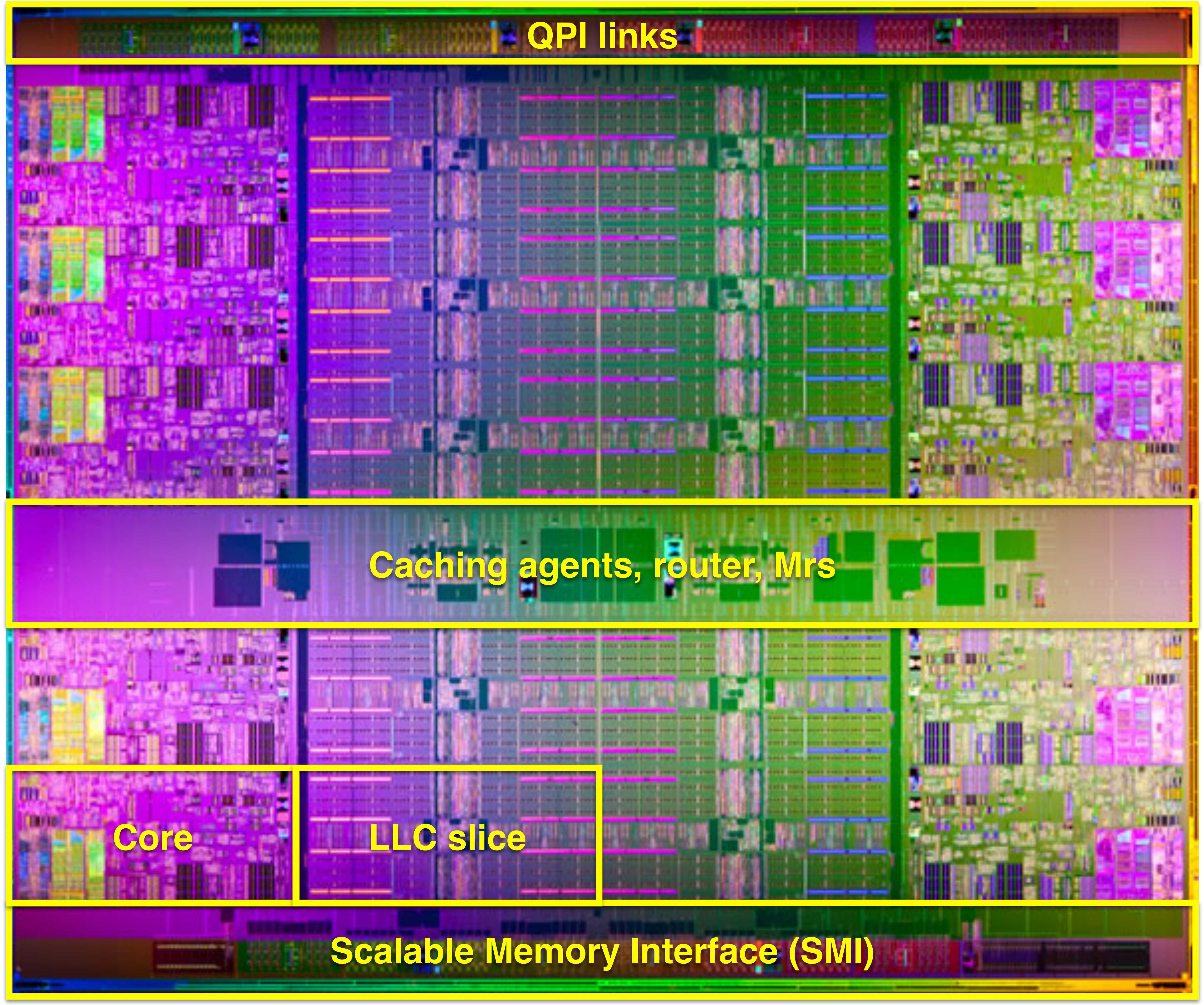


Intel Xeon E7 - Westmere-EX

DIE SHOTS - CPU OR GPU?



NVIDIA Kepler- GK110



Intel Xeon E7 - Westmere-EX

PERFORMANCE SCALING

$$Perf(\frac{ops}{s}) = \underbrace{\frac{Instructions}{cycle}}_{\propto\ PipelineCount \cdot PipelineDepth} \cdot \underbrace{frequency}_{\text{scales with feature size}}$$

CLASSICAL DENNARD SCALING

Partly by Bill Dally, Sudha Yalamanchili (UCAA Workshop, 2012)

PERFORMANCE SCALING

$$Perf(\frac{ops}{s}) = \underbrace{\frac{Instructions}{cycle}}_{\propto PipelineCount \cdot PipelineDepth} \cdot \underbrace{frequency}_{\text{scales with feature size}}$$

CLASSICAL DENNARD SCALING

$$Perf(\frac{ops}{s}) = \underbrace{Power(W)}_{\text{fixed}} \cdot \underbrace{Efficiency(\frac{ops}{Joule})}_{\text{operator cost} \; + \; \text{data movement cost}}$$

POST DENNARD SCALING

Partly by Bill Dally, Sudha Yalamanchili (UCAA Workshop, 2012)

PERFORMANCE SCALING

$$Perf(\frac{ops}{s}) = \underbrace{\frac{Instructions}{cycle}}_{\propto PipelineCount \cdot PipelineDepth} \cdot \underbrace{frequency}_{\textit{scales with feature size}}$$

CLASSICAL DENNARD SCALING

$$Perf(\frac{ops}{s}) = \underbrace{Power(W)}_{\textit{fixed}} \cdot \underbrace{Efficiency(\frac{ops}{Joule})}$$

POST DENNARD SCALING

REGIME I

$\underbrace{\textit{operator cost}}_{\textit{Specialization —> heterogeneity and asymmetry}}$ + *data movement cost*

Specialization —> heterogeneity and asymmetry

GPU COMPUTING

PROGRAMMING COMPLEXITY

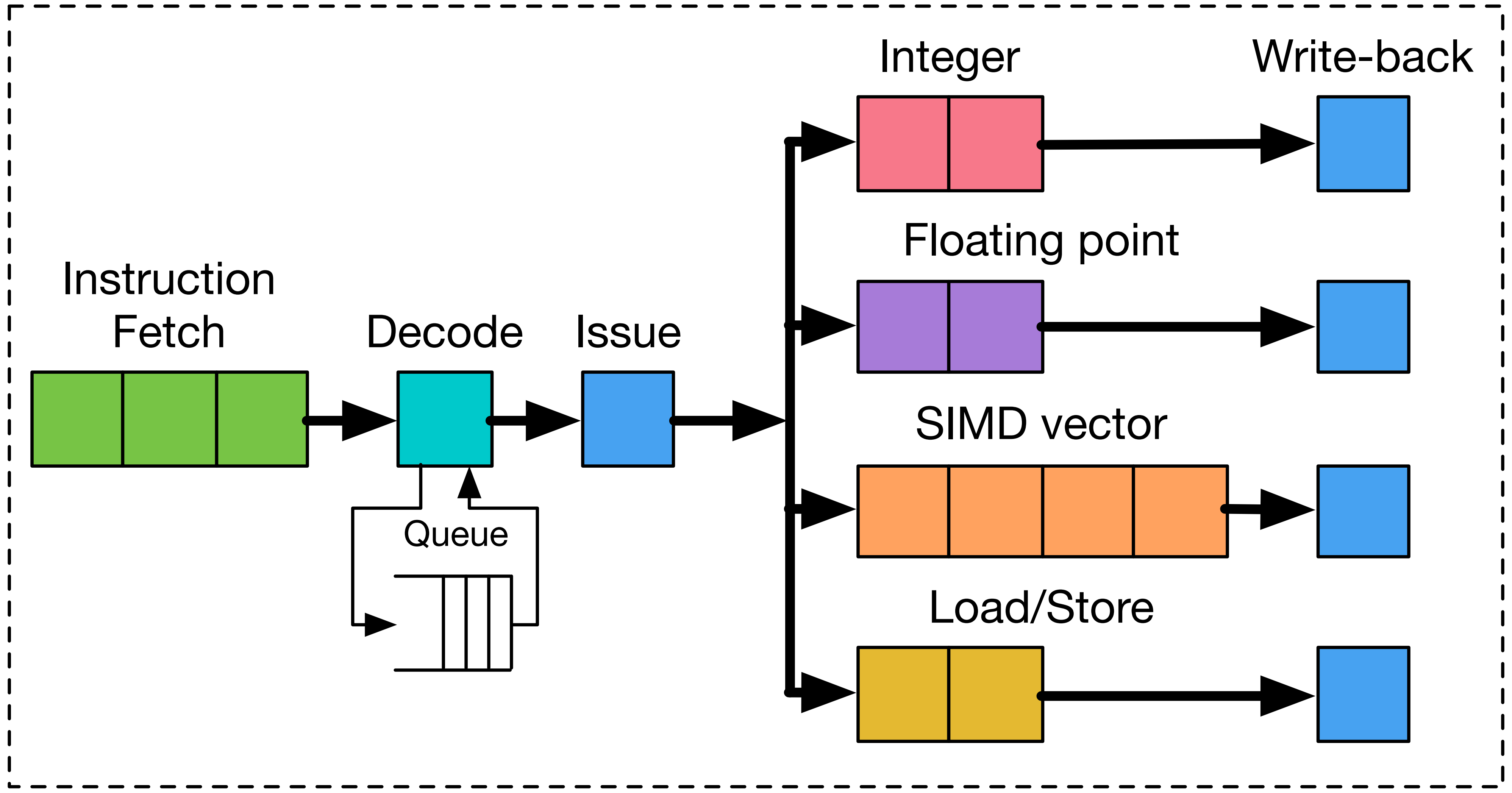
NUMA EFFECTS & LATENCY HIDING

REGIME II

Partly by Bill Dally, Sudha Yalamanchili (UCAA Workshop, 2012)

POST-DENNARD: TRANSITION TO MASSIVELY PARALLEL MICROARCHITECTURES

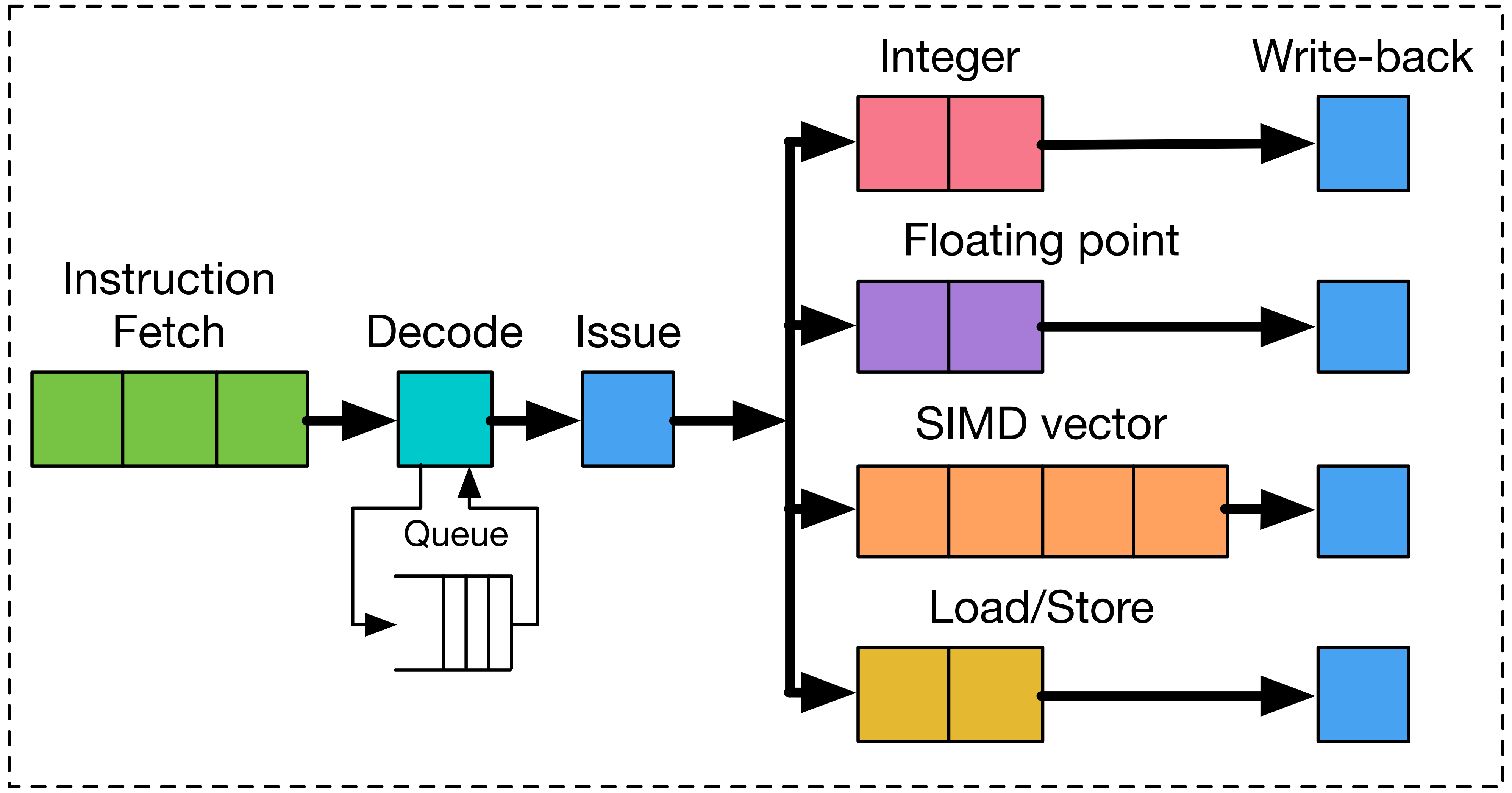
$$P = afCV^2 + VI_{leakage} \propto f^3$$

