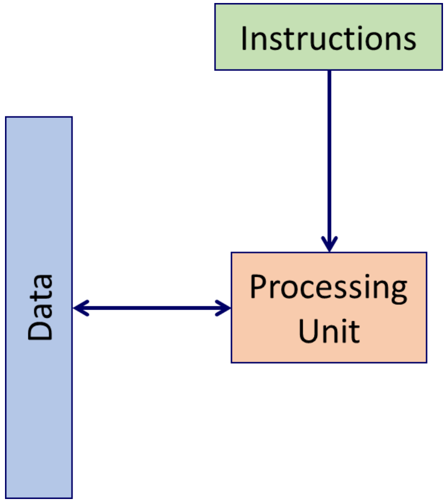
CS152: Computer Systems Architecture
SIMD Operations

Sang-Woo Jun

Winter 2021

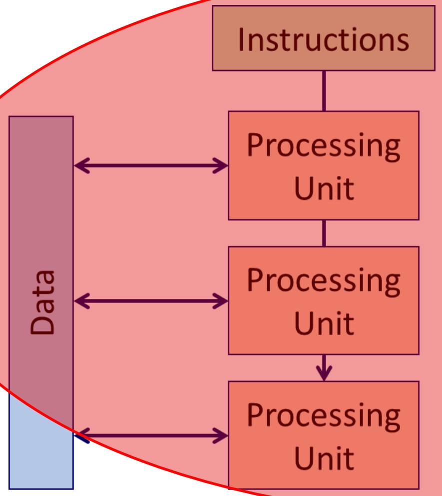
Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Flynn taxonomy

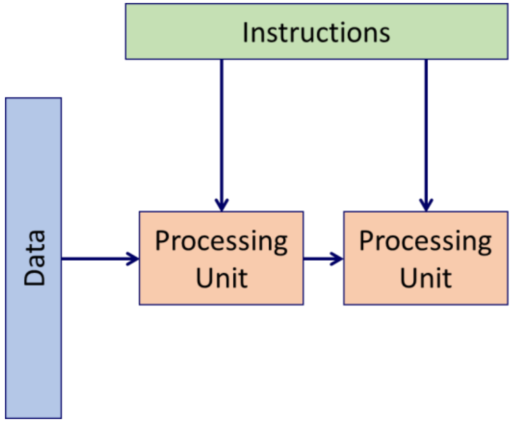


Single-Instruction
Single-Data
(Single-Core Processors)

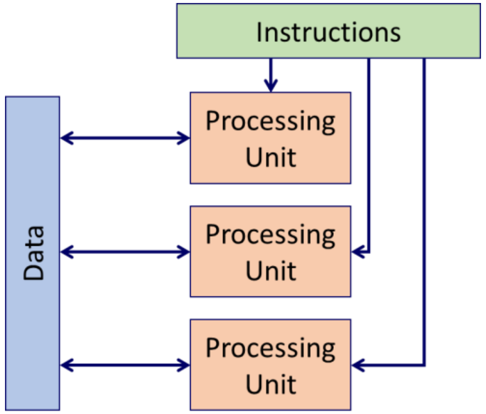
Today



Single-Instruction
Multi-Data
(GPUs, Intel SIMD Extensions)



Multi-Instruction
Single-Data
(Systolic Arrays,…)



Multi-Instruction
Multi-Data
(Parallel Processors)

Modern Processor Topics

- Transparent Performance Improvements
  - Pipelining, Caches
  - Superscalar, Out-of-Order, Branch Prediction, Speculation, …
  - Covered in CS250A and others
- Explicit Performance Improvements
  - SIMD extensions, AES extensions, …
  - …

SIMD operations

- Single ISA instruction performs same computation on multiple data
- Typically implemented with special, wider registers
- Example operation:
  - Load 32 bytes from memory to special register X
  - Load 32 bytes from memory to special register Y
  - Perform addition between each 4-byte value in X and each 4 byte value in Y
  - Store the four results in special register Z
  - Store Z to memory

For i in (0 to 7): Z[i] = X[i] + Y[i];

- RISC-V SIMD extensions (P) is still being worked on (as of 2021)

Example: Intel SIMD Extensions

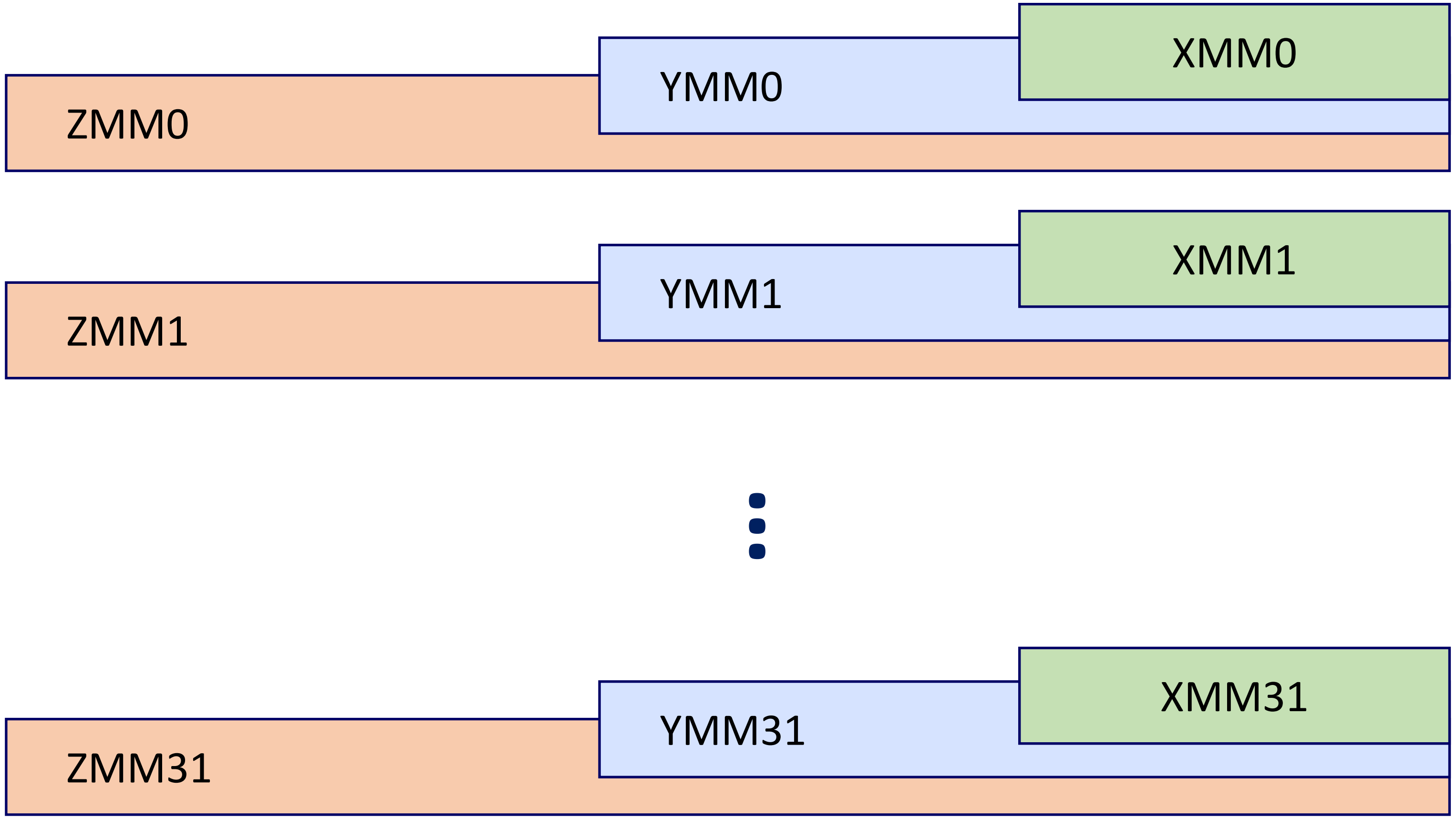
- More transistors (Moore's law) but no faster clock, no more ILP…
  - More capabilities per processor has to be explicit!
- New instructions, new registers
  - Must be used explicitly by programmer or compiler!
- Introduced in phases/groups of functionality
  - SSE – SSE4 (1999 –2006)
    - 128 bit width operations
  - AVX, FMA, AVX2, AVX-512 (2008 – 2019)
    - 256 – 512 bit width operations
  - F16C, and more to come?

