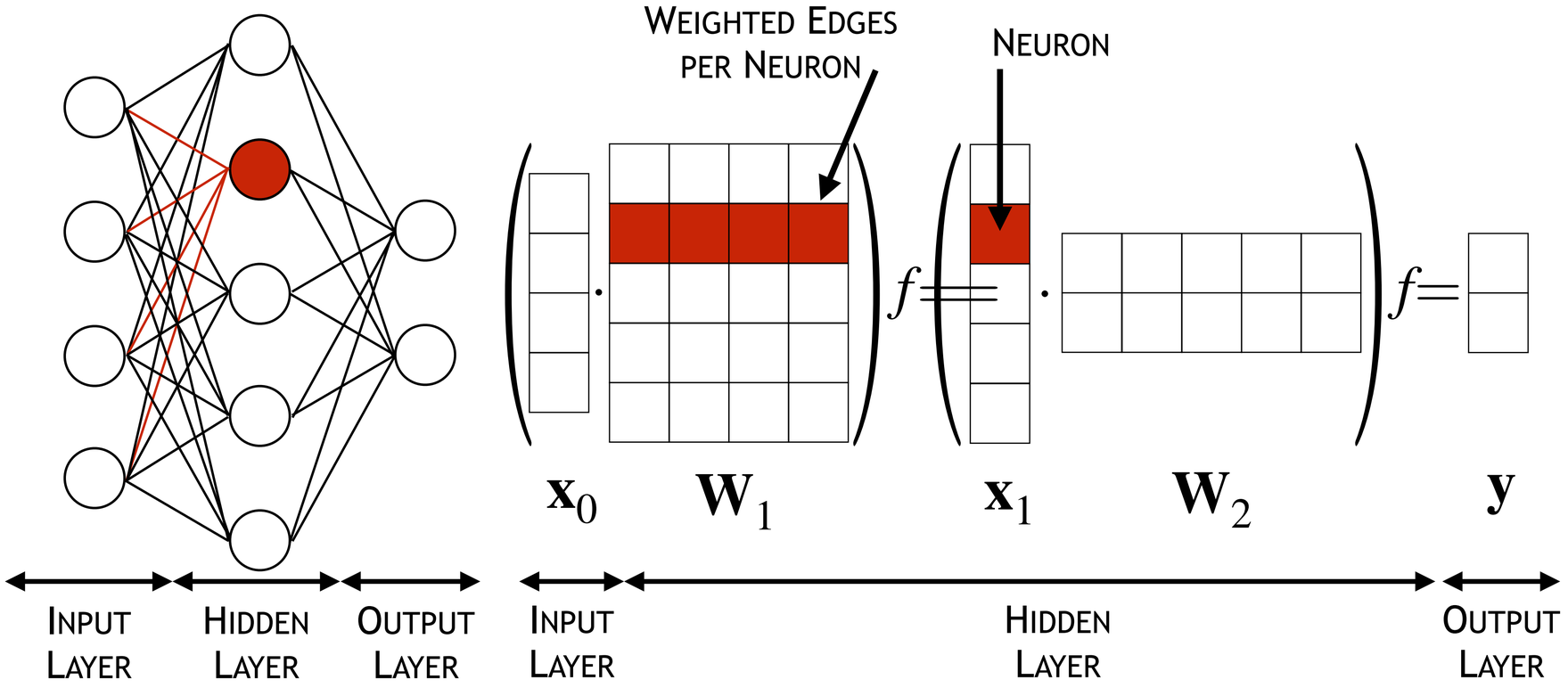
NEURAL NETWORKS FROM SCRATCH
LECTURE 03 - GPU PROGRAMMING

Hendrik Borras, Robin Janssen
{hendrik.borras, robin.janssen}@ziti.uni-heidelberg.de,
HAWAII Lab, 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?

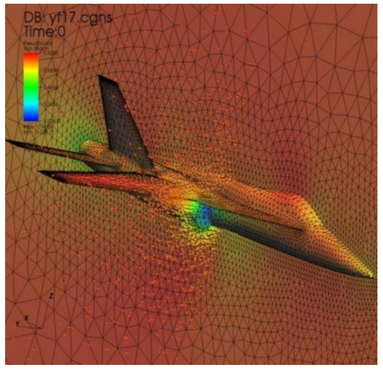
GPU COMPUTING

GPU BACKGROUND

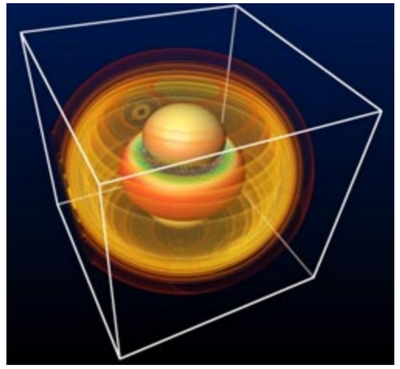
- Primary use in gaming
  - Each console has a (powerful) GPU
  - Meantime photorealistic
- Graphics: big, multi-dimensional floating-point operations in parallel
  - Programmable
- Since ~2007 used for general-purpose computing
  - CUDA

NVIDIA

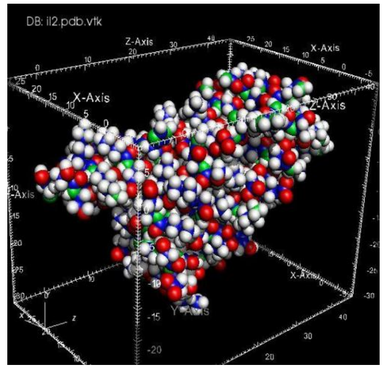
https://80.lv/articles/ue4arch-interview-about-realiscit-environments-in-ue4/



