Requirements

- Overview of thetaPGU system (Hunter)
    - Specs, login sequence, filesystems, full-node vs single-node, job script vs interactive
- How to build and submit jobs(Experiment environment)
    - Tools, toolchain, modules, packages, libraries, etc
    - Cobalt scheduler
    - qsub, qstat
    - Module (lmod)
    - Tensorflow
    - Nsight systems (nsys)
    - Nsight compute (ncu)
    - Conda
    - Top (cpu stats)
    - Nvidia-smi (gpu stats)
- Overview of the problem that we are using for benchmarking (mini app diff eq solver)
    - Problem size selections (Hunter)
    - Due to issues, we decided to stay with mini app for the whole semester (rather than openfoam or pythonfoam or tensorflowfoam)
- Issues that we experienced
    - Bare metal vs in a container
    - Waiting for gpu allocations/maintenance
    - Modifying source code to run on ThetaGPU
- Met with Romit to get info (?)
- Overview of each of the benchmarking tools (smi (Hunter), nsys, perf, etc?)
    - What it does, what data does it collect, why it's useful to us
- GPU power data -- NVIDIA smi
    - What queries are used and what are the results(plot data & necessary analysis)
    - Include our job script and commands
- CPU-GPU data movement(running the app in both CPU & GPU) -- nsys
    - Plot data & analysis
    - Zhong needs CSV
- CPU cache info and time implementation(only in CPU) -- perf
    - Plot data & analysis

Split the workload

- Zhong: make all the plots in python (Hunter will do more writing to compensate)

Exploring AI-Enabled Benchmarks on ThetaGPU

Overview

This semester, we worked under the supervision of Dr. Zhiling Lan on the SEEr-Planning project. We were assisted by Melanie Cornelius and Hannah Greenblatt. As a team of five, our goal was to explore an AI-enabled computational fluid dynamics benchmark on heterogeneous systems at Argonne National Lab. To do so, we initially planned to collect data using three different benchmarks: The SDL AI Workshop Mini-App, PythonFOAM, and TensorFlowFOAM. Our heterogeneous system is Argonne's ThetaGPU. Throughout the semester, we primarily focused on learning how to use ThetaGPU and collecting benchmark data for the mini-app. Specifically, we are interested in data movement through the memory hierarchy, as well as between the CPU and GPU. The information will give us a better understanding of the performance of in-situ data movement, analysis, and visualization.

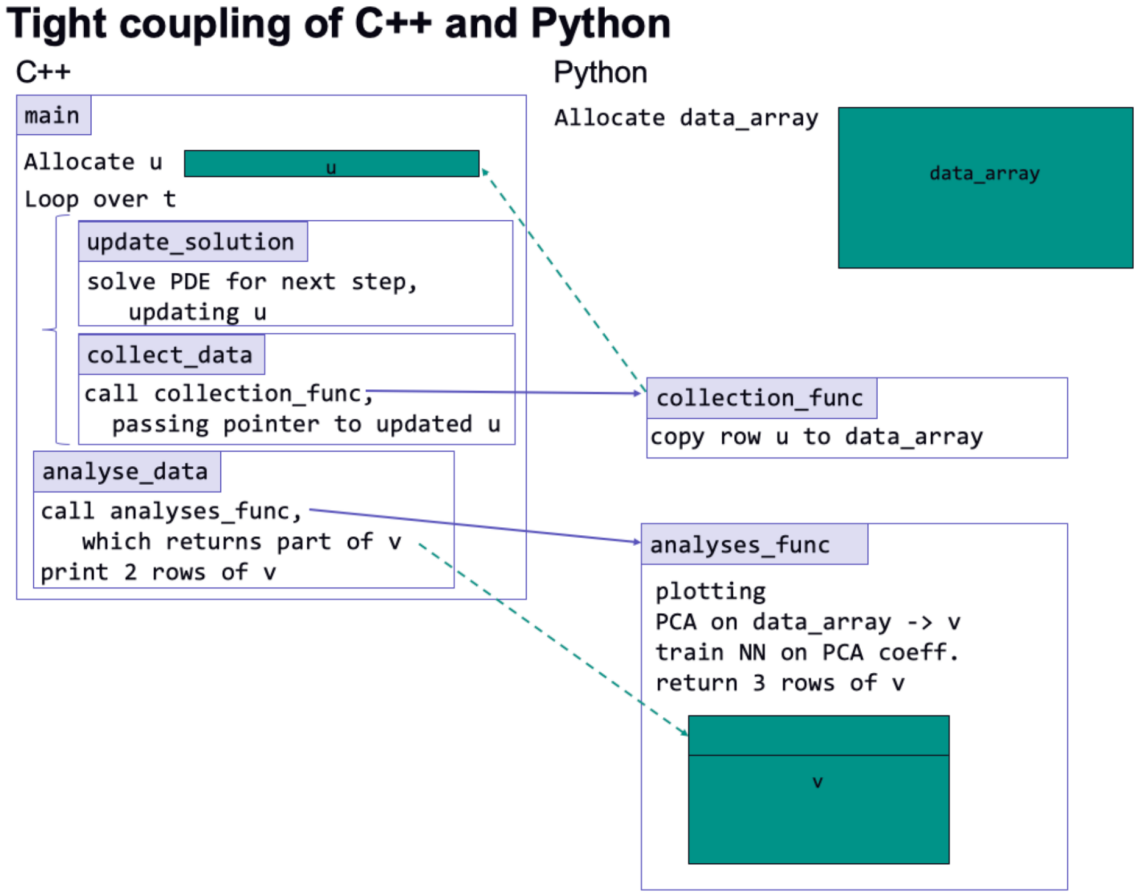
ThetaGPU is a collection of twenty-four NVIDIA DGX A100 nodes. Each node has eight NVIDIA A100 GPUs and either 320GB or 640GB of GPU memory, plus two AMD Rome 64-core CPUs and 1TB of DDR4 memory. Importantly, this is different from the Theta system, which is a Cray XC40 with Intel KNL compute nodes. We do not work with Theta on this project. In order to use ThetaGPU, we start by SSHing into the (shared) theta login nodes. These provide a landing point for all work done on Theta and ThetaGPU. From there, we SSH again into one of the ThetaGPU service nodes. This is where the bulk of work is done, including compilation and job submission.

