Kobe Davis
Prof. Karavanic
CS 405
15 June 2019

Comparing 2D Convolution Performance

Introduction

The goal of this project is to measure performance and scaling behavior of four different Gaussian blurring implementations and compare them. The Gaussian blur implementations I will be measuring and comparing are single-pass 2D convolution algorithms:

1. Sequential
2. OpenACC
3. Naive CUDA
4. Shared Memory CUDA

I expected the the shared memory CUDA implementation to perform the best out of all implementations as well as how it scales with input matrix sizes. I expected for the sequential implementation to perform the worst and for the OpenACC implementation to perform somewhere close to the naive CUDA implementation. For this project I analyzed the amount of time spent copying data to and from the device combined with the computation time of the kernel as a function of the input matrix dimension. In other words, I measured how fast each of these implementations ran over varying input matrix sizes.

2D Convolution

Convolution is the process of adding each element of the image to its local neighbors, weighted by the mask. This project focuses on Gaussian blurring specifically, but the way in which a Gaussian blur is applied to an image is through 2D convolution. For this project 3 different parallel implementations of a 2D convolution algorithm were written and one sequential implementation of the 2D convolution algorithm was written. A short recap of 2D convolution will be given in this section so that we can lead into the results and implications following this section more naturally.

Beginning with an image, we convert the image into a large matrix of decimal values. These decimal values represent the intensity of the pixels. Next, we create a “mask” or “kernel” which is usually a much smaller 3x3 - 9x9 matrix. Then we “convolve” the image matrix with the

mask matrix; we overlay the mask matrix on the image matrix for every element in the image matrix, each time multiplying the mask elements by their overlaid partners in the image matrix and summing all products to form the new center element. Finally we normalize the values in the newly formed output matrix.

There are there are a few different algorithms for 2D convolution, though the main two have to do with whether or not the mask is “separable”. When a particular mask is separable then a two-pass algorithm is generally used, otherwise a single-pass algorithm is the easiest to implement. The single pass algorithm is what was described above and is what is used in this project for all parallel implementations. The two-pass algorithm is possible when the mask used is separable and is much faster than the single-pass algorithm. To implement it we decompose the mask matrix into two 1D factor matrices. Then we convolve the image with the both matrices (one horizontally and one vertically) where the order in which this happens does not matter and the output is equivalent to that in the single-pass. Lastly a mask is separable iff all rows are multiples of each other.

Execution Environment

The execution environment used for both running and measuring these four implementations was synchrotron.cs.pdx.edu. This system was used because it had the correct version of CUDA and contained a NVIDIA device capable of running OpenACC and CUDA programs.

Methodology

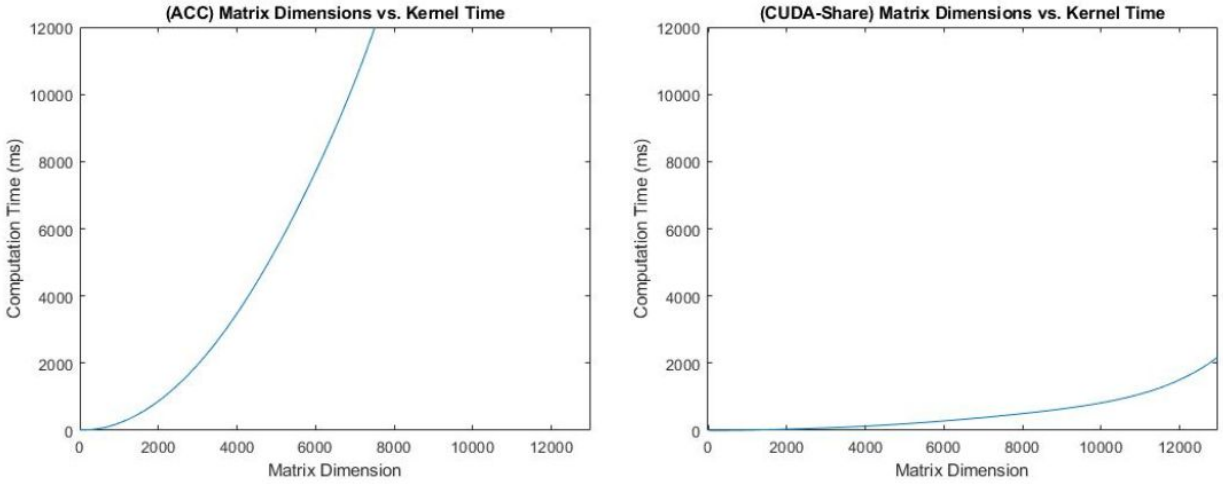
The process for creating these four implementations began with the naive CUDA implementations followed by the shared memory CUDA implementation. The sequential and OpenACC implementations came together last because the OpenACC version only needed a slightly modified sequential program to use of the pragma. For all implementations besides the sequential the outer loop was the sole focus of parallelization. The inner loops where the products of image and mask elements are summed were left sequential. Conceptually the naive CUDA and OpenACC implementations are the same; they both attempt to parallelize the same thing with no extra additions to their algorithms. The shared memory CUDA implementation also parallelizes the outer loop, but in addition it makes use of shared memory and places the mask in constant memory.

