CS4223: Multi-Core Architectures

GPU Architecture and Programming Model

Trevor E. Carlson
School of Computing
National University of Singapore

[Talk originally from Tulika Mitra]
[Partially Adapted from talk by Kayvon Fatahalian]

GPU Parallel Computing

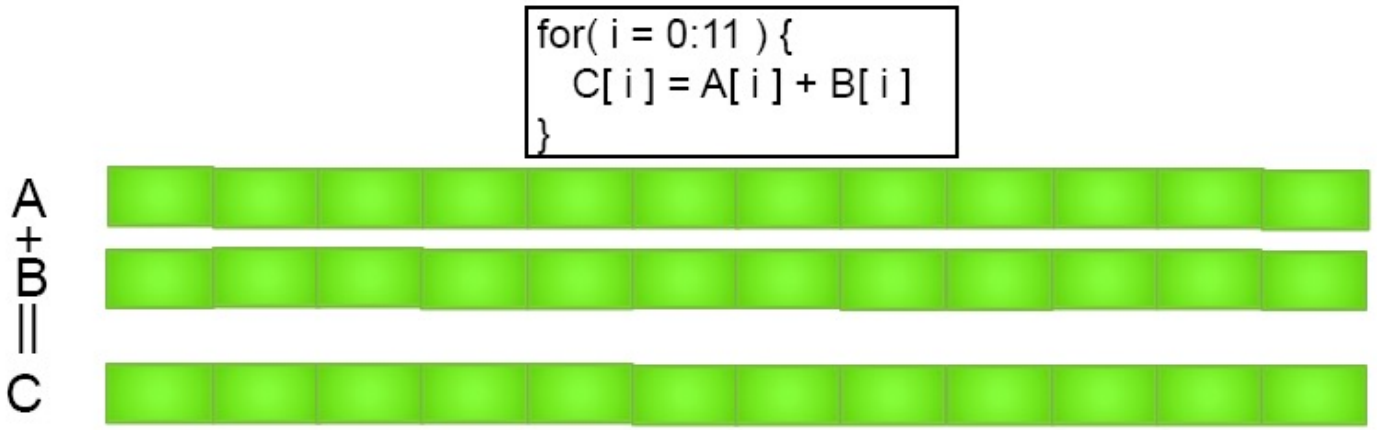
- GPU Parallelism
  - Task and data decomposition
- GPU Parallel computing
  - Software models (SPMD/SIMT)
  - Hardware architectures (SIMD)

Parallel Software - SPMD

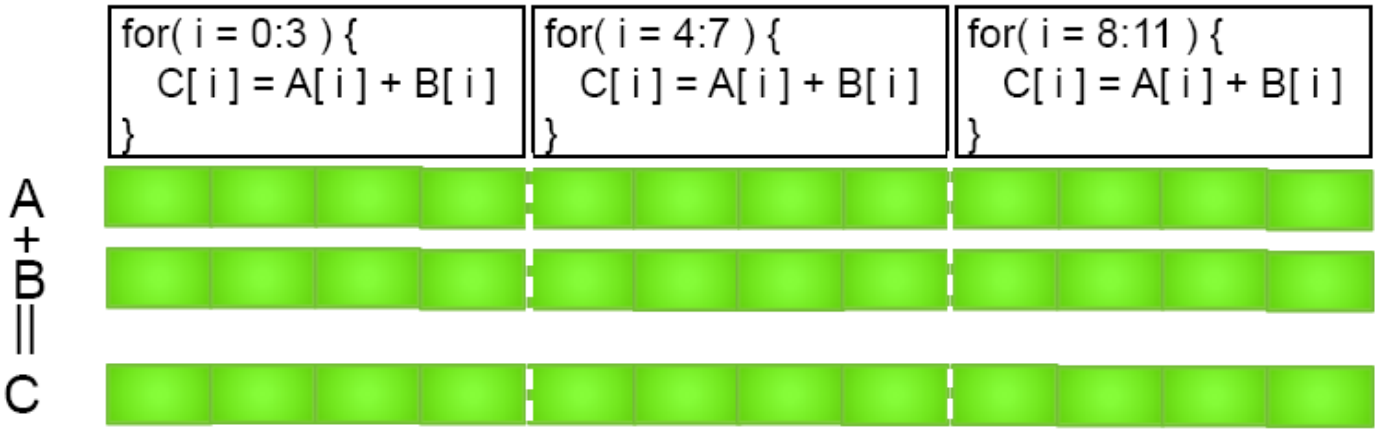
- GPU programs are called kernels and are written using the Single Program Multiple Data (SPMD) programming model
  - SPMD executes multiple instances of the same program independently, where each program works on a different portion of the data

- nVidia calls it Single-Instruction Multiple Thread (SIMT) programming model

Parallel Software - SPMD



Sequential: One thread completes the entire task



SPMD: Multiple copies of the same program run on different chunks of data

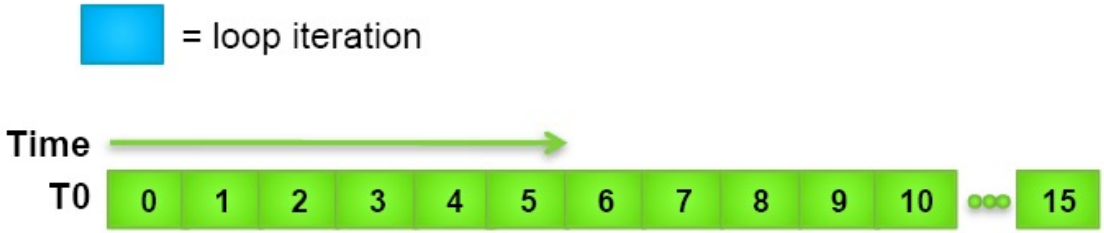
Parallel Software - SPMD

- In the vector addition example, each chunk of data could be executed as an independent thread
- On modern CPUs, the overhead of creating threads is so high that the chunks need to be large
  - In practice, usually a few threads (about as many as the number of CPU cores) and each is given a large amount of work to do
- For GPU programming, there is low overhead for thread creation, so we can create one thread per loop iteration

Parallel Software - SPMD

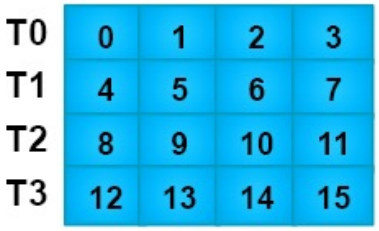
Single-threaded (CPU)

```
// there are N elements
for (i=0; i<N; i++)
  C[i] = A[i] + B[i]
```



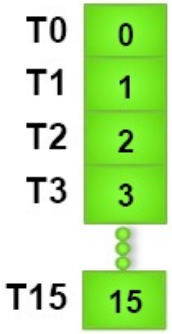
Multi-threaded (CPU)

```
// tid is thread id
//P is number of cores
for (i=0; i<tid*N/P; i++)
   C[i] = A[i] + B[i]
```



Massively Multi-threaded (GPU)

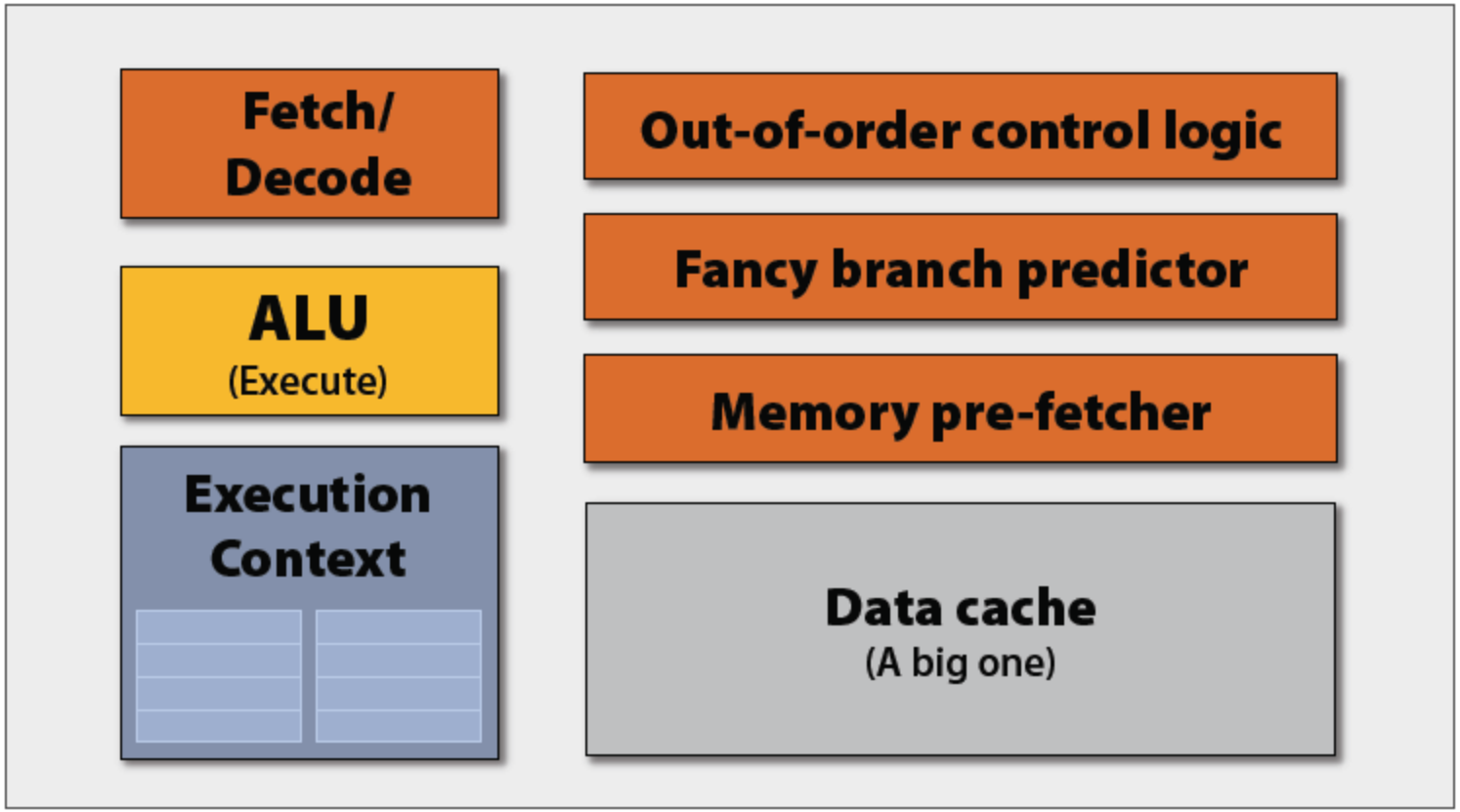
```
// tid is thread id
C[tid] = A[tid] + B[tid]
```



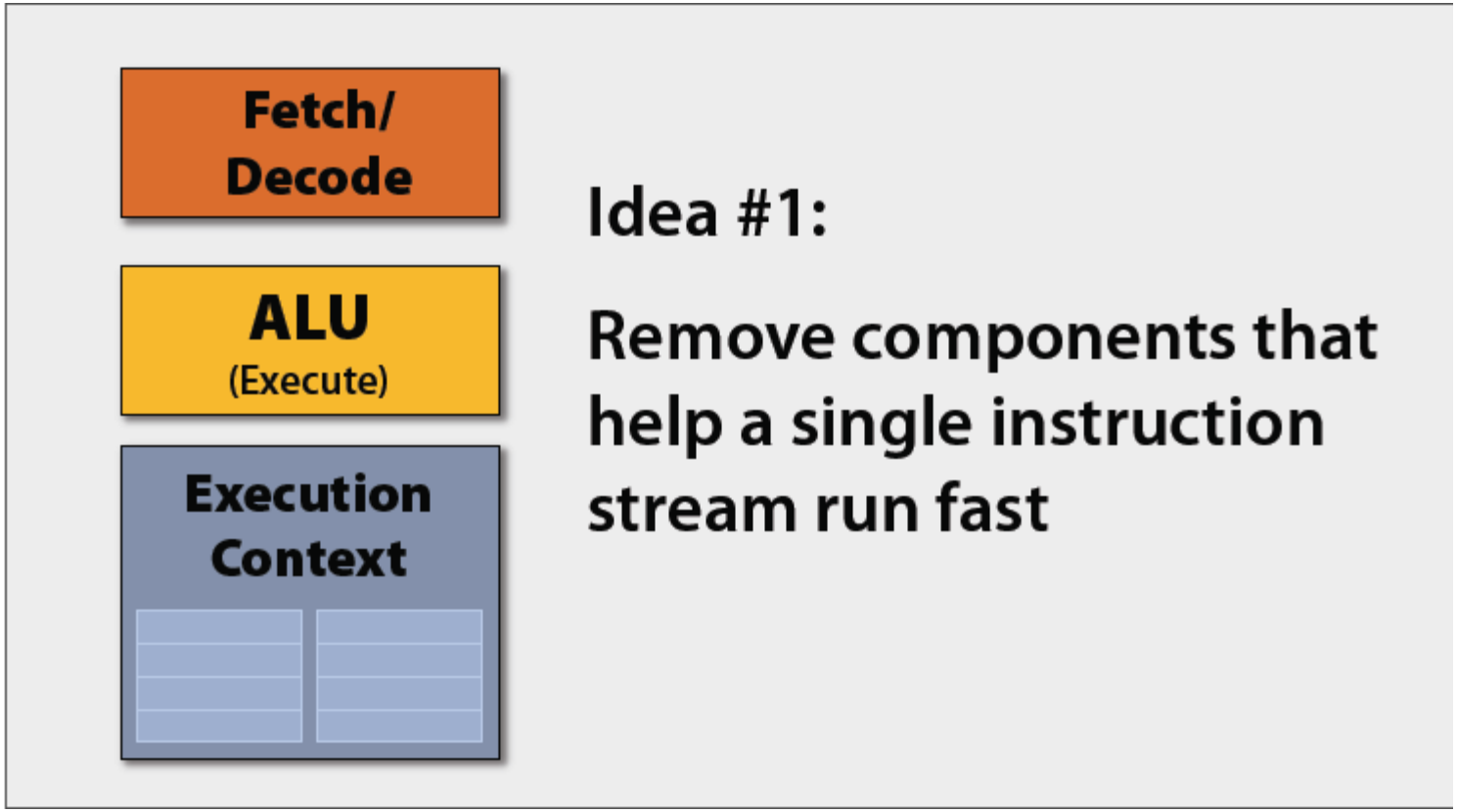
Parallel Hardware - SIMD

- In the vector addition example, a SIMD unit with a width of four could execute four iterations of the loop at once
- All current GPUs are based on SIMD hardware
  - The GPU hardware implicitly maps each SPMD thread to a SIMD “core”
    - The programmer does not need to consider the SIMD hardware for correctness, just for performance

