International Journal of Internet, Broadcasting and Communication Vol.11 No.3 1-7 (2019)
http://dx.doi.org/10.7236/IJIBC.2019.11.3.1

IJIBC 19-3-1

Performance Comparison of Parallel Programming Frameworks in Digital Image Transformation

Woochang Shin

Dept. of Computer Science, Seokyeong University, Korea
wcshin@skuniv.ac.kr

Abstract

Previously, parallel computing was mainly used in areas requiring high computing performance, but nowadays, multicore CPUs and GPUs have become widespread, and parallel programming advantages can be obtained even in a PC environment. Various parallel programming frameworks using multicore CPUs such as OpenMP and PPL have been announced. Nvidia and AMD have developed parallel programming platforms and APIs for program developers to take advantage of multicore GPUs on their graphics cards. In this paper, we develop digital image transformation programs that runs on each of the major parallel programming frameworks, and measure the execution time. We analyze the characteristics of each framework through the execution time comparison. Also a constant K indicating the ratio of program execution time between different parallel computing environments is presented. Using this, it is possible to predict rough execution time without implementing a parallel program.

Keywords: *Parallel programming framework, PPL, CUDA, OpenMP, OpenCL*

1. Introduction

Parallelism has long been employed in high performance computing, but nowadays it can be applied to general PC environment. In previous generations, the CPU clock frequency has increased year by year in order to improve CPU performance, but since 2004, the CPU manufacturer has abandoned the clock speed competition. This is because it has reached the limit of increasing the CPU clock speed due to the increase of power usage and heat problem. The CPU manufacturer then shifted toward increasing the number of CPU cores instead of increasing the clock speed. In order to speed up image processing in graphics, several companies have announced graphics cards with multicore GPUs. For example, the GTX Titan X product released by Nvidia in March 2015 contains 3072 cores.

While the hardware is turned into a parallel processing environment, software developers are accustomed to developing sequential programs. Even if the number of computing cores increases, existing sequential

Manuscript Received: May. 8, 2019 / Revised: May. 12, 2019 / Accepted: May. 17, 2019
†Corresponding Author: wcshin@skuniv.ac.kr
Tel:+82-2-940-7755, Fax: +82-2-940-7755
Dept. of Computer Science, Seokyeong University, Korea

programs will not benefit. To take full advantage of the performance of a multicore CPU or GPU, the developer must design the program in parallel and write the source code using a parallel programming framework.

As parallel programming becomes important, a number of parallel programming frameworks have been introduced to support parallel programs in various environments. Representative parallelization frameworks include OpenMP (Open Multi-Processing), PPL (Parallel Patterns Library), OpenCL (Open Computing Language), and CUDA. When programmers develop parallel programs, they first choose a parallel programming framework. In order to select the most efficient parallel framework under a specific computing environment, the prototypes applying each parallel framework must be developed and their performance compared with each other. However, this approach is time consuming and costly.

In the present paper, we examine various parallel programming frameworks that can be applied in multicore CPUs and GPUs, and evaluate the performance of each parallel programming framework in digital image processing. A constant *K* indicating the ratio of program execution time between different parallel computing environments is presented. Using this, it is possible to predict rough execution time without implementing a parallel program. The remainder of the paper is organized as follows.

In Chapter 2, related works are reviewed. In Chapter 3, the real-time digital image transformation used for performance comparison is described. Chapter 4 describes the implementation of the image transformation programs for each parallel framework. In Chapter 5, the execution time of the program developed for each parallel framework is measured and compared. Also, the execution time ratio constant *K* is presented. Chapter 6 summarizes and concludes this paper.

2. Related Work

A number of studies have been conducted to evaluate the performance of multicore CPUs and GPUs. First, the performance of multicore CPUs was analyzed and the parallel programming optimization was carried out. Research by Lee and Kang measured the performance of the parallel program according to the usage environment of shared resources among the CPU cores and studied the optimization method [1]. Research by Roh et al. analyzed the effect of high speed memory size, number of cores, number of threads and MPI processes included in the CPU on the performance of parallel programs in the many-core system [2].

