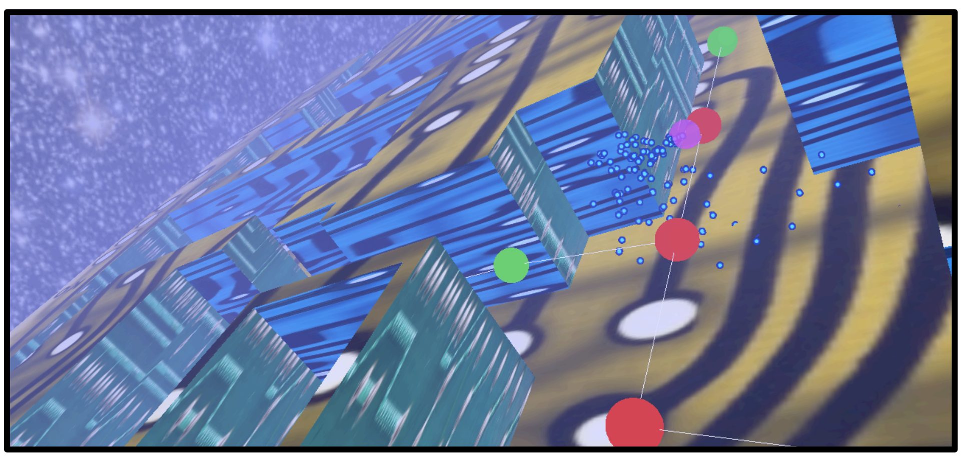
Agenda

- A Shameless self-promotion
- Introduction to GPGPUs and Cuda Programming Model
- The Cuda Thread Hierarchy
- The Cuda Memory Hierarchy
- Mapping Cuda to Nvidia GPUs
- As much of the OpenCL information as I can get through

First: Shameless Advertising

- Kurt Keutzer and I are teaching CS194-15: Engineering Parallel Software, a new undergraduate course on parallel computing at UC Berkeley
- We'll teach everything you need to know to write efficient, correct parallel software for manycore processors
- Plenty of practical experience writing parallel code for Multi-Core CPUs and GPUs in efficiency-level languages
  - In a small video game I have been developing for this purpose

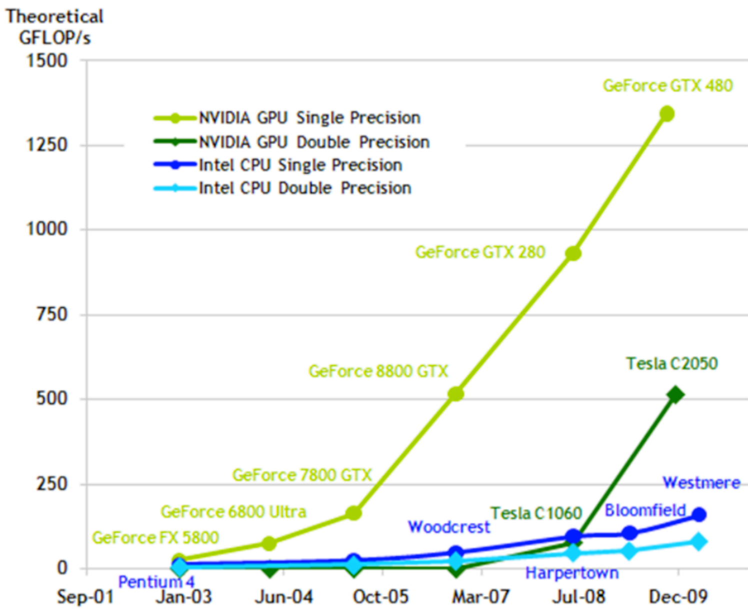


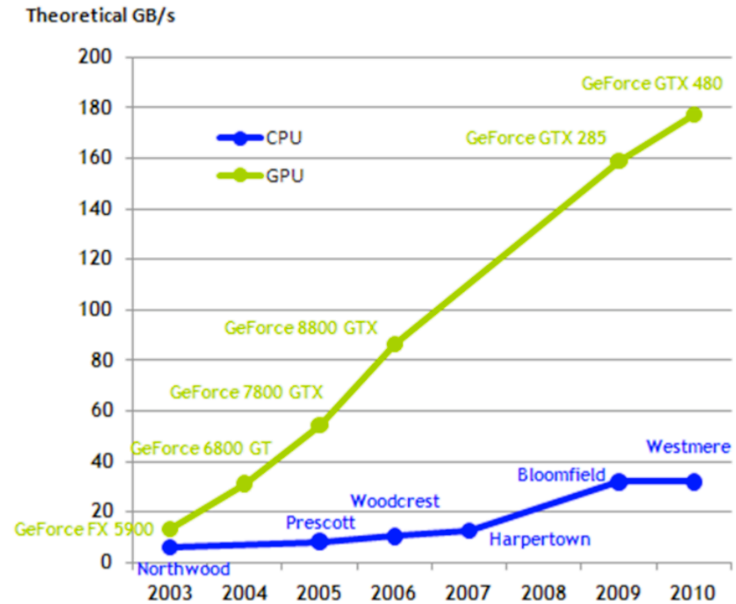
Screenshot showing an NPC object (Pink) searching a maze for its target. The navigation graph is shown in red (visited nodes) and green (unvisited nodes).

Agenda

- A Shameless self-promotion
- **Introduction to GPGPUs and Cuda Programming Model**
- The Cuda Thread Hierarchy
- The Cuda Memory Hierarchy
- Mapping Cuda to Nvidia GPUs
- As much of the OpenCL information as I can get through

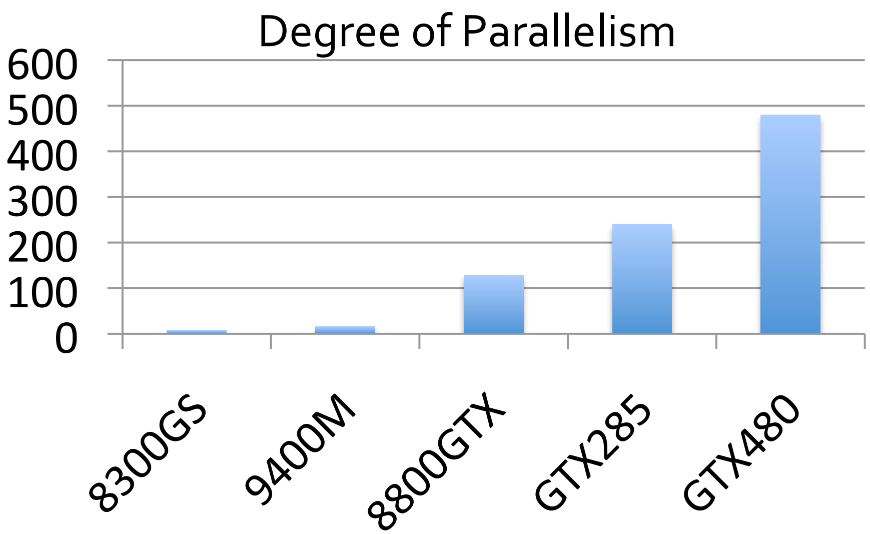
Evolution of GPU Hardware



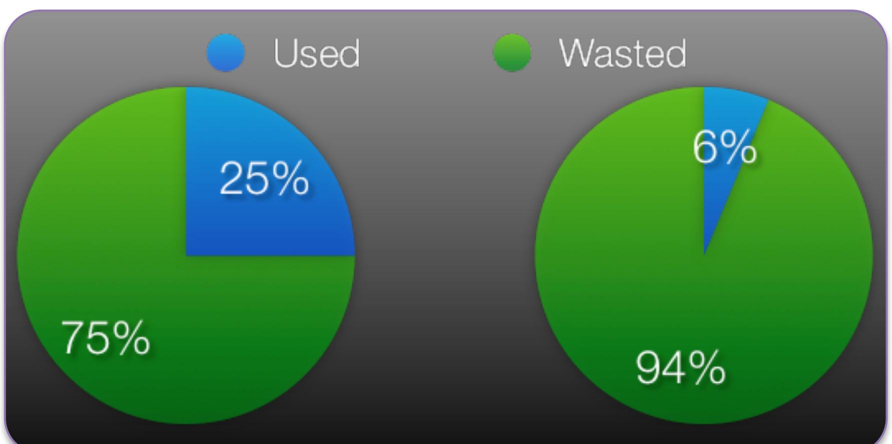


- CPU architectures have used Moore's Law to increase:
  - The amount of on-chip cache
  - The complexity and clock rate of processors
  - Single-threaded performance of **<u>legacy</u>** workloads
- GPU architectures have used Moore's Law to:
  - Increase the degree of on-chip parallelism and DRAM bandwidth
  - Improve the flexibility and performance of Graphics applications
  - Accelerate general-purpose **<u>Data-Parallel</u>** workloads

