Contents

1 Introduction

1.1 Development and Benchmark Environment

1. Operating system: Windows 10 64-bit
2. CPU: Intel® Core™ i7-7700HQ, 4 cores, 8 threads
3. RAM: 16GB DDR4-2400MHz
4. GPU: Geforce GTX 1060 (Mobile)

a.	CUDA compute capability	: 6.1
b.	(10) multiprocessors, (128) CUDA cores / MP	: 1280 CUDA cores
c.	Total amount of global memory	: 6144 MBytes
d.	Total amount of constant memory	: 65536 bytes
e.	Total amount of shared memory per block	: 49152 bytes
f.	Total number of registers available per block	: 65536
g.	Maximum number of threads per multiprocessor	: 2048
h.	Maximum number of threads per block	: 1024
i.	Max dimension size of a thread block (x, y, z)	: (1024, 1024, 64)

The GPU information is obtained through the `deviceQuery` sample provided by NVIDIA [8]. Some of the common properties are listed here as a reference for the development process.

1.2 Baseline Implementation (Naive Implementation)

The baseline implementation is a basic conversion of the CPU code to GPU code without including any further optimisation techniques. It is used as a baseline for measuring the effectiveness of different optimisations introduced during the iterative refinement.

The following implementation techniques are selected as the baseline implementation.

1. **Parallelisation Method:** Parallelise each body (N threads)
2. **Data Layout Structure:** Array of Structure
3. **Memory cache:** Global memory
4. **Threads per block**: 32 (`THREADS_PER_BLOCK`)
5. **Grid Dimension**: 1 dimension. Blocks per grid is rounded up if **N** or **D** is not a multiple of `THREADS_PER_BLOCK`.

1.2.1 Kernel 1: Compute Force

This kernel is responsible for calculating the summation of force for each n-body. Since the baseline implementation is using **N** threads, there will be a single `for` loop inside the kernel to calculate the summation part. Each thread in a block represents a single body.

```
__global__ void compute_force(...) {
    const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) {
        float local_sum_x = 0, local_sum_y = 0;

        // Summation of forces for the current n-body
        for (unsigned int j = 0; j < N; ++j) { ... }

        d_force_sum[i].x = local_sum_x;
        d_force_sum[i].y = local_sum_y;
    }
}
```

Listing 1: CUDA kernel for computing the summation of force for each n-body.

1.2.2 Kernel 2: Update Body

After the summation of force has been computed for each n-body, the result is then passed to the `update_body` kernel. The `update_body` will update the position and velocity of each body, then it will calculate the new cell of the body and increase the count. Race condition is prevented by using CUDA atomic function [4], as shown in line 11 of **Listing 2**.

```
1  __global__ void update_body(...) {
2      const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
3  
4      if (i < N) {
5          // Update x, y, vx, vy
6  
7          // Do not update `d_activity_map` if n-body is out of grid area
8          if (cell < grid_size) {
9              atomicAdd(&d_activity_map[cell], 1); // Prevent race condition
10         }
11     }
12 }
```

Listing 2: CUDA kernel for updating each body.

1.2.3 Kernel 3: Normalising Activity Map

The last step is to normalise the activity map. This kernel is relatively simpler. Each thread represents a cell in the activity map.

```
__global__ void normalise_activity_map(...) {
    const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // `grid_size` and `normalising_factor` are runtime calculated constants
    if (i < grid_size) {
        d_activity_map[i] *= normalising_factor;
    }
}
```

Listing 3: CUDA kernel for normalising the activity map.

1.3 Fixed Variables of Benchmark

1.3.1 Number of Simulation Iterations

Fixed arguments: **N** = 5000, **D** = 10

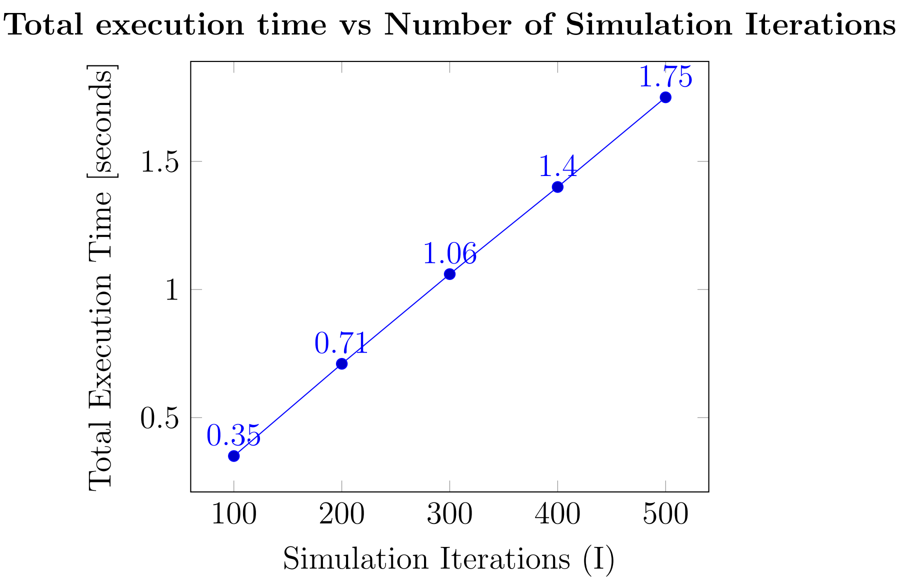


Figure 1: Relationship between the total execution time and number of simulation iterations in CUDA mode.

As shown in **Figure 1**, the total execution time is **directly proportional** to the number of simulation iterations. The value of **I** should only affect the total execution time proportionally, but not the simulation performance. Hence, **I** is excluded from the benchmark variable.

1.3.2 Number of Runs per Benchmark

Each benchmark will be run 10 times to produce an average execution time. This is to minimise the margin of error caused by uncontrollable environment variables.

1.4 Baseline Performance

Benchmark on Number of Bodies (N)

Fixed arguments: D = 10, **I** = 100

N	Average Execution Time of 10 runs (seconds)
10000	0.729
20000	2.863
30000	6.460
40000	11.494
50000	17.990

Table 1: Baseline performance of program over various values of **N**.

Benchmark on Activity Grid Dimension (D)

Fixed arguments: **N** = 1024, **I** = 100

D	Average Execution Time of 10 runs (seconds)
1000	0.087
2000	0.135
5000	0.468
10000	1.658
15000	3.620

Table 2: Baseline performance of program over various values of **D**.

1.5 Baseline Profiling on Visual Profiler

This section summarises the issues of baseline implementation reported by Visual Profiler. Detailed analysis results will be shown and discussed in other relevant sections. The profiling session is run with arguments: **N** = 10000, **D** = 10, **I** = 100.

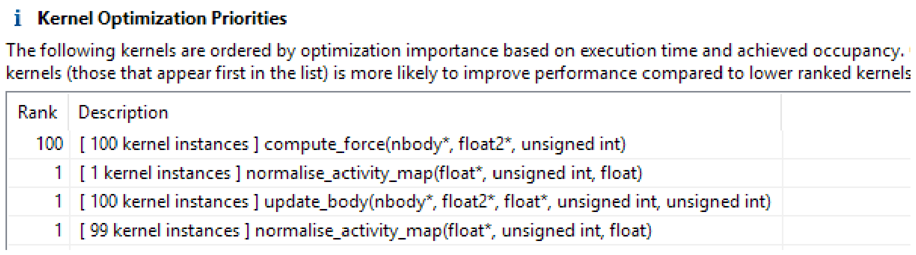
1.5.1 Analysis Results

1. Data movement and concurrency
   (a) Low Memcpy/Kernel Overlap
   (b) Low Kernel Concurrency
   (c) Low Memcpy Throughput
2. Compute Utilisation
   (a) Low Multiprocessor Occupancy (21.7% average)
3. Kernel Performance
   (a) Low Global Memory Load Efficiency (18.8% average)
   (b) Low Global Memory Store Efficiency (17.4% average)

For 1(a) and 1(c), the issue is unavoidable as the program only needs to do the `cudaMemcpy` once before the `step()` function to transfer the host n-bodies to device. For 1(b), the kernels `update_body` and `normalise_activity_map` have data dependencies on the previous kernel, so they cannot be executed in parallel.

For 2 and 3, they can be improved with the help of guided analysis. Therefore, the iterative improvement of the code will try to follow the guided analysis to identify the main bottleneck of the program and optimise it.

1.5.2 Kernel Optimisation Priorities



Kernel Optimization Priorities

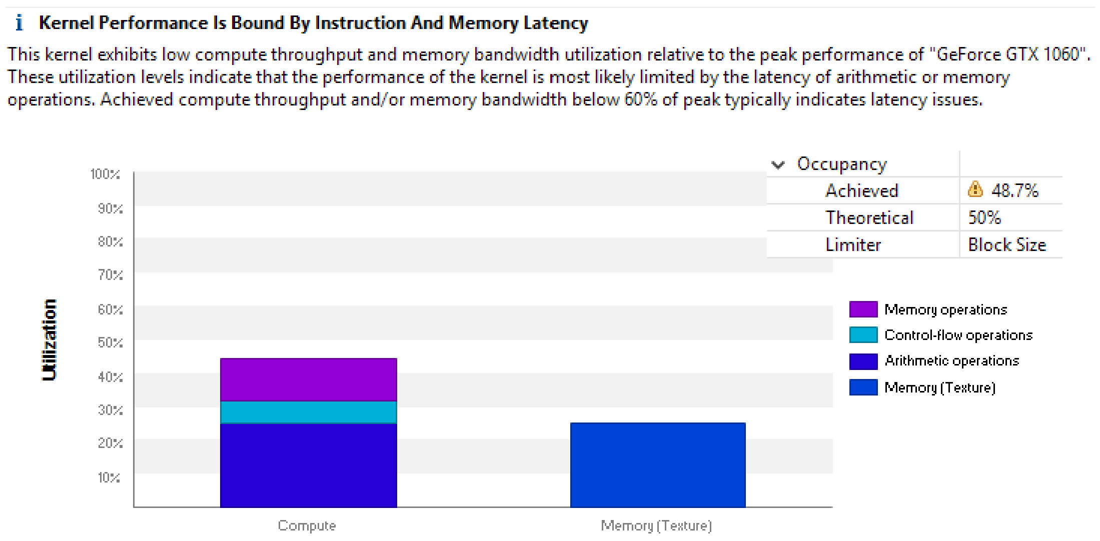
The following kernels are ordered by optimization importance based on execution time and achieved occupancy.
kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels

Rank	Description
100	[100 kernel instances] compute_force(nbody*, float2*, unsigned int)
1	[1 kernel instances] normalise_activity_map(float*, unsigned int, float)
1	[100 kernel instances] update_body(nbody*, float2*, float*, unsigned int, unsigned int)
1	[99 kernel instances] normalise_activity_map(float*, unsigned int, float)

Figure 2: Kernel optimisation priorities of baseline implementation.

