Performance of Parallelization using GPUs

Philip Sigillito
University of Nebraska Omaha
College of Information Science & Technology
Omaha, USA
psigillito@unomaha.edu

Austin Riffle
University of Nebraska Omaha
College of Information Science & Technology
Omaha, USA
ariffle@unomaha.edu

Abstract—

Graphical Processing Units, GPUs, excel at computationally intensive tasks but their use introduces computational overhead as the CPU and GPU must communicate and synchronize with each other. In this paper we look at existing research on GPU use and quantitatively measure the performance gain of using a GPU. We conduct two experiments to find the 'tipping point' of when a computation will run faster using the GPU despite the communication overhead. The first experiment computes an integration of pi where we find that the tipping point occurs at approximately 190,000 steps. The second experiment is matrix multiplication, in which target devices compute various dot products on input matrices. Some intriguing time spikes are observed. Generally we find that GPU performance benefit eventually outweighs communication overhead as the problem grows in size. Interestingly we also find that GPU processing as a portion of the total program runtime does not decrease as the final CPU reduction increases.

Keywords—GPU, Heterogeneous Processing, OpenCL, parallelization, threads

I. Introduction

Graphical Processing Units, GPUs, were initially introduced to the public as a way to increase graphics performance for gaming and video processing. While gaming is still a significant part of their use, GPUs have become significantly more important to a wide array of applications in computation. This trend of applying GPUs for general purpose computation is often referred to as GPGPU. Today, GPUs are used wherever there are processing intensive tasks such as scientific research, real-time systems consuming data to be processed by artificial intelligence, or high performance cloud systems.

GPUs are typically connected to the CPU through a high-speed bus, such as PCIe, and contain their own control chipset, memory, and processors. The details of this bus are often abstracted away from the user via device drivers and libraries. When the CPU has a computationally intensive task, it will pass off that calculation to the GPU. We call this heterogeneous processing because the computation is split between the CPU and the GPU which have different processor architectures. Whereas CPUs are designed to perform a variety of instructions such as control flow, arithmetic, and memory access; GPUs are specialized for intensive arithmetic and single instruction multiple data, SIMD, computation. A high performance CPU may have sixteen cores. Each of these cores can run independently as they each have their own control unit, arithmetic logic unit, registers, and L1/L2 caches. In comparison, a GPU can contain hundreds of lightweight cores that work together to complete computation. The GPUs cores are specialized to pipeline calculations so that lots of calculations can be processed quickly.

The time required to communicate between the CPU and GPU has been identified as one of the significant bottlenecks to system performance. GPUs have typically performed on a 'best effort' approach for returning values to the CPU. As GPGPU applications become more prevalent, interest and research in achieving tightly bound time constraints for processing has grown. There is a large body of research looking at means to reduce the cost of message passing between the CPU and GPU. Other research has looked at ways to integrate managing the GPU dynamically through the OS instead of application code.

There also exists a great number of research papers explaining why the cost of message passing is important, but there is a lack of quantitative examples indicating the actual impact of message passing cost. For our project, we propose examining the cost of message passing when utilizing a cpu-gpu architecture. We will use simple arithmetic problems of various sizes to compare the performance gained from utilizing a GPU. We intend that our research will quantitatively show when it is beneficial to offload computation to a GPU and when it is more efficient to simply process the computation on the CPU. Using existing GPU APIs the programmer can specify the number of threads used by the GPU. We also intend to quantitatively show the impact of increasing and decreasing threads used as the size of the problem being processed changes.

The rationale for our experiments is based on Amdahl's Law. Proposed in 1967 by Gene Amdahl, his observation states that the overall performance impact of improving a portion of a program through parallelization is limited by the amount of the program that is parallelizable[12]. As stated in [12], this observation can be formally written as:

$$T = T_s + T_p = \frac{S}{R_s} + \frac{P}{MR_p}$$

T is the total process time, T_s is the serialized work process time, and T_p is the parallelizable work process time. S is the serializable work and P is parallelizable work. S is divided by the serialized processor rate, R_s. P is divided by the number of parallel processors, M, times the rate of those processors, R_p.

Amdahl's law is still the basis for evaluating parallel programs as it provides a theoretical limit to the performance achievable through parallelization for a given system.

II. Background and Related Work

There are many existing APIs and tools we can use for our implementation. NVIDIA and AMD both provide proprietary APIs that can be used in a C program to configure and issue commands to the GPU named CUDA C and ATI Stream. These APIs use directives called Pragmas to state what the developer intends and low-level implementation of the details are handled by the framework. There are also open source APIs such as the OpenCL framework for writing heterogeneous architecture programs, the OpenMP API for writing multithreaded applications, and OpenGL for processing two and three-dimensional graphics.

