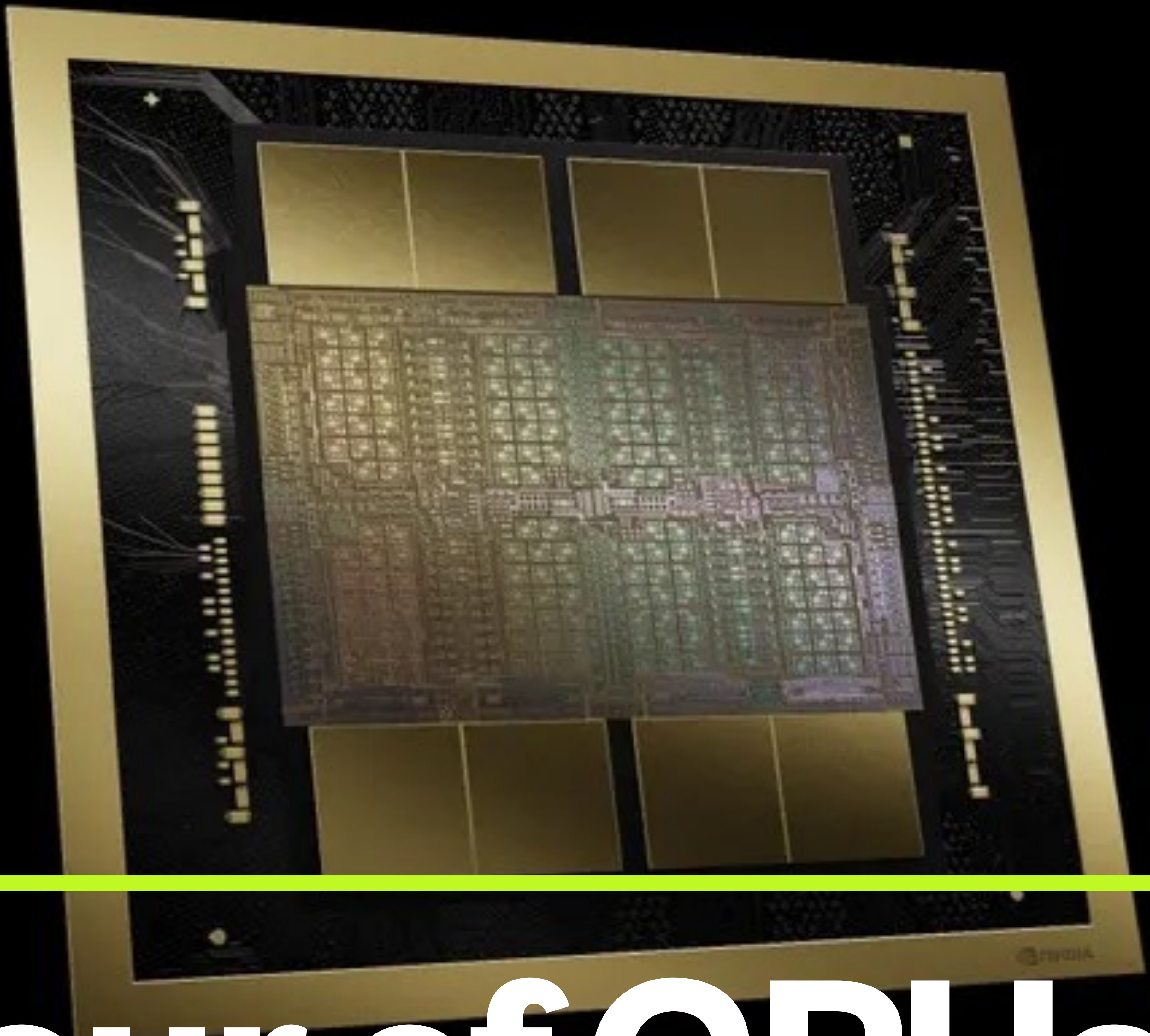
GPUs in one/two lectures



A Brief Tour of GPUs

TIM ROGERS FEB 9, 2026

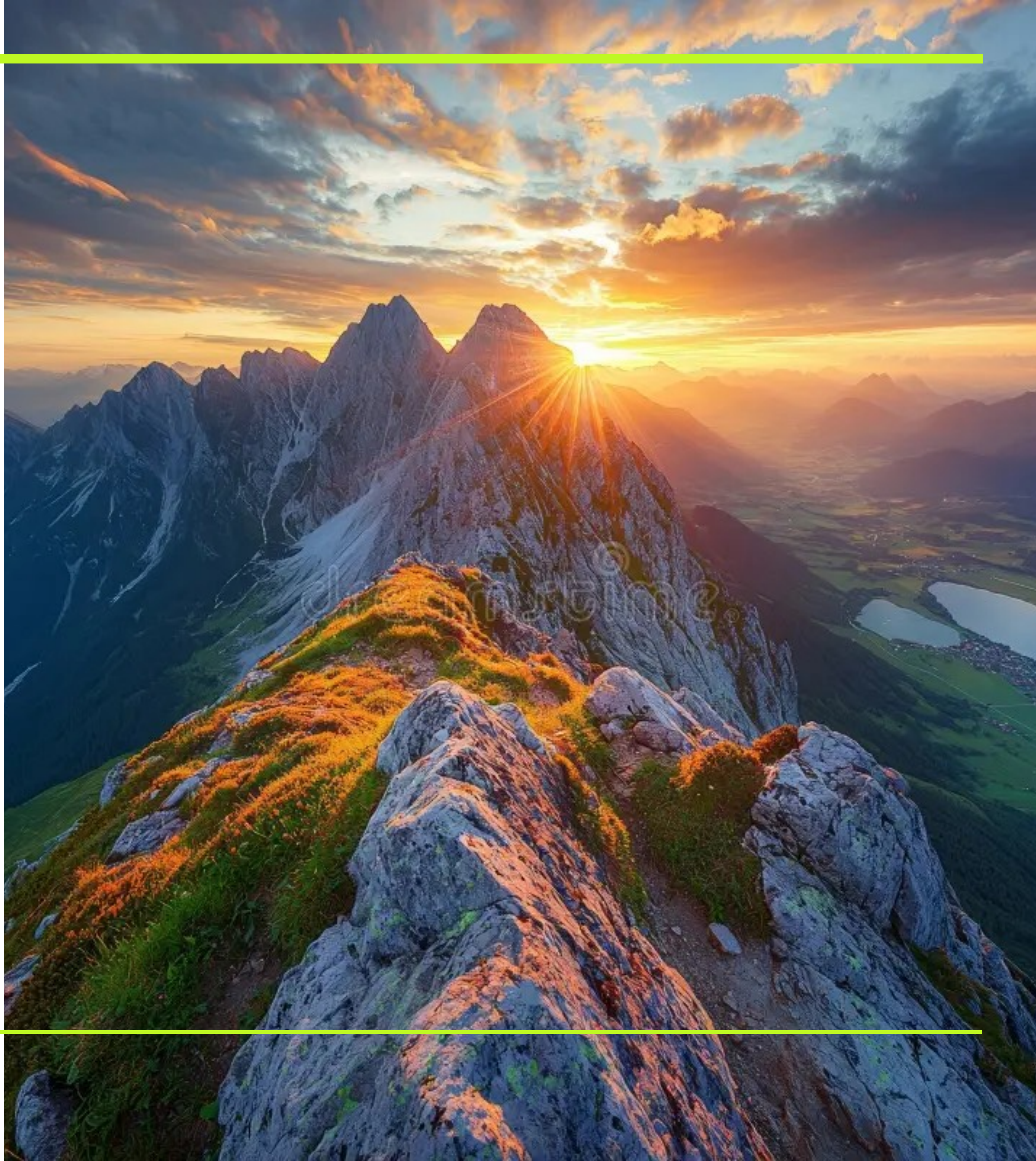
About me

TIM ROGERS FEB 13, 2025

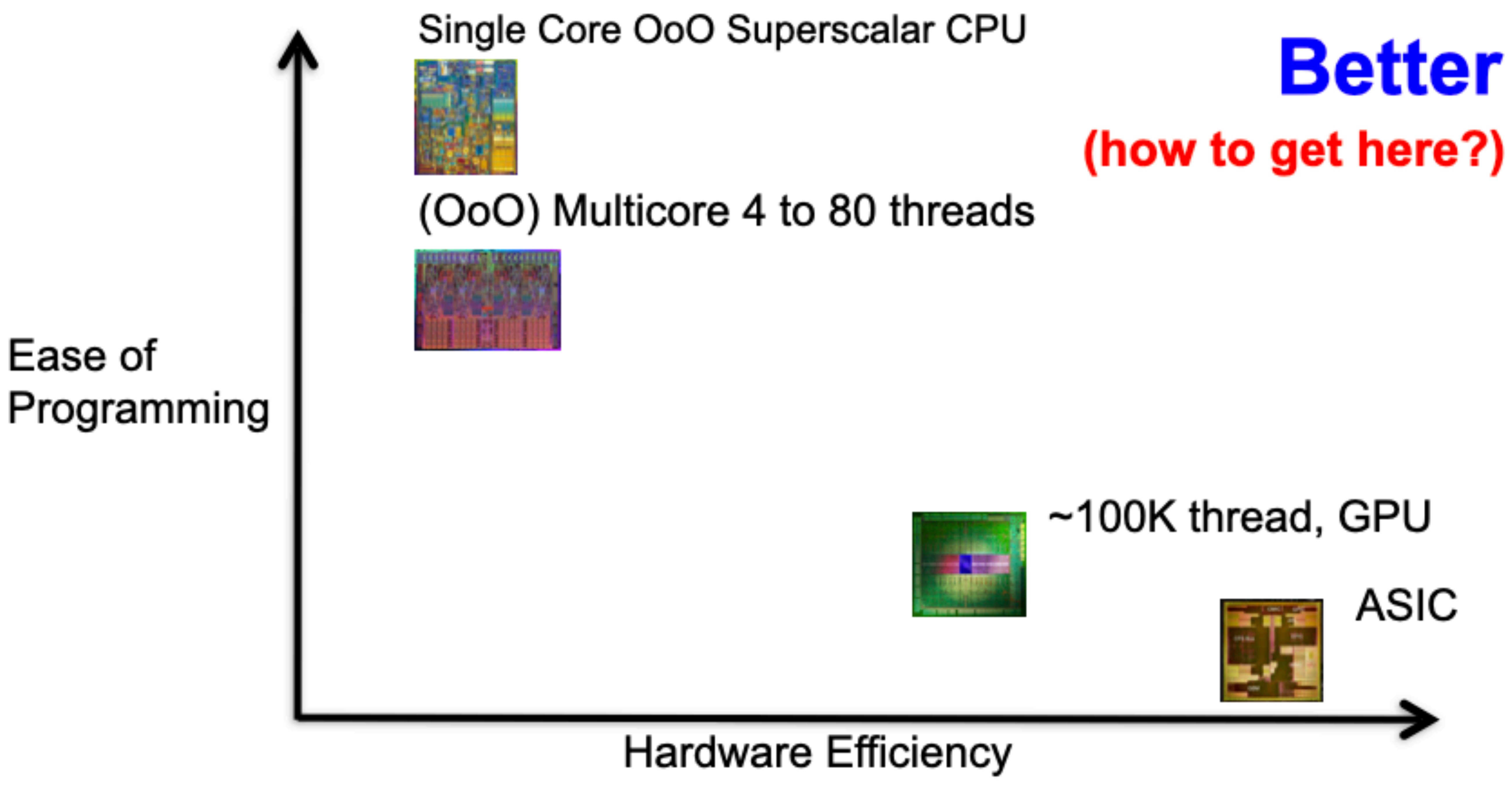
- Associate Professor
  - Started at Purdue in January 2016
- PhD in Computer Architecture from the University of British Columbia (Canada)
  - B.Eng in EE from McGill (also Canada)
  - ….Yes…. I am Canadian
- Research in Programmable Accelerators (aka GPUs)
- Interned at NVIDIA Research and AMD Research as a PhD student

- Between undergrad and grad school – spent 5 years at Electronic Arts as a software engineer.
- On eave at NVIDIA for 2024-2025
  Maintain contracting relationship with them