Data collection on these implementations was accomplished by timing the total time taken to execute a call to the CUDA kernel. This includes both time spent copying data and time spent computing the convolution. For non-CUDA implementations I measured around what would be considered the equivalent of the kernel; for the OpenACC implementation that would be around the data initialization pragma, and for the sequential implementation that would be around the triple loop which forms the convolution computation. The actual measurements were taken by wrapping the aforementioned areas with start and end timers using the C++ Chrono library. Data from each measurement was written to a file unique to each implementation and later ran through a MATLAB script which formed the data into plots.

Wall clock time was used because it was the most straightforward method to measure all implementations. Data throughput was another option when it came to measuring techniques but it was not used for this project because I did not build the program to overlap the CUDA kernels, nor did I measure with consecutive data input in mind. In addition, the simplicity of wall clock measurement left ample time for me to become more familiar with much of the new content (e.g. CUDA, OpenACC, and the Gaussian blur algorithm and implementation.)

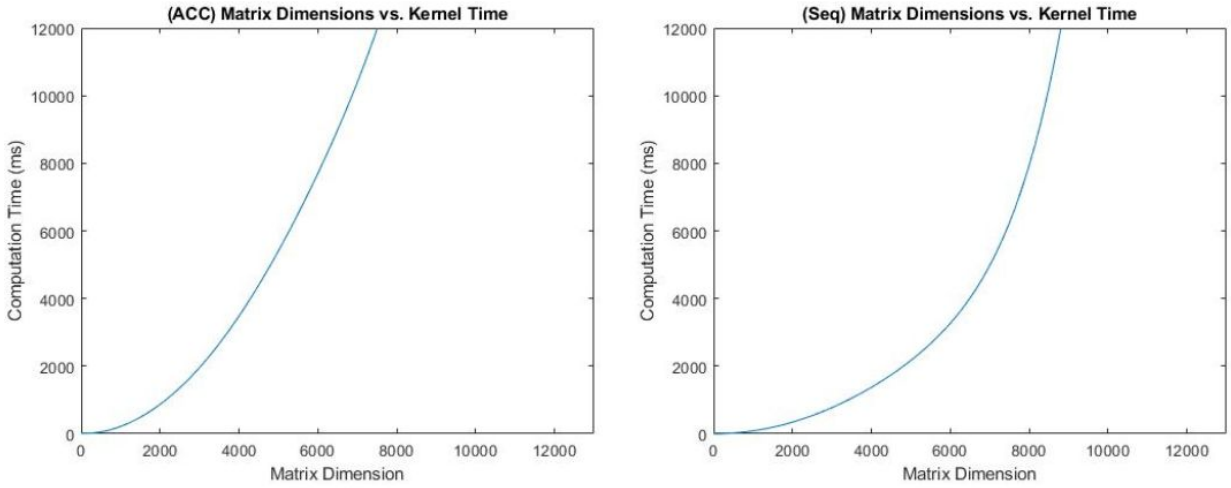
Body

The OpenACC implementation (left plot) and the shared memory CUDA (8x8 block) implementation (right plot) showed one of the largest disparities out of all of the programs. This is due to the shared memory CUDA implementation (8x8 block) performing 2nd best of all programs, and the OpenACC implementation performing the worst of all the programs.



