Quiz 1
Quiz 1 – Question 1

Compare the differences between a thread and a process. What do both contain and how do they relate to one another? Why is a thread considered "lightweight"? And if so, assess the need for a process.

- 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
- Processes are still needed to keep separate address spaces
Quiz 1 – Question 2

What are temporal and spacial cache locality? How can a programmer take advantage of both? Demonstrate a case for both localities.

- This was the most misunderstood question
- Everyone got what temporal and spacial locality definitions
- Very few applied them
Question 2 examples

What are temporal and spacial cache locality? How can a programmer take advantage of both? Demonstrate a case for both localities.

• Loops are not an application of locality; they are a description of what locality is
• This is the programs behavior that the caches take advantage of
• Not how a programmer can take advantage of locality
• The following exhibit the same behavior

```java
a = 0;
for(int i = 0; i < 10; ++i){
    a += i;
}
```

```java
a = 0;
a+=1;
a+=2;
```
Question 2 examples

What are temporal and spacial cache locality? How can a programmer take advantage of both? Demonstrate a case for both localities.

• An example for spacial 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
What are temporal and spacial cache locality? How can a programmer take advantage of both? Demonstrate a case for both localities.

- What this question is asking is how does HW affect the way software is written
- Describing what locality is shows how SW affected HW design
- Looking for you to explain and create
How I’m organizing the class

- Lectures
- Quizzes
- Labs and Tests

Bloom’s Taxonomy:

- Remember
- Understand
- Apply
- Analyze
- Evaluate
- Create

- Recall facts and basic concepts
- Define, duplicate, list, memorize, repeat, state
- Explain ideas or concepts
- Classify, describe, discuss, explain, identify, locate, recognize, report, select, translate
- Use information in new situations
- Execute, implement, solve, use, demonstrate, interpret, operate, schedule, sketch
- Draw connections among ideas
- Differentiate, organize, relate, compare, contrast, distinguish, examine, experiment, question, test
- Justify a stand or decision
- Appraise, argue, defend, judge, select, support, value, critique, weigh
- Produce new or original work
- Design, assemble, construct, conjecture, develop, formulate, author, investigate
Quiz 1 – Question 3

Explain what a SIMD unit is and what additions does it need compared to a scalar ALU. Create a scenario in which you would prefer SIMD units, when would you prefer a scalar ALU?

• 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!
• Misconceptions
  • SIMD still executes a sequence of instructions in serial. Its just that a single instruction is now a vector instruction
  • SIMD instructions are the same complexity as ALU. They both do arithmetic
Quiz 1 – Question 4

Describe the hierarchy of execution units within a GPU and relate the unit of scheduling to each level of the hierarchy. Evaluate the hierarchy in terms of programmability, performance, use cases, general vs specialization, etc..

- Sorry for the poorly written question, but most people understood the question.

<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>ALU</td>
<td>ALU</td>
<td>SIMD</td>
<td>SM</td>
</tr>
<tr>
<td>ALU Unit</td>
<td>ALU</td>
<td>SIMD</td>
<td>SM</td>
<td>GPU</td>
</tr>
<tr>
<td><strong>Threads</strong></td>
<td>Thread</td>
<td>Warp</td>
<td>Thread Block</td>
<td>Block Grid</td>
</tr>
<tr>
<td>Memory</td>
<td>Register File</td>
<td>L1 Cache</td>
<td>L2 / Memory</td>
<td></td>
</tr>
<tr>
<td>Address Space</td>
<td>Local per thread</td>
<td>Shared Memory</td>
<td>Global</td>
<td></td>
</tr>
</tbody>
</table>
Quiz 1 – Question 4

Describe the hierarchy of execution units within a GPU and relate the unit of scheduling to each level of the hierarchy. Evaluate the hierarchy in terms of programmability, performance, use cases, general vs specialization, etc..

• Good evaluations of hierarchy
• Easier to program, as we only worry about thread blocks and grids
• Reduces hardware complexity and reduces power consumption
• Scalable, just add more SMs to get more performance
• Use cases for graphics and matrix multiplication map very well to this hardware
• Allows the GPU to be programmed generally and reduces specialization
Inclusive Scan (Prefix-Sum) Definition

**Definition:** The scan operation takes a binary associative operator \( \oplus \) (pronounced as circle plus), and an array of \( n \) elements 
\[
[x_0, x_1, \ldots, x_{n-1}],
\]
and returns the array
\[
[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].
\]

**Example:** If \( \oplus \) is addition, then scan operation on the array would return
\[
[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3], \quad [3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25].
\]
An Inclusive Scan Application Example

- Assume that we have a 100-inch sandwich to feed 10 people
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
  - How do we cut the sandwich quickly?
  - How much will be left?

