



并行与分布式计算

Parallel & Distributed Computing

陈鹏飞
数据科学与计算机学院
2018-05-18
chenpf7@mail.sysu.edu.cn



Lecture 10 — Introduction to GPGPUs, CUDA & OpenCL Programming Model

Pengfei Chen

School of Data and Computer Science

May 16, 2018

chenpf7@mail.sysu.edu.cn



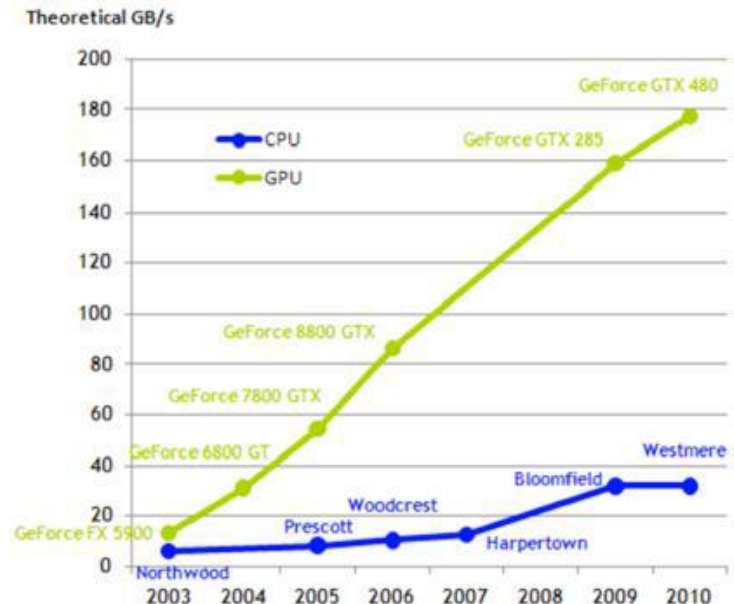
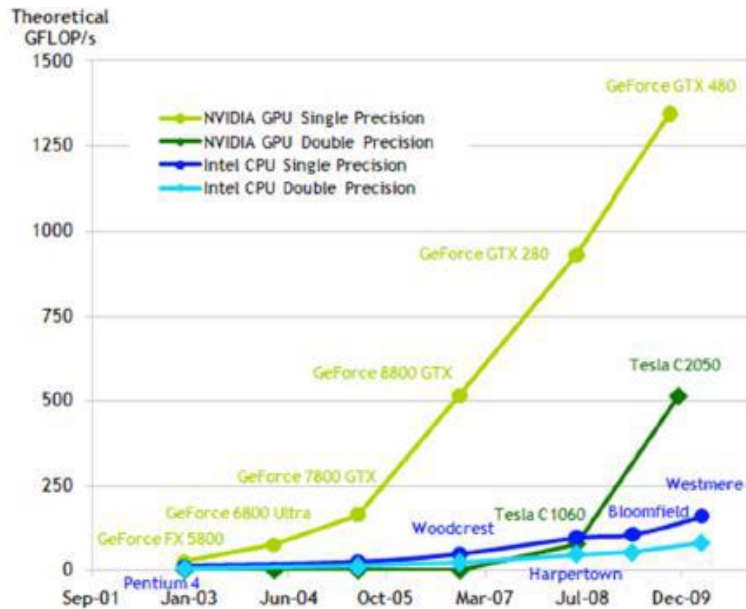
黄仁勋



Outline:

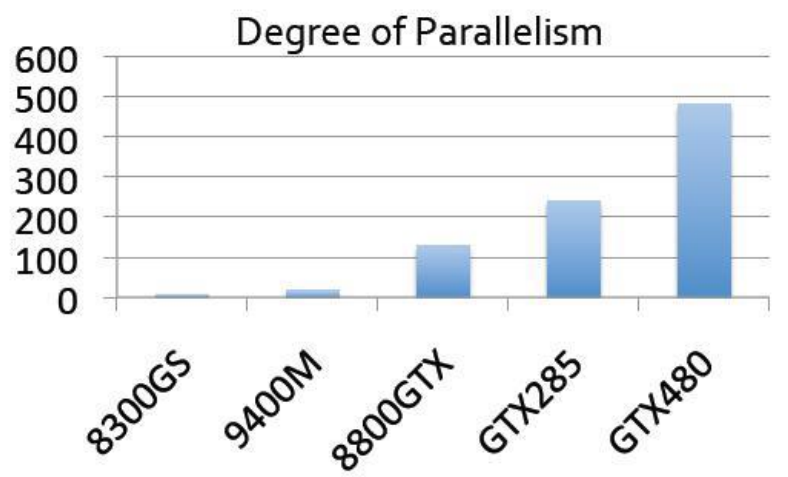
- **Introduction to GPGPUs and CUDA Programming Model**
- **The CUDA Thread Hierarchy**
- **The CUDA Memory Hierarchy**
- **Mapping CUDA to Nvidia GPUs**
- **OpenCL Introduction**

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 **legacy** 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 **Data-Parallel** workloads

Cuda Programming Model Goals



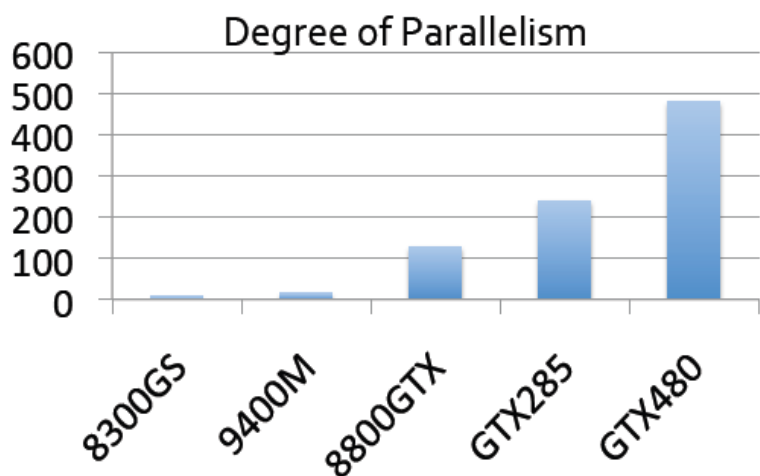
CUDA: Compute Unified Device Architecture

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

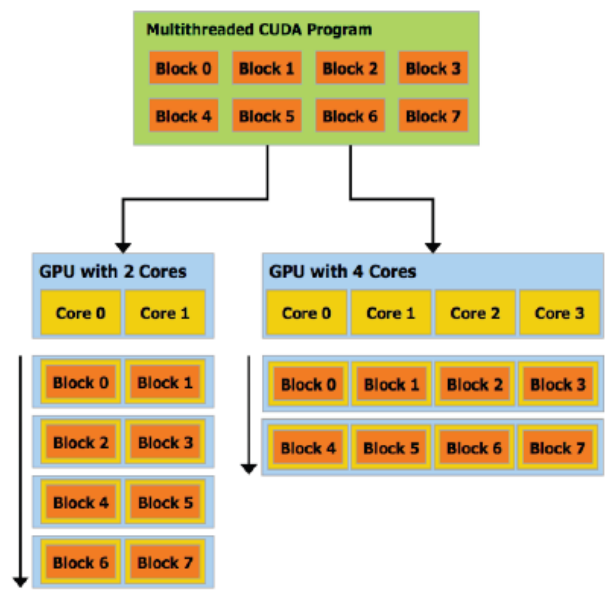


Make SIMD hardware accessible to general-purpose programmers. Otherwise, large fractions of the available execution hardware are wasted!

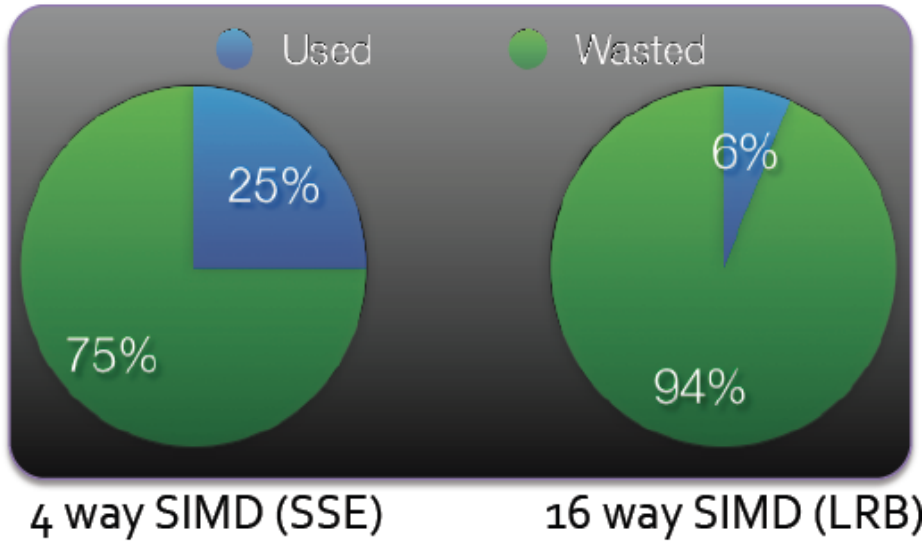
Cuda Goals: Scalability



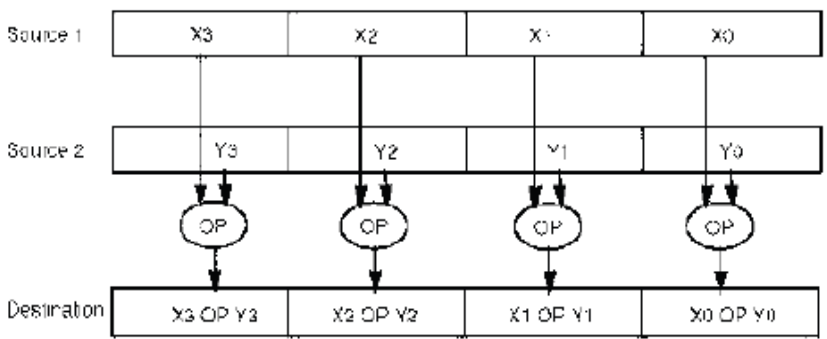
- Cuda expresses many independent blocks of computation that can be run in any order;
- Much of the inherent scalability of the Cuda Programming model stems from batched execution of "Thread Blocks";
- Between GPUs of the same generation, many programs achieve linear speedup on GPUs with more "Cores";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



Cuda Goals: SIMD Programming



- 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 Transforms
 - CULapack (3rd party proprietary) linear solvers, eigensolvers, ...
- OS-Portable: Linux, Windows, Mac OS
- A lot of momentum in Industrial adoption of Cuda!

