Hudson Connolly
Math 4390
5/6/2019

Brute Force N-Body Optimization

The goal of this project is the optimization of a Brute-force N-Body CUDA kernel. The testing environment used was a GTX 1060 at 100% theoretical occupancy (20480 bodies). Timing for each kernel is the average execution time of one thousand runs. All graphs and data tables were generated by Nvidia Nsight profiling tool for CUDA kernels. Each attached spreadsheet contains detailed information of the first one hundred runs for the corresponding kernel. The restrictions place on this project do not allow further approximations of positional and force related calculations on the bodies, other then the techniques provided in the sample code. The focus of improvement will be on leveraging a understanding of GPU architecture, and API framework, to yield faster results.

nbodyGPU5

time = 95.8ms

Analyses on the "nBodyGPU5" kernel shows the primarily cause of slow down to be Warp-stalls. A Warp is a group of 32 threads which get distributed instructions from a entity called a warp manager. A Warp-Stall can happen for a variety of reasons, but in general disrupt the flow of instructions to its threads. The main cause of the "nBodyGPU5" slowdown is due to a specific type of Warp-Stall called execution dependency. Execution dependency is when a instruction is dependent on a proceeding instruction. Examples of this include control structures and the use of multiple operators that do not match to a single CUDA instruction. In the latter case intermediate results need to loaded and stored to registers multiple times. This puts a high degree of pressure on both the load/store units and CUDA cores of a SM(streaming multiprocessor).

```
83  __device__ float3 getBodyBodyForce(float4 p0, float4 p1)
84  {
85      float3 f;
86      float dx = p1.x - p0.x;
87      float dy = p1.y - p0.y;
88      float dz = p1.z - p0.z;
89      float r2 = dx*dx + dy*dy + dz*dz;
90      float r = sqrt(r2);
91
92      float force = (G*p0.w*p1.w) / (r2)-(H*p0.w*p1.w) / (r2*r2);
93
94      f.x = force*dx / r;
95      f.y = force*dy / r;
96      f.z = force*dz / r;
97
98      return(f);
99  }
```