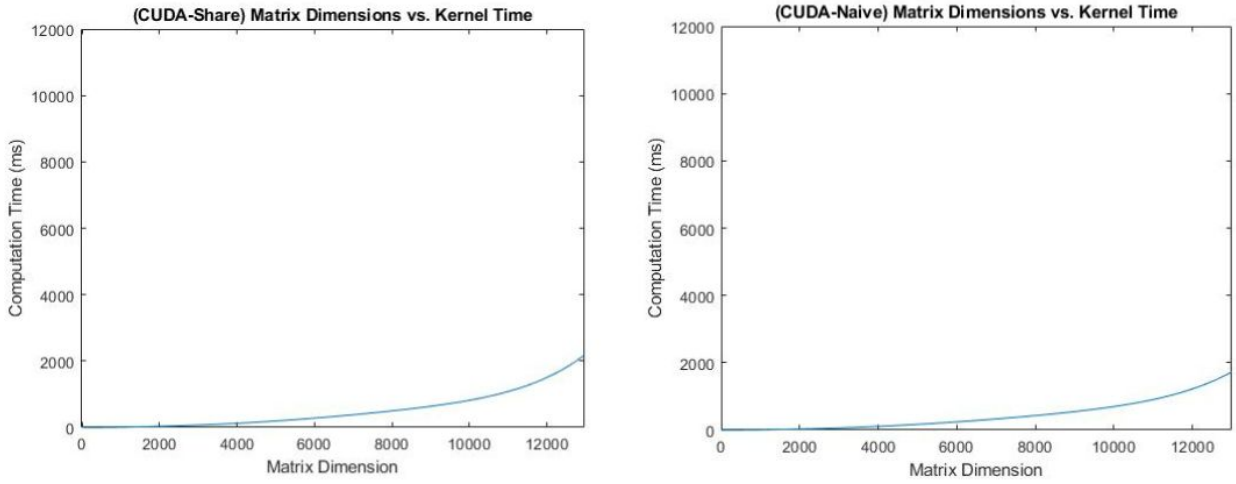
While it isn't surprising that the most heavily optimized CUDA implementation would perform well, it is surprising that a parallelized program would perform worse than the sequential despite

the OpenACC compiler confirming there to be no errors or dependencies. We can see the direct comparison of the OpenACC (left plot) implementation to the sequential implementation (right plot) in the next figure.

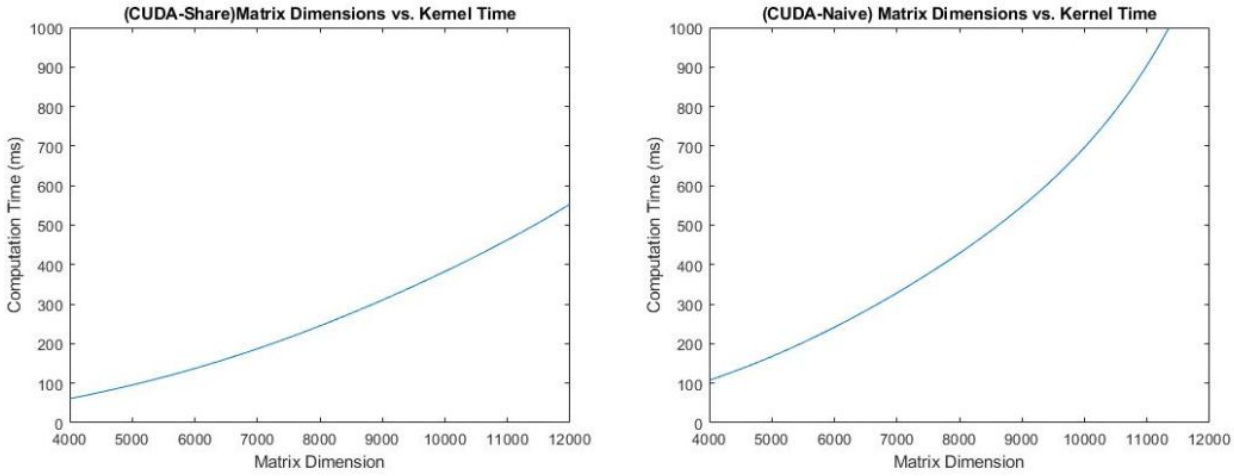


While only conjecture, it seems as if the OpenACC implementation leaves the upper limit of the plot at a larger angle than the sequential. So despite the sequential implementation performing better within the range of data collected, it's possible that the OpenACC implementation still scales better.