To actually use ThetaGPU, an allocation must be received by submitting a job request to one of the queues. Cobalt is the scheduler used on ThetaGPU. To submit a job to Cobalt, we use a command such as the following.

```
qsub -I -A SEEr-planning -n 1 -q single-gpu -t 30
```

This requests a 30-minute interactive-style job for the SEEr-planning project with 1 single gpu. For testing purposes, the single-gpu queue is sufficient. One does not need to wait as long for a submission and less resources are used. The downside is that the ThetaGPU node will be shared among other single-gpu allocations. To run a full job or to collect real benchmark data, a full node is required. In this case, submit the job to the full-node queue. Oftentimes it is impractical to wait for an interactive allocation on a full node. Therefore, it is better to write a job script and submit it without the "-I" option. For each job, Cobalt will generate .cobaltlog, .output, and .error files. These are useful for checking the termination status of the job, the stdout, and the stderr of the program.

The Mini-app[1] is a ML_PythonC++ embedding benchmark written by Dr. Romit Maulik at Argonne. The graph shown below indicates how the mini-app connects C++ and Python.



While the details of its operation are unimportant, we know generally that it is an example of how a fluid dynamics solver would work. It is written using a combination of C++ and Python. The C++ generates data by performing a computation on the CPU, and then it is transferred to the GPU to build a surrogate model using TensorFlow in Python. This host-to-device data transfer is of particular interest to us, so we highlight it down in our analysis section. Additionally, the mini-app has a set of hyperparameters that must be set before running the algorithm. We focus on NX and DT, which we manipulate to compare the data collected against various problem sizes. NX specifies the number of points in spatial discretization, and DT specifies the time step (delta t). These together control the actual size of the problem. By selecting a range of values for each, we are able to understand how they affect the benchmarks.

While learning the ThetaGPU system and setting up the mini-app, we interacted with several critical tools and libraries in the ThetaGPU environment. First and foremost is the Lmod module system. It is a system that handles environment modules by strategically manipulating the PATH variable. We found this useful on a large machine

like this because it allows us to select specific packages and versions that are made available on the system when it is not feasible to install our own. We made much use of the “conda/tensorflow” module because the mini-app is built using TensorFlow. Besides using the Conda package manager to provide Tensorflow, we also set up a Python virtual environment to install cmake, matplotlib, and sklearn. These are all the other libraries needed to build and run the mini app.