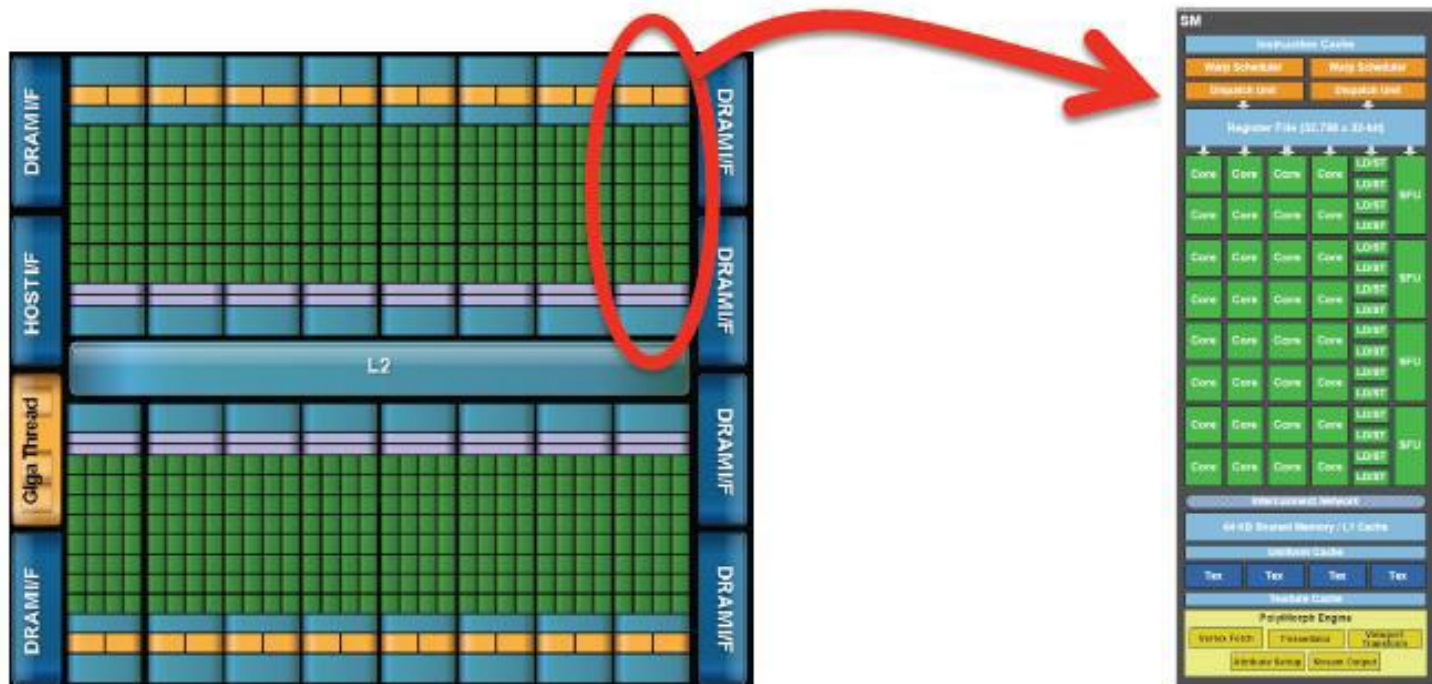
http://developer.nvidia.com/object/cuda_3_1_downloads.html