As shown in **Figure 2**, the kernel `compute_force` has the top priority. Therefore, optimisations will be prioritised for the `compute_force` kernel.

1.5.3 Kernel Analysis: Compute Force



Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1060". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Occupancy	
Achieved	⚠ 48.7%
Theoretical	50%
Limiter	Block Size

Figure 3: Kernel analysis of `compute_force`.

As shown in **Figure 3**, the kernel `compute_force` has low compute and memory utilisation. The occupancy is also limited by the block size of 32.

1.6 Summary

Problems:

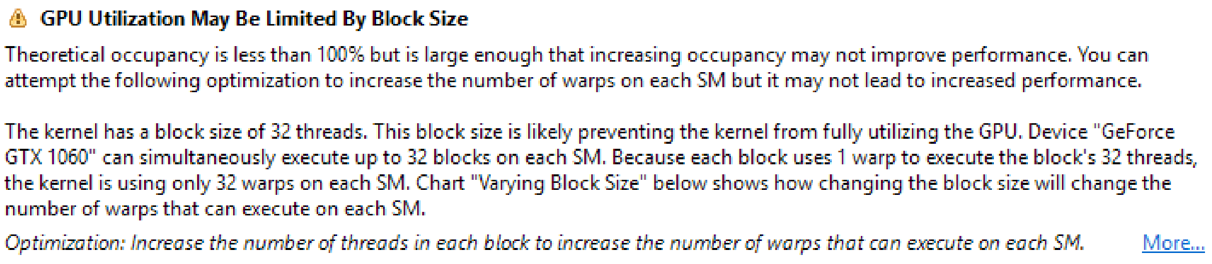
1. **Low compute utilisation:** Computation can be improved by reducing the instruction dependency and avoid heavy floating point instructions.
2. **Low memory utilisation:** Inefficient global memory load and store. It could be improved by using constant, read-only, and shared memory to reduce the global load / store instructions.
3. **Low occupancy and high latency:** This is likely to be caused by the small block size. Increasing the block sizes may improve the latency.

2 Improving Latency and Occupancy

According to **Figure 3**, Visual Profiler suggests that the kernel performance is currently limited by latency. Therefore, this optimisation is performed first.

2.1 Increase the Block Size

As shown in **Figure 4**, 32 threads per block is not enough to fully utilise the device. Moreover, the block size is also too small to hide the latency of memory instructions.



GPU Utilization May Be Limited By Block Size

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel has a block size of 32 threads. This block size is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX 1060" can simultaneously execute up to 32 blocks on each SM. Because each block uses 1 warp to execute the block's 32 threads, the kernel is using only 32 warps on each SM. Chart "Varying Block Size" below shows how changing the block size will change the number of warps that can execute on each SM.

Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM. More...

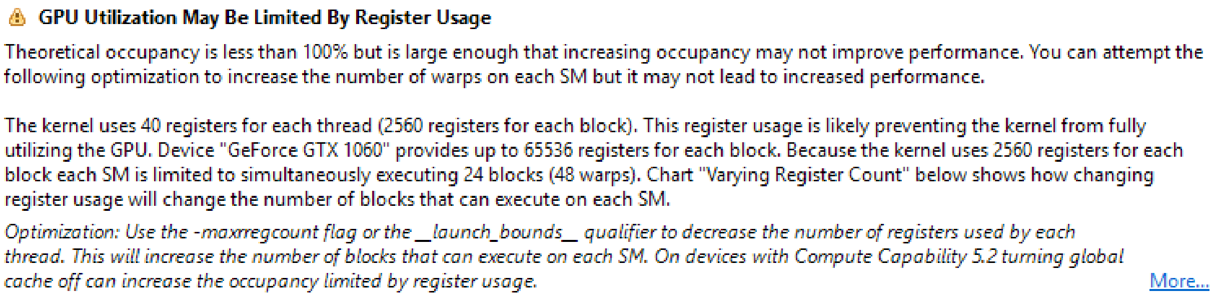
Figure 4: Visual Profiler suggesting a larger block size.

Value	Total Execution Time (seconds)				
	32 (baseline)	64	128	256	512
Number of Bodies (N)					
1024	0.087	0.081	0.072	0.072	0.073
2048	0.154	0.155	0.144	0.144	0.144
4096	0.401	0.297	0.287	0.287	0.287
10000	0.729	0.735	0.729	0.732	0.740
20000	2.863	1.863	1.849	1.862	1.873
30000	6.460	5.397	5.347	5.014	5.060
40000	11.494	7.402	7.341	7.377	7.447
50000	17.990	13.397	13.283	12.845	12.786
Average speedup	-	**1.25**	**1.29**	**1.31**	**1.30**
Activity Grid Dimension (D)					
1000	0.087	0.080	0.079	0.080	0.080
2000	0.135	0.108	0.103	0.104	0.104
5000	0.468	0.299	0.267	0.268	0.269
10000	1.658	0.983	0.855	0.854	0.861
15000	3.620	2.124	1.835	1.832	1.847
Average speedup	-	**1.46**	**1.62**	**1.61**	**1.60**

Table 3: Performance comparison of different number of threads per block.

According to the results in **Table 3** and taking the margin of error into account, 128, 256, and 512 threads per block give the best overall performance improvements. At this stage, it is decided to set the threads per block to 256. After further optimisations have been made, the threads per block might need to be changed accordingly.

2.2 Summary


GPU Utilization May Be Limited By Register Usage

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel uses 40 registers for each thread (2560 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX 1060" provides up to 65536 registers for each block. Because the kernel uses 2560 registers for each block each SM is limited to simultaneously executing 24 blocks (48 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited by register usage.

More...

Figure 5: Kernel analysis of `compute_force` after increasing block size to 256.

After increasing the threads per block to 256, both compute and memory utilisation have increased, as shown in **Figure 5**. Furthermore, the occupancy is not a limiting factor of the performance anymore.

Although the Visual Profiler still shows that the kernel performance is bound by instruction and memory latency, it is not suggesting any warnings or hints in the latency analysis. Hence, it is better to optimise the computation and memory bandwidth next.

3 Improving Computation

3.1 Use CUDA Built-In Vector Types [6]

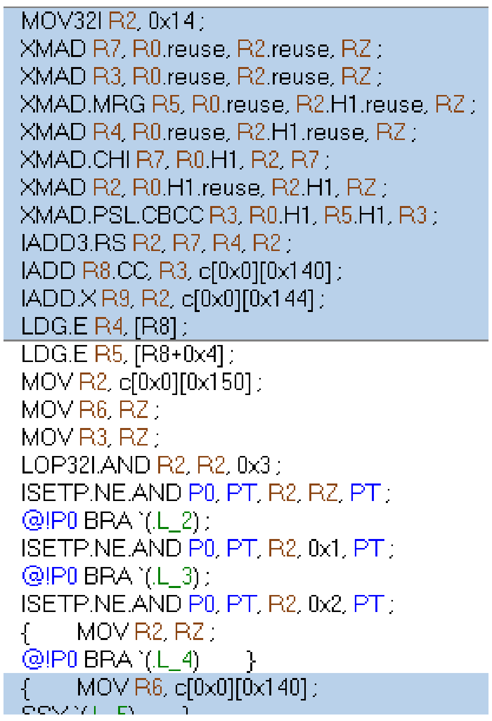
```
// Before
float local_sum_x = 0, local_sum_y = 0;
const float dist_x = d_nbodies[j].x - d_nbodies[i].x;
const float dist_y = d_nbodies[j].y - d_nbodies[i].y;

// After
float2 local_sum = make_float2(0, 0);
const float2 dist = make_float2(d_nbodies[j].x - d_nbodies[i].x,
                                d_nbodies[j].y - d_nbodies[i].y);
```

Listing 4: Converting vector variables into CUDA built-in vector types.

Value	Total Execution Time (seconds)		Speedup
	Before	After	
Number of Bodies (N)			
10000	0.734	0.732	1.0003
20000	1.863	1.862	1.0005
30000	5.013	5.009	1.0008
40000	7.385	7.379	1.0008
50000	12.846	12.840	1.0005
		Average	**1.0011**
Activity Grid Dimension (D)			
1000	0.080	0.080	1.00
2000	0.103	0.103	1.00
5000	0.267	0.267	1.00
10000	0.854	0.854	1.00
15000	1.831	1.831	1.00
		Average	**1.00**

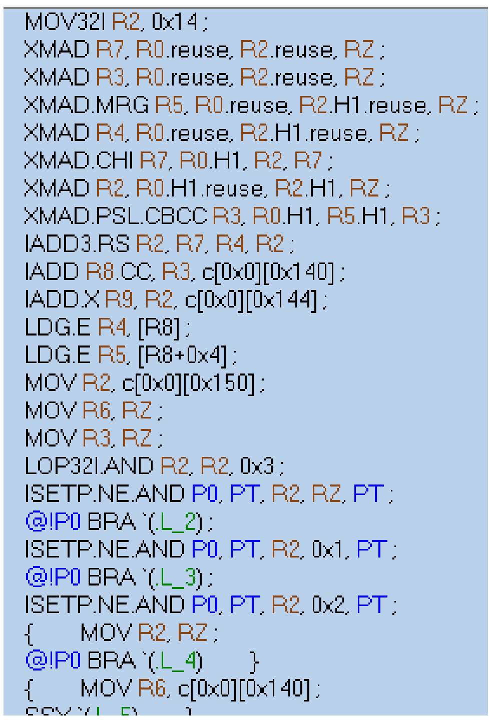
Table 4: Before and after using the CUDA built-in vector type.