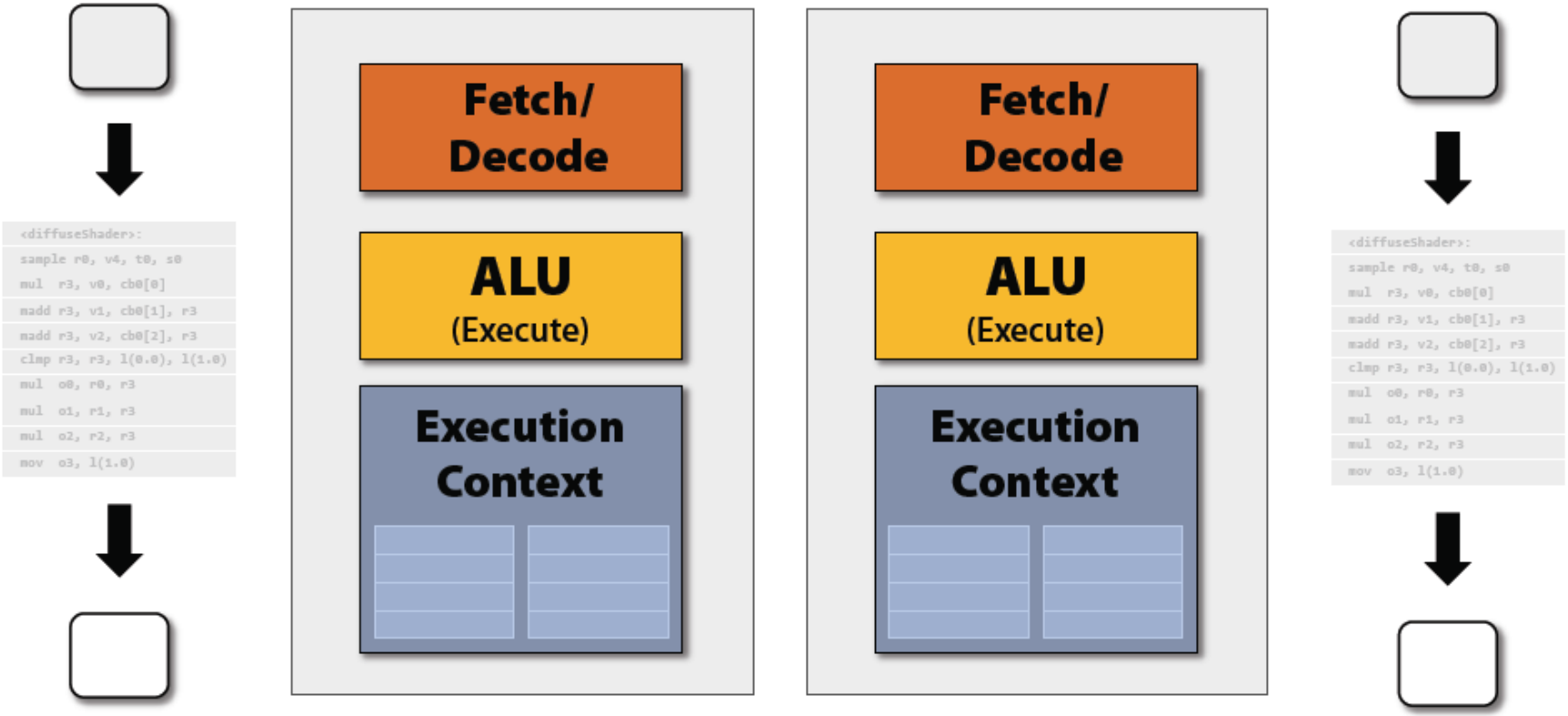
CPU Style cores



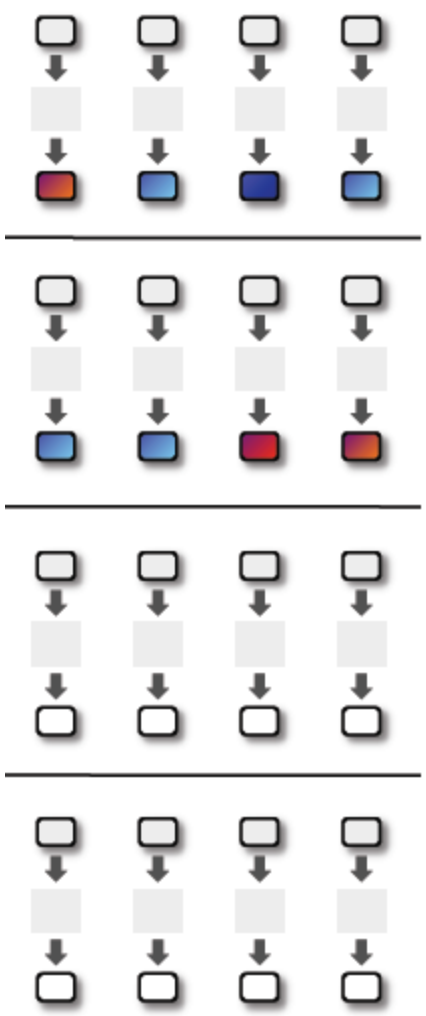
Slimming down

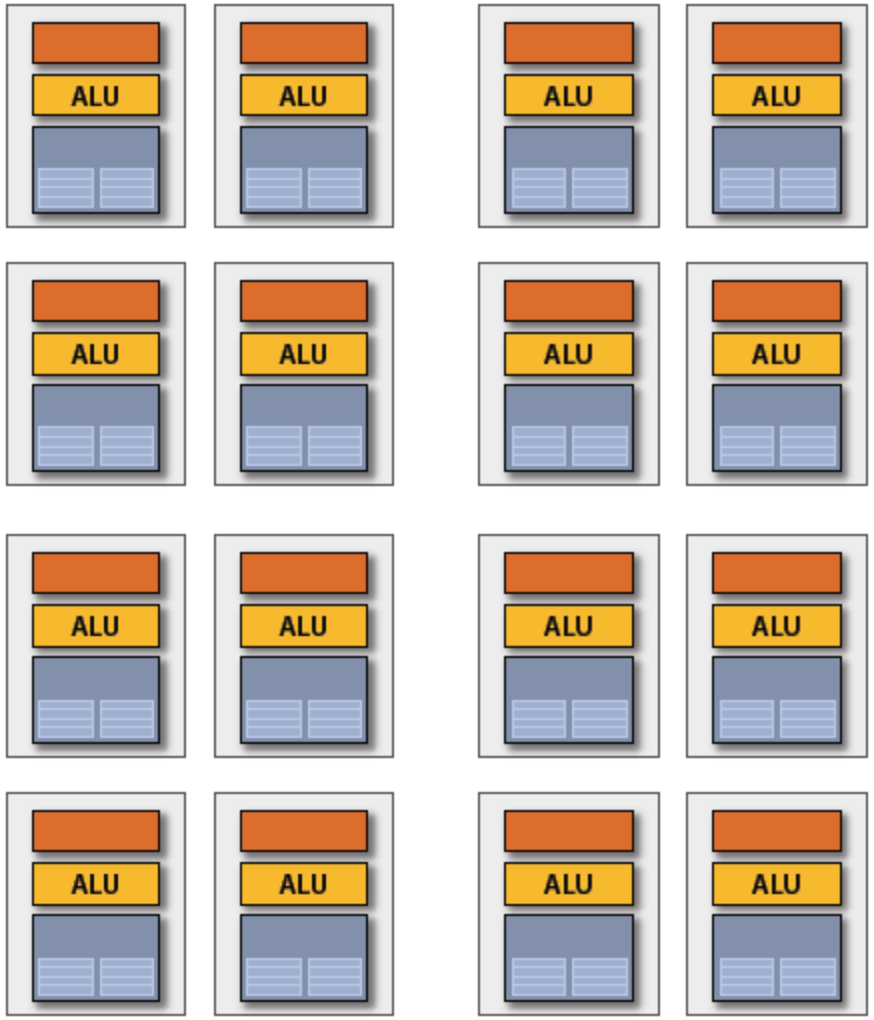


Add parallelism – 2 cores

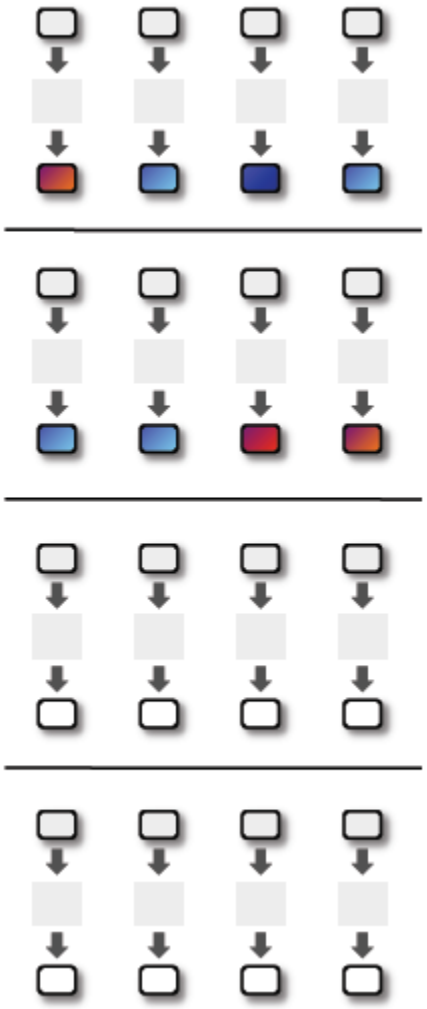


More parallelism: 16 cores



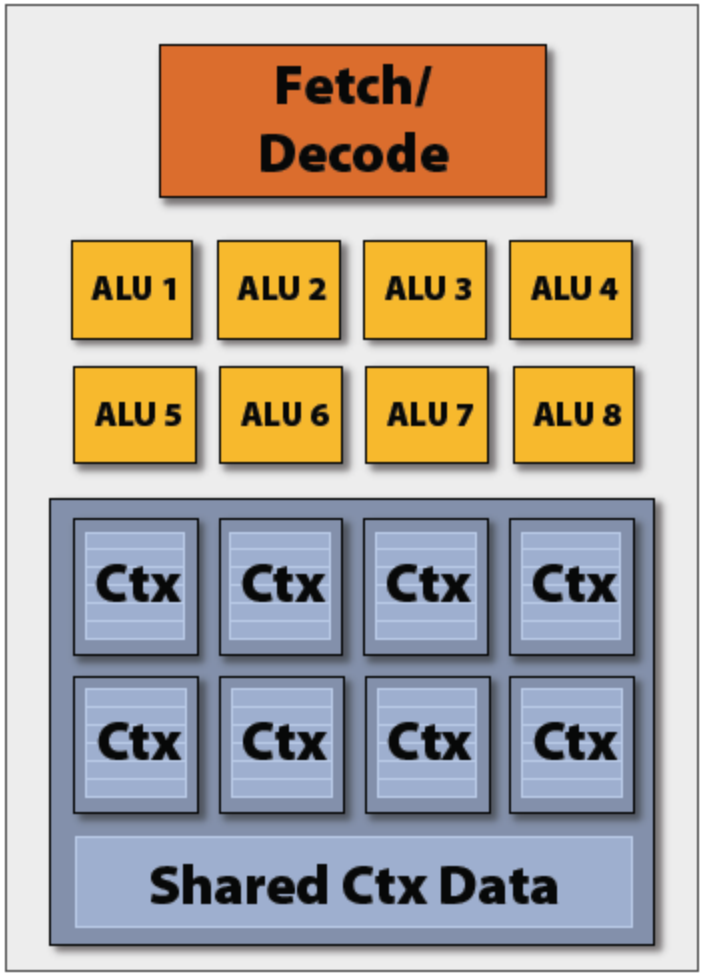


Instruction stream sharing



- But …many threads *should* be able to share an instruction stream

More sharing

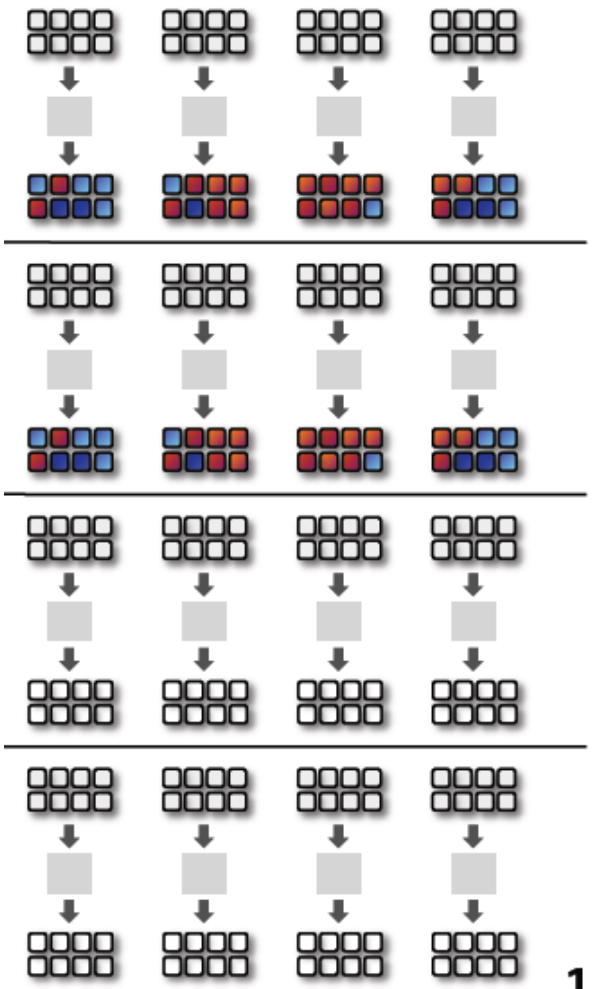


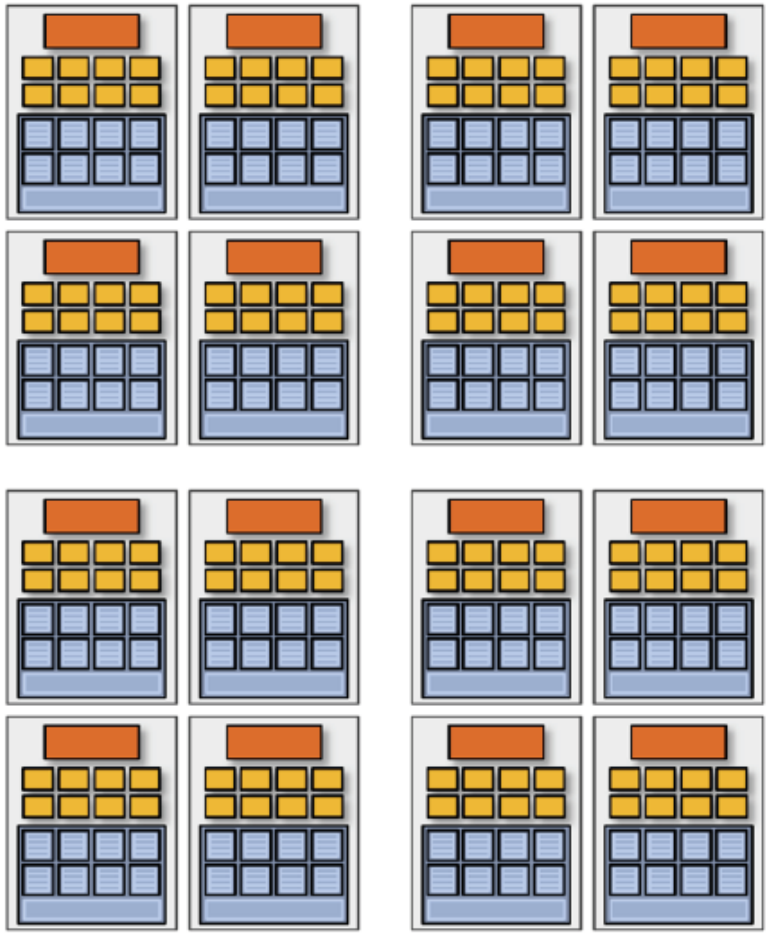
Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

