Innovation distinguishes between a leader and a follower.
Steve Jobs
Before We Get Started…

- Last time
  - Execution scheduling issues
  - Discussion of global memory access patterns

- Today
  - Shared memory, further considerations
  - Synchronization for Data Communication under CUDA
  - Atomic operations
  - CUDA Optimization/Best Practices issues

- Miscellaneous
  - Fourth assignment due tonight at 11:59 PM
  - Fifth assignment posted later today. GPU computing related
  - We’re half way through this class: please let me know what you think
    - Please provide feedback on Wednesday, Oct 9 – details to follow in an email
    - I’ll compile all of your feedback and upload on the class website
  - Exam: Th, November 7, 7-9 PM (no class on Friday, Nov. 8)
    - Review session on Wd, Nov. 6 @ 6 PM
    - Exam will draw on material covered in class and information provided in the primer
    - It’ll be a pen and paper exam. Open book and open anything

Exam: Th, November 7, 7-9 PM (no class on Friday, Nov. 8)
Shared Memory: Syntax & Semantics

- You can statically declare shared memory like in the code snippet below:

```c
__global__ void coalescedMultiply(float *a, float* b, float *c, int N) {
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

- The variable `aTile` visible to all threads in each block, and only to those threads
  - The thread that executes the kernel above sees the `aTile` declaration and understands that all its sibling-threads in the block are going to see it too. They share this variable collectively
- The same thread, when it sees the variable `row` it understands that it has sole ownership of this variable (variable stored in a register)
3 Ways to Set Aside Shared Memory

- First way: Statically, declared inside a kernel
  - See previous slide…

- Second way: Through the execution configuration
  - Not that common
  - \textbf{Ns} below indicates size (in bytes) to be allocated in shared memory

```c
__global__ void MyFunc(float*) // __device__ or __global__ function
{
    extern __shared__ float shMemArray[];
    // Size of shMemArray determined through the execution configuration
    // You can use shMemArray as you wish here…
}
```

// invoke like this
MyFunc<<< Dg, Db, Ns >>>(parameter);

- Third way: Dynamically, through the CUDA Driver API
  - Advanced feature, uses API function \texttt{cuFuncSetSharedSize()}, not discussed here
Shared Memory Architecture

- Common sense observation: in a parallel machine many threads access memory at the same time
  - To service more than one thread, memory is divided into independent banks
  - This layout essential to achieve high bandwidth

- Each SM has ShMem organized in 32 Memory banks

- Recall that shared memory and L1 cache draw on the same physical memory inside an SM; i.e., they combine for 64 KB
  - This physical memory can be partitioned as
    - 48 KB of ShMem and 16 KB of L1 cache
    - The other way around
  - Note: shared memory can store less data than the registers (48 KB vs. 128 KB)
Shared Memory Architecture

- The 32 banks of the Shared Memory are organized like benches in a movie theater
  - You have multiple rows of benches
  - Each row has 32 benches
  - In each bench you can “seat” a family of four bytes (32 bits total)
  - Note that a bank represents a column of benches in the movie theater, which is perpendicular to the screen

- Each bank has a bandwidth of 32 bits per two clock cycles
Shared Memory: Transaction Rules & Bank Conflicts

- When reading in four-byte words, 32 threads in a warp attempt to access shared memory simultaneously.

- Bank conflict: the scenario where two different threads access *different* words in the same bank.

- Note that there is no conflict if different threads access any bytes within the same word.

- Bank conflicts enforce the hardware to serialize your ShMem access, which adversely impacts bandwidth.
Shared Memory Bank Conflicts

- If there are no bank conflicts:
  - Shared memory access is fast, but not as fast as register access
  - On the bright side, latency is roughly 100x lower than global memory latency

- Share memory access, the fast case:
  - If all threads of a warp access different banks, there is no bank conflict
  - If all threads of a warp access an identical address for a fetch operation, there is no bank conflict (broadcast)

- Share memory access, the slow case:
  - Worst case: 32 threads access 32 different words in the same bank
  - Must serialize all the accesses
  - In general, cost = max # of simultaneous accesses to a single bank
How Addresses Map to Banks on Fermi

- Successive 32-bit word addresses are assigned to successive banks

- Bank you work with = (address of offset) % 32
  
  - This is because Fermi has 32 banks
  
  - Example: 1D shared mem array, `myShMem`, of 1024 floats
    
    - `myShMem[4]`: accesses bank id #4 (relative row offset: 0)
    - `myShMem[31]`: accesses bank id #31 (relative row offset: 0)
    - `myShMem[50]`: access bank id #18 (relative row offset: 1)
    - `myShMem[128]`: access bank id #0 (relative row offset: 4)
    - `myShMem[178]`: access bank id #18 (relative row offset: 5)

- NOTE: If, for instance, the third thread in a warp accesses `myShMem[50]` and the eight thread in the warp access `myShMem[178]`, then you have a two-way bank conflict and the two transactions get serialized

- IMPORTANT: There is no such thing as “bank conflicts” between threads belonging to different warps
Bank Addressing Examples
Transactions Involving 4 Byte Words

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
Bank Addressing Examples
Transactions Involving 4 Byte Words
Other Examples

- Two “no conflict read” scenarios:
  - Broadcast: all threads in a warp access the same word in a bank
  - Multicast: several threads in a warp access the same word in the same bank
Data types and bank conflicts

- No conflicts below if `shrd` is a 32-bit data type:
  
  ```
  foo = shrd[baseIndex + threadIdx.x]
  ```

- Also if accessing one byte/thread, no conflict since *different* bytes of the same word are accessed
  - No conflicts:
    ```
    extern __shared__ char shrd[];
    foo = shrd[baseIndex + threadIdx.x];
    ```

  - No conflicts:
    ```
    extern __shared__ short shrd[];
    foo = shrd[baseIndex + threadIdx.x];
    ```
Exercise: Is ShMem access below good or bad?

- Each thread loads two floats into shared memory:

