COMPARING PROGRAMMER PRODUCTIVITY IN OPENACC AND CUDA: AN EMPIRICAL INVESTIGATION

Xuechao Li[1], Po-Chou Shih[2], Jeffrey Overbey[1], Cheryl Seals[1], Alvin Lim[1]

[1]Department of Computer Science, Auburn University, AL, USA
[2]Graduate Institute of Industrial and Business Management, National Taipei University of Technology, Taipei, Taiwan

ABSTRACT

OpenACC has been touted as a "high productivity" API designed to make GPGPU programming accessible to scientific programmers, but to date, no studies have attempted to verify this quantitatively. In this paper, we conduct an empirical investigation of program productivity comparisons between OpenACC and CUDA in the programming time, the execution time and the analysis of independence of OpenACC model in high performance problems. Our results show that, for our programs and our subject pool, this claim is true. We created two assignments called Machine Problem 3(MP3) and Machine Problem 4(MP4) in the classroom environment and instrumented the WebCode website developed by ourselves to record details of students' coding process. Three hypotheses were supported by the statistical data: for the same parallelizable problem, (1) the OpenACC programming time is at least 37% shorter than CUDA; (2) the CUDA running speed is 9x faster than OpenACC; (3) the OpenACC development work is not significantly affected by previous CUDA experience

KEYWORDS

OpenACC, CUDA, empirical investigation, programming time, programmer productivity

1. INTRODUCTION

High-performance computing has been applied to more and more scientific research applications such as chemical reaction processes, explosion processes and fluid dynamic processes. When intensive computation needs to be applied to other research fields, other than image processing, the GPGPU [9] (General Purpose computing on GPU) was developed. Currently, the most popular high-performance computing device is the NVIDIA accelerator so CUDA (Compute Unified Device Architecture) [10], a mainstream parallel programming model, is widely used by programmers, developers and researchers to rapidly solve large problems with substantial inherent parallelism [11]. However, the CUDA programming framework requires programmers to fully grasp solid knowledge of software and hardware. It is an error-prone and time-consuming task if legacy code needs to be rewritten by CUDA. Furthermore, if these applications need to run on other platforms (i.e. ATI GPU, Intel GPU), the software developers need to learn new APIs and languages to rewrite all code. To overcome the above limitations of CUDA and reduce developers' coding workload, the OpenACC programming model has been released. It provides simple directives to achieve similar functions in CUDA. Also, the OpenACC codes can be run under different platforms. It is important to compare OpenACC and CUDA based on the programming productivities especially for parallel programming beginners because OpenACC can extremely reduce programmers' workload.

DOI : 10.5121/ijcsea.2016.6501

The time it takes to develop parallel solutions to the problems is one of the most important metrics in evaluating high performance APIs. To mentally and physically alleviate programmers' workload and reduce the need of the amount of developers' parallel knowledge, thousands of experts, engineers, and researchers are still using on OpenACC API although it has been released in 2011. So during the comparison of OpenACC and CUDA, we mainly investigate the question about whether the OpenACC model is more efficient than CUDA. Three research questions are proposed in this paper: (1) Is OpenACC programming time shorter than CUDA for the same parallelizable problem? (2) Is CUDA execution time shorter than OpenACC execution time? (3) Is OpenACC programming time affected by previous CUDA experience in the same problem?

We conduct this empirical investigation between CUDA and OpenACC mainly considering the following two factors: (1) CUDA is one of the most popular parallel programming models and OpenACC is an easily learned and simplified high-level parallel language, especially for parallel programming beginners; (2) One motivation for OpenACC is to simplify low-level parallel language such as CUDA.

Although other scientists did some research work on the performance comparison on OpenACC and CUDA in [22] [23] [24] [25], their work was based on the code, optimization technologies and the hardware platform. To our best knowledge, this is the first paper in evaluating productivity based on human subjects. Hence, we present the following three key contributions on this empirical investigation:

(1) This paper investigates whether OpenACC programming time is shorter than CUDA for the same parallelization problem. A WebCode website is used to automatically record programmers' work time so that a quantitative comparison can be analyzed statistically.

(2) Although high level OpenACC provides simple directives to replace CUDA functions, the performance comparison also needs to be investigated because the fact that OpenACC users take less time to code might be offset by poorer execution time performance.

(3) We investigate whether OpenACC programming work is affected by previous CUDA experience, whereby a programmer is required to accelerate the same section of serial code first with CUDA and then use OpenACC to do it again.

The remainder of this paper is organized as follows. Section 2 presents the background of CUDA and OpenACC. Section 3 discusses related work on empirical studies of the productivity in high performance languages and performance comparisons of diverse parallel programming models. Section 4 shows our target population description, data collection procedure and data analysis procedure. Section 5 presents an overall result analysis and makes conclusions about our three hypotheses about CUDA and OpenACC. Section 6 concludes this paper and presents our future work. Section 7 shows our acknowledgement.