Cuda Programming Model Goals



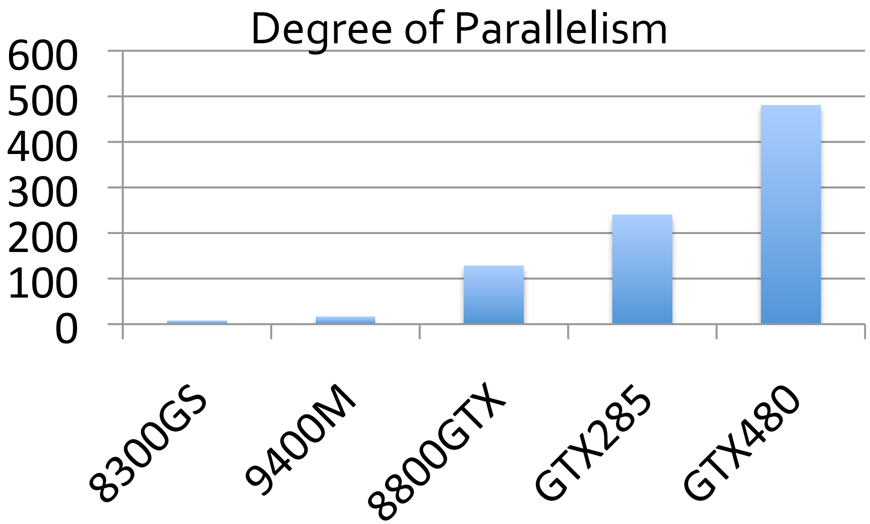
Provide an inherently scalable environment for Data-Parallel programming across a wide range of processors (Nvidia only makes GPUs, however)

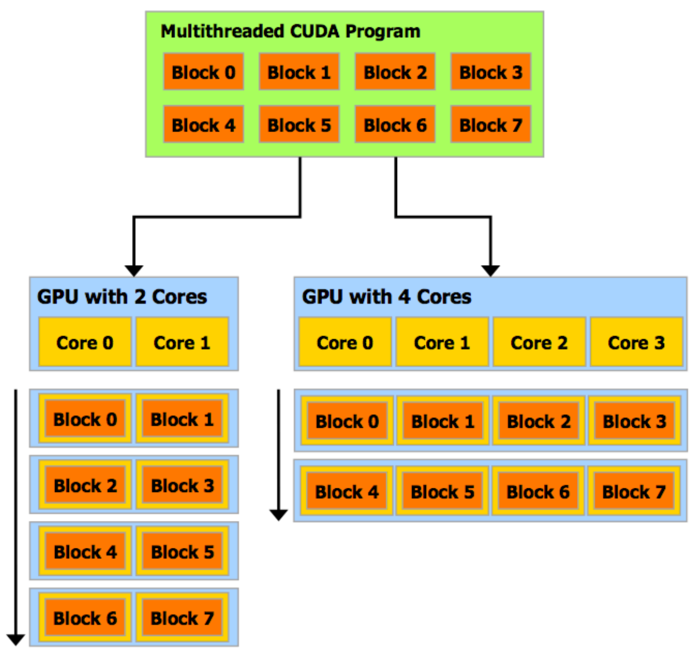


4 way SIMD (SSE) 16 way SIMD (LRB)

Make SIMD hardware accessible to general-purpose programmers. Otherwise, large fractions of the available execution hardware are wasted!

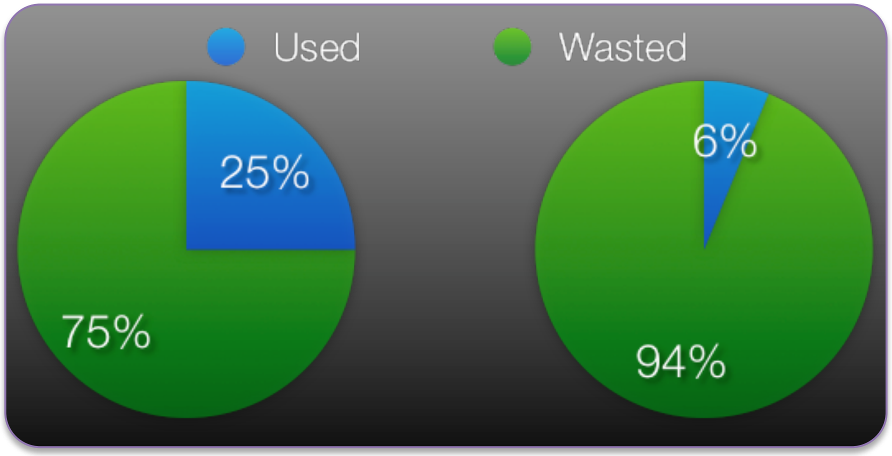
Cuda Goals: Scalability



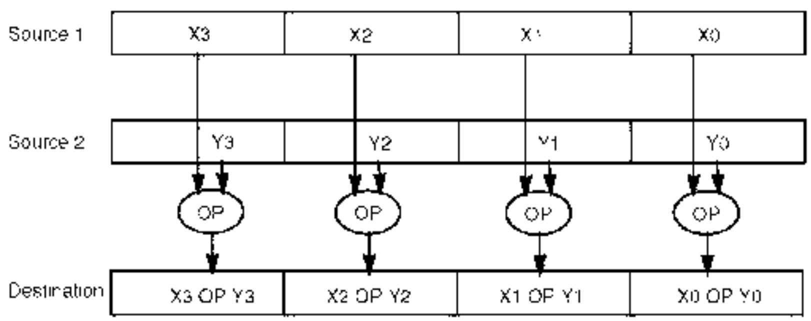


- Cuda expresses many independent blocks of computation that can be run in any order
- Much of the inherent scalability of the Cuda Programming model stems from batched execution of "Thread Blocks"
- Between GPUs of the same generation, many programs achieve linear speedup on GPUs with more “Cores”

Cuda Goals: SIMD Programming



4 way SIMD (SSE) 16 way SIMD (LRB)



- Hardware architects love SIMD, since it permits a very space- and energy-efficient implementation
- However, standard SIMD instructions on CPUs are inflexible, and difficult to use, difficult for a compiler to target
- The Cuda Thread abstraction will provide programmability at the cost of additional hardware

Cuda C Language Extensions

- Code to run on the GPU is written in standard C/C++ syntax with a minimal set of extensions:
  - Provide a MIMD Thread abstraction for SIMD execution
  - Enable specification of Cuda Thread Hierarchies
  - Synchronization and data-sharing within Thread Blocks
  - Library of intrinsic functions for GPU-specific functionality

```
__global__  void KernelFunc(...);   // define a kernel callable from host
__device__  void DeviceFunc(...);   // function callable only on the device
__device__  int  GlobalVar;       // variable in device memory
__shared__  int  SharedVar;       // in per-block shared memory
KernelFunc<<<500, 128>>>(...);      // 500 blocks, 128 threads each
// Thread indexing and identification
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
__syncthreads();    // thread block synchronization intrinsic
sinf, powf, atanf, ceil, min, sqrtf, … // <math.h> functionality
```

Cuda Host Runtime Support

- Cuda is inherently a Heterogeneous programming model
  - Sequential code runs in a CPU "Host Thread", and parallel "Device" code runs on the many cores of a GPU
  - The Host and the Device communicate via a PCI-Express link
  - The PCI-E link is slow (high latency, low bandwidth): it is desirable to minimize the amount of data transferred and the number of transfers
- Allocation/Deallocation of memory on the GPU:
  - `cudaMalloc(void**, int), cudaFree(void*)`
- Memory transfers to/from the GPU:
  - `cudaMemcpy(void*,void*,int, dir)`
  - `dir` is `cudaMemcpy{Host,Device}To{Host,Device}`

Hello World: Vector Addition

```
// Compute sum of length-N vectors: C = A + B
void
vecAdd (float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    a = new float[N];
  // ... allocate other arrays, fill with data


    vecAdd (a, b, c, N);
}
```

Hello World: Vector Addition

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays, fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Cuda Software Environment

- nvcc compiler works much like icc or gcc: compiles C++ source code, generates binary executable
- Nvidia Cuda OS driver manages low-level interaction with device, provides API for C++ programs
- Nvidia Cuda SDK has many code samples demonstrating various Cuda functionalities
- Library support is continuously growing:
  - CUBLAS for basic linear algebra
  - CUFFT for Fourier Fransforms
  - CULapack (3rd party proprietary) linear solvers, eigensolvers, ...
- OS-Portable: Linux, Windows, Mac OS
- A lot of momentum in Industrial adoption of Cuda!