- A minuscule amount of performance speedup is recorded. These speedups are negligible after considering the margin of error.
- The instruction execution section in the Visual Profiler also shows that the compiler has generated the same set of assembly instructions for both version of code (**Figure 6**).
- Therefore, there should not be any major performance difference between the two versions of code.



```
MOV32I R2, 0x14 ;
XMAD R7, R0.reuse, R2.reuse, RZ ;
XMAD R3, R0.reuse, R2.reuse, RZ ;
XMAD.MRG R5, R0.reuse, R2.H1.reuse, RZ ;
XMAD R4, R0.reuse, R2.H1.reuse, RZ ;
XMAD.CHI R7, R0.H1, R2, R7 ;
XMAD R2, R0.H1.reuse, R2.H1, RZ ;
XMAD.PSL.CBCC R3, R0.H1, R5.H1, R3 ;
IADD3.RS R2, R7, R4, R2 ;
IADD R8.CC, R3, c[0x0][0x140] ;
IADD.X R9, R2, c[0x0][0x144] ;
LDG.E R4, [R8] ;
LDG.E R5, [R8+0x4] ;
MOV R2, c[0x0][0x150] ;
MOV R6, RZ ;
MOV R3, RZ ;
LOP32I.AND R2, R2, 0x3 ;
ISETP.NE.AND P0, PT, R2, RZ, PT ;
@!P0 BRA `(.L_2) ;
ISETP.NE.AND P0, PT, R2, 0x1, PT ;
@!P0 BRA `(.L_3) ;
ISETP.NE.AND P0, PT, R2, 0x2, PT ;
{       MOV R2, RZ ;
@!P0 BRA `(.L_4)        }
{       MOV R6, c[0x0][0x140] ;
```

(a) Instruction for `float dist_x`



```
MOV32I R2, 0x14 ;
XMAD R7, R0.reuse, R2.reuse, RZ ;
XMAD R3, R0.reuse, R2.reuse, RZ ;
XMAD.MRG R5, R0.reuse, R2.H1.reuse, RZ ;
XMAD R4, R0.reuse, R2.H1.reuse, RZ ;
XMAD.CHI R7, R0.H1, R2, R7 ;
XMAD R2, R0.H1.reuse, R2.H1, RZ ;
XMAD.PSL.CBCC R3, R0.H1, R5.H1, R3 ;
IADD3.RS R2, R7, R4, R2 ;
IADD R8.CC, R3, c[0x0][0x140] ;
IADD.X R9, R2, c[0x0][0x144] ;
LDG.E R4, [R8] ;
LDG.E R5, [R8+0x4] ;
MOV R2, c[0x0][0x150] ;
MOV R6, RZ ;
MOV R3, RZ ;
LOP32I.AND R2, R2, 0x3 ;
ISETP.NE.AND P0, PT, R2, RZ, PT ;
@!P0 BRA `(.L_2) ;
ISETP.NE.AND P0, PT, R2, 0x1, PT ;
@!P0 BRA `(.L_3) ;
ISETP.NE.AND P0, PT, R2, 0x2, PT ;
{       MOV R2, RZ ;
@!P0 BRA `(.L_4)        }
{       MOV R6, c[0x0][0x140] ;
```

(b) Instruction for `float2 dist`

Figure 6: Instructions generated for **(a)** C++ primitive and **(b)** CUDA vector type.

3.2 Use CUDA Math API

The CUDA Best Practices Guide suggested that

1. The reciprocal square root should always be invoked explicitly as `rsqrtf()` for single precision [1].
2. Avoid using heavy-weight `pow()` and `powf()` functions. Use explicit multiplication for small integer powers [2].

```
1  // Before
2  float mag_add_soft = dist.x * dist.x + dist.y * dist.y + SOFTENING_SQUARE;
3  float m_div_mag = d_nbodies[j].m / (mag_add_soft * sqrtf(mag_add_soft));
4
5  // After (Version 1)
6  float mag_add_soft = dist.x * dist.x + dist.y * dist.y + SOFTENING_SQUARE;
7  float m_div_mag = d_nbodies[j].m * rsqrtf(mag_add_soft * mag_add_soft * mag_add_soft);
8
9  // After (Version 2)
10 float inv_dist = rsqrtf(dist.x * dist.x + dist.y * dist.y + SOFTENING_SQUARE);
11 float m_div_mag = d_nbodies[j].m * inv_dist * inv_dist * inv_dist;
```

Listing 5: Converting from `sqrtf()` to `rsqrtf()`.

In Assignment 1, huge performance increase was achieved by removing the usage of `powf()` function, as shown by line 2 in **Listing 5** and the formula simplification below:

$$(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^{\frac{3}{2}} \equiv (||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)\sqrt{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)}$$

CUDA Math API provides the function `rsqrtf(float x)` that is able to calculate $\frac{1}{x}$ directly in the device. With this optimisation applied, the division operation can be replaced by a multiplication operation, as demonstrated in **Listing 5**.

'Version 1' represents the following equation:

$$(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^{\frac{3}{2}} \equiv \sqrt{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^3}$$

'Version 2' represents the following equation:

$$(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)^{\frac{3}{2}} \equiv \left[\sqrt{(||\vec{x_j} - \vec{x_i}||^2 + \epsilon^2)}\right]^3$$

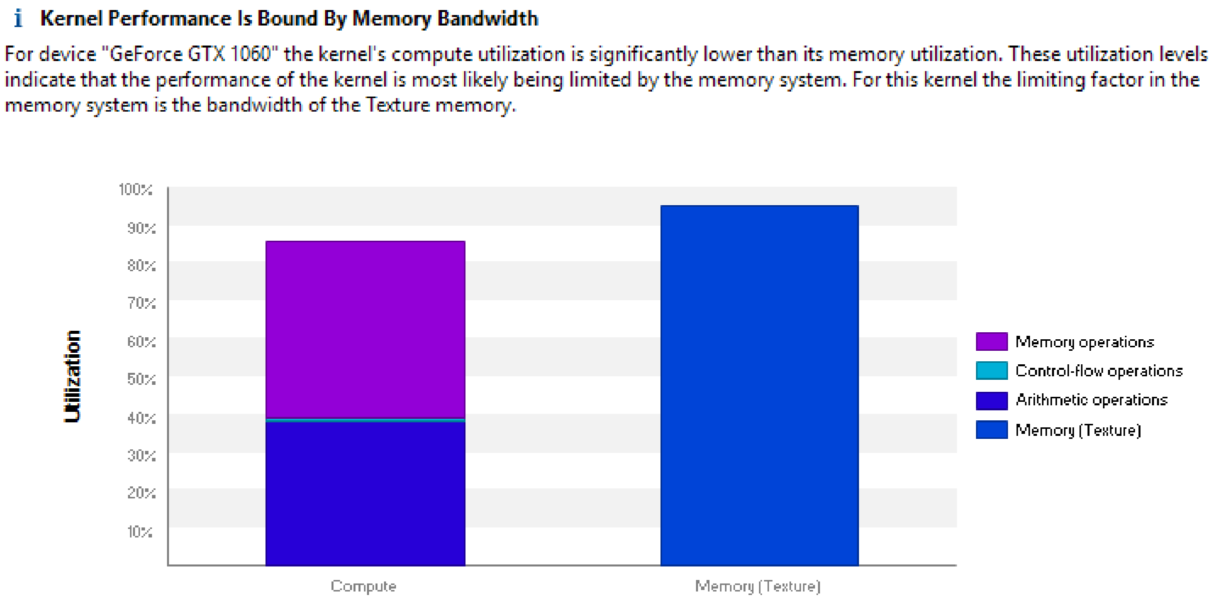
Value	Total Execution Time (seconds)		
	Before	Version 1	Version 2
Number of Bodies (N)			
1024	0.078	0.013	0.011
2048	0.144	0.024	0.023
4096	0.287	0.057	0.052
10000	0.726	0.218	0.206
20000	1.859	0.825	0.769
30000	5.028	1.839	1.714
40000	7.376	3.246	3.008
50000	12.839	5.161	4.765
Average speedup	-	**3.76**	**4.11**
Activity Grid Dimension (D)			
1000	0.080	0.019	0.018
2000	0.103	0.043	0.042
5000	0.267	0.207	0.206
10000	0.854	0.793	0.793
15000	1.831	1.770	1.770
Average speedup	-	**2.00**	**2.06**

Table 5: Performance comparison of different number of threads per block.

As shown in **Table 5**, great performance speedup is achieved by using the CUDA Math API. Although both 'Version 1' and 'Version 2' represent a different variant of the same equation, 'Version 2' provides a higher performance speedup than 'Version 1'.

However, one disadvantage is that `rsqrtf()` has a maximum ULP (Unit of Least Precision) error of 2 [9]. This error is acceptable as the actual difference would be small, and it has also provided a significant amount of performance improvements.

3.3 Summary


Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GTX 1060" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Texture memory.

Figure 7: Kernel analysis of `compute_force` after `rsqrtf()`.

1. Comparing **Figure 7** with **Figure 3**, both compute and memory utilisation have increased significantly after using CUDA's `rsqrtf()` to calculate the reciprocal square root.
2. The kernel performance of `compute_force` is now limited by the memory bandwidth.

