ECE 4822: Engineering Computation IV

Homework No. 9: Parallelizing Code Across Multiple GPUs

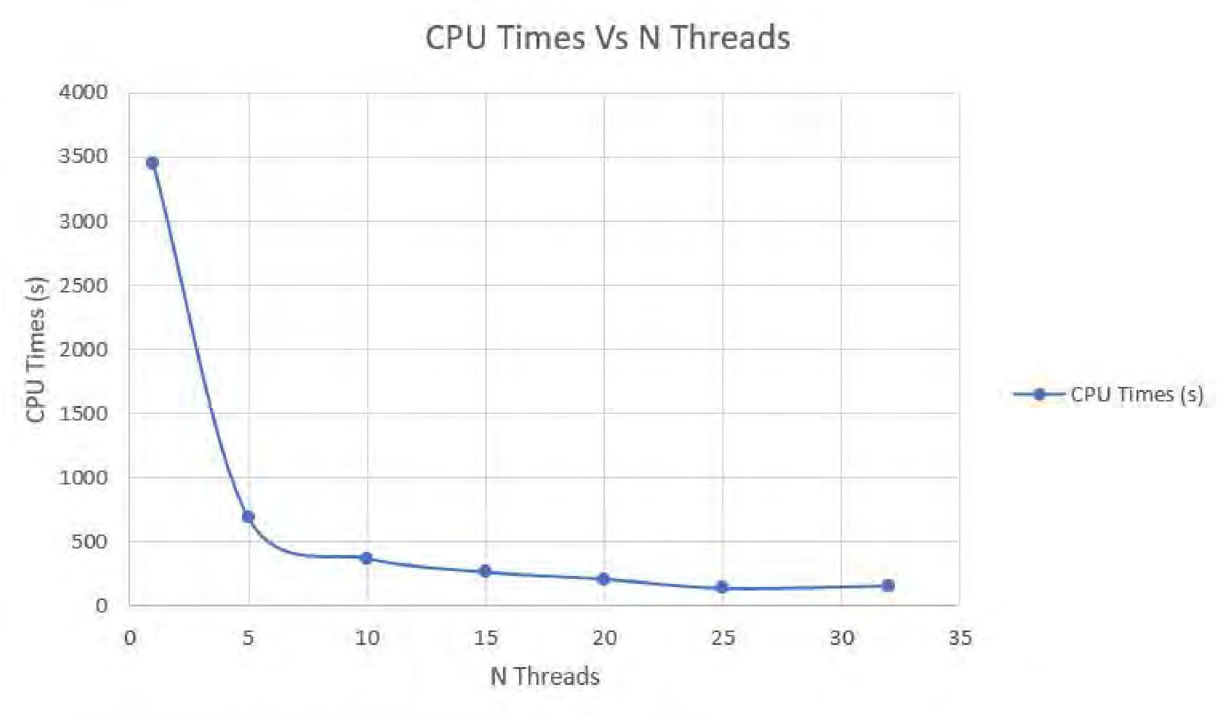
By Shahzad Khan

The goal for this assignment was to learn how to parallelize across multiple GPUs and compare the results to OpenMp parallelization.

The first task was to use OpenMp to parallelize matrix multiplication code to run using N threads and prepare a plot for $N = [1,32]$. This was relatively simple just applying OpenMp and using the built in "omp_set_num_threads()" command to limit the thread use:

```
bool multiMatrix(float* mat3, float* mat2, float* mat1, long nrows, long ncols, int threads){

  omp_set_num_threads(threads);
  if(!mat1 || !mat2 || !mat3){
    return false;
  }

#pragma omp parallel for
  for (long i = 0; i < nrows; i++) {
    for (long j = 0; j < nrows; j++) {
      float sum = 0;
      for (long k = 0; k < ncols; k++) {
        sum += mat1[i * ncols + k] * mat2[k * nrows + j];
      }
      mat3[i * nrows + j] = sum;
    }
  }
  return true;
```