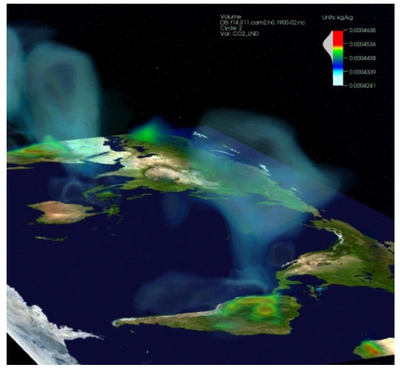
CFD



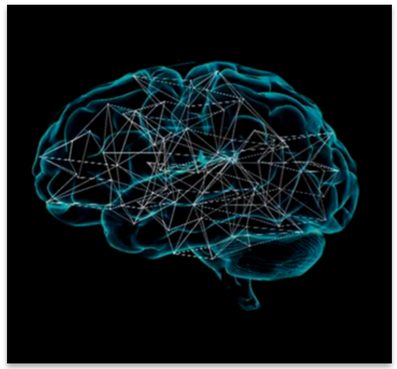
Cosmology



Molecular Dynamics



Weather/climate research

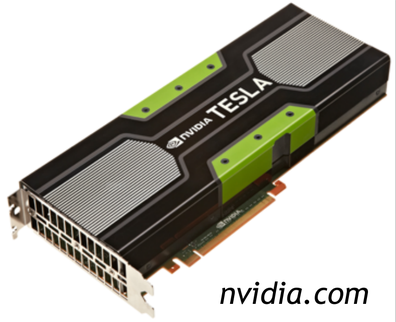


Machine Learning

Tera-FLOP/s
September 1997

15 years

Peta-FLOP/s
November 2012

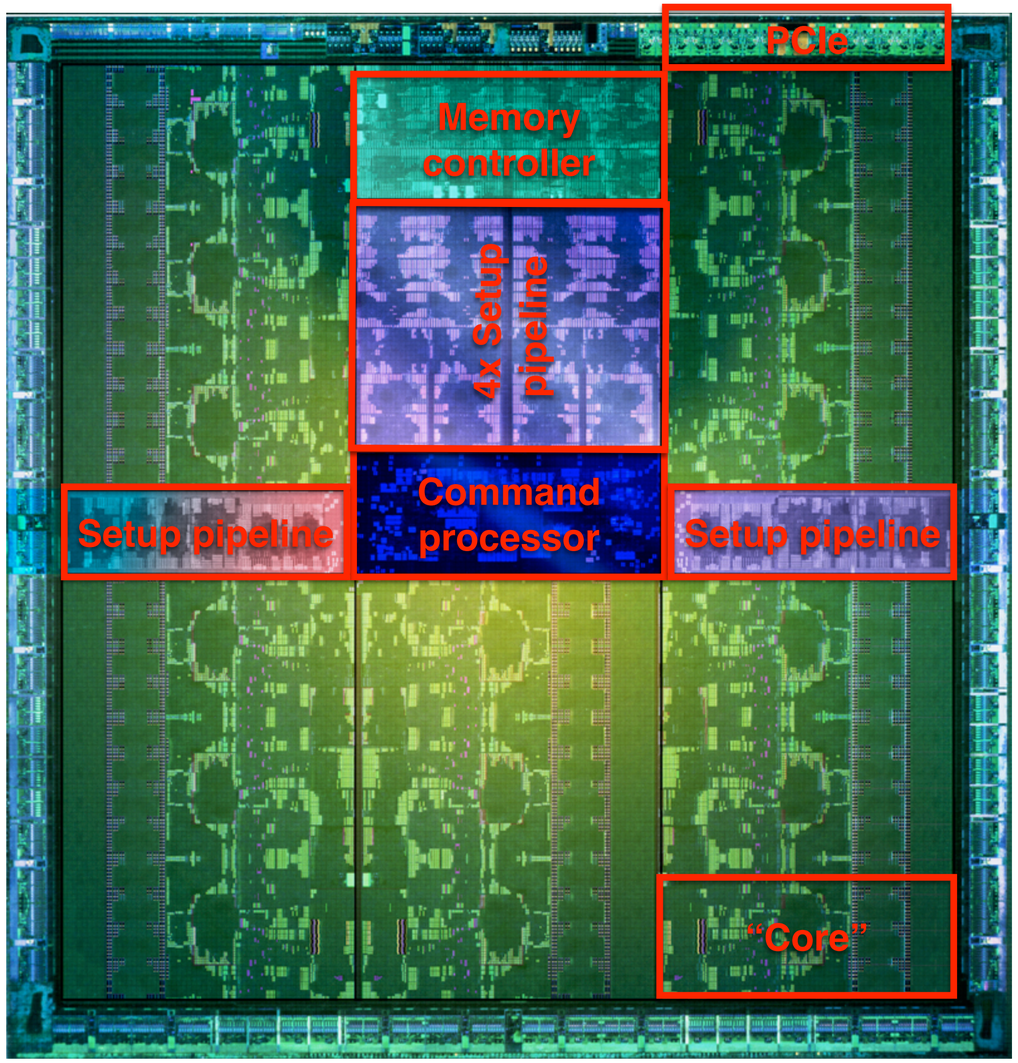


nvidia.com

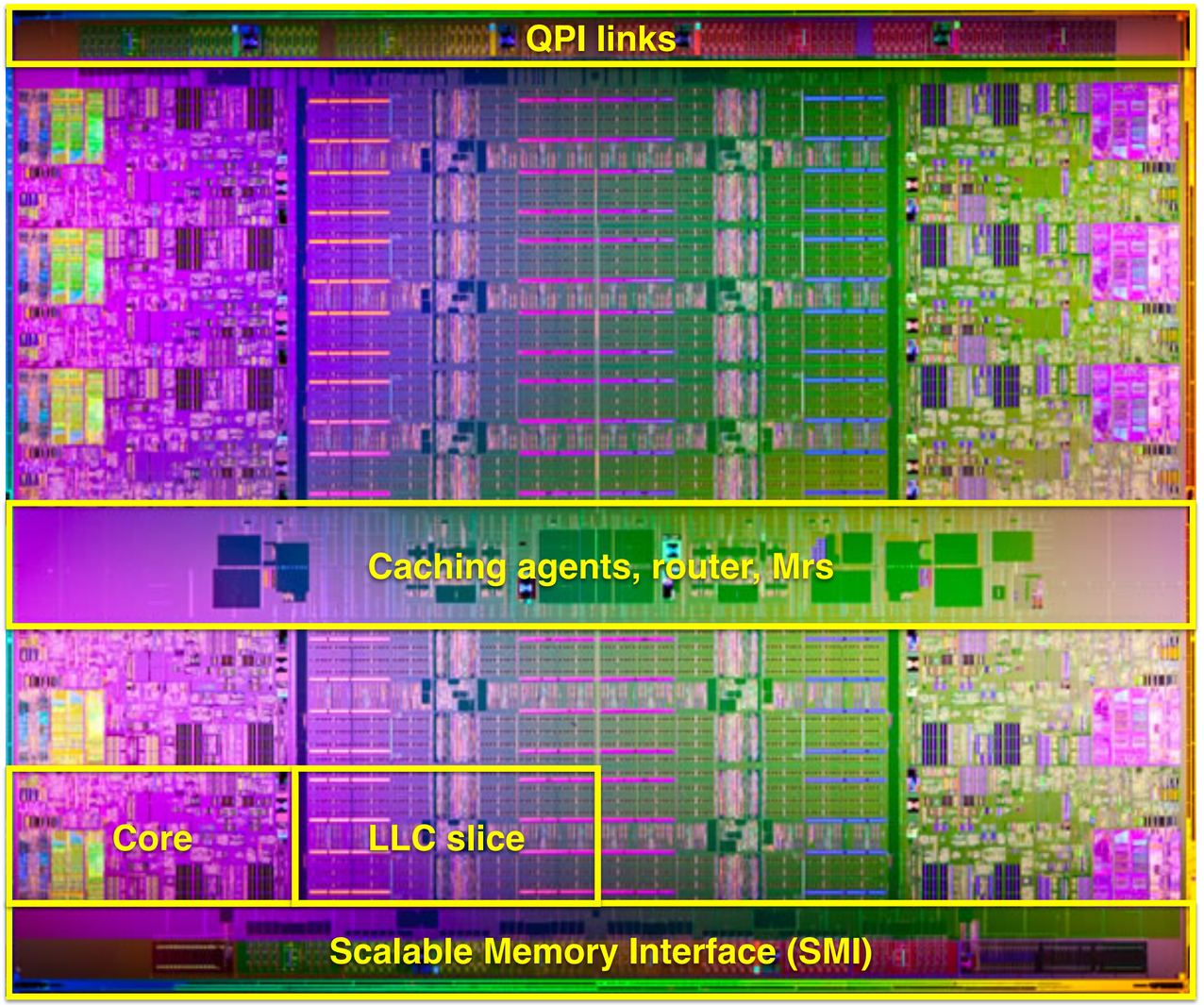
10 years

Exa-FLOP/s
June 2022

DIE SHOTS - CPU OR GPU?



