Advances in Intelligent Systems Research, volume 172
Sriwijaya International Conference on Information Technology and Its Applications (SICONIAN 2019)

Performance Analysis of Parallel Processing on GPU for Simple Mathematical Computations

Mastura Diana MARIESKA[1*], M. Ridho Putra SUFA[2], Adi WIDIANTO[3], Novi YUSLIANI[4], and Rahmat Izwan HEROZA[5]

Faculty of Computer Science, Sriwijaya University, Palembang, Indonesia
**Corresponding author: mastura.diana@informatika.org*

ABSTRACT
Over time, more and more data being produced. We need high-processing computing to process this big data. One solution to this problem is to implement parallel processing using a Graphical Computing Unit (GPU). Theoretically, processing mathematical computation on GPU will always be faster than CPU, because GPU has more than hundreds of Arithmetic Logical Units (ALUs) while CPU only has less than 10 ALUs. But to be able to process data on GPU, we need to explicitly transfer data from RAM to global memory of GPU. This process creates fairly high cost. In this research, we analyze performance of GPU compared to CPU for 2 mathematical computations, namely 1-dimensional vector addition and 2-dimensional matrix multiplication. From experimental result, we can conclude that for 1-dimensional vector addition, however big the data size, it is better to use CPU than GPU. In this case, cost of data transmission is more significant than acceleration of parallel computational process. For 2-dimensional matrix multiplication, if we use matrix larger than 96 x 96 floating point, it is better to use GPU than CPU.
Keywords: *parallel processing, NVIDIA GPU, CUDA, processing time, general purpose computing*

Introduction

In this industry 4.0 era, we need high-processing computing to process big data. One computer, even though it has very high computing power, still has limitations in processing speed. One solution to this problem is to implement parallel processing, which is processing data using more than one agents at the same time. Agent can be a processor, a separate device such as a GPU, or another PC.
One of the parallel solutions that widely used is processing with a Graphical Computing Unit (GPU), which is often called as general-purpose computing on GPU (GPGPU). GPU that was originally designed to process graphics can now be used for scientific computation processing. GPU as computing unit is not much different than CPU, they both have Arithmetic Logical Unit (ALU). ALU in GPU has lower complexity but larger amount. ALU in CPU has higher complexity but small amount. CPU has less than 10 ALUs while GPU has more than 100 ALUs [1]. This characteristic make the GPU is more suitable for simple computation of large data (data parallelism) and CPU is more suitable for complex computation (task parallelism).
Another characteristic of GPU is GPU has its own memory which is separate from RAM. Therefore, each data that we want to process in the GPU must first be explicitly transferred from RAM to GPU. This raises a fairly high cost. This study aims to find out whether simple computation processing on the GPU is faster than the CPU even though there is an overhead in terms of data transmission.
There have been several studies comparing CPU performance with GPU. However, existing research mostly use specific advanced computing cases, such as relativistic fluid dynamic [2], dense linear algebra [3], and Compressed sparse row Matrix-Vector (CsrMV) [4]. This study focuses on the case of basic computing with 1-dimensional and 2-dimensional data. Hopefully, the results of this study can become a reference for researchers when they have to choose between CPU or GPU.
There are 2 cases of basic floating-point computation which used to compare CPU and GPU. Those 2 cases are addition of 1-dimensional vector and multiplication of 2-dimensional matrix. Both cases were chosen because they have low data dependency. Cases with low data dependency are very suitable to be implemented in parallel.
In this research, we used processing time (in milliseconds) as measurement metric. Some experimental tests were conducted to measure the processing time of both cases while using variance of data amount. The benchmarking tests were conducted on CPU and GPU at the same PC. The implementation of the benchmarking application on the GPU uses NVIDIA CUDA, while the CPU uses Java as programming language.
There are 9 sections in this paper. The first section is introduction. Second section describes GPU programming

using CUDA. Other research about CPU and GPU comparison is elaborated in Section III. Section IV is explanation about thread hierarchy and memory hierarchy of CUDA Programming Model. Section V describes application design. Section VI explains about performance metrics. Section VII describes hardware specification and benchmarking scenario. The result of processing time measurement for each scenario in form of graphs is shown in Section VIII. The last section is conclusion.

