ECE 420
Lecture 10
November 4 2019

Exploiting Parallelism for Acceleration

- For signal processing applications, it's all about the math
  - How many multiplies, adds, etc. are performed
  - Sometimes performance can be memory-limited, but often SP algorithms are compute-bound
- Performance becomes correlated with compute rate of your system (FLOPs)
- Diminishing returns in absolute time for performing these operations (e.g. CPU frequencies)
- Computer architecture trends favor exploiting parallelism

Architectural Features for Parallelism

- Superscalar – multiple functional units (multipliers, adders)
- Pipelined – don't wait until operation is complete before issuing the next one
- Out of order execution – allow operations to proceed out of order if dependencies allow
- SIMD (vector instructions) – operate a single operation on a group of data
- Multithreading – separate concurrently executing code paths
- Multiprocessing – multiple processors, on same host or a cluster of hosts

Exploiting Parallel Features

- Earlier efforts focused on compiler and hardware runtime to detect and exploit parallelism
  - Instruction scheduling, out of order execution
- Limitations to how much can be gleaned through analysis

$$x = a[n] * b$$
$$y = a[n+1] * x$$
$$z = c[m] + d$$
$$a[m] = 0$$

What are ‘legal’ reorderings of this code (preserving original semantics)?

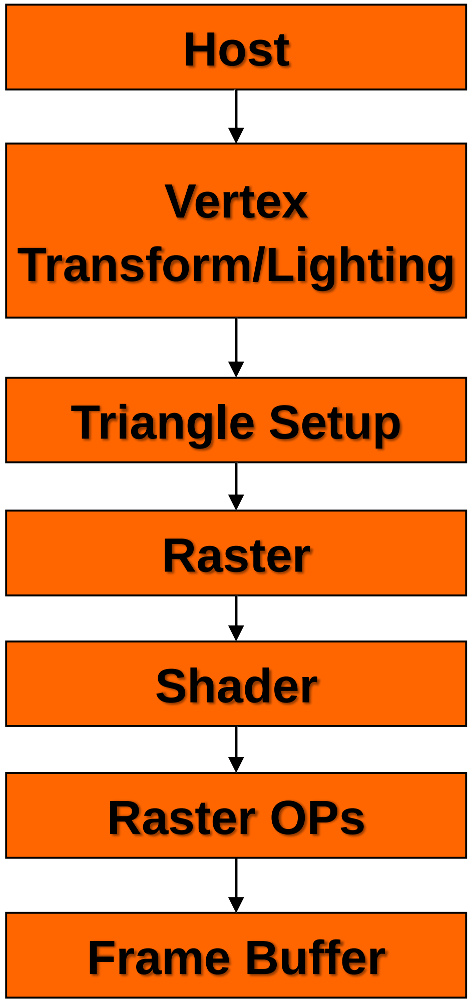
Exploiting Parallel Features

- Progressive movement to shifting responsibility to developers
  - Add ‘hints’ to code
  - Explicit threading/parallel constructs
- New tools, languages, and language enhancements improve code correctness and ease of development
  - Moves some responsibility back to the language, compiler and runtime
- A runtime/architecture will provide a particular execution model – it is up to you to determine how (or if) your algorithms map to it

Massively Parallel Processing

- In desktop land, processing improvements were largely incremental
    - e.g. two processor system, 4 cores, 4x32 bit vectors
- What can you do if you need to tackle a REALLY large problem?
    - Acceleration of 100x or more
    - Massively parallel system (likely a cluster)
- GPU technology has evolved into a massively parallel general purpose computing platform, right on your computer!
    - Disclaimer: it is not your typical architecture!