NVIDIA Kepler- GK110

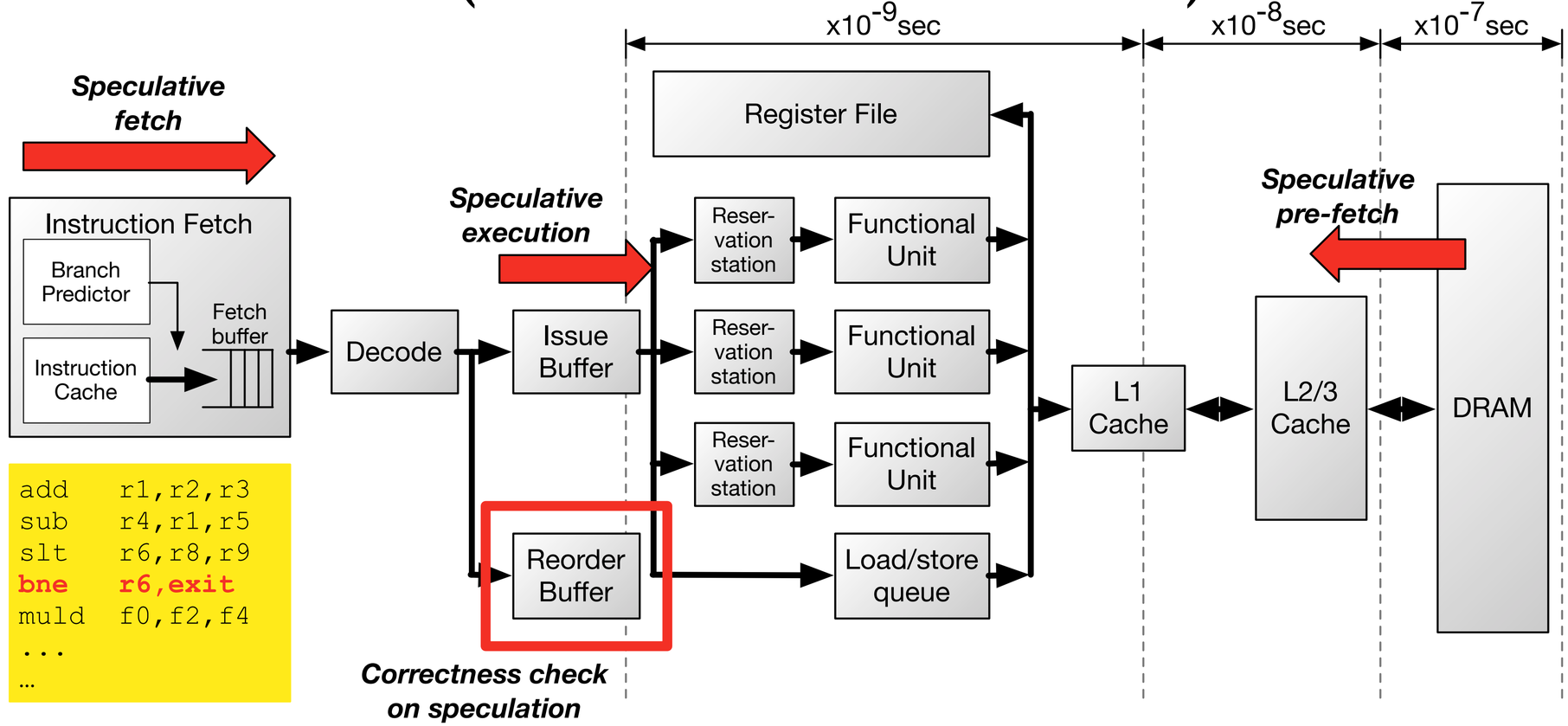


Intel Xeon E7 - Westmere-EX

	Xeon E5-2699v4 (Broadwell, 2016)	Tesla K20 (GK110) (Kepler, 2012)	NVIDIA P100 (GP100) (Pascal, 2016)	NVIDIA V100 (GV100) (Volta, 2017)
		~3-4	~1.5	
Core count	22 cores 2 FP-ALUs/core	13 SMs 64/SM(DP), 192/SM(SP)	56 SMs 32/SM(DP), 64/SM(SP)	84 SMs 32/SM(DP), 64/SM(SP)
Frequency	2.2-3.6GHz	0.7GHz	1.328-1.480GHz	1.455GHz
Effective vector width	256bit (SP/DP) AVX 2.0	1024bit (SP), 2048bit (DP) static grouping		1024bit (SP), 2048bit (DP) dynamic grouping
Peak Perf.	633.6 GF/s (DP)	1,165 GF/s (DP), SP x3	5.3 TF/s (DP), SP x2	7.5 TF/s (DP), SP x2
Use mode	latency-oriented		throughput-oriented	
Latency	minimization		toleration	
Programming	10s of threads		10,000s+ of threads	
Memory bandwidth	76.8 GB/s 128bit DDR4-2400	250 GB/s 384-bit GDDR-5	720 GB/s 4096-bit HBM2	
Memory	1.54TB	5 GB	16G	32G
Die size	456 mm^2	550mm^2	610mm^2	815mm^2
Transistor	7.2 billion	7.1 billion	15.3 billion	21.1 billion
Technology	14nm	28nm	16nm FinFET	12 nm FFN
Power	145W	250W	300W	300W
Power efficiency	4.37 GF/Watt (DP) 8.74 GF/Watt (SP)	4.66 GF/Watt (DP) 14 GF/Watt (SP)	17.66 GF/Watt (DP) 35 GF/Watt (SP)	25 GF/Watt (DP) 50 GF/Watt (SP)

HARDWARE ARCHITECTURE

MICROARCHITECTURE EXAMPLE DRIVEN BY MOORE (& DENNARD SCALING)



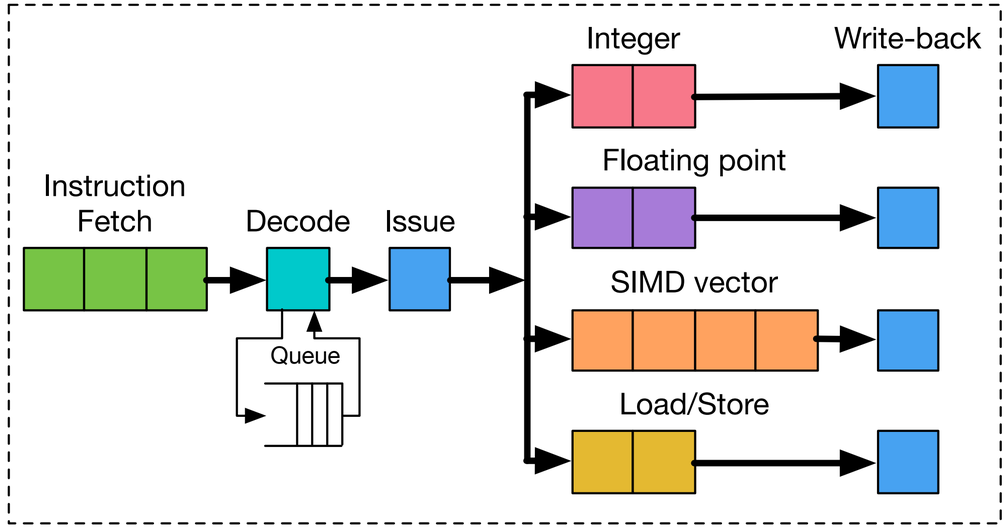
More transistors
Frequency scaling
Locality (spatial/temporal)
Predictable control flow

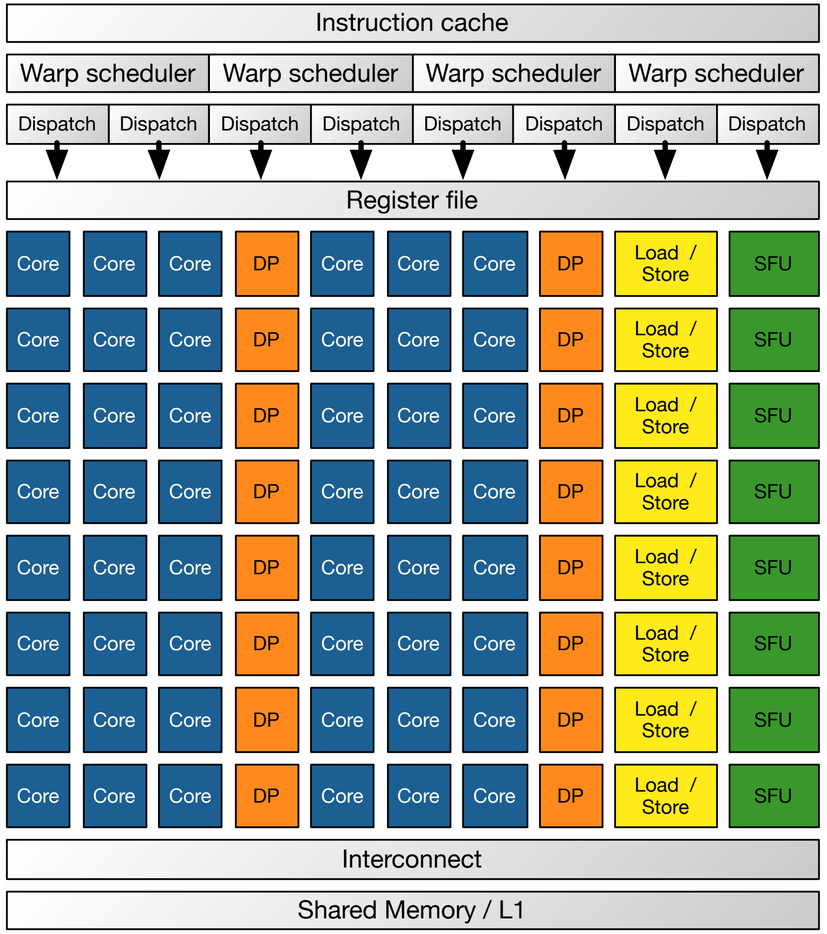
Multiple, deep pipelines
Latency minimization & hiding
Speculation everywhere

