REDUCTIONS
Reduction

**Reduce** vector to a single value
- Via an associative operator (+, *, min/max, AND/OR, ...)
- CPU: sequential implementation
  
  ```
  for(int i = 0, i < n, ++i) ...
  ```
- GPU: “tree”-based implementation

![Diagram of a reduction tree]

```
Serial Reduction

// reduction via serial iteration
float sum(float *data, int n) {
    float result = 0;
    for(int i = 0; i < n; ++i) {
        result += data[i];
    }
    return result;
}
Parallel Reduction – Interleaved

Values (in shared memory)

10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 1
Stride 1

Step 2
Stride 2

Step 3
Stride 4

Step 4
Stride 8

Thread IDs

Thread IDs

Thread IDs

Thread IDs
CUDA Reduction

```
__global__ void block_sum(float *input,
                          float *results,
                          size_t n)
{
    extern __shared__ float sdata[];
    int i = ..., int tx = threadIdx.x;

    // load input into __shared__ memory
    float x = 0;
    if(i < n)
        x = input[i];
    sdata[tx] = x;
    __syncthreads();
```
CUDA Reduction

// block-wide reduction in __shared__ mem
for(int offset = blockDim.x / 2;
    offset > 0;
    offset >>= 1)
{
    if(tx < offset)
    {
        // add a partial sum upstream to our own
        sdata[tx] += sdata[tx + offset];
    }
    __syncthreads();
}
CUDA Reduction

    // finally, thread 0 writes the result
    if(threadIdx.x == 0)
    {
        // note that the result is per-block
        // not per-thread
        results[blockIdx.x] = sdata[0];
    }
}
CUDA Reduction

// global sum via per-block reductions
float sum(float *d_input, size_t n)
{
    size_t block_size = ... , num_blocks = ...;

    // allocate per-block partial sums
    // plus a final total sum
    float *d_sums = 0;
    cudaMalloc((void**)&d_sums,
               sizeof(float) * (num_blocks + 1));
    ...
}
CUDA Reduction

// reduce per-block partial sums
int smem_sz = block_size*sizeof(float);
block_sum<<<num_blocks,block_size,smem_sz>>>(
    d_input, d_sums, n);

// reduce partial sums to a total sum
block_sum<<<1,block_size,smem_sz>>>(
    d_sums, d_sums + num_blocks, num_blocks);

// copy result to host
float result = 0;
Caveat Reductor

- What happens if there are too many partial sums to fit into \texttt{__shared__} memory in the second stage?

- What happens if the temporary storage is too big?

- Give each thread more work in the first stage
  - \texttt{Sum is associative \& commutative}
  - Order doesn’t matter to the result
  - We can schedule the sum any way we want
    → serial accumulation before block-wide reduction

- Exercise left to the hacker
Parallel Reduction Complexity

- **Log($N$)** parallel steps, each step $S$ does $N/2^S$ independent ops
  - **Step Complexity** is $O(\log N)$

- For $N=2^D$, performs $\sum_{S\in[1..D]} 2^{D-S} = N-1$ operations
  - **Work Complexity** is $O(N)$ – It is work-efficient
    - i.e. does not perform more operations than a sequential algorithm

- With $P$ threads physically in parallel ($P$ processors),
  - **time complexity** is $O(N/P + \log N)$
    - Compare to $O(N)$ for sequential reduction
PERFORMANCE CONSIDERATIONS
But First!

- Always measure where your time is going!
  - Even if you think you know where it is going
  - Start coarse, go fine-grained as need be

- Keep in mind Amdahl’s Law when optimizing any part of your code
  - Don’t continue to optimize once a part is only a small fraction of overall execution time
Performance Considerations

- Memory Coalescing
- Shared Memory Bank Conflicts
- Occupancy

Things left out due to time:
- Kernel launch overheads
- Loop iteration count divergence
MEMORY COALESCING
Memory Coalescing

- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don’t use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 32/64/128 bytes
  - Unaligned accesses will cost more
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 64B)
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 32B)
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255 *(reduce to 64B)*
Consider the stride of your accesses

