N-Body Simulation - CUDA Optimizations

Vladyslav Bondarenko

It has been decided from the beginning to implement an N parallel version of the algorithm where for each body a single thread calculates its interaction with every other body sequentially. Although N*N version would have parallelism, it would also require a lot more memory movements and would likely be limited by the memory bandwidth.

The code was benchmarked by obtaining the run-time per iteration in milliseconds in release mode for different values of N (number of bodies) with D (activity dimensions) fixed to 256. Each value was obtained over an appropriate number of iterations for a few repetitions and averaged to get an accurate estimate of the actual time complexity. The full table of results for all of the optimizations can be found in Tab.I with appropriate descriptions for each experiment outlined below. Profiling of the code has been done in Visual Profiler with 32000 bodies to ensure full utilization of device over hundreds of iterations. Initial results indicated that using a 1D block of 256 threads, resulted in good occupancy (90%) and utilization and so all further experiments are run with this set-up.

I. Baseline Implementation

Initial implementation (taken directly from CPU code) had many flaws in the form of repetitive calculations and use of slow operations. Running the profiler further confirms this, highlighting the bottom two lines of code in the top of Fig.1 as key bottlenecks. Another sub-optimal calculation occurred in activity calculation (top of Fig.3), where a D * D * N iterations resulted in a large bottleneck. Those compute limitations are further demonstrated in top of Fig.2, with improved version at the bottom.

```
diff_x = b_j->x - b_i->x;
diff_y = b_j->y - b_i->y;

sum_x += (b_j->m * diff_x) / (float)pow((double)diff_x * diff_x + (double)SOFTENING2, 1.5f);
sum_y += (b_j->m * diff_y) / (float)pow((double)diff_y * diff_y + (double)SOFTENING2, 1.5f);
```

```
// x and y vector difference in positions
diff_x = b_j->x - b_i->x;
diff_y = b_j->y - b_i->y;

// Bottom part of the fraction within the summation
denom = (diff_x * diff_x) + (diff_y * diff_y) + SOFTENING2;
denom = sqrtf(denom * denom * denom);

// Calculate the force from body j
sum_x += (b_j->m * diff_x) / denom;
sum_y += (b_j->m * diff_y) / denom;
```

Fig. 1: Initial (bottom) and improved (top) force calculation

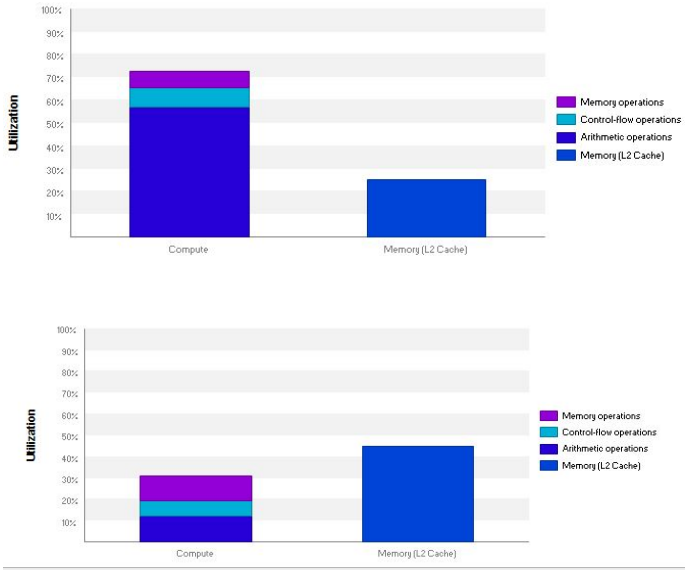

Fig. 2: Comparison of resource utilization for baseline implementation (top) and improved version (bottom).

A. Optimizing Main Loop

The calculation of the inverse square root (denominator) is redundant and needs only to be done once. As well as that, the power function is very compute-intensive and can be replaced with an sqrt. Those two improvements alone resulted in a great speed up for higher N, demonstrated by the difference in the first two columns in Tab I and further highlighted by visualization in Fig. 10.