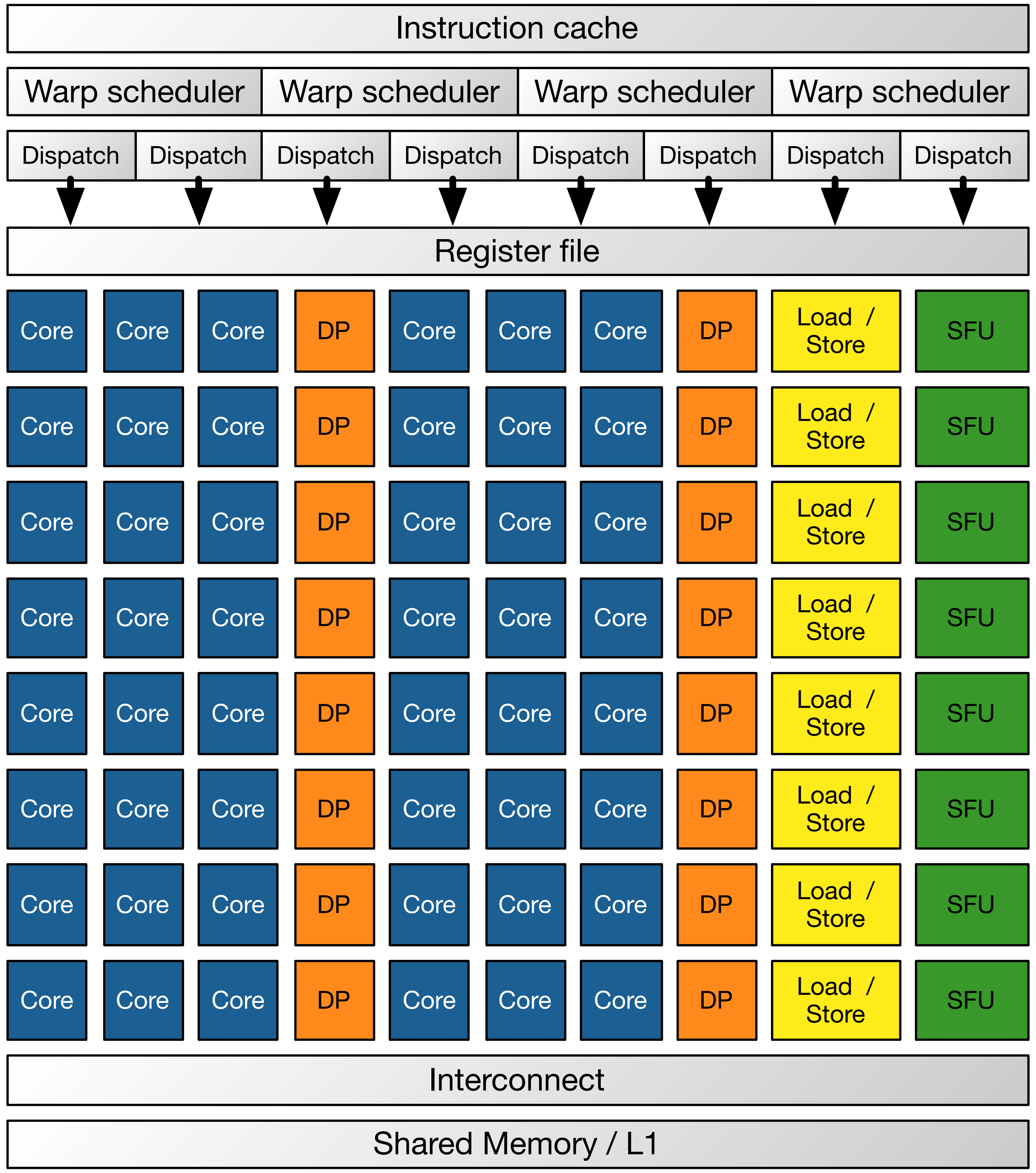


Frequency reduction
In-order pipelines

POST-DENNARD: TRANSITION TO MASSIVELY PARALLEL MICROARCHITECTURES

$$P = afCV^2 + VI_{leakage} \propto f^3$$





Frequency reduction
In-order pipelines

Replication

Massively parallel
Energy efficient

CUDA & GPU - OVERVIEW

NVIDIA CUDA

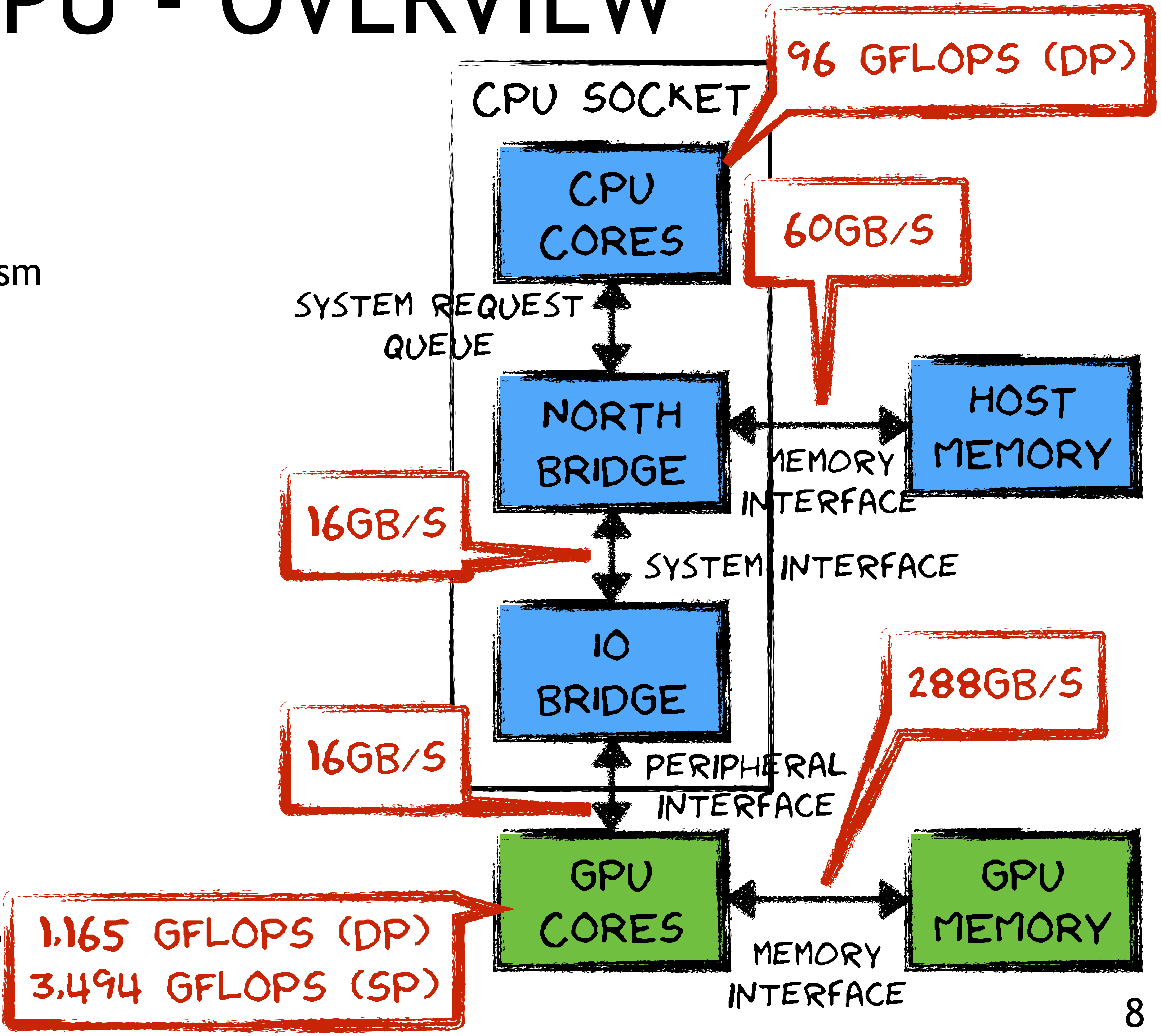
- Compute kernel as C program
- Explicit data- and thread-level parallelism
- Computing, not graphics processing
- Host communication

Memory hierarchy

- Host memory
- GPU (device) memory
- GPU on-chip memory (later)

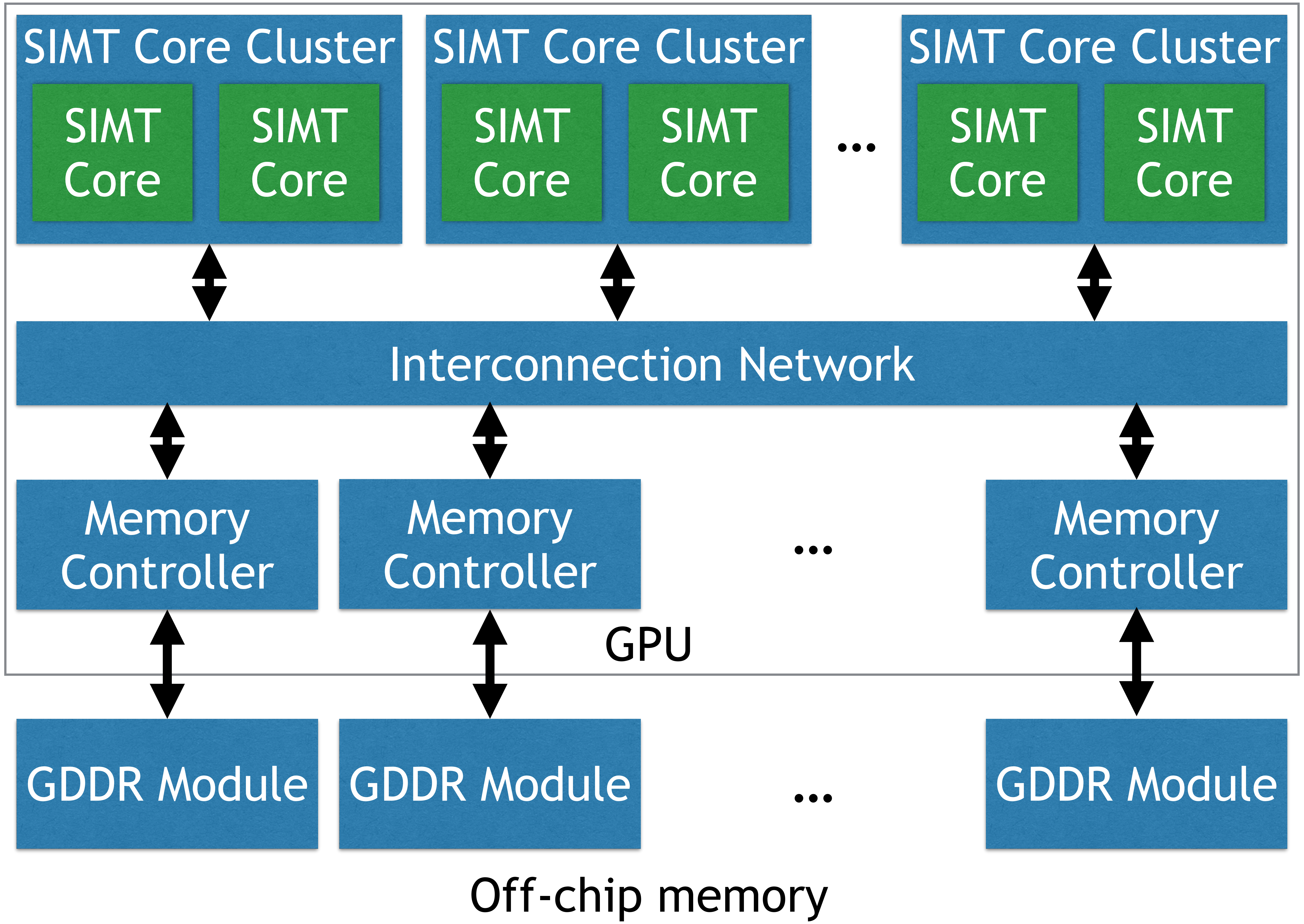
More HW details exposed

- Use of pointers
- Load/store architecture
- Barrier synchronization of thread blocks