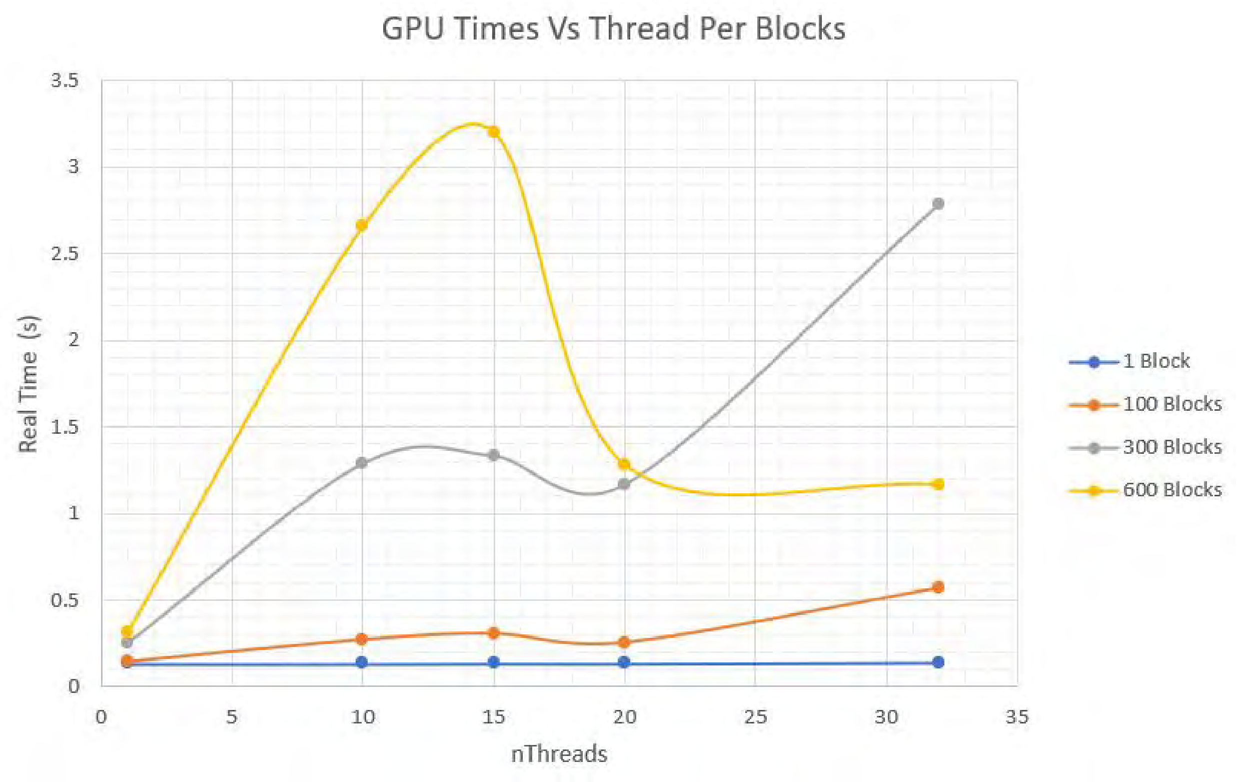
We are measuring the CPU time so in this case we can measure the real time from the bash time function. This is because the real time represents the full the it takes for the program to bring the results back to use, where as user and sys is the time the CPU spent directly on the program and the kernal performance respectively.



