The GPU Hardware and Software Model: The GPU is not a PRAM (but it's not far off)

CS 223 Guest Lecture

John Owens
Electrical and Computer Engineering, UC Davis

Credits

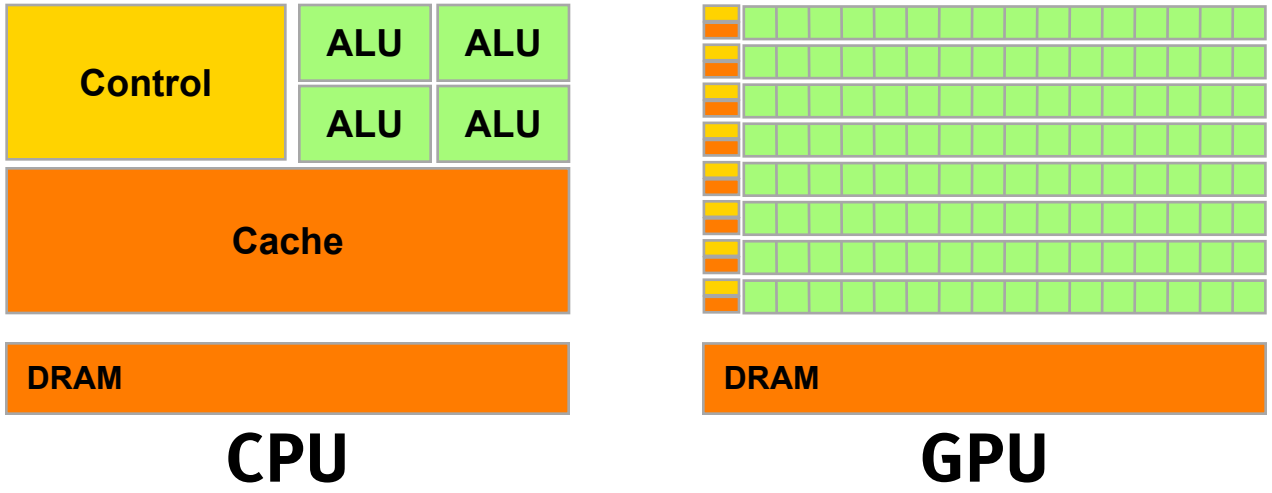
- This lecture is originally derived from a tutorial at ASPLOS 2008 (March 2008) by David Luebke (NVIDIA Research), Michael Garland (NVIDIA Research), John Owens (UC Davis), and Kevin Skadron (NVIDIA Research/University of Virginia), with additional material from Mark Harris (NVIDIA Ltd.).

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

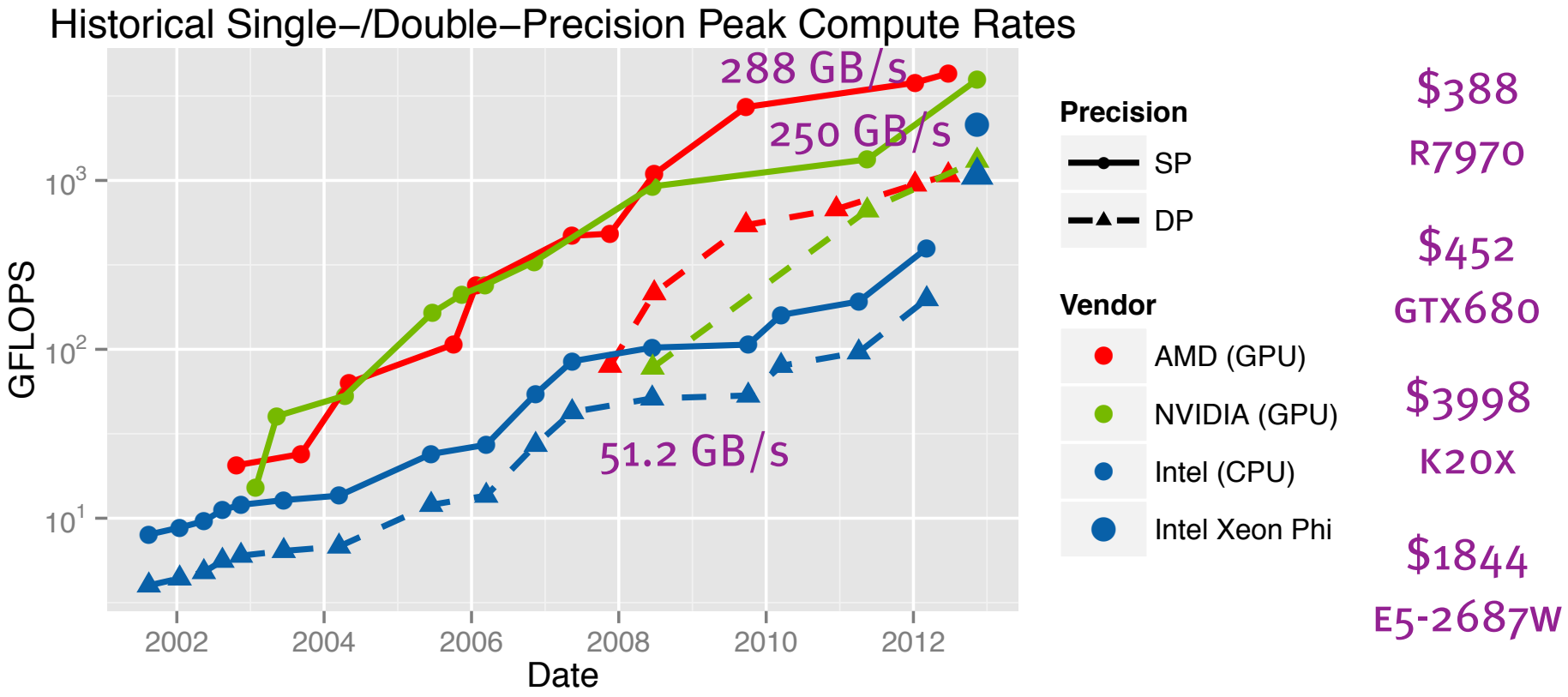
—Seymour Cray

Why is data-parallel computing fast?

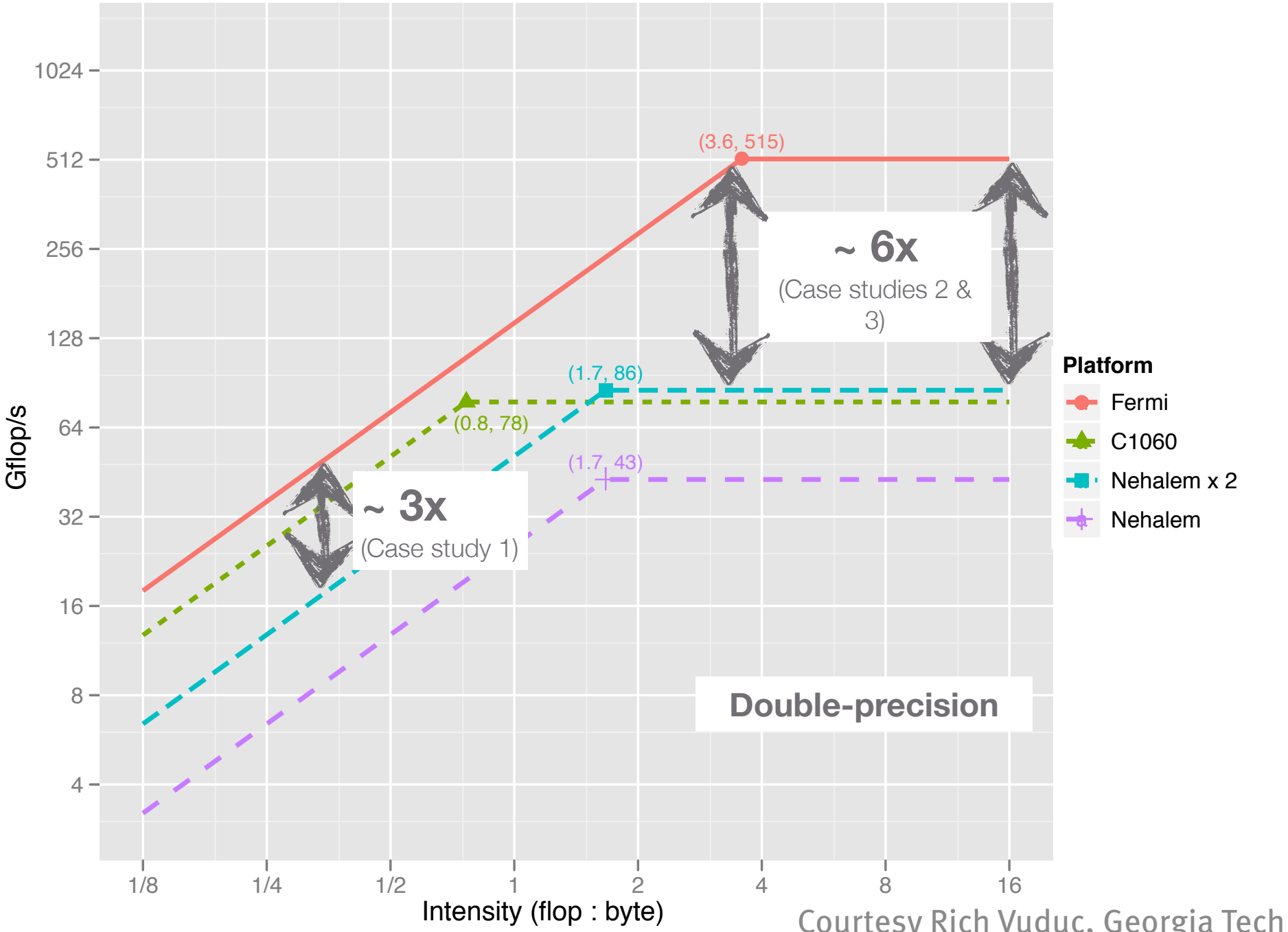
- The GPU is specialized for compute-intensive, highly parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control



