Lecture 6:

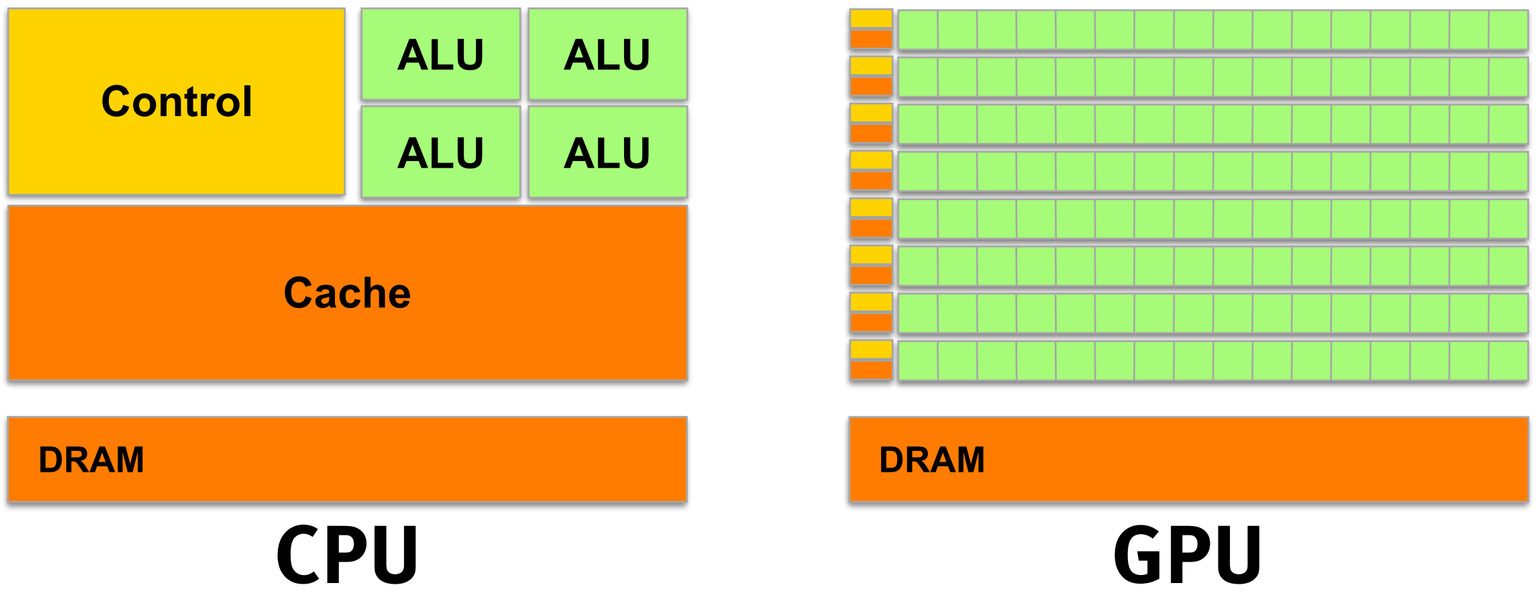
GPU Computing from a Graphics HW Perspective

EEC 277, Graphics Architecture
John Owens, UC Davis, Winter 2017

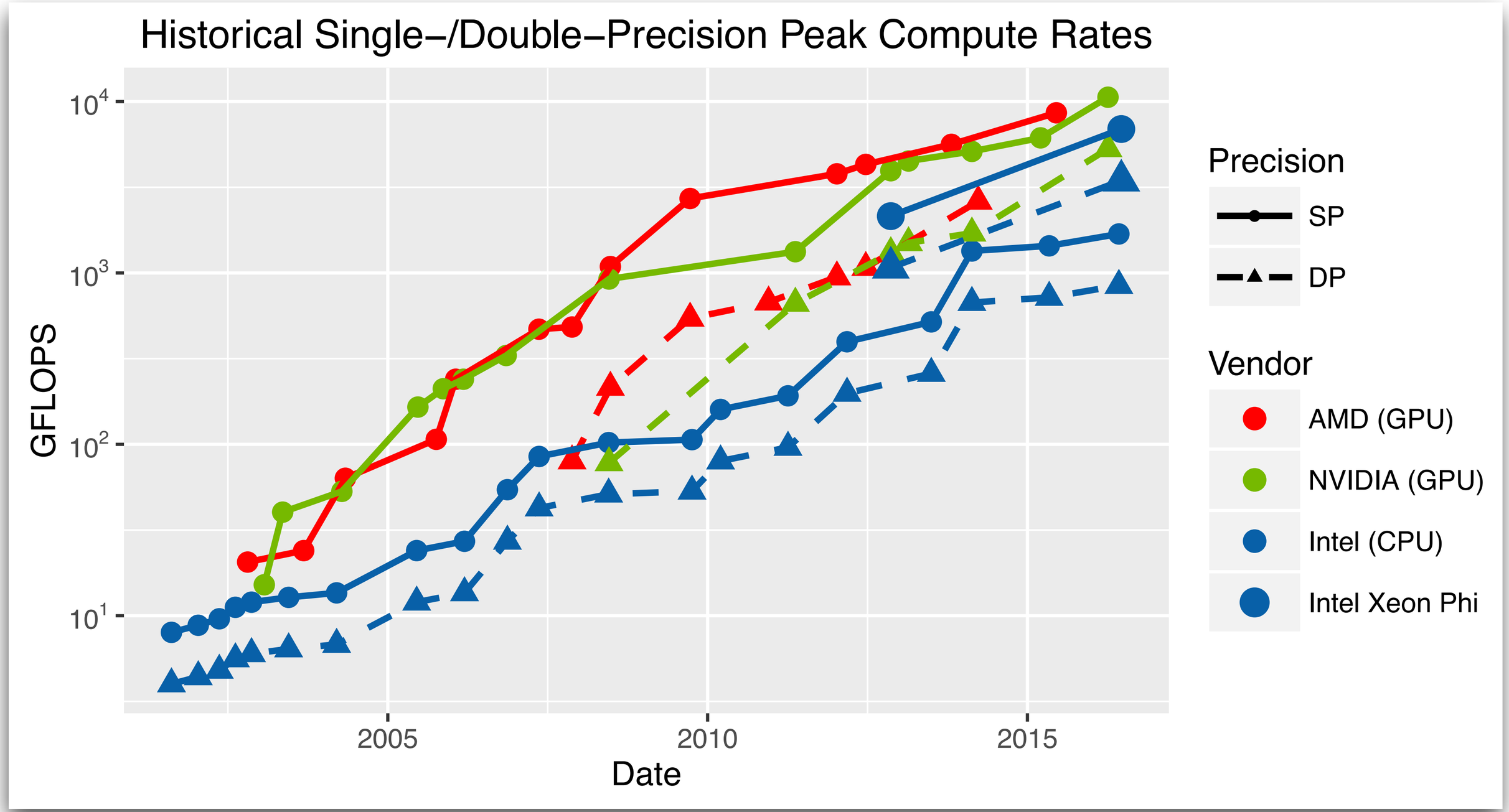
“If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?”—Seymour Cray

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



Recent GPU Performance Trends



Motivation: The Potential of GPGPU

- In short:
  - The power and flexibility of GPUs makes them an attractive platform for general-purpose computation
  - Example applications range from in-game physics simulation to conventional computational science
    - NVIDIA architect John Danskin (GH08) described the workload in a modern game: "AI (suitable for GPUs); physics (suitable for GPUs); graphics (suitable for GPUs); and a 'perl script, which can be run on a serial CPU that takes five square millimeters and consumes one percent of a processor die""
  - Goal: make the inexpensive power of the GPU available to developers as a sort of computational coprocessor

GPUs Are Fast

- Roughly 3–6x CPU bandwidth and computation
- Impressive microbenchmark performance
  - Raw math: 10+ TFLOPS sustained (NVIDIA Titan X)
  - Raw bandwidth: 512 GB per second (AMD R9 Fury X)
- Impressive useful application performance

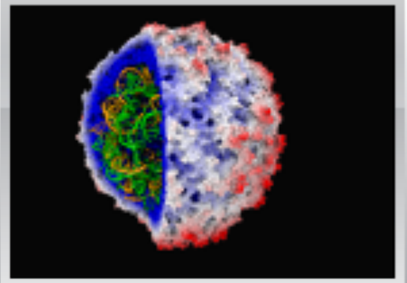
Successes on NVIDIA GPUs

[courtesy David Luebke, NVIDIA]

Successes on NVIDIA GPUs

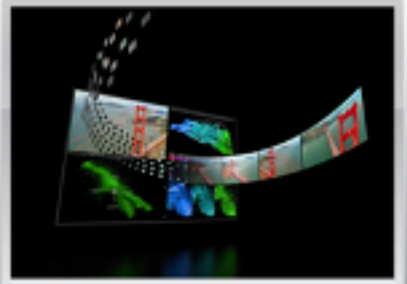
146X

Interactive visualization of volumetric white matter connectivity



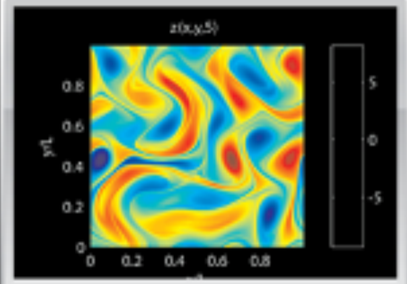
36X

Ionic placement for molecular dynamics simulation on GPU



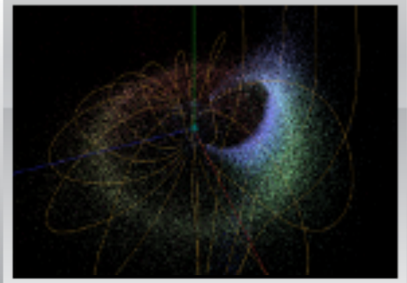
19X

Transcoding HD video stream to H.264



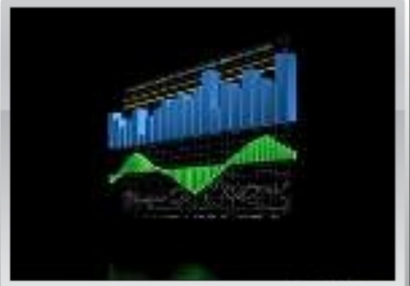
17X

Fluid mechanics in Matlab using .mex file CUDA function



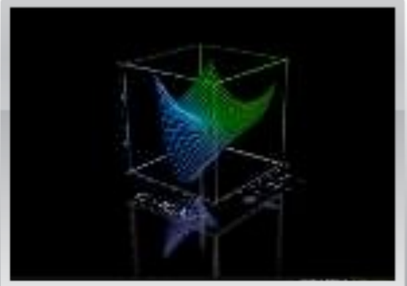
100X

Astrophysics N-body simulation



149X

Financial simulation of LIBOR model with swaptions



47X

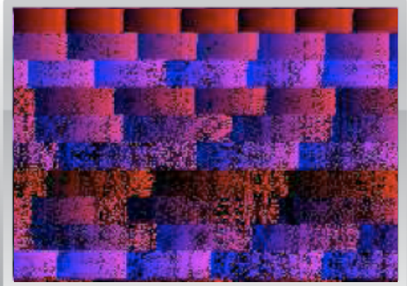
GLAME@lab: an M-script API for GPU linear algebra

20X

Ultrasound medical imaging for cancer diagnostics

24X

Highly optimized object oriented molecular dynamics

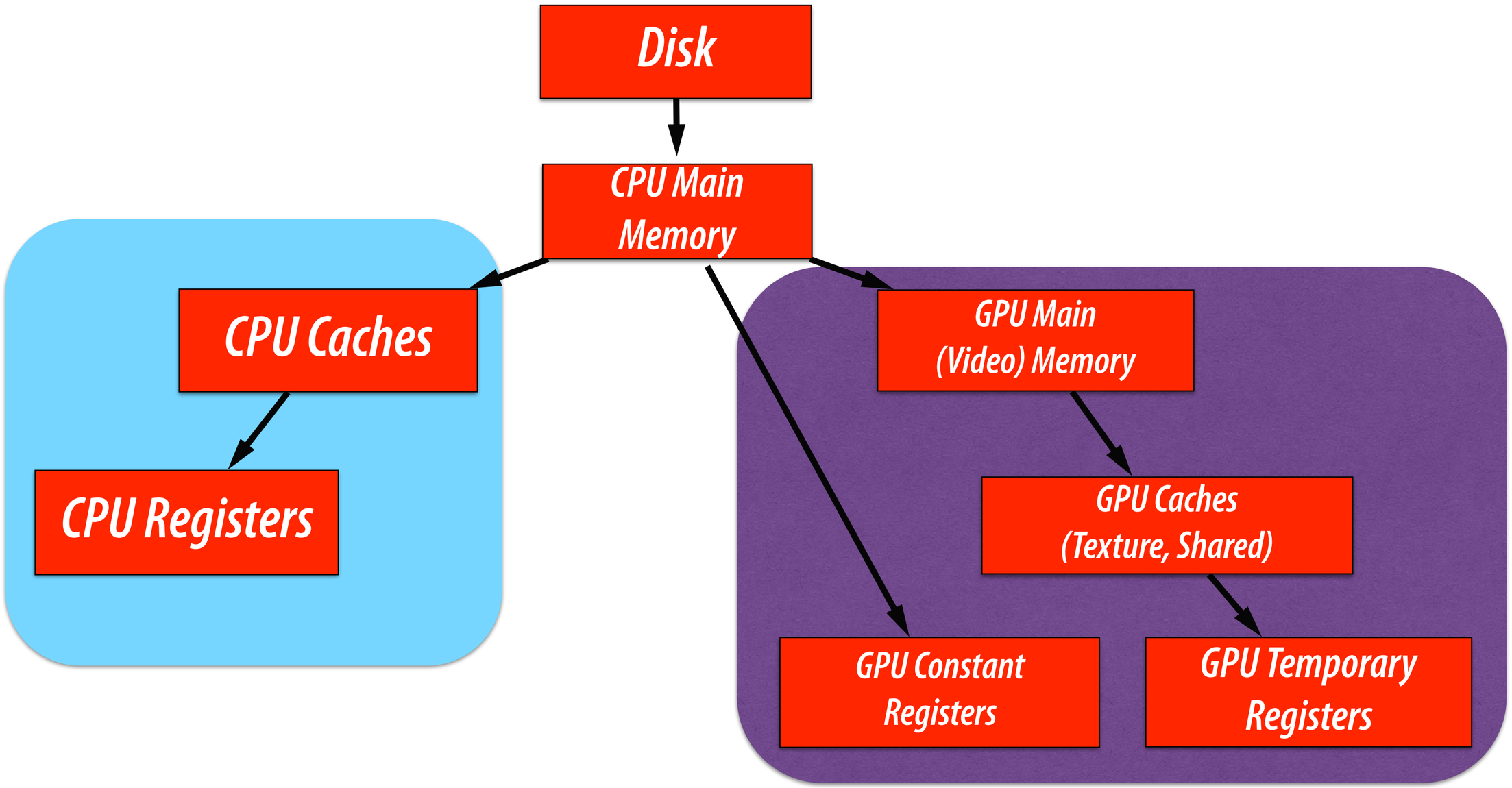


30X

Cmatch exact string matching to find similar proteins and gene sequences

[courtesy David Luebke, NVIDIA]

