A Comprehensive Evaluation of the GPUs Vs CPUs for Parallelization of Matrix Multiplication Algorithms

Yujia Ding

ABSTRACT

Linear Algebra has a fundamental influence on scientific and engineering applications. Although the Graphics Processing Unit (GPU) platform has been superior to the traditional Central Processing Unit (CPU) platform in many previous areas[1], such as image processing and computer graphics, it is not clear how efficient the GPU is compared to the CPU for parallelizing multiple Linear Algebra. Here, we use four existing CPU/GPU models, including OpenMP[2], MPI[3], CUDA[4], and OpenACC[5], to accelerate calculating matrix multiplication. Meanwhile, a comprehensive evaluation was conducted on GPUs and CPUs. Comparing and analyzing speedup, efficiency, scalability, and iterative results have concluded that GPU is beneficial for large-scale data-parallel calculation. Since perfect speedup model s(P) and cost model t(p) based on Amdahl's law are not suitable for CPU performance in reality, I develop an Ln speedup model that perfectly matches the results of MPI and OpenMP.The results show that the GPU implementation can achieve a speedup of more than 4.5 x over CPU implementation of matrix multiplication. In the scalability and iteration test, GPU implementations maintain a high efficiency than CPUs. The computational cost (runtime) of GPU implementations grows slowly with the number of parallel instances and the amount of data handled. It is worth noting that CUDA beats OpenMP, MPI, and OpenACC in all parts, especially in vast amounts of data and parallel instances.

1. INTRODUCTION

Linear Algebra plays an essential role in statistics, engineering, and machine learning. For example, covariance matrix and eigenvalue decomposition are essential parts of face detection. Also, matrix multiplication and Gaussian elimination play crucial roles in solving scientific and engineering problems, including graph theory, analysis, probability and statistics, physics, quantum mechanics, and electronics. It's vital for efficiently processing the scientific and engineering code. However, there are many challenges in choosing appropriate parallel platforms.

1.1 CPU vs GPU

A CPU is a microprocessor that executes instructions given by a program based on arithmetic, control, logic, and input-output. The main task for most CPUs is to execute a sequence of stored instructions, called program[1]. As a result, CPUs follow fetch, decode and execute steps in their operations. The structure of a CPU includes a control unit, arithmetic logic unit (ALU), address generation unit (AGU), and memory management unit (MMU)[6]. The principal component of a CPU is the ALU, which performs arithmetic and logic operations. The goal of designing a CPU is low latency, which is achieved by a large cache. Nowadays, the development of a multi-core processor allows parallel processing on the CPUs, which handle separate parts of an overall task.

A GPU was first designed to accelerate the rendering images intended for output to a display device, which is widely applied in a broad range of other application domains, such as manipulating computer graphics, image processing, and general-purpose computations. Compared to the CPU, GPU has a highly parallel structure and focuses on high throughputs instead of low latency[7]. So it has superior performance in processing large blocks of data in parallel.

In general, the CPU has more hardware allocated to fast intermediate storage but with few powerful cores with very little parallelism. On the contrary, the GPU has hundreds of weaker processing units with independent cache and control, but it takes little memory storage[8].

1.2 Parallel Computing

In processing multiple assigned tasks, a sequential processing approach can be replaced by a parallel programming approach to improve efficiency. Meanwhile parallel computing can solve the serial problem more quickly by dividing up some of the problems into independent parts so that each processing unit can execute some parts of the problem simultaneously.

There are multiple types of parallel processing; two of the most commonly used types include single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD)[5]. SIMD is a form of parallel processing in which a computer will have two or more processors executing the same instruction at any given clock cycle. In contrast, each processor handles different data element. Thus, SIMD is suited for large data sets with a high level of regularity. Typically, GPUs employ SIMD instruction and execution units[4].

MIMD is another common form of parallel processing in which each computer has two or more processors executing different instructions and will get data from separate data streams[1]. Many MIMD architectures also include SIMD execution sub-computer. The MIMD architectures widely appeared in modern supercomputers and multi-cores PCs[9].

