Parallel Computation on Graphics Processing Units in Matlab

Author:
Yuri Murakami Glauthier

Advisor: Van P. Carey

Department:
University of California Berkeley Mechanical Engineering

Contents

1. Abstract

1.1 Introduction

I have always been intrigued in unique computers whether it was raw performance or exotic methods of cooling. I wanted to utilize Matlab, an advanced computation based programing language, that it used by almost every corner of academia. Recently my usage of Matlab has increased and matlab has allowed me to run simulations as well as optimization problems. Many calculations are not time consuming and can usually be done in a matter of seconds. However, I wanted to push the limits of Matlab and see what I can make this powerful calculator do. I wanted Matlab to be able to harness all the power I could give it and see what the limitations are.

2. Setup of the GPU cluster

1.1 Graphics Cards

The choice of graphics card was simple. It must have lots of memory but still being cost efficient to be able to supply enough compute power. Another limitation is that Matlab can only utilize CUDA technology, a parallel computing platform and API, proprietary to Nvidia, a graphics card manufacturing company. The graphics card that I chose was the Nvidia GTX 1080 Ti due to its high memory bandwidth as well as superb computational power of 11.3 teraflops per second (Nvidia) being the highest in its class for performance per dollar. The choice of graphics card influenced the rest of my components in my computer.

1.2 Interfacing of the Graphics Cards

I purchased 12 GPUs to use in my clusters, 6 per cluster. I then used two motherboards and two cpus to connect the gpus too. This allowed me to make 2 clusters. Each running their own instance of Matlab. Each GPU has 11 gigabytes of video ram and for windows to be able to interface with all 6 graphics cards, I enabled 4G decoding which allowed for more memory to be allocated to the GPUs. I also had to change the PCI express lane bandwidth allocated for each graphics card since the processors I purchased only had 22 lanes. This was needed to be done to be able to send data between the CPU and all of the GPUs. This was an issue down the road that I did not realize and would negatively affect performance(see PCI Express Latency).

I created a graphics card cluster with 12 GPUs. This cluster used at peak 4 kilowatts and would generate a lot of heat. An issue I encountered during the initial testing process was that each graphics card was boosting to a different clock speed due to GPU boost, a software that was hardcoded into the GPU (graphics processing unit) to maximize performance in correlation with temperature by changing the core clock of the graphics card. To remove this limitation and get consistent results, I water cooled all 12 GPUs so that I can maximize the clock speeds at a high level. This allowed me to disable GPU boost and choose clock speeds myself to remove inconsistencies, this also gave equal cooling to all the GPUs as before water cooling the GPUs in the middle of the Cluster would be the hottest while the GPUs on the outside had the coolest temperatures due to receiving fresh air.

3. Creating a GPU Oriented Matlab Benchmark

3.1 CPU parallel computing vs. GPU parallel computing

The idea behind choosing GPU computing between GPU computing is the type of workload. CPUs can provide fast calculations on high difficulty low iterations counts. GPUs shine in high iteration counts with lots of small calculations. This is because CPUs are made with few CPU cores that run at high clock speeds to be able to make these calculations. GPUs are made of thousands of cores or Nvidia calls them CUDA cores. To take full advantage of the computational power of the GPU in Matlab I chose to do a simple calculation with many iterations that can be distributed to all the CUDA cores

3.2 x = A\b

A\b is a simple matrix operation where A is an n by n matrix and b is an n by 1 matrix. Each value in the matrix will be chosen using rand() a command that will chose a random number between 0 and 1 as a double (Matlab data type).

```
A = Array(rand(n,n));   %n by n matrix for A
b = Array(rand(n,1));   %n by 1 matrix for b
x = A\b;                %dividing A from b
```

4. Different Types of Parallel Loops

4.1 Parfor Loops

My first method of understanding how parallel computing within Matlab was to see how Matlab handles parallel loops and distributes the workload between CPU cores. Using Parfor was very simple, it was the same as a normal for loop but instead of running one iteration at a time, Matlab handled the splitting of the workload and distributing it evenly to the CPU cores so that the loop would execute quickly so that it would not waste idle CPU cores. I was easily able to see the performance difference using a normal for loop and parfor. This worked by prioritizing the speed and time of the calculation while starving other processes on the computer. This is the code for running single operation at a time. The only difference here is that Matlab will automatically distribute the workload to the CPU. This is something that I want to test with my GPUs.

```
numberofsteps = 3000;
stepsize = 1;
parfor nstep = 1:numberofsteps
    n = nstep*stepsize
    A = rand(n,n);
    b = rand(n,1);
    x = A\b;
end
```