In our survey of existing research, we found a wide array of approaches for improving GPU performance. [1] proposes parallelizing communication between the CPU and GPU to reduce the communication bottleneck. This approach recommends chunking the data during communication and processing so that the GPU can be processing chunks while still receiving data. Their approach aims to pipeline the GPUs processing so that no part of the device is idle. In their optimal solution, the GPU was fully pipelined and a portion of the data to be processed was held back to be processed by the CPU while it waited for the GPU's results to return.

[2] also looked to increase CPU-GPU architecture performance. In this paper, the authors proposed a new task scheduling algorithm. Whereas traditional task scheduling simply assigns the next task to the first idle processing unit, this algorithm aims to assign tasks to either the CPU or GPU depending on which will process the task faster. The algorithm achieves this by comparing the estimated cost of processing on the CPU and querying main memory against the cost of processing on the GPU with the cost of communication latency. The authors found a 26.5% reduction in runtime using this algorithm although they did not include the compute cost of running the algorithm in their findings.

Paper [3] investigated ways to further integrate the GPU as a device managed by the OS. The paper acknowledges that real-time systems need tightly bound time constraints for GPU processing. The paper identifies that there are several projects looking at better managing the GPU including TimeGraph, GPUSynch, and Gdev. This paper proposes leveraging Linux's support of loadable kernel modules, LKMs, to extend the OS's ability to manage the GPU. The paper proposes a new LKM framework that extends the Linux Kernel so that it can dynamically reconfigure scheduling algorithms at runtime for applications that use GPUs.

In our survey we also found text resources covering the topics of modern GPU programming. These resources are useful for implementing our tests and better understanding current best practices. [4] is a module taught at Macalester College covering the basics of GPU architecture, CUDA programming, implementing 2D and 3D matrix multiplication, and simple ray tracing. [5] is a book covering writing programs using the CUDA C API from introductory to advanced topics. The book is written by two senior developers on the CUDA team at NVIDIA and covers techniques we will implement in our experiments such as configuring the number of threads for the GPU to use.

High Performance Computing, HPC, is a popular area of research that naturally involves utilizing GPUs. In [6] the authors look at making writing HPC code easier by expanding the Fortran front-end for LLVM, named Flang. The authors expand Flang to support OpenMP directives sent to accelerator and GPU devices. LLVM is a popular open-source compiler back-end used for code optimization and generation. LLVM is very popular because it is designed to be modular and can work with different front-ends for different languages such as C, C++, Haskell, and Rust. Although the work outlined in [6] does not relate to our project's goals it is informative to see how researchers are working to implement high performance GPU code easier.

Relating to the overall sentiment given from this paper, [7] recognizes that GPUs can be underutilized, particularly for small scale computations. It is stated that the NVIDIA Fermi GPUs, Concurrent Kernel Execution (CKE) is used to alleviate this issue by performing a high volume of small scale operations concurrently. However, they recognize that the model breaks down when the host application is multi-threaded, or there are multiple processes attempting to use the same GPU. They propose a consumer-producer model to alleviate this pain, where host threads are producers, and a host consumer is responsible for submitting work to the GPU. They note a large performance increase, but suggest that it is highly dependent upon the CPU time slicing algorithm that affects their producer-consumer paradigm, and never address a minimum realistic minimum load size.

Similar to [7], [10] also discusses the effect of thread scheduling on GPU performance (albeit not as focused on small scale operations). They propose a technique called "Interleaved Thread Block Scheduling." Interleaved scheduling stands in opposition to spatial scheduling, where it breaks the notion that each application should only be allowed to submit kernels to one core. Instead, kernels are submitted to all cores, and then occupancy tests occur. Kernels are only run on cores where predefined thresholds are not met. They end on a note that host application pairs must be profiled to effectively use this technique. While this technique yielded notable performance improvements, the tuning of parameters to specific use cases may not be trivial in all cases. The relation to this paper is again via thread scheduling, a persistent notion thread utilization does have an effect on GPU proficiency.

[9] proposes that on chip GPU memory is less costly to use than the scheduling limit, and thus supplies a technique of swapping out executing "Virtual Threads" (think similar to virtual memory) which are spread across the entirety of the GPUs memory, regardless of the scheduling limitations. Essentially, thread blocks belong to Cooperative Thread Arrays (CTAs) that are marked as active while executing, but

will be marked as inactive if the warps associated with that CTA are encountering high latency. Once inactive, that CTA is switched for an active CTA. Large swaps of CTA state is avoided due to the fact that all CTAs, active or inactive, are always stored in GPU memory. Therefore, the lowest latency CTAs should spend the most time running, thus optimizing GPU usage. Again this paper recognizes that the cost of GPU threads is not free, as does this paper, but instead supplies another iteration thread scheduling to optimize thread utilization (as opposed to addressing the question of CPU viability). We recognize that in many applications, particularly for large scale operations, approaches such as [9] are indeed needed and valuable, but the lack of baseline CPU-GPU data could help assist readers in determining viability of proposals in the first place.