Classifying by parallel computer memory architectures, the CPU parallel platform includes OpenMP and MPI. OpenMP

```
for(s = 0; s < m; ++s)
  {
      for(t = 0; t < n; ++t)
      {
          C[s][t]=0;
          for(k = 0; k < m; ++k)
          C[s][t] += A[s][k] * B[k][t];
      }
  }
```

Figure 1: Matrix Multiplication Algorithm

has emerged as the standard for shared-memory parallel programming[2]. In contrast, MPI use cluster of uniprocessors and distributed memory[3].

In my work, I parallel a sequential code of complex calculations on CPU and GPU. This paper used the four most common parallel programming, comparing and analyzing their implementation algorithms and performance. For the CPU approaches, I also compare their performance with that of the theoretical model. Through comprehensive evaluation, it can be concluded that the GPU accelerators have outstanding performance over CPU approaches in complex calculations. In a nutshell:

2. METHODOLOGY

Several experiments were conducted to evaluate and analyze the performance of the CPU and the GPU. Subsection 2.1 describes the serial algorithms of matrix multiplication and the initial setup of the experiments. Subsection 2.2 describes the parallel codes of OpenMP, MPI, CUDA, and OpenACC. Subsection 2.3 describes evaluation metrics and scale tests of CPU and GPU. Subsection 2.4 includes the experimental platforms of CPU and GPU.

2.1 Matrix Multiplication Algorithm

We initially set up the default size of matrices A and B for 1024*1024 (rows*columns) and fill out the matrices with random numbers. Then each element in the nth row in matrix A was multiplied with each component in the nth column in matrix B. The sum of multiply result would fill in an empty matrix C.

2.2 Parallel Algorithm

The default size of matrices A and B is still 1024*1024 (rows*columns) and the value of the matrices is random. The parallelism part of OpenMP includes matrix filling and computation. The parallelism part of OpenMP includes matrix filling and computation. However, through analyzing the runtime matrix filling and multiplying, I found that the runtime of matrix filling is trivial in the serial codes. As a result, I only parallelized the matrix multiplication part in MPI, CUDA, and OpenACC methods. Although there exist nested loops in the matrix multiplication, the multiplying of each row and column of matrices A and B is independent. Thus we can use parallel code in this part.

2.2.1 OpenMP

```
#pragma omp parallel for shared(m, n, A, B)
{
    //fill out matrices A and B
}
#pragma omp parallel for
shared(m, n, A, B)private(nthreads)
{
  for(s = 0; s < m; ++s){
    for(t = 0; t < n; ++t){
      C[s][t]=0;
      for(k = 0; k < m; ++k)
      C[s][t] += A[s][k] * B[k][t];
    }
  }
}
```

Figure 2: Algorithm implementation in OpenMp C

It's easy to implement a parallel algorithm in OpenMP C. All parallel implementations were carried out on CPUs. They can equally access shared memory matrices A and B and the size of matrices (m rows and n columns). We initialize the number of processors and threads of OpenMP to be 64 and 64, respectively.

2.2.2 MPI

MPI parallel Processing through communicating sequential processes via an MPI library call mediated by the operating systems. Here, I used basic communication: One processor sends the data, and another receives the data. The multiplication was conducted both on the main processor (rank==0) and other processors. Meanwhile, we use a sub-block to store computing results transfer from other processors. When running the MPI programming, the default number of nodes is 4 and the number of processors is 64.

2.2.3 OpenACC

I first allocate memory matrices A, B, and C for on GPU and copy data from host to GPU when entering the region and copy the matrix C to the host when exiting region. I state them at the beginning of filling out matrices A, B. Then, I parallelize the matrix multiplication part using statement " pragma acc kernels ". It's worth noting that the acceleration is realized on the GPU while the initialization is conducted on the CPU.