Part 1: The Motivation



TIM ROGERS FEB 9, 2026



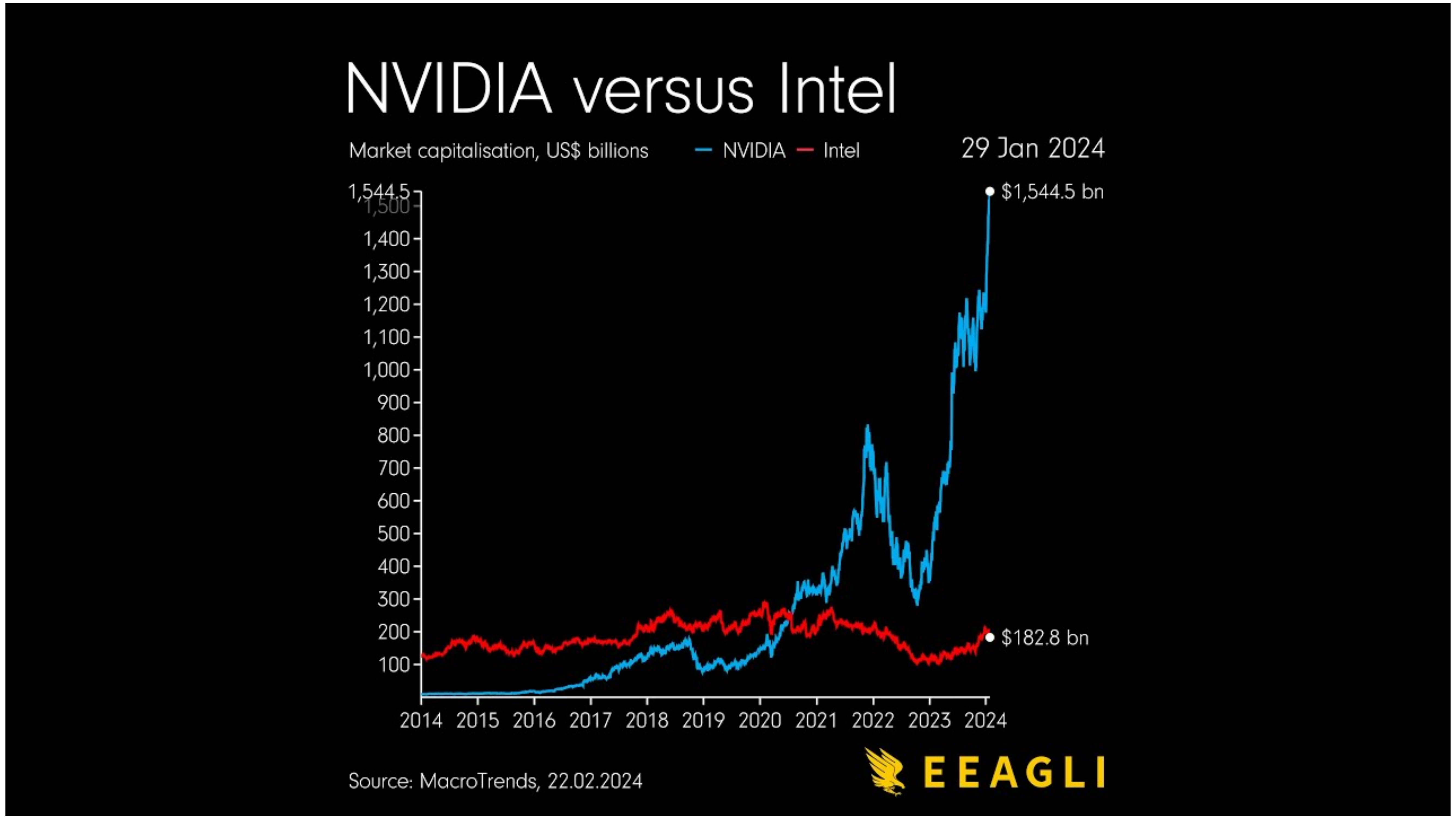
A Fundament Tradeoff

Programmability vs efficiency

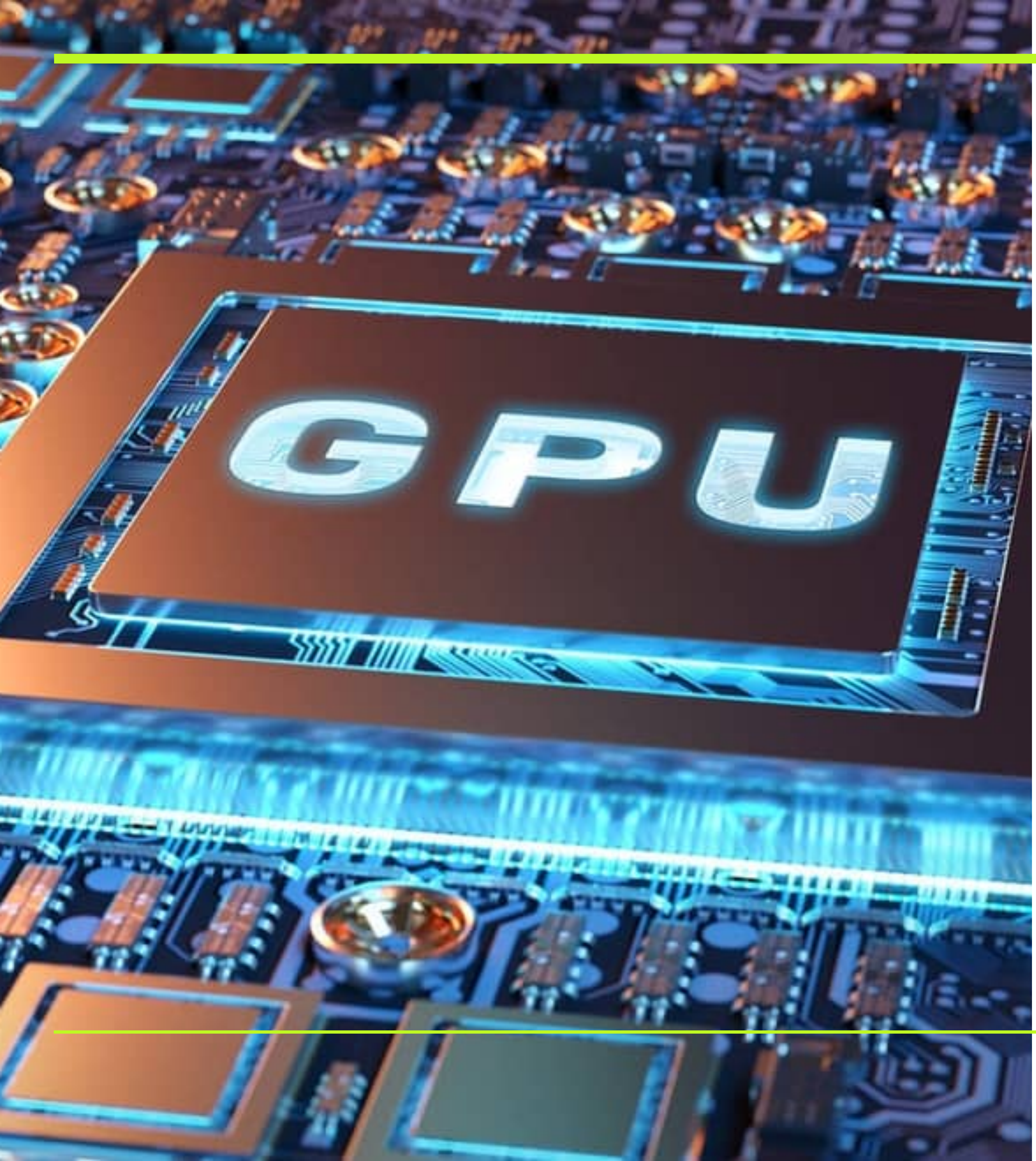

Generality
Efficiency
Tim Rogers

Why is this important now?

Intel as a proxy for general purpose



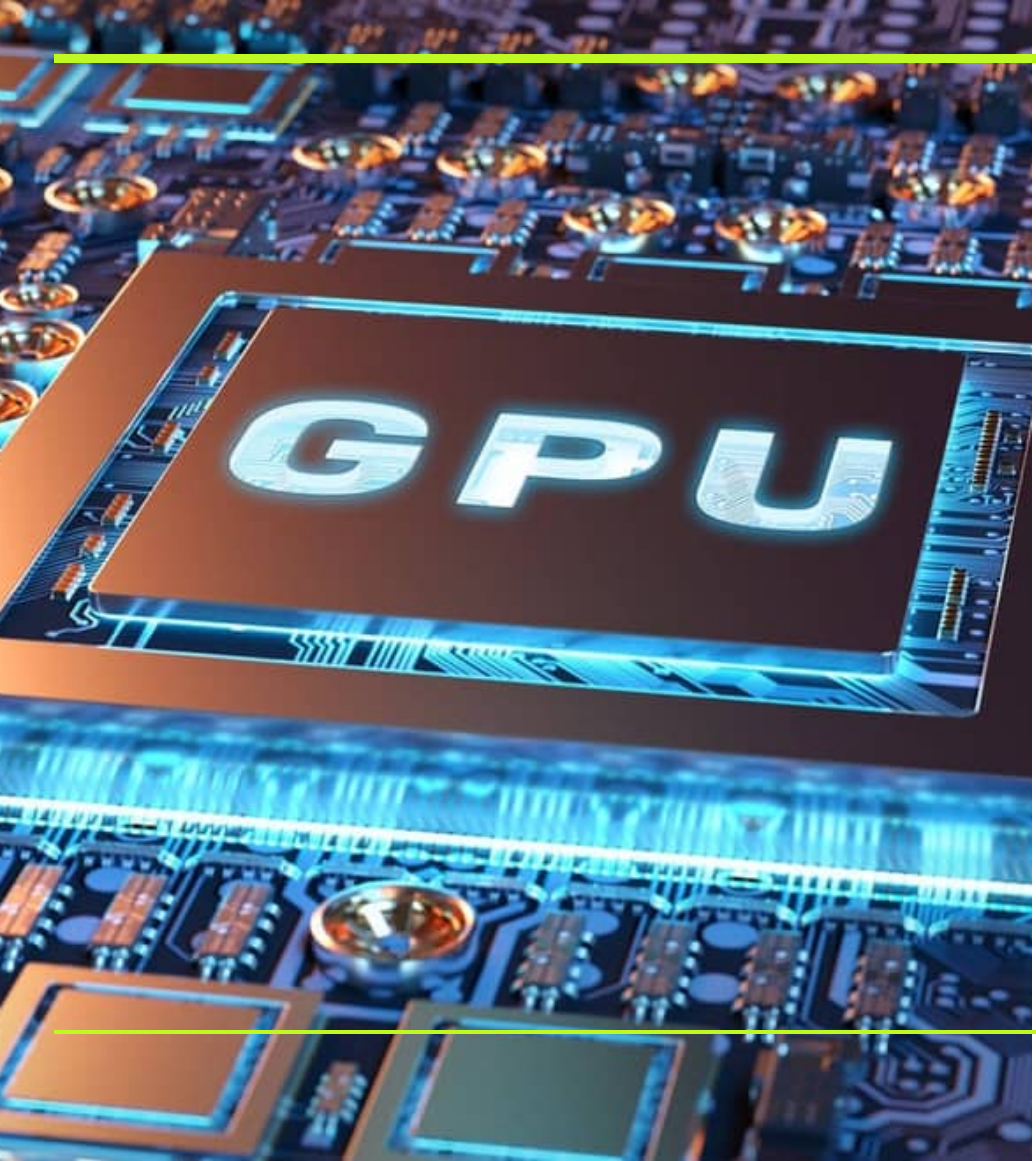
- MOS compute frequency has reached its limits
- Instruction Level Parallelism (ILP) is mostly mined out
- Branch predictors, caches, and memory dependency prediction have done great things
- However, these are energy-hungry operations
    - We are limited by power/energy in high-performance designs



What is a GPU?

Today: A Programmable Accelerator

- Hardware acceleration has been around forever
  - video/image encode/decode
  - network acceleration
  - cryptography
  - bitcoin mining
- However, they are not generally ***programmable.***
  - *i.e., not Turing Complete*
- **GPUs are programmable**



What is a GPU?

Today: **THE** Programmable Accelerator

- Hardware acceleration has been around forever
  - video/image encode/decode
  - network acceleration
  - cryptography
  - bitcoin mining
- However, they are not generally ***programmable.***
  - *i.e., not Turing Complete*
- **GPUs are programmable**

Part 2: The History

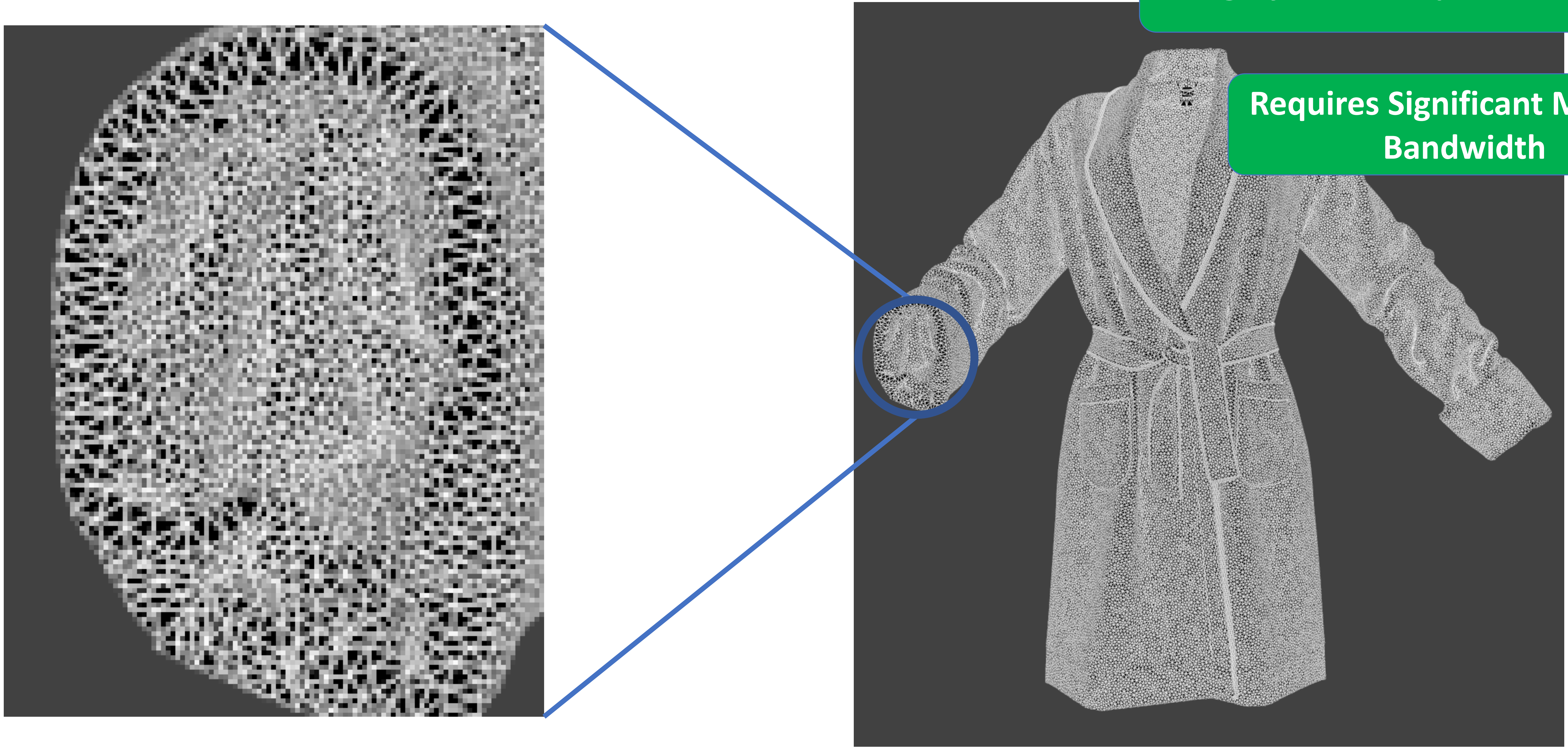
TIM ROGERS FEB 9, 2026

What was a GPU?

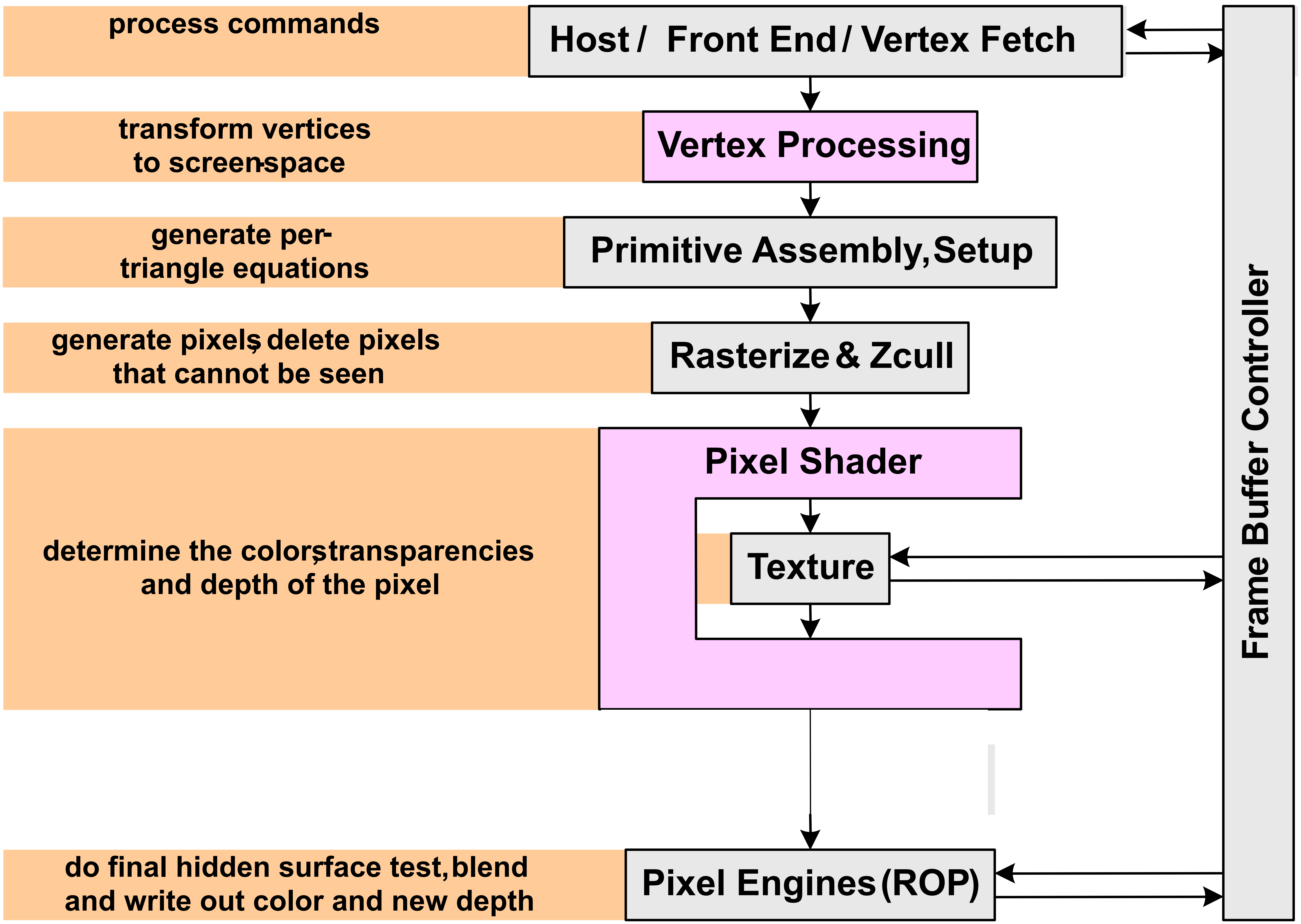
- GPU = Graphics Processing Unit
  - Accelerator for raster based graphics (OpenGL, DirectX)
  - Highly programmable
  - Commodity hardware
  - 100’s of ALUs; 10’s of 1000s of concurrent threads

Today the name GPU is not really meaningful. In reality they are highly parallel, highly programmable vector supercomputers.

