GPU Accelerated Applications with CUDA and OpenMP

Jiuhong Xiao, Zhen Li

Abstract— **The development of parallel computing systems motivates people to design more efficient parallel programming frameworks. CUDA and OpenMP are two widely used frameworks to support parallel computation on GPUs. To compare these frameworks, we implement some benchmark programs with them to evaluate the performance. The results show that CUDA has better scalability and less overhead of kernel launch, while OpenMP needs less effort for coding.**

I. INTRODUCTION

With the development of parallel computing systems, there is a growing trend to use GPUs to solve some specific problems with massive parallelism. CUDA [1] and OpenMP [2] are two widely used parallel programming frameworks. CUDA [1], introduced by NVIDIA in 2006, is a general programming framework for all NVIDIA GPUs. It provides methods for engineers to manage the computing process on GPUs and communication between the host and GPUs. OpenMP [2] is a framework for parallelism with shared memory on multi-processors. In the early years, it is mainly applied to multi-core CPU systems to improve concurrency. CUDA is an extension of C/C++ using specific function calls (cudaMalloc, cudaMemcpy, ...), while OpenMP uses the compiler directives (#pragma omp parallel, ...) for parallelism.

There were some results [3] showing that compared with CUDA, OpenMP is much easier for programming but has slightly worse performance. The programs of OpenMP in [3] were executed on x86 manycore processors but not GPUs at that time, while some recent updates shows that OpenMP has started to support offloading to accelerator devices like GPUs. Therefore, the performance of OpenMP on GPUs is worth exploring.

In this work, we compare the performance of CUDA and OpenMP on one GPU by developing some benchmark programs for two frameworks and run them on the specific GPU environment. We use CUDA profiler [1] to analyze their performance and do some comparisons to guide engineers to choose an appropriate framework for parallel programs. The major features of the two frameworks we used on the GPU are shown in Tab. I.

The rest of this work is organized as follows. In Section II we introduce the benchmark programs and environment configurations for experiments. In Section III we show the results of our experiments on benchmark programs. In Section IV we conclude the features of CUDA and OpenMP GPU programs.

The authors are with the New York University, Computer Science Department, New York, NY 10012, USA. email: {jx1190}@nyu.edu, {zl2972}@nyu.edu.

	CUDA	OpenMP
Target of parallelism	Device only	Host and device
Unified memory	Yes	Yes
Shared memory	Yes	Yes
Sychronization	Barrier	Barrier and reduction
Implementation	Compiler directives for C/C++	C/C++ extension

TABLE I: Major features of OpenMP and CUDA

II. BENCHMARKS AND ENVIRONMENTS

To evaluate the performance of CUDA and OpenMP on the GPU, we implemented several benchmark programs and built a specific environment to compile and run these programs.

A. Benchmark programs

For comparison, we implement five different GPU-friendly algorithms. Each of them has at least three versions (CPU version, GPU CUDA version and GPU OpenMP version) to compare their performance. The CPU version is used to verify the correctness of parallel implementations and others are used for performance evaluation.

1) **Matrix Multiplication (MM)** MM is a generic linear algebra application, with significant meaning in various fields related to high-performance computing. We implemented the version of square matrix multiplication with single-precision floating point numbers. This application also serves as a test suite when configuring GPU OpenMP environment.
2) **Advanced Encryption Standard (AES)** AES is a standard of a specific symmetric cryptography system, which is widely used various data encryption scenarios. We implemented AES-256 with counter mode in this project.
3) **Breadth First Search (BFS) [4]** BFS is an algorithm to traverse the nodes in one graph. It can be used to find the shortest path from source node to each reachable nodes. The implementation on the GPU [4] traverses visiting nodes to find their neighbors in each step.
4) **Kmeans [5]** Kmeans is an algorithm to partition data points into some clusters that the centroids of clusters have the least distance to belonging data points. The implementation on the GPU [6] uses reductions with explicit (CUDA) or implicit shared memory (OpenMP) [7] for the race condition.
5) **Back Propagation (BP) [8]** BP is a machine learning algorithm for feedforward neural networks. It is used to update the parameters of networks according to prediction error. We implemented a neural network for

handwriting digit recognition [9] to show the process of BP.

B. Enviroment

1) **Hardware Configuration** For evaluation of CUDA and OpenMP, we ran the benchmark programs on one of the CIMS servers with a single NVIDIA GPU. The related hardware parameters of the GPU is shown in Tab. II.

