COM6521: Parallel Computing with GPUs (2019/20)
Assignment 2

User Name: elt18sx
Student Registration Number: 190186546

Implement

The structure of the program includes standardized data preprocessing and Nbodies simulation. Preprocessing is mainly to read parameters, generate data and apply for CUDA memory space. The simulation part is processed by the step function, which contains different versions of the kernel functions, compute_volocity_CUDA and updata_location_CUDA. The step function also decides whether to update the heat map according to the parameters. As shown in the pseudo-code below,

```
function main(argc, argv)
    args <- load_args(argc, argv)

    if args.input_file then
        h_bodies <- read_file(args)
    else
        h_bodies <- generate_data(args)
    end if

    cudaMemcpy(d_bodies,h_bodies,...)

    if args.visualisation then
        initViewer(args,step)
        setNBodyPositions(d_bodies)
        setActivityMapData(heat_map)
        startVisualisationLoop()
    else
        cudaEventRecord(start)
        step()
        cudaEventRecord(stop)
        print(toc-tic)
    end if

    free(h_bodies)
    free(heat_map)
    free(d_bodies)
    free(d_heat_map)
    return 0
end function

function step()
    for i = 0 to args.iter do
        compute_volocity_CUDA<<<block,thread>>>(d_bodies)
        updata_location_CUDA<<<block,thread>>>(d_bodies)
        if args.visualisation then
            cudaMemset(d_heat_map,...)
            update_heat_map_CUDA(d_bodies,d_heat_map)
        end if
    end for
end function
```

In the code execution stage, the program will execute the corresponding CUDA kernel function according to the configuration, such as the AoS or SoA structure of the data, or different CUDA memory storage methods. (The methods will be discussed later).

Experiment Environment

The experimental code has already been tested on Instance Hub. The experimental data to be discussed below was generated by a local computer with a 6-core and 12-thread i7-9750H CPU and RTX 2060 GPU with Turing architecture.

Parallelizing Method

I used N thread to execute the program, which mainly contains 2 advantages. The first is that the code is easy to transplant, you can simply transplant the original CPU code, only a small amount of changes to the variables. The second is to use fewer threads to achieve good results. As you can see in the benchmark test later, it can be completed in about 1 second even when with N = 16384 and 200 iterations. And pseudo-code is shown as below,

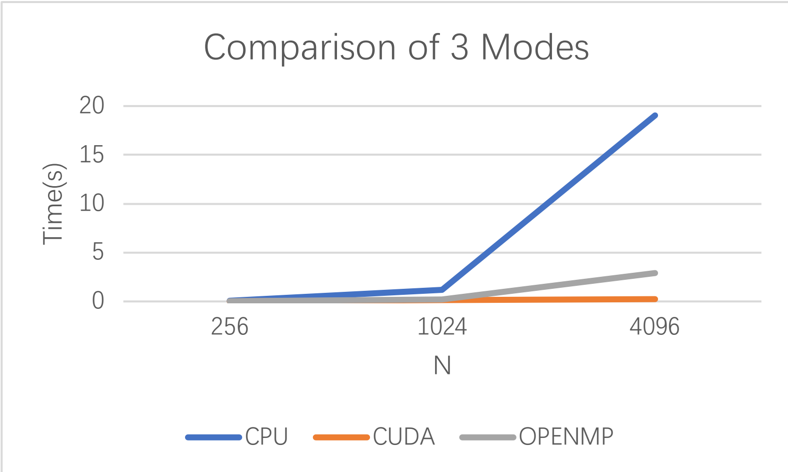
```
function compute_volocity_CUDA(bodies)
  i <- blockIdx * blockDim + threadIdx
  acceleration <- {0,0}
  target_body <- bodies[index]
  for i = 0 to N do
    external_body <- bodies[i]
    acceleration.x <- acceleration.x + acceleration from external_body
    acceleration.y <- acceleration.x + acceleration from external_body
  end for
  bodies[i].vx <- acceleration.x * dt
  bodies[i].vy <- acceleration.y * dt
end function

function updata_location_CUDA(bodies)
  i <- blockIdx * blockDim + threadIdx
  bodies[i].x <- bodies[i].x + bodies[i].vx * dt
  bodies[i].y <- bodies[i].y + bodies[i].vy * dt
end function

function uodate_heat_map_CUDA(bodies,heat_map)
        i <- blockIdx * blockDim + threadIdx
        row<-bodies[i].x/D
        line<-bodies[i].y/D
        atomicAdd(&heat_map[row][line],1/N)
end function
```