Recent GPU Performance Trends



Early data courtesy Ian Buck; from Owens et al. 2007 [CGF]; thanks to Mike Houston (AMD), Ronak Singhal (Intel), Sumit Gupta (NVIDIA)



Courtesy Rich Vuduc, Georgia Tech

What I Know About PRAM

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia

Toolbox

Print/export

Languages
Deutsch
فارسی
Français
日本語
中文

Parallel Random Access Machine

From Wikipedia, the free encyclopedia

A **Parallel Random Access Machine** (**PRAM**) is a shared memory abstract machine which is used by parallel algorithm designers to estimate the algorithm performance (like its time complexity). PRAM neglects such issues as synchronization and communication, but provides any (problem size-dependent) number of processors. Algorithm cost, for instance, is estimated as O(time x processor_number).

The read/write conflicts in accessing the same shared memory location simultaneously are resolved by one of the following strategies:

1. Exclusive Read Exclusive Write (EREW)—every memory cell can be read or written to by only one processor at a time
2. Concurrent Read Exclusive Write (CREW)—multiple processors can read a memory cell but only one can write at a time
3. Exclusive Read Concurrent Write (ERCW)—never considered
4. Concurrent Read Concurrent Write (CRCW)—multiple processors can read and write.

Here, E and C stand for 'exclusive' and 'concurrent' correspondingly. The read causes no discrepancies while the concurrent write is further defined as:

Common—all processors write the same value; otherwise is illegal
Arbitrary—only one arbitrary attempt is successful, others retire
Priority—processor rank indicates who gets to write
Another kind of *array Reduction* operation like SUM, Logical AND or MAX.

Several simplifying assumptions are made while considering the development of algorithms for PRAM. They are :

1. There is no limit on the number of processors in the machine.
2. Any memory location is uniformly accessible from any processor.
3. There is no limit on the amount of shared memory in the system.
4. Resource contention is absent.
5. The programs written on these machines are, in general, of type MIMD. Certain special cases such as SIMD may also be handled in such a framework.

Algorithms are written in pseudo-code as there are only prototype implementations of a PRAM (see below). However, these kinds of algorithms are useful for understanding the exploitation of concurrency, dividing the original problem into similar sub-problems and solving them in parallel.

GPU Truths

- There is no limit on the number of processors in the machine. *True! The programming model abstracts this.*
- Any memory location is uniformly accessible from any processor. *For global memory, true, but there's a memory hierarchy. Also, memory coalescing is important.*
- There is no limit on the amount of shared memory in the system. *Practically, if an implementation goes out of core memory, it's probably not worth pursuing anyway.*
- Resource contention is absent. *Er, not so much.*
- The programs written on these machines are, in general, of type MIMD. Certain special cases such as SIMD may also be handled in such a framework. *GPUs are SIMD across warps, otherwise MIMD.*

GPU Memory Model

- The read/write conflicts in accessing the same shared memory location simultaneously are resolved by one of the following strategies:
  - Exclusive Read Exclusive Write (EREW)—every memory cell can be read or written to by only one processor at a time
  - Concurrent Read Exclusive Write (CREW)—multiple processors can read a memory cell but only one can write at a time *(you'd better program this way)*
  - Exclusive Read Concurrent Write (ERCW)—never considered
  - Concurrent Read Concurrent Write (CRCW)—multiple processors can read and write *(I guess you could program this way—we do have a hash table code where multiple processors write to the same location at the same time and all we care about is that one of them win; also "is my sequence sorted" can be written like this)*

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.
- 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
  - *This is not so much like the PRAM’s MIMD model. Since the GPU has SIMD at its core, the question becomes what the hw/sw do to ease the difficulty of SIMD programming.*

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

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

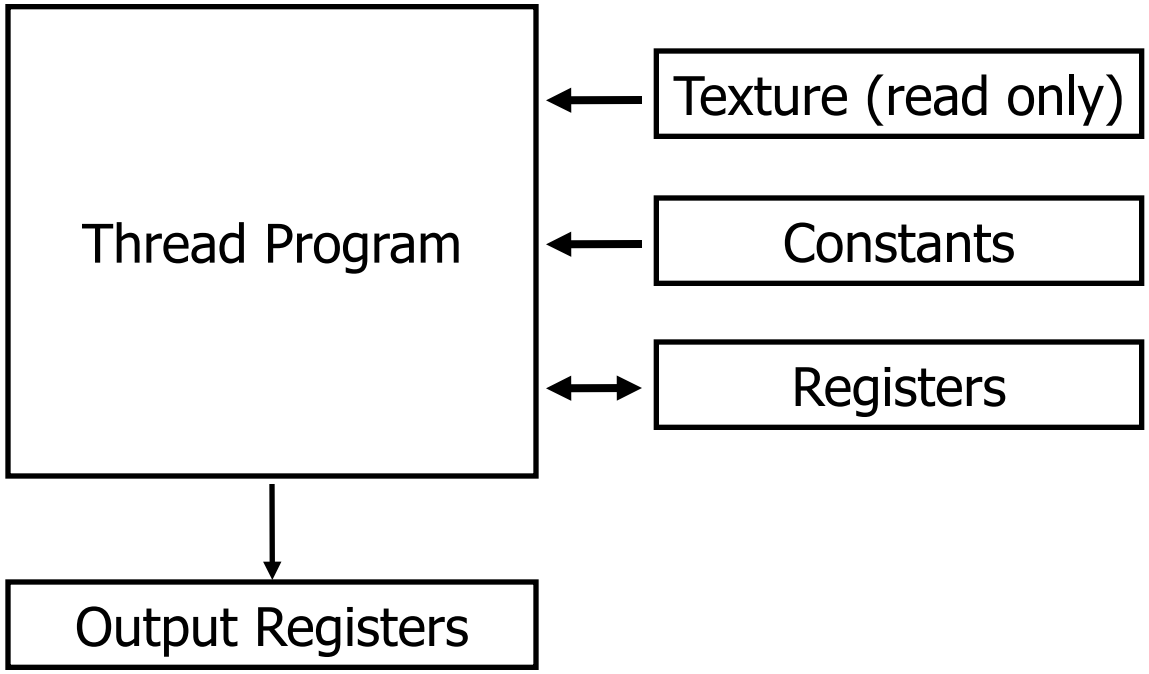
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA *must* use 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

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

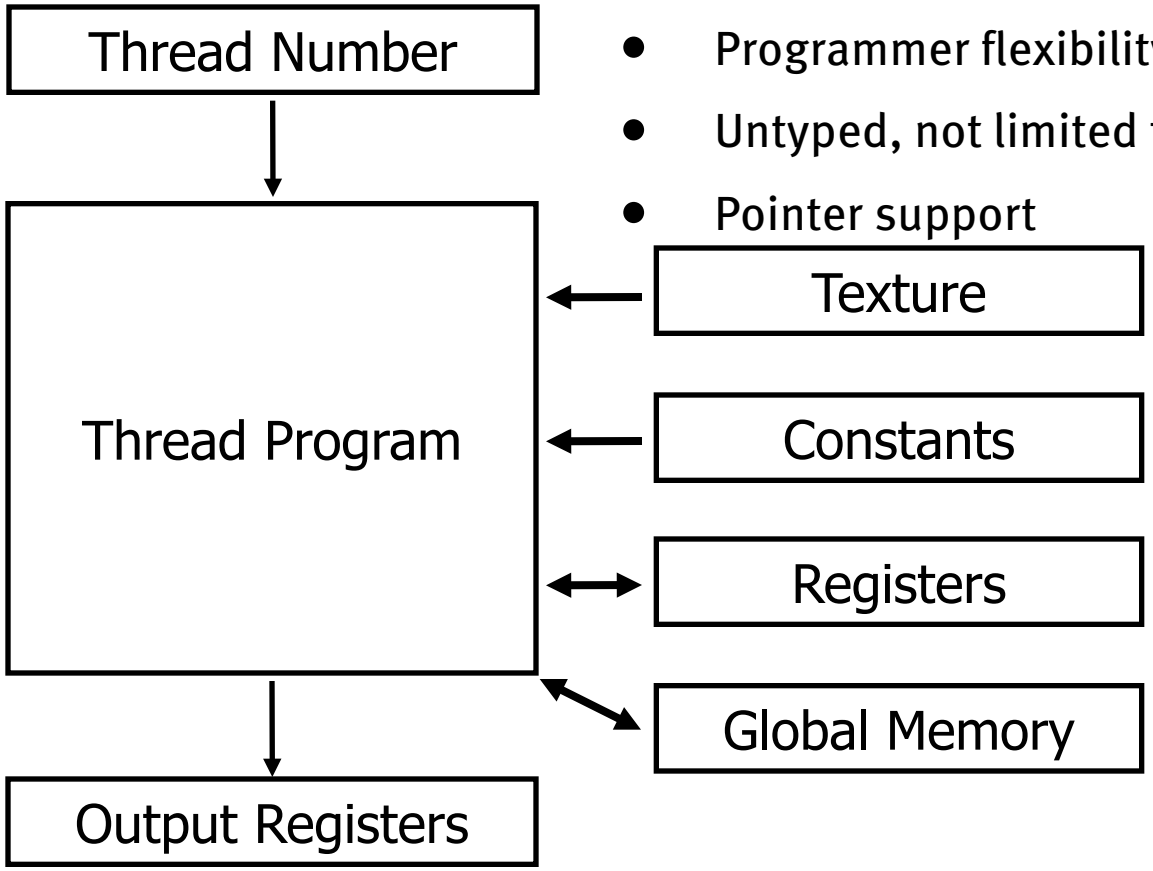
- Millions of instructions
- Full integer and bit instructions
- No limits on branching, looping



