Computer Physics Communications 182 (2011) 266–269

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters ☆

Chao-Tung Yang *, Chih-Lin Huang, Cheng-Fang Lin

Department of Computer Science, Tunghai University, Taichung City, 40704, Taiwan

ARTICLE INFO

Article history:
Received 1 March 2010
Received in revised form 18 June 2010
Accepted 25 June 2010
Available online 16 July 2010

Keywords:
CUDA
GPU
MPI
OpenMP
Hybrid
Parallel programming

ABSTRACT

Nowadays, NVIDIA's CUDA is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions – a hierarchy of thread blocks, shared memory, and barrier synchronization. This model has proven quite successful at programming multithreaded many core GPUs and scales transparently to hundreds of cores: scientists throughout industry and academia are already using CUDA to achieve dramatic speedups on production and research codes. In this paper, we propose a parallel programming approach using hybrid CUDA OpenMP, and MPI programming, which partition loop iterations according to the number of C1060 GPU nodes in a GPU cluster which consists of one C1060 and one S1070. Loop iterations assigned to one MPI process are processed in parallel by CUDA run by the processor cores in the same computational node.

1. Introduction

Nowadays, NVIDIA's CUDA [1] is a general purpose scalable parallel programming model for writing highly parallel applications. It provides several key abstractions – a hierarchy of thread blocks, shared memory, and barrier synchronization. This model has proven quite successful at programming multithreaded many core GPUs and scales transparently to hundreds of cores: scientists throughout industry and academia are already using CUDA [1] to achieve dramatic speedups on production and research codes.

This paper proposes a solution to not only simplify the use of hardware acceleration in conventional general purpose applications, but also to keep the application code portable. In this paper, we propose a parallel programming approach using hybrid CUDA, OpenMP and MPI [3] programming, which partition loop iterations according to the performance weighting of multicore [4] nodes in a cluster. Because iterations assigned to one MPI process are processed in parallel by OpenMP threads run by the processor cores in the same computational node, the number of loop iterations allocated to one computational node at each scheduling step depends on the number of processor cores in that node.

In this paper, we propose a general approach that uses performance functions to estimate performance weights for each node. To verify the proposed approach, a cluster with hybrid CUDA was built in our implementation. Empirical results show that in the hybrid CUDA clusters environments, the proposed approach improved performance over all previous schemes.

The rest of this paper is organized as follows. In Section 2, we introduce several typical and well-known parallel programming schemes. In Section 3, we define our model and describe our approach. Our system configuration is then specified in Section 4, and experimental results for three types of application program are presented. Concluding remarks and future work are given in Section 5.

2. Background review

2.1. CUDA programming

CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing [2] architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. CUDA architecture supports a range of computational interfaces including OpenGL [9] and Direct Compute. CUDA's parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions.

☆ This work is supported in part by the National Science Council, Taiwan, under grants Nos. NSC 98-2220-E-029-004- and NSC 99-2220-E-029-004-.

* Corresponding author. Tel.: +886 4 23590415; fax: +886 4 23591567.
E-mail address: ctyang@thu.edu.tw (C.-T. Yang).

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2010.06.035

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel. Such a decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables transparent scalability since each sub-problem can be scheduled to be solved on any of the available processor cores: A compiled CUDA program can therefore execute on any number of processor cores, and only the runtime system needs to know the physical processor count.

2.2. OpenMP programming

In contrast, Open Multi-Processing (OpenMP) [6], a kind of shared memory architecture API [3,5], provides a multithreaded capacity. A loop can be parallelized easily by invoking subroutine calls from OpenMP thread libraries and inserting the OpenMP compiler directives. In this way, the threads can obtain new tasks, the un-processed loop iterations, directly from local shared memory.

OpenMP is an open specification for shared memory parallelism. The basic idea behind OpenMP is data-shared parallel execution. It consists of a set of compiler directives, callable runtime library routines and environment variables that extend FORTRAN, C and C++ programs. OpenMP is portable across the shared memory architecture. The unit of workers in OpenMP is threads. It works well, when accessing shared data costs you nothing. Every thread can access a variable in shared cache or RAM.

The OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multi-processing programming in C, C++ and FORTRAN on much architecture, including UNIX and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

2.3. MPI programming

Message Passing Interface (MPI) is a specification for message passing operations. It defines each worker as a process. MPI is currently the de-facto standard for developing HPC applications on distributed memory architecture. It provides language bindings for C, C++, and FORTRAN. MPI [7] offers portability, standardization, performance, and functionality, and includes point-to-point message passing and collective (global) operations, all scoped to user-specified groups of processes. MPI provides a substantial set of libraries for writing, debugging, and performance-testing distributed programs. Our system currently uses MPICH, a portable implementation of the MPI standard. MPICH is a freely available, portable implementation of MPI, a standard for message-passing for distributed-memory applications used in parallel computing [2]. MPICH is Free Software and is available for most flavors of UNIX (including Linux and Mac OS X) [10] and Microsoft Windows. Moreover, MPICH [8] is a developed program library.