CPU/GPU Memory Hierarchy



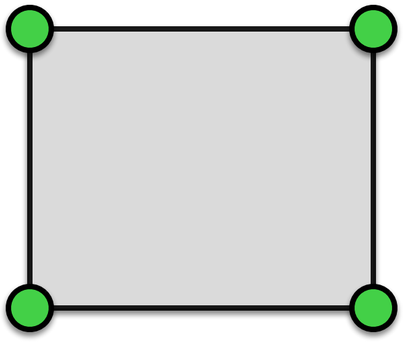
- What about hybrid (CPU/GPU same die) processors?

Structuring a GPU Program

- CPU assembles input data
- CPU transfers data to GPU (GPU "main memory" or "device memory")
- 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.
- What lessons can we draw from this sequence of operations?

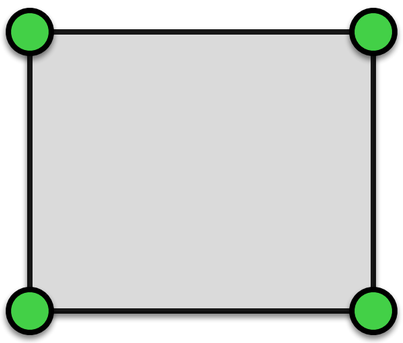
Programming a GPU for Graphics

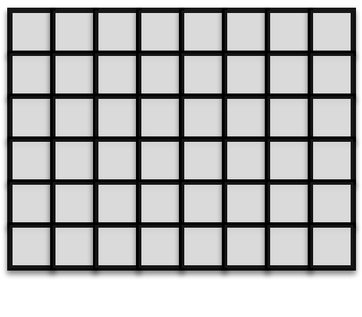
Programming a GPU for Graphics



- Application specifies geometry -> rasterized

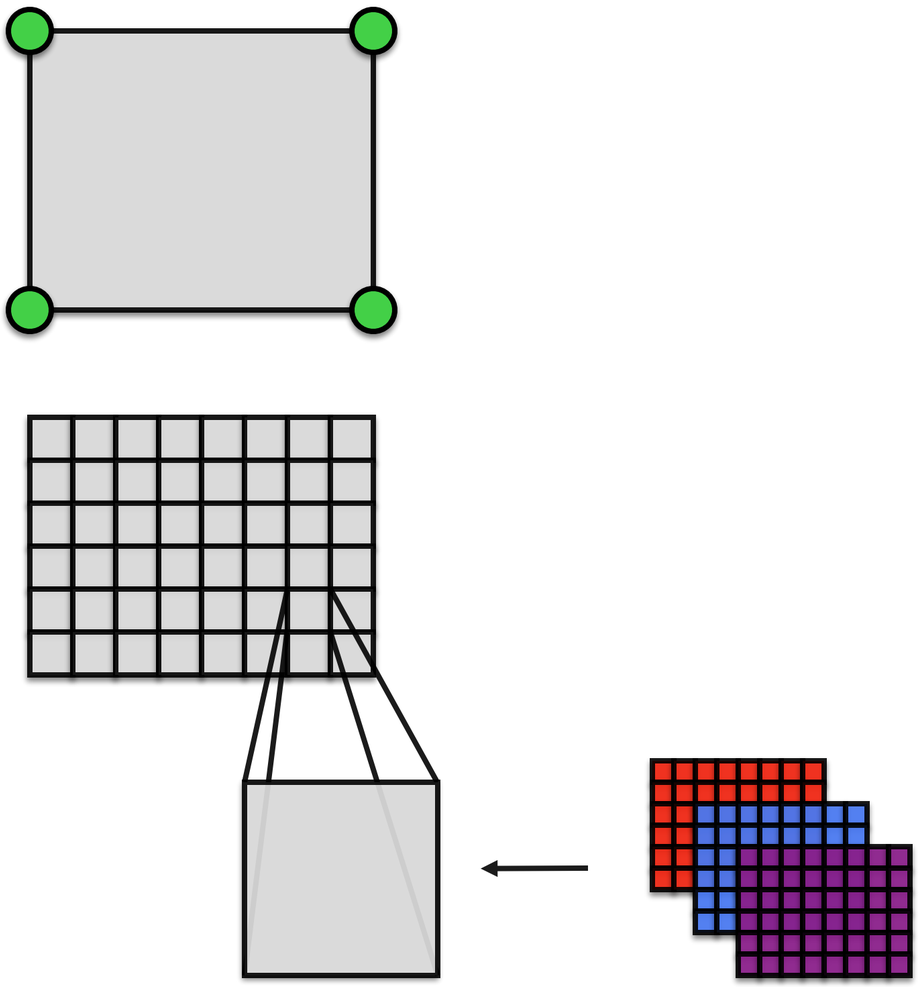
Programming a GPU for Graphics





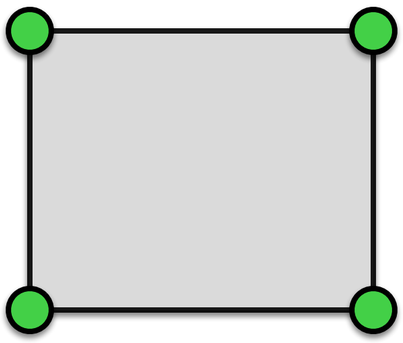
- Application specifies geometry -> rasterized
- Each fragment is shaded w/ SIMD program

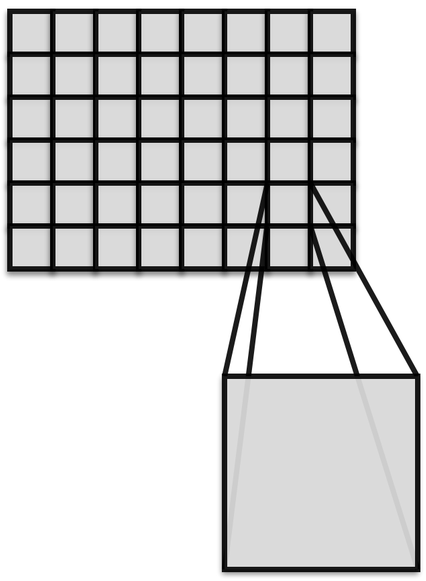
Programming a GPU for Graphics

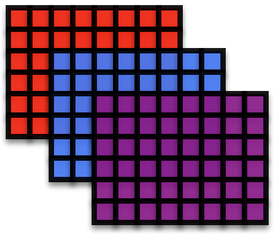


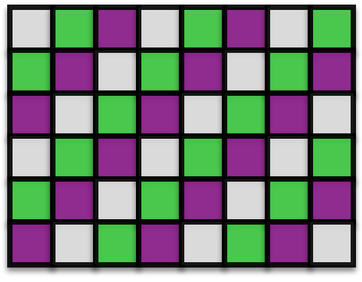
- Application specifies geometry -> rasterized
- Each fragment is shaded w/ SIMD program
- Shading can use values from texture memory

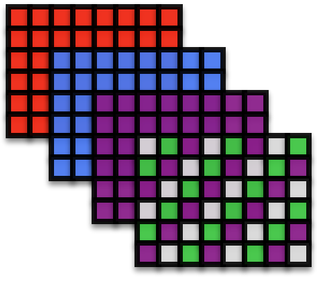
Programming a GPU for Graphics





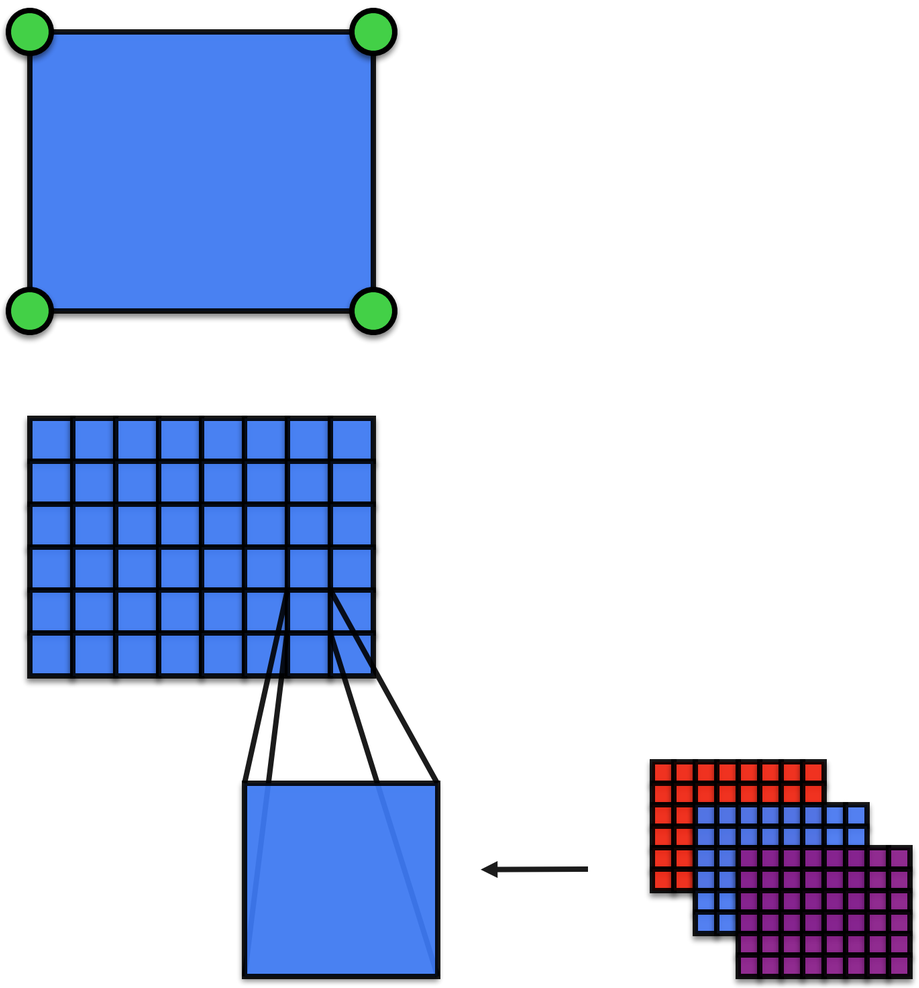






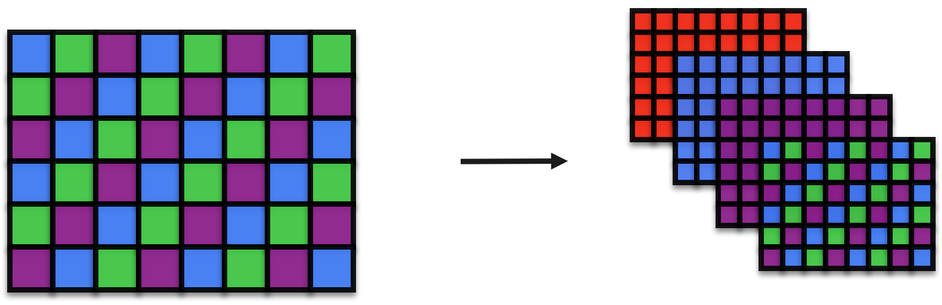
- Application specifies geometry -> rasterized
- Each fragment is shaded w/ SIMD program
- Shading can use values from texture memory
- Image can be used as texture on future passes

Programming a GPU for GP Programs (Old-School)