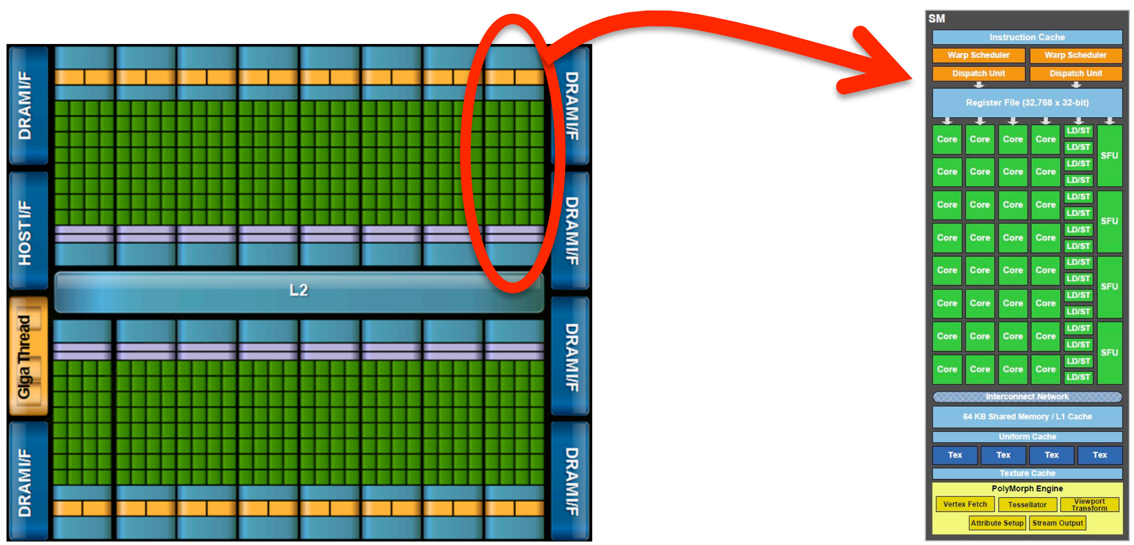
http://developer.nvidia.com/object/cuda_3_1_downloads.html

Agenda

- A Shameless self-promotion
- Introduction to GPGPUs and Cuda Programming Model
- **The Cuda Thread Hierarchy**
- The Cuda Memory Hierarchy
- Mapping Cuda to Nvidia GPUs
- As much of the OpenCL information as I can get through

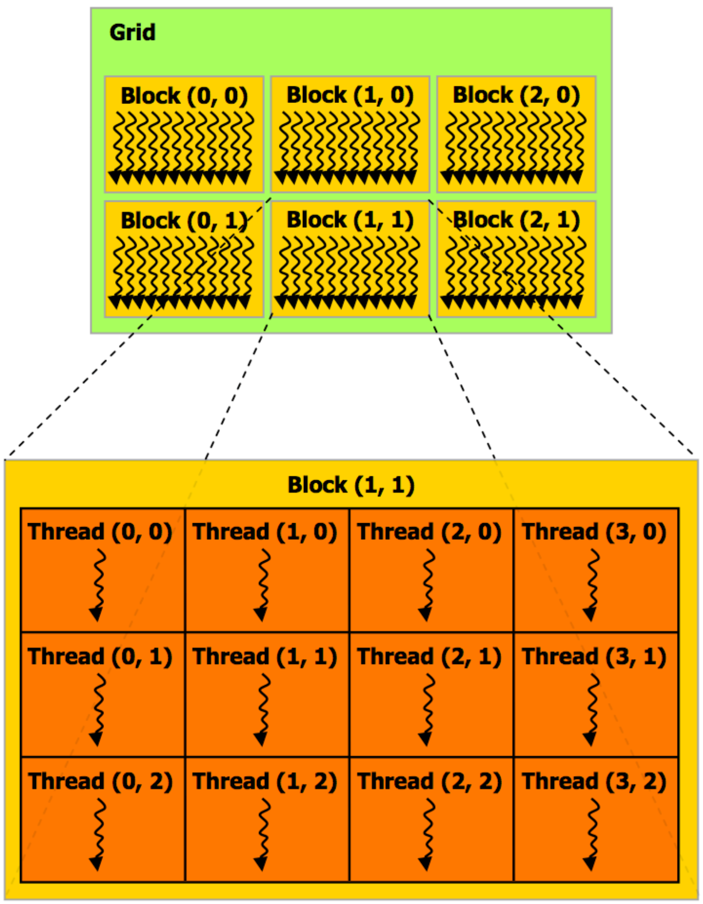
Nvidia Cuda GPU Architecture

- I'll discuss some details of Nvidia's GPU architecture simultaneously with discussing the Cuda Programming Model
  - The Cuda Programming Model is a set of data-parallel extensions to C, amenable to implementation on GPUs, CPUs, FPGAs, ...
- Cuda GPUs are a collection of “Streaming Multiprocessors”
  - Each SM is analogous to a core of a Multi-Core CPU
- Each SM is a collection of SIMD execution pipelines (Scalar Processors) that share control logic, register file, and L1 Cache



Cuda Thread Hierarchy

- Parallelism in the Cuda Programming Model is expressed as a 4-level Hierarchy:



- A ***Stream*** is a list of ***Grids*** that execute in-order. Fermi GPUs execute multiple Streams in parallel
- A ***Grid*** is a set of up to 2^{32} ***Thread Blocks*** executing the same kernel
- A ***Thread Block*** is a set of up to 1024 [512 pre-Fermi] ***Cuda Threads***
- Each ***Cuda Thread*** is an independent, lightweight, scalar execution context
    - Groups of 32 threads form ***Warps*** that execute in lockstep SIMD

What is a Cuda Thread?

- Logically, each Cuda Thread is its own very lightweight **independent MIMD execution context**
  - Has its own control flow and PC, register file, call stack, ...
  - Can access any GPU global memory address at any time
  - Identifiable uniquely within a grid by the five integers: `threadIdx.{x,y,z}, blockIdx.{x,y}`
- **Very fine granularity**: do not expect any single thread to do a substantial fraction of an expensive computation
  - At full occupancy, each Thread has 21 32-bit registers
  - ... 1,536 Threads share a 64 KB L1 Cache / `__shared__` mem
  - GPU has no operand bypassing networks: functional unit latencies must be hidden by multithreading or ILP (e.g. from loop unrolling)

What is a Cuda Warp?

- The Logical SIMD Execution width of the Cuda processor
- A group of 32 Cuda Threads that execute simultaneously
  - Execution hardware is most efficiently utilized when all threads in a warp execute instructions from the same PC.
  - If threads in a warp ***diverge*** (execute different PCs), then some execution pipelines go unused (predication)
  - If threads in a warp access aligned, contiguous blocks of DRAM, the accesses are ***coalesced*** into a single high-bandwidth access
  - Identifiable uniquely by dividing the Thread Index by 32
- Technically, warp size could change in future architectures
  - But many existing programs would break

What is a Cuda Thread Block?

- A Thread Block is a **virtualized multi-threaded core**
  - Number of scalar threads, registers, and `__shared__` memory are configured dynamically at kernel-call time
  - Consists of a number (1-1024) of Cuda Threads, who all share the integer identifiers `blockIdx.{x,y}`
- ... executing a **data parallel task** of moderate granularity
  - The cacheable working-set should fit into the 128 KB (64 KB, pre-Fermi) Register File and the 64 KB (16 KB) L1
  - Non-cacheable working set limited by GPU DRAM capacity
  - All threads in a block share a (small) instruction cache
- Threads within a block synchronize via barrier-intrinsics and communicate via fast, on-chip shared memory

What is a Cuda Grid?

- A set of Thread Blocks performing related computations
  - All threads in a single kernel call have the same entry point and function arguments, initially differing only in **`blockIdx.{x,y}`**
  - Thread blocks in a grid may execute any code they want, e.g. `switch (blockIdx.x) { ... }` incurs no extra penalty
- Performance portability/scalability requires many blocks per grid: 1-8 blocks execute on each SM
- Thread blocks of a kernel call must be **parallel sub-tasks**
  - Program must be valid for ***any interleaving*** of block executions
  - The flexibility of the memory system technically allows Thread Blocks to communicate and synchronize in arbitrary ways ...
  - E.G. Shared Queue index: **OK!** Producer-Consumer: **RISKY!**

What is a Cuda Stream?