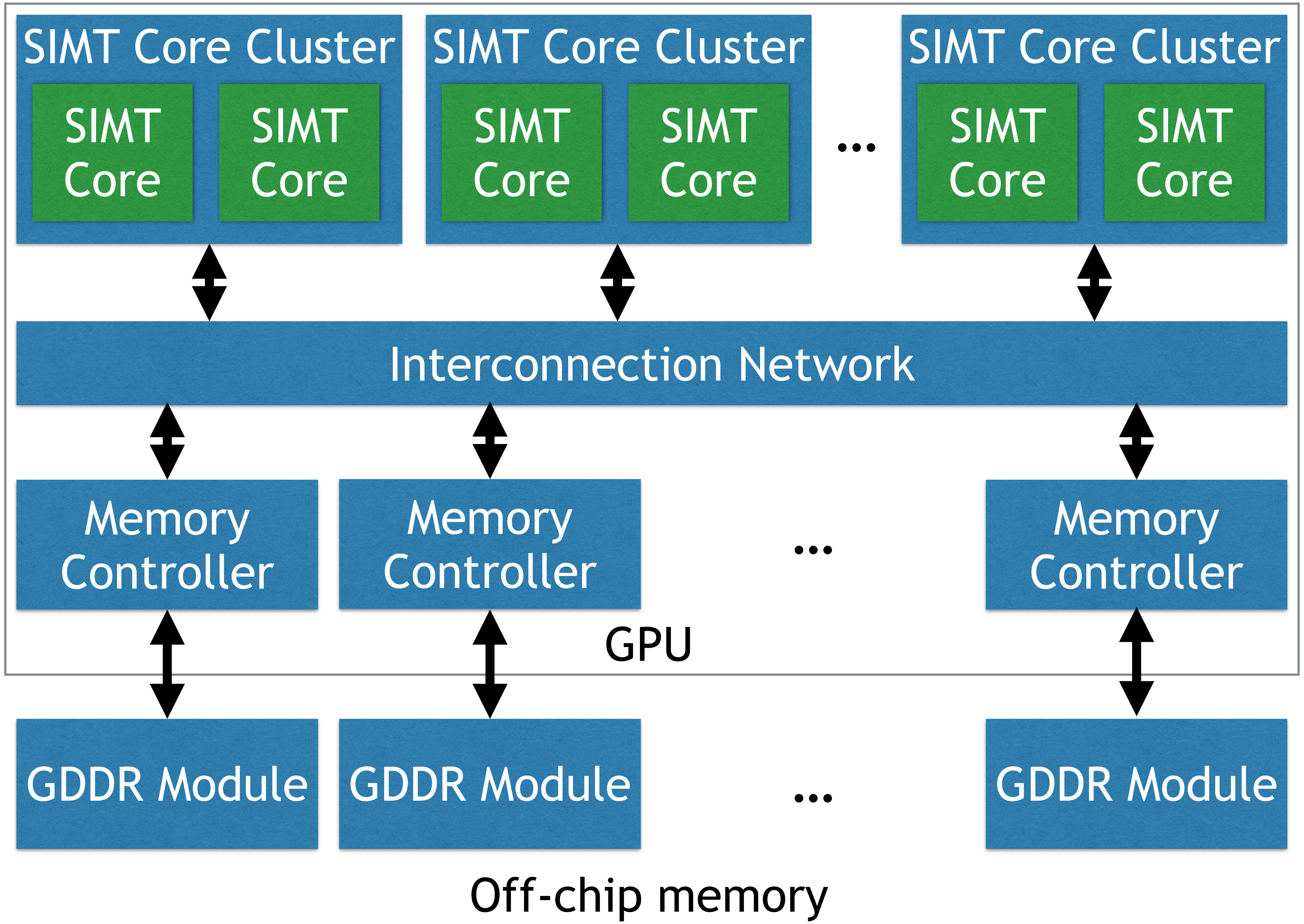


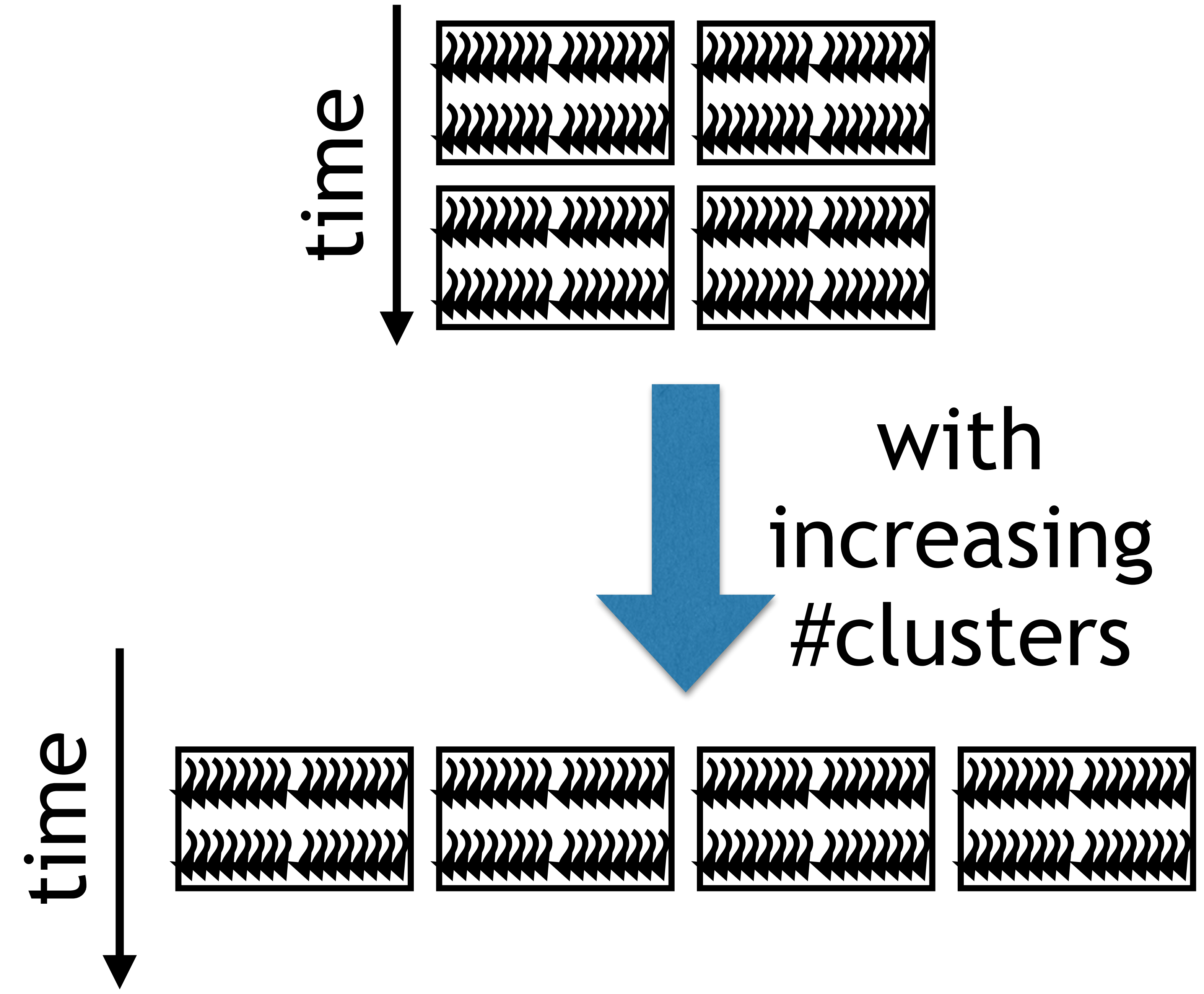
HARDWARE ARCHITECTURE

GPU ARCHITECTURE TOP-LEVEL VIEW

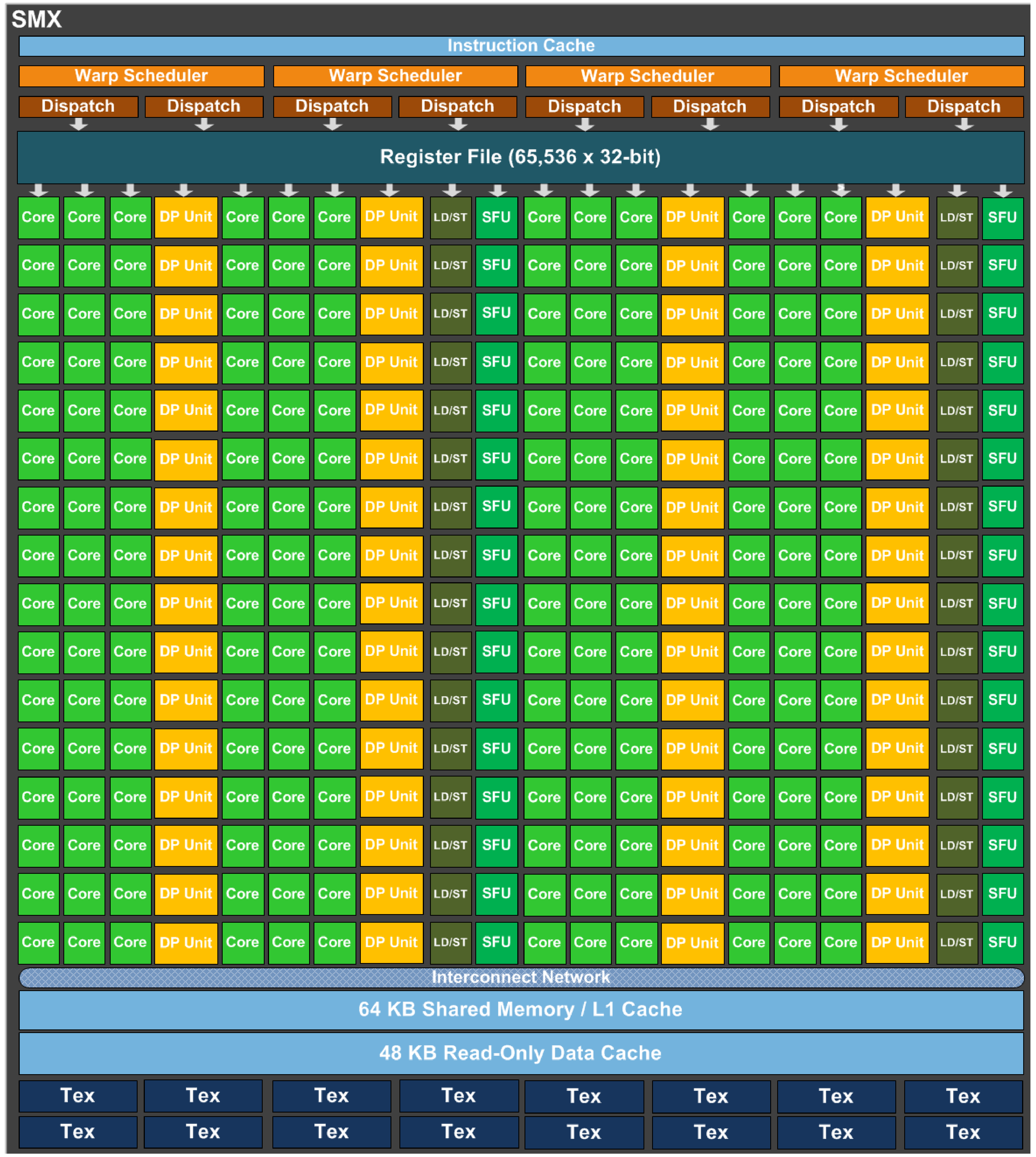


GPU ARCHITECTURE TOP-LEVEL VIEW





“NVIDIA-STYLE” SIMT CORE CLUSTER



Streaming Multi-Processor (SM)

- Multi-threaded
- Data parallel

Capabilities

- 64K registers
- 192 simple cores (Integer and SP FPU)
- 64 DP FPUs
- 32 LSUs, 32 SFUs

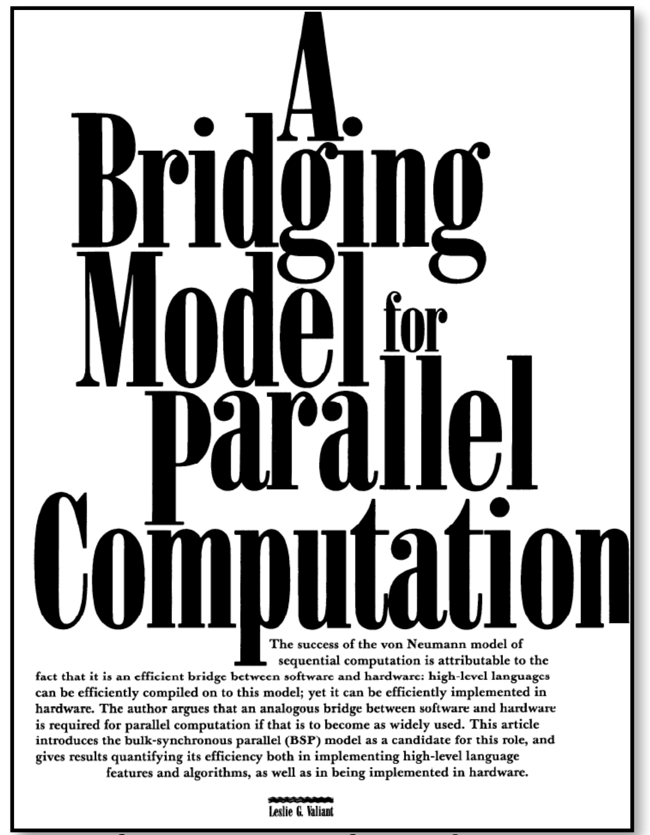
Scheduling

- 4 warp schedulers
- 2-way dispatch per warp

SOFTWARE VIEW

BULK-SYNCHRONOUS PARALLEL

In 1990, Valiant already described GPU computing pretty well



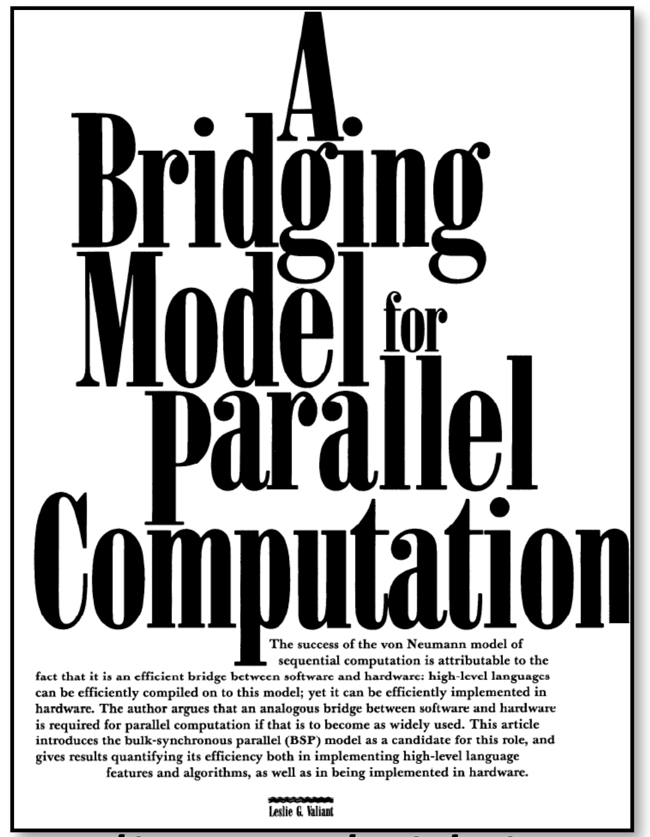
Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990

BULK-SYNCHRONOUS PARALLEL

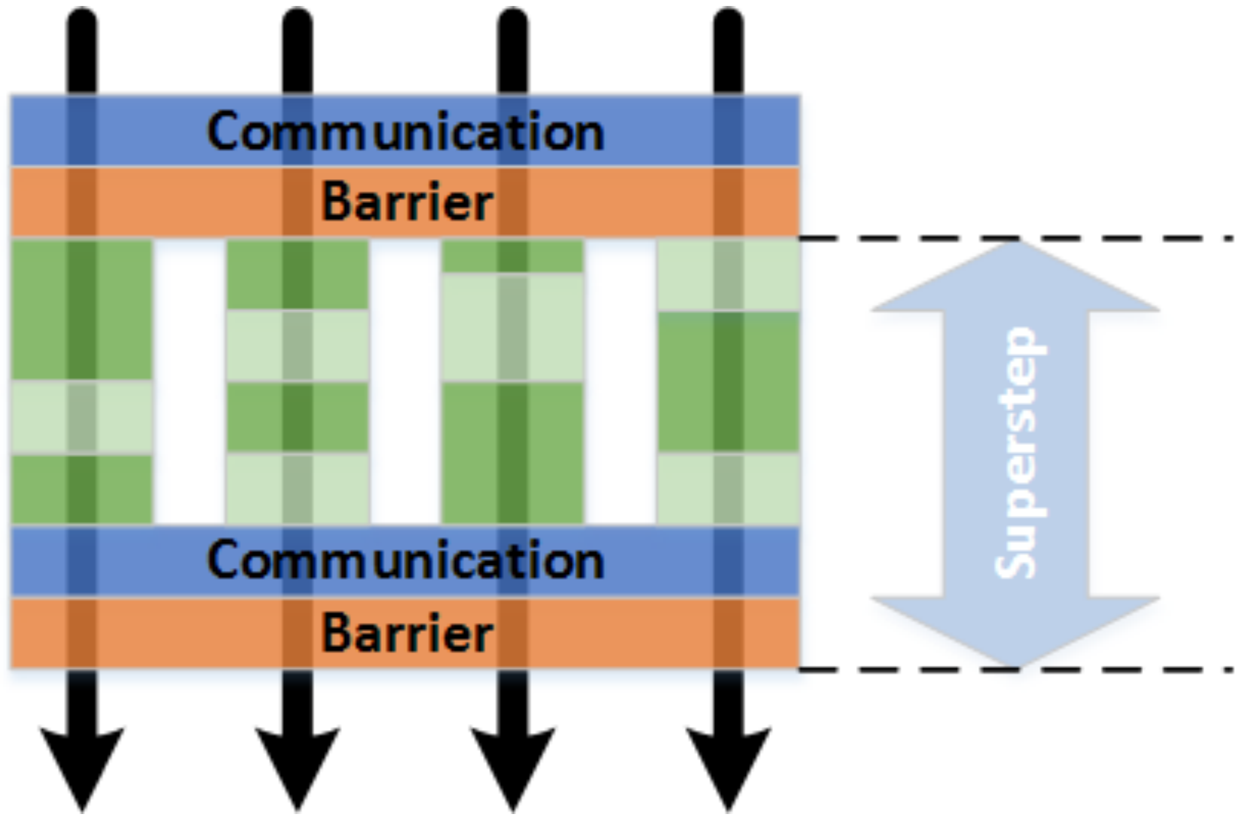
In 1990, Valiant already described GPU computing pretty well

Superstep

- Compute, communicate, synchronize



Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990



BULK-SYNCHRONOUS PARALLEL

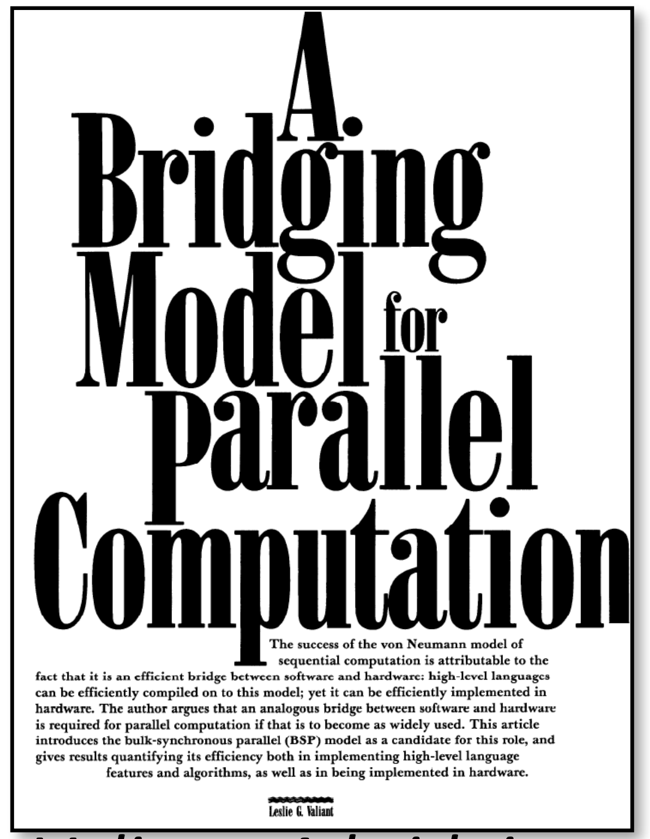
In 1990, Valiant already described GPU computing pretty well