Overall performance comparison

Observing the experimental effects of CPU, OPENMP and GPU, we found that GPU is the best, followed by OPENMP. In the polyline above, the X axis is an N variable from 256 to 4096, which increases 4 times each time, while the Y axis is the time spent in the simulation. When the parameter D is 50 and the number of iterations is 200, the polyline in the figure will be generated. (Detailed data can be seen in the benchmark)



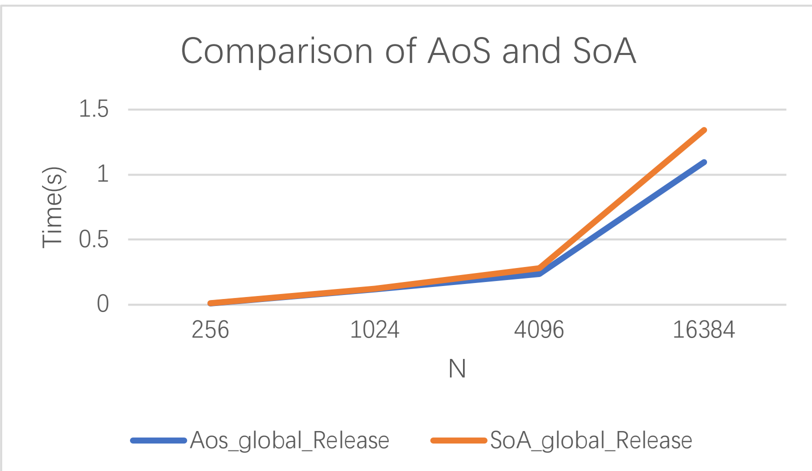
We can observe that when N is increased, the simulation time of CPU, OPENMP and GPU almost all increase linearly. But the growth rate of the GPU model is very low.

N	D	Iter	M	Time(s)
4096	50	200	CPU	19.045
			OPENMP	2.905
			CUDA	0.237

By comparing the above figure with the table, OPENMP is about 6 times faster than in CPU mode, and in GPU mode it can reach an amazing 80 times. This is perfectly reasonable because the number of threads running the simulation is 1 in CPU mode and 12 in OPENMP mode, and the GPU will have N threads, which is 4096. The speed increase is directly related to the number of running threads. However, because the core of the CPU is different from the CUDA core, such as architecture, frequency, and the running speed of a single core, the rate of speed increase will also be different. Secondly, due to the constraints of Amdahl ' s Law, the speed increase after parallelization will also be limited.

Data Structures

Different data structures also have different performance impacts on the program. Here I mainly discuss two data structures, one is AoS, and the second is SoA.



The above figure was completed with 200 iterations. It can be observed that when the amount of data

is small, the execution speed of the two data structures is almost the same. As N increases, there is a slight difference, and it takes less time in the AoS data format.
However, the trend of the difference is not obvious under the current data standard. This difference may be constant (fixed value) or cumulative (increasing with N). Therefore, it is necessary to compare when N is large.

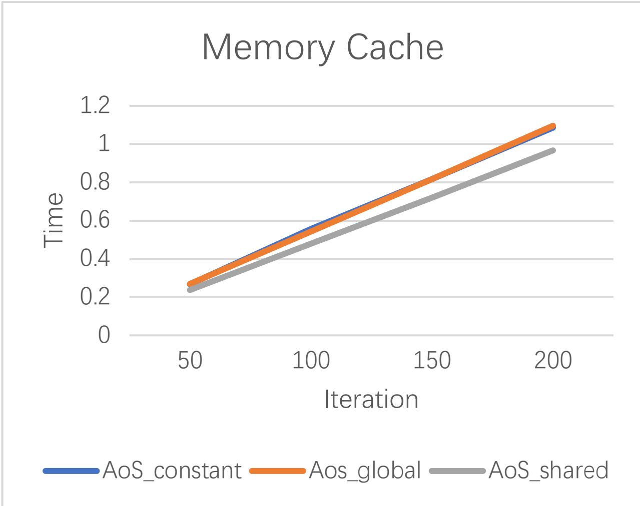
D	M	Iter	file	N	Time(s)
50	CUDA	400	Aos_global_Release	28672	4.524
				32768	8.976
			SoA_global_Release	28672	5.648
				32768	11.017