CPU Times Vs N Threads

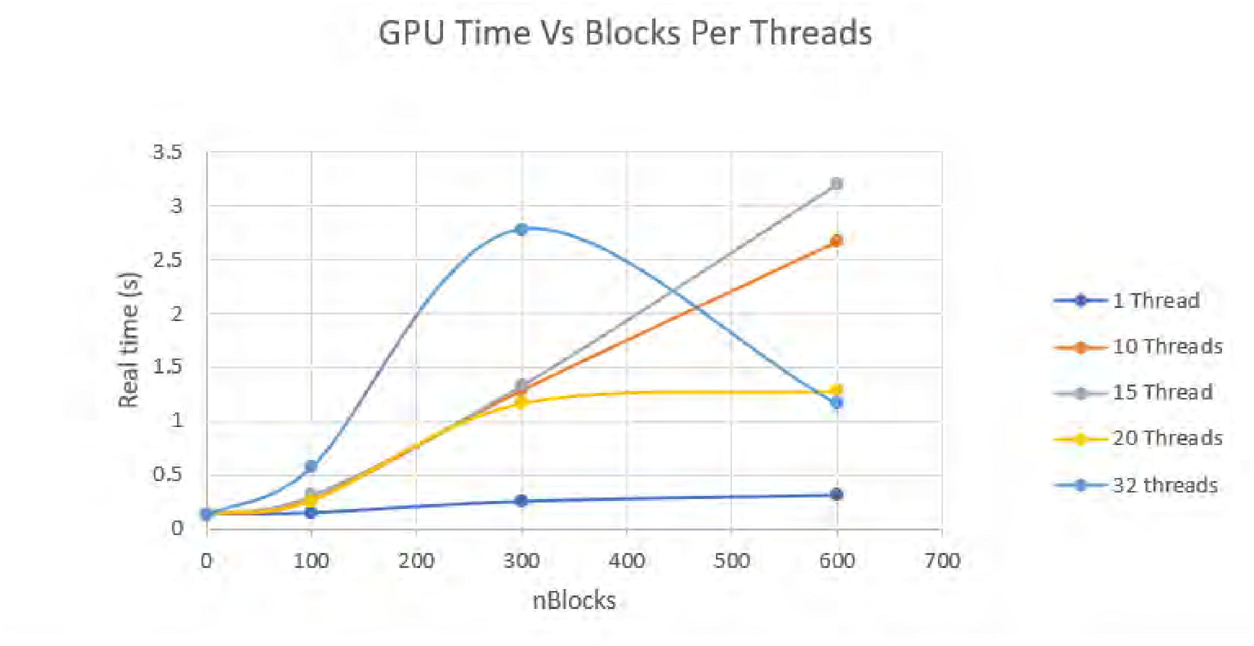
The above graph shows the relationship between the CPU time and the number of threads. As expected, the amount of time it takes for the program to run is cut drastically with the inclusion of multiple threads and decreases when it reaches 32 threads which is because the overhead of using too many threads starts to outweigh the positives.

Next the task was to parallelize our matrix multiplication using a single GPU while iterating on the number of threads and blocks. Then using the data find three values for the number of blocks and plot the time as the number of threads and vise versa for the number of threads. This again was relatively easy, just port a previous homework code over to this assignment and getting the results.



GPU Times Vs Thread Per Blocks

The graph above shows the relationship between the number of threads per block and the real time. With the number of threads for 1 block the amount of improvement was almost none, well as the number of blocks increasing. The most interesting relationship is with the higher thread counts, because the lower block sizes increase their time except for 300 blocks which shoots up switching positions with 600 blocks. Overall, this graph's output was expected, not counting the increase with 300 blocks.



GPU Times Vs Blocks Per Threads

The graph above shows the relationship between the number of blocks per the number of threads. Here we can see a similar shape to our thread per block graph but flipped. Where the number of threads themselves don't affect the performance as appose to blocks, only when it comes to once again our largest value, the runtime for 600 blocks and 32 threads beats out most of our iterations beside 1 thread.