2.2.4 CUDA

Similar to OpenACC, we need to allocate memory on GPU and Copy input data from CPU memory to that of GPU. First, the complex calculation was executed on GPU. Then, we need to Copy results from GPU memory to that of CPU. However, we need to declare kernel function and assign grids, threads, and blocks. The matrix was divided into several sub-matrices into the blocks and shared memory among threads in the matrix multiplication. Loops are over all the sub-matrices of A and B. Then, each thread computes one element of the block sub-matrix. At that time, new sub-matrices would wait until the previous calculation is done using synchronization. Finally, the result of each block sub-matrix was written to the

```
#pragma acc data copyin (A,B), copy(C)
    //fill in matrix A and matrix B
   #pragma acc kernels
    for(int s = 0; s < m; ++s)
      {
        for(int t = 0; t < n; ++t)
          {
            C[s][t]=0;
            for(int k = 0; k < m; ++k)
            {
              C[s][t] += A[s][k] * B[k][t];
            }
          }
      }
```

Figure 3: Algorithm implementation in OpenACC C

device memory. This approach avoids frequent data transfers between CPU and GPU, minimizes access to global memory, and takes advantage of fast shared memory. I initial the block size as 16.

2.3 Evaluation Metrics

2.3.1 *Amdahl's law*

$$Speedup(P) = \frac{1}{1 - P + \frac{P}{S}} \tag{1}$$

2.3.2 *Speedup*

$$Speedup(P) = \frac{Ts}{Tp} \tag{2}$$

2.3.3 *Efficiency*

$$Efficiency(P) = \frac{Speedup(P)}{N} = \frac{Ts}{Tp * N} \tag{3}$$

2.3.4 *Ideal parallel runtime*

$$t(N) = t(s) + \frac{t(P)}{N} \tag{4}$$

2.3.5 *Prefect speedup*

$$s(P) = \frac{t(s) + t(P)}{t(s) + \frac{t(P)}{N}} \tag{5}$$

2.3.6 *Ln speedup*

$$s(P) = \frac{t(s) + t(P)}{t(s) + \frac{t(P)}{1 + Ln(N)}} \tag{6}$$

2.4 Experimental Platform

The serial experiments were run on Comet with Intel Xeon E5-2680v3 CPUs and NVIDIA GPUs. Moreover, the parallel code were run on Expanse with AMD EPYC 7742 (Rome) CPU cores, NVIDIA GPUs and 220 TB total DRAM. Each GPU node contains 4 GPUs and each CPU node contains 128 cores. The compilers of OpenMP, MPI, OPenACC and CUDA is are aocc/2.2.0, openmpi/4.0.4 and openmpi/4.0.4, cuda and pgi/20.4, respectively.

3. RESULTS

3.1 Speedup of Four Models

Table 1: Speedup of Four Models

Types	Runtime (s)	Speedup
serial code	4.831849	NAN
OpenMP	0.607990	7.95
MPI	0.564553	8.98
CUDA	0.156131	36.03
OpenACC	0.183098	30.33

From Table 1 and 6, both GPU and CPU parallel approaches show significant improvement in running time over sequential algorithm when the matrices size is the default. The sequential algorithm spends 0.02 s on initializing matrices A and B and 4.81 s on completing matrix multiplication. However, CPU parallel approaches only take 0.61s and 0.56 s to complete the two instructions. Compared to the original algorithm, OpenMP and MPI's speedup is 7.95 x and 8.98 x, respectively. Furthermore, the runtime decreases to 0.16 s and 0.18 s when using GPU parallel approaches. Compared to the original algorithm, CUDA and OpenACC's speedup is 7.95 x and 8.98 x, respectively. The speedup of GPU is more than 4.5x over 64 CPU cores (CUDA vs. OpenMP). Thus, the sequential matrix multiplication algorithm benefits more from GPU parallelization, especially CUDA.