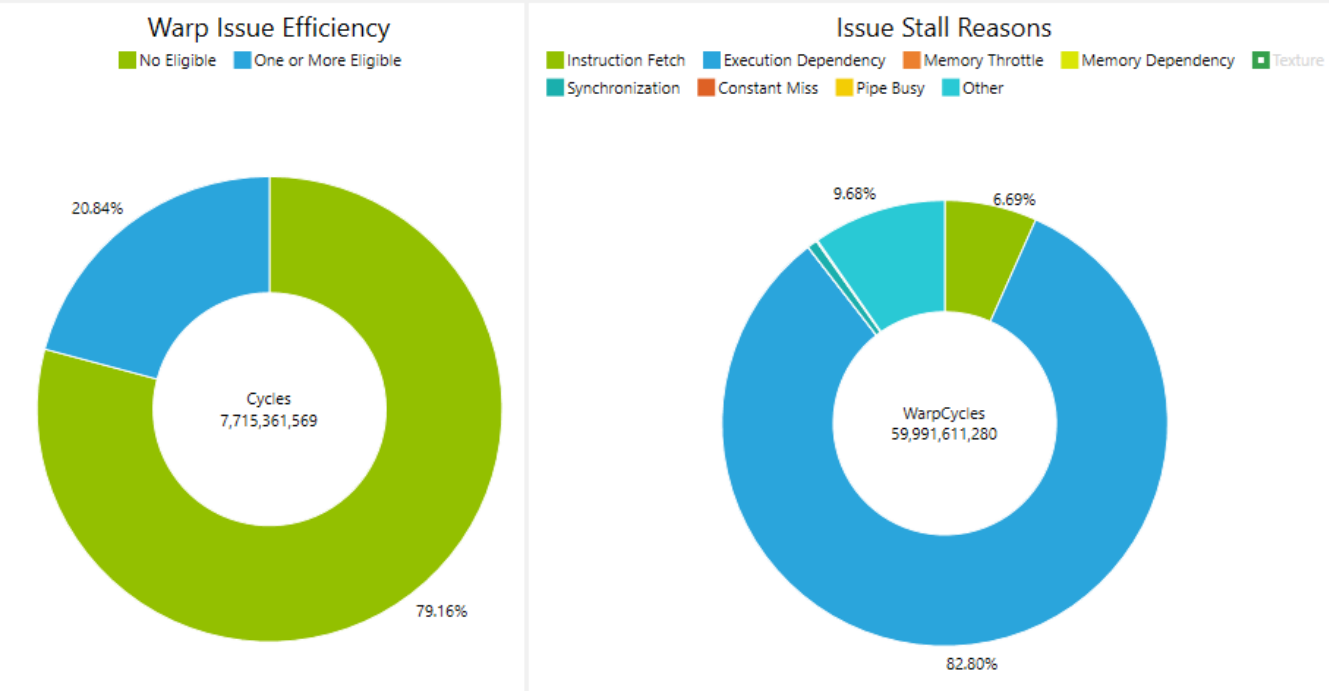
NbodyGPU5: GPU code snippet (1)

```
__global__ void getForces(float4 *pos, float3 *vel, float3 * force)
{
    int j, ii;
    float3 force_mag, forceSum;
    float4 posMe;
    __shared__ float4 shPos[BLOCK];
    int id = threadIdx.x + blockDim.x*blockIdx.x;

    forceSum.x = 0.0;
    forceSum.y = 0.0;
    forceSum.z = 0.0;

    posMe.x = pos[id].x;
    posMe.y = pos[id].y;
    posMe.z = pos[id].z;
    posMe.w = pos[id].w;

    for (j = 0; j < gridDim.x; j++)
    {
        shPos[threadIdx.x] = pos[threadIdx.x + blockDim.x*j];
        __syncthreads();

        #pragma unroll 32
        for (int i = 0; i < blockDim.x; i++)
        {
            ii = i + blockDim.x*j;
            if (ii != id && ii < N)
            {
                force_mag = getBodyBodyForce(posMe, shPos[i]);
                forceSum.x += force_mag.x;
                forceSum.y += force_mag.y;
                forceSum.z += force_mag.z;
            }
        }
    }
    if (id <N)
    {
        force[id].x = forceSum.x;
        force[id].y = forceSum.y;
        force[id].z = forceSum.z;
    }
}
```

NbodyGPU5: GPU code snippet (2)

```
144     __global__ void moveBodies(float4 *pos, float3 *vel, float3 * force)
145     {
146         int id = threadIdx.x + blockDim.x*blockIdx.x;
147         if (id < N)
148         {
149             vel[id].x += ((force[id].x - DAMP*vel[id].x) / pos[id].w)*DT;
150             vel[id].y += ((force[id].y - DAMP*vel[id].y) / pos[id].w)*DT;
151             vel[id].z += ((force[id].z - DAMP*vel[id].z) / pos[id].w)*DT;
152
153             pos[id].x += vel[id].x*DT;
154             pos[id].y += vel[id].y*DT;
155             pos[id].z += vel[id].z*DT;
156         }
157     }
```

NbodyGPU5: GPU code snippet (3)