Through data comparison, when N is 28672, the time difference is 1.124s, and when N is 32768, the time difference is 2.041s. Therefore, the time difference between two different data structures to execute code will increase with the increase of N.
The difference between the two is caused by the speed of accessing memory.
AoS: In AoS mode, all data x, y, vx, vy, and m of the same star are continuously stored in the global memory. For example, the storage distance between the data of the i-th star x and y in memory is 0 bytes, 4 bytes from vx, 8 bytes from vy, and 12 bytes from m. In parallel computing, the corresponding x, y, vx, vy, m data needs to be obtained for each star, and the difference between each data is constant because they are stored continuously.
SoA: However, in the SoA mode, the x of all stars is stored continuously, rather than all data of each star being stored continuously. For example, the storage distance between the data of the i-th star x and y in memory is (1x4)N bytes, (2x4)N bytes from vx, and (3x4)N bytes from vy, and (4x4)N bytes from m. The gap between all data of the same star is not fixed as in the AoS mode, the gap in the SoA structure will continue to increase with N.
The storage differences of different data structures in memory can also be observed through the following table. And this can also explain why the SoA structure takes more and more time as N increases.

Structure	Body	x Address	y Address	vx Address	vy Address	m Address
AoS	0th	0x0···00	0x0···04	0x0···08	0x0···12	0x0···16
	1th	0x0···20	0x0···24	0x0···28	0x0···32	0x0···36
	2th	0x0···40	0x0···44	0x0···48	0x0···52	0x0···56
SoA	0th	0x0···00	0x0···00 + 4N	0x0···00 + 8N	0x0···00 + 12N	0x0···00 + 16N
	1th	0x0···04	0x0···04 + 4N	0x0···04 + 8N	0x0···04 + 12N	0x0···04 + 16N
	2th	0x0···08	0x0···08 + 4N	0x0···08 + 8N	0x0···08 + 12N	0x0···08 + 16N

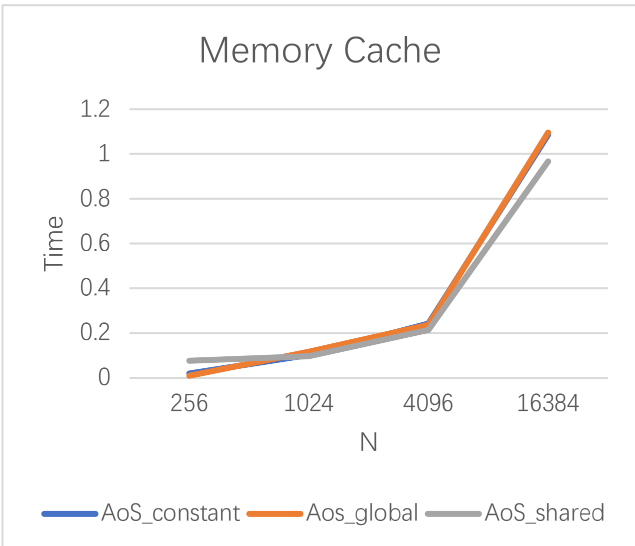
Memory Caches

The way data is cached in the GPU will also have different effects on program efficiency. Here, I mainly discuss the global, constant and shared cache methods.
In the polyline below, the X axis is the variable for the number of iterations from 50 to 200, and the Y axis is the time spent in the simulation. When the parameter D is 50 and N is 16384, the graph on the left will be generated.