4 Improving Memory Bandwidth

4.1 Store Runtime Constants into Constant Memory

The runtime constants are currently passed to the kernel function as arguments.

```
// Before
__global__ void update_body(const unsigned int N,
                            const unsigned int D,
                            const unsigned int grid_size, ...)
...
__global__ void normalise_activity_map(const unsigned int grid_size,
                                       const float normalising_factor, ...)
// After
__constant__ unsigned int c_N;
__constant__ unsigned int c_D;
__constant__ unsigned int c_grid_size;
__constant__ float c_normalising_factor;
...
__global__ void parallelise_each_body(nbody *d_nbodies, float *d_activity_map)
__global__ void normalise_activity_map(float *d_activity_map)
```

Listing 6: Store runtime constants into constant memory.

Value	Total Execution Time (seconds)		Speedup
	Before	After	
Number of Bodies (N)			
1024	0.011	0.012	0.92
2048	0.022	0.024	0.92
4096	0.052	0.051	1.02
10000	0.205	0.201	1.02
20000	0.768	0.764	1.01
30000	1.718	1.693	1.01
40000	3.006	2.996	1.00
50000	4.763	4.712	1.01
		Average	**0.99**
Activity Grid Dimension (D)			
1000	0.018	0.019	0.95
2000	0.042	0.043	0.98
5000	0.206	0.207	1.00
10000	0.793	0.793	1.00
15000	1.770	1.771	1.00
		Average	**0.98**

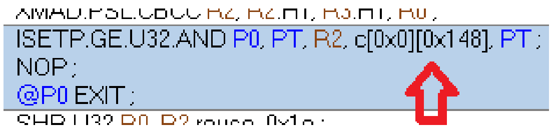
Table 6: Performance speedup of loading runtime constants from constant memory.

The results in **Table 6** have shown that loading the constants from constant memory is slightly slower than passing them as arguments for small number of **N** ($\lessapprox 4096$). When **N** is $\gtrapprox 10000$, there is a noticeable reduction in the total execution time.

According to the CUDA Programming Guide, `__global__` function parameters are passed to the device via constant memory [10]. However, as demonstrated by **Listing 7** and **Figure 8**, the variable is stored in a different constant memory location when it is used differently.

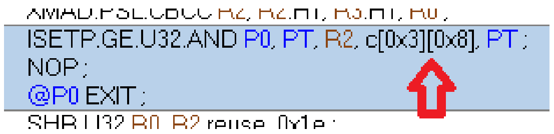
```
__constant__ float c_grid_size;

__global__ normalise_activity_map(const float grid_size) {
    const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < grid_size)   // Figure 8(a)
    if (i < c_grid_size) // Figure 8(b)
}
```

Listing 7: Implementation comparison for accessing the runtime constant.



```
XMAD.PSL.CBCC R2, R2.H1, R3.H1, R0 ;
ISETP.GE.U32.AND P0, PT, R2, c[0x0][0x148], PT ;
NOP ;
@P0 EXIT ;
SHR.U32 R0, R2.reuse, 0x1e ;
```

(a) `__global__` function parameter.

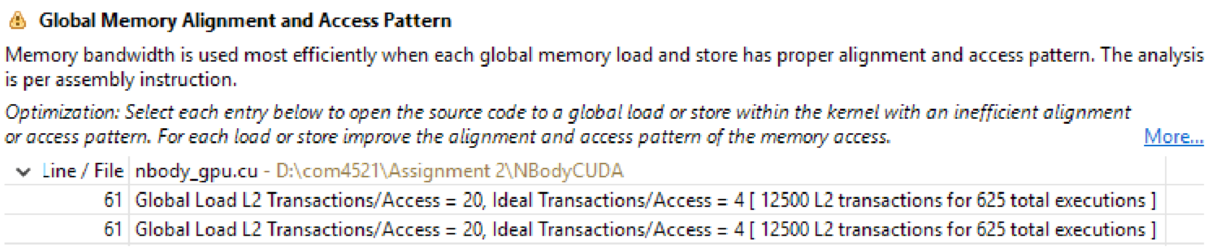


```
XMAD.PSL.CBCC R2, R2.H1, R3.H1, R0 ;
ISETP.GE.U32.AND P0, PT, R2, c[0x3][0x8], PT ;
NOP ;
@P0 EXIT ;
SHR.U32 R0, R2.reuse, 0x1e ;
```

(b) `__constant__` qualifier variable.

Figure 8: Constant memory location of different implementation.

4.2 Global Memory Alignment and Access Pattern



Global Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern. The analysis is per assembly instruction.

Optimization: Select each entry below to open the source code to a global load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access. More...

Line / File	nbody_gpu.cu - D:\com4521\Assignment 2\NBodyCUDA
61	Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [12500 L2 transactions for 625 total executions]
61	Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [12500 L2 transactions for 625 total executions]

Figure 9: Ineffective global memory access in `compute_force`.

```
61 float2 dist = make_float2(d_nbodies[j].x - d_nbodies[i].x, d_nbodies[j].y - d_nbodies[i].y);
```

Listing 8: Strided access of 5 for `d_nbodies` members, 20% bandwidth only.

As shown in **Figure 9**, the kernel `compute_force` is performing 5 times more transactions per access than ideal. This issue can be solved by changing the layout of `d_nbodies` from AoS to SoA. Three possible implementations are considered and documented in **Section 4.2.1**, **Section 4.2.2**, and **Section 4.2.3**.

4.2.1 Passing SoA `struct` by Value (Version 1)

```
nbody_soa d_nbodies;
...
/* Memory cost = the size of a nbody_soa structure (40 bytes) */
compute_force<<<blocks, threads>>>(d_nbodies, ...);
...
__global__ void compute_force(nbody_soa d_nbodies, ...)
```

Listing 9: Passing `struct` `nbody_soa` to `compute_force` by value.

4.2.2 Passing SoA `struct` Members by Reference (Version 2)

```
nbody_soa d_nbodies;
...
/* Memory cost = 3 pointers (3 * 8 bytes = 24 bytes) */
compute_force<<<blocks, threads>>>(d_nbodies.x, d_nbodies.y,  d_nbodies.m, ...);
...
__global__ void compute_force(float *x, float *y, float *m, ...)
```

Listing 10: Passing the `x`, `y`, and `m` to `compute_force` by reference.

4.2.3 Passing SoA `struct` by Reference (Version 3)

```
nbody_soa h_nbodies, *d_nbodies;
float *d_x, *d_y, *d_vx, *d_vy, *d_m; // Individual pointers to device memory

// Allocate device memory
checkCudaError(cudaMalloc((void **)(&d_nbodies), sizeof(nbody_soa)));
checkCudaError(cudaMalloc((void **)(&d_x), nbodies_soa_size));
...
// Copy host data to individual pointers first
checkCudaError(cudaMemcpy(d_x, h_nbodies.x, nbodies_soa_size, cudaMemcpyHostToDevice));
...
// Copy individual pointers to SoA pointer
checkCudaError(cudaMemcpy(d_nbodies, &h_nbodies, sizeof(nbody_soa), cudaMemcpyHostToDevice));
checkCudaError(cudaMemcpy(&d_nbodies->x, &d_x, sizeof(float *), cudaMemcpyHostToDevice));
...
/* Memory cost = 1 pointer (8 bytes) */
compute_force<<<blocks, threads>>>(d_nbodies, ...);
...
__global__ void compute_force(nbody_soa *d_nbodies, ...)
```

Listing 11: Passing `struct` `nbody_soa` by reference.

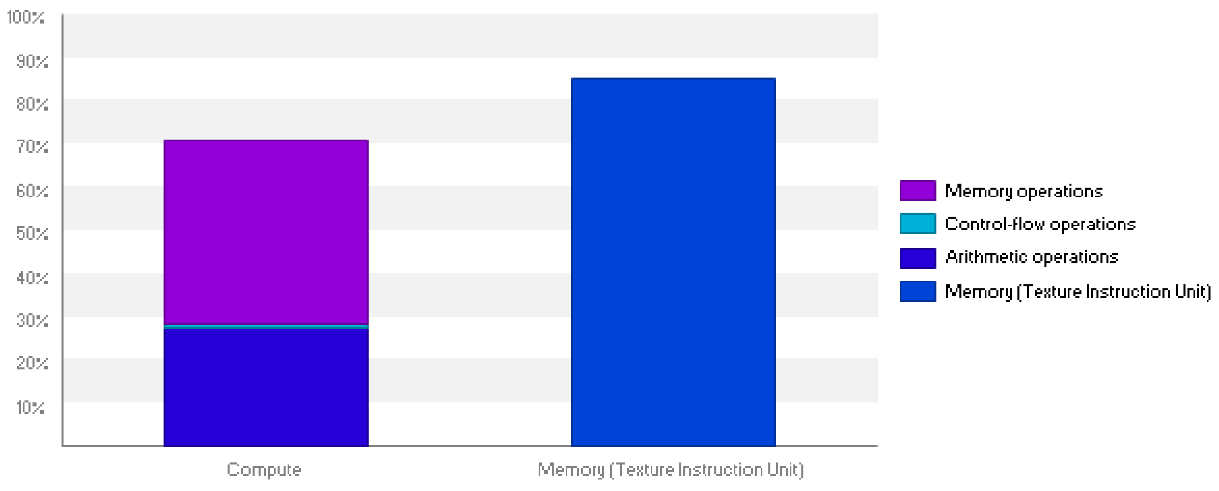
As shown in **Listing 11**, Version 3 has a more complex implementation than Version 1 and 2. This is because calling `cudaMalloc()` directly on `d_nbodies->x` will raise an 'Access violation writing location' at runtime. Therefore, it is required to allocate device memory to those individual device pointers first, then copy them to the SoA pointer members.

