Quiz 1 – Question 1

- Check all of the following that is correct in the concept of a process and a thread.
  - Processes and threads are dynamic
  - Processes contain the static input and code data but also have a global heap
  - Threads only contain their local stack and registers – this makes them lightweight
  - Process are needed to keep separate address spaces
Quiz 1 – Question 1

- A process is considered lightweight since it only contains a local stack and registers
  √ This is correct for threads; not for processes

√ Two different processes cannot access each other's address space

√ Both processes and threads are dynamic

√ Threads are the units of execution in the multi-threaded processor
Quiz 1 – Question 2

- Two different types of locality:
  - Temporal locality: if an item is referenced, it will tend to be referenced again soon
  - Spatial locality: if an item is referenced, items whose addresses are close tend to be referenced soon
Quiz 1 – Question 2 extension

- An example for spatial may be
  - Transposing a matrix to access rows instead of columns
  - Purposely putting related items next to each other in a structure
  - Computing on small region of data before moving to another

- An example for temporal may be
  - Moving computation of the same data next to each other
  - Reusing a loaded value
  - Computing on small region of data before moving to another
Quiz 1 – Question 3

Explain what the SIMD unit is and what additions does it need in hardware compared to a single-threaded processor? [tip: remember the Von-Neumann diagram] Please indicate in which scenario you would prefer the SIMD units?

Single Instruction Multiple Data (SIMD)
Quiz 1 – Question 3

- SIMD are vector processing units they execute Single Instruction on Multiple Data
- SIMD units are an array of scalar ALUs along with a wider register file (data path)
- SIMD is better for vector processing, ALU may be better for control flow or small amounts of data; SIMD does take up more power!
- SIMD still executes a sequence of instructions in serial. Its just that a single instruction is now a vector instruction
### Quiz 1 – Question 4

<table>
<thead>
<tr>
<th></th>
<th>Scalar</th>
<th>Vector</th>
<th>Core</th>
<th>Card</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU Unit</td>
<td><img src="image1" alt="ALU" /></td>
<td><img src="image2" alt="ALU" /></td>
<td><img src="image3" alt="SIMD" /></td>
<td><img src="image4" alt="SM" /> <img src="image5" alt="SM" /></td>
</tr>
<tr>
<td>SIMD Unit</td>
<td><img src="image3" alt="SIMD" /></td>
<td><img src="image3" alt="SIMD" /></td>
<td><img src="image4" alt="SM" /> <img src="image5" alt="SM" /></td>
<td><img src="image6" alt="GPU" /></td>
</tr>
<tr>
<td><strong>Threads</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td><img src="image7" alt="Thread" /></td>
<td><img src="image8" alt="Warp" /></td>
<td><img src="image9" alt="Thread Block" /></td>
<td><img src="image10" alt="Block Grid" /></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register File</td>
<td><img src="image11" alt="Register File" /></td>
<td><img src="image12" alt="L1 Cache" /></td>
<td><img src="image13" alt="L2 / Memory" /></td>
<td></td>
</tr>
<tr>
<td><strong>Address Space</strong></td>
<td><img src="image14" alt="Local per thread" /></td>
<td><img src="image15" alt="Shared Memory" /></td>
<td><img src="image16" alt="Global" /></td>
<td></td>
</tr>
</tbody>
</table>
Quiz 1 – Question 4

Scalar processors have an ALU unit in which the unit of scheduling and execution is a thread; the next level is SIMD unit in which the unit of scheduling is a warp of threads. In the higher level, each core of GPU is called SM and it's unit of scheduling is a thread block. Finally, a block grid is scheduled on the whole GPU card which consists of multiple SM cores.
CUDA MEMORIES
Hardware View of CUDA Memories
Programmer View of CUDA Memories

Grid

Block (0, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Block (1, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Thread (1, 0)

Host

Global Memory

Constant Memory
Declaring CUDA Variables

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __shared__, or __constant__
- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory
Example:
Shared Memory Variable Declaration

```c
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
    __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];

    ...
}
```
Where to Declare Variables?

