ACS Lab 1 - Matrix Multiplication

Yanzhuo Zhou (4993918,yzzhou) Xinyun Xu (5082579, xinyunxu) Lingyun Gao (4927672, lingyungao)

Report contains 5 of maximum 6 pages.

1 Introduction

The aim of this experiment is to apply basic acceleration techniques on specific computer hardware and analyse the computational characteristics. We implement matrix multiplication using SIMD extensions (AVX2/SSE), Multi-core (OpenMP) and GPU Acceleration (OpenCL) techniques. For each technique, we will evaluate the performance of every method, improve our designs, analyse our results and summarize a conclusion.

2 Experimental setup

The Graphics Processor Unit is Nvidia's high performance GTX 1080. Our implementations are tested on 64bit Linux machine. The AVX extensions complete register computing by adding vector operations. With the assistance of multi-thread operation, the OpenMP implementation uses parallelization constructions of loops and other functions to optimize its operation time. OpenCL is developed for performing complex parallel calculations on the GPU and the CPU, it is fully supperted by NVIDIA graphics card. The whole environment is based on the TU Delft HPC.
With the provided original matrix A and B, we implement matrix multiplication with both float and double precision. The matrix order varies from 16 to 1024, and the amount of matrix elements ranges from 256 to 1,048,576.
According to the set size of matrix, We implement three computational technologies to test the operation duration. With the prerequisite of certain matrix dimension, each method achieve a relatively steady average speedup factor. In addition, the plots of the relation between matrix size and operating time would be illustrated.

3 Vector inner product

3.1 Description

In this benchmark, a serial implementation of vector inner product is tested. Theoretically, for vector size of 1*N, there will be 1*N multiplication executed in CPU.

3.2 Profile

As can be seen in 1, except that the first running of float vector calculation, the time consuming of vector inner product is almost linear growing with the vector size. Besides, the performance of float calculation always outperformed that of double.

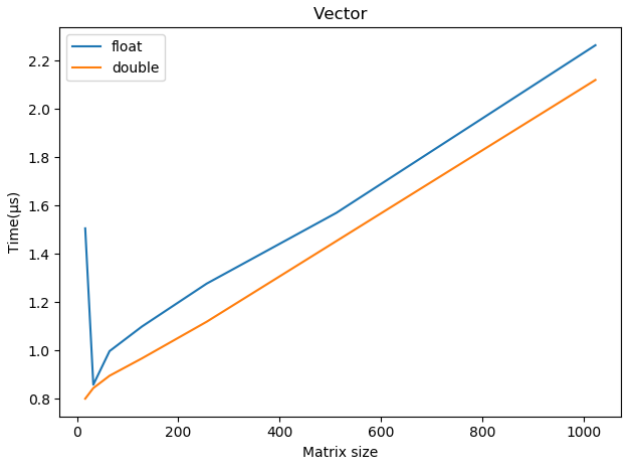

Figure 1: Vector inner product time cost

3.3 Discussion

We can notice from the graph that it's not stable for the time cost of float. Since it is measured with s, we think it's common for some cases like spikes to occur.

4 Matrix multiplication

4.1 Description

In this benchmark, a serial implementation of matrix multiplication is tested. For matrix A with size of M*P and matrix B with size of P*N, there will be M*P*N multiplication.

4.2 Estimation

In this benchmark, we only tested the performance of square matrix, which means for problem with matrix size M*M, there will be M*M*M calculations. Thus we expected a cubic curve with the growth of matrix rows.

4.3 Profile

From 2, it can be seen that the performance of float and double data type are almost the same. When problem size is under 500x500, the curve increases almost follow our estimation. We can notice that after 500x500, the curve started to increase linearly. This is because our testing points are 512 and 1024, which means we don't use points between the two values and pyplot automatically draws a line.

4.4 Discussion

From our theory, we know that the time cost will increase exponentially. From our graph, it is close to 2 exponential. However, analysis of our data shows that

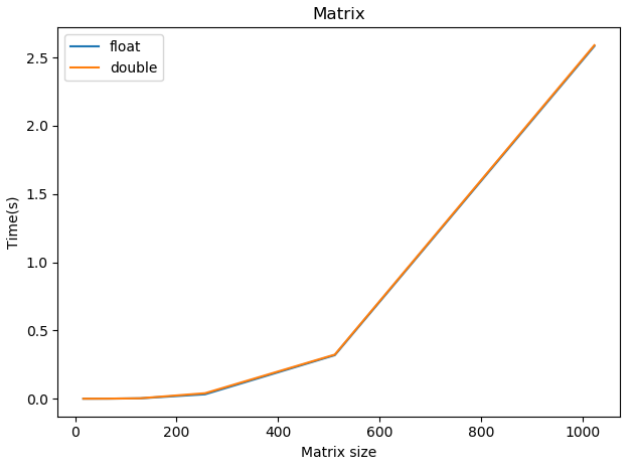

Figure 2: Matrix Multiplication Baseline Time Cost

there exists little difference. We think the consumption for random matrix and creation should be the reason.