Name	GeForce GTX TITAN X
CUDA Capability	5.2
Max clock rate	1216 MHz
Number of cores	3072
L2 cache size	3 MB
Memory bandwidth	336.5 GB/sec
Amount of global memory	12213 MBytes
Warp size	32

TABLE II: The hardware configuration of NVIDIA GPU

2) **Software Configuration** We checked out and compared several compilers and finally decided to use LLVM tool chain 11.0.0. LLVM was compiled twice, during the first of which, we compiled it using GCC 8.2 with CUDA 10.1. The second time, we compiled it using clang generated in the previous step. (This process is known as bootstrapping) For the experiments, we used CUDA 10.2 and the LLVM tool chain built in above steps (with GCC tool chain 8.2).

III. EXPERIMENTS

To evaluate the performance of two frameworks, we compare the scalability, runtime overhead and programmability of the implementations on the GPU.

A. Performance and Scalability

We evaluate the performance and scalability by measuring the running time of programs with different problem sizes, different block size and the same number of iterations (if applicable). Fig. 1-5 show the results of experiments for benchmark programs. OpenMP will automatically choose the block size. In our cases, the default block size is 128. The curves in Fig. 1-5 are named according to the framework name and the block size (e.g. cuda_blksize=32 means the implementation uses CUDA and the block size is 32). The ranges of problem sizes and block sizes are chosen according to the features of different benchmark programs.

In **MM**, we did some extra tests by explicitly specifying block sizes for OpenMP version. As shown in Fig. 1, the CUDA version performs much better than that of OpenMP in **MM**. The main reason is that the OpenMP implementation has no way to exploit shared memory while the CUDA version uses shared memory to do the tiling, which greatly reduces memory access time. We also implemented a CUDA version without tiling (shared memory), see Fig. 6 in the appendix for the timings for that version. By comparing their performance with different block size, we find OpenMP almost maintain a fixed timing (so the series overlaps with each other), while we expect it to perform better as block size grows. Actually, it performs worse when block size

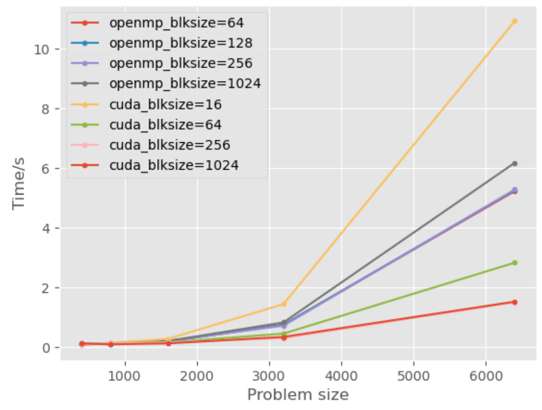

Fig. 1: The runnimg time of MM implementation

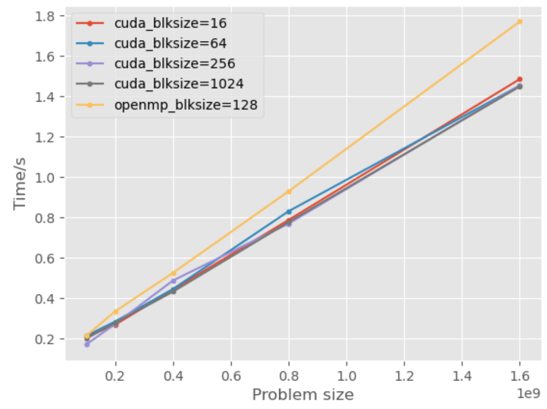

Fig. 2: The runnimg time of AES implementations

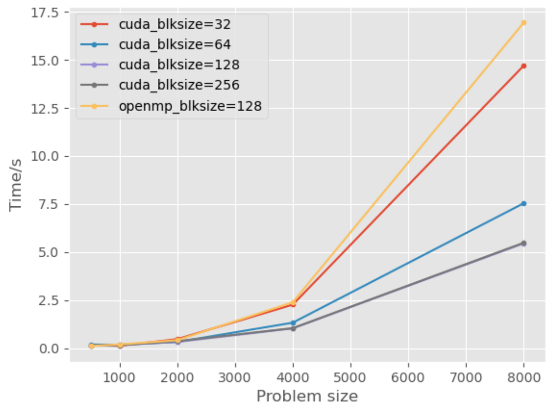

Fig. 3: The running time of BFS implementations

approaches 1024, revealing poor scalability. On the other side, CUDA does not perform well when block size is small while it does yield much better performance as block size grows.