- Can host access it?
  - global constant
  - register shared

- Outside of any Function
- In the kernel
Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
  - One in each SM
  - Accessed at much higher speed (in both latency and throughput) than global memory
  - Scope of access and sharing - thread blocks
  - Lifetime – thread block, contents will disappear after the corresponding thread finishes execution
  - Accessed by memory load/store instructions
  - A form of scratchpad memory in computer architecture
Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory

[Diagram showing global memory access patterns between threads]
**Tiling/Blocking - Basic Idea**

Global Memory

Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time
Tiling/Blocking - Basic Idea

Global Memory

On-chip Memory

Thread 1

Thread 2
Key Takeaways

- Shared Memory is a programmer specified SM memory
- Located inside of an SM Core
- Threads within a single thread block have access to the same space
- Across thread blocks can not be accessed
- Static declaration through `__shared__ int var[numElements]`
- Dynamic declaration through `func<<gridDim,BlockDim,SharedMemSize>>>(args)`
Question

- Would you use shared memory if your data gets used once? Why or why not?
PARALLEL COMPUTATION
PATTERNS - REDUCTION
“Partition and Summarize”

- A commonly used strategy for processing large input data sets
  - There is no required order of processing elements in a data set (associative and commutative)
  - Partition the data set into smaller chunks
  - Have each thread to process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer
- E.G., Google and Hadoop MapReduce frameworks support this strategy
- We will focus on the reduction tree step for now
Reduction enables other techniques

- Reduction is also needed to clean up after some commonly used parallelizing transformations
- Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location (privatization)
  - Use a reduction tree to combine the values of private locations into the original output location
What is a reduction computation?

- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product

- Often used with a user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
  - For example, the user may supply a custom “max” function for 3D coordinate data sets where the magnitude for the each coordinate data tuple is the distance from the origin.
An Efficient Sequential Reduction O(N)

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- Iterate through the input and perform the reduction operation between the result value and the current input value
  - N reduction operations performed for N input values
  - Each input value is only visited once – an O(N) algorithm
  - This is a computationally efficient algorithm.
A parallel reduction tree algorithm performs N-1 operations in log(N) steps
Work Efficiency Analysis

- For N input values, the reduction tree performs
  - \((1/2)N + (1/4)N + (1/8)N + \ldots (1)N = (1 - (1/N))N = N - 1\) operations
  - In \(\log(N)\) steps – 1,000,000 input values take 20 steps
  - Assuming that we have enough execution resources
  - Average Parallelism \((N-1)/\log(N)\)
    - For \(N = 1,000,000\), average parallelism is 50,000
    - However, peak resource requirement is 500,000
    - This is not resource efficient

- This is a **work-efficient** parallel algorithm
  - The amount of work done is comparable to the an efficient sequential algorithm
  - Many parallel algorithms are not work efficient
BASIC REDUCTION KERNEL
Parallel Sum Reduction

- Parallel implementation
  - Each thread adds two values in each step
  - Recursively halve # of threads
  - Takes log(n) steps for n elements, requires n/2 threads
Parallel Sum Reduction

– Assume an in-place reduction using shared memory
  – The original vector is in device global memory
  – The shared memory is used to hold a partial sum vector
  – Initially, the partial sum vector is simply the original vector
  – Each step brings the partial sum vector closer to the sum
  – The final sum will be in element 0 of the partial sum vector
  – Reduces global memory traffic due to reading and writing partial sum values
  – Thread block size limits n to be less than or equal to 2,048
A Parallel Sum Reduction Example

```
<table>
<thead>
<tr>
<th>DATA</th>
<th>THREAD 0</th>
<th>THREAD 1</th>
<th>THREAD 2</th>
<th>THREAD 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Active Partial Sum elements
A Naive Thread to Data Mapping

- Each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
- After each step, half of the threads are no longer needed
- One of the inputs is always from the location of responsibility
- In each step, one of the inputs comes from an increasing distance away
A Simple Thread Block Design

- Each thread block takes 2*BlockDim.x input elements
- Each thread loads 2 elements into shared memory

```c
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start + blockDim.x+t];
```
The Reduction Steps

```c
for (unsigned int stride = 1;
    stride <= blockDim.x;  stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t]+= partialSum[2*t+stride];
}
```
Barrier Synchronization

- `__syncthreads()` is needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step
- `__syncthreads()` synchronizes all threads within the block
Back to the Global Picture

- At the end of the kernel, Thread 0 in each block writes the sum of the thread block in partialSum\[0\] into a vector indexed by the blockIdx.x.
- There can be a large number of such sums if the original vector is very large.
  - The host code may iterate and launch another kernel.
- If there are only a small number of sums, the host can simply transfer the data back and add them together.
- Alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.
Question

- What is some drawback to this implementation of reduction?
A BETTER REDUCTION MODEL
Some Observations on the naïve reduction kernel

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources
- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
  - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire
Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
- Keep the active threads consecutive
  - Always compact the partial sums into the front locations in the partialSum[ ] array
An Example of 4 threads

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Quick Analysis

- For a 1024 thread block
  - No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
    - The final 5 steps will still have divergence
Question
- Are there any drawbacks to this implementation?