Implemenation and Analysis of Tile Coloring Algorithm on CPU vs GPU

Joshua Smith
Utah State University
joshua.smith4@aggiemail.usu.edu

***Abstract*—This paper is an in depth analysis and discussion of the parallelization of a tile coloring algorithm implemented on a Tesla K80 Nvidia GPU. The serial algorithm is profiled and evaluated for the process of parallelization and then implemented two different ways. The first parallel implementation uses only GPU global memory and the second makes use of block shared memory. Each version is evaluated by its run time and speedup while varying the scheduling window size. The source code and results for this analysis can be found here: https://github.com/joshua-smith4/ConcurrentNets.**

I. Introduction

PARALLELIZATION is the process of separating out parts of a serial algorithm that can be run concurrently with others. A *general purpose graphics processing unit* (GPGPU) is specialized hardware that can run many threads simultaneously. In many areas of research and industry, massive speedup has been seen by parallelizing certain algorithms (e.g. sort, matrix multiplication, etc) on GPUs.

This paper focuses on the parallelization of the tile coloring problem. Two parallel implementation on a Tesla K80 Nvidia GPU demonstrate significant speedup over the serial algorithm. The first implemenation is rather naive with respect to memory accesses while the second utilizes block shared memory to further improve the algorithm.

II. Familiarization with CUDA and GPU Device

The first part of this project was getting familiar with the CUDA API for communicating with and launching kernels on the GPU. The results of the deviceQuery sample program shipped with CUDA detail the precise specifications of the GPU used and are included in Appendix A.

The following are a few key features to note about the device:

- 11440 MB Global Memory
- 49 KB shared memory per block
- 1024 threads per block
- Max thread block dimensions (1024, 1024, 64)

Another sample program that tests the memory bandwidth of the device was run and the results are included in Appendix B. Figure 1 shows the memory bandwidth for varying data transfer sizes from device to device, host to device, and device to host.

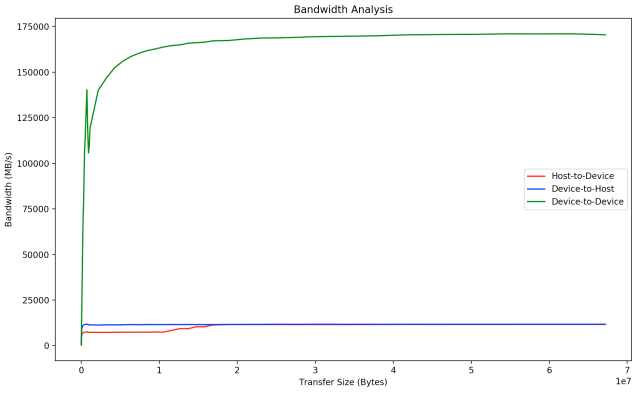

Fig. 1. Bandwidth measurements of varying data transfer sizes between devices and host.

III. CPU Analysis and Parallel Design

Per Amdahl's law, the potential speedup of an algorithm depends greatly on the size of the parallelizable portion. The tile coloring problem must be evaluated for speed up potential in order to justify the effort of parallelizing the portions that can be run concurrently. Figure 2 shows the amount of time spent per function on the serial algorithm. As can be seen, the function *findConcurrencyCPU* takes up about 98% of the algorithm's time. If this portion of code can be serialized, the resulting speedup would be very large and the effort is justified according to Amdahl's law.

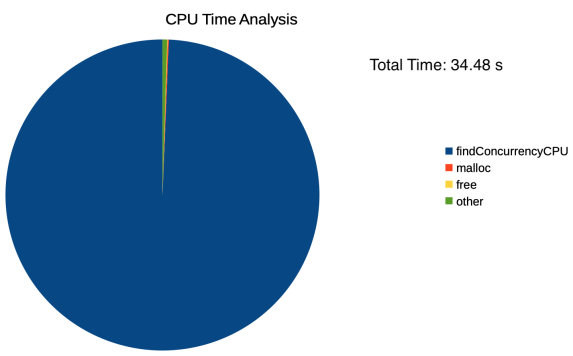

Fig. 2. Time analysis of serial algorithm executed on the CPU.

IV. Methods

The function *findConcurrencyCPU* essentially has two steps that take a lot of CPU time. The first is the coloring of the tiles. There is essentially zero data dependency between each iteration of the nested for loop structure that performs this function, making it easily and readily parallelizable. This was trivially done by alloting a thread on the GPU for each iteration of the for loop and allowing the thread to update the color of the tile associated with it. The next section goes into greater detail of how shared memory was utilized to further improve the performance of this section.

