A Case Study of Novice Programmers on Parallel Programming Models

Xuechao Li[1*], Po-Chou Shih[2], Xueqian Li[3], Cheryl Seals[1]

[1] Dept. of Computer Science, Auburn University, 345 W. Magnolia Ave, Auburn, Alabama, USA.
[2] Graduate Institute of Industrial and Business Management, National Taipei University of Technology, 1, Sec. 3, Zhongxiao E. Rd, Taipei, Taiwan.
[3] Dept. of Civil Engineering, Auburn University, 238 Harbert Engineering Center, Auburn, Alabama, USA.

* Corresponding author. Tel.: +13344984008; email: xcl@auburn.edu
Manuscript submitted May 8, 2017; accepted July 25, 2017.
doi: 10.17706/jcp.13.5.490-502

Abstract: High performance computing (HPC) languages have been developed over many years but the ability to write parallel program has not been fully studied based on the programmer effort subjects. In this paper for obtaining quantitative comparison results, we conducted an empirical experiment, which quantified four indices related to the programming productivity of CUDA and OpenACC: the achieved speedup, programmers' workload on parallel solutions, effort per line of code and the dispersion on speedup. Twenty-eight students were recruited to finish two parallel problems: Machine Problem 3(MP3) and Machine Problem 4 (MP4) and the research data was collected by a WebCode website developed by ourselves. The statistical results indicated that (1) the speedup in OpenACC is 11.3x less than CUDA speedup; (2) the workload of developing OpenACC solutions is at least 2.3x less than CUDA ones; (3) the effort per line of code in OpenACC is not significantly less than CUDA; (4) and the OpenACC dispersion on speedup is significantly less than CUDA.

Key words: Development effort, programming productivity, empirical experiment, OpenACC, CUDA.

1. Introduction

Among interfaces for GPU computing, the Compute Unified Device Architecture CUDA [1] programming model offers a more controllable interface to programmers compared to OpenACC [2]. Not surprisingly, programming GPUs is still time-consuming and error-prone work compared to the experience of serial coding [3]. For the CUDA programming model, professional knowledge of hardware and software is a big challenge to programmers, especially for novices, such as memory management of GPU, optimization technologies and understanding GPU architectures. Some applications or benchmarks are massive and complex, it is practically impossible for developers to re-write them yet it is possible theoretically [4]. To simplify the parallel programming, the OpenACC was released in 2012, which allows the compiler to convert directives to corresponding parallel code. But the conversion may consume more time. Hence users will naturally evaluate the tradeoff between the acceleration performance and the programmer effort. Specifically, even though the performance acceleration can be obtained with parallel models, writing parallel code may be not realistic if programmers need more time or effort on parallelized problems.

Many researchers have worked on performance evaluation of OpenACC and CUDA, but to date, few studies have investigated performance based on human subjects such as programmers' effort per line of

code on parallel programming. To fill this gap in the literature, we conducted an empirical experiment on the productivity comparison of parallel programming models with particular problems. We compared the productivity of OpenACC and CUDA in the following four aspects: the achieved speedup, programmers' workload, effort per line of code and the dispersion on speedup. While there is a rich body of literature on definitions of programming productivity in high performance communities, for this study we use programming times to define programming productivity in this paper and the number of lines of code to define the size of parallel solutions. To quantitatively compare the productivity of OpenACC and CUDA, we propose the following four research questions: (1) *Which API can achieve a higher speedup between OpenACC and CUDA?* (2) *Is the size of OpenACC solutions less than the size of CUDA?* (3) *Is the effort per line of code in OpenACC equal to the effort per line of code in CUDA?* (4) *Is the OpenACC dispersion on the speedup less than CUDA?*

We conducted this productivity comparison between OpenACC and CUDA based on two considerations: (1) CUDA is still the most popular parallel language in both academy and industry and OpenACC is an easy-learned and directives-based parallel language, which is remarkably beneficial for beginners; (2) One motivation of the OpenACC model is to simplify low-level CUDA such as replacing CUDA optimization methods with directives.

This paper makes the following contributions:

- The acceleration ability was investigated. Although OpenACC can assist programmers to finish parallel problems in a shorter time, the conversion from OpenACC directives to CUDA code might cost more time. For this reason, the speedup comparison was conducted.
- The comparison of the workload of OpenACC and CUDA was investigated. Because of the simplification feature of OpenACC, we are interested in whether OpenACC workload is significantly less than CUDA workload.
- We quantitatively compared programmers' efforts in the parallel programming solutions between OpenACC and CUDA based on the index of effort per line of code. To convert the subjective measurement to the objective one, we used programming time to define the human effort. Participants are required to do the parallelizing work on the WebCode developed by the research team to collect and store necessary information to be analyzed (e.g. programming time).
- The dispersion on speedup between OpenACC and CUDA was compared. Considering the high level directives in OpenACC, it might hide the skill difference between experienced programmers and novices, and as such the dispersion was also investigated.