2. BACKGROUND

The goal of high-performance parallel languages is to accelerate running time and to reduce the unnecessary waiting time of computing operations. The fully controllable parallel language CUDA is the most representative one among high performance parallel languages because it allows users to schedule resources by utilizing optimization technologies such as kernel parameters tuning, the use of local memory, data layout in memory, and CPU-GPU data transfer avoidance. However, this "controllability" advantage requires users to grasp solid hardware knowledge and debugging skills. To overcome the limitations above, OpenACC provides simple directives to accelerate code

without requiring much hardware knowledge. In this section, we simply introduce two parallel languages investigated in this paper: CUDA and OpenACC.

CUDA is a parallel computing programming model that can be run only under NVIDIA's GPUs. It fully utilizes hardware architecture and software algorithms to accelerate high-performance computation with architecture-specific directives such as shared, or global. But this utilization requires programmers to fully understand each detail of hardware and software. For example, in order to utilize memory-coalescing or tiling technology, the software-managed on-chip and off-chip memory knowledge should be understood by users before programming, which is a big challenge, especially for novices who have little knowledge of software and hardware. Also, in order to successfully launch CUDA kernels, all configurations, such as Grid/Block dimensions, computing behaviors of each thread, and synchronization problems need to carefully be tuned.

OpenACC is a cross-platform programming model and a new specification for compiler directives that allow for annotation of the compute region and data region that are offloaded to the accelerators [12] such as GPUs. The OpenACC Application Programming Interface [19] offers a directive-based approach for describing how to manage data and to execute sections of codes on the device, where parallelism often appears in regular repetition constructs such as Fortran "DO" loops and C "FOR" loops. In OpenACC, porting of legacy CPU-base code only requires programmers to add several lines of annotations before the sections where they need to be accelerated, without changing code structures [12]. However, over-simplified parallel programming model also brings to users limitations which may prevent full use of the available architectural resources, potentially resulting in greatly reduced performance when compared to highly manually tuned CUDA code [12]. For example, programmers can use synchronization to manage all threads of one block in CUDA while they cannot in OpenACC. Some architecture-specific directives such as global, shared and private have been provided by CUDA while OpenACC does not provide them.

3. RELATED WORK

NVIDIA GPU devices are currently one of the most widely used accelerators so the CUDA programming language is used by most programmers. However, it requires professional software and hardware knowledge. To alleviate or avoid these limitations above, especially for beginners, OpenACC was released. One of the OpenACC motivations is to reduce programmers' work time or workload in parallelizing serial code. On the other hand, several empirical studies of the productivity in high performance computing have been done. In this section we describe some of the classic work.

A case study on the productivity of novice parallel programmers was conducted in [1]. A metric of time to solution was proposed, which is comprised of two main components: the human effort required for developing the software and the amount of machine time required to execute it. The results showed that LOC (Line of Code) alone is not a good metric for effort and code expansion rates highlight differences in the strategies imposed by different HPC approaches.

The evaluation of development effort for high performance computing has been described and developed in [2]. In order to measure the development time, authors used two APIs (MPI and OpenMP) to collect the data of development time. The outputs of this paper are a set of experimental lessons learned.

In the area of productivity measurement on high performance computers, Marvin et al. [3] discussed the problems of defining and measuring productivity and developed a model based on program size, speedup and development cost, and a formula of measuring the productivity.

Although this work can quantitatively measure the productivity, the author mentioned that the major impact of this work is the realization that productivity may have different meanings in different application domains.

Lorin et al. [17] measured the programming effort with a combination of self-reported and automatic data. The students were required to keep an effort log to report how much time they spent in different activities. For automatic collection data, authors instrumented the compiler so that the timestamp was recorded and a copy of the submitted source code was captured. Through this pilot study, some lessons were learned such as the work time varies considerably across subjects and underreporting and over reporting are significant issues.

Besides, some research studies also worked on the performance comparison of high performance parallel language such as CUDA, OpenACC, and OpenCL. Tetsuya et al. [4] used two micro benchmarks and one real-world application to compare CUDA and OpenACC. The performance was compared among four different compilers: PGI, Cray, HMPP and CUDA with different optimization technologies, i.e. (1) thread mapping and shared memory were applied in Matrix Multiplication micro benchmark; (2) branch hoisting and register blocking were applied into 7-Point stencil micro benchmark. Finally a performance relative to the CPU performance was compared in computational fluid dynamics application with kernel specification, loop fusion, and fine-grained parallelization in the matrix-free Gauss-Seidel method optimization. The results showed that OpenACC performance is 98% of CUDA's with careful manual optimization, but in general it is slower than CUDA.