5 SIMD extensions

5.1 Description

In this benchmark, we'll use both SSE and AVX2 instruction sets to complete register computing. By adding vector operations and out-of-order execution to each core, the performance of computing can be improved greatly.

5.2 Profile

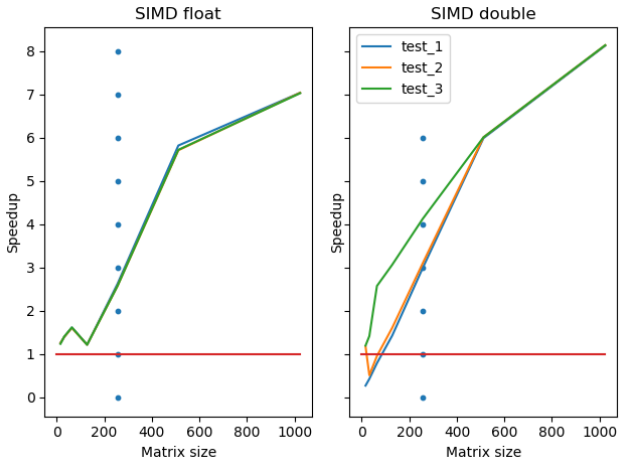

Figure 3: SIMD

In our experiments, we have made two main improvements. For the former one, we test how to make improvements via functions. For the latter one, we test how to make improvements via SIMD. From 3, it shows the speedup for different tests, the left one is for float and the right is for double. Here in our improvement, we use N*M = 256 * 256 with blue dots as a boundary for small matrix and large matrix. The red line symbols the original speed.

5.3 Estimation

The improvements we make in this benchmark are quite effective. After the first improvement, we succeed to speedup the relatively large matrix multiplication. It performs very well with a speedup of 7. The second improvement performs faster with a speedup of 8 even though it performs not well with small matrix.

5.4 Design and implementation

First, we use SSE functions instead of original computation of matrix multiplication. However, when the matrix is relatively large, the speed is nearly original one. So we use different functions and 4*4 blocking to speed up.

5.4.1 Improvement Visiting Address

The main improvement we make here is shifting the function from **_mm_set_ps** to **_mm_loadu_ps** when the matrix is quite large.This is because _mm_set_ps sets packed single-precision (32-bit) floats in dst with the supplied values, while _mm_loadu_ps loads 128-bits from memory into dst. The class Matrix is useless for _mm_loadu_ps, so we use float** to restore the data. When the matrix is below 256*256, we use original method, and when the matrix is quite large, we use loadu (visiting address) to speed up.

5.4.2 Improvement Blocking

To accelerate the speed and achieve the expected results of SIMD, we divide the computation into blocks based on the previous function improvements. In our experiments, we use **4*4 blocks**, which uses 4 vectors containing 4 64-bit double elements for multiplication. Using vectorization instructions can convert an algorithm from operating on a single value at a time to operating on a set of values at a time. This improvement enables our CPU to do SIMD operations.

5.5 Results after improvement

From 3, we can see that the speedup is not so apparent when the matrix is small. When the size of matrix multiplication grows, the speedup is much better than the original version.

5.6 Discussion

Using AVX instruction sets, we gained a great speedup compared with original matrix multiplication. Based on the SSE version, we make some improvements both from functions and SIMD.
The effect is not so good when the matrix is small, and we think it's because the computation is not complex. When the size grows larger, our improvements give great performance. Although we give one single instruction with AVX, every time we can use loading address and 4*4 parallel computing for multiply operations.

6 OpenMP

6.1 Description

OpenMP, namely Open Multi-Processing is an application programming interface. The application built with OpenMP uses parallel programming in order to

accelerate the process. This section is to construct parallel program and test the performance using OpenMP technology.