Historical GPU architecture



GP-GPU

- GPUs deliver a lot of computational horsepower, have been increasing rapidly over the years
- As graphics pipelines evolved, the vertex and shader stages became programmable units
  - Each contained their own instruction set
  - Shader programs had lots of options, including different shading and lighting modules, texture mapping, and reflections

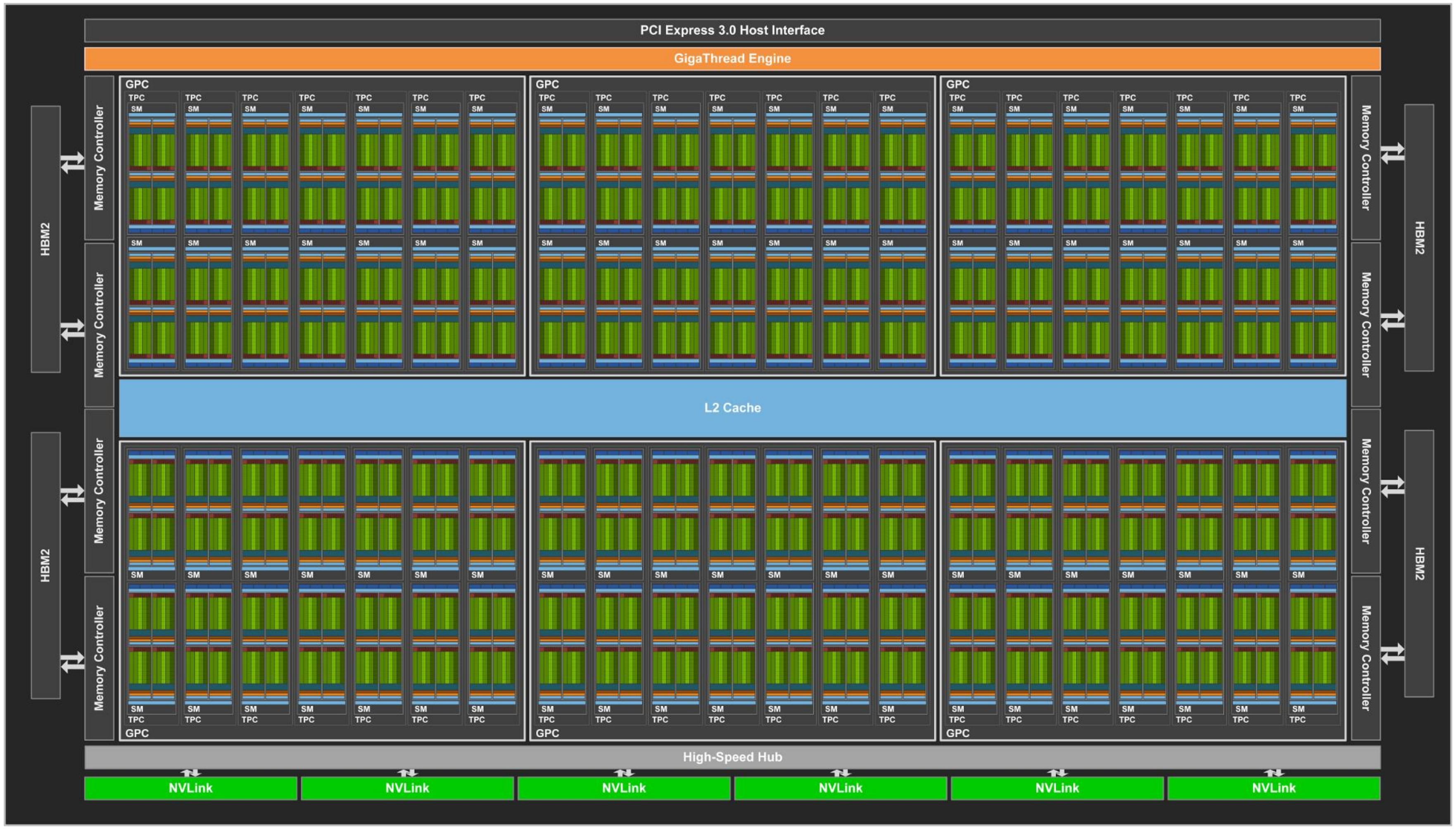
GP-GPU

- People noticed they could ‘hijack’ shader programs to do non-graphics things
  - Data input via texture buffers or other memory sources
  - Frame buffer contained output pixels
- An exciting POC but significant limitations
  - GPU wasn’t really intended to be used this way, awkward to set up
  - Instruction set not as robust as CPU operations
  - Limitations in communication among shaders
  - Datatype constraints
  - Limited tool support