```c
int tid = threadIdx.x;
sharedVar[2*tid ] = globalVar[2*tid ];
sharedVar[2*tid+1] = globalVar[2*tid+1];
```

- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic
  - Doesn’t make sense in shared memory usage where there is no cache line effects but banking effects
  - 2-way-interleaved loads result in 2-way bank conflicts

- Adding insult to injury: you don’t have coalesced global memory loads – basically you are halving the device memory bandwidth
Linear Addressing

- Given:
  ```
  __shared__ float sharedM[256];
  float foo = sharedM[baseIndex + s * threadIdx.x];
  ```

- This is bank-conflict-free if \( s \) shares no common factors with the number of banks
  - Conclusion: you are fine if \( s \) is odd
The Math Beyond Bank Conflicts

- We are in a half-warp, and the question is if thread \( t_1 \) and thread \( t_2 > t_1 \) might access the same bank of shared memory.
- Let \( b \) be the base of the array (the “shareM” pointer on previous slide).
- How should you not choose \( s \)?

\[
\begin{align*}
\{ & b + st_2 = b + st_1 + 32k, \quad \text{for some positive integer } k \\
& 0 < t_2 - t_1 \leq 32 \\
& \end{align*}
\]

\[
\begin{align*}
\{ & 32k = s(t_2 - t_1) \\
& 0 < t_2 - t_1 \leq 32 \\
& \end{align*}
\]

- If \( s=2 \), take \( k=1 \), and then any threads \( t_1 \) and \( t_2 \) which are 16 apart satisfy the condition above and will have a bank conflict ([0,16], [1,17], etc.) – two way conflict.
- If \( s=4 \), take \( k=2 \), any threads \( t_1 \) and \( t_2 \) which are 8 apart will have a bank conflict ([0,8,16,24], [1,9,17,25], etc.) – four way conflict.
- NOTE: you can’t get a bank conflict is \( s \) is odd (no quartet \( k, s, t_1, t_2 \) satisfies the bank conflict condition above). So take stride \( s=1,3,5, \) etc.
Example, ShMem Use: Vector Reduction

- Bring data in shared memory, then start adding in parallel
- Fewer and fewer threads participate
- The process is memory bound, low arithmetic ratio...
- Covered in more detail on Th (also part of the Assignment)
  - Used as a vehicle to demonstrate CUDA optimization techniques

**Diagram:**
- Data staged in shared memory
- A small number of threads finishes off

NVIDIA [M. Harris]→
Example: Vector Reduction with Bank Conflicts
(assume 2048 vector entries stored in shared memory; one block (1024 threads) carries out the reduction)
Vector Reduction **without** Bank Conflicts

(assume 2048 vector entries stored in shared memory; one block (1024 threads) carries out the reduction)
Shared Memory: A Word of Caution

- It used to be that any access to Shared Memory was a direct access (in compute capability 1.x).

- Fermi (2.x) has a load/store architecture that can bring data into registers.
  - This means that there is no guarantee for coherence between the shared memory block and the value stored in the register.

- Problem is typically addressed by making that shared memory **volatile**:
  - In 1.x, this was always ok:
    ```c
    __shared__ int myShVars[256];
    ```
  - In 2.x, you might have to do this (the compiler doesn’t optimize instructions related to `myShVars`):
    ```c
    volatile __shared__ int myShVars[256];
    ```

More information about shared memory: Programming Guide, Sections 3.2.3, 5.3.2.3, and Appendix F4.3
Example: Is 48KB of Shared Memory Enough?
[Revisiting the Matrix Multiplication Example]