4.2.4 Results and Summary

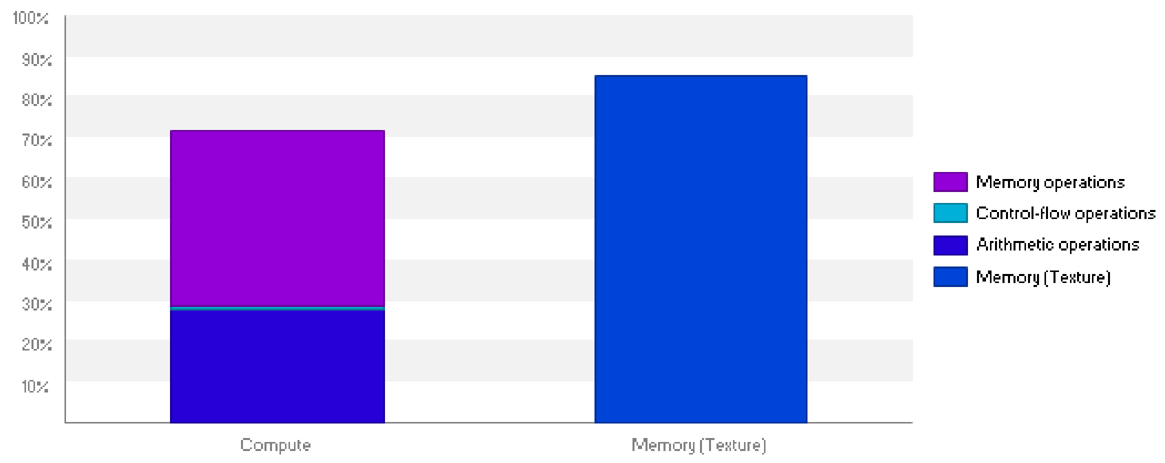
After changing the data layout of `d_nbodies` to Structure of Arrays, Visual Profiler no longer shows the access pattern warning. However, **Table 7** shows that the performance of SoA is slightly slower than AoS.

Value	Total Execution Time (seconds)			
	Before (AoS)	Version 1	Version 2	Version 3
Number of Bodies (N)				
1024	0.012	0.011	0.011	0.012
2048	0.024	0.021	0.021	0.022
4096	0.051	0.052	0.052	0.052
10000	0.202	0.227	0.227	0.233
20000	0.763	0.824	0.824	0.783
30000	1.695	2.030	2.028	1.857
40000	2.996	3.076	3.075	3.045
50000	4.707	4.754	4.751	4.869
Average speedup	**-**	**0.98**	**0.98**	**0.97**
Activity Grid Dimension (D)				
1000	0.019	0.020	0.020	0.020
2000	0.043	0.043	0.043	0.044
5000	0.207	0.207	0.207	0.208
10000	0.793	0.794	0.794	0.794
15000	1.770	1.772	1.771	1.771
Average speedup	**-**	**0.99**	**0.99**	**0.98**

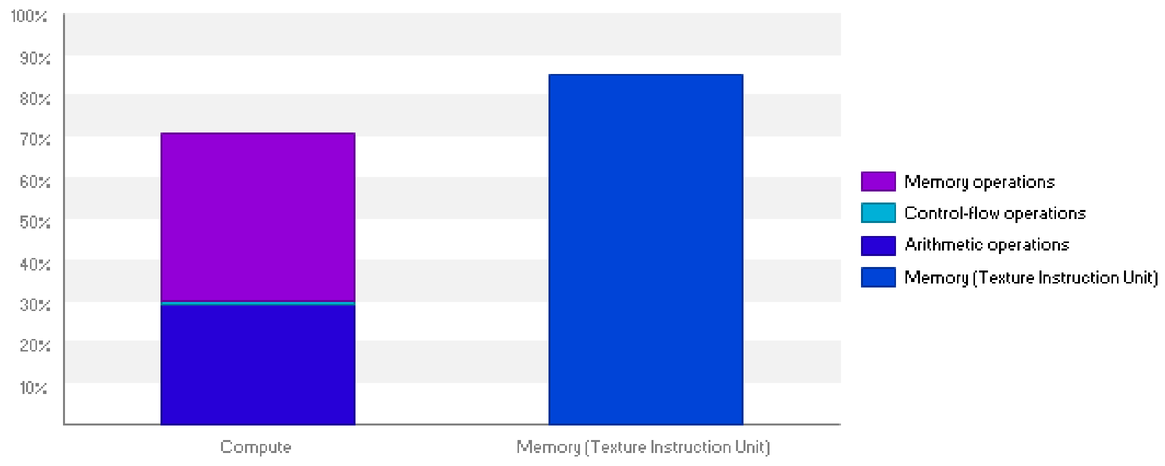
Table 7: Performance comparison of different SoA implementations.



(a) SoA Version 1, texture memory



(b) SoA Version 2, texture memory



(c) SoA Version 3, texture memory

Figure 10: Kernel analysis of `compute_force` on 3 different SoA implementations.

As illustrated by **Figure 10**, both compute and memory utilisation have decreased after changing from AoS to SoA (in comparison with **Figure 7**). This explains why the three versions of the SoA implementation has longer execution time than AoS.

In terms of compute utilisation, SoA Version 2 is the highest among the three versions ($\approx 71\%$). All three implementations are limited by the bandwidth of texture memory.

4.3 Read-Only Memory for SoA

Since SoA Version 2 (**Section 4.2.2**) and Version 3 **Section 4.2.3** are passing the pointers, they can be further optimised by using read-only memory.

```
// SoA Version 2
__global__ void compute_force(float const *__restrict__  x,
                              float const *__restrict__  y,
                              float const *__restrict__  m, ...)
// SoA Version 3
__global__ void compute_force(nbody_soa const *__restrict__  d_nbodies, ...)
```

Listing 12: Adding `__restrict__` specifier to SoA pointers.

4.3.1 Results and Summary

Value	Total Execution Time (seconds)				
	AoS (baseline)	V2 (tex.)	V2 (read)	V3 (tex.)	V3 (read)
Number of Bodies (N)					
1024	0.012	0.011	0.012	0.012	0.012
2048	0.024	0.021	0.017	0.022	0.022
4096	0.051	0.052	0.044	0.052	0.052
10000	0.202	0.227	0.187	0.233	0.229
20000	0.763	0.824	0.744	0.783	0.774
30000	1.695	2.028	1.667	1.857	1.835
40000	2.996	3.075	2.970	3.045	3.009
50000	4.707	4.751	4.655	4.869	4.826
Average speedup	-	**0.98**	**1.09**	**0.97**	**0.98**
Activity Grid Dimension (D)					
1000	0.019	0.020	0.018	0.020	0.020
2000	0.043	0.043	0.042	0.044	0.044
5000	0.207	0.207	0.206	0.208	0.208
10000	0.793	0.794	0.792	0.794	0.794
15000	1.770	1.771	1.769	1.771	1.771
Average speedup	-	**0.99**	**1.02**	**0.98**	**0.98**

Table 8: Performance comparison between SoA texture memory and read-only memory.

As shown in **Table 8**, read-only memory is faster than texture memory. In particular, SoA Version 2 with read-only memory provides a minor performance improvement. Moreover, both compute and memory utlisation of SoA Version 2 has increased when compared to texture memory (**Figure 10b**).

Conclusion: SoA Version 2 is selected as it has the best performance up to this point.

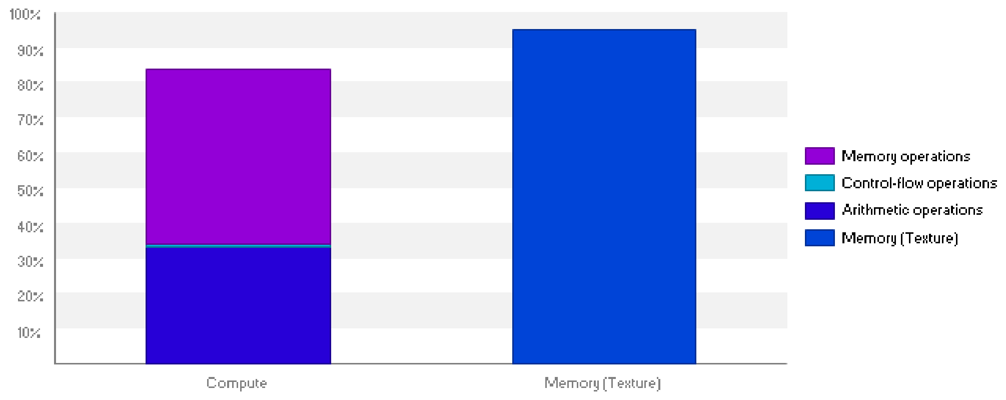


Figure 11: Kernel analysis of `compute_force` with SoA Version 2, read-only memory.

4.3.2 Optimise for Pointer Aliasing

According to a CUDA blog post [3] and the CUDA Programming Guide for [5], it is suggested to use the `__restrict__` keyword on pointers if it is known that the pointers will not be used to access overlapping regions. This tells the compiler that these pointers are not aliased, and the compiler can now perform some low-level optimisations.

Listing 13 is considered an optimisation ahead. No observable improvements are recorded as both `update_body` and `normalise_activity_map` have the lowest optimisation priority (**Figure 2**).

```
/* Kernel: update_body */
// Before
__global__ void update_body(nbody_soa d_nbodies)

// After
__global__ void update_body(float *__restrict__ x,
                            float *__restrict__ y,
                            float *__restrict__ vx,
                            float *__restrict__ vy, ...)

/* Kernel: normalise_activity_map */
// Before
__global__ void normalise_activity_map(float *d_activity_map) {

// After
__global__ void normalise_activity_map(float *__restrict__ d_activity_map) {
```

Listing 13: Adding `__restrict__` specifier to SoA pointers.

4.4 Shared Memory

4.4.1 Problem: Conditional Synchronisation, N-Bodies Out of Sync

```
1  __global__ void compute_force(...) {
2      const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
3      __shared__ float3 s_nbodies[THREADS_PER_BLOCK];
4  
5      // Conditional __syncthreads() (bad)
6      if (i < N) {
7          for (unsigned int sub_block = 0; sub_block < gridDim.x; ++sub_block) {
8              s_nbodies[threadIdx.x] = ...
9              __syncthreads();
10 
11             for (unsigned int j = 0; j < THREADS_PER_BLOCK; ++j) { ... }
12             __syncthreads();
13         }
14         ...
15     }
16 }
```

Listing 14: `if` statement (line 6) preventing `__syncthreads()` from functioning correctly.