General-Purpose Programs

Features

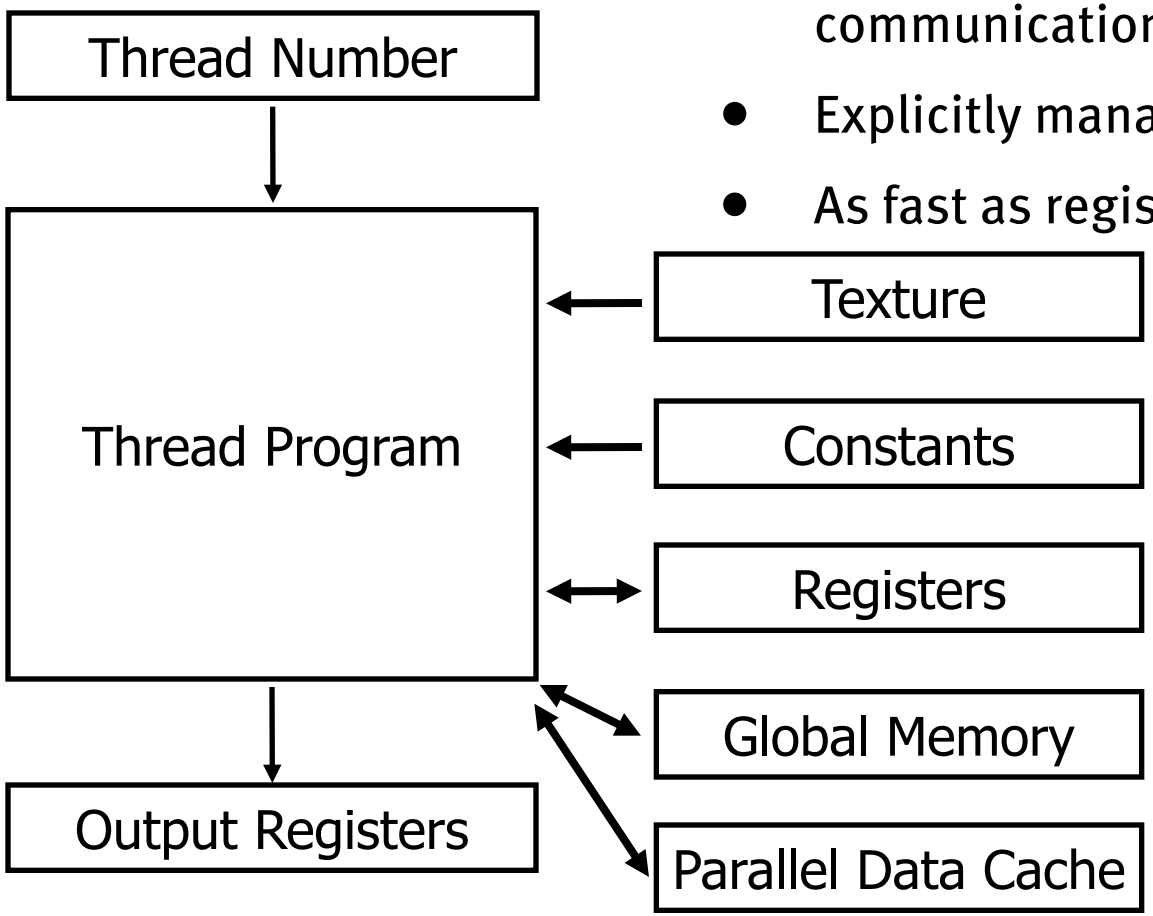
- 1D, 2D, or 3D thread ID allocation
- Fully general load/store to GPU memory: Scatter/Gather
- Programmer flexibility on how memory is accessed
- Untyped, not limited to fixed texture types
- Pointer support



Parallel Data Cache

Features

- Dedicated on-chip memory
- Shared between threads for inter-thread communication
- Explicitly managed
- As fast as registers



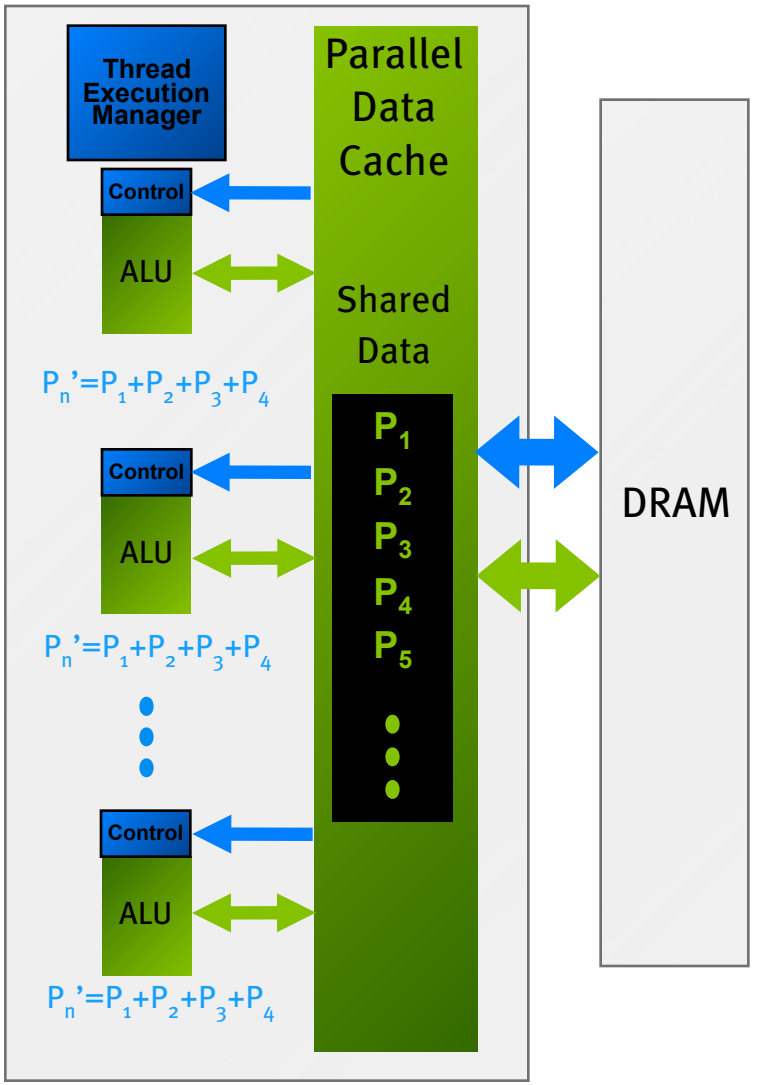
Now, if every thread runs its own copy of a program, and has its own registers, how do two threads communicate?

