Lecture 3:

GPU Hardware and Programming Model

Modern Parallel Computing
John Owens
EEC 289Q, UC Davis, Winter 2018

Today's goal

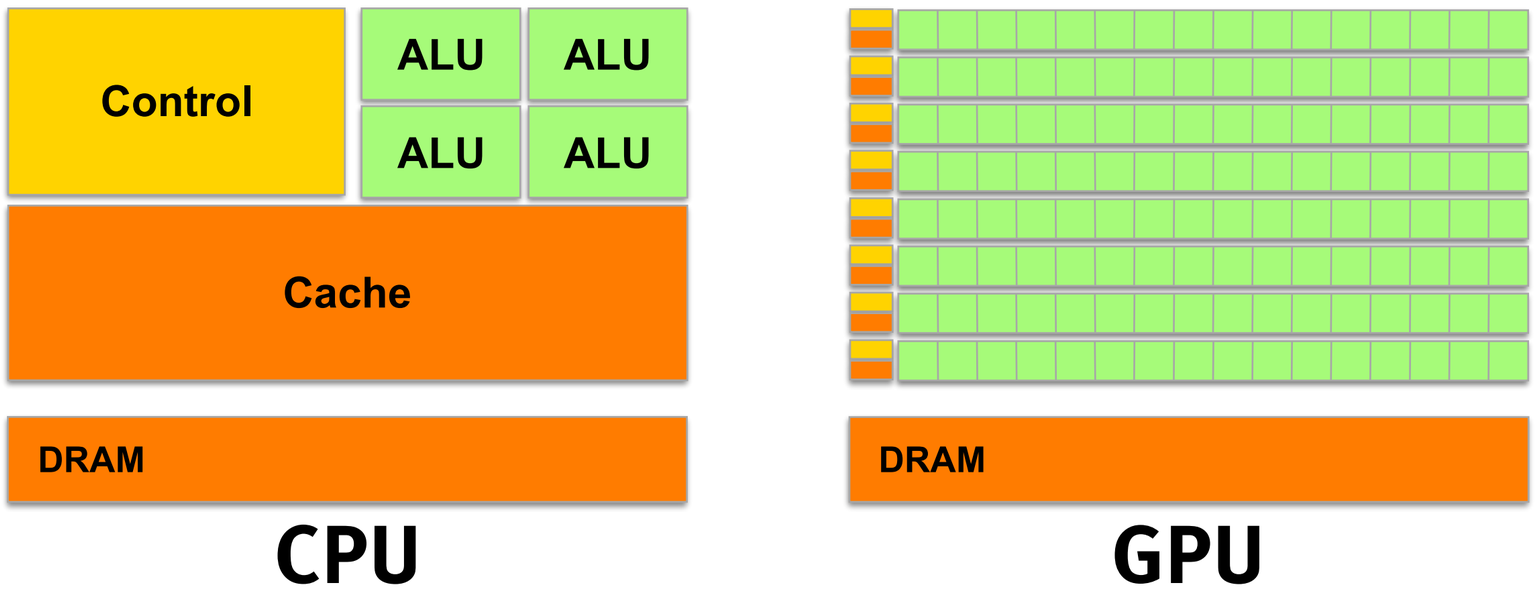
- Last lecture: why GPUs look like they do
- Today:
  - Example GPUs (follow-on from last lecture)
  - What GPUs actually look like
  - How you think about them as a programmer (the "programming model")
  - Simple CUDA example
- Next lecture: CUDA

Three key ideas in GPU hardware

- Use many "slimmed down cores" to run in parallel
- Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
  - Option 1: Explicit SIMD vector instructions (SSE, AVX, Xeon Phi)
  - Option 2: Implicit sharing managed by hardware (NVIDIA/ AMD GPUs)
- Avoid latency stalls by interleaving execution of many groups of fragments
  - When one group stalls, work on another group

Why is data-parallel computing fast?

- The GPU is specialized for compute-intensive, highly parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control



An efficient GPU workload …

- Has thousands of independent pieces of work
  - Uses many ALUs on many cores
  - Supports massive interleaving for latency hiding

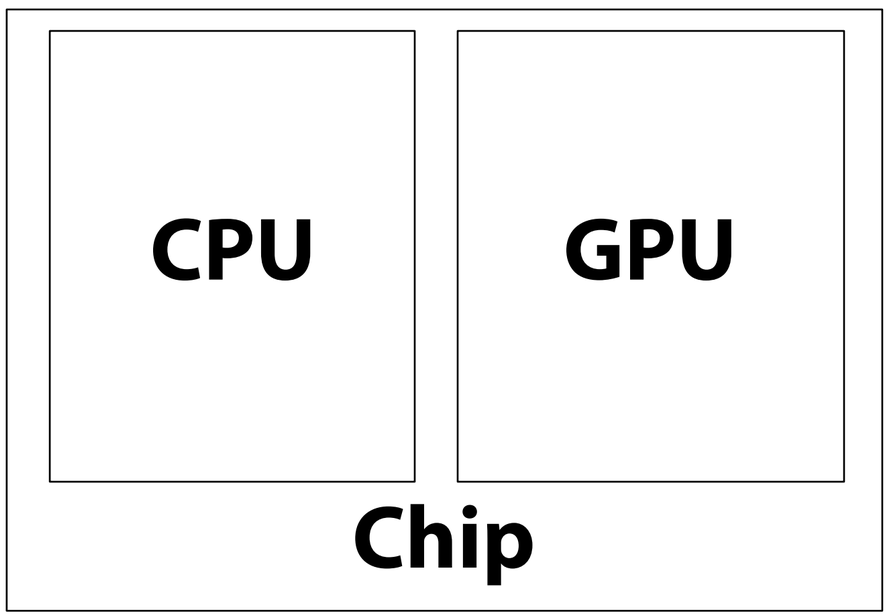
- Is amenable to instruction stream sharing
  - Maps to SIMD execution well
  - Minimizes *branch divergence*

- Is compute-heavy: the ratio of math operations to memory access is high
  - Not limited by bandwidth
  - But even if it is, GPUs have high main memory bandwidth

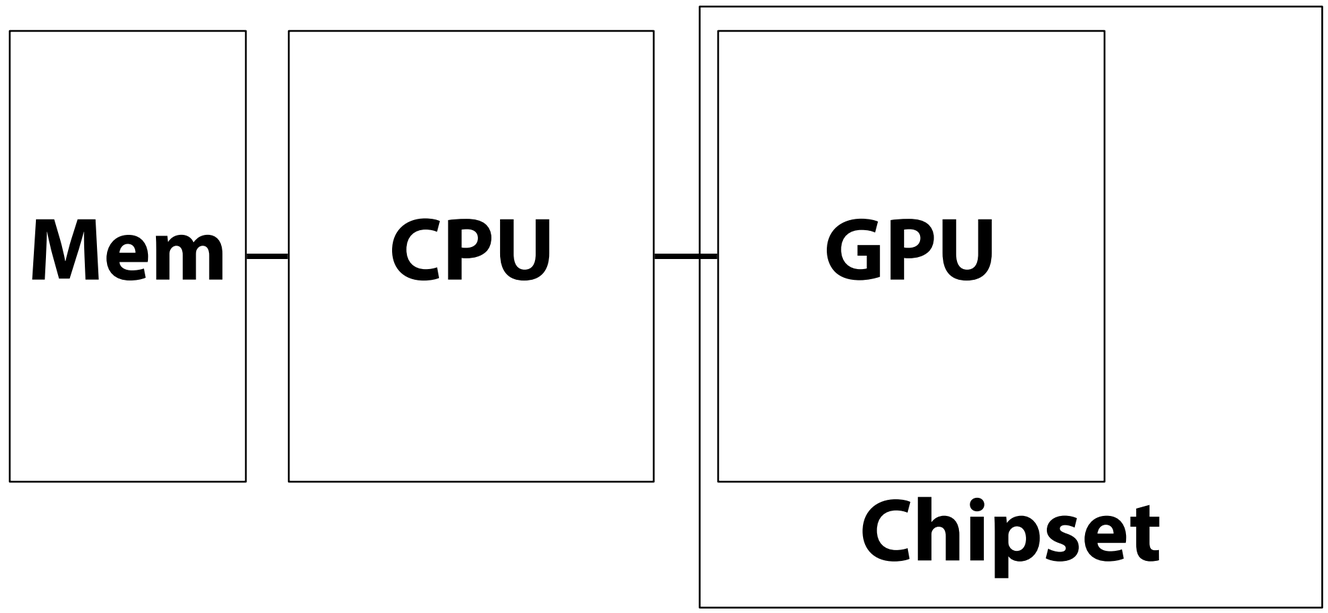
A closer look at real GPUs

- NVIDIA GeForce GTX 285
- AMD Radeon HD 4890
- Intel Larrabee (Xeon Phi/Knight's Landing)

GPU in system (3 alternatives)

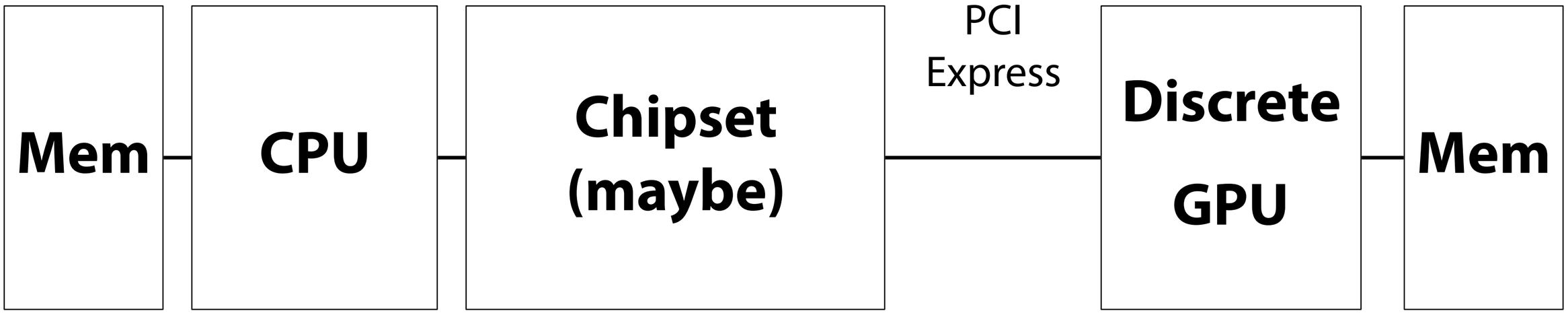


Intel Haswell, NVIDIA Tegra



this is rare today

Generally we'll be talking about this model



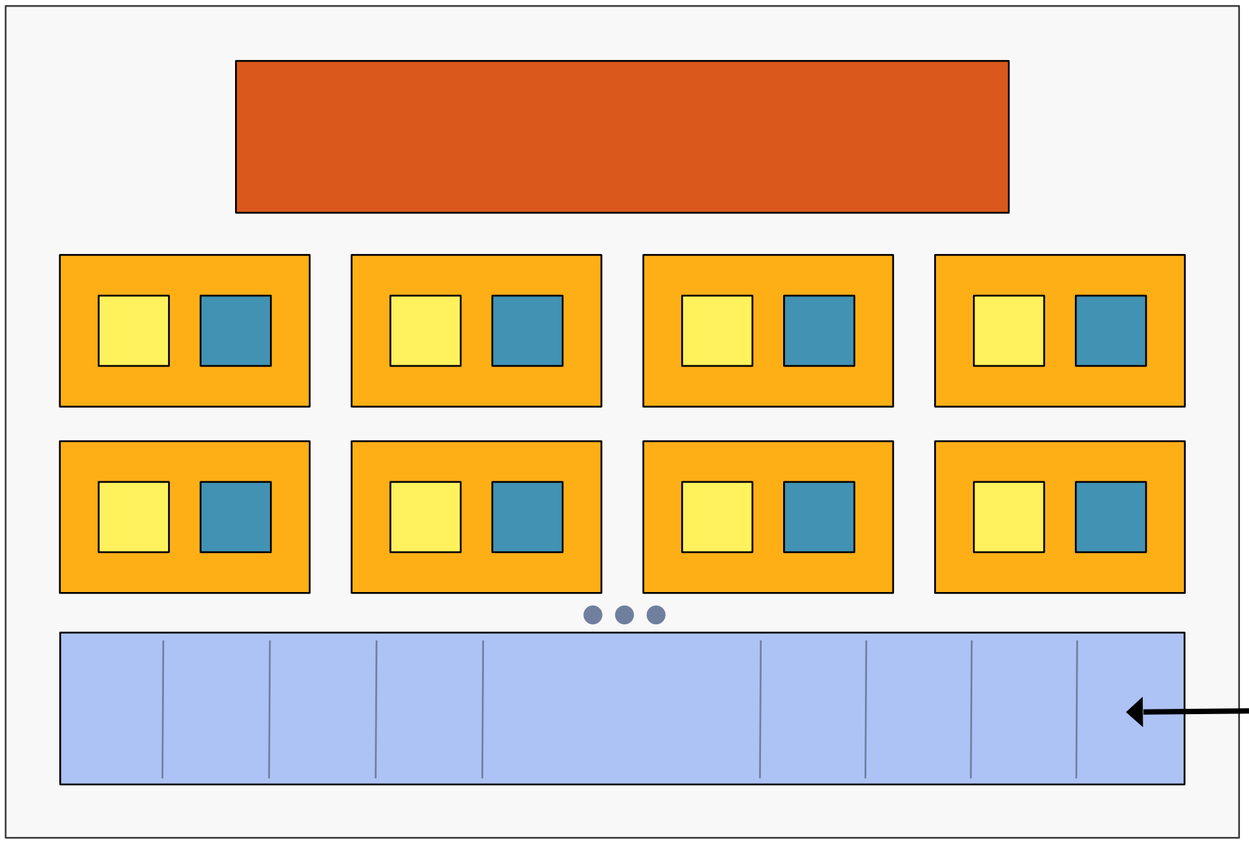
NVIDIA GeForce/Tesla, AMD Radeon

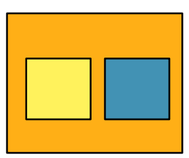
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - "SIMT execution"

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core

NVIDIA GeForce GTX 285 “core”





= SIMD functional unit, control shared across 8 units

= instruction stream decode

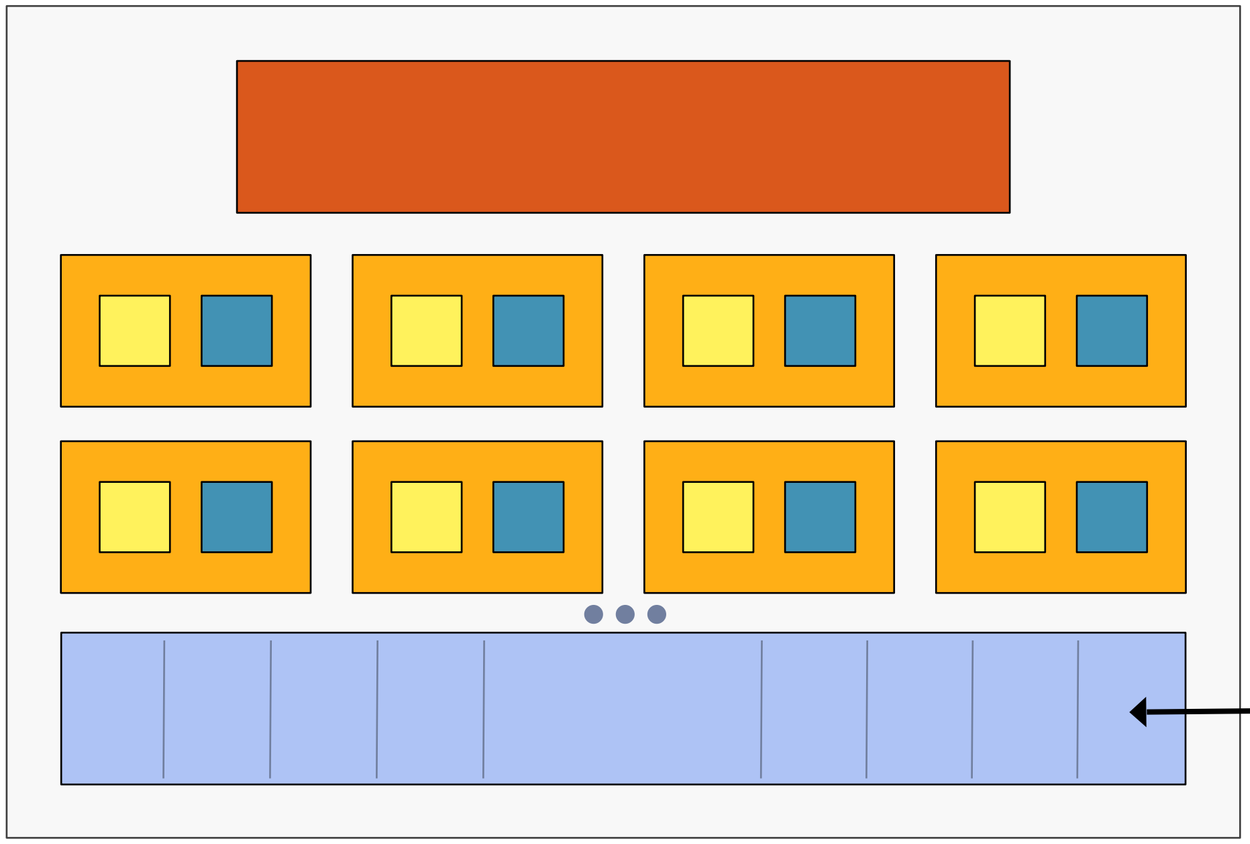
= multiply-add

= multiply

= execution context storage

NVIDIA GeForce GTX 285 “core”

- NVIDIA calls their “core” an “SM” (streaming multiprocessor)
- Groups of 32 [fragments/vertices/threads/etc.] share an instruction stream (NVIDA calls these “warps”)
- Up to 32 groups are simultaneously interleaved
- Up to 1024 fragment contexts can be stored



64 KB of storage
for fragment contexts
(registers)

NVIDIA GeForce GTX 285

- There are 30 of these things on the GTX 285: 30,000 fragments!