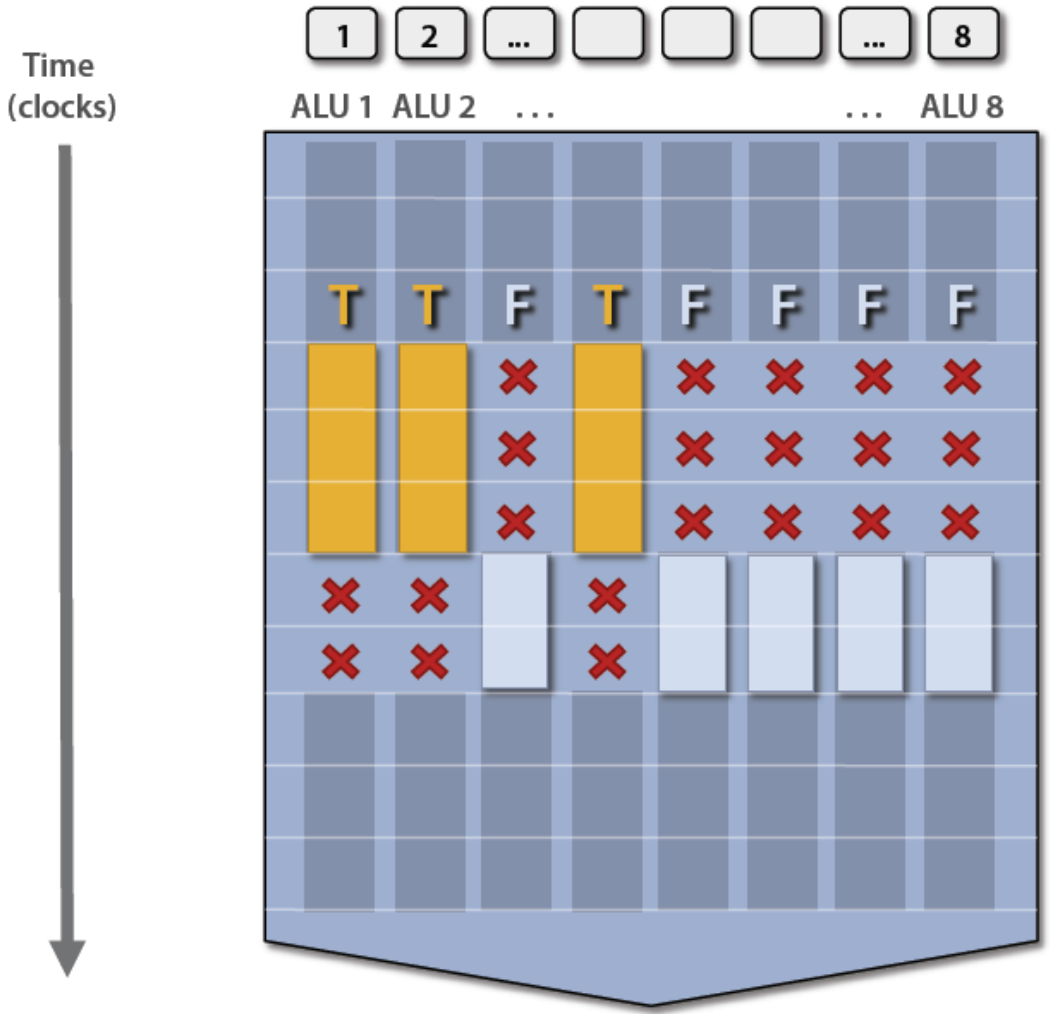
128 threads in parallel





16 cores = 128 ALUs
= 16 simultaneous instruction streams

But what about branches?




```
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

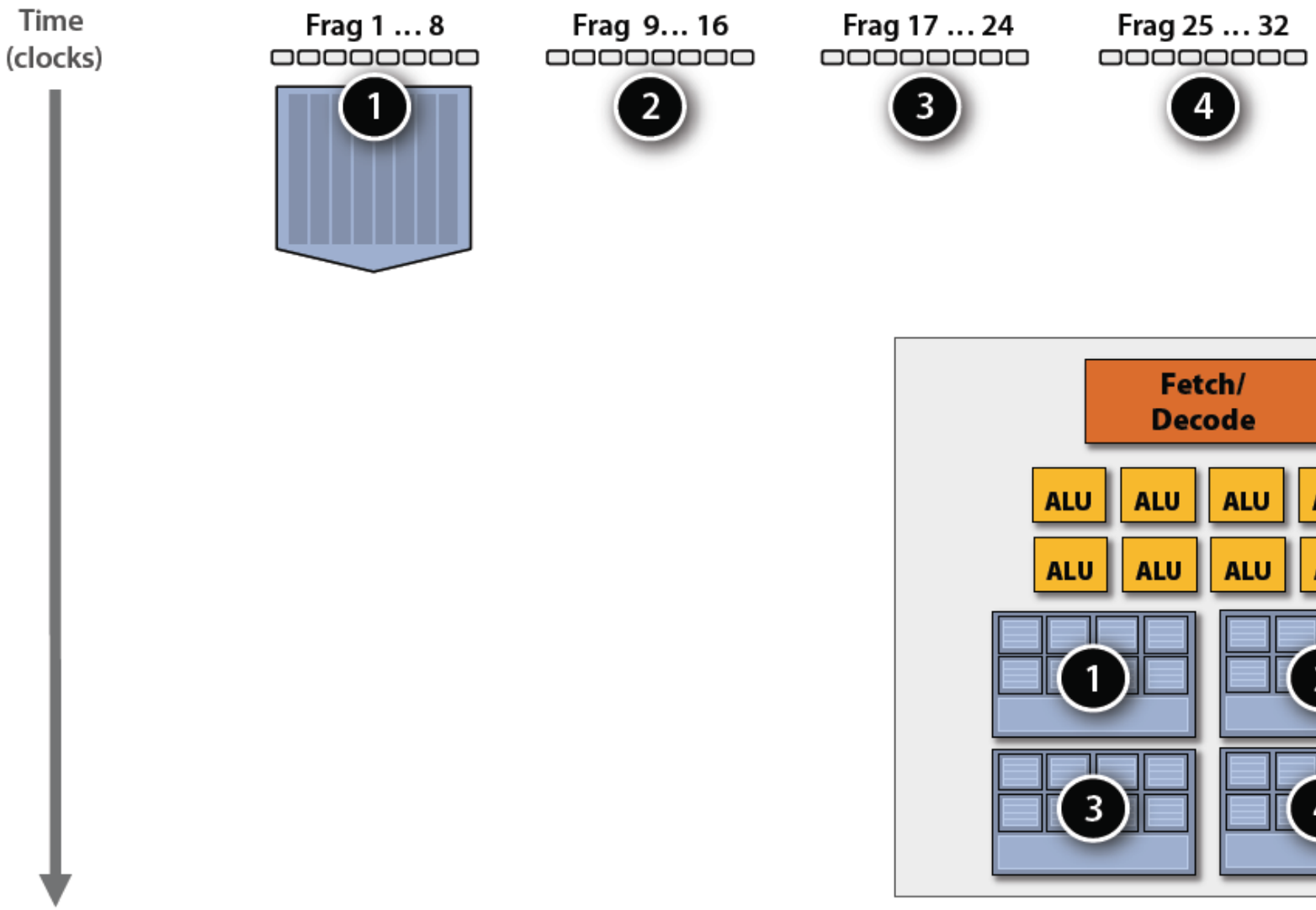
Clarification

- SIMD processing does not imply SIMD instructions
  - Option 1: Explicit vector instructions
    - Intel/AMD x86 SSE, Intel Larrabee
  - Option 2: Scalar instructions, implicit HW vectorization
    - HW determines instruction stream sharing across ALUs
    - nVidia GeForce (SIMT warps), AMD Radeon

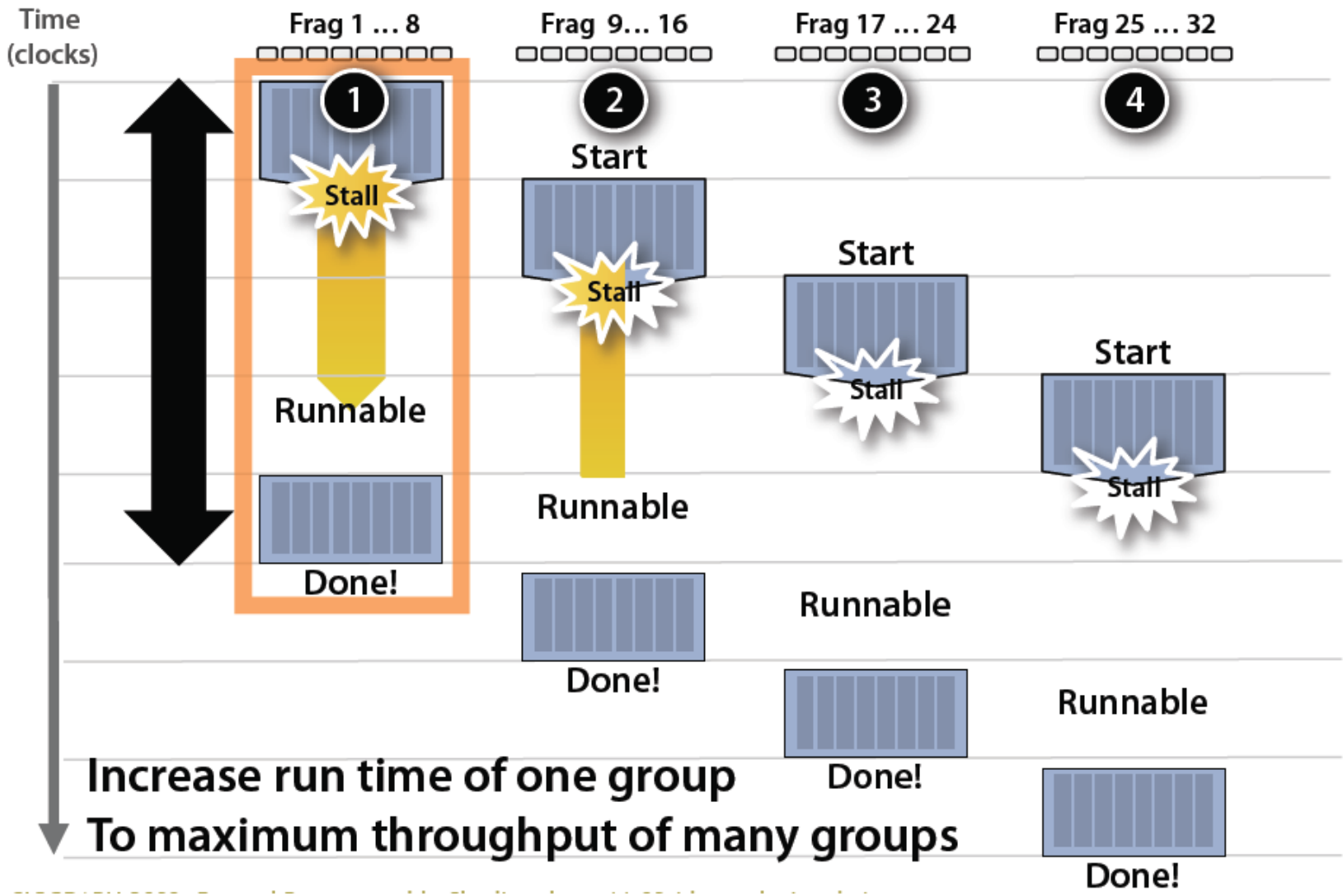
Stalls

- Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation
- The architecture has removed fancy caches and logic that helps avoid stalls
- BUT..there are LOTS of independent operations
- Idea #3: Interleave processing of many data on a single core to avoid stalls caused by high latency operations

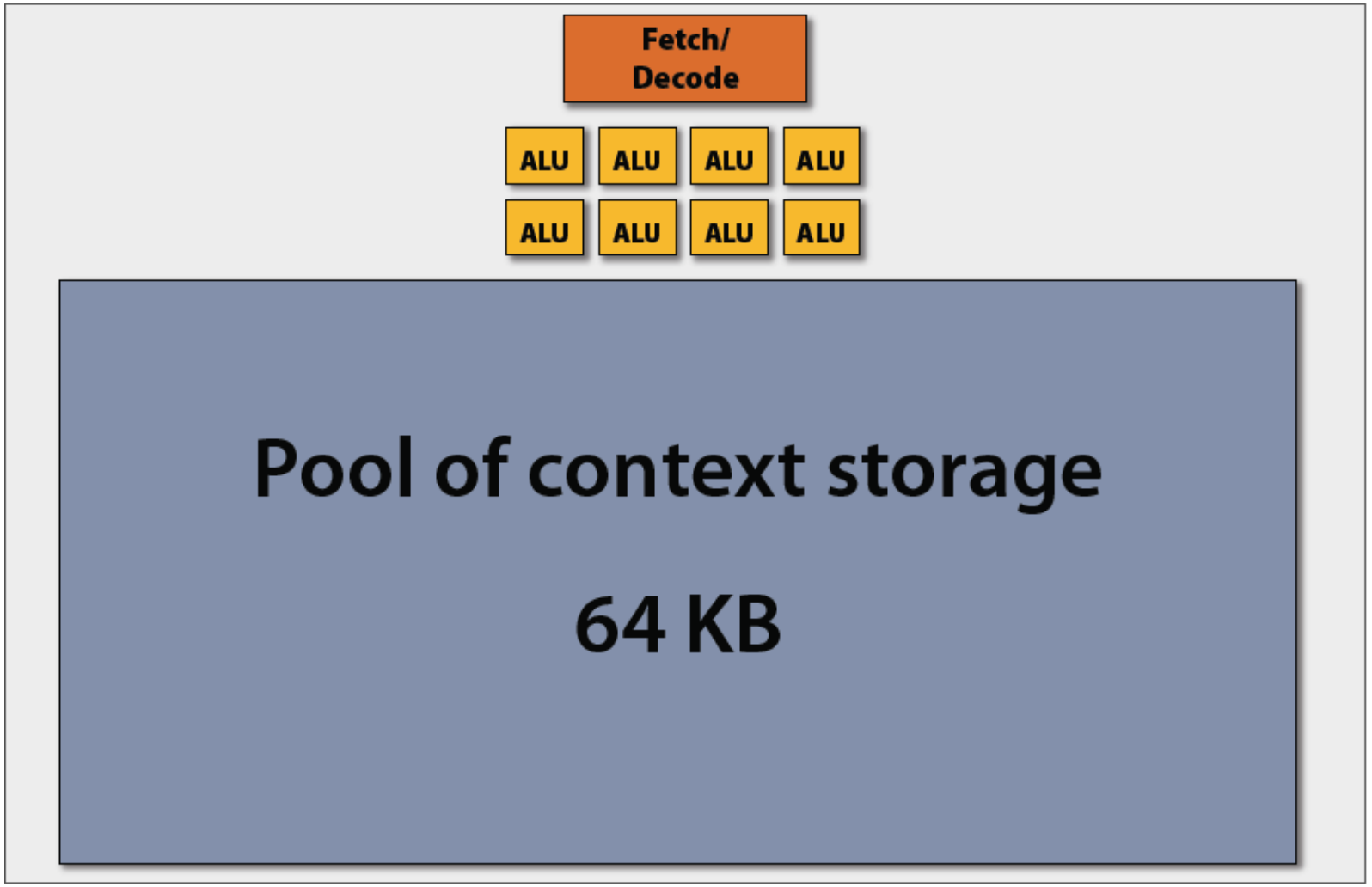
Hiding Stalls



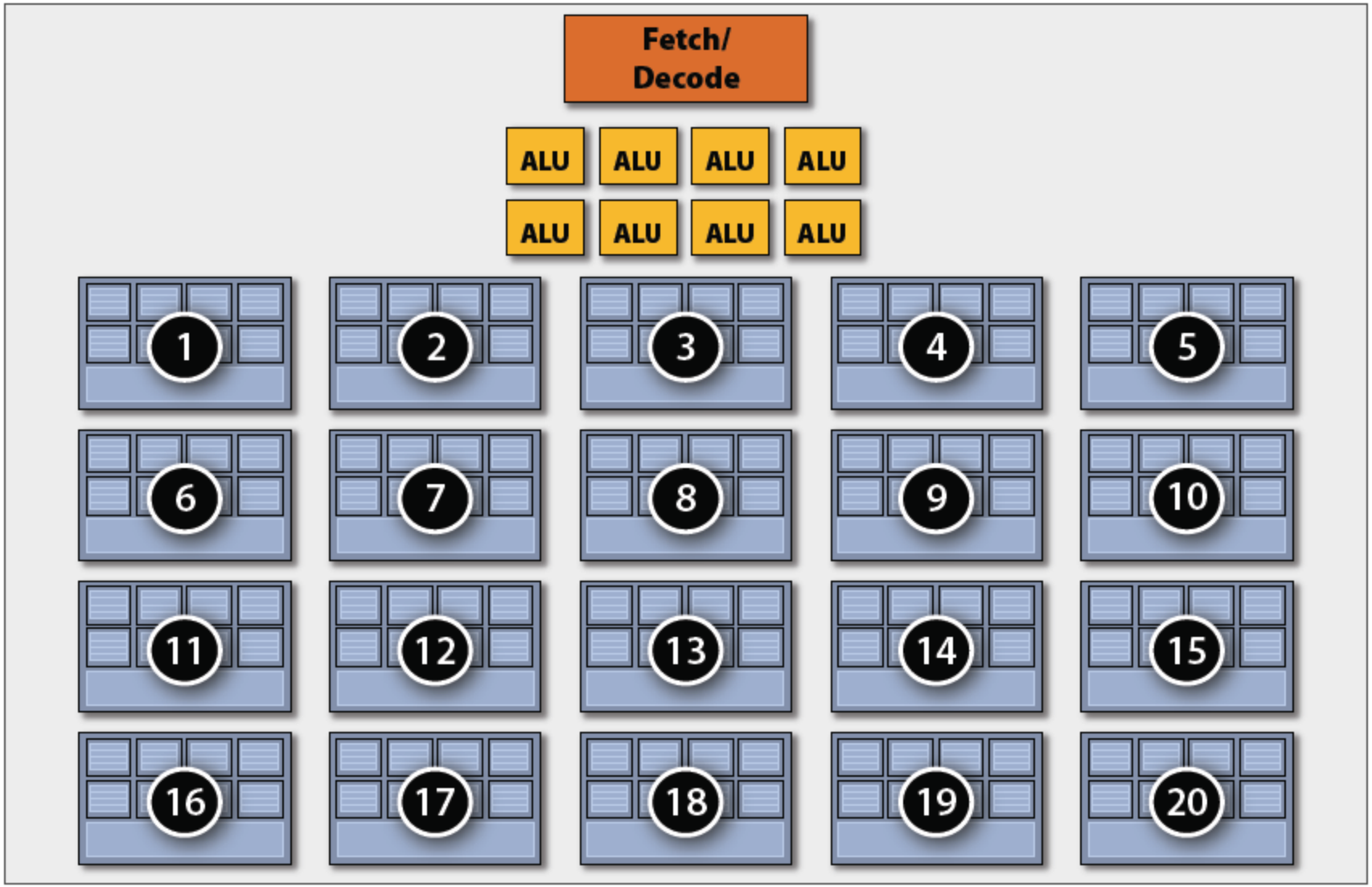
Hiding Stalls: Throughput



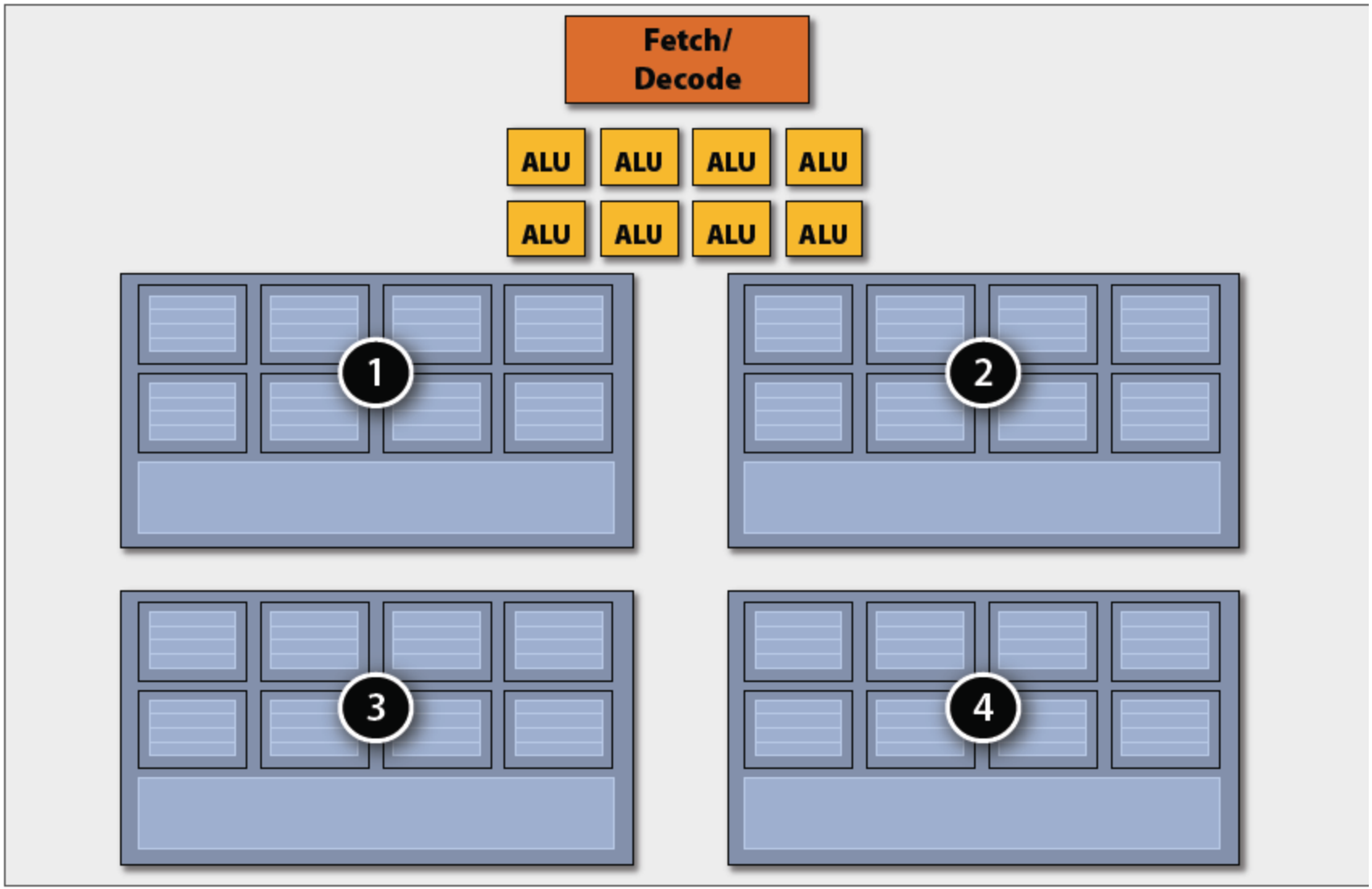
Storing contexts



Small contexts:
Maximal latency hiding capability



Large context:
Low latency hiding capability



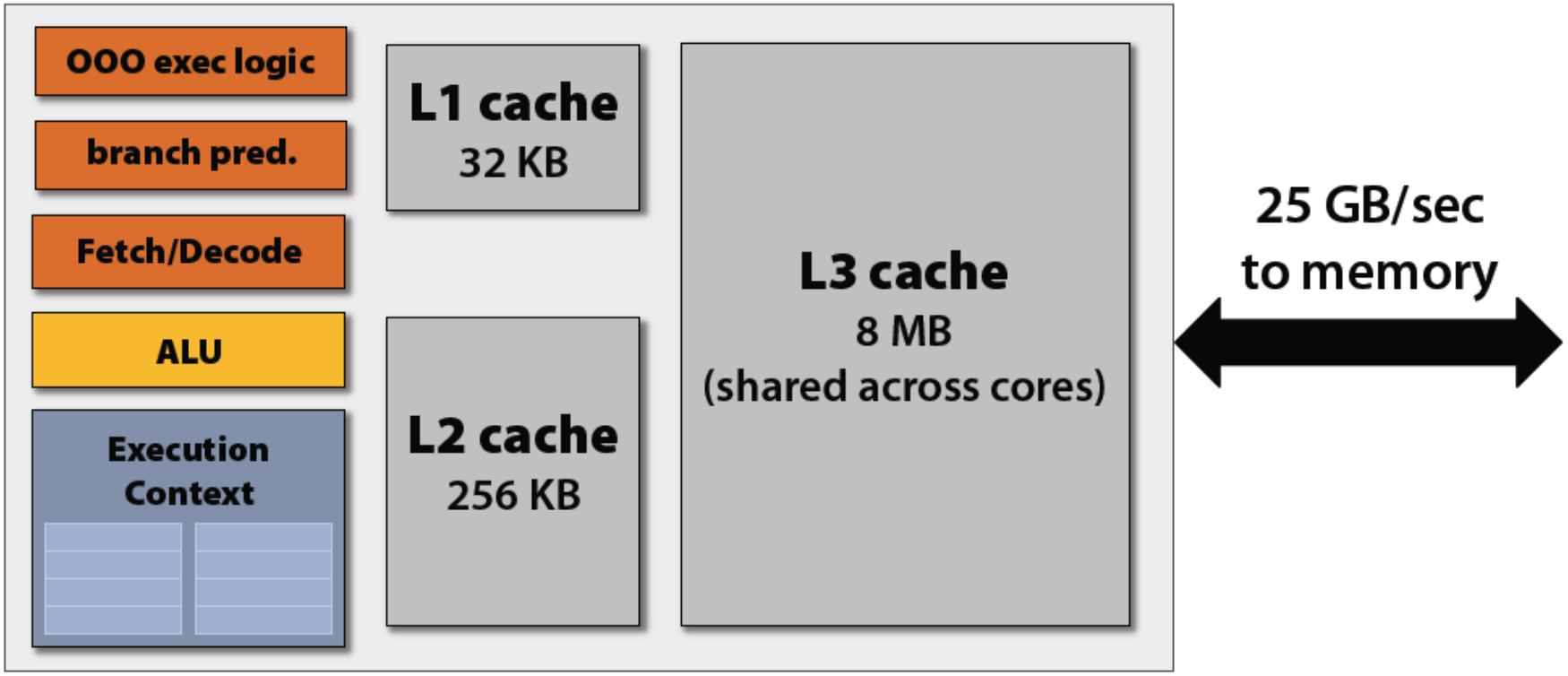
Summary: 3 Key Ideas

1. Use many "slimmed down cores" to run in parallel

2. Packs cores full of ALUs (by sharing instruction streams across threads)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by HW

3. Avoid latency stalls by interleaving execution of many groups of threads
   - When one group stalls, work on another

Moving data to processor



Complex memory hierarchy

Throughput core

More ALUs, no traditional cache hierarchy
Need high-bandwidth connection to memory

Bandwidth is critical

- On a high-end GPU
  - 11x compute performance of high-end CPU
  - 6x bandwidth to feed it
  - No complicated cache hierarchy

- GPU memory system is designed for throughput
  - Wide bus (150 GB/sec)
  - Repack/reorder/interleave memory requests to maximize use of memory bus

Bandwidth limited!

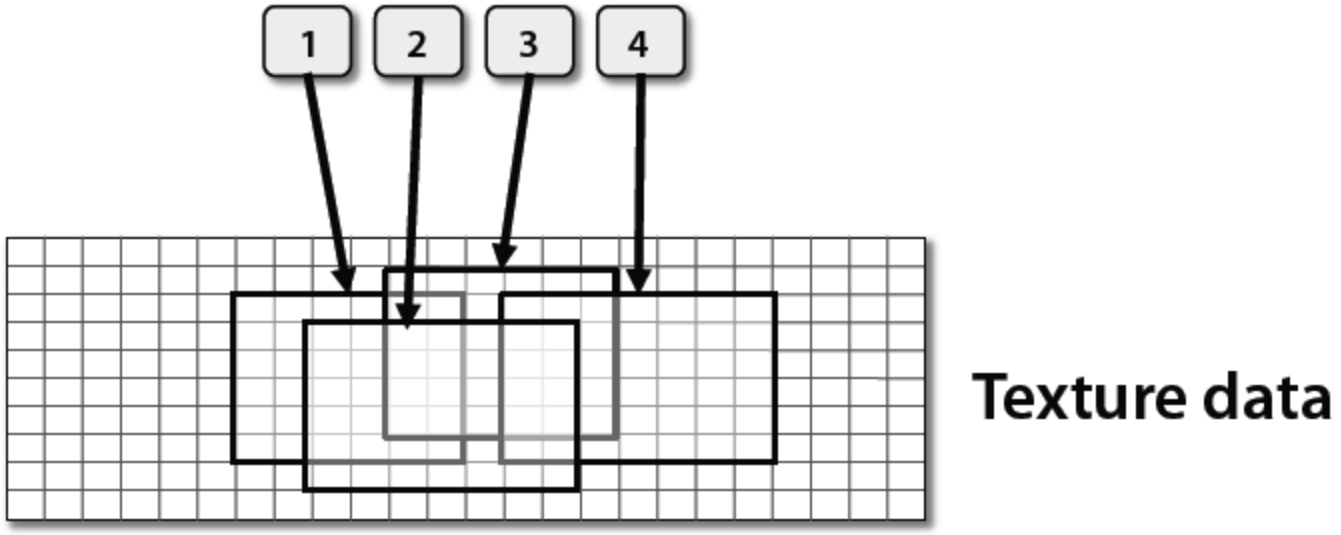
- Element-wise multiply 2 long vectors A and B