We ran into a few issues this semester but were able to learn and overcome each of them. Firstly, the SDL AI Workshop presented instructions for running the mini-app in a Singularity container. However, the mini-app needs to be run on bare metal to collect accurate data. Therefore, we just had to make a few modifications to the cmake file to get it to work. This was just a matter of changing some environment variables and changing some include directories. Secondly, one of the main reasons why we chose to evaluate the mini-app is because we consistently faced major roadblocks to getting OpenFOAM working on ThetaGPU. After talking to support about compilation and talking to Romit about the merits of the mini-app, we decided we can get useful data without the struggle of OpenFOAM. This decision allowed us to make much more progress with data collection and analysis.

CPU Data Result

We used a number of profiling tools to collect the performance data of the mini-app. Melanie’s presentation “A Simple Overview of HPC Profiling” introduced us to several tools, which was helpful. We ran the mini-app and collected CPU performance metrics CPU-GPU mode using perf.

In our experiment, by executing

perf stat -x , -a -e instructions,l2_latency.l2_cycles_waiting_on_fills,xi_sys_fill_latency -I 1000 -o ${FILENAME}.perf

We collected three metrics in six different problem sizes of the mini-app. The corresponding NX sizes are 256, 512, 1280, 2560, 3840, 5120.

The descriptions of the three metrics showed as following

- Instructions: instructions executed per second in cycles
- l2_latency.l2_cycles_waiting_on_fills: total cycles spent waiting for L2 fills to complete from L3 or memory per second
- xi_sys_fill_latency: L3 Cache Miss Latency. Total cycles for all transactions divided by 16 per second

The results are shown below by plotting.

Perf plots

As we observed, increasing the NX value from 256 to 512 speeds up the running time of the mini-app. However, when we continue to increase the NX value, the total running time grows up with no surprises.
Speedup calculation in terms of the NX value:
Running time of size 256: 34 seconds
Running time of size 412: 18 seconds
So, the speedup is 1.89 which is a sublinear speedup regarding changing NX value from 256 to 512.
Also, we observed that for both three metrics, their “cycles per second” is very high in the beginning of the running time, and they tend to decrease dramatically. That is because the C++ part was running on the CPU side at the beginning, then the host transferred calculated data to the device where python code was running.

GPU Power Data Result

The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices. We use the nvidia-smi to monitor the power consumption of the mini-app for six different problem sizes as discussed above on the GPU side when the app is running on thetaGPU. The nvidia-smi power monitor records the power data of the app every second. The script is as follows.

```
nvidia-smi --query-gpu=timestamp,index,power.draw
--loop-ms=1000 --format=csv > smi-out.csv &
```

We are interested in how the GPU power consumption behaves along with the changing of problem size of the benchmark. Therefore, we used nvidia-smi to monitor the GPU power changes when we ran the mini-app with 6 different problem sizes with respect to the NX: 256, 512, 1280, 2560. 3840, 5120. The x-axis of the graph represents checkpoints in seconds, and nvidia-smi recorded the power consumption usage in every checkpoint. The y-axis represents the amount of GPU power consumed in Watt. The graphs below reflect the GPU power changes along with the time. There is a period of time where the power consumption is around 54.46W in each graph. That is because in that period of time, the C++ code part is running on the CPU side, so the GPU power consumption doesn’t have significant change. When the Python part starts to run on the

GPU side, the GPU power consumption increases dramatically. Below is the average GPU power consumption for each problem size:

- Size 256: 58.88 W
- Size 512: 59.05 W
- Size 1280: 59.35 W
- Size 2560: 60.04 W
- Size 3840: 60.18 W
- Size 5120: 59.71 W

It is expected that the power consumption increases with increasing problem size, because it requires more data calculation and movements. However, there is an expectation in our experiment that the power consumption of size 5120 is slightly less than the size of 2560 and 3840. By calculating the data movement rate(memset, HtoD, DtoH, DtoD) which is represented as

total gpu-mem-data / total gpu-mem-time

we found that the data movement rate of size 2560 and 3840 is 3.4E-07 MB/ns, while the data movement rate of size 3840 is 2.9E-07 MB/ns. The data movement rate of the size 5120 is slightly smaller. So, we suspect that this can be one of the reasons why the power consumption of size 5120 is slightly less than size 2560 and 3840.

Another interesting point is that in the size of 2560, 3840 and 5120, the peak GPU power consumption is much higher than the size of 256, 512 and 1280. We suspect that is because instant data movement at some points is extremely higher than other points. The memory data in the table below proves our idea. In the Max (MB) column, the max data movement at some points are much higher in the problem size 1280, 3840 and 5120, especially the size 5120.