Kei *et al* [20] presented an optimizer for a histogram computation based on OpenACC model. For the problem of rewriting multiple copies of histograms with OpenACC code, this optimizer can automatically detect and rewrite the blocks of multiple copies of histograms. In the experiment phase, author employed three applications to evaluate the performance of the optimizer under three parallel platforms: AMD, NVIDIA, Intel. The results showed the achieved speedup was from 0.7x to 3.6x compared to the naive method.

In [5], the authors evaluated OpenACC, OpenCL, and CUDA based on programmer productivity, performance, and portability using Lagrangian-Eulerian explicit hydrodynamics mini-application. Through the comparison of kernel times, words of code (WOC), and portability analysis, a conclusion was made that OpenACC is an viable programming model for accelerator devices, improving programmer productivity and achieving better performance than OpenCL and CUDA.
In [21] the OpenACC performance portability was evaluated with four kernels from Rodinia benchmark. The results show the optimized OpenACC code outperforms OpenCL code in the performance portability ratio.

An overview and comparison of OpenCL and CUDA was also presented in [6]. Five application benchmarks—Sobel filter, Gaussian filter, Median filter, Motion Estimation, and Disparity Estimation—were tested under NVIDIA Quadro 4000. This paper compared and analyzed C, OpenCL, and CUDA and the two kinds of APIs—CUDA runtime API and CUDA driver API. Detailed data comparison generated by different benchmarks generally showed that CUDA had a better performance compared with OpenCL.

Christgau et al. [18] presented a very interesting application--Tsunami Simulation EasyWave—for comparing CUDA and OpenACC using two different GPU generations (NVIDIA Tesla and Fermi). Through listing runtime values under different hardware-specific optimizations—memory alignment, call by value and shared memory, three key points were revealed: (1) even the most tuned code on Tesla does not reach the performance of the unoptimized code on the Fermi GPU;

(2) the platform independent approach does not reach the speed of the native CUDA code; (3) memory access patterns have a critical impact on the compute kernel's performance.

4. INVESTIGATION DESIGN

We empirically conducted the investigation with undergraduate-level and graduate-level students at Auburn University during the Spring semester, 2016. The goal of this research is to investigate that in terms of the programming time, which API is more efficient in parallel programming work between OpenACC and CUDA. Besides, the following hypotheses were also investigated: (1) the execution time comparison (2) whether OpenACC programming work is subjectively affected by previous CUDA experience.

Three considerations of our empirical investigation are described in the following: (1) in order to simply parallelize programming work especially for novices, OpenACC provides high-level parallel directives which implement similar CUDA functions; (2) OpenACC simplifies the CUDA programming work, but this kind of simplification might be offset by poorer performance because OpenACC directives need to be converted into CUDA code before they can be executed; (3) the parallel work with OpenACC is based on serial code such as C/C++/Fortran, so the parallel style highly depends on the original coding structures and parallel directives selection. We propose the following three hypotheses:

H1: The programming time in OpenACC is shorter than CUDA.
H2: The execution time of the OpenACC solutions is longer than the execution time of the CUDA solutions.
H3: The OpenACC programming time cannot be affected by CUDA language if OpenACC implementation is followed by CUDA implementation in the same problem.

Table 1. The participants background

Background	**Education Level**		**Years of programming experiencing (any language)**			**Experience in CUDA and OpenACC**		
	under-graduate	graduate	<5 yrs	5-10 yrs	>10 yrs	novice*	can write simple code	skilled
Population	14	14	15	11	2	3	25	0

*novice: little/no experience

In order to obtain students' programming time, we created four Machine Problems(MPs) in this empirical experiment: MP1, MP2, MP3 and MP4. Before the MP3 assignment distribution, our WebCod website was still on the progress, which mainly recorded students' coding behaviors including timestamps. So the data from MP3 and MP4 were captured but the results from MP1 and MP2 cannot be obtained. Details of valid Machine Problems are described in section 4.3. Each student was required to finish each assignment with CUDA and OpenACC. For each project students are divided into two groups: in group 1 participants were required to finish the project with CUDA firstly and do it again with OpenACC while in group 2 participants were required to parallelize the project with OpenACC firstly and then do it again with CUDA. In this way we can investigate whether the OpenACC programming work is affected by previous CUDA experience.

4.1 TARGET POPULATION

The target population for this investigation is comprised of students in the computer science and software engineering department enrolled in COMP 7930/4960, GPU Programming, a one-hour,

semester-long course covering the basics of CUDA, OpenACC, and parallel algorithm design. In total 28 students are involved in our empirical investigation. Table 1 describes the participants' academic backgrounds. We can see that although 2 students have more than 10 years of programming experience (what language is unknown) in this empirical investigation, they are still novices in learning OpenACC and CUDA. Their data do not affect our conclusions in OpenACC and CUDA.