Second, studies were conducted to evaluate and optimize the performance of multicore GPUs. There have been various studies to evaluate the performance of parallel programs in Nvidia's CUDA system, and to suggest optimization techniques for CUDA programming [3-5].

These studies, after selecting a specific parallel programming framework, are aiming to improve the performance of parallel programs in this environment.

Third, there are studies on performance comparison between parallel programming frameworks. Research by S. Hua compared and analyzed the parallel computing ability between OpenMP and MPI (Message Passing Interface) [6]. And it provided some proposals in parallel programming. Research by Karimi et al. compared the performance of CUDA and OpenCL using complex, near-identical kernels [7].

In this study, we select four typical parallel programming frameworks - OpenMP, PPL, OpenCL, and CUDA - that are widely used in multicore CPUs and GPUs, and compare parallel program performance among these frameworks.

3. Digital Image Transformation

If the image displayed on a computer monitor is projected onto an external screen using a projector, the image may be distorted depending on the projection angle and the screen state. For trapezoidal distortion, we can use the keystone function in the projector settings to compensate. As shown in Figure 1, if the projected

screen surface is not flat, the image distortion occurs in a curved shape and cannot be compensated by the keystone function alone.

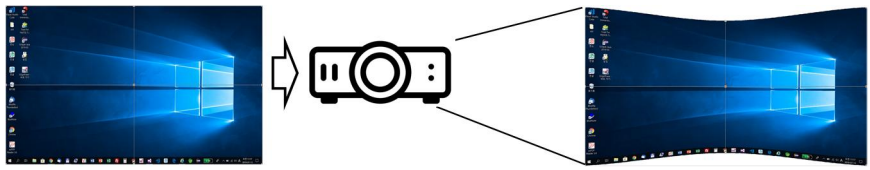

Figure 1. Curved image distortion in projection.

In order to correct the curve-shaped image distortion by software, it is necessary to capture the image of the computer monitor in real time, convert the shape of the image according to the correction information, and project the image on an external screen. As shown in Figure 2.

Figure 2. Projection after image distortion correction.

Parallel computing power of multicore CPUs and GPUs is required for distortion correction of projected image in real time. In this study, the performance of digital image transformation program is measured for each parallel programming framework, and compared and analyzed.

4. Implementation

In order to compare the performance of parallel programming frameworks, we have developed a program that (a) captures the computer screen in real time, (b) corrects the distortion of the captured image, and (c) outputs it to the projector. The development and execution environment of this program is as follows.

- Software
  - Microsoft Windows 10
  - Visual Studio 2017, x64
  - Language: C++/ C
- Hardware
  - CPU: Intel Skylake I7 (3.6GHz, 4 cores)
  - Graphic card: Nvidia GeForce GTX 960 (CUDA Core # 1024, 1164MHz, Memory Speed 7.01Gbps)

In this program, the part of "(b) corrects the distortion of the captured image" was developed as a separate version for each parallel program framework.

4.1 Single Core

The image distortion correction was limited to the vertical direction. That is, the pixels of the image are constant in width but can vary in height. In order to compensate for this, information (*mappingTable*) for

expressing different heights for each pixel of the image is stored, and the target image (*targetBuffer*) to be outputted to the projector is generated by correcting the original image (*srcBuffer*) in real time.

The code used for the correction is shown in Figure 3. The *getBlendedPixel* function computes the pixel value of the (x, y) coordinates of the distortion-corrected image using the *mappingTable*.

```
void MappingMgr::map(char *srcBuffer, char *targetBuffer, float *mappingTable) {
    for (int col = 0; col < _imageWidth; ++col) {
        uint32_t *tptr = ((uint32_t *)targetBuffer) + col;
        for (int row = 0; row < _imageHeight; ++row, tptr += _imageWidth)
            *tptr = getBlendedPixel(srcBuffer, mappingTable, col, row);
    }
}
```

Figure 3. Code for image distortion correction.