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

2. Background

In this section, we simply introduce three components related to our research work in this paper: (1) productivity work in high performance computing; (2) the CUDA programming model and (3) the OpenACC programming model.

As far as the development of parallel solutions is concerned, the productivity measurement needs to consider developing, porting, rewriting, tuning and maintenance. One of the mainstream methods is to

measure the total time to solution of a parallel problem [5]. Additionally, the design of the empirical experiment to measure the productivity needs to be carefully conducted because the time programmers spent thinking about the development cannot be easily and precisely captured. Usually researchers like to quantify the code development process in a specialized environment so that the related data can be collected [6].

CUDA is a parallel API designed only for NVIDIA's GPUs and the motivation of CUDA development is that engineers try to fully utilize the power of the GPUs to process computing operations. The workflow of CUDA parallel solutions can be briefly described as the following: the host sends data from the main memory to the graphic processing units. And then all cores simultaneously perform the tasks with different data. Until all computations are done, the results will not be sent back to the host [7].

To overcome the limitations above, researchers and companies are establishing alternative parallel models with the motivation of lowering the barriers of programming parallel accelerators [8]. One promising parallel model is OpenACC. It uses compiler directives to generate a part of codes programming GPUs. Unlike the CUDA model, what users need to do is to select proper directives and insert them into the existing code to parallelize it, which dramatically reduces programmers' workload. The PGI [9] compiler has been widely used to compile OpenACC code because directives can be automatically converted into executable parallel codes run on the GPU. The OpenACC Application Programming Interface [2], [9] offers a directive-based approach for describing how to manage data and to execute sections of codes on the device. The "directives" attribution not only reduces programmers' workload, but lowers programming error occurrences.

3. Related Work

In this section, we collect and review a fair amount of work on the productivity measurements and development effort evaluations on high performance computing as well as the performance analysis and evaluations between OpenACC and CUDA. Additionally, we also briefly describe research work of other parallel models such as OpenCL and OpenMP.

Victor *et al.* [10] generated a set of feasibly experimental methods to collect the development time of parallel solutions and then evaluated their own experimental protocols. Subjects were required to fill out series of forms to report their effort for manual collection. In addition, authors created a wrapper for the compiler in order to automatically collect the development time. Through analyzing the data, a set of experimental lessons learned was present and twelve well-formed hypotheses were supported.

Another interesting experiment for improving programming effort measurement was conducted by Lorin *et al.* [11]. Authors used an interval-based hybrid measure with the combination of self-reported data and automatic data collected by the instrumented compiler. The results showed that the effort log can potentially increase the accuracy of the effort measured, but it is not clear whether this similar approach can be adopted in the industrial environment.

Ken *et al.* [12] explored the ratio between programming effort and performance. The authors proposed the use of two dimensionless ratios, relative power and relative efficiency to measure the productivity of programming interfaces. Matlab, Java and Fortran were used to illustrate the power-efficiency ratio.

A framework for measuring supercomputer productivity was proposed in [13]. According to common economic definitions of productivity and the Utility Theory, this paper tried to capture the essential aspects of HPC systems and discussed important metrics for measuring productivity such as performance-oriented system metrics and performance-oriented application metrics.

Thomas [14] established a set of mutually consistent and complementary metrics for the model of productivities. With the help of the formulation of the concept of computational productivity, the author

quantitatively presented the measurement theories. Another interesting measurement was proposed by Marvin *et al.* [6]. The authors used the ratio of relative speedup and relative effort to calculate the productivity. Two different programming teams were employed to solve the same problem on the same hardware platform. The conclusions showed that their formula was validated by a set of experiments.

In [15] the performance of OpenACC and CUDA was compared based on kernel benchmarks and a memory-bound CFD application. This evaluation showed that in general OpenACC performance is 50% lower than CUDA. Taking a closer look at the difference between two models, the authors pointed out that the lack of the OpenACC interface for *shared* memory is a major problem resulting in loss of performance. Additionally, the OpenACC performance portability was also evaluated by Amit *et al.* [16]. In this experiment, twelve OpenACC programs and three architectures (NVIDIA CUDA, AMD GCN, Intel MIC) were employed and the effects of various optimizations and programming settings on different architectures were also discussed. The conclusion was made that while code portability was achievable, performance portability eludes a common programming model.

