Ryan A. Margraf

WES 237B

Assignment 4

9/5/2022

Lab Part 1

Example 1 (ex1.cu)

This program prints the index corresponding to each block/thread using 2 blocks of 2 threads each.

Example 2 (ex2.cu)

This program prints the index corresponding to each thread using 1 block of 4 threads. The threads are arranged in 2 dimensions. It accomplishes something similar, but provides an alternative way to arrange the the threads which may be preferred depending on the use case.

Matrix Multiply (lw)

The first lw allocates memory on the device by creating first creating pointers then copying it from the host. On the other hand, lw_managed directly allocates memory in the unified memory shared between the host and device. This is likely slower memory, but it removes the need to copy the inputs/results to/from the host.

Lab Part 2

Performance Comparison

The submitted version of the code runs each of conversion to grayscale, inversion, and blur algorithms. For this performance comparison, unnecessary code was commented out. For example, the blur is not calculated for the grayscale or inversion tests, and the inversion is not done for the grayscale or blur tests. Additionally, only the final output image was shown. Since grayscale conversion was necessary for the specified inversion and blur algorithms, it was included in those tests.

These tests were done with the default size of 768x768. The timing was estimated by looking at the terminal outputs produced by the provided timing method.

	Convert to Grayscale	Invert	Blur (size 5)
CPU	0.0045s	0.0048s	0.028s
GPU	0.0031s	0.0040s	0.007s
GPU w/ Unified Memory	0.0029s	0.0030s	0.004s

Using the GPU with unified memory was the fastest for all 3 algorithms.

Assignment: Sobel Filter

Performance Comparison

These tests were done with the "imshow" command commented out. The timing was estimated by looking at the terminal outputs produced by the provided timing method.

Image size	**512x512**	**1024x1024**	**1536x1536**	**2048x2048**	**3072x3072**	**4096x4096**
OpenCV	7.9ms	29ms	61ms	107ms	236ms	436ms
CPU	4.8ms	18.5ms	41ms	73ms	161ms	283ms
GPU	1.6ms	6ms	14ms	13ms	21ms	21ms

The GPU showed a significant improvement over the other methods, showing a 3x to over 10x speedup over the CPU depending on the size. I was a bit surprised OpenCV was the slowest, but it is possible that its matrix overhead seen in Assignment 2 is a contributing factor.

For the larger sizes, the time taken did not increase as fast as expected. It is possible this has something to do with the block and thread layout. For example, my implementation defaults to blocks of 1024 threads, so for the 1536 size, some blocks had half of their threads go unused. This also likely has implications on the cache efficiency.

Assignment: Block Matrix Multiplication

Best Block Size

The table below shows the GPU execution times on a 1024x1024 matrix for various block sizes.

To find these, the program was run 5 times for each size and the median GPU execution time and its corresponding speedup were recorded.

Block Size	**4x4**	**8x8**	**16x16**	**32x32**
Time	255.64ms	52.59ms	44.04ms	47.91ms
Speedup	5.78x	28.78x	35.03x	32.56x

The block size of 16x16 threads appeared to be the fastest. This matches the 256 cores on the Jetson's GPU, which may be a contributing factor to its efficiency. Smaller block sizes require more non-interacting blocks to run, which makes sense that it would be less efficient than all of the threads working together. Larger block sizes do not have enough cores in this case for all the threads, so there is going to be some overhead in switching out the threads while the block executes.

Speedup

The following table shows the speedup compared to CPU with various sizes of the first matrix (NxM) with a block size of 16. To find these, the program was run 5 times for each size and the median speedup was recorded.

Matrix A Size	**16x16**	**16x1024**	**1024x16**	**64x64**	**128x512**	**768x256**	**1024x1024**
Speedup	0.52x	1.18x	3.59x	1.34x	5.39x	9.20x	22.18x

This shows a general trend of increasing speedup for larger matrices. The 16x16 case was actually slower on the GPU, but this is not surprising since it is not actually splitting the work into smaller blocks and it has some extra memory operations.

It is notable that the speedup for the 1024x16 matrix was significantly more than the 16x1024 one. This is likely because its output matrix size was 1024x1024 rather than 16x16. The latter, however, was still able to be faster than the CPU even though it wasn't for the 16x16 input size. This is due to the extra blocks in the other dimension that needed to be calculated.