GPU PROGRAMMING USING CUDA

The GPU is a microprocessor. Main task of GPU is to manipulate pixel data and generate display to monitor. This process includes performing certain tasks such as fast mathematical calculations. The CPU can sequentially execute processes with multiple cores and must wait until the end of execution to start a new one. GPUs have thousands of cores. These cores are capable of performing thousands of parallel mathematical operations.
Technically, programming using a GPU is faster than using a CPU, because the GPU is specifically designed for intensive computing. The calculation in GPU is very parallel, as it for graphical rendering. Therefore, in GPU design, more transistors are devoted to data processing rather than data caching and flow control. Comparison of hardware architecture CPU and GPU is illustrated in Figure 1.

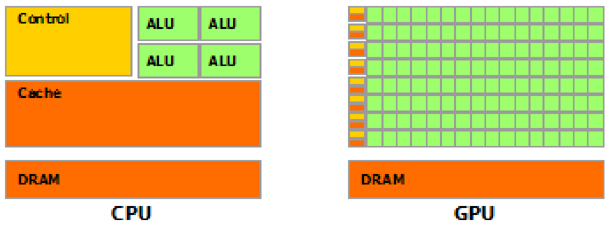

Figure 1. Hardware Architecture of CPU and GPU [5]

NVIDIA corp. introduced CUDA® in November 2006. CUDA is a general-purpose parallel computing platform and application programming interface (API) model that uses NVIDIA GPU parallel computing machines to solve complex computing problems in a more efficient way than on CPUs. By using this platform, software engineers and software developers will be able to use GPU for general purpose computation. This approach is called GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that provides direct access to the GPU's virtual instruction set and parallel computational elements to run the compute kernels.

RELATED RESEARCH

There are some researches in field of GPU-CUDA performance analysis. Among others is [6], which is conducted comparison between CPU-GPU performance using benchmark application GpuTest and Furmark. The conclusion of that research is the performance of the GPU when compared to that of CPU is always step ahead in all application. In reference [3], GPU-CUDA performance analysis is conducted for dense linear algebra. Conclusion of that research is the optimizations attain 80%-90% of the peak speeds possible for large matrices. Reference [4] compares the performance of CPU and GPU-CUDA for Compressed sparse row Matrix-Vector (CsrMV). It concludes that GPU-CUDA has better performance and more consistent than than traditional CsrMV in CPU.
Other researches that focused on GPU-CUDA performance analysis are [2] and [7]. That researches focus on parallel computing on case study. In [2], CUDA-based GPU code performed parallel simulations of relativistic fluid dynamic. The result is CUDA-based GPU code is approximately two orders of magnitude faster than CPU. In [7] proposed GPU based parallel clustering method for electric power big data . The algorithm based on CUDA proposed in this paper solved the problem of massive load curve clustering in CPU with high speedup ratio and strong adaptability performance. In this research, we conducted performance analysis of parallel processing with matrix multiplication in GPU-CUDA, which compared to traditional CPU.