CUDA Architecture

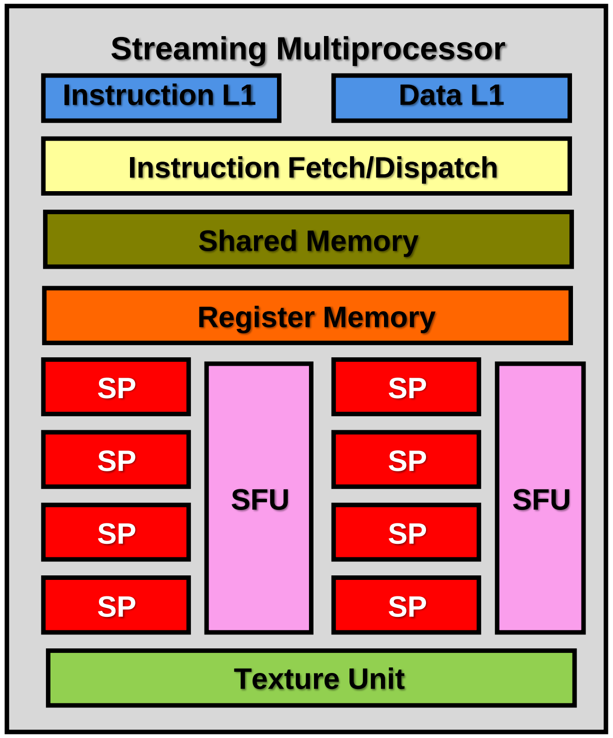
- “Compute Unified Device Architecture”
  - No more discrete vertex processors and shaders
  - Massive array of processing elements
- General purpose programming model
  - SPMD (single program, multiple data) paradigm
  - Facilitate execution of more arbitrary computational tasks with a variety of datatypes
  - Expanded instruction set for computation
  - Official tools for developing, debugging
- Applications
  - Image processing, physical modeling, machine learning, matrix algebra, convolution, correlation, sorting