Realizing that CPU and GPU architecture is vastly different, [8] discusses techniques used to make algorithms more efficient when used by a GPU. They discuss a technique known as algorithm flattening. The intent of this technique is to remove branching from the algorithm, as unlike CPUs, GPUs do not handle branching well. Removing branching will ensure that the GPU is fully utilized, and time spent waiting for one GPU thread (while others are waiting) is minimized. This paper focuses on simple operations, rather than algorithms with complex branching, so [8] does not directly pertain to the work proposed here, but readers should be aware when weighing the tradeoffs between CPU and GPU cores that techniques such as this do exist, and that some algorithmic translation may offset other bottlenecks.

In contrast to direct message passing, [11] proposes using the Unified Virtual Address (UVA) available in NVIDIA Fermi GPUs as means to avoid bus access altogether. They propose a task structure that defines abstracted input and output queues to the user, while a background thread places requested computation in task slots. These slots are then marked as ready for processing by the GPU, and then store the results accordingly. The interesting architecture proposed by this paper completely eliminates the message passing over the bus from the CPU to the GPU, as the CPU is instead writing data to a shared memory region. Additionally, the GPU is also not passing messages across a bus to a CPU, but is instead performing memory write operations to the shared memory region. Unfortunately, while this paper recognizes the bottleneck of extraneous bus access, no quantitative results on the actual performance are provided - simply alternatives to avoid it in the first place.

III. Collecting GPU Performance Data

We are using OpenCL as our primary tool for testing. [14] and [15] outline the structure of OpenCL. OpenCL, Open Compute Language, is a language and framework designed for heterogeneous parallel programming. It allows a host program to allocate work to other hardware devices. The set of devices working together share a common context. Devices in the same context use asynchronous messaging to send instructions and data. The host program defines a queue for each device through which instructions and data are passed.

OpenCL uses the concept of a global dimension to describe a parallelism problem and identify the number of work-items or threads required. [14] provides an insightful example. Suppose a program processes images that are 10x10 pixels in size. If the program performs an operation on each pixel, then the global dimension would be 100 because it needs 100 work-items to process the image in one batch. If the program instead performed an operation on each row of pixels then the dimension would shrink to ten because there are ten rows.

Work-items are grouped into work-groups which are executed simultaneously on a single compute unit. The size of a work-group relates directly to the size of the compute unit in hardware. Threads within a work-group can synchronize with each other but threads in different work-groups cannot. For example, a GPU may consist of many compute units that each have 16 cores. One 16 core compute unit on the GPU could process a work group with up to 16 threads.

If a work-group is smaller than the compute unit, the unused cores are idle. This reflects OpenCL's intentional low level design. It is up to the programmer to be aware of the hardware architecture and design their parallelization appropriately.

The instructions sent to a device are called the kernel. The kernel is written in the OpenCL language which is an extension of C99. The kernel is compiled separately from the host program and is often lazy compiled during runtime.

OpenCL is a good candidate framework for our experiments because it provides low level access to customize parallelization. It allows us to alter and fine tune our experiments in order to see how changes in software impact performance on hardware.

Utilizing OpenCL, we have created an application which profiles device performance while performing various operations. At run time, the program is specified to run against either a CPU or GPU device. Using OpenCL, information surrounding the device is obtained, the device is added to a newly created context, and a command queue is initialized. Next, the program creates all required read and write buffers shared between the host and the device. The last step prior to device execution is to build the OpenCL program, compile the kernel, and configure all read and write buffers.

After the kernel is loaded, the device is ready to begin execution. A timestamp is taken using the host's high resolution clock (via the C++ standard library). Device execution begins and the host blocks until it is complete. After completing execution a timestamp is immediately taken to calculate execution time. The timing information is printed, results validated, memory cleaned up, and the program exits. For our analysis, the application was run on both CPU and GPU target devices with varying size problems.