Shared vs Global: From the leftmost diagram, you can observe that the shared cache method is faster than the global and constant methods. This is mainly because programs accessing shared data is faster than accessing global data. This is determined by the GPU architecture, each block has its own shared cache, and global is accessible to all threads. Shared cache under this architecture is like L1 cache, and read and write will be faster.

Constant vs Global: As for the constant cache method, its speed is similar to the global speed. Theoretically, the access speed of constant data will be much faster than that of the global cache, but because the constant cache is read-only data, and because of the particularity of the data in the experiment, the constant data can be used is limited, so the speed is not much different. Unlike the shared cache, which can be read and written, the constant cache only has 64k of read-only storage space, so the constant approach also has certain limitations.



Discussion1: is global always faster than global?

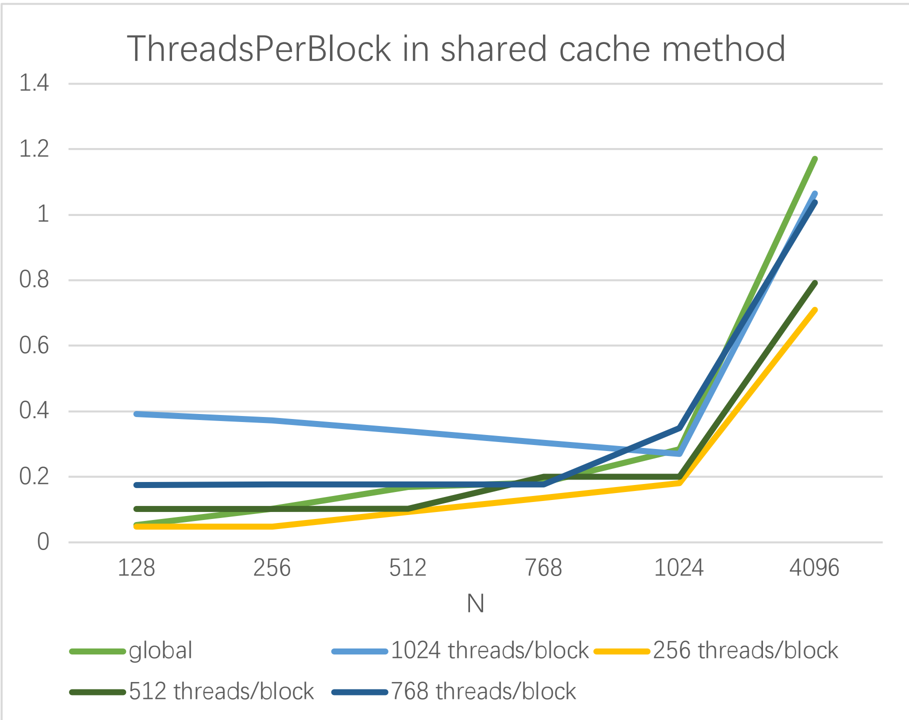
In the polyline above, the X axis is the N variable from 256 to 16384, and the Y axis is the time spent in the simulation. When the parameter D is 50 and the number of iterations is 200, the graph on the left will be generated.

We can find that the shared cache mode is not always faster than the global cache mode. Specifically, we can observe that when N is smaller than 1024, global is faster. When N is 1024, the two are about

the same speed. When N is greater than 1024, the speed of shared is obviously improved.
But why is this so? This is because the shared cache method is to store data in each block for the threads in each block to read. In other words, the greater the number of blocks used for shared cache, the faster the shared cache method will execute. In the current program, I set the number of threads in each block to 1024 (this is also the maximum number of threads in a single block). Therefore, when N is less than or equal to 1024, only one block of shared cache is used to store data. In this case, compared to the global cache, additional operations are required to transfer data to the shared cache. It also needs to wait for the synchronization signal. So the shared cache mode is slower when N is less than 1024 (the number of threads in a single block), and When N is greater than 1024, the shared cache is faster.

Discussion2: ThreadsPerBlock influences shared cache performance

In order to explore more about the content mentioned above, I will set a different number of threads per block for the shared cache method.



Here, we need to compare setting different threads per block in the shared cache method with the global method.