Aside: Do I Have SIMD Capabilities?

- less /proc/cpuinfo

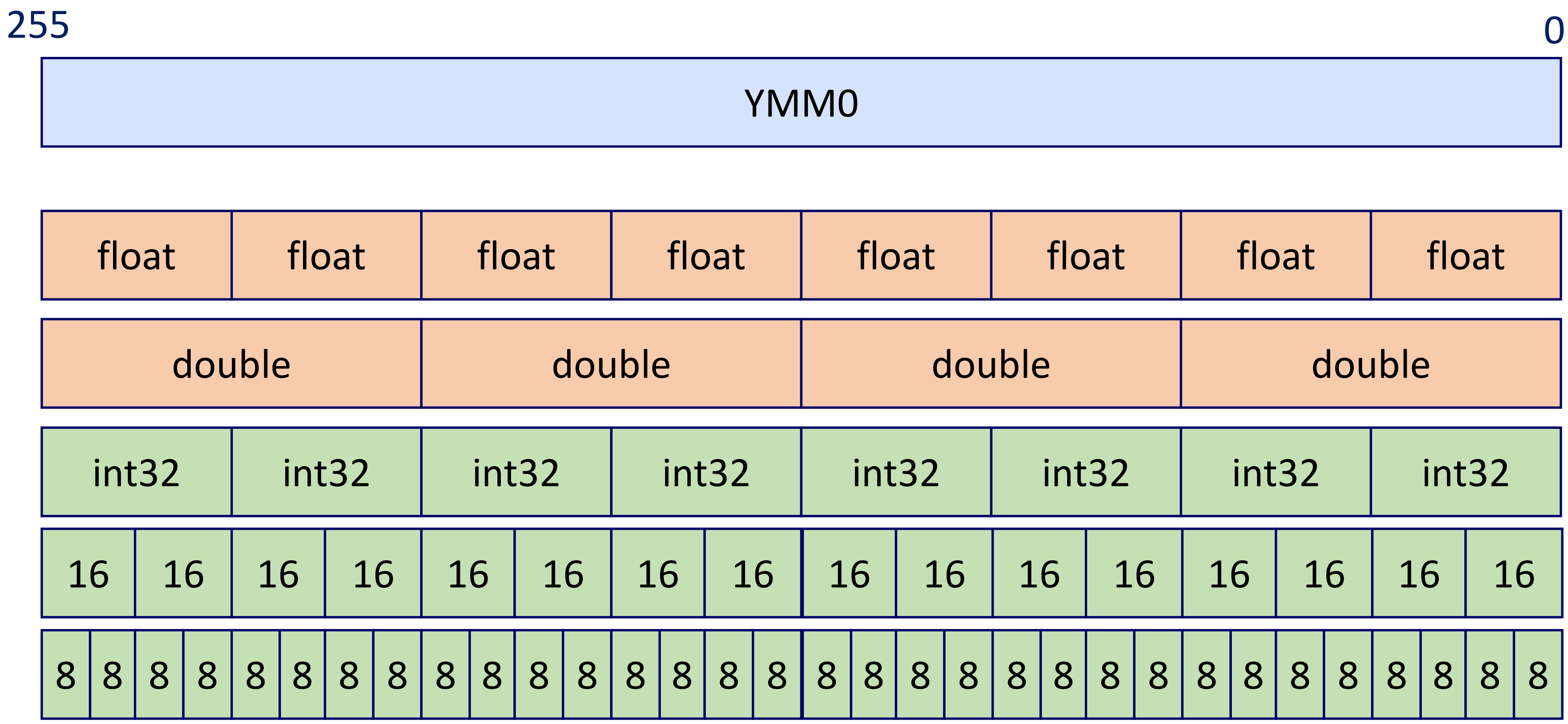
```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat p
se36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm con
stant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmp
erf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx1
6 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f
16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibp
b stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2
 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves
dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```

Intel SIMD Registers (AVX-512)



- XMM0 – XMM15
  - 128-bit registers
  - SSE
- YMM0 – YMM15
  - 256-bit registers
  - AVX, AVX2
- ZMM0 – ZMM31
  - 512-bit registers
  - AVX-512

SSE/AVX Data Types



Operation on
32 8-bit values
in one instruction!

Processor Microarchitectural Effects on Power Efficiency

- The majority of power consumption of a CPU is not from the ALU
  - Cache management, data movement, decoding, and other infrastructure
  - Adding a few more ALUs should not impact power consumption
- Indeed, 4X performance via AVX does not add 4X power consumption
  - FromiI7 4770K measurements:
  - Idle: 40 W
  - Under load : 117 W
  - Under AVX load : 128 W

Compiler Automatic Vectorization

- In gcc, flags “-O3 –mavx –mavx2” attempts automatic vectorization
- Works pretty well for simple loops

```c
int a[256], b[256], c[256];
void foo () {
    for (int i=0; i<256; i++) a[i] = b[i] * c[i];
}
```

```
.L2:
        vmovdqa xmm1, XMMWORD PTR b[rax]
        add     rax, 16
        vpmulld xmm0, xmm1, XMMWORD PTR c[rax-16]
        vmovaps XMMWORD PTR a[rax-16], xmm0
        cmp     rax, 1024
        jne     .L2
```

Generated using GCC explorer: https://gcc.godbolt.org/

- But not for anything complex
  - E.g., naïve bubblesort code not parallelized at all

Intel SIMD Intrinsics

- Use C functions instead of inline assembly to call AVX instructions
- Compiler manages registers, etc
- Intel Intrinsics Guide
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide
  - One of my most-visited pages…

e.g.,