NVIDIA GeForce GTX 285

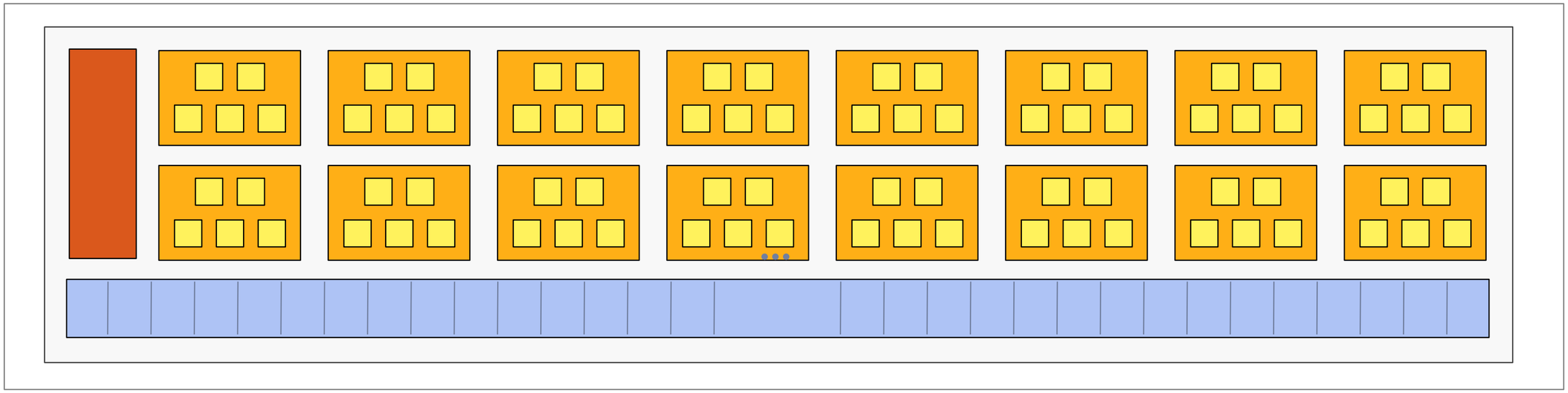
- Generic speak:
  - 30 processing cores
  - 8 SIMD functional units per core
  - Best case: 240 mul-adds + 240 muls per clock

AMD Radeon HD 4890

- AMD-speak:
  - 800 stream processors
  - HW-managed instruction stream sharing (like “SIMT”)

- Generic speak:
  - 10 cores
  - 16 “beefy” SIMD functional units per core
  - 5 multiply-adds per functional unit

AMD Radeon HD 4890 “core”



= SIMD functional unit, control shared across 16 units

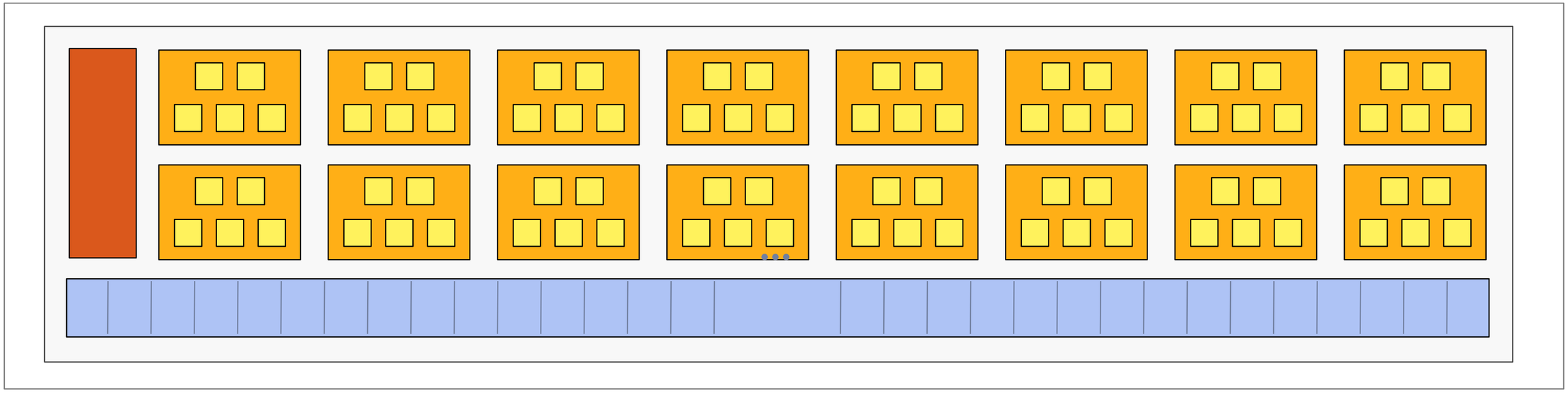
= multiply-add

= instruction stream decode

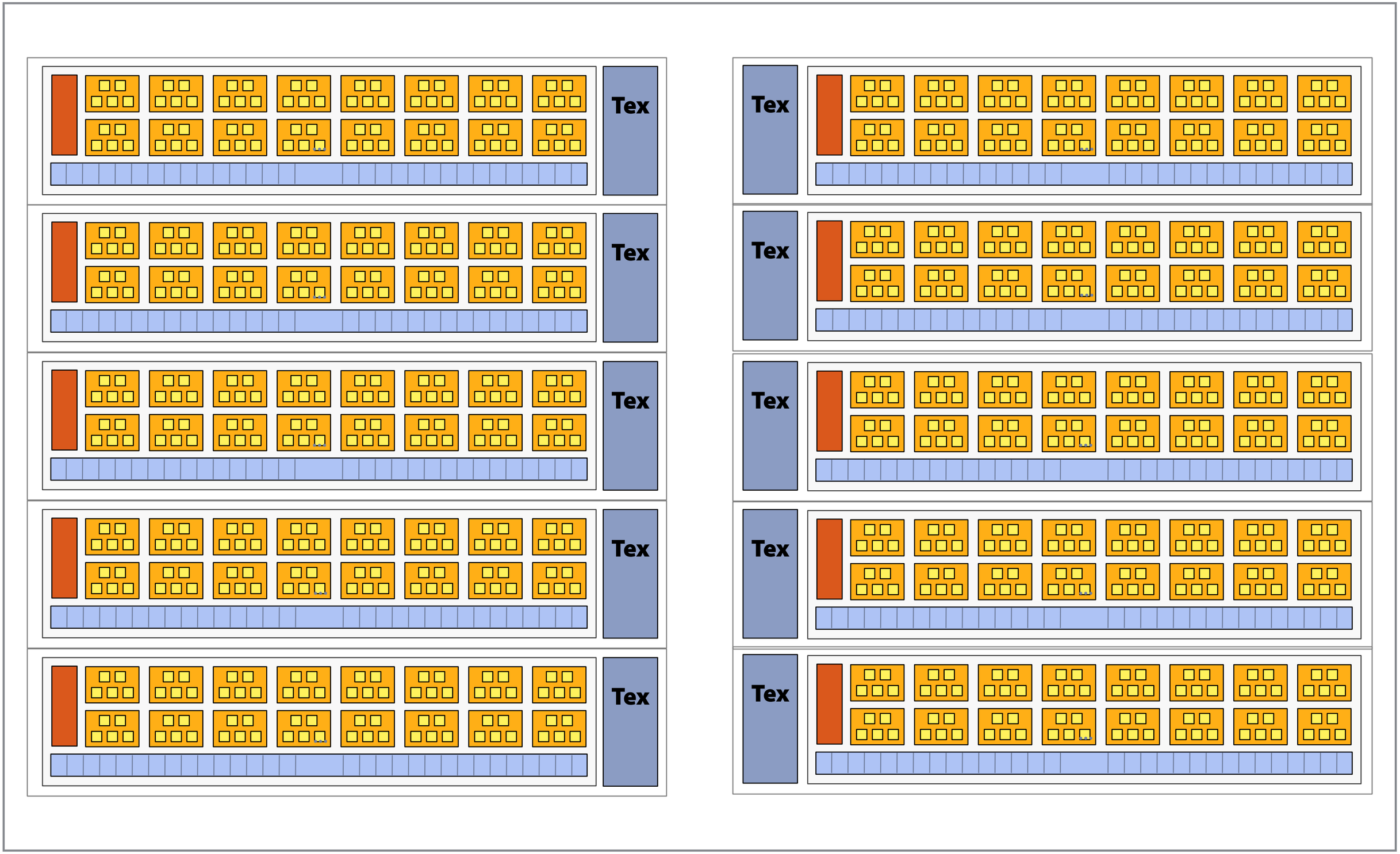
= execution context storage

AMD Radeon HD 4890 "core"

- Groups of 64 [fragments/vertices/etc.] share instruction stream (AMD calls these "wavefronts")
  - One fragment processed by each of the 16 SIMD units
  - Repeat for four clocks



AMD Radeon HD 4890



AMD Radeon HD 4890

- Generic speak:
  - 10 processing "cores"
  - 16 "beefy" SIMD functional units per core
  - 5 multiply-adds per functional unit
  - Best case: 800 multiply-adds per clock

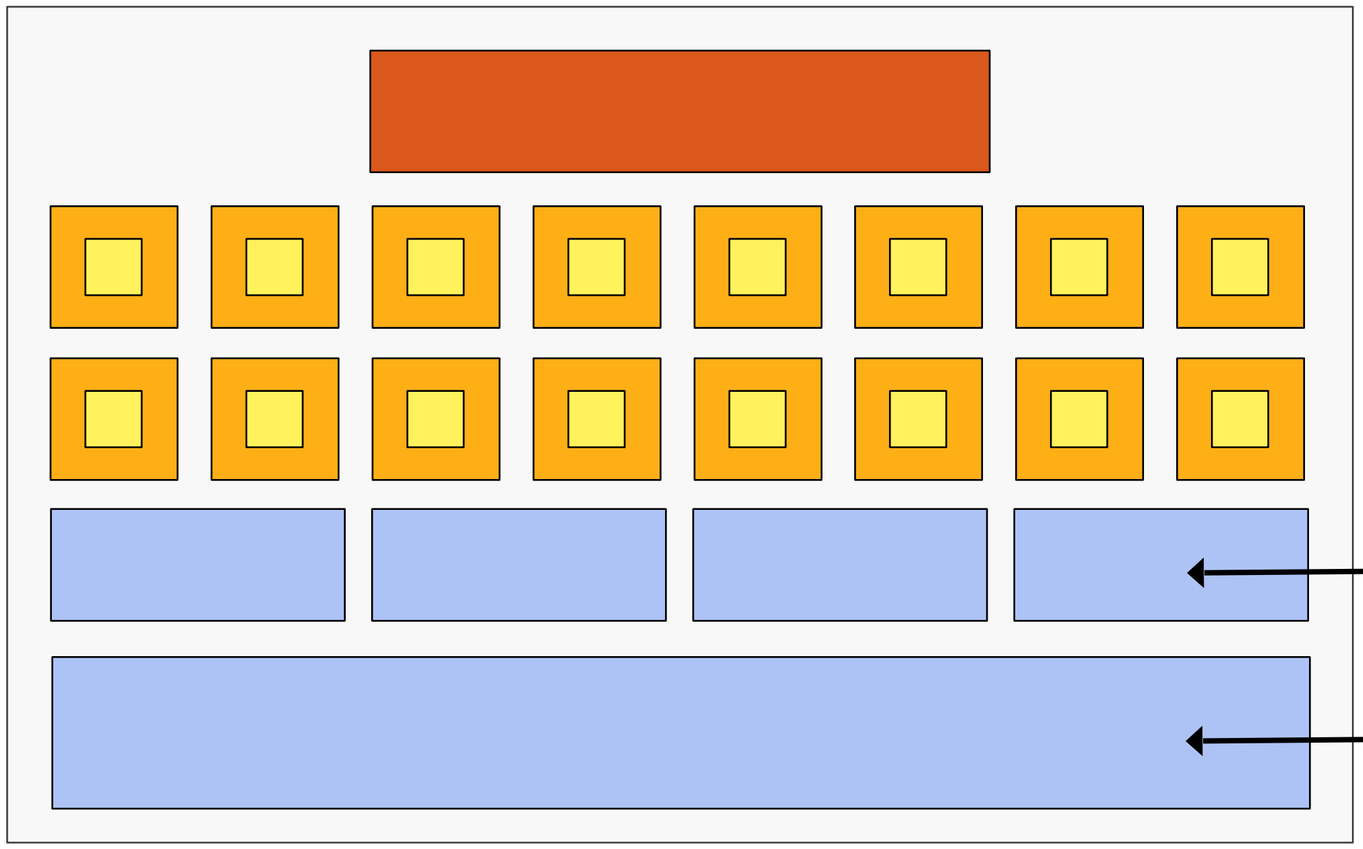
- Scale of interleaving similar to NVIDIA GPUs

Intel Larrabee

- Intel speak:
  - Intel originally didn't say anything about core count or clock rate
  - Explicit 16-wide vector ISA
  - Each core interleaves four x86 instruction streams
  - Software implements additional interleaving

- Generic speak:
  - That was the generic speak

Intel Larrabee "core"