  - Load input A[i]
  - Load input B[i]
  - Multiply
  - Store result C[i]
- 3 memory operations every 4 cycles (12 bytes)
- Need ~1TB/sec bandwidth on high-end GPU
- 7x of available bandwidth

Bandwidth limited!

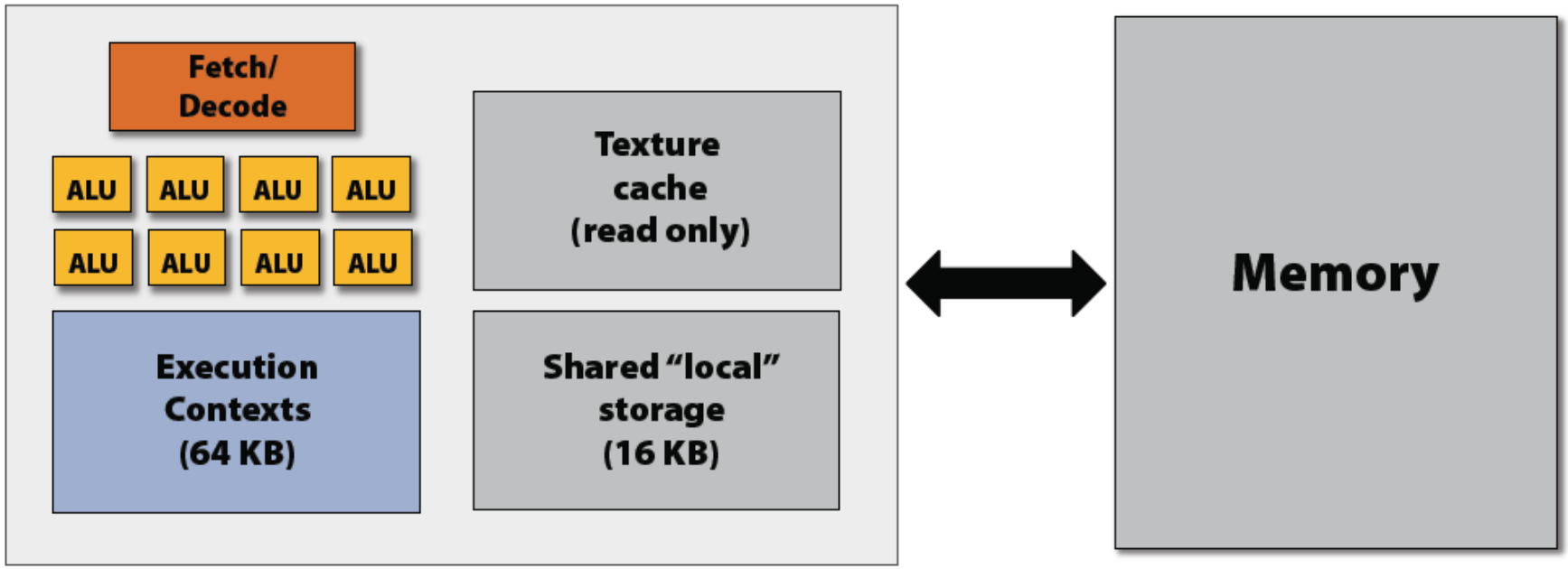
- If processors request data at too high a rate, the memory system cannot keep up
- No amount of latency hiding can help
- Reduce required bandwidth
  - Request data less often (do more math)
  - Share/reuse data across threads: increase on-chip storage

Reducing required bandwidth

- On-chip storage
  - Texture cache

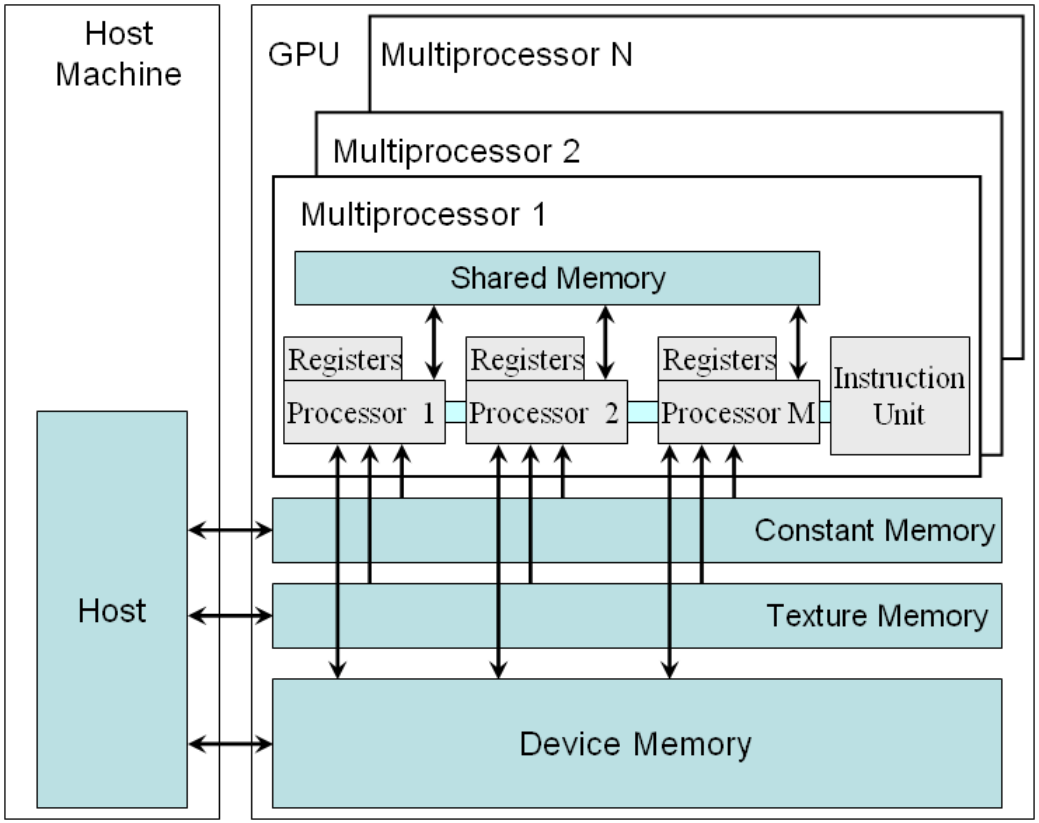


GPU memory hierarchy



GPU Memory Hierarchy

- **Four memories.**
  - **Device (a.k.a. global)**
    - ***slow*** **– 400-600 cycles access latency**
    - ***large*** **– 256MB – 1GB**
  - **Shared**
    - ***fast*** **– 4 cycles access latency**
    - ***small*** **– 16KB**
  - **Texture – read only**
  - **Constant – read only**



An efficient GPU workload..

- GPU is a multi-core optimized for maximum throughput
- Workload should have thousands of independent threads
  - Uses many ALUs on many cores
  - Supports massive interleaving for latency hiding
- Is amenable to instruction steam sharing
  - Maps to SIMD execution well
- Is compute-heavy: the ratio of math operations to memory access is high
  - Not limited by bandwidth

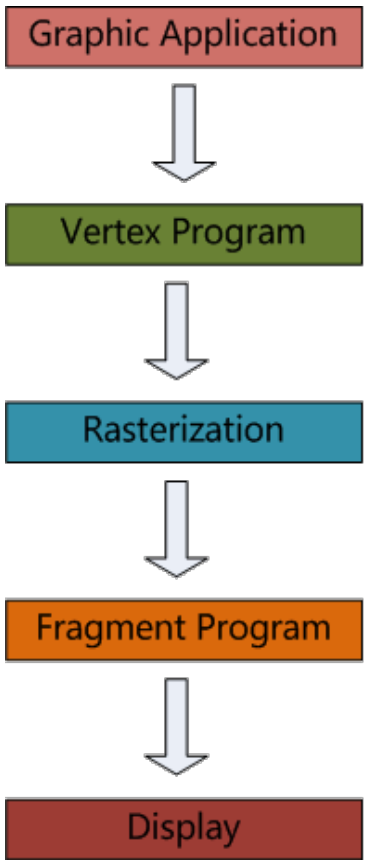
GPU Computing

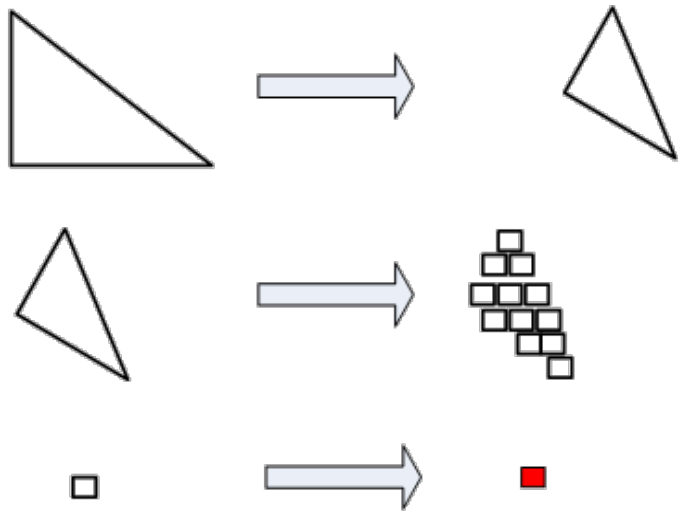
- GPU is a massively parallel processor
  - Supports thousands on active threads
- GPU computing requires a programming model that can efficiently express that kind of parallelism
  - Most importantly, data parallelism
- CUDA implements such a programming model

CUDA

- Compute Unified Device Architecture
- A powerful parallel programming model for issuing and managing computations on the GPU without mapping them to a graphics API
  - Heterogeneous - mixed serial-parallel programming
  - Scalable - hierarchical thread execution model
  - Accessible - minimal but expressive changes to C

GPU programming model





GPGPU programming model

