Characterizing the performance of OpenMP, OpenCL, CUDA, and OpenACC on GPUs

Jason Chan

jwc516@nyu.edu

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, New York 10003

December 18, 2018

Abstract

With the rise of utilizing graphics processing units for parallel computing, many different vendors, companies or consortiums have released frameworks to aid developers and computer scientists. We looked at the four popular frameworks and characterized their performance by performing real-world tests. We ran each framework through a series of photo blurring tests. We concluded that directive based libraries such as OpenACC and OpenMP trade performance for ease of use. While programming language extension frameworks such as OpenCL and CUDA can offer the best performance as long as the code is parallelized well.

1 Introduction

With the many different graphics processing unit (GPU) frameworks out there, which provides the best performance? In this paper, we will explore the four popular libraries and characterize their performance with the real world scenario of photo box blurring. The four libraries are CUDA, OpenCL, OpenACC, and OpenMP. Each framework has a different target objective, thus each framework includes trade-offs to performance to meet that target. We will compare the performance of each framework by not only box blurring a single photo, but also a batch box blurring. We will then look into the reasons on why one framework outperforms another.

2 Background

I chose box blurring as it mimics a real-world use case of parallel computing. In the area of photo editing, one of the most popular programs is Adobe Lightroom. Complaints are

bountiful online regarding the lack of editing acceleration even with GPU support in Lightroom [1]. While Lightroom is GPU accelerated, it only uses OpenGL. Adobe, the developer of Lightroom, is leveraging OpenGL to help render the images for larger 4k displays [2] but not for much for the photo editing aspect. Adobe acknowledged the benefits of OpenCL for image editing as it is more suited for heavy computation such as image blurring over the 3D rendering benefits OpenGL in a photo editing software. The lack of editing acceleration is even more apparent on Adobe Lightroom where users are allowed to edit photos in batches.

3 Existing Research

Various studies in the past compared the one or more of the four parallel computing frameworks. Studies ranged from the application of possible use cases such as machine learning text mining, specifically TFIDF [3] with CUDA to comparing frameworks to each other. A study compared the data movement and kernel times between CUDA and OpenCL [4]. In this study, CUDA code was ported to OpenCL. This was done so there will be minimal variance to how the parallel code works and minimal overhead was added. While the findings were not surprising that CUDA was slightly faster, their study did not explain which tool they used to profile the performances. They also did not explain how they ported their CUDA code to OpenCL code beyond using Nvidia tools. The most comprehensive study found was from Linnaeus University [5]. Their study compared OpenCL, OpenACC, OpenMP and CUDA with a battery of benchmark tests. They observed time, temperatures and the efficiency of creating a parallel program.

4 Proposed Solution

Past studies either looked for practical use cases to apply CUDA to or compared frameworks with hypothetical benchmarks. I propose, to better characterize the performance of each framework, we simulated a practical real problem with the four frameworks. To do this, I propose comparing the performance of blurring a photo and blurring of multiple photos with OpenCL, OpenACC, OpenMP and CUDA.

The blurring of multiple images in a batch is important because, in the real-world application of photo blurring, users will want to not only blur a single photo but blur multiple images in batch. By doing both, we will see what type of performance gains we can potentially achieve if we applied the frameworks to a program like Adobe Lightroom. This study comes at a critical time as the megapixel count from popular camera manufacturers such as Canon are passing the 50-megapixel mark. As the problem size of megapixel count grows, the editing speed will become slower and slower on the CPU as the growth of CPU speed has decreased [6]. We will find which framework is overall the fastest in running the parallel code and investigate the reasons for their acceleration. For the simple parallel directive based frameworks such as OpenMP and OpenACC, we will also find out how well each framework was able to parallelize the code compared to the C extension based CUDA and OpenCL.

To fairly compare the framework, we compared each framework by implementing a box blur. A box blur averages all the neighboring pixels within the predetermined radius of a

target pixel. As the radius increases, the computation intensity increases. Before we start our experiment, we will start with a benchmark time with the sequential code for the CPU. We will test each framework against the CPU for both single image blur and batch image blur.