Partly by Sudha Yalamanchili, Architectural Alternatives for Energy Efficient Performance Scaling, VLSI Conference, 2013

POST-DENNARD: TRANSITION TO MASSIVELY PARALLEL MICROARCHITECTURES

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



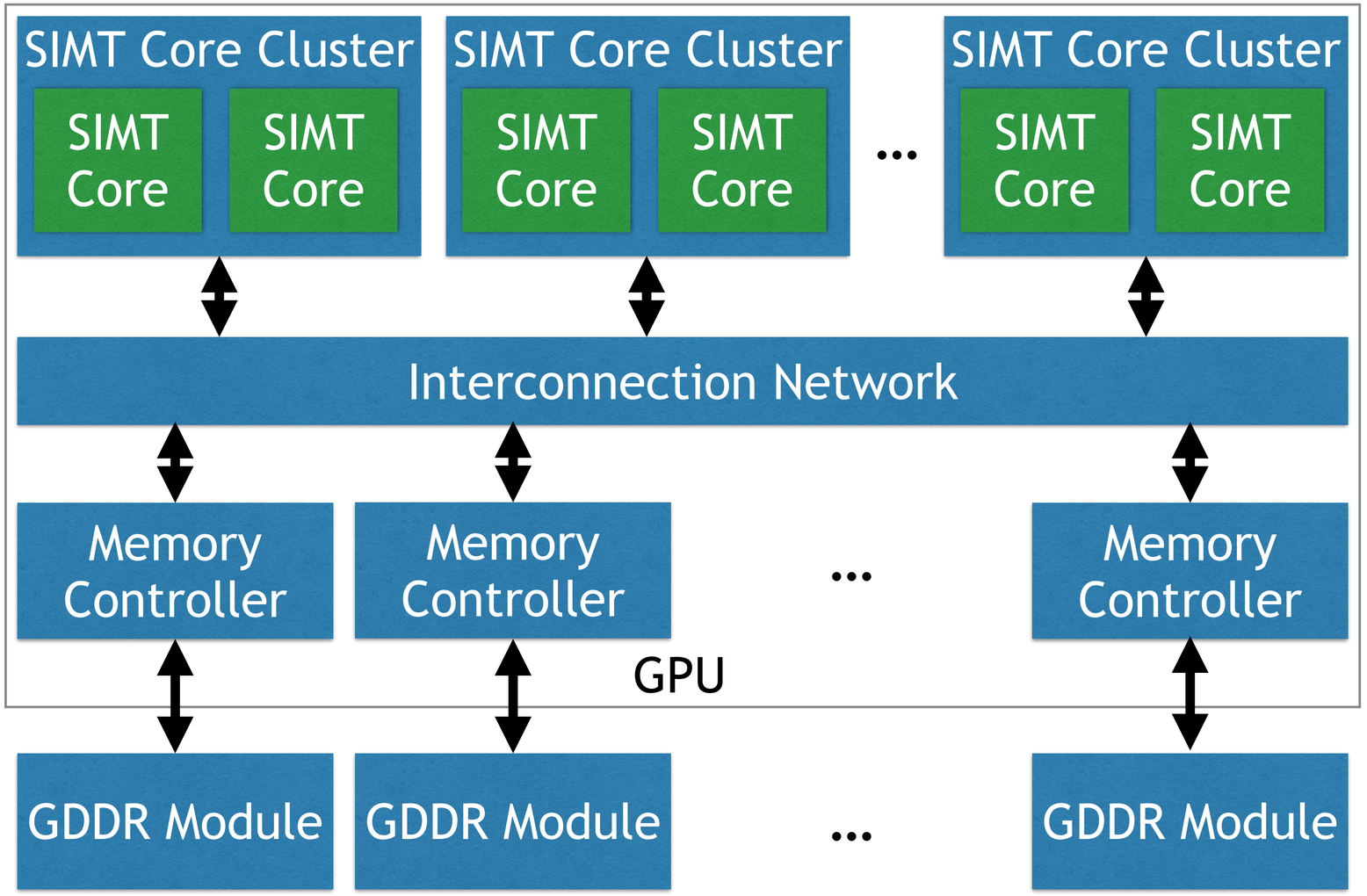


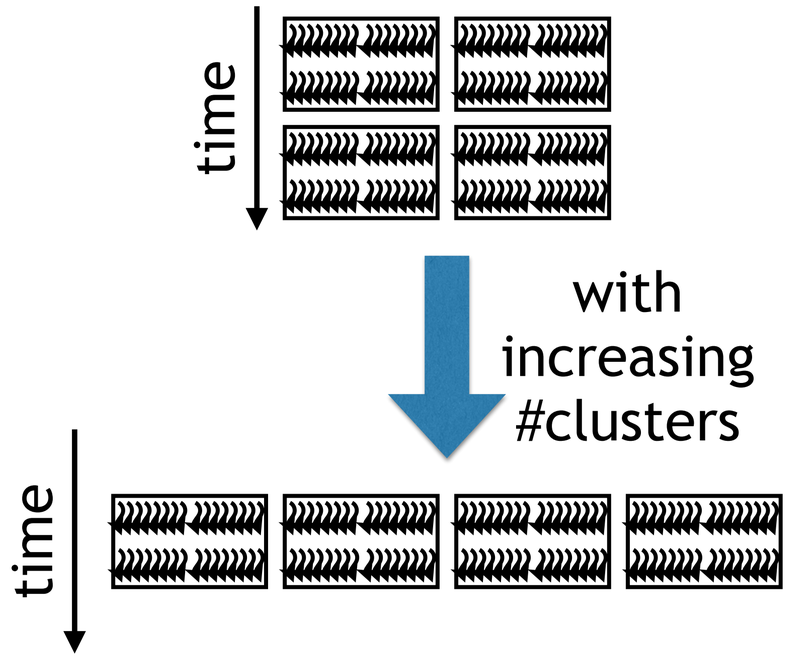
Frequency reduction
In-order pipelines

Replication

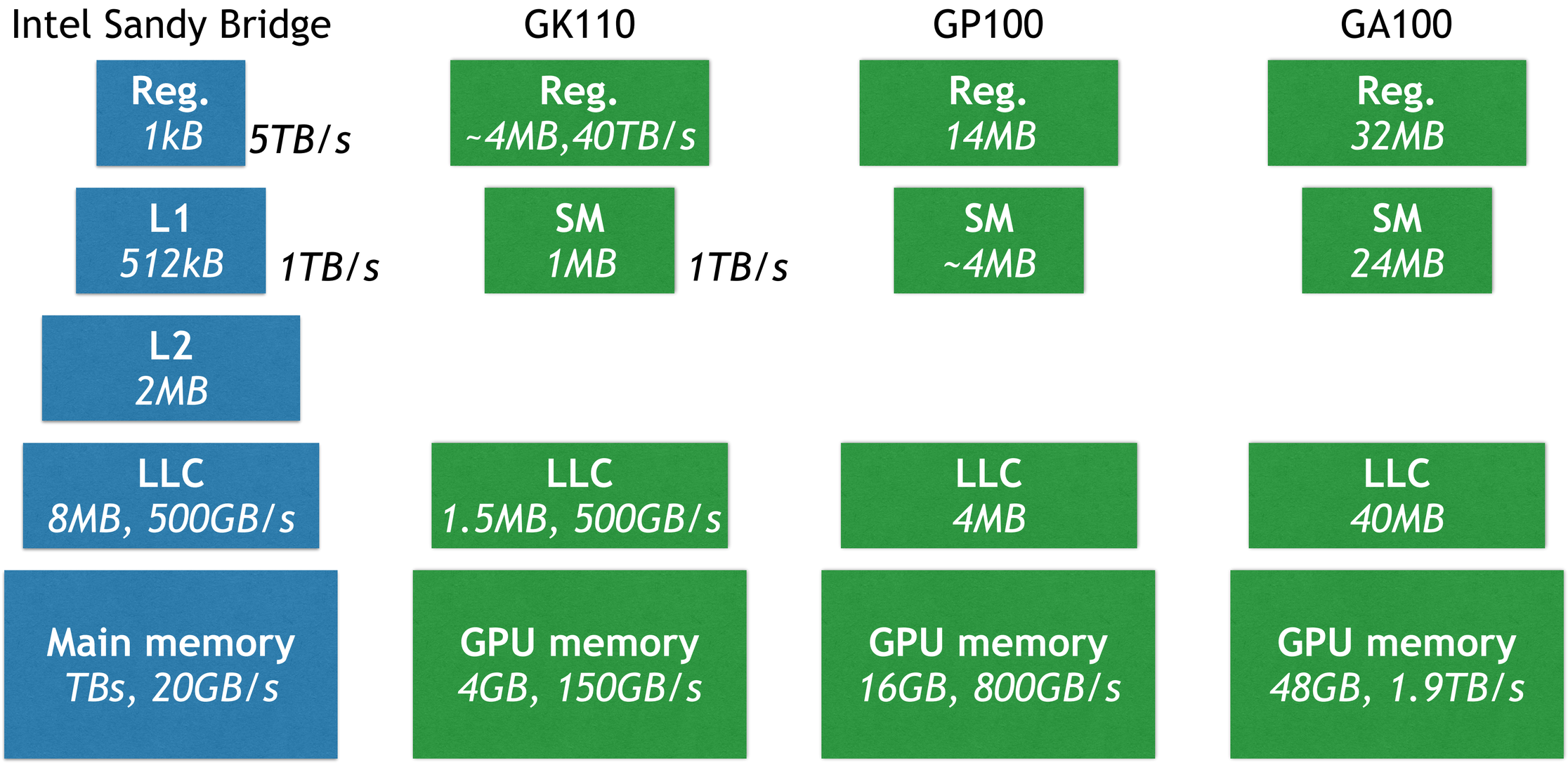
Massively parallel
Energy efficient

GPU ARCHITECTURE TOP-LEVEL VIEW





MIND THE MEMORY HIERARCHY



SOFTWARE VIEW

BULK-SYNCHRONOUS PARALLEL

In 1990, Valiant already described GPU computing pretty well

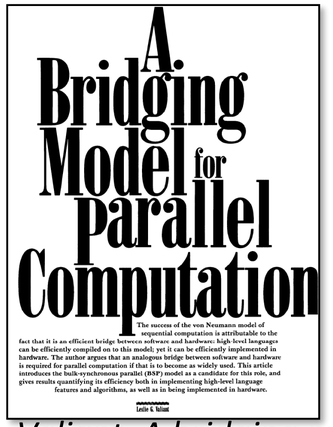
Superstep

- Compute, communicate, synchronize

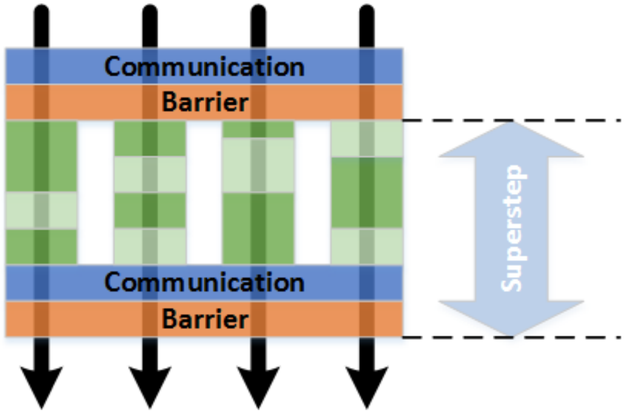