- Trick the GPU into general-purpose computing by casting problem as graphics
  - Turn data into images ("texture maps")
  - Turn algorithms into image synthesis ("rending passes")
- Drawback:
  - Tough learning curve
  - potentially high overhead of graphics API
  - highly constrained memory layout & access model
  - Need for many passes drives up bandwidth consumption

CUDA Programming Model

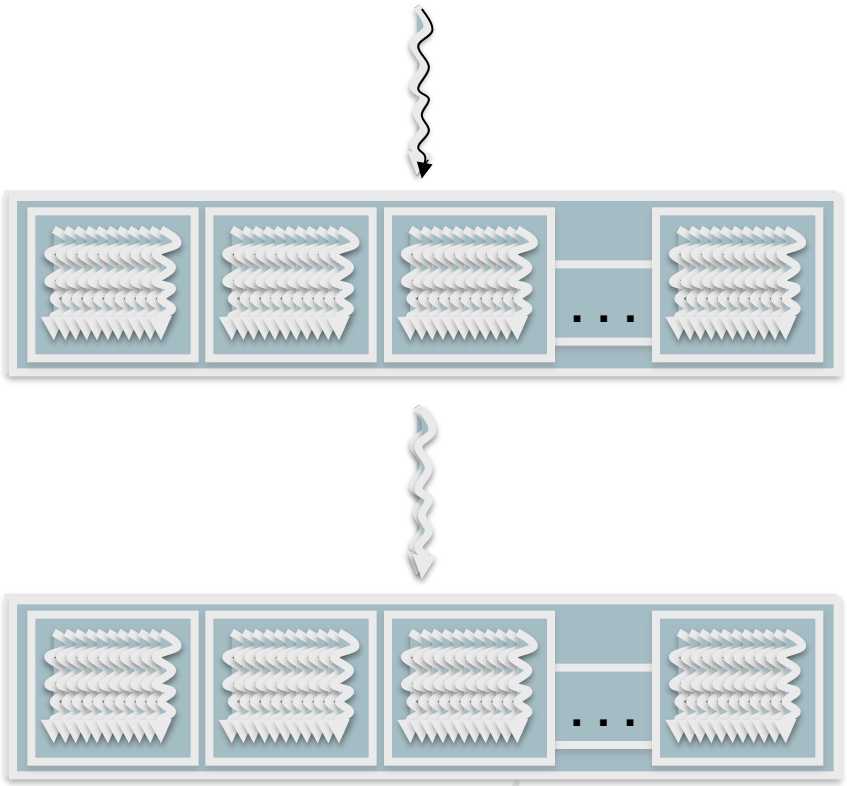
- Program the GPU in C
- Scalable data parallel execution model
- C with minimal yet powerful extensions

Heterogeneous programming model

- Integrated host + device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);



Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

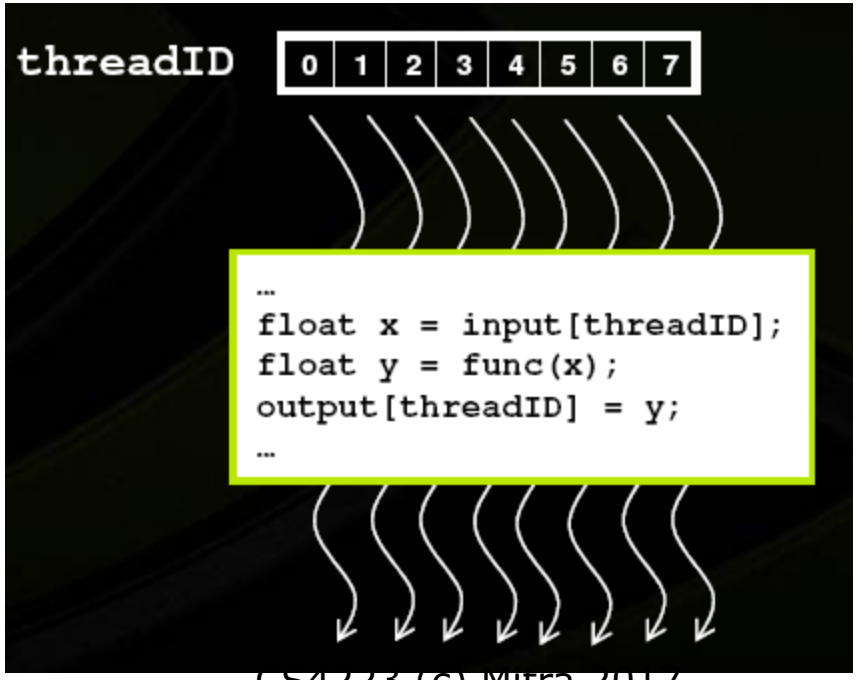
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel

- Difference between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can only use a few

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

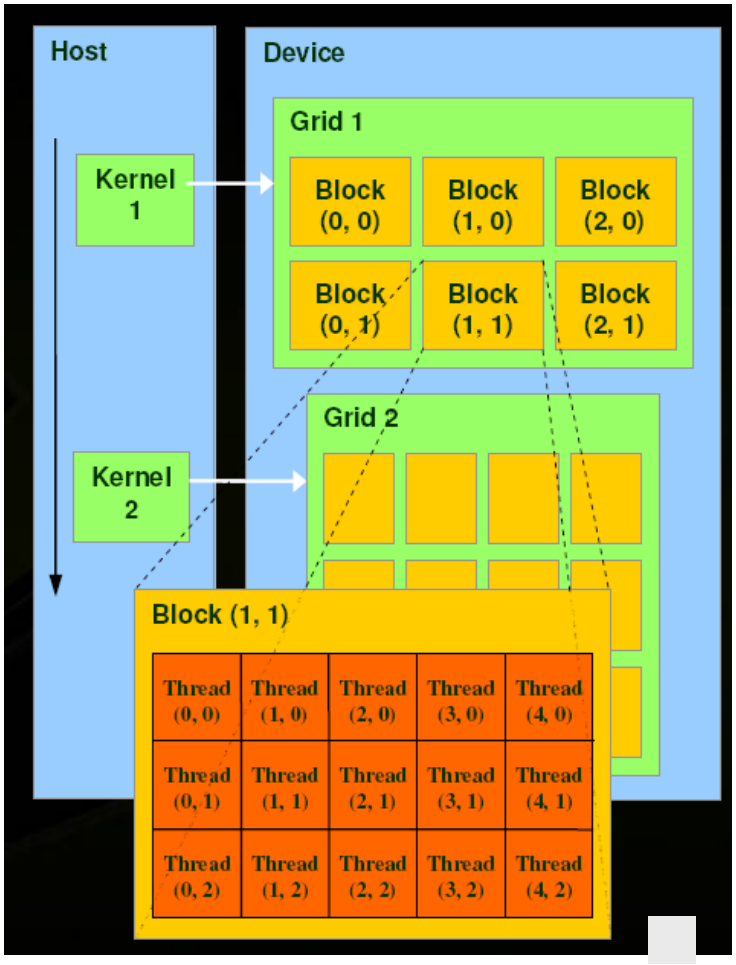


Thread Cooperation

- The missing piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to save computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA

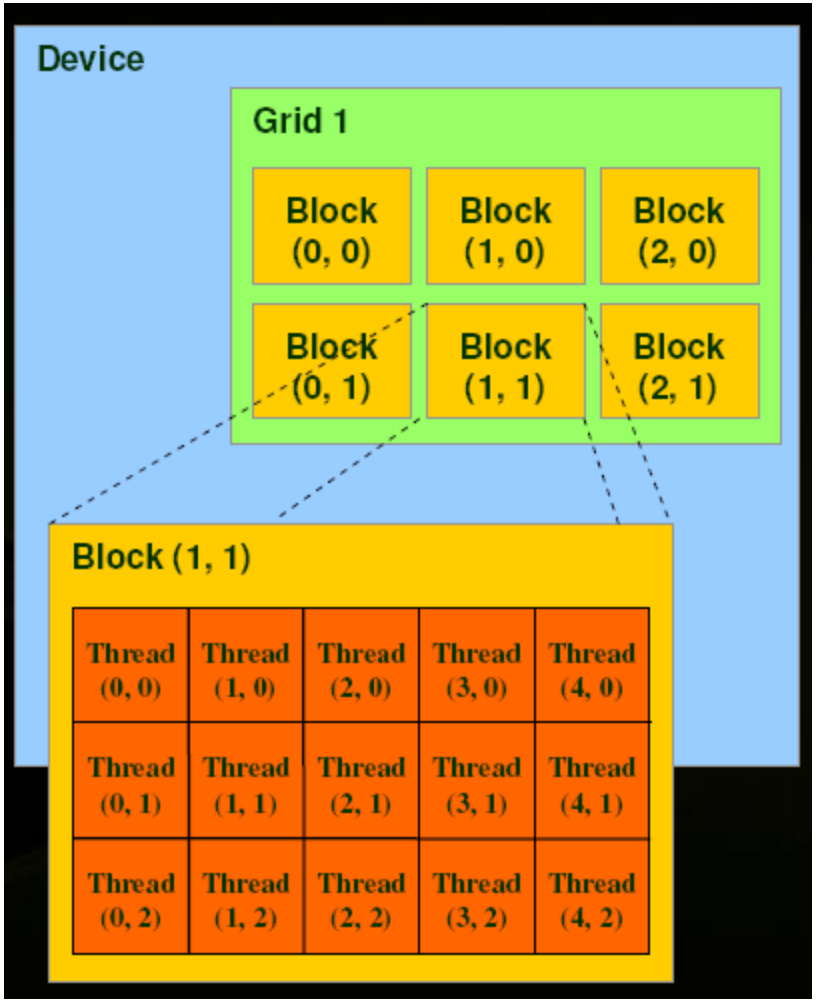
CUDA programming model

- A kernel is executed by a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



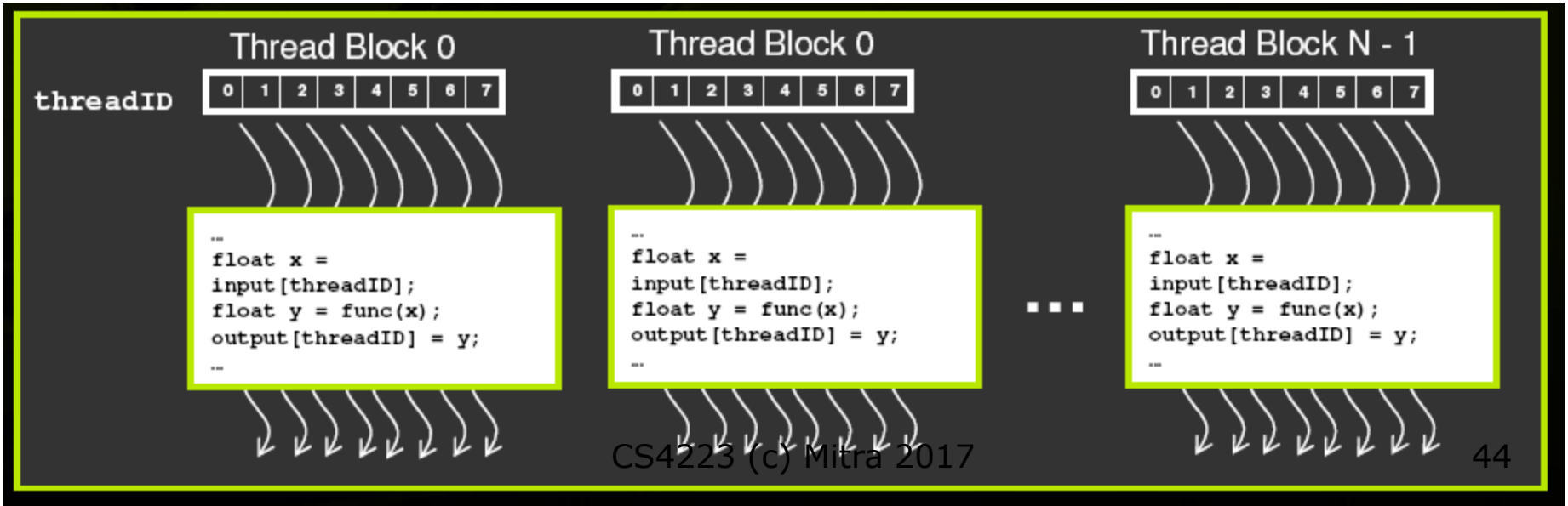
Thread and Block IDs

- Thread and blocks have IDs
  - So each thread can decide what data to work on
- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multi-dimensional data



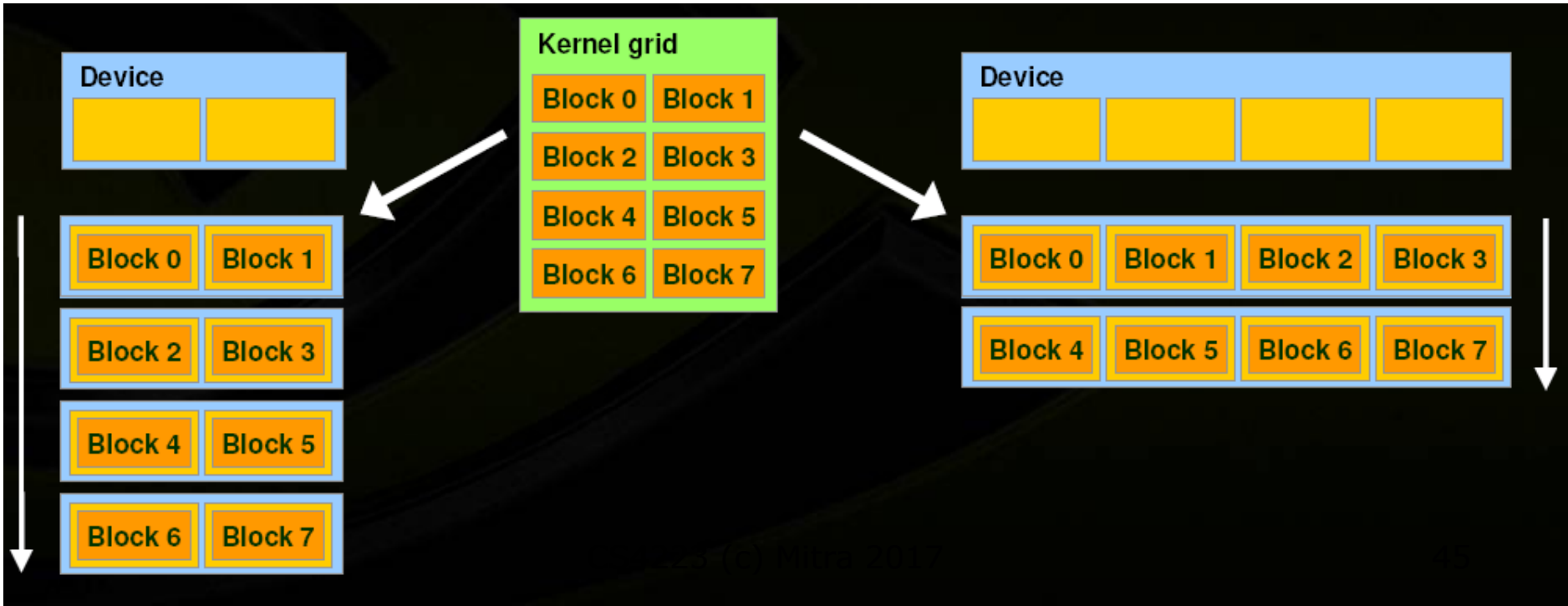
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory
  - Threads in different blocks cannot cooperate
- Enables programs to transparently scale to any number of processors



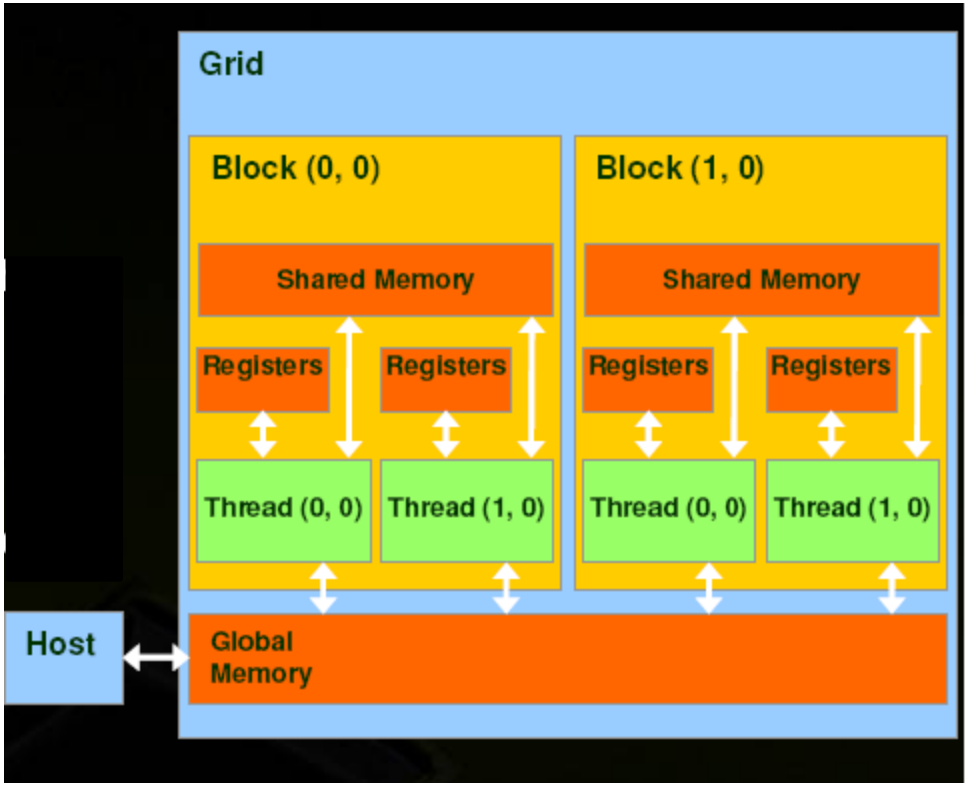
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor at any time
  - A kernel scales across any number of parallel multiprocessors



Kernel Memory Access

- Registers
- Global memory
  - Kernel input and output data reside here
  - Off-chip, large
  - Un-cached
- Shared memory
  - Shared among threads in a single block