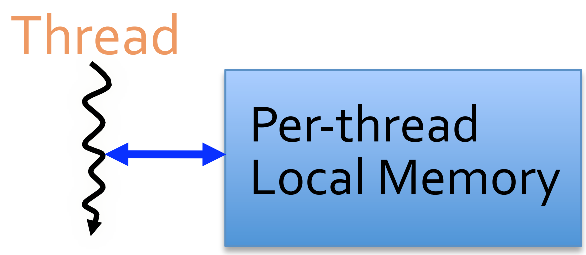
- A sequence of commands (kernel calls, memory transfers) that execute in order.
- For multiple kernel calls or memory transfers to execute concurrently, the application must specify multiple streams.
  - Concurrent Kernel execution will only happen on Fermi
  - On pre-Fermi devices, Memory transfers will execute concurrently with Kernels

```
cudaStream_t s0, s1;
cudaStreamCreate (&s0);  cudaStreamCreate (&s1);

cudaMemcpyAsync (a0, cpu_a0, N0*sizeof(float),
                 cudaMemcpyHostToDevice, s0);
vecAdd <<<N0/256, 256, 0, s0>>> (a0, b0, c0, N0);

cudaMemcpyAsync (a1, cpu_a1, N1*sizeof(float),
                 cudaMemcpyHostToDevice, s1);
vecAdd <<<N1/256, 256, 0, s1>>> (a1, b1, c1, N1);
```

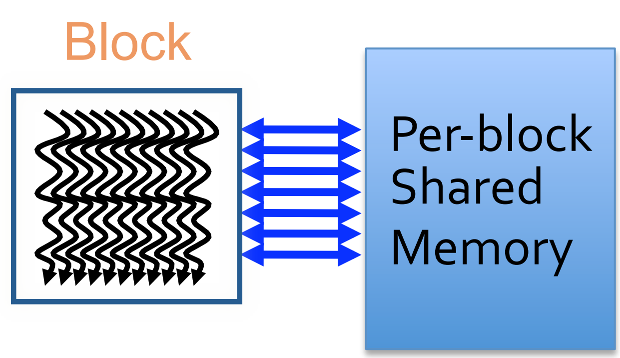
Agenda

- A Shameless self-promotion
- Introduction to GPGPUs and Cuda Programming Model
- The Cuda Thread Hierarchy
- **The Cuda Memory Hierarchy**
- Mapping Cuda to Nvidia GPUs
- As much of the OpenCL information as I can get through

Cuda Memory Hierarchy



- Each Cuda Thread has private access to a configurable number of registers
  - The 128 KB (64 KB) SM register file is partitioned among all resident threads
  - The Cuda program can trade degree of thread block concurrency for amount of per-thread state
  - Registers, stack spill into (cached, on Fermi) “local” DRAM if necessary

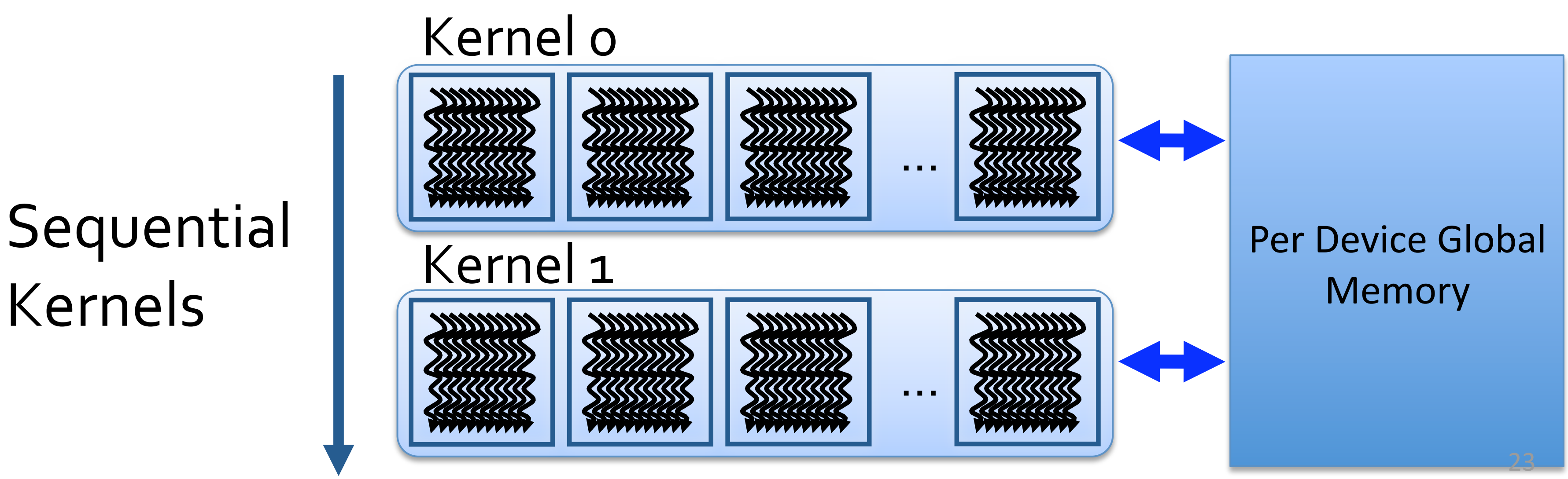


- Each Thread Block has private access to a configurable amount of scratchpad memory
  - The Fermi SM’s 64 KB SRAM can be configured as 16 KB L1 cache + 48 KB scratchpad, or vice-versa*
  - Pre-Fermi SM’s have 16 KB scratchpad only
  - The available scratchpad space is partitioned among resident thread blocks, providing another concurrency-state tradeoff

* selected via cudaFuncSetCacheConfig()

Cuda Memory Hierarchy

- Thread blocks in all Grids share access to a large pool of "Global" memory, separate from the Host CPU's memory.
  - Global memory holds the application's persistent state, while the thread-local and block-local memories are temporary
  - Global memory is much more expensive than on-chip memories: O(100)x latency, O(1/50)x (aggregate) bandwidth
- On Fermi, Global Memory is cached in a 768KB shared L2

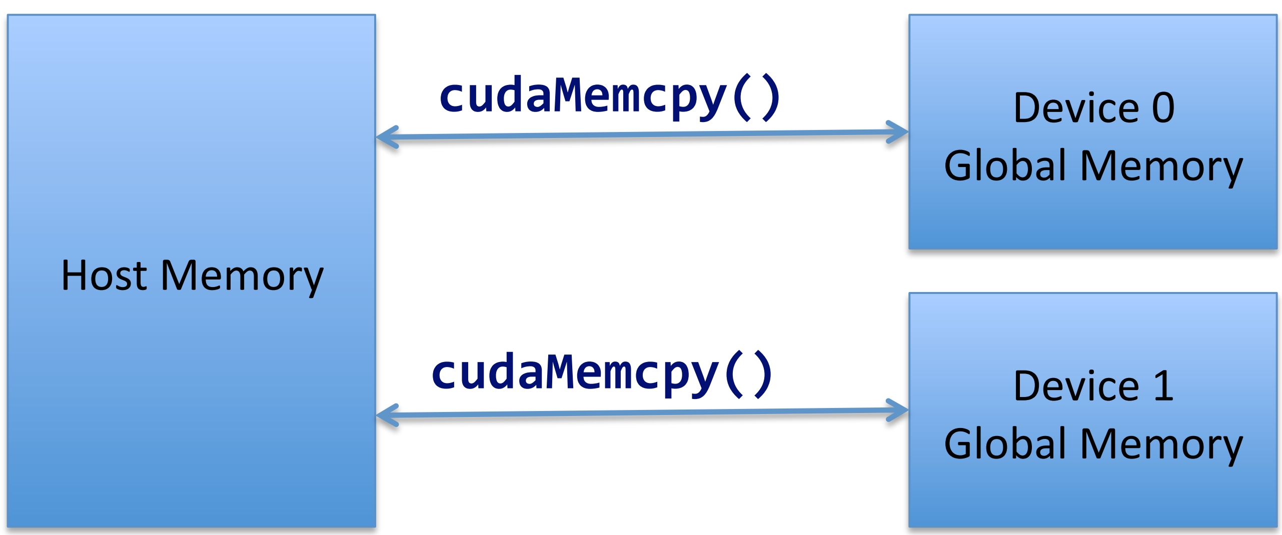


Cuda Memory Hierarchy

- There are other read-only components of the Memory Hierarchy that exist due to the Graphics heritage of Cuda
- The 64 KB Cuda **Constant Memory** resides in the same DRAM as global memory, but is accessed via special read-only 8 KB per-SM caches
- The Cuda **Texture Memory** also resides in DRAM and is accessed via small per-SM read-only caches, but also includes interpolation hardware
  - This hardware is crucial for graphics performance, but only occasionally is useful for general-purpose workloads
- The behaviors of these caches are highly optimized for their roles in graphics workloads.