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

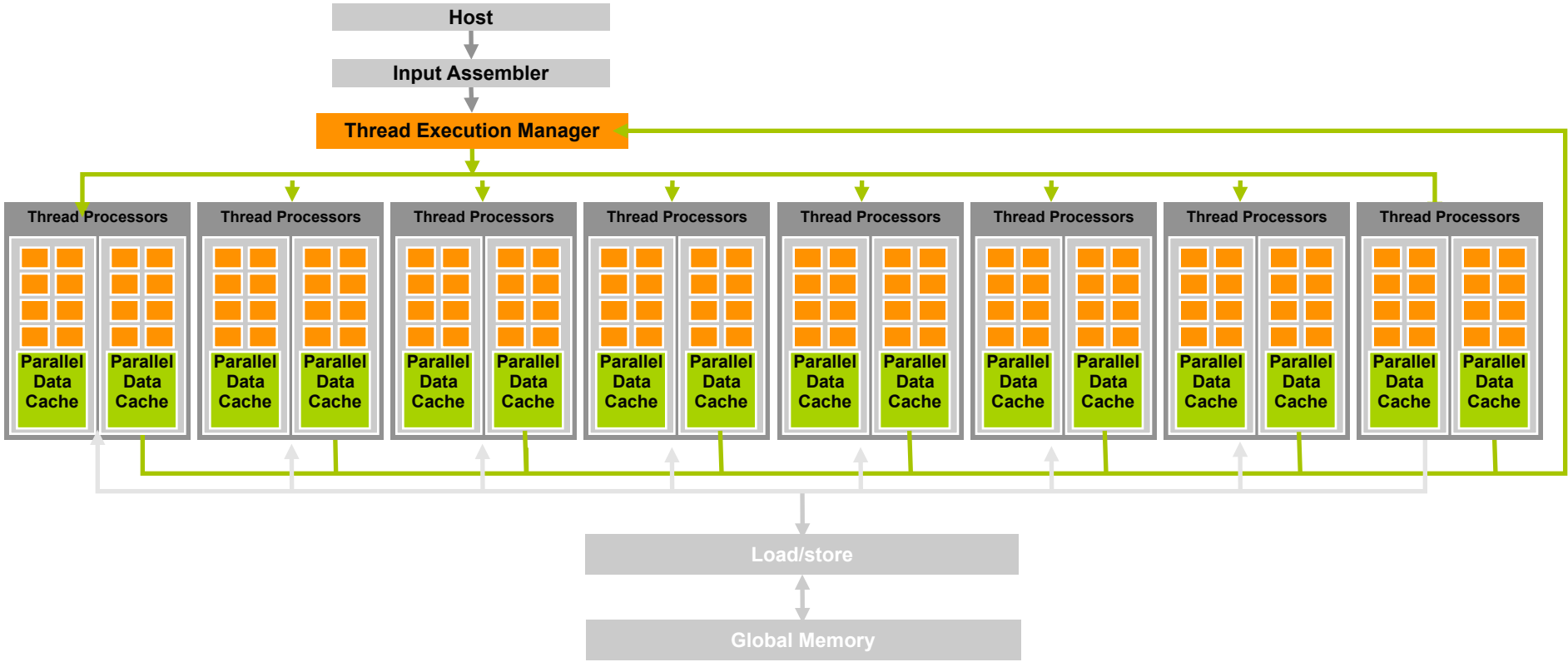
This parallel data cache seems pretty cool. Since we can have thousands of threads active at any time, why can’t all of them communicate through it?

Realities of modern DRAM

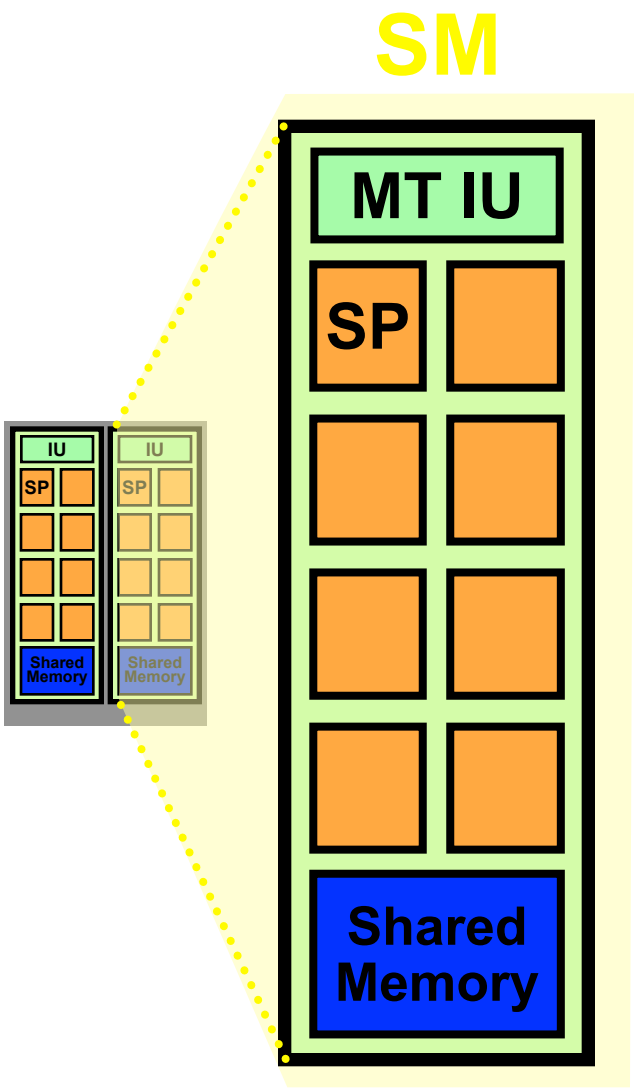
- DRAM {throughput, latency} not increasing as fast as processor capability
- Vendor response: DRAM “grains” are getting larger
- Consequence: Accessing 1 byte of DRAM same cost as accessing 128 consecutive bytes
- Consequence: Structure your memory accesses to be contiguous (NVIDIA: “coalesced”)

GPU Computing (G80 GPUs)

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

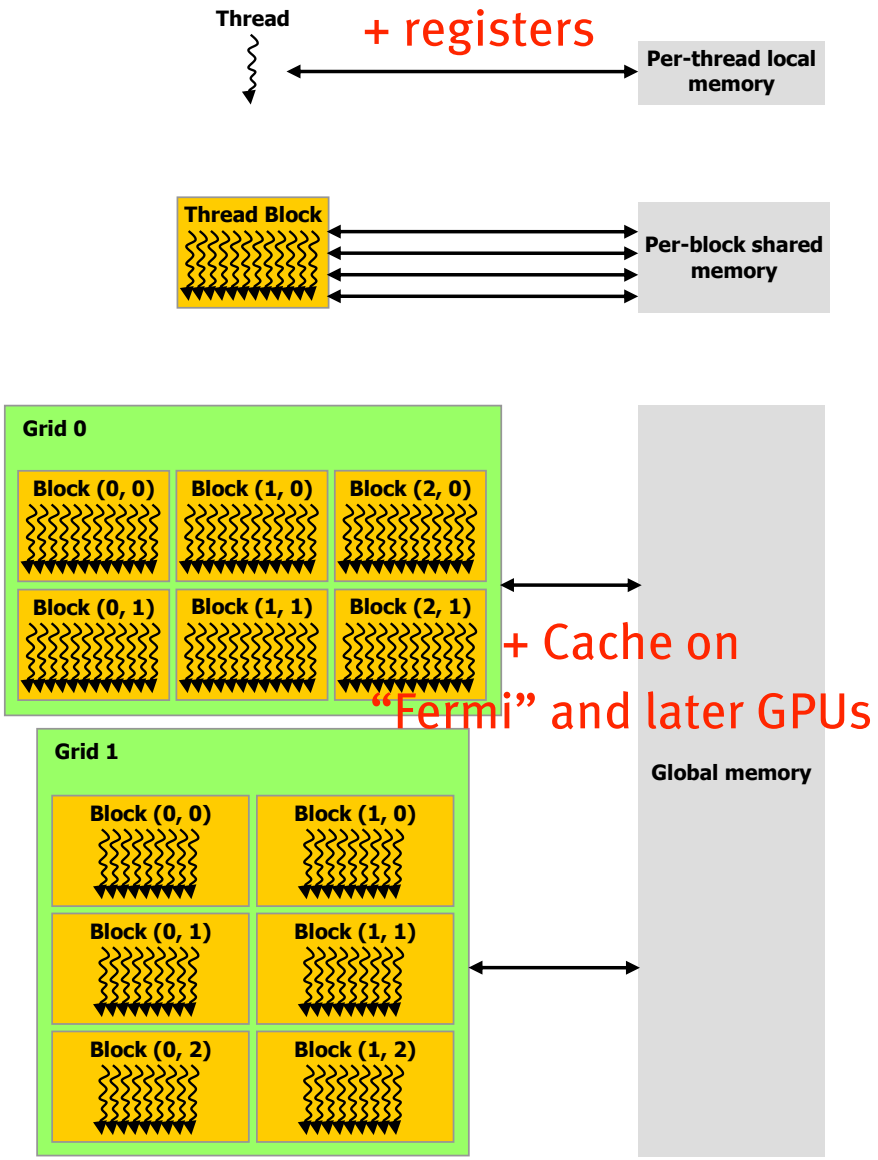


SM Multithreaded Multiprocessor



- **Each SM runs a *block* of threads**
- SM has 8 SP Thread Processors
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
- Scalar ISA
- Up to 768 threads, hardware multithreaded
- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store