4.2 Parfor Loops and GPUs

I quickly learned that Parfor cannot be used with GPUs, and while I could write to the GPU memory the issue I encountered was that multiple CPU cores would try to utilize only one GPU's memory.

```
numberofsteps = 3000;
stepsize = 1;
parfor nstep = 1:numberofsteps
    n = nstep*stepsize
    A = gpuArray(rand(n,n));
    b = gpuArray(rand(n,1));
    x = A\b;
end
```

4.3 Single Program Multiple Data loops

SPMD (Single Program Multiple Data) allows the user to write a single program and assign the data to a worker called `labindex`. This allows for complex data algorithms to run in parallel on multiple workers. The challenging part about controlling the SPMD blocks is that you can

only run one program and that program can not change based on the worker due to its parallelism. Loops outside of SPMD loops are also slowed down due to workers being issued data multiple times rather than being distributed equally and can ruin parallelism.

```
spmd
    A = gpuArray(rand(n-1+labindex,n-1+labindex));
    b = gpuArray(rand(n-1+labindex,1));
    x = A\b;
end
```

4.4 Implementing SPMD and For Loops Simultaneously

To remove the issue of the lower performance of having a nested for loop I decided to place the SPMD block inside of the for loop. This allows the user to have control over each iteration of the SPMD loop and modify the parallelism or change a variable if needed. Having control over changing the SPMD block allows for easier programing while may have some drawbacks due to restarting a SPMD loop every for loop iteration.

```
spmd
    for n = nstart:workerswanted:nend
        A = gpuArray(rand(n-1+labindex,n-1+labindex));
        b = gpuArray(rand(n-1+labindex,1));
        x = A\b;
    end
end
```

5. Distributing workload based on difficulty

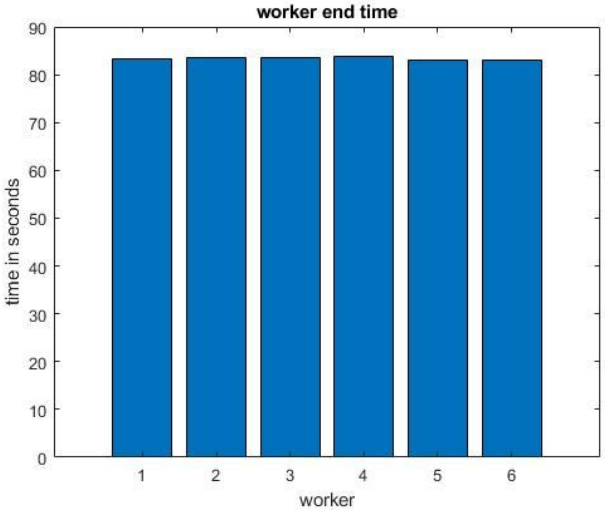
5.1 Linear versus Exponential Difficulty

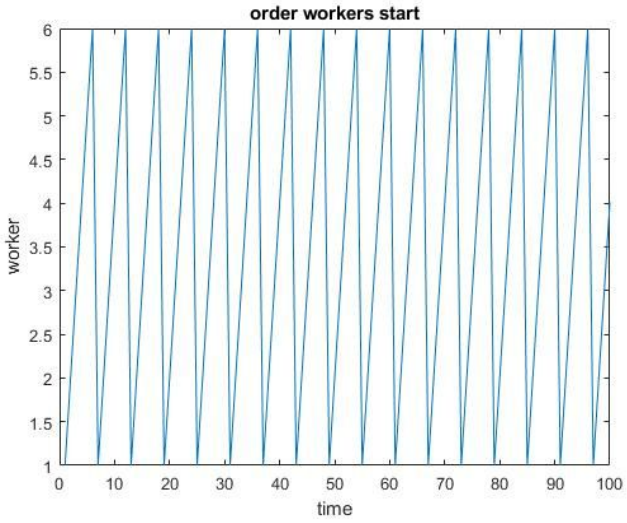
To distribute the workload evenly between workers, there needs to be a way to model the difficulty of a certain workload. To do so I simulated an exponential workload where every iteration becomes a linearly growing difficulty.