Superstep

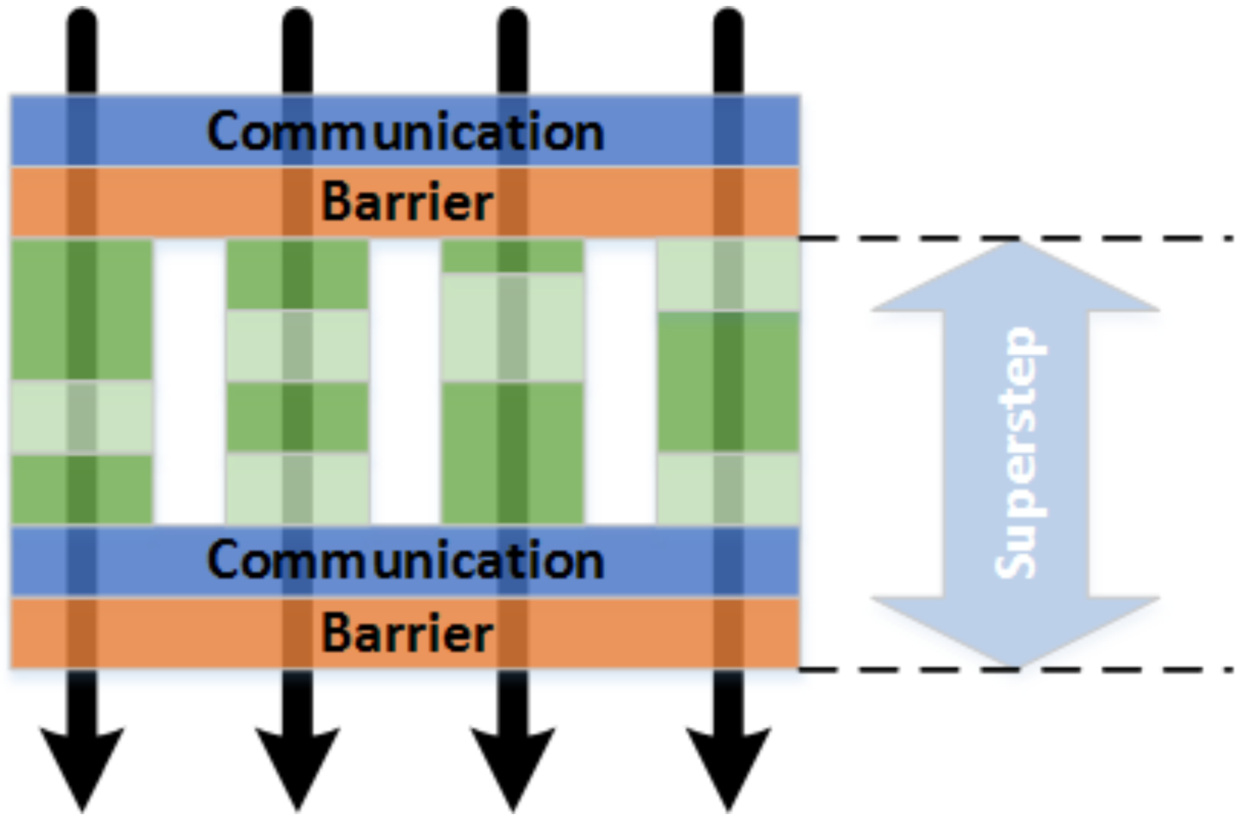
- Compute, communicate, synchronize

Parallel slackness: # of virtual processors v, physical processors p

- v = 1: not viable
- v = p: unpromising wrt optimality
- v >> p: leverage slack to schedule and pipeline computation and communication efficiently



Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990



BULK-SYNCHRONOUS PARALLEL

In 1990, Valiant already described GPU computing pretty well

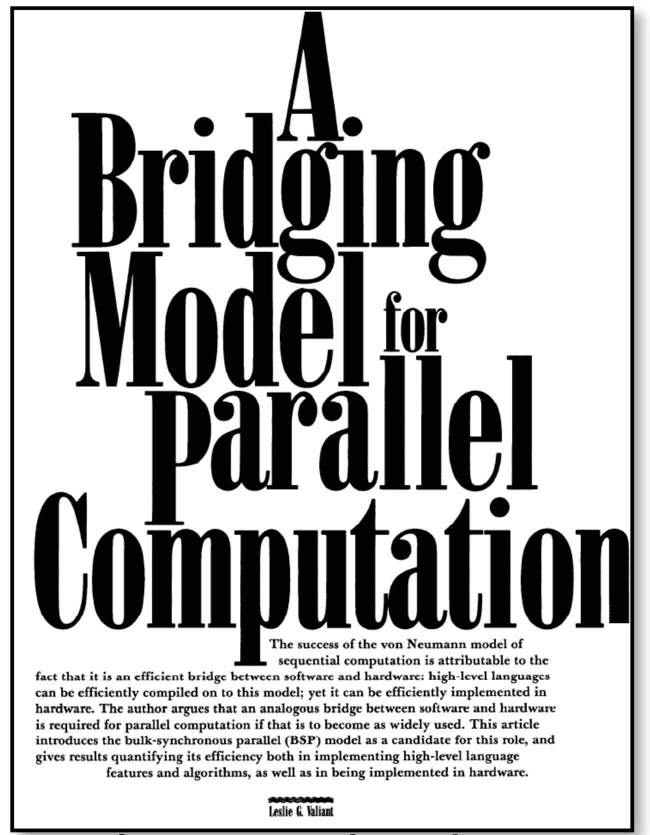
Superstep

- Compute, communicate, synchronize

Parallel slackness: # of virtual processors v, physical processors p

- v = 1: not viable
- v = p: unpromising wrt optimality
- v >> p: leverage slack to schedule and pipeline computation and communication efficiently

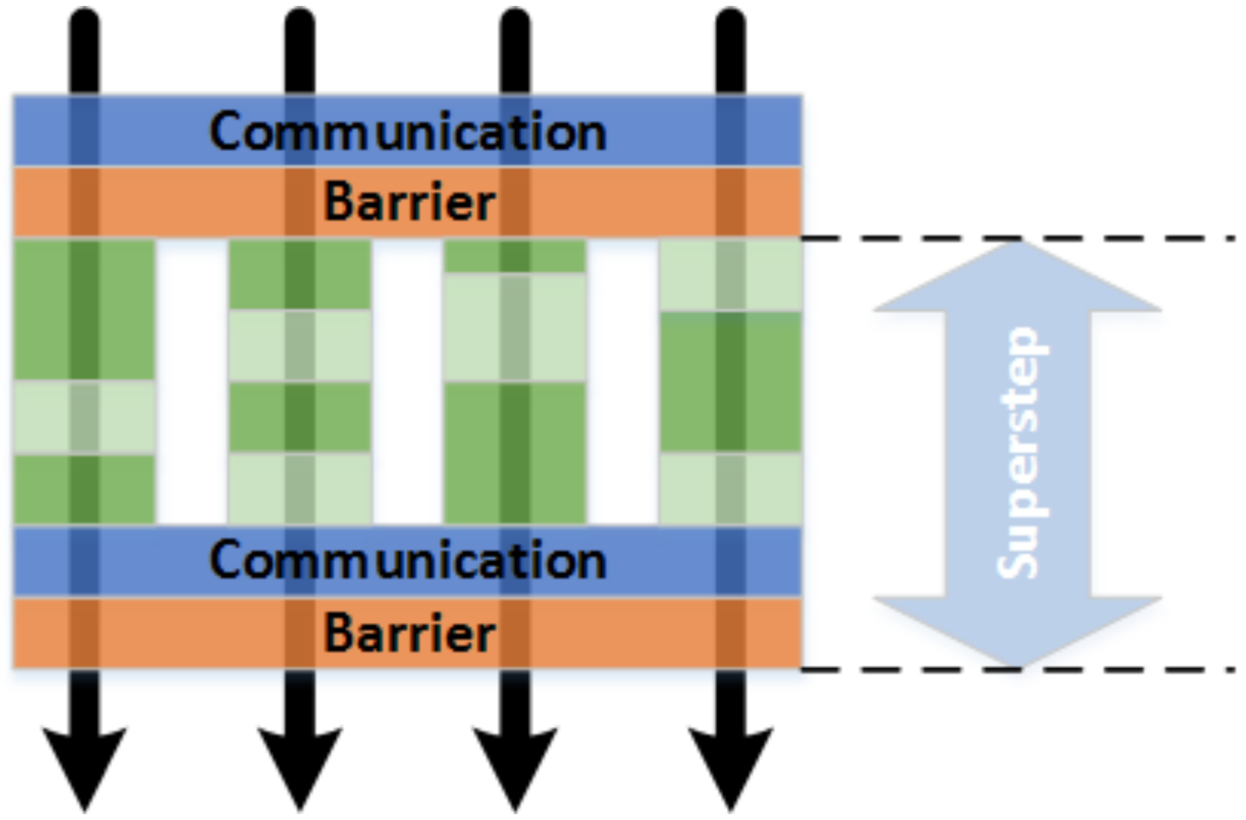
Extremely scalable, bad for unbalanced parallelism

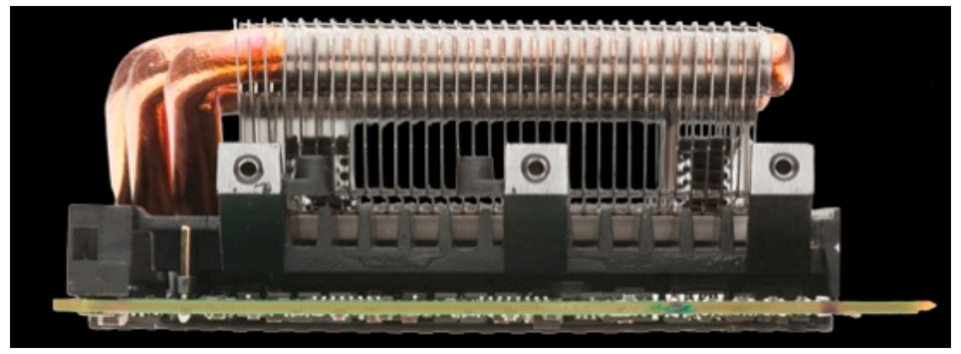

A Bridging Model for Parallel Computation

The success of the von Neumann model of sequential computation is attributable to the fact that it is an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware. The author argues that an analogous bridge between software and hardware is required for parallel computation if that is to become as widely used. This article introduces the bulk-synchronous parallel (BSP) model as a candidate for this role, and gives results quantifying its efficiency both in implementing high-level language features and algorithms, as well as in being implemented in hardware.