Each HW context:
32 vector registers

32 KB of L1 cache
256 KB of L2 cache

= SIMD functional unit, control shared across 16 units

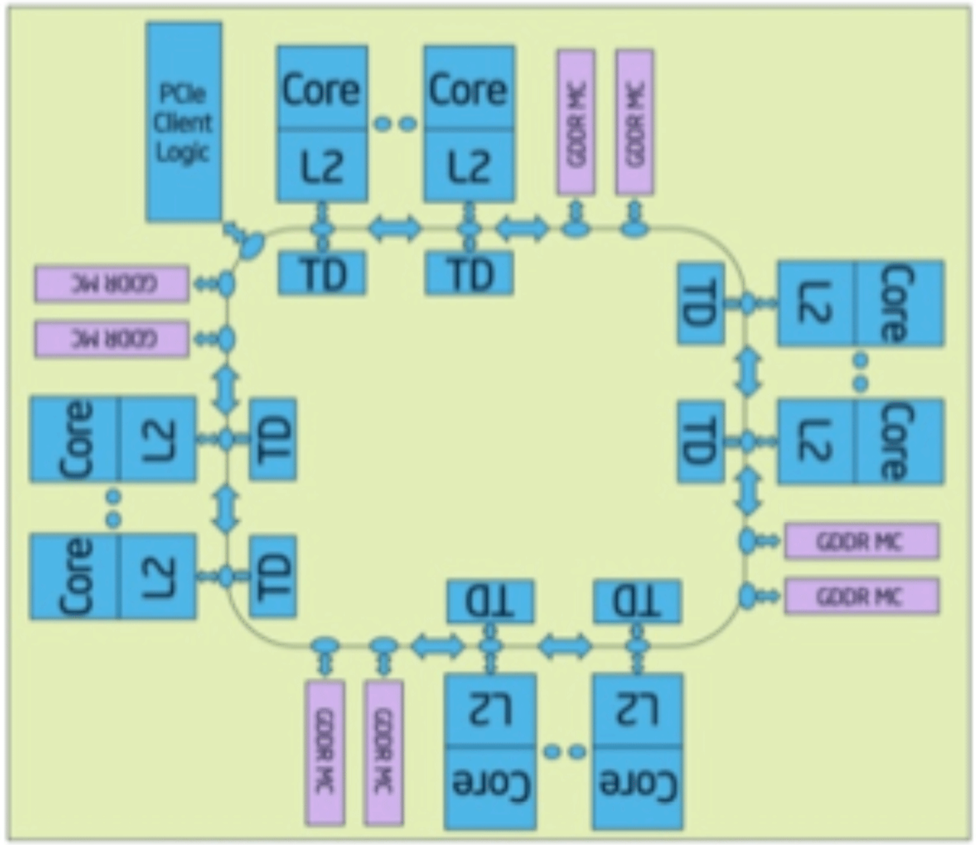
= mul-add

= instruction stream decode

= execution context storage/ HW registers

Intel Xeon Phi

- 60 cores
- Each core: 512b-wide (16 x 32b) SSE-style vector unit, plus scalar
  - 2147 GFLOPS SP, 1074 DP
  - Compare to NVIDIA K20X: 3950/1310
- 8 GB DRAM, 352 GB/s theoretical peak, 200 GB/s achievable



Knights Landing will be built using up to 72 Airmont (Atom) cores with four threads per core,[42][43] supporting for up to 384 GB of "far" DDR4 RAM and 8–16 GB of stacked "near" 3D MCDRAM, which is similar to Micron's Hybrid Memory Cube. Each core will have two 512-bit vector units and will support AVX-512F (AVX3.1) SIMD instructions with Intel AVX-512 Conflict Detection Instructions (CDI), Intel AVX-512 Exponential and Reciprocal Instructions (ERI), and Intel AVX-512 Prefetch Instructions (PFI), along with Intel's full x86 instruction set except TSX.[44] Knights Landing's TDP will range from 160 to 215 W.[Wikipedia]

http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

What sort of features do you have to put in hardware to make it possible to support thousands (or millions) of threads?

Structuring a GPU Program

- CPU assembles input data
- CPU transfers data to GPU (GPU "main memory" or "device memory")
  - CUDA 9 hides this from the programmer—`cudaMallocManaged`
- CPU calls GPU program (or set of kernels). GPU runs out of GPU main memory.
- When GPU finishes, CPU copies back results into CPU memory
- Recent interfaces allow overlap between communicate/compute

Programming Model: A Massively Multi-threaded Processor

- Move data-parallel application portions to the GPU
- Differences between GPU and CPU threads
  - Lightweight threads
  - GPU supports 1000s of threads

Programming Model Big Idea #1

- One thread per data element.
- Doesn't this mean that large problems will have millions of threads?

Programming Model Big Idea #2

- Write one program.
- That program runs on ALL threads in parallel.
- Software terminology here is "SPMD": single-program, multiple-data.
- Hardware terminology here is "SIMT": single-instruction, multiple-thread.
  - Roughly: SIMD means many threads run in lockstep; SIMT means that some divergence is allowed and handled by the hardware

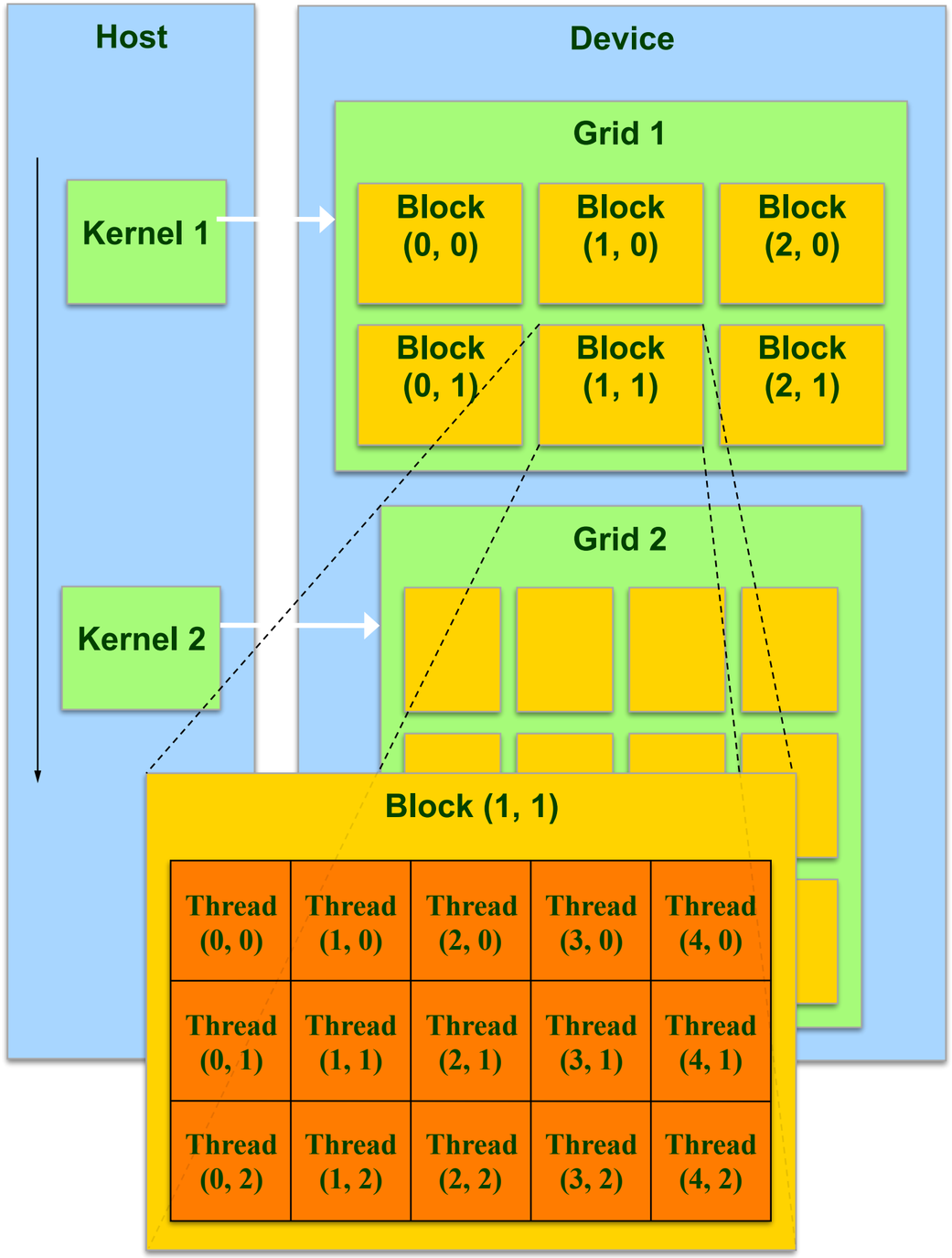
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One SIMT kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA must use 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:
Device = GPU;
Host = CPU;
Kernel = function that runs on the device

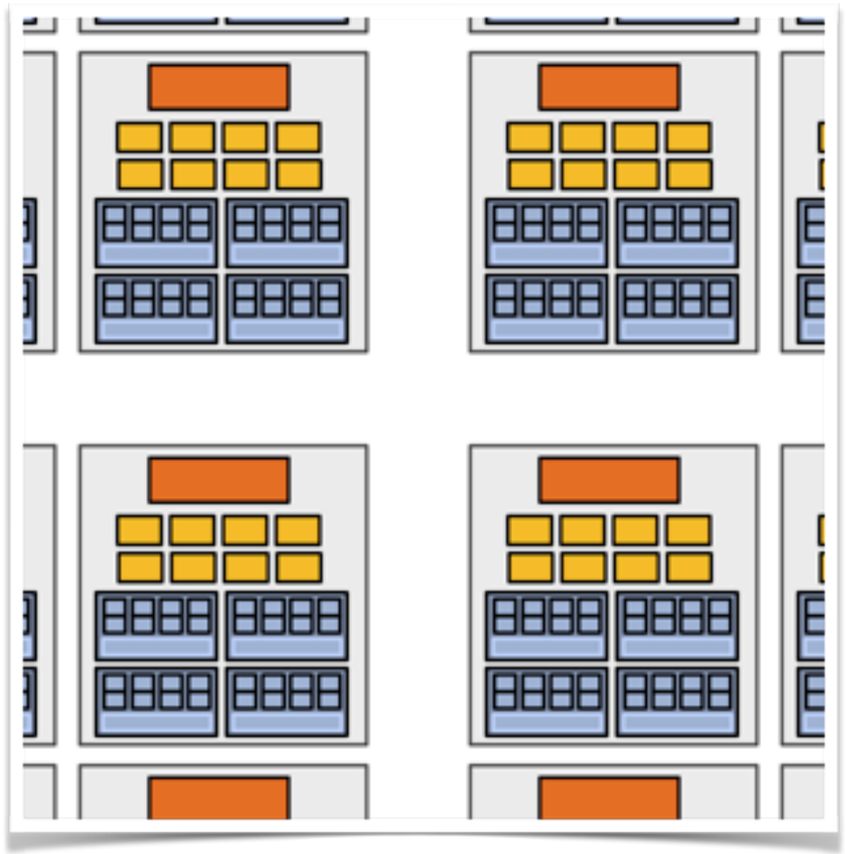
Programming Model Overview

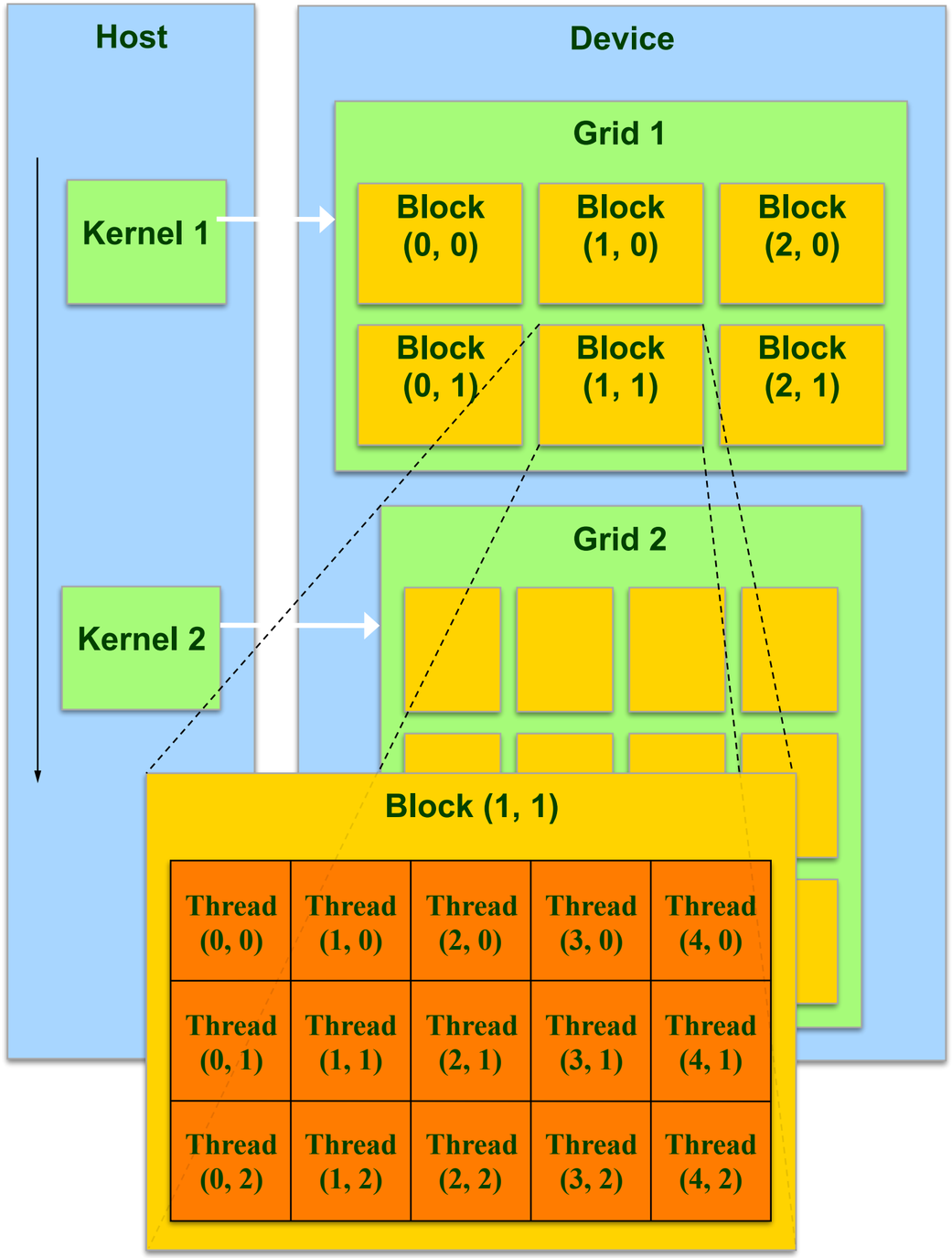
- A kernel is executed as a grid of thread blocks
- The programmer specifies the dimensions of the grid and thread block
- A thread block is a batch of threads that can cooperate with each other (shared memory, synchronization)
- Two threads from two different blocks cannot cooperate
  - Blocks are independent



What The Hardware Does

- Grids run on the entire machine
- Blocks are mapped to cores
- Threads are mapped to scalar processors





Let's look at CUDA code!

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

(1) Use __global__ to indicate a GPU function

```
// CUDA Kernel function to add the elements
// of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

```
// CUDA Kernel function to add the elements
// of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

(2) Allocate GPU-accessible memory

```
// Allocate Unified Memory --
// accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

```
// CUDA Kernel function to add the elements
// of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}



// Allocate Unified Memory --
// accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
```

(3) Launch the kernel on the GPU

```
// Run kernel on 1M elements
// on the GPU
add<<<1, 1>>>(N, x, y);
// kernel calls are non-blocking

// Wait for GPU to finish
// before accessing on host
cudaDeviceSynchronize();
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

Non-idealities:

- Runs on one thread. No parallelism.

- Launching more than one thread results in a race condition (all threads would compute the entire array)

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

(1) Launch 256 threads instead of 1

```
add<<<1, 256>>>(N, x, y);
```

This runs 256 threads in SIMD-parallel fashion on one core

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

(2) Divide work among *stride* threads

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

Each thread knows its thread and block ID, and the dimensions of the grid and block.

```
add<<<1, 256>>>(N, x, y);
```

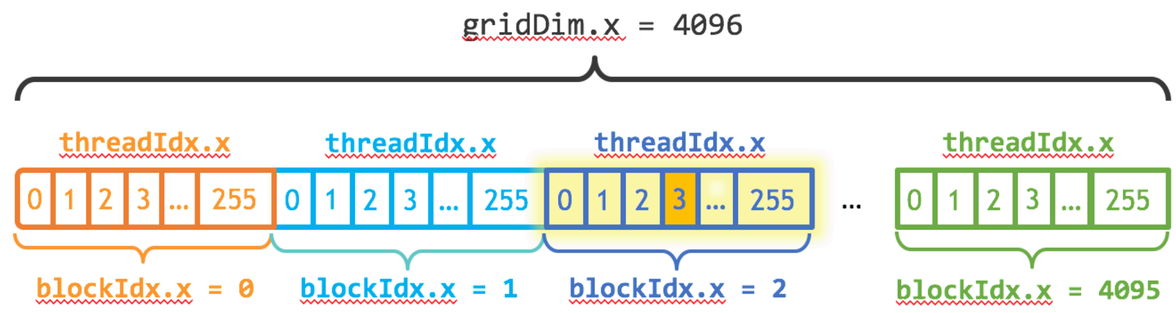
https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 256>>>(N, x, y);
  cudaDeviceSynchronize();

  // … for space, remove error checking/free

  return 0;
}
```

(3) Divide work among *more** threads

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x *
    blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Each thread knows its thread and block ID, and the dimensions of the grid and block.