Parallel slackness: # of virtual processors v, physical processors p

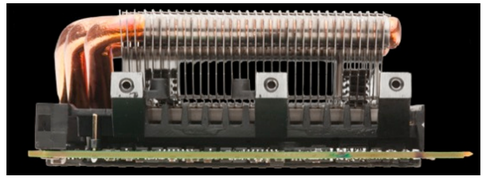
- v = 1: not viable
- v = p: unpromising wrt optimality
- v >> p: leverage slack to schedule and pipeline computation and communication efficiently

Extremely scalable, bad for unbalanced parallelism



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

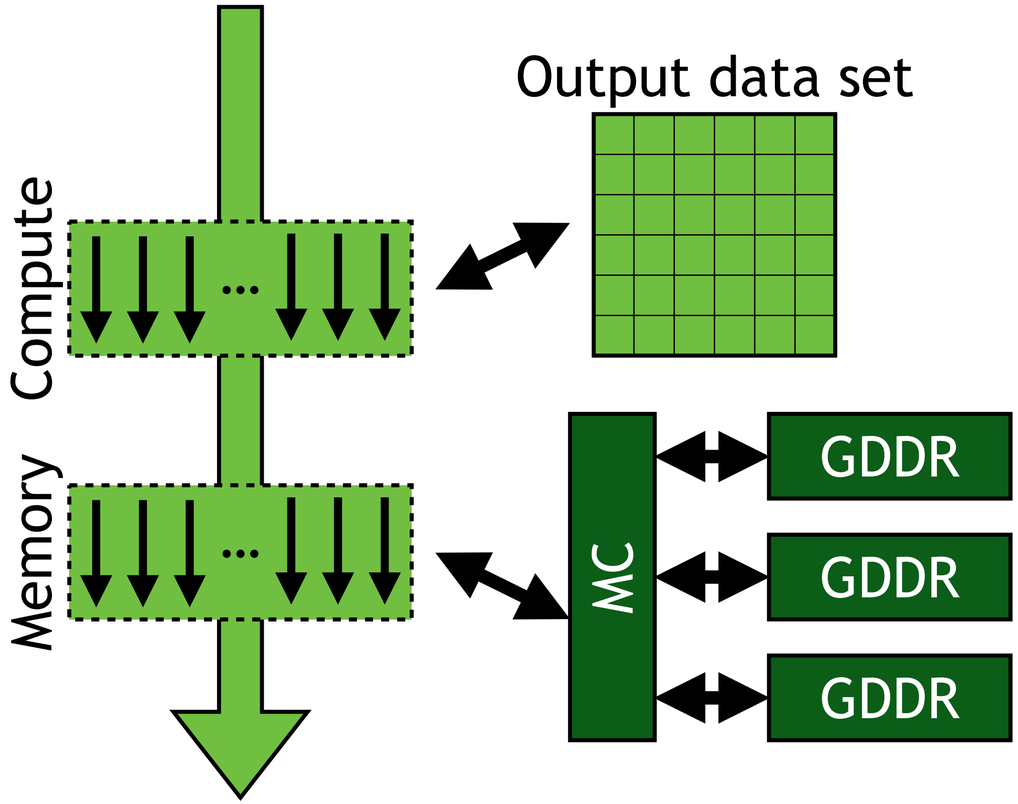
- Thread ID determines data element

GPU collaborative computing

- One thread per output element
- Schedulers exploit parallel slackness

GPU collaborative memory access

- One thread per data element



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

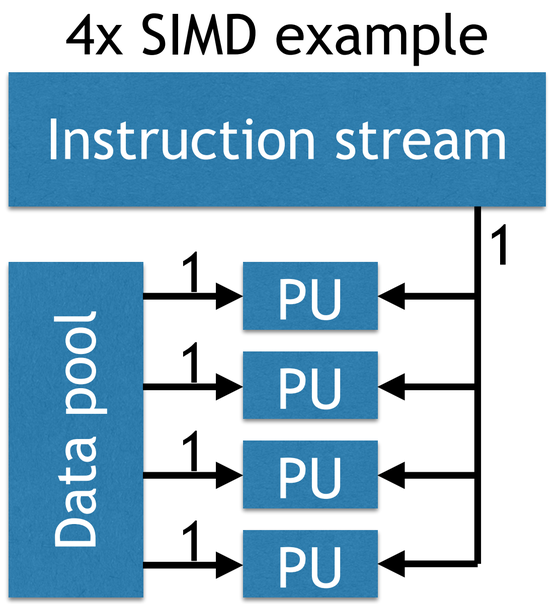
PROGRAMMABILITY OF MASSIVE PARALLELIZATION

