**NAME : RASHMI ANANTHAN**

**ID NO : A04165399**

**PROJECT 6**

*Introduction:*

This project involves heterogeneous parallelization of Travelling Salesman Problem using cuda ( for gpu ) and openmp ( for cpu ). The reason for using openmp over other API’s is that, openmp provides parallelization at higher level and also it minimizes the difficulty in writing the parallel programs. Cuda’s parallelizing capability of executing thousands of threads concurrently and in an efficient way made me to choose it as another paradigm. So far I have been implementing this problem on a single parallelization method. I wanted to find the effectiveness of using hybrid methods of parallelization which led me to choose openmp and cuda together in the same problem. I have found that this hybrid parallelization provides better runtime values than GPU parallelization but not better than CPU parallelization using openmp.

*Background Information:*

I had implemented the Travelling Salesman Problem using openMP earlier. The code scaled well and there were no parallelization overhead. It achieved a better speedup. When implementing the same problem using cuda, I observed that the two CPU’s in Stampede were faster than GPU and parallelization using GPU provided overhead due to main memory accesses. So the runtimes weren’t much better. Thus I am combining the two parallelization methods to see if they provide better runtimes than the two individual implementations.

*Project Implementation:*

Travelling Salesman Problem provides the shortest path of each city from the origin city given the list of cities and the distance between each pair of cities. The input size ( samples ) is divided among the two parallelization methods. The first half of the input size is given to a cuda function and the second half is given to an openmp function. In both cuda and openmp function, we parallelize the loop iterating the number of samples. In the case of OpenMP, we create a pragma omp parallel loop before iterate over the sample loop. This loop is iterated over the second half of input samples size. A local length value is calculated by each thread and then finally reduced to global length using critical section. In case of cuda part, the first half of the input size is passed to the CUDA kernel along with other parameters. Each thread is made to calculate a local length which is then finally reduced to global length using atomic function and this is done by thread0 in each block. The calculated length from both the functions is taken and the best value among the two is given as the final result.

*Measurement Evaluation:*

I have measured the runtimes for various input graphs and samples three times and took the best among them for evaluation. I have calculated two speedup’s with respect to the runtime from GPU and CPU column in the below runtime table. The whole project was executed on stampede and the batch file submitted along with this report has details of the appropriate queue, number of nodes, and number of tasks I have used in the submission script.

*Reference:*

CPU + GPU = hybrid parallelization using openmp + cuda

GPU = parallelization using cuda only.

CPU = parallelization using openmp only.

|  |  |  |  |
| --- | --- | --- | --- |
| *Runtimes:* | CPU + GPU | GPU | CPU |
| ts225 | 0.2192 | 0.4218 | 0.1686 |
| rat575 | 0.6009 | 1.1939 | 0.4149 |
| d1291 | 1.4893 | 2.9469 | 0.9195 |

*Speedup relative to GPU runtime taken from the above table:*

|  |  |  |
| --- | --- | --- |
|  | over GPU | over CPU |
| ts225 | 0.7572 | 1.9242 |
| rat575 | 0.6811 | 1.9868 |
| d1291 | 0.6161 | 1.9787 |
|  |  |

*Observation:*

The runtime was predicted to be better than CPU and GPU runtimes. But, these values for hybrid parallelization is two times lesser than GPU runtime and a little lesser than two times higher than CPU runtimes. The runtime increases as the input size increases which is as expected. The speedup table also shows the same.

In this project, half the set of samples which involve parallelizing using cuda will take more amount of time to compute the length because each thread end up accessing main memory every time. Always accessing DRAM is very slow and these accesses are not coalesced. This breaks the feature of gpu which works well when there is a coalesced memory access. However the runtime by the other set of samples executed by openmp provides a better runtime but this is equalized by the poorer runtimes when executed in cuda.

Had the problem of coalescing fixed, then the runtimes could have been way better than the CPU and GPU. Hybrid parallelization doesn’t have much overhead due to separate parallelization using CPU and GPU. This is because GPU kernel calls are asynchronous and they do not wait for GPU to finish and so in the mean time CPU continues processing. In this project we call cudamemcpy after calling the openmp function. Thus both parallelization method is executed in parallel.

*Summary:*

Hybrid parallelization using cuda and openmp was possible to implement and is better when the GPU’s features are utilized at its best level like coalesced memory access. When the amount of code or data to be parallelized is higher, then hybrid parallelization is of good use. Otherwise parallelization using CPU gives better results.

*Future Works:*

1. In this project, we are finding the length values from cuda and openmp; finding the best among them and concluding the best runtime as the result. Some mechanism can be used to find the length from both the methods.
2. There is no coalesced access of tour array as they are permuted randomly. They do not fix in the shared memory. Some fix for this would minimize the runtimes when executed.