From the perspective of years of programming experience, 53.6% participants (15 out of 28) have less than 5 years of programming experience and the programming experience of 39.2% participants (11 out of 28) is greater than 5 years but less than 10 years. So the experimental results are in a fair condition.

For specific experience in our target programming languages: OpenACC and CUDA, there are only 3 novices (little or no experience) and the rest of students can write some simple OpenACC or CUDA code. So the target population is in a relatively "balanced" environment.
Finally, the number of undergraduate students is equal to the number of graduate students in educational level (i.e.14 undergraduate students and 14 graduate students). All of the students met the entrance criteria of not being experts in parallel programming and are all novices in this area. The difference in their educational level (graduate vs undergraduate student) did not significantly affect our conclusions.

4.2 DATA COLLECTION PROCEDURE

For data collection purposes, we develop the WebCode website (the source code is available at https://github.com/joverbey/webcode) to automatically record programmers' coding behaviors including programming timestamps. The participants were required to parallelize two projects (MP3 and MP4) on WebCode so that programming information can be stored in the database.

Figure 1. A snapshot of our WebCode website

As Figure 1 illustrates, students were allowed to write the code in the main window and all compilation information was displayed in the lower window so that students can easily debug their code as they do the coding work in other compiling environment such as Ubuntu. The only difference of coding on this website from other environments was that all coding behaviors with timestamps are stored in the database so that the data can be collected and analyzed for our empirical study.

To precisely measure users' programming time of each project under either OpenACC or CUDA, we use the "Project ID" primary key as a keyword to track the programming time of the same project among different tables in our database. Because we did not require students to code the projects in a particular time range and specific work place such as classrooms, most of students did their coding work intermittently instead of continuously. This situation brings us a challenge in accurately counting programming time because once the coding work pauses, there are two possibilities: (1) the programmer may be thinking about coding questions, so the coding work is in progress; (2) the programmer would do something else instead of coding. After reviewing the data in the database, we set up a time threshold of 10 minutes to properly measure the programming time. That is, if the pausing interval of coding behaviors is less than 10 minutes, we assume that this programmer was thinking and still worked on the coding so the pausing time was counted into the programming time. Otherwise if the pausing interval is greater than 10 minutes, we assume that this programmer would not continue to code so this pausing time was not counted into the programming time.

In order to collect participants' academic background, we distributed a programming experience questionnaire to each student in the classroom environment and they voluntarily returned it after filling it out.

4.3 Projects Description

In order to truly estimate students' programming efficiency in OpenACC and CUDA, we did not intentionally select hard or easy projects to be the test samples. In this way the impact from the difficulty level of test samples on efficiency can be avoided. The nature of two Machine Problems(MPs) in this experiment is described in the following way:

MP3: A program simulates the heat transfer across a long, thin rod. Initially the temperature is 0 at every point of a rod. When the left-hand side of a rod is heated, the temperature at each point will raise, which can be calculated by

$$T_{new}[i] = t_{old}[i] + \alpha(t_{old}[i-1] + t_{old}[i+1] - 2t_{old}[i])$$

Where $t_{old}[i]$ denotes the temperature of the i-th points at a particular unit of time and α is a constant (the "thermal diffusivity").

The key part of serial code is showed in Figure 2 and the complexity is $O(n^2)$.

```
// Compute temperatures at each sample point in the rod over time
int time;
for (time = 0; time < MAX_TIMESTEPS; time++) {
    new_t[0] = old_t[0];
    for (int i = 1; i < N-1; i++) {
        new_t[i] = old_t[i] + ALPHA*(old_t[i-1] + old_t[i+1] - 2*old_t[i]);
    }
    new_t[N-1] = old_t[N-1];

    // Swap old and new buffers for next iteration (double-buffering)
    temp = old_t;
    old_t = new_t;
    new_t = temp;
}
```

Figure 2. The key part of MP3 serial code

MP4: To encrypt and decrypt the messages, both the sender and the intended recipient need to a secret key. A brute-force key search was used to figure out this key with Tiny Encryption Algorithm (TEA). Instead of 128-bit key, 28-bit key was used in this key search. The key part of serial code is showed in Figure 3 and the complexity is $O(n^2)$.

```
void encrypt(uint32_t *data, const uint32_t *key) {
    uint32_t v0=data[0], v1=data[1], sum=0, i;           /* set up */
    uint32_t delta=0x9e3779b9;                           /* a key schedule constant */
    uint32_t k0=key[0], k1=key[1], k2=key[2], k3=key[3];  /* cache key */
    for (i=0; i < 32; i++) {                             /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                                    /* end cycle */
    data[0]=v0; data[1]=v1;
}
 /* Try every possible 28-bit integer... */
    for (uint32_t k = 0; k <= 0x0FFFFFFF; k++) {
        /* Try encrypting the data with the key { k, k, k, k } */
        data[0] = orig_data[0];
        data[1] = orig_data[1];
        key[0] = key[1] = key[2] = key[3] = k;
        encrypt(data, key);
        /* Did we get the correct encrypted values? */
        if (data[0] == encrypted[0] && data[1] == encrypted[1]) {
            break;
        }
    }
```