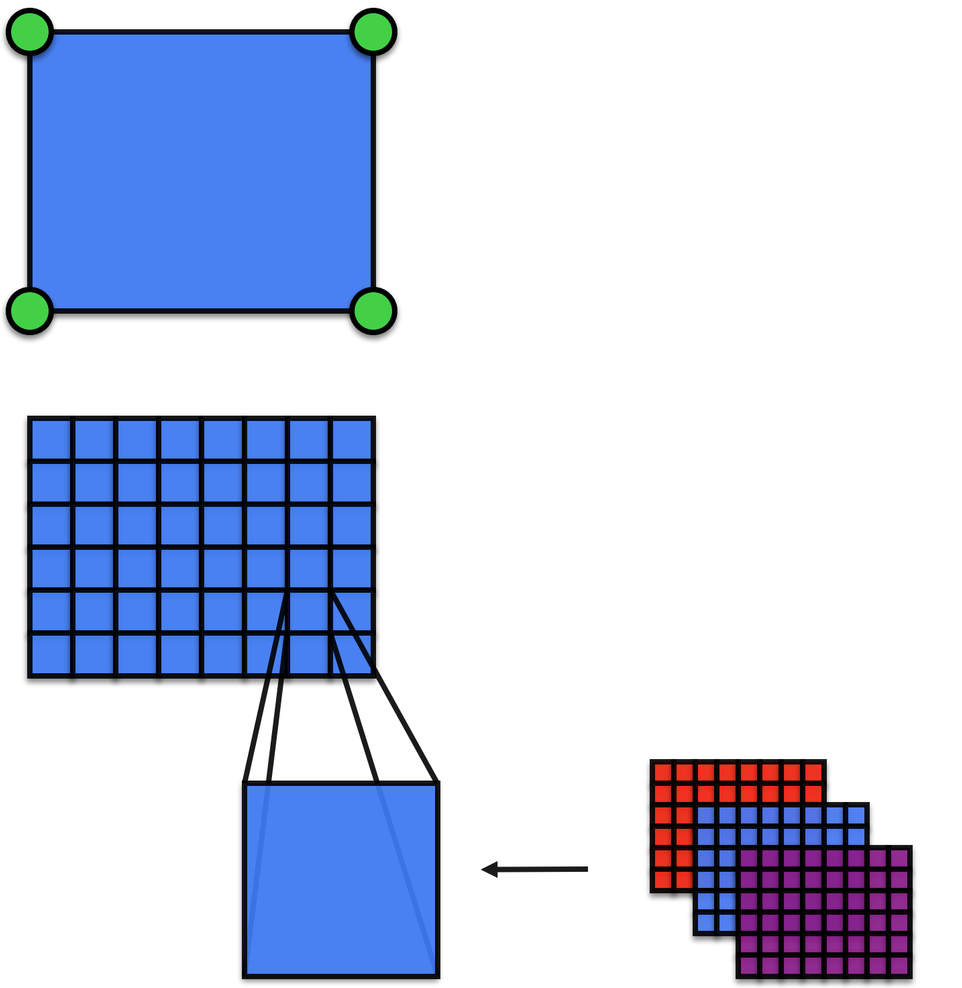
- Draw a screen-sized quad
- Run a SIMD program over each fragment

- "Gather" is permitted from texture memory

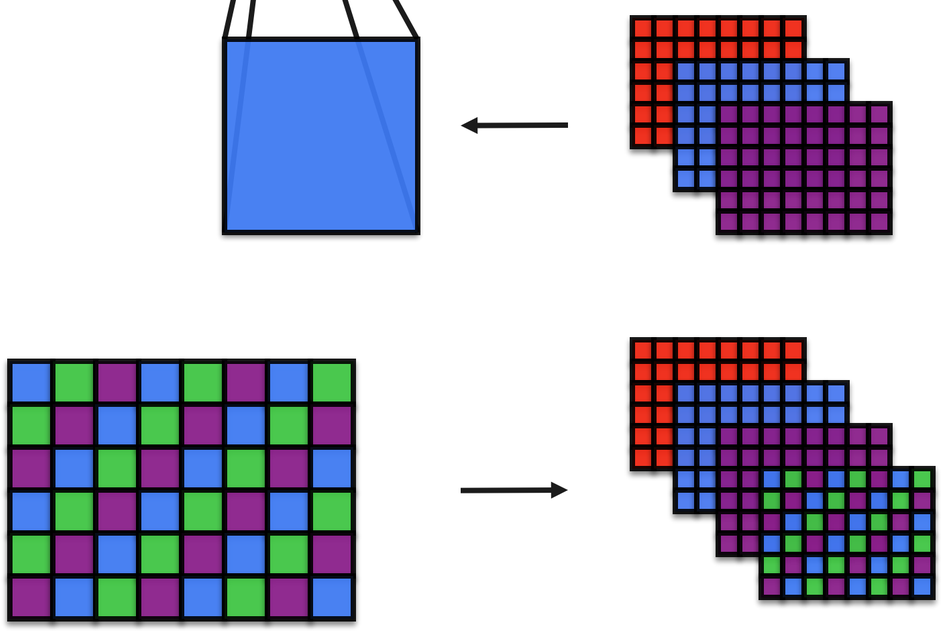


- Resulting buffer can be treated as texture on next pass

Programming a GPU for GP Programs (New School)



- Define a computation domain over elements
- Run a SIMD/SPMD program over each element/thread



- Global memory supports both gather and scatter

- Resulting buffer can be used in other general-purpose kernels or as texture

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
- Today:
  - GPU hardware
  - CUDA programming environment

Big Idea #1

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

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

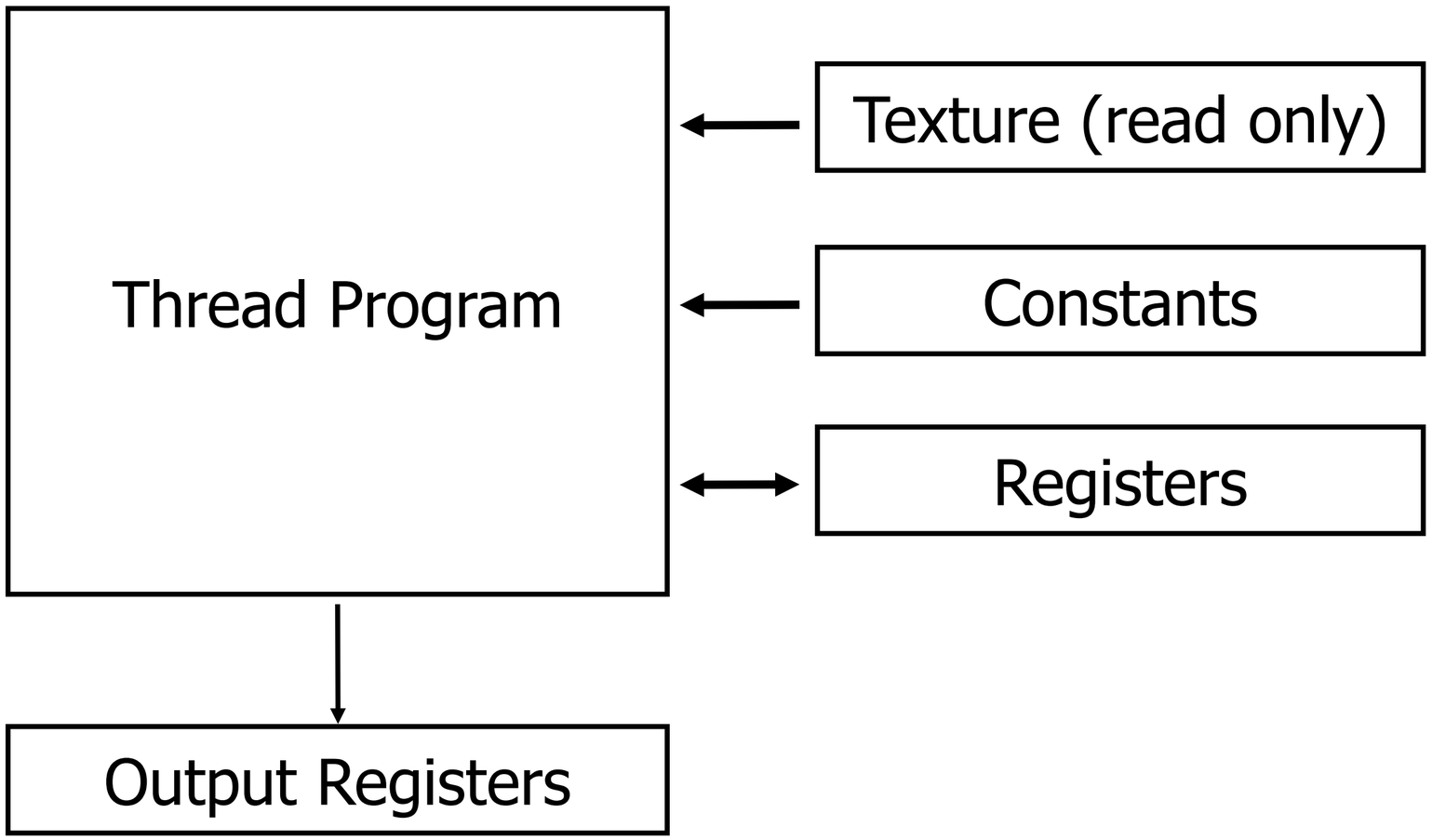
- Parallel portions of an application are executed on the device as kernels
  - One SIMT kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - 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