We are conducting our tests using two kernels. The first kernel implements the matrix multiplication via the dot product. We start with two matrices, **A** of size m x n and **B** of size n x p. The multiplication of these matrices results in **C**, which is of size m x p. Any given element c_{ij} in **C** where (i = 1, …, m and j = 1, …, n) is the dot product of row i in **A** and column j in **B**, such that [16]:

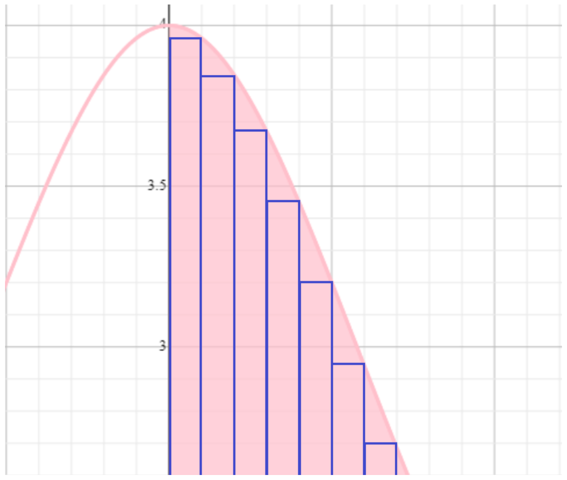
$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

This kernel's input parameters are matrices **A** and **B**, and integers *m* and *n*. Values *i* and *j* are calculated dynamically, based upon the running kernels assigned global work id *n*, where $n = 1, \ldots, mn$. c_{ij} is then calculated and placed into the corresponding output write buffer representing **C.**

The second kernel performs the integration:

$$\int_0^1 \frac{4.0}{(1 + x^2)}$$

As stated in [13], this integration is a common "hello world" program for parallel programming. The result of this integration is π, making it easy to check for accuracy. Our kernel solves this integration by dividing the area under the curve into rectangles as shown below:



After each rectangle's area is found, the results are summed to provide an approximation of pi. One can easily see that adding more rectangles will result in a more accurate estimation, but will also require more calculation. Our experiment compares the execution time of different OpenCL worker-group configurations against serialized processing as the number or rectangles increases. We also measure the execution time on the GPU as a portion of total execution time.

Process execution time can vary slightly due to other operations that happen to be running on our test machine. To minimize incidental impact to performance, we are running each experiment five times and average the results. Results are also highly dependent on hardware. For all results we are using an AMD Ryzen 5 5600x 6-Core Processor CPU and a GeForce GTX 1070 Gaming x8 GPU. This GPU uses the Nvidia pascal architecture, has 8 GB of GDDR5 memory, and 1920 cores.

IV. π Integration Results

Our first experiment compares the execution time of a single threaded program running on a CPU and a program that offloads processing to a GPU to calculate the integration:

$$\int_0^1 \frac{4.0}{(1 + x^2)}$$

Our approach consists of dividing the area under the curve into rectangles (steps), finding each rectangle's area, and summing the areas together. This approach provides an approximation of π. As the number of steps increases, the accuracy of the approximation increases. Our test measures how the execution times increase as the number of steps increases. Step counts are powers of two. This ensures the step count is divisible by the GPU's work group size. For all tests, the GPU work groups are 32 threads, except in cases where there are less than 32 steps to calculate.

OpenCL documentation instructs developers to tune the worker group size for their hardware [15]. Since the best worker group size is hardware dependent, it is only important that our worker group size be reasonable to show the trend of performance improvement.

We ran each test five times and averaged them together to reduce the impact of incidental factors such as other processes running concurrently. Three times were measured for the GPU offloading process: total time to complete the process, time spent executing on the GPU, and the time when results were returned from the GPU. Only total execution time was measured for the single threaded program.

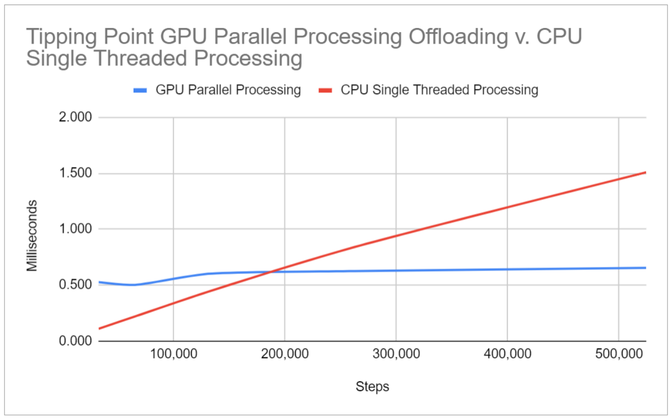
We used C++'s STL chrono functions to measure the total execution time and time required to receive results from the GPU. We used the API provided by OpenCL to measure GPU execution time.

