COP6616 Parallel Computing
Instructor: Scott Piersall
Student: Aaron Goldstein
Fall 2024
Assignment 3

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

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

   Answer:

Data spatial locality refers to the tendency of a program to access data locations that are close to each other in memory within a short time period. This means that if a particular data item is accessed, nearby data items (e.g., the next elements in an array) are likely to be accessed soon. Temporal locality, on the other hand, refers to the tendency of a program to access the same data location multiple times within a relatively short time frame. This is common in scenarios like loops where the same variables are accessed repeatedly. Both types of locality are leveraged to optimize cache performance: spatial locality through fetching contiguous blocks of data (cache lines) and temporal locality through keeping recently accessed data in the cache for quick reuse.-
---

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:

Although a Structure of Arrays (SoA) generally offers better memory access performance than an Array of Structures (AoS), 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 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 all 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, bank 31 will store word 31, 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 or banks.

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.

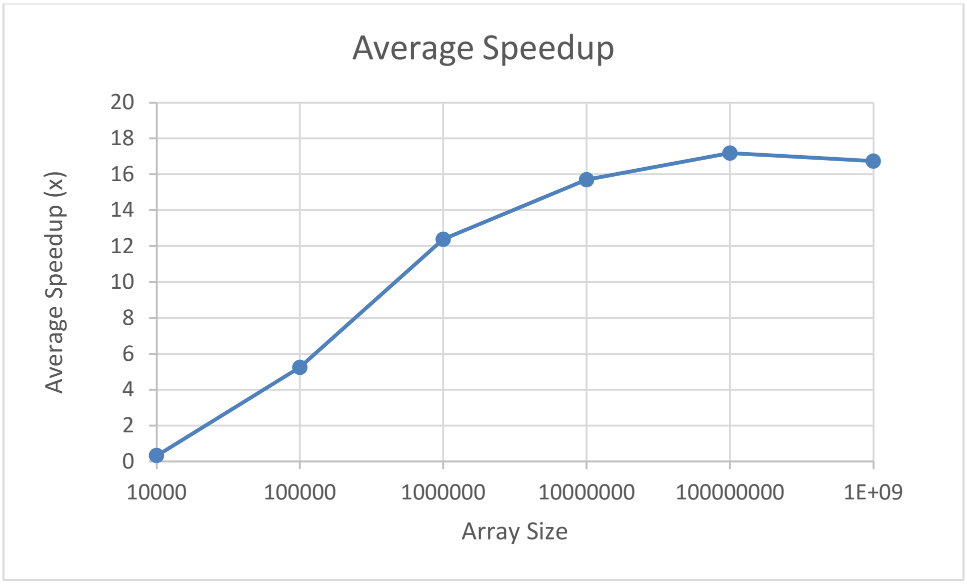
2. **(15 points)** Please write **a CUDA program** to compute the Euclidean distance, similar to the problem in assignment 2.

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?

  Answer:



Starting at an array size of 10,000, CUDA is slower than serial execution (about 0.33 times as fast). This tells us the problem size of 10^4 does not justify the overhead of parallelization (launching kernels, moving data between devices, and synchronizing threads, etc). However, if we move up to 100,000, we see CUDA jump up to about 5.25 times as fast as serial execution. We see significant gains all the way up to 10^7 at 15.71 times speedup where it then starts to level off and the speedup starts to become consistent, only growing slightly to 17.19 times speedup at 10^8, then dropping down to 16.74 times speedup at 10^9. This is likely because eventually the maximum number of simultaneous threads that can be executed at once are being executed around the 10^7 mark, and the GPU is being fully utilized, thus increasing the problem size will not provide any more significant speedup aside from random fluctuations and chance. It appears for this GPU, NVIDIA RTX 4090, 10^7 is an optimal minimum problem size to take advantage of the maximum speedup provided by CUDA parallelization.

3. **(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?

Answer:

The GPU implementation shows speedup over the CPU implementation ranging from 1.33x to 2.50x for the image I tested, although the speedup may not be too impressive for small pictures due to them not fully utilizing GPU parallelism and overhead. Overhead in this case would be launching the kernel and memory transfer between devices. This problem is well suited for parallelism because each pixel conversion is independent, and the conversion formula is simple arithmetic. Each thread processes one pixel, while blocks of 32 * 32 threads work in parallel. The total number of blocks is equal to your grid x dimension * grid y dimension however, the actual number of threads that can be run in parallel is limited to your GPU hardware. The RGB values are contiguous in memory and accessed consecutively by warps, allowing for coalescence, combining thread memory access into a single transaction, providing improved performance. The above-mentioned factors make the GPU architecture well suited to this task. I have attached an example input to my program, as well as two example outputs for the serial and parallel implementations.