  - On-chip, small
  - As fast as registers
- The host can read and write global memory but not shared memory



Execution Model

- Kernels are launched in grids
  - One kernel execute at a time
- A block executes on one multiprocessor
  - Does not migrate
- Several blocks can reside concurrently on one multiprocessor
  - Control limitations (of G8X/G9X GPUs)
    - At most 8 concurrent blocks per SM
    - At most 768 concurrent threads per SM
  - Number is further limited by SM resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks

CUDA Model Summary

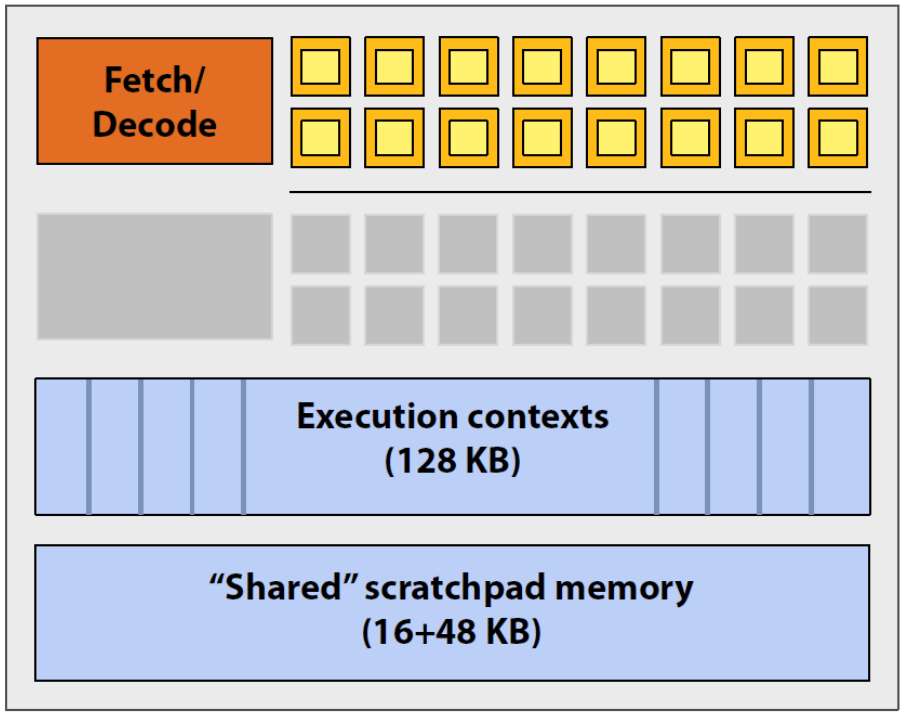
- Thousands of lightweight concurrent threads
  - No switching overhead
  - Hide instruction and memory latency
- Shared memory
  - User-managed L1 cache
  - Thread communication/cooperation within blocks
- Random access to global memory
  - Any thread can read/write any location(s)

Memory	Location	Cached	Access	Scope ("Who?")
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host

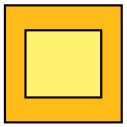
Putting Ideas Into Practice

- Nvidia GeForce GTX 480 (Fermi)
  - Followed up by Kepler, Maxwell, Pascal
- Nvidia-speak
  - 480 stream processors (“CUDA cores”)
  - SIMT execution
- Generic-speak
  - 15 cores
  - 2 groups of 16 SIMD functional units per core

Nvidia GeForce GTX480 “core”



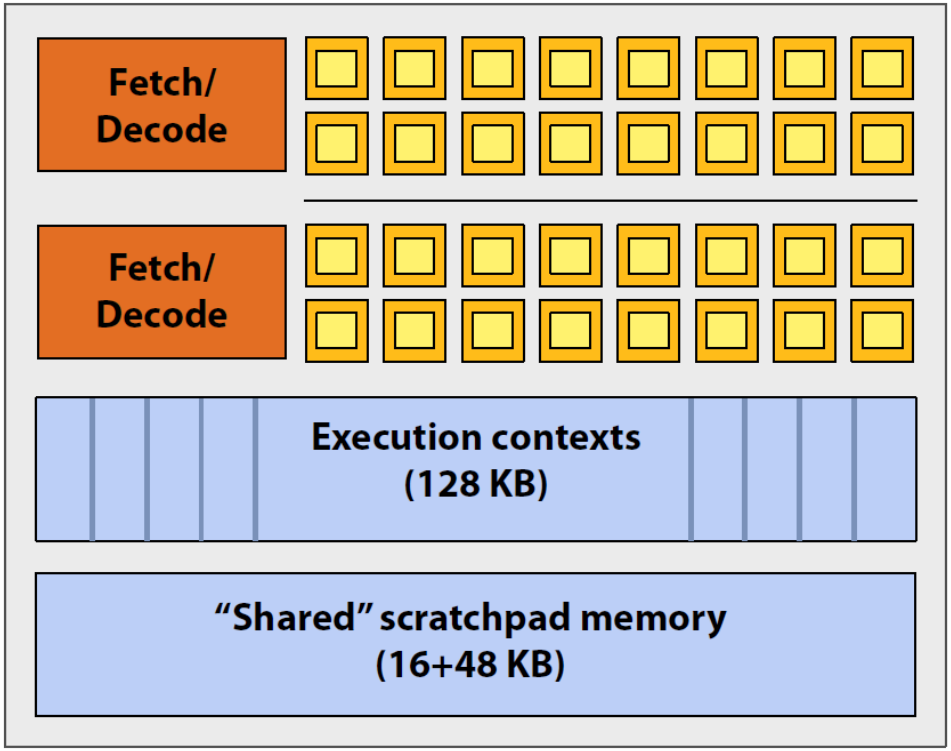
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G



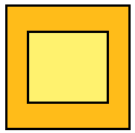
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- Groups of 32 fragments share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

Nvidia GeForce GTX480 "core"



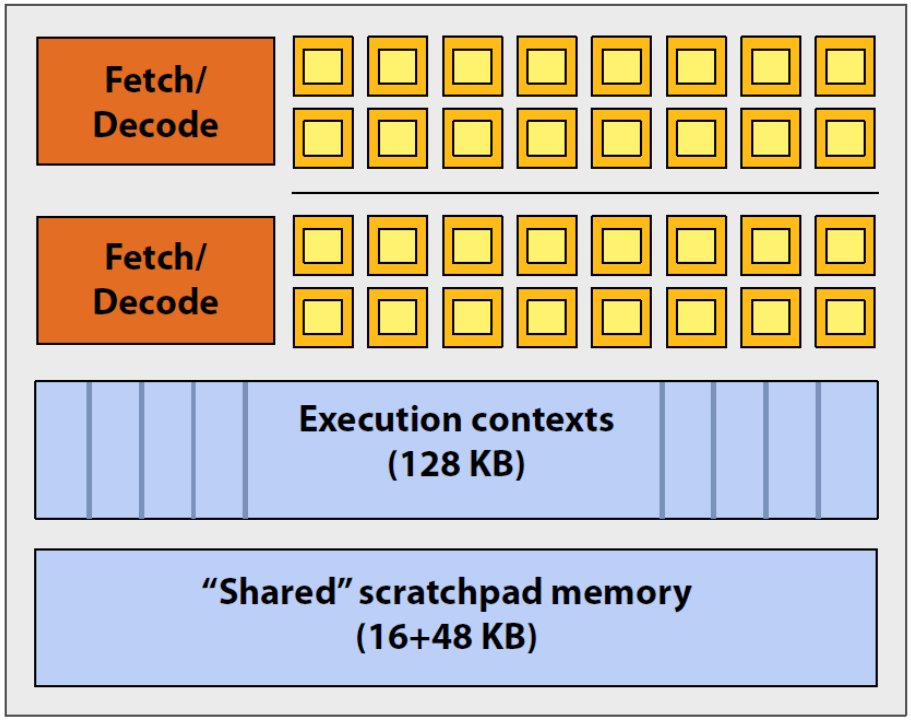
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G



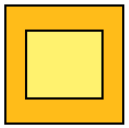
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

Nvidia GeForce GTX480 "SM"



Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

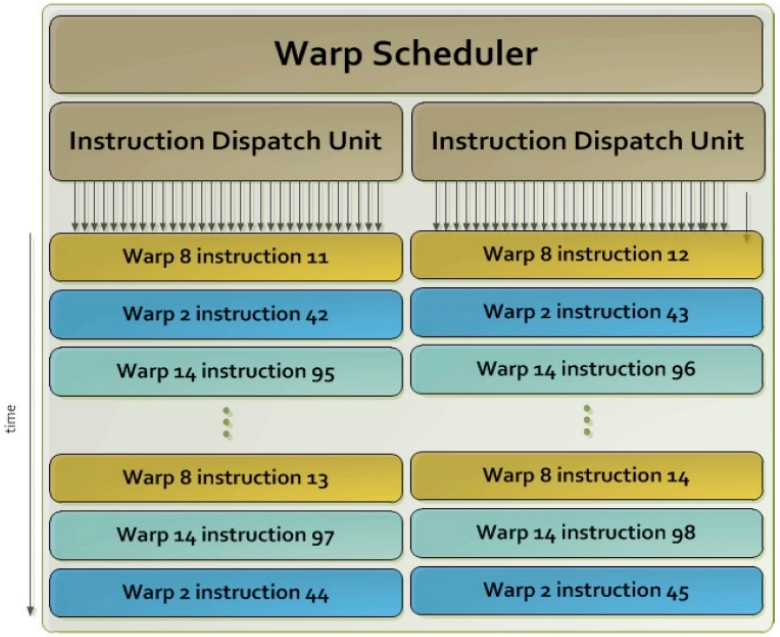


= CUDA core
(1 MUL-ADD per clock)

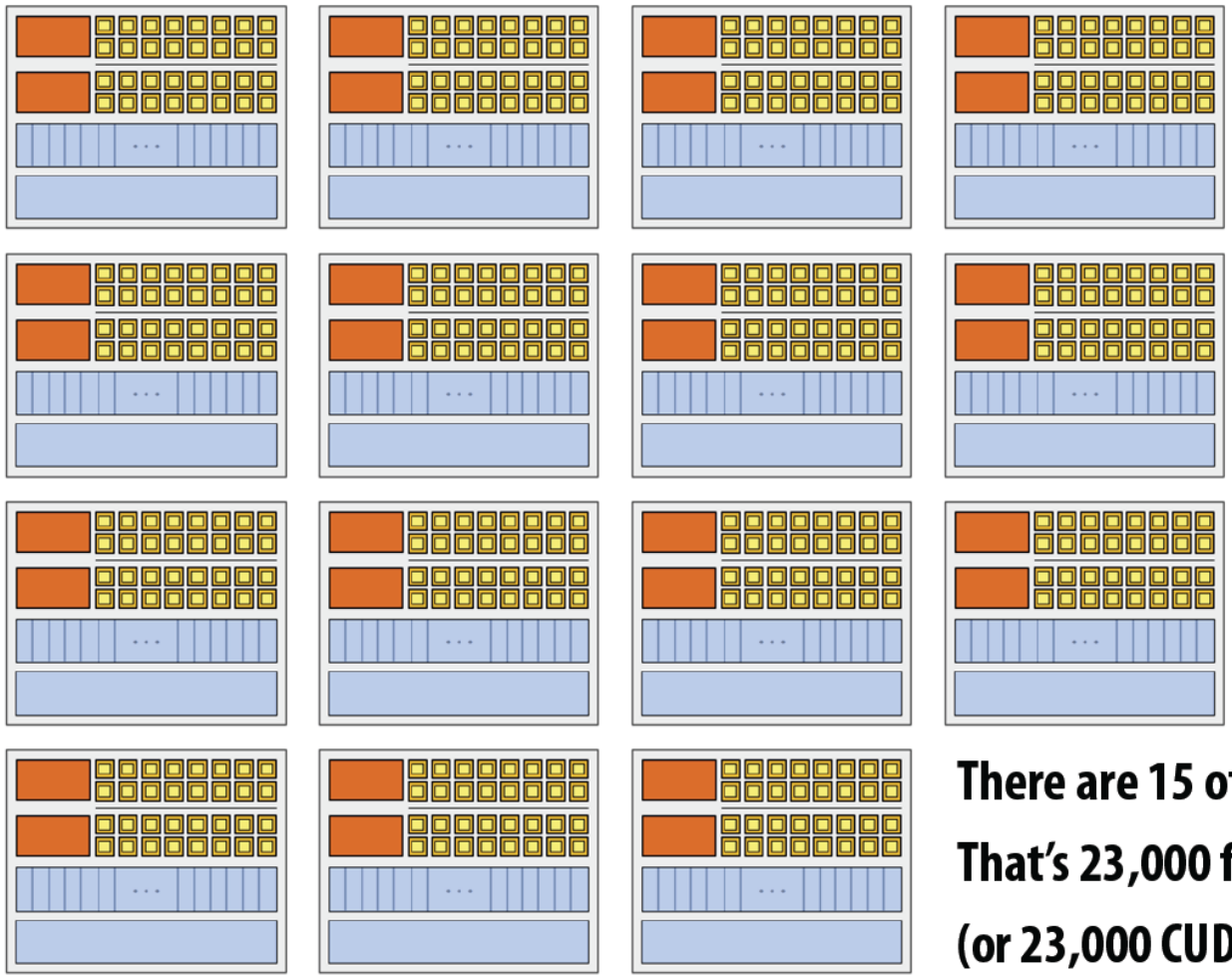
- The SM contains 32 CUDA cores
- Two warps are selected each clock (decode, fetch, and execute two warps in parallel)
