The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.

Edsger Dijkstra
Before We Get Started…

- Last time
  - Example, working w/ large arrays
  - Timing a kernel execution
  - The CUDA API

- Today
  - The memory ecosystem

- Miscellaneous
  - Fourth assignment will be posted today and due on Monday, October 7, at 11:59 PM
    - GPU computing related
    - Kicks off a series of four challenging assignments
  - Read pages 73 through its end:
    http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWInterface.pdf
    - Please post suggestions for improvement
Before Diving In: A Word On CUDA Streams

- A sequence of CUDA calls should be visualized as belonging to a stream.

- IMPORTANT: All CUDA calls (items) in a stream are strictly executed in the order in which they were “deposited” into this stream AND no item in the stream starts before the following item in the stream finishes.
  - Example:
    - Imagine you have a sequence of cudaMemcpy-1 followed by kernel-call followed by cudaMemcpy-2.
    - cudaMemcpy-2 will not start before the GPU finishes execution of the kernel-call.

- Recall that kernel calls are asynchronous (implications when timing calls).

- cudaMemcpy() is synchronous (blocks the execution of CPU).
  - There is an asynchronous version as.
    - Even for the asynchronous version, the strict execution order in the stream is observed.
End API discussion
…… transitioning into…
The Memory Ecosystem
Fermi: Global Memory

- Up to 6 GB of “global memory”
- “Global” in the sense that it doesn’t belong to an SM but rather all SM can access it
GPU vs. CPU – Memory Bandwidth

[GB/sec]
The Fermi Architecture

- 64 KB L1 cache & shared memory
- 768 KB L2 uniform cache (shared by all SMs)
- Memory operates at its own clock rate
- High memory bandwidth
  - Close to 200 GB/s
CUDA Device Memory Space Overview

[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on an SM

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memory

IMPORTANT NOTE: Global, constant, and texture memory spaces are persistent across kernels called by the same host application.
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads

**NOTE:** We will not emphasize texture memory in this class.
Note the presence of local memory, which is virtual memory:
- If too many registers are needed for computation (“high register pressure”) the ensuing data overflow is stored in local memory.
- “Local” means that it’s local, or specific, to one thread.
- In fact local memory is part of the global memory.
- Long access times for local memory (in Fermi, local memory is cached).
## Storage Locations

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A – resident</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A – resident</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

Off-chip means on-device; i.e., slow access time.
Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached
The Three Most Important Parallel Memory Spaces

- **Register**: per-thread basis
  - Private per thread
  - Can spill into local memory (potential performance hit if not cached)

- **Shared Memory**: per-block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication

- **Global Memory**: per-application basis
  - Available for use to all threads
  - Used for: Inter-thread communication
  - Also used for inter-grid communication

Thread

Block

Global Memory

Grid 0

Grid 1

Sequential Grids in Time
SM Register File (RF) [Tesla C1060]

- Register File (RF)
  - 64 KB (Tesla: 16,384 four byte words)
  - Provides 4 operands/clock cycle
  - Note: typical CPU has less than 20 registers per core

- TEX pipe can also read/write RF

- Global Memory Load/Store pipe can also read/write RF
Programmer View of Register File

- Number of 32 bit registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla
  - 32K on Fermi
  - 64K on Kepler

- Size of Register File dependent on your compute capability

- Registers are dynamically partitioned across all Blocks assigned to the SM

- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks

- A thread in a Block can only access registers assigned to itself
  - Kepler: a thread can have up to 255 registers

Possible per-block partitioning scenarios of the RF available on the SM
Matrix Multiplication Example

[Tesla C1060]

- If each Block has 16X16 threads and each thread uses 20 registers, how many blocks can run on each SM?
  - Each Block requires 20*256 = 5120 registers
  - 16,384 = 3 * 5120 + pocket change
  - As such, three blocks can run on an SM as far as registers are concerned

- What if each thread increases the use of registers from 20 to 22?
  - Each Block now requires 22*256 = 5632 registers
  - 16,384 < 16896 = 5632 *3
  - Only two Blocks can run on an SM, about 33% reduction of parallelism!!!

- This example shows why understanding the underlying hardware is essential if you want to squeeze performance out of parallelism
  - One way to find out how many registers you use per thread is to invoke the compile flag -ptax-options=-v when you compile with nvcc
More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers

  - One can run a smaller number of threads that require many registers each, or run a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.