Leslie G. Valiant

Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990





THE BEAUTY OF SIMPLICITY

Thread-collective computation and memory accesses

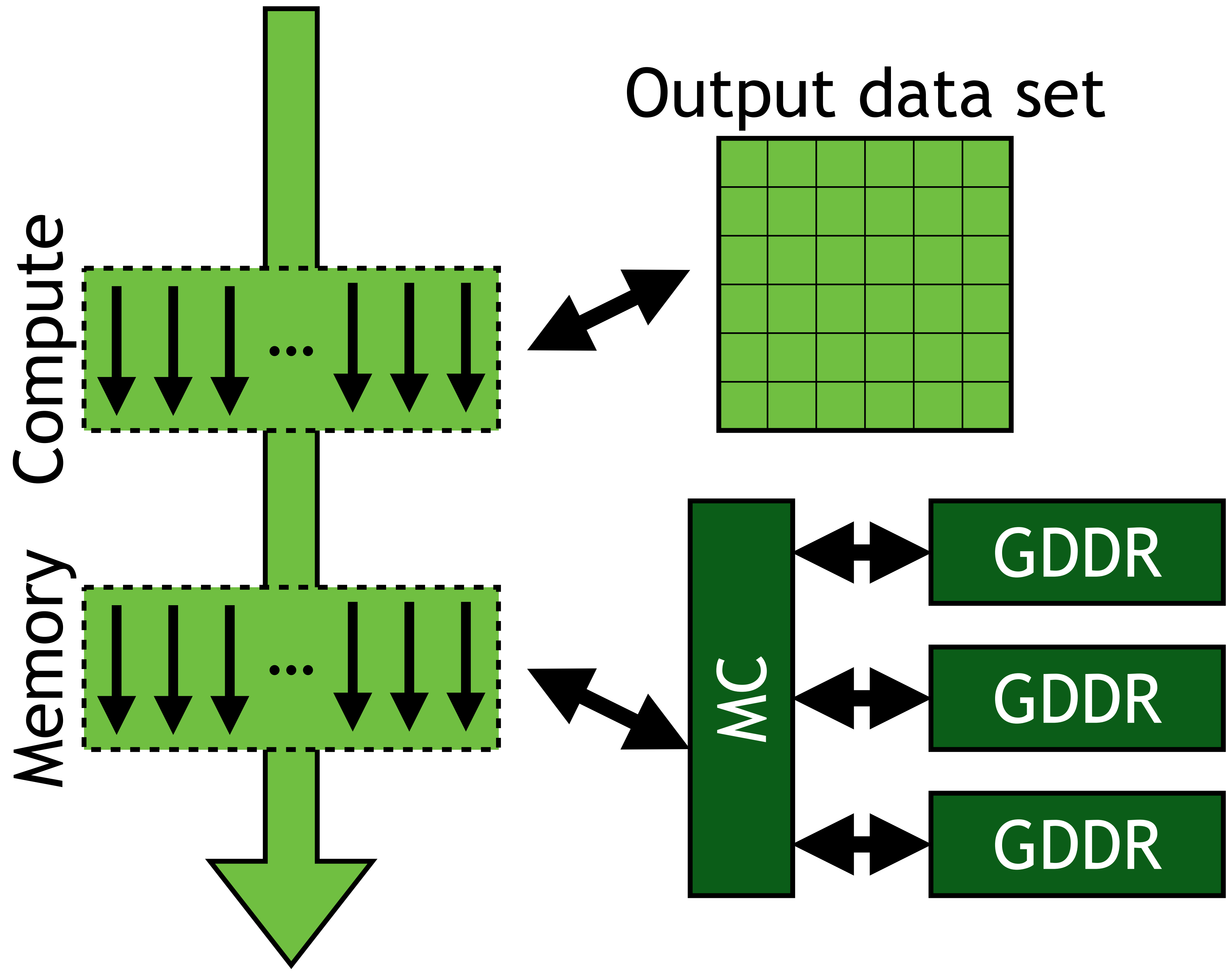
- Thread ID determines data element

GPU collaborative computing

- One thread per output element
- Schedulers exploit parallel slackness

GPU collaborative memory access

- One thread per data element



THE BEAUTY OF SIMPLICITY

Thread-collective computation and memory accesses

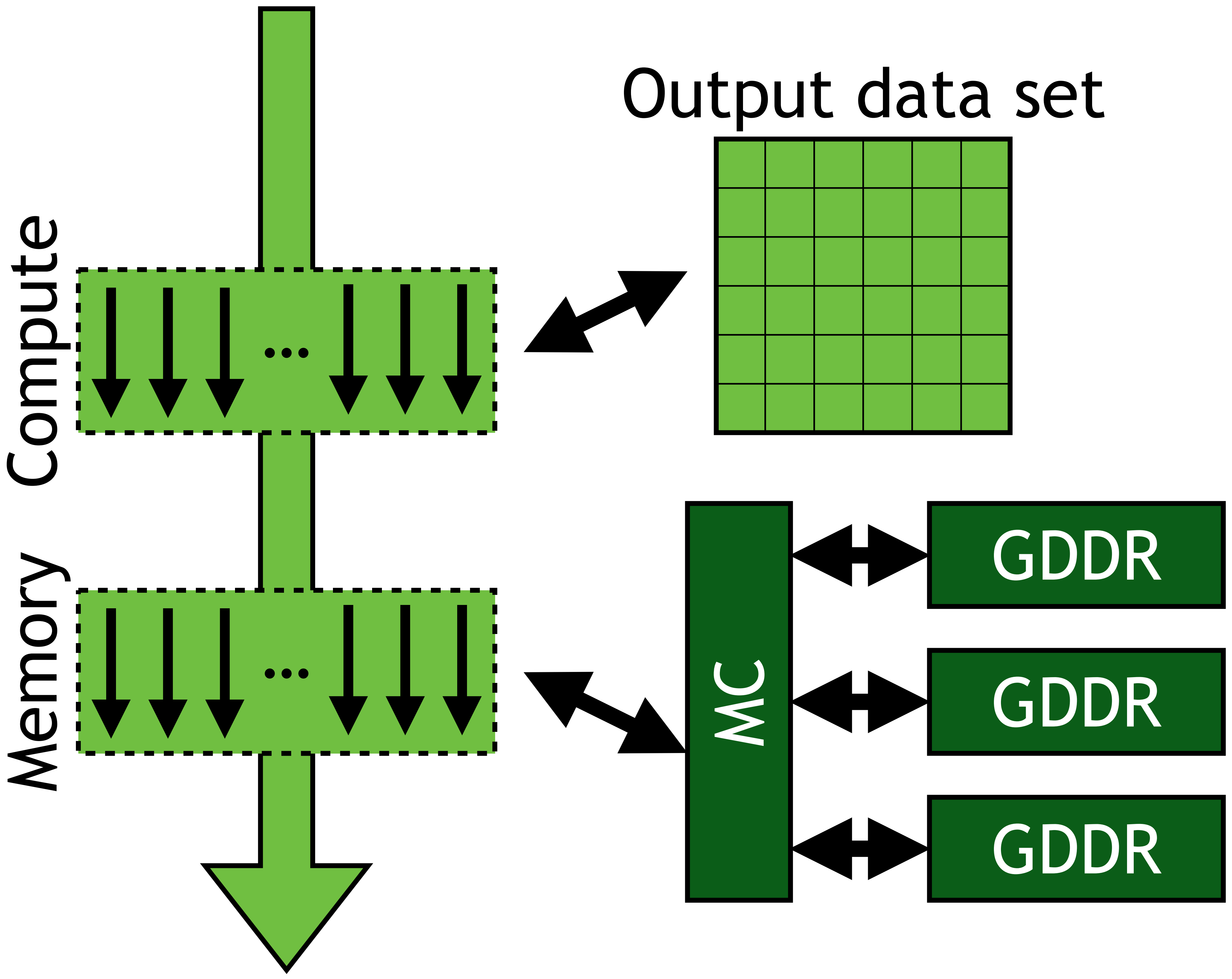
- Thread ID determines data element

GPU collaborative computing

- One thread per output element
- Schedulers exploit parallel slackness

GPU collaborative memory access

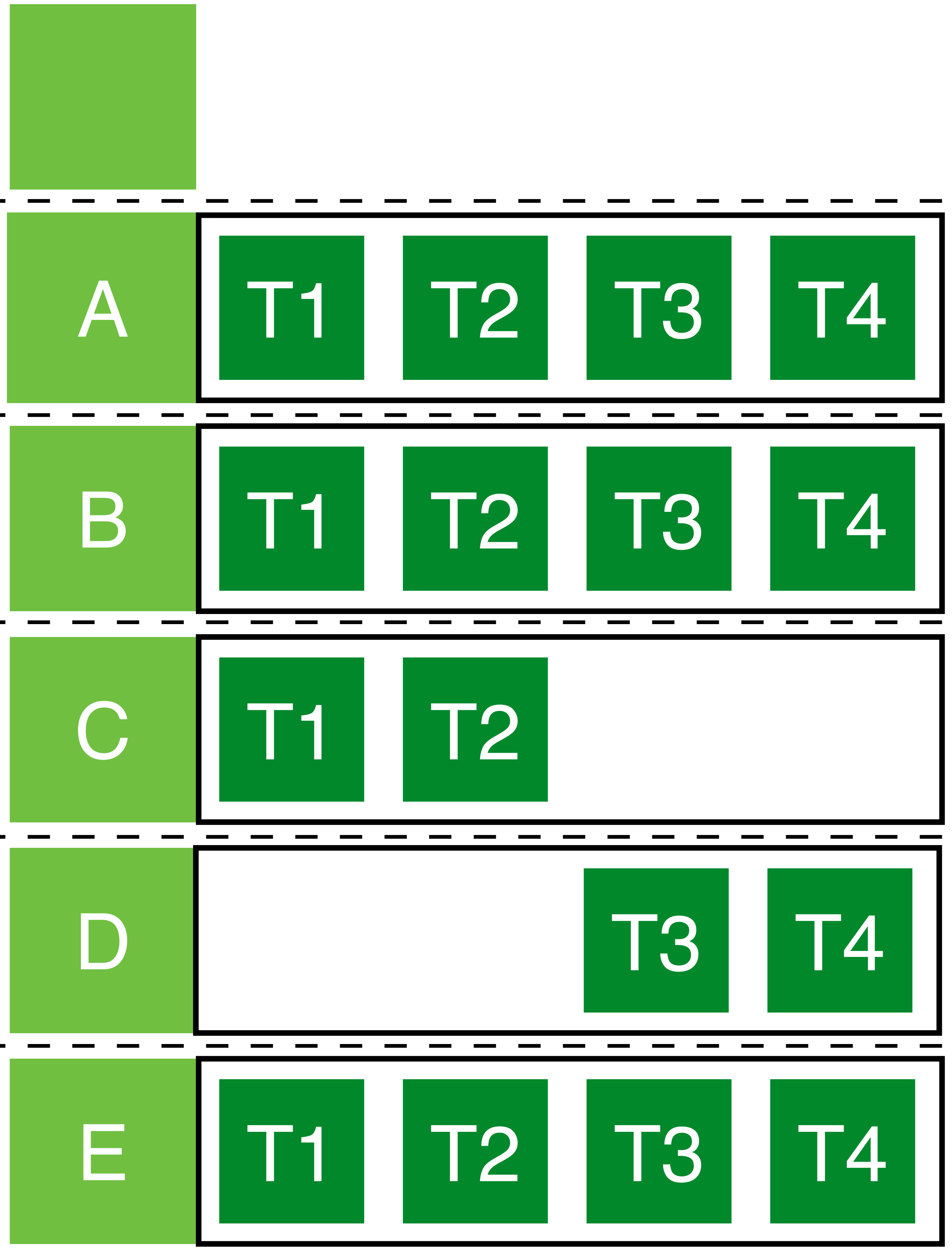
- One thread per data element



-> If you do something on a GPU, do it collaboratively with all threads

SIMT EXECUTION MODEL

```
foo[] = {4,8,12,16};

A: v = foo[tid.x];

B: if (v < 10)

C:     v = 0;

   else
D:     v = 10;

E: w = bar[tid.x] + v;
```



Programmer sees independent scalar threads

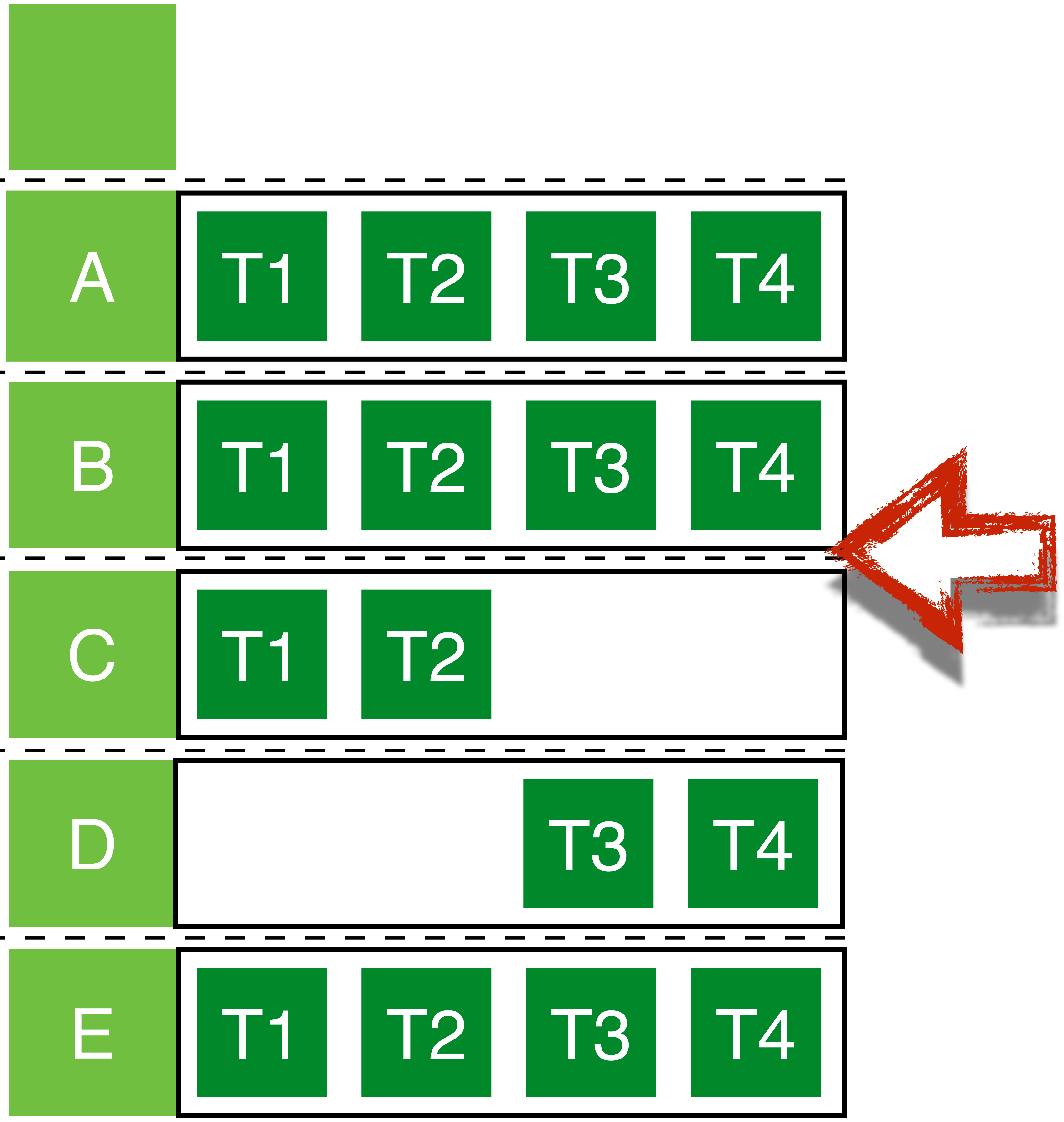
- Illusion

GPU HW bundles threads into warps

- Warps run in lockstep on vector-like hardware (SIMD)
- How is divergent control flow handled?

SIMT EXECUTION MODEL

```
foo[] = {4,8,12,16};

A: v = foo[tid.x];

B: if (v < 10)

C:     v = 0;
   else
D:     v = 10;

E: w = bar[tid.x] + v;
```



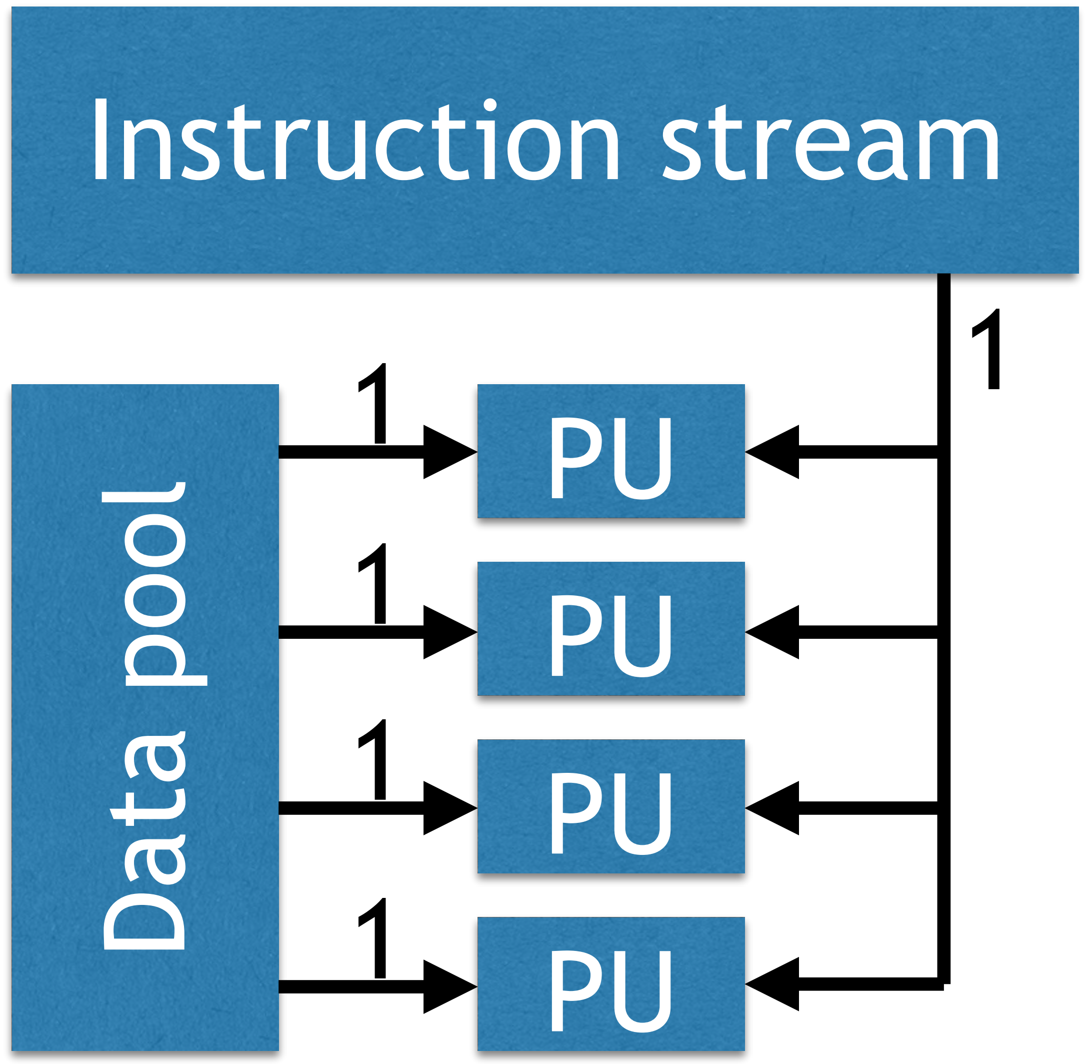
- Programmer sees independent scalar threads
  - Illusion
- GPU HW bundles threads into warps
  - Warps run in lockstep on vector-like hardware (SIMD)
  - How is divergent control flow handled?

PROGRAMMABILITY OF MASSIVE PARALLELIZATION

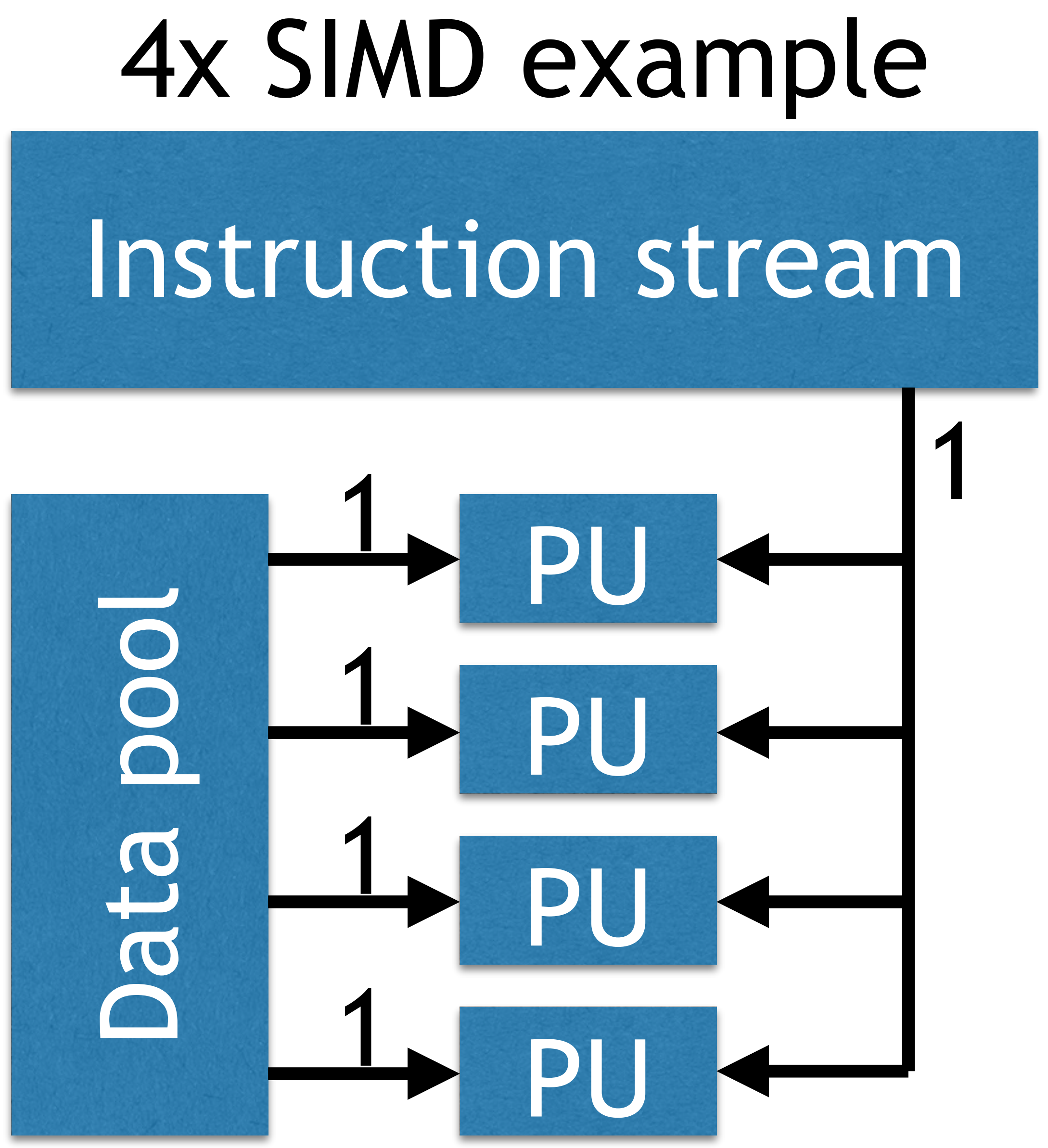
Vector ISAs are great

- Compact: one instruction for multiple data elements
- Parallel: N operations are independent
- Expressive: complex memory accesses (irregular strides)