In the figure above, the X axis is different from N, and the Y axis is the time it takes to run the program. Observed from the above figure, the green line represents the performance of global under different N. We first observe the orange line segment, which means setting 256 threads per block. It can be observed that when N is less than 256, the shared speed is faster than global. Dark green indicates that 512 threads per block is set. It can be observed that when N is 512, the shared speed is faster than global. The remaining data also basically conforms to this pattern.

In general, when N is greater than the number of threads per block, shared cache can improve the performance of the program.

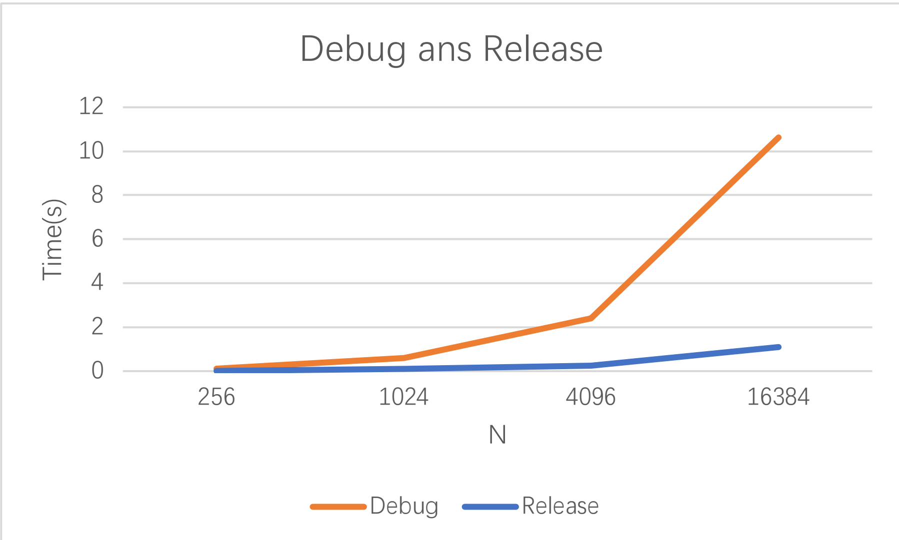
Improvement in Stimulation

1. Data Structure: As discussed above, using AoS will be faster than SoA.

2. Memory Caches: Similarly, as discussed above, the use of shared cache with a reasonable number of threads per block can improve the performance of the code.

In addition, using constant cache to cache commonly used data, such as the values of N and D, can also slightly improve code efficiency. Although the effect is minimal in the current experiment, in the future when a large number of parameters or a large number of fixed values are used, the use of constant cache may bring good benefits.

3. Release and Debug: By changing the code from debug to release mode, the execution efficiency can also be improved.



It can be seen from the above figure that the execution efficiency of Release is about 10 times that of Debug.