- Up to 48 warps are interleaved, totaling 1536 CUDA threads

Warp Scheduler (Fermi)

- Scoreboard for long-latency operations
  - Keeps track of registers not yet ready with valid data
  - Dependency checker analyzes register usage of fully decoded warp instructions against scoreboard to determine warps eligible to issue
- Inter-warp scheduling decisions
- Thread-block level scheduling



Nvidia GeForce GTX480



There are 15 of these things on the GTX 480:

That's 23,000 fragments!

(or 23,000 CUDA threads!)

Looking Forward

- Bigger and Faster
  - more cores, more FLOPS
- Addition of (select) CPU-like features
  - More traditional caches
  - Stacked memory
- Tight integration with CPUs
  - Unified memory architecture
  - NVLink
- Support for alternative programming interfaces (OpenCL)

NVIDIA Volta GV100

- 21.1 billion transistors, 815 mm^2 die size, 12nm FinFET
- 7.8 TFlops double-precision, 125 TFlops tensor core throughput
- 300W TDP, 50% more energy-efficient vis-à-vis Pascal

NVIDIA Volta GV100

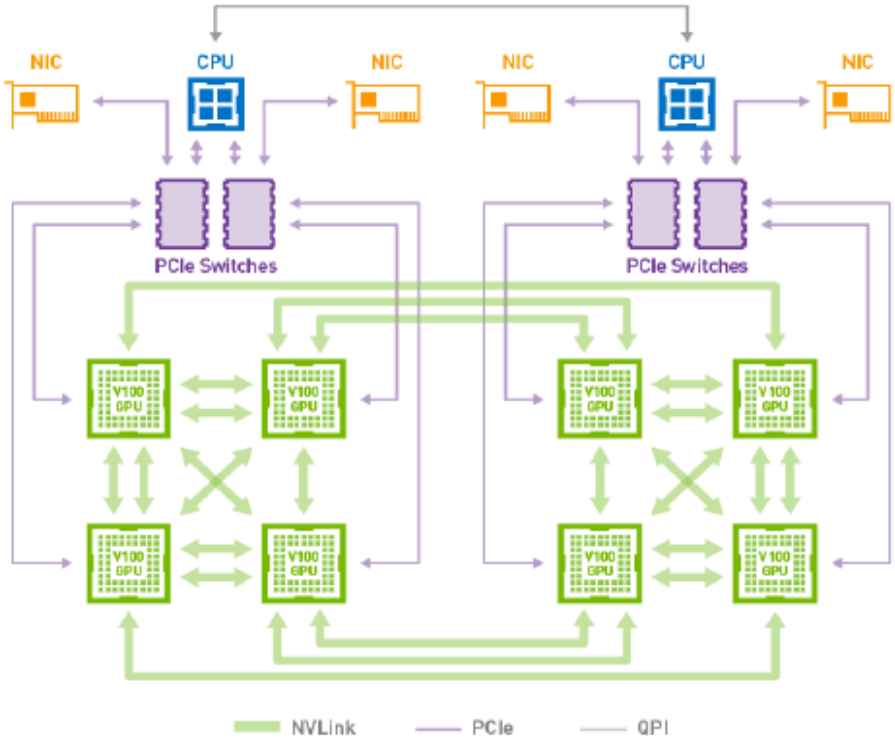
- Six GPU Processing Clusters:
  - Seven Texture Processing Cluster (TPC)
  - 14 Streaming Multiprocessors (SM)
- Total 6x14 = 84 Volta SMs
  - 64 FP32 cores
  - 64 INT32 cores
  - 32 FP64 cores
  - 8 Tensor Cores
  - 4 Texture units
  - 256KB register file
  - L0 instruction cache
  - 128KB L1 data cache/shared memory
- 6,144KB shared L2 cache
- Eight 512-bit memory controllers, 16GB memory, 900 GB/sec bandwidth

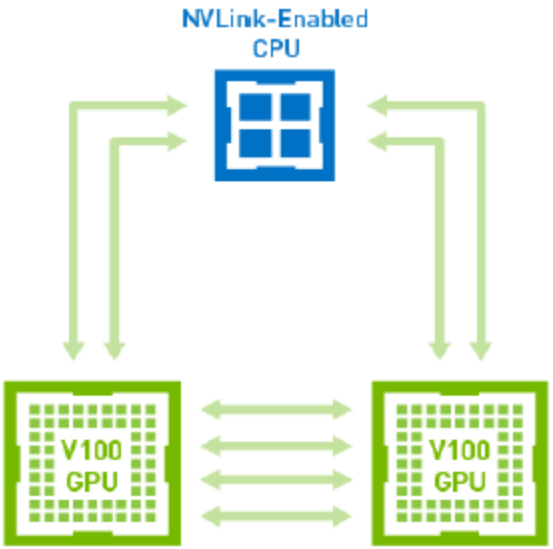
Volta SM


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

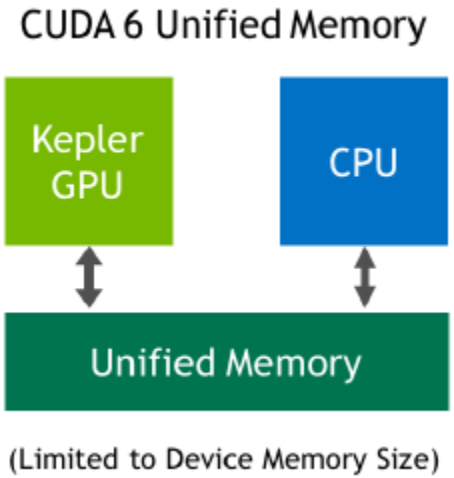
NVLink: Fast Connection

- Six NVLinks, 300GB/sec bandwidth





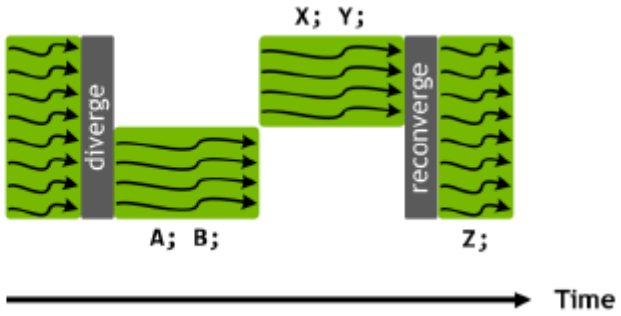
Unified Memory Architecture



NVIDIA SIMT Model

- Threads from the same warp in divergent regions cannot signal to each other or exchange data
- Can easily lead to deadlock
- GPU programmers avoid fine-grained synchronization and rely on lock-free algo

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

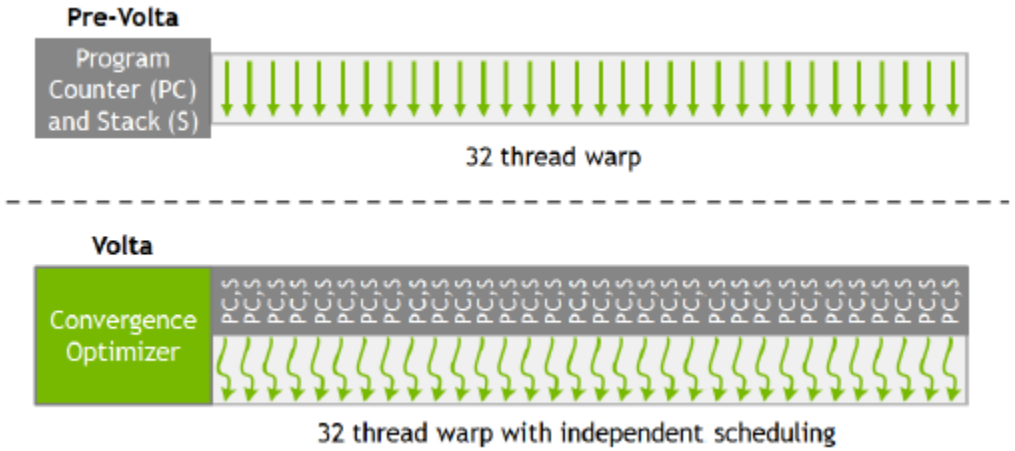


Deadlock Example