The representative values that I chose are:

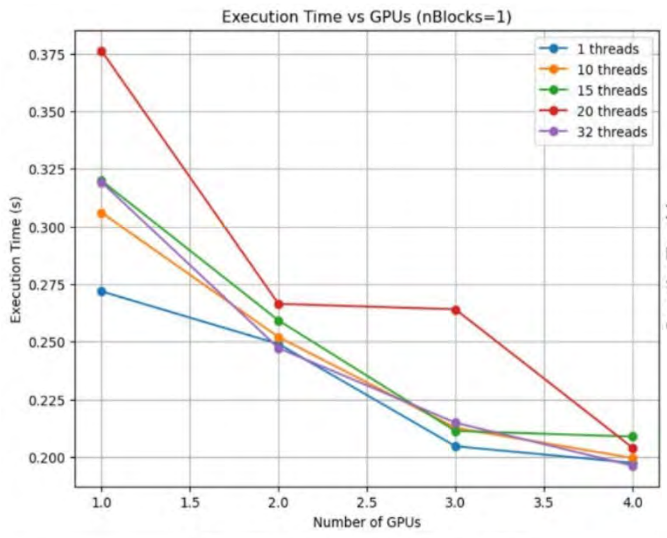
$$nBlocks = 1, 300, 600$$

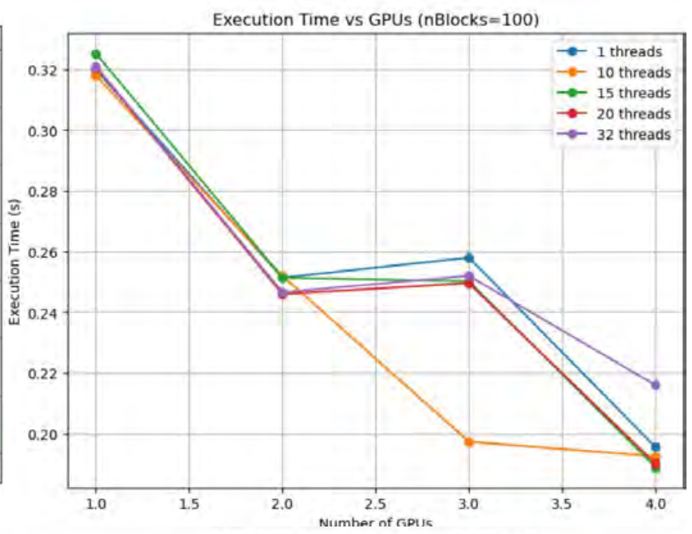
$$nThreads = 1, 15, 32$$

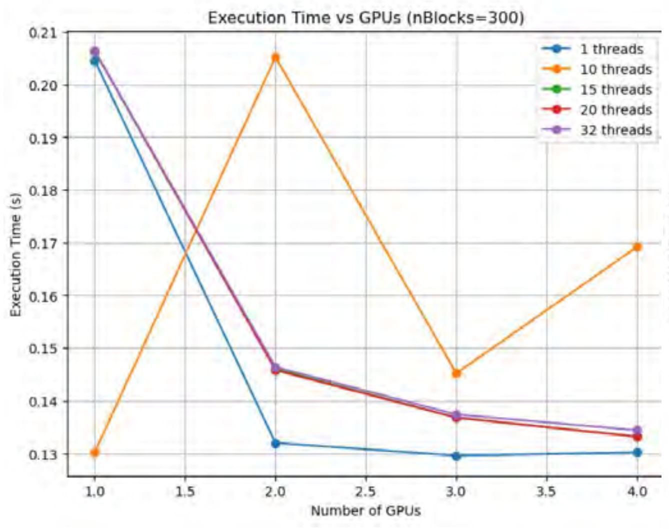
nBlocks	nThreads = 1	nThreads = 15	nThreads = 32
1	0.1344	0.1350	0.1368
300	0.2520	1.3314	2.7816
600	0.3132	3.1992	1.1670