4x SIMD example



PROGRAMMABILITY OF MASSIVE PARALLELIZATION

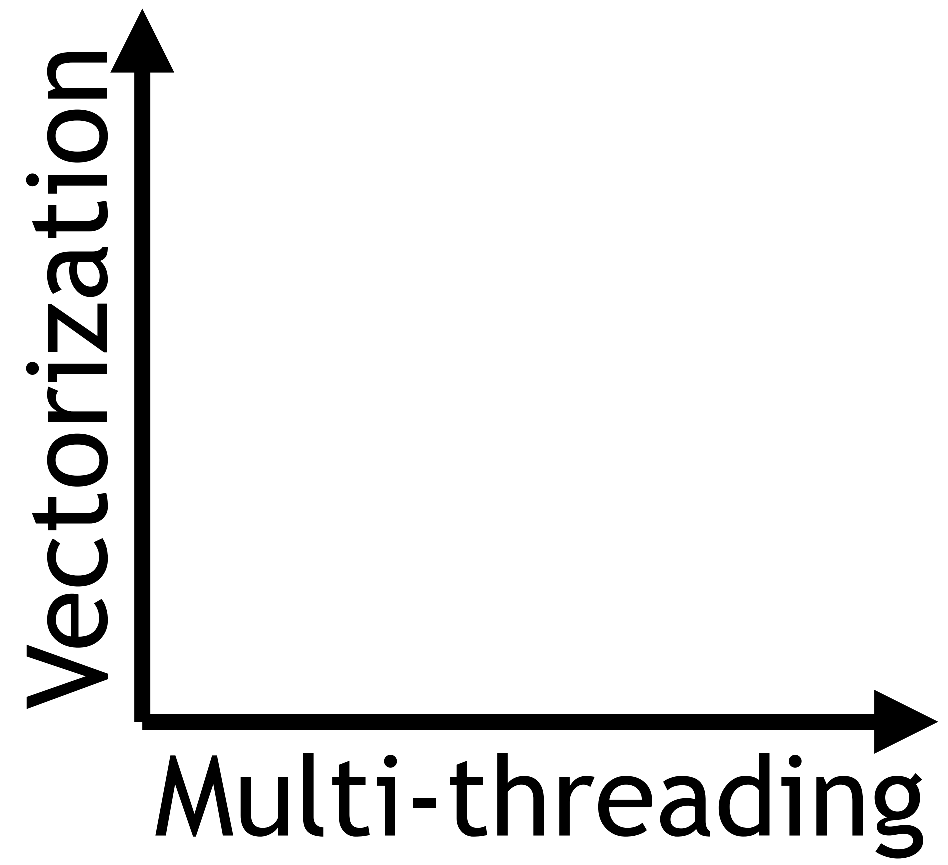


Vector ISAs are great

- Compact: one instruction for multiple data elements
- Parallel: N operations are independent
- Expressive: complex memory accesses (irregular strides)

Vector ISAs are bad

- Orthogonal to multi-threading
- Static in size, static in selection, mixed semantic model for vector/scalar instructions, C/C++ is scalar



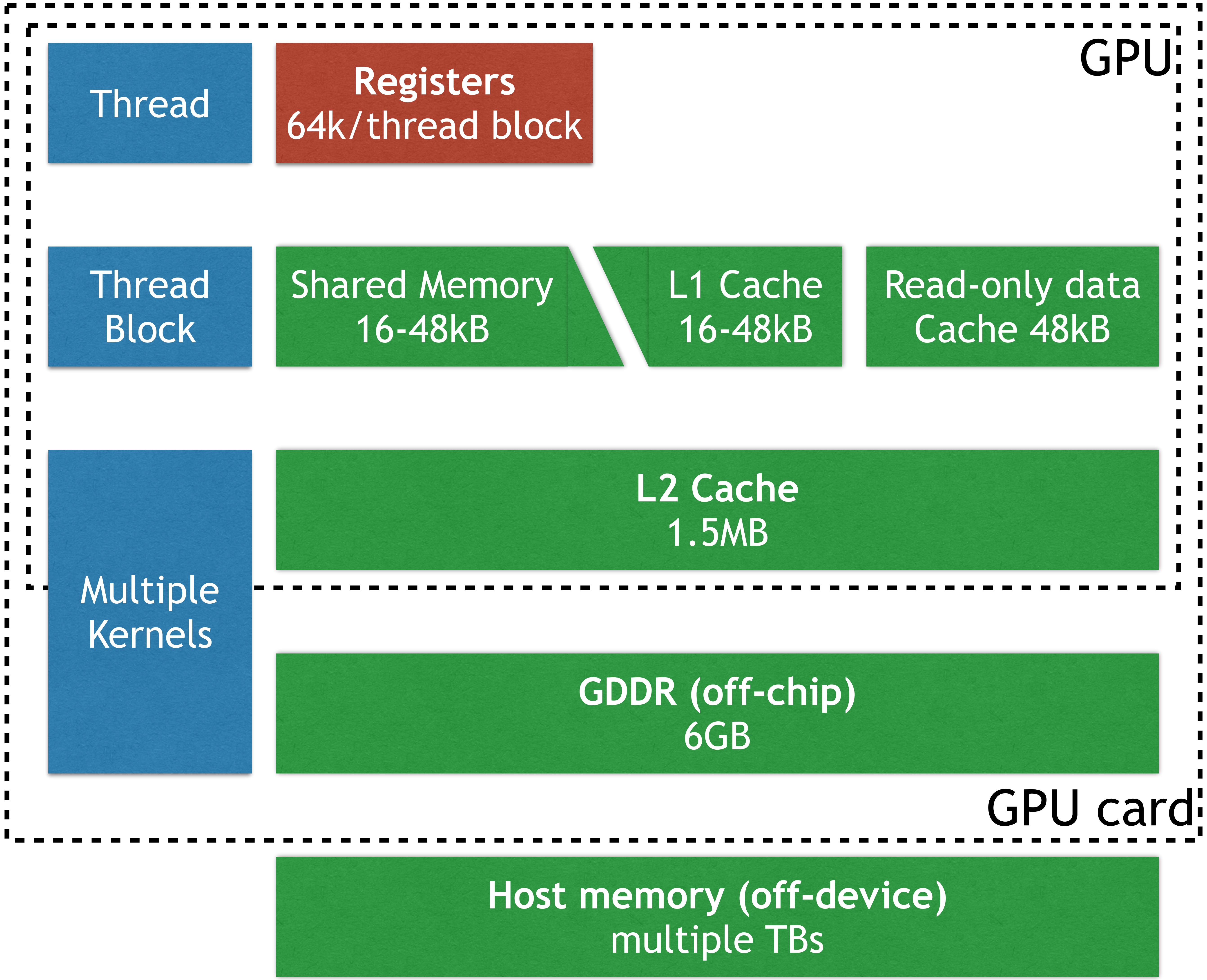
ACCESSING MEMORY

Explicit memory hierarchy

- Manual GPU memory fills & spilling
- Manual shared memory fills

Explicit memory hierarchy simplifies coherence & consistency

- No guarantees except for kernel completion boundaries
- Software-controlled coherence



MIND THE MEMORY HIERARCHY

Intel Sandy Bridge	GK110	GP100	GA100
Reg. *1kB* *5TB/s*	Reg. *~4MB,40TB/s*	Reg. *14MB*	Reg. *32MB*
L1 *512kB* *1TB/s*	SM *1MB* *1TB/s*	SM *~4MB*	SM *24MB*
L2 *2MB*			
LLC *8MB, 500GB/s*	LLC *1.5MB, 500GB/s*	LLC *4MB*	LLC *40MB*
Main memory *TBs, 20GB/s*	GPU memory *4GB, 150GB/s*	GPU memory *16GB, 800GB/s*	GPU memory *48GB, 1.9TB/s*

OUR VIEW OF A GPU

Software view: a programmable many-core scalar architecture

- Huge amount of scalar threads, operates in lock-step
- SIMT: single instruction, multiple threads

Hardware view: a programmable multi-core vector architecture

- SIMD: single instruction, multiple data
- Illusion of scalar threads: hardware packs them into compound units

OUR VIEW OF A GPU

Software view: a programmable many-core scalar architecture

- Huge amount of scalar threads, operates in lock-step
- SIMT: single instruction, multiple threads

Hardware view: a programmable multi-core vector architecture

- SIMD: single instruction, multiple data
- Illusion of scalar threads: hardware packs them into compound units

<u>IT'S A VECTOR ARCHITECTURE THAT HIDES ITS VECTOR UNITS</u>

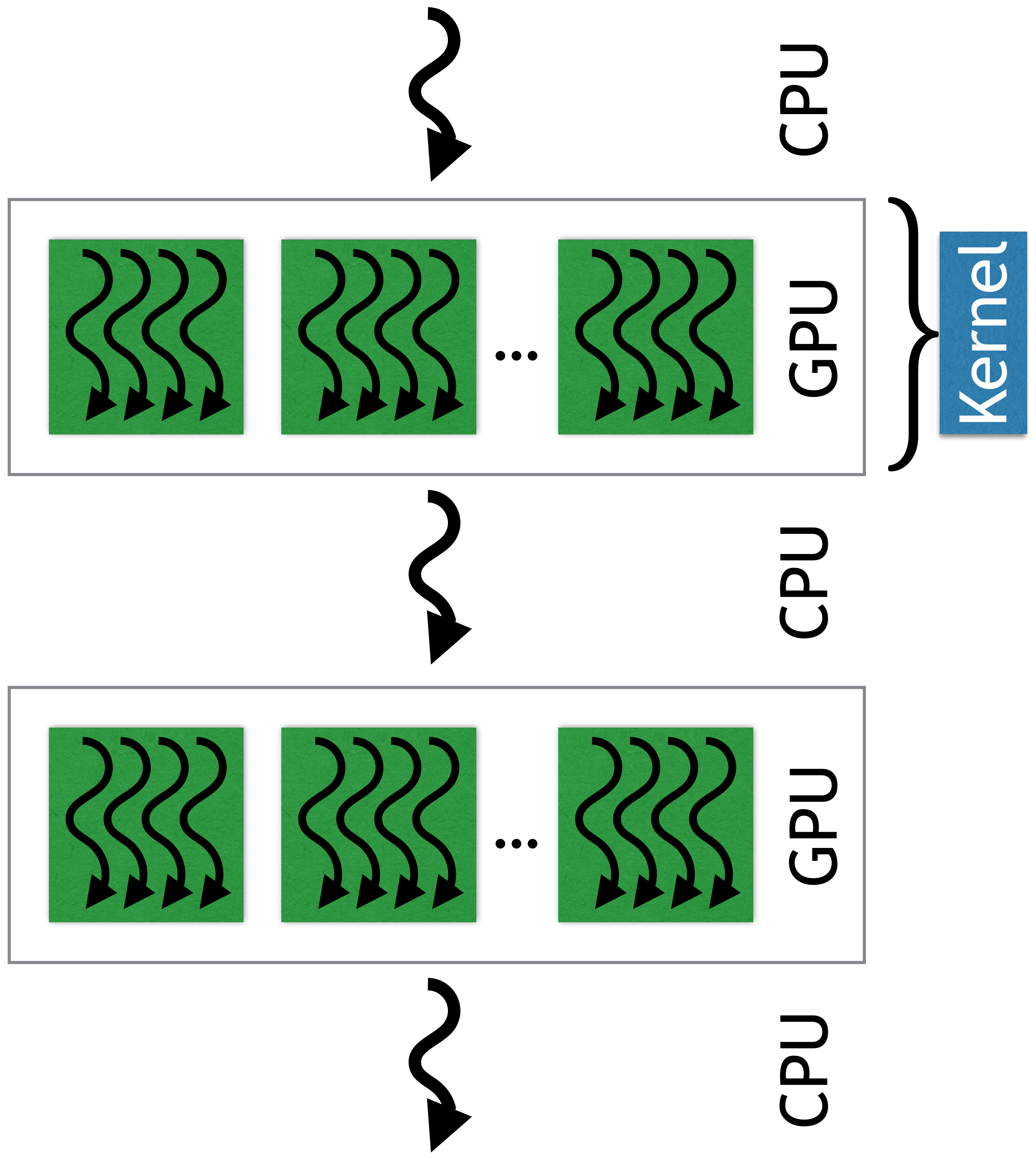
MAKING GPU USAGE EASY

GPU LIBRARIES

PROGRAMMING MODEL

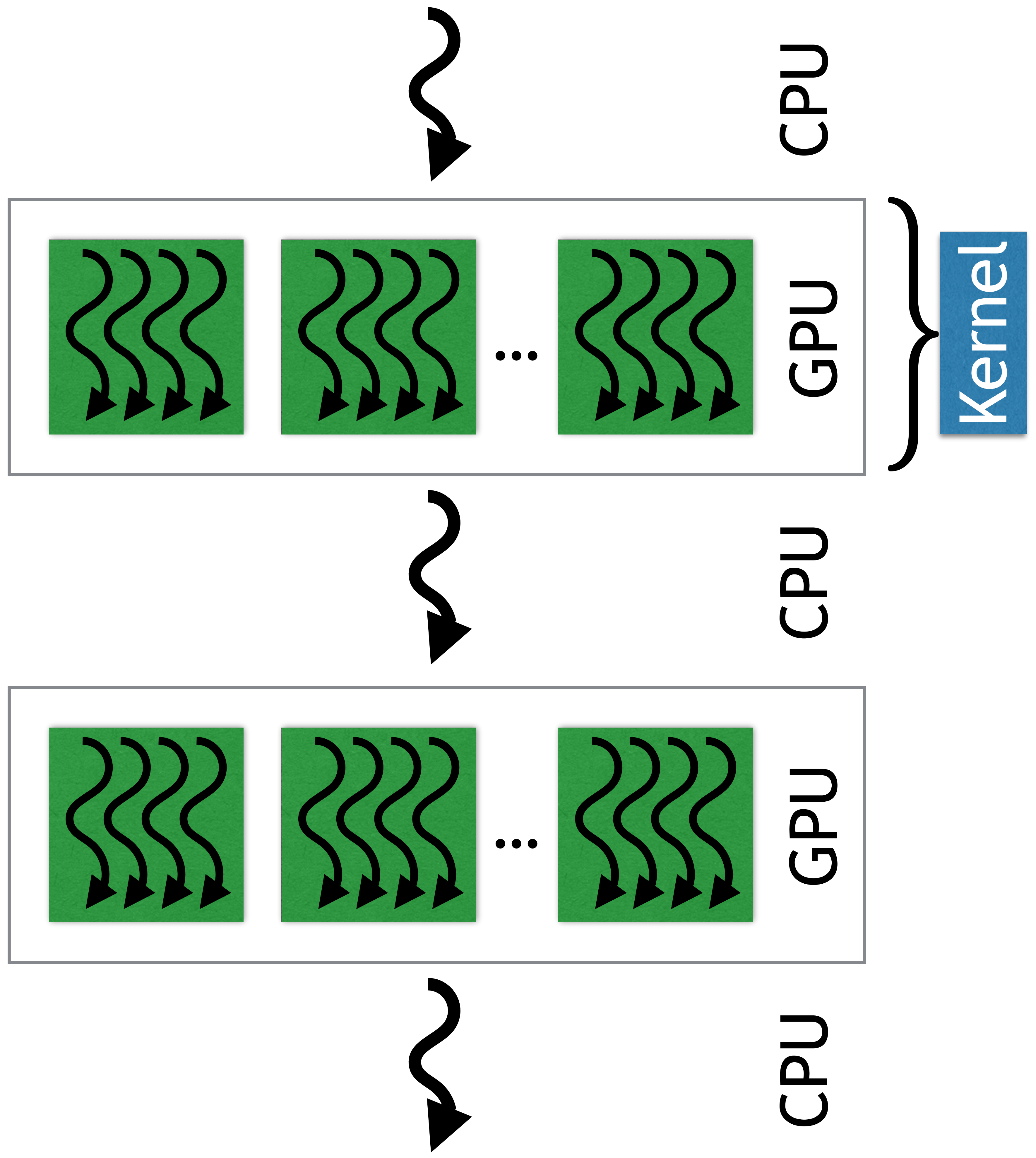
CUDA program consists of CPU & GPU part

- CPU part: part of the program with no or little parallelism
- GPU part: high parallel part, SPMD-style