```
__global__ kernel() {
  __shared__ int semaphore;
  semaphore=0;
  __syncthreads();
  while (true) {
    int prev=atomicCAS(&semaphore,0,1);
    if (prev==0) {
      //critical section
      semaphore=0;
      break;
    }
  }
}
```

Volta SIMT Model

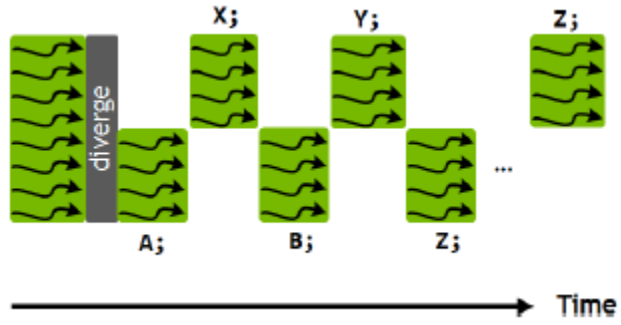
- Independent Thread Scheduling: Program Counter (PC) and call stack per thread



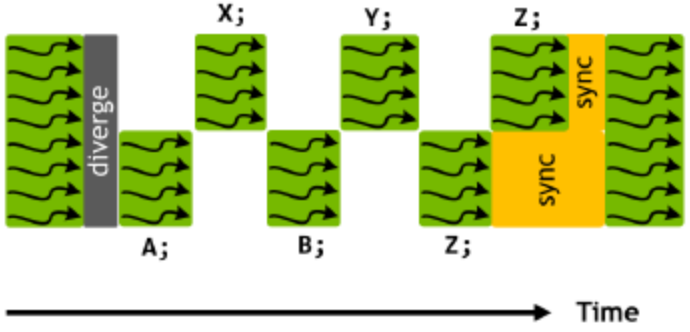
Volta Interleaved Execution

- Programs use explicit synchronization to re-converge threads in a warp

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

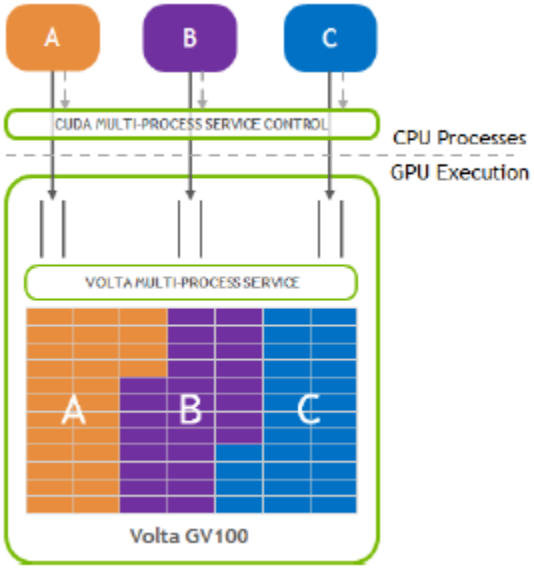



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



Volta Multi-Process Service

- Isolation for multiple compute applications sharing the GPU



A100 Optimizations

Accelerating Sparse Deep Neural Networks

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic,
Dusan Stosic, Ganesh Venkatesh, Chong Yu, Paulius Micikevicius

NVIDIA

{asitm, jalbericiola, jpool, darkos, dstosic, chongy, pauliusm}
@nvidia.com

Abstract. As neural network model sizes have dramatically increased, so has the interest in various techniques to reduce their parameter counts and accelerate their execution. An active area of research in this field is sparsity – encouraging zero values in parameters that can then be discarded from storage or computations. While most research focuses on high levels of sparsity, there are challenges in universally maintaining model accuracy as well as achieving significant speedups over modern matrix-math hardware. To make sparsity adoption practical, the NVIDIA

A100 Optimizations

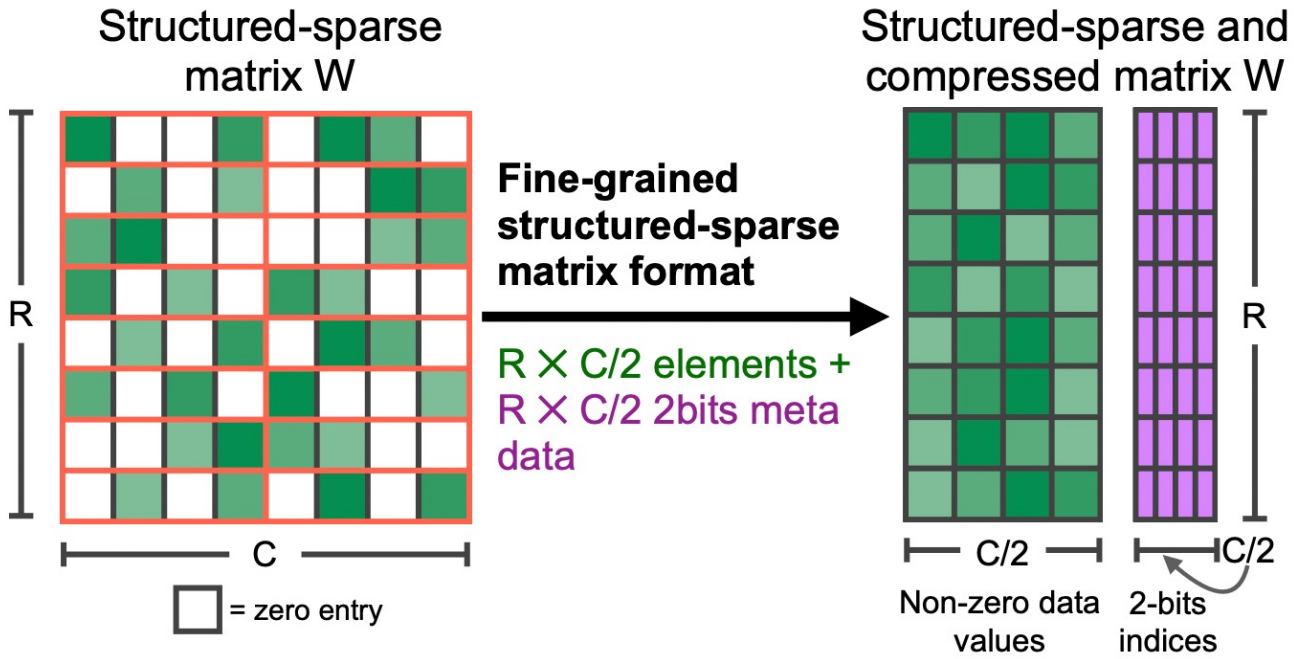


Fig. 1. Structured-sparse matrix (W) storage format. The uncompressed matrix is of dimension R × C and the compressed matrix is of dimension R × $\frac{C}{2}$.

A100 Optimizations

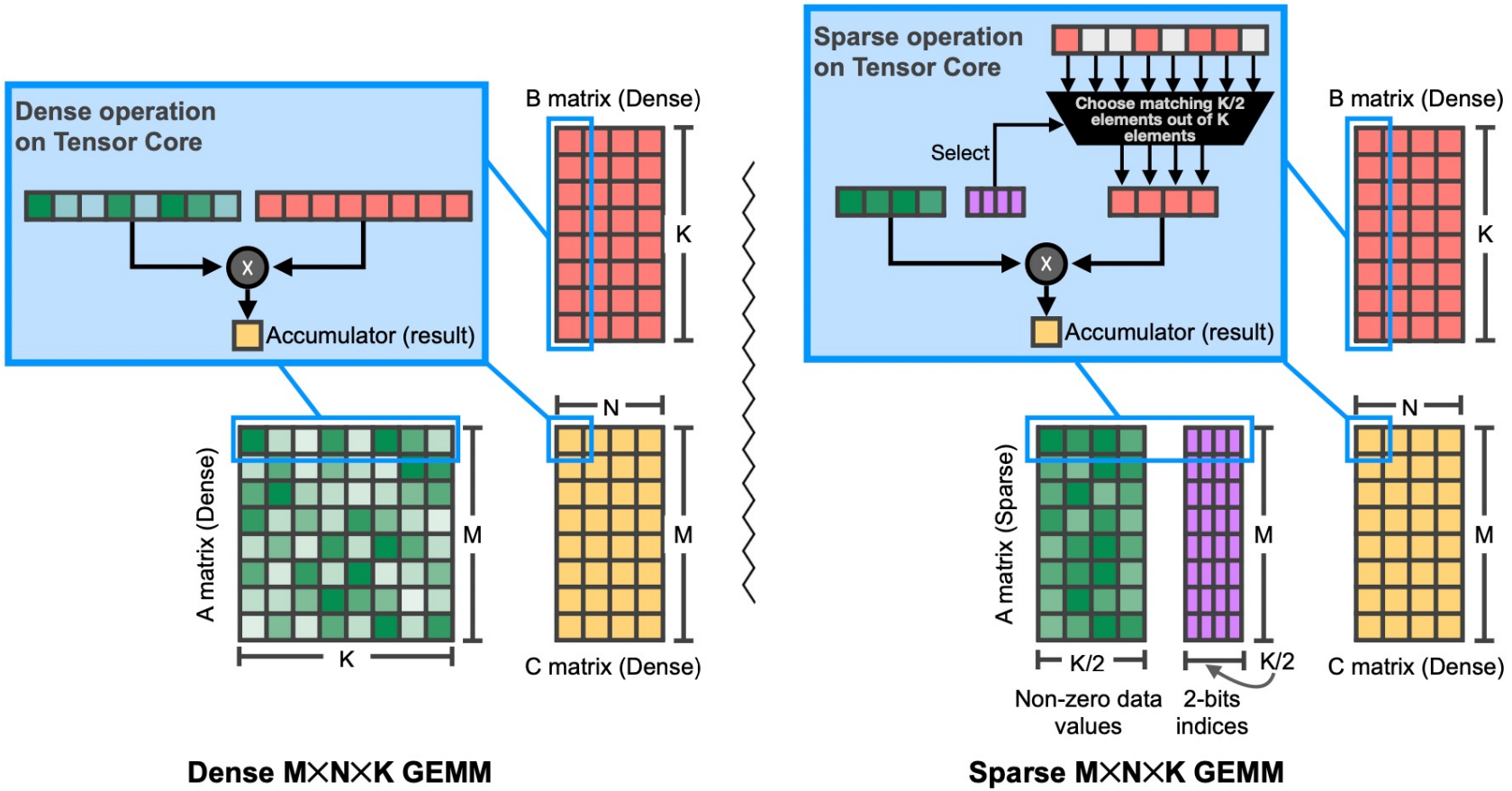
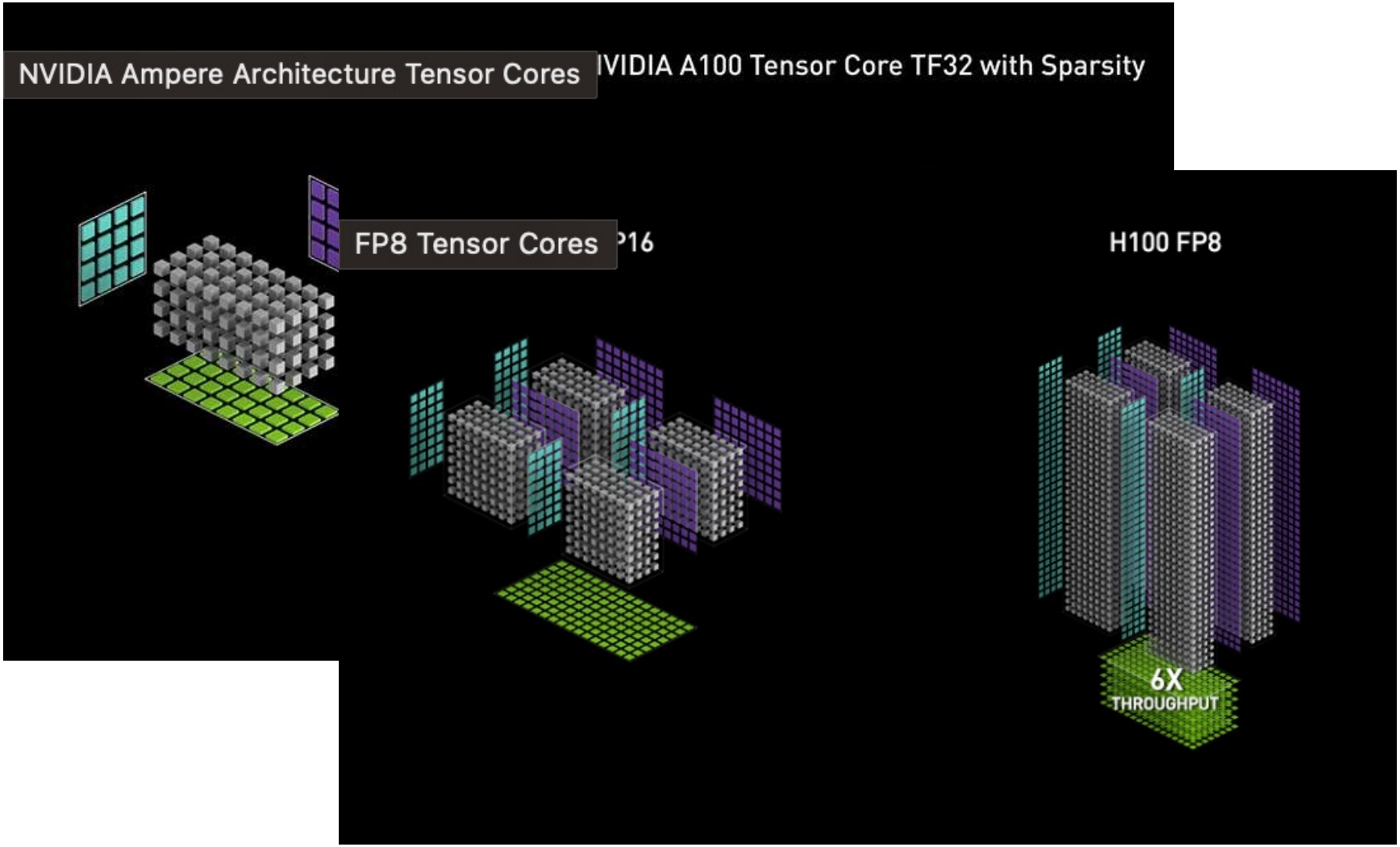
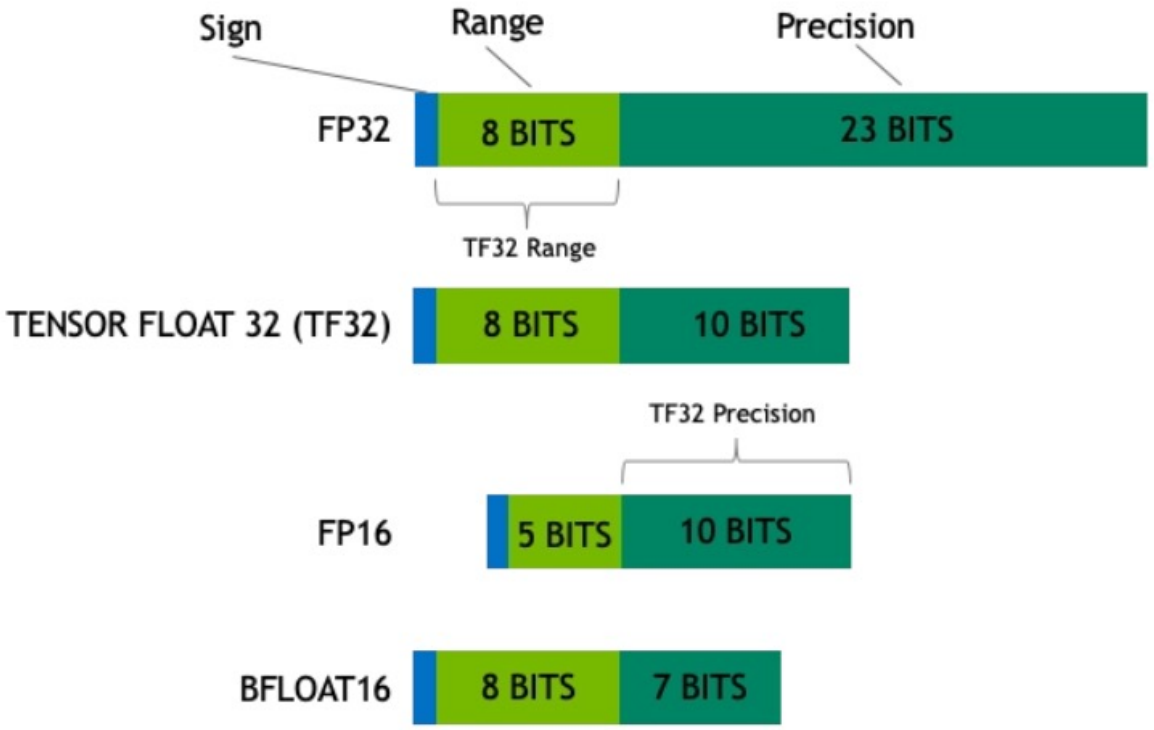


Fig. 2. Mapping a M×N×K GEMM onto a Tensor Core. Dense matrix A, of size M×K, (left side) becomes M×$\frac{K}{2}$ (right side) after pruning with 2:4 sparsity. Sparse Tensor Core hardware selects only the elements from B that correspond to the nonzero values in A, skipping the unnecessary multiplications by zero. In both dense and sparse GEMMs, B and C are dense K×N and M×N matrices, respectively.

Tensor Cores



New FP Formats



TF32 strikes a balance that delivers performance with range and accuracy.

FP8

FP8 Formats For Deep Learning

Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi, Michael Siu, Hao Wu
NVIDIA
{pauliusm, dstosic, pjudd, jkamalu, soberman, mshoeybi, msiu, skyw}@nvidia.com

Neil Burgess, Sangwon Ha, Richard Grisenthwaite
Arm
{neil.burgess, sangwon.ha, richard.grisenthwaite}@arm.com

Naveen Mellempudi, Marius Cornea, Alexander Heinecke, Pradeep Dubey
Intel
{naveen.k.mellempudi, marius.cornea, alexander.heinecke, pradeep.dubey}@intel.com

Abstract

FP8 is a natural progression for accelerating deep learning training inference beyond the 16-bit formats common in modern processors. In this paper we propose an 8-bit floating point (FP8) binary interchange format consisting of two encodings - E4M3 (4-bit exponent and 3-bit mantissa) and E5M2 (5-bit exponent and 2-bit mantissa). While E5M2 follows IEEE 754 conventions for representatio of

FP8

3 FP8 Binary Interchange Format

FP8 consists of two encodings - E4M3 and E5M2, where the name explicitly states the number of exponent (E) and mantissa (M) bits. We use the common term "mantissa" as a synonym for IEEE 754 standard's trailing significand field (i.e. bits not including the implied leading 1 bit for normal floating point numbers). The recommended use of FP8 encodings is E4M3 for weight and activation tensors, and E5M2 for gradient tensors. While some networks can train with just the E4M3 or the E5M2 type, there are networks that require both types (or must maintain many fewer tensors in FP8). This is consistent with findings in [20, 16], where inference and forward pass of training use a variant of E4M3, gradients in the backward pass of training use a variant of E5M2.

FP8 encoding details are specified in Table 1. We use the S.E.M notation to describe binary encodings in the table, where S is the sign bit, E is the exponent field (either 4 or 5 bits containing biased exponent), M is either a 3- or a 2-bit mantissa. Values with a 2 in the subscript are binary, otherwise they are decimal.

Table 1: Details of FP8 Binary Formats

	E4M3	E5M2
Exponent bias	7	15
Infinities	N/A	$S.11111.00_2$
NaN	$S.1111.111_2$	$S.11111.\{01, 10, 11\}_2$
Zeros	$S.0000.000_2$	$S.00000.00_2$
Max normal	$S.1111.110_2 = 1.75 * 2^8 = 448$	$S.11110.11_2 = 1.75 * 2^{15} = 57,344$
Min normal	$S.0001.000_2 = 2^{-6}$	$S.00001.00_2 = 2^{-14}$
Max subnorm	$S.0000.111_2 = 0.875 * 2^{-6}$	$S.00000.11_2 = 0.75 * 2^{-14}$
Min subnorm	$S.0000.001_2 = 2^{-9}$	$S.00000.01_2 = 2^{-16}$

FP8

4.1 Training

In the FP8 training experiments we retain the same model architectures, weight initializations, and optimizer hyper-parameters as are used for higher-precision baseline training sessions. Baselines were trained in either FP16 or bfloat16, which have been shown to match the results of single-precision training sessions [14, 9]. In this study we focused on the input tensors for math-intensive operations - convolutions and matrix multiplies, to which we'll refer as GEMM-operations as they involve dot-product computations. Thus, unless otherwise specified, we clip to FP8-representable values the activation, weight, and activation gradient tensors that are inputs to GEMMs. Output tensors were left in higher precision as they typically are consumed by non-GEMM operations, such as a non-linearities or normalizations, and in a number of cases get fused with the preceding GEMM operation. Moving more tensors to FP8 is the subject of future study.

Table 2: Image Classification Models, ILSVRC12 Validation Top-1 Accuracy

Model	Baseline	FP8
VGG-16	71.27	71.11
VGG-16 BN	73.95	73.69
Inception v3	77.23	77.06
DenseNet 121	75.59	75.33
DenseNet 169	76.97	76.83
Resnet18	70.58	70.12
Resnet34	73.84	73.72
Resnet50 v1.5	76.71	76.76
Resnet101 v1.5	77.51	77.48
ResNeXt50	77.68	77.62
Xception	79.46	79.17
MobileNet v2	71.65	71.04
DeiT small	80.08	80.02

FP8

Table 4: NLP Models, Perplexity

Model	Baseline	FP8
Transformer-XL Base	22.98	22.99
Transformer-XL Large	17.80	17.75
GPT 126M	19.14	19.24
GPT 1.3B	10.62	10.66
GPT 5B	8.94	8.98
GPT 22B	7.21	7.24
GPT 175B	6.65	6.68

FP8 – Inference

4.2 Inference

8-bit inference deployment is greatly simplified by FP8 training, as inference and training use the same datatypes. This is in contrast to int8 inference with networks trained in 32- or 16-bit floating point, which require post-training quantization (PTQ) calibration and sometimes quantization-aware training (QAT) in order to maintain model accuracy. Furthermore, even with quantization aware training some int8-quantized models may not completely recover the accuracy achieved with floating point [1].

We evaluate FP8 post-training quantization of models trained in 16-bit floating point. Table 5 lists inference accuracies for FP16-trained models quantized to either int8 or E4M3 for inference. Both quantizations use per-channel scaling factors for weights, per-tensor scaling factors for activations, as is common for int8 fixed-point. All input tensors to matrix-multiply operations (including attention batched matrix multiplies) were quantized. Max-calibration (choosing the scaling factor so that the maximum magnitude in a tensor is represented) is used for weights, activation tensors are calibrated using the best calibration chosen from max, percentile, and MSE methods. BERT language model evaluation on Stanford Question Answering Dataset shows that FP8 PTQ maintains accuracy while int8 PTQ leads to a significant loss of model accuracy. We also tried casting the tensors to FP8 without applying a scaling factor, which resulted in a significant accuracy loss, increasing the perplexity to 11.0. Evaluation of GPT models on wikitext103 dataset shows that while FP8 PTQ is much better at retaining model accuracy compared to int8.

Table 5: Post training quantization of models trained in 16-bit floating point. For F1 metrics higher is better, for perplexity lower is better. Best 8-bit result is bolded.

Model	Dataset (metric)	16-bit FP	int8	E4M3
BERT Base	SQuAD v1.1 (F1)	88.19	76.89	**88.09**
BERT Large	SQuAD v1.1 (F1)	90.87	89.65	**90.94**
GPT3 126M	wikitext103 (perplexity)	19.01	28.37	**19.43**
GPT3 1.3B	wikitext103 (perplexity)	10.19	12.74	**10.29**
GPT3 6.7B	wikitext103 (perplexity)	8.51	10.29	**8.41**

NVIDIA TESLA GRAPHICS CARD	NVIDIA H100 (SMX5)	NVIDIA H100 (PCIE)	NVIDIA A100 (SXM4)	NVIDIA A100 (PCIE4)	TESLA V100S (PCIE)	TESLA V100 (SXM2)	TESLA P100 (SXM2)	TESLA P100 (PCI-EXPRESS)	TESLA M40 (PCI-EXPRESS)	TESLA K40 (PCI-EXPRESS)
GPU	GH100 (Hopper)	GH100 (Hopper)	GA100 (Ampere)	GA100 (Ampere)	GV100 (Volta)	GV100 (Volta)	GP100 (Pascal)	GP100 (Pascal)	GM200 (Maxwell)	GK110 (Kepler)
Process Node	4nm	4nm	7nm	7nm	12nm	12nm	16nm	16nm	28nm	28nm
Transistors	80 Billion	80 Billion	54.2 Billion	54.2 Billion	21.1 Billion	21.1 Billion	15.3 Billion	15.3 Billion	8 Billion	7.1 Billion
GPU Die Size	814mm2	814mm2	826mm2	826mm2	815mm2	815mm2	610 mm2	610 mm2	601 mm2	551 mm2
SMs	132	114	108	108	80	80	56	56	24	15
TPCs	66	57	54	54	40	40	28	28	24	15
FP32 CUDA Cores Per SM	128	128	64	64	64	64	64	64	128	192
FP64 CUDA Cores / SM	128	128	32	32	32	32	32	32	4	64

Tensor Cores	528	456	432	432	640	640	N/A	N/A	N/A	N/A
Texture Units	528	456	432	432	320	320	224	224	192	240
Boost Clock	TBD	TBD	1410 MHz	1410 MHz	1601 MHz	1530 MHz	1480 MHz	1329MHz	1114 MHz	875 MHz
TOPs (DNN/AI)	3958 TOPs	3200 TOPs	1248 TOPs 2496 TOPs with Sparsity	1248 TOPs 2496 TOPs with Sparsity	130 TOPs	125 TOPs	N/A	N/A	N/A	N/A
Memory Interface	5120-bit HBM3	5120-bit HBM2e	6144-bit HBM2e	6144-bit HBM2e	4096-bit HBM2	4096-bit HBM2	4096-bit HBM2	4096-bit HBM2	384-bit GDDR5	384-bit GDDR5
Memory Size	Up To 80 GB HBM3 @ 3.0 Gbps	Up To 80 GB HBM2e @ 2.0 Gbps	Up To 40 GB HBM2 @ 1.6 TB/s Up To 80 GB HBM2 @ 1.6 TB/s	Up To 40 GB HBM2 @ 1.6 TB/s Up To 80 GB HBM2 @ 2.0 TB/s	16 GB HBM2 @ 1134 GB/s	16 GB HBM2 @ 900 GB/s	16 GB HBM2 @ 732 GB/s	16 GB HBM2 @ 732 GB/s 12 GB HBM2 @ 549 GB/s	24 GB GDDR5 @ 288 GB/s	12 GB GDDR5 @ 288 GB/s
L2 Cache Size	51200 KB	51200 KB	40960 KB	40960 KB	6144 KB	6144 KB	4096 KB	4096 KB	3072 KB	1536 KB
TDP	700W	350W	400W	250W	250W	300W	300W	250W	250W	235W