Memory hierarchy

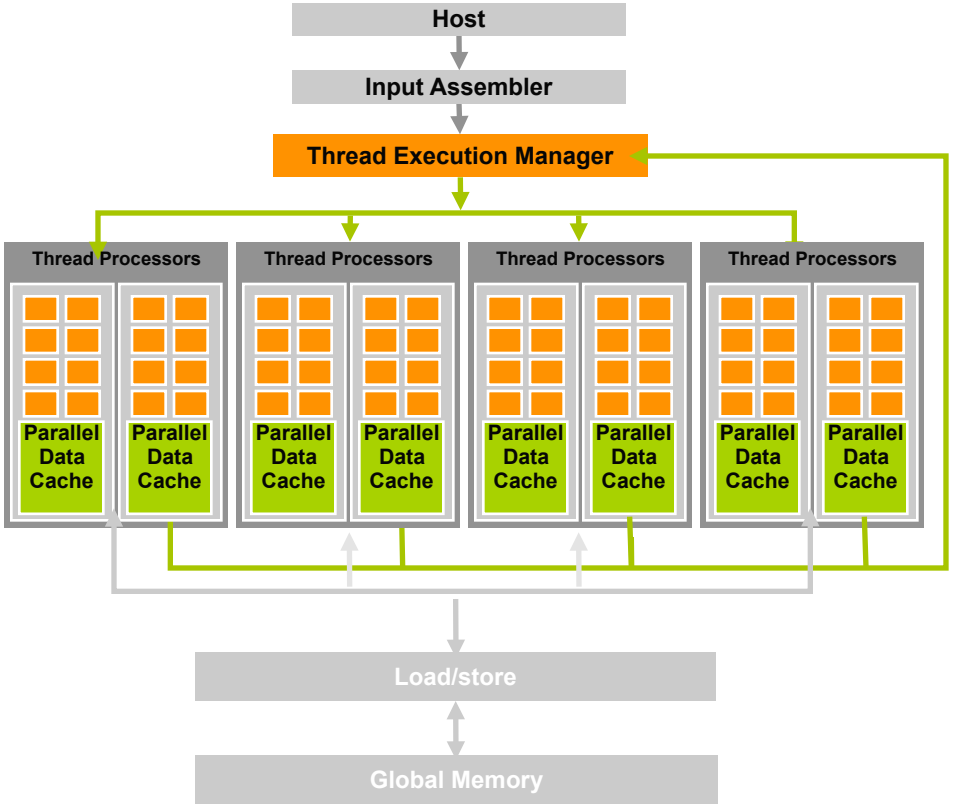


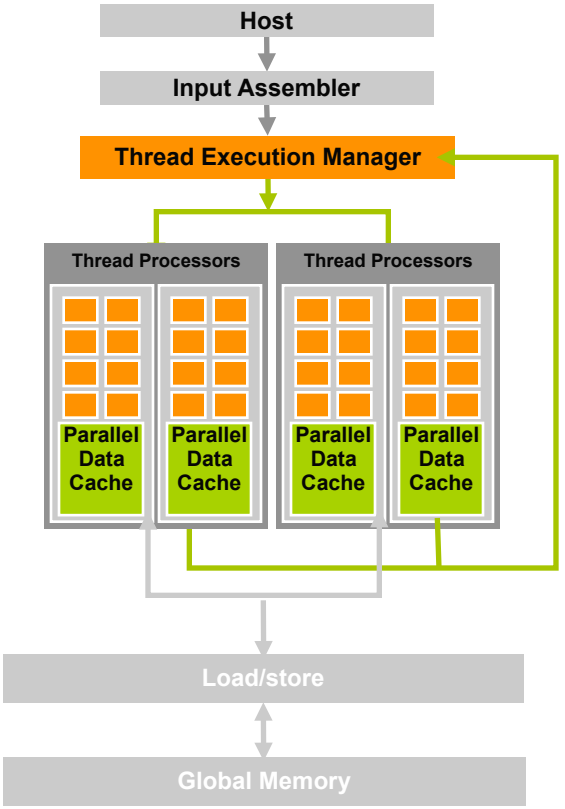
Big Idea #3

- 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 SIMD “warp” and an efficient program usually has 128–256 threads in a block.

Scaling the Architecture

- Same program
- Scalable performance





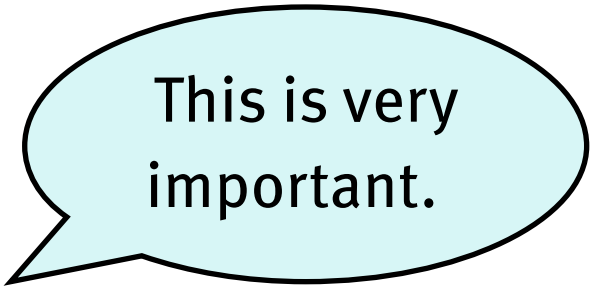
There’s lots of dimensions to scale this processor with more resources. What are some of those dimensions? If you had twice as many transistors, what could you do with them?