The advantage for the user is that MPI is standardized on many levels. For example, since the syntax is standardized, you can be sure your MPI code will execute under any MPI implementation running on your architecture. Since the functional behavior of MPI calls is also standardized, your MPI calls should behave the same regardless of the implementation, thus guaranteeing the portability of your parallel programs. Performance, however, may vary from implementation to implementation.

The MPI library is often used for parallel programming [11] in cluster systems because it is a message-passing programming language. However, MPI is not the most appropriate programming language for multicore [4] computers because even when there

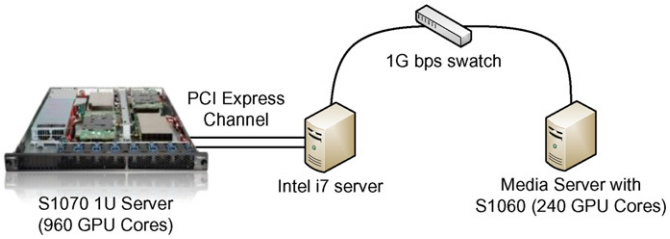

Fig. 1. System model: The hybrid CUDA GPU cluster.

are still many tasks assigned to overloaded slave processors remaining in shared memory, other slave MPI processors on the same computing node cannot access the tasks. Instead, all slave processors must communicate directly with the master MPI processor to obtain new tasks. In large cluster systems, the master processor may become a bottleneck on system performance because of excessive amounts of communication. The cluster computations exploit message-passing, because computers in cluster have distributed memory. When one process needs data from another one then you should manage data passing over the network. It is time-consuming operation. So, if you want to write such hybrid program, you should implement data broadcasting operations (e.g. MPI_Bcast from MPI) for each data access in OpenMP. This will kill parallel performance at all.

3. System hardware

3.1. Tesla C1060 GPU computing processor

The NVIDIA® Tesla™ C1060 transforms a workstation into a high-performance computer that outperforms a small cluster. This gives technical professionals a dedicated computing resource at their desk-side that is much faster and more energy-efficient than a shared cluster in the data center. The NVIDIA® Tesla™ C1060 computing processor board which consists of 240 cores is a PCI Express 2.0 form factor computing add-in card based on the NVIDIA Tesla T10 graphics processing unit (GPU). This board is targeted as high-performance computing (HPC) solution for PCI Express systems. The Tesla C1060 [15] is capable of 933 GFLOPs/s [13] of processing performance and comes standard with 4 GB of GDDR3 memory at 102 GB/s bandwidth.

A computer system with an available PCI Express $\times 16$ slot is required for the Tesla C1060. For the best system bandwidth between the host processor and the Tesla C1060, it is recommended (but not required) that the Tesla C1060 be installed in a PCI Express $\times 16$ Gen2 slot. The Tesla C1060 is based on the massively parallel, many-core Tesla processor, which is coupled with the standard CUDA C programming [14] environment to simplify many-core programming.

3.2. Tesla S1070 GPU computing system

The NVIDIA® Tesla™ S1070 [12] computing system speeds the transition to energy-efficient parallel computing [2]. With 960 processor cores and a standard C compiler that simplifies application development, Tesla S1070 scales to solve the world's most important computing challenges – more quickly and accurately. The NVIDIA® Tesla™ S1070 Computing System is a 1U [12] rack-mount system with four Tesla T10 computing processors. This system connects to one or two host systems via one or two PCI Express cables. A Host Interface Card (HIC) [5] is used to connect each PCI Express cable to a host. The host interface cards are compatible with both PCI Express 1x and PCI Express 2x systems.

The Tesla S1070 GPU computing system is based on the T10 GPU from NVIDIA. It can be connected to a single host system via