```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

* *more* = numBlocks × blockSize = *N*

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

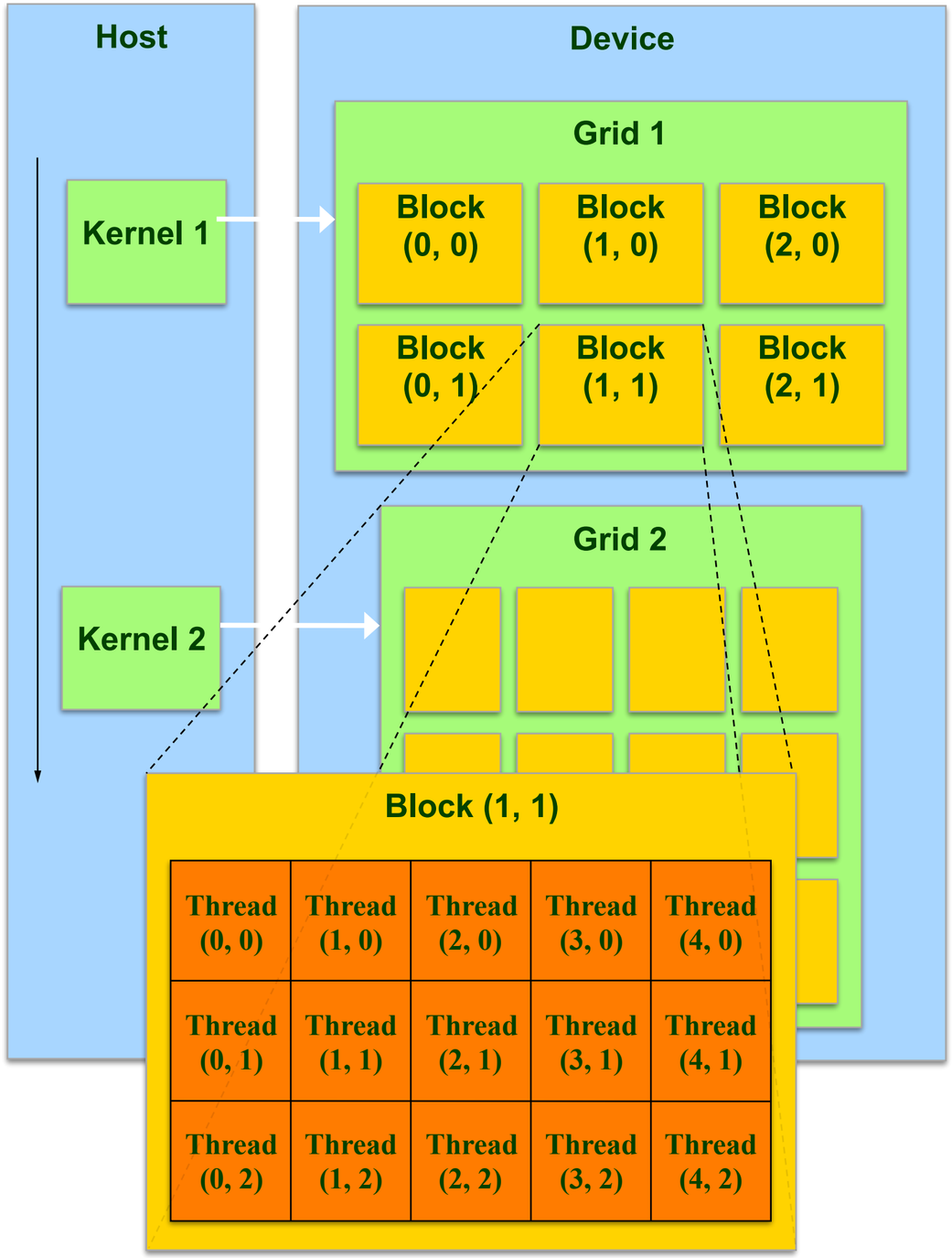
Performance

Version	Laptop (GT 750M)		Server (Tesla K80)	
	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411 ms	30.6 MB/s	463 ms	27.2 MB/s
1 CUDA Block	3.2 ms	3.9 GB/s	2.7 ms	4.7 GB/s
Many CUDA blocks	0.68 ms	18.5 GB/s	0.094 ms	134 GB/s
	Wikipedia says peak: 32 GB/s		*Wikipedia says peak: 240 GB/s*	

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

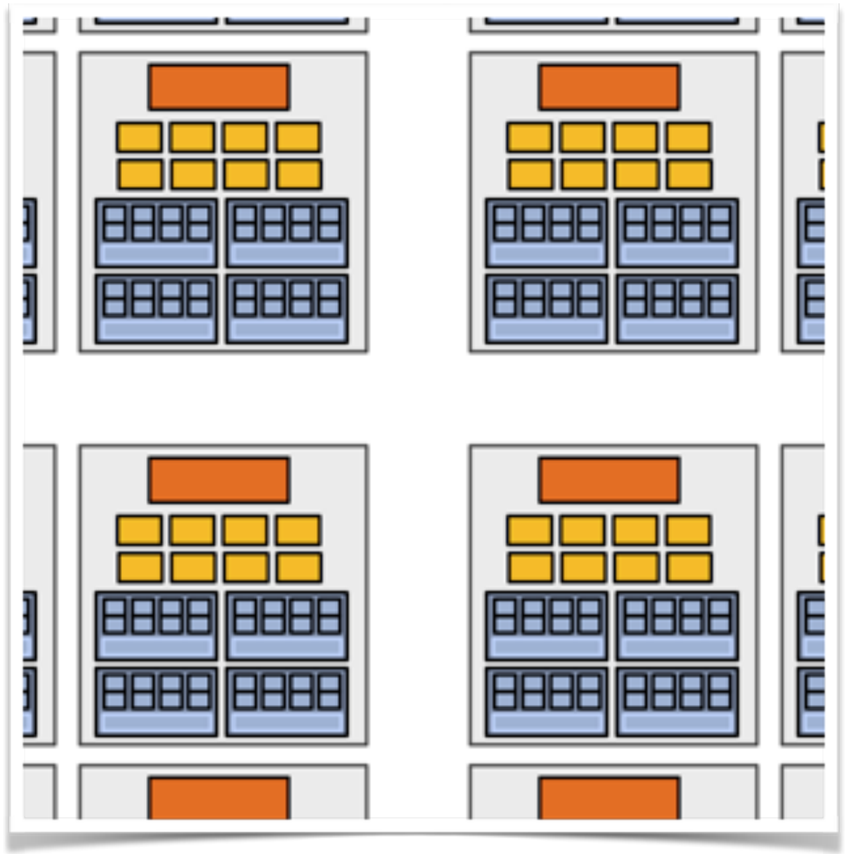
Recap: Programming Model

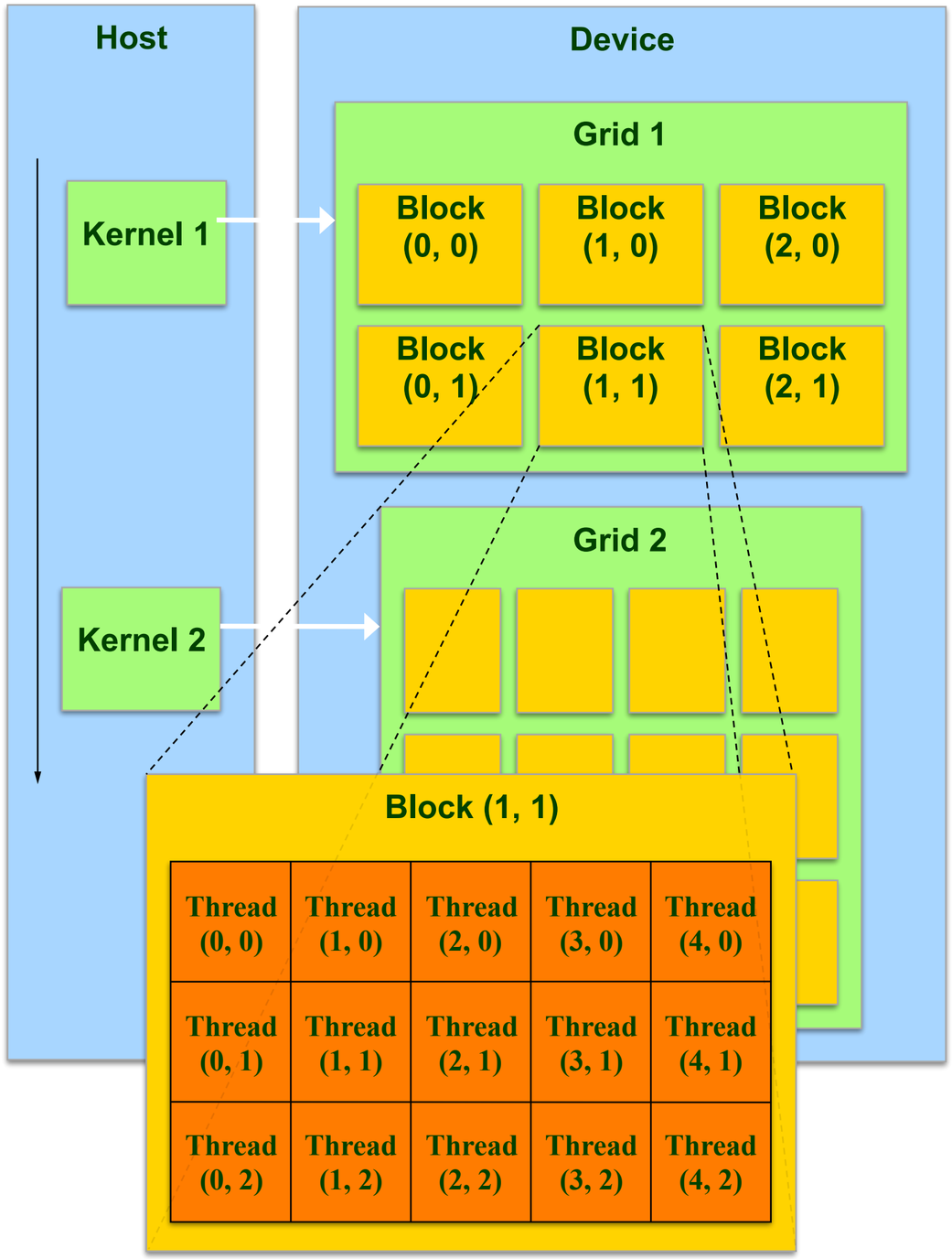
- A kernel is executed as a grid of thread blocks
- The programmer specifies the dimensions of the grid and thread block
- A thread block is a batch of threads that can cooperate with each other (shared memory, synchronization)
- Two threads from two different blocks cannot cooperate
  - Blocks are independent



Recap: What The Hardware Does

- Grids run on the entire machine
- Blocks are mapped to cores
- Threads are mapped to scalar processors





Programming Model Big Idea #3

- *Scalable execution*
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

- Hierarchical execution model
  - Decompose problem into sequential steps (kernels)
  - Decompose kernel into computing parallel blocks
  - Decompose block into computing parallel threads

This is very important!

- Hardware distributes independent blocks to SMs as available

Blocks must be independent

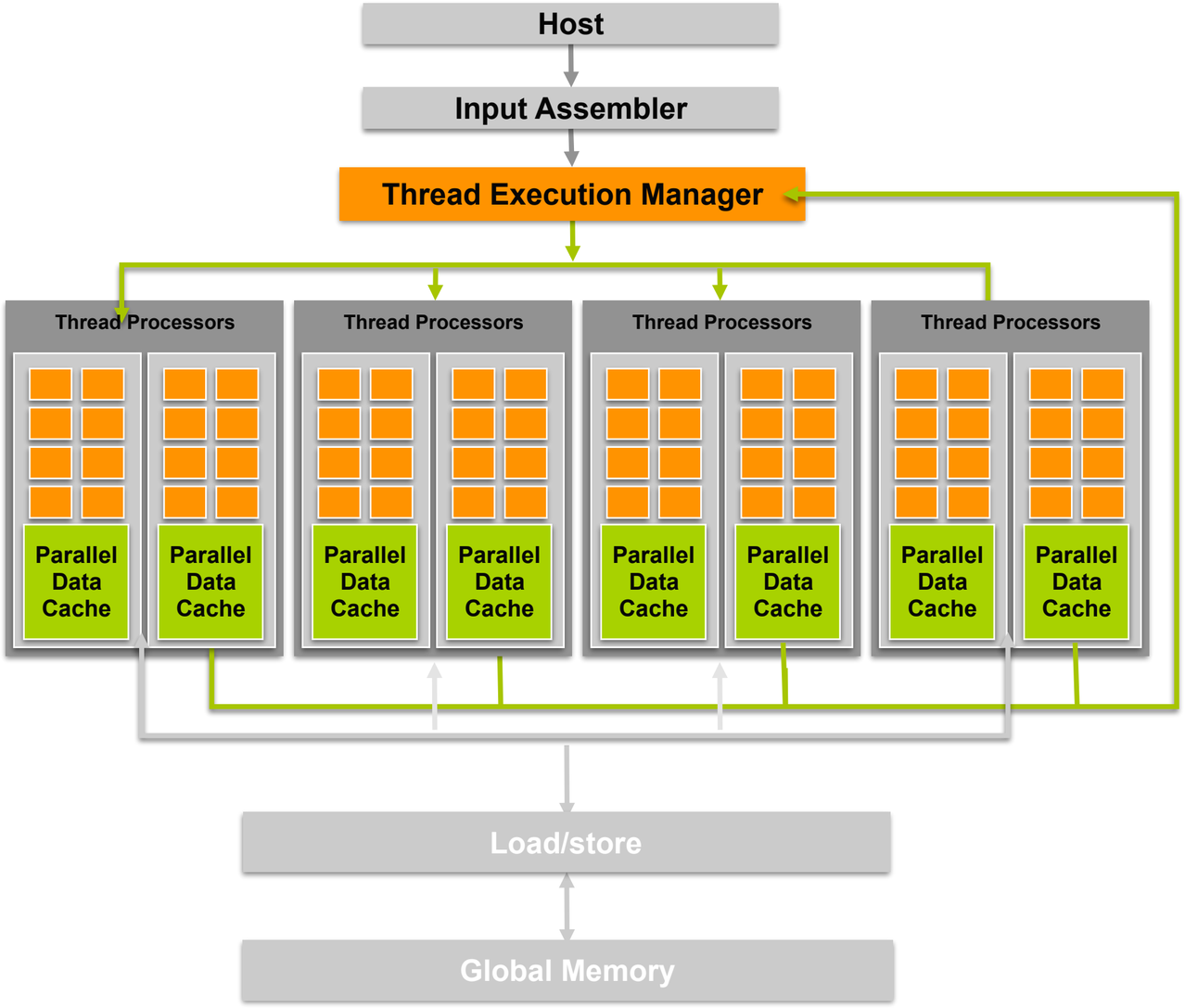
- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may *coordinate* but not *synchronize*
  - shared queue pointer: OK
  - shared lock (A locks, B unlocks): BAD … can easily deadlock
- Independence requirement gives scalability