For **AES**, CUDA version performs slightly better than

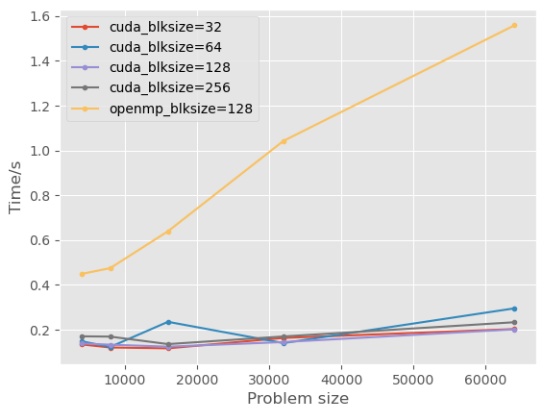

Fig. 4: The running time of Kmeans implementations

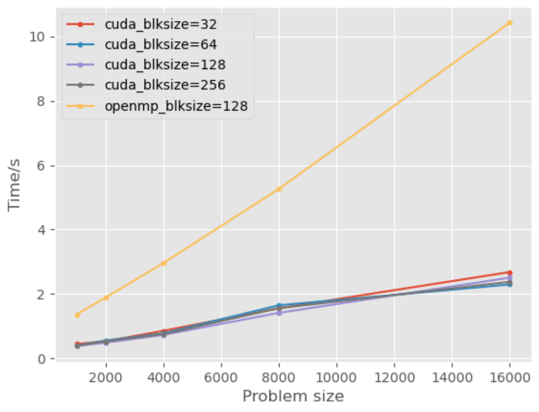

Fig. 5: The runnimg time of BP implementations

the OpenMP version, as shown in Fig. 2. CUDA version uses shared memory to cache the constant table while the OpenMP version does not. There are also other versions of CUDA implementations, the version in Fig. 2 has each thread operates at the AES block level, Fig. 7 has each thread operates at the word level and Fig. 8 has each thread operates at byte level. All of those figures reveals good scalabilities. As a common intuition, each thread in a block should operate at the word level to yield best performance, but in this application, we find the version operating at the AES block level has the best performance. This might due to the uniqueness of **AES** that there are lots of table lookups (that is why we used shared memory to cache the lookup tables), so there it will be expensive to synchronize warps. But, without caching constant table, CUDA version performs much worse than the OpenMP version. Despite not very sure why the program has such behavior, after ruling out several potential factors, we conjecture it was because Clang 11 generates a more optimized target code for than that of NVCC 10.2 when dealing with this uniqueness.

From Fig. 3 we can see CUDA **BFS** implementations outperform OpenMP **BFS** implementation. Please note that

	$T_{\text{first}}(\text{ns})$	$T_{\text{avg}}(\text{ns})$
CUDA	1.177×10^8	3.547×10^3
OpenMP	1.133×10^8	1.469×10^6

TABLE III: The overhead of kernel launch

for the similar performance of CUDA implementations with block size = 128 and 256, the purple curve that represents block size = 128 is covered by the gray curve. the results show that when the problem size becomes more than 4000, the implementations with large block size have much less running time than others. This is because they benefit from large block size to fully utilize the GPU.

Fig. 4-5 show that the performance of OpenMP is much worse than that of CUDA on **Kmeans** and **BP** algorithms. The difference comes from shared memory utilization. Due that OpenMP on GPUs has not implemented the explicit on-chip shared memory allocation, we have only limited methods to use shared memory. Currently OpenMP supports explicit off-chip shared memory and implicit on-chip shared memory by using **reduction** clause. Meanwhile, the reduction clause cannot be applied to variable-length arrays, which is required in **BP**. Therefore, the implementations of **Kmeans** and **BP** of OpenMP cannot benefit from shared memory, which worsen the performance. Besides, our results show that different block sizes have little influence on the running time because these block size can take enough on-chip shared memory. If the block size is too large, it may not allocate all data on shared memory.

B. *Overhead Analysis*

The analysis of runtime overhead is done by running empty kernels for many times and measure the running time. These kernel will do nothing so that the running time is exactly the overhead of kernel launch. In the test program, we launch kernels for 100 times for stable results. Tab. III shows the results, where T_{first} is the running time of the first launch and T_{avg} is the average time of the following launches. The first launch is not included in the calculation of T_{avg}.

From Tab. III, we can see T_{first} of CUDA is approximately equal to that of OpenMP, which means the overhead of the first launch of two frameworks are almost the same. However, T_{avg} of OpenMP is much more than that of CUDA. That means that OpenMP has larger overhead if we launch kernels for many times. This overhead will slow down the computation of kernels. Since we have many kernel calls in our benchmark program, this may be one reason why OpenMP has worse performance in Section III-A.

C. *Programmability*

To measure the programmer productivity, we follow the metrics discussed in [3] and [10]. We use the line of code (LOC), introduced in [11], to measure the line of code without comments in our implementations. Tab. IV shows the results of our measurement. LOC_{CUDA} and $\text{LOC}_{\text{OpenMP}}$ are defined as the line of code involving CUDA and OpenMP parallel computing respectively.The lines of kernel function

Benchmark Programs	LOC_{CUDA}	LOC_{OpenMP}
MM	46	11
AES	142	121
BFS	56	36
Kmeans	64	47
BP	156	77

TABLE IV: Programmability of CUDA and OpenMP

and any function calls or compiler directives related to CUDA or OpenMP are taken into account.