Modern GPUs: Good at drawing triangles



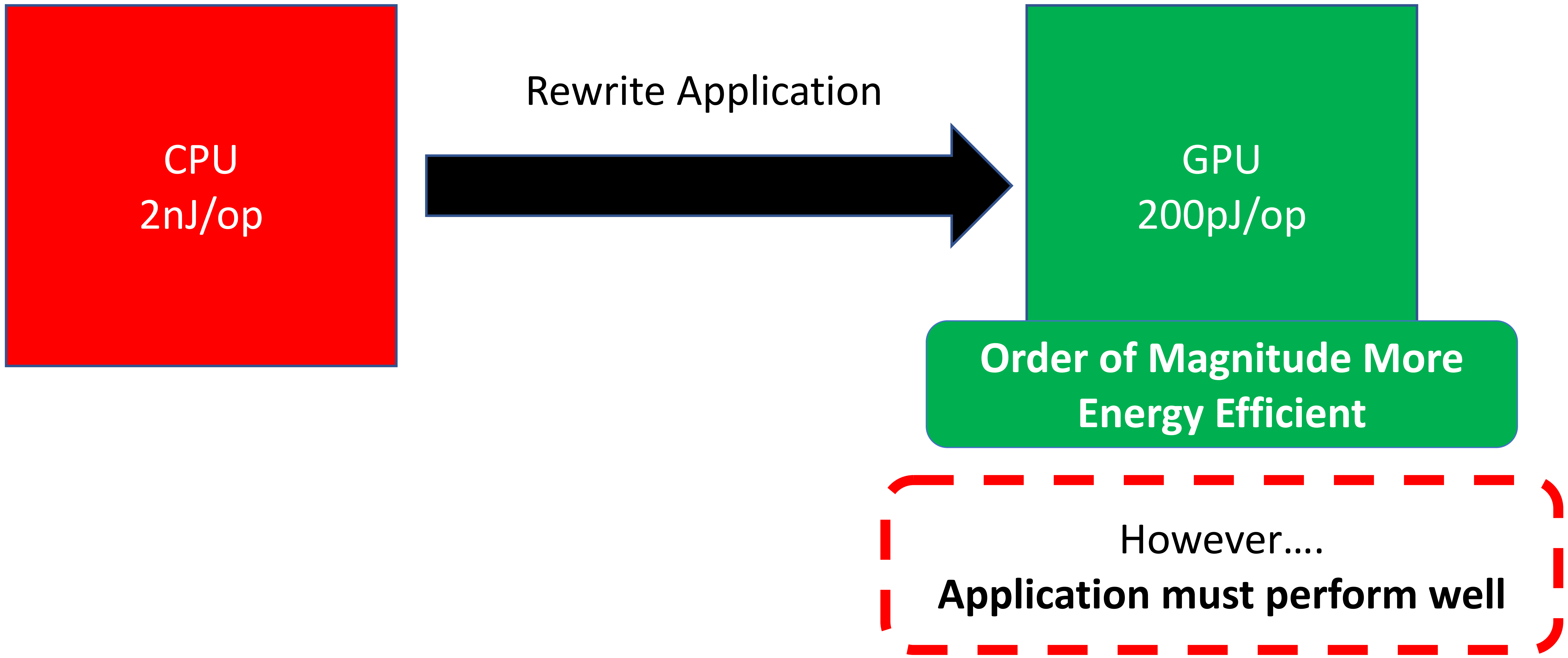
GPU: The Life of a Triangle



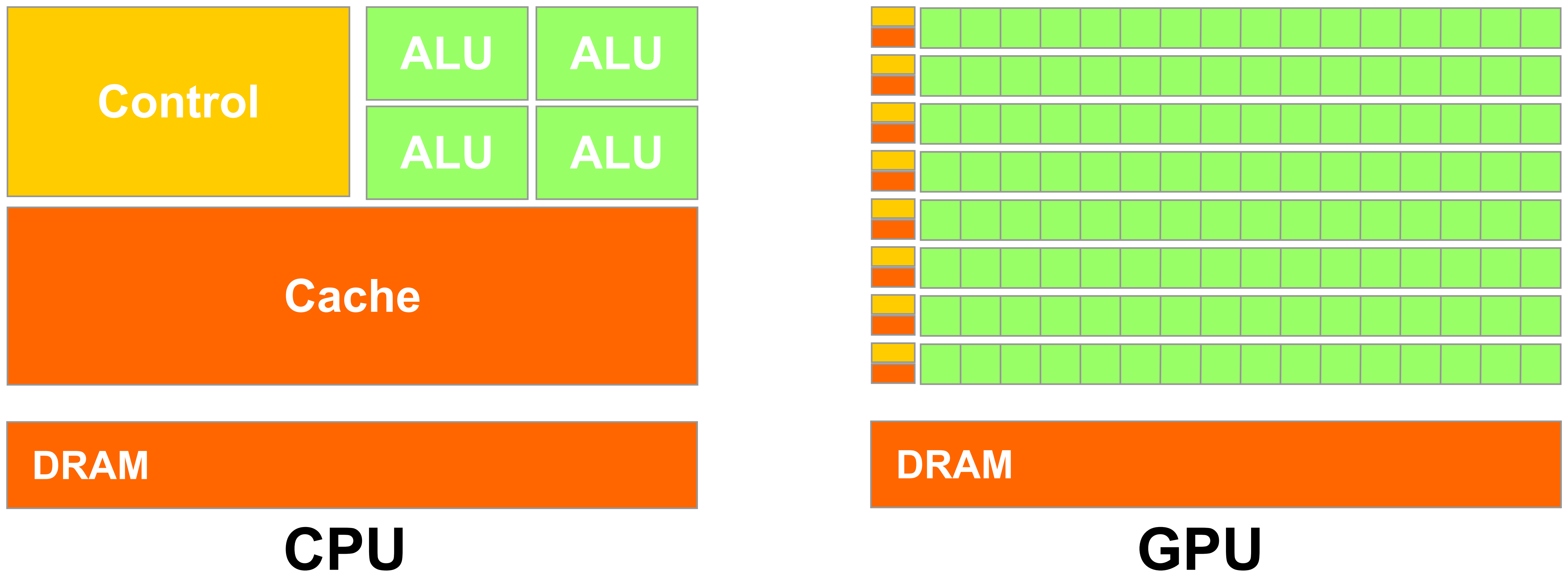
[David Kirk / Wen-mei Hwu]

Why use a GPU for computing?

- GPU uses larger fraction of silicon for computation than CPU.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.



GPU uses larger fraction of silicon for computation than CPU



[NVIDIA]

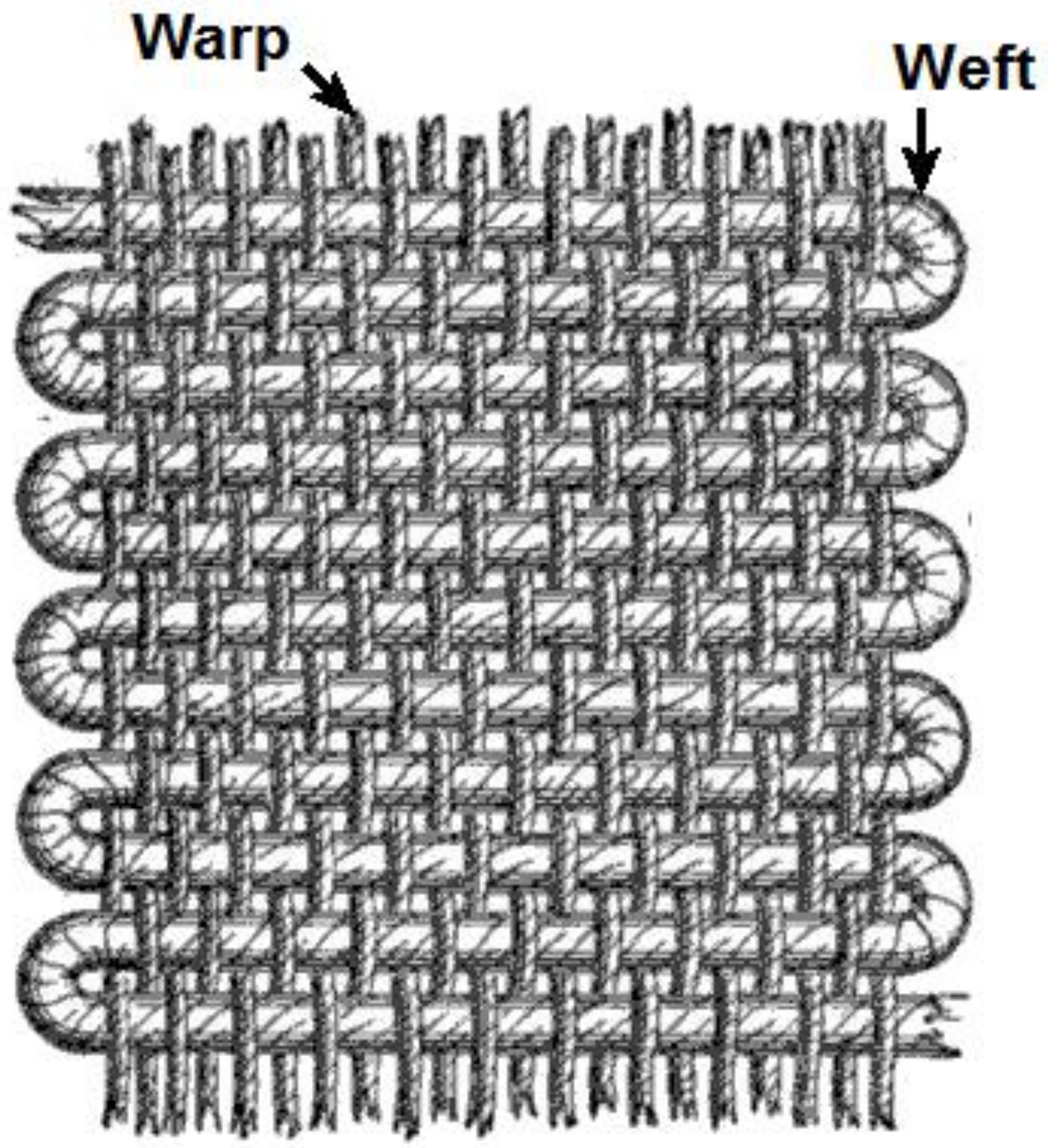
GPGPUs vs. Vector Processors

- Similarities at hardware level between GPU and vector processors.

- SIMT programming model moves hardest parallelism detection problem from compiler to programmer.

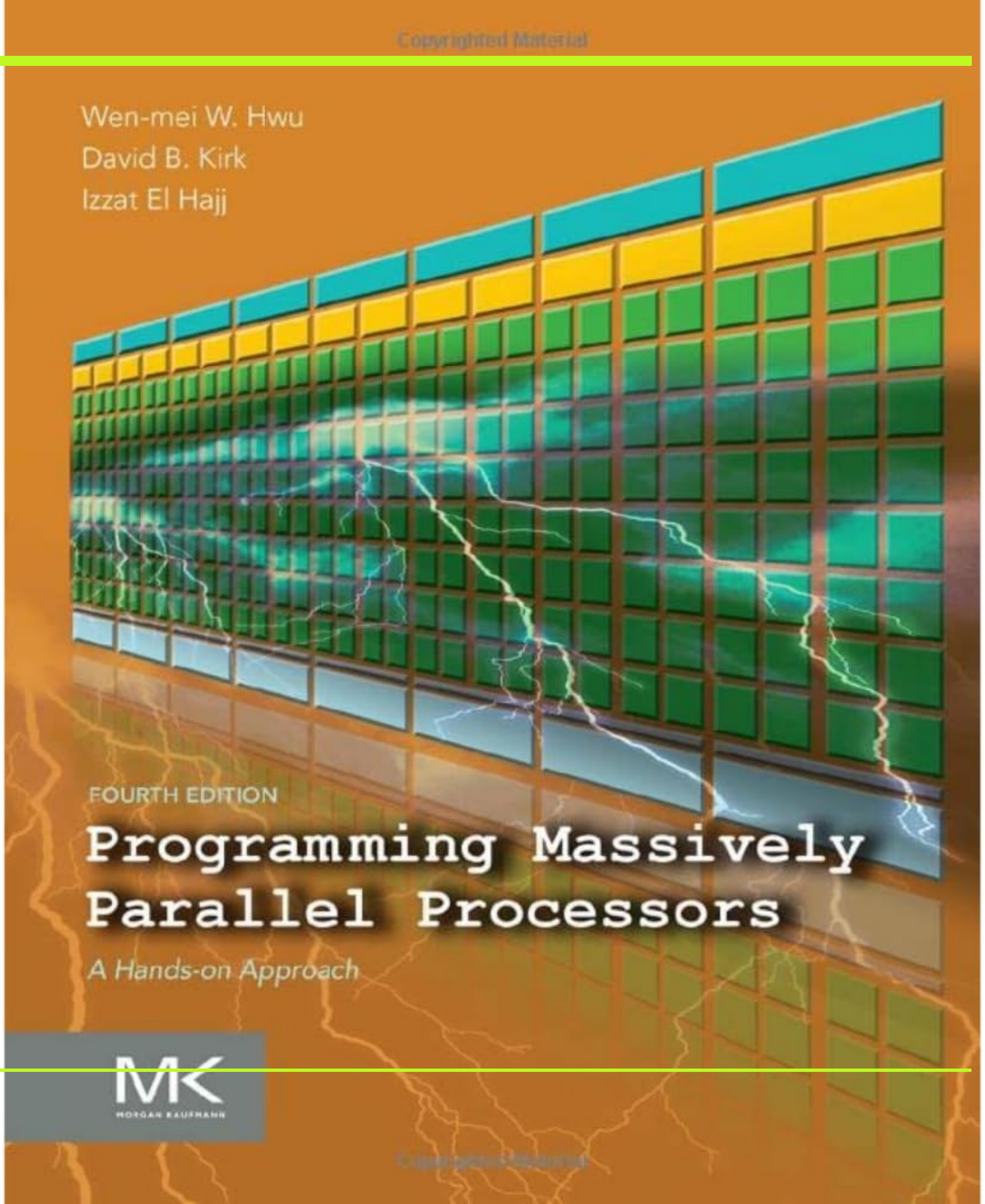
Single Instruction Multiple Thread (SIMT) Execution Model

- Programmers sees MIMD threads (scalar)
- GPU bundles threads into warps (wavefronts) and runs them in lockstep on SIMD hardware
- An NVIDIA warp groups 32 consecutive threads together (AMD wavefronts group 64 threads together)
- Aside: Why "Warp"? In the textile industry, the term "warp" refers to "the threads stretched lengthwise in a loom to be crossed by the weft" [Oxford Dictionary].