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

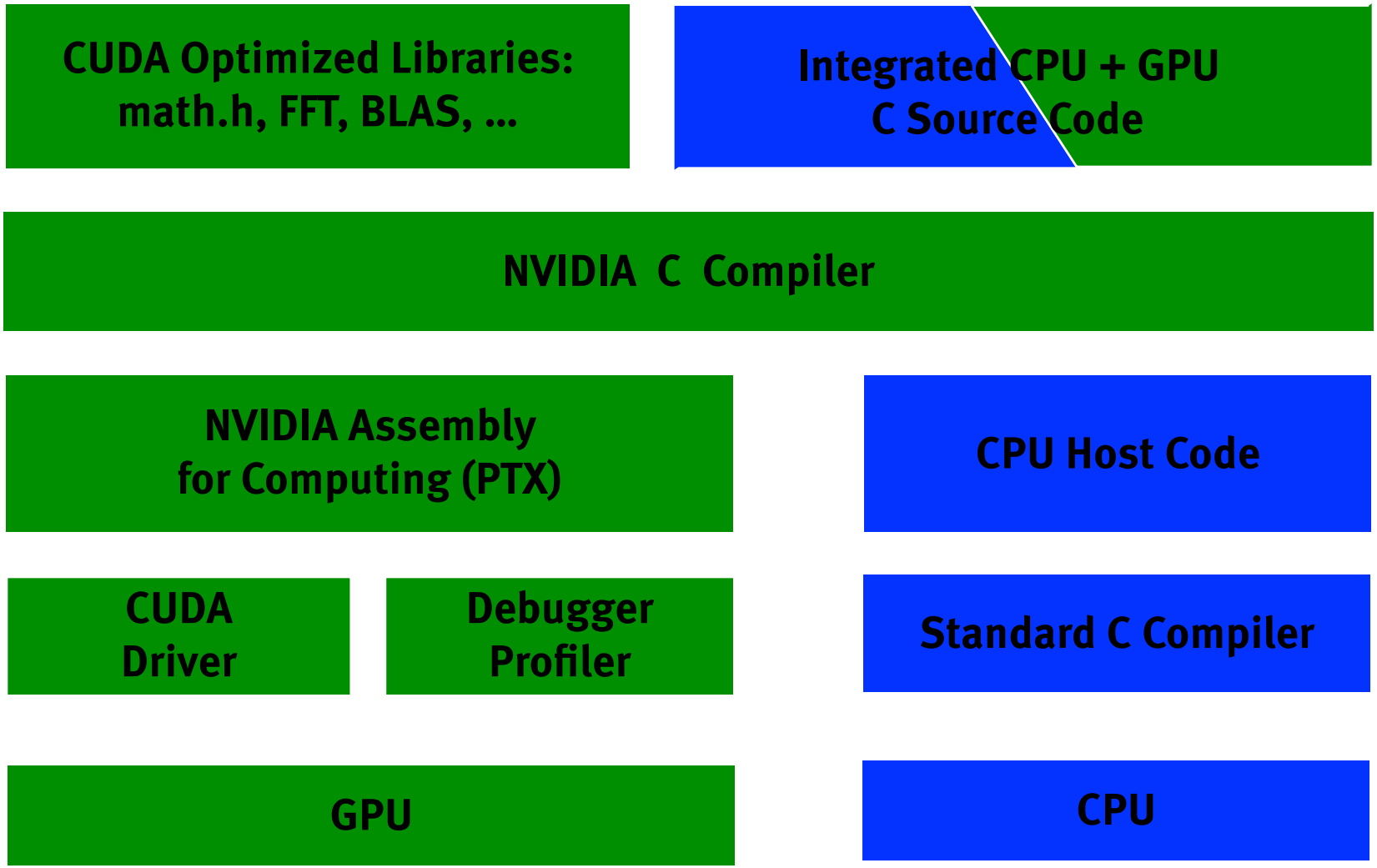


- Hardware distributes ***independent*** blocks to SMs as available

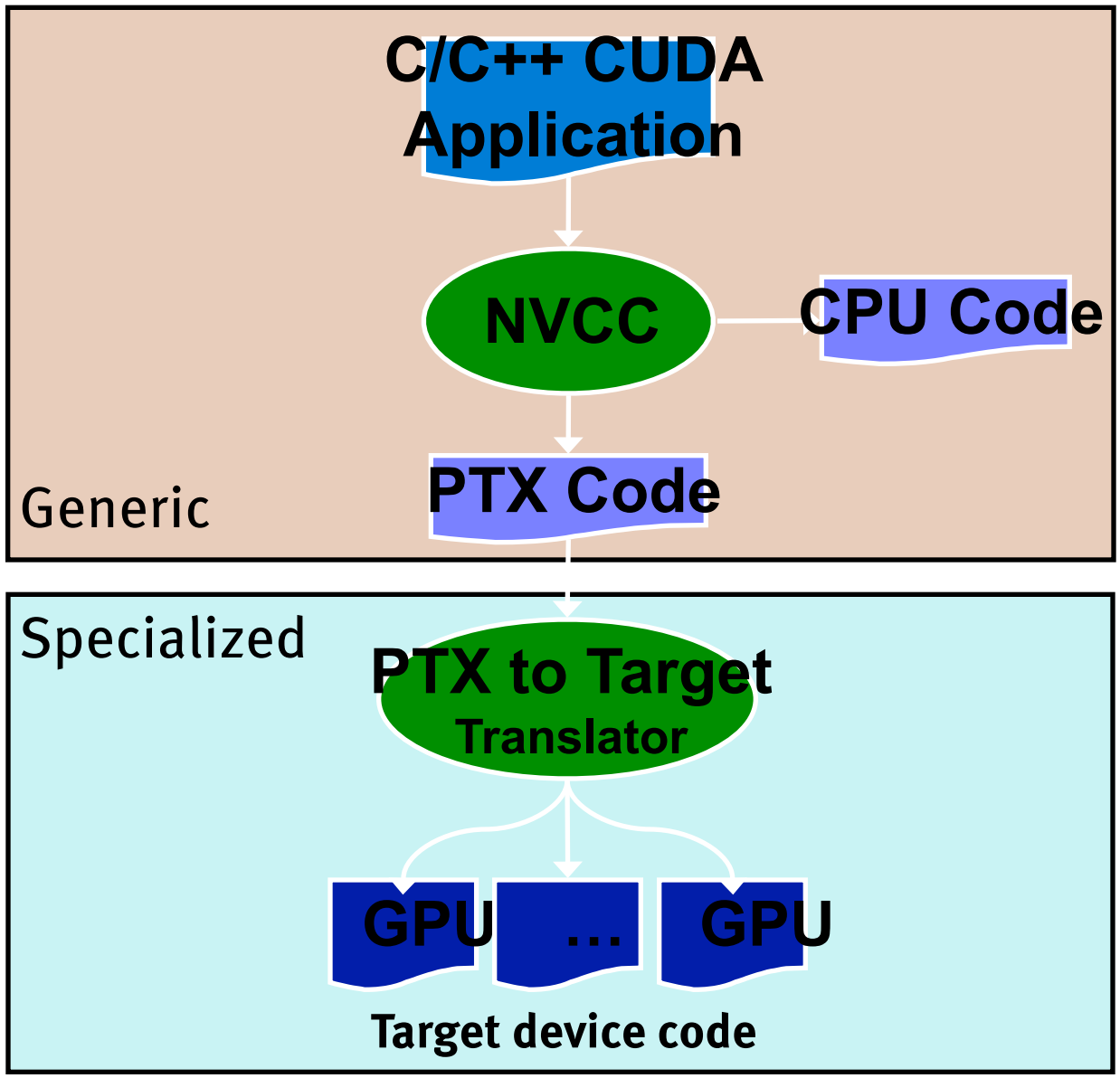
Programming Model: A Highly Multi-threaded Coprocessor

- 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

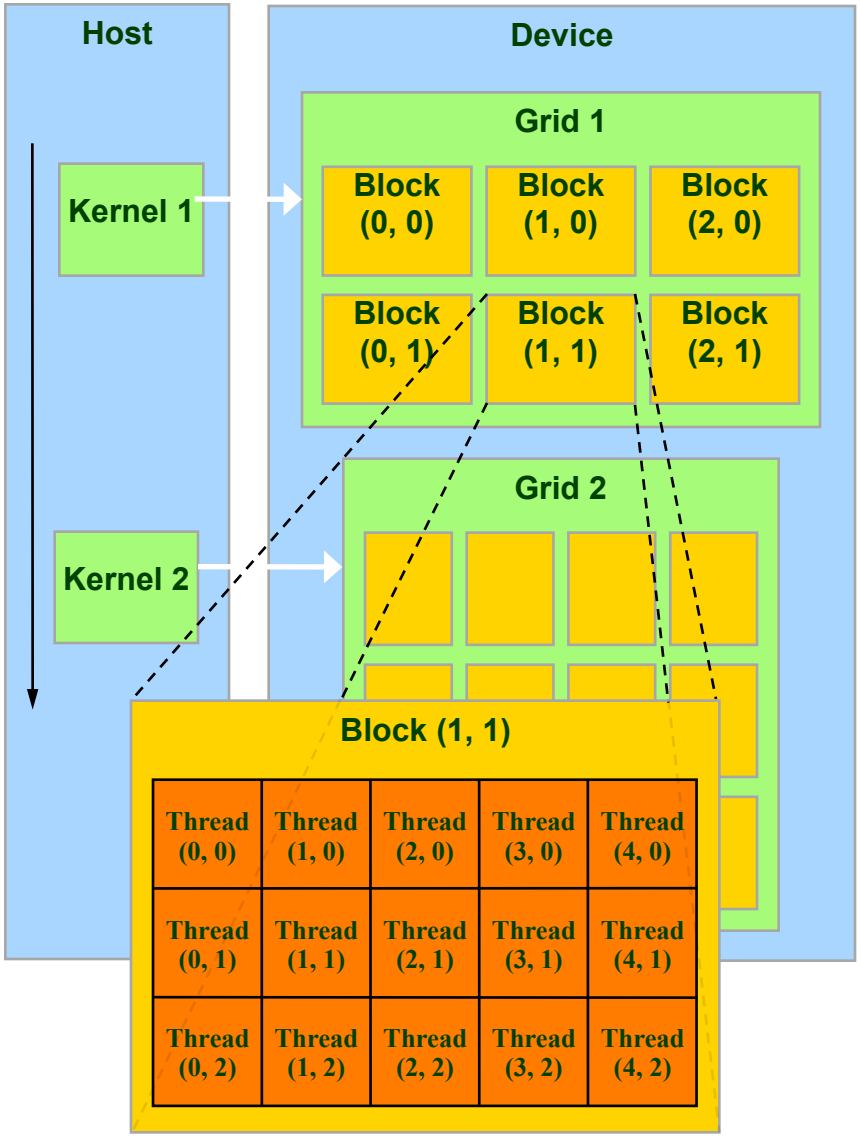


Compiling CUDA for GPUs



Programming Model (SPMD + SIMD): Thread Batching

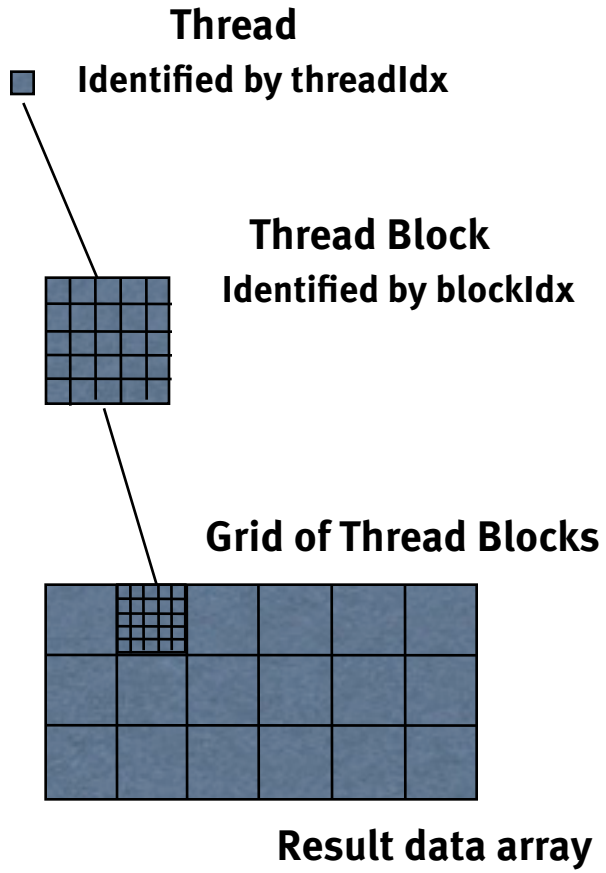
- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Two threads from two different blocks cannot 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, does not release resources until exit
- Several blocks can reside concurrently on one multiprocessor (SM)
  - Control limitations (of compute capability 3.0+ GPUs):
    - At most 16 concurrent blocks per SM
    - At most 2048 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

Execution Model



Multiple levels of parallelism

- Thread block
  - Up to 1024 threads per block (CC 3.0+)
  - Communicate through shared memory
  - Threads guaranteed to be resident
  - **threadIdx, blockIdx**
  - **__syncthreads()**
- Grid of thread blocks
  - **f<<<nblocks, nthreads>>>(a,b,c)**
- Global & shared memory atomics