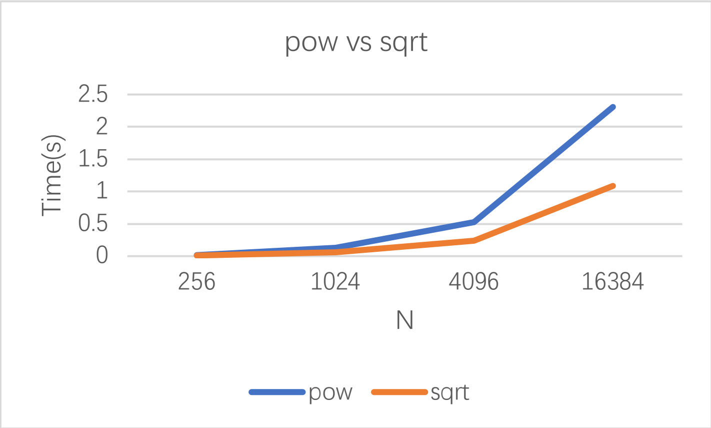
Other Interesting Aspects

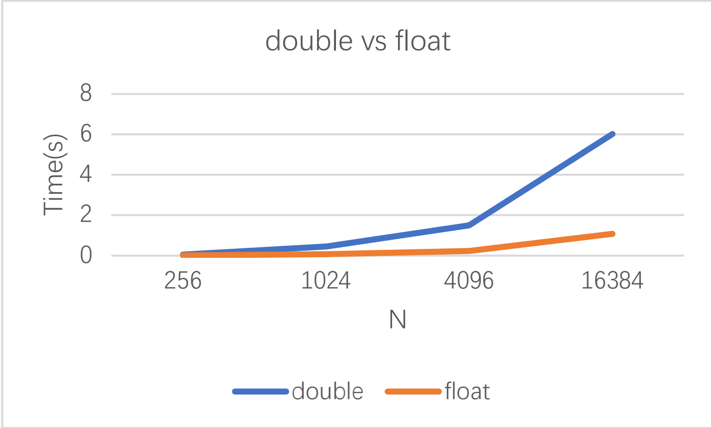
1. sqrt is faster than pow: In the part of calculating the acceleration between the stars. It requires to solved the following formula:

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

The most intuitive way is to use pow (x, 3/2), but when sqrt (x) * x is used instead, the speed becomes surprisingly fast. By observing the line chart below, it can increase the code speed by 2 times.

2. float is faster than double: In the formula of the calculation formula, I initially used the double variable to store the acceleration data, the purpose is to retain the accuracy, but when I changed the double to the float type, the speed increased by 6 times. This is because when a double variable is used, when it is stored in a float variable, a forced data conversion is required.





Experimental Data Table

Benchmark Results table

file	N	D	M	Iter	Time(s)
AoS_constant_Release	256	50	CUDA	50	0.005
AoS_constant_Release	256	50	CUDA	100	0.01
AoS_constant_Release	256	50	CUDA	150	0.015
AoS_constant_Release	256	50	CUDA	200	0.02
AoS_constant_Release	1024	50	CUDA	50	0.03
AoS_constant_Release	1024	50	CUDA	100	0.061
AoS_constant_Release	1024	50	CUDA	150	0.092
AoS_constant_Release	1024	50	CUDA	200	0.102
AoS_constant_Release	4096	50	CUDA	50	0.085
AoS_constant_Release	4096	50	CUDA	100	0.169
AoS_constant_Release	4096	50	CUDA	150	0.193
AoS_constant_Release	4096	50	CUDA	200	0.242
AoS_constant_Release	16384	50	CUDA	50	0.267
AoS_constant_Release	16384	50	CUDA	100	0.555
AoS_constant_Release	16384	50	CUDA	150	0.818
AoS_constant_Release	16384	50	CUDA	200	1.086
Aos_global_Release	256	50	CUDA	50	0.002
Aos_global_Release	256	50	CUDA	100	0.005
Aos_global_Release	256	50	CUDA	150	0.007
Aos_global_Release	256	50	CUDA	200	0.01
Aos_global_Release	1024	50	CUDA	50	0.014
Aos_global_Release	1024	50	CUDA	100	0.029
Aos_global_Release	1024	50	CUDA	150	0.082
Aos_global_Release	1024	50	CUDA	200	0.118
Aos_global_Release	4096	50	CUDA	50	0.091
Aos_global_Release	4096	50	CUDA	100	0.152
Aos_global_Release	4096	50	CUDA	150	0.178
Aos_global_Release	4096	50	CUDA	200	0.237