```c
__global__ void foo(int* input, float3* input2)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    // Stride 1
    int a = input[i];
    // Stride 2, half the bandwidth is wasted
    int b = input[2*i];
    // Stride 3, 2/3 of the bandwidth wasted
    float c = input2[i].x;
}
```
Example: Array of Structures (AoS)

```c
struct record {
    int key;
    int value;
    int flag;
};

record *d_records;
cudaMalloc((void**)&d_records, ...);
```
Example: Structure of Arrays (SoA)

```c
struct SoA
{
    int * keys;
    int * values;
    int * flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);
```
Example: SoA vs. AoS

```c
__global__ void bar(record *AoS_data,
                     SoA SoA_data)
{
    int i = blockDim.x * blockIdx.x
             + threadIdx.x;
    // AoS wastes bandwidth
    int key = AoS_data[i].key;
    // SoA efficient use of bandwidth
    int key_better = SoA_data.keys[i];
}
```
Memory Coalescing

- Structure of array is often better than array of structures
  - Very clear win on regular, stride 1 access patterns
  - Unpredictable or irregular access patterns are case-by-case
SHARED MEMORY BANK
CONFLICTS
Shared Memory

- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
  - Threads can each access different banks
  - Or can all access the same value

- Consecutive words are in different banks

- If two or more threads access the same bank but different value, get bank conflicts
Bank Addressing Examples

No Bank Conflicts

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

No Bank Conflicts

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

Thread 15 → Bank 15
Bank Addressing Examples

2-way Bank Conflicts

Thread 0 → Bank 0
Thread 1 → Bank 1
Thread 2 → Bank 2
Thread 3 → Bank 3
Thread 4 → Bank 4
Thread 8 → Bank 5
Thread 9 → Bank 6
Thread 10 → Bank 7
Thread 11 → Bank 15

8-way Bank Conflicts

Thread 0 → Bank 0
Thread 1 → Bank 1
Thread 2 → Bank 2
Thread 3 → Bank 3
Thread 4 → Bank 4
Thread 5 → Bank 5
Thread 6 → Bank 6
Thread 7 → Bank 7
Thread 15 → Bank 15
Trick to Assess Impact On Performance

- Change all SMEM reads to the same value
  - All broadcasts = no conflicts
  - Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for SMEM writes
  - So, replace SMEM array indices with `threadIdx.x`
  - Can also be done to the reads
Reminder: Thread Scheduling

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM*
  - Warps whose next instruction has its inputs 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

\[\text{TB} = \text{Thread Block}, \ W = \text{Warp}\]
Thread Scheduling

- What happens if all warps are stalled?
  - No instruction issued $\rightarrow$ performance lost

- Most common reason for stalling?
  - Waiting on global memory

- If your code reads global memory every couple of instructions
  - You should try to maximize occupancy
What determines occupancy?

- Register usage per thread & shared memory per thread block
Pool of registers and shared memory per SM

- Each thread block grabs registers & shared memory
- If one or the other is fully utilized -> no more thread blocks
Resource Limits (2)

- Can only have 8 thread blocks per SM
  - If they’re too small, can’t fill up the SM
  - Need 128 threads / TB (gt200), 192 thread/ TB (gf100)

- Higher occupancy has diminishing returns for hiding latency
Hiding Latency with more threads

Throughput, 32-bit words
How do you know what you’re using?

- Use `nvcc -Xptxas -v` to get register and shared memory usage.
- Plug those numbers into CUDA Occupancy Calculator.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

### Varying Block Size

![Graph showing varying block sizes and occupancy.]

### Varying Register Count

![Graph showing varying register counts and occupancy.]

### Varying Shared Memory Usage

![Graph showing varying shared memory usage and occupancy.]

### GPU Occupancy Data Displayed Here and in the Graphs

1. **Active Threads per Multiprocessor**: N/A
2. **Active Warps per Multiprocessor**: N/A
3. **Active Thread Blocks per Multiprocessor**: N/A
4. **Occupancy of each Multiprocessor**: N/A

### Physical Limits for GPU Compute Capability

- **Threads per Warp**: N/A
- **Warps per Multiprocessor**: N/A
