Homework 4- Raytracing using CUDA

Compile and running instructions:

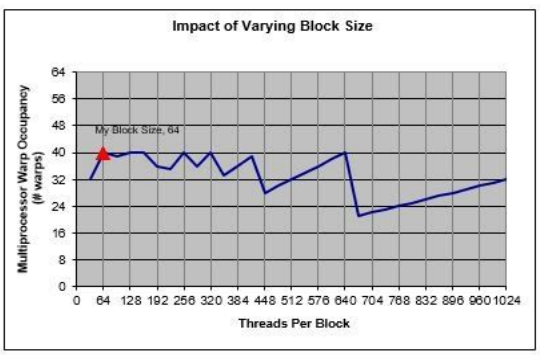
The code was compiled using the flags `--resource-usage`.
The code was executed on the peanut cluster (Titan) in interactive mode
The repo includes `CMakeLists.txt`.

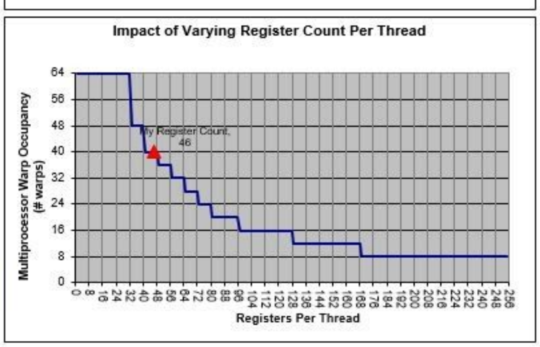
Functionality:

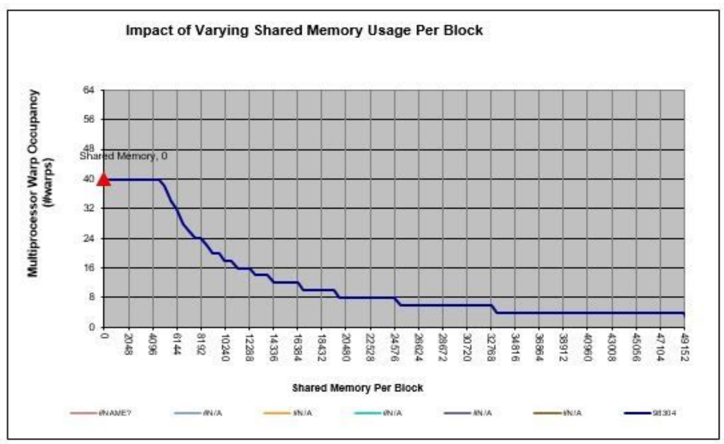
The repo contains two `.cu` files, `rayTracing.cu` and `bitmap.cu`. The latter is just bitmap.c with a changed extension. The code in the repo uses the algorithm specified in the document to simulate ray tracing on a sphere at a fixed distance from a light source at a fixed point. The output of the program includes the wall time for the runtime and a `demo.bmp` file that shows the illuminated sphere.

Optimum block size calculation:

Using the excel sheet from the Nvidia website, I calculated the optimum block size for my program, which used 46 registers per thread, as reported by `--resource-usage`. Prior to that, it used 48 registers, which is why there is a lot of commented out code in the file. For some reason, I couldn't get the register count down below that. The max occupancy was at 64 threads/block, at 63% occupancy. The occupancy could hit 100 if the register count went down.







CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	**6.1**	(Help)
1.b) Select Shared Memory Size Config (bytes)		
1.d) Select Global Load Caching Mode	L1+L2 (ca)	

2.) Enter your resource usage:		
Threads Per Block	64	(Help)
Registers Per Thread	46	
User Shared Memory Per Block (bytes)	0	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	**1280**	(Help)
Active Warps per Multiprocessor	**40**	
Active Thread Blocks per Multiprocessor	**20**	
Occupancy of each Multiprocessor	**63%**	

Physical Limits for GPU Compute Capability:	**6.1**
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	98304
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	2	64	32
Registers (Warp limit per SM due to per-warp reg count)	2	40	20
Shared Memory (Bytes)	0	49152	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	32		
Limited by Registers per Multiprocessor	**20**	**2**	**40**
Limited by Shared Memory per Multiprocessor	32		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 40 / 64 = 63%

Optimum Grid Dimensions test:

Since 64 threads was the block size that gave optimum occupancy, I used these values (8x8) with varying grid dimensions to see which grid dimension was the optimum one. The image size used for these tests was 4096x4096.

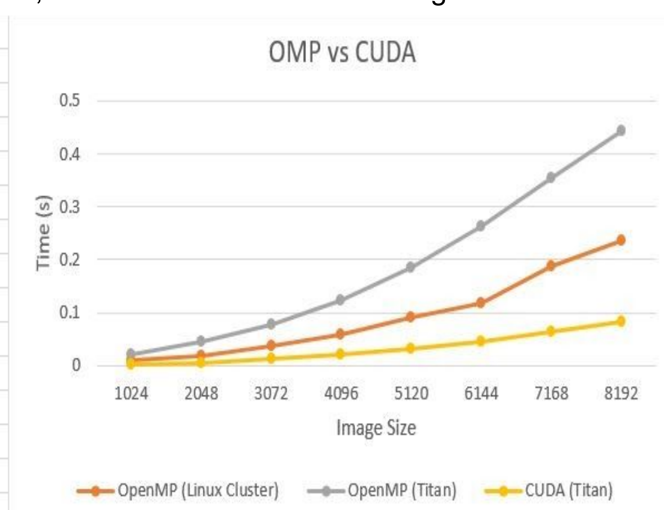
Grid size	Time (ms)
2x2	346.7
4x4	90
8x8	34.27
16x16	20.8
32x32	21.36
64x64	21.8
128x128	22.7
256x256	26.77
512x512	34.95
1024x1024	35.86

Evidently, 16x16 was the optimum grid dimension for this problem, as there is a slight increase in execution time from that point on. I expected a higher block count to be better but turns out, it's not.

OMP vs CUDA:

The next performance test was to compare the performance of the code from homework 3. From the graph in HW3, 24 threads gave the best performance. So I ran the program with 24 threads with varying image sizes (1024-8192, step size 1024) and compared them to the CUDA code (with optimum block and grid dimensions) with the same intervals. One thing I noticed was the OMP code gave starkly different execution times based on whether I ran it on the Linux cluster or on the Titan partition of the Peanut cluster, so I included both those timings as well.

Image Size	OpenMP (Linux Cluster)	OpenMP (Titan)	CUDA (Titan)
1024	0.009682	0.021603	1.46E-03
2048	0.019725	0.044199	5.41E-03
3072	0.037263	0.078503	1.19E-02
4096	0.05777	0.123256	2.08E-02
5120	0.090104	0.185394	3.23E-02
6144	0.11886	0.264085	4.64E-02
7168	0.188114	0.353153	6.29E-02
8192	0.235239	0.443543	8.19E-02



Observations: The OMP code ran on the Titan partition had the worst performance by a good measure. Linux cluster OMP version was second in terms of performance, and had quite a different performance from the same code on Titan. The CUDA version was the best though, and the rate of increase in execution time was also much slower than the other two versions.