6.2 Profile

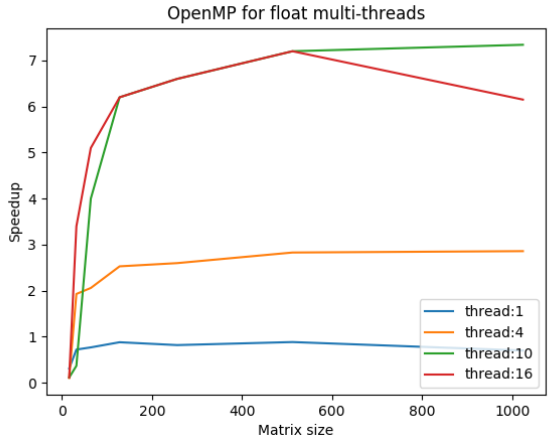

Figure 4: OpenMP for float computing

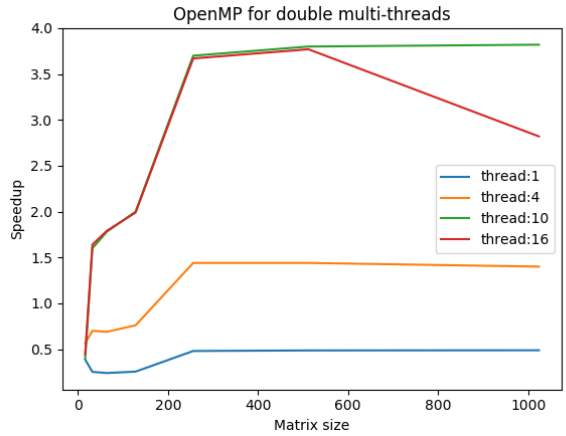

Figure 5: OpenMP for double computing

With the default matrix,we constructed multi-thread matrix multiplication. There can be different operating duration of each thread setting. Compared to the original multiple duration, the speedup factor can be figured out. The variation of matrix size would affect the performance of multi-thread implementation, the larger matrix tends to have large speedup factor with certain thread operation.

6.3 Estimation

When the implementation of a given algorithm can be parallelized, it will benefit from the availability of more processing cores. The speedup factor we achieve should be positively correlate with the thread numbers, as we have enough cores to support the number of threads. Practically, the amount of threads we implement is limited because there is the constraint of hardware and the time of data processing.

6.4 Design and implementation

We designed multi-thread parallel structure,and the matrix multiplication operate on the certain GPU, Nvidia's high performance GTX 1080. So, the performance of multi-thread definitely depends on the testing system. It means that the speedup and throughput could be much higher when the hardware platform is upgraded.

6.5 Improvement

During the experiment, we design the number of threads and the matrix size. Firstly, we attempted to increase the thread of the algorithm. Besides, we found some optimization of OpenMP, especially for loops and iterations. The matrix size is an important factor other than the number of threads, the speedup factor can be affected to some extent.

6.6 Results after improvement

With several thread numbers taken into consideration, the speedup can be calculated. When the thread number is 1, the speedup factor is below 1. With far more tests, the speedup result is around 1. The best average speedup we achieved is when the thread is 10 and the matrix order is 1024. We increase the number a bit higher than 10, the performance seems to be sightly decreasing.

6.7 Discussion

Briefly, OpenMP can be a portable way to optimize the matrix multiplication. Under the certain processing hardware, multi-thread is an effective approach to reduce the operating period with precise codes, while it is not simple to maximize its advantages. Both the hardware platform and the code structure would influence the operating result. After all, parallel programming could help under the multi-core circumstances, it would act in workload division pattern.

7 OpenCL

7.1 Description

In this section, a matrix multiplication program by OpenCL performing on a GPU device(NVidia GTX 1080Ti with 3584 cores). We constructed a two dimensional parallel computing work group. Every work item in this group performs the same kernel to compute an element of the result matrix.

7.2 Profile

The performance of low-dimensional (smaller than 100x100) matrix multiplication is extremely low comparing with serial computing performance mentioned in section 4. The initialization takes around 1.2 seconds and all the low-dimensional multiplication would take around 0.25 seconds.

Only the problem size is high enough, the computing performance (larger than 200x200) becomes better than that of serial computing. As it can be seen in 6, from the baseline, with matrix dimension rising from 200x200 to 1400x1400, the computing performance increases slightly. From our profiling, the unsatisfied per-