```
__m256 a, b, c;
__m256 d = _mm256_fmadd_ps(a, b, c); // d[i] = a[i]*b[i]+c[i] for i = 0 ...7
```

Intrinsic Naming Convention

- _mm<width>_[function]_[type]
  - E.g., _mm256_fmadd_ps :
    perform fmadd (floating point multiply-add) on
    256 bits of
    packed single-precision floating point values (8 of them)

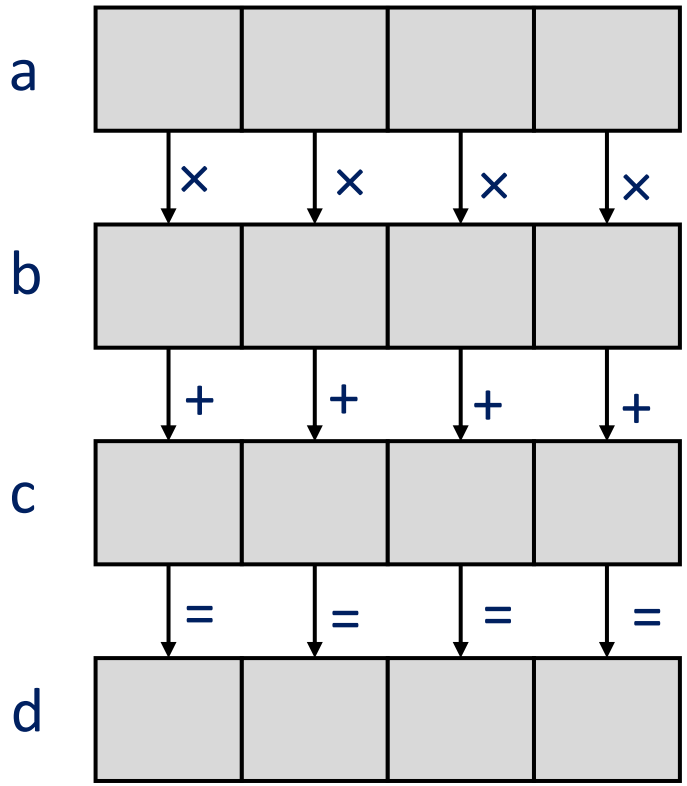
Width	Prefix
128	_mm_
256	_mm256_
512	_mm512_

Not all permutations exist! Check guide

Type	Postfix
Single precision	_ps
Double precision	_pd
Packed signed integer	_epiNNN (e.g., epi256)
Packed unsigned integer	_epuNNN (e.g., epu256)
Scalar integer	_siNNN (e.g., si256)

Example: Vertical Vector Instructions

- Add/Subtract/Multiply
  - _mm256_add/sub/mul/div_ps/pd/epi
    - Mul only supported for epi32/epu32/ps/pd
    - Div only supported for ps/pd
    - Consult the guide!
- Max/Min/GreaterThan/Equals
- Sqrt, Reciprocal, Shift, etc…
- FMA (Fused Multiply-Add)
  - (a*b)+c, -(a*b)-c, -(a*b)+c, and other permutations!
  - Consult the guide!
- …




```
__m256 a, b, c;
__m256 d = _mm256_fmadd_pd(a, b, c);
```

Integer Multiplication Caveat

- Integer multiplication of two N bit values require 2N bits
- E.g., __mm256_mul_epi32 and __mm256_mul_epu32
  - Only use the lower 4 32 bit values
  - Result has 4 64 bit values