The basis of having the workload distributed is to minimize the ammount of downtime a GPU has aswell as how to make sure that when all the GPUs finish their tasks that they all finish at the same time. The model I created would also show how long each GPU is doing work and comparing the time it took to complete a task. This model was clearly shown when run on the GPU tests.

```
gputimes = zeros(1,6);
worker = zeros(1,1000);
iteration = 0;
```

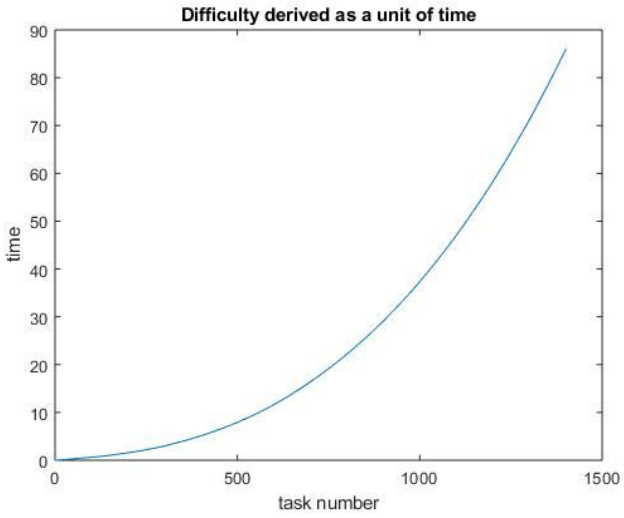
```
for seconds = 0.001:0.001:1
    iteration = 1+iteration;
   [lowestval,GPU] = min(gputimes);
   gputimes(GPU) = gputimes(GPU) + seconds;
   worker(iteration) = GPU;
end
```