Figure 3. The key part of MP3 serial code

4.4 Analysis Procedure

We use paired sample t-test [7] to analyze results of programming time comparison in order to see whether the OpenACC coding time is significantly less than CUDA. Similarly, in the comparison of the execution time between OpenACC and CUDA, we still use paired sample t-test to make our conclusion. To investigate whether OpenACC programming time is subjectively affected by previous CUDA experience, the independent t-test[7] was used in our third hypothesis because

whether the equal relationship of OpenACC programming time between two groups in each project was investigated. We will list reasons to select proper statistical index to analyze data in different hypotheses. Finally, in all analyses above, we use a significant level of 0.05 to make conclusions.

5. HYPOTHESES INVESTIGATED

The data collected from two projects (MP3 and MP4) give us a way to address issues about what are the differences between two API models (CUDA and OpenACC) in programming time, OpenACC model independence and the dispersion of the programming time. In total 28 students in the computer science and software engineering department of Auburn University participated in this empirical study. Because we only use the data from students who correctly solve projects with both OpenACC and CUDA, the valid population is, respectively, 18 in MP3 and 16 in MP4. For example, some students correctly solved MP3 with OpenACC but failed it with CUDA or some students solved MP3 with both OpenACC and CUDA, but the results are not correct or partially correct. This kind of data is not counted in our valid population.

In order to quantitatively compare OpenACC and CUDA to investigate whether our hypotheses were supported, different statistical methods were used to analyze data. Because the proper selection of statistical index highly depends on problem attributes, we specifically describe them in the following way. In section5.1 each student correctly parallelizes the same problem with both OpenACC and CUDA, so we compared the programming time of OpenACC to the programming time of CUDA and analyzed the data with paired t-test to see whether or not there is statistically a significant difference in programming time users spend on their tasks. In section5.2 in order to investigate whether the fact that OpenACC takes less time to parallelize the serial code might be offset by the poorer performance, we compared the execution time of OpenACC and CUDA using paired sample t-test. In section5.3 the investigated question is OpenACC programming time is affected if CUDA are already used to solve the same problem and statistically we examine whether OpenACC programming time in group1 is equal to one in group2 so the independent t-test[7] was selected. We use two scenarios to illustrate our second hypothesis:

S1: users first parallelize serial code with CUDA and then do it again with OpenACC
S2: users first parallelize serial code with OpenACC and then do it again with CUDA

5.1. PROGRAMMING TIME

The programming time or efficiency of parallel APIs is one of the most important factors when users choose APIs to parallelize serial code. The CUDA requires programmers to understand details of software and hardware such as memory management to accelerate codes. Therefore, it is an error-prone and time-consuming work. To overcome those limitations, OpenACC provides simple directives to achieve similar functions in CUDA with less knowledge and effort. So the programmers' programming time and workload can be extremely reduced. To our best knowledge, this is the first paper to investigate empirically our first hypothesis:

H1: The programming time in OpenACC is shorter than CUDA

To evaluate this hypothesis, the comparison of programming time is abstracted in a statistical way.

$$\begin{cases} H_0: \mu_{OpenACC} \geq \mu_{CUDA} \\ H_1: \mu_{OpenACC} < \mu_{CUDA} \end{cases}$$

We use $\mu_{OpenACC}$ to denote the mean programing time in OpenACC and μ_{CUDA} to denote the mean programing time in CUDA.

Table 2. The programming time data

Project	API	Programming time (hours)	P-value (one-tail)
MP3	CUDA	Mean 3.622, Std. 1.905, n=18	0.0465
	OpenACC	Mean 2.597, Std. 1.816, n=18	
MP4	CUDA	Mean 3.640, Std.1.782, n=16	0.0008
	OpenACC	Mean 2.302, Std.1.539, n=16	

The data in table 2 have supported our hypothesis *H1*. In MP3 the mean programming time in OpenACC and CUDA, respectively, is 2.6 hours and 3.6 hours. Through the one-tail p-value of 0.0465, we can see OpenACC programming time is significantly shorter than one in CUDA. Similarly, in MP4 on average, the students spent 2.3 hours on the parallelization work with OpenACC while 3.6 hours was spent with CUDA. The one-tail p-value is 0.0008, which means the OpenACC programming time is also significantly shorter than CUDA. The data above show us that in order to parallelize serial code, CUDA users generally spend 1.4x longer than OpenACC users. Therefore, we make a conclusion: The programming time in OpenACC is shorter than CUDA.