NbodyGPU5: Warp Efficiency Graph

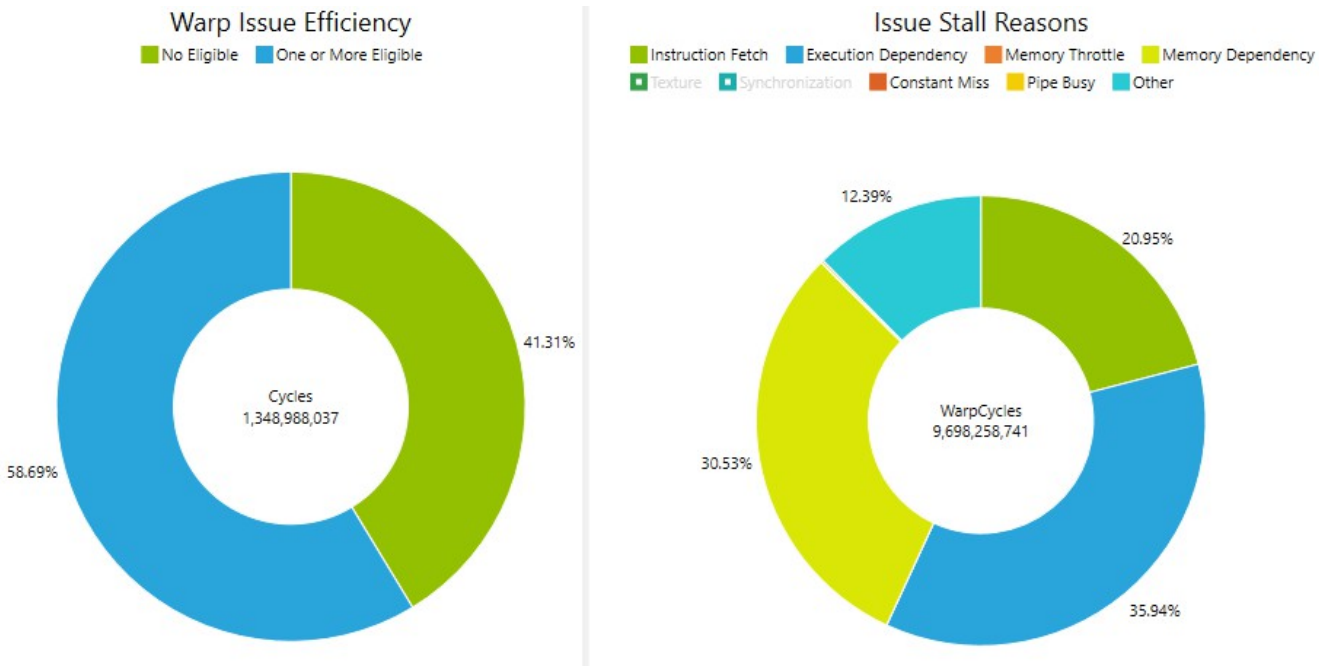
Nbody kernel v1

time = 16.5ms, speedup = 5.81

The first area of improvement is to optimize the attraction force equation by eliminating costly mathematical instructions. Secondly, reduce potential thread divergence by reducing the amount of conditional statements where possible. These two changes reduced Warp-Stalls by 37.85% increasing the kernels instruction throughput. The final modification is the reduction of the kernels global memory footprint. Because force is not persistence between kernel executions it can be applied directly to velocity without any performance penalty. Many of the conditional statements in "nbodyGPU5" served the purpose of maintaining a efficient memory access pattern for each thread. By removing the shared memory, and associated conditionals, memory dependency is accounting for 30.53% of our Warp-Stalls. Because each thread in this kernel is making a separate transaction to the same global memory address the SM is unable to fulfill all requests in parallel. In practice CUDA detects and attempts minimize this problem by broadcasting the retrieved data between threads in a half-warp. A broadcast is this case would reduce the total global memory transactions from a block of 1024 to 64 transactions (blockSize/(warpSize/2)). However, 64 transactions per block is still to large for a single SM to handle concurrently which results in threads waiting for their data.

```
27  __global__ void getVelocity(float4 *pos, float3 *vel) {
28      const unsigned int id = threadIdx.x + blockDim.x * blockIdx.x;
29      if (id >= N) return;
30
31      const float4 myPos = pos[id];
32      float4 force = {0.0f, 0.0f, 0.0f};
33
34      for (int i = 0; i < N; i++) {
35          float4 p = pos[i];
36
37          float dx = p.x - myPos.x;
38          float dy = p.y - myPos.y;
39          float dz = p.z - myPos.z;
40
41          float r2 = dx*dx + dy*dy + dz*dz + EPSILON;
42          float r = 1.0f / sqrtf(r2);
43          float mag = (G*p.w*myPos.w) / (r2)-(H*p.w*myPos.w) / (r2*r2);
44
45          force.x += mag * dx * r;
46          force.y += mag * dy * r;
47          force.z += mag * dz * r;
48      }
49      vel[id].x += ((force.x - DAMP*vel[id].x) / myPos.w)*DT;
50      vel[id].y += ((force.y - DAMP*vel[id].y) / myPos.w)*DT;
51      vel[id].z += ((force.z - DAMP*vel[id].z) / myPos.w)*DT;
52  }
53
54  __global__ void move(float4 *pos, float3 *vel) {
55      int id = threadIdx.x + blockDim.x * blockIdx.x;
56      if (id >= N) return;
57      pos[id].x += vel[id].x*DT;
58      pos[id].y += vel[id].y*DT;
59      pos[id].z += vel[id].z*DT;
60  }
```