- E.g., __mm256_mullo_epi32 and __mm256_mullo_epu32
  - Uses all 8 32 bit values
  - Result has 8 truncated 32 bit values
- And more options!

Case Study: Matrix Multiply

- Branch :
  Boser & Katz,
  “CS61C: Great Ideas In Computer Architecture”
  Lecture 18 – Parallel Processing – SIMD

CS152: Computer Systems Architecture
GPU Computing Introduction

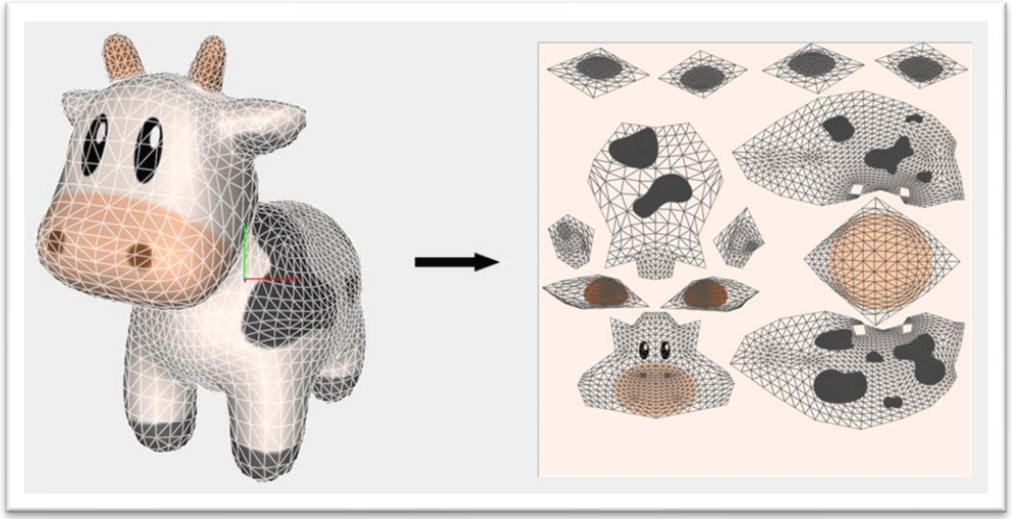
Sang-Woo Jun

Winter 2021

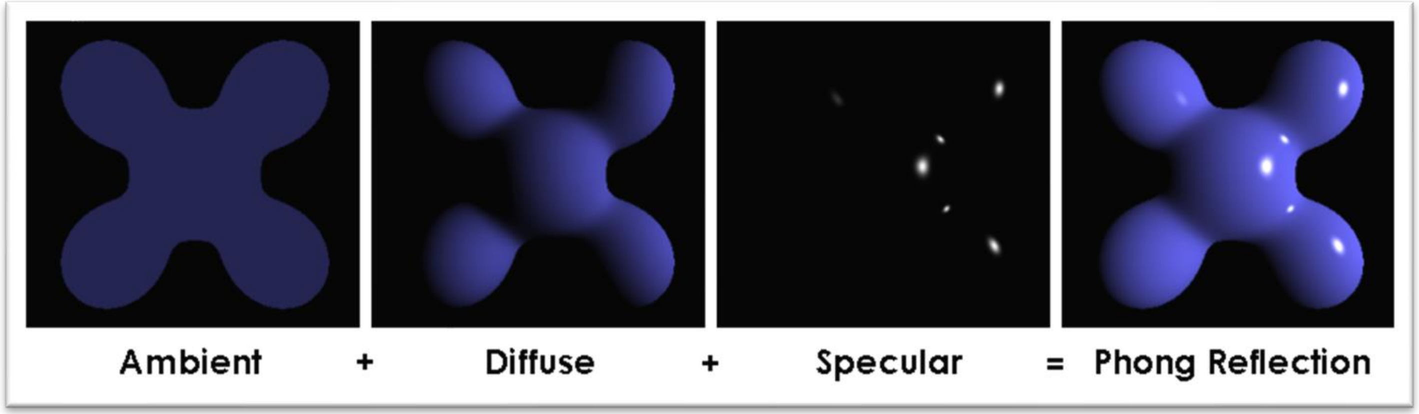
UCI

Graphic Processing – Some History

- ❑ 1990s: Real-time 3D rendering for video games were becoming common
  - Doom, Quake, Descent, … (Nostalgia!)
- ❑ 3D graphics processing is immensely computation-intensive



Texture mapping



Shading

Warren Moore, “Textures and Samplers in Metal,” Metal by Example, 2014
Gray Olsen, “CSE 470 Assignment 3 Part 2 - Gourad/Phong Shading,” grayolsen.com, 2018

Graphic Processing – Some History

- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played

Doom (1993) : "Affine texture mapping"

- Linearly maps textures to screen location, disregarding depth
- Doom levels did not have slanted walls or ramps, to hide this

Graphic Processing – Some History

- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played

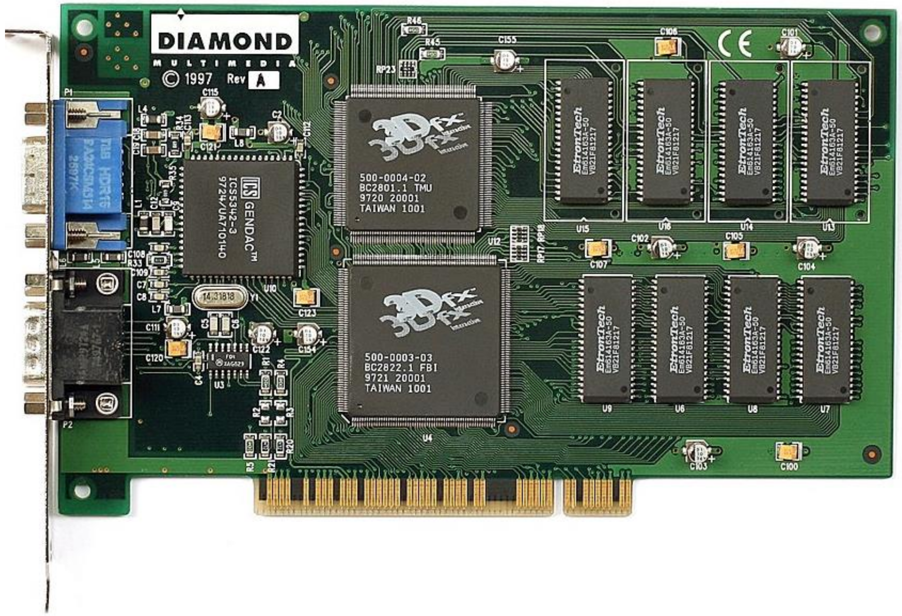
Quake III arena (1999) : “Fast inverse square root” magic!