5.2 EXECUTION TIME

Through the analysis in section 5.1, the OpenACC programming efficiency is significantly higher than CUDA efficiency. But the fact that less time on OpenACC programming might bring the problem of the poorer performance, in terms of the execution time, also needs to be investigated. On the other hand, the execution time of parallel models is an important performance evaluation index so in this human-subject experiment toward novices we also investigate which API is faster in the acceleration.

OpenACC is a high level compiler-directive parallel model, but the conversion process from directives to CUDA codes may consume more time compared to the low level CUDA. Based on this consideration, we proposed the second hypotheses.

H2: The execution time in OpenACC is longer than the execution time in CUDA.
The statistical formula is described in the following for these hypotheses.

$$\begin{cases} H_0: E_{OpenACC} \geq E_{CUDA} \\ H_1: E_{OpenACC} < E_{CUDA} \end{cases}$$

We use $E_{OpenACC}$ to denote the execution time of OpenACC solutions and E_{CUDA} to denote the execution time of CUDA solutions. The hypotheses H_0 assumes that the execution time of OpenACC solutions is equal to or greater than the execution time of CUDA solutions while The hypotheses H_1 assumes that the execution time of OpenACC solutions is less than the execution time of CUDA solutions.

Table 3. The execution time comparison of parallel solutions

Project	API	Execution time (hours)	P-value (one-tail)
MP3	CUDA	Mean 0.2158, Std. 0.0863, n=18	3.38E-11

	OpenACC	Mean 1.987, Std. 0.5064, n=18	
MP4	CUDA	Mean 0.1206, Std. 0.0987, n=16	1.59E-09
	OpenACC	Mean 30.40, Std. 9.90, n=16	

The data in Table 3 showed that in MP3 the mean execution time (0.2158) in CUDA is significantly shorter than OpenACC one (1.987) because one-tail p-value (3.38E-11) is less than 0.05. Similarly, in MP4 the mean execution time (0.1206) in CUDA is significantly shorter than OpenACC one (30.40) because one-tail p-value (1.59E-09) is less than 0.05. Hence we make the conclusion that the execution time in OpenACC is longer than the execution time in CUDA.

5.3 INDEPENDENCE OF OPENACC

Because OpenACC is a high-level directive-based API, programmers need to add proper directives or annotations in original serial code such as C, C++, Fortune while CUDA requires users to rewrite the original code. Considering these different coding behaviors above, we need to know whether or not the programming time of OpenACC is affected if students use CUDA first and then use OpenACC second in parallelizing the same serial code. Therefore, we propose the third hypothesis:

H3 The OpenACC programming time cannot be affected by CUDA language if OpenACC implementation is followed by CUDA implementation in the same problem.

To evaluate the independence of OpenACC, the comparison is abstracted in the following statistical way:

$$\begin{cases} H_0: \mu_{OpenACC_2} = \mu_{OpenACC_1} \\ H_1: \mu_{OpenACC_2} \neq \mu_{OpenACC_1} \end{cases}$$

$\mu_{OpenACC2}$denotes the mean programming time of OpenACC in group 2 while $\mu_{OpenACC1}$denotes the mean programming time of OpenACC in group 1. To achieve this evaluation, the students are divided into two groups to measure the independence of OpenACC: In group 1 students are required to parallelize code with CUDA first and then do it with OpenACC second while in group 2 students do it with OpenACC first and then do it with CUDA second.

Table 4. The data of OpenACC independence

Project	API	Independence(hours)	P-value (two-tail)
MP3	$OpenACC_1$	Mean 3.159, Std. 1.769, n=8	0.2523
	$OpenACC_2$	Mean 2.148, Std. 1.814, n=10	
MP4	$OpenACC_1$	Mean 1.856, Std.1.301, n=7	0.3233
	$OpenACC_2$	Mean 2.649, Std.1.693, n=9	

In MP3 the p-value(two-tail) of table 4 is greater than 0.05, so we have no significant evidence to reject *H0*. Therefore, although students in group 1 first use CUDA to solve the problem and then use OpenACC to solve the same problem, the programming time of OpenACC is not affected by the previous CUDA experience. Similarly, in MP4 the p-value(two-tail) of 0.3233 is also greater than 0.05. That is, the programming time of OpenACC is also not affected by the previous CUDA coding.

Thus, *H3* is supported in both problems (MP3 and MP4). That is the OpenACC programming time is not affected by previous CUDA coding experience.

5.4 Threats To Validity

The fact that this empirical study was run in a classroom environment and that the place of the coded projects is unknown means that there are threats to validity of our conclusions that should be discussed in this section when interpreting the results. Here we discuss the following five threats to validity for this study.