3.2 Performance of OpenMP in Relation to the Number of Threads

It is essential to find the relationships between the number of processors(threads) and the computational cost of OpenMP. In the perfect model, the runtime would decrease, and the speedup and efficiency would improve following Amdahl's law. In 7, it shows that the relative speedup of perfect models linearly increases with the number of threads because the proportion P / S is INFI. However, the experimental result isn't according to the perfect models. When the number of threads is less than ten, the relative speedup of OpenMP dramatically increases. However, the relative speedup of OpenMP slowly increases when the number of threads is more than ten. Thus, I come up with a new performance model to new model to fit the actual results. The Ln speedup model, which describes in 2.3.6, can almost match the output. The actual speedup of OpenMP is slightly higher than Ln speedup model. The relationship between the number and the computational cost of OpenMP is a natural logarithm.

3.3 Performance of MPI in Relation to the Number of Processors

Here, I analyze the relationships between the number of processors and the computational cost of MPI. I use the MPI

```
if (rank==0){ // main processor-0
        fill out matrices A and B
        for (int e=1;e<numprocs;e++)//send A to other processors
            MPI_Send(A,n*m,MPI_DOUBLE,e,0,MPI_COMM_WORLD);
        for (int f=1; f<numprocs; f++)//send B other processors
            MPI_Send(B+(f-1)*numline*m,m*numline,MPI_DOUBLE,f,1,MPI_COMM_WORLD);
        for (int k=1;k<numprocs;k++){
            //calculating sub blocks in other processors and receive the subset
        MPI_Recv(sub,n*numline,MPI_DOUBLE,k,3,MPI_COMM_WORLD,MPI_STATUS_IGNORE);}
            for (int i=0;i<numline;i++){
                for (int j=0;j<m;j++){
                    C[((k-1)*numline+i)*m+j] = sub[i*m+j];}}
        }//calculating render parts of B
        for (int i=(numprocs-1)*numline;i<m;i++){
            for (int j = 0; j < n; j++) {
             int temp = 0;
             for (int k = 0; k<m; k++) {
                 temp +=A[i*m+k]*B[k*m+j];}
            C[i*m+j] = temp;}
        }
else{ // other processors calculate the sub blocks
       MPI_Recv(A,n*m,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
       MPI_Recv(buff,m*numline,MPI_DOUBLE,0,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
       for (int i=0;i<numline;i++){
           for (int j=0;j<n;j++){
               int temp=0;
               for(int k=0;k<m;k++){
                   temp += buff[i*m+k]*A[k*m+j];}
               sub[i*m+j]=temp;}
       }
       MPI_Send(sub,n*numline,MPI_DOUBLE,0,3,MPI_COMM_WORLD);
  }
```

Figure 4: Algorithm implementation in MPI C

```
__global__ void matrixMul( double* A, double* B, double* C, int nn)
{
    __shared__ double AS[BLOCK_SIZE][BLOCK_SIZE]; //subset
    __shared__ double BS[BLOCK_SIZE][BLOCK_SIZE];

    int bx = blockIdx.x; // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int Col = bx * BLOCK_SIZE + tx;
    int Row = by * BLOCK_SIZE + ty;
   double sub = 0;
  for(int a = 0;a < nn / BLOCK_SIZE;a++){ //for each bloack
            AS[ty][tx] = A[Row * nn + (a * BLOCK_SIZE + tx)]; // WA*ty
            BS[ty][tx] = B[Col + (a * BLOCK_SIZE + ty) * nn];
            __syncthreads();
            for(int k = 0;k < BLOCK_SIZE;k++){
               sub += AS[ty][k] * BS[k][tx];}
            __syncthreads();
        }
        C[Row * nn + Col] = sub; //by * BLOCK_SIZE * wB + ty * wB +bx * BLOCK_SIZ
}
int main(){
//fill out matrices A and B
//allocate GPU memory
cudaMemcpy(d_A, &A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, &B, size, cudaMemcpyHostToDevice);
dim3 gridSize(m/width,n/width);
dim3 blockSize(width,width);
matrixMul<<<gridSize,blockSize>>>(d_A, d_B, d_C, m);
cudaDeviceSynchronize();
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);}
//free GPU memory
```