4.2 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran [8]. Visual Studio 2017 used in the experiment supports OpenMP 2.0.

It is very easy to convert sequential source code into parallelized source code with OpenMP. Just put the "#pragma omp parallel for" directive statement before the for-loop. Figure 4 shows the image distortion correction code written in parallel with OpenMP.

```
void MappingMgr::map(char *srcBuffer, char *targetBuffer, float *mappingTable) {
#pragma omp parallel for
    for (int col = 0; col < _imageWidth; ++col) {
        uint32_t *tptr = ((uint32_t *)targetBuffer) + col;
        for (int row = 0; row < _imageHeight; ++row, tptr += _imageWidth)
            *tptr = getBlendedPixel(srcBuffer, mappingTable, col, row);
    }
}
```

Figure 4. Image distortion correction code with OpenMP.

4.3 PPL

The PPL (Parallel Patterns Library) is a Microsoft C++ library that provides features for multicore programming. It was first bundled with Visual Studio 2010. The sequential loop can be made into a parallel loop by replacing the *for* with a *parallel_for* as shown in Figure 5.

```
void MappingMgr::map(char *srcBuffer, char *targetBuffer, float *mappingTable) {
    Concurrency::parallel_for(0, _imageWidth-1, [&](int col) {
        uint32_t *tptr = ((uint32_t *)targetBuffer) + col;
        for (int row = 0; row < _imageHeight; ++row, tptr += _imageWidth)
            *tptr = getBlendedPixel(srcBuffer, mappingTable, col, row);
    });
}
```

Figure 5. Image distortion correction code with PPL.

4.4 OpenCL

OpenCL (Open Computing Language) is a framework for writing parallel computing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors or hardware accelerators [9]. In this paper, we wrote the OpenCL program to use a multicore GPU. The graphics card used in the experiment is Nvidia GeForce GTX 960, and the OpenCL v1.2 library is included in the CUDA toolkit v10.1 which supports the graphics card.

The OpenCL program consists of a host program running on a CPU and a kernel program running on a device (graphics card). In order to correct the image distortion, the original image is transferred to the device memory, and the kernel program is operated in parallel for each image pixel to generate the corrected image. The corrected image is transmitted to the host memory again.

4.5 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by Nvidia that makes using a GPU for general purpose computing simple and elegant [10]. In this experiment, CUDA toolkit v10.1 was used.

Similar to the OpenCL program, the CUDA program consists of a host program running on a CPU and a kernel program running on a device. In this experiment, we applied various CUDA optimization techniques introduced in the book by Jung [11] to maximize the speed of image distortion correction.

5. Performance Comparison

Under the same conditions, the execution time of the image distortion correction function was measured by executing five versions of the program 100 times each. The results are shown in Table 1 and Figure 6.

Table 1. Execution time of parallel programming frameworks.

Size of image	Single core		OpenMP		PPL		OpenCL		CUDA	
	avg. time	stdev	avg. time	stdev	avg. time	stdev	avg. time	stdev	avg. time	stdev
800x600	5.8104	0.0960	1.6282	0.1606	1.5812	0.0459	0.9929	0.0629	0.5710	0.0277
1024x768	10.5784	0.0562	3.0214	0.3740	2.8722	0.0704	1.5705	0.0708	0.8310	0.0248
1920x1080	28.7554	0.4226	7.7859	1.5000	7.1788	0.2468	3.8669	0.1242	2.0474	0.0251