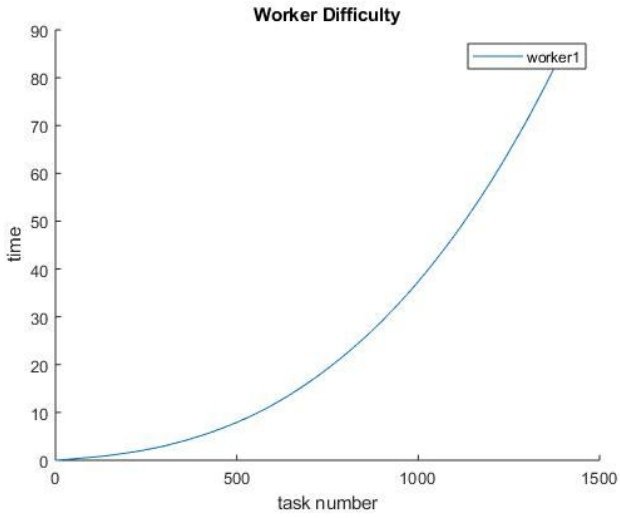


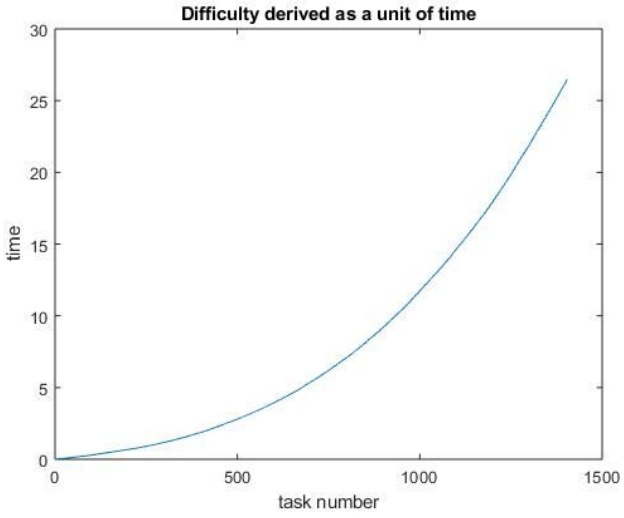


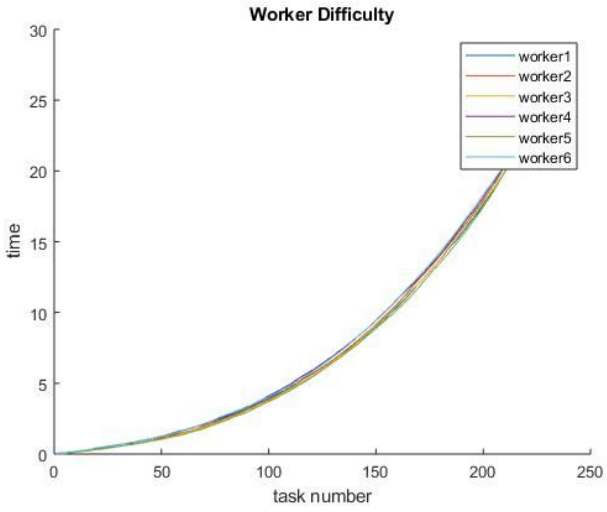
Here we can see how the work is distributed evenly among workers and how the order the workers start is in a sequential order from first worker to last worker. This code can also be used for different difficulty models.

6. Performance of Multiple GPUs









Here we can see how the performance of one GPU relative to the time it took to solve one of the iterations compared to how 6 GPUs were able to scale. Something interesting was that adding more GPUs allowed for scaling to be easily seen. Here when adding 6 GPUs we can see that we gained a performance increase of 3 fold. Looking at the difficulty derived as a unit of time graph we can see how the difficulty in relation to time has decreased meaning that the time to solve the benchmark was much faster with more GPUs. While the increase in performance was not a linear gain I was able to utilize as much of the performance that I could. To improve the performance of this I need to be able to split the matrixes up and distribute the workload to GPUs evenly and complete more operations at once rather than having the GPUs do all the work alone. While I was able to distribute the workload based on difficulty that is not all that needs to be done to have optimal parallel computing.

7. Sending Data Between Clusters

7.1 Data Transmission Between Clusters

Using the internet protocol TCP sharing data over a network is made very simple. I was able to send and receive data from cluster to cluster purely using Matlab code. This streamlined many things and made it easy for code to execute and send data. I could initialize the code that I was using for the size of the array I was solving for, as well as how many gpus I wanted to use. Receiving data was a little harder due to the issue of needing the size of the data. This would require some initialization but would not be too difficult. In my networking configuration I set one cluster to be the host and one to be the slave where the host would be at the ip address of 10.0.0.1 and would be the main hub for the data. To remove any bottlenecks of data transfers I used an ethernet cable to connect the two computers together.

Sending Data:

```
t= tcpip('10.0.0.1',25655,'Networkrole','client');
fopen(t)
data = logspace(1,10,1024); %data that would be sent would be written here.
for x = 1:length(data)
    fwrite(t,num2str(data(x)))
end
fclose(t)
```

Receiving Data:

```
t = tcpip(address,port);
lengthofread = datalength+1;
t.OutputBufferSize = 3000;
t.Timeout = 999999;
t.ByteOrder = 'littleEndian';
fopen(t);
address = '10.0.0.1';
port = 25656;
finishdata = 0;
datalength = 8;
for run = 0:iterations
    if run == iterations
        finishdata = 1;
    end
    data = fread(t,lengthofread,'char');
    index = run + 1;
    dataset(:,index)=[data(1:datalength)];
end
fclose(t)
```