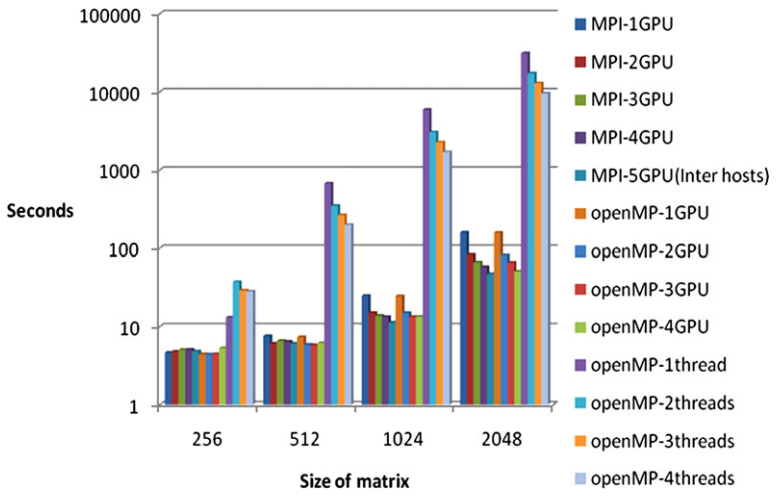

Fig. 2. Matrix multiplication with problem sizes from 256 to 2048. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

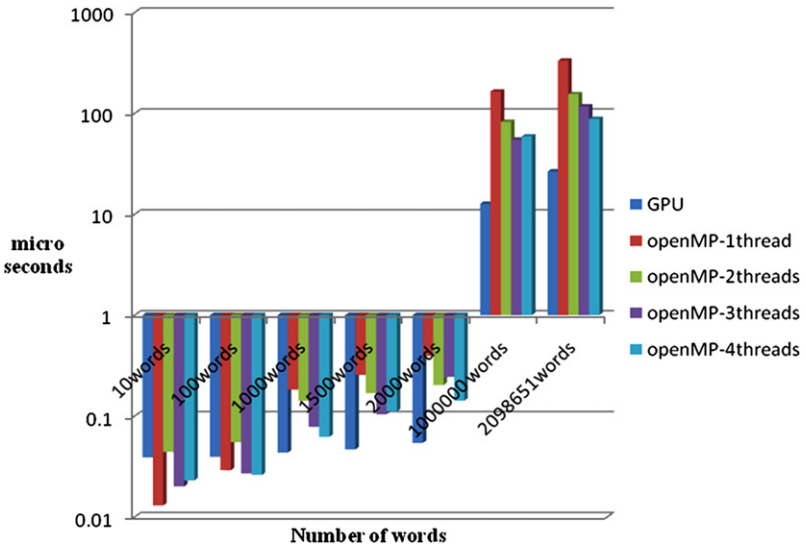

Fig. 3. Md5 hashing on 10 to 2,098,651 words. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

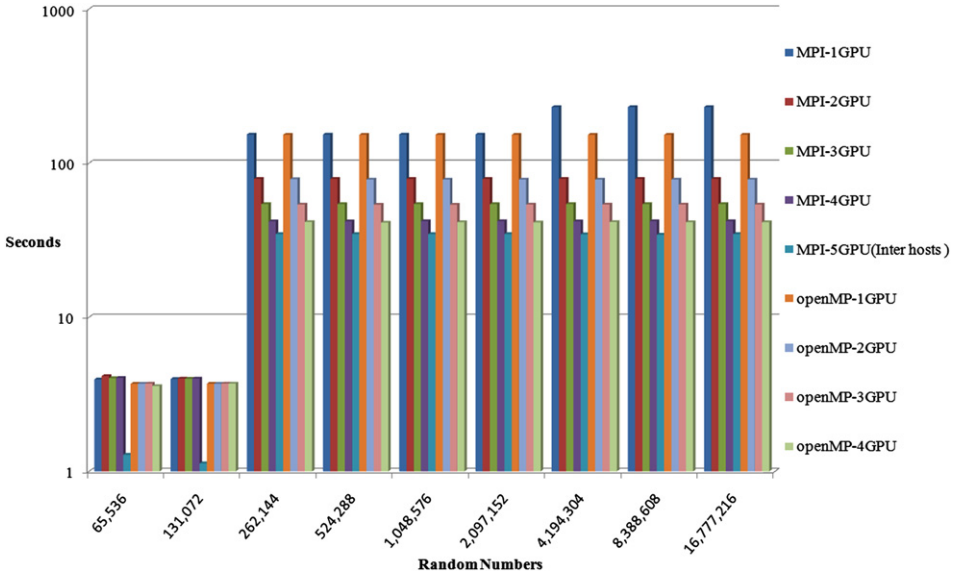

Fig. 4. Sorting numbers 640 times from 65,536 to 16,777,216 floating point numbers. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

two PCI Express connections to that host, or connected to two separate host systems via one PCI Express connection to each host. Each NVIDIA switch and corresponding PCI Express cable connects to two of the four GPUs in the Tesla S1070. If only one PCI Express cable is connected to the Tesla S1070, only two of the GPUs will be used. To connect all four GPUs in a Tesla S1070 to a single host system, the host must have two available PCI Express slots and be configured with two cables.