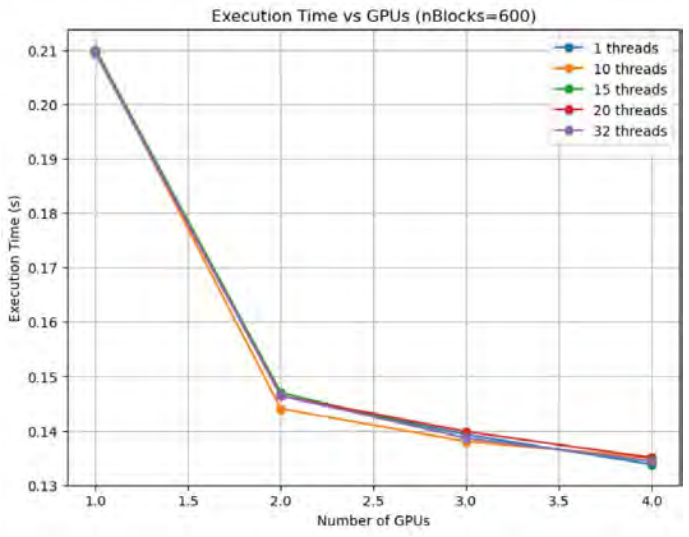
With the optimized times with a singular GPU found the next step is to allow for the use of multiple GPUs once again porting code from a previous homework assignment over.

```
int gpusToUse = std::min(deviceCount, gpuNum);
int rowsPerGPU = nrows / gpusToUse;
printf("Detected %d GPU(s), using %d\n", deviceCount, gpusToUse);


randomizeMatrix(A, nrows, ncols);
randomizeMatrix(A, nrows, ncols);


 for (int iter = 0; iter < niter; iter++) {
 // printf("\n--- Iteration %d ---\n", iter + 1);
 for (int g = 0; g < gpusToUse; g++) {

   cudaSetDevice(g);
   int startRow = g * rowsPerGPU;

   int nrowsA = (g == gpusToUse - 1) ? (nrows - startRow) : rowsPerGPU;
```









The graphs above show a interesting relationship between the time and the number of GPUs used. Which is expected, with the more GPUs used the faster the program becomes. For the most part this is shown in every graph, except for 100 and 300 blocks. 300 specificly using 2 and 4 GPUs shot the time to be higher than expected and almost the same time as just using 1 GPU. This is most likely because of a load imbalance where because of how the data is split between the blocks and threads it might not be split between the GPUs evenly which causes one GPU to have to wait on the other to finish, which causes large overhead which outweighs the speed increases of using mutliple GPUs. Its possible this is what is happening for 100 blocks as well.

# GPUs	nBlocks	nThreads	Real Time (s)	Notes
1	1	1	0.2718	Single GPU baseline
1	300	15	0.2064	Performance degraded due to under-utilization
1	600	32	0.209	Best single-GPU configuration
2	300	10	0.205	Load imbalance
2	600	32	0.1464	Faster than 1 GPU
3	300	1	0.1296	Fast, but inefficient thread usage
4	600	32	0.1344	**Chosen Optimal Configuration**

But overall it looks like the biggest speed increase obtatined is by using 4 GPUs with 600 blocks and 32 threads for a $10{,}000x10{,}000$ matrix. Which is the expected outcome, with the large matrix that we are using we need large blocks and thread to split the matrix evenly between the GPUs

The next task is to compare these times between the different GPUs between nedc_006, nedc_008, nedc_011, nedc_012.

Machine	nGPU	nBlocks	nThreads	Real Times (s)	Notes
nedc_006	4	600	32	0.9276	Runs slowest because of the old GPU
nedc_008	4	600	32	0.9084	A little faster but still slow because of old hardware
nedc_011	4	600	32	0.8458	Moderate speed older GPU but better than previous.
nedc_012	4	600	32	0.14932	Once again fastest because it's the newst hardware

In conclusion, nedc_012 is indeed the fastest GPU machine, but only when the workload is divided into sufficiently many blocks to keep all GPU cores active. When the number of blocks is too low (e.g., 300), load imbalance and idle GPU cores cause execution times to increase dramatically, sometimes exceeding the performance of older GPUs. Increasing the block count to 600 ensures even workload distribution and allows nedc_012 to achieve the best performance. Therefore, the optimal configuration does not change for threads or GPU count, but the optimal number of blocks must be increased on the newest hardware.