(1) The target population. Although 30 students are in our classroom, the valid data in MP3 are from 18 students and 16 students in MP4 respectively. The prerequisite of programming time comparison is that students can correctly finish each project with both OpenACC and CUDA. If the data only contain the programming time in either OpenACC or CUDA, the data of this student are excluded from our data pool. Thus the population size can impact our empirical results.

(2) The population academic background. As well known the programming efficiency highly depends on programmers' academic level. Generally, the programmers in a high educational level are more proficient. Although our empirical investigation was conducted in a classroom of 14 undergraduate students and 14 graduate students and the programming experience of most students is less than 10 years, the data of 2 students who have more than 10 years' experience still impact our conclusion, to some extent.

(3) The programming skills. Due to the difference of participants' academic background, the programming skills also need to be investigated. We reasonably believe that well-organized code structure improves the programming productivity.

(4) The number of test samples. We used two projects (MP3 and MP4) to conduct this empirical study. Although more projects can make our empirical results more convincing, considering the teaching work, we do not have more time to accelerate more projects. So the number of test samples can also impact our conclusions.

(5) The experiment configuration. Because our empirical investigation focuses on the programming time of accelerating serial code with both OpenACC and CUDA, the serial solutions to projects (MP3 and MP4) have been present before students did their assignments. If the serial code still needed to be finished by students, our data would vary. Besides, the number of parallelizable sections in serial code can impact the empirical conclusion. There is only one parallelizable section in each project so the easy identification of these sections can shorten their programming time.

(6) The process of data collection. The WebCode website automatically records students' coding behaviors with the timestamps. The time data in the database showed that most students did not continuously finish projects. So we assume that if there is no coding action within 10 minutes, the programming time was terminated. For example, if a student did not do anything for more than 10 minutes, we assume this student temporarily paused the coding work. But if a student did not code within 9 minutes, we assume that this student was thinking. The "10-minute" threshold may impact our conclusions.

6. CONCLUSION AND FUTURE WORK

In this paper, we conducted a programming productivity comparison of OpenACC and CUDA in the programming time, the independence of OpenACC and the dispersion discussion of the programming time. Through all of the discussion and analysis above, we can see the following empirical conclusions.
(1) OpenACC enables users to finish a parallel solution in a shorter time than CUDA. During the test of MP3 and MP4 projects, this conclusion can be clearly seen. In MP3, the CUDA mean time (3.6 hours) is significantly longer than the OpenACC one (2.6 hours) with one-tail p-value of 0.0465. In MP4, the CUDA mean time (3.6 hours) is significantly longer than the OpenACC one (2.3 hours) with one-tail p-value of 0.0008.

(2) Although in the comparison of the programming time the programming productivity of OpenACC model is higher than CUDA model, the performance of CUDA is better than OpenACC model in terms of the execution time. On the average, the execution time in CUDA is 9x shorter than execution time in OpenACC.

(3) The OpenACC programming work is not affected by previous CUDA experience. In the experiment configuration, students in the classroom were divided into two groups to parallelize the same serial project. In order to investigate whether the OpenACC programming time is significantly affected by CUDA programming experience, in group 1 the OpenACC solution was required to be done first while in group 2 the CUDA solution was required to be done first. With two-tail p-value of 0.2532(MP3) and 0.3233(MP4), there is no significant change in the OpenACC programming time with or without previous CUDA experience.

In future work, we plan to invite expert programmers to solve the same problems that were involved in this paper. It would allow us to see how different students are from experts. Besides, another change on measuring programming time is that the original serial code will not be provided so the time may vary in a longer time. Finally in both MP3 and MP4 there are 28 participants who finish each project with OpenACC and CUDA, so in total we have 114 coding samples to be analyzed to see whether we can extract the common programming styles as described in [13][14][15]. Finally, we will investigate the influence of the programming styles on the programming productivity.

7. ACKNOWLEDGEMENT

We would like to thank Dr. Jeff Carver at University of Alabama for conversations on the initial design of this empirical investigation.

REFERENCES

[1] Lorin, H., Jeff, C., Forrest, S., Sima, A., Victor, B., Jeffrey, K. H., and Marvin, V. Z. “Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers”. In Proceedings of SuperComputing, 2005

[2] Basili, V., Asgari, S., Carver, J., Hochstein, L., Hollingsworth, J., Shull, F., Zelkowitz, M. “A Pilot Study to Evaluate Development Effort for High Performance Computing”. University of Maryland Technical Report CS-TR-4588. April 2004.