Tab. IV shows that OpenMP implementations need less effort than CUDA implementations. The difference is from memory management and kernel implementation. For memory management, even with unified memory, CUDA needs more effort for memory management by using function calls like **cudaFree**, while OpenMP uses **map** clauses to automatically allocate and free device memory [2]. For kernel implementation, CUDA needs programmers to rewrite specific kernel functions for the GPU, but with OpenMP we only needs add the CPU version code with compiler directives (# pragma omp parallel for, ...) to exploit parallelism on GPU. Consequently, OpenMP reduces the effort of coding for parallelism.

IV. CONCLUSION

In this work, we evaluate the performance of benchmark programs using CUDA and OpenMP on the GPU. The results show that the CUDA implementations have better scalability and less overhead of kernel launch. The OpenMP implementations, on the other hand, needs fewer lines of code for parallelism. We explore the influence of block sizes. If the block size is too small, it will affect the running time significantly. The difference of performance can come from unfinished implementation of explicit on-chip shared memory and more overhead of kernel launch. OpenMP gives programmers an efficient way to implement parallelism without much code effort, while programmers should use CUDA if they want to get better performance.

REFERENCES

[1] NVIDIA, *CUDA C Programming Guide*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[2] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[3] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, 2017, pp. 1–6.

[4] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International conference on high-performance computing*. Springer, 2007, pp. 197–208.

[5] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," 1967.

[6] A. Minnaar, "A cuda implementation of the k-means clustering algorithm," 2019. [Online]. Available: http://alexminnaar.com/2019/03/05/cuda-kmeans.html

[7] G. Gan, X. Wang, J. Manzano, and G. R. Gao, "Tile reduction: The first step towards tile aware parallelization in openmp," in *International Workshop on OpenMP*. Springer, 2009, pp. 140–153.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[10] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the upc language," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, 2004, p. 254.

[11] R. E. Park, "Software size measurement: A framework for counting source statements," Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Tech. Rep., 1992.

Appendix

For **MM** and **AES**, we have some additional CUDA implementations. They are shown in Fig. 6-8 to support our conclusions.

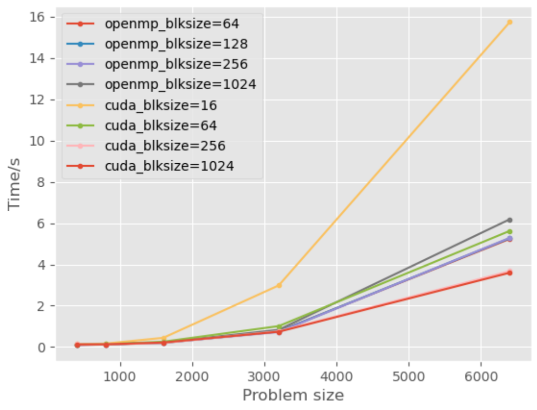

Fig. 6: The running time of CUDA without tiling

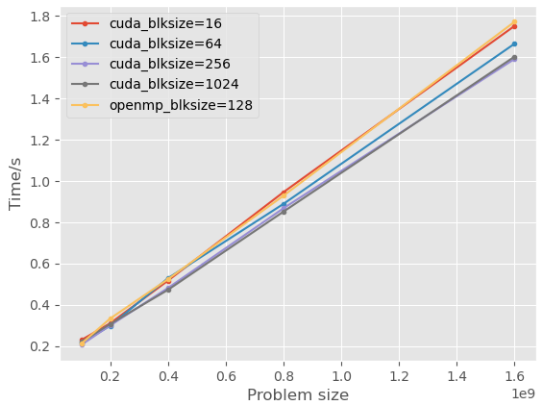

Fig. 7: The running time of AES with each thread operating at the word level

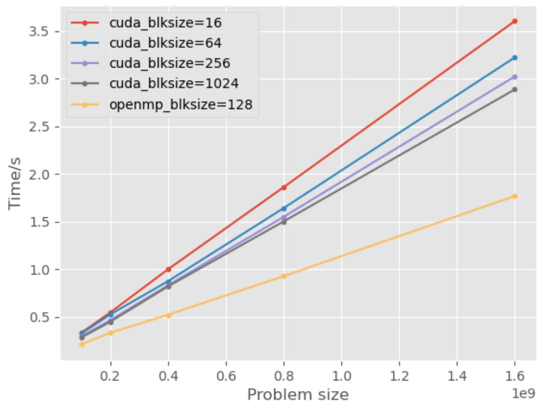

Fig. 8: The runnimg time of AES with each thread operating at the byte level