Nvidia Cuda GPU Architecture

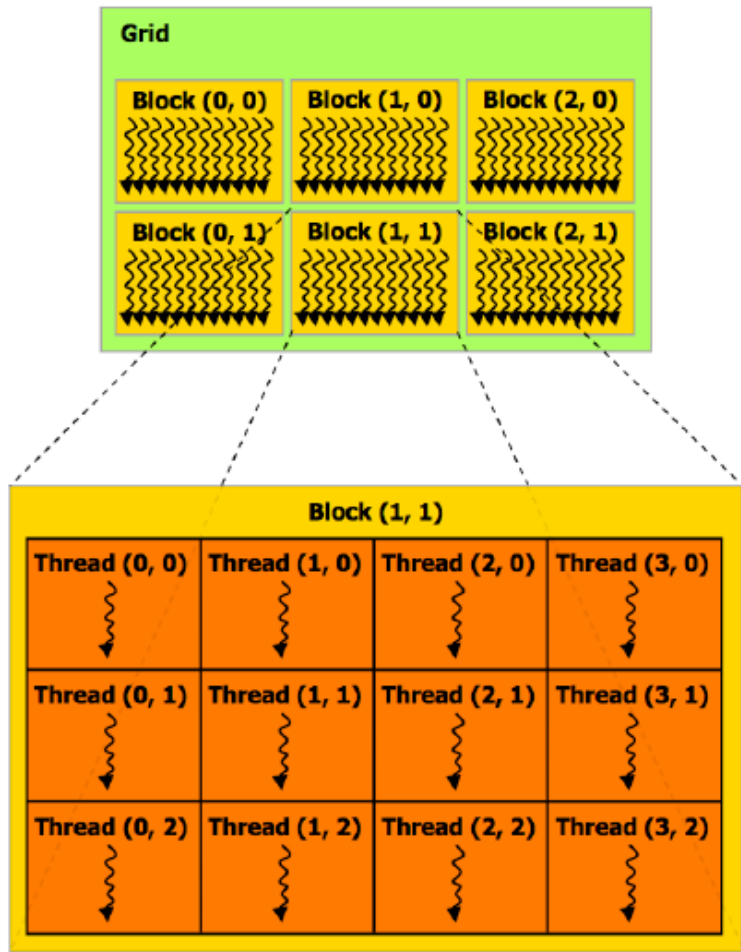
Nvidia Cuda GPU Architecture

- ❑ 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 **Grid** s that execute in-order. Fermi GPUs execute multiple Streams in parallel
- A **Grid** is a set of up to 2^{32} **read 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 mul/threading 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** (合并) 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);
```



CUDA Memory Hierarchy

Cuda Memory Hierarchy

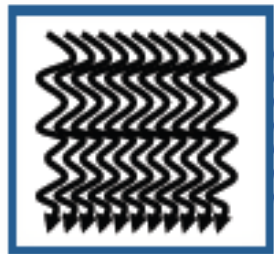
Thread



Per-thread
Local Memory

- 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

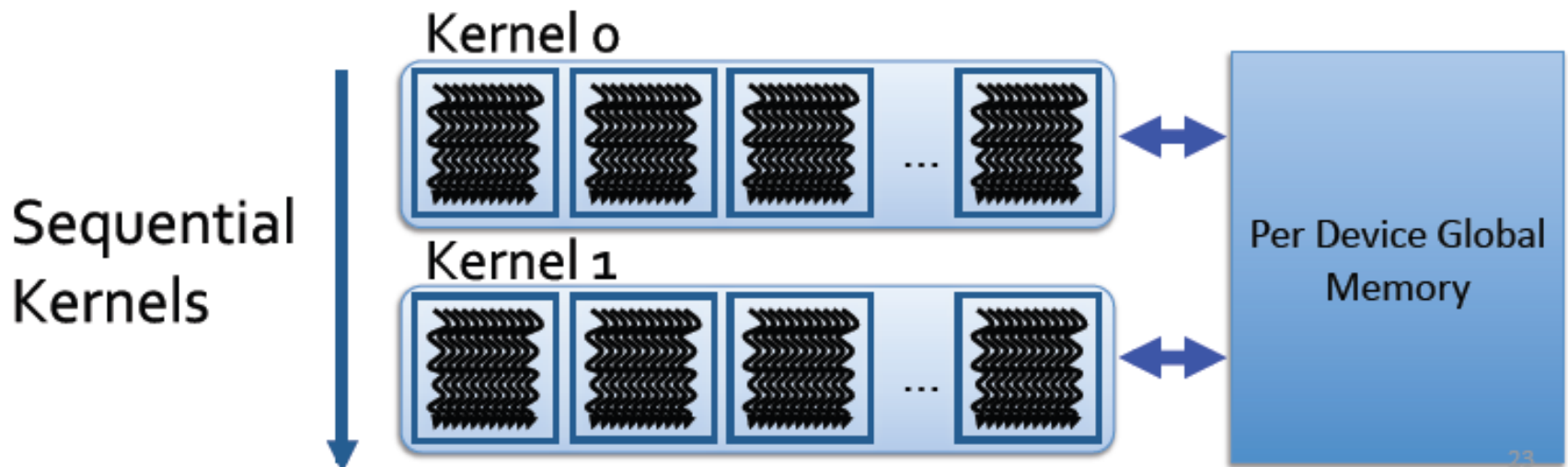
Block



Per-block
Shared
Memory

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: $O(100)\times$ latency, $O(1/50)\times$ (aggregate) bandwidth
- On Fermi, Global Memory is cached in a 768KB shared L2



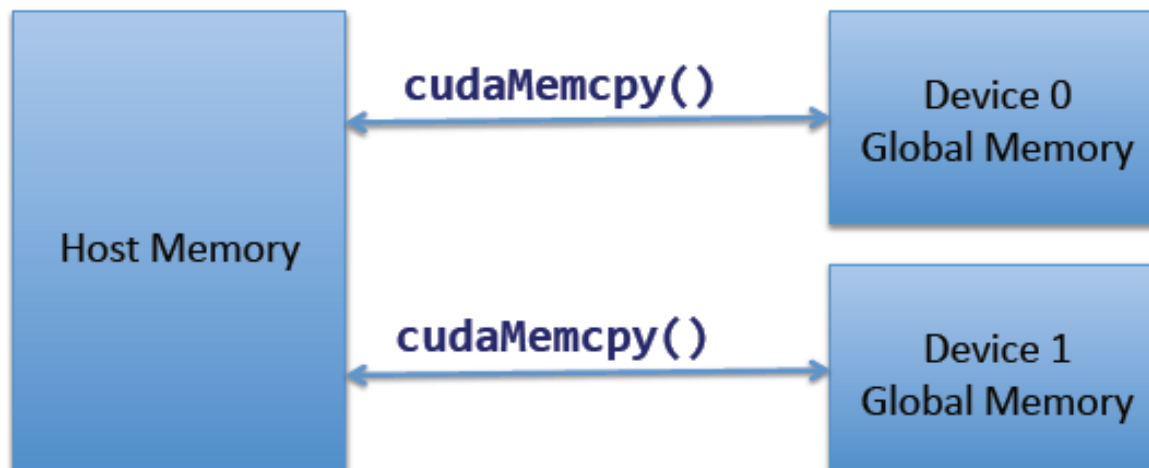


Cuda Memory Hierarchy

- 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

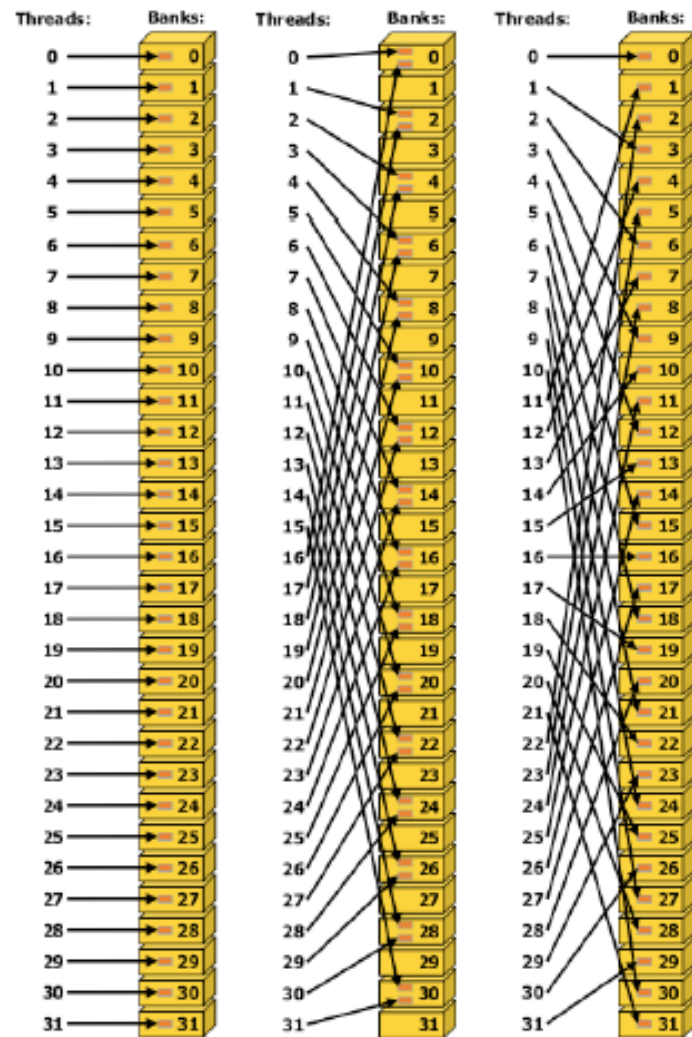
- Each SM has 64 KB of private memory, divided 16KB/48KB (or 48KB/16KB) into so hardware-managed scratchpad and hardware-managed, non-coherent cache
 - Pre-Fermi, the SM memory is only 16 KB, and is usable only as so hardware-managed scratchpad
- Unless data will be shared between Threads in a block, 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 $(k+1) \bmod 32$
- 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

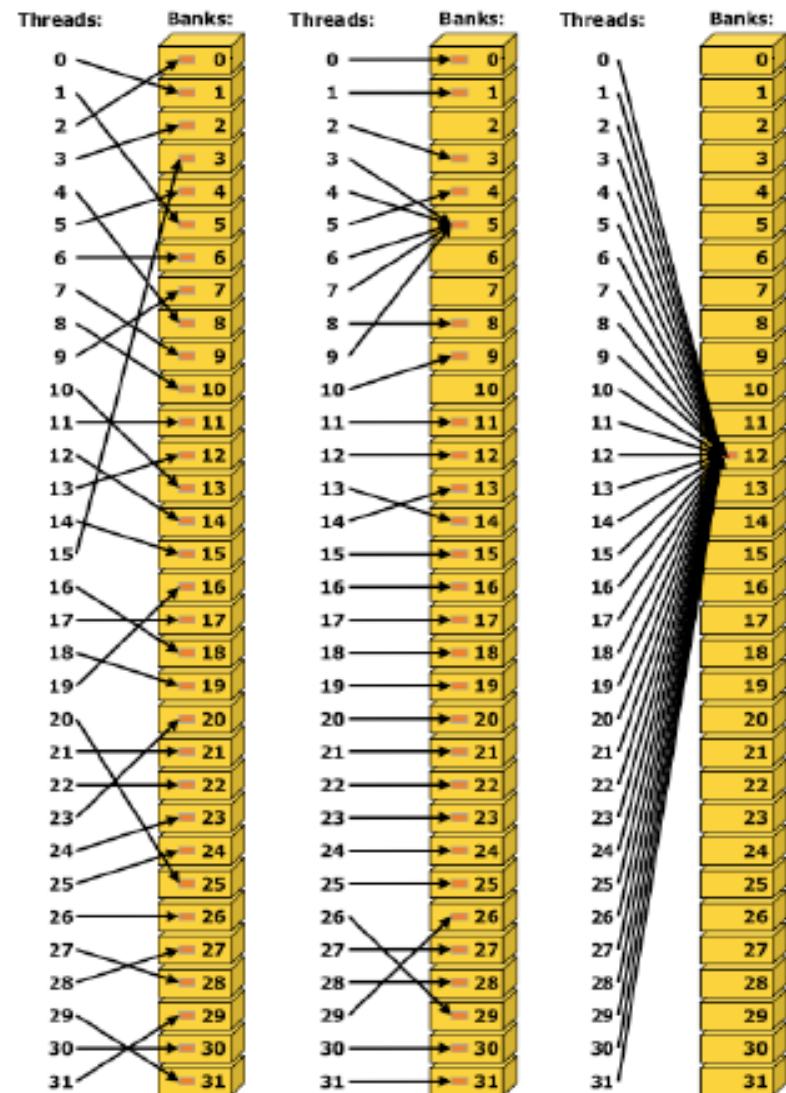
Shared Memory Bank Conflicts



➤ 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 Code 3.1
- Permutations 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
 - ❑ With 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