- **Threads per Multiprocessor**: N/A
- **Thread Blocks per Multiprocessor**: N/A
- **Total # of 32-bit registers per Multiprocessor**: N/A
- **Register allocation unit size**: N/A
- **Register allocation granularity**: N/A
- **Shared Memory per Multiprocessor (bytes)**: N/A
- **Shared Memory Allocation unit size**: N/A
- **Warp allocation granularity (for register allocation)**: N/A

### Allocation Per Thread Block

- **Warps**: N/A
- **Registers**: N/A
- **Shared Memory**: N/A

### Maximum Thread Blocks per Multiprocessor

- **Blocks Limited by Max Warps / Blocks per Multiprocessor**: N/A
- **Blocks Limited by Registers per Multiprocessor**: N/A
- **Blocks Limited by Shared Memory per Multiprocessor**: N/A
- **Thread Block Limit Per Multiprocessor highlighted**: N/A

### CUDA Occupancy Calculator

**Version**: 2.0

**Copyright and License**
CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 1.3

2.) Enter your resource usage:
   - Threads Per Block: 128
   - Registers Per Thread: 25
   - Shared Memory Per Block (bytes): 640

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
3.) GPU Occupancy Data is displayed here and in the graphs:

| Active Threads per Multiprocessor | 512 |
| Active Warps per Multiprocessor   | 16  |
| Active Thread Blocks per Multiprocessor | 4  |
| Occupancy of each Multiprocessor  | 50% |

Physical Limits for GPU Compute Capability:

| Threads per Warp | 32 |
| Warps per Multiprocessor | 32 |
| Threads per Multiprocessor | 1024 |
| Thread Blocks per Multiprocessor | 8 |
| Total # of 32-bit registers per Multiprocessor | 16384 |
| Register allocation unit size | 512 |
| Register allocation granularity | block |
| Shared Memory per Multiprocessor (bytes) | 16384 |
| Shared Memory Allocation unit size | 512 |
| Warp allocation granularity (for register allocation) | 2 |

Allocation Per Thread Block

| Warps | 4 |
| Registers | 3584 |
| Shared Memory | 1024 |

These data are used in computing the occupancy data in blue.

Maximum Thread Blocks Per Multiprocessor

| Limited by Max Warps / Blocks per Multiprocessor | 8 |
| Limited by Registers per Multiprocessor | 4 |
| Limited by Shared Memory per Multiprocessor | 16 |
| Thread Block Limit Per Multiprocessor highlighted | RED |
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.
How to influence how many registers you use

- Pass option `-maxrregcount=X` to nvcc

- This isn’t magic, won’t get occupancy for free

- Use this very carefully when you are right on the edge
Performance Considerations

- Measure, measure, then measure some more!
- Once you identify bottlenecks, apply judicious tuning
  - What is most important depends on your program
  - You’ll often have a series of bottlenecks, where each optimization gives a smaller boost than expected
Questions?
Backup
Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance
__global__ void per_thread_sum(int *indices,
                             float *data,
                             float *sums)
{
    ...

    // number of loop iterations is data dependent
    for(int j=indices[i]; j<indices[i+1]; j++)
    {
        sum += data[j];
    }
    sums[i] = sum;
}
Iteration Divergence

- A single thread can drag a whole warp with it for a long time
- Know your data patterns
- If data is unpredictable, try to flatten peaks by letting threads work on multiple data items
Shared Memory

**Uses:**
- Inter-thread communication within a block
- Cache data to reduce global memory accesses
- Use it to avoid non-coalesced access

**Organization:**
- 16 banks, 32-bit wide banks (Tesla)
- 32 banks, 32-bit wide banks (Fermi)
- Successive 32-bit words belong to different banks

**Performance:**
- 32 bits per bank per 2 clocks per multiprocessor
- smem accesses are per **16**-threads (half-warp)
- **Serialization:** if \( n \) threads (out of 16) access the same bank, \( n \) accesses are executed serially
- **Broadcast:** \( n \) threads access the same word in one fetch
__global__ void per_thread_sum(...) 
{
    while(!done)
    {
        for(int j=indices[i];
            j<min(indices[i+1],indices[i]+MAX_ITER);
            j++)
        {
            ...
        }
    }
}