```c
float Q_rsqrt( float number )
{
    const float x2 = number * 0.5F;
    const float threehalfs = 1.5F;

    union {
        float f;
        uint32_t i;
    } conv = {number}; // member 'f' set to value of 'number'.
    conv.i  = 0x5f3759df - ( conv.i >> 1 );
    conv.f  *= ( threehalfs - ( x2 * conv.f * conv.f ) );
    return conv.f;
}
```

Introduction of 3D Accelerator Cards

- Much of 3D processing is short algorithms repeated on a lot of data
  - pixels, polygons, textures, …
- Dedicated accelerators with simple, massively parallel computation

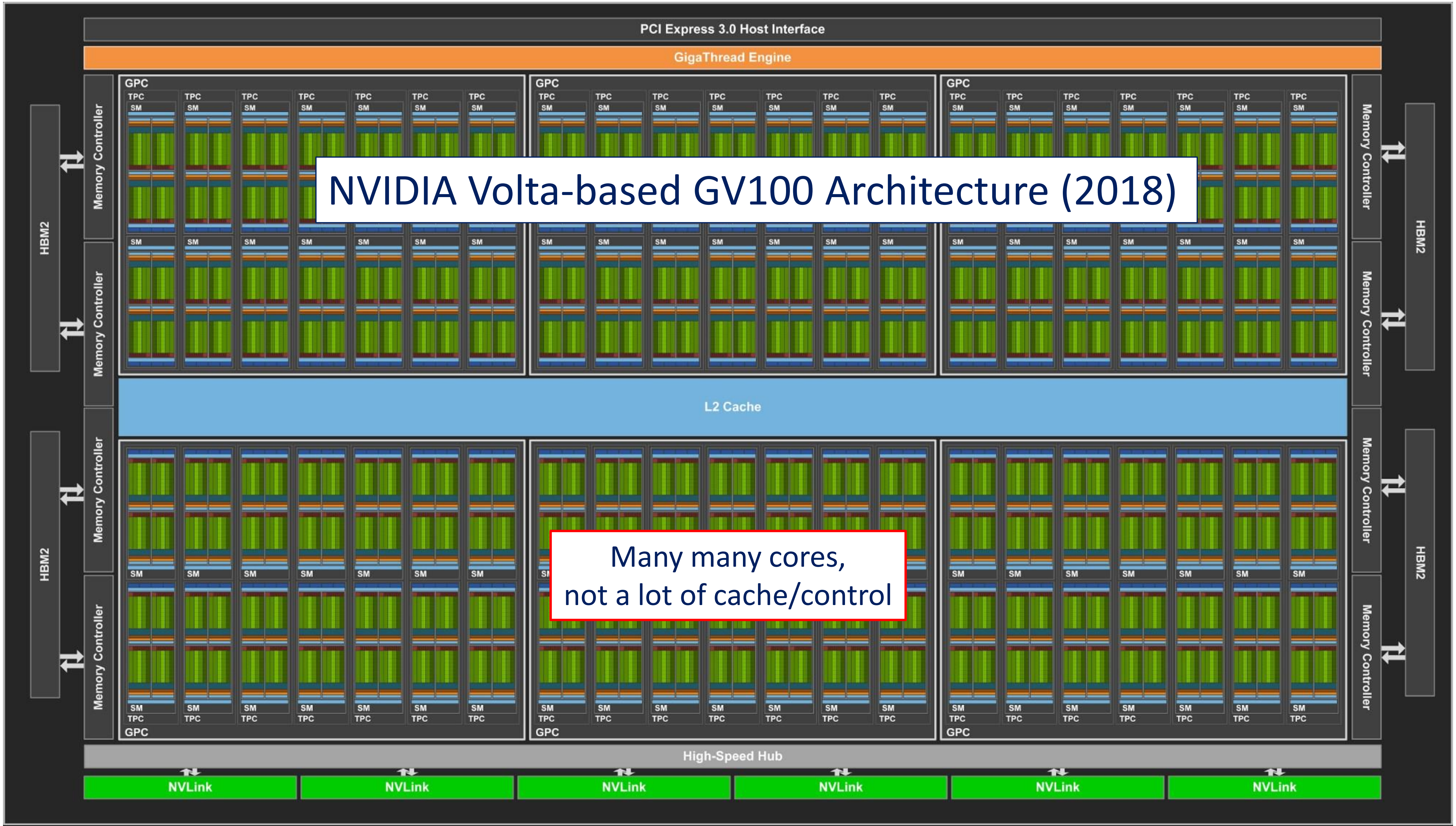


A Diamond Monster 3D, using the Voodoo chipset (1997)
(Konstantin Lanzet, Wikipedia)

General-Purpose Graphic Processing Units (GPGPU)

- Massively parallel architecture created for graphics processing, opened up for general purpose programming
  - Thousands of simple cores with high floating-point processing capability
    - Floating point operations important for graphics processing
  - Very fast off-chip memory originally used for graphics processing

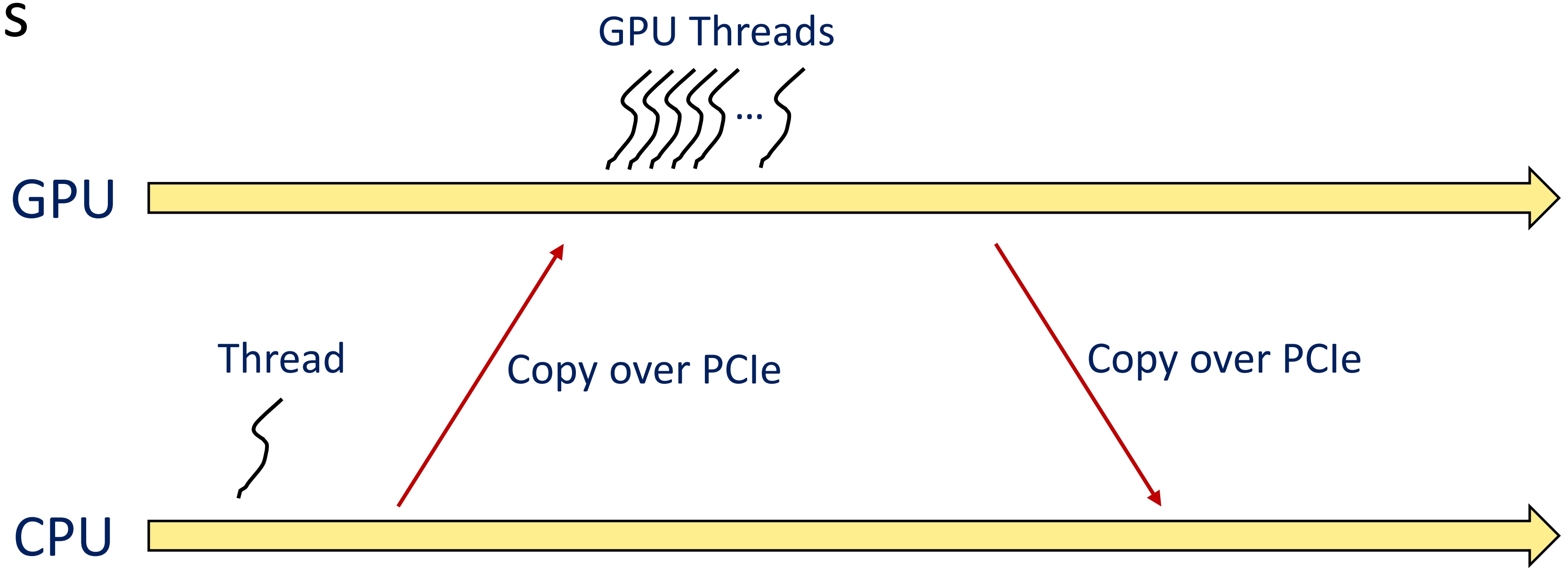