PROGRAMMING MODEL

- CUDA program consists of CPU & GPU part
  - CPU part: part of the program with no or little parallelism
  - GPU part: high parallel part, SPMD-style
- Concurrent execution
  - Non-blocking thread execution
  - Explicit synchronization



PROGRAMMING MODEL

CUDA program consists of CPU & GPU part

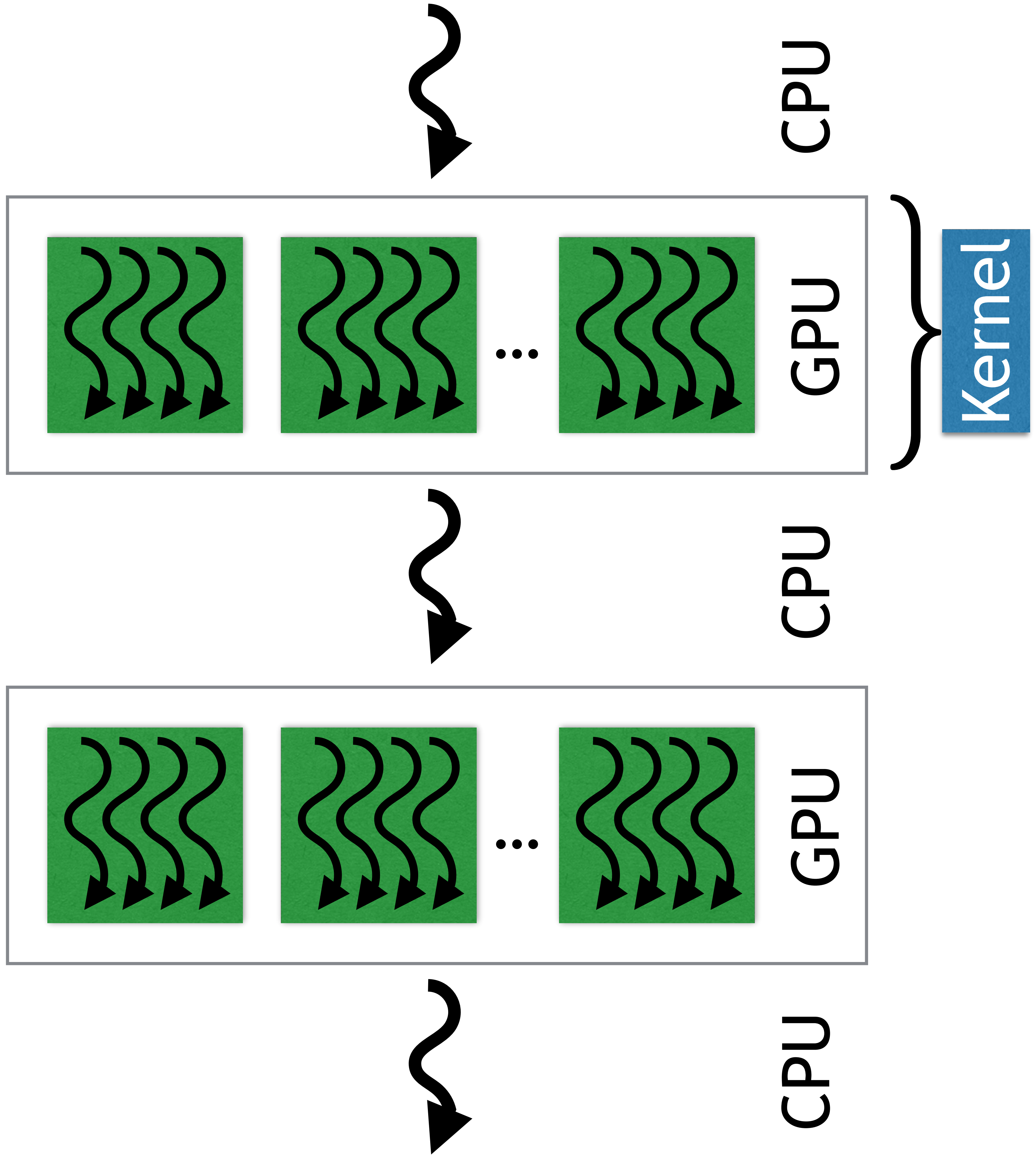
- CPU part: part of the program with no or little parallelism
- GPU part: high parallel part, SPMD-style

Concurrent execution

- Non-blocking thread execution
- Explicit synchronization

C Extension with three main abstractions

1. Hierarchy of threads
2. Shared memory
3. Barrier synchronization



PROGRAMMING MODEL

CUDA program consists of CPU & GPU part

- CPU part: part of the program with no or little parallelism
- GPU part: high parallel part, SPMD-style

Concurrent execution

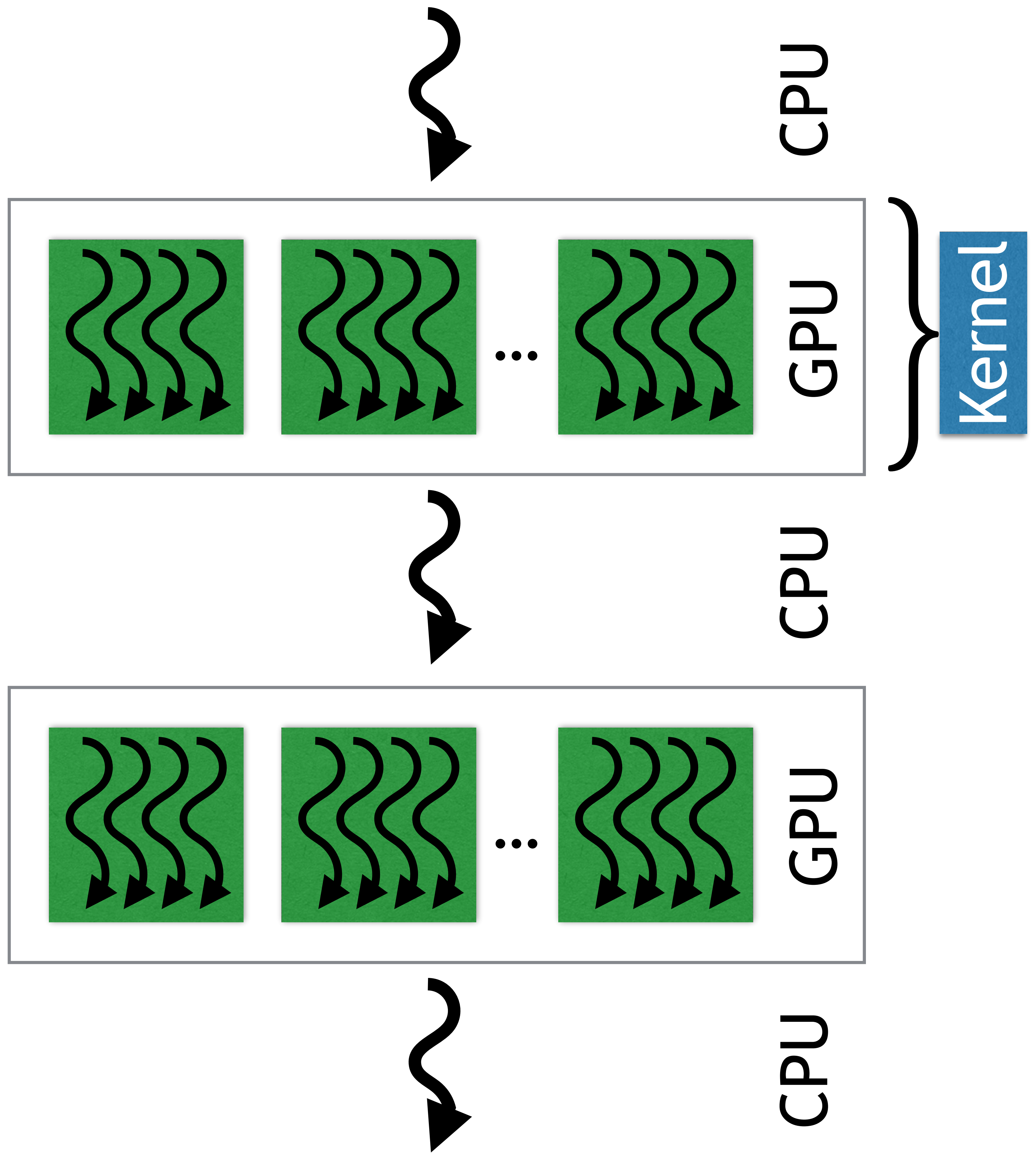
- Non-blocking thread execution
- Explicit synchronization

C Extension with three main abstractions

1. Hierarchy of threads
2. Shared memory
3. Barrier synchronization

Exploiting parallelism

- Fine-grain data-level parallelism (DLP)
- Thread-level parallelism (TLP)



PROGRAMMING MODEL

CUDA program consists of CPU & GPU part

- CPU part: part of the program with no or little parallelism
- GPU part: high parallel part, SPMD-style

Concurrent execution

- Non-blocking thread execution
- Explicit synchronization

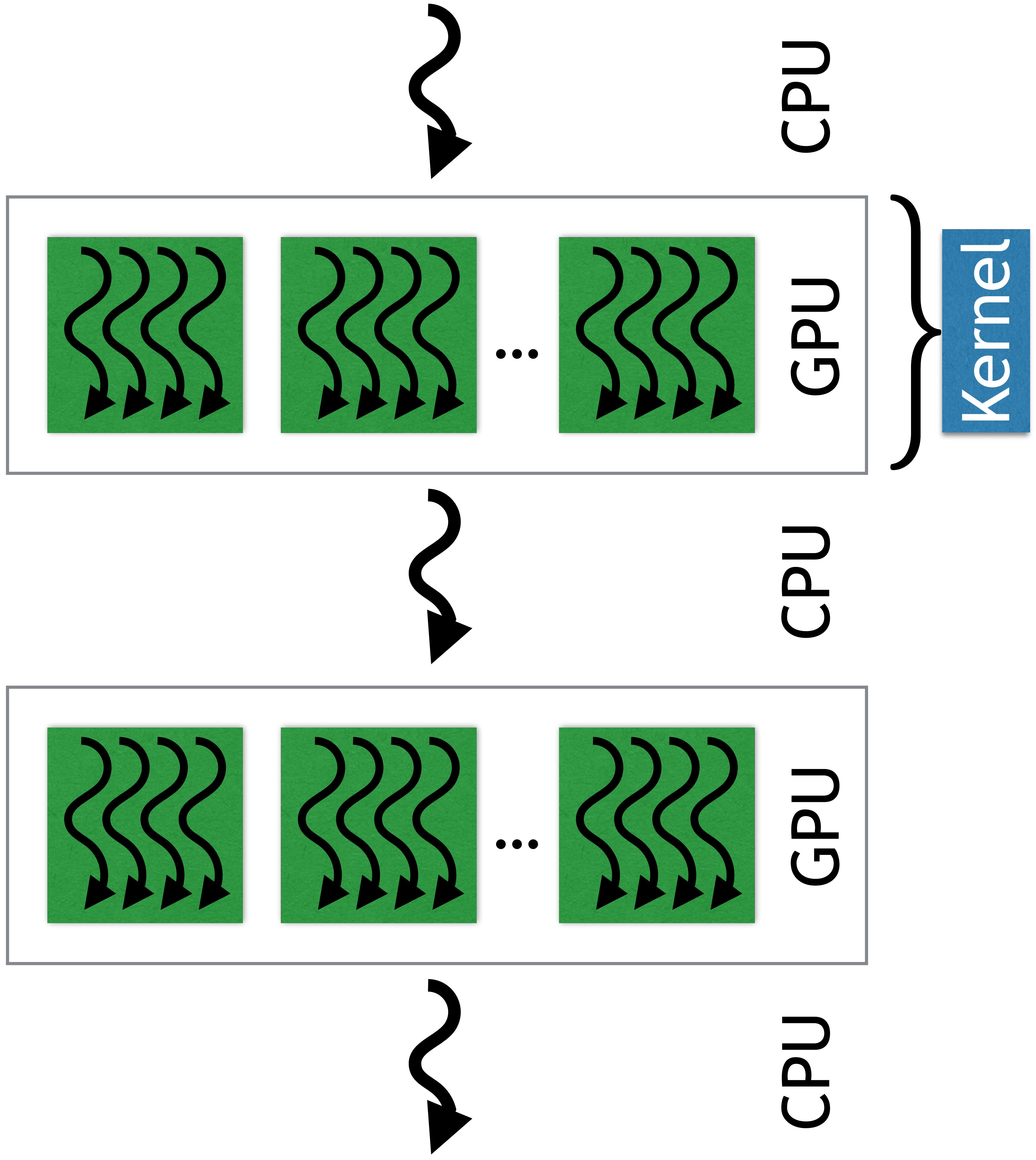
C Extension with three main abstractions

1. Hierarchy of threads
2. Shared memory
3. Barrier synchronization

Exploiting parallelism

- Fine-grain data-level parallelism (DLP)
- Thread-level parallelism (TLP)

Inner loops
Threads
Kernels



JUST-IN-TIME COMPILATION

Device code only supports C-subset of C++ (getting better)