5 Experimental Setup

The box blurring programs for each respective framework were developed on Linux machine. The machine was running Ubuntu 18.04 with an Intel 6 core i7-8700 clocked at 3.2ghz with a Nvidia GTX 1070 for the GPU. Programs were also tested on New York Universitys Courant Institute of Mathematical Sciences clusters. Each program ran 10 times and the average time performance was recorded.

Each program takes 4 arguments: the width dimensions, height dimensions, number of images in the batch, and the pixel radius size. The dimension we tested were 10,000 by 10,000 or 100 megapixels. We used a blur radius of 15. The number of photos in a batch were 1 and 10. The program file names follow their respective framework name + blur. For example, the CUDA version will be cudaBlur.cu. CUDA version was compiled with NVCC. OpenCL, OpenMP, and OpenACC versions were compiled with the PGI compiler.

All code will be bare-bones minimal to reduce any extra overhead that is not related to the parallel code. To be fair, the parallel code for each program were as similar to each other as possible to avoid any unintentional optimization. The box blur algorithm was based on the Programming Massively Parallel Processors blur algorithm [7]. I also did not use any optimization techniques to see how well the frameworks accelerated the parallel code to see how much of a minimal performance gain we got over the sequential CPU code.

Nvprof and NVIDIA System Management Interface (Nvidia-SMI) tools were used to monitor GPU performance and utilization. We used the Linux time function to track the wall time. In the code, a timer was also used to track the GPU activities. GPU activities start when data is being moved from Host to Device, and ends when data is moved back to host. This will be useful for frameworks that are not accessible by any of the Nvidia tools such as OpenCL.

6 Findings and Analysis

After running all four frameworks through the box blur algorithm, we can see how each framework compared to the sequential CPU version. First, the results will be presented, then further down this section, an analysis will be provided on why the results came out the way they did.

One hickup during the tests was the lack of a suitable compiler that could compile OpenMP for the GPU. The compilers available on the NYU clusters did not have support for OpenMP GPU offloading. Currently, only the IBM XL compiler is out of prototype for OpenMP GPU offloading [8]. Unfortunately, the IBM XL compiler only works on Power architecture and not x86.

Table 1 shows the performance of applying box blur to a 100-megapixel image with a blur radius of 15 pixels to a single image. The table includes the total time, GPU time and

Table 1: Performance of a Single Image Blur

Framework	Total Time	GPU Activities	% time for GPU
CUDA	.810s	.046s	54.3%
OpenCL	.820s	0.049s	60.1%
OpenACC	8.52s	7.939s	93.1%
OpenMP (Multicore)	133.857s	N/A	N/A
CPU (Sequential)	323.03s	N/A	N/A

the percentage of the total time spent on GPU time. GPU time or GPU activities include moving data from host to device, kernel, and device back to host. This step was needed as Nvidia's profiling tools did not work with OpenCL.

Table 2 shows the performance of applying box blur to ten 100 megapixel images with a blur radius of 15 pixels.

Table 2: Performance in of a 10 Image Batch Blur

Framework	Total Time	GPU Activities	% time for GPU
CUDA	1.650s	.467s	28.3%
OpenCL	2.020s	0.507s	25.1%
OpenACC	79.732s	78.439s	98.3%
OpenMP (Multicore)	1320.147s	N/A	N/A
CPU (Sequential)	3210.6s	N/A	N/A

Comparing the results from both single image and 10 image batch blur, we see that CUDA and OpenCL provided the most speedup over the CPU sequential version (See Fig. 1). For the first test of a single image, the wall time was fairly close between CUDA and OpenCL. This should be the case as both had nearly identical kernels. While we were unable to profile the OpenCL's GPU activities with NVPROF, we were able to do it with CUDA. For the CUDA version, 98% of the total time for GPU activities was moving data from device to host and host to device. When we compare the 10 image batch, we also noticed that the total time was longer for the OpenCL version when compared to the CUDA version while the GPU activities time is still very similar. Since the programs were nearly identical from the main method to the kernel, the only major difference between OpenCL and CUDA was that the OpenCL had extra processes that included querying for devices, creating context, creating queues and multiple tear-down steps. While not noticeable with a single image, with a batch of 10, this extra overhead resulted in the OpenCL version of the test to take 23% longer.