The results of our experiment are listed below:

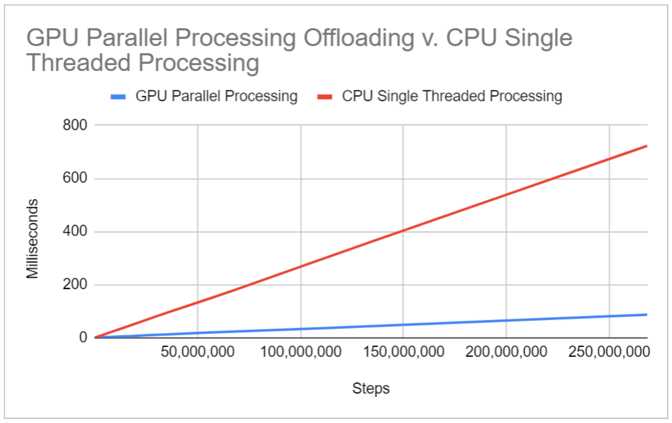
Pi Integration GPU Offloading						
Steps		Local Group Size	# Work Groups	Total Time (ms)	Receive Result from GPU (ms)	GPU Exec Time (ms)
2^0	1	1	1	0.503	0.503	0.023
2^1	2	2	1	0.622	0.622	0.020
2^5	32	32	1	0.456	0.456	0.024
2^10	1,024	32	32	0.458	0.458	0.024
2^15	32,768	32	1,024	0.528	0.528	0.032
2^16	65,536	32	2,048	0.503	0.503	0.040
2^17	131,072	32	4,096	0.602	0.602	0.055
2^18	262,144	32	8,192	0.626	0.626	0.089
2^19	524,288	32	16,384	0.654	0.654	0.153
2^20	1,048,576	32	32,768	0.892	0.892	0.281
2^21	2,097,152	32	65,536	1.229	1.229	0.540
2^22	4,194,304	32	131,072	1.863	1.863	1.055
2^23	8,388,608	32	262,144	3.250	3.250	2.118
2^24	16,777,216	32	524,288	5.932	5.932	4.139
2^25	33,554,432	32	1,048,576	12.011	12.011	8.876
2^26	67,108,864	32	2,097,152	23.054	23.054	17.242
2^27	134,217,728	32	4,194,304	43.271	43.271	32.913
2^28	268,435,456	32	8,388,608	87.030	87.030	64.850

Pi Integration Single Threaded CPU		
Steps		Total Time (ms)
2^0	1	0
2^1	2	0
2^5	32	0
2^10	1,024	0.004
2^15	32,768	0.110
2^16	65,536	0.221
2^17	131,072	0.442
2^18	262,144	0.840
2^19	524,288	1.509
2^20	1,048,576	2.877
2^21	2,097,152	5.671
2^22	4,194,304	11.353
2^23	8,388,608	22.745
2^24	16,777,216	44.957
2^25	33,554,432	90.111
2^26	67,108,864	177.389
2^27	134,217,728	361.129
2^28	268,435,456	723.100

Unsurprisingly, our results found that the single threaded CPU program ran considerably faster than the GPU offloading program when executing a small number of steps. This is expected as offloading processing to the GPU requires creating a messaging queue, sending the data and kernel program to the GPU, compiling the kernel, and then waiting for the results to return. As the number of threads increased, the GPU offloading program outperformed the CPU program. This tipping point occurred at approximately 190,000 steps.



Both processes increased in execution time linearly as the number of steps increased. The notable difference is their rate of increase. As shown in the following graph, the single threaded process increased in runtime significantly faster than the GPU offloading program.

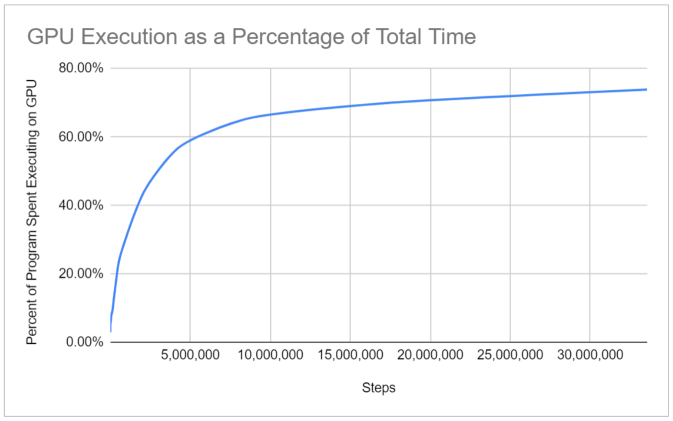


These results are expected as GPU offloading reduces the effect of increased steps by dividing work between parallel threads. The GPU hardware further reduces the impact of increased steps as its cores are designed to execute arithmetic operations, whereas CPU cores are designed for general application.

One interesting characteristic of GPGPU processing we found during testing is that you cannot synchronize worker groups before returning results to the host program. This means that you cannot determine the integral value with one call to the GPU. For our experiment, each worker group sums its result on the GPU and then each worker group's result is summed by the CPU. The time of the CPU summation is reflected in the difference between the total execution time and time when results are returned by the GPU.