B. Activity

Activity map update has been optimized by calculating the index within the activity map directly from the body position rather than manually checking every index. Optimized code is shown at the bottom of Fig. 3 where **between_1** is a defined macro function that checks if the body is within the limits of activity grid. To avoid race conditions between threads in the update, **atomicAdd** operation has been used. This optimization resulted in a great improvement in run time across all values of N with most noticeable results for smaller N. This shows that previously activity map calculation could have been a bottleneck with high values of D. Further profiling and running experiments without activity map calculation resulted in similar run-times, highlighting that no more improvements for this calculation are required.

N	Baseline	+Compute	+Activity	SOA	Read Only	Shared	rSqrt
100	5.3	5.3	0.2	0.1	0.1	0.1	0.06
1000	6.7	6	0.6	0.7	0.6	0.2	0.1
2000	7.5	6.9	1.2	1.3	1	0.3	0.1
4000	9.6	7.8	2.2	2.5	2.1	0.5	0.3
8000	14	10.8	5.4	5.2	4.1	1	0.6
16000	29	20.4	13.9	10.8	8.2	2.8	1.7
32000	120.4	73.7	60.4	44.9	30.1	10.6	6.2
64000	366	215.1	190.2	137.2	91.7	35.3	22.5

TABLE I: Table of run-time in milliseconds per iteration for CUDA Optimizations

```
for (i = 0; i < D; i++) {
    y_min = i * d;
    y_max = (i + 1) * d;
    for (j = 0; j < D; j++) {
        x_min = j * d;
        x_max = (j + 1) * d;
        if (x_min <= b.x && x_max >= b.x && y_min <= b.y && y_max >= b.y) {
            idx = (i * D) + j;
            activity[idx] += val;
        }
    }
}
```

```
// Update activity map:
if (between_1(b_i->x, b_i->y)) {
    idx = (int)floorf(d_D * b_i->x) + (int)(d_D * floorf(d_D * b_i->y));
    atomicAdd(&d_a_buff[idx], val);
}
```

Fig. 3: Initial implementation of activity calculation (top) vs improved (bottom).

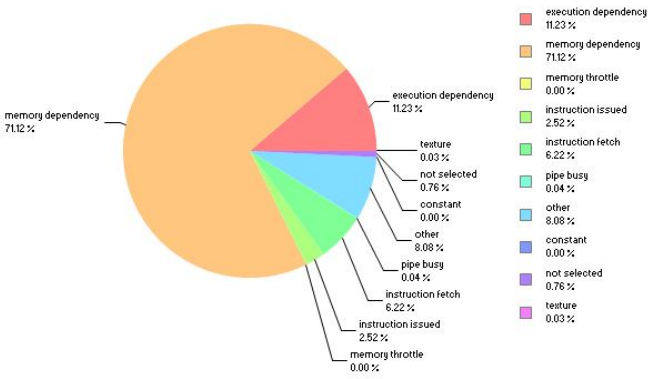

Fig. 4: Operations split for the improved baseline implementation. (Note: Orange is memory dependencies)

.

II. Memory Optimizations

Profiling the improved version of the baseline solution demonstrated that the core bottleneck is memory-related. This is demonstrated both in bottom of Fig.2 and Fig.4 where memory utilization is much higher than that of other operations. Analyzing memory statistics shows that there is a lot of data movement from and to global memory, causing significant inefficiency. Solutions to this problem are outlined below.

A. Structure of Arrays (SOA)

The data for bodies are being stored in an Array of Structures (AOS) each consisting of 5 floats totalling in 20 bytes per structure. Since individual memory transactions from L2 cache have 32bit size, only one structure can be transferred per transaction leaving much-unutilized bandwidth. Thus improved memory coalescing could result in higher performance. This is done by changing the data into Structure of Arrays where we only pass around a single structure with an array for each attribute of the body containing data about all of the N bodies. Effectiveness of this approach is demonstrated in TabI where performance improvement increases as we increase the number of bodies. Ruining the profiler shows that there has been some improvement in a memory dependency, but it is still a critical bottleneck.

B. Double Buffering and Read Only

The current implementation reads and writes data from the same structure, making a lot of data movements there and back. This can be solved by implementing double buffering where two structures store data for N bodies, one gets updated inside the kernel, and then the pointers between two structures are switched. Although double-buffering alone has not contributed much to performance improvement, it has allowed implementing more advance memory access approach using read-only storage.