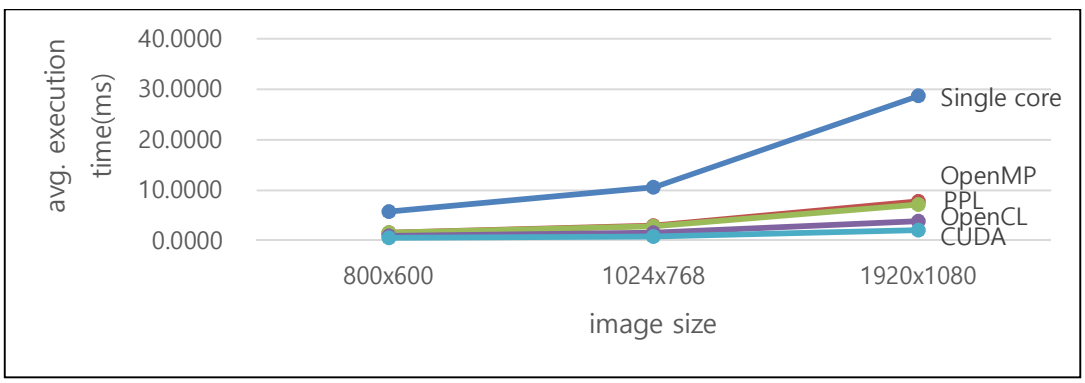

Figure 6. Performance comparison of parallel programming frameworks.

5.1 Single Core vs. Others

The sequential image distortion correction function using a single CPU core took an average of 28.76ms to process an image of 1920x1080 size. On the other hand, OpenMP took 7.79ms, PPL 7.18ms, OpenCL 3.87ms, and CUDA 2.05ms. OpenMP and PPL programs using multicore CPU were 3.69 times and 4.01 times faster than single core program, respectively. Considering that the number of cores of the CPU in the experimental environment is 4, it can be seen that OpenMP and PPL almost completely used the performance of 4 multicore in the image correction process. OpenCL and CUDA using multicore GPU improved 7.44 times and 14.04 times, respectively, compared to the single CPU core program. As the data to be processed becomes smaller, the performance improvement rate through parallelization is somewhat reduced.

However, OpenMP, PPL, OpenCL, and CUDA programs show significantly improved performance compared to the sequential processing program.

5.2 OpenMP vs. PPL

The programs in both frameworks run on the same 4 cores CPU, but the program written in PPL is 8.5% faster than the OpenMP program. Also, for the standard deviation of execution time, the value of PPL program was smaller than the value of OpenMP. This shows that the execution time of the PPL program is more uniform.

Experimental results show that when developing parallel programs on multicore CPUs in Visual Studio C++, PPL is better than OpenMP in terms of execution time and deviation.

5.3 OpenCL vs. CUDA

Although the two framework programs run on the same multicore GPU, the CUDA program has 1.89 times better performance than the OpenCL program. One of the main reasons for performance differences is that the CUDA program use ‘streams’ to optimize code, but the OpenCL program do not. According to [7], CUDA programs are reported to be 1.13 times faster than OpenCL programs.

5.4 Multicore CPU vs. GPU

Comparing the performance of PPL program that shows good performance with multicore CPU to that of CUDA program optimized for multicore GPU, the CUDA program is 3.51 times faster than the PPL program. Direct performance comparison is unreasonable because the performance of parallel programs depends on the type of CPU and GPU.

In this paper, we compare the performance based on the clock speed of both devices. The CPU used in the experiments consists of 4 cores operating at 3.6GHz, so that the total clock speed is 14.4GHz. The GPU used in the experiment has 1024 cores operating at 1.164 GHz. Therefore, the total clock speed is 1191.9GHz. With the total clock hertz ratio and execution time ratio of the two devices, the execution time ratio constant K is obtained as follows.

- Execution-time-ratio = (Execution-time-in-PPL / Execution-time-in-CUDA)
- Total-clock-hertz-ratio = (Total-GPU-cores-Hz / Total-CPU-cores-Hz)
- K = (Execution-time-ratio) / (Total-clock-hertz-ratio) = 3.51 / 82.77 = 0.04236

$K_{(I7,PPL)\ to\ (GTX960,CUDA)}$ represents the approximate execution time ratio of PPL program (running on Intel I7 CPU) and CUDA program (running on Nvidia GTX960 GPU) based on clock.