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

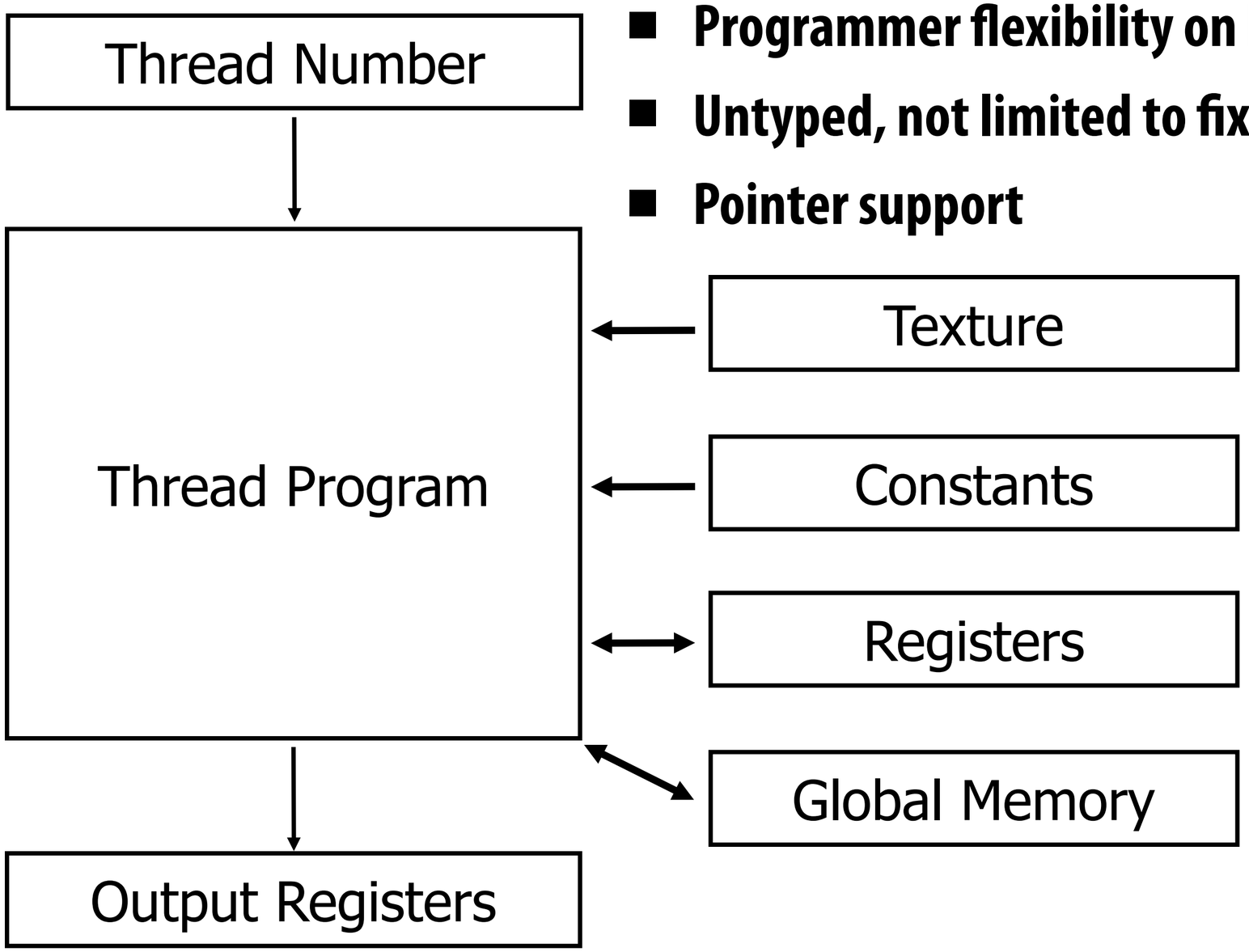
Graphics Programs

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



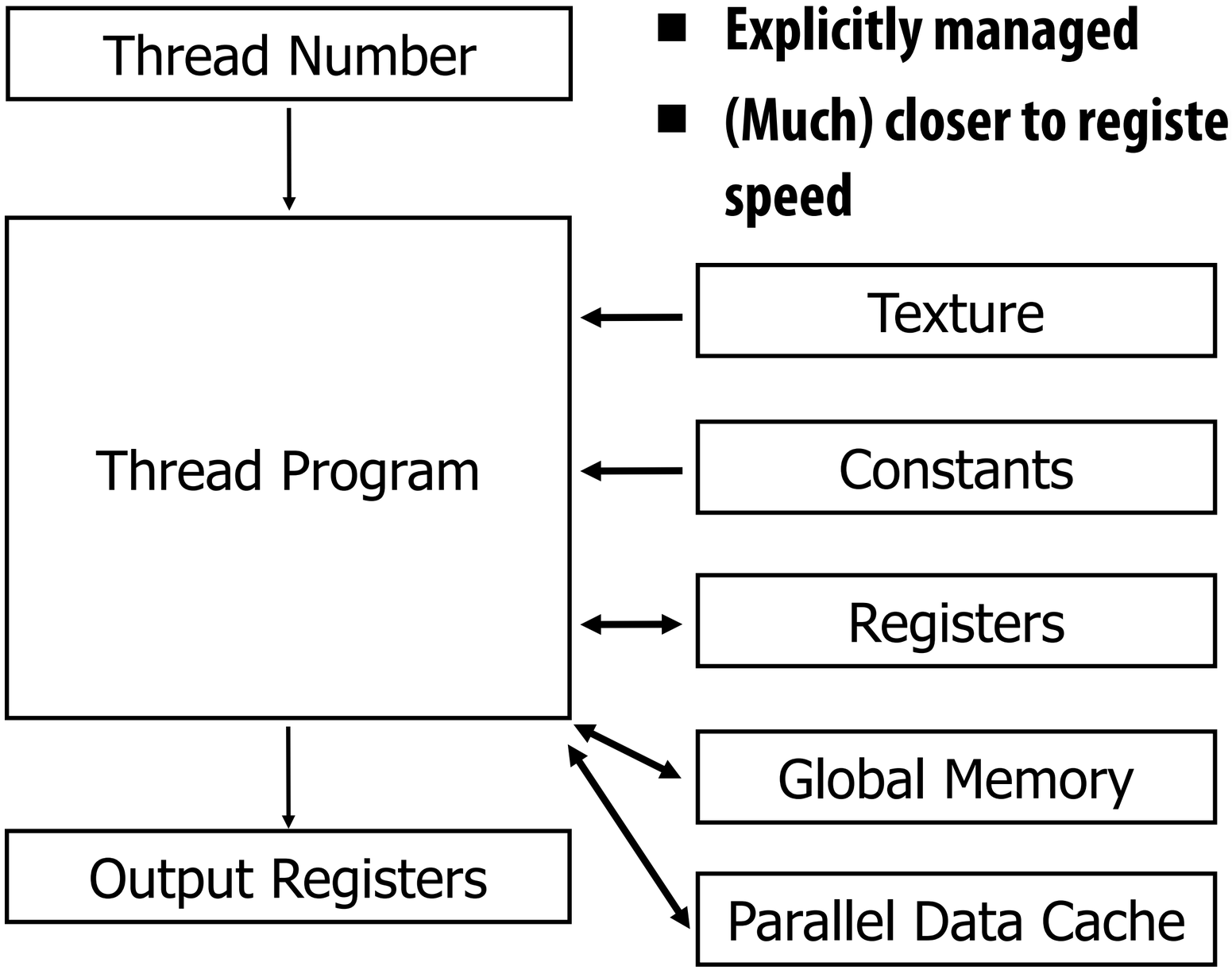
General-Purpose Programs

- 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

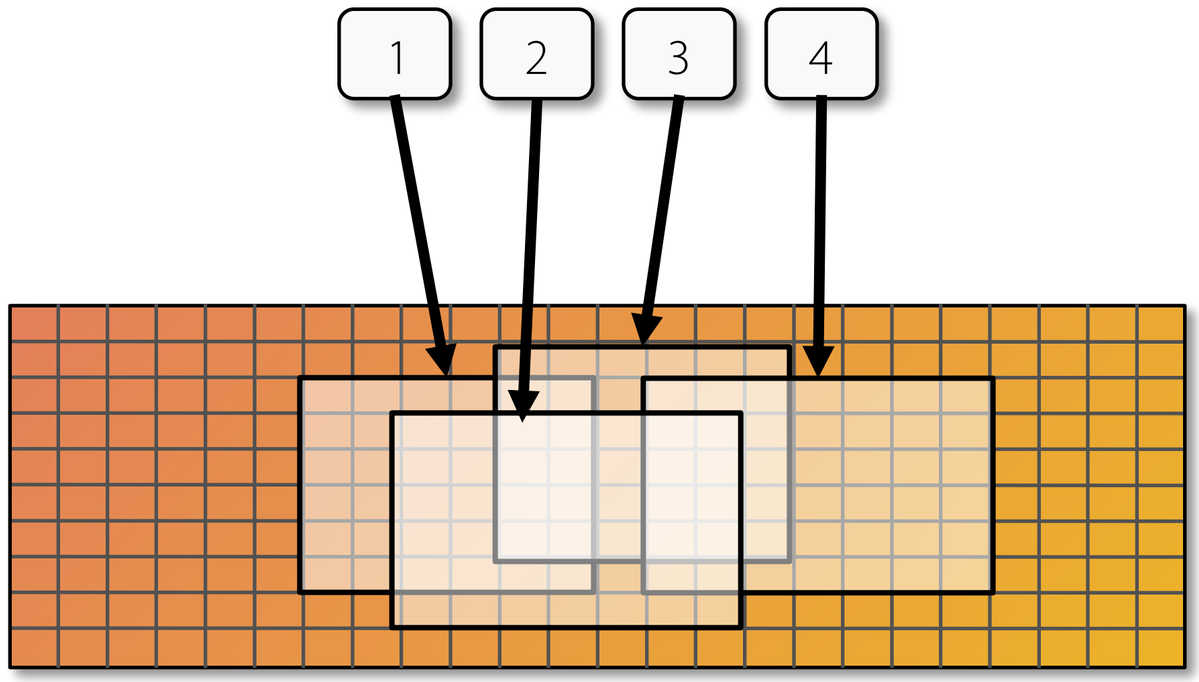


Parallel Data Cache

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

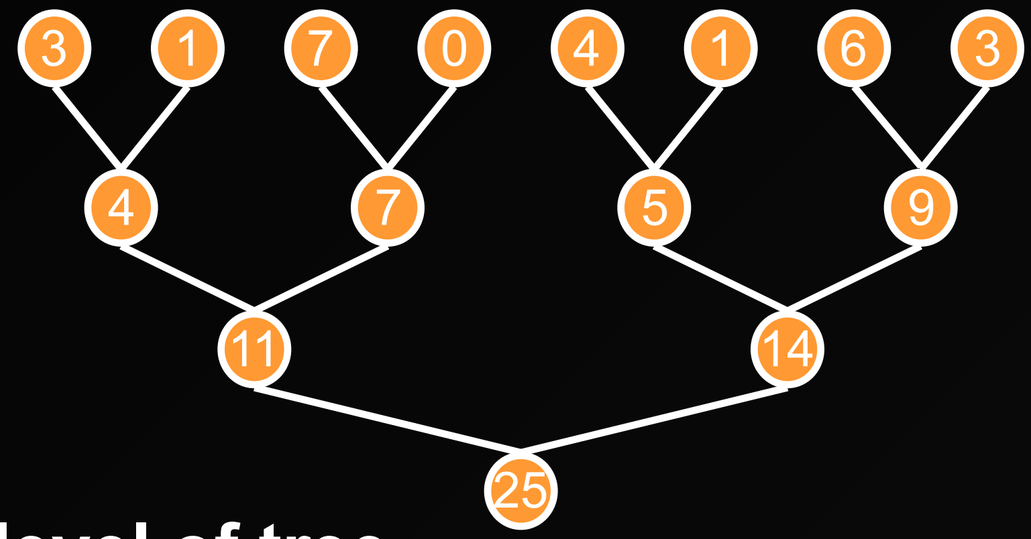


Reuse (texture & general-purpose)



Texture data

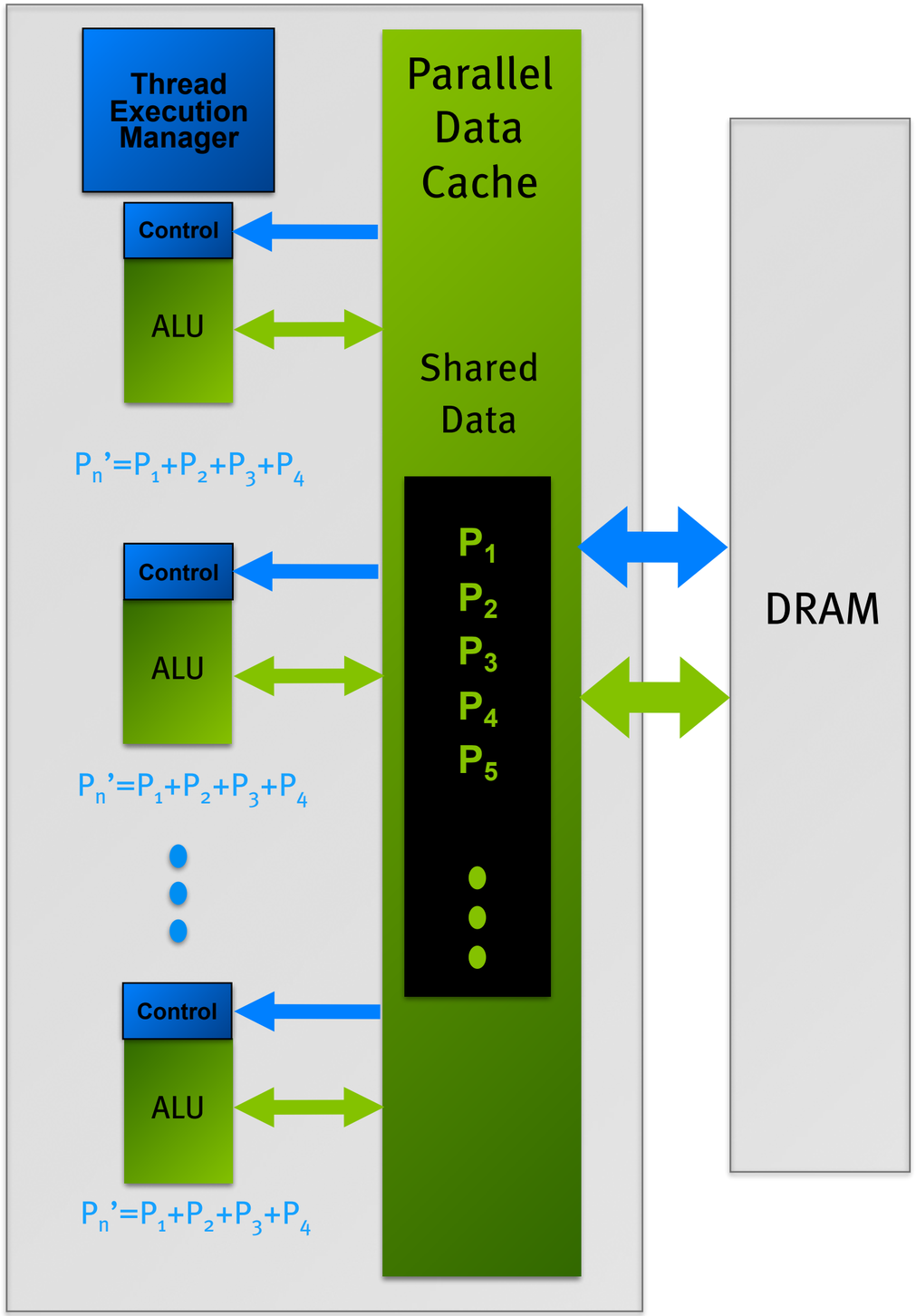
Tree-Based Parallel Reductions



- Traditional algorithm
  - Requires synchronization at each level of tree
  - Synchronized through main memory, making it …
  - … completely bandwidth-bound
  - Memory writes and reads are off-chip, no reuse of intermediate sums
- CUDA solves this by exposing on-chip per-block shared memory
  - Reduce blocks of data in shared memory to save bandwidth

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



Parallel execution through cache

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?

Compute shaders

If I Know GLSL,
What Do I Need to Do Differently to Write a Compute Shader?

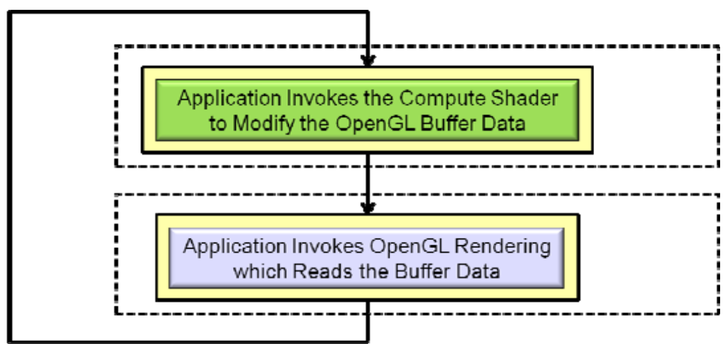
Not much:

1. A Compute Shader is created just like any other GLSL shader, except that its type is GL_COMPUTE_SHADER (duh). You compile it and link it just like any other GLSL shader program.

2. A Compute Shader must be in a shader program all by itself. There cannot be vertex, fragment, etc. shaders in there with it. (why?)