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate prefix sum:
  - [3, 8, 10, 17, 45, 52, 52, 60, 61] (39 inches left)
Typical Applications of Scan

- Scan is a simple and useful parallel building block

- Convert recurrences from sequential:
  
  ```
  for(j=1; j<n; j++)
  out[j] = out[j-1] + f(j);
  ```

- Into parallel:
  
  ```
  forall(j) { temp[j] = f(j) };
  scan(out, temp);
  ```

- Useful for many parallel algorithms:
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms, ….
Other Applications

- Assigning camping spots
- Assigning Farmer’s Market spaces
- Allocating memory to parallel threads
- Allocating memory buffer space for communication channels
- ...
An Inclusive Sequential Addition Scan

Given a sequence \( [x_0, x_1, x_2, \ldots] \)

Calculate output \( [y_0, y_1, y_2, \ldots] \)

Such that

\[
\begin{align*}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2 \\
& \quad \vdots
\end{align*}
\]

Using a recursive definition

\[
y_i = y_{i-1} + x_i
\]
A Work Efficient C Implementation

\[
y[0] = x[0];
\]

\[
\text{for } (i = 1; i < \text{Max}_i; i++) \ y[i] = y[i-1] + x[i];
\]

Computationally efficient:

N additions needed for N elements - O(N)!
Only slightly more expensive than sequential reduction.
A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

\[
\begin{align*}
y_0 &= x_0 \\
y_1 &= x_0 + x_1 \\
y_2 &= x_0 + x_1 + x_2
\end{align*}
\]

“Parallel programming is easy as long as you do not care about performance.”
Why is this a naive implementation? How can we make it better?
A Better Parallel Scan Algorithm

1. Read input from device global memory to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration

- Active threads \(\text{stride} = n-1\) (n-stride threads)
- Thread \(j\) adds elements \(j\) and \(j\)-\(\text{stride}\) from shared memory and writes result into element \(j\) in shared memory
- Requires barrier synchronization, once before read and once before write
A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration.
A Better Parallel Scan Algorithm

1. Read input from device to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory
Handling Dependencies

- During every iteration, each thread can overwrite the input of another thread
  - Barrier synchronization to ensure all inputs have been properly generated
  - All threads secure input operand that can be overwritten by another thread
  - Barrier synchronization is required to ensure that all threads have secured their inputs
  - All threads perform addition and write output
A Work-Inefficient Scan Kernel

```c
__global__ void work_inefficient_scan_kernel(float *X, float *Y, int InputSize) {
    __shared__ float XY[SECTION_SIZE];
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < InputSize) {XY[threadIdx.x] = X[i];}
    // the code below performs iterative scan on XY
    for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2) {
        __syncthreads();
        float in1 = XY[threadIdx.x + stride];
        __syncthreads();
        XY[threadIdx.x] += in1;
    }
    __syncthreads();
    if (i < InputSize) {Y[i] = XY[threadIdx.x];}
}
```
Work Efficiency Considerations

- This Scan executes $\log(n)$ parallel iterations
  - The iterations do $(n-1), (n-2), (n-4), \ldots (n - n/2)$ adds each
  - Total adds: $n \times \log(n) - (n-1) \rightarrow O(n \times \log(n))$ work

- This scan algorithm is not work efficient
  - Sequential scan algorithm does $n$ adds
    - A factor of $\log(n)$ can hurt: 10x for 1024 elements!

- A parallel algorithm can be slower than a sequential one when execution resources are saturated from low work efficiency
Improving Efficiency

- **Balanced Trees**
  - Form a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- **For scan:**
  - Traverse down from leaves to the root building partial sums at internal nodes in the tree
    - The root holds the sum of all leaves
  - Traverse back up the tree building the output from the partial sums
Parallel Scan - Reduction Phase

Time

\( x_0 \) \( \sum_{x_0} x_1 \) \( x_1 \) \( \sum_{x_2} x_3 \) \( x_2 \) \( \sum_{x_4} x_5 \) \( x_3 \) \( \sum_{x_4} x_5 \) \( x_4 \) \( \sum_{x_6} x_7 \) \( x_5 \) \( \sum_{x_6} x_7 \) \( x_6 \) \( \sum_{x_6} x_7 \) \( x_7 \) \( \sum_{x_0} x_7 \)