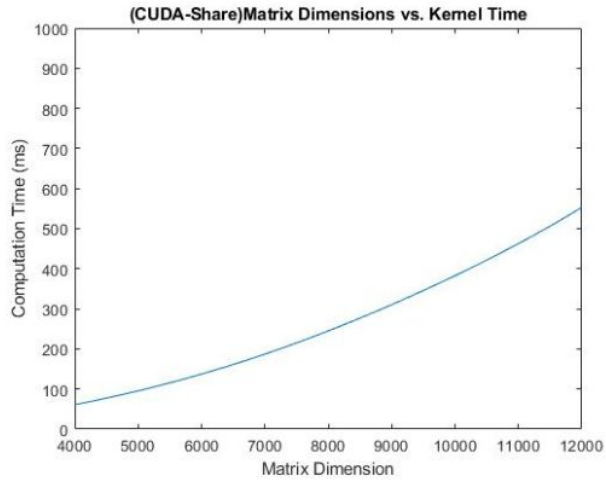
The next figure compares the performance of the 8x8 block, 5x5 mask naive (right plot) and shared memory(left plot) CUDA implementations. To paraphrase the K&H text, the difference in performance between these two implementations at smaller block and kernel sizes is bordering on negligible. In fact, it appears that the overhead of handling the shared memory is putting the naive implementation slightly ahead towards the larger input data sizes.

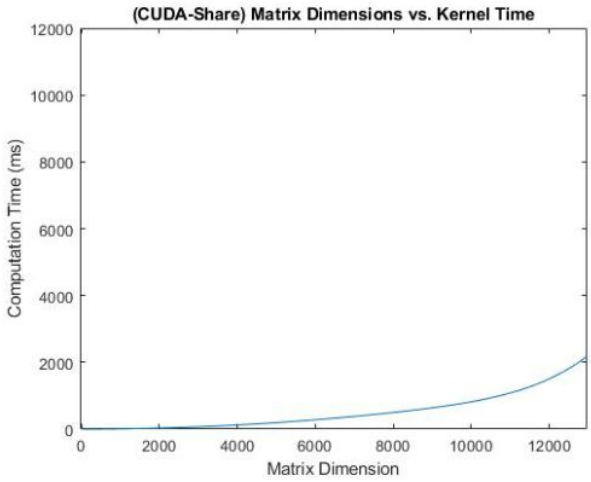


This becomes more apparent when viewing their comparison again, but with block size 64x64 and mask size 9x9. In the following figure we see the shared memory (left plot) and naive (right plot) CUDA implementations taking very different paths as the input data sizes grow. In short, the increased global memory fetches begin to cause significant congestion for the naive implementation, whereas the elements loaded into shared memory for the shared memory implementation are finally showing their worth.



Another exciting result, shown in the plot below, was the comparison between the 64x64 block, 9x9 mask CUDA implementations (left plot) and the 8x8 block, 5x5 mask CUDA implementations (right plot). Keep an eye on the y-axis, despite how the plots look the numbers are going to be what's important here. The left plot representing the larger block and mask size shows the computation time reaching ~600 ms at an input data size of 12000x12000. The right plot representing the smaller block and mask size shows the computation time reaching ~1500 ms at an input data size of 12000x12000. Comparing the raw data confirmed that this pattern was the case for almost all input data sizes besides the very small ones. The larger block and mask size roughly cut in half the computation time needed to complete a Gaussian blur.





Discussion

Most of the unexpected results during this project came from the OpenACC implementation. It is likely that the measurements seen from this implementation were the result of some subtle or strange interaction with how I structured the sequential program. While I would not consider this to be an error (as verified by the OpenACC compiler), I would note that there is likely some small OpenACC-specific changes that could be made by more expert eyes to vastly improve performance. In other words since 2D convolution is parallelizable there must be a way to produce some speedup. What's interesting about this outcome is that it demonstrates the subtleties involved in creating an optimal parallel program with OpenACC. Though more nuanced than it originally seems, it is still much easier to handle than a CUDA program written from scratch.