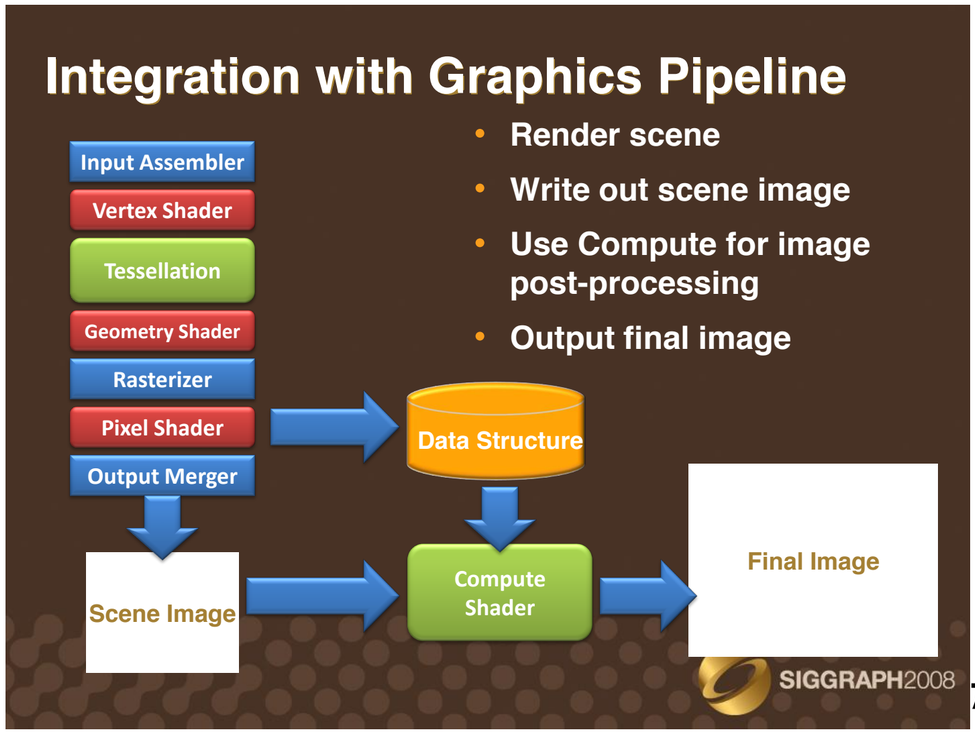
3. A Compute Shader has access to uniform variables and buffer objects, but cannot access any pipeline variables such as attributes or variables from other stages. It stands alone.

4. A Compute Shader needs to declare the number of work-items in each of its work-groups in a special GLSL *layout* statement.



[courtesy Mike Bailey, Oregon State]

[courtesy Chas Boyd, Microsoft]



Compute shaders

```
#version 430
layout(local_size_x = 1, local_size_y = 1) in;
layout(rgba32f, binding = 0) uniform image2D img_output;
// sample compute shader from
// http://antongerdelan.net/opengl/compute.html
void main() {
  // base pixel colour for image
  vec4 pixel = vec4(0.0, 0.0, 0.0, 1.0);
  // get index in global work group i.e x,y position
  ivec2 pixel_coords = ivec2(gl_GlobalInvocationID.xy);

  //
  // interesting stuff happens here later
  //

  // output to a specific pixel in the image
  imageStore(img_output, pixel_coords, pixel);
}
```

What Do I Need to I

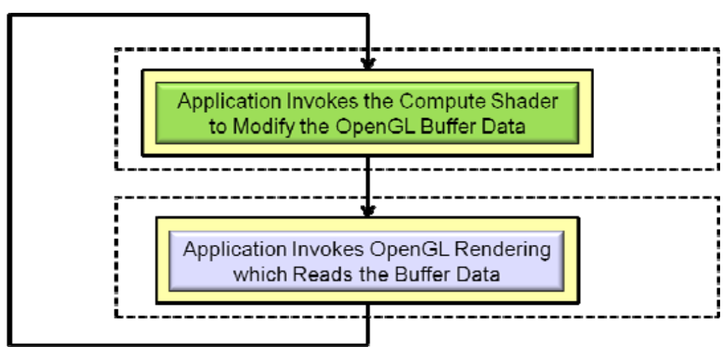
Not much:

1. A Compute Shader is
   its type is GL_COMPU
   just like any other GL

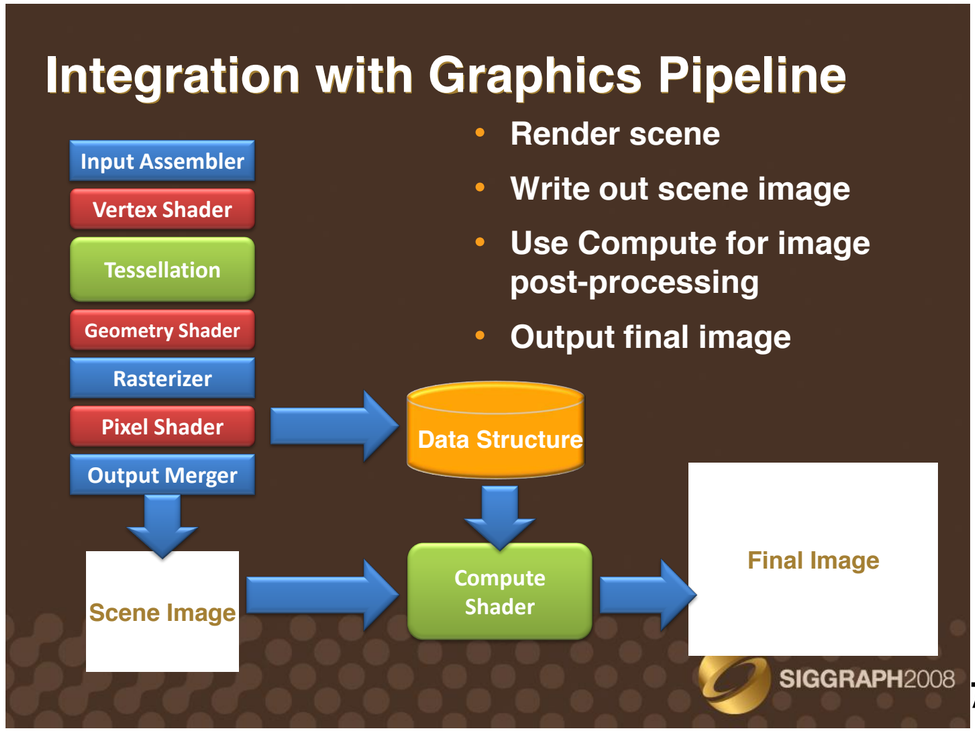
2. A Compute Shader m
   cannot be vertex, frag

3. A Compute Shader has access to uniform variables and buffer objects, but cannot access any pipeline variables such as attributes or variables from other stages. It stands alone.

4. A Compute Shader needs to declare the number of work-items in each of its work-groups in a special GLSL *layout* statement.



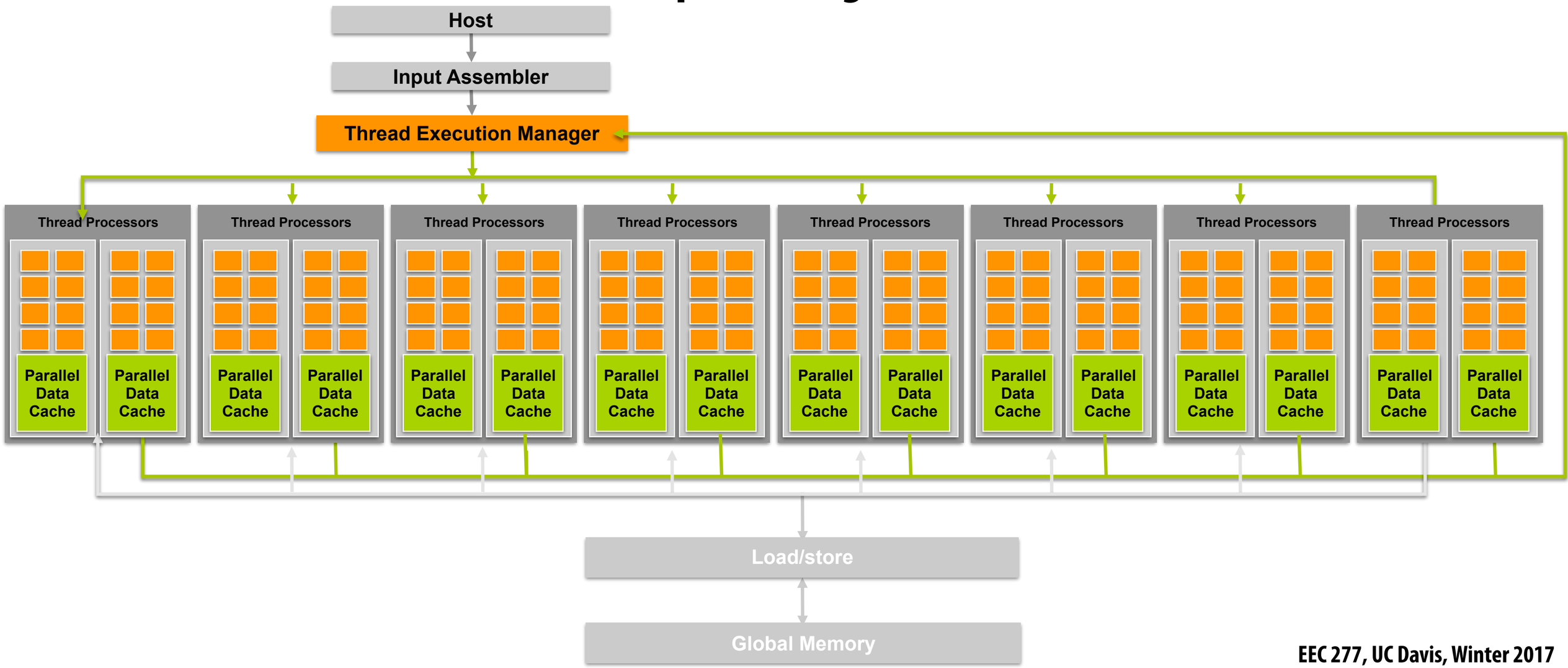
[courtesy Mike Bailey, Oregon State]



[courtesy Chas Boyd, Microsoft]

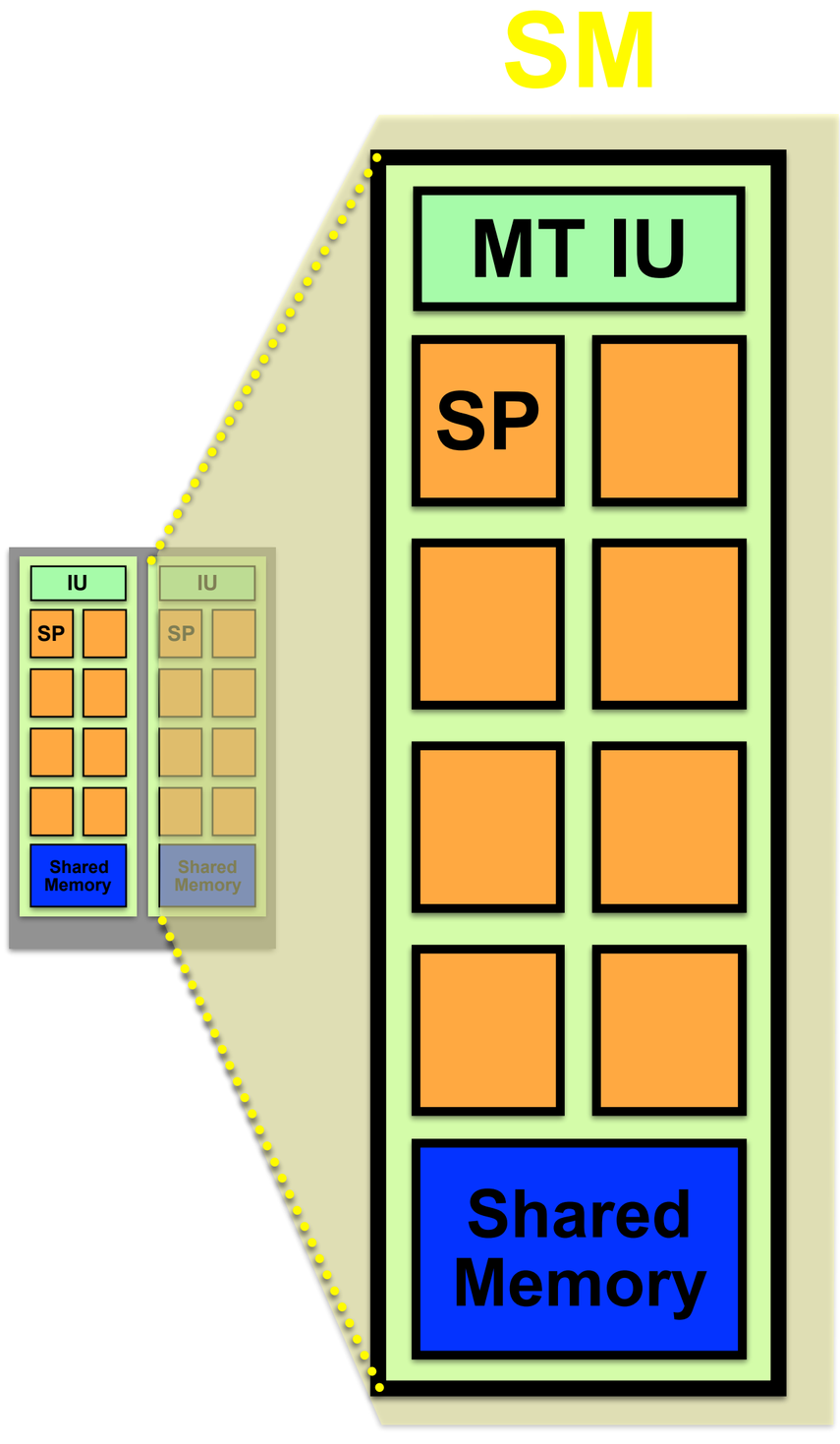
GPU Computing (G80 GPUs)

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors
- Parallel Data Cache accelerates processing



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 (at least this particular SM)
  - 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*