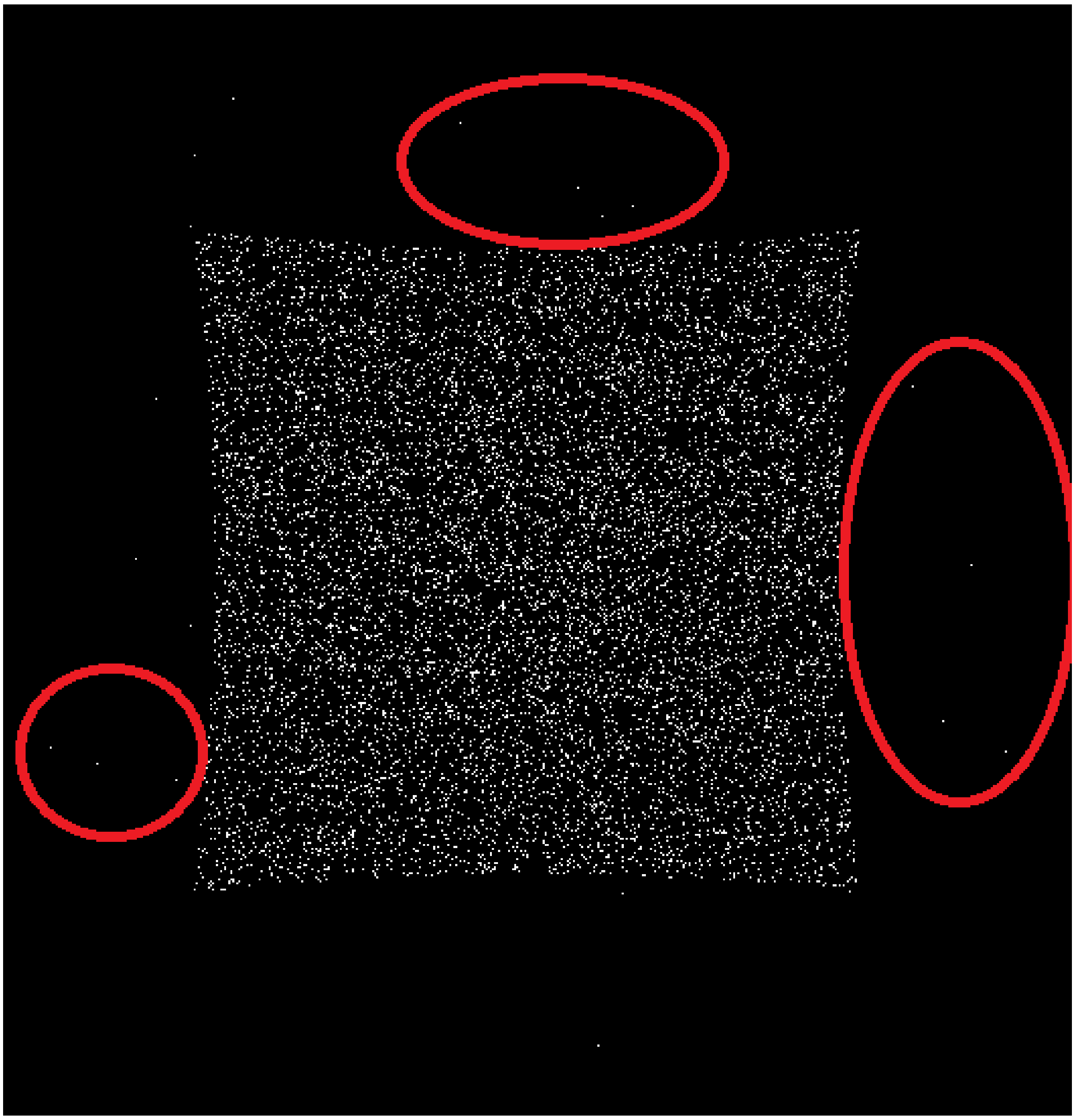


Figure 12: Some n-bodies (red circles) are out of sync.

As demonstrated in **Figure 12**, some n-bodies converge slower than the group of n-bodies. This problem does not exist when the number of bodies (**N**) is a multiple of `THREADS_PER_BLOCK`. When **N** is not a multiple of `THREADS_PER_BLOCK`, then there will be `N % THREADS_PER_BLOCK` n-bodies out of sync. For example:

- **N** = 10240, 256 threads per block: No n-bodies out of sync.
- **N** = 10270, 256 threads per block: 30 n-bodies out of sync.

Cause: According to the CUDA Programming Guide [7], `__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

4.4.2 Solution

```
1  __global__ void compute_force(float const *__restrict__ x, ...) {
2      __shared__ float3 s_nbodies[THREADS_PER_BLOCK];
3      const unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
4  
5      for (unsigned int sub_block = 0; sub_block < gridDim.x; ++sub_block) {
6          // Inline conditional below to prevent reading beyond allocated
7          s_nbodies[threadIdx.x] = sub_i < c_N
8                                   ? make_float3(x[sub_i], ...)
9                                   : make_float3(0, 0, 0);
10         __syncthreads();
11 
12         // Do not compute force for non-existent n-body
13         if (i < c_N) {
14             for (unsigned int j = 0; j < THREADS_PER_BLOCK; ++j) { ... }
15         }
16 
17         __syncthreads();
18     }
19 }
```

Listing 15: Solution for the problem in **Figure 12**.

The problem can be solved by moving the conditional inside the loop, as shown in **Listing 15**. By doing this, all threads are guaranteed to reach the `__syncthreads()`. However, this will introduced an additional conditional branch inside the loop to prevent reading beyond allocated (line 7).

4.4.3 Results

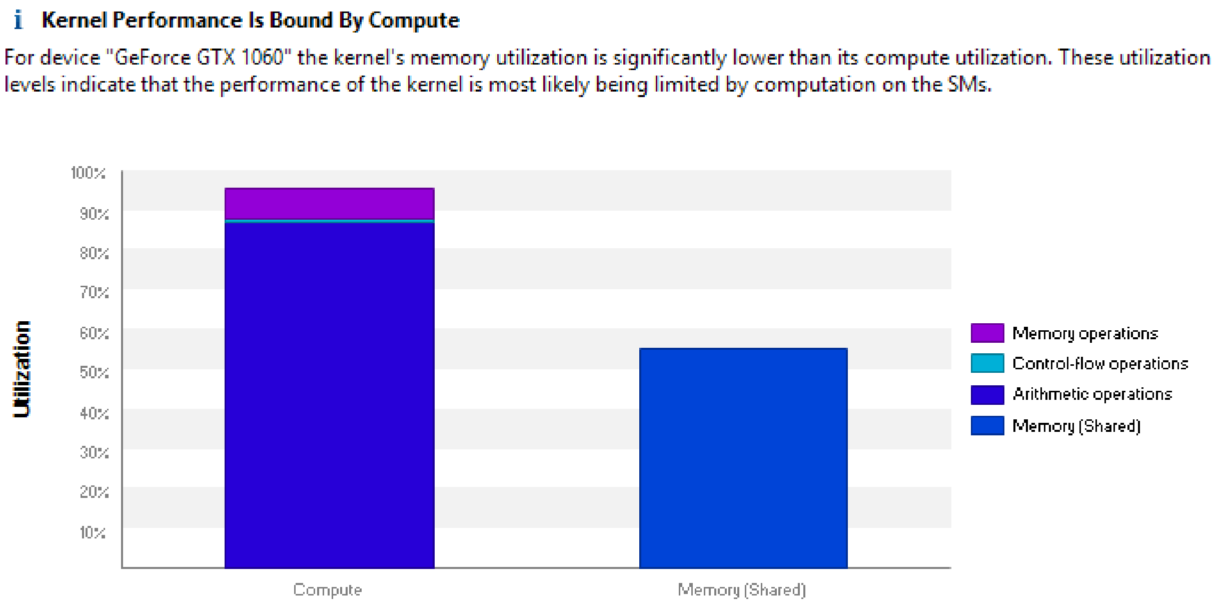
Value	Total Execution Time (seconds)		
	Before (**Section 4.3.2** version)	After	Speedup
Number of Bodies (N)			
1024	0.011	0.005	2.20
2048	0.018	0.009	2.00
4096	0.044	0.022	2.00
10000	0.187	0.093	2.01
20000	0.744	0.361	2.06
30000	1.667	0.846	1.97
40000	2.968	1.426	2.08
50000	4.658	2.269	2.05
		Average	**2.05**
Activity Grid Dimension (D)			
100	0.018	0.012	1.50
1000	0.042	0.036	1.17
2000	0.206	0.200	1.03

5000	0.792	0.787	1.01
10000	1.769	1.763	1.00
		Average	**1.14**

Table 9: Performance speedup of using shared memory for n-bodies array.

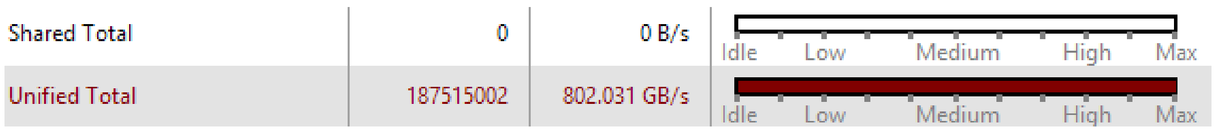
As shown in **Table 9**, the performance for various values of **N** is 2 times faster when using shared memory. For **D**, the performance speedup decreases as **D** increases.

4.4.4 Summary


Kernel Performance Is Bound By Compute

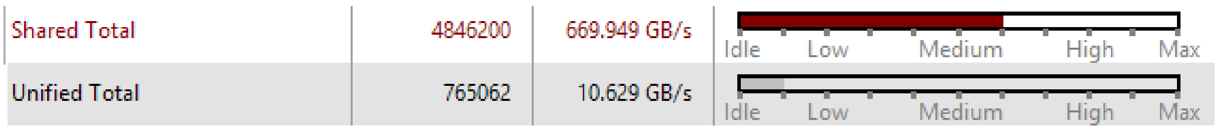
For device "GeForce GTX 1060" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

Figure 13: Kernel analysis of `compute_force` after using shared memory.