Figure 5: Algorithm implementation in CUDA C

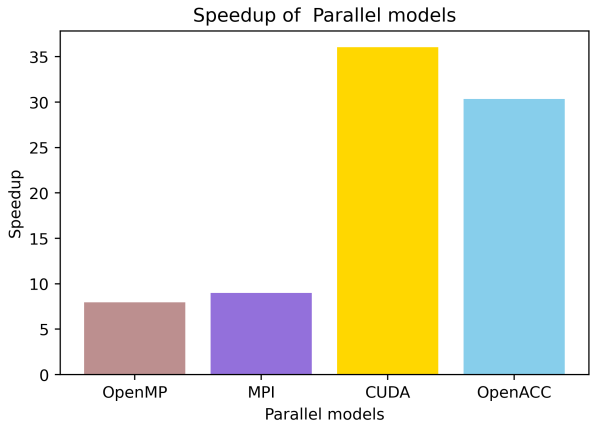

Figure 6: Speedup of Parallel models

Table 2: Computational time in relation to the number of threads and processors

	Processors(16)	Processors(32)	Processors(64)
Threads(1)	4.715958 s	4.586382 s	4.554028 s
Threads(2)	2.585233 s	2.183906 s	2.679290 s
Threads(4)	1.358626 s	1.822314 s	1.587834 s
Threads(8)	1.316069 s	1.100572 s	1.033179 s
Threads(16)	1.141501 s	0.578901 s	0.681321 s
Threads(32)	0.855132 s	0.746020 s	0.613854 s
Threads(64)	0.746981 s	0.651756 s	0.607990 s

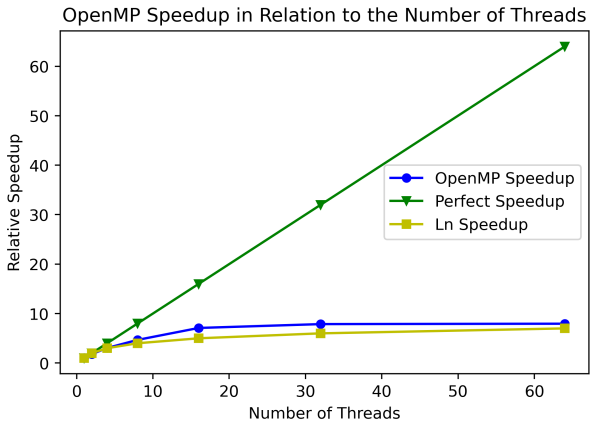

Figure 7: OpenMP Speedup in Relation to the Number of Threads

Table 3: Performance of OpenMP in relation to the number of processors

Threads	Runtime (s)	Speedup	Effi	s(P)	t(N) s
1	4.554028	1.06	1.06	1	4.71
2	2.67929	1.8	0.9	2	1.29
4	1.587834	3.04	0.76	4	0.34
8	1.033179	4.68	0.58	8	0.16
16	0.681321	7.09	0.44	16	0.071
32	0.613854	7.87	0.25	32	0.027
64	0.60799	7.95	0.12	64	0.012

approach to paralyze the matrix multiplication part. When changing the number of processors, it's interesting that the total runtime of the processor decreases in the range of 1 to 64 but slightly increases with 128 processors. Also, the perfect model was calculated considering actual serial time Ts and parallel time Ps. The proportion P / S is 240. Similally, the runtime of the perfect model decreases, and the speedup and efficiency increase following Amdahl's law. In 8. As we expected before, the experimental result isn't according to the perfect models. The relative speedup of OpenMP slowly increases with the number of threads. Thus, I fit the actual results with the Ln speedup model, which describes in 2.3.6. It can perfectly match the output except for 64 processors. The relationship between the number and the computational cost of OpenMP is a natural logarithm.