Having two data structures allows to read data about bodies from one structure and update the other one. Declaring the read-only structure as **const* __restrict__** lets the compiler know that no data will be written to it and it can be stored and accessed in a more efficient manner. This change has resulted in a good increase in performance as demonstrated by results in Tab.I and almost eliminated the L2 cache memory dependency issues. Using texture cache has also been considered; however, the 1D approach taken to solving this problem as well as load size do not land naturally to using texture memory. Constant memory is also not applicable as body size is runtime defined, and memory is dynamically allocated.

C. Shared Memory

Finally, we can observe that a lot of the data is reused during the calculation as each body has to be loaded at least N times to compare against every other body. Shared memory could be used to load information about each of the bodies once and then reused across the block. This process is demonstrated in the code snippet in Fig. 5 where each block would load 256 (number of threads) bodies into shared memory and perform the force calculation, then this repeats until the force interaction has been calculated for all of the bodies. Some care had to be taken to ensure all bodies are loaded in, but no illegal memory access occurs, thus the presence of 2 **if** statements.

```
for (int s = 0; s < d_N; s += THREADS_PER_BLOCK, step++) {
    int s_idx = MOD(blockIdx.x + step, gridDim.x) * blockDim.x + threadIdx.x;
    if (s_idx < d_N) {
        // Only need to have shared access to poisition and mass:
        float4 b_s = { d_bodies_read->x[s_idx], d_bodies_read->y[s_idx], d_bodies_read->m[s_idx] };
        sharedBodies[threadIdx.x] = b_s;
        __syncthreads();
        if (within) {
            force = calc_force_d(pos, force);
        }
        __syncthreads();
    }
}
```

Fig. 5: Loading body data into shared memory

Using shared memory resulted in a more than 3-fold increase in the performance, highlighting the importance of practical memory usage on the GPU device. Profiling results for this approach are demonstrated in Fig. 7 where at the top, we can see that share memory usage is the dominant one and is coping well with the amount of data. The chart at the bottom of the figure demonstrates that there are almost no memory dependency issues anymore, and the major bottleneck is caused by other(unknown) and execution dependencies. Examining the execution profile (Fig.6 points to one line of code which could potentially be optimized.

```
// x and y vector difference in positions
diff.x = b_j.x - b_i.x;
diff.y = b_j.y - b_i.y;

// bottom part of the fraction within the summation
denom = (diff.x * diff.x) + (diff.y * diff.y) + SOFTENING2;
denom = sqrtf(denom * denom * denom);

force.x += b_j.z * diff.x * denom;
force.y += b_j.z * diff.y * denom;
```

Fig. 6: Line by line profiling of main force calculation

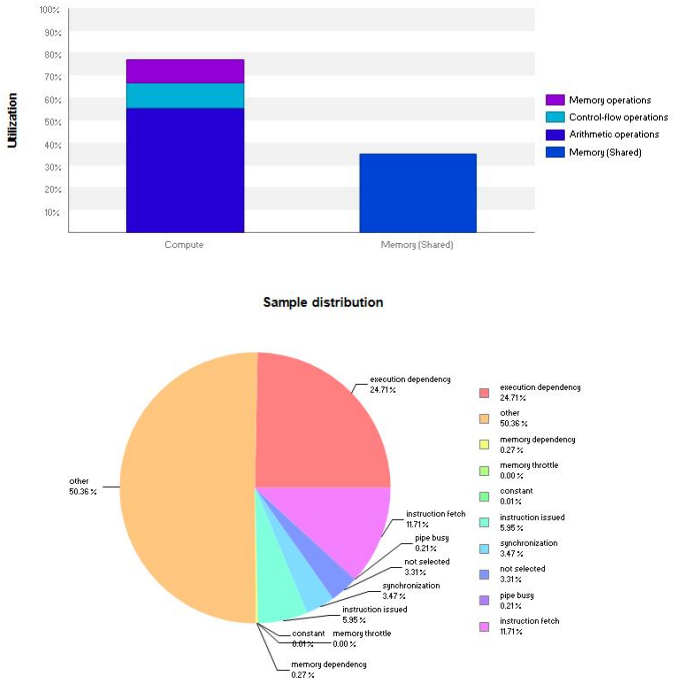

Fig. 7: Profile results for shared memory implementation. (Note: orange section denotes other usage)