[https://en.wikipedia.org/wiki/Warp_and_woof]

Part 3: The Programming



TIM ROGERS FEB 9, 2026

Parallelism in general

Many different definitions/types of parallelism

- Instruction Level Parallelism
  - Different machine instructions in the same thread can execute in parallel
- Task Level Parallelism
  - Higher level tasks can run concurrently
- Bit level Parallelism
  - In VHDL exploit the ability to do level bit-level computation in parallel (i.e. longer words, carry-lookahead adders)
- Data Level Parallelism
  - Identical computation just on different data
  - Single Instruction Multiple Data (SIMD) instructions exploit data parallelism
  - Single Program Multiple Data (SPMD) applications exploit data parallelism

GPUs are designed to exploit DLP

Remember: Can’t get around Ahmadl's Law

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

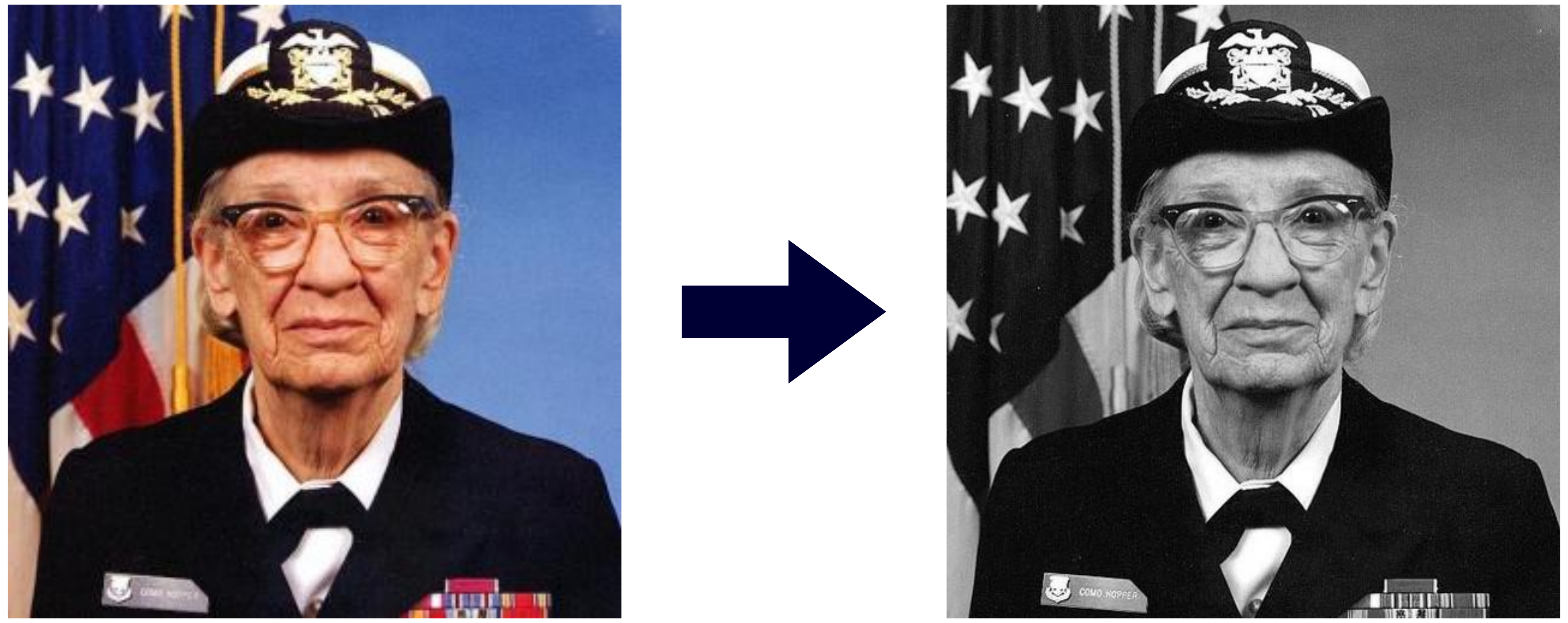
- There will always be some serial work that needs to be done
- CPUs are much better designed to handle serial work
- CPU and a parallel accelerator will almost certainly always work together.
  - OoO, superscalar CPU = Serial Accelerator
  - GPU = Parallel Accelerator

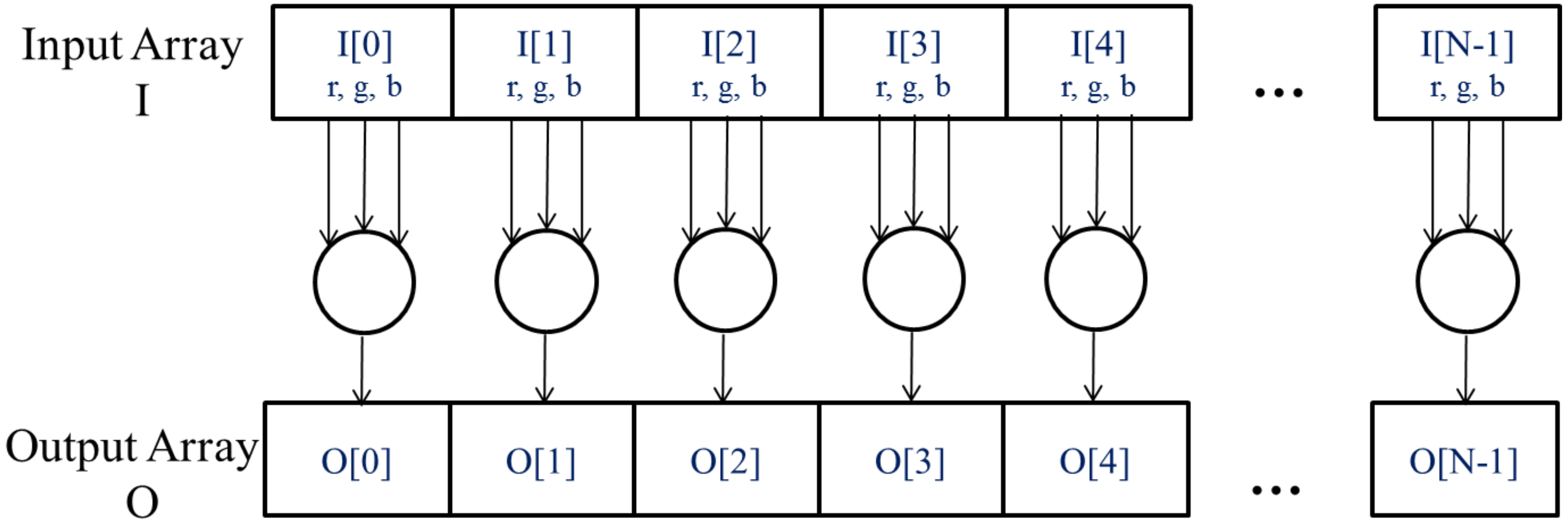
Bottom line:
Without parallelism in the program,
GPUs are useless

Example Application: Conversion to grey-scale

- Every pixel has 3 values to determine the color (R,G,B)
- Compute the **Luminance** value of the pixel
  - Embarrassingly **data-parallel** operation
  - Same operation on every pixel, all independent
  - Parallelism scales 1:1 with input data

Example of data parallelism

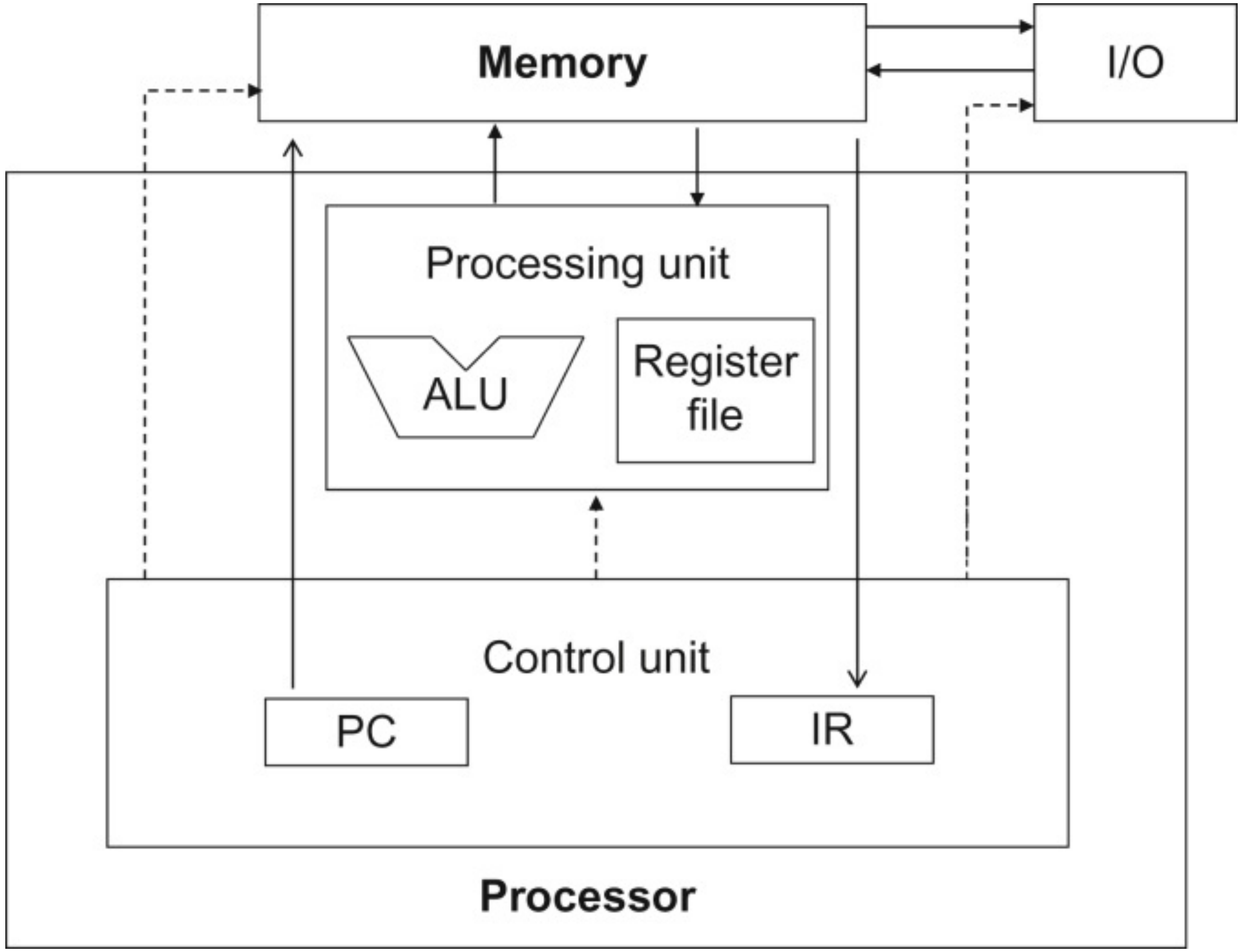




Why Data Parallelism?

- Easy to build efficient hardware to capture it
- The **regularity** in the computation can be exploited to reduce control hardware and make effective use of memory bandwidth

Conceptual model of a Von Neumann thread

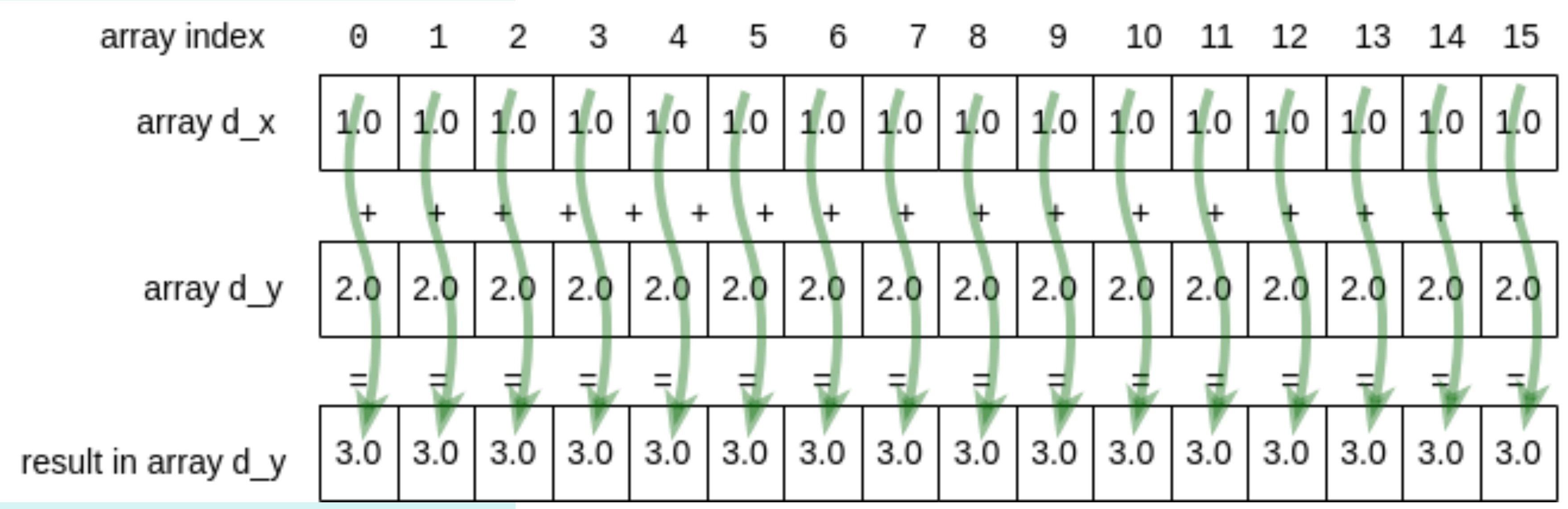


Conceptually you can think of a thread this way:
In reality – **the hardware does not actually look like this.**

The Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```




```
int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

The Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Host Code