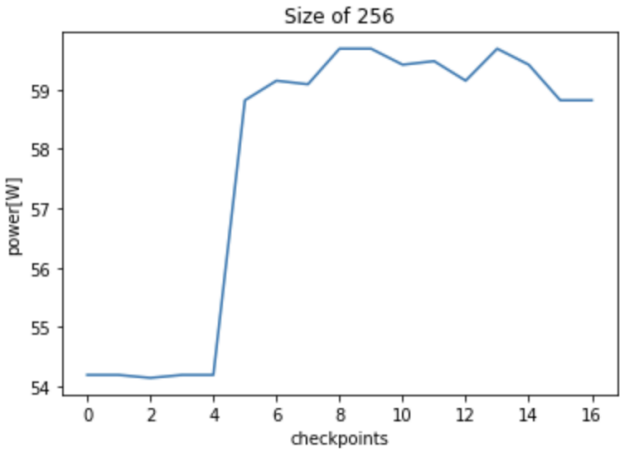
	2560						
Total (MB)	Count	Avg (MB)	Med (MB)	Min (MB)	Max (MB)	StdDev (MB)	Operation
61.083	53446	0.001	0	0	0.216	0.013	[CUDA memcpy HtoD]
34.555	45598	0.001	0.001	0	0.001	0	[CUDA memcpy DtoD]
25.206	605	0.042	0.044	0	0.082	0.033	[CUDA memset]
8.269	10690	0.001	0	0	0.04	0.005	[CUDA memcpy DtoH]
	3XXX						
Total (MB)	Count	Avg (MB)	Med (MB)	Min (MB)	Max (MB)	StdDev (MB)	Operation
61.083	53446	0.001	0	0	0.216	0.013	[CUDA memcpy HtoD]
34.555	45598	0.001	0.001	0	0.001	0	[CUDA memcpy DtoD]
25.206	605	0.042	0.044	0	0.082	0.033	[CUDA memset]
8.269	10690	0.001	0	0	0.04	0.005	[CUDA memcpy DtoH]
	5XXX						
Total (MB)	Count	Avg (MB)	Med (MB)	Min (MB)	Max (MB)	StdDev (MB)	Operation
605.151	503446	0.001	0	0	2.16	0.044	[CUDA memcpy HtoD]
322.555	405598	0.001	0.001	0	0.001	0	[CUDA memcpy DtoD]
48.337	100690	0	0	0	0.216	0.01	[CUDA memcpy DtoH]
25.206	605	0.042	0.044	0	0.082	0.033	[CUDA memset]

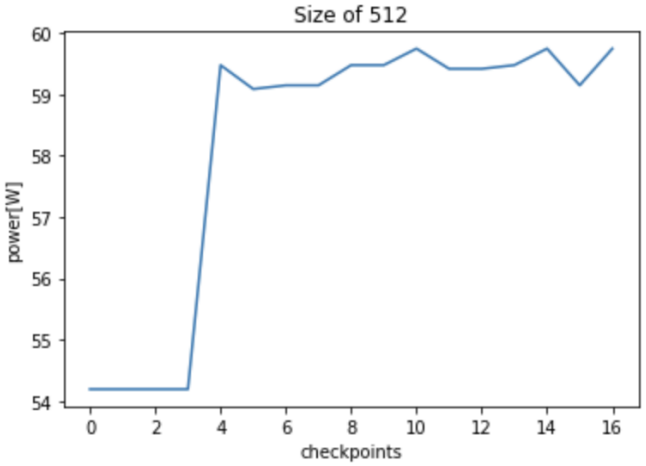
Time (%)	Total Time (	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
56.2	210063510	45598	4606.9	4608	4384	24959	161.9	[CUDA memcpy DtoD]
36	134771625	53446	2521.6	2432	2303	187327	1403.5	[CUDA memcpy HtoD]
7.1	26627536	10690	2490.9	2432	2336	5504	242	[CUDA memcpy DtoH]
0.7	2455349	605	4058.4	4832	2368	5408	1136.9	[CUDA memset]