CUDA Architecture



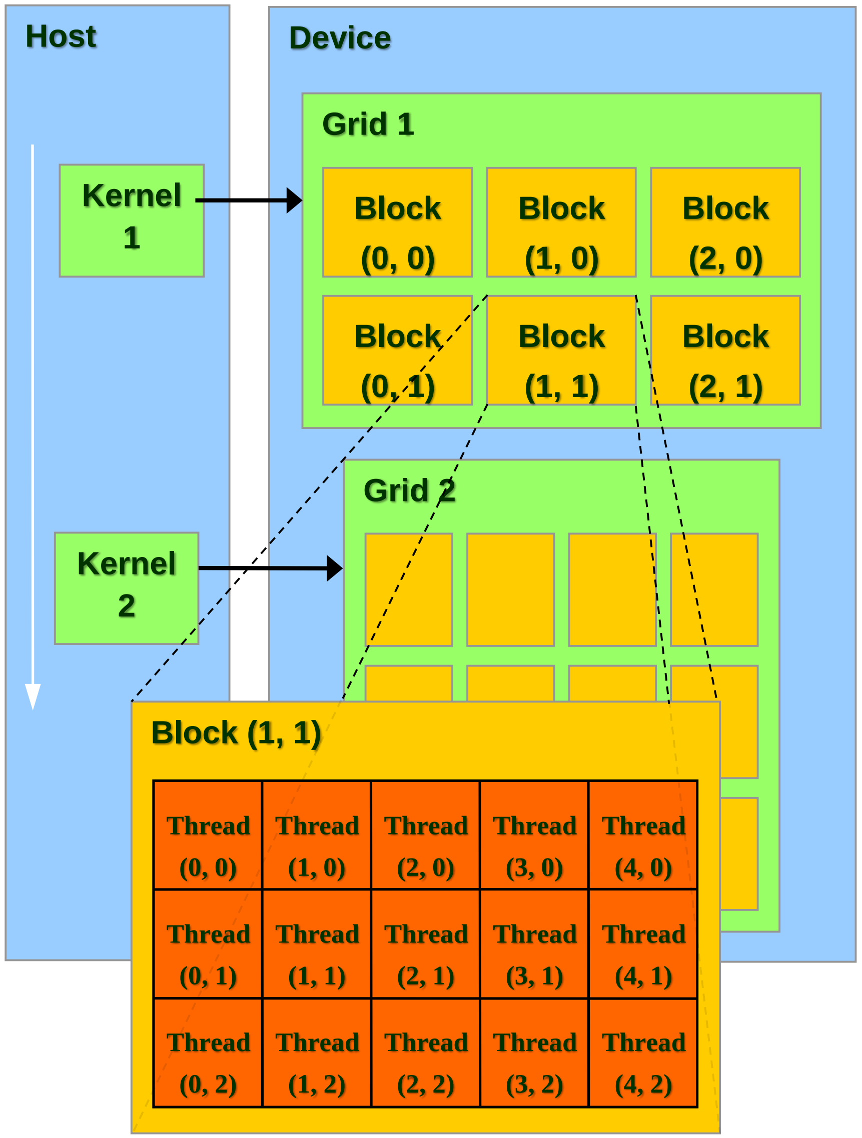
Streaming Multiprocessors

- Streaming Multiprocessor (SMP) contains
  - Streaming Processors (SP)
  - Super (special) Function Units (SFU)
  - Texture Unit
  - Shared memory, register memory
  - L1 caches for instruction, data
- Streaming processors are ‘simple’ processing elements
  - No vector operations, no OOO execution
  - Executes a thread from the larger task pool
  - Relies on switching among threads to cover latency of computation and memory operations



CUDA Threading Organization

- A kernel denotes a program to be executed on the GPU device
- All threads are executing the same kernel (SPMD)
- Threads are organized into groups called Blocks
- The overall problem is partitioned into blocks (denoted a Grid)
- Every thread is provided ID info threadIdx and blockIdx along with blockDim and gridDim



Simple Example

- Consider a simple operation where we want to scale every pixel by a certain factor
- Conventional CPU code

```
void scale(float* img, int N, int M, float scaling) {
  for (int y = 0; y < M; y++) {
     for (int x = 0; x < N; x++) {
         img[x + N * y] *= scaling;
     }
  }
}
```

- GPU Kernel code

```
void scale(float* img, int N, float scaling) {
   xindex = threadIdx.x + blockIdx.x * blockDim.x
   yindex = threadIdx.y + blockIdx.y * blockDim.y
   img[xindex + N * yindex] *= scaling;
}
```

Thread Scheduling

- Each Thread Block is divided into Warps
- Warps are a group of threads that execute concurrently on an SMP
- Warps are scheduling units in SMP
  - The SMP has zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

Threading and Block Scheduling

- Threads are an extremely light weight construct in CUDA
  - For full utilization, 1000s of threads will be required
  - Each thread requires some resources (registers, shared memory)
- The resource requirements of a block is the aggregate requirements of all threads in the block
- Blocks are scheduled to run on the SMPs
  - No particular order or organization guarantees
  - Number of blocks assigned to each SMP depends on block resources vs. SMP resources
- A key design decision is that threads within a block can cooperate but threads from different blocks cannot

Memory Hierarchy

- Registers
  - Private per thread
  - Stores intermediate values in computation
  - Compiler-determined
- Shared Memory
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory
  - Shared by all blocks, all threads
  - Inter-block* and inter-grid communication

Concurrency and Communication

- Threads within a block logically execute concurrently but only threads within a given warp physically execute concurrently
- Ordering of execution among warps is not guaranteed
- Communication among all threads (within or between warps) is possible via Shared Memory
    - Shared memory is low-latency memory within an SMP
    - Required shared memory declared as part of block properties
    - Shared memory is visible to all threads in a block but not with other blocks
    - Communication via write/read operations to shared memory
- Synchronization primitive __synchthreads() provides a barrier among all warps to enforce correctness

Branch divergence

- All threads within a warp execute the same instruction at the same time
- Threads can conditionally execute code based on thread index or other input

```
void kernel_func() {
    if (threadIdx.x % 2) {
        // do something on odd pixels
    } else {
        // do something on even pixels
    }
}
```