CUDA PROGRAMMING MODEL

Thread Hierarchy

Parallel execution in CUDA is implemented by creating many threads that carried task at the same time. Each thread has index called threadIdx. Thread in CUDA is a 3-component vector. Each thread can be identified using a one-dimensional, two-dimensional or three-dimensional thread index. These threads can form one-dimensional, two-dimensional or three-dimensional blocks of threads, called thread block. This design provides a natural way to calculate all the elements of a domain, such as vectors, matrices or volumes.
The thread index and the thread ID are directly related: for one-dimensional blocks, they are identical; for two-dimensional block sizes (Dx, Dy), the thread ID of the index thread (x, y) is (x + y Dx); for three-dimensional block sizes (Dx, Dy, Dz), the thread ID of the index thread (x, y, z) is (x + y Dx + z Dx Dy).
The number of threads per block is limited because all block threads must be in the same processor core and must share limited memory resources from that core. In the current GPU, thread blocks can contain up to 1024 threads. However, the kernel can be executed by multiple blocks of threads in the same way, so that the total number of threads is equal to the number of threads per block multiplied by the number of blocks.
The blocks are organized into a thread blocks grid of one-dimensional, two-dimensional or three-dimensional, as shown in Figure 2. The number of thread blocks in the grid is generally determined by the size of the data being processed or the number of system processors, which can greatly exceed.
Programmer can specify number of threads per block and number of blocks per grid using <<< ... >>> syntax. The parameter can be of type int or dim3. Two-dimensional blocks or grids can be determined as in the example above. Each block in the grid can be identified by a one-dimensional, two-dimensional or three-dimensional index accessible in the kernel through the default blockIdx variable. The dimension of the thread block in the kernel can be accessed through the default blockDim variable.

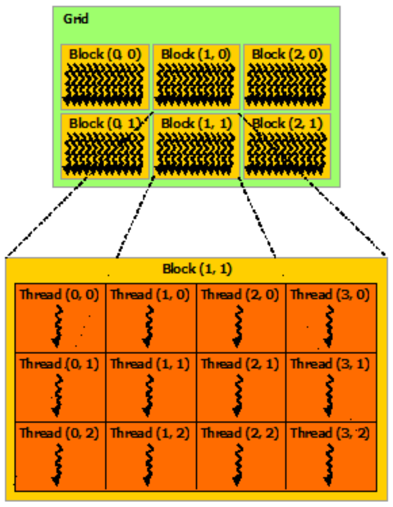

Figure 2. Grid and Thread Blocks [5]

Memory Hierarchy

During its execution, each CUDA threads can access data from multiple memory spaces. This memory hierarchy is illustrated by Figure 3. Each thread has local personal memory. Each block of threads has shared memory visible to all block threads and has the same lifespan as the block. All threads have access to the same global memory.
In addition, there are two other kind of memory; constant and texture memory space. These two memories are read-only memory spaces available for all threads. The global, constant and texture memory space have their own specializations and optimized for different memory uses. The global, constant and textures memory spaces are persistent when starting the kernel with the same application. Texture memory also provides different addressing modes. For some specific data formats, texture memory provides data filtering.

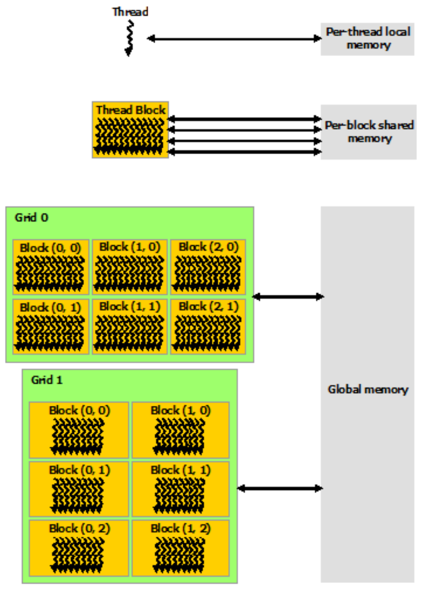

Figure 3. Memory Hierarchy of NVIDIA GPU [5]

APPLICATION DESIGN

Vector Addition

Vector addition is a simple mathematic problem using 1 dimension of data. Given 2 vector with the same size, A and B. Every element of vector A and B will be added to vector C. Vector addition is a basic and simple computation in scientific computing. It has low data dependency. Every ith element can be added separately. The pseudocode of vector addition is as follow.

```
procedure vectorAddition (A,B,C)
    for i = 1 to n do
        C[i] = A[i] + B[i];
    end for;
```