[3] Marvin, Z., Victor, B., Sima, A., Lorin, H., Jeff, H., Taiga, N. “Measuring Productivity on High Performance Computers”. METRICS '05 Proceedings of the 11th IEEE International Software Metrics Symposium. 2005

[4] Hoshino, T. Maruyama, N. Matsuoka, S. and Takaki, R." CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a memory-bound CFD Application". 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 136-143. May 2013

[5] Herdman, J.A. Gaudin, W.P. Mclntosh-Smith, S. and Boulton, M. "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA". '12 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC). pp. 465-471 Nov. 2012

[6] Po-Yu, C., Chun-Chieh, L., Long-Sheng, H. and Kuo-Hsuan, W. "Overview and Comparison of OpenCL and CUDA Technology for GPGPU". IEEE Asia Pacific Conference on Circuits and Systems (APCCAS). pp. 448 – 451. Dec 2012

[7] Howell, D. D. Statistical Methods for Psychology. 5th ed. 2002, Pacific Grove: Duxbury.

[8] Kai, W., Charles A. P., Arnold M. S., Michael A. L. "Entropy Explorer: an R package for computing and comparing differential Shannon entropy, differential coefficient of variation and differential expression". DOI 10.1186/s13104-015-1786-4

[9] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., kruger, J., Lefohn, A.E. and Purcell, T.J. "A Survey of General-Purpose Computation on Graphics Hardware". Computer Graphics Forum, vol.26, pp. 80-113. March 2007

[10] NVIDIA Corporation, NVIDIA CUDA C Programming Guide v5.0, October 2012.

[11] Fang, J., Varbanescu, A. and Sips, H. "A Comprehensive Performance Comparison of CUDA and OpenCL." International Conference of Parallel Processing (ICPP), pp. 216-225, Sept. 2011

[13] The OpenACC Application Programming Interface, Version1.0, November 2011.

[14] Anthony C., and Maryanne F. "Programming Style: Influences, Factors, and Elements". ACHI '09 Second International Conferences on Advances in Computer-Human Interactions, 2009.

[15] Matthew, C. J. "A First Look at Novice Compilation Behaviour Using BlueJ". Computer Science Education, 15(1), pp. 25-40. Doi: 10.1080/08993400500056530

[16] Iris V. "Expertise in debugging computer programs: A process analysis". International Journal of Man-Machine Studies, 23(5), pp. 459-494. Doi:10.1016/S00207373(85)80054-7

[17] Lorin H., Victor R. B., Marvin V. Z., Jeffrey K. H. and Jeff C. "Combining self-reported and automatic data to improve effort measurement." Proceedings of Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005). pp.356-365. 2005

[18] Christgau, S. Spazier, J. Schnor, B. Hammitzsch, M. Babeyko, A. and Waechter, J. "A comparison of CUDA and OpenACC: Accelerating the Tsunami Simulation EasyWave". 27th International Conference on Architecture of Computing Systems (ARCS). pp. 1-5. Feb. 2014

[19] The OpenACC Application Programming Interface, Version1.0, November 2011.

[20] Kei, I., Fumihiko, I. and Kenichi, H. "An OpenACC Optimizer for Accelerating Histogram Computation on a GPU." 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp.468-477. 2016

[21] Suttinee, S., James, L., Simon, S., Francois, B. and Satoshi, M. "Understanding Performance Portability of OpenACC for Supercomputers." 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), vol. 00, pp. 699-707, 2015. Doi:10.1109/IPDPSW.2015.60

[22] Ronald, D., Resit, S., Frederick, J. V. "On the performance and energy-efficiency of multi-core SIMD CPUs and CUDA-enabled GPUs" pp. 174-184, 2013, doi:10.1109/IISWC.2013.6704683

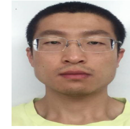
[23] Tetsuya, H., Naoya, M., Satoshi, M. "An OpenACC Extension for Data Layout Transformation" pp.12-18, 2014, doi:10.1109/WACCPD.2014.12

[24] Jose, M. N., Marisa, L.V. "A Performance Study of CUDA UVM versus Manual Optimizations in a Real-World Setup: Application to a Monte Carlo Wave-Particle Event-Based Interaction Model", IEEE Transactions on Parallel & Distributed Systems, vol. 27. pp. 1579-1588, June 2016, doi:10.1109/TPDS.2015.2463813

[25] Ryan, S. L., Qinru, Q. "Effective Utilization of CUDA Hyper-Q for Improved Power and Performance Efficiency". 2016 IEEE International Parallel and Distributed Processing Symposium Workshops. pp. 1160-1169. May 23-27, 2016

AUTHOR

Xuechao Li
Ph.D candidate
Auburn University USA



Po-Chou Shih
Visiting scholar
Auburn University USA
Ph.D candidate
National Taipei University of Technology Taiwan

Jeffrey Overbey
Assistant Professor
Computer Science Department
Auburn University USA

Cheryl Seals
Associate Professor
Computer Science Department
Auburn University USA

Alvin Lim
Professor
Computer Science Department
Auburn University USA