4. Methodology

Any quantification involving human subjects is a surprisingly difficult task. The development effort measurement of programmersis also not an exception. The external and internal human factors need to be carefully processed in order to obtain accurate data. For example, some participants write code continuously for many hours while some programmers prefer to write code intermittently. Given those sets of challenges in the experiment, we set out to develop the data collection mechanisms and experiment protocols to accurately measure their programming time.

While conducting this experiment, we face two challenges: how to convert the subjective measurement to the objective on parallel programming models and how to accurately and consistently measure human efforts. Although there have been various productivity definitions in the previous work, we used *the time to solution* as the measurement index because one of the motivations of OpenACC was to reduce programmers' workload in parallel solutions so that this programming model can be easily and quickly applied to users at different programming levels. For the second challenge, we carefully review a number of historical methods: manual reports by participants and automatic reports by instrumented compilers or environments. Because of the inaccuracy and unreliableness of manual reports, the automatic report was adopted in our experiment.

The target samples in this experiment consists of novices or participants who have little parallel programming experience based on the following considerations: (1) novices prefer to use high level compiler-directive programming models such as OpenACC; (2) one of the motivations of OpenACC is to simplify CUDA programming and to lower the barriers of parallel programming; (3) it would be hard to distinguish the productivity between OpenACC and CUDA if experienced programmers are hired because the advantages of OpenACC directives may be hidden by the skills of highly experienced programmers.

To objectively evaluate the parallelized ability of OpenACC and CUDA, the assignment selection also need to be seriously considered. If the problems are very easily solved or coded, the programming time will likely be the same or close. Similarly, if high amounts of effort is required to finish the assignments or the problems are complex, we may not obtain the data because few participants can solve them. Hence, in this experiment we do not intend to use very easy problems such as matrix multiplication or comprehensive assignments to be the test samples.

For measurement objectives, we tried to select the most concerned indices about OpenACC and CUDA in industry and academia as objectives to compare them. There are four total indices we collected and analyzed: (1) the speedup. For any parallel programming model, a key question is whether it can be easily

used to achieve the speedup and then we did a deeper analysis: which programming model can assist users to achieve a higher speedup? (2) the size of the solution. We use the number of lines of code to define the size of the solution. It is naturally believed that the larger the size is, the more time programmers need to spend; (3) Effort per line of code. Although the programming time can reflect the efficiency of the programming model, the size of the solution also needs to be seriously considered. So the effort per line of code can perfectly solve this problem. (4) The dispersion on speedup. For OpenACC, the feature of high level directives might not obviously distinguish the performance difference between novices and experienced programmers. On the other hand, for the programmers highly skilled in CUDA could easily obtain the peak performance while it is highly possible for beginners to fail in CUDA coding. So among the correct solutions, we also investigate the dispersion on the speedup of two APIs.

5. Experiment Design

We empirically conducted this experiment at Auburn University, U.S. during Spring semester 2016 and the participants were students enrolled in COMP 7930/4960 GPU Programming.

Our goal was to objectively evaluate which API is more productive between OpenACC and CUDA in terms of the programmers' efforts. So four indices related to the productivity are investigated: (1) effort per line of code; (2) the programmer workload; (3) the achieved speedup; and (4) the dispersion on speedup. The goals are mainly based on the following three considerations: (1) OpenACC can provide high-level sets of directives to achieve the similar CUDA functionalities. In this way OpenACC API will simplify the parallel work so that productivity can be improved and programmers' effort is correspondingly lowered. (2) If the directives can replace low level functional codes, it is naturally believed that the size of OpenACC solutions is supposed to be smaller than CUDA one for the same problem. (3) Due to the fact that conversion from OpenACC directives to CUDA code consumes more time, the comparison on the achieved speedup is conducted. (4) The performance difference highly depends on the programming experience if CUDA is used while high level OpenACC directives can alleviate the performance gap between beginners and skilled users. Considering this, the dispersion on speedup is also investigated.

Hence, we propose the following four hypotheses in this paper:

H1: The achieved speedup in OpenACC is less than CUDA.

H2: The size of OpenACC solutions is smaller than the size of CUDA solutions.

H3: The effort per line of code for OpenACC is not significantly less than CUDA.

H4: In terms of speedup, the dispersion in CUDA is significantly larger than the dispersion in OpenACC.

5.1. Target Samples

The participants were recruited from COMP 7930/4960, GPU Programming, a one-hour, semester-long course covering the basics of CUDA, OpenACC, and parallel algorithm design at Auburn University, U.S. during Spring semester 2016. Twenty-eight students total were involved in our experiments. In order to understand students' academic background, we also created a programming experience questionnaire which mainly evaluates students' programming background from 3 categories: Education Level, Years of Programming Experience (any language), and Experience in CUDA and OpenACC.