- One **block** computes one tile $C_{\text{sub}}$ of size $\text{Block} \_\text{Size}$

- One **thread** computes one element of $C_{\text{sub}}$

- Assume that the dimensions of $A$ and $B$ are multiples of $\text{Block} \_\text{Size}$ and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
Example: Matrix Multiplication
Shared Memory Usage - WIDTH = 16

- Each Block requires $2 \times WIDTH^2 \times 4$ bytes of shared memory storage

  - For WIDTH = 16, each BLOCK requires 2KB, up to 24 Blocks can fit into the Shared Memory of GTX480
    - Note that if you have the setting for ShMemory that gives you 16 KB you can still fit 8 blocks on one SM

  - Since each SM scheduler on GTX480 can only manage 1536 threads (48 warps), each SM can only take 6 Blocks of 256 threads each
    - Then, you have 100% occupancy

- Shared memory size is not a constraint for our implementation of the Matrix Multiplication
Example: Matrix Multiplication
Shared Memory Usage - WIDTH = 32

- Each Block requires $2 \times WIDTH^2 \times 4$ bytes of shared memory storage
  - For WIDTH = 32, each BLOCK requires 8KB, up to 6 Blocks can fit into the
    Shared Memory of GTX480
    - Note that if you have the setting for ShMemory that gives you 16 KB you can still
      fit 2 blocks on one SM
  - Since each SM on GTX480 can only manage 1536 threads, each SM can only take 1 Block of 1024 threads
    - Then, you have 66% occupancy
  - Conclusion: It’s likely that this will run slower than the WIDTH=16 options
    - Not necessarily true, since there are other factors (number of registers used, potential
      for compiler code optimization, etc.) that come into play
Synchronization Issues
Global Communication

- Keep this in mind: in this segment we are **not** talking about execution scheduling
  - Focus is on making data available to other threads and required synchronization

- How can Global Communication occur?
  - For threads in different blocks and different grids:
    - Locations in global memory (global variables)
  - For threads in same blocks:
    - Locations in global memory
    - Locations in shared memory (\texttt{__shared__} variables)

- Looming danger: race conditions…
Race Conditions

[1/2]

- Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is a write

```c
// race.cu, the kernel
__global__ void race(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    *x = i;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudamemcpy(x, d_x, sizeof(int), cudamemcpyDeviceToHost);
```
Race Conditions

Programs with race conditions may produce unexpected, seemingly arbitrary results
- Updates may be missed, and updates may be lost

```c
// race.cu, the kernel
__global__ void race(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    *x = *x + 1;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudamemcpy(x, d_x, sizeof(int), cudamemcpyDeviceToHost);
```
Synchronization for Data Communication

- Accesses to shared locations need to be correctly synchronized (coordinated) to avoid race conditions.

- In many common shared memory multithreaded programming models, one uses coordination objects such as locks to synchronize accesses to shared data.

- CUDA provides several scalable synchronization mechanisms, such as efficient barriers and atomic memory operations.

- Whenever possible, try hard to design algorithms with few synchronizations.
  - Synchronization impacts execution speed.