Assuming that the execution time of a PPL program operating at a total clock X GHz is A, The expected execution time B, when developed with a CUDA program operating at a total Y GHz, can be calculated as follows. (The execution time ratio constant for both execution environments is Ka)

- $B = (A \times X)/(Ka \times Y)$

The execution time ratio constant K may vary depending on the CPU and the GPU, and may vary depending on the structure of programs, the degree of optimization, and the amount of data. In this experiment, when the image size is 800x600, K value is calculated as 0.03346. The execution time rate constant K between single core and CUDA is 0.04242, similar to the K value between PPL and CUDA.

6. Conclusion

In this paper, we developed digital image transformation programs that runs on each of the major parallel programming frameworks - OpenMP, PPL, OpenCL, and CUDA - respectively, and compared the performance of the programs.

OpenMP, PPL, OpenCL, and CUDA programs show significantly improved performance compared to the sequential processing program. Experimental results show that when developing parallel programs on multicore CPUs in Visual Studio C++, PPL is better than OpenMP in terms of execution time and deviation. Also CUDA programs performed better than OpenCL programs.

A constant K indicating the ratio of program execution time between different parallel computing environments is presented. Using this, it is possible to predict rough execution time without implementing a parallel program.

Acknowledgement

This Research was supported by Seokyeong University in 2019.

References

[1] M.H. Lee and J.S. Kang, “Performance Analysis and Characterization of Multi-Core Servers, ” Korea Information Processing Society Review A Vol.15-A, No.5, pp. 259-268, Korea Information Processing Society, 2008. DOI: https://doi.org/10.3745/kipsta.2008.15-a.5.259.

[2] S.W. Roh, J.E. Choi, D.N. Nam, G.C. Park, and C.Y. Park, “An Analysis Tool and Benchmark Performance Experiment for Optimizing Parallel Programming in Mani-Core System,” Korea Information Processing Society Review Vol.25, No.1, pp. 78-88, Korea Information Processing Society, 2018.

[3] S.S. Kim, D.H. Kim, S.K. Woo, and I.S. Ihm, “Analysis of Programming Techniques for Creating Optimized CUDA software,” Journal of KIISE : Computing Practices and Letters 16(7), pp. 775-787, Korea Information Science Society, 2017.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron, “A Performance Study of General-Purpose Applicaions on Graphics Processors Using CUDA ,” Journal of Parallel and Distributed Computing, University of Virginia, 2008. DOI: https://doi.org/10.1016/j.jpdc.2008.05.014.

[5] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu, “Optimization Principles and Application Performance Evaluation of a Multi-threaded GPU Using CUDA,” Proc. 13th ACM SIG-PLAN Symp. Principles and Practice of Parallel Programming, ACM Press, 2008. DOI: https://doi.org/10.1145/1345206.1345220.

[6] S. Hua, “Comparison and Analysis of Parallel Computing Performance Using OpenMP and MPI,” The Open Automation and Control Systems Journal vol. 5, pp. 38-44, 2013. DOI: https://doi.org/10.2174/1874444301305010038.

[7] K. Karimi, N.G. Dickson, and F. Hamze, “A Performance Comparison of CUDA and OpenCL,” Computing Research Repository – CORR, arXiv:1005.2581, 2010.

[8] OpenMP Architecture Review Board, OpenMP Application Program Interface, *http://openmp.org/forum/*

[9] O. Takashi, OpenCL NYUMON-GPU & Multi-core CPU heiretus Programming, HongRung Publishing, 2012.

[10] M. Ebersole, What Is CUDA?, *https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.*

[11] Y.H. Jung, CUDA Parallel Programming, FreeLec Publishing, 2011.