- From the perspective of the educational level, there were 14 undergraduate students and 14 graduate students. Our experimental conclusions would be influenced by the percentage of students in different educational levels. For example, if the percentage of graduate students is more than 90%, the average programming effort might be less. But if the percentage of undergraduate students is more than 90%, the average programming effort might be more.
- In the "Years of Programming experience (any language)" field, because the students were recruited

from the department of computer science and software engineering, 53.6% of participants (15 out of 28) have less than 5 years programming experience (in any programming languages), 39.2% of participants (11 out of 28) have programming experience of more than 5 years but less than 10 years, and finally there are only 2 students who have written code for more than 10 years. Based on their programming experience background, we think that it is a fair comparison because few students are very experienced (more than 10 years in programming) and 92.8% of students have programming experience (less than 10 years).

- For "Experience in CUDA and OpenACC", we are also interested in another question: to what extent do students already understand our target languages—OpenACC and CUDA—before they participated in the experiment? The data from the questionnaire showed that 89.3% of students (25 out of 28) can write some simple code with OpenACC or CUDA and 10.7% of them (3 out of 28) have no or little experience in both APIs. No one is an experienced programmer. Overall this is a "balanced" environment to conduct this comparison experiment.

5.2. Experiment Configuration

The biggest challenge in this experiment is to accurately and consistently collect the programming time. To achieve that, we developed the WebCode website where students did all of the programming work, and all of coding behaviors were stored in the database with timestamps. (Source code is available at https://github.com/joverbey/webcode). The host processor was a 10-core Intel Xeon E5-2650 at 2.3 GHz with 16 GB DDR4 RAM and 25 MB cache. The device was an NVIDIA Tesla K40c at 0.75 GHz with 11,520 MB global memory.

Another challenge is to precisely count the programming time in the database, because we did not require participants to finish their assignments in a particular place such as the classroom and a specific time range. For example, if the coding behavior for some participant paused for 5 minutes, there would be two possibilities: (1) the programmer was thinking and tried to solve some bugs; (2) the programmer temporarily stopped the coding work and after 5 minutes this programmer came back and continued to do it. Based on 112 solutions, we used 10 minutes as the threshold to distinguish those two situations. Specifically, if the pausing length was greater than 10 minutes, we assumed that the programmer had stopped the coding work; otherwise the programmer was thinking about the assignment.

5.3. Test Samples

In order to obtain data for verifying our hypotheses, we created five parallel problems: MP0 (Machine Problem 0), MP1 (Machine Problem 1), MP2 (Machine Problem 2), MP3 (Machine Problem 3) and MP4 (Machine Problem 4). Before MP3 was distributed to students, our WebCode website was still in progress so the data from MP3 and MP4 can be captured. But the results from MP0, MP1 and MP2 were excluded. Below we briefly describe details of MP3 and MP4.

MP3: Students were required to simulate the process of heat transfer on a long, thin rod. The left end of the rod was heated by the source from temperature 0°F. We used $t_{old}[i]$ to denote the temperature of the i-th points at a particular unit of time. The section of the code is shown in Fig. 1.

```
// 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]);
```

Fig. 1. The parallelizable code in MP3.

MP4: This is an encryption and decryption problem. In order to send the messages in a secure way, the sender and the receiver share a secret key which was used to encrypt and decrypt the message. Students were required to use a brute-force key search to figure out someone's 28-bit key. Since most of computing time was consumed in the encryption and decryption phases, Fig. 2 displays serial code in two phases.

```
encrypt(uint32_t *data, const uint32_t *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 */
}
decrypt(uint32_t *data, const uint32_t *key) {
    for (i=0; i<32; i++) {                               /* basic cycle start */
        v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        sum -= delta;
    }                                                    /* end cycle */
}
```

Fig. 2. The parallelizable part of encryption and decryption code in MP4.

5.4. Data Collection Procedure

For verifying whether our hypotheses can be supported by the data from test samples of MP3 and MP4, we need to record programmers' coding behaviors, such as programming time. The WebCode website can achieve these recording tasks in the experiment. All students were required to finish their code on this platform and each coding action, such as "paste" and "copy", was stored in the database with a timestamp. Through retrieving the database, the data can be easily analyzed for the hypotheses.

5.5. Data Analysis

For each problem (MP3 or MP4), every student submitted both OpenACC and CUDA solutions. Hence we used paired-sample t-test [17] to analyze the speedup data in order to see whether or not the achieved speedup in OpenACC is significantly less than the achieved speedup in CUDA. Similarly, we used the same statistical method to verify our hypothesis 2 and hypothesis 3. In hypothesis 4, in order to evaluate the speedup dispersion between OpenACC and CUDA, we used F-test for two sample variances (left-tail) [18], [19]. This test allows us to investigate the hypothesis whether in both problems, the CUDA variance is significantly greater than the OpenACC variance. Finally for all statistical analyses above, we use a significant levelα of 0.05 to make our conclusions.