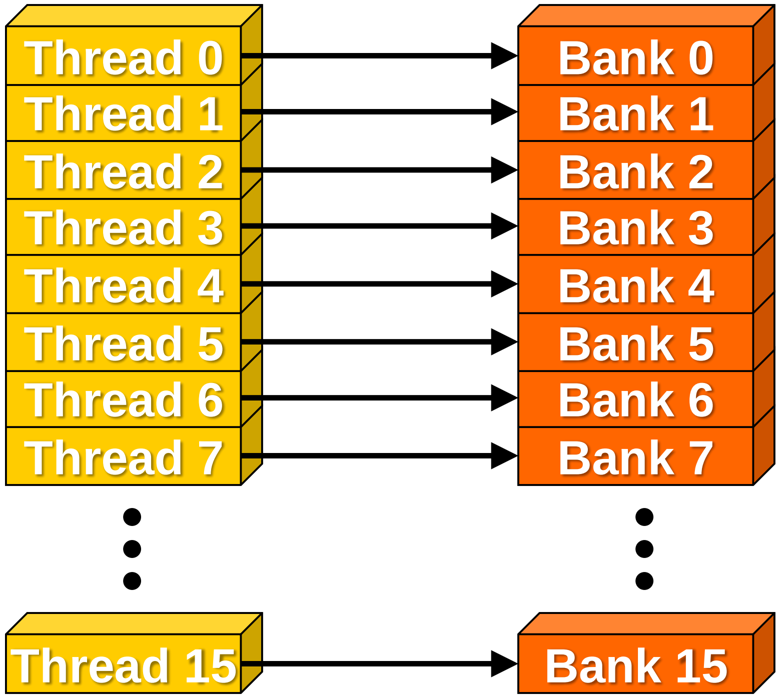
- This is managed in the hardware by executing both paths and using predicates
  - If the predicate (conditional) for an instruction is false, the instruction becomes a no-op
- Incurs runtime overhead of executing both branch paths (branch divergence)
- If all threads within a warp branch the same way, this is avoided

Memory Bank Conflicts

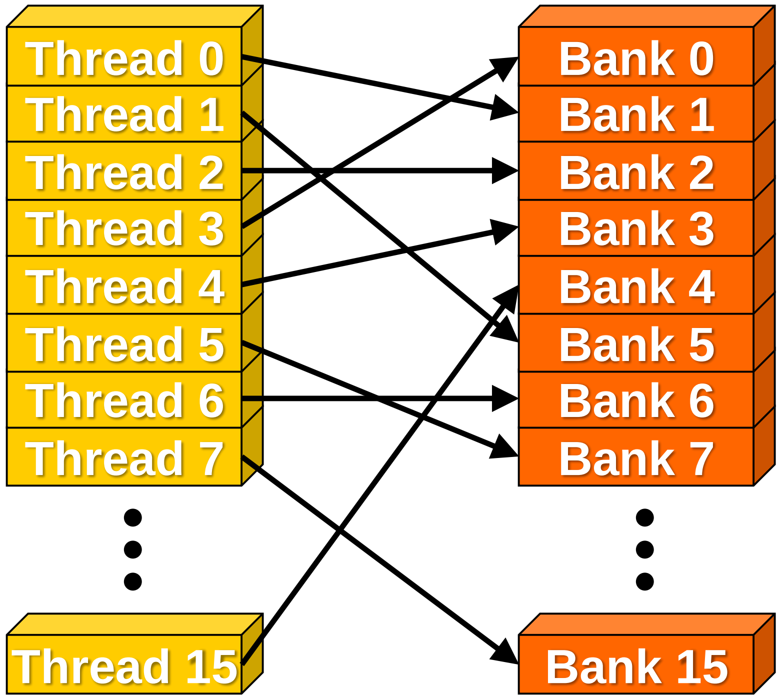
- In a parallel machine, many threads access memory at the same time
  - In order to achieve high memory bandwidth, memory is divided into banks
- Each bank can service one address per cycle
  - Total memory subsystem can service as many simultaneous accesses as it has banks
- There are as many banks and SPs, however access patterns are not restricted
  - Multiple simultaneous accesses to a bank result in a bank conflict and are serialized