```
int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

A little more on Kernel Launch

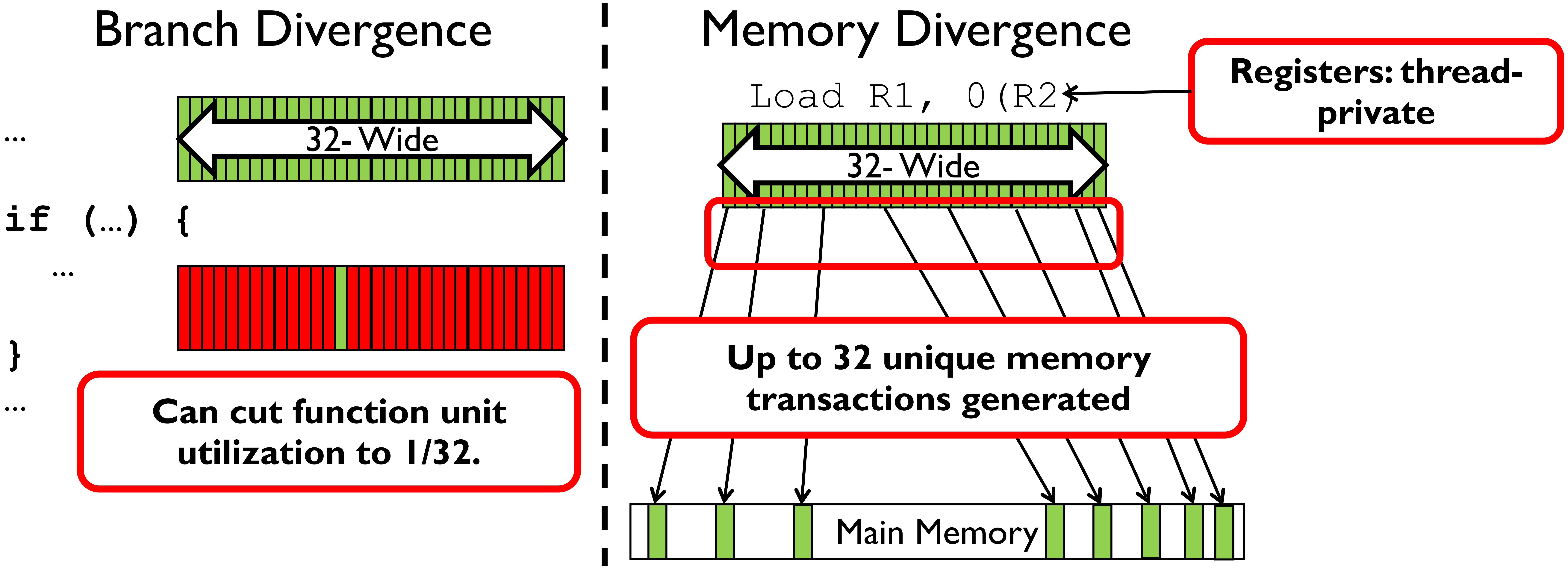
Host Code

```
int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  dim3 DimGrid(n/256, 1, 1);
  if (n%256) DimGrid.x++;
  dim3 DimBlock(256, 1, 1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

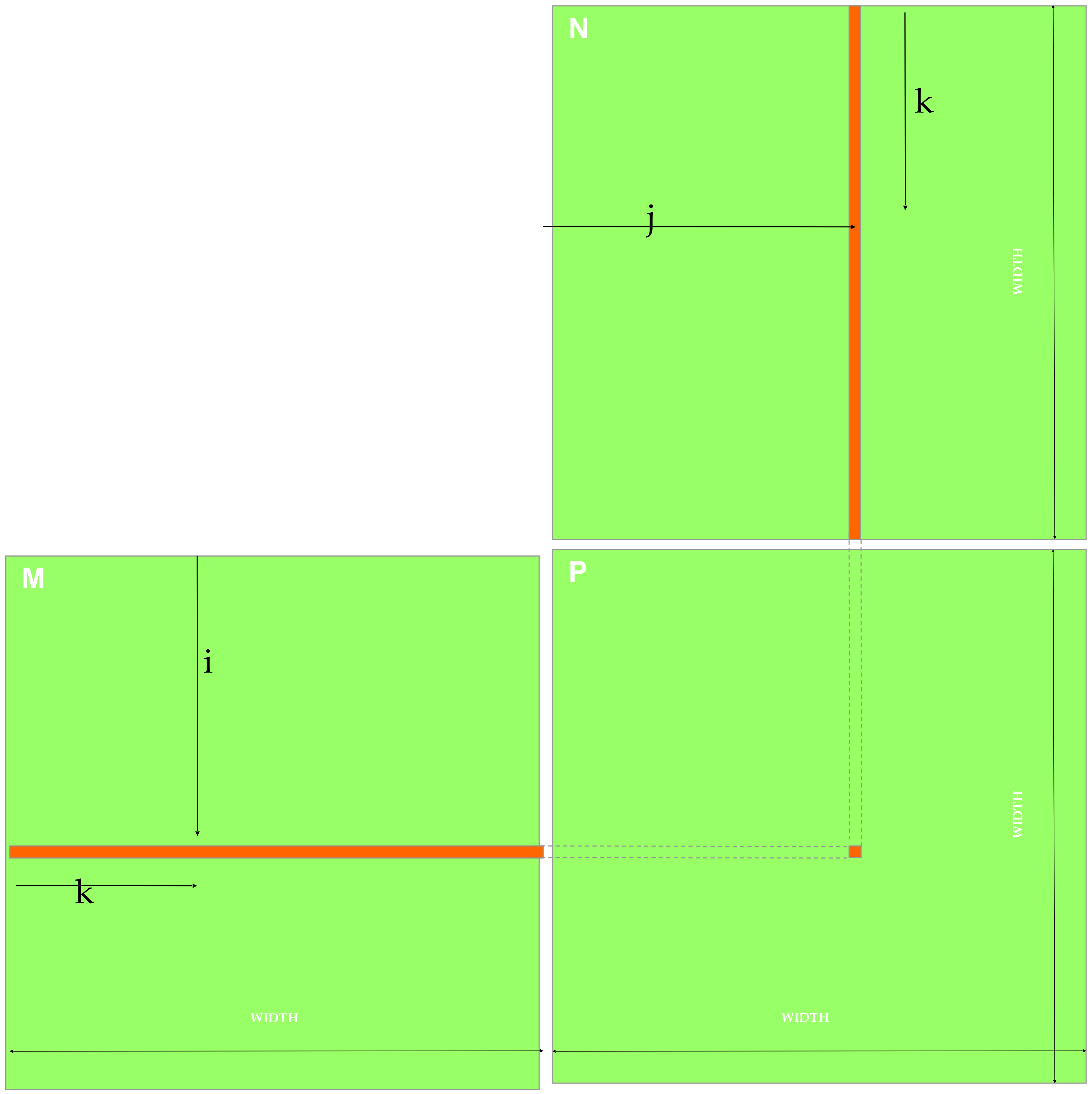
- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Divergence: Source of Inefficiency

- Regular hardware that amortizes front end and overhead
- Irregular software with many different control flow paths and less predicable memory accesses.



Matrix Multiplication
Where the money is made



// Matrix multiplication on the (CPU) host in single precision

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

Side-by-side kernel comparison

Tiled + shared memory

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__  float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__  float ds_N[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p)  {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __syncthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Simple MM

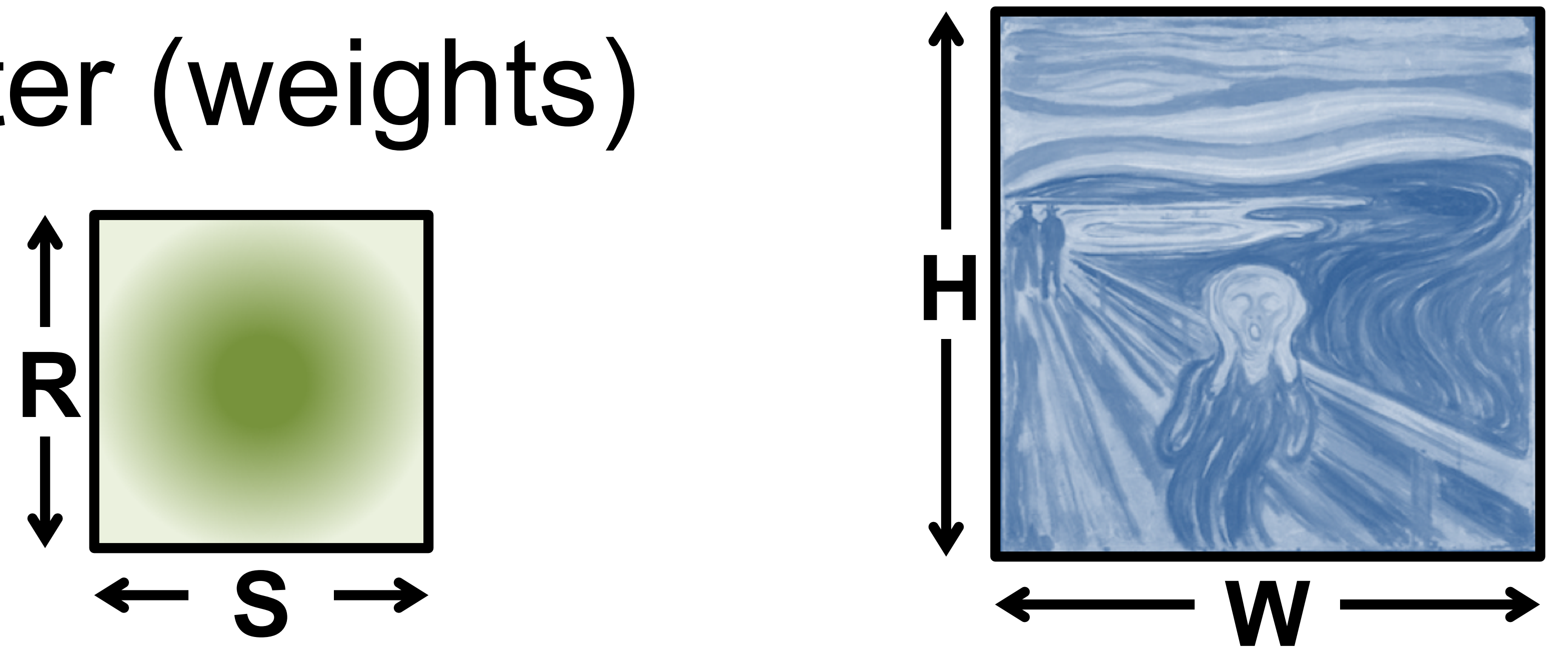
```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
int Row = blockIdx.y*blockDim.y+threadIdx.y;
int Col = blockIdx.x*blockDim.x+threadIdx.x;

