There are three rules to follow when parallelizing large codes.
Unfortunately, no one knows what these rules are.

W. Somerset Maugham and Gary Montry
Before We Get Started…

- **Last time**
  - Scheduling issues
    - Thread divergence
  - Device memory
    - Coalesced memory accesses
    - L1 & L2 caches

- **Today**
  - Shared memory, further considerations
  - Synchronization issues
  - Atomic operations
  - CUDA Optimization/Best Practices issues

- **Other issues**
  - HW6 due at 11:59 PM
  - Half page proposal for your Midterm Project due today
    - For default project (solving dense banded linear system): no need to submit anything.
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
  - \( Ns \) 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 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 #4 (physically, the fifth one – first row)
    - `myShMem[31]`: accesses bank #31 (physically, the last one – first row)
    - `myShMem[50]`: access bank #18 (physically, the 19th one – second row)
    - `myShMem[128]`: access bank #0 (physically, the first one – fifth row)
    - `myShMem[178]`: access bank #18 (physically, the 19th one – sixth row)
  - NOTE: If, for instance, the third thread in a warp accesses `myShMem[50]` and the eight thread in the warp accesses `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

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7

...Thread 31...

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7

...Bank 31...

- No Bank Conflicts
  - Random 1:1 Permutation

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7

...Thread 31...

Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7

...Bank 31...
Bank Addressing Examples
Transactions Involving 4 Byte Words
Other Examples

- Two “no conflict” 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
Linear Addressing

- Given:
  ```c
  __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.
Data types and bank conflicts

- No conflicts below if `shrd` is a 32-bit data type:

  ```c
  foo = shrd[baseIndex + threadIdx.x]
  ```

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

  - No conflicts:
    ```c
    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;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[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 mem bandwidth
A Better Array Access Pattern

- Here's a better way of doing it
  - Each thread loads one element in every consecutive group of blockDim elements.

```c
shared[tid] = global[tid];
shared[tid + blockDim.x] = global[tid + blockDim.x];
```
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

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

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

- Assume that the dimensions of $A$ and $B$ are multiples of Block_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 on GTX480 can only manage 1536 threads, 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

- 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 (\_\_shared\_\_ variables)

- Looming danger: race conditions…
Race Conditions

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

// 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

- 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
Synchronization

- Assume thread T1 reads a value defined by thread T0

// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}

// main.cpp
update_race<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);

- Program needs to ensure that thread T1 reads location after thread T0 has written location
Synchronization within Block

- Threads in same block: can use `__syncthreads()` to specify synchronization point that orders accesses

```c
// update.cu
__global__ void update(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
    __syncthreads();
    if (i == 1) *y = *x;
}

// main.cpp
update<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Important: all threads within the block must reach the `__syncthreads()` statement
Synchronization between Grids

- Threads in different grids: system ensures writes from kernel happen before reads from subsequent grid launches.

```
// update.cu
__global__ void update_x(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
}

__global__ void update_y(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 1) *y = *x;
}

// main.cpp
update_x<<<1,2>>>(d_x, d_y);
update_y<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Synchronization within Grid

- Often not reasonable to split kernels to synchronize reads and writes from different threads to common locations
  - Values of `__shared__` variables are lost unless explicitly saved
  - Kernel launch overhead is non-trivial, and introducing extra launches can degrade performance

- CUDA provides atomic functions (commonly called atomic memory operations) to enforce atomic accesses to shared variables that may be accessed by multiple threads

- Programmers can synthesize various coordination objects and synchronization schemes using atomic functions.
Atoms
Atom memory operations (atomic functions) are used to solve all kinds of synchronization and coordination problems in parallel computer systems.

General concept: provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads.

This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.
Atomic Functions

Atomic functions perform read-modify-write operations on data residing in global and shared memory.

```c
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = atomicAdd(x, 1);  // j = *x;
}

// main.cpp
int x = 0;
cudaMemcpy(d_x, x, cudaMemcpyHostToDevice);
update<<<1,128>>>>;
cudaMemcpy(&x, d_x, cudaMemcpyHostToDevice);
```

Atomic functions guarantee that only one thread may access a memory location while the operation completes.

Order in which threads get to write is not specified though…
Atomic Functions

Atomic functions perform read-modify-write operations on data residing in global and shared memory.

Synopsis of atomic function $\text{atomicOP}(a,b)$ is typically

```c
    t1 = *a;       // read
    t2 = t1 OP (*b);  // modify
    *a = t2;       // write
    return t1;
```

The hardware ensures that all statements are executed atomically without interruption by any other atomic functions.

The atomic function returns the initial value, not the final value, stored at the memory location.
Atomic Functions

- The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates.

- The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary.

- However, none of the atomic updates will be lost.

- Many different kinds of atomic operations:
  - Add (add), Sub (subtract), Inc (increment), Dec (decrement)
  - And (bit-wise and), Or (bit-wise or), Xor (bit-wise exclusive or)
  - Exch (Exchange)
  - Min (Minimum), Max (Maximum)
  - Compare-and-Swap
Histogram Example

// Compute histogram of colors in an image

//
//  color – pointer to picture color data
//  bucket – pointer to histogram buckets, one per color
//

__global__ void histogram(int n, int* color, int* bucket)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
    {
        int c = colors[i];
        atomicAdd(&bucket[c], 1);
    }
}
Work Queue Example

// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense
// to continuously grab work from a queue

__device__ int do_work(int x)
{
    return f(x-1) + f(x) + f(x+1);
}

__global__ void process_work_q(int* work_q, int* q_counter,
                                int* output, int queue_max)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[i] = result;
}

NVIDIA [J. Balfour]→
Performance Notes

- Atomics are slower than normal accesses (loads, stores)
- Performance can degrade when many threads attempt to perform atomic operations on a small number of locations
- Possible to have all threads on the machine stalled, waiting to perform atomic operations on a single memory location
- Atomics: convenient to use, come at a typically high efficiency loss…
Example: Global Min/Max (Naive)

- Compute maximum across all threads in a grid
- One can use a single global maximum value, but it will be VERY slow

```c
__global__ void global_max(int* values, int* global_max) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = values[i];
    atomicMax(global_max, val);
}
```
Example: Global Min/Max (Better)

- Introduce local maximums and update global only when new local maximum found

```c
__global__ void global_max(int* values, int* global_max, int *local_max, int num_locals)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = values[i];
    int li = i % num_locals;
    int old_max = atomicMax(&local_max[li], val);
    if (old_max < val)
    {
        atomicMax(global_max, val);
    }
}
```

- Reduces frequency at which threads attempt to update the global maximum, reducing competition access to location

NVIDIA [J. Balfour]→
Lessons from global Min/Max

- Many updates to a single value causes serial bottleneck
- One can create a hierarchy of values to introduce more parallelism and locality into algorithm
- However, performance can still be slow, so use judiciously
Important note about Atomics

- Atomic updates are not guaranteed to appear atomic to concurrent accesses using loads and stores

```c
__global__ void broken(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0)
    {
        *x = *x + 1;
    }
    else
    {
        int j = atomicAdd(x, 1); // j = *x; *x = j + i;
    }
}
```

// main.cpp
broken<<<1,128>>>(128, d_x); // d_x = d_x + {1, 127, 128}
Summary of Atomics

- When to use: Cannot use normal load/store for reliable inter-thread communication because of race conditions

- Use atomic functions for infrequent, sparse, and/or unpredictable global communication

- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism

- Attempt to use shared memory and structure algorithms to avoid synchronization whenever possible