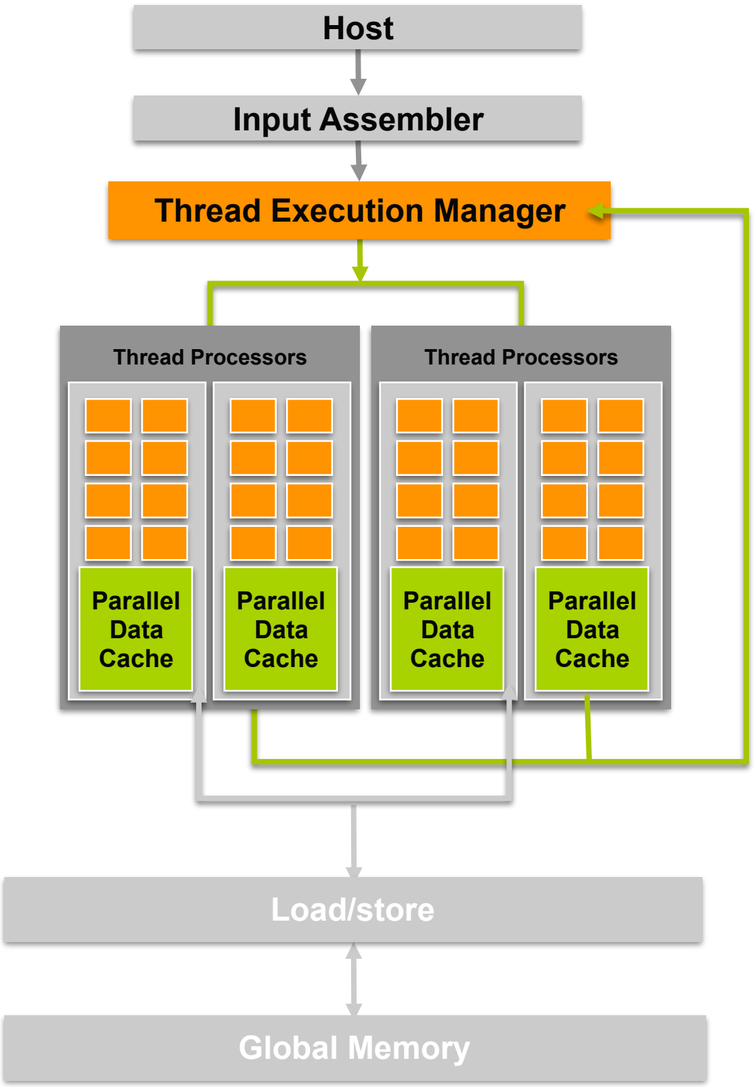
Big Idea #4

- Organization into independent blocks allows scalability / different hardware instantiations
  - If you organize your kernels to run over many blocks ...
  - ... the same code will be efficient on hardware that runs one block at once and on hardware that runs many blocks at once

Scaling the Architecture

- Same program
- Scalable performance

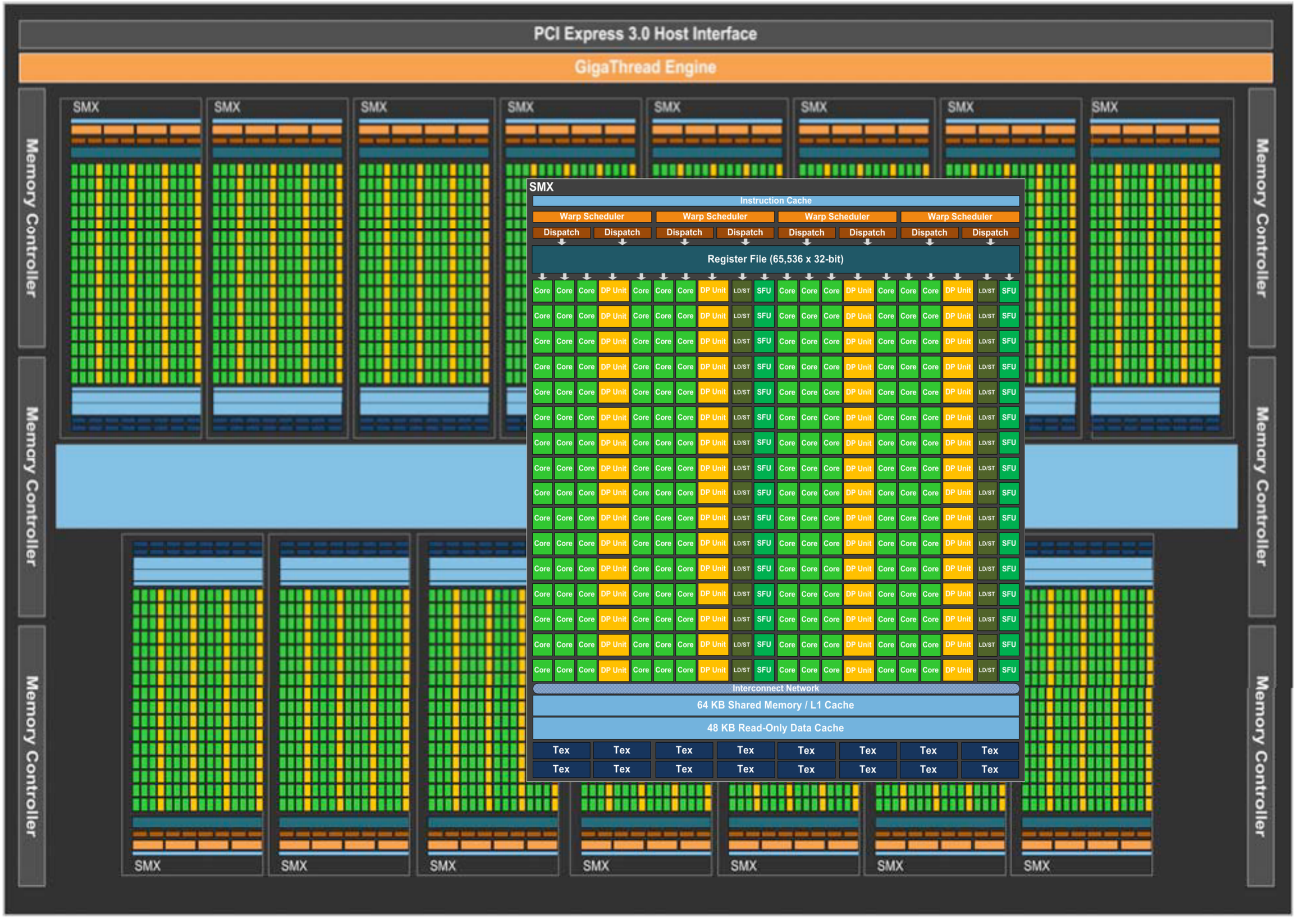




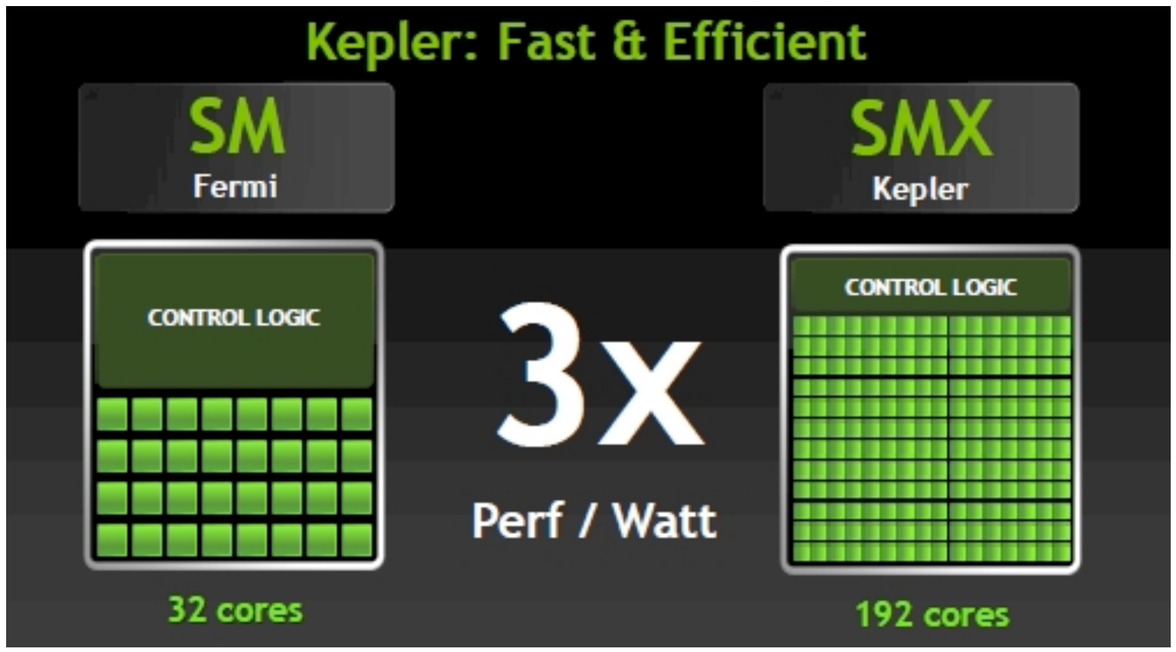
There's lots of dimensions to scale this processor with more resources. What are some of those dimensions? If you had twice as many transistors, what could you do with them?

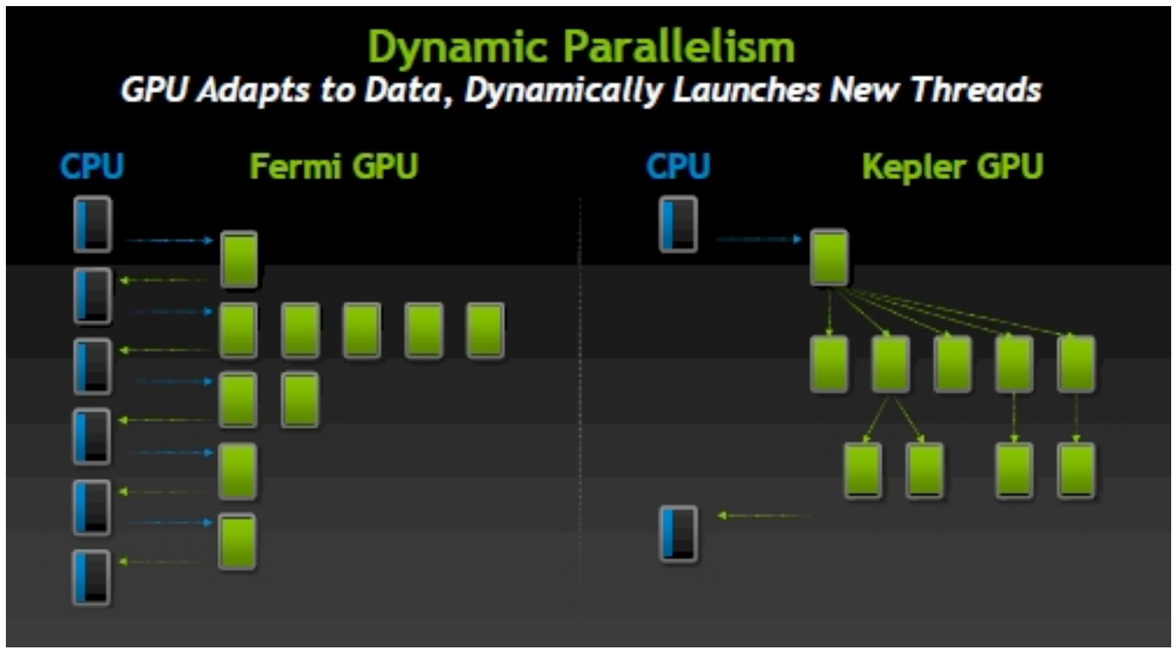
What *should* you do with more resources? In what dimension do you think NVIDIA will scale future GPUs?

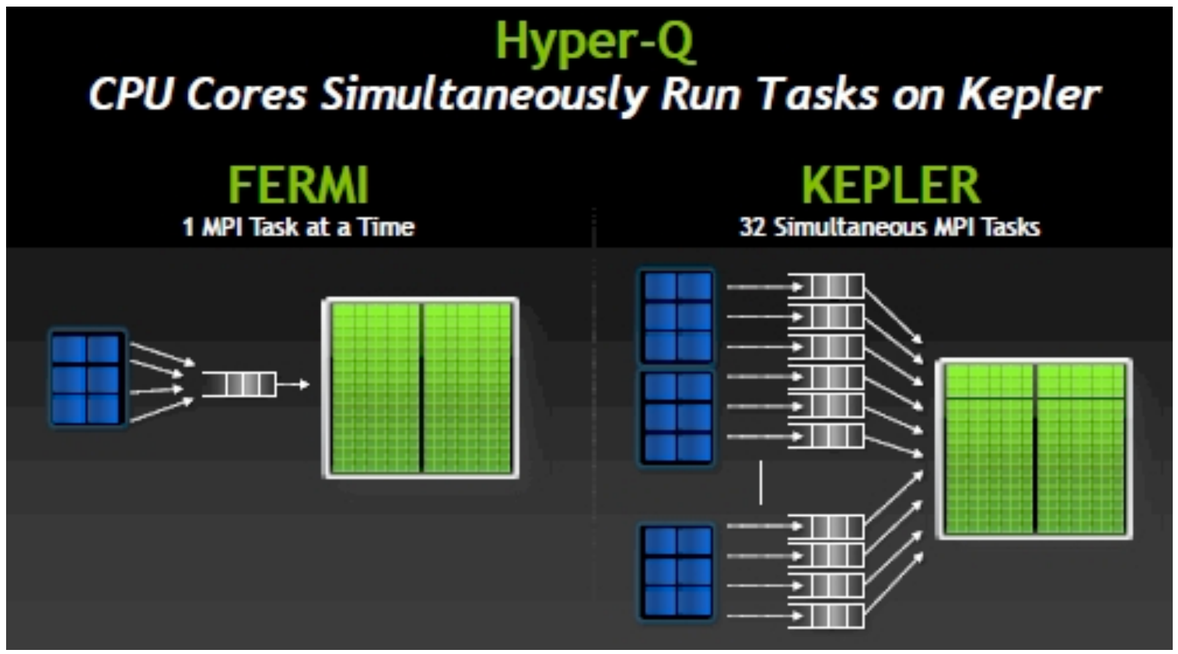
Kepler GK110 block diagram

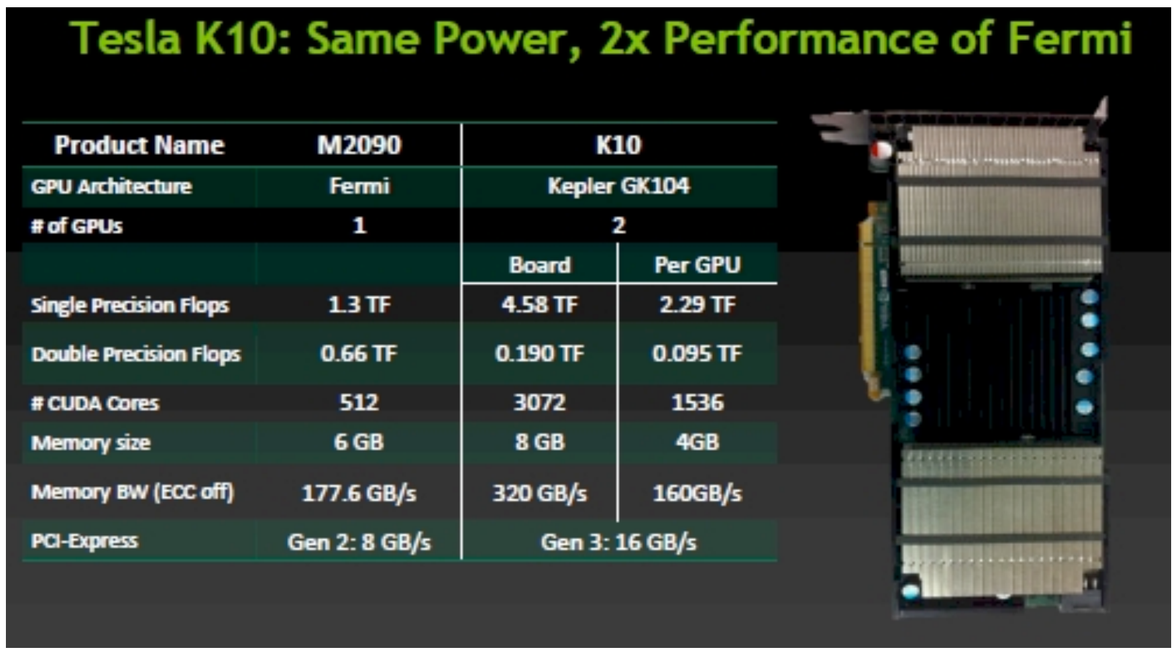


NVIDIA Kepler









Product Name	M2090	K10	
GPU Architecture	Fermi	Kepler GK104	
# of GPUs	1	2	
		Board	Per GPU
Single Precision Flops	1.3 TF	4.58 TF	2.29 TF
Double Precision Flops	0.66 TF	0.190 TF	0.095 TF
# CUDA Cores	512	3072	1536
Memory size	6 GB	8 GB	4GB
Memory BW (ECC off)	177.6 GB/s	320 GB/s	160GB/s
PCI-Express	Gen 2: 8 GB/s	Gen 3: 16 GB/s	

http://www.theregister.co.uk/2012/05/15/nvidia_kepler_tesla_gpu_revealed/page2.html