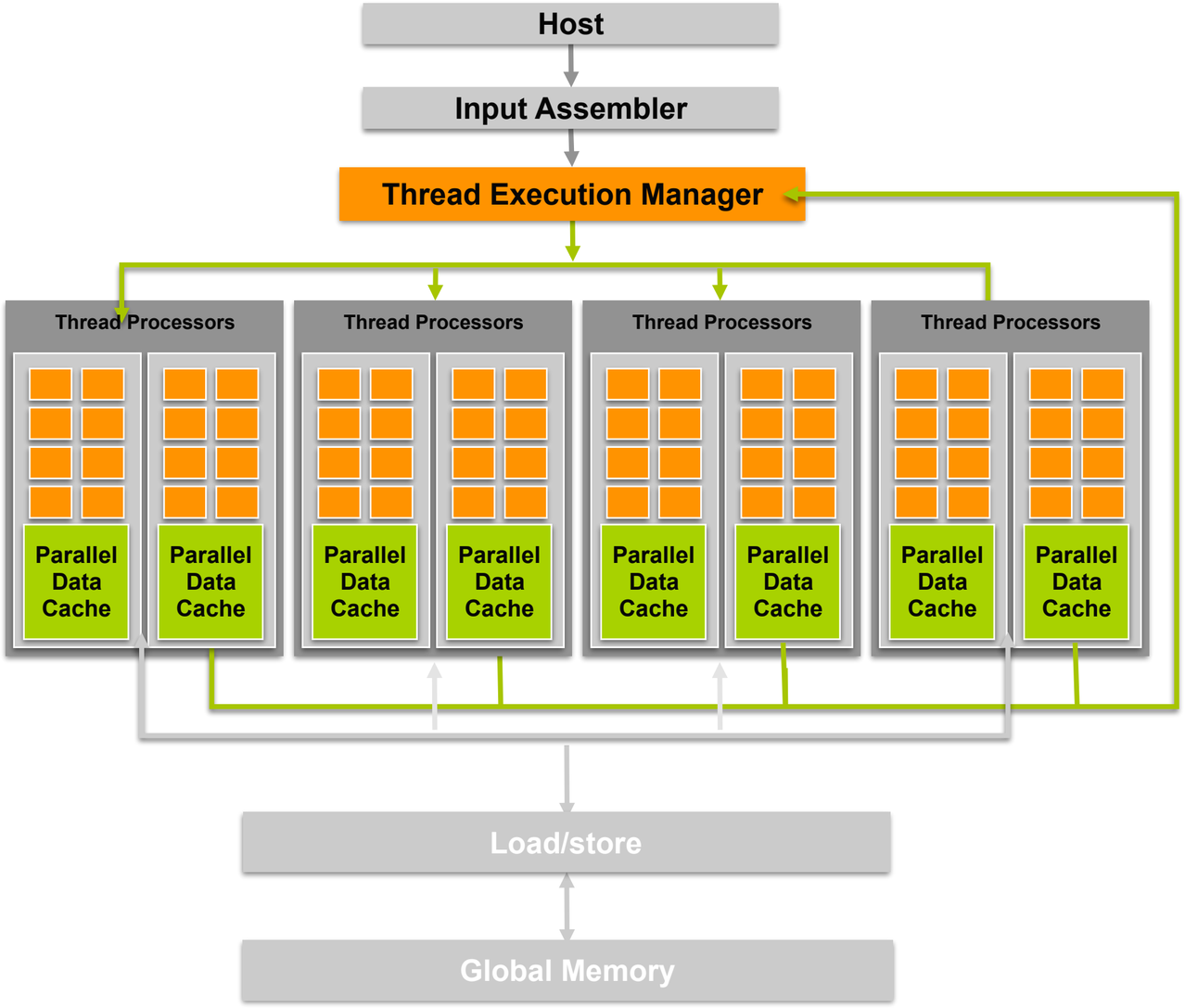
What do you think goes in a single thread processor? How would you design it?

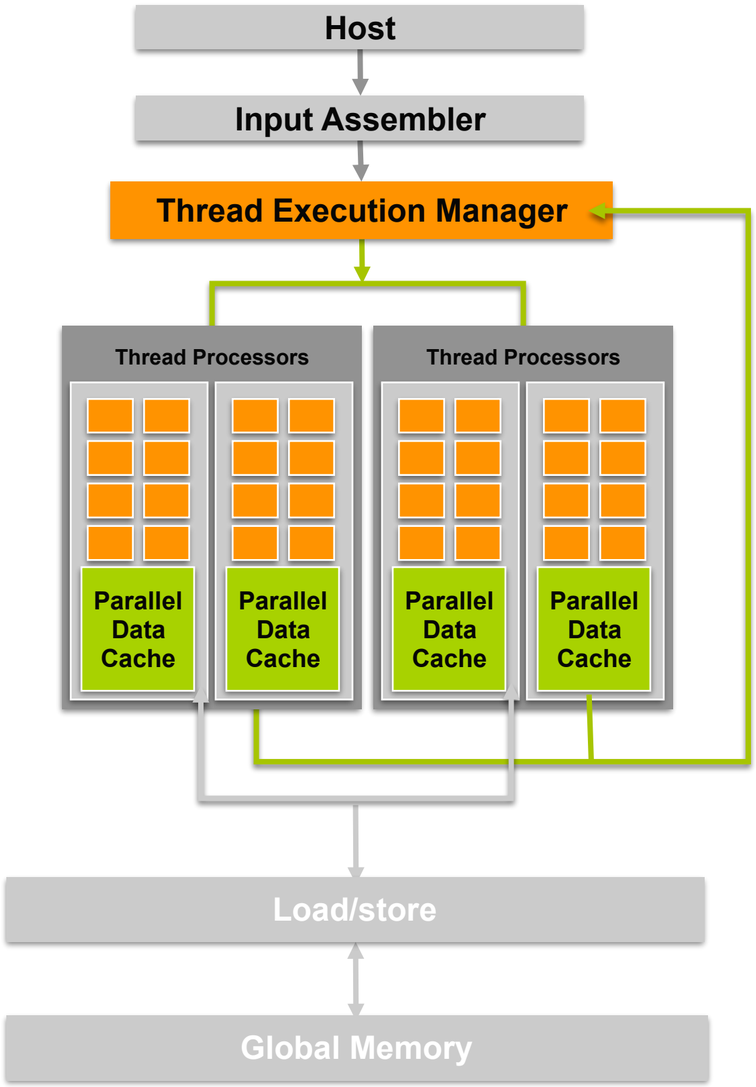
Big Idea #3

- Latency hiding.
  - It takes a long time to go to memory.
  - So while one set of threads is waiting for memory ...
  - ... run another set of threads during the wait.
    - In practice, 32 threads run in a "warp" and an efficient program might have 128–256 threads in a block.

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?

HW Goal: Scalability

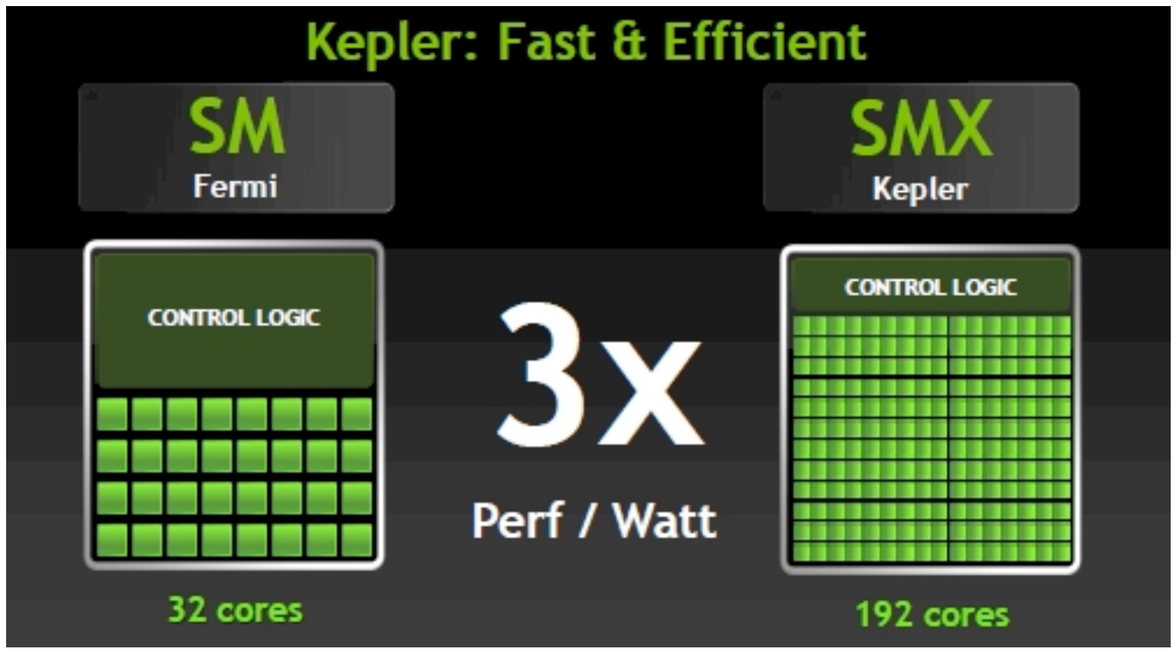
- Scalable execution
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

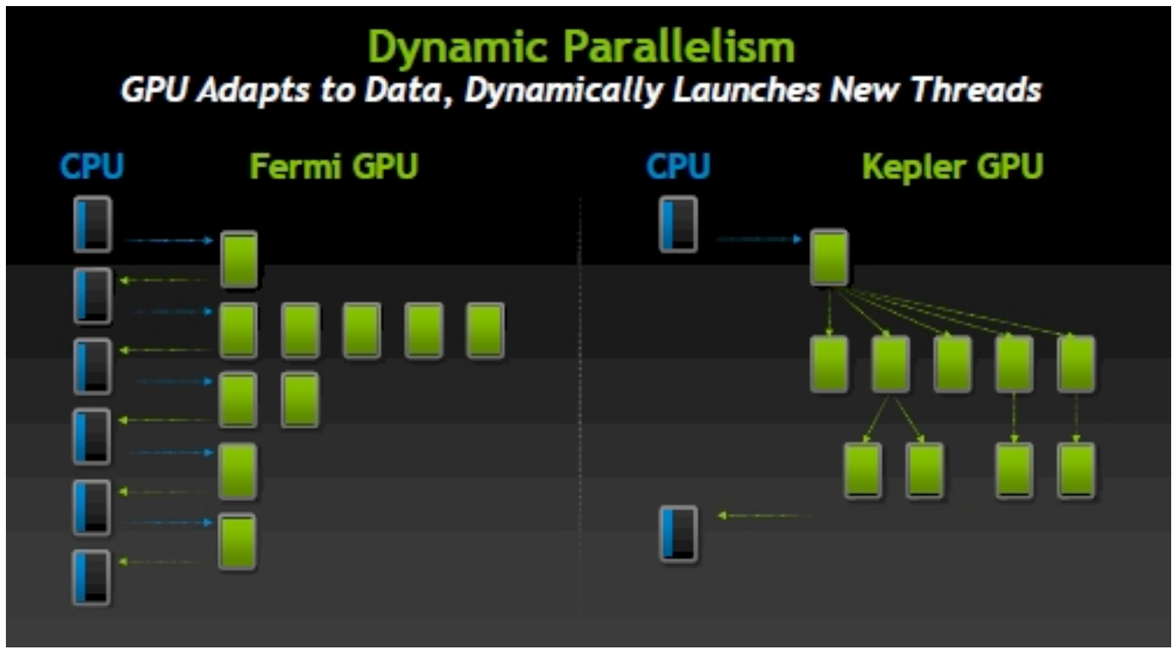
- Hierarchical execution model
  - Decompose problem into sequential steps (kernels)
  - Decompose kernel into computing parallel blocks
  - Decompose block into computing parallel threads

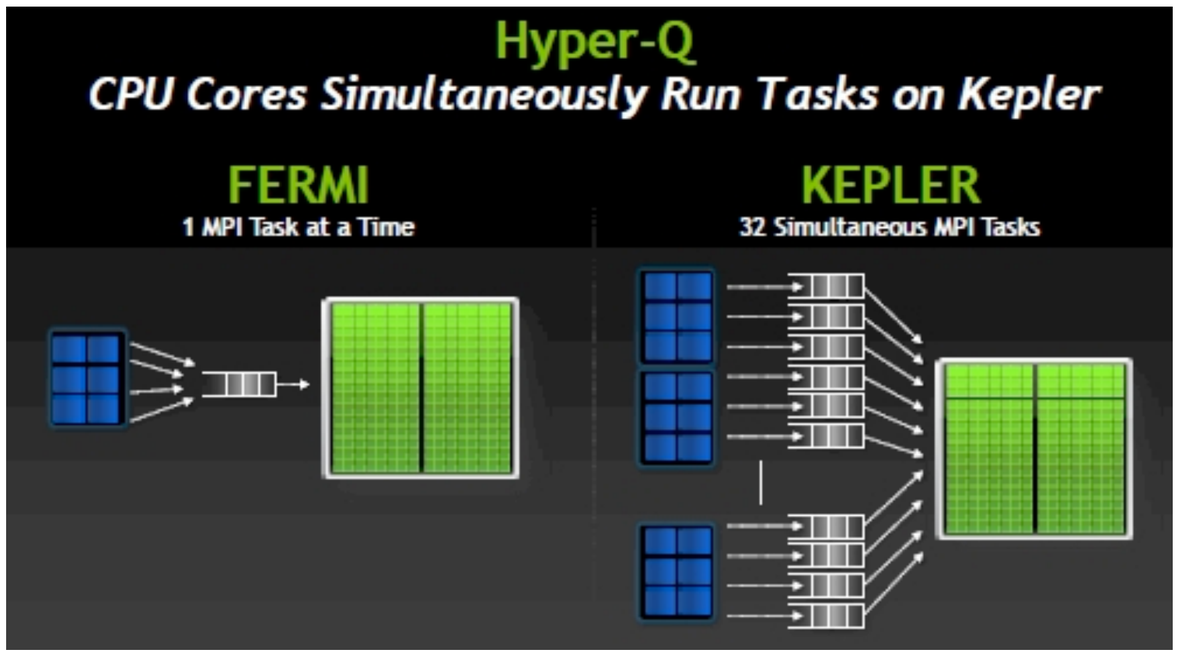
This is very important!