Shared Total	0	0 B/s	Idle Low Medium High Max
Unified Total	187515002	802.031 GB/s	Idle Low Medium High Max

(a) Before using shared memory



Shared Total	4846200	669.949 GB/s	Idle Low Medium High Max
Unified Total	765062	10.629 GB/s	Idle Low Medium High Max

(b) After using shared memory

1. According to **Figure 13**, memory operations have reduced significantly and the kernel performance is now limited by arithmetic operations.
2. Global memory loads are reduced by using shared memory to cache the n-bodies.

4.5 Dynamically Calculate Shared Memory Usage

Currently the shared memory is using 256 threads per block (inherited from baseline implementation). **Listing 16** attempts to improve the performance by using the optimal block size calculated by CUDA API.

```
int requiredSM(int blockSize) {
    return blockSize * sizeof(float3);
}
...
cudaOccupancyMaxPotentialBlockSizeVariableSMem(&minGridSize, &block_size,
                                               compute_force, requiredSM);
...
compute_force<<<blocksPerGrid, block_size, requiredSM(block_size)>>>(...);
```

Listing 16: Calculating block size that achieves maximum potential occupancy.

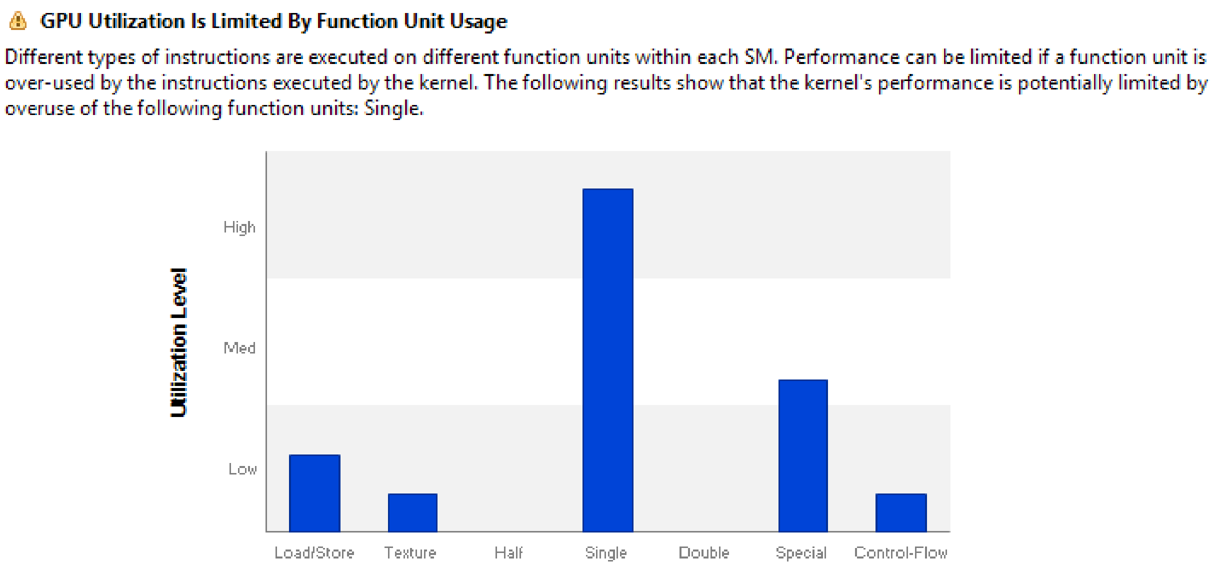
Value	Total Execution Time (seconds)		Speedup
	Before (Static shared, 256 block size)	After (Dynamic)	
Number of Bodies (N)			
1024	0.005	0.014	0.36
2048	0.009	0.023	0.39
4096	0.022	0.045	0.49
10000	0.093	0.113	0.82
20000	0.360	0.446	0.81
30000	0.850	1.002	0.85
40000	1.425	1.781	0.80
50000	2.270	2.728	0.83
		Average	**0.67**
Activity Grid Dimension (D)			
100	0.012	0.019	0.63
1000	0.036	0.043	0.84
2000	0.200	0.207	0.97
5000	0.786	0.794	0.99
10000	1.763	1.770	1.00
		Average	**0.88**

Table 10: Performance comparison of static and dynamic size for shared memory.

Summary: Using the block size calculated by CUDA produces worse performance. CUDA returned a block size of 1024, which causes the larger performance loss when the value of **N** is smaller. This has led to inefficient execution as the 'tail' of the blocks are large. Therefore, it is not suitable to use this technique.

5 Compute Analysis

5.1 Limitations


GPU Utilization Is Limited By Function Unit Usage

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is potentially limited by overuse of the following function units: Single.

Figure 15: Single precision instructions limiting kernel performance.

```
1  const float2 dist = make_float2(s_nbody.x - x[i], s_nbody.y - y[i]);
2  const float inv_dist = rsqrtf(dist.x * dist.x + dist.y * dist.y + SOFTENING_SQUARE);
3  const float m_div_mag = s_nbody.z * inv_dist * inv_dist * inv_dist;
4
5  // Execution dependency, line 6, 7 depends on line 1, 3
6  local_sum.x += m_div_mag * dist.x;
7  local_sum.y += m_div_mag * dist.y;
```

Listing 17: Most compute intensive part of the `compute_force` kernel.

As shown in **Figure 15**, the only limiting factor is the single precision instructions. Special instructions are unavoidable as the equation requires square root calculation. Some conditional statements are also required to prevent reading beyond allocated when **N** is not a multiple of block size.

There is also an unavoidable execution dependency in **Listing 17**. This increases the latency of `compute_force` kernel as warps would be stalling waiting for the previous instructions to complete.

5.2 Possible Improvements

Question: Is it possible to speed up the single precision computation?

In terms of code optimisation, the possibility is low. A significant improvement was already achieved by using CUDA `rsqrtf()` function in **Section 3.2**, and the calculations are all in its simplest form. Running the simulation on a better GPU with higher processing power (GFLOPS) might be the easiest way.

Question: Is it possible to reduce the amount of computation of the current implementation?

The answer is yes. The current implementation is calculating the summation of force n^2 times. In **Equation (1)**, it is required to calculate the term $\frac{(\vec{x_j}-\vec{x_i})}{(||\vec{x_j}-\vec{x_i}||^2+\epsilon^2)^{\frac{3}{2}}}$ n times for each body.

$$\vec{F_i} = Gm_i \sum_{j=1}^{N} \frac{m_j(\vec{x_j}-\vec{x_i})}{(||\vec{x_j}-\vec{x_i}||^2+\epsilon^2)^{\frac{3}{2}}} \tag{1}$$

However, it is known that

$$\vec{x_j}-\vec{x_i} \equiv -(\vec{x_i}-\vec{x_j}) \tag{2}$$

$$||\vec{x_j}-\vec{x_i}||^2 \equiv ||\vec{x_i}-\vec{x_j}||^2 \tag{3}$$

Let matrix D, where $d_{ij} = \vec{x_j}-\vec{x_i}$ initially represents the distance between i^{th} and j^{th} body,

$$D = \begin{bmatrix} d_{11} & \dots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{n1} & \dots & d_{nn} \end{bmatrix}$$

Substituting (2) into matrix D,

$$D = \begin{bmatrix} 0 & d_{12} & d_{13} & \dots & d_{1n} \\ -d_{12} & 0 & d_{23} & \dots & d_{2n} \\ -d_{13} & -d_{23} & 0 & \dots & d_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -d_{1n} & -d_{2n} & -d_{3n} & \dots & 0 \end{bmatrix} \tag{4}$$

As seen in (4), D is now similar to a symmetric matrix except that the elements in lower triangular are the opposite of the one in upper triangular.

According to (3), the term $(||\vec{x_j}-\vec{x_i}||^2+\epsilon^2)^{\frac{3}{2}}$ can therefore be applied into matrix D as well,

$$D = \begin{bmatrix} 0 & \frac{d_{12}}{(||d_{12}||^2+\epsilon^2)^{\frac{3}{2}}} & \dots & \dots & \frac{d_{1n}}{(||d_{1n}||^2+\epsilon^2)^{\frac{3}{2}}} \\ \frac{-d_{12}}{(||d_{12}||^2+\epsilon^2)^{\frac{3}{2}}} & 0 & \dots & \dots & \frac{d_{2n}}{(||d_{2n}||^2+\epsilon^2)^{\frac{3}{2}}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{-d_{1n}}{(||d_{1n}||^2+\epsilon^2)^{\frac{3}{2}}} & \frac{-d_{2n}}{(||d_{2n}||^2+\epsilon^2)^{\frac{3}{2}}} & \dots & \dots & 0 \end{bmatrix} \tag{5}$$

As shown in (5), it is only required either the upper triangular or lower triangular of the matrix. Since each body could have different mass, it cannot be integrated into matrix D.

Assuming the size of the matrix is N,

$$\text{Total elements in upper triangular} = \frac{N(N-1)}{2}$$
$$\text{Total elements in the matrix} = N^2$$

$$\therefore N^2 - \frac{N(N-1)}{2} = \frac{N(N+1)}{2} \quad \text{less computations to do}$$

5.3 Feasibility of the Matrix Implementation

A potential $\mathcal{O}(n^2)$ performance improvement is good, but the implementation difficulty might have exceeded the scope of this assignment.