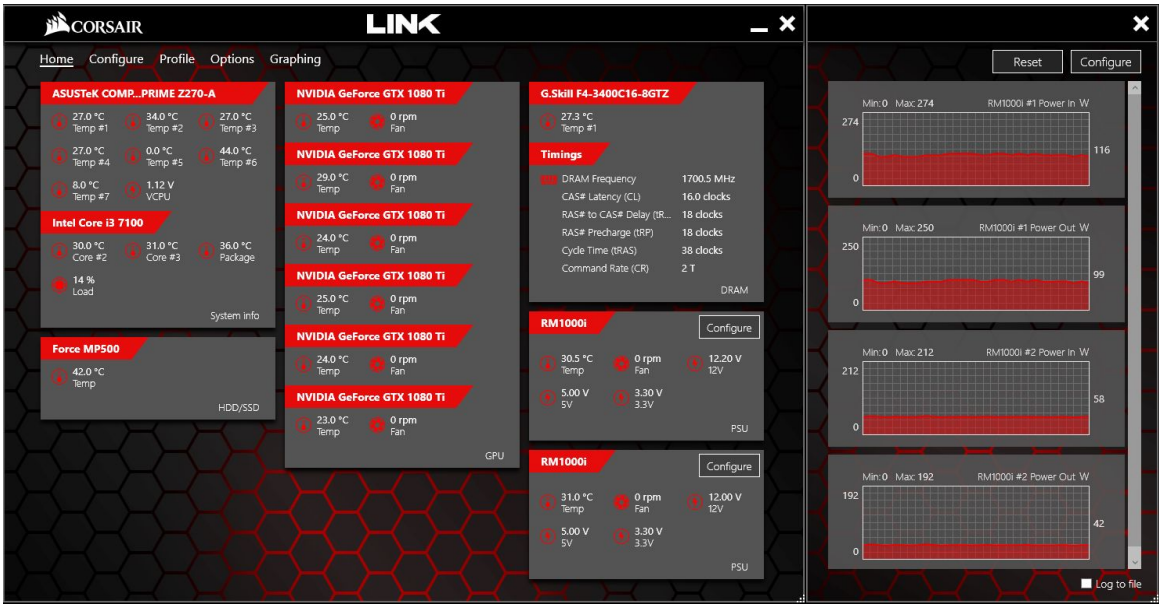
7.2 Performance of Multiple Clusters

The performance of multiple clusters is very linear do to less hardware limitations. As the data between the multiple Clusters is shared the bottleneck is how fast they can share the data amongst the other clusters. While this operation can be done simultaneously while running the GPUs the scaling is near perfect with my use of clusters. While in bigger environments it is evident how sharing data between clusters can be a challenge. The amount of data shared between the clusters does not matter for the next iteration in the solve therefore this is not an issue with A\b however if using an optimization model which needs the previous iteration (such as in controls) adding clusters can add a delay.

8. Energy Consumption of GPUs

8.1 Core Energy Usage versus Memory Energy Usage

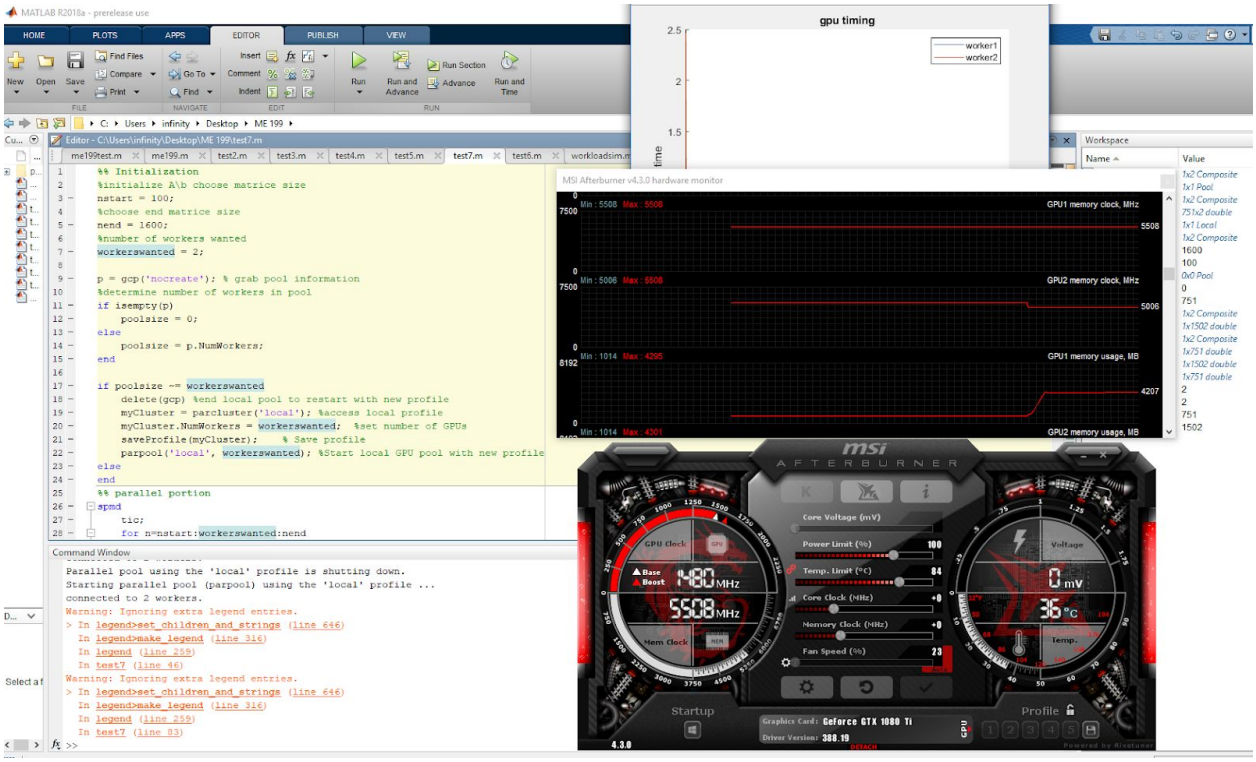
A\b's calculation was a mostly memory based benchmark which was one of the flaws in my design of a benchmark. A benchmark is suppose to be able to test many aspects of a system however mine was unable to show significant changes in energy usage. Using Corsair Link specialized power monitoring utility included with the power supplies I was able to take a baseline reading of power usage at idle which was about 130 to 150 watts. When running the benchmark I found that the power usage barely changed and with the degree of measurement I was able to collect with the Corsair Link memory based applications did not affect performance at all.