3.3. System model and approach

The system model is presented in Fig. 1, a hybrid CUDA GPU cluster is built with two GPU Servers as shown as S1070 and S1060, which connected with a Gigabit Swatch. The S1070 1U server attached to Intel i7 Server is connected with double PCI express channel for enhancing the internal-communication. We take the Intel Core i7 which contains four cores as the control group for comparing with the performance for GPU and CPU. In order to execute MPI and OpenMP application by CUDA, the simplest way forward for combining MPI and OpenMP upon CUDA GPU is to use the CUDA compiler-NVCC [16] for everything. The NVCC compiler wrapper is somewhat more complex than the typical mpicc compiler wrapper, so it's easier to translate MPI and OpenMP codes into.cu and compile with NVCC than the other way around.

4. Experimental results

We built a hybrid CUDA GPU cluster consisting of one Tesla C1060 and a Tesla S1070, each with Gigabit Ethernet NIC interconnected via a D-LINK DGS-3100-24 Gigabit switch. To verify our approach, illustrate our cluster environment, and describe the terminology for our application, we implemented programs with MPI/OpenMP for execution on our testbed. We then verify the performance of our scheme upon the hybrid CUDA GPU cluster to solve problems in Matrix Multiplication, MD5 and Bubble Sorting. From Figs. 2 to 4, we take log of 10 at execution time to emphasize the differences. Fig. 2 shows that the performance of GPU on processing the massively parallel execution as the application of Matrix Multiplication form 256 to 2048. In this case, the execution results on MPI and OpenMP upon GPU are close. Comparing to the performance between GPU and CPU with this instance, the performance of GPU obviously exceeds GPU. With the small problem size such as 256 by 256 Matrix Multiplication; the speedup of performance is negligible. The degree of speedup accumulates with the increasing of the problem size. Also, Fig. 3 reveals that single GPU presents better performance than single CPU with multiple threads on MD5 hashing computation. Again, the performance of GPU could not be observed in the small problem size due to the constraint on the internal overhead of starting execution. Finally, Fig. 4 shows that the comparison of performance on multiple GPU with MPI and OpenMP. The results of MPI and OpenMP are approximate to each other.

5. Conclusion

In conclusion, we propose a parallel programming approach using hybrid CUDA and MPI programming, which partition loop iterations according to the number of C1060 GPU nodes in a GPU cluster which consists of one C1060 and one S1070. During the experiments, loop iterations assigned to one MPI process are processed in parallel by CUDA run by the processor cores in the same computational node. The experiments reveal that the hybrid parallel multicore GPU currently processing with OpenMP and MPI as a powerful approach of composing high performance clusters.

References

[1] Download CUDA, http://developer.nvidia.com/object/cuda.htm.
[2] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, S. Tureka, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Comput. 33 (Nov. 2007) 685–699.
[3] P. Alonso, R. Cortina, F.J. Martínez-Zaldívar, J. Ranilla, Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA, J. Supercomputing, in press, doi:10.1007/s11227-009-0360-z, SpringerLink Online Date: Nov. 18, 2009.
[4] Francois Bodin, Stephane Bihan, Heterogeneous multicore parallel programming for graphics processing units, J. Sci. Programming 17 (4) (2009) 325–336, doi:10.3233/SPR-2009-0292.
[5] Specification Tesla S1070 GPU Computing System, http://www.nvidia.com/docs/IO/43395/SP-04154-001_v02.pdf.
[6] OpenMP Specification, http://openmp.org/wp/about-openmp/.
[7] Message Passing Interface (MPI), http://www.mcs.anl.gov/research/projects/mpi/.
[8] MPICH, A Portable Implementation of MPI, http://www.mcs.anl.gov/research/projects/mpi/mpich1/index.htm.
[9] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1, 6th edition, Addison–Wesley Professional, Reading, MA, ISBN 0321481003, 2007.
[10] Intel 64 Tesla Linux Cluster Lincoln webpage, [online] available: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/, 2008.
[11] R. Dolbeau, S. Bihan, F. Bodin, HMPP: A hybrid multi-core parallel programming environment, in: The Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), Boston, Massachussets, USA, October 4th, 2007, http://www.caps-entreprise.com/upload/ckfinder/userfiles/files/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf.
[12] The NVIDIA Tesla S1070 1U computing system – scalable many core supercomputing for data centers, http://www.nvidia.com/object/product_tesla_s1070_us.html.
[13] Top 500 supercomputer sites, what is Gflop/s, http://www.top500.org/faq/what_gflop_s.
[14] NVIDIA CUDA programming guide, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.
[15] NVIDIA Tesla C1060 Computing Processor, http://www.nvidia.com/object/product_tesla_c1060_us.html.
[16] The CUDA Compiler Driver NVCC, http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc_2.0.pdf.