One approach to mitigate a significant final summation cost is to send the results of worker groups back to the GPU for further reduction [14]. This requires another kernel and data set to be sent to the GPU. If we were trying to create the fastest running code, we could benchmark the execution time of summing the results locally against offloading the summation to the GPU. We could then continue to send the results of summations to the GPU until the summation could be more quickly processed by the CPU.

The final summation running on the CPU does not appear to have significantly impacted our results. As the following graph depicts, the GPU dominates processing time as the number of steps increases.



This result is surprising as we expect the GPU execution percentage to eventually decrease as the final summation cost increases. Additional testing could be performed to determine if the final summation eventually dominates the process' runtime.

V. Matrix Multiplication Results

The second experiment conducted performed matrix multiplication. Each element of the resulting matrix is a dot product of a row and column from the input matrices. The kernel performs the following dot product operation to calculate a specific element in the output matrix. While there exists techniques that provide improved execution time of this, such as memoization, this naive implementation is used for straightforward benchmarking:

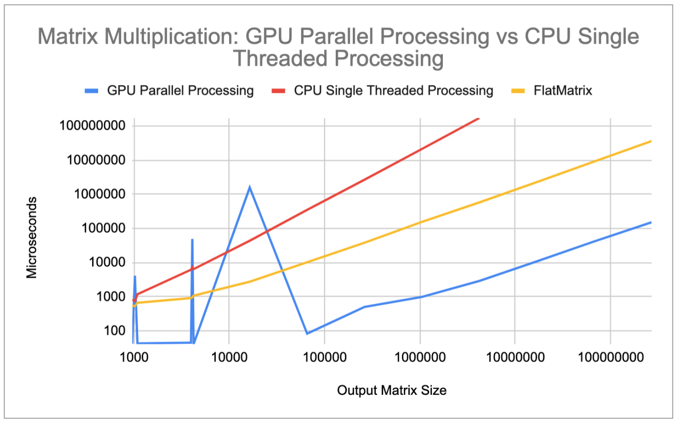
$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

All matrices were stored in "flat" memory buffers, meaning the underlying buffer is one dimensional, and each matrix element being stored as the C int type. This was done to increase memory performance by providing a contiguous memory block, as opposed to blocks pointing to blocks as is typical in naive multidimensional arrays. This tradeoff provides improved memory performance at the cost of some small computations.

Additionally, all input matrices were randomly initialized using a uniform distribution. Staying consistent with the first experiment, C++ and OpenCL API were used to profile execution times, test results were averaged over 5 runs, and a consistent/reasonable work group size (for the target hardware) was used. The results are in the following table.

Matrix Multiplication GPU Offloading					
Matrix Size		Local Group Size	GPU Time (μs)	CPU Time (μs)	Flatten Time (μs)
31x31	961	32	43	827	557
32x32	1024	32	4168	753	529
33x33	1089	32	43	1191	674
63x63	3969	32	46	6234	925
64x64	4096	32	49733	7341	970
65x65	4225	32	41	6503	1089
128x128	16384	32	1589401	43913	2764
256x256	65536	32	84	352318	10335
512x512	262144	32	508	2689454	38422
1024x1024	1048576	32	996	21314885	156723
2048x2048	4194304	32	2921	171908690	582605
4096x4096	16777216	32	10940		2291666
8192x8192	67108864	32	41999		9141804
16384x16384	268435456	32	152439		36404934

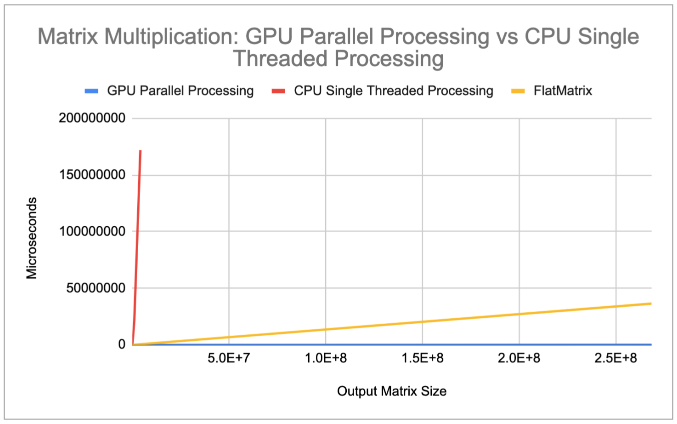
This table contains some intriguing/inconsistent results which are highlighted in yellow. These outliers are notably multiples of 32, however this pattern does not continue as the matrix size increases, nor at adjacent matrix sizes (e.g. the spike was observed at an output matrix size of 32x32, but not 31x31). In general, the growth rate for the GPU is nearly linear when the outliers found at output matrix sizes of 1024, 4096, and 16384 are viewed aporetically. This is best illustrated in the following log-log plot.