The graphs on the right are the power usage with power in and power out. I started my benchmark roughly in the middle of this graph however due to the design of my benchmark I was unable to detect this change. This also lead to the issue that since the amount of energy consumed was so low that measuring the energy efficiency to thermal efficiency of the computer was made near impossible.

8.2 Overclocking GPU memory

To be able to see a larger change in performance I decided to do the GPU memory overclocking using only two graphics cards on full PCI e x16 slot bandwidth to minimize any bottlenecks.



In this image we can clearly see when the benchmark began we can see that the memory usage on the graphics card shot up to about 4 GB of usage. The usage does not go down once the benchmark ends however because it is saved in the GPUs memory and will not disappear till the ParPool has been shutdown.

To determine the performance of the memory overclocking test I ran the test 10 times and found the average for the standard stock clock performance compared to a 204 Mhz increase to core clock (a 4.075% increase in memory clock). To find the best performance I decided to sum up the total time from the graph of gpu timing. This will add up the total time that it took both GPUs to complete the benchmark. This means that a lower score is better.

The stock clocks had a total T value of 4994.0 while the overclocked memory had a T value of 4899.5. This is a 1.9% increase in performance and for a small increase in memory clock a 1.9%

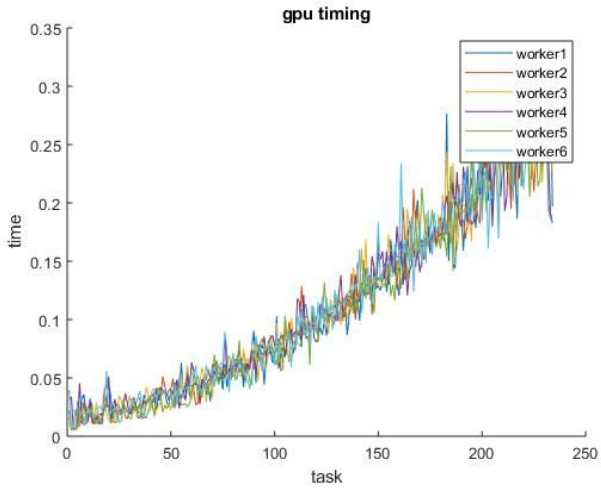
performance increase can go a long way. In more difficult tasks this extra performance compounds and speeds up general performance.