In terms of positive surprises, the naive and shared memory CUDA implementations landed at practically equal performance when executed with a smaller block size. While at the much larger 64x64 block size, the shared memory implementation jumped ahead. This is inline with what the textbook states, nevertheless it is always confounding for a time sink like the shared memory implementation to not perform any better than the sequential implementation. With a much larger block and kernel size there is more congestion from the fetches in global memory so the shared memory helps to alleviate that traffic and speed up the overall process, hence the shared memory implementation doing much better at the larger block size.

Conclusion

To recap what was covered in the body:

- The naive and shared memory CUDA implementations perform and scale similarly at smaller block and kernel sizes.
- At much larger block and kernel sizes the shared memory CUDA implementation not only outperformed the naive CUDA implementation at all input sizes, but scaled better too.
- Both CUDA implementations performance *significantly* better than the sequential and OpenACC implementations. This was despite the lengthy copy times involved and no kernel overlapping.
- The OpenACC implementation performed slightly worse than the sequential implementation.

The clearest takeaway from these measurements is the substantial performance boost resulting solely from increasing block size in the CUDA implementations. Both naive and shared memory implementations experienced a reduction in computation time upon raising block size from 8x8 to 64x64 equivalent to ~2X speedup (this can be inferred from the plots). Doing this is not always easy though as raising the block size to this extent excludes smaller input data; with the maximum number of resident blocks per SM being a minimum of 16 on NVIDIA devices this amounts to a minimum of 256x256 input data size (and this only grows for the rest of the devices where maximum resident blocks per SM reaches 32!)

The other priority takeaway is the improved scaling of the shared memory CUDA implementation using block size 64x64 and mask size 9x9 vs. block size 8x8 and mask size 5x5. Referencing the K&H textbook, the image array access reduction ratio for an 8x8 block paired with a 5x5 mask is a low 11.1. Moving up to a 64x64 block paired with a 9x9 mask, the text shows the image array access reduction ratio at 64, this is a substantial increase from the former. In other words, the shared memory implementation only becomes useful if given a larger block and kernel size. While this is certainly backed by the text, this is also evident as shown by the aforementioned plots. The shared memory implementation with 64x64 block size rises in computation time at a much steadier rate than its naive counterpart.

Improvements

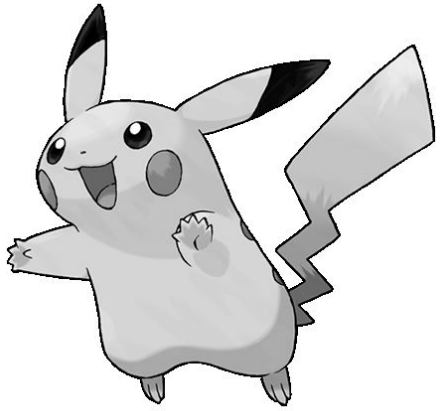
There are two improvements I would like to see in future work on this project. Most important would be the lack of NVIDIA resources used in debugging errors and measuring

performance. NVIDIA provides tools like CUDA-GDB for solving errors in CUDA programs and Nsight Visual Studio for monitoring and measuring analytics. With more time these tools could provide essential information about the CUDA implementations. Likewise for the OpenACC implementation; the OpenACC compiler comes with a variety of tools made for the purpose of measuring performance and providing feedback. But these tools, much like CUDA and OpenACC, are practically languages in and of themselves. Becoming proficient in these resources will require far more time than what is available for side projects during an academic quarter.

Data throughput is another concept I would enjoy using more in this project. While in their current state the CUDA implementations would not showcase useful measurements in data throughput because only one kernel is executed at a time with no overlapping. It is stated in the K&H text and mentioned in class that GPUs hide their transfer latency by overlapping kernels. Overlapping kernels allows back-to-back execution which increases data throughput when running multiple iterations as was done in this project to obtain the data over thousands of input data sizes. I believe this type of measurement would showcase GPUs in the environment they shine most in.

Appendix

Original Image:



Blurred Image: