CSE 6230 Final Project

Optimization of a GPGPU Lattice Boltzmann Method Implementation

Arjun Chintapalli[*], Edwin Goh[†]

Georgia Institute of Technology, Atlanta, GA

Contents

[*]School of Computational Science and Engineering
[†]Graduate Research Assistant, School of Aerospace Engineering

1. Initial Performance Assessment - Grid Size Parameter Sweep

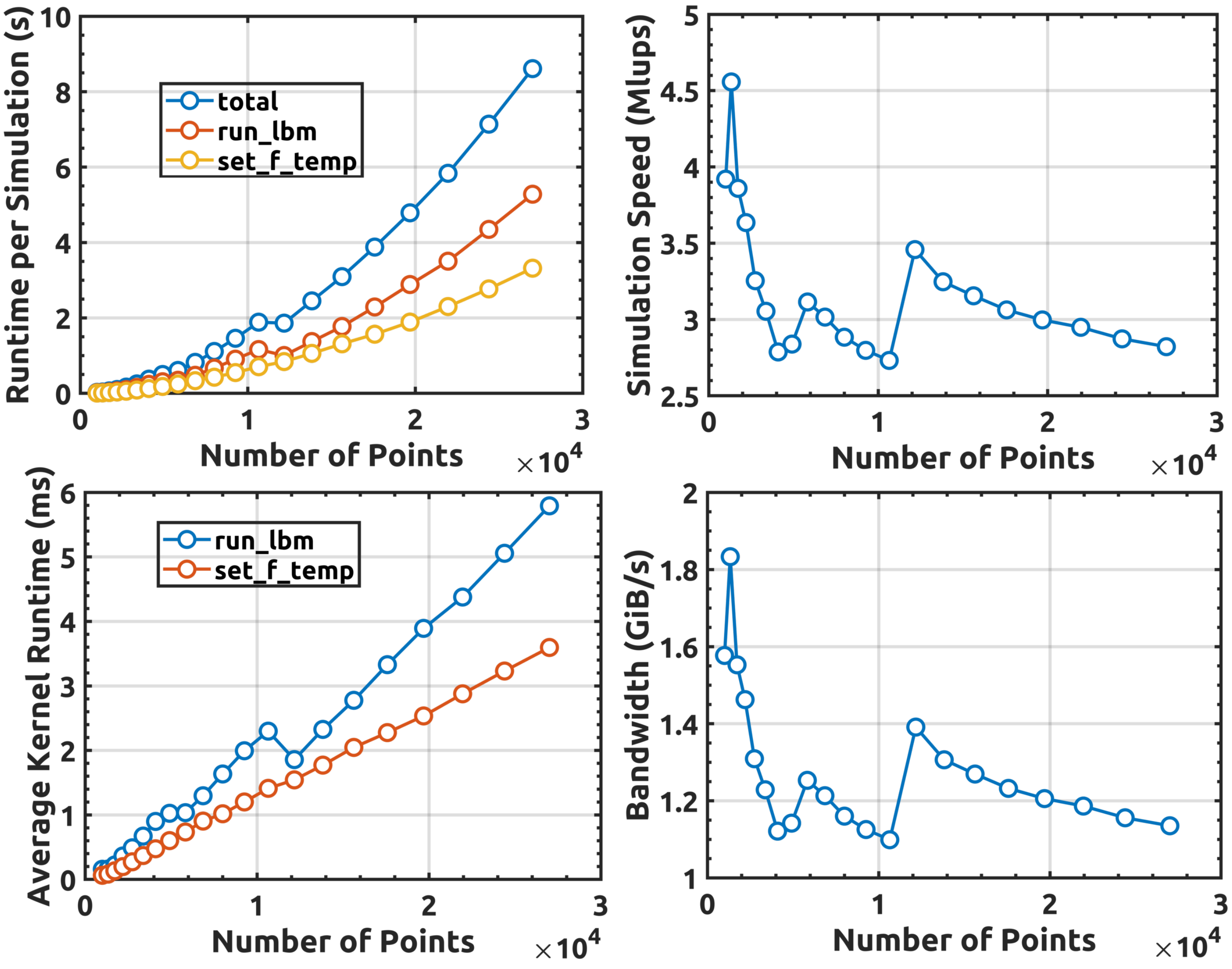


Figure 1: Variation of important performance metrics as a function of number of total points simulated.

Figure 1 shows the results obtained for LBM simulations of different problem sizes. The runtime is obtained by calculating the time taken to complete N^2 iterations of the LBM stream-collide-save algorithm, where N is the number of points in a single spatial dimension. The simulation speed is defined as Mega Lattice-Updates Per Second (Mlups), and is calculated by dividing the total number of points updated (i.e. total number of points $\times$ number of iterations) by the total time taken to run the simulation. Average kernel runtimes are also obtained for the `run_lbm` and `set_f_temp` functions.

The total runtime per simulation increases at a rate that is between quadratic and cubic, i.e. $O(N^{\sim 2.5})$. Under ideal conditions where communication is free and the number of GPU cores is unlimited, the total runtime should only scale as $O(N^2)$ because the number of iterations performed in each iteration is N^2. In reality, problem sizes that are too large to fit in cache or too large to be executed simultaneously will pose constraints that increase the dependence of the runtime on problem size. In addition, the runtime for the `run_lbm` function scales more strongly than that of `set_f_temp`.

From the average kernel runtime curves, we see that the `set_f_temp` kernel scales almost perfectly linearly with number of points. Although the same amount of work is being done by the kernel, the fact that the average runtime to perform the instructions shown in Listing 1 below increases linear with number of points simulated implies that the time taken to read and write data between `f` and `f_temp`.

Listing 1: Listing of the set_f_temp kernel

```

    __global__ void set_f_temp(PetscReal (*f)[27], double (*f_temp)[27], size_t n) {
      int x = threadIdx.x;
      int y = threadIdx.y;
      int z = blockIdx.x;
      int pos_index = x + n * (y + n * z); // current (x, y, z)
      for (int i = 0; i < 27; i++) {
        f_temp[pos_index][i] = f[pos_index][i];
      }
    }
```

2. Initial Performance Assessment - Small Grid Size

2.1 BitBucket Commit

BitBucket Commit: d8a7a7ee3077515e94fae2c6da3b738d48e063b0

2.2 Nvprof Results

Listing 2: Profiling results obtained using nvprof for the small problem size limit.

```
[egoh6@ae-comb-305038 parallel] $ nvprof ./test_lbm -num_tests 12 -scale 11
==23667== NVPROF is profiling process 23667, command: ./test_lbm -num_tests 12 -scale 11
==23667== Profiling application: ./test_lbm -num_tests 12 -scale 11
==23667== Profiling result:
            Type  Time(%)       Time     Calls       Avg       Min       Max  Name
 GPU activities:   64.46%  242.62ms      1452  167.10us  161.06us  181.44us  run_lbm(double[27]*, ...
     double[27]*, unsigned long, double, IndexBlock)
                   35.24%  132.64ms      1452  91.351us  82.625us  113.57us  set_f_temp(double[27]*, ...
                      double[27]*, unsigned long)
                    0.15%  553.93us        13  42.609us  1.2480us  46.625us  [CUDA memcpy HtoD]
                    0.15%  546.21us        12  45.517us  45.377us  45.632us  [CUDA memcpy DtoH]
      API calls:   65.95%  569.48ms      1452  392.21us  244.84us  4.5726ms  cudaDeviceSynchronize
                   30.67%  264.85ms        16  16.553ms  18.931us  264.28ms  cudaFree
                    1.79%  15.455ms      2904  5.3210us  4.1840us  35.078us  cudaLaunch
                    0.47%  4.0607ms        27  150.40us  9.7380us  2.5190ms  cudaMalloc
                    0.44%  3.7962ms        12  316.35us  292.55us  342.35us  cudaGetDeviceProperties
                    0.24%  2.0783ms        25  83.130us  14.244us  143.28us  cudaMemcpy
                    0.18%  1.5575ms     11616     134ns     103ns  10.942us  cudaSetupArgument
                    0.09%  798.39us        12  66.532us  61.944us  82.827us  cudaMemGetInfo
                    0.07%  593.41us      2904     204ns     133ns  9.8740us  cudaConfigureCall
                    0.07%  585.77us       185  3.1660us     145ns  130.00us  cuDeviceGetAttribute
                    0.01%  98.325us         2  49.162us  48.774us  49.551us  cuDeviceTotalMem
                    0.01%  91.173us         2  45.586us  36.263us  54.910us  cuDeviceGetName
                    0.00%  12.373us        16     773ns     622ns  1.7790us  cudaEventCreateWithFlags
                    0.00%  11.752us        16     734ns     531ns  2.0820us  cudaEventDestroy
                    0.00%  9.5630us         1  9.5630us  9.5630us  9.5630us  cudaSetDeviceFlags
                    0.00%  5.8580us         2  2.9290us  2.5040us  3.3540us  cudaThreadSynchronize
                    0.00%  4.7290us        11     429ns     285ns  1.1730us  cudaDeviceGetAttribute
                    0.00%  2.0430us         4     510ns     133ns     991ns  cuDeviceGetCount
                    0.00%  1.6080us         1  1.6080us  1.6080us  1.6080us  cudaGetDevice
                    0.00%  1.1700us         3     390ns     234ns     657ns  cuDeviceGet
                    0.00%     662ns         1     662ns     662ns     662ns  cuInit
                    0.00%     508ns         1     508ns     508ns     508ns  cuDriverGetVersion
                    0.00%     459ns         1     459ns     459ns     459ns  cudaGetDeviceCount
```

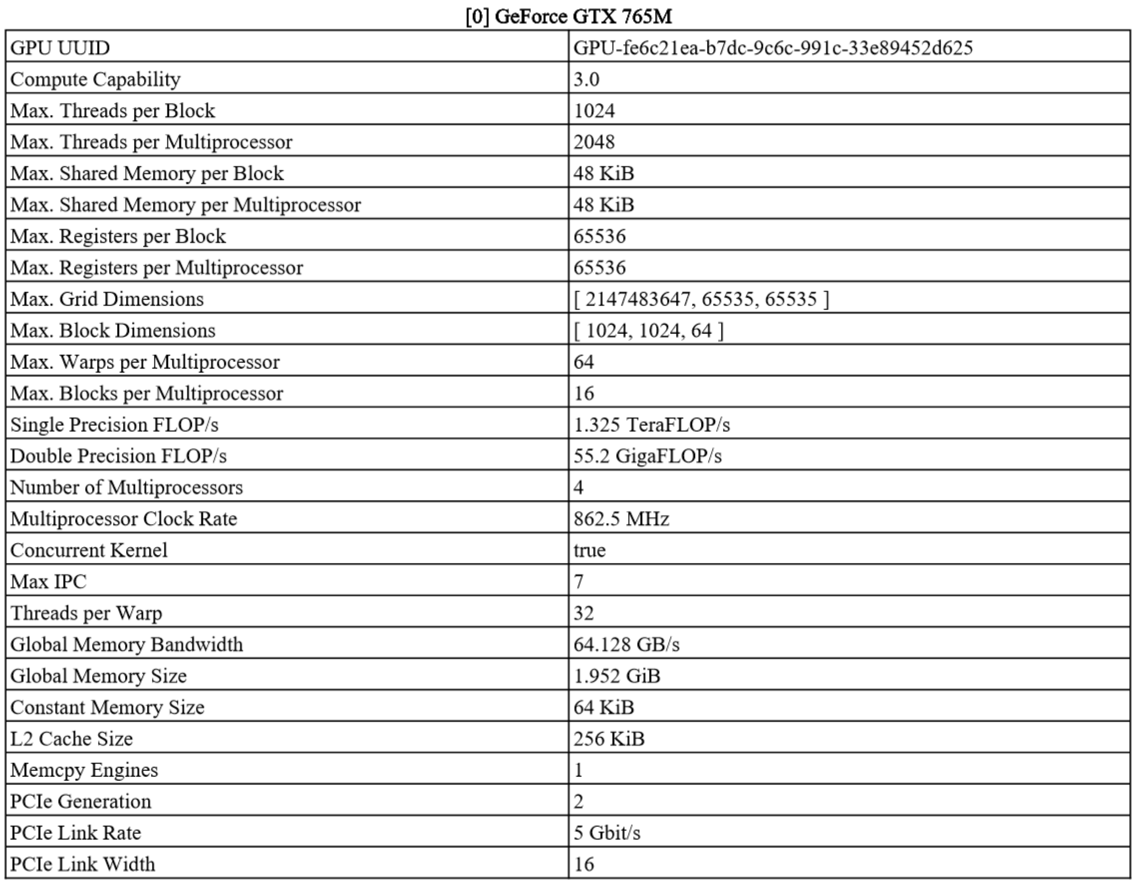
2.3 Profiling with the Nvidia Visual Profiler

The Nvidia Visual Profiler was used to interactively profile the initial code listed in the above commit.

For preliminary profiling, a local machine (a notebook with a GTX 765M GPU) was used instead of the Bridges cluster in order to decrease turnaround time. Moreover, the Bridges cluster has been found to be inaccessible at times, as shown in the output from `squeue` below:

```
JOBID   PARTITION     NAME     USER    ST TIME  NODES NODELIST(REASON)
2057068 GPU-share   lbm-k80    arjunch PD 0:00    1   (PartitionDown)
2057341 GPU-share   lbm-p100   egoh6   PD 0:00    1   (PartitionDown)
```

The specifications of this GPU as given by the Nvidia Visual Profiler are shown in Figure 2 below:

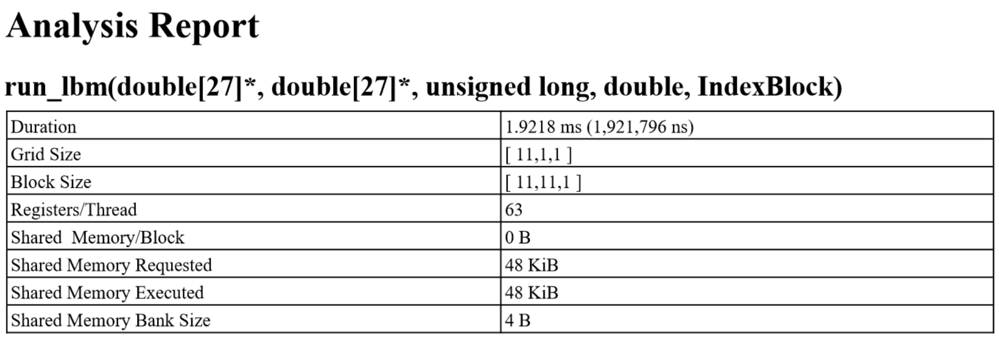


[0] GeForce GTX 765M

GPU UUID	GPU-fe6c21ea-b7dc-9c6c-991c-33e89452d625
Compute Capability	3.0
Max. Threads per Block	1024
Max. Threads per Multiprocessor	2048
Max. Shared Memory per Block	48 KiB
Max. Shared Memory per Multiprocessor	48 KiB
Max. Registers per Block	65536
Max. Registers per Multiprocessor	65536
Max. Grid Dimensions	[2147483647, 65535, 65535]
Max. Block Dimensions	[1024, 1024, 64]
Max. Warps per Multiprocessor	64
Max. Blocks per Multiprocessor	16
Single Precision FLOP/s	1.325 TeraFLOP/s
Double Precision FLOP/s	55.2 GigaFLOP/s
Number of Multiprocessors	4
Multiprocessor Clock Rate	862.5 MHz
Concurrent Kernel	true
Max IPC	7
Threads per Warp	32
Global Memory Bandwidth	64.128 GB/s
Global Memory Size	1.952 GiB
Constant Memory Size	64 KiB
L2 Cache Size	256 KiB
Memcpy Engines	1
PCIe Generation	2
PCIe Link Rate	5 Gbit/s
PCIe Link Width	16

Figure 2: Nvidia Visual Profiler output showing specifications and compute capabilities of the Nvidia GTX 765M

Because the `test_lbm` harness calculates multiple test cases of various grid sizes, only the `run_lbm` kernels which ran smallest grid size (11 x 11 x 11) was analyzed in order to determine the bottlenecks in the limit of small problem sizes. This is shown in Figure 3 below



Analysis Report

run_lbm(double[27]*, double[27]*, unsigned long, double, IndexBlock)

Duration	1.9218 ms (1,921,796 ns)
Grid Size	[11,1,1]
Block Size	[11,11,1]
Registers/Thread	63
Shared Memory/Block	0 B
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Figure 3: Nvidia Visual Profiler output showing CUDA kernel being analyzed. The small case (11 x 11 x 11) is being analyzed in this figure.

In this limit of small problem sizes, the initial code uses a block of 121 threads per block, and uses a total of 11 blocks to update the discretized velocity PDF at each of the $11 \times 11 \times 11$ grid points.

2.4 Identified Bottlenecks

2.4.1 Issue # 1: Kernel is Compute/Memory Latency Bound

Figure 4 shows the utilization of the GPU's compute capabilities and L1 memory during the execution of the `run_lbm` kernel for small problem sizes. The severe underutilization of even a mobile-class laptop GPU indicates that, for small problem sizes, most of the GPU's compute cores L1 memory are idly standing by without receiving any tasks to perform. This may that there is a latency issue where instructions are either:

1. Taking too long to finish, signifying arithmetic operation latency; or
2. Waiting a long time for the required data to be fetched into or from the GPU, i.e. high memory operation latency.

In other words, the performance of the `run_lbm` kernel when problem sizes are small is limited by the dearth of work to keep the GPU busy.
This further supports our conclusions from Checkpoint 2, where we said our LBM algorithm is bound by memory bandwidth limitations. This issue can be overcome by reducing the need for the kernel to repeatedly access global memory stores, by either prefetching the necessary information or architecting a shared memory system. .

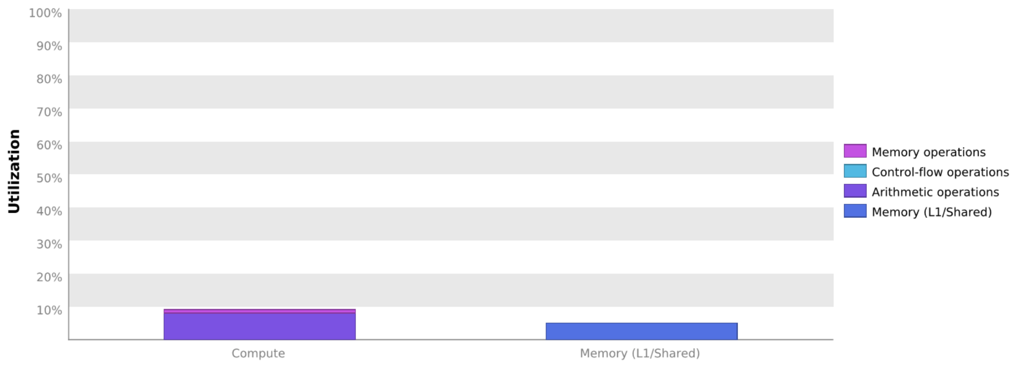
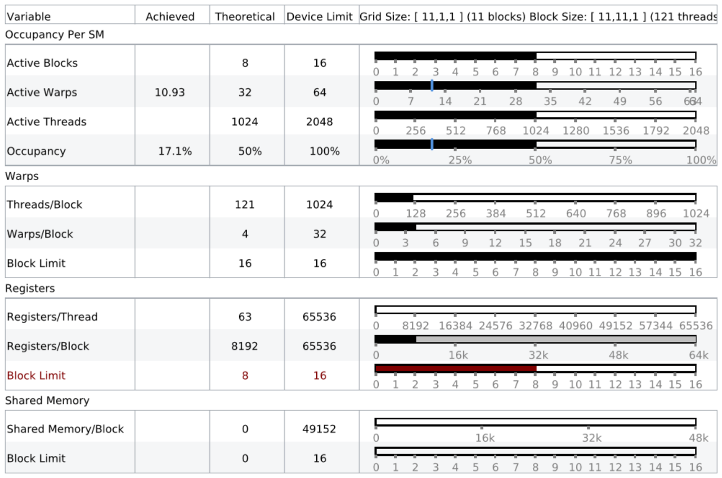


Figure 4: Nvidia Visual Profiler output showing the levels of Compute and L1 Memory utilization for the current limit of small problem sizes.

2.4.2 Issue # 2:

The Nvidia Visual profiler tracks the number of threads, blocks, and warps that are concurrently running during the execution of a kernel, the results of which are shown in Figure 5.

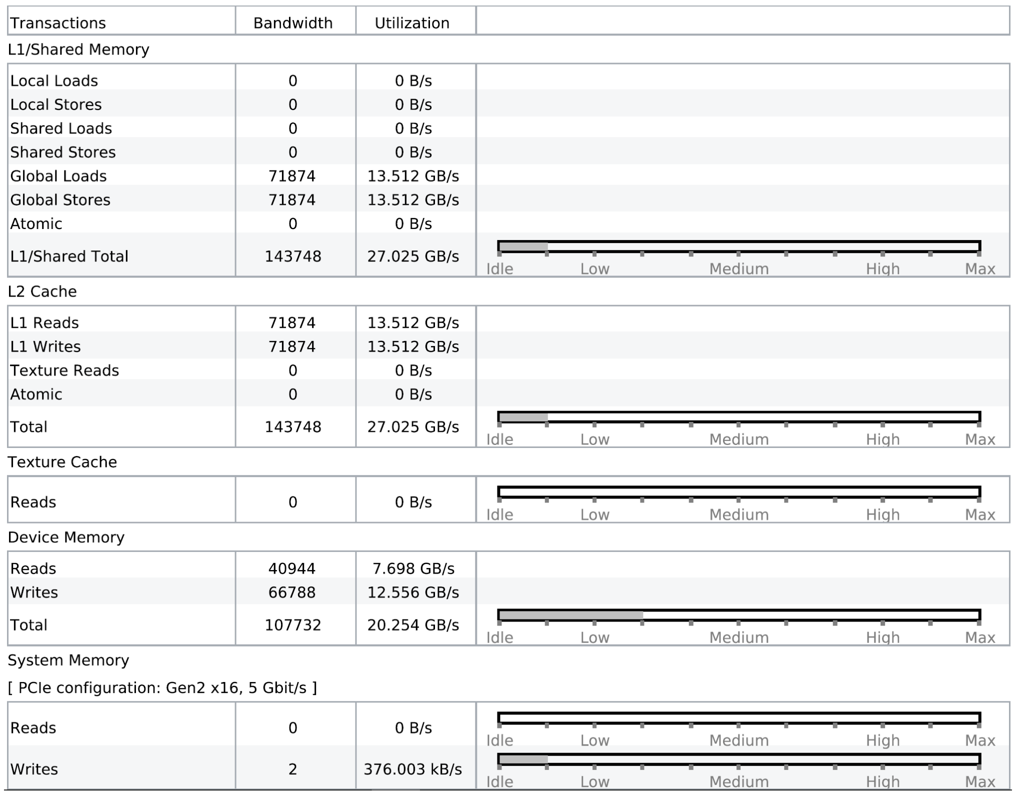


Variable	Achieved	Theoretical	Device Limit	Grid Size: [11,1,1] (11 blocks) Block Size: [11,11,1] (121 threads
Occupancy Per SM				
Active Blocks		8	16	
Active Warps	10.93	32	64	
Active Threads		1024	2048	
Occupancy	17.1%	50%	100%	
Warps				
Threads/Block		121	1024	
Warps/Block		4	32	
Block Limit		16	16	
Registers				
Registers/Thread		63	65536	
Registers/Block		8192	65536	
Block Limit		8	16	
Shared Memory				
Shared Memory/Block		0	49152	
Block Limit		0	16	

Figure 5: Nvidia Visual Profiler output showing the utilization of blocks, threads, and warps during the execution of the `run_lbm` kernel for small problem sizes.

Based on Figure 5, only 50% of the maximum number of active warps are capable of running concurrently. Even among this theoretical 50%, the current version of the LBM code uses only 10 out of the 32 warps, i.e. 320 threads. For a problem size with $11 \times 11 \times 11 = 1331$ points, the GPU should ideally be able to use 1,331 out of its 2,048 threads to execute the `run_lbm` kernel in one fell swoop. In such a situation the number of active threads would be 1,331, the corresponding number of warps of which is 42. The kernel does not execute enough blocks to hide memory and operation latency. Typically the kernel grid size must be large enough to fill the GPU with multiple "waves" of blocks. Based on theoretical occupancy, device "GeForce GTX 765M" can simultaneously execute 8 blocks on each of the 4 SMs, so the kernel may need to execute a multiple of 32 blocks to hide the compute and memory latency. If the kernel is executing concurrently with other kernels then fewer blocks will be required because the kernel is sharing the SMs with those kernels.

2.4.3 Issue # 3: Memory Bandwidth Usage is Limited



Transactions	Bandwidth	Utilization	
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	71874	13.512 GB/s	
Global Stores	71874	13.512 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	143748	27.025 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	71874	13.512 GB/s	
L1 Writes	71874	13.512 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	143748	27.025 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	40944	7.698 GB/s	
Writes	66788	12.556 GB/s	
Total	107732	20.254 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	2	376.003 kB/s	Idle Low Medium High Max

Figure 6: Nvidia Visual Profiler output displaying memory bandwidth utilization for different types of memory available to the GTX 765M device.

Figure 6 obtained from the Nvidia Visual Profiler shows that the `run_lbm` kernel uses only a small amount of the memory bandwidth available on the GTX 765M. Furthermore, there are essentially no operations performed using shared memory. This exclusive reliance on device memory is a limiting factor in terms of performance, and optimization ought to be performed to enable operations using memory shared between threads in a block.

3. Initial Performance Assessment - Large Grid Size

3.1 BitBucket Commit

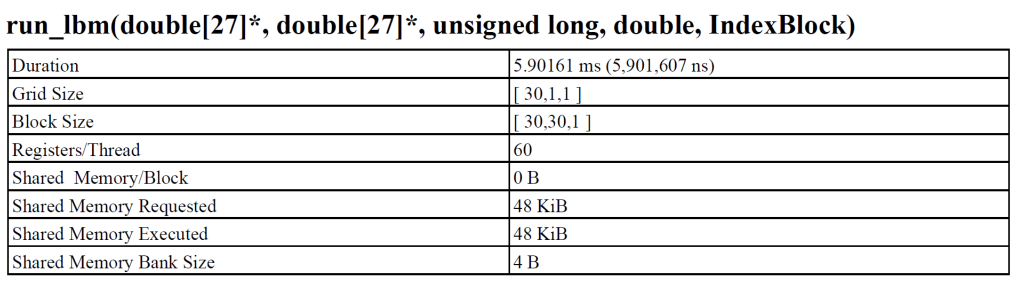
BitBucket Commit: d8a7a7ee3077515e94fae2c6da3b738d48e063b0

3.2 Nvprof Results

Listing 3: Profiling results obtained using nvprof for the large problem size limit.

```
[egoh6@ae-comb-305038 parallel] $ nvprof ./test_lbm -num_tests 12 -scale 30
==24214== NVPROF is profiling process 24214, command: ./test_lbm -num_tests 12 -scale 30
==24214== Profiling application: ./test_lbm -num_tests 12 -scale 30
==24214== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   61.39%  63.3723s     10800  5.8678ms  5.7919ms  6.0652ms  run_lbm(double[27]*, ...
     double[27]*, unsigned long, double, IndexBlock)
                   38.58%  39.8285s     10800  3.6878ms  3.5961ms  3.8662ms  set_f_temp(double[27]*, ...
                       double[27]*, unsigned long)
                    0.01%  13.326ms        12  1.1105ms  1.0235ms  1.1812ms  [CUDA memcpy DtoH]
                    0.01%  11.035ms        13  848.88us  1.3120us  974.21us  [CUDA memcpy HtoD]
      API calls:   99.58%  104.126s     10800  9.6413ms  9.2174ms  14.411ms  cudaDeviceSynchronize
                    0.24%  254.99ms        16  15.937ms  21.038us  252.38ms  cudaFree
                    0.12%  127.30ms     21600  5.8930us  4.1040us  341.67us  cudaLaunch
                    0.03%  26.473ms        25  1.0589ms  14.855us  1.3385ms  cudaMemcpy
                    0.01%  12.040ms     86400     139ns     103ns  304.76us  cudaSetupArgument
                    0.01%  7.8426ms        27  290.47us  9.1790us  2.3155ms  cudaMalloc
                    0.00%  4.5785ms     21600     211ns     153ns  10.775us  cudaConfigureCall
                    0.00%  3.9233ms        12  326.94us  308.64us  366.61us  cudaGetDeviceProperties
                    0.00%  748.70us        12  62.391us  59.361us  75.068us  cudaMemGetInfo
                    0.00%  599.90us       185  3.2420us     132ns  133.07us  cuDeviceGetAttribute
                    0.00%  98.106us         2  49.053us  48.863us  49.243us  cuDeviceTotalMem
                    0.00%  76.516us         2  38.258us  37.724us  38.792us  cuDeviceGetName
                    0.00%  11.156us        16     697ns     518ns  1.8250us  cudaEventCreateWithFlags
                    0.00%  11.074us        16     692ns     457ns  2.1540us  cudaEventDestroy
                    0.00%  9.2610us         1  9.2610us  9.2610us  9.2610us  cudaSetDeviceFlags
                    0.00%  7.4040us         2  3.7020us  3.3660us  4.0380us  cudaThreadSynchronize
                    0.00%  4.8480us        11     440ns     296ns  1.1880us  cudaDeviceGetAttribute
                    0.00%  1.8340us         4     458ns     174ns  1.1890us  cuDeviceGetCount
                    0.00%  1.6130us         1  1.6130us  1.6130us  1.6130us  cudaGetDevice
                    0.00%  1.0260us         3     342ns     166ns     606ns  cuDeviceGet
                    0.00%     556ns         1     556ns     556ns     556ns  cuInit
                    0.00%     475ns         1     475ns     475ns     475ns  cudaGetDeviceCount
                    0.00%     437ns         1     437ns     437ns     437ns  cuDriverGetVersion
```

3.3 Profiling with the Nvidia Visual Profiler



run_lbm(double[27]*, double[27]*, unsigned long, double, IndexBlock)

Duration	5.90161 ms (5,901,607 ns)
Grid Size	[30,1,1]
Block Size	[30,30,1]
Registers/Thread	60
Shared Memory/Block	0 B
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB
Shared Memory Bank Size	4 B

Figure 7: Nvidia Visual Profiler output showing the kernel being profiled to be of the problem size 30 x 30 x 30

3.4 Identified Bottlenecks

3.4.1 Issue # 1: Kernel Performance Is Bound By Instruction And Memory Latency

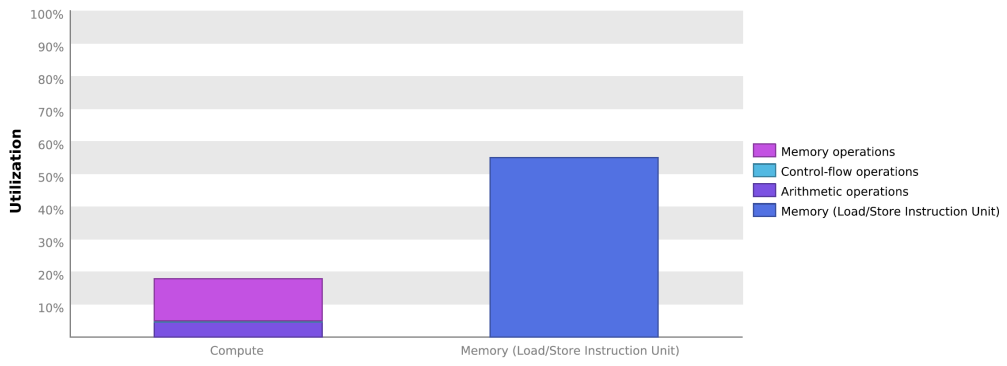
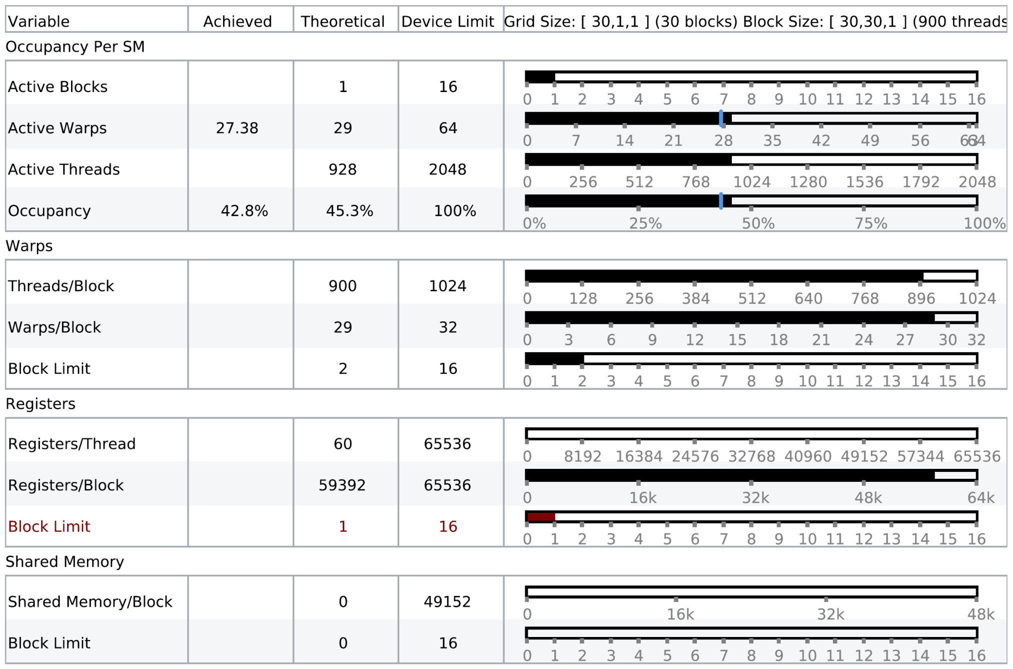


Figure 8: Nvidia Visual Profiler Output showing the levels of Compute and L1 memory utilization for the limit of large problem sizes

3.4.2 Issue # 2: Low Theoretical Occupancy

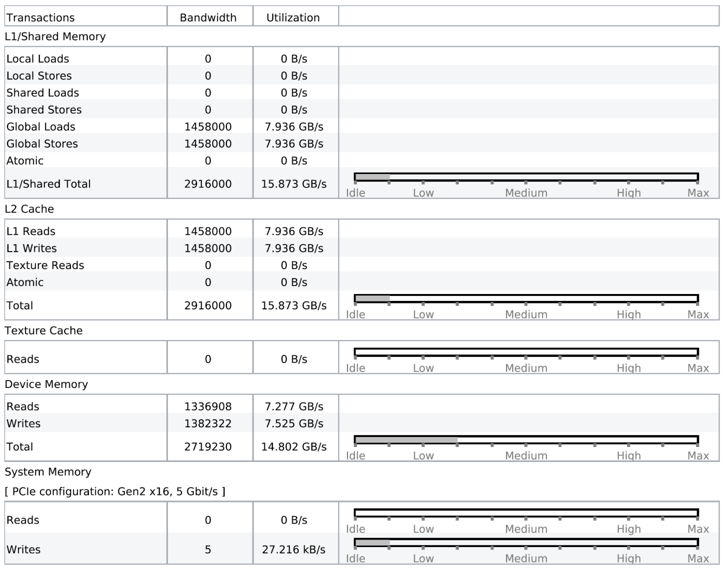


Variable	Achieved	Theoretical	Device Limit	Grid Size: [30,1,1] (30 blocks) Block Size: [30,30,1] (900 threads
Occupancy Per SM				
Active Blocks		1	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Active Warps	27.38	29	64	0 7 14 21 28 35 42 49 56 6364
Active Threads		928	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	42.8%	45.3%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		900	1024	0 128 256 384 512 640 768 896 1024
Warps/Block		29	32	0 3 6 9 12 15 18 21 24 27 30 32
Block Limit		2	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Registers				
Registers/Thread		60	65536	0 8192 16384 24576 32768 40960 49152 57344 65536
Registers/Block		59392	65536	0 16k 32k 48k 64k
Block Limit		1	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Shared Memory				
Shared Memory/Block		0	49152	0 16k 32k 48k
Block Limit		0	16	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figure 9: Nvidia Visual Profiler output showing the utilization of blocks, threads, and warps during the execution of the `run_lbm` *kernel for small problem sizes.*

The large grid size dramatically more so than the smaller grid size demonstrates the memory latency issues inherent with the code. Optimizations with memory latency would speed up the code more so than compute optimizations.

3.4.3 Issue # 3: Low Memory Bandwidth Utilization



Transactions	Bandwidth	Utilization	
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	1458000	7.936 GB/s	
Global Stores	1458000	7.936 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	2916000	15.873 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	1458000	7.936 GB/s	
L1 Writes	1458000	7.936 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	2916000	15.873 GB/s	Idle Low Medium High Max
Texture Cache			
Reads	0	0 B/s	Idle Low Medium High Max
Device Memory			
Reads	1336908	7.277 GB/s	
Writes	1382322	7.525 GB/s	
Total	2719230	14.802 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	27.216 kB/s	Idle Low Medium High Max

Figure 10: Nvidia Visual Profiler output displaying memory bandwidth utilization for different types of memory available to the GTX 765M device

4. Notebook Entry # 1: Reducing Memory Accesses with Temporary Local Variables

4.1 BitBucket Commit and Modified Code

BitBucket Commit: 1890a95

```
__global__ void run_lbm(PetscReal (*f)[27], double (*f_temp)[27], size_t n, double tau, struct IndexBlock ...
    iBlock)
{
  const double tau_reciprocal = 1/tau;
  const double one_minus_tau_reciprocal = 1 - tau_reciprocal;
  int x = threadIdx.x;
  int y = threadIdx.y;
  int z = blockIdx.x;

  int pos_index = x + n * (y + n * z); // current (x, y, z) corresponds to a pos_index in the global f array

  double rho = 0, u = 0, v = 0, w = 0;
  PetscReal f_loc[27];

  for (int dir = 0; dir < 27; dir++) {
    size_t x_periodic = (n + x - gpu_lbm_velocities[dir][0]) % n;
    size_t y_periodic = (n + y - gpu_lbm_velocities[dir][1]) % n;
    size_t z_periodic = (n + z - gpu_lbm_velocities[dir][2]) % n;
    // Convert Cartesian coordinates to n*n*n row index:
    size_t old_pos = x_periodic + y_periodic * n + z_periodic * n * n;

 //   f[pos_index][dir] = f_temp[old_pos][dir];
    f_loc[dir]=f_temp[old_pos][dir];
    rho += f_loc[dir];
    u += f_loc[dir] * gpu_lbm_velocities[dir][0];
    v += f_loc[dir] * gpu_lbm_velocities[dir][1];
    w += f_loc[dir] * gpu_lbm_velocities[dir][2];
  }

  double rhoinv = 1.0/rho;

  u *= rhoinv; v *= rhoinv; w *= rhoinv;
  for (int dir = 0; dir < 27; dir++) {
    // calculate u_alpha (the dot product)
    double u_alpha = gpu_lbm_velocities[dir][0] * u +
      gpu_lbm_velocities[dir][1] * v + gpu_lbm_velocities[dir][2] * w;
    // calculate equilibrium distribution
    double f_equil_i = gpu_lbm_weights[dir] * rho *
      (1. + 3. * u_alpha + 4.5 * u_alpha * u_alpha - 1.5*(u*u + v*v + w*w));
    f[pos_index][dir] = tau_reciprocal * f_equil_i + one_minus_tau_reciprocal * f_loc[dir];
  }
}
```

4.2 Proposed Change and Expected Improvements

Since memory transfers and latencies were determined to be the the primary constraint on our code, optimizations to limit memory read/writes from global device memory were found. The very first initial optimization in this line of thought was to preload each thread's own f distribution values instead of repeatedly accessing them because they don't change throughout the execution of a thread. Priorly, with each iteration a thread would load from global device memory its corresponding lattice nodes f_i and $ftemp_i$ distribution values multiple times in the execution, and thus add to the memory dependency latency. An optimization was thus implemented to store these distribution values in a local array with the expected outcome of reducing the time for the run_lbm kernel specifically as well as the overall time.

4.3 Profiling Results

Table 1: Raw Performance Statistics

	Optimazation 1 (Average across 10 runs)			**Base Case (Average across 10 runs)**		
N	Total Time (ms)	Avg run_lbm (μs)	Avg set_ftemp (μs)	Total Time (ms)	Avg run_lbm (μs)	Avg set_ftemp (μs)
10	300.12	194.58	105.54	368.23	266.62	101.61
15	2208.4	508.53	471.92	2436	617.21	465.28
20	8409.731	1074	1027	9865.6	1394.5	1070.1
30	63171.2	3227	3789	84640	5747.7	3924.1

Table 2: Performance Boost Statistics

	Performance Boost		
N	Total Time %	run_lbm %	set_ftemp %
10	0.18	0.27	-0.03
15	0.09	0.17	-0.01
20	0.14	0.23	0.04
30	0.25	0.44	0.03

4.4 Analysis

As expected the profiling results and tables demonstrate that prefetching the f_i distributions dramatically increased performance in the run_lbm kernel specifically as well as the Total Time. Since nothing was modified in the set_ftemp kernel as expected there is no statistically significant increase in performance. In conclusion, as expected reducing the number of device global memory reads lead to performance boosts over 15% at all grid sizes, which further confirms that memory latency is the primary bottleneck.

The logical next step of implementing a shared memory system was also considered but due to the memory limitations of the streaming processors this was deemed infeasible. Consider the base case implementation with the maximum tested grid size of $N = 15$, then each thread block would have 225 threads corresponding to 225 lattice nodes and in total 6075 distributions for each f_i and f_temp_i and given the data size of 4 bytes per distribution this would have necessitated a a total shared memory of 48,600 bytes but the Nvidia K20 multiproccessors only have 49,152 bytes of shared memory. Additionally with the prefetching implementation, there is only one read and one write per direction of distribution per call to lbm_run, and each thread would read/write information pertaining to its own lattice node, thus shared memory would be of limited utility in this situation.

5. Notebook Entry # 2: Changing Grid/Block Size

5.1 BitBucket Commit

BitBucket Commit: b91639b

5.2 Proposed Change and Expected Improvements

In the initial code, the grid-block structure adopted was one where the threads in the block were allocated such that each thread in the x and y dimension represented points in the x and y directions in the simulated grid. Blocks in the CUDA grid's x direction were used to represent points in the z dimension in space. This implementation, however, led to constraints on the number of spatial points in a given dimension, because the maximum number of threads per block is limited to 1,024. Consequently, results such as those shown in Figure 1 only include problem sizes up to 30 x 30 x 30, because larger problem sizes would break the limit on threads per block.

The new grid-block structure can be seen in the following code:

```
threads_per_block = PetscMin(prop.maxThreadsPerBlock, threads_per_block);
int numBlocks = ceil(N*N*N / threads_per_block + 1);
dim3 grid_of_blocks(numBlocks, 1, 1);
dim3 block_of_threads(threads_per_block, 1, 1);
...
...
int pos_index = threadIdx.x + blockIdx.x * blockDim.x;
if (pos_index ≥ n * n * n) {
  return;
}
int x = (int) pos_index % (int) n;
int y = ((int)pos_index / (int) n) % (int) n;
int z = (int) pos_index / ((int) n * (int)n);
```

In the updated version of the LBM code, the `threads_per_block` is a user-defined input to set the number of threads per block. Using this, the user can determine the optimized block size through the `-block_size` input in order to attain maximum performance. The current project found a block size of around 720 to be optimum. This proposed change would increase the occupancy of the code, an issue which was identified in the initial profiling with the Nvidia Visual Profiler. An increased occupancy would result in more threads working simultaneously at any given time, thereby hiding latency and increasing performance.

5.3 Profiling Results

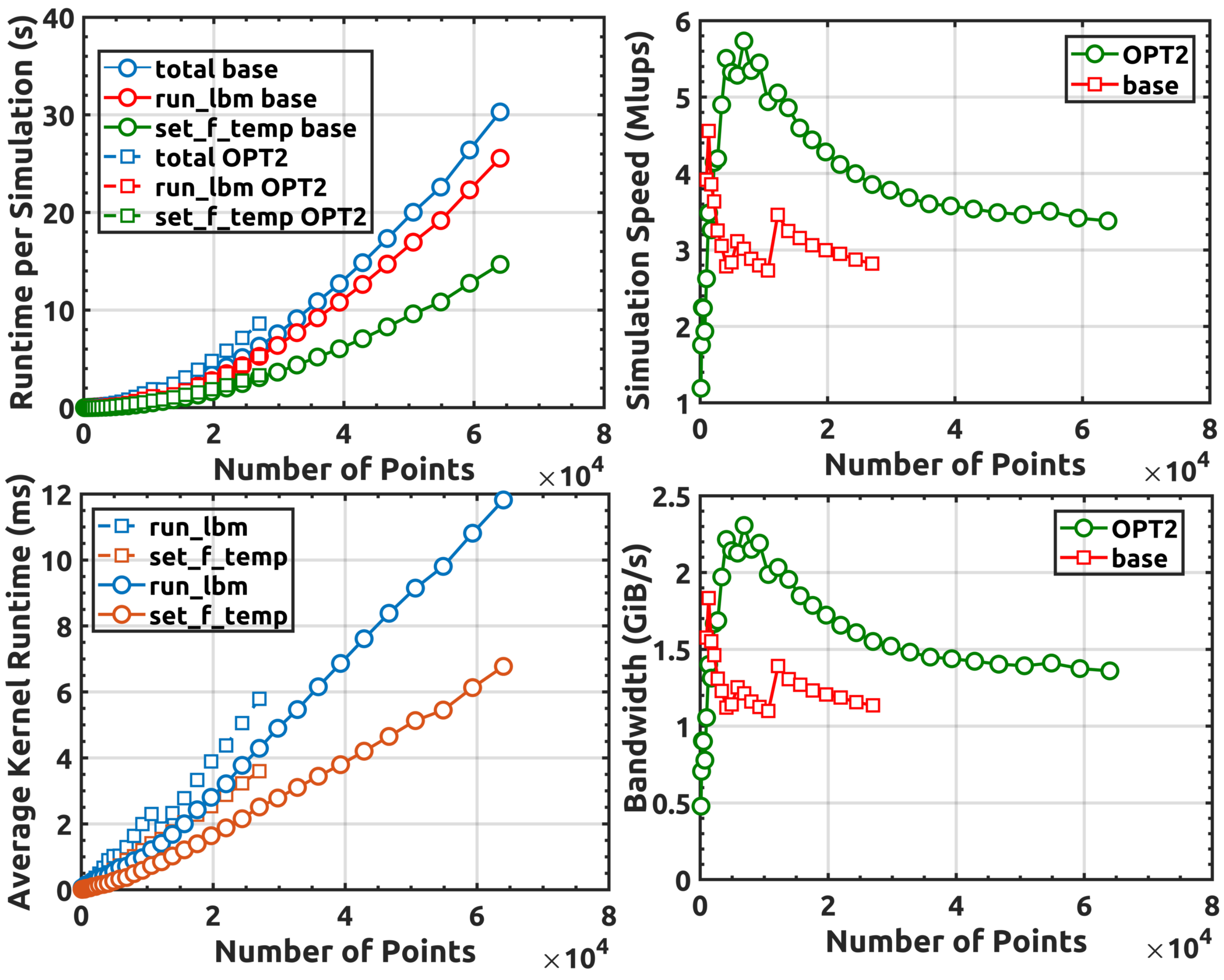


Figure 11: Comparison between the performance of the intial LBM code and the version of the code using the optimized grid-block structure.

5.4 Analysis

Figure 11 above shows the performance from the LBM code using the optimized grid-block structure. It can be seen that not only does the new code, labeled "OPT2" provide decreased runtime, it also increases the range of problem sizes that can be simulated. The increased concurrency results in increased bandwidth, as expected. Furthermore, this change also results in a decreased dependence of average kernel runtimes with the number of points. This might be an artifact resulting from the increased bandwidth due to higher concurrency, which hides the latency behind memory operations in the two kernels.

6. Notebook Entry # 3: Removing cudaDeviceSynchronize

6.1 BitBucket Commit

BitBucket Commit: 3d0c302

```
  // LBM_ITERATE_DEVICE MAIN SIMULATION LOOP
  double begin = seconds();
  for (int i = 0; i < num_iter; i++) {
//    set_f_temp <<< grid_of_blocks, block_of_threads >>> (fLocal[0], f_temp, N); // set f_temp = f //TODO: ...
    move this into run_lbm
    run_lbm <<<grid_of_blocks, block_of_threads>>> (fLocal[0], f_temp, N, tau, iBlock[0]); // *fLocal[0] is ...
        an array of length 27, with n^3 rows
//    cudaDeviceSynchronize();
  }
```

6.2 Proposed Change and Expected Improvements

With the original code it was noticed that the cudaDeviceSynchronize took up most of the time for the code. Thus it was experimented to see if the function was necessary between iterations.
In a similar train of thought it was also noticed if the set_ftemp function was really necessary and if a work around could be achieved.

6.3 Profiling Results

Listing 4: Profiling results obtained using nvprof for the LBM code without set_f_temp.

```
nvprof --log-file "nvprof_no_ftemp_Working.txt" ./test_lbm_noftemp >> "test_lbm_noftemp_Working.txt"
  Test 7:
    Test dimensions: [10 x 10 x 10], 100 iterations, tolerance 2.
    Initializing LBM calculation...
    Total points = 10 * 10 * 10 = 1000
    Allocated grid of blocks with dimensions: (10, 1, 1); 10 blocks total
    Allocated 10 thread blocks with dimensions: (10, 10, 1); 100 threads per block
    10 blocks * 100 threads per block = 1000 threads total
Remote CUDA information:
Device name: GeForce GTX 765M
# of multiprocessors: 4
compute capability: 3.0
GPU global memory available: 1999.2 MiB

Performance Statistics:
GPU Memory Allocated: 0.4 (MiB)
Runtime: 0.001 (s)
Speed: 139.74 (Mlups)
Bandwidth: 56.2 (GiB/s)
      Error 0.128923
      Passed.
  Test 8:
    Test dimensions: [15 x 15 x 15], 225 iterations, tolerance 0.888889
    Initializing LBM calculation...
    Total points = 15 * 15 * 15 = 3375
    Allocated grid of blocks with dimensions: (15, 1, 1); 15 blocks total
    Allocated 15 thread blocks with dimensions: (15, 15, 1); 225 threads per block
    15 blocks * 225 threads per block = 3375 threads total
Remote CUDA information:
Device name: GeForce GTX 765M
# of multiprocessors: 4
compute capability: 3.0
GPU global memory available: 1999.2 MiB

Performance Statistics:
```

```
GPU Memory Allocated: 1.4 (MiB)
Runtime: 0.002 (s)
Speed: 477.92 (Mlups)
Bandwidth: 192.3 (GiB/s)
      Error 0.0589498
      Passed.
  Test 9:
    Test dimensions: [13 x 13 x 13], 169 iterations, tolerance 1.18343
    Initializing LBM calculation...
    Total points = 13 * 13 * 13 = 2197
    Allocated grid of blocks with dimensions: (13, 1, 1); 13 blocks total
    Allocated 13 thread blocks with dimensions: (13, 13, 1); 169 threads per block
    13 blocks * 169 threads per block = 2197 threads total
Remote CUDA information:
Device name: GeForce GTX 765M
# of multiprocessors: 4
compute capability: 3.0
GPU global memory available: 1999.2 MiB

Performance Statistics:
GPU Memory Allocated: 0.9 (MiB)
Runtime: 0.001 (s)
Speed: 313.93 (Mlups)
Bandwidth: 126.3 (GiB/s)
      Error 0.904949
      Passed.
==28035== NVPROF is profiling process 28035, command: ./test_lbm_noftemp
==28035== Profiling application: ./test_lbm_noftemp
==28035== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.68%  285.54ms      1121  254.72us  125.73us  615.08us  run_lbm(double[27]*, ...
     double[27]*, unsigned long, double, IndexBlock)
                    0.16%  459.97us        11  41.815us  1.3440us  112.42us  [CUDA memcpy HtoD]
                    0.16%  445.00us        10  44.499us  18.048us  118.66us  [CUDA memcpy DtoH]
      API calls:   48.53%  320.13ms        14  22.866ms  11.785us  319.63ms  cudaFree
                   48.50%  319.92ms        21  15.234ms  15.086us  138.41ms  cudaMemcpy
                    1.15%  7.5785ms        23  329.50us  8.5700us  4.9184ms  cudaMalloc
                    0.92%  6.0355ms      1121  5.3840us  4.5520us  29.509us  cudaLaunch
                    0.50%  3.3096ms        10  330.96us  302.72us  417.78us  cudaGetDeviceProperties
                    0.12%  768.36us      5605     137ns     104ns  11.006us  cudaSetupArgument
                    0.11%  736.31us       185  3.9800us     143ns  225.30us  cuDeviceGetAttribute
                    0.10%  635.25us        10  63.524us  60.417us  71.256us  cudaMemGetInfo
                    0.04%  252.68us      1121     225ns     183ns  10.906us  cudaConfigureCall
                    0.02%  100.80us         2  50.400us  50.019us  50.782us  cuDeviceTotalMem
                    0.01%  74.045us         2  37.022us  33.875us  40.170us  cuDeviceGetName
                    0.00%  11.417us        16     713ns     488ns  1.9620us  cudaEventDestroy
                    0.00%  11.289us        16     705ns     507ns  2.0260us  cudaEventCreateWithFlags
                    0.00%  8.8070us         2  4.4030us  4.3800us  4.4270us  cudaThreadSynchronize
                    0.00%  6.7100us         1  6.7100us  6.7100us  6.7100us  cudaSetDeviceFlags
                    0.00%  4.5880us        11     417ns     274ns  1.2610us  cudaDeviceGetAttribute
                    0.00%  1.7550us         4     438ns     148ns  1.0840us  cuDeviceGetCount
                    0.00%  1.7160us         1  1.7160us  1.7160us  1.7160us  cudaGetDevice
                    0.00%  1.0450us         3     348ns     180ns     622ns  cuDeviceGet
                    0.00%     603ns         1     603ns     603ns     603ns  cuDriverGetVersion
                    0.00%     550ns         1     550ns     550ns     550ns  cuInit
                    0.00%     448ns         1     448ns     448ns     448ns  cudaGetDeviceCount
```

6.4 Analysis

Although removing the cudaDeviceSynchronize was found to dramatically improve the time without affecting the accuracy of the simulations. This is expected because removing the scynchronization reduces the latent time waiting for all blocks to finish. The base code naturally ensures accuracy by keeping the original distributions separate from the distributions post collision.
Removing the set_ftemp function was found to work only under certain circumstances. Only if a prior make and run of the code with the set_ftemp function included was made, then only would a make and run without the set_ftemp function would work. Even with freeing up the prior allocated arrays, an explanation for why the code runs successfully in only some circumstances couldn't be found. Still though the code was found to be faster than the base case, but due to the

unreliability in performance it was not included in the final submission. As expected, the code without set_ftemp works better because of the reduction in memory read/writes needed to overwrite the original distributions with each iteration. This reasoning is explained in more detail in the following section.

7. Notebook Entry # 4: Push-Pull LBM Implementation

7.1 BitBucket Commit

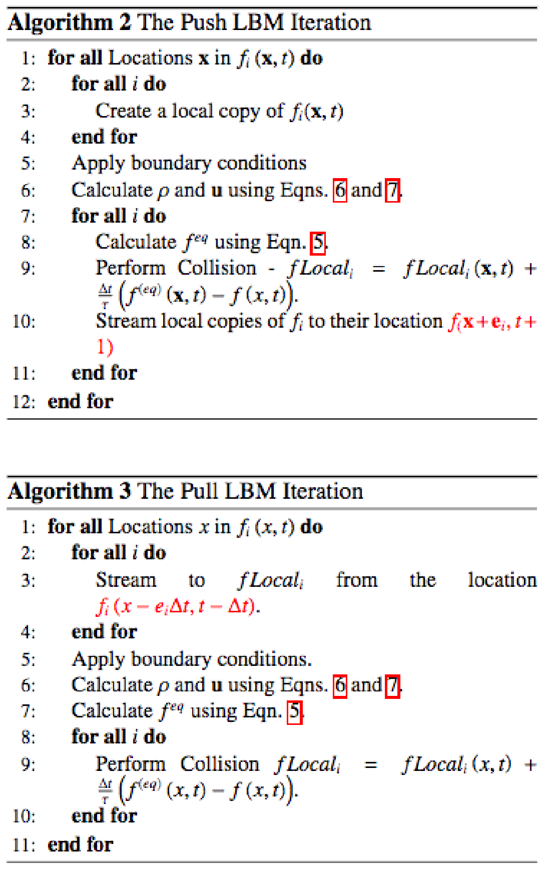
BitBucket Commit: 228babc

```
  // MAIN SIMULATION LOOP
  time_t begin = clock();
  for (int i = 0; i < num_iter; i++) {
    push_lbm <<<grid_of_blocks, block_of_threads>>> (fLocal[0], f_temp, N, tau, iBlock[0]); // *fLocal[0] is ...
        an array of length 27, with n^3 rows

    pull_lbm <<<grid_of_blocks, block_of_threads>>> (fLocal[0], f_temp, N, tau, iBlock[0]); // *fLocal[0] is ...
        an array of length 27, with n^3 rows
  }
    set_f_temp <<< grid_of_blocks, block_of_threads >>> (f_temp, fLocal[0], N);
    time_t end = clock();
    double runtime = (double)(end-begin)/CLOCKS_PER_SEC;
    double gpu_runtime = 0.001*0.0f;

    size_t doubles_read = 27; // per node every time step
    size_t doubles_written = 27;
    size_t doubles_saved = 0; // per node every NSAVE time steps

    // note NX*NY overflows when NX=NY=65536
    size_t nodes_updated = num_iter*size_t(N*N*N);
    size_t nodes_saved   = 0;
    double speed = nodes_updated/(1e6*runtime);

    double bandwidth = (nodes_updated*(doubles_read + ...
        doubles_written)+nodes_saved*(doubles_saved))*sizeof(double)/(runtime*bytesPerGiB);

    printf("Performance Statistics: \n");
    printf("GPU Memory Allocated: %.1f (MiB)\n",432*N*N*N/bytesPerMiB);
    printf("Number of Iterations: %u\n",num_iter);
    printf("Runtime: %.3f (s)\n",runtime);
    printf("Speed: %.2f (Mlups)\n",speed);
    printf("Bandwidth: %.1f (GiB/s)\n",bandwidth);



  PetscFunctionReturn(0);
}

__global__ void set_f_temp(PetscReal (*f)[27], double (*f_temp)[27], size_t n) {
  int x = threadIdx.x;
  int y = threadIdx.y;
  int z = blockIdx.x;
  int pos_index = x + n * (y + n * z); // current (x, y, z)
  for (int i = 0; i < 27; i++) {
    f_temp[pos_index][i] = f[pos_index][i];
  }
}

__global__ void push_lbm(PetscReal (*f)[27], double (*f_temp)[27], size_t n, double tau, struct IndexBlock ...
    iBlock)
{
  const double tau_reciprocal = 1/tau;
  const double one_minus_tau_reciprocal = 1 - tau_reciprocal;
  int x = threadIdx.x;
  int y = threadIdx.y;
  int z = blockIdx.x;

  int pos_index = x + n * (y + n * z); // current (x, y, z) corresponds to a pos_index in the global f array

  double rho = 0, u = 0, v = 0, w = 0;
  PetscReal f_loc[27];
  PetscReal f_loc2[27];
  for (int dir = 0; dir < 27; dir++) {
    f_loc[dir]=f[pos_index][dir];

    rho += f_loc[dir];
    u += f_loc[dir] * gpu_lbm_velocities[dir][0];
    v += f_loc[dir] * gpu_lbm_velocities[dir][1];
    w += f_loc[dir] * gpu_lbm_velocities[dir][2];
  }

  double rhoinv = 1.0/rho;

  u *= rhoinv; v *= rhoinv; w *= rhoinv;
  for (int dir = 0; dir < 27; dir++) {

    size_t x_periodic = (n + x + gpu_lbm_velocities[dir][0]) % n;
    size_t y_periodic = (n + y + gpu_lbm_velocities[dir][1]) % n;
    size_t z_periodic = (n + z + gpu_lbm_velocities[dir][2]) % n;
    // Convert Cartesian coordinates to n*n*n row index:
    size_t new_pos = x_periodic + y_periodic * n + z_periodic * n * n;
    // calculate u_alpha (the dot product)
    double u_alpha = gpu_lbm_velocities[dir][0] * u +
      gpu_lbm_velocities[dir][1] * v + gpu_lbm_velocities[dir][2] * w;
    // calculate equilibrium distribution
    double f_equil_i = gpu_lbm_weights[dir] * rho *
      (1. + 3. * u_alpha + 4.5 * u_alpha * u_alpha - 1.5*(u*u + v*v + w*w));
    double temp = tau_reciprocal * f_equil_i + one_minus_tau_reciprocal * f_loc[dir];
    f_temp[pos_index][dir] = temp;
    f[new_pos][dir] = temp;

  }
}
__global__ void pull_lbm(PetscReal (*f)[27], double (*f_temp)[27], size_t n, double tau, struct IndexBlock ...
    iBlock)
{
  const double tau_reciprocal = 1/tau;
  const double one_minus_tau_reciprocal = 1 - tau_reciprocal;
  int x = threadIdx.x;
  int y = threadIdx.y;
  int z = blockIdx.x;

  int pos_index = x + n * (y + n * z); // current (x, y, z) corresponds to a pos_index in the global f array

  double rho = 0, u = 0, v = 0, w = 0;
  PetscReal f_loc[27];

  for (int dir = 0; dir < 27; dir++) {
    size_t x_periodic = (n + x - gpu_lbm_velocities[dir][0]) % n;
    size_t y_periodic = (n + y - gpu_lbm_velocities[dir][1]) % n;
    size_t z_periodic = (n + z - gpu_lbm_velocities[dir][2]) % n;
    // Convert Cartesian coordinates to n*n*n row index:
    size_t old_pos = x_periodic + y_periodic * n + z_periodic * n * n;
    f_loc[dir] = f_temp[pos_index][dir];
    rho += f_loc[dir];
    u += f_loc[dir] * gpu_lbm_velocities[dir][0];
    v += f_loc[dir] * gpu_lbm_velocities[dir][1];
    w += f_loc[dir] * gpu_lbm_velocities[dir][2];
  }

  double rhoinv = 1.0/rho;

  u *= rhoinv; v *= rhoinv; w *= rhoinv;
  for (int dir = 0; dir < 27; dir++) {
    // calculate u_alpha (the dot product)
    double u_alpha = gpu_lbm_velocities[dir][0] * u +
      gpu_lbm_velocities[dir][1] * v + gpu_lbm_velocities[dir][2] * w;
    // calculate equilibrium distribution
    double f_equil_i = gpu_lbm_weights[dir] * rho *
```

```
        (1. + 3. * u_alpha + 4.5 * u_alpha * u_alpha - 1.5*(u*u + v*v + w*w));
      f_temp[pos_index][dir] = tau_reciprocal * f_equil_i + one_minus_tau_reciprocal * f_loc[dir];

    }
}
```

7.2 Proposed Change and Expected Improvements

After looking through the literature on implementations of the Lattice Boltzmann Method on GPU's, optimal methods that reduced memory transfers were further considered. Finally a paper by Mawson & Revell [1] was chosen to be implemented because the high performance boost was reported and the implementation didn't seem too ambitious given the time constraints. A limitation that was easily recognized was the need to replace f_temp_i distributions with f_i distributions with every iteration. This replacement is necessary in the base case implementation because a lattice nodes original distributions are required to do the collision and propagation step but the stream step propagates the distribution to neighboring nodes. Thus, 2 distributions need to be maintained to keep track of the original distribution in that iteration and the distribution after collisions. We found an alternate implementation in this article that gets around the need to swap the original and post collision distributions by doing an iteration of pushing the original distributions and an iteration pulling the post-collision distribution. These iterations would avoid the need to have to replace the original distribution array at each iteration. The proposed pseudo code from their work is given for reference.



Algorithm 2 The Push LBM Iteration

```
1: for all Locations x in $f_i(\mathbf{x}, t)$ do
2:     for all $i$ do
3:         Create a local copy of $f_i(\mathbf{x}, t)$
4:     end for
5:     Apply boundary conditions
6:     Calculate $\rho$ and $\mathbf{u}$ using Eqns. 6 and 7.
7:     for all $i$ do
8:         Calculate $f^{eq}$ using Eqn. 5.
9:         Perform Collision - $fLocal_i = fLocal_i(\mathbf{x}, t) + \frac{\Delta t}{\tau}\left(f^{(eq)}(\mathbf{x}, t) - f(x, t)\right)$.
10:        Stream local copies of $f_i$ to their location $f_i \mathbf{x} + \mathbf{e}_i, t + 1)$
11:    end for
12: end for
```

Algorithm 3 The Pull LBM Iteration

```
1: for all Locations $x$ in $f_i(x, t)$ do
2:     for all $i$ do
3:         Stream to $fLocal_i$ from the location $f_i(x - e_i\Delta t, t - \Delta t)$.
4:     end for
5:     Apply boundary conditions.
6:     Calculate $\rho$ and $\mathbf{u}$ using Eqns. 6 and 7.
7:     Calculate $f^{eq}$ using Eqn. 5.
8:     for all $i$ do
9:         Perform Collision $fLocal_i = fLocal_i(x, t) + \frac{\Delta t}{\tau}\left(f^{(eq)}(x, t) - f(x, t)\right)$.
10:    end for
11: end for
```

Figure 12: Pseudo code describing the algorithm behind the push and pull implementation of the GPU-based LBM implementation

7.3 Profiling Results

Successful implementation of the pseudocode couldn't be achieved as it fails one of the tests in the harness, although the implemented version in the given commit comes close to the tolerance of .88 with an error of .92 for the $n = 15$ gridsize. We

[1]Mawson, Mark J., and Alistair J. Revell. "Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs." Computer Physics Communications 185.10 (2014): 2566-2574.

believe this implementation is correct except for a small bug which we couldn't figure out. The result from the terminal is given below for validation:

Listing 5: Profiling results obtained using nvprof for the push-pull implementation.

```
edwin@Edwin-MacBuntu:~/final/Lattice-Boltzmann-Method/parallel$ make clean && make && nvprof ./test_lbm
==18515== NVPROF is profiling process 18515, command: ./test_lbm
Has 1140850688 communicators
Running 10 tests of Lattice Boltzmann method with 1 devices
  Test 0:
    Test dimensions: [13 x 13 x 13], 169 iterations, tolerance 1.18343
    Initializing LBM calculation...
    Total points = 13 * 13 * 13 = 2197
    Allocated grid of blocks with dimensions: (13, 1, 1); 13 blocks total
    Allocated 13 thread blocks with dimensions: (13, 13, 1); 169 threads per block
    13 blocks * 169 threads per block = 2197 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 0.9 (MiB)
Number of Iterations: 169
Runtime: 0.003 (s)
Speed: 110.05 (Mlups)
Bandwidth: 44.3 (GiB/s)
      Error 0.394037
      Passed.
  Test 1:
    Test dimensions: [8 x 8 x 8], 64 iterations, tolerance 3.125
    Initializing LBM calculation...
    Total points = 8 * 8 * 8 = 512
    Allocated grid of blocks with dimensions: (8, 1, 1); 8 blocks total
    Allocated 8 thread blocks with dimensions: (8, 8, 1); 64 threads per block
    8 blocks * 64 threads per block = 512 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 0.2 (MiB)
Number of Iterations: 64
Runtime: 0.002 (s)
Speed: 21.73 (Mlups)
Bandwidth: 8.7 (GiB/s)
      Error 0.601047
      Passed.
  Test 2:
    Test dimensions: [8 x 8 x 8], 64 iterations, tolerance 3.125
    Initializing LBM calculation...
    Total points = 8 * 8 * 8 = 512
    Allocated grid of blocks with dimensions: (8, 1, 1); 8 blocks total
    Allocated 8 thread blocks with dimensions: (8, 8, 1); 64 threads per block
    8 blocks * 64 threads per block = 512 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 0.2 (MiB)
Number of Iterations: 64
```

```
Runtime: 0.001 (s)
Speed: 24.69 (Mlups)
Bandwidth: 9.9 (GiB/s)
      Error 0.601047
      Passed.
  Test 3:
    Test dimensions: [9 x 9 x 9], 81 iterations, tolerance 2.46914
    Initializing LBM calculation...
    Total points = 9 * 9 * 9 = 729
    Allocated grid of blocks with dimensions: (9, 1, 1); 9 blocks total
    Allocated 9 thread blocks with dimensions: (9, 9, 1); 81 threads per block
    9 blocks * 81 threads per block = 729 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 0.3 (MiB)
Number of Iterations: 81
Runtime: 0.003 (s)
Speed: 19.10 (Mlups)
Bandwidth: 7.7 (GiB/s)
      Error 0.497029
      Passed.

  Test 5:
    Test dimensions: [8 x 8 x 8], 64 iterations, tolerance 3.125
    Initializing LBM calculation...
    Total points = 8 * 8 * 8 = 512
    Allocated grid of blocks with dimensions: (8, 1, 1); 8 blocks total
    Allocated 8 thread blocks with dimensions: (8, 8, 1); 64 threads per block
    8 blocks * 64 threads per block = 512 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 0.2 (MiB)
Number of Iterations: 64
Runtime: 0.001 (s)
Speed: 24.27 (Mlups)
Bandwidth: 9.8 (GiB/s)
      Error 0.601047
      Passed.
  Test 6:
    Test dimensions: [8 x 8 x 8], 64 iterations, tolerance 3.125
    Initializing LBM calculation...
    Total points = 8 * 8 * 8 = 512
    Allocated grid of blocks with dimensions: (8, 1, 1); 8 blocks total
    Allocated 8 thread blocks with dimensions: (8, 8, 1); 64 threads per block
    8 blocks * 64 threads per block = 512 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 0.2 (MiB)
Number of Iterations: 64
Runtime: 0.002 (s)
Speed: 20.24 (Mlups)
Bandwidth: 8.1 (GiB/s)
      Error 0.601047
      Passed.
  Test 7:
    Test dimensions: [10 x 10 x 10], 100 iterations, tolerance 2.
    Initializing LBM calculation...
    Total points = 10 * 10 * 10 = 1000
    Allocated grid of blocks with dimensions: (10, 1, 1); 10 blocks total
    Allocated 10 thread blocks with dimensions: (10, 10, 1); 100 threads per block
    10 blocks * 100 threads per block = 1000 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 0.4 (MiB)
Number of Iterations: 100
Runtime: 0.002 (s)
Speed: 49.85 (Mlups)
Bandwidth: 20.1 (GiB/s)
      Error 0.415925
      Passed.
  Test 8:
    Test dimensions: [15 x 15 x 15], 225 iterations, tolerance 0.888889
    Initializing LBM calculation...
    Total points = 15 * 15 * 15 = 3375
    Allocated grid of blocks with dimensions: (15, 1, 1); 15 blocks total
    Allocated 15 thread blocks with dimensions: (15, 15, 1); 225 threads per block
    15 blocks * 225 threads per block = 3375 threads total
Remote CUDA information:
Device name: GeForce GT 650M
# of multiprocessors: 2
compute capability: 3.0
GPU global memory available: 976.5 MiB

Performance Statistics:
GPU Memory Allocated: 1.4 (MiB)
Number of Iterations: 225
Runtime: 0.005 (s)
Speed: 167.45 (Mlups)
Bandwidth: 67.4 (GiB/s)
      Error 0.92373
[0]PETSC ERROR: --------------------- Error Message ...
    ------------------------------------------------------------
[0]PETSC ERROR: Error in external library
[0]PETSC ERROR: Test 8 failed residual test at threshold 0.888889 with value 0.92373

system msg for write_line failure : Bad file descriptor
==18515== Profiling application: ./test_lbm
==18515== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 50.61%  339.08ms       952  356.18us  165.72us  713.42us  pull_lbm(double[27]*, double[27]*, unsigned long, ...
     double, IndexBlock)
 49.07%  328.82ms       952  345.40us  162.05us  675.25us  push_lbm(double[27]*, double[27]*, unsigned long, ...
     double, IndexBlock)
  0.20%  1.3705ms         9  152.28us  76.734us  461.53us  set_f_temp(double[27]*, double[27]*, unsigned long)
  0.06%  391.19us        10  39.119us  1.5680us  115.90us  [CUDA memcpy HtoD]
  0.06%  373.53us         9  41.503us  18.559us  114.81us  [CUDA memcpy DtoH]

======== Error: Application returned non-zero code 76
```

7.4 Analysis

Although this implementation fails for gridsizes larger than 15, the below results were achieved for a gridsize of 10. The total time for this implementation was 293.7 ms in comparison to a total time of 368.23 ms for the base case code on the same machine. Thus, this implementation would have led to a speed up in the performance in comparison to the base case.

This optimization though performs better than the first optimization with prefetching combined with the base case, which achieved a total time of 300.12 ms. Thus, if the bug contributing to elevated errors could have been figured out the push pull implementation would have been included in the final.
We believe our implementation of the given psuedocode fails at larger test sizes due to a mixup between the f_i and f_temp_i distributions somewhere in the code. Even as is though, at smaller test sizes this code is better than the first optimization implemented and holds good potential if debugged. The improvement in performance that can already be seen is due to the removal of the need to read and write over an array the size of the number of points with each iteration.

Listing 6: Results for gridsize 10 obtained using nvprof for the push-pull implementation.

```
    ==19702== Profiling application: ./test_lbm -scale 10
==19702== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 60.02%  176.58ms      1000  176.58us  172.70us  223.68us  push_lbm(double[27]*, double[27]*, unsigned long, ...
     double, IndexBlock)
 39.45%  116.07ms      1000  116.07us  112.96us  125.18us  pull_lbm(double[27]*, double[27]*, unsigned long, ...
     double, IndexBlock)
  0.27%  804.78us        10  80.478us  79.935us  80.990us  set_f_temp(double[27]*, double[27]*, unsigned long)
  0.13%  385.15us        11  35.013us  1.4400us  45.055us  [CUDA memcpy HtoD]
  0.12%  356.12us        10  35.612us  34.943us  36.608us  [CUDA memcpy DtoH]
```

8. Final Analysis

8.1 Profiling Results from Best-Performing Code

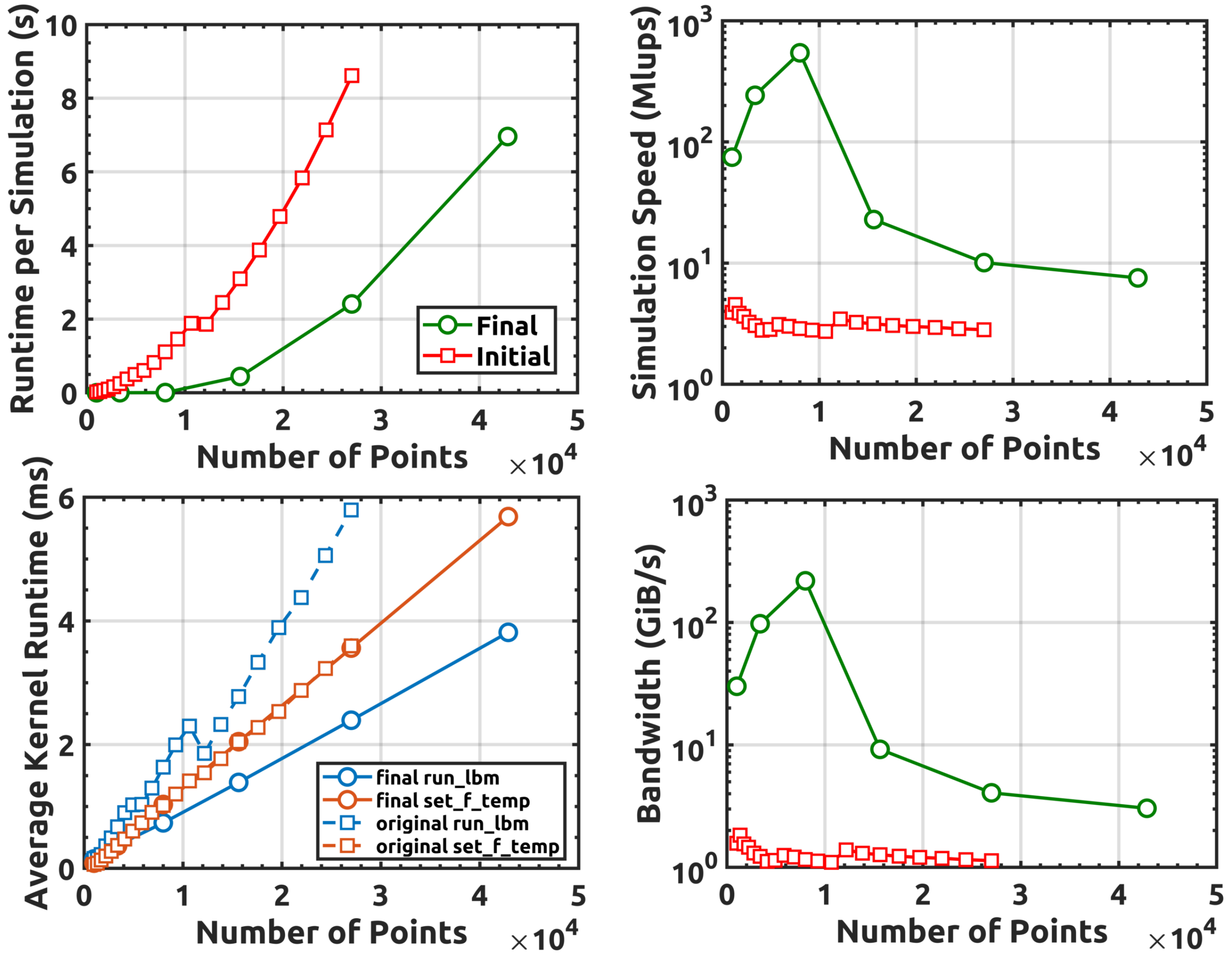


Figure 13: Comparison of important performance metrics between best-performing and original code.
result

Comparison with the prior submission shows the dramatic improvement in performance our new submission achieves. If n is the total number of points, then the total time for the final submission scales quadratically with problem size but at a slower rate than the initial submission. Similar to before, time to transfer in and out of device still scales in O(n). The set_ftemp functions could be used as proxy for memory latency because it only has memory operations and as shown in the final figure, it scales in O(n). It hasn't changed at all because the optimizations to remove the need for set_ftemp were unsuccessful. On the other hand, the final optimizations dramatically improved the performance of the run_lbm kernel. Optimization with grid and block sizes allowed dramatic increases in bandwidth because the sizes were optimized so as to allow more blocks to be run at a given time. All these optimizations dramatically increased the speed of the final submission especially because they combat the memory transfer and latency bottlenecks.

8.2 Analysis and Discussion of Improvements and Remaining Bottlenecks

The three main optimizations that were included in the final code was memory prefetching, optimizing the grid/block size, and removing the CudaDeviceSynchronize.
Fundamentally, the LBM method doesn't require any calls from CPU to device calls, and so the optimizations looked into were to minimize device memory read/writes and latency as well as grid size optimizations. Prefetching the f distributions

reduced the amount of memory reads, and the optimal grid sizing ensured there was enough shared memory for all the threads in a thread block so that storing in global memory could be avoided.
After further analysis, the compute operations were deemed to be already optimal and no further optimizations in this area could be thought of. Even in the original implementation that was submitted for check point 3, optimizations such as storing the output of divisions were precomputed and stored as much as possible. Thus, no additional compute optimizations were found.
As discussed in checkpoint 2, it was deemed that the primary avenue for optimizations with the Lattice Boltzmann Method should be to optimize the memory latency and transfers. The memory latency and transfer bottlenecks were as expected and optimizations in this area dramatically sped up performance. The analysis with varying grid sizes at the beginning of this report further confirmed this hunch. Additionally, memory transfer optimizations such as prefetching gave the expected boost in performance achieving a 20% in improvement. Similarly, optimizing the grid size ensured enough memory cache locally for threads so as to speed up performance.

8.3 Further Optimization

As mentioned before, if the push-pull implementation could have been debugged this would have been included in the final submission because this would have dramatically reduced the read/writes per iteration, which is especially beneficial as we have determined our algorithm is memory transfer and latency bound. A shared memory structure combined with grid size optimization could have been implemented so as to ensure that each thread block has enough memory to store distributions in shared memory by limiting the number of threads per block. But as is, the optimizations implemented in the final submission have lead to a dramatic speedup in speed compared to the original.

Further improvements that could be attempted to increase memory bandwidth include reorganizing the array of velocity densities and the kernels such that streaming could be performed across threads in a block. This would require the use of halo points which make this a complex endeavor, but the potential performance benefits from having data transfer at the L1 level might be worth the extra programming effort.

9. Conclusions and Summary

In this project, a serial implementation of the Lattice Boltzmann Method (LBM) was developed (see "serial" folder in d8a7a7). The Taylor-Green Vortex, an idealized flow whereby analytical solutions to the Navier-Stokes equations could be found, was used for validation of the implementation's correctness.

A parallel implementation was then performed, with the platform of choice being a GPU-based solution due to the high memory transfer requirements in the streaming operations involved in the LBM algorithm. The GPU implementation was optimized by:

1. Changing the grid-block structure to allow for more threads/warps to be executed at a given time, i.e. higher concurrency. This allowed for increased bandwidth and therefore reduced runtime (See Section 5.
2. Using local copies of frequently used variables within the CUDA kernels in order to reduce memory accesses to and from global *device* memory (See Section 4).
3. Removing the `cudaDeviceSynchronize` function with each iteration reduced the overall time of the simulation without affecting accuracy. This resulted in the largest performance improvement in this brief optimization of the LBM code. (See Section 6)