This plot illustrates nicely just how immense the outlying values were. The cause of these outliers is currently unknown, however our conclusions are established on the assumption that the general GPU processing time trendline represents a reasonable expectation of performance. Possible reasons for the outliers could be: implementation error, OpenCL framework issue, or hardware specific issues.

The yellow line visualizes execution time spent using a random number generator to initialize the "flat" matrices. This value is important as it did grow to dominate experiment execution time, and was a factor in the number of data points collected.

Next we see that the GPU device offered many orders of magnitude of improved performance over the CPU device. Like all log-log plots, that one has the discernible property that the slope of a given function represents the polynomial degree of that function. Thus, here we are able to visualize the exponential growth of CPU offloading compared to that of the near-linear GPU offloading. In order to truly appreciate the GPU performance, we can look at a linear plot of the same data.



This linear plot clearly visualizes the exponential CPU growth, and how at large matrix sizes, GPU is the only sane offloading option to execute the implemented kernel. The largest output matrix size obtained using the CPU target device was 2048x2048 at an execution time of ~2.87 minutes. Given the growth rate, any larger output matrix sizes would have taken many hours to days on a CPU device to compute, thus making collection of that data infeasible for our experiment. Again, we can see that the random initialization time began to dominate execution time in comparison to the GPU target device.

In contrast to the first experiment, explicit synchronization between work groups was not required. This is due to the fact that the input matrices were read-only (multiple readers with 0 writes requires no synchronization), and the output memory region (residing in the output matrix) for each dot product operation is inherently mutually exclusive with all other kernels (true on both CPU or GPU).

VI. Conclusions

In conclusion, we have shown GPU offloading does have a cost and that a tipping point between single threaded CPU processing and parallelized GPU processing does exist. In the first experiment we determined that this tipping point was approximately 190,000 steps. The second experiment only observed an undisputed GPU advantage starting with an output matrix containing 65536 elements, although some questionable GPU execution time outliers result in merely a semi-confident proclamation.

We can also conclude that fine tuning is important. The workgroup size and global size parameters make a large impact on overall device performance as observed in the first experiment. Ultimately, the higher the GPU execution percentage, the more advantageous GPU offloading becomes due to their sheer processing power.

Algorithmic design is also important in the same sense; inefficient work group reduction can artificially inflate device execution time on any device. Similarly, the final reduction performed by the host device was not shown to have significant impact on the target device (GPU) runtime percentage as the problem size increases. It is noted that there exists alternative techniques to perform this reduction, such as a secondary GPU kernel.

Finally, we can leave developers with a bit of advice: understanding the underlying target hardware is integral to achieving high performance parallelization code. In this paper, we have specifically shown that the number of processing cores has a fundamental and direct impact on design decisions such as work group sizes; thus the application should be tailored to the specific heterogeneous system it executes on. These parameters can be determined empirically with some guided intuition based on the problem at hand for a solid initial starting point.

VII. Future Work

During our implementation we experienced some shortcomings and challenges. Although OpenCL is supported by a consortium of hardware vendors, AMD ceased including the ability to send OpenCL kernel code to its CPUs with its latest drivers. Instead of using a single threaded program, we intended to test the performance of using an OpenCL queue to send the kernel code to the CPU. This would have made the two programs more similar for measuring performance.

We found that synchronization within a workgroup requires additional processing overhead. Synchronization is achieved through OpenCL functions used inside the kernel code. OpenCL provides each thread with a unique id and includes a thread barrier function that halts all threads until each thread reaches the barrier. Synchronizing threads is necessary for summing the results of a workgroup [14]. [17] provided a template for our reduction code. Once each thread's result is calculated, the kernel code enters the reduction loop. Each loop combines the results of two adjacent threads.

For example, consider four threads: 0, 1, 2, and 3. During the first reduction loop, thread 0 combines the results of 0 and 1. Thread 2 combines the results of 2 and 3. A synchronization barrier stops any thread from continuing until both combinations are completed. Once all threads reach the barrier, the second loop starts and thread 0 combines with 2 to get the total result. It would be informative to measure how the reduction loops impact processing time.

Further experimentation on the final work group reduction done by the CPU is needed. It did not end up dominating the total run time as expected. This experimentation could take the form of greatly increasing the step size (and thus total run time) to observe performance impact or perhaps implementing other reduction techniques such as a secondary GPU kernel.

Additionally, we did not compare against multithreading on the CPU. Our results reflect both the benefits of parallelization and the GPU hardware. This does not discredit the results of our experiment. Although a multithreaded CPU program would outperform a single threaded program, the GPU would still outperform the CPU with large data sets. The tipping point of our experiment would require more steps but, our results would maintain a similar conclusion.