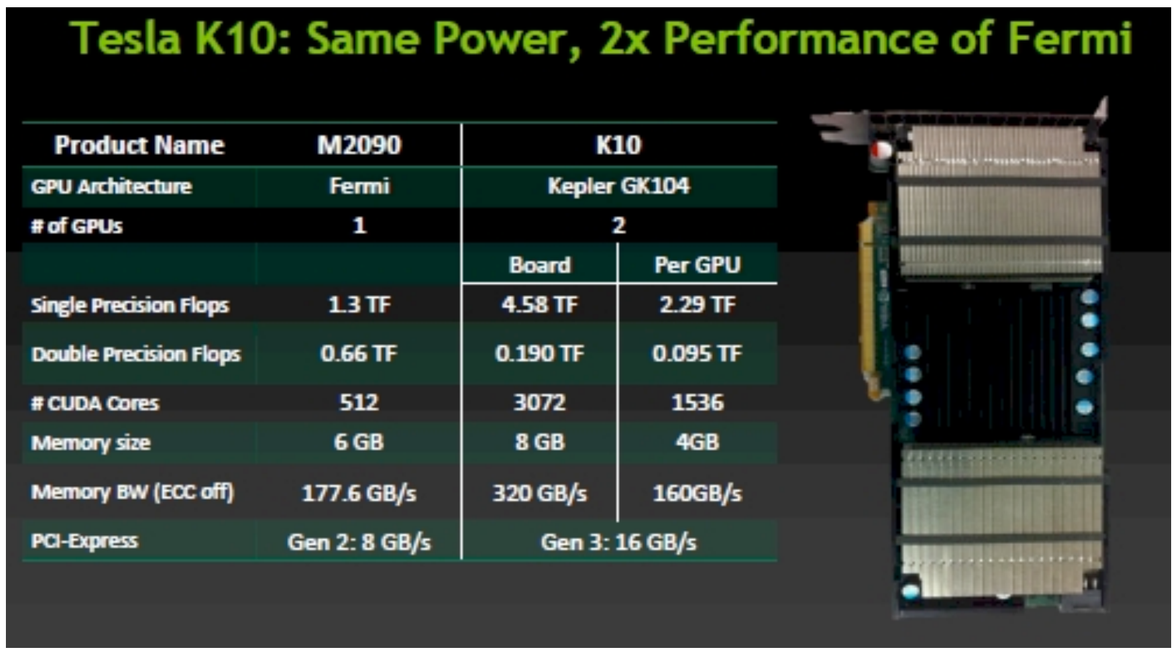
- Hardware distributes independent blocks to SMs as available

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

Pascal SM

- 64 FP32 ALUs/SM
- 32 FP64 ALUs/SM
- 64 kB shared mem/SM
- 256 KB registers/SM

- Each SM has 2 "processing blocks":
  - 32 FP32 ALUs
  - 1 instruction buffer
  - 1 warp scheduler
  - 2 dispatch units

https://devblogs.nvidia.com/parallelforall/inside-pascal/

Programming Model: A Highly Multi-threaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application execute on the device as kernels that run many cooperative threads in parallel
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

CUDA Software Development Kit

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

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

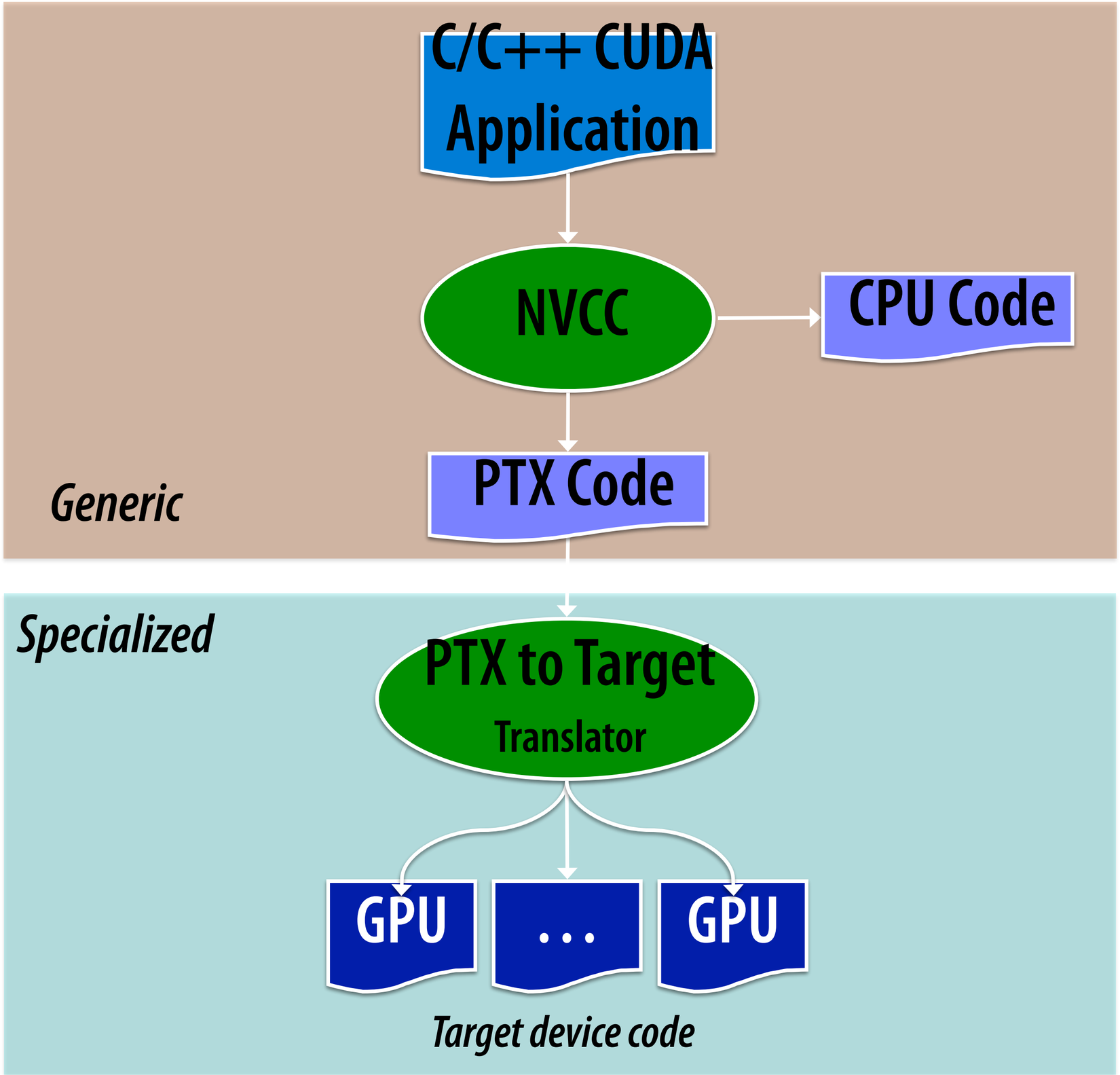
Debugger
Profiler

Standard C Compiler

GPU

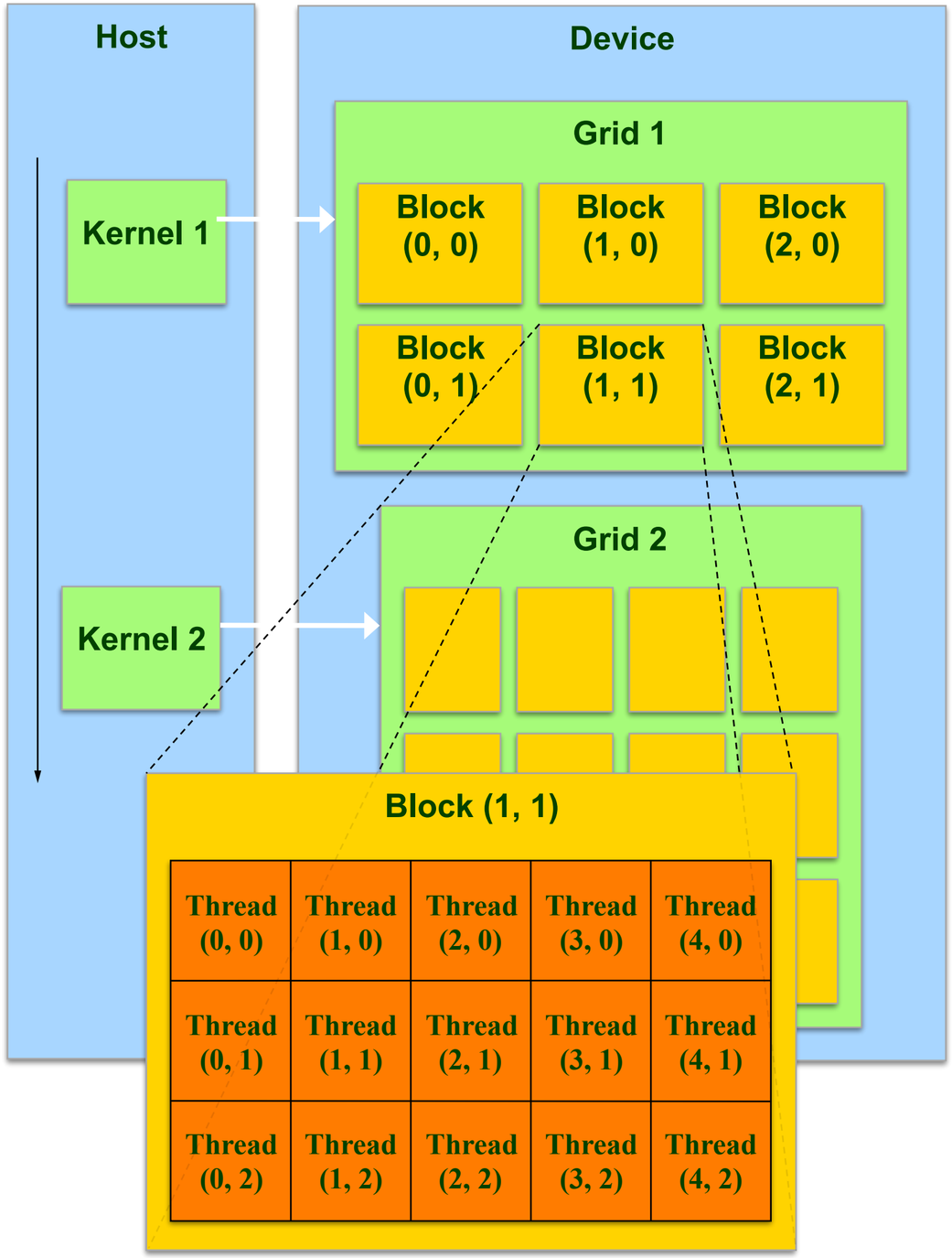
CPU

Compiling CUDA for GPUs



Programming Model (SPMD + SIMD): Thread Batching

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Two threads from two different blocks cannot cooperate
  - Blocks are independent

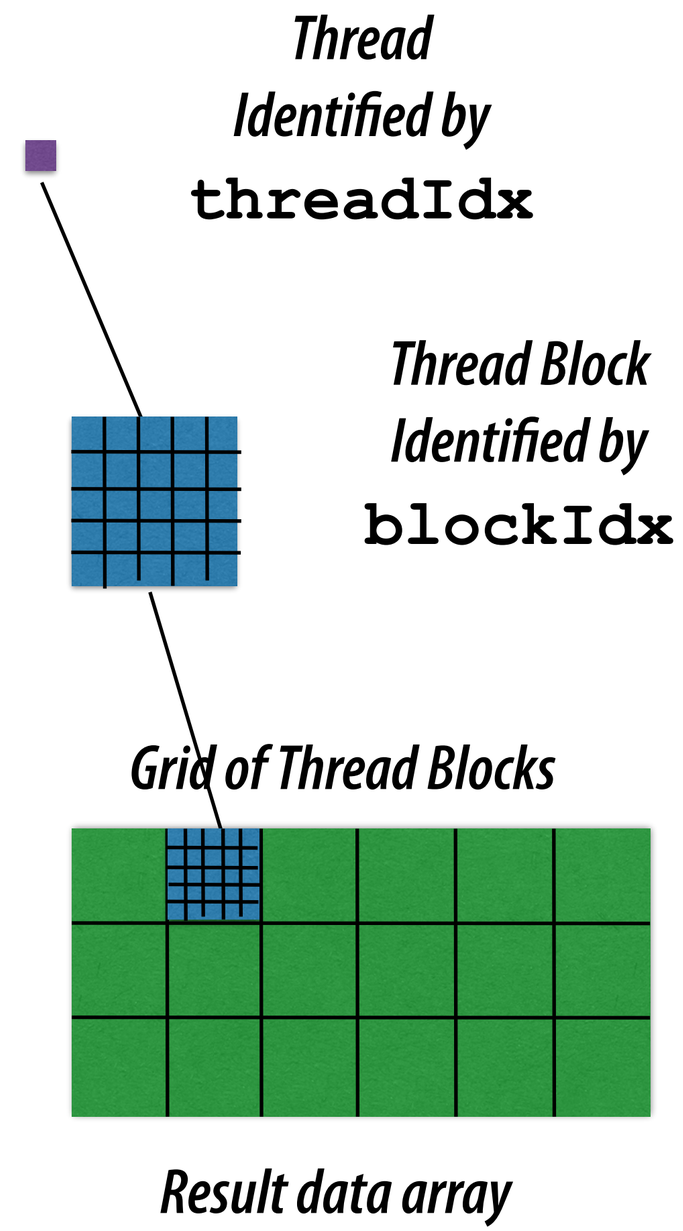


Execution Model

- 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 (of G8X/G9X GPUs):
    - At most 8 concurrent blocks per SM
    - At most 768 concurrent threads per SM (1024 in new hw)
  - Number is further limited by SM resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks