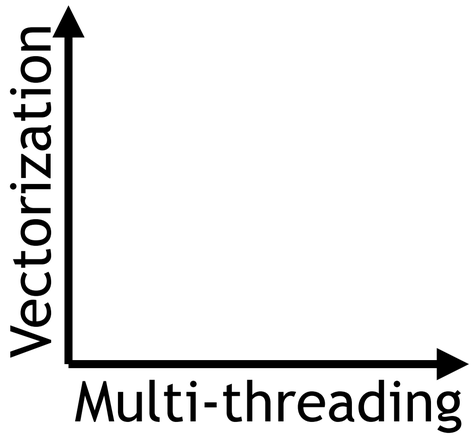
Vector ISAs are great

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

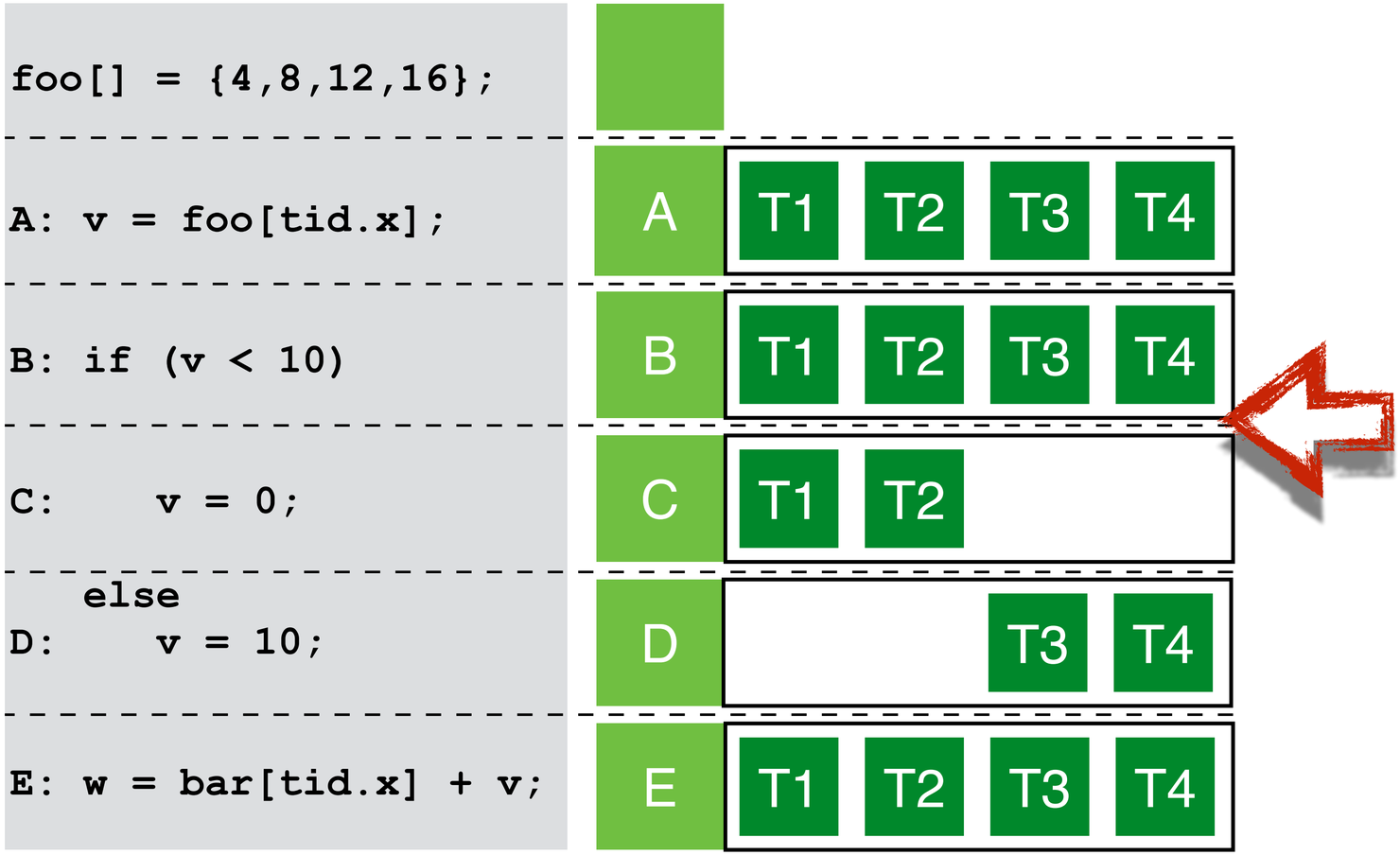
Vector ISAs are bad

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





SIMT EXECUTION MODEL



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?