- Tradeoff between instruction-level parallelism (CPU) and thread level parallelism (GPU)
  - TLP: many threads are launched
  - ILP: few threads are launched, but for each thread several instructions can be executed simultaneously
Constant Memory

- This comes handy when all threads use the same *constant* value in their computation
  - Example: \( \pi \), some spring force constant, e=2.7173, etc.

- Constants are stored in DRAM but cached on chip
  - There is a limited amount of L1 cache per SM
  - Might run into slow access if for example have a large number of constants used to compute some complicated formula (might overflow the cache…)

- A constant value can be broadcast to all threads in a warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block
  - When all threads in a warp read the same constant memory address this is as fast as a register
Example, Use of Constant Memory
[For compute capability 2.0 (GTX480, C2050) – due to use of “printf”]

```c
#include <stdio.h>

// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;

// Some dummy function that uses the constant variable
__global__ void myExample() {
    float circum = 2.f*dansPI*threadIdx.x;
    printf("Hello thread %d, Circ=%5.2f\n", threadIdx.x, circum);
}

int main(int argc, char **argv) {
    float somePI = 3.141579f;
    cudaMemcpyToSymbol(dansPI, &somePI, sizeof(float));
    myExample<<<1, 16>>>(0); //-dimensional kernel launch
    cudaThreadSynchronize();
    return 0;
}
```

Hello thread 0, Circ= 0.00
Hello thread 1, Circ= 6.28
Hello thread 2, Circ=12.57
Hello thread 3, Circ=18.85
Hello thread 4, Circ=25.13
Hello thread 5, Circ=31.42
Hello thread 6, Circ=37.70
Hello thread 7, Circ=43.98
Hello thread 8, Circ=50.27
Hello thread 9, Circ=56.55
Hello thread 10, Circ=62.83
Hello thread 11, Circ=69.11
Hello thread 12, Circ=75.40
Hello thread 13, Circ=81.68
Hello thread 14, Circ=87.96
Hello thread 15, Circ=94.25
Matrix Multiplication Example, Revisited

- **Purpose**
  - See an example where the use of multiple blocks of threads plays a central role
  - Emphasize the role of the shared memory
  - Emphasize the need for the `__syncthreads()` function call

- **NOTE:** A one dimensional array stores the entries in the matrix
Why Revisit the Matrix Multiplication Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction (“arithmetic intensity”) very low
  - Each arithmetic computation required one fetch from global memory
  - The matrix M (its entries) is copied from global memory to the device N.width times
  - The matrix N (its entries) is copied from global memory to the device M.height times

- When solving a numerical problem the goal is to go through the chain of computations as fast as possible
  - You don’t get brownie points moving data around but only computing things
The Common Pattern to CUDA Programming

- **Phase 1**: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU.

- **Phase 2**: Let the GPU crunch the numbers based on the kernel that you defined.

- **Phase 3**: Bring back the results from the GPU. Free memory on the device (clean up…). You’re done.

**Rules of Thumb for Efficient GPU Computing:**

1. Get the data on the GPU and keep it there
2. Give the GPU enough work to do
3. Focus on data reuse within the GPU to avoid memory bandwidth limitations
A Common Programming Pattern
BRINGING THE SHARED MEMORY INTO THE PICTURE

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory

- An advantageous way of performing computation on the device is to partition (“tile”) data to take advantage of fast shared memory:
  - **Partition** data into **data subsets (tiles)** that each fits into shared memory
  - **Handle** each data subset (tile) with one thread block by:
    - Loading the tile from global memory into shared memory, **using multiple threads to exploit memory-level parallelism**
    - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory back to global memory
Multiply Using Several Blocks

- One block computes one square sub-matrix $C_{sub}$ of size $Block_Size$

- One thread computes one entry of $C_{sub}$

**Assumption:** $A$ and $B$ are *square matrices* and their dimensions are multiples of $Block_Size$

- Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
- In this example work with $Block_Size=16x16$

---

**NOTE 1:** Similar example provided in the CUDA Programming Guide 3.2
- Available on the 2011 class website

**NOTE 2:** A similar technique is used on CPUs to improve cache hits. See slide “Blocking Example” at 