Table 4: Computational time in relation to the number of processors

Processors	Runtime (s)
1	6.813183
2	6.849331
4	3.474572
8	2.010411
16	1.314662
32	1.255821
64	0.564553
128	0.773205

Table 5: Performance of MPI in relation to the number of processors

Processors	Runtime (s)	Speedup	Effi	s(P)	t(N) s
1	6.813183	0.71	0.71	1	4.85
2	6.849331	0.71	0.35	1.99	2.43
4	3.474572	1.4	0.35	3.95	1.23
8	2.010411	2.43	0.3	7.78	0.62
16	1.314662	3.74	0.23	15.07	0.32
32	1.255821	3.92	0.12	28.37	0.17
64	0.564553	8.98	0.14	50.79	0.095
128	0.773205	6.47	0.05	83.98	0.058

3.4 Scalability Results

It's meaningful to analyze the computational cost on CPU

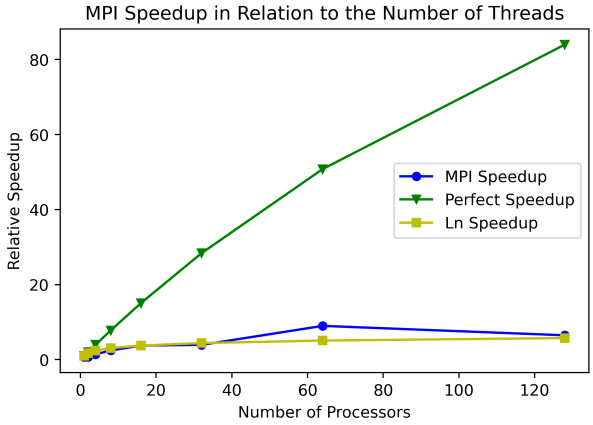

Figure 8: MPI Speedup in Relation to the Number of Processors

and GPU with the amount of data handled. The scalability is essential to evaluate the parallel method because it's most likely to quickly require large processing amounts of data in science and engineering applications. From the simple time complexity analysis of serial code, the matrix multiplication takes O $(m*n*n)$, Since the time complexity of initialization is O $(2*m*n)$ and m is equal to n, the total complexity is O(n^3). If we increase the matrix size k times, the total complexity is O$((kn)^3)$. The runtime will dramatically increase with the amount of data. Most algorithms crush when handling massive data. Here, I expand the matrix size from 1024*1024 to 8192*8192. Firstly, the serial algorithm increases dramatically when the matrix size is twice as original, taking 61.7 s. When the matrix size grows four times, the total runtime is almost 275 times the original, taking 1327.1 s. It's concluded that serial algorithms aren't good at handling large amounts of data.

When using CPU parallel approaches, the runtime indeed decreases than the serial one. For example, the computational cost of OpenMP is 5.63 s, 95.08, and 540s when matrix size increases 2, 4, and 8 times, respectively. Similarly, the runtime of OpenMP is 7.43 s, 58 s, and 471 s when matrix size increases 2, 4, and 8 times, respectively9. Although the CPU approaches save time and improve the efficiency of processors on data explosion, the rate of time increase is still higher. When the matrix size is two times, the increasing rate of OpenMP and MPI are similar to the serial one.

As a result, using GPU parallel approaches is helpful for data explosion. In 10, we can see that CUDA and OpenACC have outstanding performance when the amount of data increasing. Compared to the CPU, the GPU maintains a higher efficiency than the rapid increase of data. The computational cost of CUDA is 0.24 s, 0.625 s, and 2.44 s, and the rate of time increase is 1.5, 4, and 15.6 when matrix size increases 2, 4, and 8 times, respectively. Similarly, the runtime of OpenACC is 0.39 s, 1.3 s, and 5.66 s, and the rate of time increase is 2.2, 7.22, and 31.44 when matrix size increases 2, 4, and 8 times, respectively. Furthermore, using CUDA and OpenACC save computational cost in matrix multiplication. The speedup of CUDA and OpenAcc is 221 and 95 over OpenMP when the matrix size is 8192*8192. Thus, it takes advantage of dealing with complex and huge calculations. Moreover, the CUDA and OpenAcc keep high efficiency. Specifically, the rate of time increase of increase is almost a linear relationship. Last but not least, the scalability of CUDA is better than OpenACC.