Nbody kernel v1: GPU code snippet



Nbody kernel v1: Warp Efficiency Graph

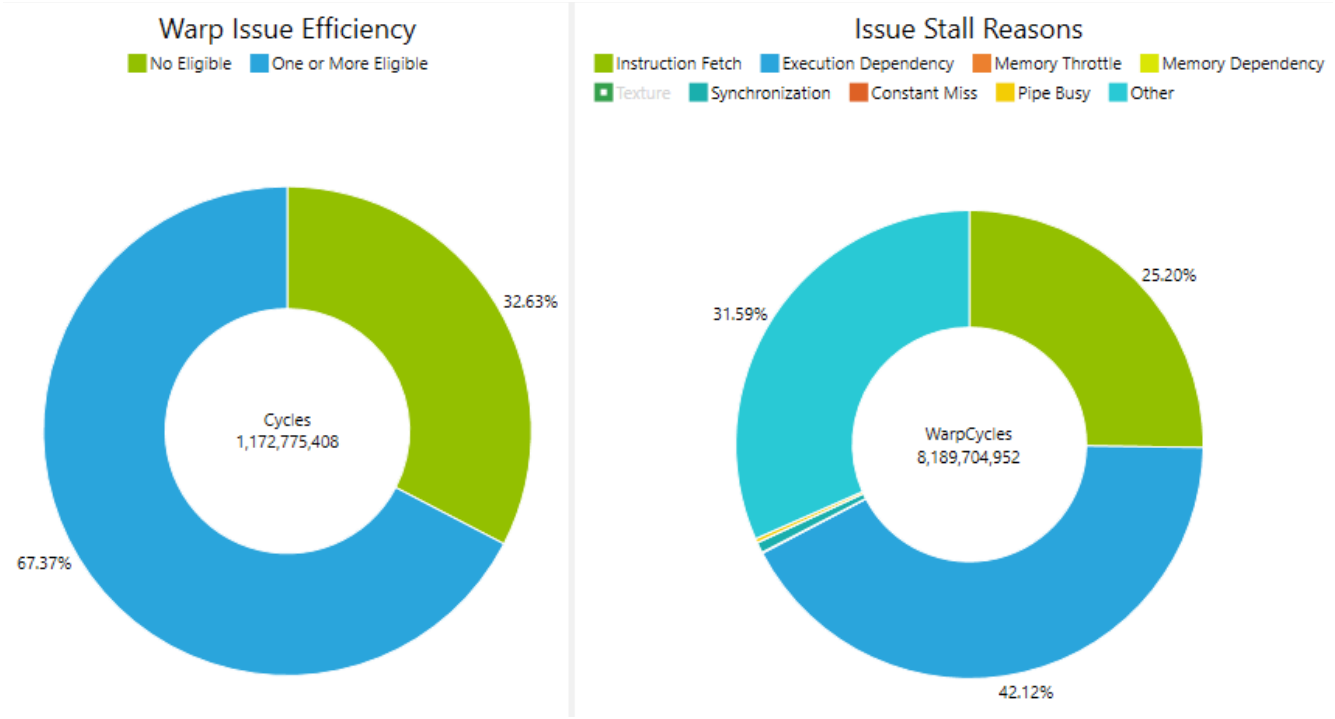
Nbody kernel v3

time = 14.64ms, speedup = 6.54

While building on the improvements in the previous kernel, attempts are made to reduce the Warp-Stalls caused by memory dependency. Nivida GPU's provide systems for improved global memory accessing under specific circumstances. If each thread within a block accesses a unique memory address which results in a contiguous access pattern, then a single memory transaction occurs for the contiguous memory chunk. The size of each of these memory chunks will be the amount of threads in a block. Because each thread needs to calculate the forces acting upon its body by all other bodies, shared memory is used to temporarily store the data pulled from global memory. To ensure shared memory is visible to all threads in a block, synchronization must occur after new data is pulled from global memory. Due to the use of shared memory this kernel has better overall performance verses previous iterations. The problems with this approach come with the restrictions imposed by the use of shared memory. Because blocks must synchronize, processing any number of bodies which are not divisible by the block size will cause out of bounds memory accesses. Decreasing the block size, from maximum size, will decrease the kernels performance in several ways. Firstly, as block size decreases the amount of global memory transactions, per block, increases. Secondly, the occupancy will be reduced if the block size is not a power of two. The trade-offs for efficient memory access come at the price of kernel flexibility.

```
28  __global__ void getVelocity(float4 *pos, float3 *vel) {
29      
30      const unsigned int id = threadIdx.x + blockDim.x * blockIdx.x;
31      __shared__ float4 sharedPos[BLOCK];
32      const float4 myPos = pos[id];
33      float4 force = { 0.0f, 0.0f, 0.0f };
34
35      for (int gMem_Index = threadIdx.x; gMem_Index < N; gMem_Index += blockDim.x) {