Kepler review

- Both Kepler and Fermi schedulers contain similar hardware units to handle scheduling functions, including, (a) register scoreboarding for long latency operations (texture and load), (b) inter-warp scheduling decisions (e.g., pick the best warp to go next among eligible candidates), and (c) thread block level scheduling (e.g., the GigaThread engine); however, Fermi's scheduler also contains a complex hardware stage to prevent data hazards in the math datapath itself. A multi-port register scoreboard keeps track of any registers that are not yet ready with valid data, and a dependency checker block analyzes register usage across a multitude of fully decoded warp instructions against the scoreboard, to determine which are eligible to issue.
- For Kepler, we realized that since this information is deterministic (the math pipeline latencies are not variable), it is possible for the compiler to determine up front when instructions will be ready to issue, and provide this information in the instruction itself. This allowed us to replace several complex and power-expensive blocks with a simple hardware block that extracts the pre-determined latency information and uses it to mask out warps from eligibility at the inter-warp scheduler stage.
- The short story here is that, in Kepler, the constant tug-of-war between control logic and FLOPS has moved decidedly in the direction of more on-chip FLOPS. The big question we have is whether Nvidia's compiler can truly be effective at keeping the GPU's execution units busy. Then again, it doesn't have to be perfect, since Kepler's increases in peak throughput are sufficient to overcome some loss of utilization efficiency. Also, as you'll soon see, this setup obviously works pretty well for graphics, a well-known and embarrassingly parallel workload. We are more dubious about this arrangement's potential for GPU computing, where throughput for a given workload could be highly dependent on compiler tuning. That's really another story for another chip on another day, though, as we'll explain shortly.

http://techreport.com/articles.x/22653

CUDA Software Development Kit

CUDA Optimized Libraries:
math.h, FFT, BLAS, …

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler (LLVM-based)

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Debugger
Profiler

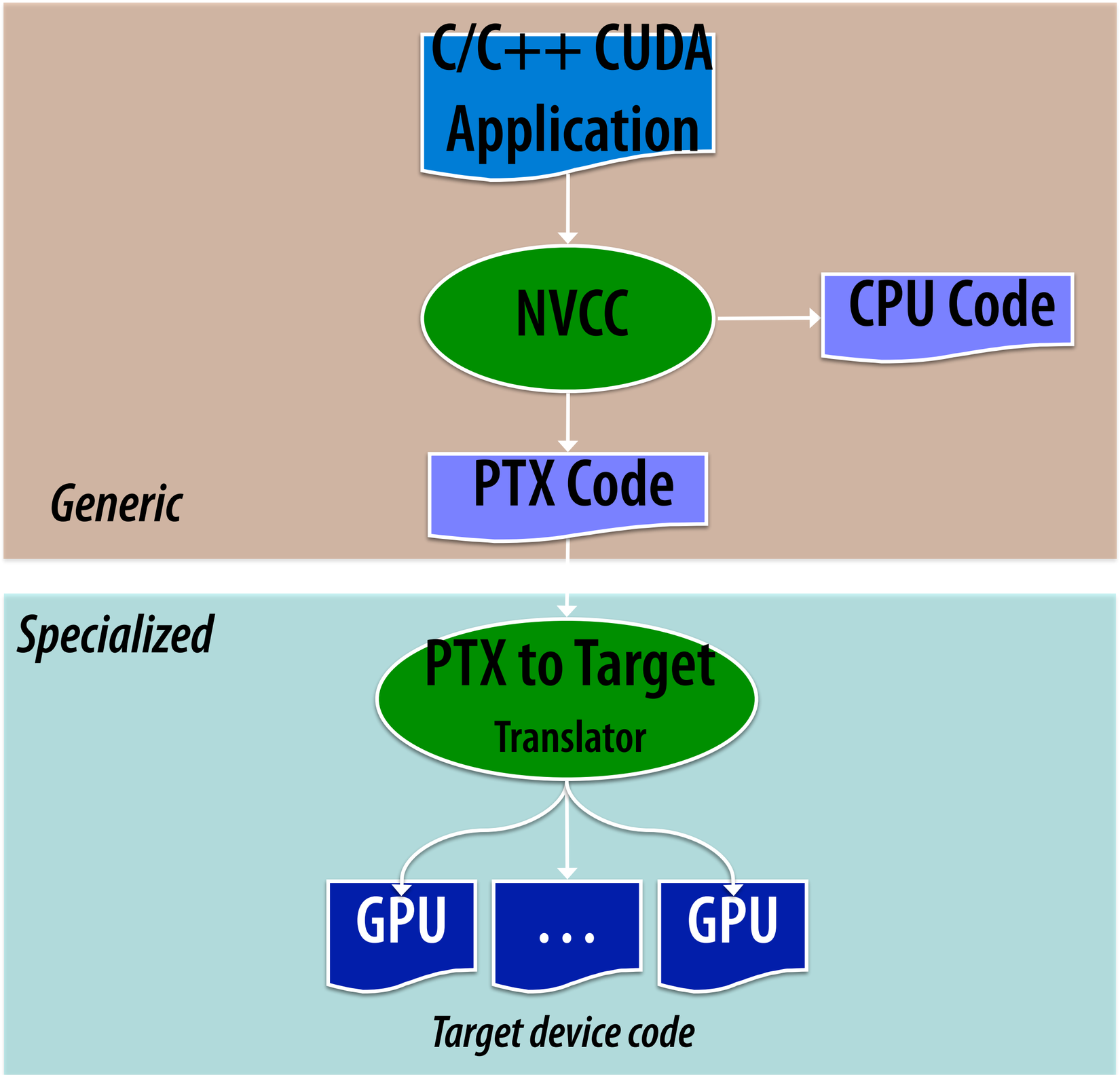
Standard C Compiler

SASS (GPU machine code)

GPU

CPU

Compiling CUDA for GPUs



OpenCL vs. CUDA

- OpenCL: Controlled by Khronos Group (consortium)
- Supported on:
  - GPUs (AMD, Intel, NVIDIA, Apple, Qualcomm, Samsung, etc.)
  - CPUs (Intel, AMD, etc.)
  - DSPs, FPGAs
- Very similar programming model (SPMD kernels, grid/blocks/threads, etc.)
- CUDA has single-program model. OpenCL separates programs into host vs. kernel code.
- Fewer {language, functional} features than CUDA.
- OpenCL programs have more boilerplate (harder for beginners)
- Generally *functionally* compatible across devices, but not necessarily *performance*-compatible

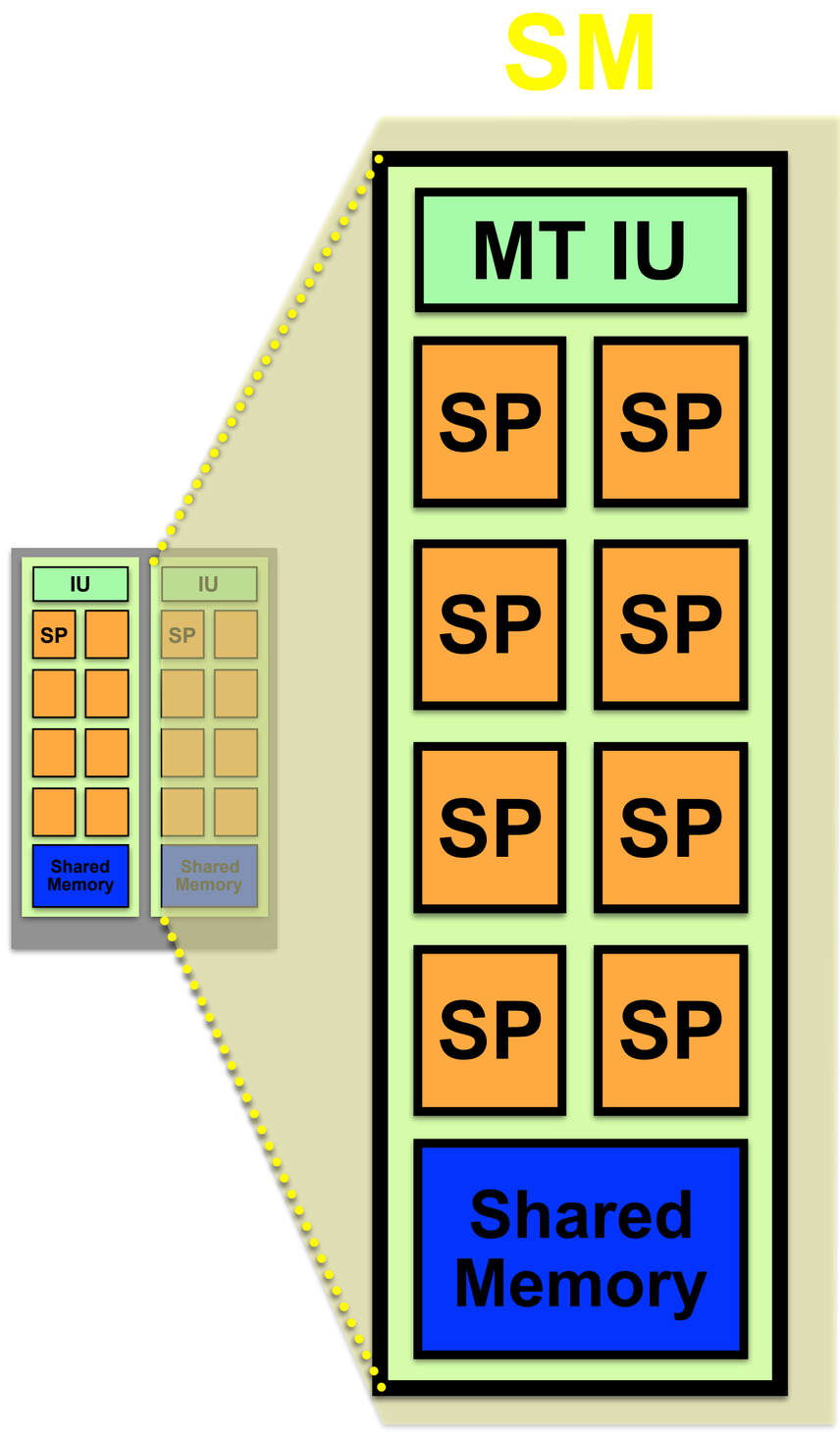
CUDA vs. OpenCL terminology

CUDA term	OpenCL term
GPU	Device
(Streaming) Multiprocessor	Compute unit
Scalar core	Processing element
Global memory	Global memory
Shared (per-block) memory	Local memory
Local memory (automatic, or local)	Private memory
Block	Work-group
Thread	Work-item

[Matt Harvey, "Experiences porting from CUDA to OpenCL"]

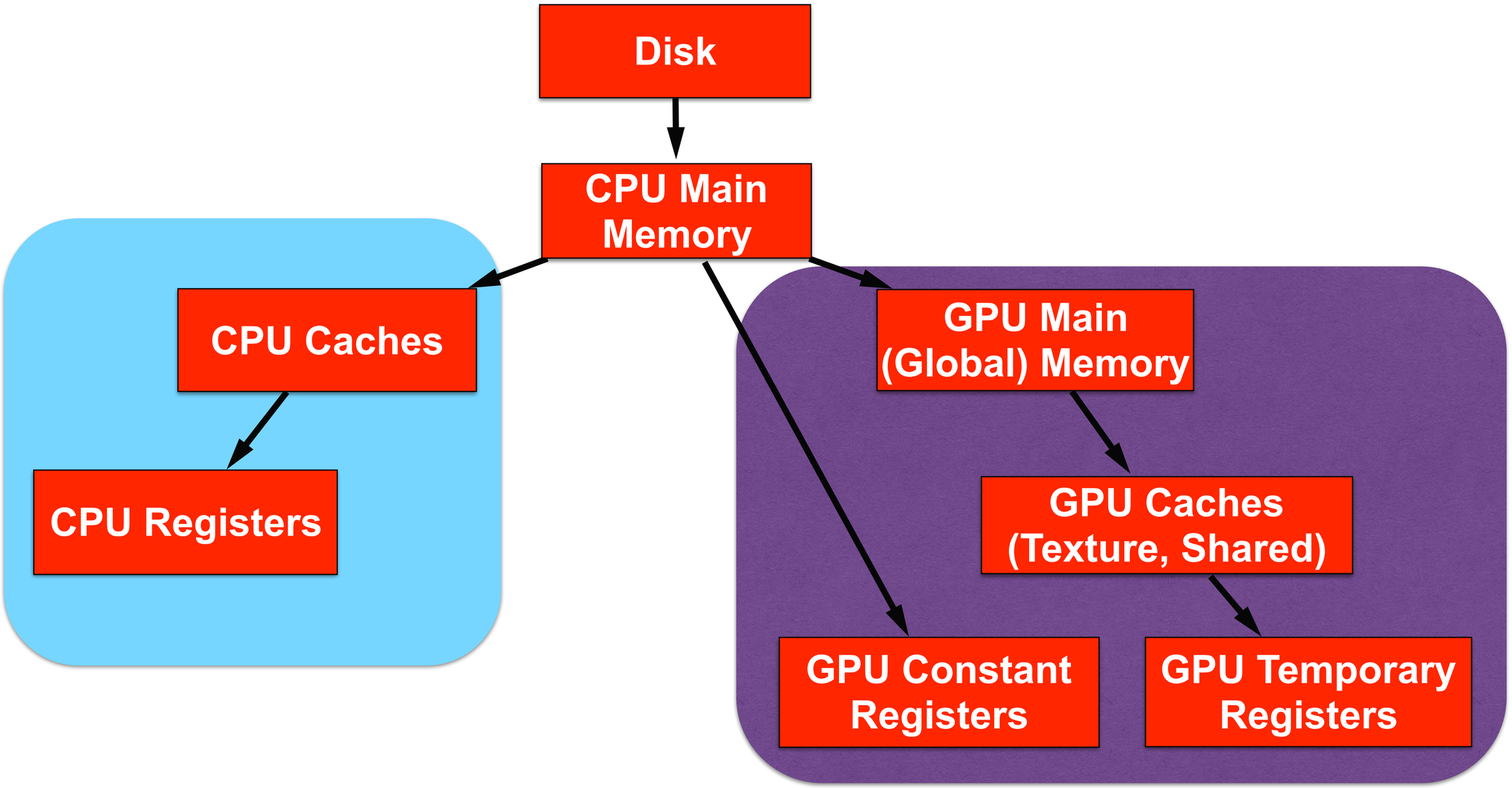
SM Multithreaded Multiprocessor

- Each SM runs a *block* of threads
- SMs have 8, 16, or 32 SP Thread Processors
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
- Scalar ISA
- Up to 768 threads, hw multithreaded (1024 in newer hw)
- 16KB Shared Memory (64KB in newer hw)
  - Concurrent threads share data
  - Low latency load/store
- 32 elements run at same time (SIMD) as a *warp*



What do you think goes in a single thread processor? How would you design it?

