2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies

Improving Performance of GPU Specific OpenCL Program on CPUs

Qiang Lan, Changqing Xun, Mei Wen, Huayou Su, Lifang Liu, Chunyuan Zhang
Computer College
National University of Defense Technology
Changsha, China
{*lanqiang,xunchangqing,meiwen,shyou,lfliu,cyzhang*}*@nudt.edu.cn*

***Abstract*—OpenCL provides unified programming interface for various parallel computing platforms. The OpenCL framework manifests good functional portability, the programs can be run on platforms supporting OpenCL programming without any modification. However, most of the OpenCL programs are optimized for massively parallel processors, such as GPU, it's hard to achieve good performance on general multi-core processors without sophisticate modification to the GPU specific OpenCL programs. The major reason is the immense gap between CPU and GPU architecture. In this paper, we evaluate the performance portability of OpenCL programs between CPU and GPU, and analyse the reasons why GPU specific OpenCL programs are not fit for CPU. Based on the profiling, we proposed three optimization strategies for improving performance of GPU specific OpenCL programs on CPU, including increasing the granularity of task partition, optimizing the usage of memory hierarchy and block-based data accessing. In addition, we applied the proposed techniques on several benchmarks. The experimental results show that the performance of the optimized OpenCL programs achieve high performance in terms of speedup ratio from 2 to 4 on CPUs, when compared with their corresponding GPU specific ones.**

***Keywords*-OpenCL; functional portability; performance portability; GPU specific**

I. Introduction

In order to make use of the maximum performance of modern parallel processors, hardware vendors would provide special parallel programming model, such as CUDA [1] for GPU from NVIDIA corporation, StreamC/KernelC [2] and StreamIt [3] for stream processor, and CellSs [4] for Cell processor, etc. But too many programming models increase the burden of the programmers. To unify the programming of different processors, the industry released OpenCL [5], OpenCL is a cross-platform computing framework, and programs based on OpenCL can be run on any platform which supports OpenCL without any change in the programs. The current mainstream processor platforms have supported the OpenCL programming, such as GPU, CPU, Cell Processor, FPGA, etc.

Although OpenCL model has good functional portability, its performance portability is not satisfactory. In [6], the authors point out that the performance of the OpenCL kernels is not highly portable, they propose using auto-tuning to better explore these kernels parameter space using search heuristics. In [7], Komatsu discusses the difference between NVIDIA and AMD OpenCL implementations by comparing the performance of their GPUs for the same programs and the performance comparison shows that some parameters have to be optimized for each GPU to obtain its best performance. To run an OpenCL program on two platforms whose architectures are significant different from each other, such as CPU and GPU, and both achieving high performance is impossible. Because the OpenCL model abstracts all the target machines into a unified platform model, a unified storage model and a unified execution model. However the target machine in the real world varies in architecture, when programmers write OpenCL program for different platform, they need to adopt platform dependent optimization strategies. In [8], Seo etc. summarize different optimizations for CPU and GPU when implementing the NAS parallel benchmarks in OpenCL.

Currently, most of OpenCL programs are designed and optimized for GPU architecture, it's hard to gain good performance when running them on CPU. This paper analyses the problem of performance portability of the GPU-optimized OpenCL programs in terms of the hardware architecture of CPU and GPU, and then presents three optimization techniques of transforming OpenCL from GPU specific to CPU specific, including increasing the granularity of task partition, optimizing the usage of memory hierarchy and block-based data accessing. With these techniques in mind, we optimize several GPU specific OpenCL programs for CPU, and the experiment results show that the performance of the CPU-optimized OpenCL programs based on GPU-optimized has a 2.0∼4.0 time improvement relative to the GPU specific OpenCL programs both running on the Intel E5620 CPU.

The rest of this paper is organized as follows: Section II describes the background knowledge of OpenCL and the motivation of this paper. Section III summarizes three optimization techniques of transforming OpenCL programs from GPU specific to CPU specific. Section IV gives the experimental evaluation and section V is the conclusion and future work.