III. Final Results

A. Inverse Square Root Approximation

Final optimization has tackled the problematic line of code from the end of the previous section. It has been found that it is possible to adequately approximate inverse square root calculation using the Newton Raphson Method with an accurate first estimation achieved via bit-wise operations (Fig.8). It is possible to increase the precision by setting the appropriate **PRECISION** constant to a larger value. It was found that even the first approximation is accurate and produced the desired visualization. Depending on the context in which the simulation is used, this parameter could be increased to accommodate for the accuracy required. Increasing the **PRECISION** even to 5 resulted only in a minimal reduction of performance. As demonstrated in Tab.I, the performance gained from using this approximation is significant. Profile results from Fig.9 indicate that no further reasonable performance improvements could be identified.

```
__device__ float fast_rsqrt_d(float x) {
    float xhalf = 0.5f * x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i >> 1);
    x = *(float*)&i;
    for (int j = 0; j < PRECISION; j++)
        x *= (1.5f - (xhalf * x * x));
    return x;
}
```

Fig. 8: Code for inverse square root approximation

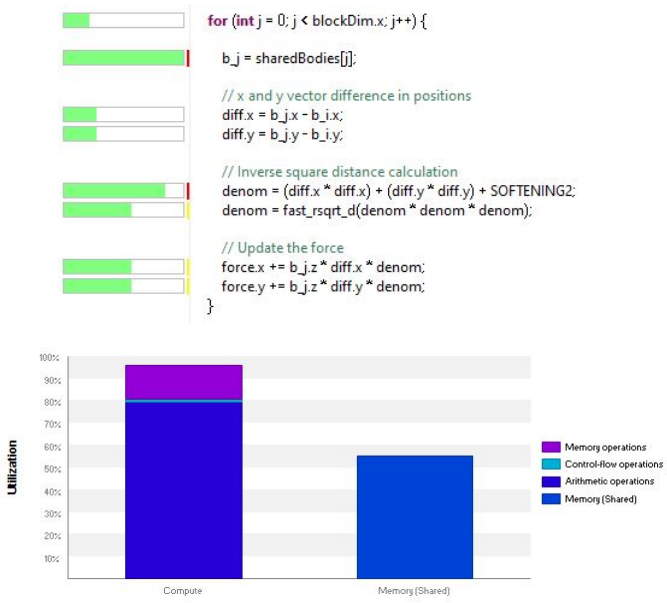

Fig. 9: Profile results for fast sqrt implementation

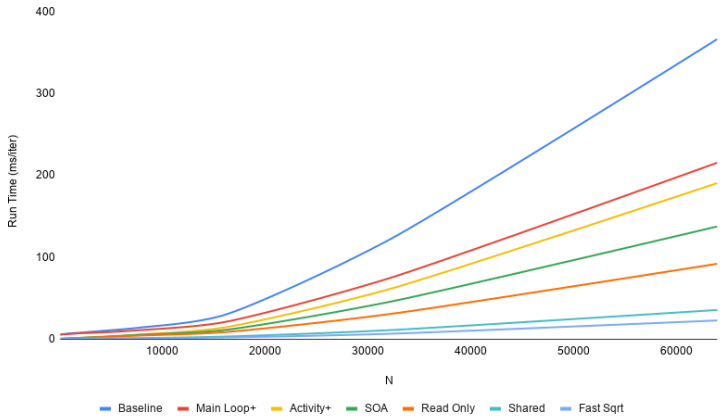

Fig. 10: Graphed results from Tab. I

B. CPU vs GPU

Final results from Tab.II demonstrate the incredible power of effective parallelization of a program on GPU, particularly as the problem grows, the difference becomes more apparent. The critical difference in three is how many processors are performing computations in parallel. With a serial CPU implementation, only a single processor is dealing with each body one at a time. With OPENMP, we can parallelize this process on CPU but are still limited to the small number of processing units (threads) on the CPU. GPU has much more threads and given the small complexity of each computation that has to be performed with the individual body, allowing to run all of the bodies in parallel.

N	CPU	OPENMP	CUDA
100	0.2	0.1	0.06
1000	17	5.6	0.1
2000	70	21	0.1
4000	280	75	0.3
8000	1.1K	290	0.6
16000	4.4K	1K	1.7
32000	17K	4.2K	6.2
64000	68K	19K	22.5

TABLE II: Table of run-time in milliseconds per iteration. Where K indicates in thousands.