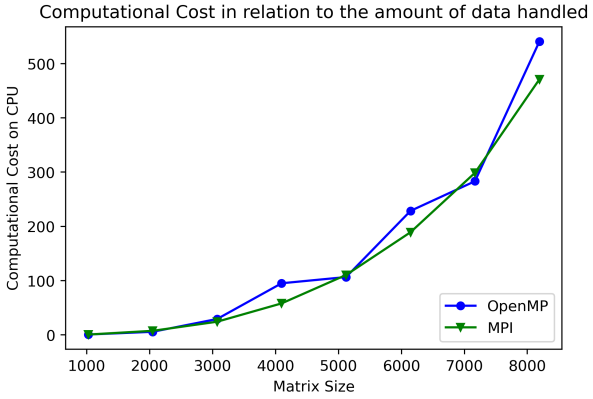

Figure 9: Computational Cost on CPU in relation to the amount of data handled

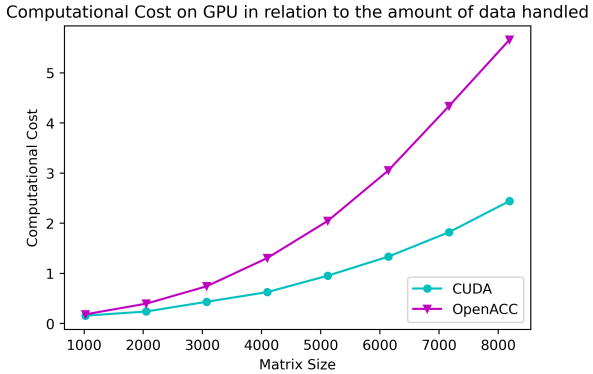

Figure 10: Computational Cost on GPU in relation to the amount of data handled

3.5 Iteration Results

Here, I discuss the computational cost of CPU and GPU with the number of parallel instances. From the simple time complexity analysis of serial code, the matrix multiplication takes O $(m*n*n)$, Since the time complexity of initialization is O $(2*m*n)$ and m is equal to n, the total complexity is O(n^3). If we repeat the matrix multiplication k times, the total complexity is O$(k*n^3)$. In my assumption, the computational cost would linearly increase with the amount of independent instances increase. As expected, when I iterate the separate computation 100 times, the runtime of the serial code is 100 times than before.

In the parallel platform, it's interesting to discuss the relationship between the computational cost and independent parallel instance. Firstly, the correctness of iteration is verified. Then, each iteration is independent, and the nested

Table 6: Scalability Results of four models

K	serial(s)	OpenMP(s)	MPI(s)	CUDA(s)	OpenACC(s)
1	4.831849	0.607990	0.564553	0.156131	0.183098
2	61.742528	5.638206	7.432202	0.238868	0.394832
3	454.37922	29.250321	24.219017	0.432722	0.742139
4	1327.129509	95.083375	57.999021	0.625417	1.304685
5	NAN	106.524016	110.359027	0.953780	2.046905
6	NAN	228.532423	189.164034	1.333959	3.052786
7	NAN	283.227820	298.770309	1.820468	4.335629
8	NAN	540.870028	471.072542	2.441942	5.661031

parallel function doesn't influence the outer iteration because the output of each iteration is the same. It's exciting to find that using the parallel method saves the iterative computational cost. The runtime of CUDA and OpenAcc is slightly increasing with the increase of parallel instances. Beyond the expectation, the increased rate of computational cost is lower than linear. Similar to the above experiment, the CUDA beats other functions with the increase of iteration times.