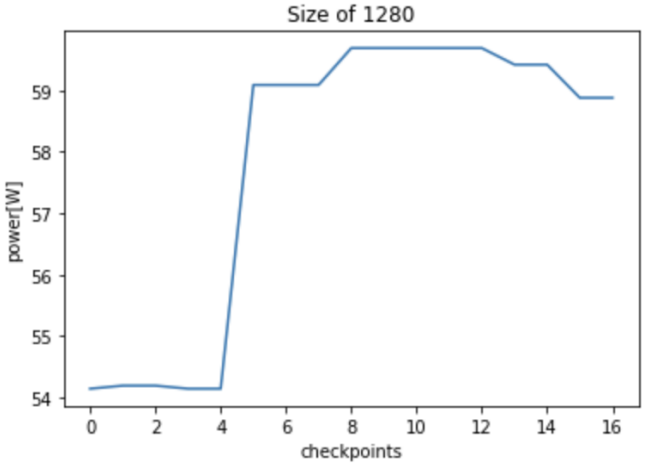
Time (%)	Total Time (	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
56.4	210037421	45598	4606.3	4608	4352	12575	101.2	[CUDA memcpy DtoD]
35.8	133406523	53446	2496.1	2432	2272	15680	829	[CUDA memcpy HtoD]
7.2	26653719	10690	2493.3	2432	2336	9184	251.7	[CUDA memcpy DtoH]
0.7	2454699	605	4057.4	4832	2368	5664	1136.3	[CUDA memset]

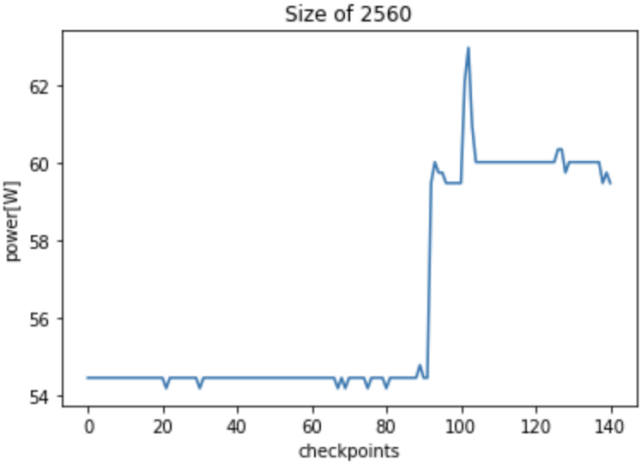
5120

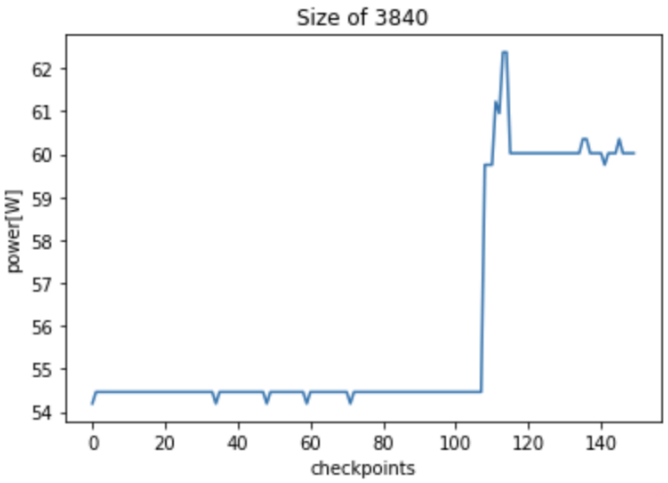
Time (%)	Total Time (	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
55.3	1890632657	405598	4661.3	4672	4352	199615	736.7	[CUDA memcpy DtoD]
37.4	1279557724	503446	2541.6	2463	2272	202687	3565.1	[CUDA memcpy HtoD]
7.2	245934684	100690	2442.5	2400	2303	13247	434.9	[CUDA memcpy DtoH]
0.1	2524543	605	4172.8	4960	2336	6112	1225.2	[CUDA memset]

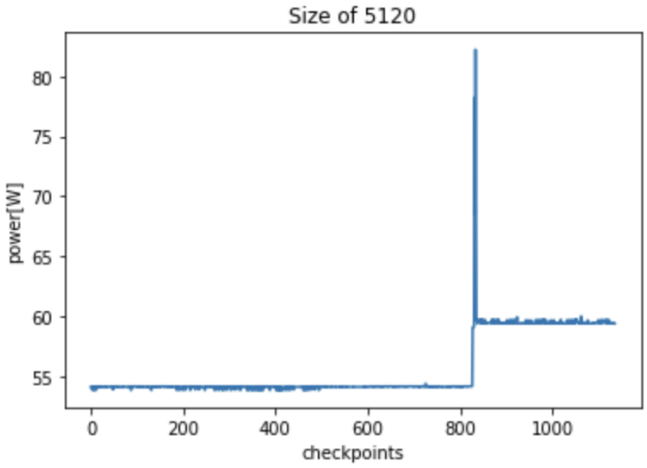












GPU Memory Data Analysis

Nvidia Nsight Systems is a system-wide performance analysis tool that collects detailed GPU runtime information. Here is the command used to run this on ThetaGPU. Note that when requesting a ThetaGPU allocation to run this, the `--attrs="perf=true"` must be specified so the perf paranoia level is appropriately set to 0 when running.

```
nsys profile --gpu-metrics-device=all -o ~/nsys-out ./app
```