978-0-7695-4879-1/12 $26.00 © 2012 Crown Copyright
DOI 10.1109/PDCAT.2012.81

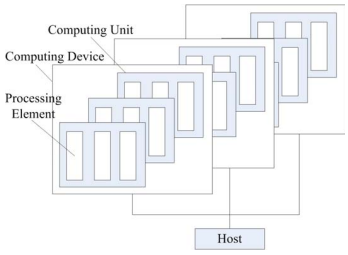


Figure 1. OpenCL platform model

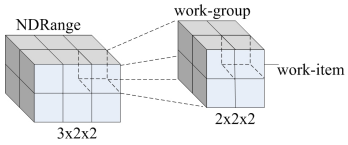


Figure 2. OpenCL execution model: NDRange

II. Background and Motivation

A. Overview of OpenCL

The OpenCL programming model consists of three parts, platform model, execution model and memory model.

The OpenCL platform model includes one host and several connected computing devices, each device has one or several computing units, and each computing unit includes one or several processing elements. Fig. 1 shows the OpenCL platform model. From the view of Nvidia platform, GPU is looked as a device, SM is looked as a computing unit, and SP represents the processing element.

In the OpenCL execution model, a very important concept is the index space of work-item(each thread which executes real calculation is called work-item), which is called NDRange, the space can be one-dimensional, two-dimensional or three-dimensional, the size of the NDRange determines the granularity of task partition, work-item within NDRange executes the same kernel code, but each work-item will access different data according to its respective ID, which is similar to the STMD execution mode of CUDA. Fig. 2 shows an example of NDRange, it is three dimensional, and there are a total of 3x2x2 work groups, and each group has 2x2x2 work items.

In OpenCL, the memory model of a target machine is abstracted into four level, they are global memory, constant memory, local memory and private memory. Global memory can be read and written by all work-items, constant memory can be only read by work-items, local memory can be shared by all the work-items within a work-group, each work-item has its own private memory which can not be accessed by other work-items, private memory represents the registers in real hardware. Fig. 3 shows the OpenCL memory model.

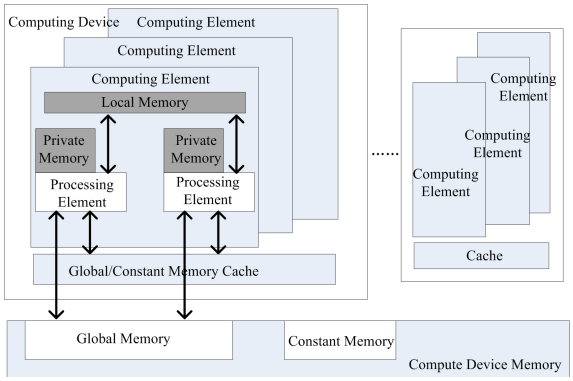


Figure 3. OpenCL memory model

B. Motivation

To explain the problem of performance portability existing in the OpenCL model, we choose several GPU specific benchmarks from SNU_NPB-1.0.2 [8] implemented in OpenCL. We run each benchmark on Nvidia C2050 GPU and Intel E5620 CPU respectively. In order to illustrate the performances of GPU specific OpenCL programs running on E5620 CPU, we also run optimized OpenMP versions of those benchmarks on E5620 CPU. Analyzing the results of the above experiments as shown in Fig. 4, we find that the performances of benchmarks in OpenCL version running on GPU are pretty good, while on CPU are poor and only reaching 40%∼70% of that in OpenMP version. That is to say GPU specific OpenCL programs do not match the CPU architecture well. We find that the difference between performance on CPU and GPU mainly because of the difference in architecture between CPU and GPU. The first six rows in table 1 show some characteristics of GPU specific OpenCL programs, including the size of NDRange, the usage of the local memory and the data access manner. And the rest rows in table 1 describe the hardware property and the optimized data access manner of CPU and GPU. The differences between architecture of CPU and GPU exist mainly in the number of processors, optimized memory hierarchy and data access manner. We can see that the NDRange of the GPU specific benchmarks are all large to make full use of computing resources in GPU, all the GPU specific benchmarks use the local memory to improve the memory bandwidth and adopt the coalesced access manner to improve the speed of accessing memory which can either be global memory or local memory. So the GPU specific OpenCL programs run well on GPU while bad on CPU.

Presently many OpenCL programs are oriented to the GPU platform. Researches on optimization techniques of transforming OpenCL programs from GPU specific to CPU specific are of great significance. This paper summarizes three transforming techniques which are only implemented manually.