When comparing the OpenACC version to the sequential CPU version, the OpenACC version's speedup is not as great as OpenCL and CUDA. Fig. 1 shows the speed of four different frameworks. OpenACC and OpenMP converts sequential code into parallel code. We rely on the developers of both of these frameworks to decide on what and how to parallelize the code. For OpenACC and OpenMP's tests, examining the GPU activity times and nvprof traces, 92-97% of the time was spent executing the kernel. The data movement

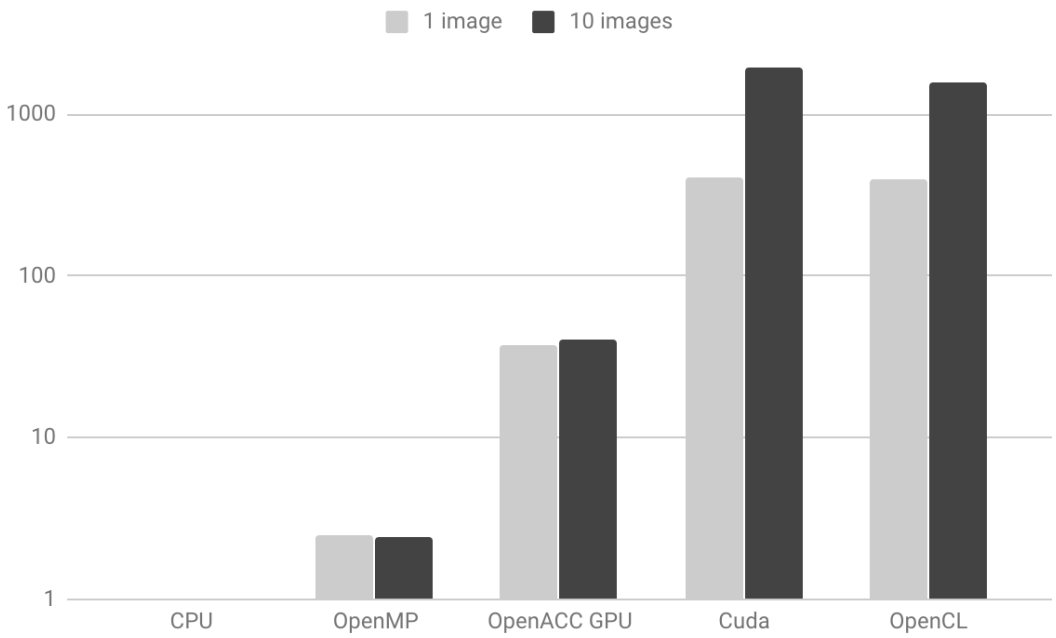

Figure 1: GPU Speedup (CPU Time/GPU Time)

from device to host and vice versa was only a 20ms of the 7.9s of the GPU activities. Thus, this leads me to conclude that the way OpenACC is parallelizing the sequential code, while it does work, it is not as effective as having the programmer program explicit parallel code such as in OpenCL and CUDA. Looking at the compilation information ouput with the "-Minfo" option, we also see that OpenACC is organizing threads into blocks of 128 automatically. The technic in determining geometry is unknow to the programmer. By letting OpenACC decide on the grids and blocks, we will have to do another study to try different algorithms and problem sizes to see how OpenACC manages geometry. With some manual optimization, we can possibly see a better speed up for OpenACC. For OpenMP multicore performance, we did see some acceleration. For both single and ten images, the speed up was about 2.5 times. The acceleration times are consistent over multiple images as there are no extra steps of launching a kernel or allocating memory on the GPU device. Both OpenACC and OpenMP could probably benefit from a more efficient box blur algorithm that is easier to parallelize. However, to be fair with the tests, as we mentioned before, we keep the algorithm fairly similar to see how each framework would perform given the same problem to solve.