 if ((Row < Width) && (Col < Width)) {
      float Pvalue = 0;
      for (int k = 0; k < Width; ++k)
    Pvalue += d_M[Row*Width+k] *  d_N[k*Width+Col];
   d_P[Row*Width+Col] = Pvalue;
 }
}
```

PRETTY MUCH EVERY USEFUL AI ALGORITHM YOU LEARN WILL TURN INTO GEMM + SOMETHING ON A GPU

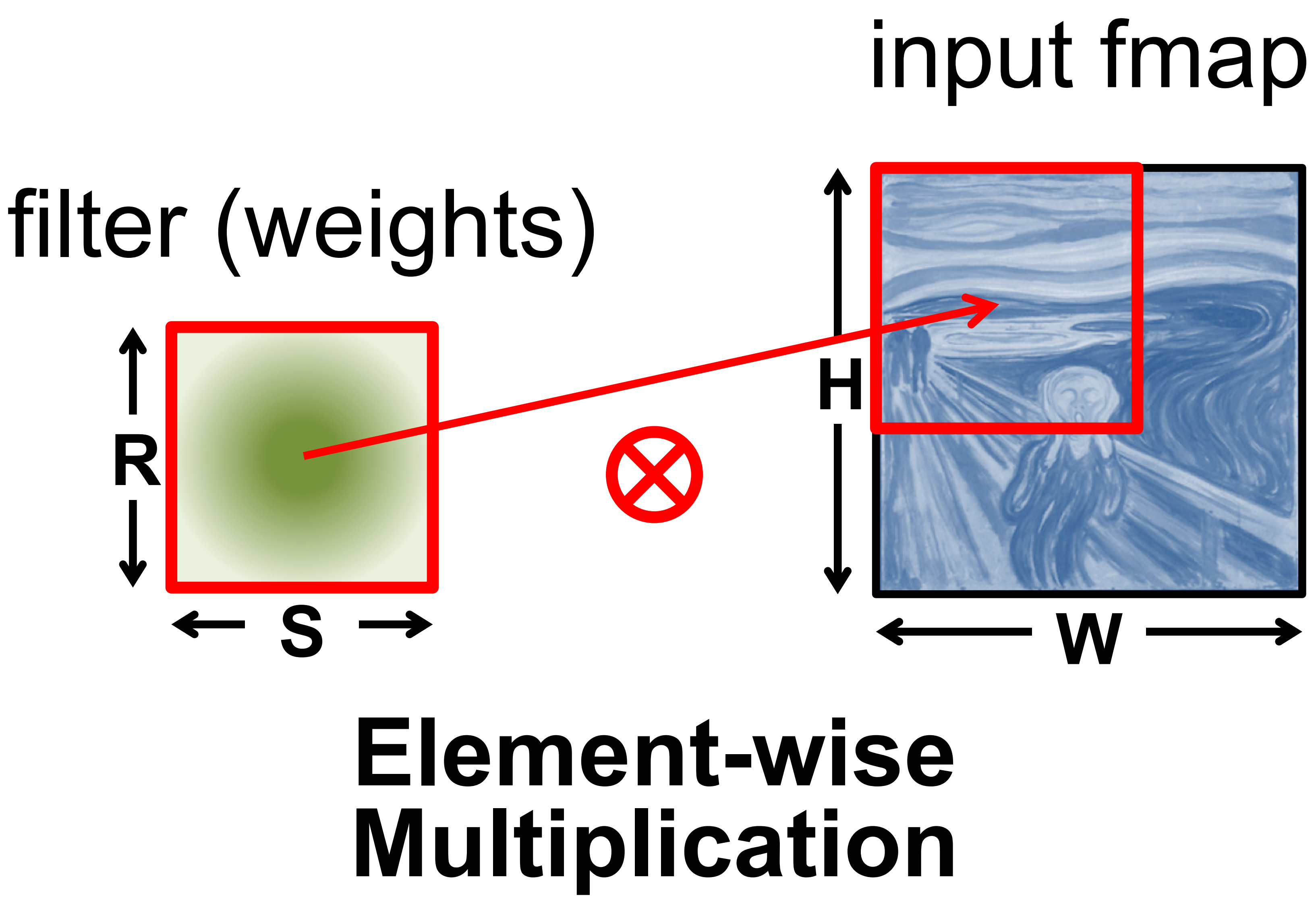
Convolution (CONV) Layer

a plane of input activations
a.k.a. **input feature map (fmap)**

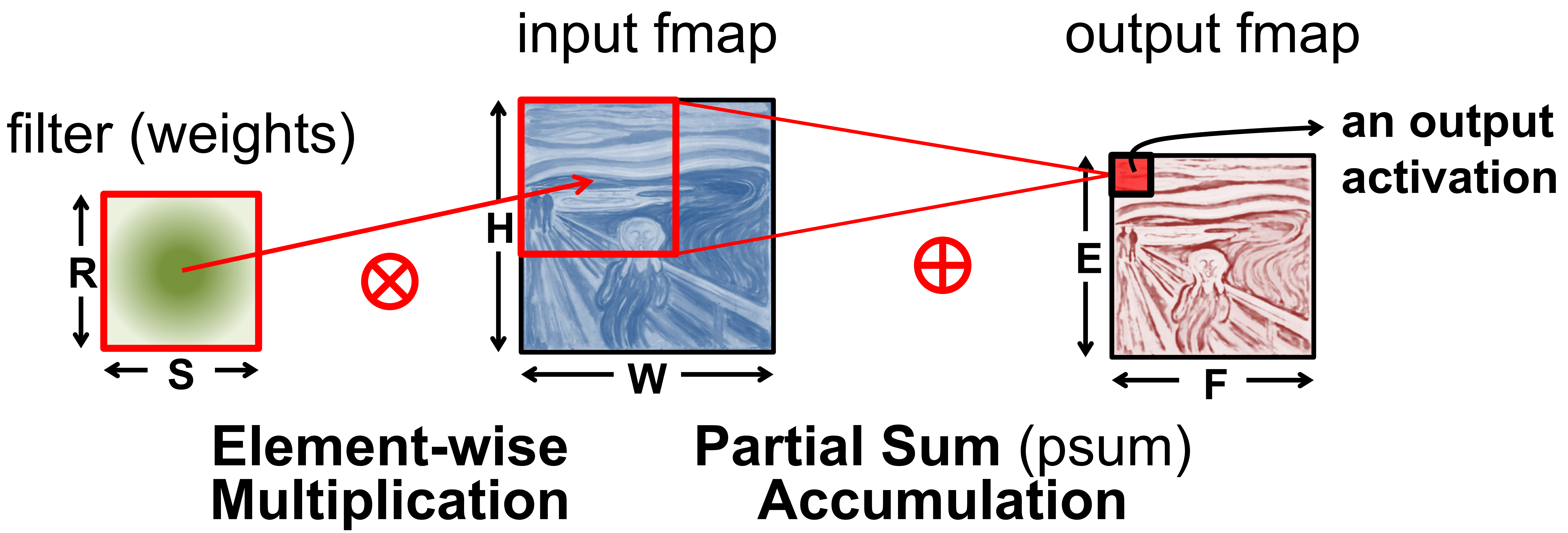
filter (weights)



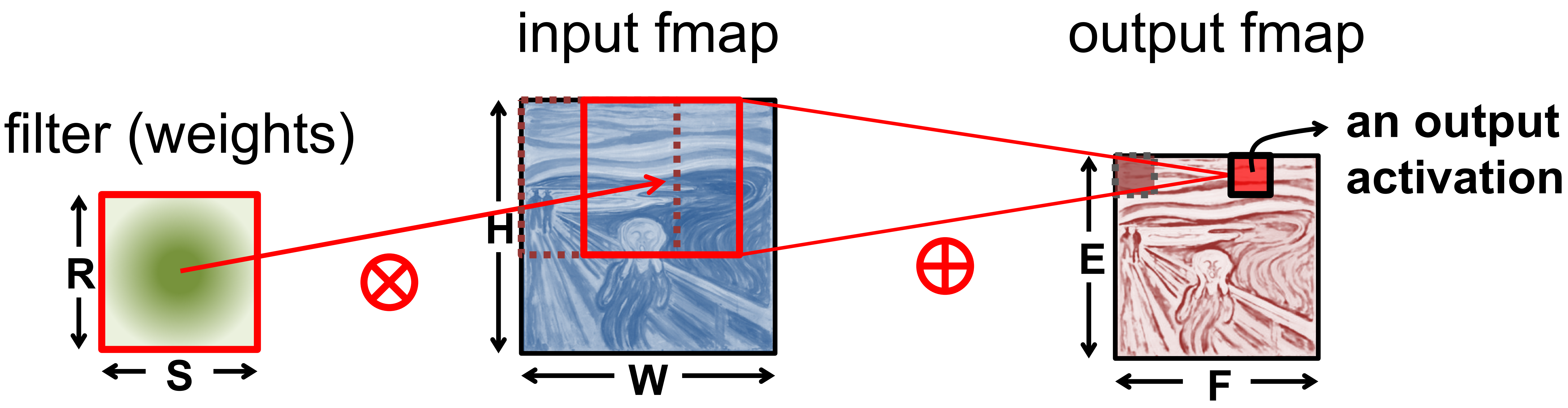
Convolution (CONV) Layer



Convolution (CONV) Layer

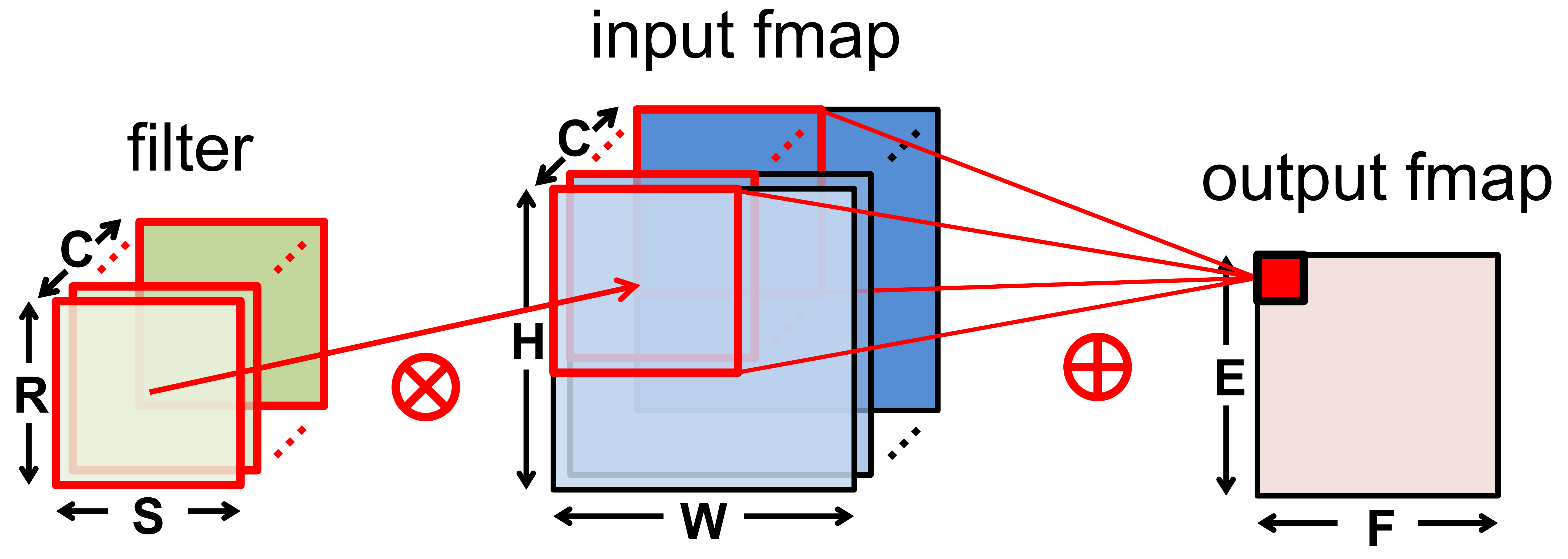


Convolution (CONV) Layer



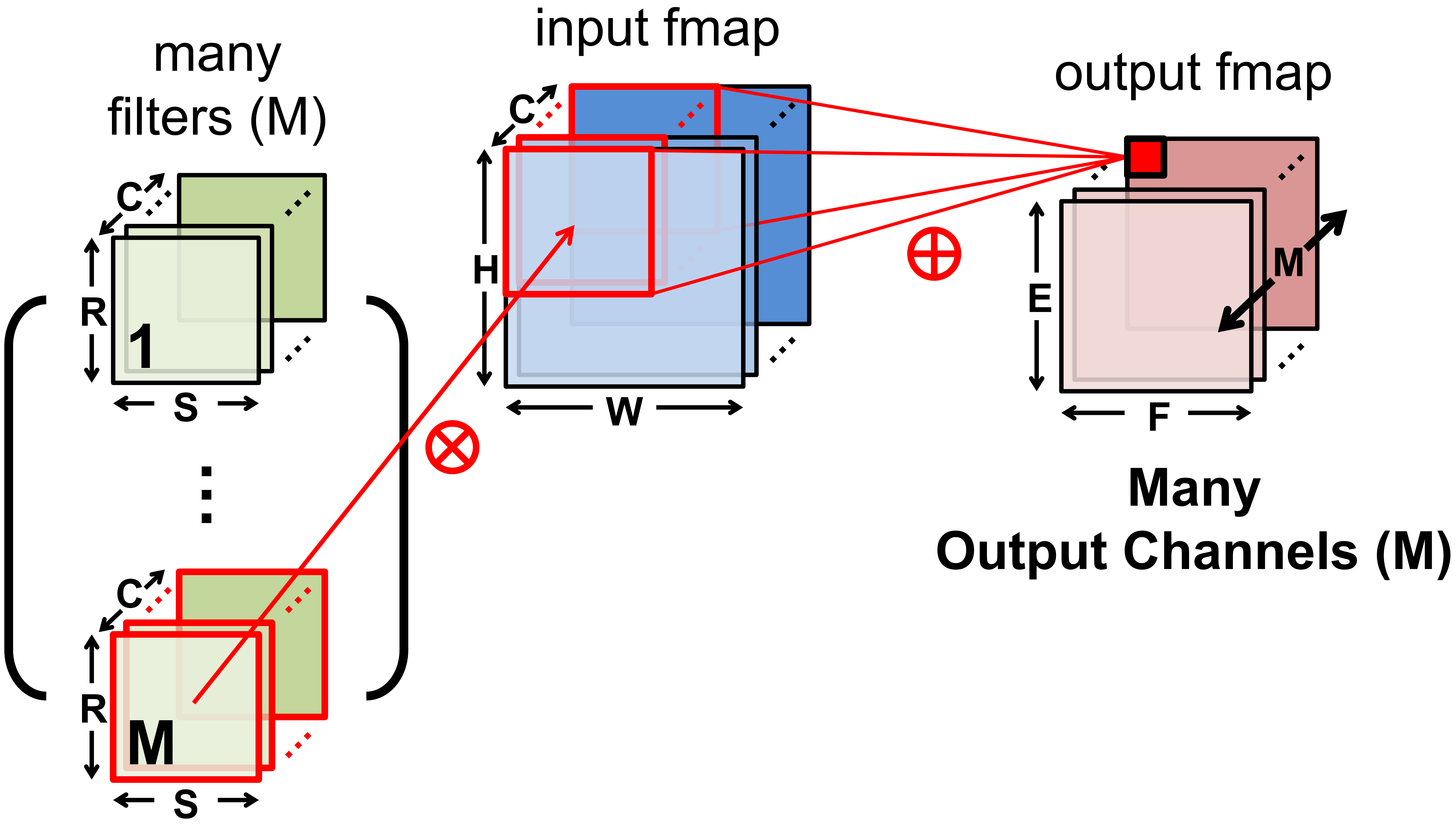
Sliding Window Processing

Convolution (CONV) Layer

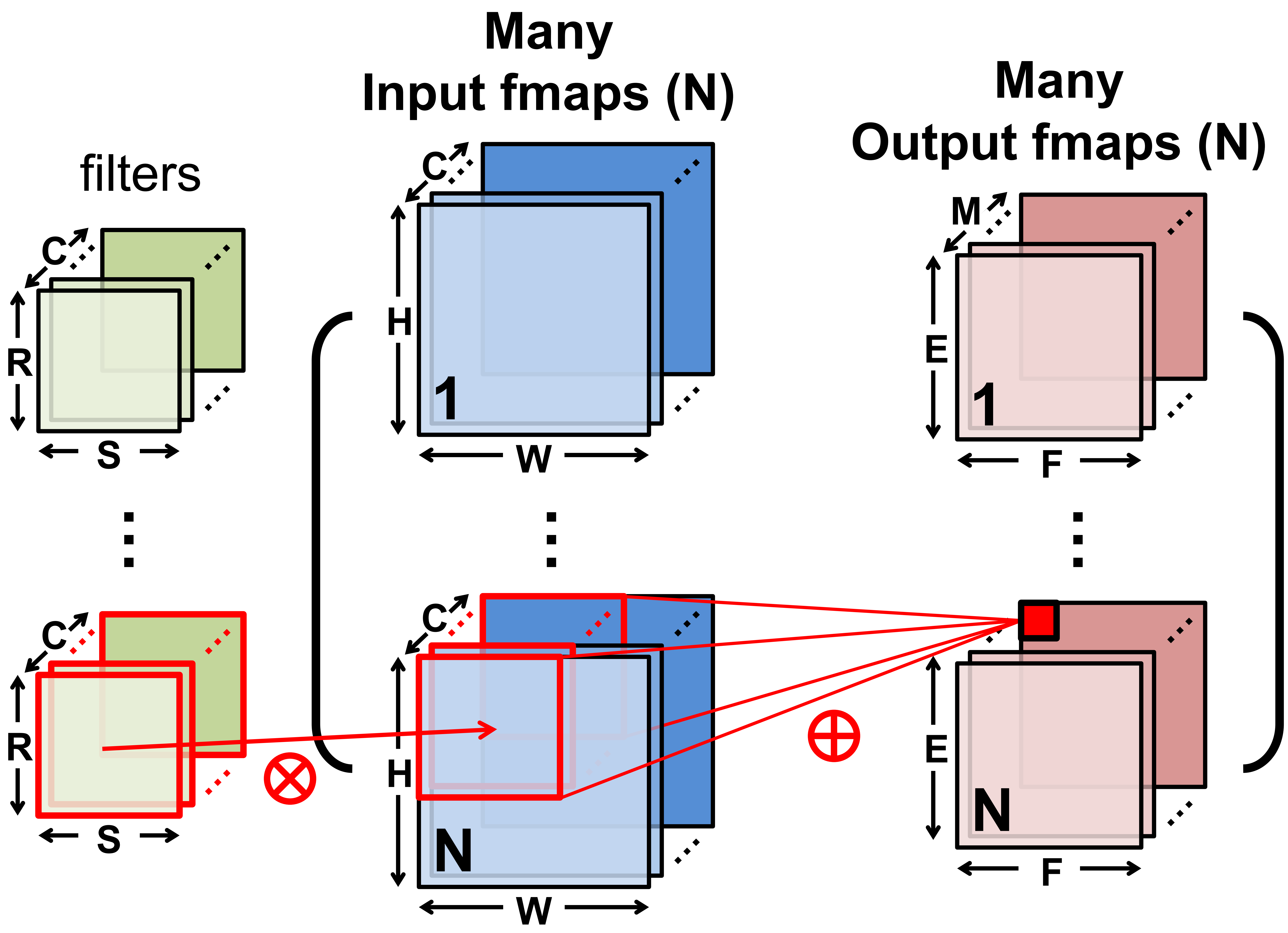


Many Input Channels (C)

Convolution (CONV) Layer



Convolution (CONV) Layer



CNN Decoder Ring

- N – Number of input fmaps/output fmaps (batch size)
- C – Number of 2-D input fmaps /filters (channels)
- H – Height of input fmap (activations)
- W – Width of input fmap (activations)
- R – Height of 2-D filter (weights)
- S – Width of 2-D filter (weights)
- M – Number of 2-D output fmaps (channels)
- E – Height of output fmap (activations)
- F – Width of output fmap (activations)

CONV Layer Tensor Computation

Output fmaps (O)

Biases (B)

Input fmaps (I)

Filter weights (W)

$$\underline{\mathbf{O}[n][m][x][y]} = \text{Activation}(\underline{\mathbf{B}[m]} + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} \underline{\mathbf{I}[n][k][Ux+i][Uy+j]} \times \underline{\mathbf{W}[m][k][i][j]}),$$

$$0 \le n < N, 0 \le m < M, 0 \le y < E, 0 \le x < F,$$

$$E = (H - R + U)/U, F = (W - S + U)/U.$$

Shape Parameter	Description
N	fmap batch size
M	# of filters / # of output fmap channels
C	# of input fmap/filter channels
H/W	input fmap height/width
R/S	filter height/width
E/F	output fmap height/width
U	convolution stride

CONV Layer Implementation

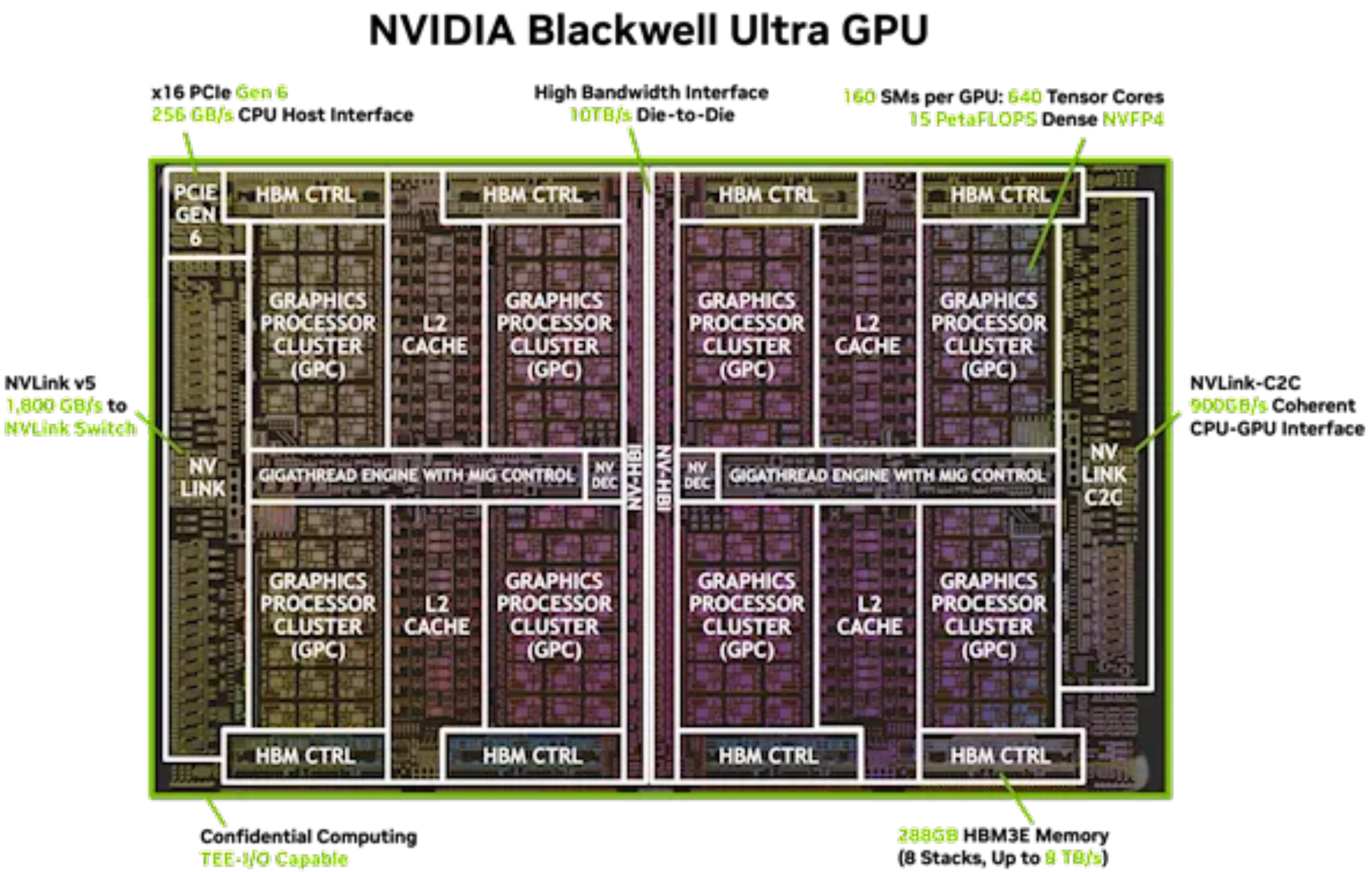
Naïve 7-layer for-loop implementation:

```
for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (x=0; x<F; x++) {
            for (y=0; y<E; y++) {

                O[n][m][x][y] = B[m];
                for (i=0; i<R; i++) {
                    for (j=0; j<S; j++) {
                        for (k=0; k<C; k++) {
                            O[n][m][x][y] += I[n][k][Ux+i][Uy+j] × W[m][k][i][j];
                        }
                    }
                }