Memory Bank Conflicts

- No Bank Conflicts

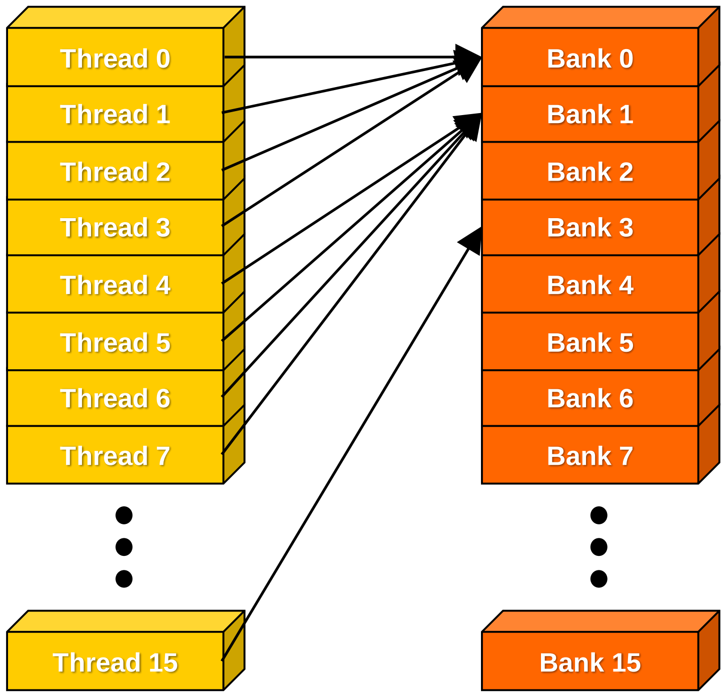


- No Bank Conflicts



Memory Bank Conflicts

- Bank Conflicts



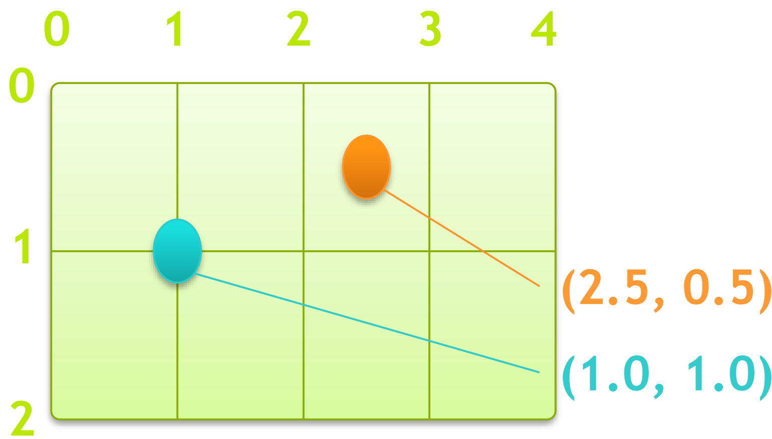
- Note: Bank conflicts only occur if accessing different memory addresses that map to the same bank. Reading from the same address is OK.

Memory Coalescing

- Access to global memory is via a very wide bus
- Reading or writing to a particular address involves accessing the full block of data at that address
- In order to make memory access efficient, threads should access contiguous pieces of data, so that individual requests by different threads coalesce into a single memory operation
- Without coalescing, global memory accesses are serialized, inducing extra latency and wasting the memory bus bandwidth

Texture Unit

- Present in each SMP
- Accesses read-only texture memory from dedicated cache
- Includes hardware for performing linear interpolation among samples
- Automatically handles boundary effects
- Has some associated latency, but allows for off-loading some computation / memory effort



Efficient GPU Programming

- Key features of highly efficient GPU algorithms
  - Massively parallel
  - Little to no branch divergence
  - Low memory requirements
- High occupancy (number of warps scheduled for an SMP)
  - SMP resources divided by resource requirements per block (registers, shared memory)
- Other tweaks
  - Memory optimizations (preload into shared memory, bank conflict elimination, global memory coalescing)
  - Control flow to reduce branch divergence
  - Leverage texture unit

OpenCL / GPU on Android

- It is possible to develop CUDA programs natively for nVidia based chipsets
- OpenCL is a standardized abstraction for high performance computing platforms and encompasses CPUs, GPUs, and other hardware accelerators
    - Using OpenCL can provide most cross-platform support among Android devices
- OpenCL is supported as part of the Android SDK and is recommended
- Some library routines in OpenCV can also take advantage of GPU resources for acceleration

This Week

- Revised Final Project Proposals (with ‘final’ Assigned Lab results) due this week
- Final project work
  - Milestone 1 demo next week