The second part of the function *findConcurrencyCPU* is essencially a histogram operation where a bin associated with a color or subnet is incremented as a nested for loop structure iterates over the now colored tiles. In general, the number of colors or subnets is much smaller than the number of tiles, meaning that, in a parallel environment, many threads would be attempting to increment the same memory locations. This introduces possible data races between threads that must be handled. This part is also parallelizable but several precautions have to be made in order to retain safety and increase efficiency.

The CUDA API has a function called atomicAdd which is an indivisible operation that increments the value at a provided memory location by the argument. This guarantees that no race conditions can occur between threads desiring to increment the same location in the histogram. This solves the race condition problem but introduces a bottleneck into the program. Because there are many more tiles than colors, in general, many threads attempt to increment the same bins simultaneously resulting in a lot of thread communication and waiting. This is undesirable as it decreases the effectiveness of the parallelization.

In order to overcome this issue, several copies of the bins were made and only certain threads, based on their indices, would were allowed to access each copy. By significantly increasing the number of memory locations, the probability of collisions sharply decreased, allowing the algorithm to continue approximately fully parallelized. After each copy was updated, the results were trivially gathered by another kernel whose objective was to accumulate the values in each bin copy.

Three kernel functions were used, each with the functions described above.

1) colorTiles - color the tiles
2) histCalc - calculate the histogram using many copies of the bin array
3) sumHist - accumulated the values of the distributed histograms

V. Shared Memory vs Global Memory Implemenations

As mentioned above, in the color the tiles section of the algorithm there are two arrays that all threads refer to often to evaluate the color of the tile. The first implementation stores these arrays, called a and b in global memory where each access is costly. The second implementation first copies these arrays into shared memory, synchronizes threads in the block, then uses the shared memory to evaluate the tile color. This change resulted in a the speedups shown in Figure 3.

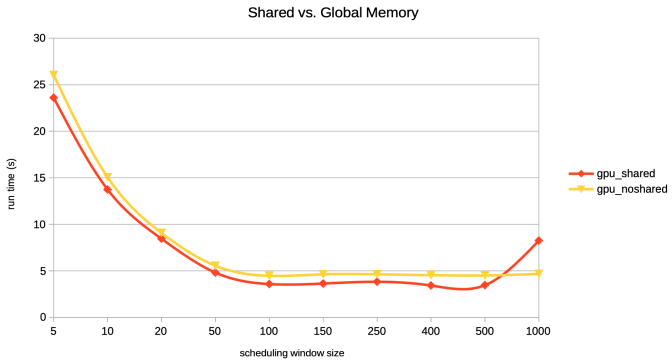

Fig. 3. Speedup analysis of both implementations: global memory and shared memory.

As can be seen in Figure 3, the two implementations perform similarly with the shared memory version consistently faster. This only changes at the end when the scheduling window size causes the two arrays, a and b, to significantly exceed the amount of shared memory proportioned to each block (49 KB available - 128 KB needed). This causes the inverted behavior observed at the scheduling window size of 1000.

Figures 4 and 5 show the time percentages taken up by different portions of code. As can be seen, these are significantly different than Figure 2. The CPU no longer takes up 98% of the algorithm's time. The GPU is able to perform much more work in a lot less time by leveraging millions of concurrent threads.

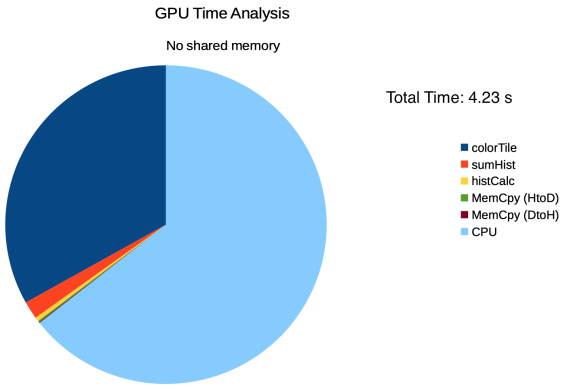

Fig. 4. Time analysis of GPU implementation without shared memory.

VI. Speedup Analysis

Figure 6 shows the CPU and GPU with shared memory implementations side by side with the associated speedup measurement with varying scheduling window size. As shown, the CPU performs well on very small window sizes. This is expected as the amount of parallelization is proportional to the window size. As scheduling window size increases, the GPUs ability to expoit parallelism becomes evident with

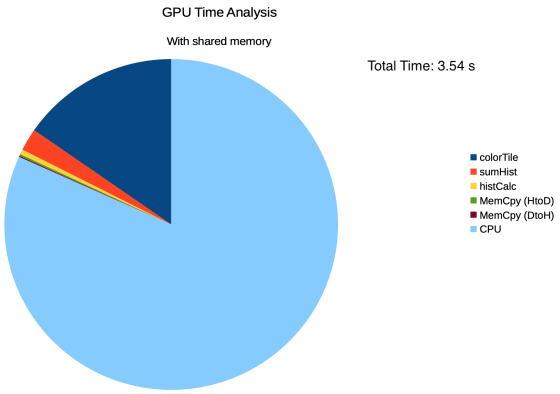

Fig. 5. Time analysis of GPU implementation with shared memory.

massive speedup over the CPU implementation. The run time of the CPU version seems to increase exponentially with scheduling window size whereas the GPU version run time remains relatively constant at large window sizes.