                O[n][m][x][y] = Activation(O[n][m][x][y]);
            }
        }
    }
}
```

for each output fmap value

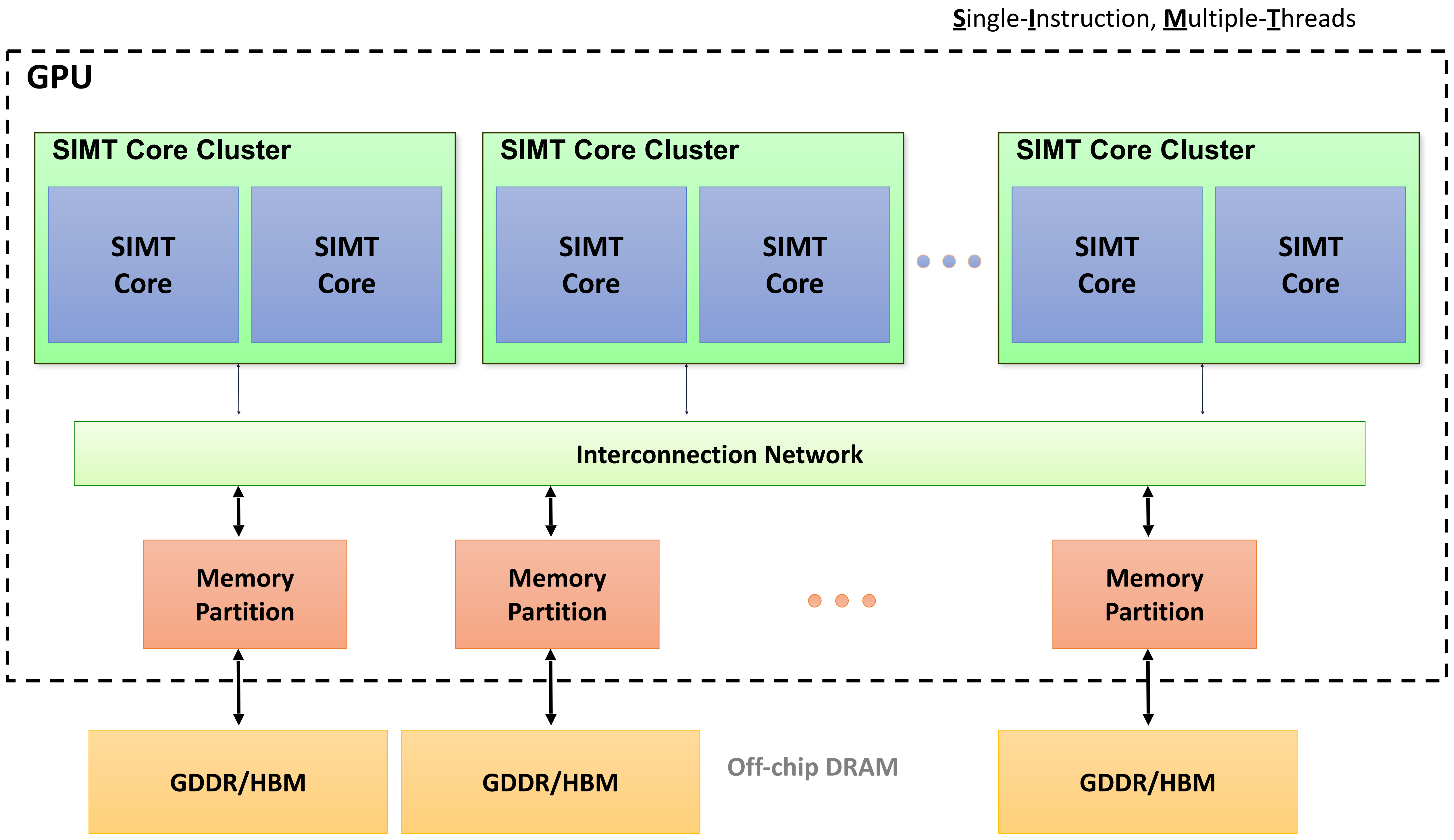
convolve a window and apply activation

Part 4: The Hardware

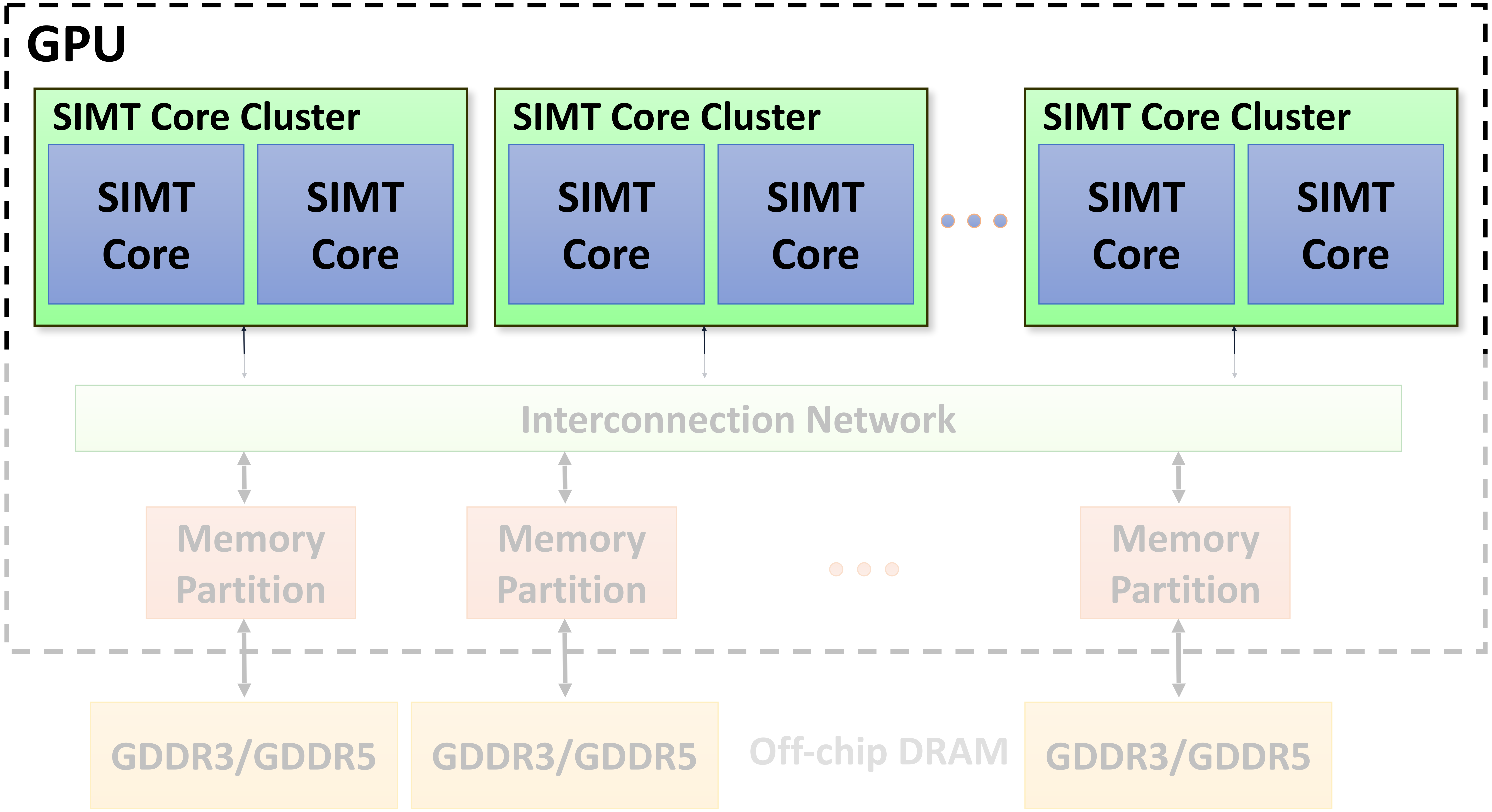


TIM ROGERS FEB 9, 2026

Part 4: The Hardware



GPU Microarchitecture Overview



GPU pipelines -- HOW

- Wide and deep unlike only deep traditional vectors
  - Eg 32 deeply-pipelined ALUs (32-deep multiplier)
  - use VLSI transistors (early Cray Vector machines predate CMOS)
  - Wider → slower clock → more energy-efficient than only deeper
- Super-simple pipelines (energy-efficient)
  - 32 “lanes” in lock-step
  - Each lane has Regrd, Ex, Mem, WB
- No bypass, no branch prediction!
  - Multithreading can hide 300-cycle memory latency and 4-cycle pipeline latency without bypass/ br pred
- In-order issue but out-of-order complete
  - For special long-latency graphics functions like cosine, sinh
  - Scoreboard for scheduling [565]

Why energy efficient

- Simple pipelines help
- Lock-step lanes
  - 1 32-wide structure more efficient than 32 1-wide structures
    - Eg register file, caches
    - Common control overheads amortized 32-way
      - Eg SRAM address decode for register file or caches
- Instruction processing overhead amortized 32-way
  - Decode, control signals

What about branches

- Lock-step lanes - same performance problem as vectors
  - Programmable: Good old familiar/easy branches, not vector masks
- Lanes may not all have same branch outcome
  - Some taken (if) and others not-taken (else)
  - If same outcome (eg if i<n in SAXPY), no problem
- First run the if-path lanes until if-else reconvergence point while else-path lanes idle → lose performance
- Then run else-path lanes until if-else reconvergence point while if-path lanes idle → lose performance
  - Compiler finds if-else reconvergence point

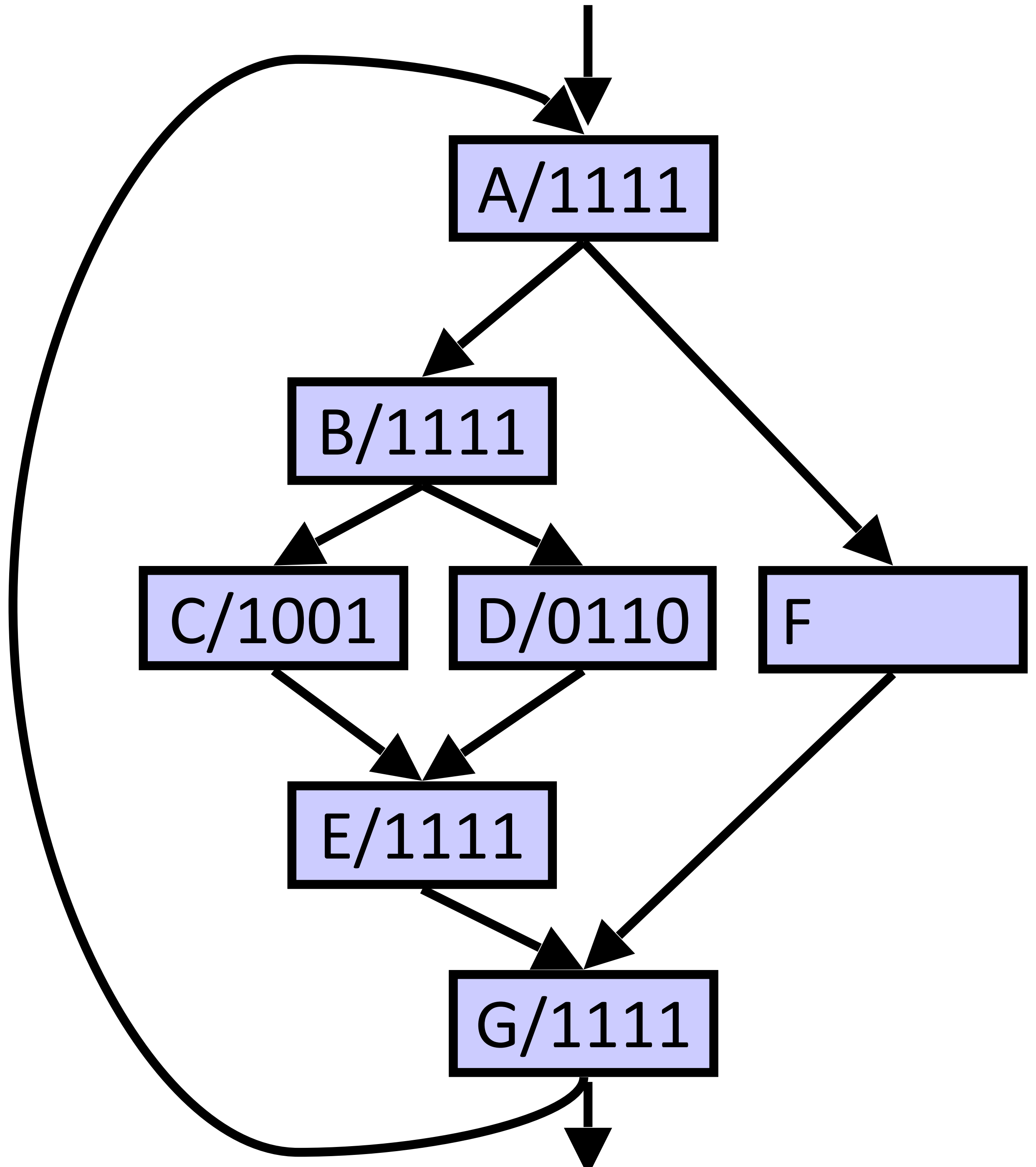
What about branches

- Hardware stack for nested if-elses
  - More programmable than vector masks
  - Code looks like if—else
  - Hardware tracks if path and else path and reconvergence
- Hurts performance if many data-dependent branches