NVIDIA Volta-based GV100 Architecture (2018)



Many many cores,
not a lot of cache/control

Massively Parallel Architecture For Massively Parallel Workloads!

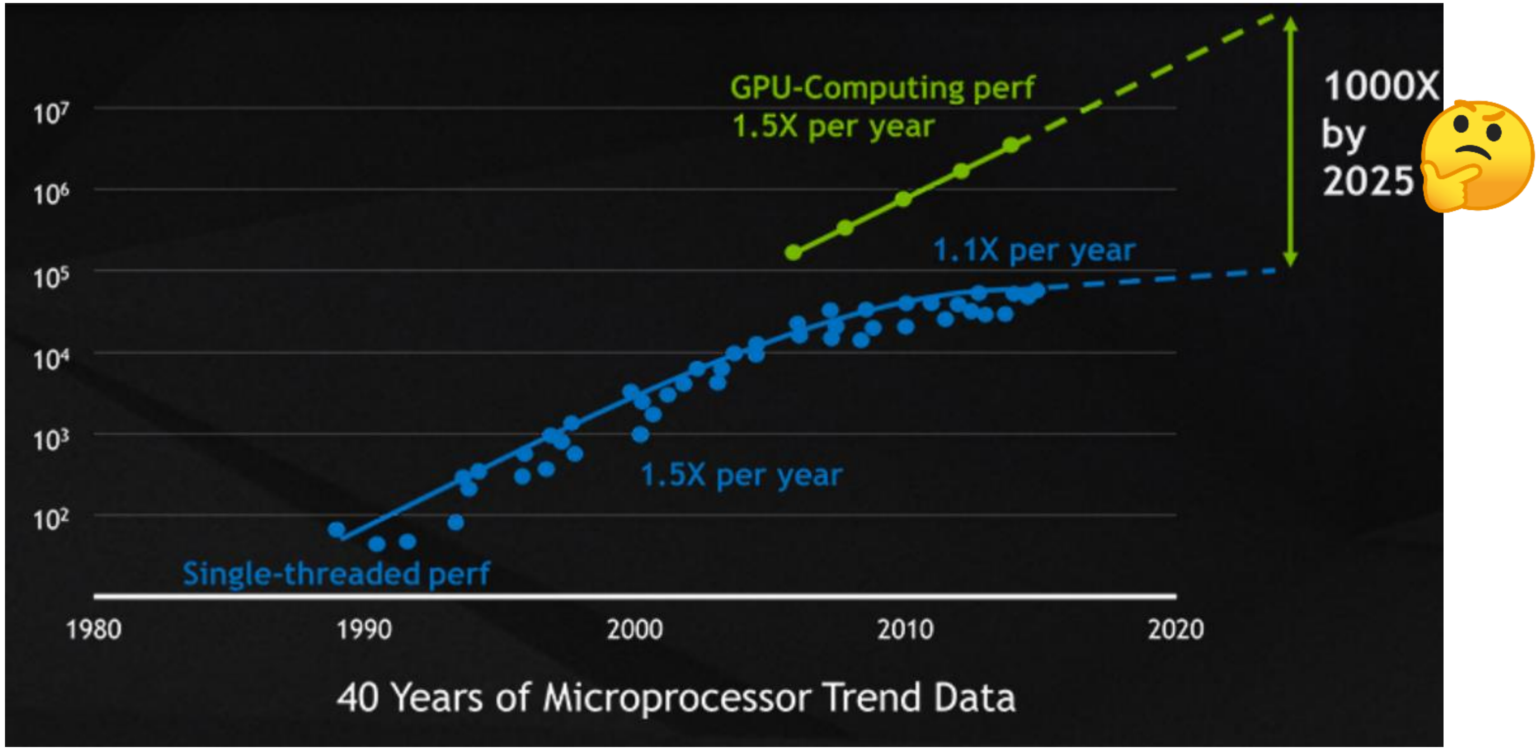
- NVIDIA CUDA (Compute Uniform Device Architecture) – 2007
  - A way to run custom programs on the massively parallel architecture!
- OpenCL specification released – 2008
- Both platforms expose synchronous execution of a massive number of threads



Peak Performance vs. CPU

	Throughput	Power	Throughput/Power
Intel Skylake	128 SP GFLOPS/4 Cores	100+ Watts	~1 GFLOPS/Watt
NVIDIA V100	15 TFLOPS	200+ Watts	~75 GFLOPS/Watt

Also,



GPU programming abstraction

- “SIMT” (Single Instruction Multiple Threads), introduced by NVIDIA
  - Simply put: Identical program (“Kernel”) executed on multiple threads
  - Thread ID is given as a parameter to the program,
    so each thread can perform different work despite identical code
  - Another kernel parameter is “block size”, the number of threads to use

CPU Code example

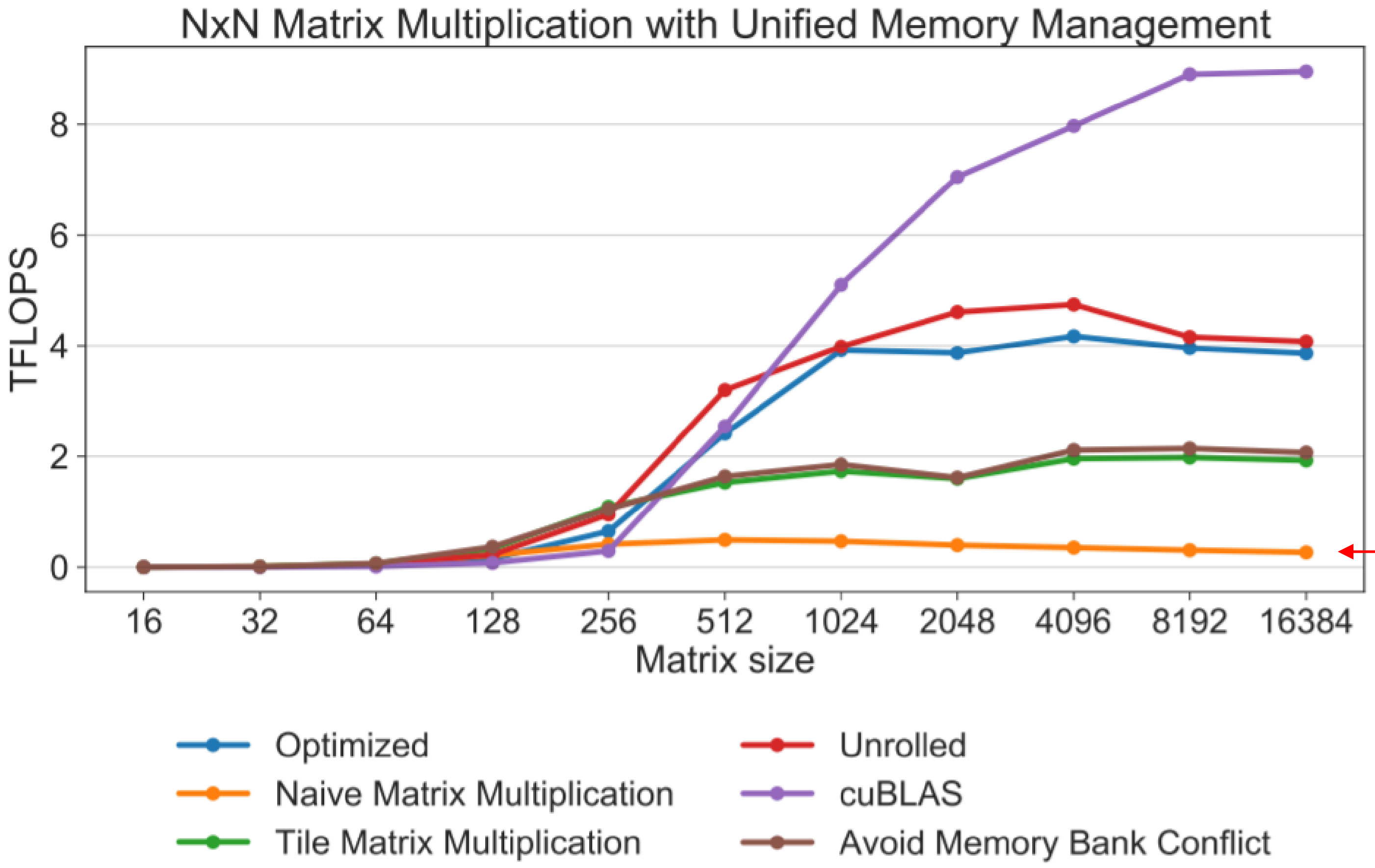
```
for (ii = 0; ii < cnt; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