6. Experiment Results Discussion

In this section we used statistical analyses to discuss the results from projects MP3 and MP4. The paired sample t test allowed us to make conclusions in comparisons of the achieved speedup, the size of solutions, effort per line of code, and the dispersion on speedup between CUDA and OpenACC.

To objectively conduct a convincing comparison, only the correct solutions in both OpenACC and CUDA were used as samples to be analyzed. Therefore, the valid sample is 18 in MP3 and 16 in MP4.

Also, the order of the OpenACC programming and the CUDA programming would affect the coding productivity of participants. Taking MP3 as an example, if participants first used OpenACC to parallelize MP3 and then used CUDA to do it again, the OpenACC work done previously may provide some clues or hints when this participant parallelized the MP3 with CUDA. To avoid this potential negative impact, we divided 28 students into two groups, and MP3 and MP4 were finished in the order described in Table 1.

Table 1. The Order of API Selection for Projects

Group	MP3	MP4
1	CUDA first and then OpenACC	OpenACC first and then CUDA
2	OpenACC first and then CUDA	CUDA first and then OpenACC

6.1. Acceleration

The goal of high-performance parallel programming models is to accelerate the original serial code, improve computing efficiency, and reduce the waiting time of resources. OpenACC can assist programmers to automatically set up dimensions in parallel computing, but the high level directives still need to be converted into CUDA code before the code is executed by GPUs. Therefore, although programmers can achieve the speedup with OpenACC or CUDA for the same problem, the achieved speedup of OpenACC might be slower than CUDA. Based on this analysis, we proposed our first hypothesis:

H1: The achieved speedup in OpenACC is significantly less than CUDA

To verify this hypothesis, we abstract the comparison of the achieved speedup in a statistical format, as in (1).

$$\begin{cases} \mathrm{H}_0 : \mu_{OpenACC} \geq \mu_{CUDA} \\ \mathrm{H}_1 : \mu_{OpenACC} < \mu_{CUDA} \end{cases} \tag{1}$$

We used $\mu_{OpenACC}$ to denote the mean speedup of OpenACC and μ_{CUDA} to denote the mean speedup of CUDA. The hypothesis H_0 assumes that the mean speedup of OpenACC is equal to or greater than the mean speedup of CUDA while the hypothesis H_1 assumes that the mean speedup of OpenACC is less than the mean speedup of CUDA.

Through the data in Table 2, we can see that compared to the serial code, the OpenACC model can obtain 3.142x speedup while the CUDA can obtain 34.74x speedup in MP3. The OpenACC speedup is significantly slower than CUDA because the p-value (2.18E-05) is less than significant level $\alpha(0.05)$. Similarly, in MP4 the participants can obtain 2.514x speedup with OpenACC while the speedup is 762.4x if participants use CUDA to accelerate the serial code. The OpenACC speedup is also significantly slower than CUDA with the p-value (4.96E-05). Therefore, there is sufficient evidence to reject H_0 and accept H_1. Hence, we make the conclusion that the OpenACC speedup is significantly less than CUDA for the same serial code.

Table 2. The Speedup Comparison

Project	API	Speedup			One-tail P-value
		Mean	Std.	Samples	
MP3	CUDA	34.74	24.44	18	2.18E-05
	OpenACC	3.14	0.60	18	
MP4	CUDA	7.62E02	5.82E02	16	4.96E-05
	OpenACC	2.51	02.58	16	

6.2. Size of Solutions

In addition to the programming time as an important factor in the productivity comparison, the size of parallel solutions also cannot be ignored. In this paper we used lines of code (LOC) to quantify the size. Because the original serial code was given, we are interested in code expansion [20] instead of simply counting the number of lines of solutions. Specifically, we investigated how the size of parallel solutions changes from serial code to OpenACC or CUDA code.

There are different parallelizing ways in different parallel models. For example, OpenACC requires users to add few directives into serial code while significant code changes need to be made in CUDA. Hence we proposed the second hypothesis.

H2: In terms of lines of code, the size of CUDA solutions is significantly larger than OpenACC one.

A similar method in section 6.1 is used in the comparison of the size. We abstract the comparison of the size of parallel solutions in a statistical format, as in (2).

$$\begin{cases} \mathrm{H}_0 : S_{OpenACC} \geq S_{CUDA} \\ \mathrm{H}_1 : S_{OpenACC} < S_{CUDA} \end{cases} \tag{2}$$

We used $S_{OpenACC}$ to denote the mean number of lines expansion of code of OpenACC solutions and S_{CUDA} to denote the mean number of lines expansion of code of CUDA solutions. The hypothesis H_0 assumes that the mean number of lines expansion of code of OpenACC solutions is equal to or greater than the mean number of lines expansion of code of CUDA solutions while The hypothesis H_1 assumes that the mean number of lines expansion of code of OpenACC solutions is less than the mean number of lines expansion of code of CUDA solutions.

