Lyon Kee

keel@oregonstate.edu

Project #5: CUDA: Monte Carlo Simulation

Key snippets of code

```
30    #define NUMBLOCKS    ( NUMTRIALS / BLOCKSIZE )
```

Anatomy of a CUDA Program: Getting Ready to Execute 7

```
// setup the execution parameters:
dim3 grid( DATASET_SIZE / THREADS_PER_BLOCK, 1, 1 );

dim3 threads( THREADS_PER_BLOCK, 1, 1 );
```

Grid Size and Block Size

Line 30: As seen in cuda slides, the number of blocks is given by DATASET_SIZE/THREADS_PER_BLOCK.

```
151     float* dbeforey, * daftery, * ddistx;
152     int*   dsuccesses;
153
154     cudaMalloc((void**)(&dbeforey), NUMTRIALS * sizeof(float));
155     cudaMalloc((void**)(&daftery),  NUMTRIALS * sizeof(float));
156     cudaMalloc((void**)(&ddistx),   NUMTRIALS * sizeof(float));
157
158     cudaMalloc((void**)(&dsuccesses), NUMTRIALS * sizeof(int));
159
160     CudaCheckError();
161
162
163     // copy host memory to the device:
164
165     cudaMemcpy(dbeforey, hbeforey,  NUMTRIALS * sizeof(float),  cudaMemcpyHostToDevice);
166     cudaMemcpy(daftery,  haftery,   NUMTRIALS * sizeof(float),  cudaMemcpyHostToDevice);
167     cudaMemcpy(ddistx,   hdistx,    NUMTRIALS * sizeof(float),  cudaMemcpyHostToDevice);
168
169     CudaCheckError();
```

Line 154-158: we allocate memory on the device.

Line 165-167: we write data from host to device.

```
190     // execute the kernel:
191     MonteCarlo<<< grid, threads >>>( dbeforey, daftery, ddistx, dsuccesses );
192
193     // record the stop event:
194     cudaEventRecord(stop, NULL);
195
196     // wait for the stop event to complete:
197     cudaEventSynchronize(stop);
198
199     float msecTotal = 0.0f;
200     cudaEventElapsedTime(&msecTotal, start, stop);
201     CudaCheckError();
202
203     // copy result from the device to the host:
204     cudaMemcpy(hsuccesses, dsuccesses, NUMTRIALS * sizeof(int), cudaMemcpyDeviceToHost);
205     CudaCheckError();
```

Line 191: We run the kernel with the set grid and threads.

Line 204: We copy the results from device back to host to compute probability of successes.

Tables of data

Columns: # Trials

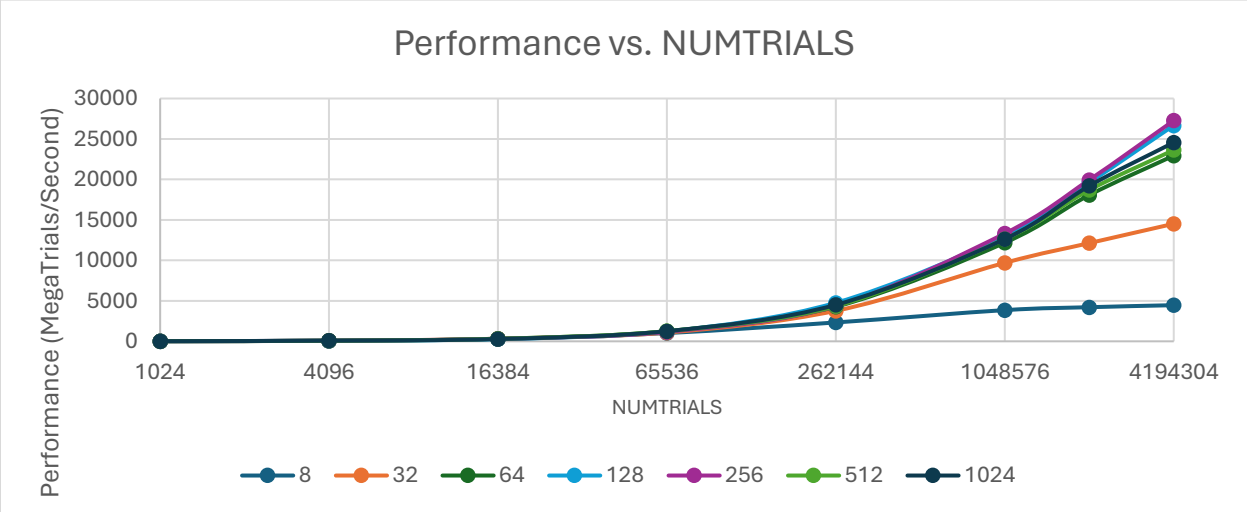
Rows: Block Size

Values: megaTrialsPerSecond

Sum of megaTrialsPerSecond	NUM Trials								
Block Size	**1024**	**4096**	**16384**	**65536**	**262144**	**1048576**	**2097152**	**4194304**	**Grand Total**
8	22.7273	74.0741	280.7017	1022.977	2332.5741	3857.3278	4216.1606	4464.9136	16271.4562
32	20.4082	67.7966	320	1085.3206	3769.9033	9700.4141	12136.296	14524.823	41624.9618
64	20.8333	75.4717	320	1254.1335	4280.0417	12185.9428	18073.9111	22946.778	59157.1121
128	20	66.6667	280.7017	1211.1177	4765.5612	13060.1833	19627.4331	26678.6082	65710.2719
256	16.6667	81.6327	320	1175.6602	4357.4468	13325.7416	19956.1505	27278.2516	66511.5501
512	20.4082	75.4717	320	1261.0837	4350.5045	12637.1001	18697.8607	23637.8716	61000.3005
1024	20	80	290.9091	1256.4418	4508.5306	12641.9756	19207.5035	24568.323	62573.6836
Grand Total	**141.0437**	**521.1135**	**2132.3125**	**8266.7345**	**28364.5622**	**77408.6853**	**111915.3155**	**144099.569**	**372849.3362**