GPU Code example

```
__global__ void KernelFunction(…) {
   int tid = threadIdx.x;
   int blocksize = ceiling(cnt/blockDim.x);
   for (i = 0; i < blocksize; ++i ) {
      int ii = blocksize*tid+i;
      if ( ii < cnt ) C[ii] = A[ii] + B[ii];
   }
}
```

Thread dimensions given as part of request from host software

Matrix Multiplication Performance Engineering



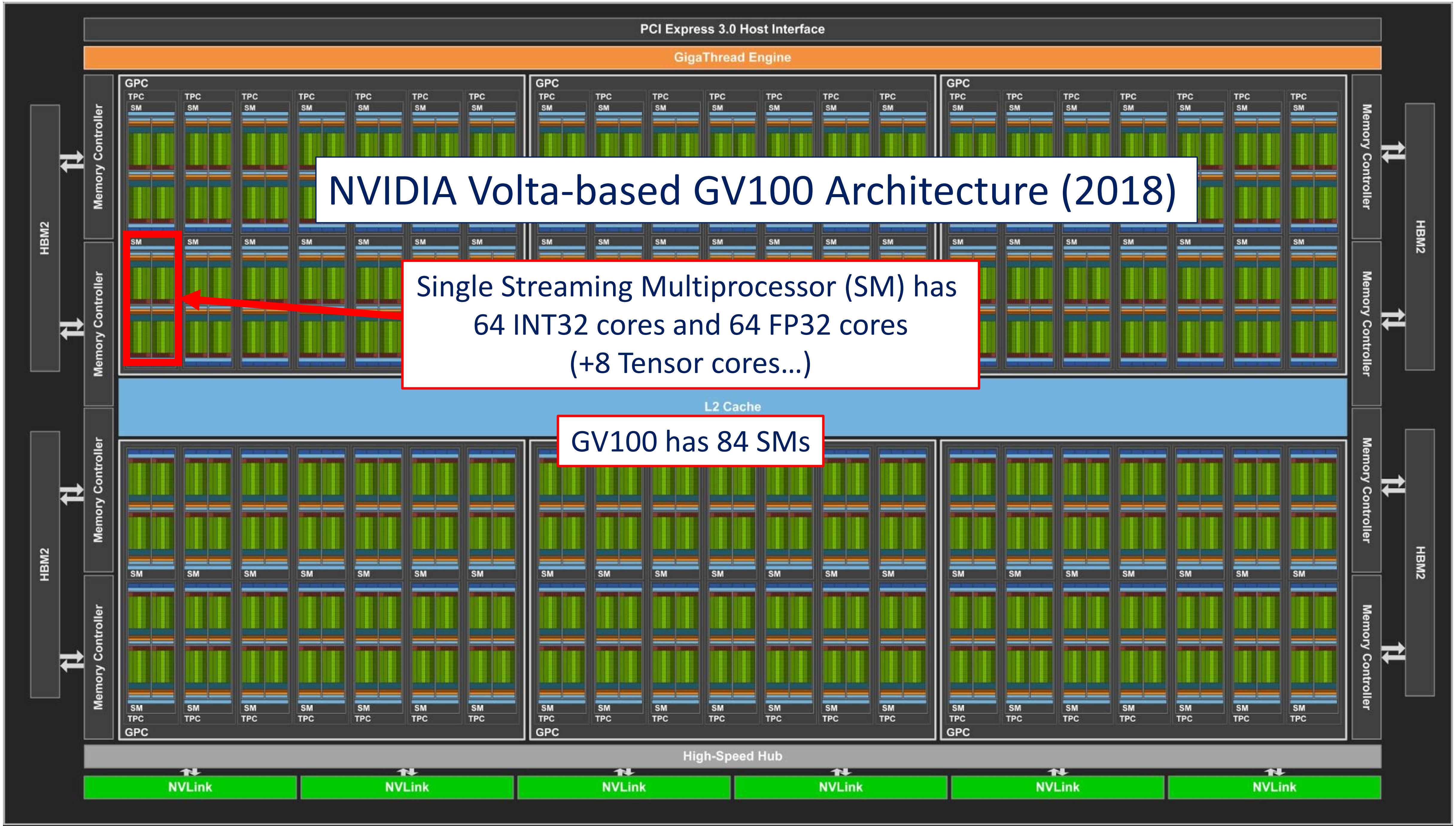
No faster than CPU

Results from NVIDIA P100

Architecture knowledge is needed (again)

Coleman et. al., “Efficient CUDA,” 2017

NVIDIA Volta-based GV100 Architecture (2018)



Single Streaming Multiprocessor (SM) has 64 INT32 cores and 64 FP32 cores (+8 Tensor cores…)

GV100 has 84 SMs

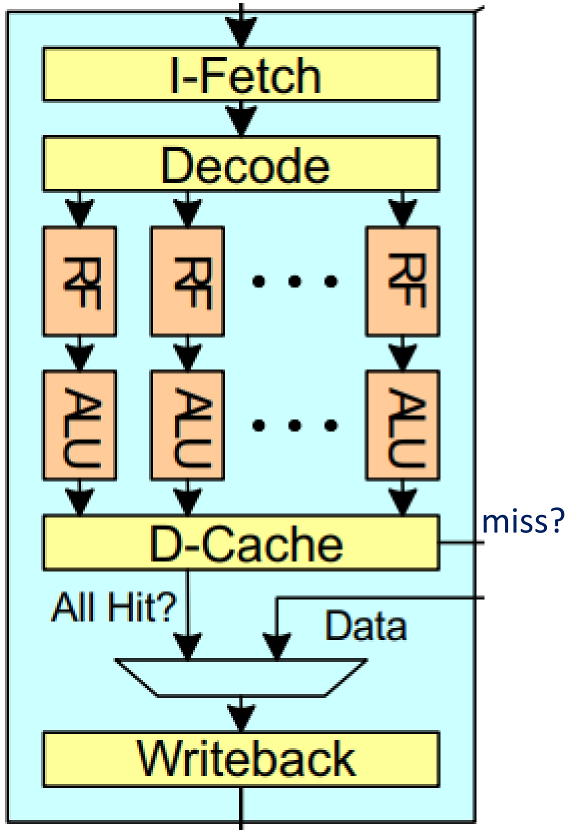
GPU processor architecture

- GPUs have thousands of threads running concurrently at multiple gigabytes!

- Much simpler processor architecture
  - Dozens of threads scheduled together in a SIMD fashion
  - Much simpler microarchitecture (doesn’t need to boot Linux!)
- Much higher power budget
  - CPUs try to maintain 100 W power budget (Pentium 4 till now)
  - GPUs regularly exceed 400 W

GPU processor architecture

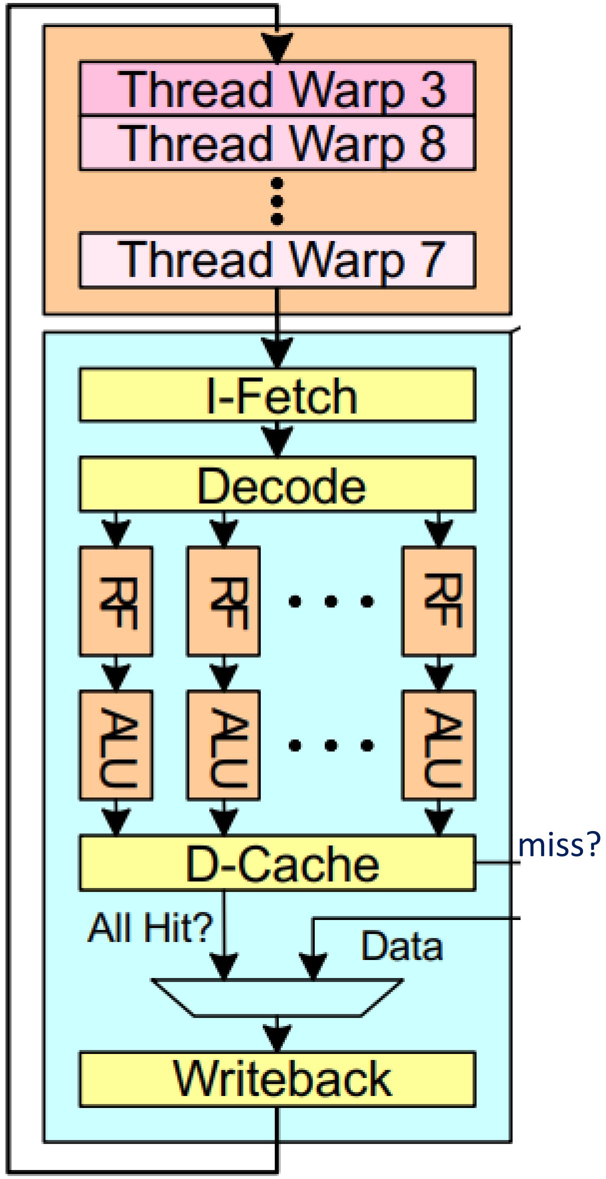
- Cores are organized into units of “warps”
  - Threads in a warp share the same Fetch and decode units
  - Drastically reduces chip resource usage
    - One reason why GPUs can fit so many cores
  - Basically a warp is one SIMD thread
    - But exposes multithread abstraction to the programmer
  - Typically 32 threads per warp for NVIDIA, but may change
    - Warp size information is not part of programming abstraction



Source: Tor Aamodt

GPU processor architecture

- ❑ Each warp hardware can handle many sets of threads
  - o Context switch in case of memory access request, to hide memory access latency
- ❑ A large block of threads can map across many streaming multiprocessors
  - o Thread 0 to 31 map to warp 0,
    Thread 32 to 63 map to warp 1, …