Another improvement that could be made is the collection of execution time metrics. We used the C++ STL's high resolution clock to take time stamps and then calculated differences between the timestamps. Feature rich benchmarking software such as mixbench [18] could provide further insight into performance.

Lastly, in regards to the matrix multiplication experiment specifically, two improvements should be made. First, we should use floating point values as opposed to integer values. This will better demonstrate the benefit of using a GPU device over CPU. Second, we need to determine the cause for the spikes in values. While we generally do not think they jeopardized the integrity of our conclusion or results in general, the issue should be addressed and appropriate mitigation taken (run across multiple hardware sets, fix kernel bugs, upgrade OpenCL, etc).

References

[1] N. V. Sunitha, K. Raju and N. N. Chiplunkar, "Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead," 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT), 2017, pp. 211-215, doi: 10.1109/ICICCT.2017.7975190.

[2] L. Wang, Y. -z. Huang, X. Chen and C. -y. Zhang, "Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environment," 2008 International Conference on Computer Science and Information Technology, 2008, pp. 228-232, doi: 10.1109/ICCSIT.2008.27.

[3] Y. Suzuki, Y. Fujii, T. Azumi, N. Nishio and S. Kato, "Real-Time GPU Resource Management with Loadable Kernel Modules," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 6, pp. 1715-1727, 1 June 2017, doi: 10.1109/TPDS.2016.2630697.

[4] Zhao, Y. (2017, February 23). GPU Programming CSInParallel Project. CSinParallel. Retrieved February 27, 2022, from https://csinparallel.org/csinparallel/modules/gpu_programming.html

[5] Sanders, J., Kandrot, E., & Dongarra, J. J. (2015). *Cuda by example: An introduction to general-purpose Gpu programming*. Addison-Wesley/Pearson Education.

[6] G. Özen, S. Atzeni, M. Wolfe, A. Southwell and G. Klimowicz, "OpenMP GPU Offload in Flang and LLVM," 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), 2018, pp. 1-9, doi: 10.1109/LLVM-HPC.2018.8639434.

[7] F. Wende, F. Cordes and T. Steinke, "On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering," 2012 Symposium on Application Accelerators in High Performance Computing, 2012, pp. 74-83, doi: 10.1109/SAAHPC.2012.12.

[8] L. V. Lucas Vespa, "Unraveling the Divergence of GPU Threads," 2018 International Conference on Computational Science and Computational Intelligence (CSCI), 2018, pp. 1398-1403, doi: 10.1109/CSCI46756.2018.00270.

[9] M. K. Yoon, K. Kim, S. Lee, W. W. Ro and M. Annavaram, "Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 609-621, doi: 10.1109/ISCA.2016.59.

[10] M. Awatramani, J. Zambreno and D. Rover, "Increasing GPU throughput using kernel interleaved thread block scheduling," 2013 IEEE 31st International Conference on Computer Design (ICCD), 2013, pp. 503-506, doi: 10.1109/ICCD.2013.6657093.

[11] Shuai Zhang, Tao Li, Qiankun Dong, Xuechen Liu and Yulu Yang, "CPU-assisted GPU thread pool model for dynamic task parallelism," 2015 IEEE International Conference on Networking, Architecture and Storage (NAS), 2015, pp. 135-140, doi: 10.1109/NAS.2015.7255234.

[12] G. M. Amdahl, "Computer Architecture and Amdahl's Law," in *Computer*, vol. 46, no. 12, pp. 38-46, Dec. 2013, doi: 10.1109/MC.2013.418.

[13] *Introduction to OpenMP.* (2013). *www.openmp.org*. Retrieved March 25, 2022, from https://www.openmp.org/resources/tutorials-articles/.

[14] Uppsala Programming for Multicore Architectures Research Center. (2016). *Introduction to OpenCL*. Retrieved March 26, 2022, from https://youtu.be/V4RfPfHQPC8.

[15] Khronos Group. (2021, November 19). *The OpenCL™ Specification*. Khronos Group. Retrieved March 26, 2022, from https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.h

[16] Wikipedia contributors. (2022, March 26). Matrix multiplication. In Wikipedia, The Free Encyclopedia. Retrieved 02:50, March 31, 2022, from https://en.wikipedia.org/w/index.php?title=Matrix_multiplication&oldid=1079431539

[17] Bailey, M. (2022, June 14). *Performing Reductions in OpenCL*. Retrieved April 4, 2022, from https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/opencl.reduction.2pp.pdf

[18] ekondis. (2022, January 1). GitHub - ekondis/mixbench: A GPU benchmark tool for evaluating GPUs on mixed operational intensity kernels (CUDA, OpenCL, HIP, SYCL). GitHub. https://github.com/ekondis/mixbenc