Matrix Multiplication

Matrix multiplication is a simple mathematic problem using 2 dimensions of data. By definition, the product of an a x b matrix A by an u x v matrix B is an a x v matrix C. The pseudocode of matrix multiplication is as follow.

```
procedure matrix_multiplication (A,B,C)
    for i = 1 to a do
        for j = v to k do
            $C_{ij} = 0$;
            for s = 1 to b do
            $C_{ij} = C_{ij}+(A_{is} * B_{sj})$;
            end for;
        end for;
    end for;
```

Matrix multiplication is a common computation in parallel programming because it has low data dependencies. Because the data dependency is low, we can break that data into smaller parts and then work on each of data simultaneously. For example, for multiplying 2 matrices of size a x v, we can take a subset of matrices into of size m x n where 0 < m < a and 0 < n < v. This subset then can be multiplied separately in 1 thread. Linkages between data are taken into account when adding up the results of each element of the multiplication.

PERFORMANCE METRIC

In this research, we use only one performance metric, which is processing time in milliseconds. To measure the time, at the beginning part of the code we will record the start time, while the end part of the code we will record the end time. Processing time is the difference between start time and end time.

Processing Time of CPU

The program implementation on the CPU uses the Java programming language. To measure processing time, we use the System library, precisely the nanoTime() function. The output of this function will be divided by 1,000,000 to get the processing time value in milliseconds. Recording processing time starts after data initializationand ended after the vector addition or matrix multiplication calculation is complete.

Processing Time of GPU

The program implementation on the GPU uses CUDA. Processing time on the GPU is recorded using the cudaEventCreate() function. This function produces values in milliseconds and has a resolution of approximately half a microsecond. The timing of this function is measured on the GPU clock, so the resolution is independent from operating system influence [8]. Start time is recorded after data initialization is complete, but before copying data to GPU memory. This is because copying data to GPU memory is an overhead that must be done in order to be able to process data on GPU. End time recording is done after copying the resulting data from GPU memory to RAM.

Specification and Scenarios

In this research we use one PC that has NVIDIA GPU as graphic card and intel processor as CPU. Hardware specification used for this research is shown from Table 1 below.

Table 1. Hardware Specifications

CPU	
Processor	Intel Core i7-6700HQ
Number of Core	4 cores @ 2.60GHz
L1 Cache	4 x 32 KBytes
L2 Cache	4 x 256 KBytes
L3 Cache	6 MBytes
Memory	8GB
GPU	
GPU Name	Nvidia GeForce GTX 950M
Total Memory	4096MBytes
Number of Core	5 cores
Max Clock Rate	928MHz
Memory Clock Rate	2505Mhz
Memory Bus Width	128-bit
Total Constant Memory	65536 bytes
Total Shared Memory per Block	49152 bytes
Total Number of Register per Block	65536
Maximum Thread per Multiprocessor	2048
Maximum Thread per Block	1024
Max Dimension of a Thread Block	(1024, 1024, 64)
Max Dimension Size of a Grid Size	(2147483647, 65535, 65535)

Benchmarking scenarios are divided into 2 scenarios; small data scenario and large data scenario. Small and large data sizes differ for each case of vector addition and matrix multiplication. Small and large values are determined through experiments. In vector addition, small data size is in the range of 100,000 to 1,000,000 data with a difference of 100,000 data for each variant. While the size of large data in vector addition is in the range of 10,000,000 to 100,000,000 data with a difference of 10,000,000 data for each variant. So, there are a total of 20 variants of data for vector addition cases.

In matrix multiplication, the data used is 2 square matrices of size n x n. In the small data scenario, the value of n is in the range of 16 to 192 data with a difference of 16 data for each variant. Whereas for large data scenarios, the size of the matrix is in the range 192 to 1024 data with a difference of 64 data for each variant. So, there are 12 variants of data for small data scenarios and 14 variants of data for large data scenarios.