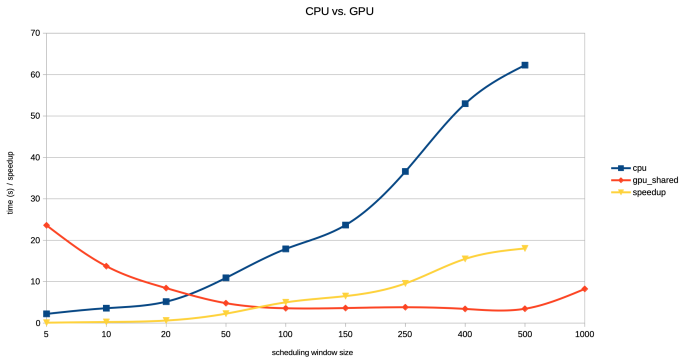

Fig. 6. Speedup analysis of CPU and GPU with shared memory implementations.

VII. Results and Conclusion

As shown by Table I, the GPU implementations provide significant speedup by efficiently exploiting the potential parallelism in the tile coloring problem. The use of shared memory further increases the efficiency by reducing the amount of time spent waiting for memory accesses.

TABLE I
RUN TIMES: SCHEDULING WINDOW SIZE OF 256.

CPU	GPU Global	GPU Shared
34.48 s	4.23 s	3.54 s

Appendix A
Device Query Results

```
./deviceQuery Starting...

 CUDA Device Query Runtime API version CUDART static linking

Detected 2 CUDA Capable devices

Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11440 MBytes 11995578368 bytes
  13 Multiprocessors, 192 CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz 0.82 GHz
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size x,y,z         1D=65536, 2D=65536, 65536, 3D=4096, 4096, 4096
  Maximum Layered 1D Texture Size, num layers  1D=16384, 2048 layers
  Maximum Layered 2D Texture Size, num layers  2D=16384, 16384, 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block x,y,z: 1024, 1024, 64
  Max dimension size of a grid size    x,y,z: 2147483647, 65535, 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engines
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing UVA:      Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 134 / 0
  Compute Mode:
     < Default multiple host threads can use ::cudaSetDevice with device simultaneously >

Device 1: "Tesla K80"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11440 MBytes 11995578368 bytes
  13 Multiprocessors, 192 CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz 0.82 GHz
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size x,y,z         1D=65536, 2D=65536, 65536, 3D=4096, 4096, 4096
  Maximum Layered 1D Texture Size, num layers  1D=16384, 2048 layers
  Maximum Layered 2D Texture Size, num layers  2D=16384, 16384, 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block x,y,z: 1024, 1024, 64
  Max dimension size of a grid size    x,y,z: 2147483647, 65535, 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engines
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing UVA:      Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 135 / 0
  Compute Mode:
     < Default multiple host threads can use ::cudaSetDevice with device simultaneously >
> Peer access from Tesla K80 GPU0 -> Tesla K80 GPU1 : Yes
> Peer access from Tesla K80 GPU1 -> Tesla K80 GPU0 : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.1, CUDA Runtime Version = 9.1, NumDevs = 2
Result = PASS
```