The lines of serial code in MP3 is 79 and the data in Table 3 showed that the mean number of lines expansion (43.56) in CUDA is significantly larger than OpenACC (19.17) because the one-tail p-value (2.6E-05) is less than 0.05. Similarly, the lines of serial code in MP4 is 77 and the mean lines expansion (50.06) in CUDA is significantly larger than OpenACC one (13.75) because the one-tail p-value (3.94E-06) is less than 0.05. Hence we make the conclusion that in terms of lines of code, the size of CUDA solutions is significantly larger than OpenACC for parallelizing the same problem.

Table 3. The Comparison of Sizes of Parallel Solutions

Project	API	LOC(expansion)			One-tail P-value
		Mean	Std.	Samples	
MP3	CUDA	43.56	16.80	18	2.60E-05
	OpenACC	19.17	21.43	18	
MP4	CUDA	50.06	18.96	16	3.94E-06
	OpenACC	13.75	12.18	16	

6.3. Effort Per Line of Code

In order to accelerate the serial code, normally the workload in CUDA is significantly greater than in OpenACC. CUDA allows users to write low level directives to fully utilize hardware architecture, such as *shared* memory for achieving the speedup of the problems, so programmers need more lines of code to parallelize the code. On the other hand, the high level OpenACC model provides directives to assist users to avoid the obstacle of hardware software knowledge, so few lines of code need to be added into the existing code.

For fairly evaluating the programmers' effort on the coding work, we selected cost per LOC as an index based on two considerations: (1) if we simply use the total programming time to evaluate or measure the human effort, it is not an accurate or fair index because the workload such as LOC also has an important impact on evaluating effort of different programming models. Therefore the combination of the total effort and the total lines of code into effort per LOC can perfectly solve this problem; (2) for the previous research work done on programming productivity evaluation, the index of cost per LOC has been widely adopted. Hence, we posed the third hypothesis.

H3: The effort per line of code for OpenACC is not significantly less than CUDA.

The statistical formula is described in the following way for this hypothesis, as in (3).

$$\begin{cases} \mathrm{H}_0 : E_{OpenACC} \geq E_{CUDA} \\ \mathrm{H}_1 : E_{OpenACC} < E_{CUDA} \end{cases} \tag{3}$$

We used $E_{OpenACC}$ to denote the mean time of each line of code in OpenACC solutions and E_{CUDA} to denote the mean time of each line in CUDA solutions. The hypothesis H_0 assumes that the mean time spent on each line of code in OpenACC solutions is equal to or greater than the mean time in CUDA solutions while The hypothesis H_1 assumes that the mean time of OpenACC solutions is less than the mean time of CUDA solutions.

The data in Table 4 shows that although the mean effort (1.545 minutes) spent on LOC in OpenACC is slightly less than the mean effort (1.810 minutes) in CUDA in MP4, it is not significant because the p-value (0.2388) is greater than 0.05. Similarly, for MP3 students spent an average 1.601 minutes on each line of OpenACC code and 1.810 minutes on each line of CUDA code. The effort spent on OpenACC is not significantly less than CUDA because the p-value (0.1918) is greater than 0.05. Statistically, in both MP3 and MP4, the p-value is greater than 0.05(α). Therefore, there is no sufficient evidence to reject H_0. Hence, we make the conclusion that the effort per line of code for OpenACC is not significantly less than CUDA.

Table 4. The Effort per LOC Comparison of Parallel Solutions

Project	API	Time			One-tail P-value
		Mean	Std.	Samples	
MP3	CUDA	1.81	1.02	18	0.2388
	OpenACC	1.60	0.98	18	
MP4	CUDA	1.72	0.76	16	0.1918
	OpenACC	1.55	1.07	16	

6.4. The Dispersion on Speedup

One of the features in OpenACC is to provide high level directives for accelerating the serial code. But the limited "parallel directives" cannot fully utilize available resources for the peak performance. On the other hand, low level CUDA allows programmers to accelerate or optimize code with all of potential resources, especially with the utilization of memory. Based on the features analysis above, we can see that the convenience of OpenACC annotations also limits the performance while the complex CUDA directives can help experienced programmers to obtain the peak performance.