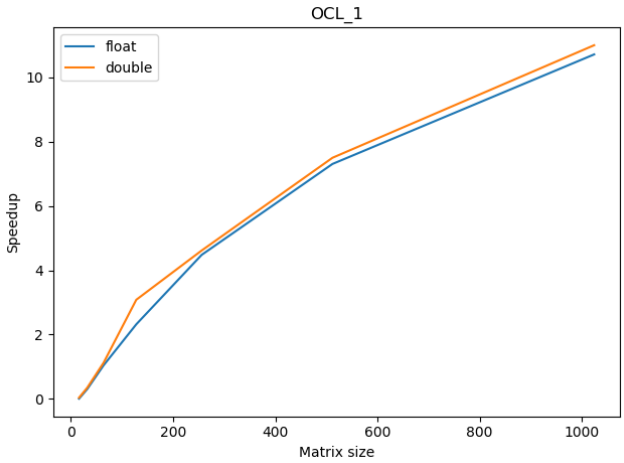

Figure 6: OpenCL Baseline Speedup

formance in low-dimensional computing may cause by time cost of reading and writing data from the host to the device, finding platforms and devices, building kernel and other initialization.

7.3 Estimation

In the baseline, the initialization of finding platforms and devices, building context, kernel, queue and release those space would happen every time when started a new matrix multiplication, but those initialization actually only need ones. By pre-compiling and moving part of the initialization to be global variables, serial part of the program should be reduced a lot.

The improvement of performance is hard to estimate because with the dimension of matrix multiplication rises, the influence of reducing serial part would be different. However, we can ensure that the performance of openCL would exceed serial computing much earlier.

7.4 Design and implementation

We reduced the serial part of the matrix multiplication mainly by moving initialization outside the multiply function.

7.4.1 Improvement by Moving Initialization Outside

As mentioned in section 7.3, initialization functions for finding platforms and devices, building context and building queue are executed in every calling of multiply function. As these setting actually remain the same during the calculation, we can move these initialization outside the multiply function, set global variables to store them and release these variables when finishing multiplication.

7.4.2 Improvement by Pre-compiling

Pre-compiling is to pre-compile the program and store values in a binary file at the first running. Computing after that could just load that binary file. By this operation, the time of loading kernel can be reduced. When improvement of moving initialization was not implemented, the execution time of all matrix size drops by 0.02 seconds on average. However, when first improvement applied, the drop of time can hardly be seen. This improvement thus is not implemented in the final version.

7.5 Results after improvement

From 7, it can be seen that the performance raised significantly after moving initialization and corresponding releasing,. When matrix size is of 1000x1000, the speedup achieved at 73 compared to OpenCL baseline at around 10. the performance is extremely increased. The increasing of performance raised up fast at first, then the gradient of the curve goes down and started to raise up fast again at problem size of approximately 500x500.

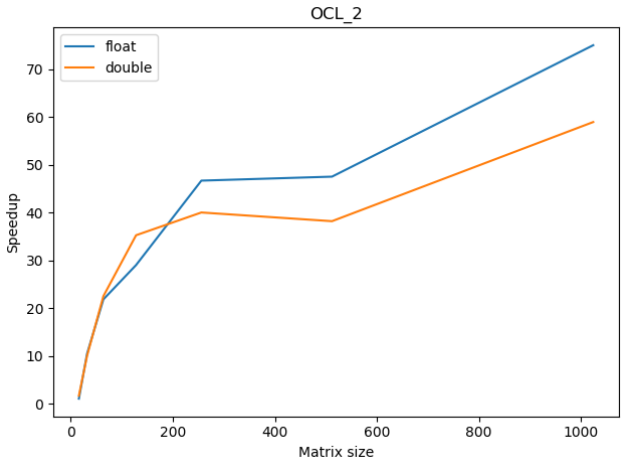

Figure 7: OpenCL Speedup After Improvement

7.6 Discussion

We implemented matrix multiplication by OpenCL and improved the baseline by reducing the serial part of the program. There are two worth-discussing issues during the experiment. The first issue is when matrix multiplication size is lower than 100x100, the time cost of parallel programming by OpenCl almost remain the same. This is because when the program id performed on the device (Nividia 1080 Ti) which has more 3500 cores, every core would perform one element calculating. Thus when problem size is under 60x60(3600), the cores can not be efficiently used. The second issue is that when problem size is higher than(500x500), the performance ratio goes up again, this may be due to the memory limit of the host and devices.

8 Conclusion

In this lab, we complete three experiments implementing matrix multiplication.

For AVX2, this method performs not so well if just use original instructions, but after our improvements it performs better, especially when the size of matrix is very large and the 4*4 blocks can speed up a lot.

For OpenMP, it depends on the hardware of PC. Since our laptop has eight cores, it seems that the number of threads should not be too larger nor too smaller than 8.

For OpenCL, this technology relies on the performance

of GPU. Since our GPU is quite good, the results are much better than the two previous methods.
Since we do three experiments separately, if we can combine these three technologies to work together, the performance must be much better than our results.