Cuda Memory Hierarchy

- Each Cuda device in the system has its own Global memory, separate from the Host CPU memory
  - Allocated via cudaMalloc()/cudaFree() and friends
- Host ⇔ Device memory transfers are via cudaMemcpy() over PCI-E, and are extremely expensive
  - microsecond latency, ~GB/s bandwidth
- Multiple Devices managed via multiple CPU threads



Thread-Block Synchronization

- Intra-block barrier instruction `__syncthreads()` for synchronizing accesses to `__shared__` and global memory
  - To guarantee correctness, must `__syncthreads()` before reading values written by other threads
  - All threads in a block must execute the same `__syncthreads()`, or the GPU will hang (not just the same number of barriers !)
- Additional intrinsics worth mentioning here:
  - `int __syncthreads_count(int), int __syncthreads_and(int), int __syncthreads_or(int)`

```
extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```

Using per-block shared memory

- The per-block shared memory / L1 cache is a crucial resource: without it, the performance of most Cuda programs would be hopelessly DRAM-bound
- Block-shared variables can be declared statically:

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

- Software-managed scratchpad is allocated statically:

```
__shared__ int scratch[128];
scratch[threadIdx.x] = ... ;
```

- ... or dynamically:

```
extern __shared__ int scratch[];

kernel_call <<< grid_dim, block_dim, scratch_size >>> ( ... );
```

- Most intra-block communication is via shared scratchpad:

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

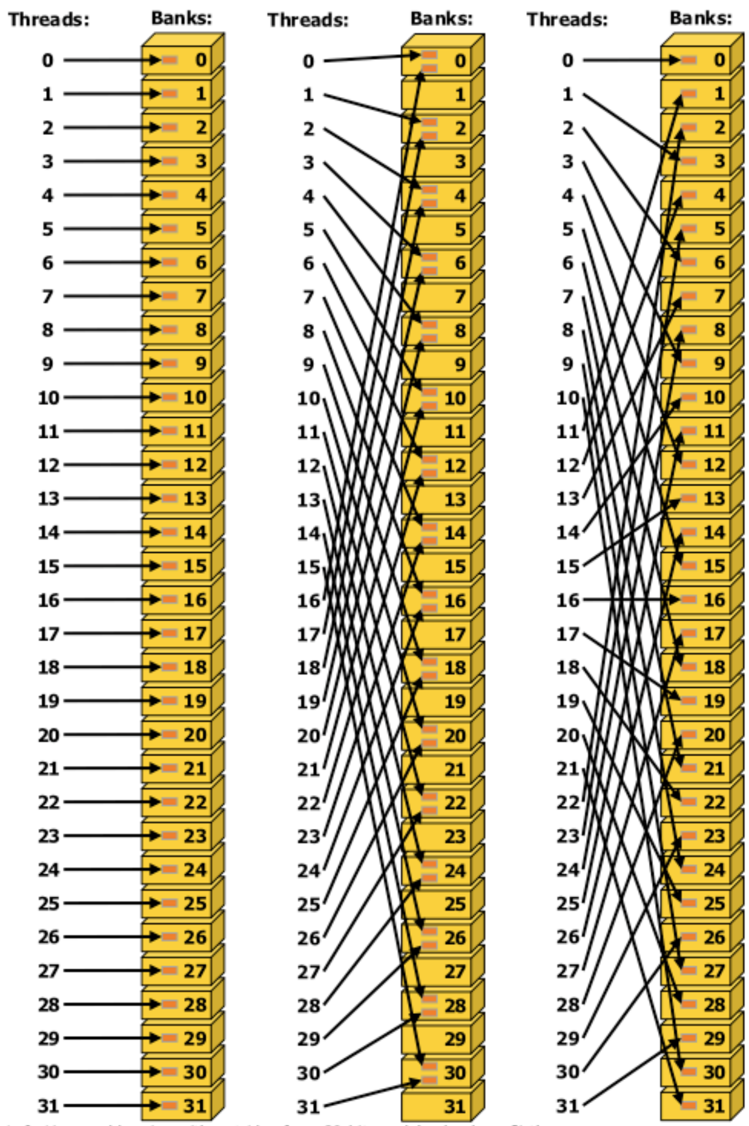
Using per-block shared memory

- Each SM has 64 KB of private memory, divided 16KB/48KB (or 48KB/16KB) into software-managed scratchpad and hardware-managed, non-coherent cache
  - Pre-Fermi, the SM memory is only 16 KB, and is usable only as software-managed scratchpad
- Unless data will be shared between Threads in a block, it should reside in registers
  - On Fermi, the 128 KB Register file is twice as large, and accessible at higher bandwidth and lower latency
  - Pre-Fermi, register file is 64 KB and equally fast as scratchpad

Shared Memory Bank Conflicts

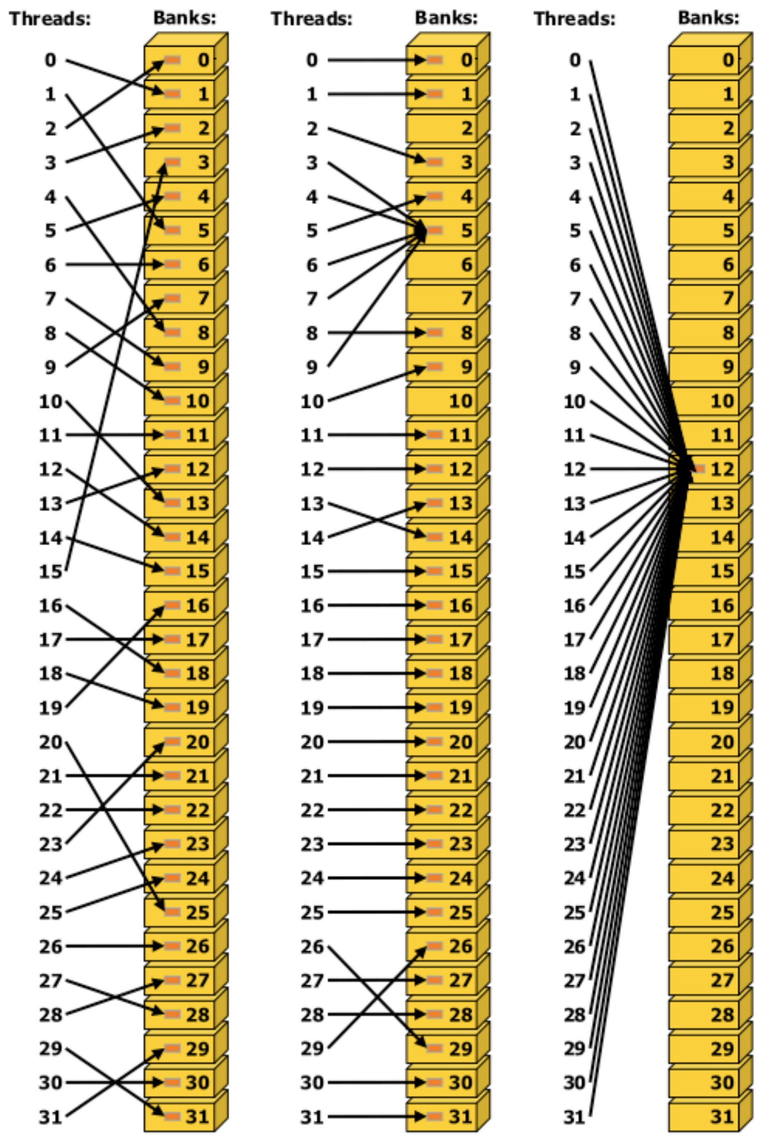
- Shared memory is ***banked***: it consists of 32 (16, pre-Fermi) independently addressable 4-byte wide memories
  - Addresses interleave: `float *p` points to a float in bank ***k,*** `p+1` points to a float in bank (***k+1) mod 32***
- Each bank can satisfy a single 4-byte access per cycle.
  - A ***bank conflict*** occurs when two threads (in the same warp) try to access the same bank in a given cycle.
  - The GPU hardware will execute the two accesses serially, and the warp's instruction will take an extra cycle to execute.
- Bank conflicts are a second-order performance effect: even serialized accesses to on-chip shared memory is faster than accesses to off-chip DRAM

Shared Memory Bank Conflicts



- Figure G-2 from Cuda C Programming Gude 3.1
- Unit-Stride access is **conflict-free**
- Stride-2 access: thread ***n*** conflicts with thread ***16+n***
- Stride-3 access is **conflict-free**

Shared Memory Bank Conflicts



- Three more cases of conflict-free access
  - Figure G-3 from Cuda C Programming Gude 3.1
- Permuations within a 32-float block are OK
- Multiple threads reading the **same memory address**
- **All threads** reading the same memory address is a ***broadcast***

Atomic Memory Operations

- Cuda provides a set of instructions which execute atomically with respect to each other
  - Allow non-read-only access to variables shared between threads in shared or global memory
  - Substantially more expensive than standard load/stores
  - Wth voluntary consistency, can implement e.g. spin locks!