Hence, another research work also needs to be investigated: although both OpenACC and CUDA can achieve the speedup, the CUDA dispersion on the speedup might be larger than the dispersion in OpenACC. The fourth hypothesis is posed:

H4: For speedup, the dispersion in CUDA is larger than the dispersion in OpenACC

Statistically, the dispersion comparison can be formulated in the following format, as in (4).

$$\begin{cases} \mathrm{H}_0 : D_{OpenACC} \geq D_{CUDA} \\ \mathrm{H}_1 : D_{OpenACC} < D_{CUDA} \end{cases} \tag{4}$$

We used $D_{OpenACC}$ to denote the speedup dispersion in OpenACC and D_{CUDA} to denote the speedup dispersion in CUDA. The hypothesis H_0 assumes that for speedup, the dispersion in OpenACC is greater or equal to the dispersion in CUDA while The hypothesis H_1 assumes thatfor speedup, the dispersion in

OpenACC is less than the dispersion in CUDA.

In Table 5, we can see that the speedup variance in CUDA is significantly greater than the speedup variance in OpenACC for MP3. And a similar result is also from MP4 data. With the p-value which is less than 0.05, the speedup variance in CUDA is significantly greater than the speedup variance in OpenACC. Statistically, since the p-value is less than 0.05, there is sufficient evidence to reject H_0 and acceptH_1. Hence, the fourth hypothesis is supported.

Table 5. The Dispersion Comparison of Parallel Solutions

Project	API	Speedup dispersion			left-tail P-value
		Std.	Variance	Samples	
MP3	CUDA	24.44	597.20	18	0 (<10E-30)
	OpenACC	0.60	0.357	18	
MP4	CUDA	5.81E02	3.38E05	16	0 (<10E-30)
	OpenACC	2.58	6.67	16	

6.5. Threats to Validity

In this human-subject empirical experiment, we conducted each trial in an objective manner. However, there were some threats to the validity of our conclusions. In this section, we list the potential factors that threaten this experiment.

From the perspective of programming styles, the efficiency on the programming is highly influenced by the programming skills. The well-organized code structure could shorten the programming time, reduce effort spent on tasks, and improve the code execution efficiency.

The "novice" programming background may affect our conclusions. In this experiment, most of the participants were students who were learning parallel languages for the first time, so in measuring the achieved speedup, size of solutions, effort per LOC and the dispersion on the speedup, it would be different if we recruit experienced programmers such as experts to parallelize the same problem. Also all participants are from Auburn University, U.S.so the results are specific to this institution and may produce different results when conducted in another academic institution.

The number and the difficulty level of assignments would affect our data. Our conclusions in this paper are based on the data from two assignments. The more test samples we have, the more accurate the results are. If more assignments can be parallelized, our conclusions can be more convincing. In addition, if we distributed some easier or harder assignments than ones in this experiment, the results would be different.

The 10-minute threshold in calculating programming times may be reconsidered. The automatic data collection makes authors have no way to understand programmers' status if programmers paused during solving the assignments. The 10-minutes threshold is based on our experience during reviewing 112 solutions. Hence if this time threshold was changed, some data also need to be adjusted correspondingly.

The experiment configuration may affect participants' programming performance. In order to purely evaluate parallelizing efficiency of OpenACC and CUDA, we presented the serial code to participants before they began to code. This strategy may affect their programming efficiencies.

7. Conclusion and Future Work

We have successfully conducted an initial empirical experiment and a number of lessons has been learned.

In terms of the execution time, the OpenACC model, on the average, can obtain 15x speedup than the serial code while the CUDA model can obtain 55x speedup than the serial code. So the conversion from OpenACC directives to CUDA code and the implementation of the OpenACC directives have significantly

negative impact on the execution time. Although the CUDA model outperforms the OpenACC model in terms of speedup, the portability issue of CUDA needs to be considered if the code might be run on other platform instead of Nvidia GPU in the future. So an alternative model is OpenCL which is more difficult to code.

In terms of lines expansion of code, the OpenACC model can significantly reduce programmers' workload. But from the perspective of effort per line of code, OpenACC is not significantly better than CUDA. Finally, the conclusion from the dispersion on speedup suggests that highly skilled programmers can obtain better computing performance with CUDA model while the OpenACC model fits for novices for the general acceleration.

In the future work, we plan to invite experts to finish the same problems and then analyze the difference between experts and novices. In addition, the influence of programming styles on effort also needs to be conducted. Finally, the programming skills also play an important role in programming work such as speedup and workload, so we plan to accurately and objectively measure programming skills based on standards.

References

[1] Morgan, K. (2010). *Programming Massively Parallel Processors: A Hands-on Approach* (2nd ed.).