In-place calculation

Value after reduce
Reduction Phase Kernel Code

// XY[2*BLOCK_SIZE] is in shared memory
for (unsigned int stride = 1; stride <= BLOCK_SIZE; stride *= 2) {
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
    __syncthreads();
}

threadIdx.x+1 = 1, 2, 3, 4....
istride = 1,
    index = 1, 3, 5, 7, ...
Parallel Scan - Post Reduction Reverse Phase

Move (add) a critical value to a central location where it is needed
Parallel Scan - Post Reduction Reverse Phase

\[ x_0 \quad \sum_{0..1} x_1 \quad x_2 \quad \sum_{0..3} x_3 \quad x_4 \quad \sum_{4..5} x_5 \quad x_6 \quad \sum_{0..7} x_7 \]

\[ \sum_{0..2} x_0 \cdot x_2 \quad \sum_{0..4} x_0 \cdot x_4 \quad \sum_{0..6} x_0 \cdot x_6 \]
Putting it Together
Post Reduction Reverse Phase Kernel Code

for (unsigned int stride = BLOCK_SIZE/2; stride > 0; stride /= 2) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index+stride < 2*BLOCK_SIZE) {
        XY[index + stride] += XY[index];
    }
}
__syncthreads();
if (i < InputSize) Y[i] = XY[threadIdx.x];

First iteration for 16-element section
threadIdx.x = 0
stride = BLOCK_SIZE/2 = 8/2 = 4
index = 8-1 = 7
Work Analysis of the Work Efficient Kernel

- The work efficient kernel executes $\log(n)$ parallel iterations in the reduction step
  - The iterations do $n/2, n/4, ... 1$ adds
  - Total adds: $(n-1) \rightarrow O(n)$ work

- It executes $\log(n) - 1$ parallel iterations in the post-reduction reverse step
  - The iterations do $2^{-1}, 4^{-1}, ..., n/2^{-1}$ adds
  - Total adds: $(n-2) - (\log(n) - 1) \rightarrow O(n)$ work

- Both phases perform up to no more than $2 \times (n-1)$ adds

- The total number of adds is no more than twice of that done in the efficient sequential algorithm
  - The benefit of parallelism can easily overcome the $2X$ work when there is sufficient hardware
Some Tradeoffs

- The work efficient scan kernel is normally more desirable
  - Better Energy efficiency
  - Less execution resource requirement
- However, the work inefficient kernel could be better for absolute performance due to its single-phase nature (forward phase only)
  - There is sufficient execution resource
Handling Large Input Vectors

- Build on the work efficient scan kernel
- Have each section of $2^{*}$blockDim.x elements assigned to a block
  - Perform parallel scan on each section
- Have each block write the sum of its section into a Sum[] array indexed by blockIdx.x
- Run the scan kernel on the Sum[] array
- Add the scanned Sum[] array values to all the elements of corresponding sections
- Adaptation of work inefficient kernel is similar.
Overall Flow of Complete Scan

Initial Array of Arbitrary Values

Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3

Store Block Sum to Auxiliary Array

Scan Block Sums

Add Scanned Block Sum \(i\) to All Values of Scanned Block \(i + 1\)

Final Array of Scanned Values
**Exclusive Scan Definition**

**Definition:** The exclusive scan operation takes a binary associative operator $\oplus$, and an array of $n$ elements

$$[x_0, x_1, \ldots, x_{n-1}]$$

and returns the array

$$[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})].$$

**Example:** If $\oplus$ is addition, then the exclusive scan operation on the array [3 1 7 0 4 1 6 3], would return [0 3 4 11 11 15 16 22].
Why Use Exclusive Scan?

- To find the beginning address of allocated buffers

- Inclusive and exclusive scans can be easily derived from each other; it is a matter of convenience

\[
\begin{align*}
&\text{Exclusive} \quad [0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22] \\
&\text{Inclusive} \quad [3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22 \ 25]
\end{align*}
\]
A Simple Exclusive Scan Kernel

- Adapt an inclusive, work inefficient scan kernel
- Block 0:
  - Thread 0 loads 0 into XY[0]
  - Other threads load X[threadIdx.x-1] into XY[threadIdx.x]
- All other blocks:
  - All thread load X[blockIdx.x*blockDim.x+threadIdx.x-1] into XY[threadIdx.x]
- Similar adaption for work efficient scan kernel but ensure that each thread loads two elements
  - Only one zero should be loaded
  - All elements should be shifted to the right by only one position