```
int atomicAdd (int*,int), float atomicAdd (float*, float), ...
...
int atomicMin (int*,int),
...
int atomicExch (int*,int), float atomicExch (float*,float), ...
int atomicCAS (int*, int compare, int val), ...
```

Voluntary Memory Consistency

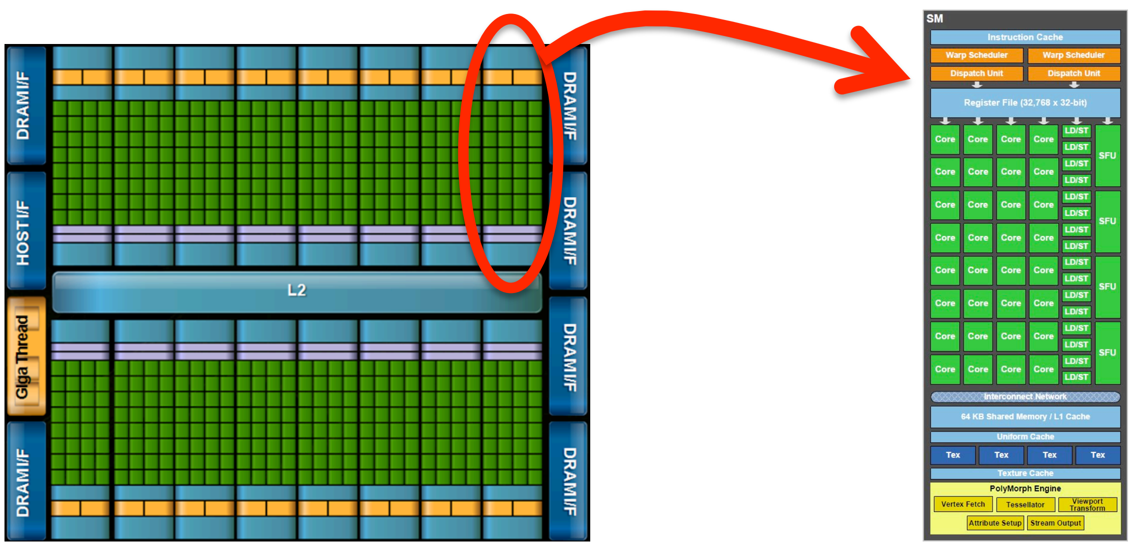
- By default, you cannot assume memory accesses are occur in the same order specified by the program
  - Although a thread's ***own*** accesses appear to that thread to occur in program order
- To enforce ordering, use ***memory fence*** instructions
  - `__threadfence_block()`: make all previous memory accesses visible to all other threads ***within the thread block***
  - `__threadfence()`: make previous ***global*** memory accesses visible to all other threads ***on the device***
- Frequently must also use the `volatile` type qualifier
  - Has same behavior as CPU C/C++: the compiler is forbidden from register-promoting values in volatile memory
  - Ensures that pointer dereferences produce load/store instructions
  - Declared as `volatile float *p; *p` must produce a memory ref.

Agenda

- A Shameless self-promotion
- Introduction to GPGPUs and Cuda Programming Model
- The Cuda Thread Hierarchy
- The Cuda Memory Hierarchy
- **Mapping Cuda to Nvidia GPUs**
- As much of the OpenCL information as I can get through

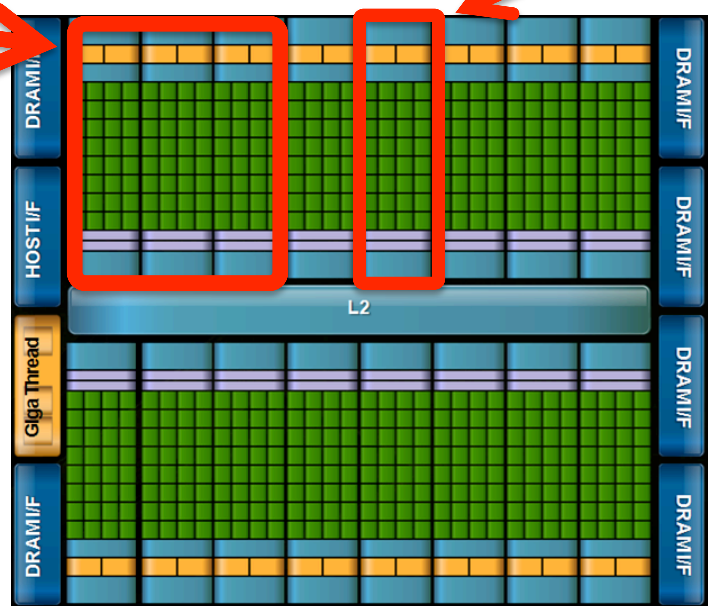
Mapping Cuda to Nvidia GPUs

- Cuda is designed to be "functionally forgiving": Easy to get correct programs running. The more time you invest in optimizing your code, the more performance you will get
- Speedup is possible with a simple "Homogeneous SPMD" approach to writing Cuda programs
- Achieving performance requires an understanding of the hardware implementation of Cuda



