**COP6616 Parallel Computing**

**Instructor: Scott Piersall**

**Fall 2024**

**Assignment 3**

1. **Short Answer Questions (14 points).**

1). What is the difference between data spatial locality and temporal locality?

**Answer:**

Spatial data locality refers to the prefetching into the cache of instructions that are stored nearby to the recently executed instruction because we assume they have a high chance of execution. Sequential data access patterns benefit from spatial locality while complex access patterns may find it costly, especially if preloaded cached data are never used, resulting in an increased number of cache misses. On the other hand, temporal locality focuses on caching data that is used most frequently, which is good for repetitive operations where the same data and instruction are used repeatedly by an application. However, a disadvantage is that filling the cache with the presumption of temporal locality may lead to the premature removal of important data from the cache, especially when data is used sparingly.

----

2). In the MPI Gather routine, does the root contribute data?

**Answer:**

Yes, each process, including the root sends the contents of its send buffer to the root process. Afterwards the root process receives the messages and stores them in rank order.

----

3). For the following MPI code, decide whether there is/are any potential problem(s) with it. If there are problems, please fix them.

if (rank =0){

MPI\_Barrier(…);

} else {

do something;

}

**Answer:**

if (rank == 0) {

// do something

}

else {

// do something else

}

MPI\_Barrier(MPI\_COMM\_WORLD); // Move barrier here so all processes call it

Explanation: MPI\_Barrier() is a collective operation that will only complete once invoked by all the MPI processes. No processes can exit the barrier until all processes have entered it, if some processes don’t reach it, it will hang forever.

4). In CUDA programs, if we compare “structure of arrays” and “array of structures”, which one is offers better memory access performance? WHY?

**Answer:**

This is no straightforward best response to this question, as the optimal solution depends on the access pattern of the algorithm. For example, if you are processing the R,G, and B components together of an array of RGB pixels, then AOS (array of structures) would be preferable as the elements of each structure would be contiguous in memory allowing for better cache utilization. This would also facilitate contiguous access by threads where for example, thread 0 accesses memory address N, thread 1 accesses memory address N, + 1 thread 2 should access N+2, and so on. Contiguous access is required by coalescing which refers to combining multiply memory accesses by threads into a single memory transaction, which greatly improves bandwidth. If you still wanted to operate on a series of RGB pixels, but you only wanted to process all of the R components separately, structure of arrays would suit you better as all the r components will be contiguous in memory, allowing for coalescing.

5). Discuss the difference between the following GPU memory: global, local, shared, texture, and constant. Which one is the fastest?

**Answer:**

Here is a brief description of each of the above mentioned above, starting from the slowest, all the way to the fastest.

**Slow Memory:**

Global Memory :

* Largest memory pool in the GPU and can be accessed by all threads across all blocks.
* Is relatively slow and resides in off-chip DRAM memory.
* Typically used for things like input / output data and global constants.
* Requires optimization to avoid performance degradation (careful access patterns such as coalescing)

Local memory:

* Is about as slow as global memory
* Memory that is private to each thread and lasts the lifetime of that thread
* Like global memory, it resides off-chip in device RAM
* Local memory is cached
* Used for storing temporary or automatic variables
* Used when register space is insufficient (register spilling)

----

**Moderate Speed Memory:**

Texture Memory:

* Resides off-chip and is cached
* Read only memory on the device that can be faster than global memory when all reads in a warp are physically adjacent.
* All threads across all blocks can access these memory types
* It is a special type of memory optimized for textures.
* Optimized for 2D / 3D spatial locality, images being an example.

Constant Memory:

* Resides off-chip and is cached
* It is a special type of memory optimized for constant memory
* Used for read-only data shared across threads
* Optimized for broadcast access where all threads in a warp read the same memory location.

**Fast Memory:**

Shared Memory:

* Smaller but fast memory pool
* Resides on chip
* Data is visible to all threads within that block and lasts for the duration of the block.
* Allows for threads to communicate and share data between one another playing a key role in inter-thread communication
* Divided into banks and susceptible to conflicts when threads try to access the same bank at the same time.

----

6). If shared memory in a GPU is defined and used in the following way, under what scenario about the variable “s”, is there a bank conflict? Under what scenario is there no bank conflict? Why?