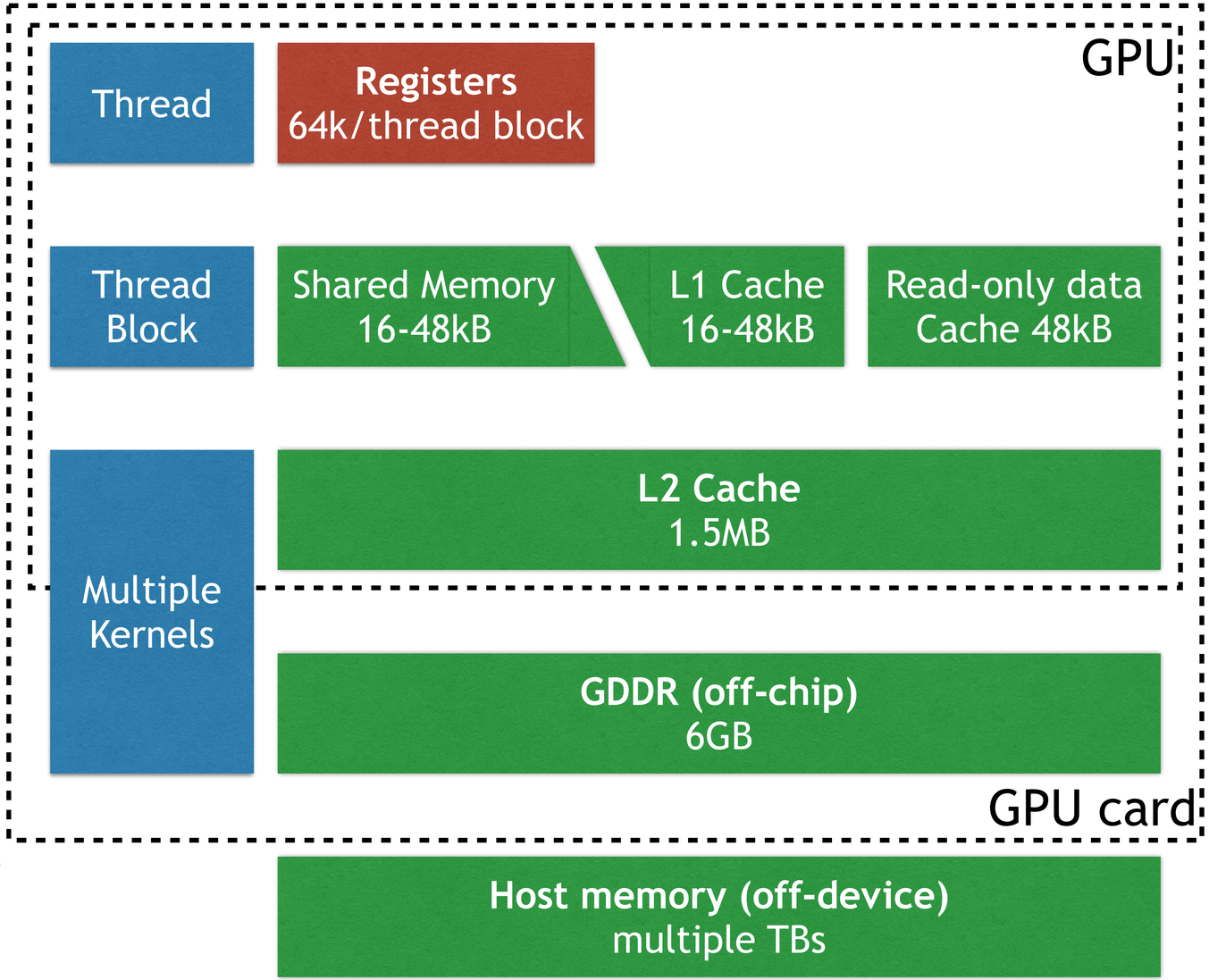
ACCESSING MEMORY

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



OUR VIEW OF A GPU

Software view: a programmable many-core scalar architecture

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

IT'S A (ALMOST) PERFECT INCARNATION OF THE BSP MODEL

Hardware view: a programmable multi-core vector architecture

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

IT'S A VECTOR ARCHITECTURE THAT HIDES ITS VECTOR UNITS

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

Concurrent execution

- Non-blocking thread execution
- Explicit synchronization

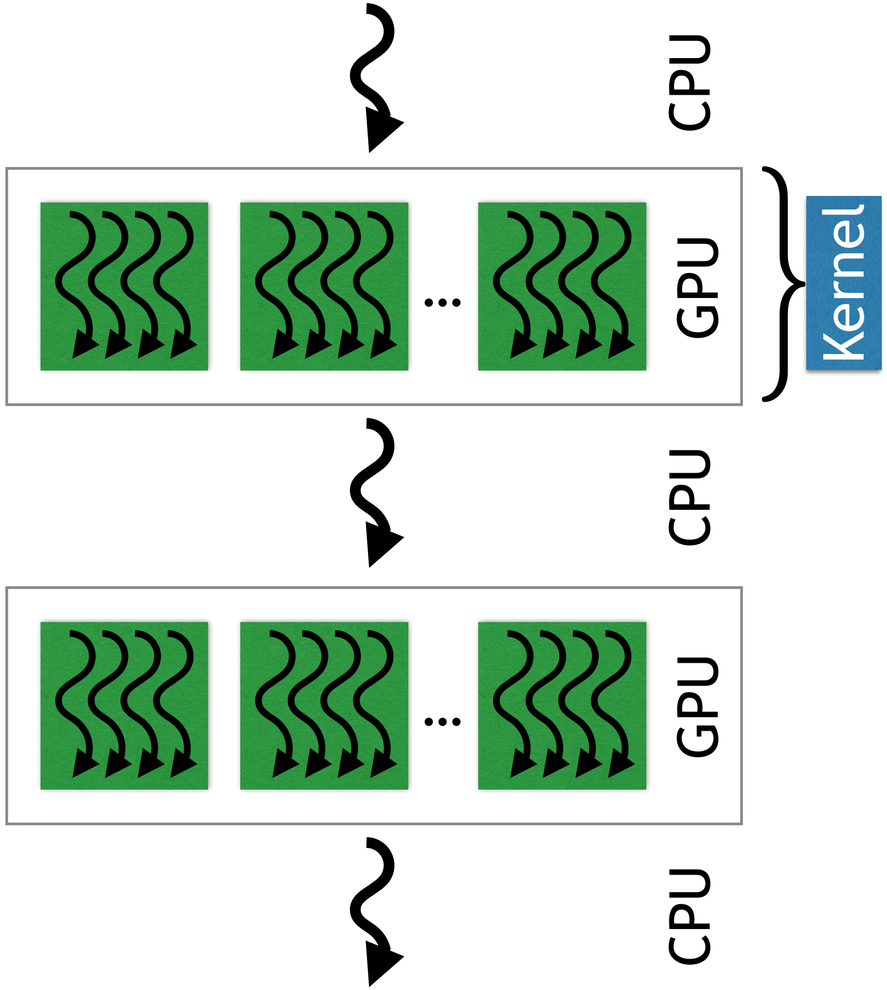
C Extension with three main abstractions

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

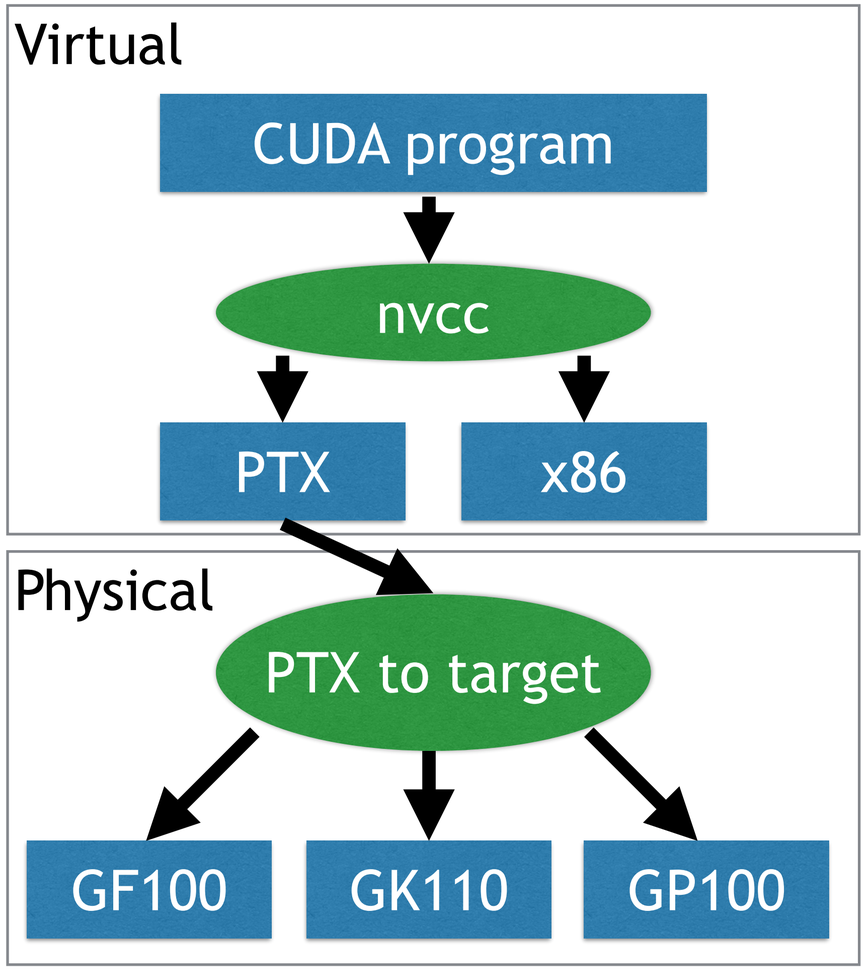
- C code (host CPU Code)
- Either PTX object code, or source code for run-time interpretation

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 demonstrate GPU programming

- Will start with lowest programming paradigm: CUDA
- End at highest level abstraction: CuPy

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

Taichi

- JIT compiled Python code as 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