36
37          sharedPos[threadIdx.x] = pos[gMem_Index]; __syncthreads();
38
39          for (int i = 0; i < blockDim.x; i++) {
40              float dx = sharedPos[i].x - myPos.x;
41              float dy = sharedPos[i].y - myPos.y;
42              float dz = sharedPos[i].z - myPos.z;
43
44              float r2 = dx*dx + dy*dy + dz*dz + EPSILON;
45              float r = 1.0f / sqrtf(r2);
46              float mag = (G*myPos.w*sharedPos[i].w) / (r2)-(H*myPos.w*sharedPos[i].w) / (r2*r2);
47
48              force.x += mag * dx * r;
49              force.y += mag * dy * r;
50              force.z += mag * dz * r;
51          }
52      }
53      vel[id].x += ((force.x - DAMP*vel[id].x) / myPos.w)*DT;
54      vel[id].y += ((force.y - DAMP*vel[id].y) / myPos.w)*DT;
55      vel[id].z += ((force.z - DAMP*vel[id].z) / myPos.w)*DT;
56  }
57
58  __global__ void move(float4 *pos, float3 *vel) {
59      int id = threadIdx.x + blockDim.x * blockIdx.x;
60      if (id >= N) return;
61      pos[id].x += vel[id].x*DT;
62      pos[id].y += vel[id].y*DT;
63      pos[id].z += vel[id].z*DT;
64  }
```

Nbody kernel v3: GPU code snippet



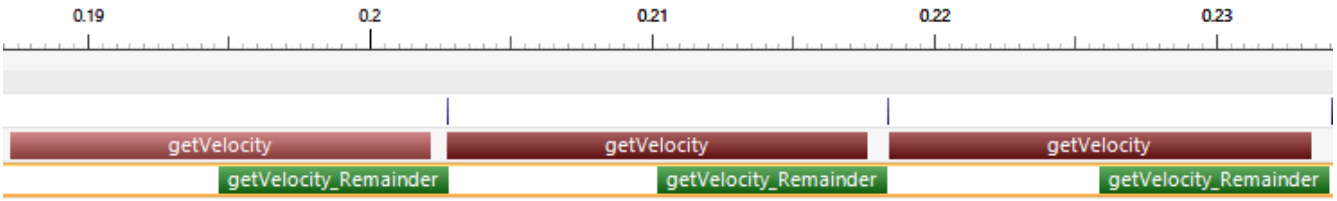
Nbody kernel v3: Warp Efficiency Graph

Nbody kernel v5

Best Case Time = 14.64ms, speedup = 6.54

Worst Case Time = 15.88ms, speedup = 6.03

The third and final approach attempts to hybridize the previous two kernels by selective use depending on the situation. The "Nbody kernel v3" will be assigned a subset of the bodies that are divisible by its block size. If a remainder of bodies exists, then "Nbody kernel v1" will be used to and both kernels will run in parallel with streams. In in the best case scenario all bodies can be processed by "Nbody kernel v3". In the worst case scenario the amount remainder bodies is very close to, but less then, the block size. "Nbody kernel v5" gives us flexible kernel while retaining as much of the performance from "Nbody kernel v3" as possible.