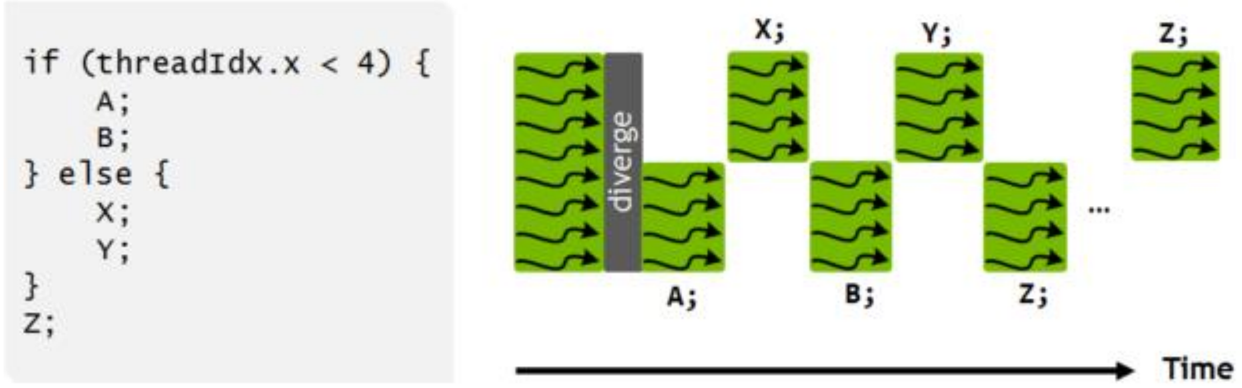


Warp scheduling caveats

- Remember: Threads within a block share the same fetch, decode units
  - All threads in a warp are always executing the same instruction
  - What if their execution diverges?
    - e.g., if (tid%2) func1(), else func2()
    - e.g., if (A[tid] < 100) X++, else Y++
- Divergence across warps don’t matter
  - Different warps, different fetch+decode
- What about intra-warp divergence?

Warp scheduling caveats

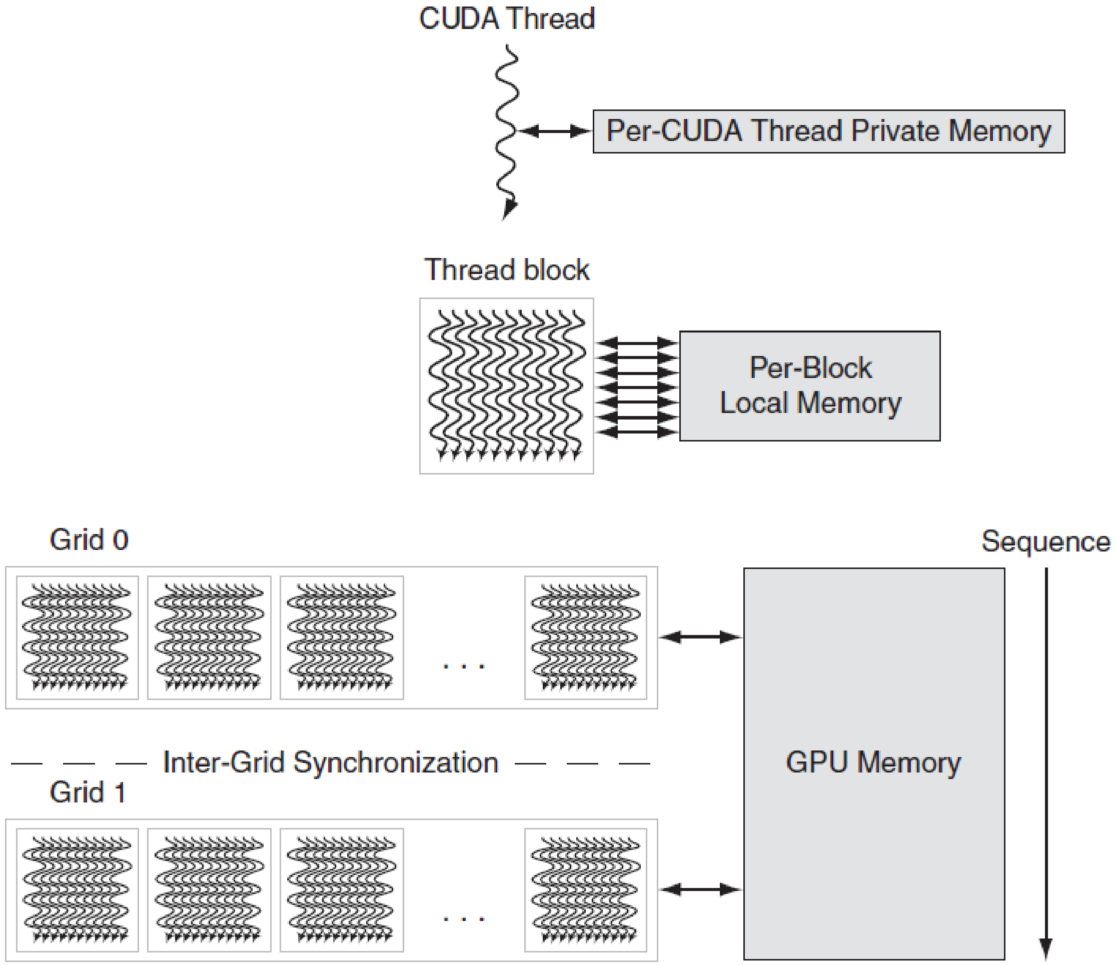
- Intra-warp execution divergence incurs "control divergence"
  - The warp processor must execute both paths, one after another
    - Whole warp will execute one direction first with some threads suspended, and the other direction with the other threads suspended
  - If 32 threads go down 32 different branches, no performance gain with SIMD!
- Warps have been 32-threads so far, but may change in the future



2018, "Using CUDA Warp-Level Primitives," NVIDIA

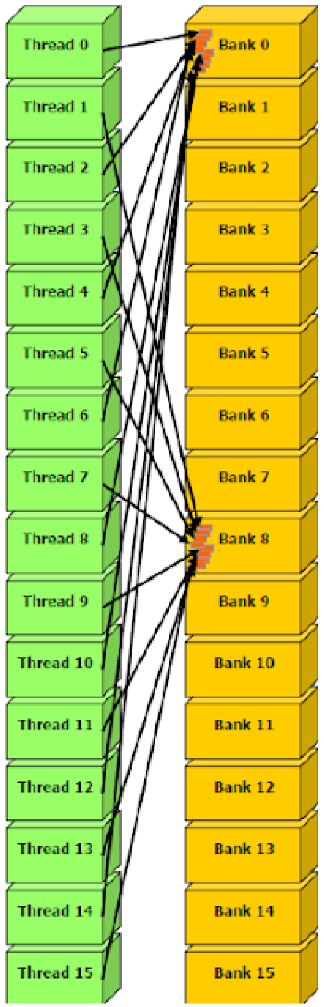
GPU memory architecture

- Not much on-chip memory per thread
  - 1024 Registers per FP32 core
  - 96 KB Shared memory
- Relatively fast off-chip “global” memory
  - But not fast enough!
  - GDDR5 or HBM2 can deliver up to ~1TB/s
  - Shared across 2048+ threads…
- Pretty much no memory consistency between blocks
  - Once data goes to off-chip main memory, explicit synchronization critical!



GPU memory architecture

- Remember: A warp has 32 threads
  - They can all be accessing shared memory at once
    - Difficult to have multiple ports on same memory region
  - Serializing memory access will kill performance
    - Performance will be limited by one shared memory access per thread per cycle
- Organized into banks to distribute access
  - Best performance if all threads in warp access different banks
  - Best performance if all threads access the same back (broadcast)
  - Otherwise, bank conflicts drastically reduce performance



8-way bank conflict
1/8 memory bandwidth

So what are GPUs good for?

- Bottlenecks to watch:
  - PCIe bandwidth is slow, so communication/computation ratio should be low
  - SIMD operations at 32-thread warps, so less branching
    - "Regularly structured" computation
- Good example is matrix multiplication
- Also, Computing convolutions
  - Deep neural networks became feasible with GPUs!