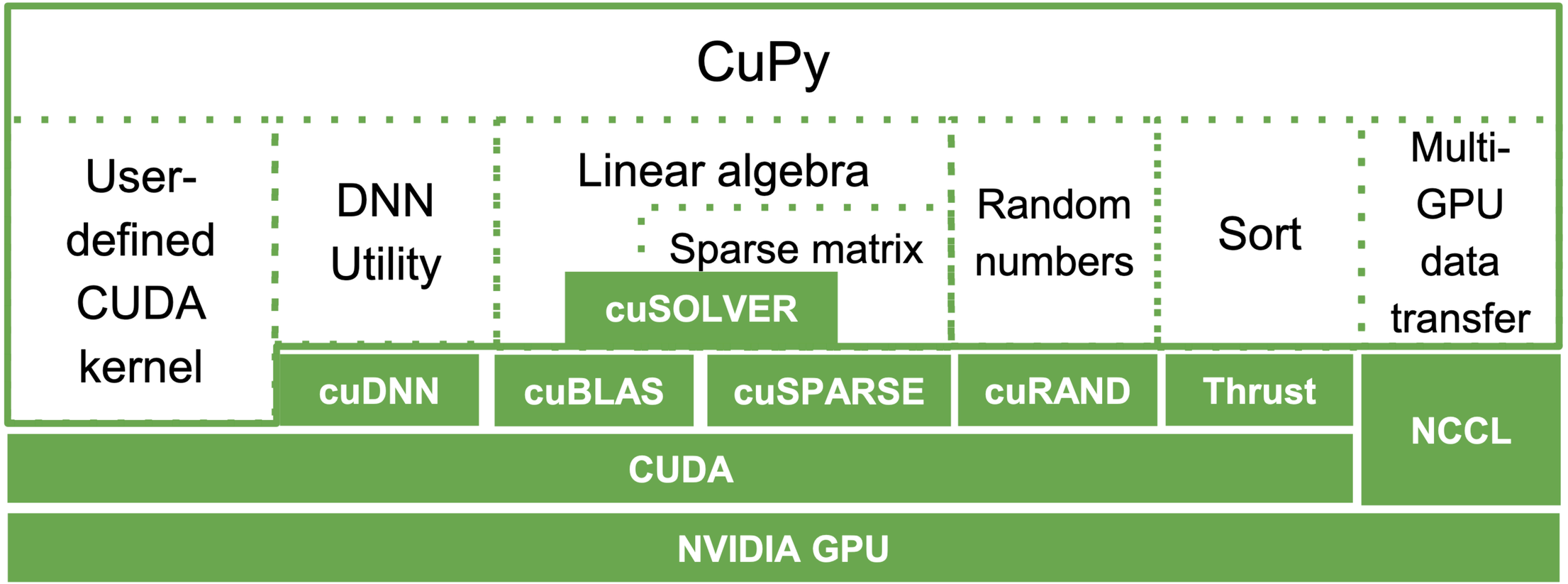
Imports:

Host <-> Device interaction:

Execute operation:

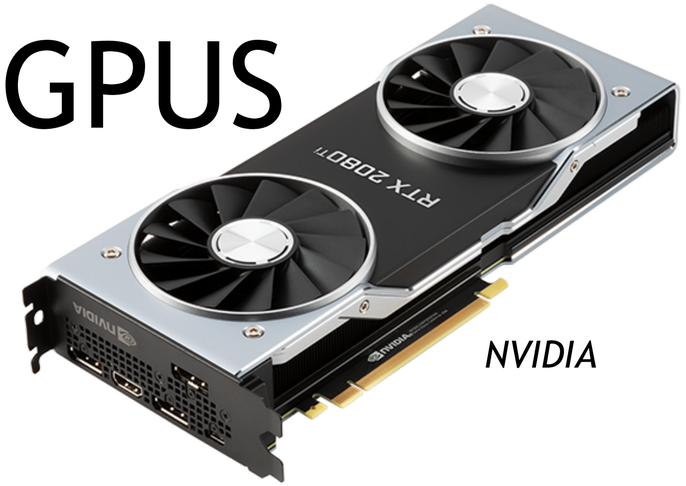
```
import cupy
import numpy as np

# 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: SOFTWARE ARCHITECTURE



WRAPPING UP

NEURAL NETWORKS AND GPUS



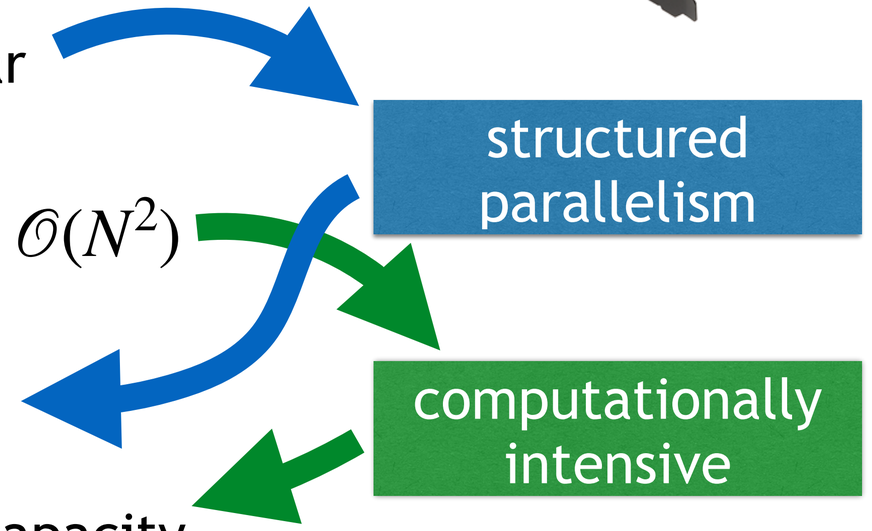
NVIDIA

Matrix multiplication $Y = W \cdot X$ is at the heart of neural networks

- N^2 elements - independent but very similar computations
- Compute scales with $\mathcal{O}(N^3)$, memory with $\mathcal{O}(N^2)$

GPUs are

- massively parallel vector-based processors
- excellent in compute, limited in memory capacity
- Bonus: highly energy efficient => many operations per Watt



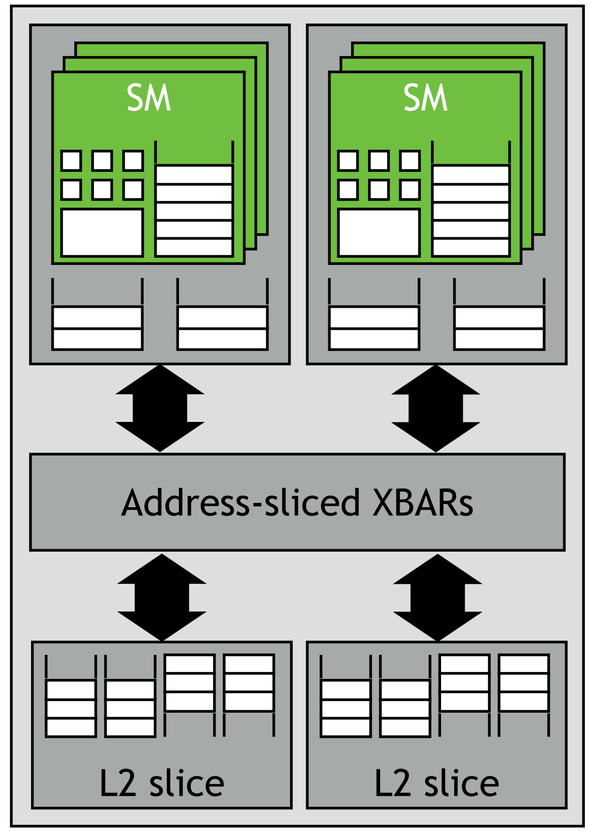
SUMMARY

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 highly depends on the use-case



DISCUSSION
EXERCISE 02

Groups presenting today:
Code: Group 6 (Daniela & Finn)
Plots/Observations: Group 5 (Vincent & Julius)

GENERAL REMARKS

LR observation across groups

Low LRs are slow, higher LRs fast & just as good

But: Our current task (MNIST) is pretty simple, be careful to generalise this

Two equivalent ways to compute sigmoid derivative:

$$\sigma(x) = \frac{1}{(1 + e^{-x})} \rightarrow \sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \sigma(x)(1 - \sigma(x))$$

Gradient normalisation

Gradient must be normalised to batch size, otherwise LR depends on batch size

Please use robin.janssen@ziti.uni-heidelberg.de (not @stud…) 🤷

THIRD (AND FINAL) EXERCISE

Weights and Biases logging (wandb.ai)

Port NumPy implementation to CuPy

- Code template: ex2 solution
- GPU access: HAWAII cluster (*how_to_cluster*
- Account credentials via email
- Experiment with different network sizes
- Compare CPU and GPU execution times

To be submitted via e-mail by:

- Wednesday 9:00

Discussion after project examples

https://hawaii.ziti.uni-heidelberg.de/teaching/ap_nn_from_scratch_materials_wise2025/