Nbody kernel v5: Worst Case scenario timeline snippet

Because the getVelocity from "Nbody kernel v3" is only designed for retrieving body positions in block sized chunks, the memory access pattern will have to account for position arrays of varying size. The inner loop can keep track of which indexes in shared memory have been updated by using the "__syncthreads_count()" instruction. If all threads in a block evaluate the predicate to be false, then the loop can safely be exited. Even with the added conditionals, the getVelocity from "Nbody kernel v3" maintains the same performance as the original.

```
29  __global__ void getVelocity(float4 *pos, float3 *vel) {
30      const unsigned int id = threadIdx.x + blockDim.x * blockIdx.x;
31      __shared__ float4 sharedPos[BLOCK];
32      const float4 myPos = pos[id];
33      float3 force = { 0.0f, 0.0f, 0.0f };
34
35      for (int gMem_Index = threadIdx.x; true; gMem_Index += blockDim.x) {
36
37          if (gMem_Index < N) sharedPos[threadIdx.x] = pos[gMem_Index];
38          int len = __syncthreads_count(gMem_Index < N);
39
40          if (len == 0) break;
41          for (int i = 0; i < len; i++) {
42              float dx = sharedPos[i].x - myPos.x;
43              float dy = sharedPos[i].y - myPos.y;
44              float dz = sharedPos[i].z - myPos.z;
45
46              float r2 = dx*dx + dy*dy + dz*dz + EPSILON;
47              float r = 1.0f / sqrtf(r2);
48              float mag = (G*myPos.w*sharedPos[i].w) / (r2)-(H*myPos.w*sharedPos[i].w) / (r2*r2);
49
50              force.x += mag * dx * r;
51              force.y += mag * dy * r;
52              force.z += mag * dz * r;
53          }
54      }
55      vel[id].x += ((force.x - DAMP*vel[id].x) / myPos.w)*DT;
56      vel[id].y += ((force.y - DAMP*vel[id].y) / myPos.w)*DT;
57      vel[id].z += ((force.z - DAMP*vel[id].z) / myPos.w)*DT;
58  }
```

Nbody kernel v5: GPU code snippet (1)

The only modification required for the “Nbody kernel v1” getVelocity, renamed to getVelocity_Remainder, is a offset to mark the tail-end of the body positions array.

```
60  __global__ void getVelocity_Remainder(float4 *pos, float3 *vel, int offset) {
61      const unsigned int id = (threadIdx.x + blockDim.x * blockIdx.x) + offset;
62      const float4 myPos = pos[id];
63      float4 force = { 0.0f, 0.0f, 0.0f };
64
65      for (int i = 0; i < N; i++) {
66          float4 p = pos[i];
67
68          float dx = p.x - myPos.x;
69          float dy = p.y - myPos.y;
70          float dz = p.z - myPos.z;
71
72          float r2 = dx*dx + dy*dy + dz*dz + EPSILON;
73          float r = 1.0f / sqrtf(r2);
74          float mag = (G*p.w*myPos.w) / (r2)-(H*p.w*myPos.w) / (r2*r2);
75
76          force.x += mag * dx * r;
77          force.y += mag * dy * r;
78          force.z += mag * dz * r;
79      }
80      vel[id].x += ((force.x - DAMP*vel[id].x) / myPos.w)*DT;
81      vel[id].y += ((force.y - DAMP*vel[id].y) / myPos.w)*DT;
82      vel[id].z += ((force.z - DAMP*vel[id].z) / myPos.w)*DT;
83  }
```

Nbody kernel v5: GPU code snippet (2)

Reference Material

- Nvidia Fermi Architecture
- CUDA C Best Practices Guide
- Cuda C Programming Guide
- Nvida TDCI Architecture
- Vasily Volkov - Better Performance at Lower Occupancy