On the subjective end when comparing the four frameworks, the amount of effort required to convert sequential code into parallel code for a specific framework varied greatly. OpenMP and OpenACC only took 2-3 extra lines of code to convert sequential code into parallel code. The compilers will do the rest of the work in converting the sequential code into working parallel code for the GPU. As we've seen, while performance may not be the best, we gained a quick and easy implementation process. On the other spectrum, OpenCL requires a lot more overhead and setup code just to get to launch the kernel. But the performance is near CUDA. The ease of implementing CUDA falls somewhere in between OpenCL and

OpenACC. The amount of effort to implement parallel CUDA code has been dropping. With the introduction of unified memory, the amount of setup overhead code to launch the kernel has been reduced.

Table 3: Implementation Time

Framework	Environment Setup	Programming Time	Debugging Time
CUDA	5 hrs	8 hrs	3 hrs
OpenCL	3 hrs	12 hrs	9 hrs
OpenACC	2 hrs	1 hrs	2 hrs
OpenMP (Multicore)	2 hrs	1 hrs	3 hrs
CPU (Sequential)	1 hr	3 hrs	3 hrs

Table 3 shows the time I've spent creating each box blurring program for their respective framework. The learning curve for each library may also depend on ones own prior experience. Arguable, CUDA for Python will probably be very approachable compared to CUDA for Fortran. Thus, learning of how to use each framework was excluded. The ease of use for OpenACC and OpenMP were reflected all of the implementation time metrics. Both took about an 1 hour to implement if you already have the sequential code, which I did. Programming took the longest for OpenCL as the process to program the kernel to launch required a lot more setup/tear down code. For debugging, OpenCL also took longer (9 hrs) than CUDA (3 hrs) as Nvidia excluded OpenCL from their profiler.

7 Conclusion

Base on our tests, while OpenMP, OpenACC, CUDA and OpenCL as frameworks, all have the ability to accelerate a problem such as performing an image blur. The common trade-off for performance is ease of use. OpenACC as not able to create parallel code as good as a programmer, thus resulted in lower performance. We conclude that directive based libraries such as OpenACC and OpenMP traded performance for ease of use. While C programming language extension frameworks, such as OpenCL and CUDA, were more programming intensive but they offered superior speedups. It appears that there is a cost to OpenCL's extra setup and tear down, and that is performance. While CUDA is more programming intensive, this characterization of CUDA might change in the future as Nvidia keeps making strides to improve the ease of use of CUDA.

References

[1] *Why Your Lightroom CC May Actually Be Slower with the New GPU Acceleration*, available at `petapixel.com/2015/05/08/why-your-lightroom-cc-may-actually-be-slower-with-the-new-gpu-acceleration/`.

[2] *Can OpenGL And OpenCL Overhaul Your Photo Editing Experience?*, available at `tomshardware.com/reviews/photoshop-cs6-gimp-aftershot-pro,3208-7.html`.

[3] Y. Zheng, F. Mueller, X. Cui, T. Potok, GPU-Accelerated Text Mining, *North Carolina State University*, 2011.

[4] K. Karimi, N. Dickson, F. Hamze, A Performance Comparison of CUDA and OpenCL, *Computing Research Repository*, 2011.

[5] S. Memeti, L. Li, S. Pllana, J. Koodziej, and C. Kessler, Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption *ARMS-CC '17*, 2017.

[6] Intel Corp, *FORM 10-K*, Retrieved from http://www.sec.gov/edgar.shtml.

[7] D. Kirk, W. Hwu, 3rd ed, *Programming Massively Parallel Processors*, Morgan Kaufmann, 2016, Print.

[8] OpenMP, *OpenMP Compilers Tools Home¿Resources¿OpenMP Compilers Tools*, Retrieved from openmp.org/resources/openmp-compilers-tools/.