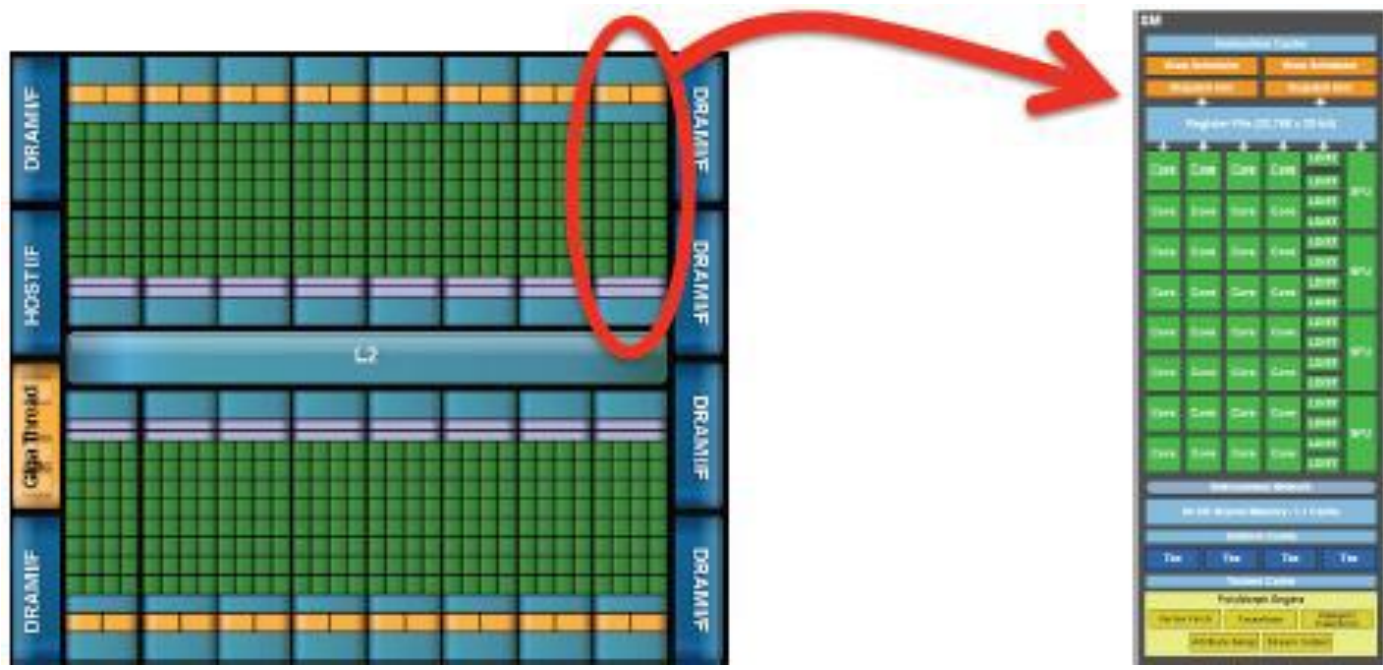
- By default, you cannot assume memory accesses are occur in the same order specified by the program
 - ❑ Although a thread's **own** accesses appear to that thread to occur in program order
- To enforce ordering, use **memory fence** instructions
 - ❑ `__threadfence_block()`: make all previous memory 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.



Mapping Cuda to Nvidia GPUs

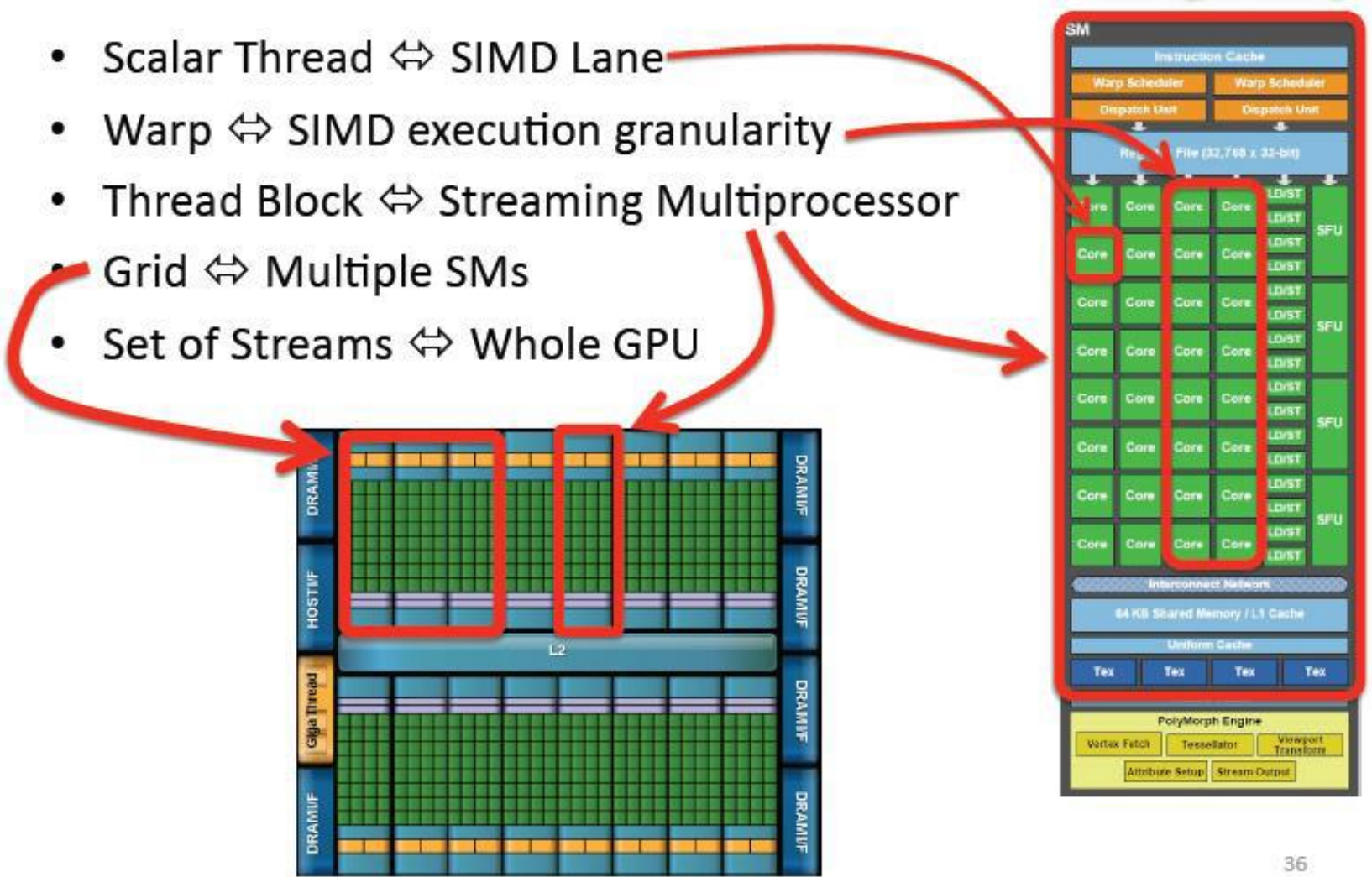
Mapping CUDA to Nvidia GPUs

- 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



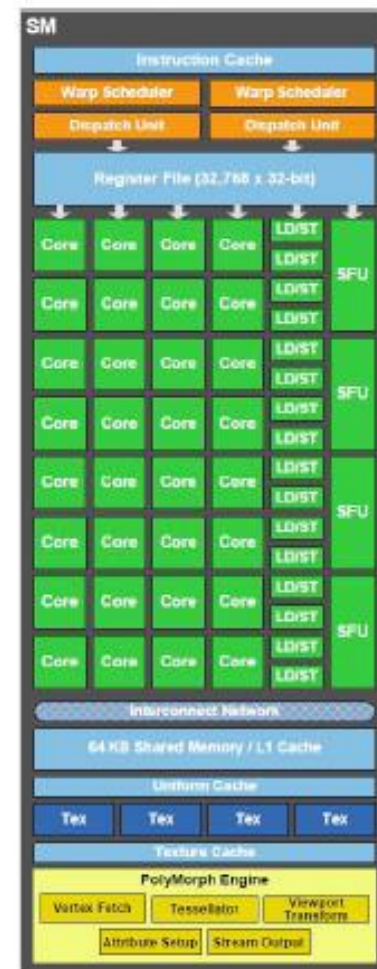
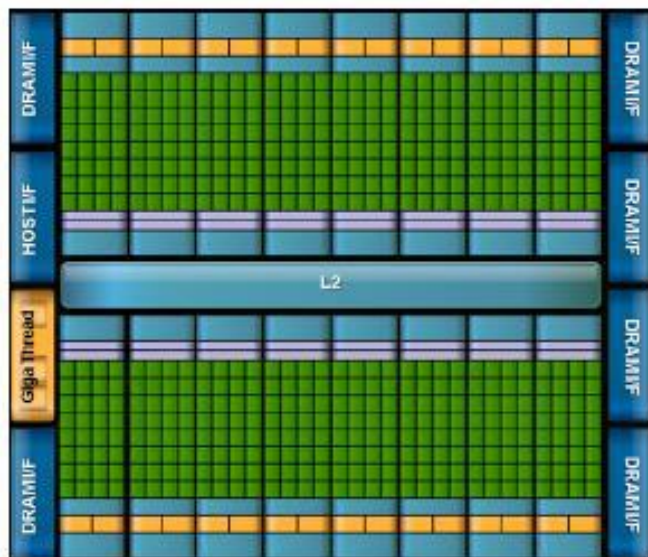
Mapping CUDA to Nvidia GPUs

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

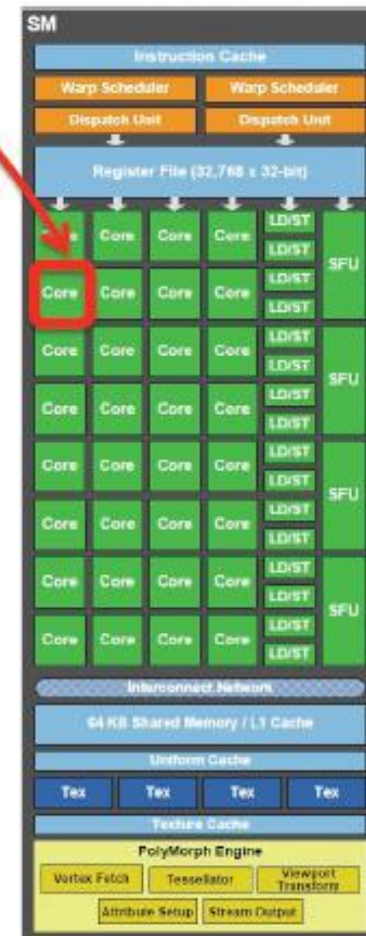


Mapping CUDA to Nvidia GPUs

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

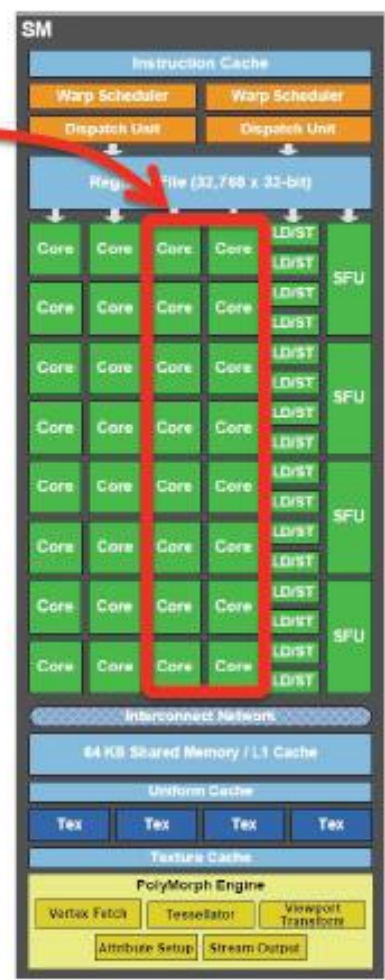
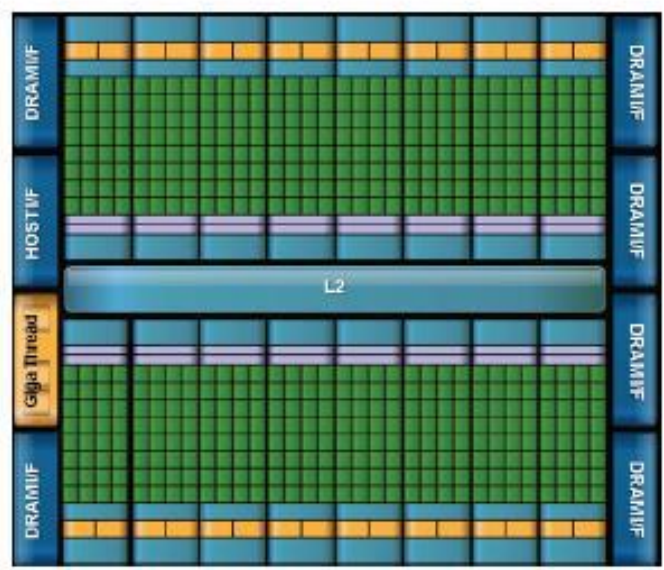


- **Scalar Thread** ⇔ **SIMD Lane**
- Warp ⇔ Logical SIMD width
- Thread Block ⇔ Streaming Multiprocessor
- Grid ⇔ Multiple SMs
- Set of Streams ⇔ Whole GPU



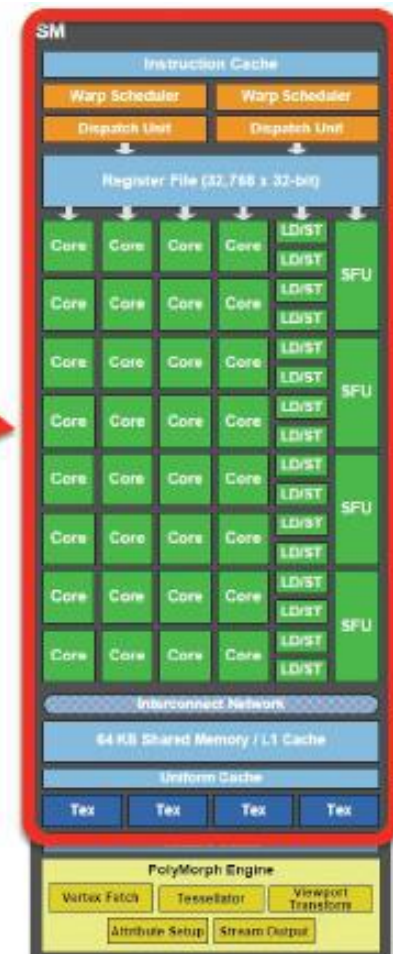
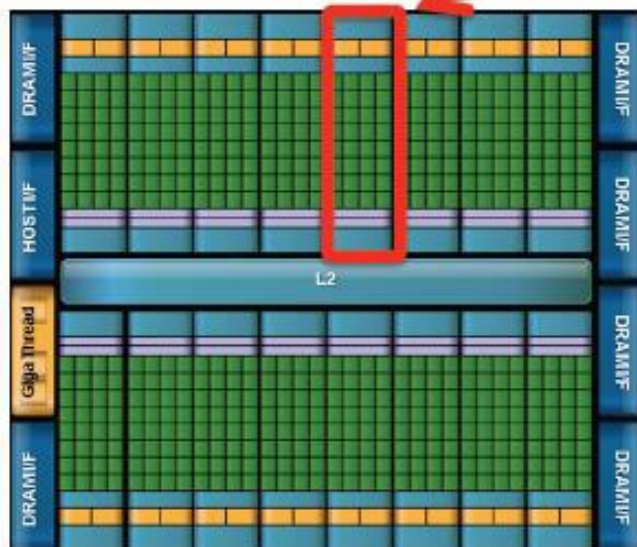
Mapping CUDA to Nvidia GPUs

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



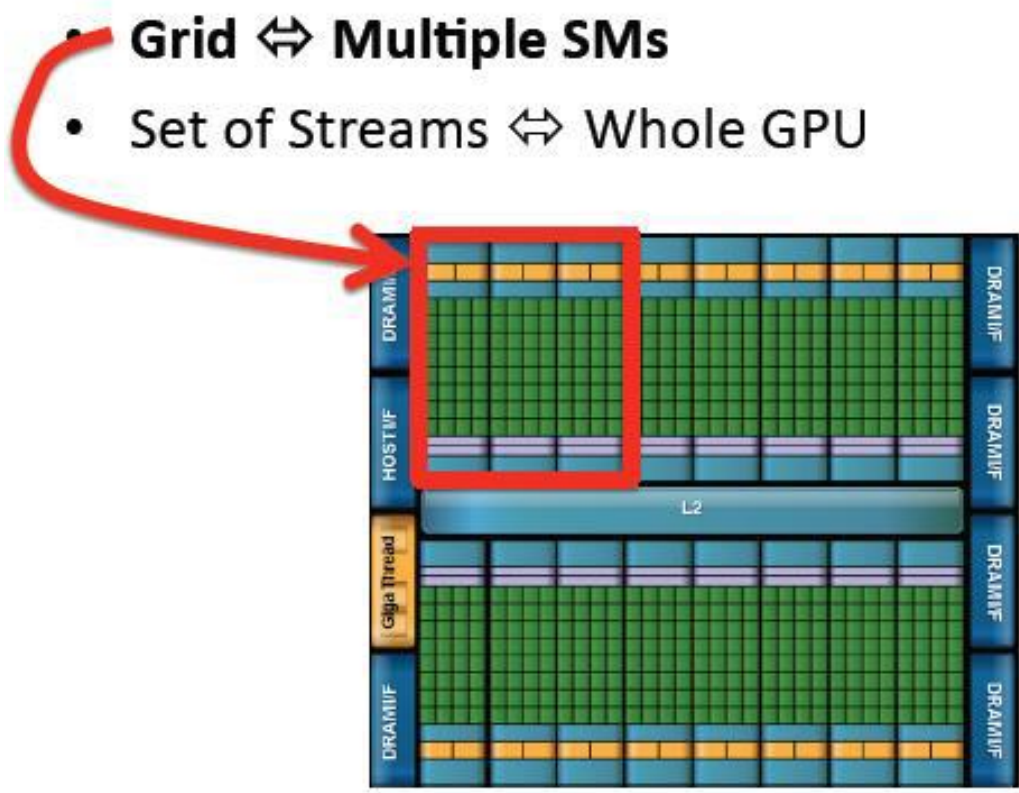
Mapping CUDA to Nvidia GPUs

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



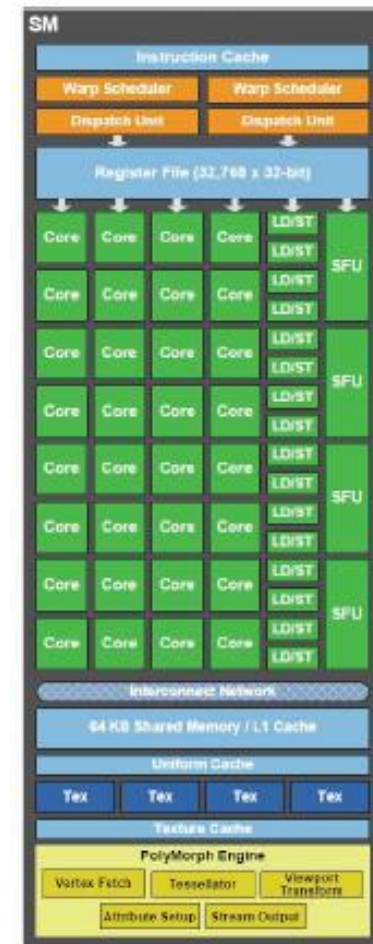
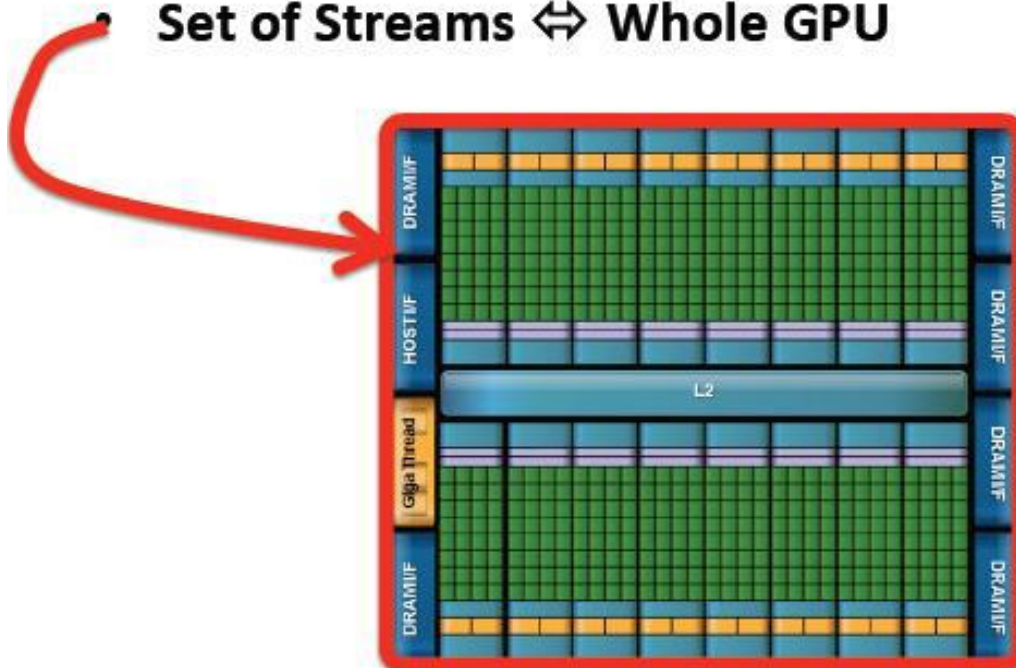
Mapping CUDA to Nvidia GPUs

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



Mapping CUDA to Nvidia GPUs

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



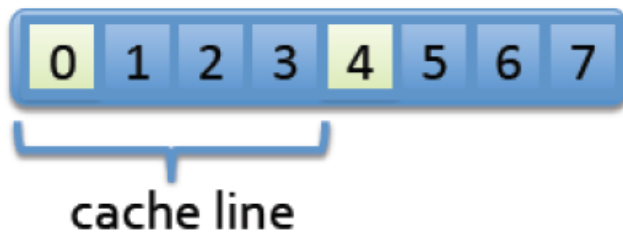


Mapping CUDA to Nvidia GPUs

- ❑ 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 \longleftrightarrow fewer threads
 - More shared memory per block \longleftrightarrow fewer blocks

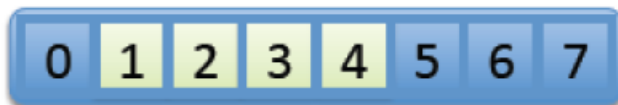
Mapping CUDA to Nvidia GPUs

- ❑ A many core processor \equiv 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:
 $\frac{1}{4}$ effective bandwidth

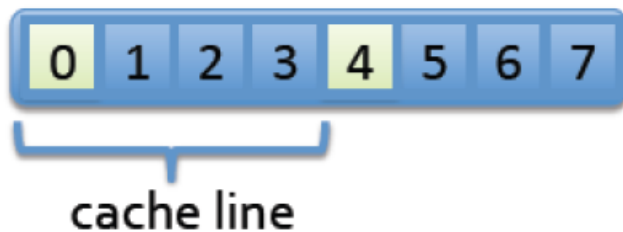
- **Unaligned accesses waste bandwidth!**



4 words used, 8 words loaded:
 $\frac{1}{2}$ effective bandwidth

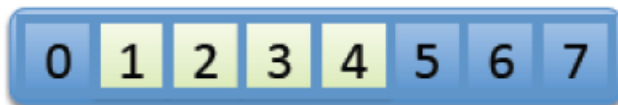
Mapping CUDA to Nvidia GPUs

- ❑ A many core processor \equiv 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:
 $\frac{1}{4}$ effective bandwidth

- **Unaligned accesses waste bandwidth!**



4 words used, 8 words loaded:
 $\frac{1}{2}$ 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



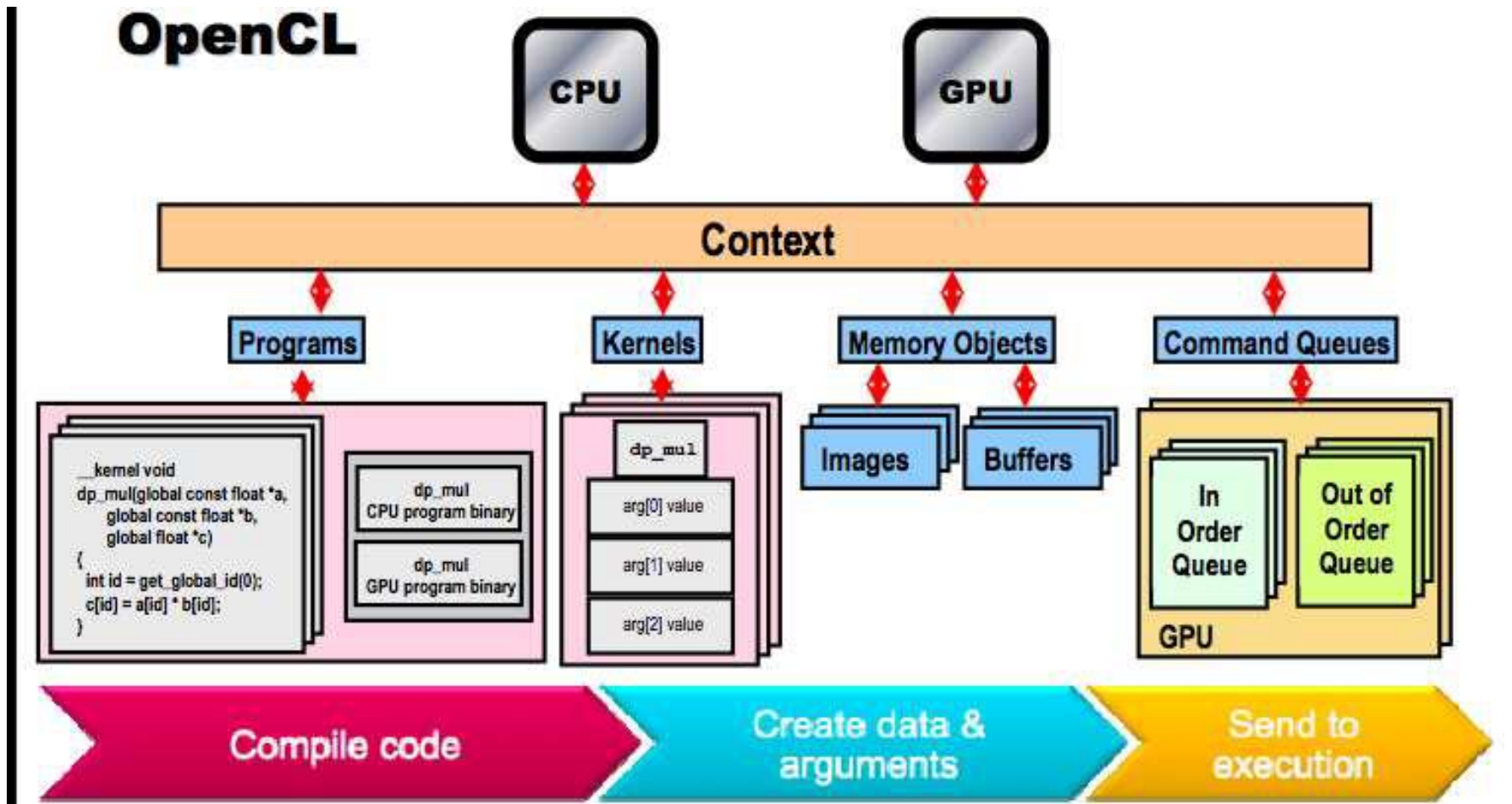
Open CL



OpenCL Review

- **OpenCL allows parallel computing on heterogeneous Devices**
 - ❑ CPUs, GPUs, other processors (Cell, DSPs, FPGAs, etc.)
 - ❑ Provides portable accelerated code
- **Basic concepts in OpenCL**
 - ❑ Platform model
 - ❑ Execution model
 - ❑ Memory model
 - ❑ Programming model

Big Picture





Parallel Software – SPMD

- GPU programs (*kernels*) written using the Single Program Multiple Data (SPMD) programming model
 - ❑ SPMD executes multiple instances of the same program independently, where each program works on a different portion of the data
- For data-parallel scientific and engineering applications, combining SPMD with loop strip mining is a very common parallel programming technique
 - ❑ Message Passing Interface (MPI) is used to run SPMD on a distributed cluster
 - ❑ POSIX threads (pthreads) are used to run SPMD on a shared memory system
 - ❑ Kernels run SPMD within a GPU

Parallel Software – SPMD

◆ Consider the following vector addition example

```
for( i = 0:11 ) {  
    C[ i ] = A[ i ] + B[ i ]  
}
```

Serial program:

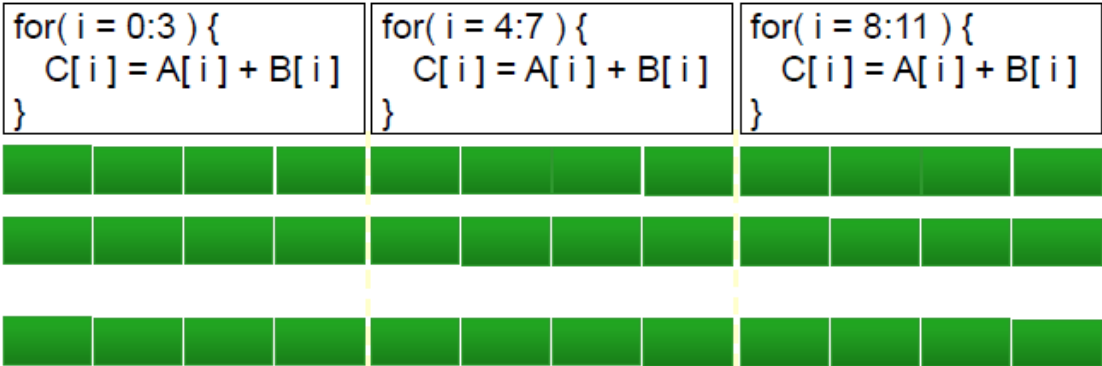
one program completes
the entire task



◆ Combining SPMD with loop strip mining allows multiple copies of the same program execute on different data in parallel

SPMD program:

multiple copies of the
same program run on
different chunks of the
data






Parallel Software – SPMD

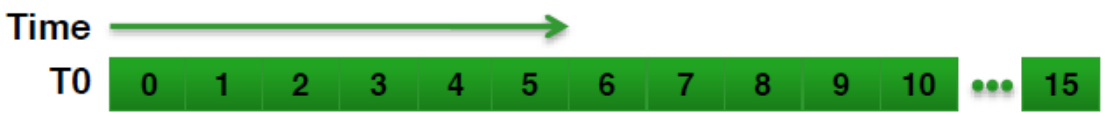
- **In the vector addition example, each chunk of data could be executed as an independent thread**
- **On modern CPUs, the overhead of creating threads is so high that the chunks need to be large**
 - **In practice, usually a few threads (about as many as the number of CPU cores) and each is given a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

Parallel Software – SPMD

 = loop iteration

Single-threaded (CPU)

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



Multi-threaded (CPU)

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

T0	0	1	2	3
T1	4	5	6	7
T2	8	9	10	11
T3	12	13	14	15

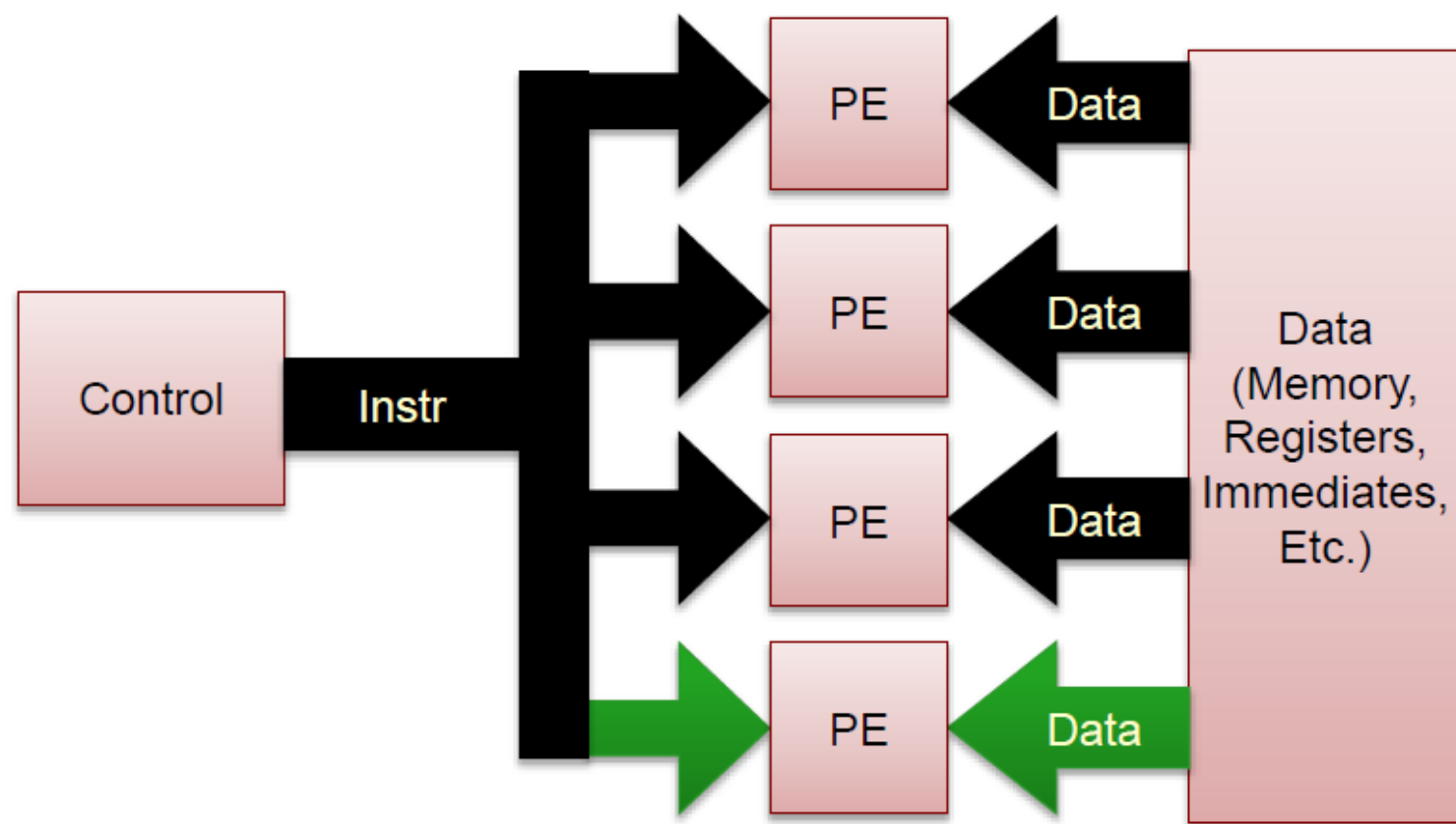
Massively Multi-threaded (GPU)

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

T0	0
T1	1
T2	2
T3	3
...	...
T15	15

Parallel Software – SPMD

➤ A SIMD hardware unit



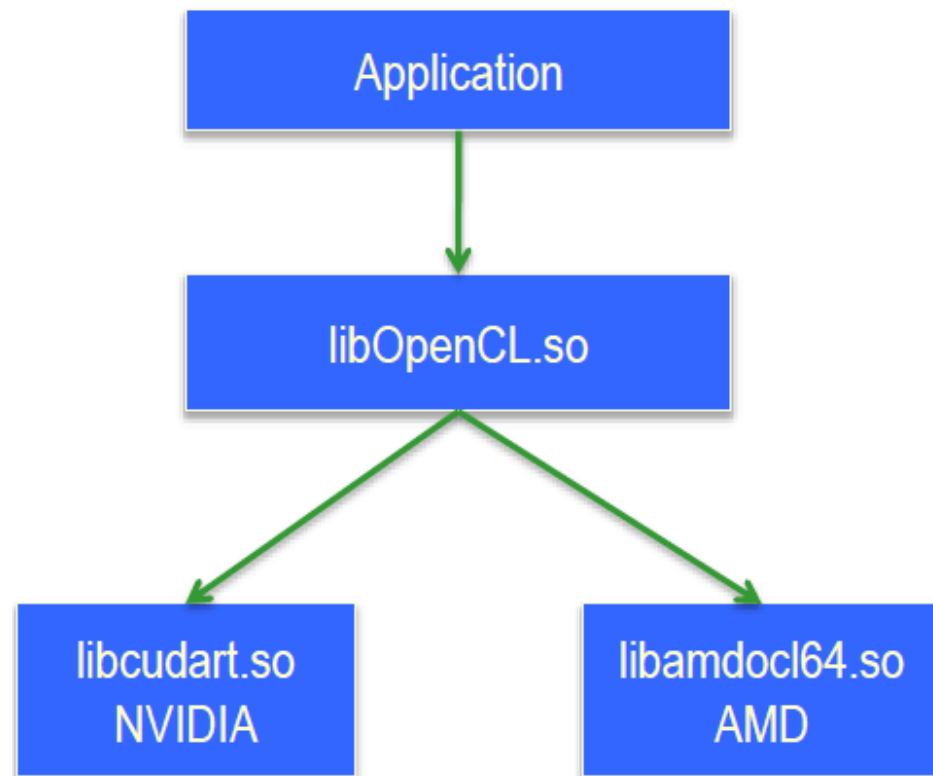


Platform Model

- Each OpenCL implementation (i.e. an OpenCL library from AMD, NVIDIA, etc.) defines *platforms* which enable the host system to interact with OpenCL-capable devices
 - ❑ Currently each vendor supplies only a single platform per implementation
- OpenCL uses an “Installable Client Driver” model
 - ❑ The goal is to allow platforms from different vendors to coexist
 - ❑ Current systems’ device driver model will not allow different vendors’ GPUs to run at the same time

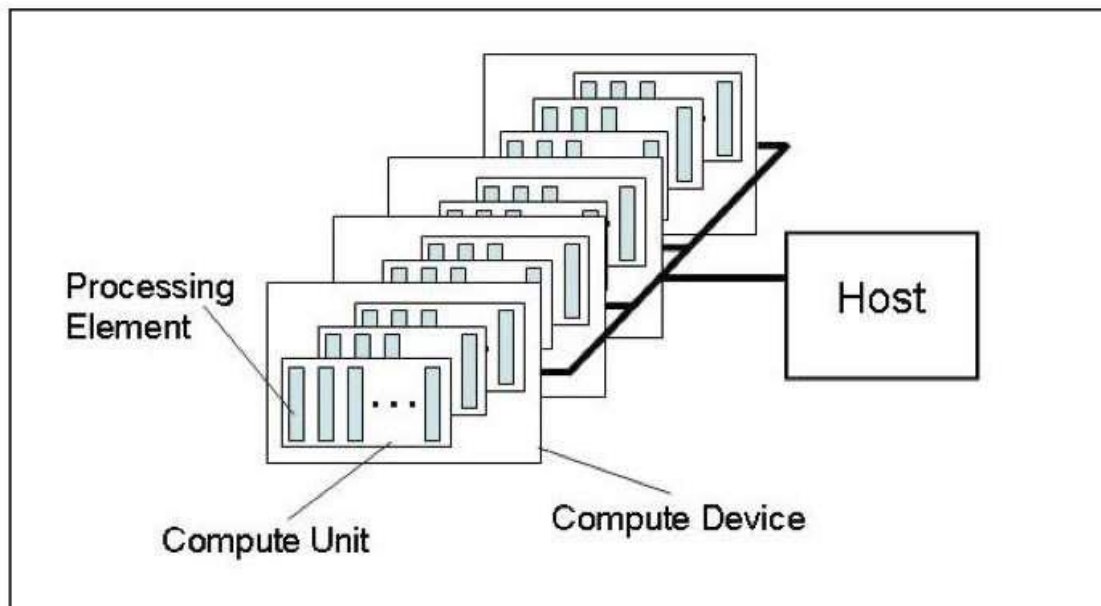
Installable Client Driver

- ICD allows multiple implementations to co-exist
- Code only links to libOpenCL.so
- Application selects implementation at runtime
 - ❑ clGetPlatformIDs() and clGetPlatformInfo() examine the list of available implementations and select a suitable one
- Current GPU driver model does not easily allow devices from different vendors in same platform



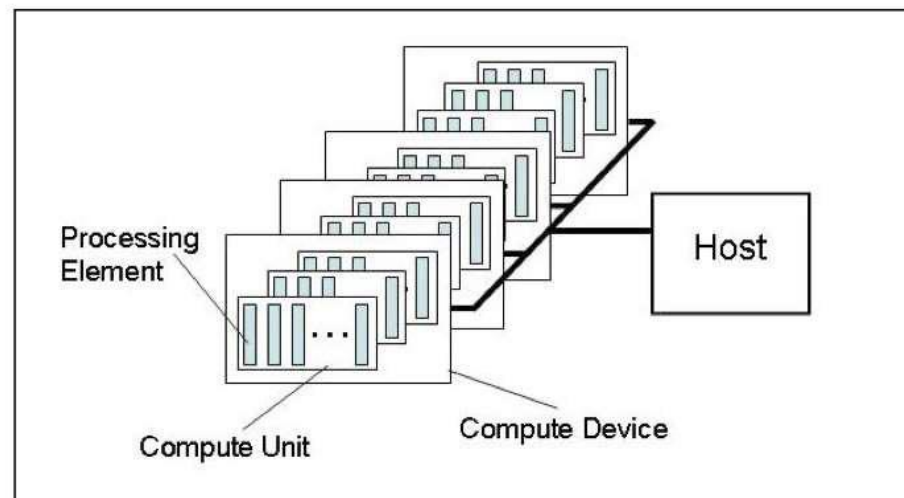
Platform Model

- The model consists of a host connected to one or more OpenCL devices
- A device is divided into one or more compute units
- Compute units are divided into one or more processing elements
 - ❑ Each processing element maintains its own program counter



Host/Devices

- The host is whatever the OpenCL library runs on
 - ❑ x86 CPUs for both NVIDIA and AMD
- Devices are processors that the library can talk to
 - ❑ CPUs, GPUs, and generic accelerators
- For AMD
 - ❑ All CPUs are combined into a single device (each core is a compute unit and processing element)
 - ❑ Each GPU is a separate device



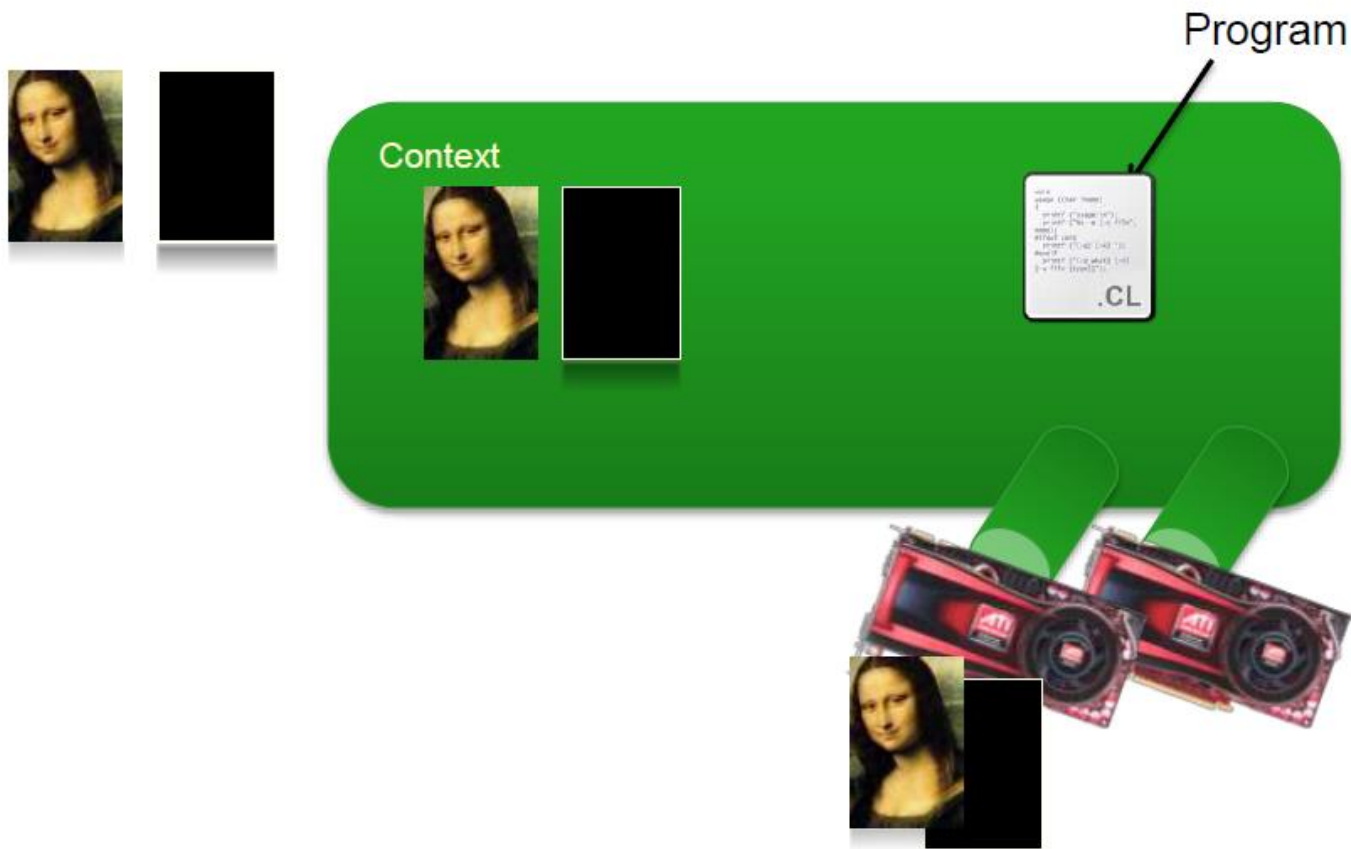


Programs

- A program object is basically a collection of OpenCL kernels
 - ❑ Can be source code (text) or precompiled binary
 - ❑ Can also contain constant data and auxiliary functions
- Creating a program object requires either reading in a string (source code) or a precompiled binary
- To compile the program
 - ❑ Specify which devices are targeted
 - Program is compiled for each device
 - ❑ Pass in compiler flags (optional)
 - ❑ Check for compilation errors (optional, output to screen)

Programs

- A program object is created and compiled by providing source code or a binary file and selecting which devices to target





Creating Programs

```
cl_program  clCreateProgramWithSource (cl_context context,  
                                       cl_uint count,  
                                       const char **strings,  
                                       const size_t *lengths,  
                                       cl_int *errcode_ret)
```

- This function creates a program object from strings of source code
 - ❑ *count* specifies the number of strings
 - ❑ The user must create a function to read in the source code to a string
- If the strings are not NULL-terminated, the *lengths* fields are used to specify the string lengths



Compiling Programs

```
cl_int      clBuildProgram (cl_program program,  
                           cl_uint num_devices,  
                           const cl_device_id *device_list,  
                           const char *options,  
                           void (CL_CALLBACK *pfn_notify)(cl_program program,  
                                                           void *user_data),  
                           void *user_data)
```

- This function compiles and links an executable from the program object for each device in the context
 - ❑ If *device_list* is supplied, then only those devices are targeted
- Optional preprocessor, optimization, and other options can be supplied by the *options* argument

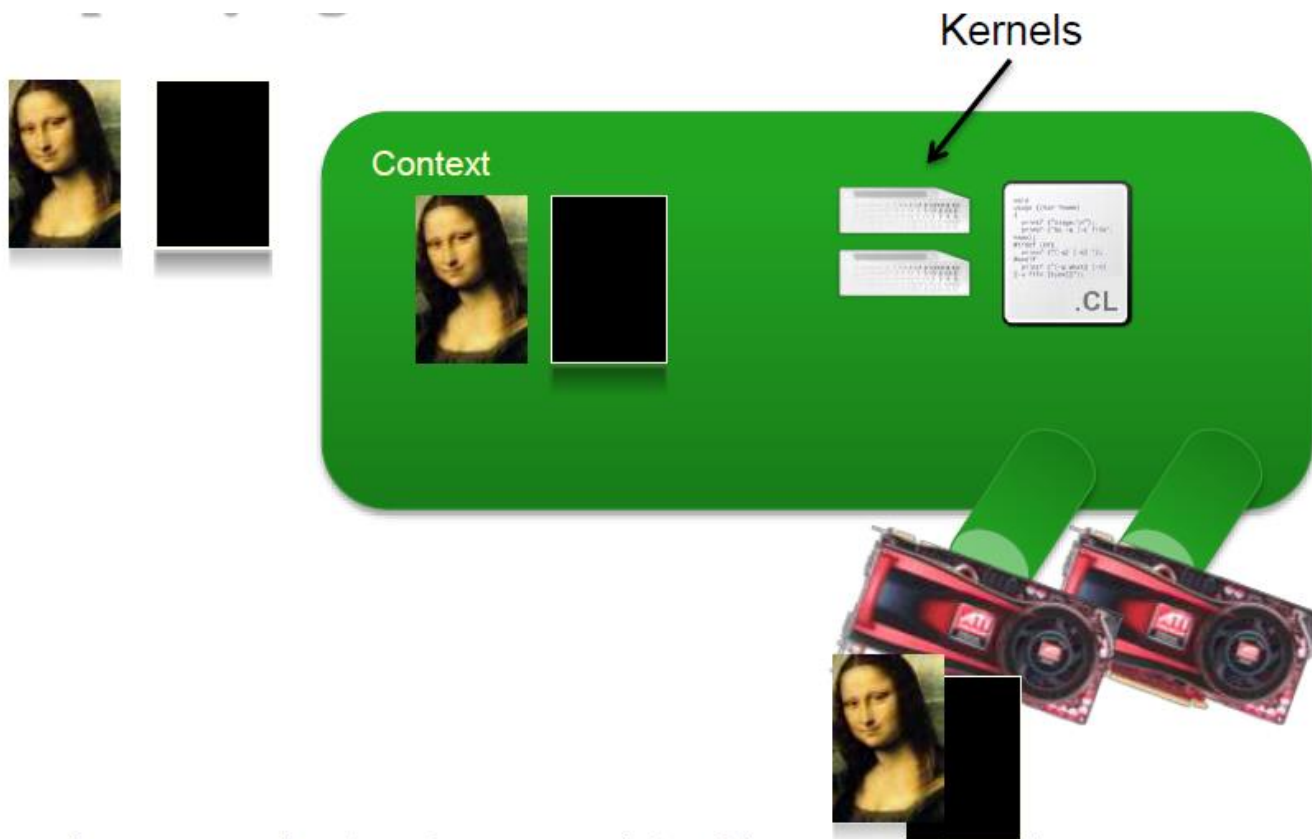


Kernels

- **A kernel is a function declared in a program that is executed on an OpenCL device**
 - ❑ **A kernel object is a kernel function along with its associated arguments**
- **A kernel object is created from a compiled program**
- **Must explicitly associate arguments (memory objects, primitives, etc) with the kernel object**

Kernels

- Kernel objects are created from a program object by specifying the name of the kernel function





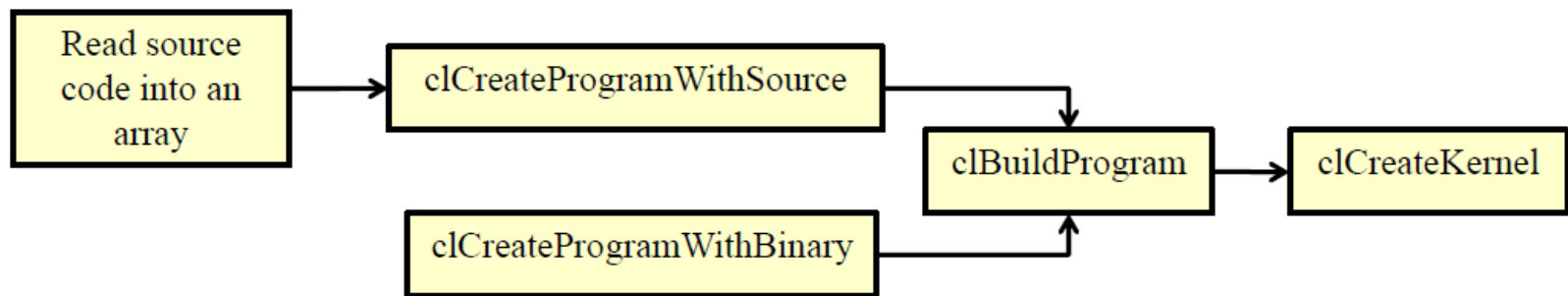
Kernels