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

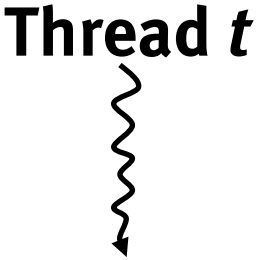
- Scale to 100s of cores, 1000s of parallel threads
- Let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language
- Enable heterogeneous systems (i.e., CPU+GPU)
  - CPU & GPU are separate devices with separate DRAMs

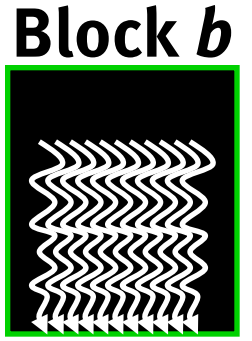
Key Parallel Abstractions in CUDA

- 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: Programming GPU in C

- Philosophy: provide minimal set of extensions necessary to expose power
- 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;         // shared within thread block
```

- Extend function invocation syntax for parallel kernel launch

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

- Special variables for thread identification in kernels

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

- Intrinsics that expose specific operations in kernel code

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

CUDA: Features available on GPU

- Standard mathematical functions

  sinf, **powf**, **atanf**, **ceil**, **min**, **sqrtf**, etc.

- Atomic memory operations (not in the class hw)

  atomicAdd, **atomicMin**, **atomicAnd**, **atomicCAS**, etc.

- Texture accesses in kernels

```
texture<float,2> my_texture;     // declare texture reference


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

```
// 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];
}
```

Host Code

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

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
  - Very interesting recent 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

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

Summary: CUDA exposes parallelism

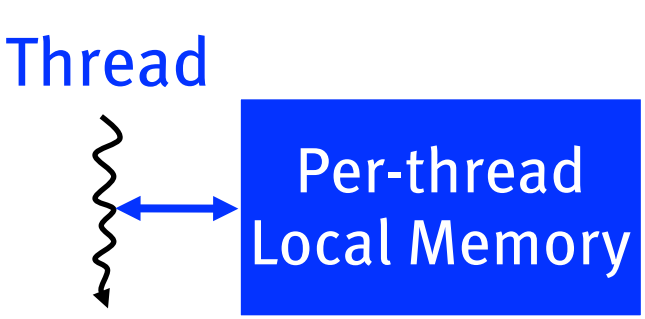
- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels

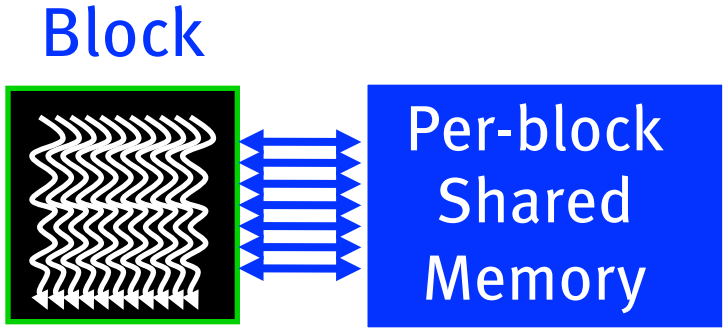
GPUs are not PRAMs

- Computational hierarchy, motivated by …
- … memory hierarchy
- Cost of execution divergence (SIMD vs. MIMD)
- Memory layout (coalescing)
- Synchronization (not a bulk synchronous model)

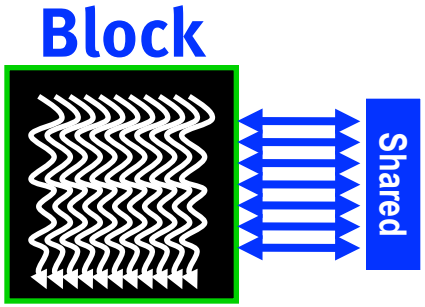
How would you program (in CUDA) two separate (parallel) tasks (to run in parallel)? How would that map to the GPU?

Memory model





Using per-block shared memory



- Variables shared across block

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

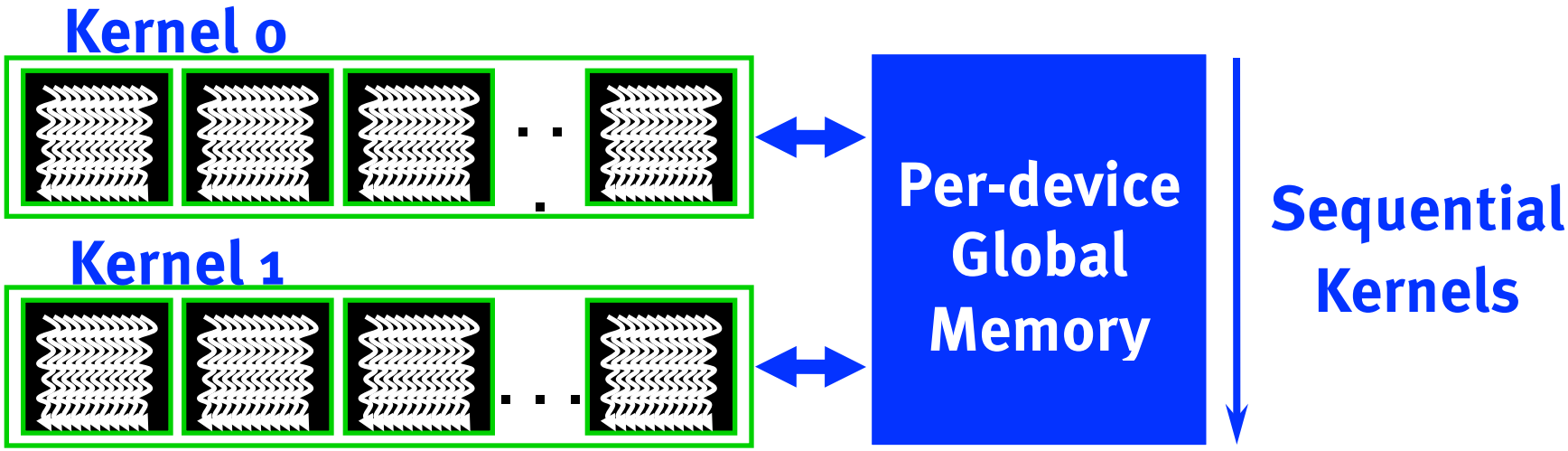
- Scratchpad memory

```
__shared__ int scratch[blocksize];

scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

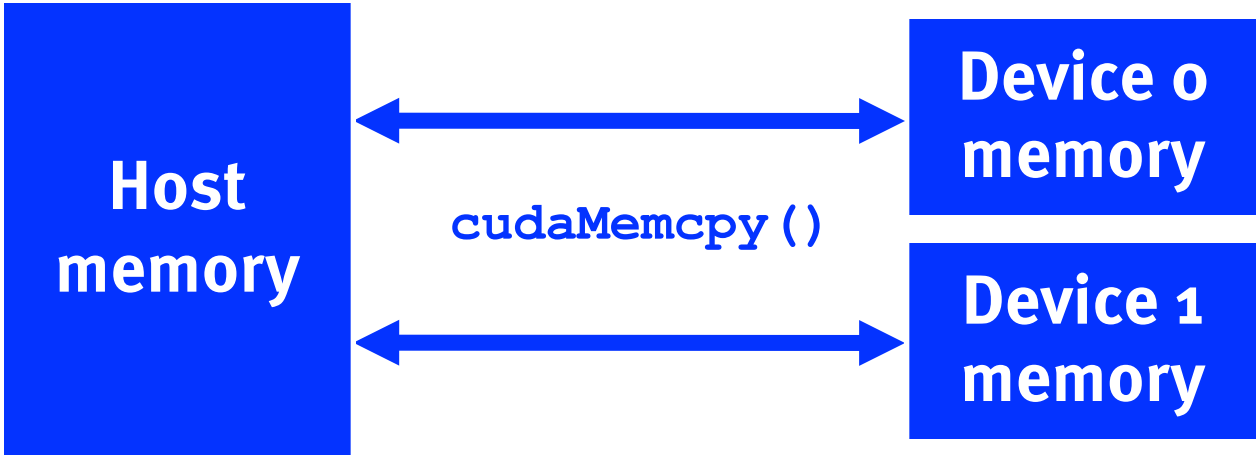
- Communicating values between threads

```
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()`, ...