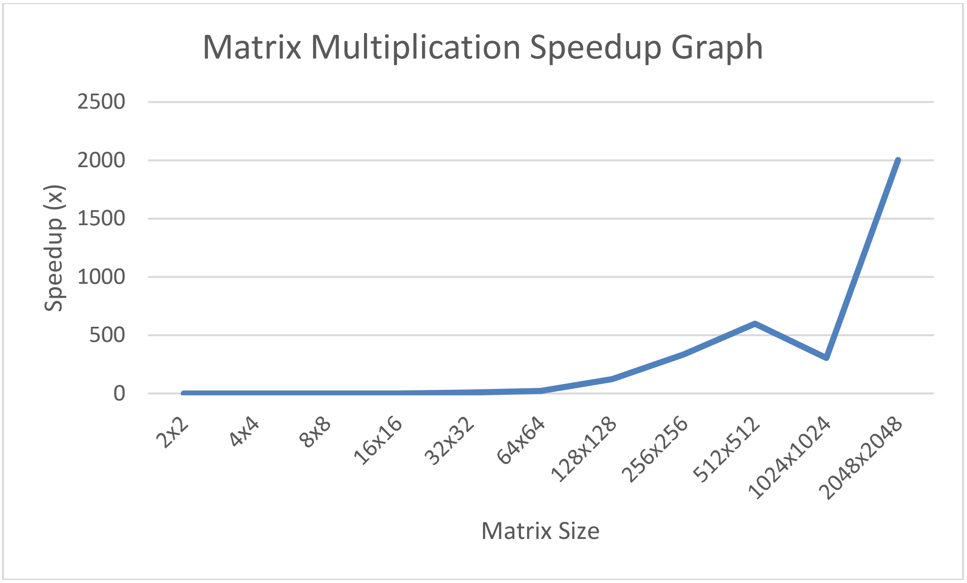
Before Conversion:

After Conversion (CPU - OpenCV library)

After Conversion (GPU):

4. **(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.

Answer:



Looking at the above graph showcasing my speedup results, we can see that I tested the speed of my CPU and GPU implementations for matrix multiplication with several powers of 2 from 2^1 to 2^11. 2^12 was chosen as the cutoff point because it simply took long to execute to be feasible for this experiment. Up until but not including matrix sizes 2^5, GPU implementation is slower due to kernel launching overhead. For a 32x32 matrix sizes, the speedup jumps to 6.73x. Next we see a relatively moderate increase to 21.59x speedup in a 64x64 matrix. After, the speedup skyrockets to 123.41x for a 256x256 matrix. The pattern increases up to and including 512x12 matrices where it reaches 598.91x speedup, then it drops at 1024x1024 to 305.23x speedup. When we move up to 2048x2048 matrices, the speedup increases again to a staggering 2002.61x speedup! It appears that this pattern will only continue as we move to larger and larger matrix sizes. This shows that the speedup is absolutely affected by matrix size. Both the GPU and CPU implementation are similar in that they calculate the value of each element in the result matrix as the dot product of that element's corresponding row and column in the A and B input matrices respectively, the key difference is that the CPU implementation calculates each element one by one in row major order while the GPU implementation calculates each element in parallel where each thread is responsible for a specific element. Let's look at a simple example of how the CPU execution works with a simple 2x2 matrix then move to a more complicated example with a 4x4 matrix using the GPU implementation to show how we use global thread indexing with multiple grid dimensions and multiple block dimensions to have each thread calculate an element in the resulting 4x4 matrix. Take a look at the 2x2 matrix below.

Matrix A:
| A[0,0] A[0,1] |
| A[1,0] A[1,1] |
Matrix B:
| B[0,0] B[0,1] |
| B[1,0] B[1,1] |
Result Matrix C (calculated serially):
| C[0,0] C[0,1] |
| C[1,0] C[1,1] |

CPU Algorithm Steps:

The CPU algorithm will calculate result matrix C in the following order, one by one.

CPU Algorithm Steps:

1. Calculate C[0,0] = A[0,0] * B[0,0] + A[0,1] * B[1,0]
2. Calculate C[0,1] = A[0,0] * B[0,1] + A[0,1] * B[1,1]
3. Calculate C[1,0] = A[1,0] * B[0,0] + A[1,1] * B[1,0]
4. Calculate C[1,1] = A[1,0] * B[0,1] + A[1,1] * B[1,1]

This is done using a triple nested for loop, the outer two, ensure we traverse through element in the result matrix, and the inner ensures we multiply each corresponding element in that element's corresponding row and column in the A and B matrices respectively, add their products, and store the result in the correct location in the result matrix. This has a time complexity of O(N^3) and can be very time consuming on the CPU when the matrices get large, although it not too bad when we are dealing with small matrices like this.