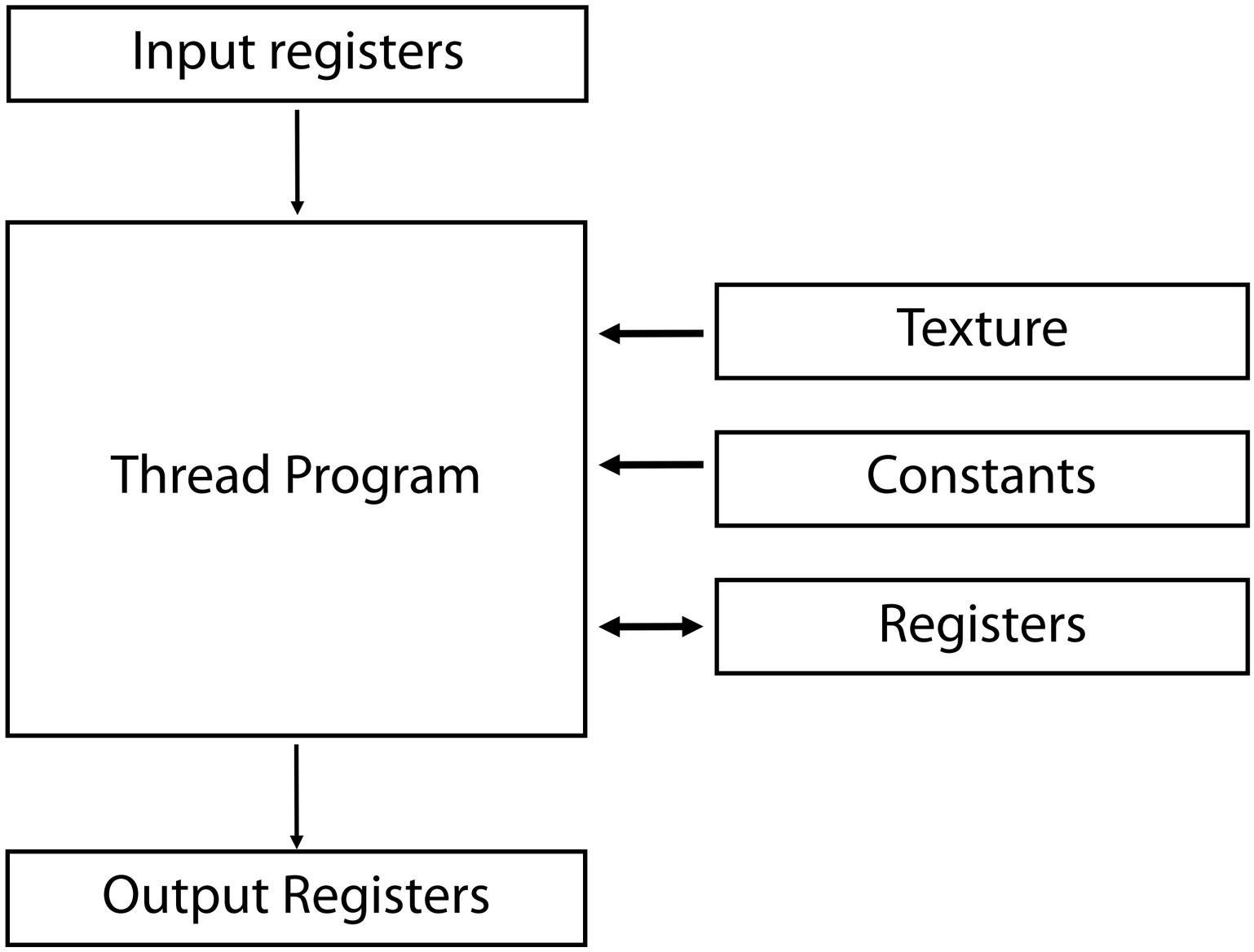
CPU/GPU Memory Hierarchy



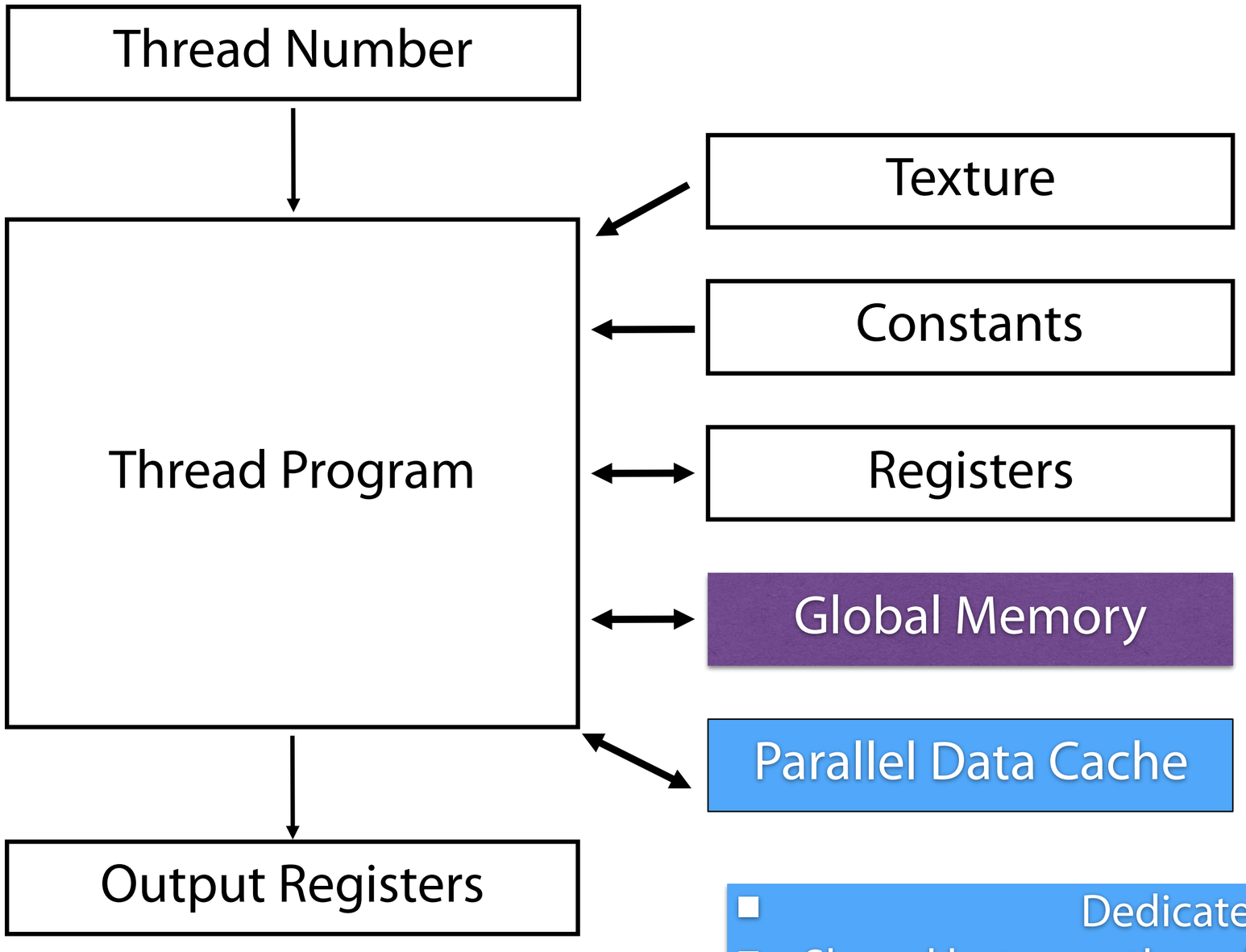
What about hybrid (CPU/GPU same die) processors?

Graphics Programs

- **Features**
  - **Millions of instructions**
  - **Full integer and bit instructions**
  - **No limits on branching, looping**



GPU Memory Spaces

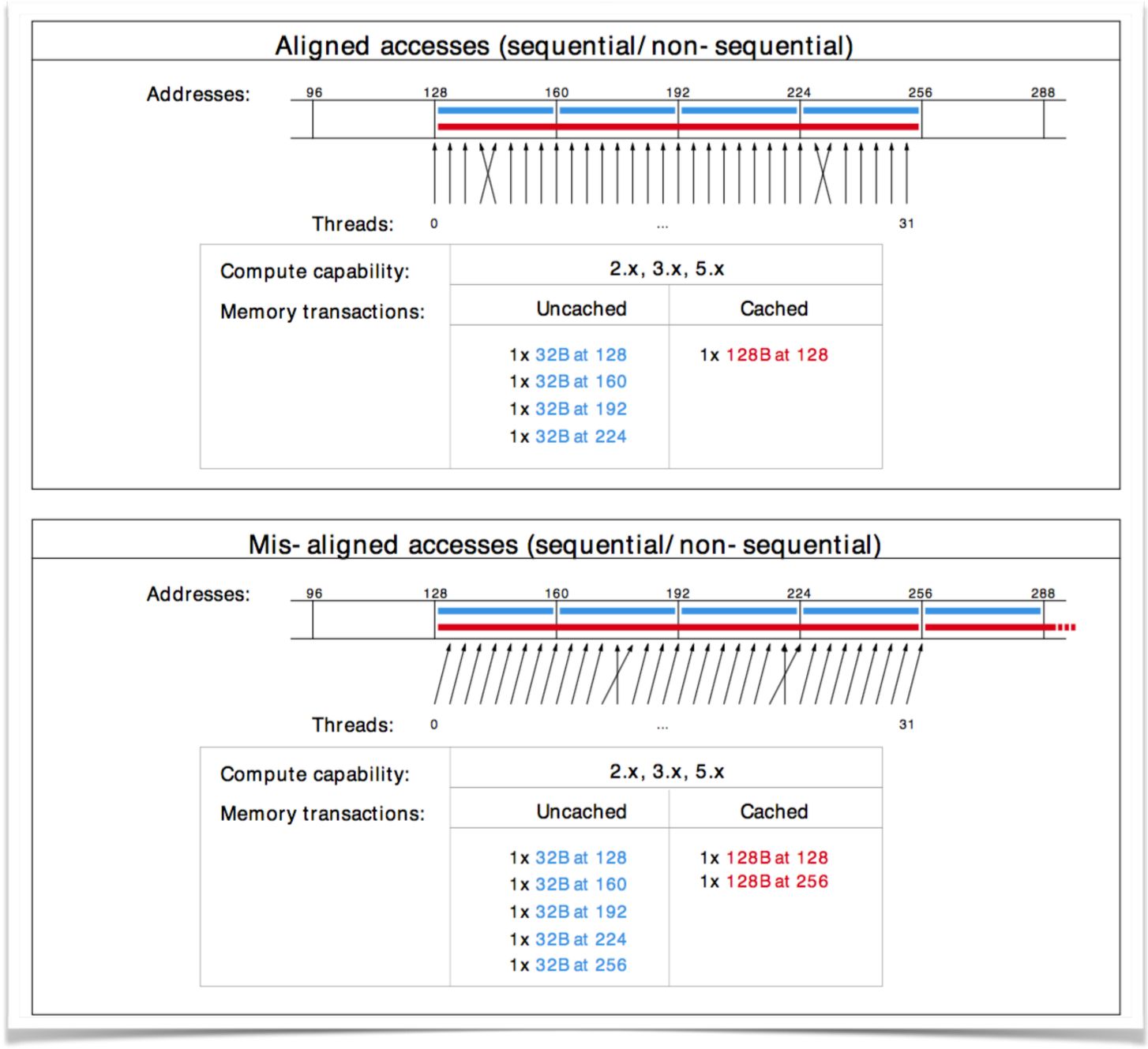


- 1D, 2D, or 3D thread ID allocation
- Fully general load/store to GPU memory: Scatter/Gather
- Programmer flexibility on how memory is accessed
- Untyped, not limited to fixed texture types
- Pointer support

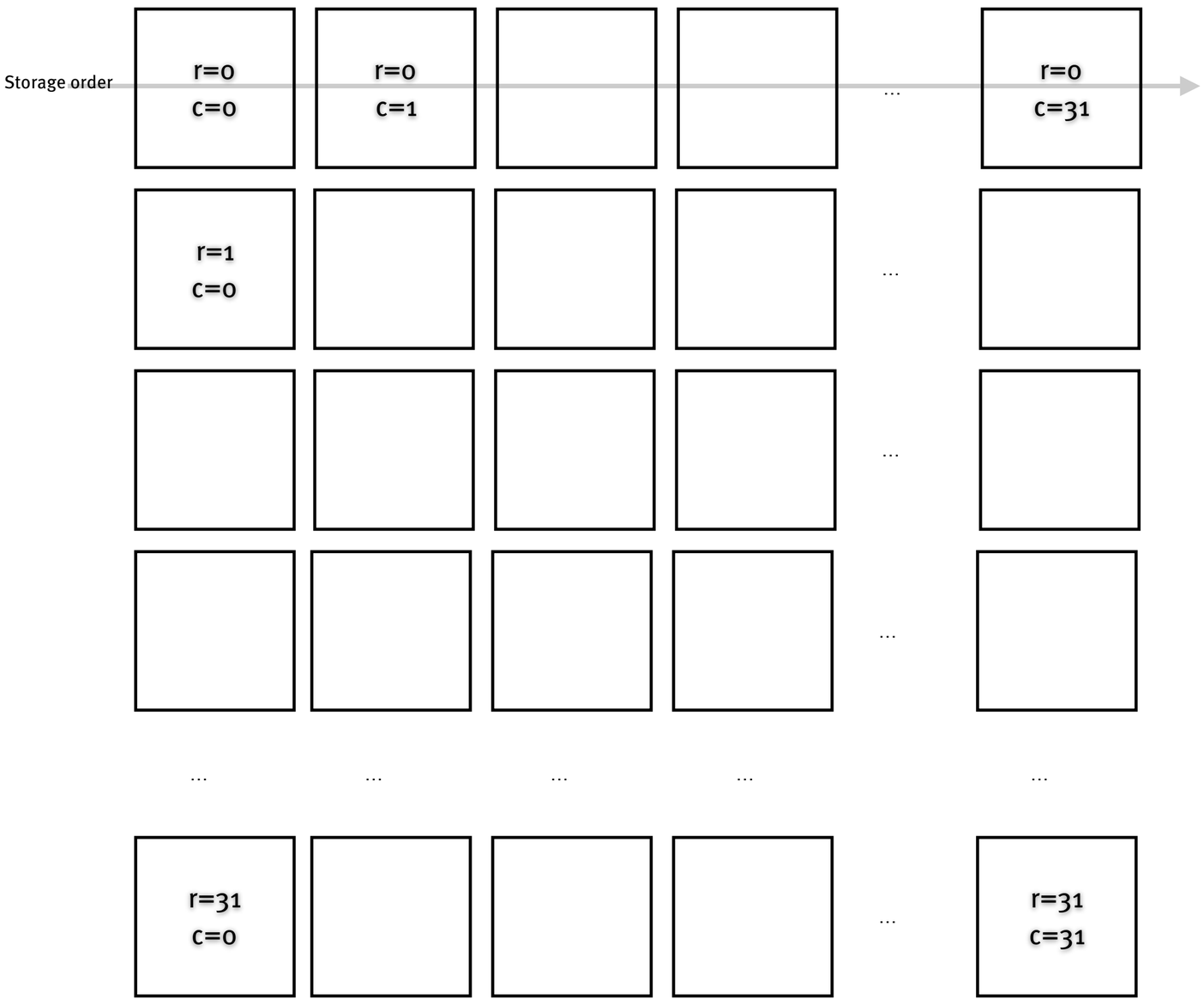
- Dedicated on-chip memory
- Shared between threads for inter-thread communication
- Explicitly managed
- (Much) closer to register speed than memory speed

Memory Coalescing in Global Memory

- **Your mental model should be that:**
  - **Once you touch anything in a memory chunk, every other item in that chunk is free**
  - **Your memory cost is the cost of all chunks touched**