Graphs of data

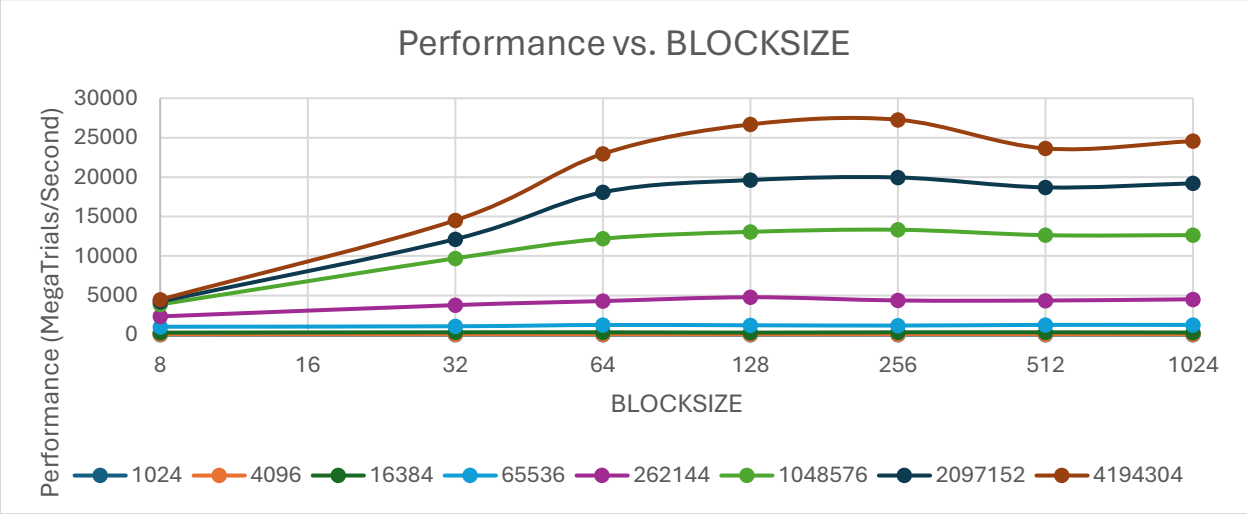
Graph of Performance vs. NUMTRIALS



The graph above shows that as the number of trails increases, the performance increases for each block size. This is obviously the case because as the number of trails increases, we have a higher

parallel fraction given that there are more work to do thus leading to a smaller overhead cost, keeping the GPUs busy to maximize resource usage. This is also due to a concept known as latency hiding as there are more threads to execute, it allows the GPU to switch to ready threads to hide the latency of memory operations. Lower number of trails it is simply underutilizing the available number of threads on the GPU.

Graph of Performance vs. BLOCKSIZE



The above graph shows that the block size that yields the highest performance is 256 with block sizes above and below having slower performance. This is because every GPU has its own optimal configuration and for the ones I am using, it turns out that 256 block size is the sweet spot, and any more or less will result in larger overhead costs coming from memory, utilization, and the granularity of parallelism.

PDF Commentary

1. What machine you ran this on

I ran this on the one Tesla V100 of the A100s that the class has access to, therefore I get dedicated CPU&GPU time and it is very reliable.

2. What do you think this new probability is?

Looking at runs with the highest number of trials, we can safely conclude that the probability hovers around 83.80285714 as the average of all the results with the highest number of trials.

3. Show the rectangular table and the two graphs

Shown in the above sections.

4. What patterns are you seeing in the performance curves?

The commentary is in the **Graphs of data** section above.

5. Why do you think the patterns look this way?

The commentary is in the **Graphs of data** section above.

6. Why is a BLOCKSIZE of 8 so much worse than the others?

The GPU is underutilized, just like the example given in class, a grape grabbing machine only using 8 sockets when there are more.

7. How do these performance results compare with what you got in Project #1? Why?

The optimal performance on project 1 is about 20 times slower than the optimal performance when using GPU given the same dataset size, however we observe that the CUDA version is able to perform faster in it's own optimal configuration.

8. What does this mean for what you can do with GPU parallel computing?

This means that with GPU, we can concurrently do more computations as quickly as possible with GPUs as they are always "ready". The number of speed up for non-GPU and GPU will depend on both the GPU's design and also the availability of the GPU in the system when the program is the same.

How did things turn out?

Things turned out very well with no blockages. The only thing that provokes thought is the pattern of the graph at a high number of block size.

Why do you think they turned out that way?

A higher number of block size can be caused by many reasons as stated the **Graphs of data** section above.