Compile with nvcc

- Compiler Driver
- Calls other tools as required
- cudacc, g++, clang, …

Output

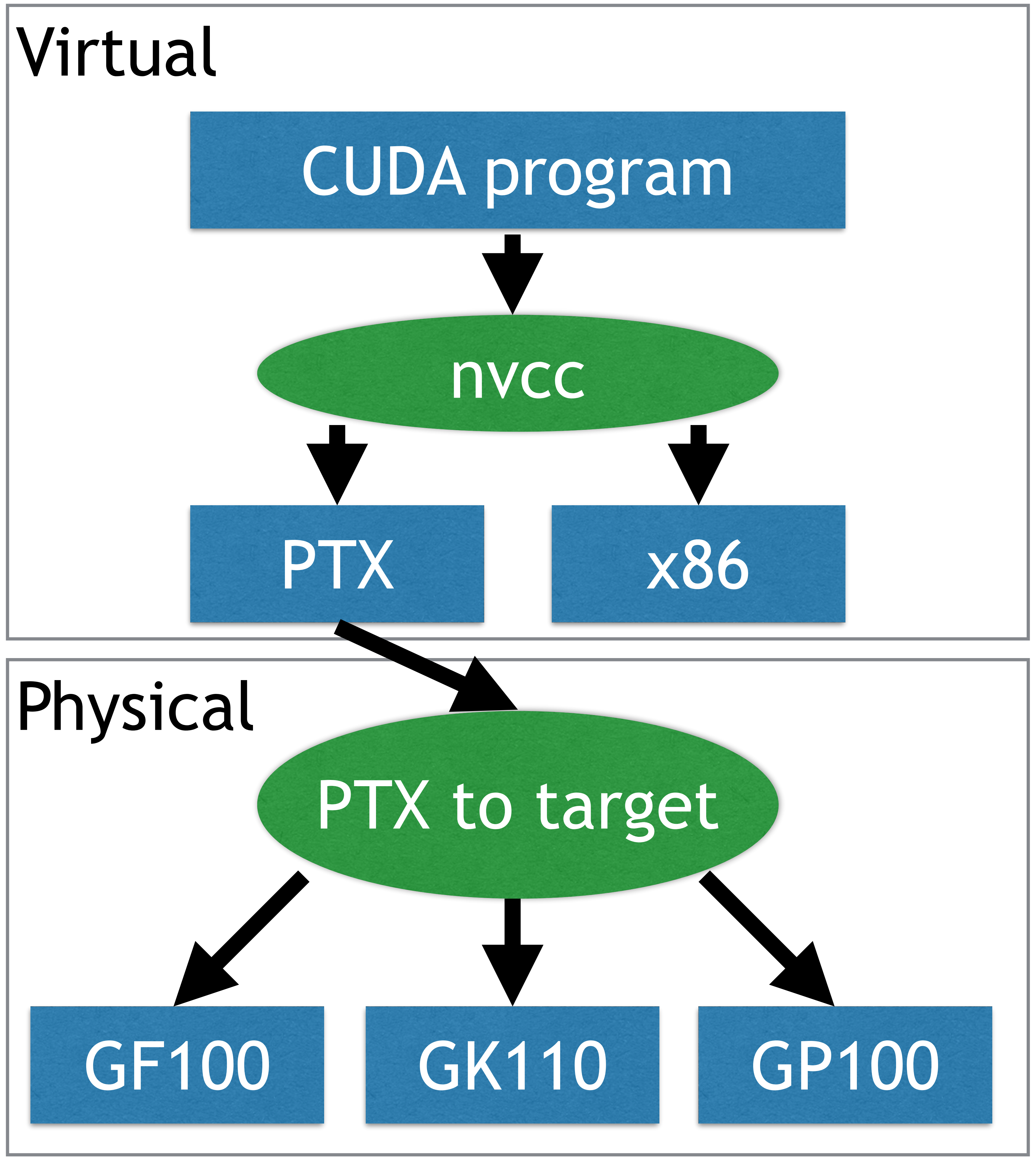
- C code (host CPU Code)
- Either PTX object code, or source code for run-time interpretation

PTX (Parallel Thread Execution)

- Virtual Machine and ISA
- Execution resources and state

Linking

- CUDA runtime library cudart
- CUDA core library cuda



SAXPY EXAMPLE

$$y[i] = \alpha \cdot x[i] + y[i]$$

SAXPY: Scalar Alpha X Plus Y

Simple test to compare GPU and CPU performance

- Objective: runtime reduction
- Max. gridSize * threadsPerBlock elements
- 65535*1k -> ~ 64M elements
- Memory requirement = 32M elements * 2 arrays * 4 Byte/element = 256MB

Source code contains kernels for the GPU and the CPU

CUDA EXAMPLE

Kernel definition:

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

Host <-> Device interaction:

Kernel execution:

Host <-> Device interaction:

```
int main(void)
{
    int N = 20 * (1 << 20);
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

    for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
    }

    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
    saxpy<<<(N+511)/512, 512>>>(N, 2.0f, d_x, d_y);

    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

    cudaDeviceSynchronize();

    // Free memory
    cudaFree(d_X);
    cudaFree(d_Y);

    // Do some printing
}
```

LOW-LEVEL LIBRARIES

Require good understanding of the CUDA execution model

cuda-python

Just plain CUDA C++ with some Python for device control

numba-cuda

CUDA, but as Python not as C++, still very close to CUDA. JIT-compiled

Triton

Abstraction to Tensor operations, less flexible, but often better optimized

NUMBA-CUDA

Imports:

Kernel definition:

Host <-> Device interaction:

Kernel execution:

```
import numpy as np
from numba import cuda

# Kernel definition
@cuda.jit
def f(a, x, y):
    # like threadIdx.x + (blockIdx.x * blockDim.x)
    tid = cuda.grid(1)
    size = len(y)

    if tid < size:
        y[tid] = a*x[tid] + y[tid]

# Vector allocation and copy to Device
N = 100000
x = cuda.to_device(np.random.random(N))
y = cuda.to_device(np.random.random(N))
alpha = 2.

# Kernel execution
# Enough threads per block for several warps per block
nthreads = 256
# Enough blocks to cover the entire vector depending on its length
nblocks = (len(a) // nthreads) + 1
f[nblocks, nthreads](a, x, y)

# Copying data back to host and print
print(y.copy_to_host())
```

HIGH-LEVEL

- Tachi
  - Still able to write custom GPU kernels
  - No more detailed GPU thread control required
- **CuPy**
  - No more kernel writing
  - Basically Numpy, but on a GPU
  - Adds a few functions for data transfer and device control
- Deep Learning focused (include AutoGrad)
  - Jax
  - TensorFlow
  - PyTorch

CUPY

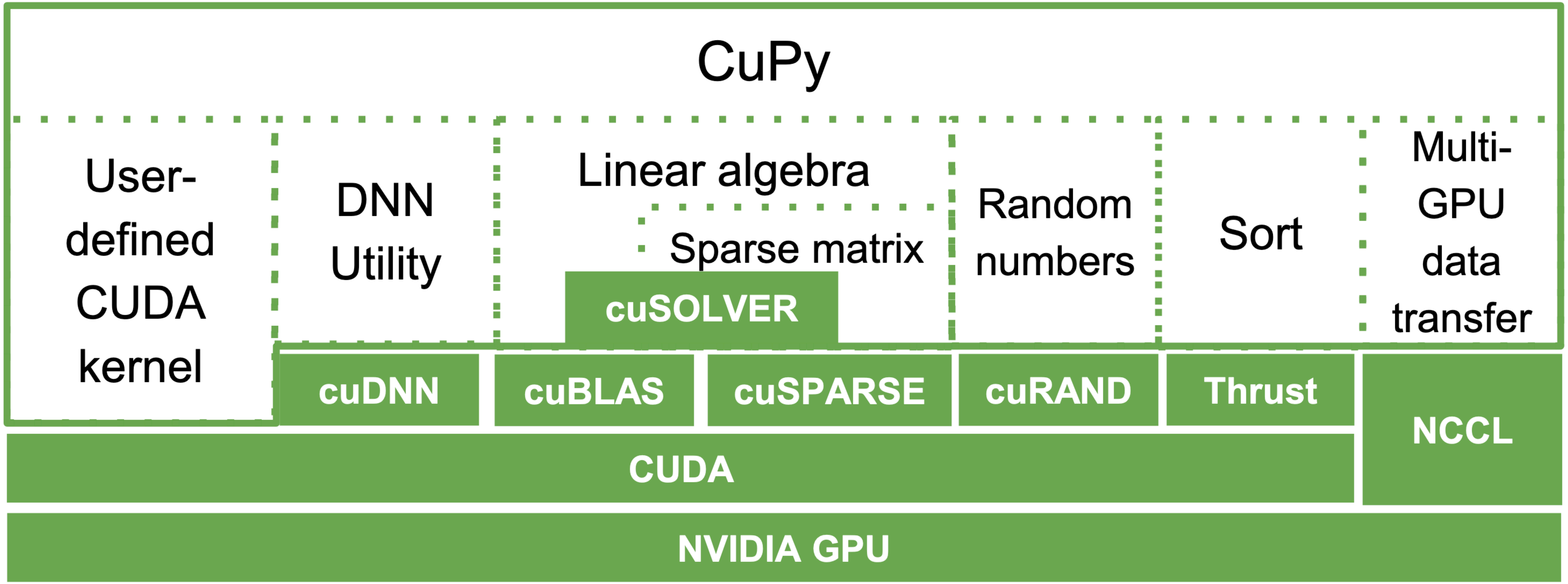
Imports:

Host <-> Device interaction:

Execute operation:

```python
import cupy
import numpy as np

# Vector allocation and copy to Device
N = 1000000
x = cupy.asarray(np.random.random(N))
y = cupy.asarray(np.random.random(N))
alpha = 2.0

# Execute saxpy op
y += alpha * x

# Explicit copy back to host and print
# (implicit often also works)
print(cupy.asnumpy(y))
```

CUPY


CuPy
User-defined CUDA kernel
DNN Utility
Linear algebra
Sparse matrix
cuSOLVER
Random numbers
Sort
Multi-GPU data transfer
cuDNN
cuBLAS
cuSPARSE
cuRAND
Thrust
NCCL
CUDA
NVIDIA GPU

WRAPPING UP

SUMMARY

GPU Computing is using GPUs for non-graphical computations

- More performance (compute, memory)
- Better energy-efficiency (how I learned to love the picoJoule)

Key differences to a CPU

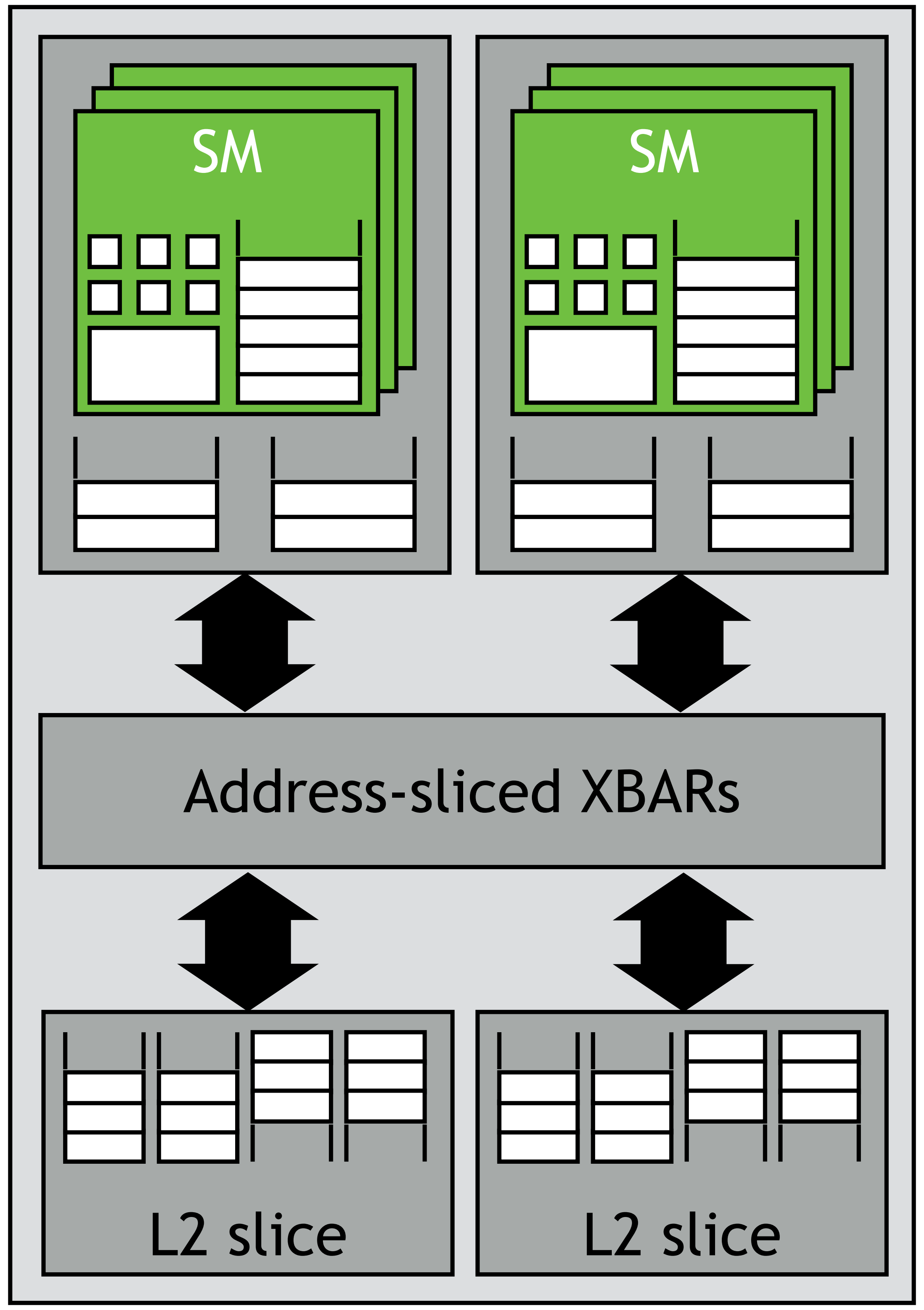
- Much (many much'es) more parallelism
- Latency is not minimized, but tolerated
- Offload compute model
- No general-purpose programming (yet?)
- Memory capacity is small
- Single-thread performance is a nightmare

Programming GPUs

- Both low- and high-level abstractions are available
- The best library strongly depends on the use-case

More reading

- https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law



5 MIN BREAK

Then Exercises

NN from scratch in Numpy by Group 4 (Arjan, Iona, Julian, Jonas)

THIS WEEKS EXERCISE

NN from scratch in Numpy by Group 4 (Arjan, Iona, Julian, Jonas)

EXERCISE 2

General comments:

- Two different gradient calculations were used for the sigmoid
  - See next slide
  - Both are valid
- Some groups didn’t normalize the linear layer gradient to the batch size
  - LR suddenly depends on the batch size
  - Models effectively learn with a higher LR than set

EXERCISE 2

Direct gradient formulation:

```
class Sigmoid():
    def __init__(self, in_features: int, batch_size: int):
        super(Sigmoid, self).__init__()
        self.input = np.zeros(batch_size)

    def forward(self, input):
        self.input = input
        return 1./(1.+np.exp(-input))

    def backward(self, grad_output):
        grad_input = grad_output * np.exp(-self.input) / np.power(1. + torch.exp(-self.input), 2)
        return grad_input
```

Formulation with the sigmoid itself:

```
class Sigmoid:
    def __init__(self, in_features: int, batch_size: int):
        super(Sigmoid, self).__init__()
        self.input = np.zeros(batch_size)

    def forward(self, input):
        self.input = input
        output = 1 / (1 + np.exp(-input))
        return output

    def backward(self, grad_output):
        sigmoid = self.forward(self.input)
        grad_input = sigmoid * (1 - sigmoid) * grad_output
        return grad_input
```

EXERCISE 2

Gradient normalization for the linear layer:

```
class Linear():
    def __init__(self, in_features: int, out_features: int, batch_size: int, lr=0.1):
        super(Linear, self).__init__()
        self.batch_size = batch_size
        self.lr = lr
        self.weight = np.random.normal(size=(in_features, out_features)) * np.sqrt(1. / in_features)
        self.bias = np.random.normal(size=(out_features,)) * np.sqrt(1. / in_features)
        self.grad_weight = np.zeros((in_features, out_features))
        self.grad_bias = np.zeros(out_features)
        self.input = np.zeros((batch_size, in_features))


    def forward(self, input):
        self.input = input
        output = np.matmul(input, self.weight) + self.bias
        return output

    def backward(self, grad_output):
        grad_input = np.matmul(grad_output, self.weight.T)
        self.grad_weight = (1. /self.batch_size) * np.matmul(self.input.T, grad_output)
        self.grad_bias = (1. /self.batch_size) * grad_output.sum(0)
        return grad_input

    def update(self):
        self.weight = self.weight - self.lr * self.grad_weight
        self.bias = self.bias - self.lr * self.grad_bias
```

NEXT WEEKS EXERCISE

NEXT WEEKS EXERCISE

Port numpy implementation to CuPy

- Experiment with different network sizes
- Compare CPU and GPU execution times

Submission deadline: Tuesday 09:00 am

https://csg.ziti.uni-heidelberg.de/teaching/ap_nn_from_scratch_materials/