- 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**

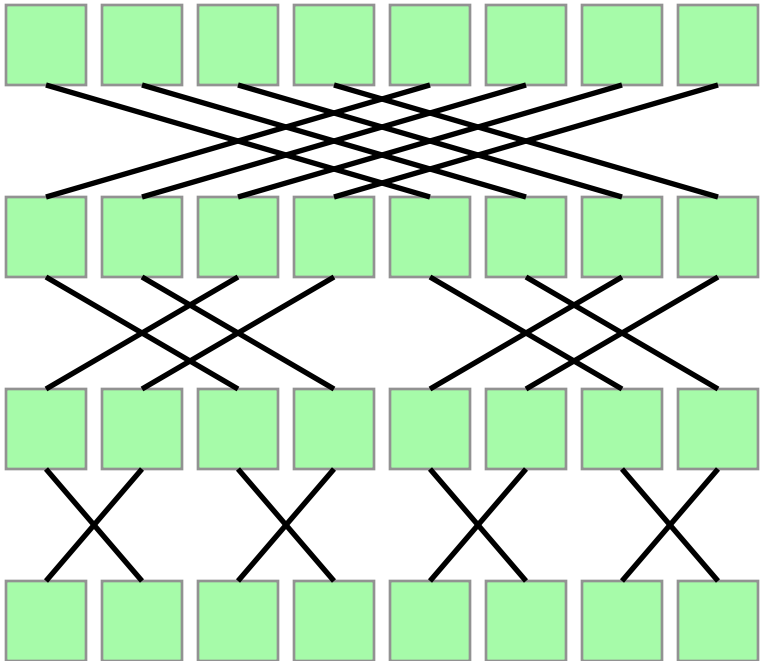
```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

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);
```

Example: Parallel Reduction

- Summing up a sequence with 1 thread:

```
int sum = 0;
for(int i=0; i<N; ++i)  sum += x[i];
```

- Parallel reduction builds a summation tree
  - each thread holds 1 element
  - stepwise partial sums
  - *n* threads need log *n* steps
  - one possible approach: Butterfly pattern

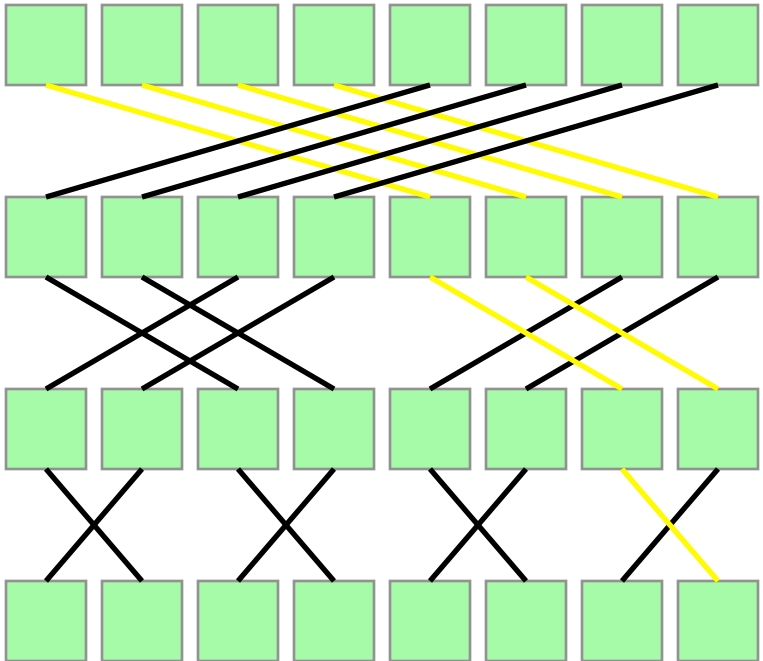


Example: Parallel Reduction

- Summing up a sequence with 1 thread:

```
int sum = 0;
for(int i=0; i<N; ++i)  sum += x[i];
```

- Parallel reduction builds a summation tree
  - each thread holds 1 element
  - stepwise partial sums
  - *n* threads need log *n* steps
  - one possible approach: Butterfly pattern

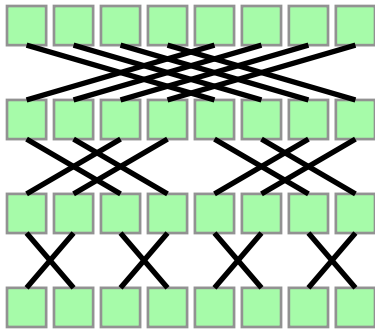


Parallel Reduction for 1 Block

```
// INPUT: Thread i holds value x_i
int i = threadIdx.x;
__shared__ int sum[blocksize];
```

```
// One thread per element
sum[i] = x_i; __syncthreads();
```

```
for(int bit=blocksize/2; bit>0; bit/=2)
{
    int t=sum[i]+sum[i^bit]; __syncthreads();
    sum[i]=t;                __syncthreads();
}
// OUTPUT: Every thread now holds sum in sum[i]
```



Example: Serial SAXPY routine

Serial program: compute y = α x + y with a loop

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = a*x[i] + y[i];
}
```

Serial execution: call a function

```
saxpy_serial(n, 2.0, x, y);
```

Example: Parallel SAXPY routine

Parallel program: compute with 1 thread per element

```
__global__
void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    if( i<n )  y[i] = a*x[i] + y[i];
}
```

Parallel execution: launch a kernel

```
uint size   = 256;        // threads per block
uint blocks = (n + size-1) / size; // blocks needed

saxpy_parallel<<<blocks, size>>>(n, 2.0, x, y);
```

SAXPY in PTX 1.0 ISA

```
cvt.u32.u16  $blockid, %ctaid.x;  // Calculate i from thread/block IDs
cvt.u32.u16  $blocksize, %ntid.x;
cvt.u32.u16  $tid, %tid.x;
mad24.lo.u32 $i, $blockid, $blocksize, $tid;
ld.param.u32 $n, [N];       // Nothing to do if n ≤ i
setp.le.u32  $p1, $n, $i;
@$p1 bra  $L_finish;


mul.lo.u32    $offset, $i, 4;  // Load y[i]
ld.param.u32 $yaddr, [Y];
add.u32   $yaddr, $yaddr, $offset;
ld.global.f32    $y_i, [$yaddr+0];
ld.param.u32 $xaddr, [X];      // Load x[i]
add.u32   $xaddr, $xaddr, $offset;
ld.global.f32    $x_i, [$xaddr+0];


ld.param.f32 $alpha, [ALPHA]; // Compute and store alpha*x[i] + y[i]
mad.f32   $y_i, $alpha, $x_i, $y_i;
st.global.f32    [$yaddr+0], $y_i;

$L_finish:    exit;
```

Sparse matrix-vector multiplication

- Sparse matrices have relatively few non-zero entries
- Frequently $O(n)$ rather than $O(n^2)$
- Only store & operate on these non-zero entries

Example: Compressed Sparse Row (CSR) Format

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

		Row 0	Row 2	Row 3
Non-zero values	`Av[7] = { 3, 1, 2, 4, 1, 1, 1 };`			
Column indices	`Aj[7] = { 0, 2, 1, 2, 3, 0, 3 };`			
Row pointers	`Ap[5] = { 0, 2, 2, 5, 7 };`			

Sparse matrix-vector multiplication

```
float multiply_row(uint  rowsize, // number of non-zeros in row
                   uint  *Aj,   // column indices for row
                   float *Av,   // non-zero entries for row
                   float *x)    // the RHS vector
{
    float sum = 0;

    for(uint column=0; column<rowsize; ++column)
        sum += Av[column] * x[Aj[column]];

    return sum;
}
```

Row 0 Row 2 Row 3

Non-zero values `Av[7] = { 3, 1, 2, 4, 1, 1, 1 };`

Column indices `Aj[7] = { 0, 2, 1, 2, 3, 0, 3 };`

Row pointers `Ap[5] = { 0, 2, 2, 5, 7 };`

Sparse matrix-vector multiplication

```
float multiply_row(uint size, uint *Aj,
float *Av, float *x);


void csrmul_serial(uint *Ap, uint *Aj, float *Av,
                   uint num_rows, float *x, float *y)
{
    for(uint row=0; row<num_rows; ++row)
    {
        uint row_begin = Ap[row];
        uint row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin,
                              Aj+row_begin,
                              Av+row_begin,
                              x);
    }
}
```

Sparse matrix-vector multiplication

```
float multiply_row(uint size, uint *Aj,
float *Av, float *x);


__global__
void csrmul_kernel(uint *Ap, uint *Aj, float *Av,
                   uint num_rows, float *x, float *y)
{
    uint row = blockIdx.x*blockDim.x + threadIdx.x;

    if( row<num_rows )
    {
        uint row_begin = Ap[row];
        uint row_end   = Ap[row+1];

        y[row] = multiply_row(row_end-row_begin,
                              Aj+row_begin, Av+row_begin, x);
    }
}
```

Adding a simple caching scheme

```
__global__ void csrmul_cached(... ... ... ... ... ...) {
    uint begin = blockIdx.x*blockDim.x,  end = begin+blockDim.x;
    uint row   = begin + threadIdx.x;

    __shared__ float cache[blocksize];          // array to cache rows
    if( row<num_rows)  cache[threadIdx.x] = x[row];        // fetch to cache
    __syncthreads();

    if( row<num_rows ) {
        uint row_begin = Ap[row],  row_end = Ap[row+1];  float sum = 0;

        for(uint col=row_begin; col<row_end; ++col) {
            uint j = Aj[col];

            // Fetch from cached rows when possible
            float x_j = (j>=begin && j<end) ? cache[j-begin] : x[j];

            sum += Av[col] * x_j;
        }

        y[row] = sum;
    }
}
```

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