Since we are mostly interested in data movement for this project, we highlight the relevant statistics in the series of tables below. Notice that for each data movement type (Host-to-Device, Device-to-Device, and Device-to-Host), the quantity of data moved increases proportionally with the number of steps taken by the algorithm. This is specified by the DT parameter, which indicates the step size. It is also worth noting that the percentage of the overall runtime that each movement operation takes is fairly consistent across problem sizes. Therefore, we have a solid basis for what to expect in terms of runtime for each of them, given the problem size. Interestingly, we don't immediately see a correlation between the NX parameter of the problem size and the size or runtime of the movement of data. Of course, NX and DT are correlated by nature of the problem, but their impact on performance differs slightly.

Host-to-Device Data Movement				
NX	DT	HtoD size (MB)	HtoD time %	HtoD time (ns)
256	0.001	6.676	29.3	22018785
512	0.001	6.676	28.9	21226008
1280	0.001	6.676	31.3	22045571
2560	0.0001	61.083	36	134771625
3840	0.0001	61.083	35.8	133406523

5120	0.00001	605.151	37.4	1279557724

Device-to-Device Data Movement				
NX	DT	DtoD size (MB)	DtoD time %	DtoD time (ns)
256	0.001	5.755	61.4	46093649
512	0.001	5.755	61.8	45364044
1280	0.001	5.755	58.7	41311701
2560	0.0001	34.555	56.2	210063510
3840	0.0001	34.555	56.4	210037421
5120	0.00001	322.555	55.3	1890632657

Device-to-Host Data Movement				
NX	DT	DtoH size (MB)	DtoH time %	DtoH time (ns)
256	0.001	4.262	6	4507004
512	0.001	4.262	5.9	4345658
1280	0.001	4.262	6.4	4512697
2560	0.0001	8.269	7.1	26627536
3840	0.0001	8.269	7.2	26653719
5120	0.00001	48.337	7.2	245934684

Nsight Systems also provides a lot of details about CUDA API calls. We decided to highlight some of that data below, since we found some interesting trends related to the problem sizes. For example, for extremely small problem sizes (NX=256 and NX=512), a majority of the runtime is spent just calling CudaFree. Therefore, it overshadowed the quick runtime of CudaLaunchKernel by a considerable amount. On the other hand, the largest problem size passes a critical point where calls to CudaFree are not dominating the runtime. Instead, most of the time is spent executing CudaLaunchKernel, which makes sense because that is where the main work for solving the problem is done.

Relationship Between Calls to CudaFree and CudaLaunchKernel					
NX	DT	cudeFree (ns)	cudaFree %	cudaLaunchKernel (ns)	cudaLaunchKernel %
256	0.001	6979390756	65.5	914930021	8.6
512	0.001	6049313274	62.4	919640655	9.5

1280	0.001	8462017197	60	1081153868	7.7
2560	0.0001	6530913080	39.8	5221552457	31.8
3840	0.0001	5085010691	35.7	5530759653	38.8
5120	0.00001	4981846961	6.8	46960114363	64.1

Conclusion

The research we did this semester was both interesting and educational. It provided us with a great opportunity to learn how to use ThetaGPU. Thank you to Dr. Lan, Melanie, and Hannah for helping us with this work. We look forward to the opportunity for summer research.

References

1. MiniApp: https://github.com/argonne-lcf/sdl_ai_workshop/tree/master/05_Simulation_ML/ML_PythonC%2B%2B_Embedding/ThetaGPU
2. Getting Started on ThetaGPU: https://alcf.anl.gov/events/getting-started-thetagpu
3. NVIDIA Performance Tools For A100 GPU Systems: https://www.alcf.anl.gov/events/nvidia-performance-tools-a100-gpu-systems
4. Theta and ThetaGPU ALCF Reference: https://www.alcf.anl.gov/support-center/theta-and-thetagpu