For each data variation in each case, the processing time was measured 10 times, then we calculated average values from those measurements. This is because in the general purpose operating system, a running process can be interrupted by operating system if there are other processes that have a higher priority. For this reason, it is important that processing time calculations cannot be done only for once but must be an average of several experiments.

RESULT ANALYSIS

Vector Addition

Figure 4 shows comparation of processing time in CPU and GPU for vector addition.

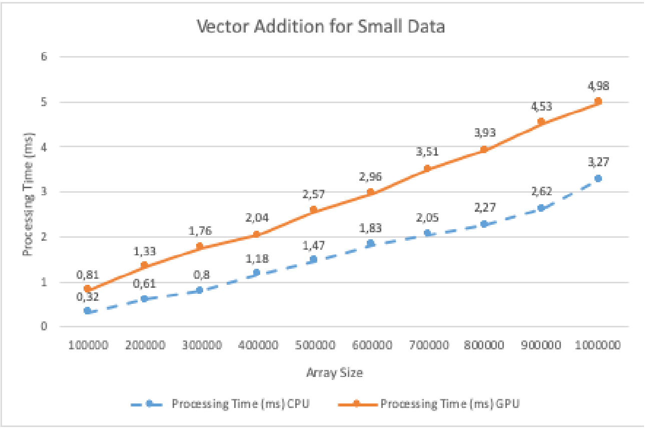

Figure 4. Processing Time of Vector Addition for Small Data

From figure 4 and figure 5, we can conclude that for vector addition, CPU is always faster than GPU. In the small data scenario, the processing time difference between the CPU and GPU is only two times. But in the large data scenario, the processing time difference increases to 3 times. The cost of sending data from RAM to GPU memory is too high and is not proportional to the acceleration of parallel computing.

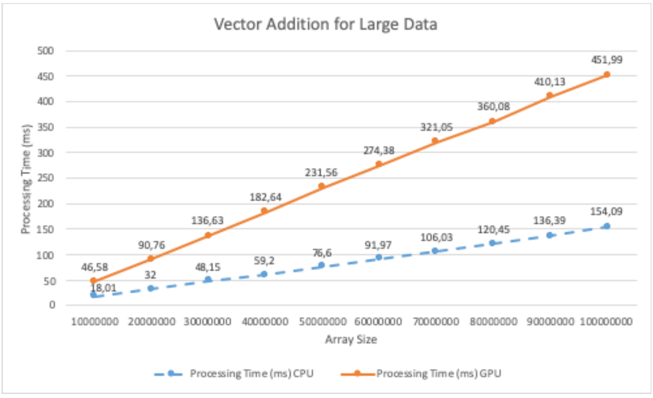

Figure 5. Processing Time of Vector Addition for Large Data

Matrix Multiplication

Graph below is comparation of processing time in CPU and GPU for matrix multiplication.