```
cl_kernel      clCreateKernel (cl_program program,  
                               const char *kernel_name,  
                               cl_int *errcode_ret)
```

- Creates a kernel from the given program
 - ❑ The kernel that is created is specified by a string that matches the name of the function within the program

Runtime Compilation

- There is a high overhead for compiling programs and creating kernels
 - Each operation only has to be performed once (at the beginning of the program)
 - The kernel objects can be reused any number of times by setting different arguments





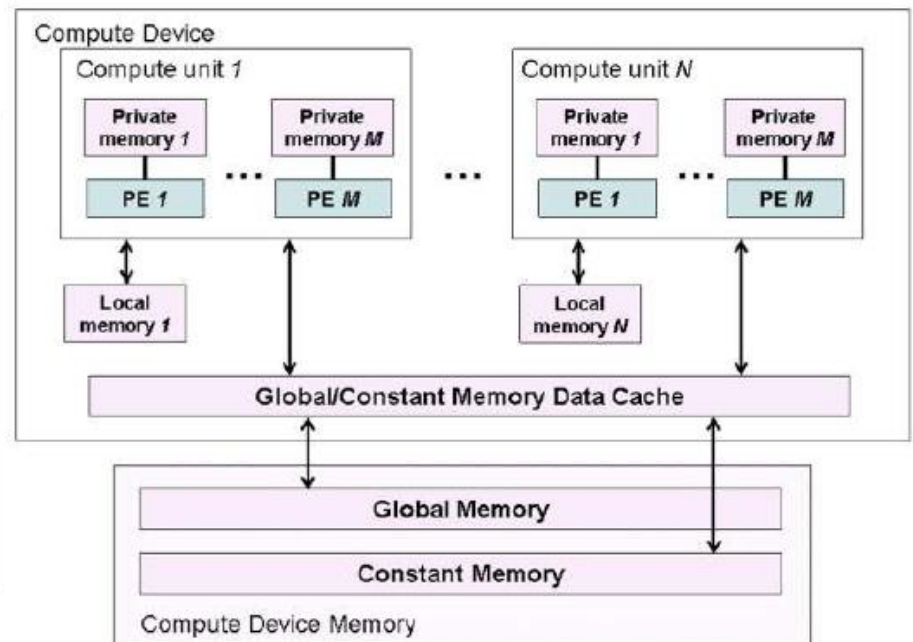
Thread Structure

- **Massively parallel programs are usually written so that each thread computes one part of a problem**
 - ❑ **For vector addition, we will add corresponding elements from two arrays, so each thread will perform one addition**
 - ❑ **If we think about the thread structure visually, the threads will usually be arranged in the same shape as the data**

Memory Model

- The OpenCL memory model defines the various types of memories (closely related to GPU memory hierarchy)

Memory	Description
Global	Accessible by all work-items
Constant	Read-only, global
Local	Local to a work-group
Private	Private to a work-item





Memory Model

- **Memory management is explicit**
 - ❑ **Must move data from host memory to device global memory, from global memory to local memory, and back**
- **Work-groups are assigned to execute on compute units**
 - ❑ **No guaranteed communication/coherency between different work-groups (no software mechanism in the OpenCL specification)**



Address Space Identifiers

- **__global** – memory allocated from global address space
- **__constant** – a special type of read-only memory
- **__local** – memory shared by a work-group
- **__private** – private per work-item memory
- **__read_only/__write_only** – used for images
- Kernel arguments that are memory objects must be global, local, or constant

Example Kernel

◆ Simple vector addition kernel:

`__kernel`

```
void vecadd(__global int* A,  
            __global int* B,  
            __global int* C) {  
    int tid = get_global_id(0);  
    C[tid] = A[tid] + B[tid];  
}
```



Programming Model

➤ **Data parallel**

- ❑ **One-to-one mapping between work-items and elements in a memory object**
- ❑ **Work-groups can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the workgroups)**

➤ **Task parallel**

- ❑ **Kernel is executed independent of an index space**
- ❑ **Other ways to express parallelism: enqueueing multiple tasks, using device-specific vector types, etc.**

➤ **Synchronization**

- ❑ **Possible between items in a work-group**
- ❑ **Possible between commands in a context command queue**



Running the Example Code

- For example, a simple vector addition OpenCL program
- Before running, the following should appear in your `.bashrc` file:
 - ❑ `export LD_LIBRARY_PATH=/opt/AMDAPP/lib/x86_64`
- To compile:
 - ❑ Make sure that `vecadd.c` and `vecadd.cl` are in the current working directory
 - ❑ `gcc -o vecadd vecadd.c -I/opt/AMDAPP/include -L/opt/AMDAPP/lib/x86_64 -lOpenCL`



Summary

- OpenCL provides an interface for the interaction of hosts with accelerator devices
- A context is created that contains all of the information and data required to execute an OpenCL program
 - ❑ Memory objects are created that can be moved on and off devices
 - ❑ Command queues allow the host to request operations to be performed by the device
 - ❑ Programs and kernels contain the code that devices need to execute