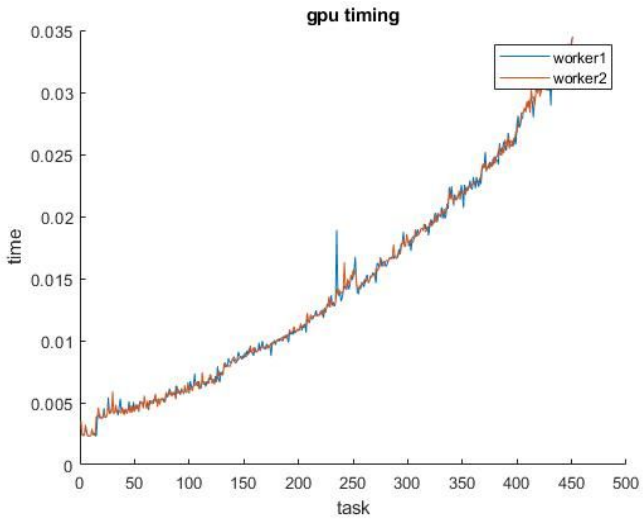
9. Hardware Issues

9.1 PCI Express Latency

One of the first issues that I realized would be a limitation of Matlab would be that when using a GPU to compute is the RAM access time difference between the CPU and GPU, even though a GPU would be faster at solving the equation the CPU would show a lower time because the GPU has to copy the data to the onboard video ram on the graphics card. We can see this when testing different sizes of arrays, while the CPU holds strong for the first few thousand sizes and is able to swap to ram faster at a certain point the GPU can solve faster than the CPUs swap time. This can be measured in the speed of how fast a GPU can solve a problem.

Another even bigger issue is the hardware interfacing of the graphics card with the motherboard. If comparing two computers one running with an inferior hardware interface we can clearly see how the larger bandwidth of the PCIe lane allows for roughly 10 times faster performance.





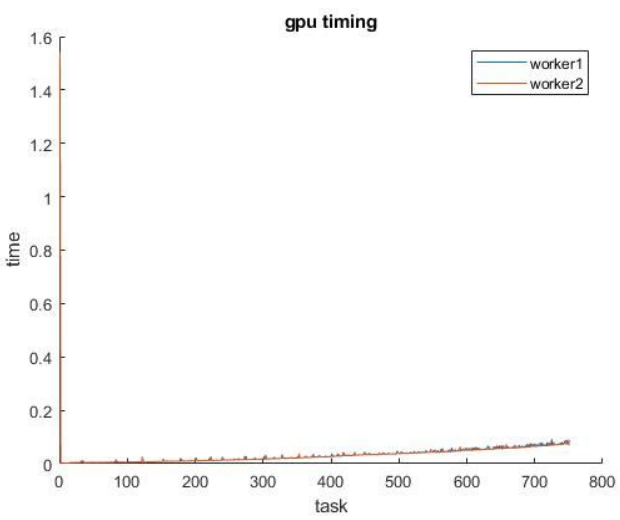
Here we can see how two computers with different hardware interfacing has different performance. Both have the same make and model as each other but with inferior performance due to the different in generation of PCIe as well as number of lanes connected to the CPU. The left side is using PCIe Gen.2 1x while the right hand side is using PCIe Gen.3 16x. The difference between these two and how much bandwidth they have is astonishing with a difference of 500 MB/s per card on the left while the right has 15.8 GB/s per card.

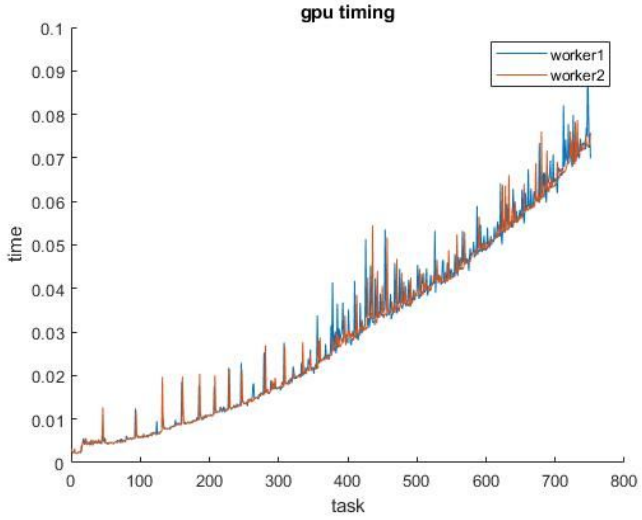
9.2 Temperature Differences Between GPUs

With GPUs generating a lot of heat there are issues when many GPUs are placed next to each other. To solve the heat issue of heat being transferred between GPUs I decided to watercool my GPUs to have equal temperatures across the board. This successfully allowed me to overclock the GPUs to equal levels at stable higher clock speeds.