Appendix B
Bandwidth Test

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla K80
 Shmoo Mode

.....................................................................
 Host to Device Bandwidth, 1 Devices
 PINNED Memory Transfers
   Transfer Size Bytes        Bandwidth MB/s
   1024                       317.5
   2048                       634.8
   3072                       913.4
```

4096	1171.3
5120	1406.6
6144	1632.3
7168	1842.6
8192	2013.5
9216	2220.9
10240	2383.9
11264	2515.7
12288	2706.0
13312	2848.3
14336	2979.4
15360	3109.0
16384	3220.8
17408	3346.2
18432	3456.4
19456	3565.0
20480	3658.8
22528	3794.4
24576	4011.6
26624	4165.6
28672	4312.9
30720	4430.3
32768	4560.4
34816	4679.6
36864	4782.1
38912	4804.2
40960	4976.3
43008	5027.6
45056	5134.2
47104	5224.6
49152	5298.3
51200	5361.9
61440	5662.8
71680	5893.5
81920	6059.0
92160	6223.6
102400	6365.1
204800	6982.5
307200	7250.2
409600	7379.0
512000	7426.0
614400	7479.7
716800	7519.3
819200	7550.3
921600	7287.0
1024000	7304.0
1126400	7316.8
2174976	7273.8
3223552	7271.5
4272128	7328.6
5320704	7352.7
6369280	7373.6
7417856	7396.0
8466432	7413.0
9515008	7444.9
10563584	7432.2
11612160	8442.1
12660736	9321.4
13709312	9319.0
14757888	10354.1
15806464	10354.7
16855040	11339.8
18952192	11548.8
21049344	11580.4
23146496	11569.9
25243648	11577.5
27340800	11553.3
29437952	11561.2
31535104	11559.3
33632256	11553.5
37826560	11588.3
42020864	11623.2
46215168	11607.5
50409472	11639.9
54603776	11647.8
58798080	11673.8
62992384	11686.4
67186688	11678.7

.....

Device to Host Bandwidth, 1 Devices
PINNED Memory Transfers

Transfer Size Bytes	BandwidthMB/s
1024	438.8
2048	894.5
3072	1328.7
4096	1690.6
5120	2238.5
6144	2572.7
7168	3087.9
8192	3348.1
9216	3738.5
10240	4023.4
11264	4257.5
12288	4549.3
13312	4744.2
14336	4955.8
15360	5153.4
16384	5332.8
17408	5508.0
18432	5678.4
19456	5827.4
20480	5944.5
22528	6256.8
24576	6489.6
26624	6577.9
28672	6941.7
30720	7142.3
32768	7324.9
34816	7505.7
36864	7601.9
38912	7818.2
40960	7960.3
43008	8018.7
45056	8209.4
47104	8336.2
49152	8432.7
51200	8519.2
61440	8986.3
71680	9337.9
81920	9609.8
92160	9834.6
102400	10033.7
204800	10964.5
307200	11352.4
409600	11539.2
512000	11605.1
614400	11689.1
716800	11739.0
819200	11784.6
921600	11364.2
1024000	11440.3
1126400	11416.8
2174976	11274.6
3223552	11427.2
4272128	11415.1
5320704	11430.7
6369280	11557.9
7417856	11485.7
8466432	11584.4
9515008	11555.0
10563584	11542.9
11612160	11580.1
12660736	11545.4
13709312	11559.3
14757888	11564.2
15806464	11548.9
16855040	11568.3
18952192	11631.0
21049344	11640.7
23146496	11677.2
25243648	11726.1
27340800	11697.5
29437952	11722.2
31535104	11736.4
33632256	11706.2
37826560	11709.3
42020864	11706.0
46215168	11708.4
50409472	11705.4
54603776	11708.2
58798080	11708.6
62992384	11707.8
67186688	11709.1

.....

Device to Device Bandwidth, 1 Devices
PINNED Memory Transfers

Transfer Size Bytes	BandwidthMB/s
1024	341.3
2048	704.5
3072	1026.5
4096	1448.5
5120	1713.3
6144	2042.3
7168	2397.0
8192	2662.3
9216	2969.0
10240	3363.5
11264	3939.4
12288	4700.1
13312	5049.7
14336	5394.6
15360	5818.3
16384	6199.2
17408	6572.4
18432	7016.8
19456	7333.0
20480	7708.0
22528	8497.1
24576	9300.6
26624	10547.8
28672	10848.0
30720	11019.0
32768	12210.8
34816	12756.6
36864	13161.0
38912	13789.7
40960	14019.7
43008	13951.6
45056	14555.7
47104	15339.9
49152	16017.1
51200	16748.0
61440	19593.5
71680	22647.3
81920	26395.0
92160	28784.2
102400	32089.1
204800	64102.6
307200	80913.1
409600	105214.4
512000	115458.5
614400	127947.0
716800	140305.3
819200	118985.3
921600	105711.2
1024000	109342.2
1126400	119507.0
2174976	140101.1
3223552	146877.7
4272128	152427.4
5320704	156036.8
6369280	158614.5
7417856	160457.5
8466432	161828.8
9515008	162786.3
10563584	163838.0
11612160	164594.0
12660736	164999.4

```
   13709312                    165879.6
   14757888                    166193.4
   15806464                    166454.7
   16855040                    167214.4
   18952192                    167409.8
   21049344                    168245.3
   23146496                    168721.8
   25243648                    168829.5
   27340800                    169060.6
   29437952                    169422.2
   31535104                    169593.5
   33632256                    169695.8
   37826560                    169958.7
   42020864                    170517.7
   46215168                    170661.3
   50409472                    170761.3
   54603776                    171038.1
   58798080                    170994.5
   62992384                    171063.1
   67186688                    170529.6

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
```