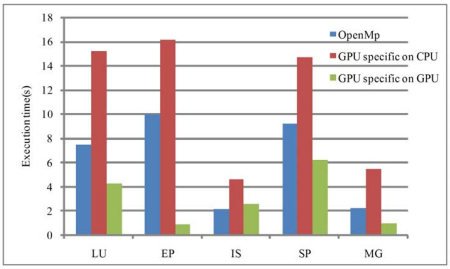


Figure 4. Performance portability of OpenCL

Table I
CHARACTERISTICS OF THE BENCHMARKS AND PLATFORMS

Benchmarks	NDRange	Local memory usage	Data access manner
LU	(192,160)	Yes	coalesced access
EP	65536		
IS	2^{27}		
SP	(192,162,162)		
MG	(66048,256)		
Platforms	processors	Local memory	optimized data access manner
Nvidia C2050	448	shared memory	coalesced access
Intel E5620	16	main memory	block access

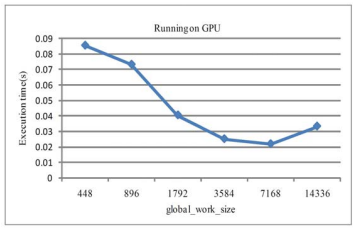
III. Optimization Techniques of Transforming OpenCL Programs from GPU Specific to CPU Specific

This section takes matrix vector multiplication as an example to describe three optimization techniques of transforming OpenCL programs from GPU specific to CPU specific, including increasing the granularity of task partition, optimizing the usage of memory hierarchy and effective data access manner.

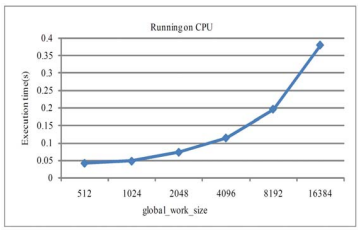
A. Increasing the Granularity of Task Partition

There are many computing units and processing elements with weak power in GPU device, while there are few computing units and processing elements with strong power in CPU device. So in the GPU specific OpenCL programs, the index space of work-items is as large as possible. For GPU, work-items within work-group are executed in the SM, work-groups switch frequently on the SM through hardware mechanism to hide the latency of data access. While for CPU, the overhead of work-items switching is large, because the switching job is completed by software, so it is an optimization way to set the index space of work-items as large as possible in the GPU specific OpenCL programs, and as the same size as the number of the processor cores within CPU.

In the program of matrix vector multiplication, we set up 6 kinds of sizes of index space, chart (a) and (b) in Fig. 5 respectively show the performance impact of the programs on GPU and CPU with the changes of the size of index space. The abscissas of charts in Fig. 5 stand for the size



(a) The effect of index space size on GPU



(b) The effect of index space size on CPU

Figure 5. The performance effect of index space size on GPU and CPU

of global work size. As is shown in Fig. 5, with the size of global work size enlarged, the performance of matrix vector multiplication program running on GPU gradually ascends until the global work size reaches to 7168, and the performance begins to descend when the global work size reaches to 14336, since the computing resources have been fully used, while the performance on CPU gradually descends.

B. Optimizing the Usage of Memory Hierarchy

The memory model of OpenCL is similar to the hardware memory hierarchy of GPU. The local memory of OpenCL model can be mapped to the share memory of GPU. When we write an OpenCL program for a GPU device, if a small data block in global memory is accessed frequently by all work-items within one work-group, we should load the data block from global memory to local memory first to fully utilize the memory bandwidth of GPU, while for CPU, local memory is actually mapped to the main memory of CPU, using local memory means that data would be transferred within main memory, and the operation caused by synchronization because of the usage of local memory would cause extra overhead. In order to improve the performance of the GPU specific OpenCL program on CPU, we should remove the utilization of local memory. The left diagram of Fig. 6 shows part of codes of matrix vector multiplication which are optimized for GPU, where local memory is used. The work-items in the same work-group would execute a synchronization statement after storing the result to local memory, which would reduce the performance for CPU. The right diagram of Fig. 6 is the optimized kernel codes in which the usage of local memory is removed. The dominate effect is that every work-group would store the result to global memory directly. Fig. 7 shows the performance of