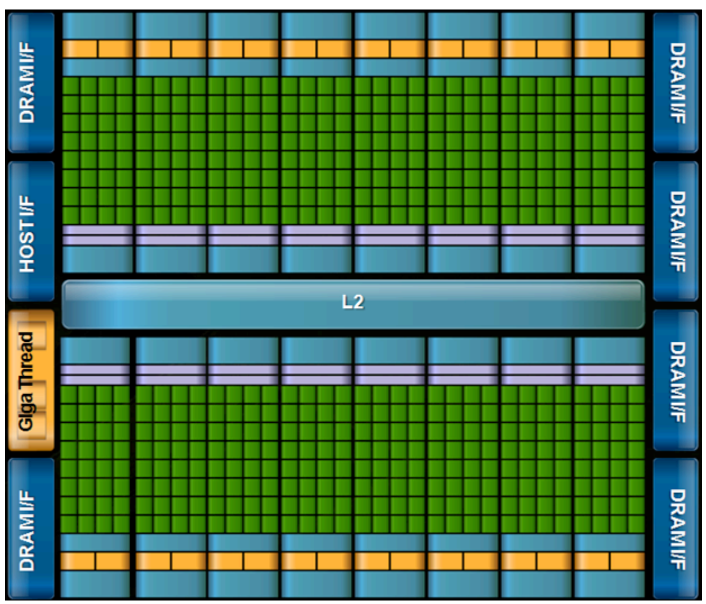
Mapping Cuda to Nvidia GPUs

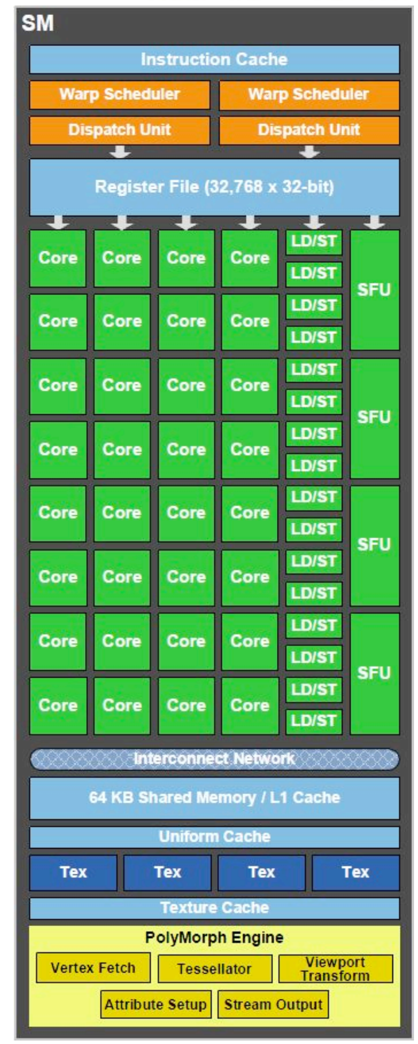
- Scalar Thread ⇔ SIMD Lane
- Warp ⇔ SIMD execution granularity
- Thread Block ⇔ Streaming Multiprocessor
- Grid ⇔ Multiple SMs
- Set of Streams ⇔ Whole GPU



Mapping Cuda to Nvidia GPUs

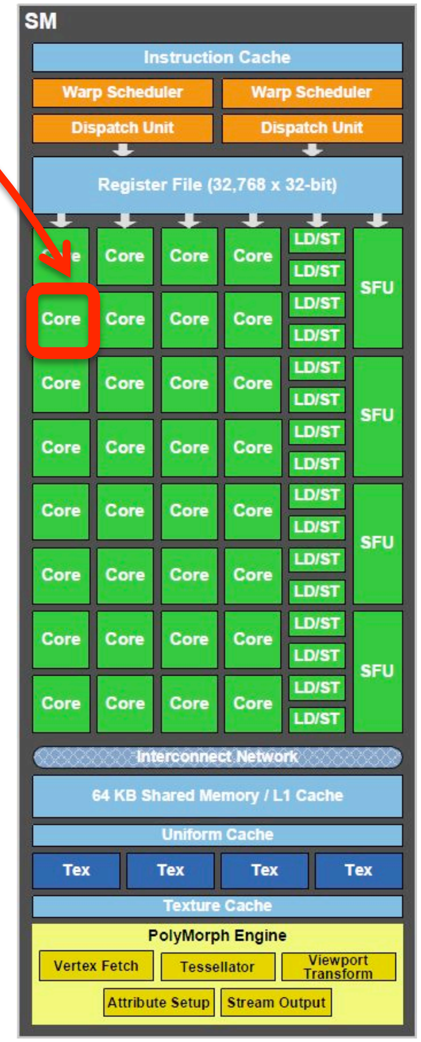
- Scalar Thread ⇔ SIMD Lane
- Warp ⇔ Logical SIMD width
- Thread Block ⇔ Streaming Multiprocessor
- Grid ⇔ Multiple SMs
- Set of Streams ⇔ Whole GPU

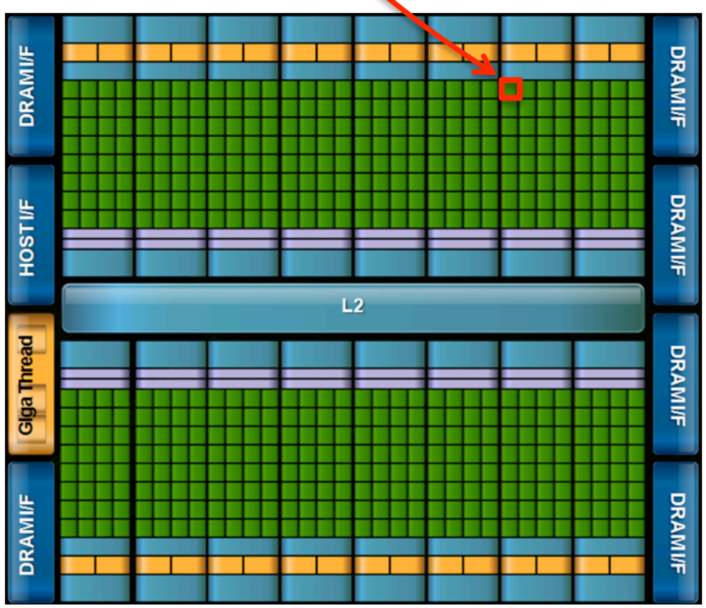




Mapping Cuda to Nvidia GPUs

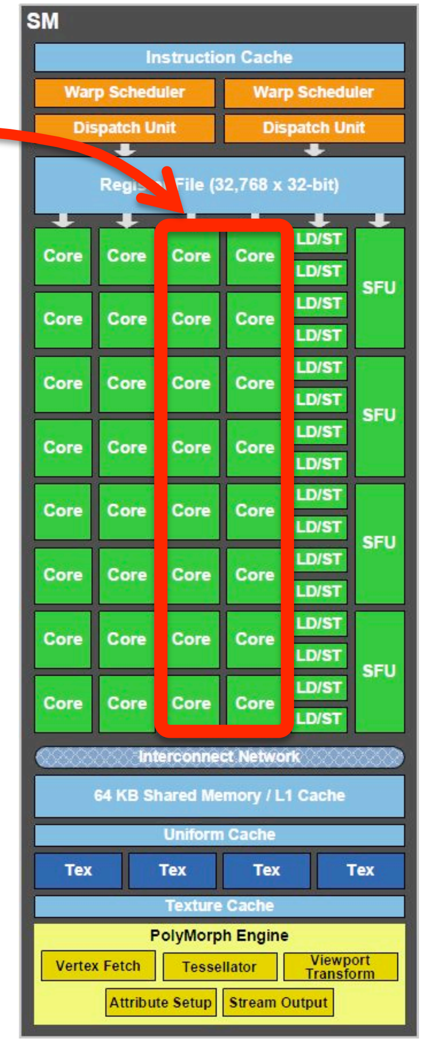
- **Scalar Thread ⇔ SIMD Lane**
- Warp ⇔ Logical SIMD width
- Thread Block ⇔ Streaming Multiprocessor
- Grid ⇔ Multiple SMs
- Set of Streams ⇔ Whole GPU

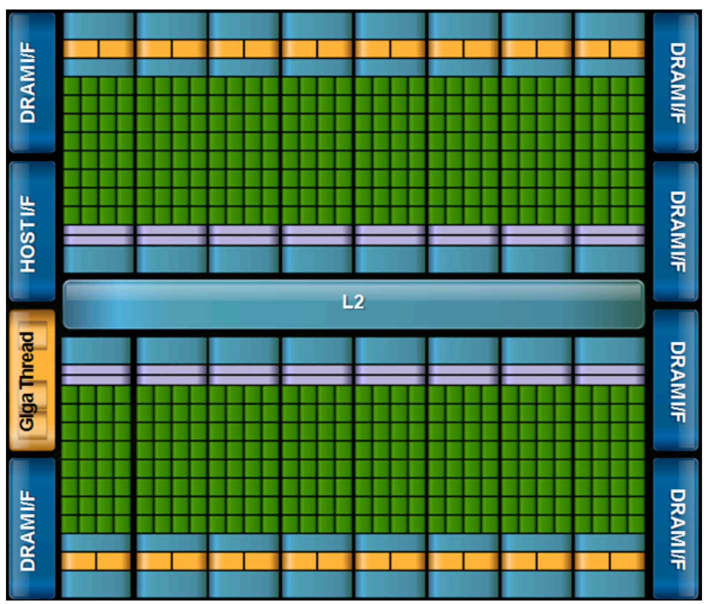




Mapping Cuda to Nvidia GPUs

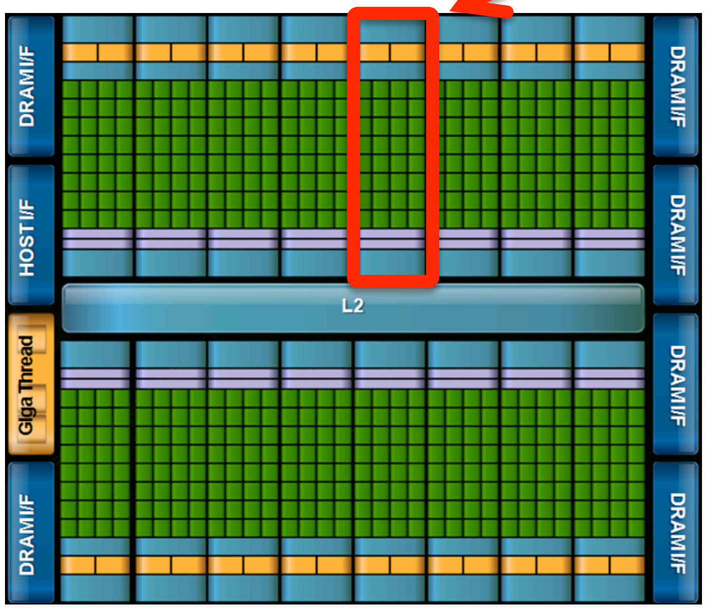
- Scalar Thread ⇔ SIMD Lane
- **Warp ⇔ Logical SIMD width**
- Thread Block ⇔ Streaming Multiprocessor
- Grid ⇔ Multiple SMs
- Set of Streams ⇔ Whole GPU





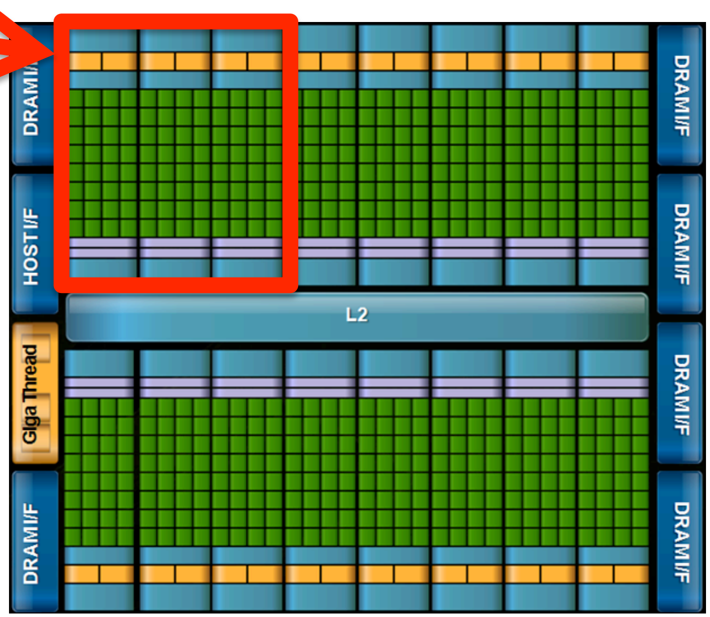
Mapping Cuda to Nvidia GPUs

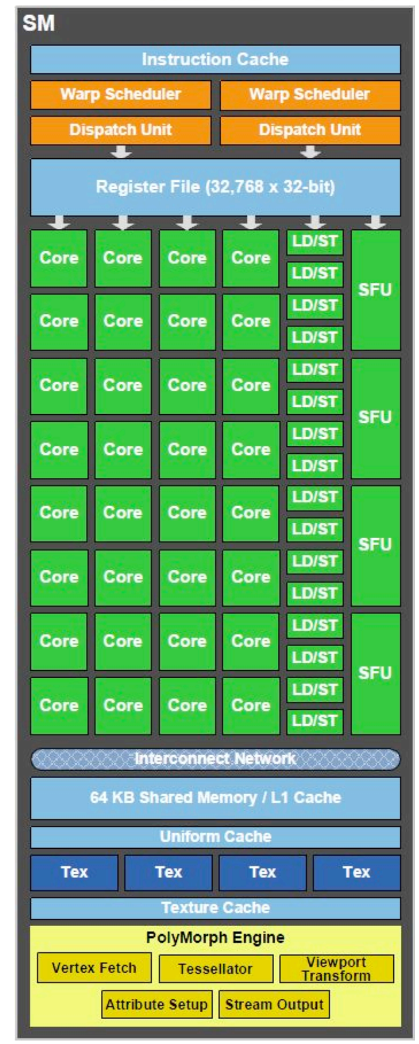
- Scalar Thread ⇔ SIMD Lane
- Warp ⇔ SIMD execution granularity
- **Thread Block ⇔ Streaming Multiprocessor**
- Grid ⇔ Multiple SMs
- Set of Streams ⇔ Whole GPU



Mapping Cuda to Nvidia GPUs

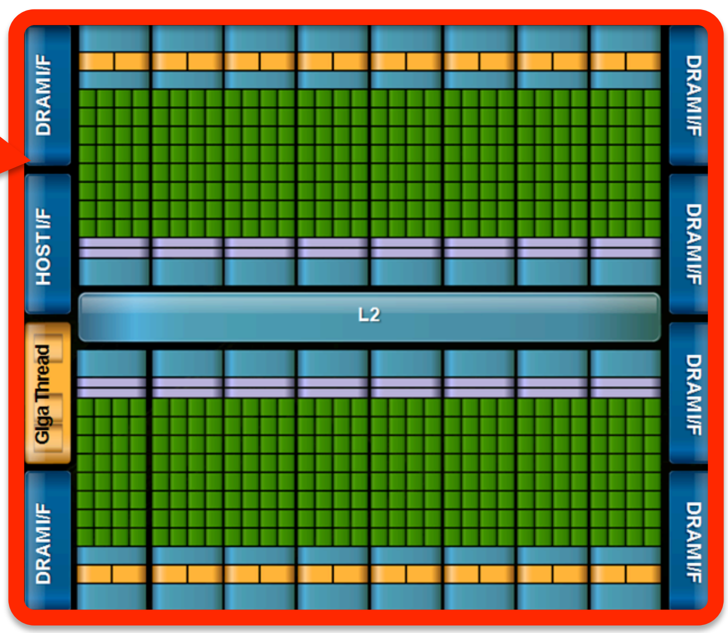
- Scalar Thread ⇔ SIMD Lane
- Warp ⇔ Logical SIMD width
- Thread Block ⇔ Streaming Multiprocessor
- **Grid ⇔ Multiple SMs**
- Set of Streams ⇔ Whole GPU

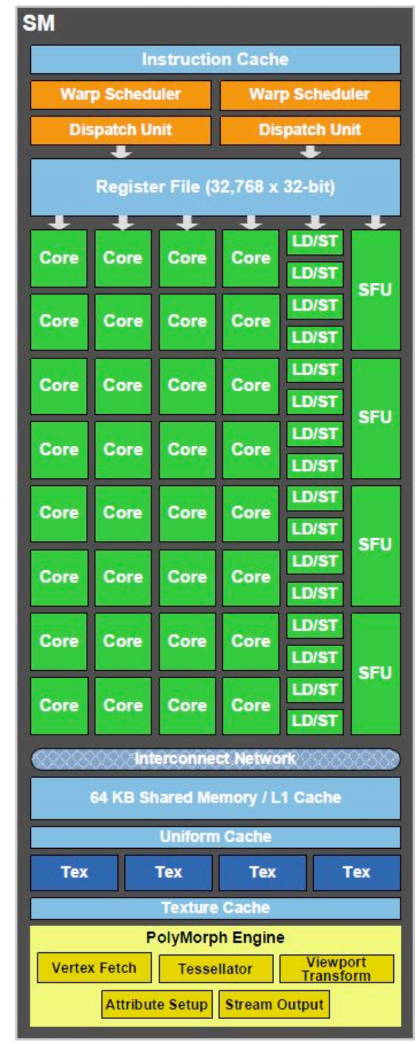




Mapping Cuda to Nvidia GPUs

- Scalar Thread ⇔ SIMD Lane
- Warp ⇔ Logical SIMD width
- Thread Block ⇔ Streaming Multiprocessor
- Grid ⇔ Multiple SMs