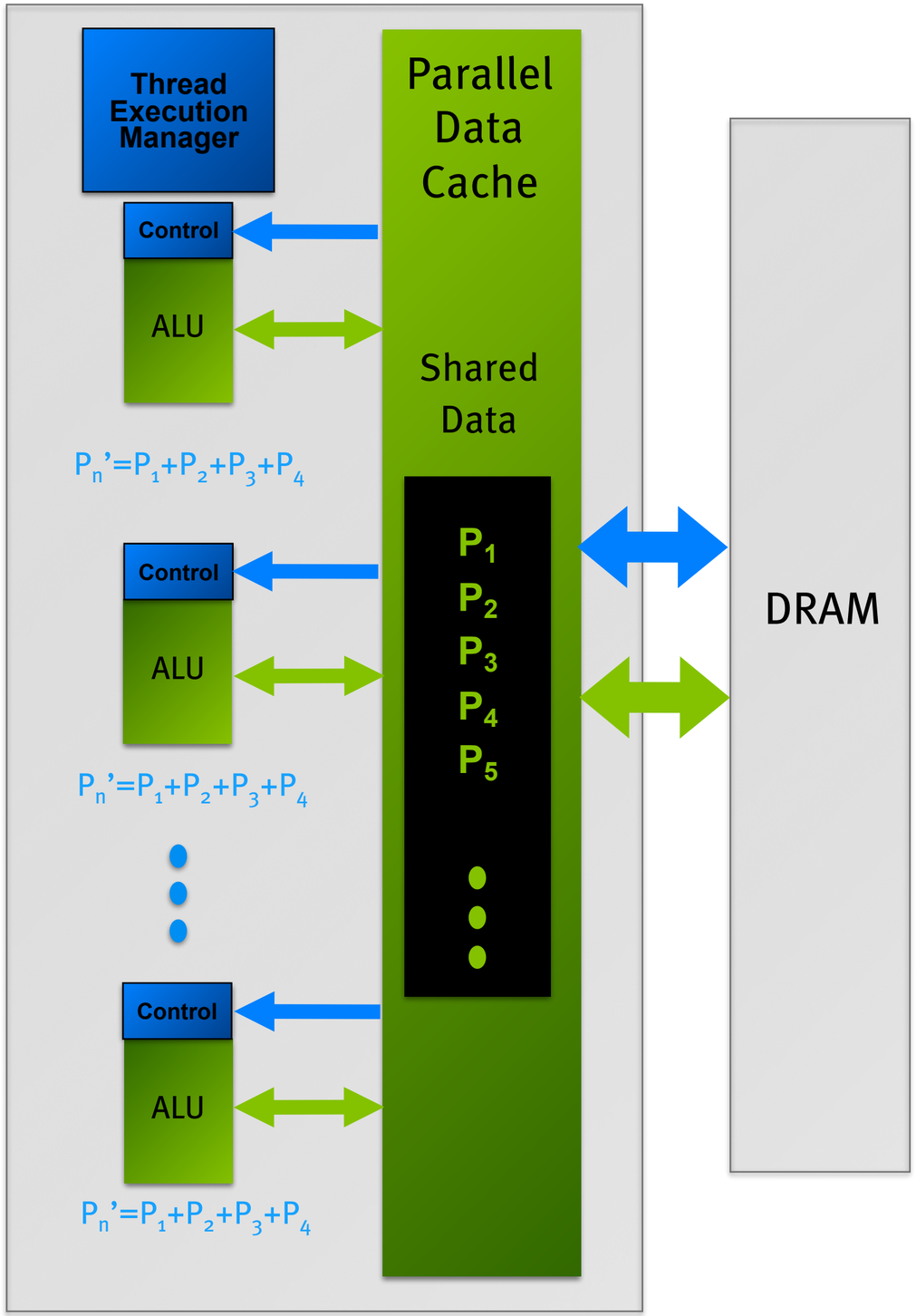
Memory Coalescing Example



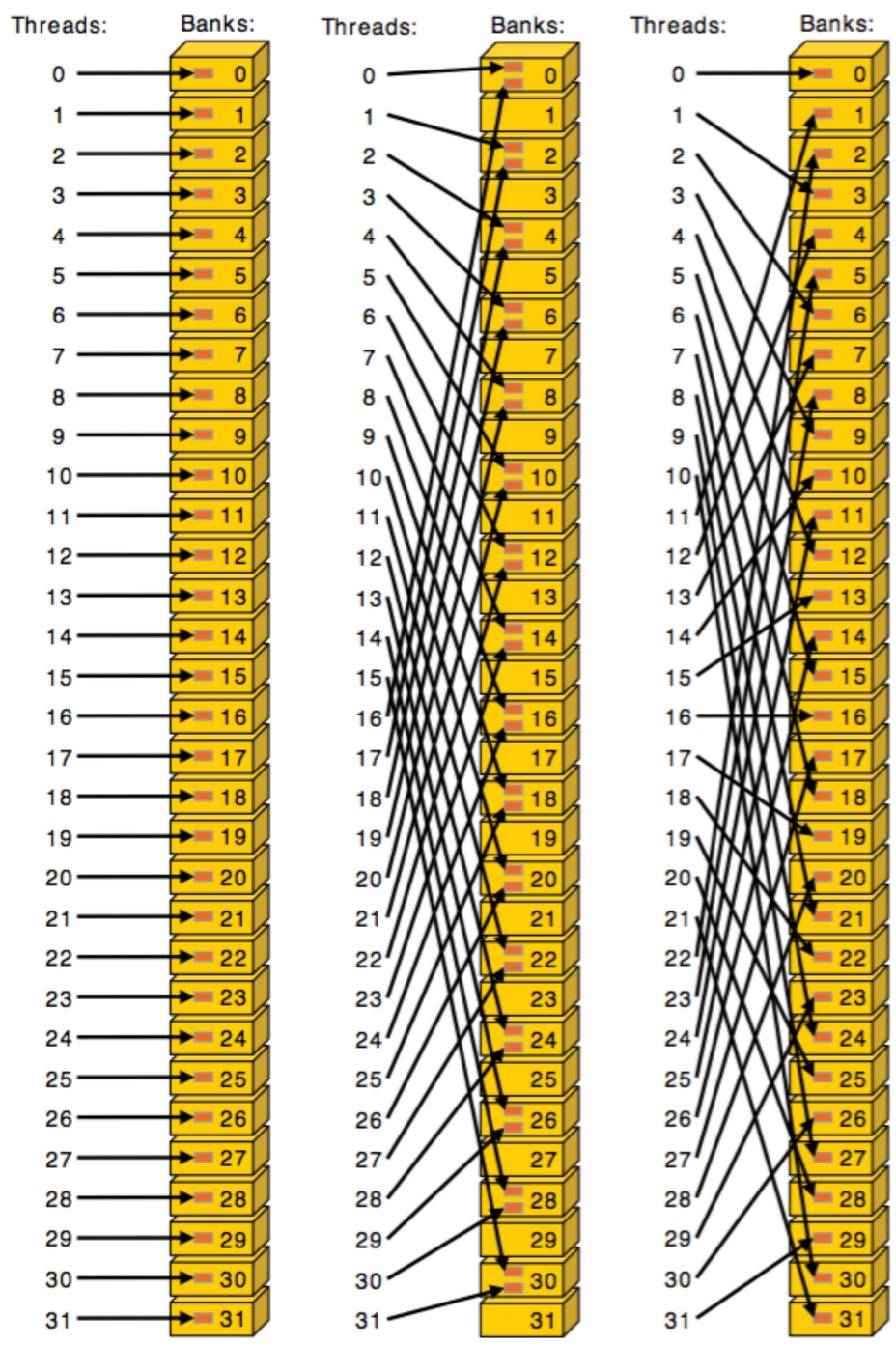
- Read a 32x32 block of memory, stored in row-major order, from DRAM
- Read it row-major: thread i reads (r=i/32, c=i%32):
  - Fully coalesced
  - 32 memory transactions
- Read it column-major: thread i reads (r=i%32, c=i/32):
  - Fully uncoalesced
  - 32x32 memory transactions

Parallel Data Cache

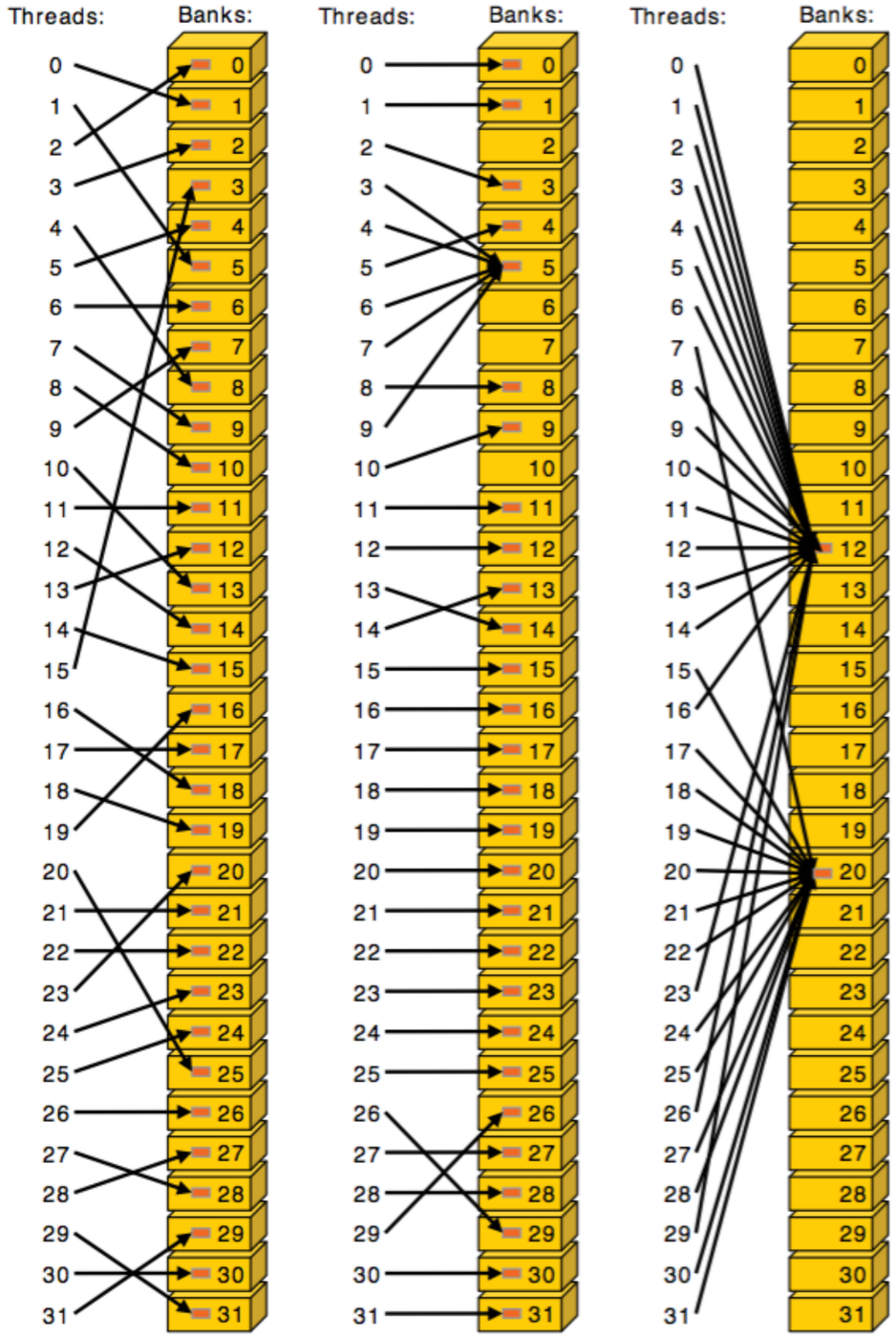
- Addresses a fundamental problem of stream computing
- Bring the data closer to the ALU
- Stage computation for the parallel data cache
- Minimize trips to external memory
- Share values to minimize overfetch and computation
- Increases arithmetic intensity by keeping data close to the processors
- User managed generic memory, threads read/write arbitrarily



Shared memory has banks



Left
Linear addressing with a stride of one 32-bit word (no bank conflict).
Middle
Linear addressing with a stride of two 32-bit words (two-way bank conflict).
Right
Linear addressing with a stride of three 32-bit words (no bank conflict).



Left
Conflict-free access via random permutation.
Middle
Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right
Conflict-free broadcast access (threads access the same word within a bank).

This parallel data cache seems pretty cool. Since we can have tens of thousands of threads active at any time, why can't all of them communicate through it?

What did we learn about the hardware?

- Base unit of the hardware is a SIMD (lockstep) warp of 32 threads
  - Avoid branching within a warp
- GPU "SMs" (cores) run blocks of threads
  - SMs run warps when they are ready.
  - Warps may be from the same or different blocks.
  - The hardware schedules blocks onto SMs if sufficient SM resources.
- Memory hierarchy, {small, fast} -> {big, slow}: registers, shared memory + L1, L2, global memory (DRAM)

Hardware design points

- {More, weaker} cores vs. {fewer, more powerful} cores (SMs)
- {More, weaker} thread processors vs. {fewer, more powerful}
- Registers per thread
- Managing branch divergence & warp width
- Caches, and user-managed vs. hardware-managed
- Allowable communication mechanisms between {threads, warps, blocks, GPUs}
- Atomics/synchronization and where they can be used

Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate, runs to completion
- Several blocks can reside concurrently on one multiprocessor (SM)
  - Control limitations (~current GPUs, compute version 5.0):
    - At most 32 concurrent blocks per SM
    - At most 2048 concurrent threads per SM
  - Number is further limited by SM resources
    - Register file (64–128k) is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks

What is a thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled
- CUDA threads might be physical threads
  - as on NVIDIA GPUs
- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU
  - Interesting research on this topic

What is a thread block?

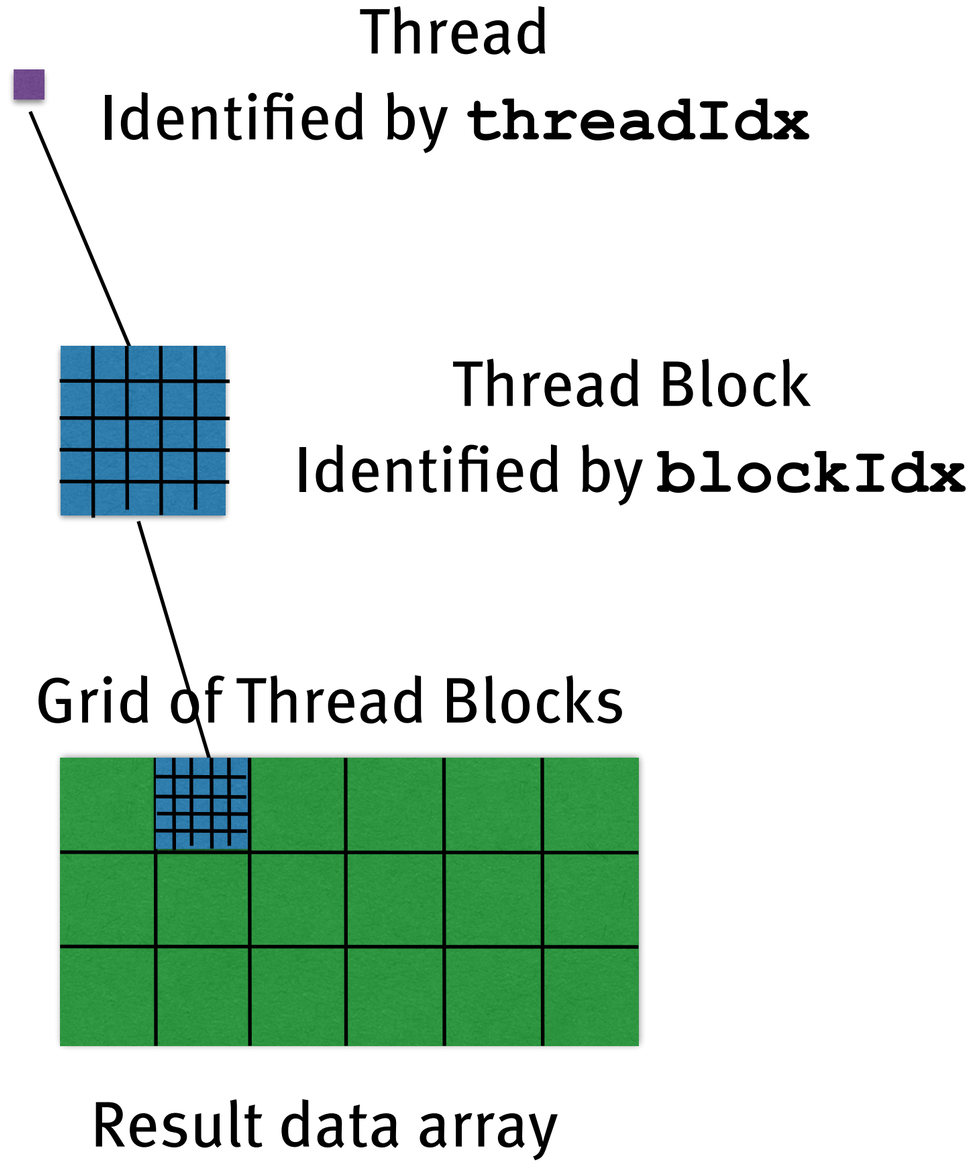
- Thread block = virtualized multiprocessor
  - freely choose processors to fit data
  - freely customize for each kernel launch
- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want
- Thread blocks of kernel must be independent tasks
  - program valid for any interleaving of block executions

Levels of parallelism

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels

Preview of CUDA details (next lecture)

- Multiple levels of parallelism
- Thread block
  - Up to 2048 threads per block
  - Communicate through shared memory
  - Threads guaranteed to be resident
    - `threadIdx, blockIdx`
    - `__syncthreads()`
- Grid of thread blocks
    - `f<<<nblocks, nthreads>>>(a,b,c)`



Summary: CUDA Design Goals

- Scale to 100s of cores, 1000s of parallel threads
- Let programmers focus on parallel algorithms
  - not mechanics of a parallel programming language
- Enable heterogeneous systems (i.e., CPU+GPU)
  - CPU & GPU are separate devices with separate DRAMs