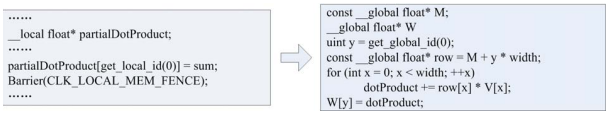


Figure 6. Transforming GPU specific codes with using local memory to CPU specific code with using global memory

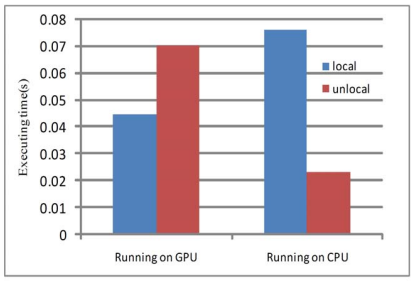


Figure 7. The performance effect on CPU and GPU after using the technique of optimizing the usage of memory hierarchy

the GPU specific and CPU specific both running on CPU and GPU, according to the result, after the utilization of local memory is removed, the performance of CPU specific improve 3.26 time relative to GPU specific when both of them are running on CPU while a 1.57 time depression on GPU.

C. Effective Data Access Manner

In CUDA programming model, 16 threads from a thread block, which are called a half-warp, are executed simultaneously. These threads access the global memory with aligned access manner. In OpenCL programs, work-items of a work group access the global memory and the local memory with aligned access manner as well. Aligned access manner effectively improves the performance of programs running on GPU, but contrarily, it degrades performance when programs run on CPU. The reason is that for CPU, data to be accessed by a work group are expected to locate in a continuous area in order to make the best use of caches. Therefore, in order to optimize GPU specific OpenCL programs for CPU, aligned access manner should be avoided. A demonstration of aligned access and unaligned access is shown in Fig. 8, where threadN denotes the work-item with the largest ID number in a work group. Fig. 9 contains parts of GPU specific kernel codes and optimized kernel codes for CPU. In the GPU specific kernel codes with aligned access, work-items from the same work group cooperate to perform a vector multiplication. Every work item takes local_work_size which is the number of the work-items in one work-group as the interval to access vector elements. However, in the optimized kernel codes for CPU, the multiplication of a row vector and the column vector is accomplished by a single work-item, data needed for each work-item is located in a continuous block.

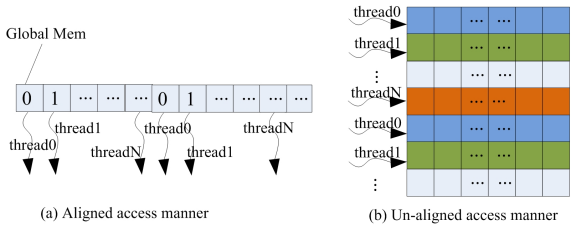


Figure 8. Aligned and unaligned access manner

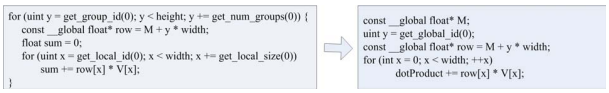


Figure 9. Transforming GPU specific codes with aligned access manner to CPU specific code with block access manner

Performance of matrix vector multiplications implemented with aligned access and unaligned access on both GPU and CPU are shown in Fig. 10. Unaligned access manner leads to 2.34 time performance improvement relative to the aligned access manner both running on CPU, but a 1.63 time performance depression on GPU.

IV. Experimental Evaluation

In this section, we optimize the GPU specific OpenCL programs for CPU using techniques introduced in section III. We choose several benchmarks which are all optimized for Nvidia's GPU, the information about benchmarks are listed in table 2, most of the benchmarks are from Nvidia SDK 4.0, except that IS is from SNU_NPB-1.0.2, and Stenciled2D is from shoc-1.1.2. In the table, IGTP is shorted for increasing the granularity of the task partition, OUMH for optimizing the usage of memory hierarchy and EDAM for Effective Data Access Manner.

We do our experiments on platforms of Nvidia's C2050 GPU and Intel's E5620 CPU. For each benchmark, there are two kinds of implementations, one is GPU specific OpenCL programs, and the other is CPU specific OpenCL programs. We run them both on E5620 CPU and C2050 GPU, the results are show in Fig. 11.