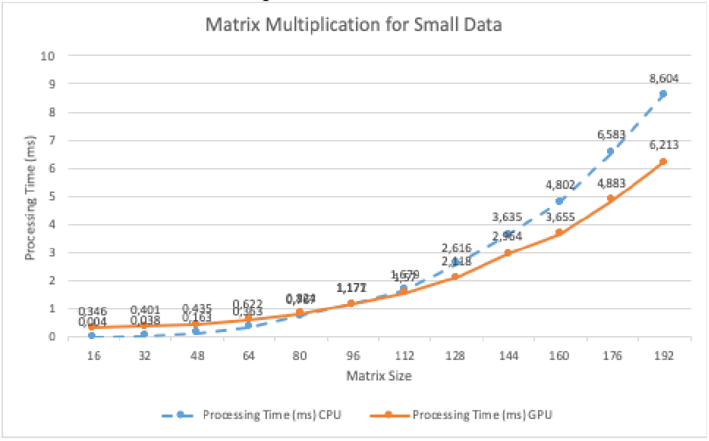

Figure 6. Processing Time of Matrix Multiplication for Small Data

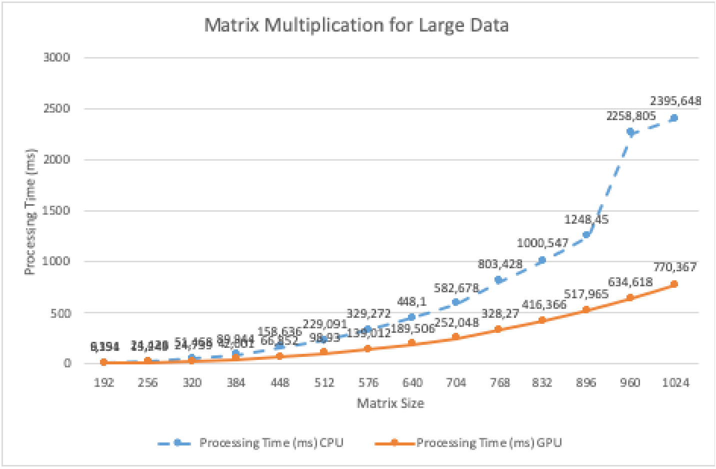

Figure 7. Processing Time of Matrix Multiplication for Large Data

For matrix multiplication case, we can conclude that for small size of data (less than 96 x 96 floating point), processing time of CPU is lower than GPU. For matrix size larger than 96 x 96, processing time of CPU is significantly higher than GPU. At the largest data, which is 1024 x 1024, processing time of CPU is more than 3 times higher than GPU.

Result of this research is a little bit different than other related research. In other research such in [2][3][4], concluded that processing in GPU is always faster than processing in CPU. It turns out that for complex case such as matrix multiplication, GPU is able to process data faster that CPU. This makes the overhead of data transmission from RAM to GPU is less significant. But for simple computation using 1 dimensional data, like in vector addition, CPU can perform faster data processing. Because in simple computation, data transmission cost is more significant than acceleration of parallel processing.

CONCLUSION

We have evaluated performance of GPU compared to CPU for simple mathematical computation. We implemented 2 different kind of computational cases using different data dimension, namely 1-dimensional vector addition and 2-dimensional matrix multiplication. From experimental result, we can conclude that for 1-dimensional vector addition, however much the data size, it is better to use CPU than GPU. In this case, cost of data transmission is more significant than acceleration of computational process. For 2-dimensional matrix multiplication, if we use matrix larger than 96 x 96 floating point, it is better to use GPU than CPU.

REFERENCES

[1] A. Rege, ‘An Introduction to Modern GPU Architecture’, *NVIDIA Present.*, 2009.

[2] D. Bazow, U. Heinz, and M. Strickland, ‘Massively parallel simulations of relativistic fluid dynamics on graphics processing units with CUDA’, *Comput. Phys. Commun.*, vol. 225, pp. 92–113, 2018.

[3] V. Volkov and J. W. Demmel, ‘Benchmarking GPUs to tune dense linear algebra’, *2008 SC - Int. Conf. High Perform. Comput. Networking, Storage Anal. SC 2008*, no. November, 2008.

[4] D. Merrill and M. Garland, ‘Merge-based Parallel Sparse Matrix-Vector Multiplication’, no. November, 2016.

[5] NVIDIA, ‘Cuda C Programming Guide’, *Program. Guid.*, no. September, pp. 1–261, 2015.

[6] C. Naikodi, ‘Performance Evaluation of CPU-GPU with CUDA Architecture Performance Evaluation of CPU-GPU with CUDA Architecture’, no. March, 2017.

[7] C. Ji, Z. Xiong, C. Fang, H. Lv, and K. Zhang, ‘A GPU based parallel clustering method for electric power big data’, *Proc. - 2017 4th Int. Conf. Inf. Sci. Control Eng. ICISCE 2017*, pp. 29–33, 2017.

[8] NVIDIA, ‘CUDA Toolkit Documentation’. [Online]. Available: https://docs.nvidia.com/cuda/. [Accessed: 05-Aug-2019].