Table 7: Iteration Results of four models

Types	OpenMP(s)	CUDA(s)	OpenACC(s)
Iteration 100	14.011531	0.406461	1.234428
Iteration 200	29.554252	0.422461	2.305260
Iteration 300	38.512396	0.524049	3.367714
Iteration 400	58.104848	0.625137	4.420174
Iteration 500	87.846226	0.736350	5.496017

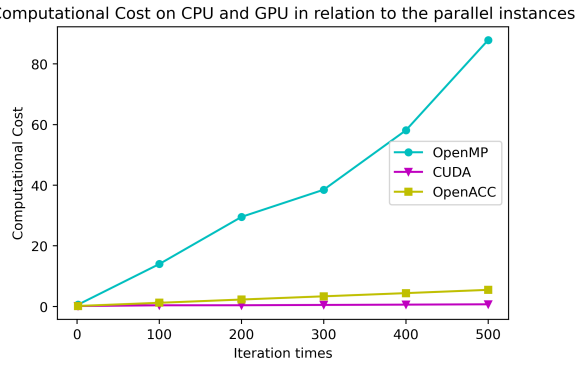

Figure 11: Computational Cost on CPU and GPU in relation to the parallel instances

4. CONCLUSION

Linear Algebra plays a critical role in scientific and engineering applications. Thus, matrix multiplication is usually used as an essential benchmark to test the performance of computer architecture. In this paper, the matrix multiplication algorithm is used to test the performance of CPU and GPU parallel algorithms[10][11]. After implementing these parallel algorithms, comprehensive evaluation was conducted, including speedup, efficiency, scalability, and iteration. According to Amdahl's law, I analyze the relationships between the number of processors(threads) and the computational cost of MPI and OpenMP. Unfortunately, the perfect speedup model s(P) and cost model t(p), seen in 2.3.4 and 2.3.5, is not suitable for the actual speedup and runtime of MPI and OpenMP. Hence, I develop an Ln speedup model that perfectly matches the results of MPI and OpenMP.

Furthermore, the scalability and iteration test is also meaningful. I discuss the relationships between the computational cost on CPU and GPU Platform s and the amount of data and independent parallel instance. Although the CPU approaches maintain the high efficiency of processors on data explosion and iterative cases, the rate of time increase is still higher. Both OpenMP and MPI is not good at dealing with massive complex calculation. Fortunately, such calculation benefits more from GPU parallelization. Especially, CUDA has an outstanding performance than the other three methods in all parts. Compared to OpenMP 64 CPU cores, the speedup of CUDA is 4.5 x and 221 x, respectively, when matrix size is 1024*1024 and 8192*8192. Also, the speedup of CUDA is 118.7 x over OpenMP with 64 CPU cores when repeating the default matrix 500 times.

In conclusion, the CUDA takes advantage of dealing with the vast amount of data in Linear Algebra calculation.

5. REFERENCES

[1] A. Syberfeldt and T. Ekblom, "A comparative evaluation of the gpu vs. the cpu for parallelization of evolutionary algorithms through multiple independent runs," *International Journal of Computer Science & Information Technology (IJCSIT)*, vol. 9, no. 3, pp. 1–14, 2017.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2008.

[3] Y.-Y. Jiao, Q. Zhao, L. Wang, G.-H. Huang, and F. Tan, "A hybrid mpi/openmp parallel computing model for spherical discontinuous deformation analysis," *Computers and Geotechnics*, vol. 106, pp. 217–227, 2019.

[4] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE micro*, vol. 28, no. 4, pp. 13–27, 2008.

[5] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 136–143, IEEE, 2013.

[6] B. M. Reddy, S. Shanthala, and B. VijayaKumar, "Performance analysis of gpu v/s cpu for image processing applications," *Issue II, International Journal for Research in Applied Science and Engineering Technology (IJRASET)*, vol. 5, pp. 437–443, 2017.

[7] D. Castaño-Díez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. S. Frangakis, "Performance evaluation of image processing algorithms on the gpu," *Journal of structural biology*, vol. 164, no. 1, pp. 153–160, 2008.

[8] Y. Zhai, N. Danandeh, Z. Tan, S. Gao, F. Paesani, and A. W. Götz, "Parallel implementation of machine learning-based many-body potentials on cpu and gpu," 2018.

[9] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *2008 International Conference on Computer Science and Software Engineering*, vol. 3, pp. 198–201, IEEE, 2008.

[10] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 55–55, 2001.

[11] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 133–137, 2004.