  - Works well in graphics workloads because branches often coarse-grain not data-dependent (SAXPY eg) → lanes stay together

SIMT Using a Hardware Stack

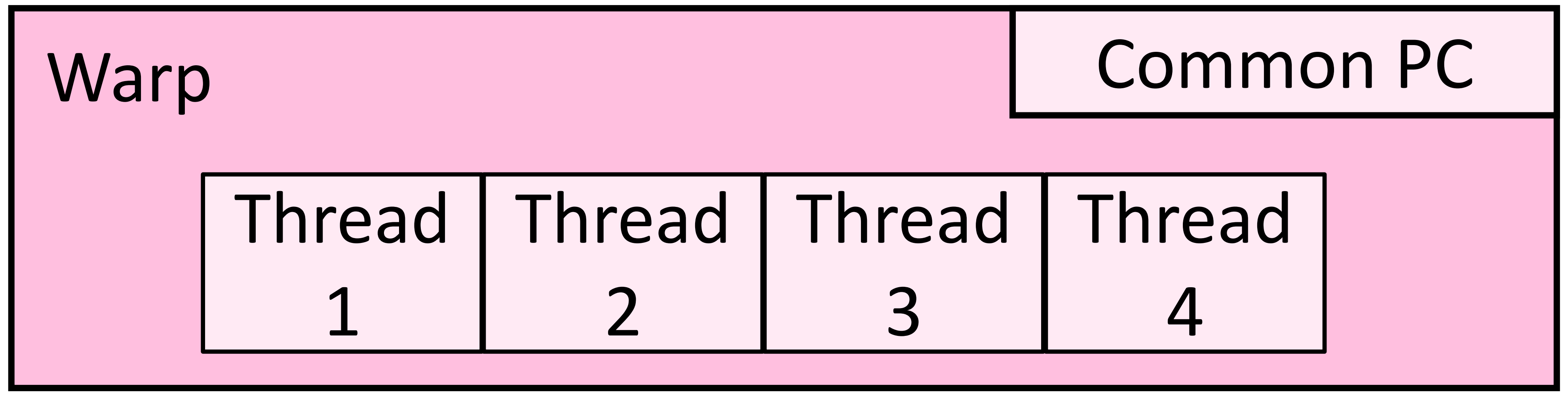
Stack approach invented in early 1980's

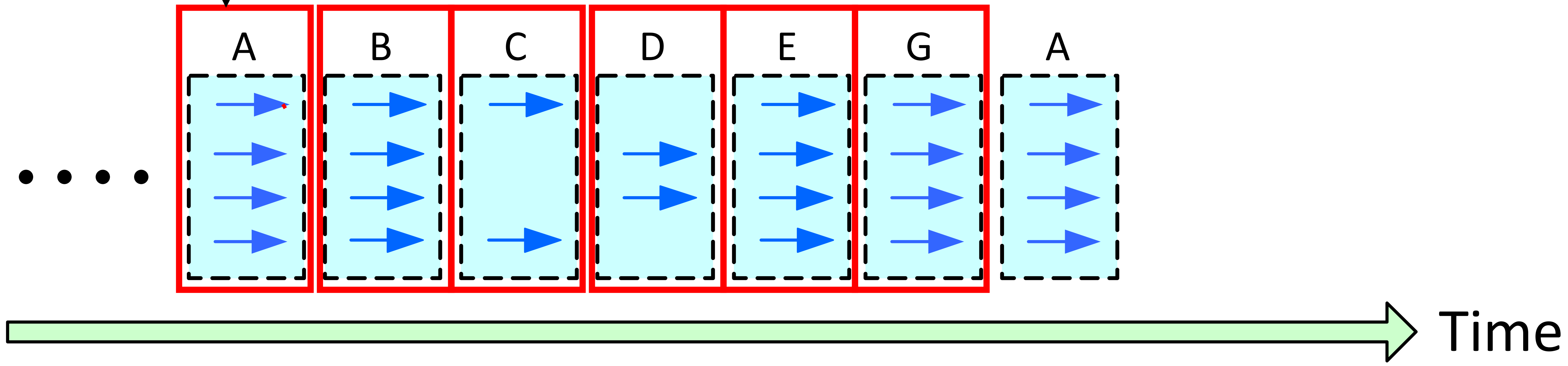
Version here from [Fung et al., MICRO 2007]



Stack

	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001

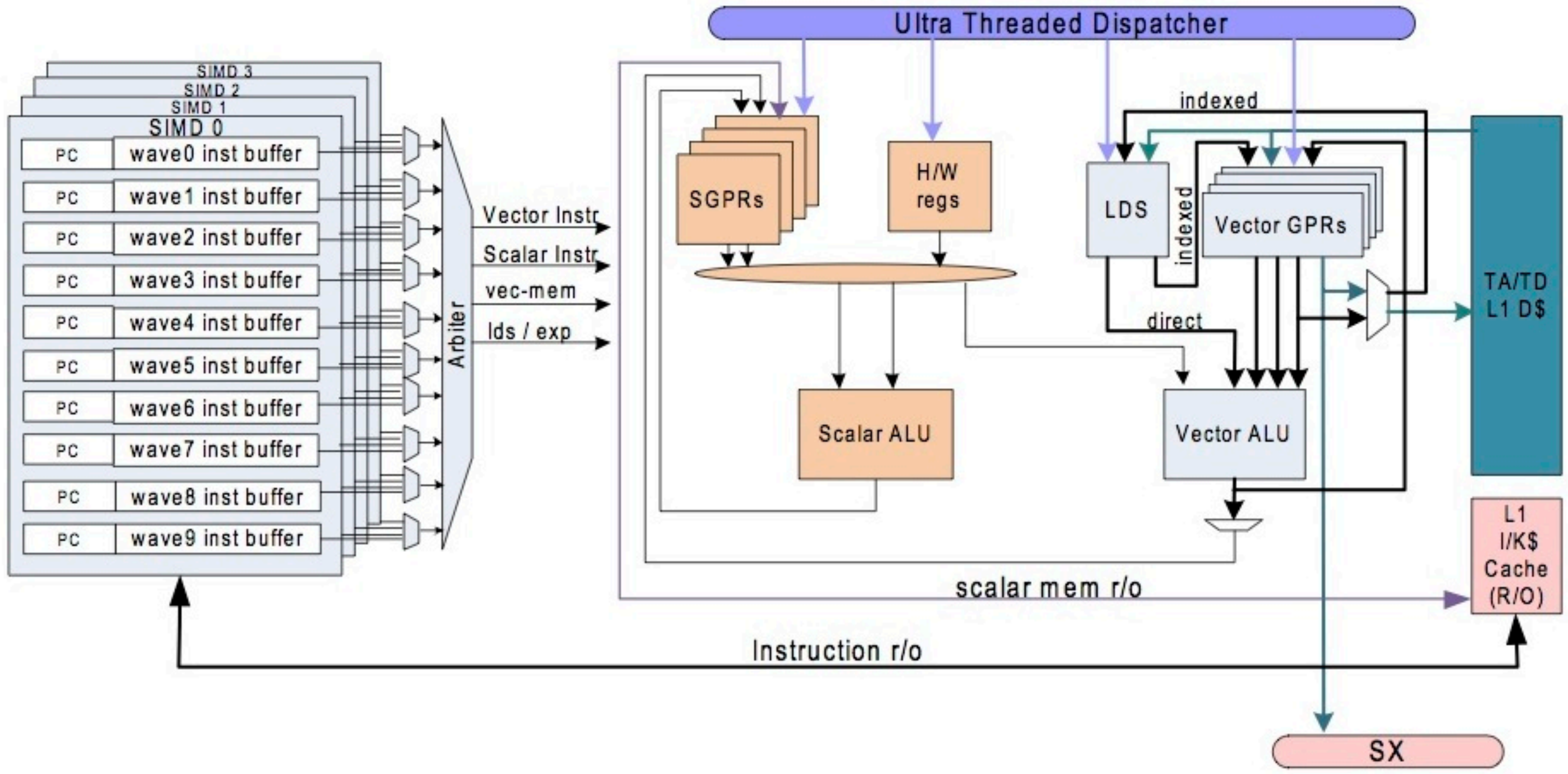




SIMT = SIMD Execution of Scalar Threads

AMD Southern Islands SIMT-Core

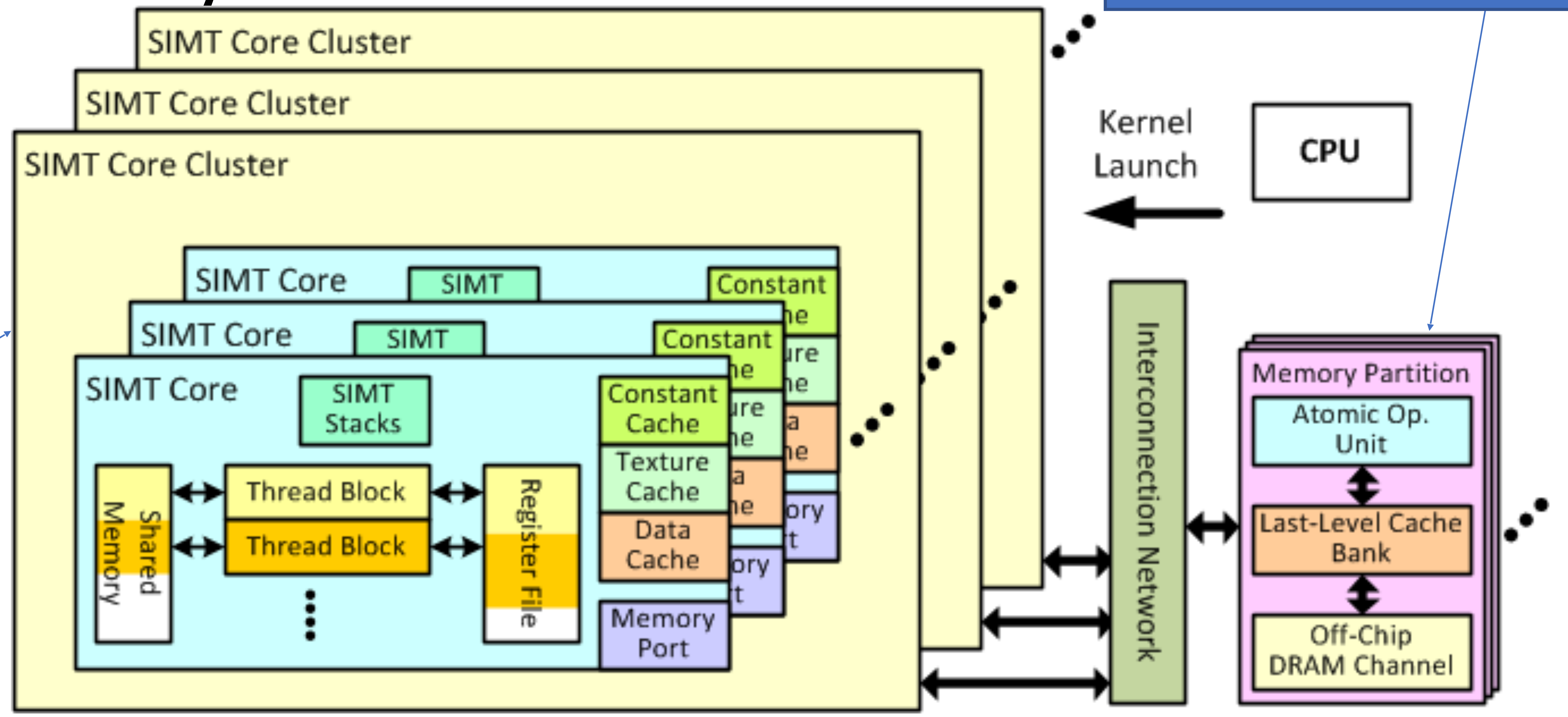
ISA visible scalar unit executes computation identical across SIMT threads in a wavefront



On-chip Interconnection Network/Memory Partitions

- Multiple DRAM chips
  - Similar to CPUs – but in GPUs the chips are on the board.
- Multiple memory partitions.
  - 1 memory partition is connected to 1 memory channel
- Various patents describe how memory is interleaved
  - 256 or 1024B/partition.

Memory Partition



From the perspective of the interconnect:
SIMT Core Cluster is one node

What is High-Bandwidth Memory (HBM)?

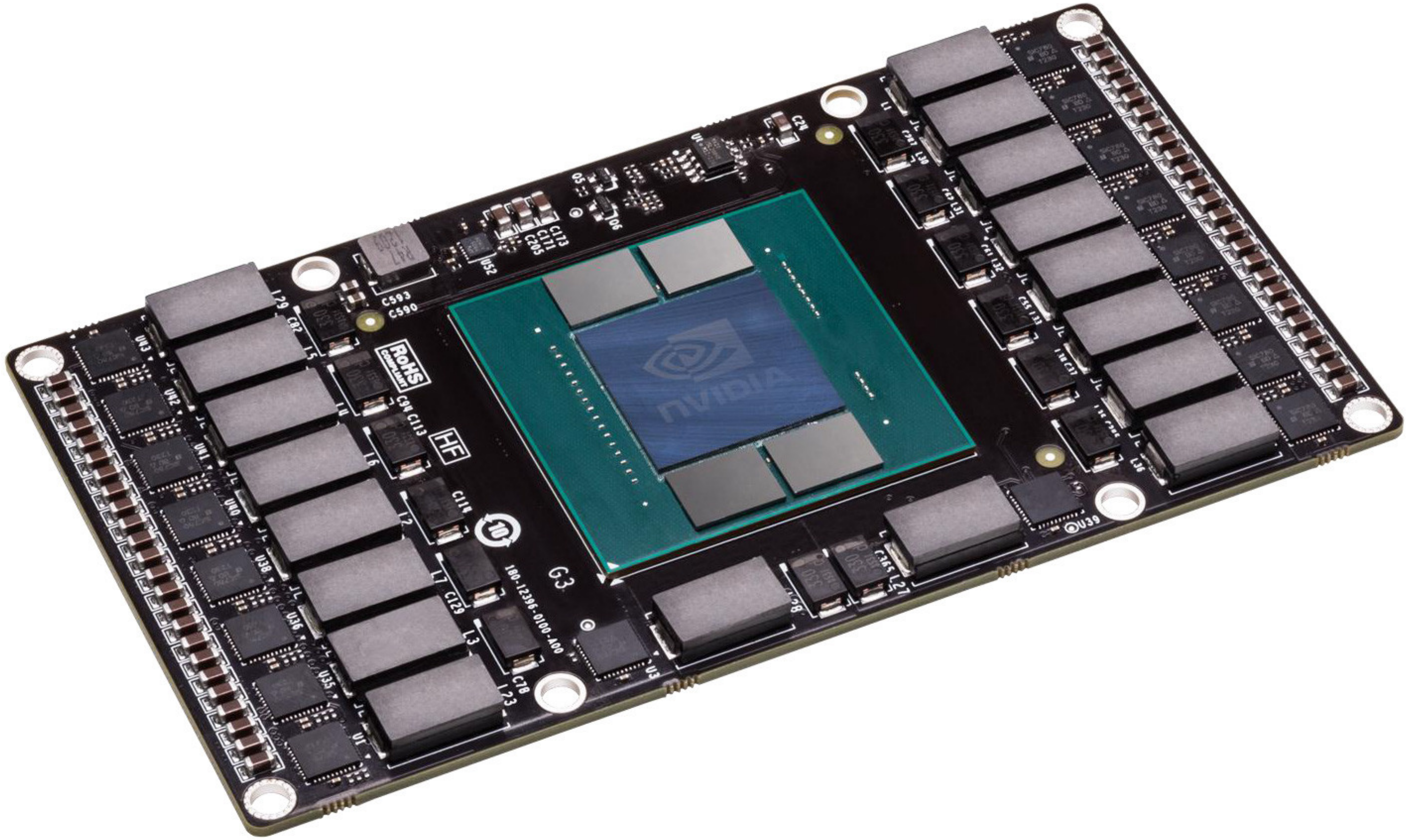
Memory standard designed for needs of future GPU and HPC systems:

- Exploit very large number of signals available with die-stacking technologies for very high memory bandwidth
- Reduce I/O energy costs
- Enable higher fraction of peak bandwidth to be exploited by sophisticated memory controllers
- Enable ECC/Resilience Features

JEDEC standard JESD235, adopted Oct 2013.

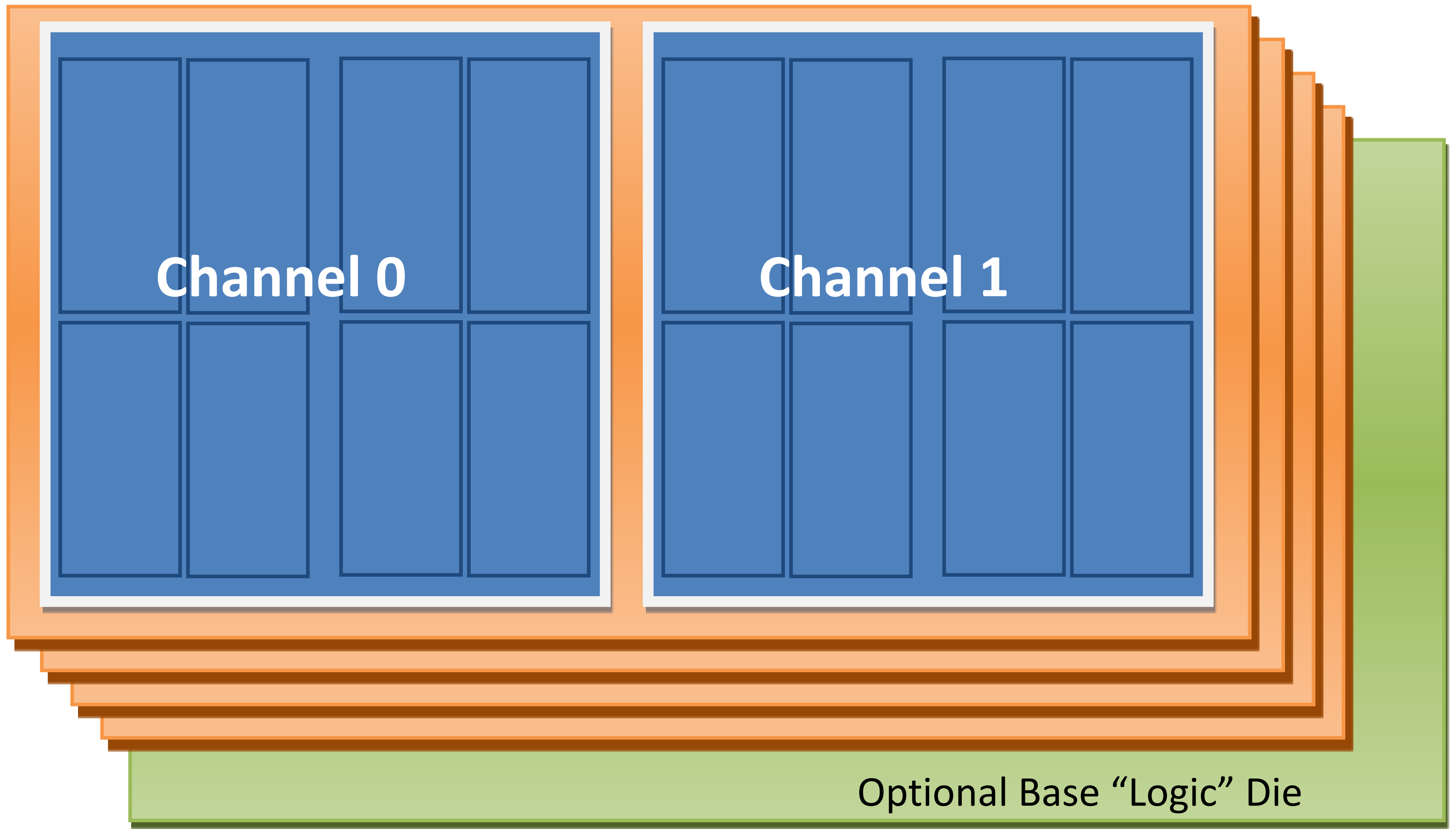
- Initial work started in 2010

What is High-Bandwidth Memory (HBM)?



Enables systems with extremely high bandwidth requirements like future high-performance GPUs

HBM Overview



Each HBM stack provides 8 independent memory channels

- These are completely independent memory interfaces
  - Independent clocks & timing
  - Independent commands
  - Independent memory arrays
  - In short, nothing one channel does affects another channel

Part 5: Where we are going

A cyclic trajectory

GPUs are becoming increasingly specialized

- Tensor cores: Dedicated GEMM units
- Specialized Tensor DMA engines (Tensor Memory Accelerator)
- More and more area/every devoted to the compute for machine learning
- Evolving to a future of “**AI in everything**”