Now, let's look at the GPU algorithm for a 4x4 matrix

| A[0,0] A[0,1] A[0,2] A[0,3] |
| A[1,0] A[1,1] A[1,2] A[1,3] |
| A[2,0] A[2,1] A[2,2] A[2,3] |
| A[3,0] A[3,1] A[3,2] A[3,3] |
Matrix B:
| B[0,0] B[0,1] B[0,2] B[0,3] |
| B[1,0] B[1,1] B[1,2] B[1,3] |
| B[2,0] B[2,1] B[2,2] B[2,3] |
| B[3,0] B[3,1] B[3,2] B[3,3] |
Result Matrix C:
| C[0,0] C[0,1] C[0,2] C[0,3] |
| C[1,0] C[1,1] C[1,2] C[1,3] |
| C[2,0] C[2,1] C[2,2] C[2,3] |
| C[3,0] C[3,1] C[3,2] C[3,3] |

Thread Indexing in CUDA

We'll use 2x2 blocks for this example, which means we'll have a 2x2 grid of blocks. Each block will contain 2x2 threads. Note: This block size is just for simplicity to split a small matrix across multiple blocks to see how indexing the elements with threads works. Our actual block sizes will be much bigger and a matrix of this size (4x4) would be handled by one block. For my actual implementation I used block sizes of 32x32 for a total 1024 threads per block.

We determined the number of blocks in the x dimension by dividing the matrix width (4), and then dividing by the number of threads in the block x dimension (2). 4 / 2 tells us we need 2 blocks in the grid's x dimension.

We determined the number of blocks in the grid y dimension a similar way, by dividing the matrix height (4) by the number of threads in the block y dimension (2). 4 / 2 tells us we need 2 blocks in the grid's y dimension. This tells us we need a 2x2 grid of blocks as you can see below in the table showing how blocks and their threads' corresponding global indices are organized. The global 2D index for each element is calculated with the below 2 formulas.

Here's how the blocks and threads are organized:

```
Block (0,0)    Block (0,1)
[0,0] [0,1]   [0,2] [0,3]
[1,0] [1,1]   [1,2] [1,3]

Block (1,0)    Block (1,1)
[2,0] [2,1]   [2,2] [2,3]
[3,0] [3,1]   [3,2] [3,3]
```

The global 2D index for each element is calculated with the below 2 formulas.

Row = blockIdx.y * blockDim.y + threadIdx.y

Column = blockIdx.x * blockDim.x + threadIdx.x

Now, let's look at how thread indexing works for a few threads to confirm the table is correct:

1. Thread in top-left corner (Block (0,0), Thread (0,0)):
   - blockIdx.x = 0, blockIdx.y = 0
   - threadIdx.x = 0, threadIdx.y = 0
   - row = blockIdx.y * blockDim.y + threadIdx.y = 0 * 2 + 0 = 0
   - col = blockIdx.x * blockDim.x + threadIdx.x = 0 * 2 + 0 = 0 This thread calculates C[0,0]
2. Thread in bottom-right corner of first block (Block (0,0), Thread (1,1)):
   - blockIdx.x = 0, blockIdx.y = 0
   - threadIdx.x = 1, threadIdx.y = 1
   - row = blockIdx.y * blockDim.y + threadIdx.y = 0 * 2 + 1 = 1
   - col = blockIdx.x * blockDim.x + threadIdx.x = 0 * 2 + 1 = 1 This thread calculates C[1,1]

3. Thread in the top-left of the second block (Block (0,1), Thread (0,0)):
    - blockIdx.x = 1, blockIdx.y = 0
    - threadIdx.x = 0, threadIdx.y = 0
    - row = blockIdx.y * blockDim.y + threadIdx.y = 0 * 2 + 0 = 0
    - col = blockIdx.x * blockDim.x + threadIdx.x = 1 * 2 + 0 = 2 This thread calculates C[0,2]
4. Thread in the bottom-right corner of the last block (Block (1,1), Thread (1,1)):
    - blockIdx.x = 1, blockIdx.y = 1
    - threadIdx.x = 1, threadIdx.y = 1
    - row = blockIdx.y * blockDim.y + threadIdx.y = 1 * 2 + 1 = 3
    - col = blockIdx.x * blockDim.x + threadIdx.x = 1 * 2 + 1 = 3 This thread calculates C[3,3]

Sometimes the matrix width will not be evenly divisible by the block x dimension, the same goes for the matrix height and the block y dimension. In this case we round up the quotient to the nearest integer which will increase the x or y dimension of the grid by 1. In my program, the width and height are the same, as well as the x and y dimensions of my blocks, so both would need to be rounded up. This also means that some global thread indexes won't be valid if this happens which is why we have a check in the GPU kernel to make sure the element at resultMatrix[row][column] is a valid element and avoid out of bounds errors before calculating that element's value as the dot product of the corresponding row and column of the A and B input matrices respectively.

```
if (row < size && col < size)
{
    float sum = 0.0f;
    for (int k = 0; k < size; k++)
    {
        sum += A[row * size + k] * B[k * size + col];
    }
    C[row * size + col] = sum;
}
```

The global thread indexing essentially replaces the two outer loops of the sequential implementation while the inner loop remains the same, calculating a dot product of a row and column in the input matrices. The combination of the formula to determine the number of blocks based on the block dimensions and problem size, and the formulas to assign a thread's row and column based on the block index, block dimension, and thread index ensures that every element in the result matrix has a corresponding thread assigned to calculate it. You can see that when I show how the blocks and threads were organized in the previous table, each thread's two-dimensional global index corresponded to an element in the result matrix. These threads execute in parallel, meaning multiple result elements are being calculated at once where only one was being computed at a time in the sequential execution.

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.