Aos_global_Release	16384	50	CUDA	50	0.269
Aos_global_Release	16384	50	CUDA	100	0.541
Aos_global_Release	16384	50	CUDA	150	0.818
Aos_global_Release	16384	50	CUDA	200	1.096
AoS_shared_Release	256	50	CUDA	50	0.019
AoS_shared_Release	256	50	CUDA	100	0.039
AoS_shared_Release	256	50	CUDA	150	0.058
AoS_shared_Release	256	50	CUDA	200	0.077
AoS_shared_Release	1024	50	CUDA	50	0.013
AoS_shared_Release	1024	50	CUDA	100	0.037
AoS_shared_Release	1024	50	CUDA	150	0.065
AoS_shared_Release	1024	50	CUDA	200	0.097
AoS_shared_Release	4096	50	CUDA	50	0.1
AoS_shared_Release	4096	50	CUDA	100	0.157
AoS_shared_Release	4096	50	CUDA	150	0.161
AoS_shared_Release	4096	50	CUDA	200	0.214
AoS_shared_Release	16384	50	CUDA	50	0.237
AoS_shared_Release	16384	50	CUDA	100	0.479
AoS_shared_Release	16384	50	CUDA	150	0.721
AoS_shared_Release	16384	50	CUDA	200	0.967
SoA_constant_Release	256	50	CUDA	50	0.003
SoA_constant_Release	256	50	CUDA	100	0.006
SoA_constant_Release	256	50	CUDA	150	0.009
SoA_constant_Release	256	50	CUDA	200	0.012
SoA_constant_Release	1024	50	CUDA	50	0.017
SoA_constant_Release	1024	50	CUDA	100	0.035
SoA_constant_Release	1024	50	CUDA	150	0.097
SoA_constant_Release	1024	50	CUDA	200	0.137
SoA_constant_Release	4096	50	CUDA	50	0.134
SoA_constant_Release	4096	50	CUDA	100	0.259
SoA_constant_Release	4096	50	CUDA	150	0.208
SoA_constant_Release	4096	50	CUDA	200	0.277
SoA_constant_Release	16384	50	CUDA	50	0.309
SoA_constant_Release	16384	50	CUDA	100	0.621
SoA_constant_Release	16384	50	CUDA	150	0.938
SoA_constant_Release	16384	50	CUDA	200	1.252
SoA_global_Release	256	50	CUDA	50	0.003
SoA_global_Release	256	50	CUDA	100	0.006
SoA_global_Release	256	50	CUDA	150	0.009
SoA_global_Release	256	50	CUDA	200	0.012
SoA_global_Release	1024	50	CUDA	50	0.017
SoA_global_Release	1024	50	CUDA	100	0.035
SoA_global_Release	1024	50	CUDA	150	0.096
SoA_global_Release	1024	50	CUDA	200	0.122

SoA_global_Release	4096	50	CUDA	50	0.076
SoA_global_Release	4096	50	CUDA	100	0.153
SoA_global_Release	4096	50	CUDA	150	0.209
SoA_global_Release	4096	50	CUDA	200	0.279
SoA_global_Release	16384	50	CUDA	50	0.331
SoA_global_Release	16384	50	CUDA	100	0.663
SoA_global_Release	16384	50	CUDA	150	1.008
SoA_global_Release	16384	50	CUDA	200	1.343
SoA_shared_Release	256	50	CUDA	50	0.026
SoA_shared_Release	256	50	CUDA	100	0.052
SoA_shared_Release	256	50	CUDA	150	0.079
SoA_shared_Release	256	50	CUDA	200	0.105
SoA_shared_Release	1024	50	CUDA	50	0.013
SoA_shared_Release	1024	50	CUDA	100	0.027
SoA_shared_Release	1024	50	CUDA	150	0.04
SoA_shared_Release	1024	50	CUDA	200	0.054
SoA_shared_Release	4096	50	CUDA	50	0.053
SoA_shared_Release	4096	50	CUDA	100	0.12
SoA_shared_Release	4096	50	CUDA	150	0.21
SoA_shared_Release	4096	50	CUDA	200	0.213
SoA_shared_Release	16384	50	CUDA	50	0.229
SoA_shared_Release	16384	50	CUDA	100	0.458
SoA_shared_Release	16384	50	CUDA	150	0.688
SoA_shared_Release	16384	50	CUDA	200	0.926

CPU，OPENMP，GPU Results

N	D	M	Iter	Time(s)
256	50	CPU	200	0.073
1024	50	CPU	200	1.19
4096	50	CPU	200	19.045
256	50	CUDA	200	0.01
1024	50	CUDA	200	0.118
4096	50	CUDA	200	0.237
256	50	OPENMP	200	0.025
1024	50	OPENMP	200	0.216
4096	50	OPENMP	200	2.905