- **Set of Streams ⇔ Whole GPU**



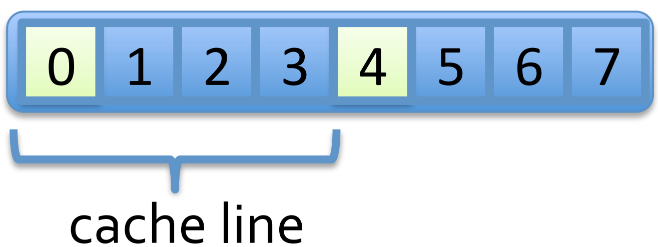


Mapping Cuda to Nvidia GPUs

- Each level of the GPU's processor hierarchy is associated with a memory resource
  - Scalar Threads / Warps: Subset of register file
  - Thread Block / SM: shared memory (l1 Cache)
  - Multiple SMs / Whole GPU: Global DRAM
- Massive multi-threading is used to hide latencies: DRAM access, functional unit execution, PCI-E transfers
- A highly performing Cuda program must carefully trade resource usage for concurrency
  - More registers per thread ⇔ fewer threads
  - More shared memory ber block ⇔ fewer blocks

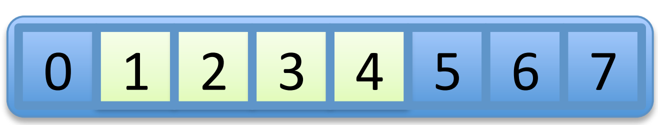
Memory, Memory, Memory

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem
  - **Memory concerns dominate performance tuning!**
- Memory is SIMD too! The memory systems of CPUs and GPUs alike require memory to be accessed in aligned blocks
  - **Sparse accesses waste bandwidth!**



2 words used, 8 words loaded:
¼ effective bandwidth

  - **Unaligned accesses waste bandwidth!**



4 words used, 8 words loaded:
½ effective bandwidth

Cuda Summary

- The Cuda Programming Model provides a general approach to organizing Data Parallel programs for heterogeneous, hierarchical platforms
  - Currently, the only production-quality implementation is Cuda for C/C++ on Nvidia's GPUs
  - But Cuda notions of "Scalar Threads", "Warps", "Blocks", and "Grids" can be mapped to other platforms as well!
- A simple "Homogenous SPMD" approach to Cuda programming is useful, especially in early stages of implementation and debugging
  - But achieving high efficiency requires careful consideration of the mapping from computations to processors, data to memories, and data access patterns