Execution Model

- Multiple levels of parallelism
- Thread block
  - Up to 1024 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)`



Divergence in Parallel Computing

- Removing divergence pain from parallel programming
- SIMD Pain
  - User required to SIMD-ify
  - User suffers when computation goes divergent
- GPUs: Decouple execution width from programming model
  - Threads can diverge freely
  - Inefficiency only when divergence exceeds native machine width
  - Hardware managed
  - Managing divergence becomes performance optimization
  - Scalable

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

Key Parallel Abstractions in CUDA

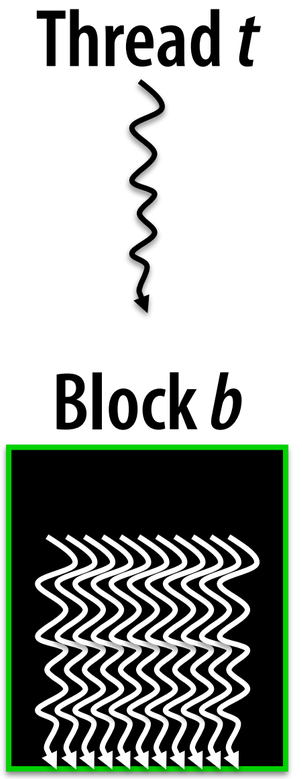
- Hierarchy of concurrent threads

- Lightweight synchronization primitives

- Shared memory model for cooperating threads

Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program
  - (This is "SIMT")
- Threads are grouped into thread blocks
  - threads in the same block can cooperate
- Threads/blocks have unique IDs
  - Each thread knows its "address" (thread/block ID)



CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   // kernel callable from host
__device__ void DeviceFunc(...);   // function callable on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);    // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   // barrier synchronization
```

CUDA: Features available on GPU

- Standard mathematical functions
  - `sinf, powf, atanf, ceil, min, sqrtf, etc.`
- Atomic memory operations
  - `atomicAdd, atomicMin, atomicAnd, atomicCAS,` etc.
- Texture accesses in kernels
  - `// declare texture reference`
    `texture<float,2> my_texture;`
  - `float4 texel = texfetch(my_texture, u, v);`

Example: Vector Addition Kernel

- Compute vector sum C = A+B means:
- n = length(C)
-
```
for i = 0 to n-1:
    C[i] = A[i] + B[i]
```
- So C[0] = A[0] + B[0], C[1] = A[1] + B[1], etc.

Example: Vector Addition Kernel

Device Code

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

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

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

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

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

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

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

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

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

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

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

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

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

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Host Code

Synchronization of blocks

- Threads within block may synchronize with barriers
  - … Step 1 …

```
__syncthreads();
```

    … Step 2 …
- Blocks coordinate via atomic memory operations
  - e.g., increment shared queue pointer with atomicInc()
- Implicit barrier between dependent kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```

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
  - CUDA is now in LLVM (compiler) so this is an interesting area of research

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

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: BAD … can easily deadlock
- Independence requirement gives scalability

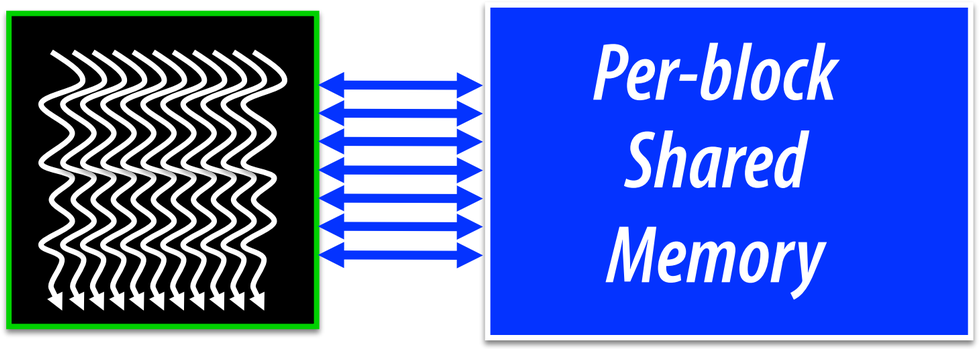
Big Idea #4

- 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

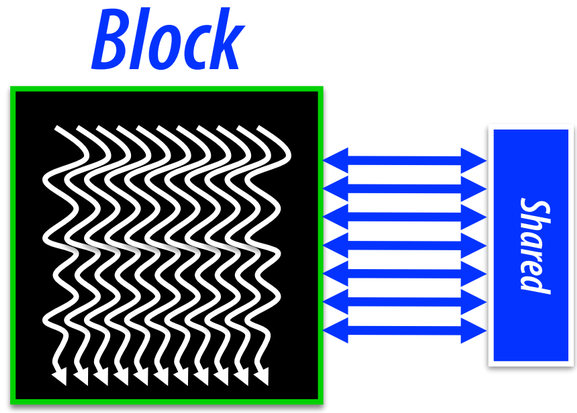
Levels of parallelism

- Instruction level parallelism
  - Within a thread: can you issue multiple instructions? (Depends on hardware capabilities)
- 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

Memory model



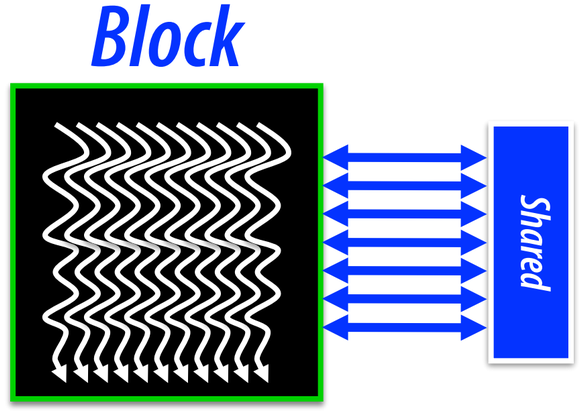
Using per-block shared memory



Using per-block shared memory

- Variables shared across block

```
__shared__ int *begin, *end;
```



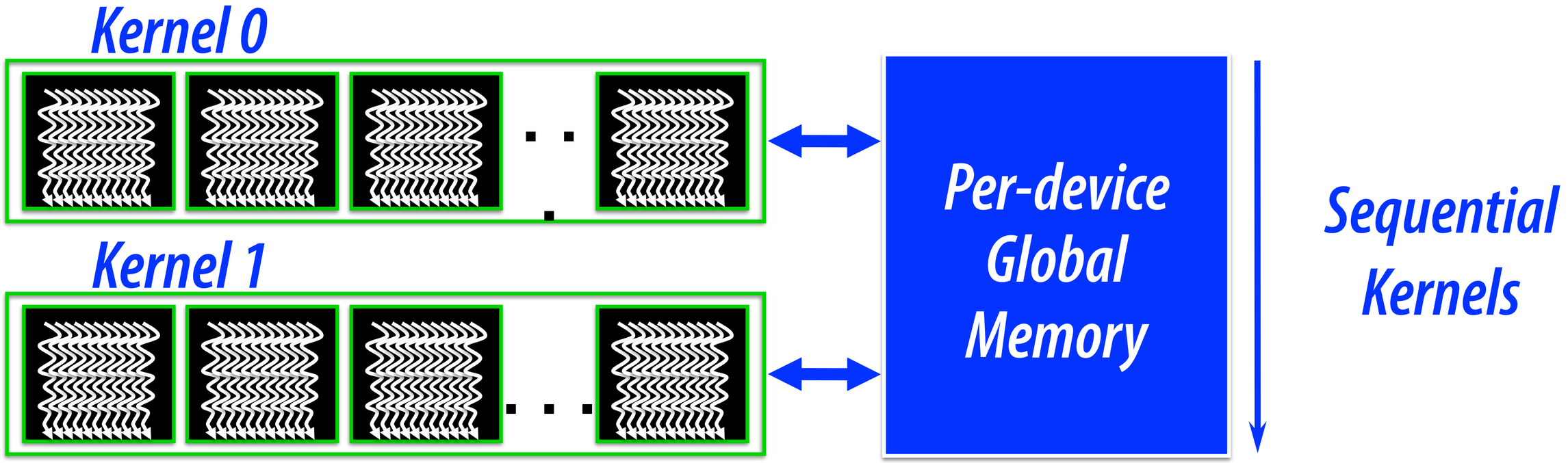
- Scratchpad memory

```
__shared__ int scratch[blocksize];

scratch[threadIdx.x] = begin[threadIdx.x];

// … compute on scratch values …

begin[threadIdx.x] = scratch[threadIdx.x];
```

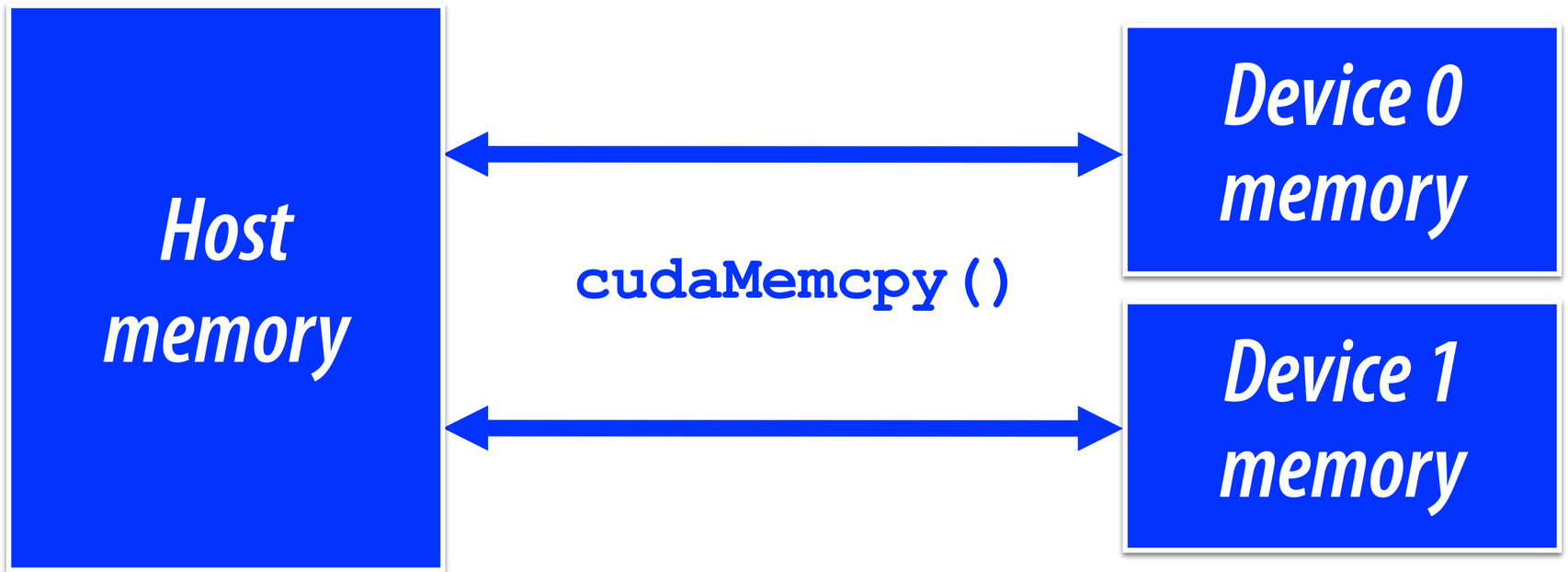
- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];

__syncthreads();

int left = scratch[threadIdx.x - 1];
```

Memory model



Memory model



CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc(), cudaFree()`
- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy(), cudaMemcpy2D(), ...`
- Implicit memory copy for host ↔ device, device ↔ device
  - `cudaMallocManaged()`
- Texture management
  - `cudaBindTexture(), cudaBindTextureToArray(), ...`
- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject(), cudaD3D9MapVertexBuffer(), ...`

Example: Vector Addition Kernel

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

int main(){
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Host code for vecAdd

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

This manual management and copy can now be done automatically with CUDA managed memory

Example: Host code for vecAdd

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
```

This manual management and copy can now be done automatically with CUDA managed memory

```
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Example: Host code for vecAdd

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

This manual management and copy can now be done automatically with CUDA managed memory

Example: Host code for vecAdd

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

This manual management and copy can now be done automatically with CUDA managed memory

Basic Efficiency Rules

- Develop algorithms with a data parallel mindset
- Minimize divergence of execution within blocks
- Maximize locality of global memory accesses
- Exploit per-block shared memory as scratchpad
- Expose enough parallelism

Summing Up

- CUDA = C + a few simple extensions
  - makes it easy to start writing basic parallel programs

- Three key abstractions:
  - hierarchy of parallel threads
  - corresponding levels of synchronization
  - corresponding memory spaces

- Supports massive parallelism of manycore GPUs