9.3 Matlab ParPool Startup Time

When starting up the ParPool in Matlab one of the issues I encountered was that The parpool time would negatively influence the first iteration of the spmd loop. To avoid having large spikes in solve time for the GPU difficulty graphs I would always have to run the test twice once with the spike and afterwards it would go away. I believe that this is an initialization issue.





Side by side we can see how this issue can negatively affect benchmark numbers artificially slowing down the test results when in reality it is a perfectly good test. The total difference in test time would be roughly 1.5 seconds as we can see from the graph on the left.

10. Conclusion and Future Changes

While I was only able to scratch the surface of parallel computing I was able to learn a lot about how it works and where there are flaws in it. I was successfully able to make a benchmark based on the principle of A\b and would like to explore more options to create different types of benchmarks. In the future I believe that I will better be able to comprehend how a function needs to be broken down and distributed and even the parts of the function to be manipulated for easier computation for the computer. In the future I would change the model for determining the

difficulty of the function and rather than slowly ramping up the difficulty and have the GPUs struggle more towards the end I want to be able to pin the GPUs utilization to 100% all of the time so that no performance is wasted. While no GPUs had down time some workloads were lighter than others. I have also learned that independent studies are difficult and hard to complete when the topic chosen is obscure and not commonly referenced. I believe that this was an amazing challenge and would love to continue my work and bring parallel computing to other parts of Matlab as well.

11. Code used for Project

```
%% Difficulty generation
gputimes = zeros(1,6);
worker = zeros(1,1000);
iteration = 0;
for seconds = 0.001:0.001:1
    iteration = 1+iteration;
   [lowestval,GPU] = min(gputimes);
   gputimes(GPU) = gputimes(GPU) + seconds;
   worker(iteration) = GPU;
end
figure()
bar(gputimes)
title('worker end time')
xlabel('worker')
ylabel('time in seconds')
figure()
plot(worker)
title('order workers start')
xlabel('time')
ylabel('worker')
xlim([0 100])
%% Initialization
% initialize A\b choose matrix size
nstart = 1000;
% choose end matrice size
nend = 1450;
% number of workers wanted
workerswanted = 1;

p = gcp('nocreate'); % grab pool information
% determine number of workers in pool
if isempty(p)
    poolsize = 0;
else
    poolsize = p.NumWorkers;
end

if poolsize ~= workerswanted
    delete(gcp) % end local pool to restart with new profile
    myCluster = parcluster('local'); % access local profile
    myCluster.NumWorkers = workerswanted;  % set number of GPUs
    saveProfile(myCluster);    % Save profile
    parpool('local', workerswanted); %Start local GPU pool with new profile
```

```
else
end
%% parallel portion
spmd
    tic;
    for n=nstart:workerswanted:nend
        A = gpuArray(rand(n-1+labindex,n-1+labindex));
        b = gpuArray(rand(n-1+labindex,1));
        x = A\b;
        run = (n-nstart)/workerswanted+1;
        tgpu(run) = toc;
    end
end
%% simulation timing
% gpu times
figure()
for x = 1:workerswanted
    hold on
    plot(tgpu{x})
end
title('gpu timing')
ylabel('time')
xlabel('task number')
legend('worker1','worker2','worker3','worker4','worker5','worker6')
hold off
figure()
% task completion times
timemat = zeros(1,run{1}*workerswanted);
z = 0;
for x = 1:workerswanted
    time = tgpu{x};
    for y = 1:run{1}
        z = z+1;
        timemat(z) = time(y);
    end
end
sortedtimemat=sort(timemat);
plot(sortedtimemat)
title('Difficulty')
ylabel('time')
xlabel('task number')
hold off
%task difficulty
figure()
difficulty = zeros(run{1},workerswanted);
for x = 1:workerswanted
    timemat2 = tgpu{x};
    for r = 1:run{1}
        if r == 1
            difficulty(r,x) = timemat2(r);
        else
            difficulty(r,x) = timemat2(r)-timemat2(r-1);
        end
    end
end
hold on
plot(difficulty) ,
```

```
title('gpu timing')
ylabel('time')
xlabel('task')
legend('worker1','worker2','worker3','worker4','worker5','worker6')
hold off
```

Sources

"Geforce GTX 1080 Ti Graphics Cards |Nvidia Geforce." *Nvidia*, www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/.

"Documentation." *MATLAB Documentation*, www.mathworks.com/help/matlab/.