AoS and SoA Results table

file	N	D	M	Iter	Time(s)
Aos_global_Release	4096	50	CUDA	400	0.472
Aos_global_Release	8192	50	CUDA	400	0.963
Aos_global_Release	12288	50	CUDA	400	1.52
Aos_global_Release	16384	50	CUDA	400	2.201
Aos_global_Release	20480	50	CUDA	400	2.921
Aos_global_Release	24576	50	CUDA	400	3.702
Aos_global_Release	28672	50	CUDA	400	4.524
Aos_global_Release	32768	50	CUDA	400	8.976
SoA_global_Release	4096	50	CUDA	400	0.572
SoA_global_Release	8192	50	CUDA	400	1.17
SoA_global_Release	12288	50	CUDA	400	1.859
SoA_global_Release	16384	50	CUDA	400	2.769
SoA_global_Release	20480	50	CUDA	400	3.657
SoA_global_Release	24576	50	CUDA	400	4.619
SoA_global_Release	28672	50	CUDA	400	5.648
SoA_global_Release	32768	50	CUDA	400	11.017

Debug and Release Results table

file	N	D	M	Iter	Time(s)
Debug	256	50	CUDA	200	0.114
Debug	1024	50	CUDA	200	0.592
Debug	4096	50	CUDA	200	2.395
Debug	16384	50	CUDA	200	10.628
Release	256	50	CUDA	200	0.01
Release	1024	50	CUDA	200	0.118
Release	4096	50	CUDA	200	0.237
Release	16384	50	CUDA	200	1.096

ThreadsPerBlock Results table

file	N	D	M	Iter	Time(s)
global	128	50	CUDA	1000	0.053
global	256	50	CUDA	1000	0.102
global	512	50	CUDA	1000	0.17
global	768	50	CUDA	1000	0.182
global	1024	50	CUDA	1000	0.284
global	4096	50	CUDA	1000	1.171
1024 threads/block	128	50	CUDA	1000	0.391
1025 threads/block	256	50	CUDA	1000	0.373
1026 threads/block	512	50	CUDA	1000	0.339
1027 threads/block	768	50	CUDA	1000	0.304
1028 threads/block	1024	50	CUDA	1000	0.27
1029 threads/block	4096	50	CUDA	1000	1.065
256 threads/block	128	50	CUDA	1000	0.048
257 threads/block	256	50	CUDA	1000	0.048
258 threads/block	512	50	CUDA	1000	0.092
259 threads/block	768	50	CUDA	1000	0.136
260 threads/block	1024	50	CUDA	1000	0.18
261 threads/block	4096	50	CUDA	1000	0.71
512 threads/block	128	50	CUDA	1000	0.102
513 threads/block	256	50	CUDA	1000	0.102
514 threads/block	512	50	CUDA	1000	0.103
515 threads/block	768	50	CUDA	1000	0.201
516 threads/block	1024	50	CUDA	1000	0.201
517 threads/block	4096	50	CUDA	1000	0.792
768 threads/block	128	50	CUDA	1000	0.175
769 threads/block	256	50	CUDA	1000	0.176
770 threads/block	512	50	CUDA	1000	0.176
771 threads/block	768	50	CUDA	1000	0.177
772 threads/block	1024	50	CUDA	1000	0.348
773 threads/block	4096	50	CUDA	1000	1.038

Math: pow and sqrt Results table

file	N	D	M	Iter	Time(s)
pow	256	50	CUDA	200	0.016
pow	1024	50	CUDA	200	0.133
pow	4096	50	CUDA	200	0.529
pow	16384	50	CUDA	200	2.307
sqrt	256	50	CUDA	200	0.01
sqrt	1024	50	CUDA	200	0.057
sqrt	4096	50	CUDA	200	0.237
sqrt	16384	50	CUDA	200	1.086

Math: double and float Results table

file	N	D	M	Iter	Time(s)
double	256	50	CUDA	200	0.05
double	1024	50	CUDA	200	0.461
double	4096	50	CUDA	200	1.49
double	16384	50	CUDA	200	6.019
float	256	50	CUDA	200	0.01
float	1024	50	CUDA	200	0.057
float	4096	50	CUDA	200	0.237
float	16384	50	CUDA	200	1.078