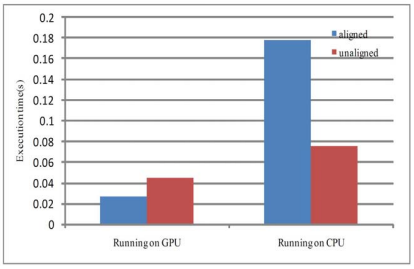


Figure 10. The performance effect on CPU and GPU after using the technique of effective data access manner

Table II
TECHNIQUES ADOPTED IN BENCHMARKS TESTED IN THE EXPERIMENTS

Benchmark	Source	IGTP	OUMH	EDAM
IS	SNU_NPB-1.0.2	✗	✓	✓
VectorAdd	Nvidia SDK 4.0	✓	✗	✗
Blackschole	Nvidia SDK 4.0	✗	✗	✓
matrixMult	Nvidia SDK 4.0	✗	✓	✗
copyArray	Nvidia SDK 4.0	✗	✗	✓
MatVecMul	Nvidia SDK 4.0	✓	✓	✓
Stencile2D	shoc-1.1.2	✗	✓	✗

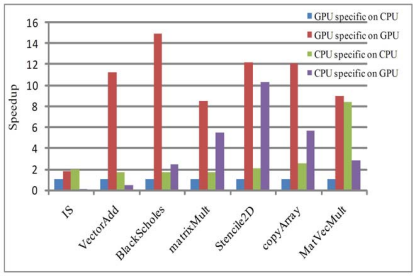


Figure 11. The performance comparison between the GPU specific and CPU specific running on CPU and GPU

GPU specific on GPU in Fig. 11 represents the GPU specific OpenCL program running on GPU, CPU specific on CPU represents the CPU specific OpenCL programs running on CPU and so on. We regard the result of the GPU specific on CPU as the baseline. After applying our optimizing techniques to GPU specific OpenCL programs, we gain nearly 2 time performance improvement of the CPU specific OpenCL programs running on CPU relative to the performance of GPU specific OpenCL programs on CPU. Meanwhile the performances of optimized CPU specific OpenCL programs decrease obviously relative to the GPU specific on GPU, especially for VectorAdd benchmark, because we adopt the first technique introduced in previous section to optimize the GPU specific program, decreasing the size of the index space which is good for CPU, while bad for GPU, because there are not enough threads running on the computing resources on GPU.

V. Conclusions and Future Work

In this paper, we evaluate the performance portability of OpenCL programs, the result shows that the performance portability of OpenCL is not satisfying. And we summarize three optimization techniques of transforming OpenCL programs from GPU specific to CPU specific. We also evaluate the performance of the GPU specific and CPU specific OpenCL programs both running on Intel's E5620 CPU and Nvidia's C2050 GPU. We achieve a good performance improvement in the CPU specific relative to GPU specific running on CPU.

The work of transforming the GPU specific OpenCL programs to CPU specific OpenCL programs is done manually currently, in order to ease the work of programmers, our future work will focus on developing an automatic transforming tool to implement our present work.

Acknowledgment

This research is supported by the Nature Science Foundation of China under NSFC No.61033008, 60903041 and 61103080, and Research Found for the Doctoral Program of Higher Education of China under SRFDP No. 20104307110002, Hunan Provincial Innovation For Postgraduate under No.CX2010B028, Found of Innovation in Graduate School of NUDT under No.B100603.

References

[1] *CUDA Application Design and Development*. Elsevier Inc., 2011.

[2] P. R. Mattson, "A programming system for the imagine media processor," Master's thesis, Stanford University, 2002.

[3] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*.

[4] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*.

[5] *OpenCL Programming Guide*. Pearson Education,Inc., 2012.

[6] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance portable solution for multi-platform GPU programming," *Parallel Computing*, 2012.

[7] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating Performance and Portability of OpenCL Programs," in *The Fifth International Workshop on Automatic Performance Tuning*.

[8] S. Seo, G. Jo, and J. Lee, "Performance Characterization of the NAS Parallel Benchmarks in OpenCL," in *2011 IEEE International Symposium on Workload Characterization(IISWC)*.