\_\_shared\_\_ float shared[16];

float foo = shared[baseIndex + s \* threadIdx.x];

**Answer:**

Shared memory is divided into equally sized memory banks that can be accessed simultaneously. Consecutive 32-bit words are mapped to consecutive banks, where each bank can service one memory request per clock cycle. Modern GPUs typically have 32 banks, so bank 0 will store word 0, bank 1 will store word 1, etc. Bank 0’s next word will be word 32. If s is 1 and assuming a base index of 0, the consecutive threads will map directly to the banks, thread 0 to bank 0, thread 1 to bank 1, etc. However, if s is 32, a multiple of the number of banks, you will have terrible bank conflicts because thread 1 will try to access word 32, thread 2 will try to access word 64. These are all words in bank 0, and the bank can only service one of these requests at a time, resulting in serialized memory access and reduced performance! To avoid this issue, we should take care to use access patterns that result in consecutive threads accessing consecutive memory locations.

7). If shared memory in GPU is defined and used in the following way, what kind of bank conflict (such as 2-way, 4-way, 8-way) does it have? Why?

\_\_shared\_\_ **char** shared[];

foo = shared[baseIndex + threadIdx.x];

**Answer:**

For modern GPUs, each bank handles 32-bit or 4 byte words and each character is 1 byte. This means that 4 consecutive chars fit into one 32-bit word which will be stored in a bank. Now consider a scenario where shared[] is an array with 4 characters and you have four threads. These four threads will access four consecutive characters or bytes of the same word, hence they all are accessing the same word, and thus the same bank! This means that we have a 4-way bank conflict. Luckily this doesn’t cause a performance penalty for GPUs devices with a Compute Capability of 2.0+. This is because when multiple threads of the same warp access any address within the same 32-bit word, even though the addresses fall in the same bank, in the case of read access as in the code excerpt above, modern GPUs broadcast the word to all requesting threads simultaneously.

1. **(15 points)** Please write **a CUDA program** to compute the Euclidean distance, similar to the problem in assignment 2.

You may design your code using the following steps:

* Declare the arrays (host and device). All arrays should be dynamically allocated; the host arrays can be allocated either with malloc or new, while the device arrays should be allocated with cudaMalloc.
* Print the number of CUDA-enabled hardware devices attached to the system by calling cudaGetDeviceCount.
* Print at least 3 interesting properties of Device 0, including the device name, by calling cudaGetDeviceProperties. The first argument to this function is a pointer to a struct of type cudaDeviceProp.
* Calls InitArray to initialize the host arrays. InitArray initializes an integer array with random numbers within a fairly small range (0 to 99).
* Calls cudaMemcpy to copy the host input arrays to the device.
* Calls the CUDA kernel, which computes the square of the difference of the components for each dimension, reduce all the elements of the output array in parallel (you may need to investigate how to implement an efficient parallel reduce in CUDA), and takes the square root of the sum.

Run experiments using varying size of inputs. Graph and discuss the speedup provided by the GPU/CUDA implementation over varying input sizes. Is there an input size where the speedup stops?

1. **(10 points)** You are working on team that is building an edge detection module. The first step in edge detection is to remove color information and work directly in black-and-white. You do not need to do edge detection, just the first step: removing the color information from images. So, please write **a CUDA program** to convert a color image to grayscale using the Colorimetric method. I suggest that you use PNG or BMP images as input. I want to see the original images that you tested with and the output/resulting image. You may do this assignment in C or Python, and are free to create a Jupyter notebook. If you do a Jupyter notebook, please include the notebook in your assignment report. How does using a GPU for this vs. a serial CPU-based implementation perform? Is there a speedup? How (be specific) is the GPU architecture well-suited to this task?
2. **(11 points)** Write a matrix multiplication program that uses GPU/CUDA. Your matrices should consist of floating-point values (NOT INTEGERS). You may do this assignment in C, Python, and are free to create a Jupyter notebook. If you do a Jupyter notebook, please include the notebook in your assignment report. You should run MANY experiments with varying sized matrices. What is the speedup? Is the speedup affected by matrix size? Explain (diagrams are useful here) how the GPU implementation of this code differs from a serial (CPU-based) implementation.

**What to submit:**

1. Answer all questions and provide any code inline. Do not paste screenshots of code, post your source code, I want to see graphs/visualizations of all of your experimental results results. Crete a PDF and upload the PDF to the assignment specification in canvas.