Below is a preliminary analysis of the matrix implementation procedure:

1. Allocate a triangular array of size `sizeof(float2) * N * (N - 1) / 2`. Vector type is required to represent the distance with `x` and `y`.
2. Define a new kernel that is responsible for calculating the upper (or lower) triangular
3. For simplicity, launch the kernel with 1D block size.
4. Since the upper triangular is now represented by an 1D array, it will need special equations to calculate the corresponding $(row, column)$ for a given linear index. Navarro and Hitschfeld [12] has provided an equation in that uses block-wise mapping to calculate the $(row, column)$.
5. For each linear index, calculate the $(row, column)$.
6. Compute $\dfrac{(\vec{x_{col}} - \vec{x_{row}})}{(||\vec{x_{col}} - \vec{x_{row}}||^2 + \epsilon^2)^{\frac{3}{2}}}$
7. Store the result into the array.

There are several challenges that must be overcome:

1. **Memory Access Pattern:** In this implementation, each thread of a warp are not accessing the adjacent `x` and `y`. The access pattern is likely to be strided or offset access. There are several algorithms proposed by Gorawsk and Lorek [11] that could be adapted for efficient thread-wise traversal.
2. **Memory Latency:** How to divide the triangular array efficiently into blocks to utilise shared memory?
3. **Conditional Branches:** The summation of force for each body is traversing the matrix row-by-row. Since the triangular array does not represent the whole square matrix, two conditional branches are required for detecting the diagonal and the other triangular part.

In summary, this is a possible implementation that can be made. However, majority of the techniques including indices mapping and triangular matrix traversal require an intermediate level of knowledge in computer algorithm. This has exceeded the scope of this assignment and therefore is impractical to implement it.

6 Alternative Method: Parallelise Each Body Interaction (N $\times$ N Threads)

6.1 Implementation

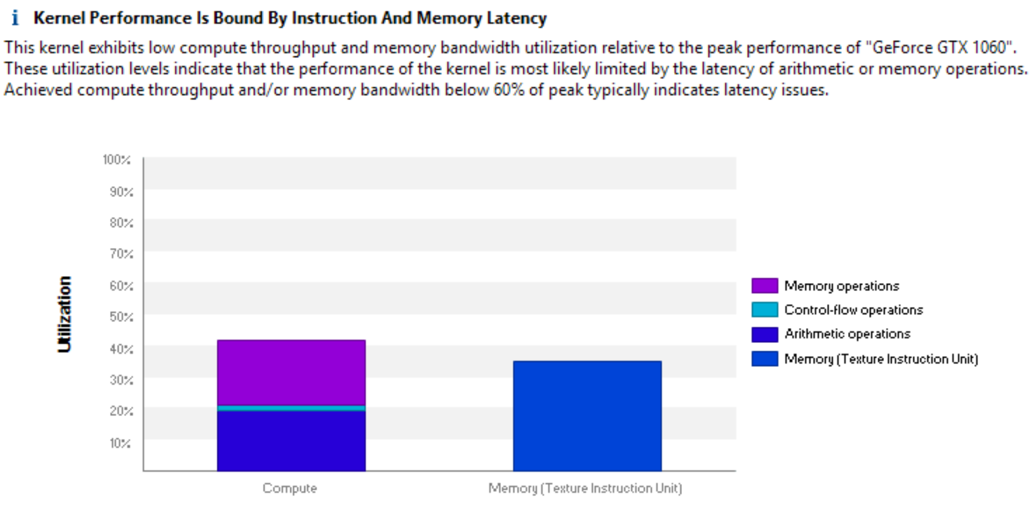
Primary differences between Parallelising Each Body (N threads):

1. **Grid and block dimension:** Inheriting the 256 threads per block, the `compute_force` kernel is launched with a 2D block size of 32×8.
2. **Calculating the summation of forces**: Each thread in a block is now responsible for calculating one force only. For each body B, there are two way to calculate the summation of forces:
   (a) **Atomic operations:** This is a simpler but slower implementation. Each force calculated by a thread is added to the `force_sum` variable through `atomicAdd()`.
   (b) **Matrix row summation:** A similar concept to **Section 5.3**. However, the whole $N \times N$ matrix is calculated, where each element in the matrix (f_{ij}) represents the force between i^{th} and j^{th} body. The summation of force for

6.2 Results

Value	Total Execution Time (seconds)		
	N threads (baseline)	$N \times N$, atomic	$N \times N$, matrix
Number of Bodies (N)			
1024	0.005	0.018	0.017
2048	0.009	0.055	0.025
4096	0.022	0.221	0.049
10000	0.094	1.395	0.177
20000	0.364	5.380	0.699
30000	0.860	12.741	1.633
40000	1.440	22.106	2.783
50000	2.302	34.890	4.320
Average speedup	**-**	**0.11**	**0.47**
Activity Grid Dimension (D)			
1000	0.012	0.023	0.015
2000	0.036	0.049	0.043
5000	0.200	0.232	0.217
10000	0.787	0.886	0.834
15000	1.764	2.124	1.831
Average speedup	**-**	**0.77**	**0.89**

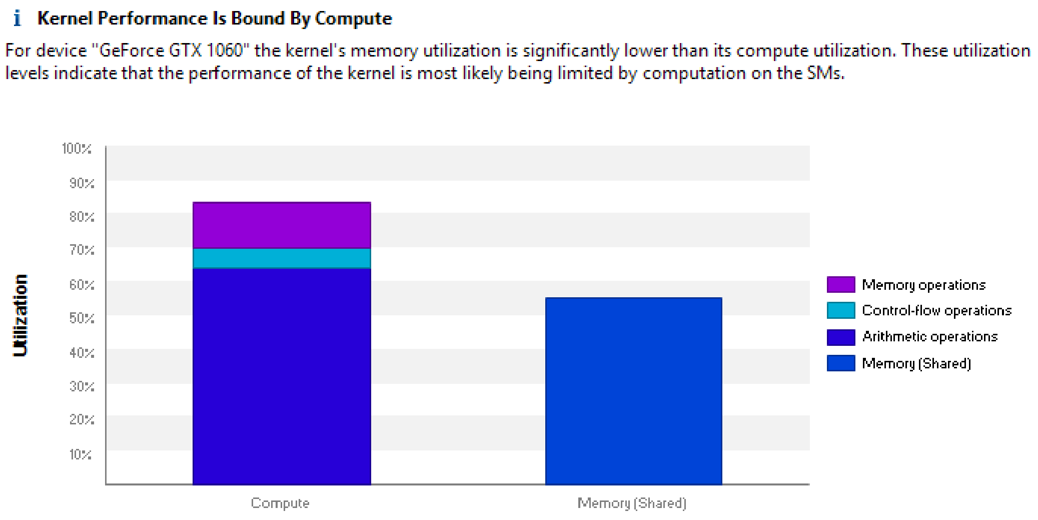
Table 11: Performance of Parallelise Each Body Interaction method.



Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1060". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

(a) Atomic operations



Kernel Performance Is Bound By Compute

For device "GeForce GTX 1060" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

(b) Matrix row summation

Figure 16: Kernel analysis of `compute_force` on parallelise each body interaction method.

6.3 Why N threads version is faster

As shown in **Table 11**, parallelising each body interaction (i.e. the force) is significantly slower than parallelising each body. Justifications are given below:

1. For atomic operations method, the performance loss increases exponentially when **N** increases. When **N** is large, a huge amount of time is spent on executing the force summation in serial due to the atomic section.

2. The matrix row summation method is faster than atomic operations. In comparison with N thread version, this method requires an extra amount of computation. After calculating the forces between each body, it is still need to sum each row of the matrix to obtain the summation of forces for each body. This extra step is likely to have added a substantial amount of execution time.

7 Summary

Effective Optimisation	Avg. Speedup for **N**	Cumulative Speedup for **N**
Baseline implementation	1.00	1.00
Block size from 32 to 256	1.31	1.31
Use CUDA `rsqrtf()`	4.11	5.38
Read-only mem. for SoA	1.09	5.87
Shared memory	2.05	**12.03**

Table 12: Summary of speedup achieved for **N** by effective optimisations.

Effective Optimisation	Avg. Speedup for **D**	Cumulative Speedup for **D**
Baseline implementation	1.00	1.00
Block size from 32 to 256	1.61	1.61
Use CUDA `rsqrtf()`	2.06	3.32
Read-only mem. for SoA	1.02	3.38
Shared memory	1.14	**3.86**

Table 13: Summary of speedup achieved for **D** by effective optimisations.

1. 12.03x speedup on **N** and 3.86x speedup on **D** compared to the baseline implementation
2. Kernel performance in only limited by single precision computations

References

[1] *11.1.3. reciprocal square root*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#reciprocal-square-root` (visited on 29/04/2020).

[2] *11.1.6. math libraries*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#math-libraries` (visited on 29/04/2020).

[3] J. Appleyard, *Cuda pro tip: Optimize for pointer aliasing*, NVIDIA, 7th Aug. 2014. [Online]. Available: `https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing` (visited on 09/05/2020).

[4] *B.12. atomic functions*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions` (visited on 07/05/2020).

[5] *B.2.5. __restrict__*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#restrict` (visited on 09/05/2020).

[6] *B.3. built-in vector types*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#built-in-vector-types` (visited on 29/04/2020).

[7] *B.6. synchronization functions*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#synchronization-functions` (visited on 30/04/2020).

[8] *Cuda-samples/devicequery*, NVIDIA. [Online]. Available: `https://github.com/NVIDIA/cuda-samples/tree/master/Samples/deviceQuery` (visited on 28/04/2020).

[9] *E.1. standard functions*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#standard-functions` (visited on 29/04/2020).

[10] *F.3.9.3. function parameters*, NVIDIA. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#function-parameters` (visited on 30/04/2020).

[11] M. Gorawsk and M. Lorek, 'Efficient processing of large data structures on gpus: Enumeration scheme based optimisation', *International Journal of Parallel Programming*, vol. 46, pp. 1063–1093, 4th Jul. 2017. DOI: `10.1007/s10766-017-0515-0`.

[12] C. A. Navarro and N. Hitschfeld, 'Gpu maps for the space of computation in triangular domain problems', pp. 375–382, Aug. 2014. DOI: `10.1109/HPCC.2014.64`.