[2] OpenACC Consortium. The OpenACC application programming interface, version 1.0. Retrieved from http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf

[3] Seyong, L., & Jeffrey, S. V. (2012). Early evalustion of deirective-based gpu programming models for productive exascale computing. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.*

[4] Makoto, S., Shoichi, H., Kazuhiko, K., Hiroyuki, T., & Hiroaki, K. (2013). A comparison of performance tunabilities between OpenCL and OpenACC. *Proceedings of the 7th International Symposium on Embeded Multicore/Manycore System-on-Chip* (pp. 147-152).

[5] Marvin, Z., Victor, B., Sima, A., Lorin, H., Jeff, H., & Taiga, N. (2005). Measuring productivity on high performance computers. *Proceedings of the 11th IEEE International Software Metrics Symposium* (pp. 6-15).

[6] Forrest, S., Jeffrey, C., Lorin, H., & Victor, B. (2005). Empirical study design in the area of high-performance computing (HPC). *Proceedings of the International Symposium on Empirical Software Engineering (ISESE)* (pp. 305-314).

[7] Cleversion, L. L., Carlos, M. D. Z., & Julio, C. S. A. (2013). Comparative analysis of OpenACC, OpenMP and CUDA using sequential and parallel algorithms. *Proceedings of the* Duxbury *11th Workshop on Parallel and Distributed Processing.*

[8] Mark, S., Mark, P., & Derek, G. (2006). A performance-oriented data parallel virtual machine for GPUs. *ACM SIGGRAPH Sketches, 184.*

[9] PGI Group. (2016). PGI_accelerator compilers openacc getting started guide. Retrieved from http://www.pgroup.com/doc/openacc_gs.pdf

[10] Basili, V., Asgari, S., Carver, J., Hochstein, L., Hollingsworth, J., Shull, F., & Zelkowitz, M. (2004). A pilot study to evaluate development effort for high performance computing. *University of Maryland Technical Report CS-TR-4588.*

[11] Hochstein, L., Basili, V., Zelkowitz, M., Hollingsworth, J., & Carver, J. (2005). Combining self-reported and automatic data to improve effort measurement. *Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 356-365).

[12] Ken, K., Charles, K., & Robert, S. (2004). Defining and measuring the productivity of programming

languages. *International Journal of High Performance Computing Applications, 18(4),* 441-448.

[13] Marc, S., & David. A. B. (2004). A framework for measuring supercomputer productivity. *International Journal of High Performance Computing Applications, 18(4)*, 417-432.

[14] Thomas, S. (2004). Productivity metrics and models for high performance computing. *International Journal of High Performance Computing Applications, 18(4),* 433-440.

[15] Tetsuya, H., Naoya, M., Satoshi, M., & Ryoji, T. (2013). CUDA vs OpenACC: Performance case studies with kernel benchmarks and a memory-bound cfd application. *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (pp. 136-143).

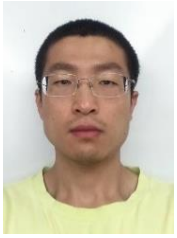
[16] Amit, S., Putt, S., Seyong, L., & Jeffrey, S. (2014). Evaluating performance portability of OpenACC. *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing* (pp. 51-66).

[17] Howell, D. D. (2002). *Statistical Methods for Psychology* (5th ed.). Duxbury: Pacific Grove.

[18] Trott, M. (2004). *The Mathematica GuideBook for Programming.* Springer-Verlag.

[19] Sawilowsky, S. (2002). Fermat, schubert, einstein, and behrens – fisher: The probable difference between two means when σ12≠σ22. *Journal of Modern Applied Statistical Methods, 1(2),* 461-472.

[20] Hochstein, L., Carver, J., Shull, F., Asgari, S., Basili,V., Hollingsworth, J., & Zelkowitz, M. (2005). Parallel programmer productivity: A case study of novice parallel programmers. *Proceedings of the ACM/IEEE conference on Supercomputing* (pp. 1-9).



Xuechao Li is a Ph.D candidate in the Department of Computer Science and Software Engineering at Auburn University, USA. Currently, his research work includes the optimization of high performance compilers, empirical study in software engineering and software modeling.

Po-Chou Shih is a Ph.D candidate of graduate institute of industrial and business management at National Taipei University of Technology (NTUT). His research interests are predicted by using artificial neural network and metaheuristic algorithms.

Xueqian Li is pursuing the master degree in the Department of Civil Engineering at Auburn University, USA. His research areas are geographic information system, the application of object-oriented programming languages on the groudwater system.

Cheryl Seals received her Ph.D degree from Virginia Tech in 2004. She is an associate professor in Auburn University's Department of Computer Science and Software Engineering. Her research areas of expertise are human computer interaction, user interface design, usability evaluation and educational gaming technologies. She also works with outreach initiatives to improve computer science education at all levels.