International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 9 – No.7, September 2015 – www.ijais.org

Auto Conversion of Serial C Code into Cuda-C-Code for Faster Execution Utilizing GPU

Dipak V. Patil
Department of Computer Engineering
G.E.S's R.H. Sapat College of Engineering,
Management and Research, Nasik, M.S., India

ABSTRACT

The primary accusative of this implementation is to expand the use of NVIDIA Graphics Processing Units (GPUs) to accelerate the all-purpose applications outside the graphics arena. CUDA is a programming language particularly designed for parallel computation to work. Now a day, C programming is glaringly used in industries to develop general purpose applications. Normally, a C program instruction executes sequentially and do not support data parallel computation, it increases the time complexity of a program. CUDA renders C like interface, configured for programming NVIDIA GPU which supports parallel computation of different parts of same instructions on different cores of GPU. For ordinary programmers it is very sticky to write CUDA programs because it involves various irksome tasks. Today, most of the machines come with NVIDIA graphics card which contains GPU having numerous processing cores. It is mainly used during execution of gaming, graphics and image processing kind of applications. It remains otiose during execution of general-purpose applications which results into surplus time. To properly employ the potential of available GPU cores on graphics cards for accelerating execution of applications outside graphics domain, the system implemented here provides an automatic tool that converts the directive based sequential C program and generates equivalent parallel CUDA program which will significantly enhance the speed of execution of program with help of parallel processing support. The C programmers can use this tool to enhance the speed of execution of their applications by transforming their directive based C code to CUDA C code. This tool provides simple user interface and helps to enhance the performance of the system.

Keywords

Parallel Computing, Serial Computing, CUDA, GPU, HPC

1. INTRODUCTION

Compute Unified Device Architecture (CUDA) is a parallel computing system and API designed and developed by NVIDIA. It permits to use a CUDA-enabled graphics cards (GPU) for all purpose processing a methodology recognized as GPGPU. Basically graphics card contains GPU having multiple processing units are used for performing computer graphics related applications like computer gaming, animation and playing movies.

The GPU [1] remains idle during running of general purpose applications. To enhance the system performance, the computing capability of the GPU available can get properly utilized during execution of applications outside the graphics domain. Brook [2] supports simple data-parallel statements and promotes the use of the GPU as a co-processor. GPUs have recently increased beamy popularity among investigators and creators as accelerators for applications outside the domain of conventional Computer Graphics [3]. This evolution, known as General-Purpose computing on the GPU or GPGPU, Largely outcomes from the big improvements in GPU programmability. Everyone is interested to get the fast response from computer for this purpose evolution of HPC is needed. Immediate outcome is the need of society whenever large amount of data processing taken place. A emblematic GPU is a multi-core architecture with each core capable of running thousands of threads concurrently. Hence, an petition with a large quantity of parallelism can use GPUs to understand essential performance benefits. GPUs have recently appeared as almighty platform for general purpose high performance computing. Programming for GPU is a complex task as compared to programming CPU and parallel programming models such as shared memory[4].

The CUDA is a programming language specifically designed to program NVIDIA GPUs. It is an enlargement to C, CUDA has rapidly become popular and drawn more and more non-graphics computer programmer to port existing applications to CUDA[5,6].

However, experience shows that the porting process is highly challenging task. In specific, CUDA places on the coder the burden of packaging GPU code in isolated functions of explicitly carry off data transfer between the host memory and several GPU memories, and of manually optimizing the utilization of the GPU memory. The experiment involve different schemes of partitioning computation among GPU threads, of optimizing single-thread code, and of utilizing the GPU memory. As a outcome, the coder has to make important code alterations, perhaps many times, before attaining coveted performance. Practically this operation is very irksome and error prone. Many of the tasks entangled are mechanical and can be automated by a this system, system is an attempt to attain such automation. In this work, research introduces methods for transfer the load from CPU to GPU for HPC[7][8].

Hardware accelerators, such as GPGPUs, are probable parallel platforms for HPC. While a GPGPU renders a inexpensive, highly parallel method to application coders, its programming complexity poses an essential challenge for programmer. Even though the CUDA[9] programming model, recently novice by NVIDIA, offers a more user friendly interface, programming GPGPUs is tough and error prone, compared to programming CPUs and parallel programming models such as OpenMP[10].

GPGPUs have lately appeared as compelling conveyance for general purpose High Performance Computing. CUDA programming framework [13,14] from NVIDIA offers improved programmability for common computing, hence, the manual development of high performance codes in CUDA is more participating than in other parallel programming [15] models such as OpenMP. Manual procedure places burden on programmer to port computation on processors. It is not bendable to change with programs, input problem sizes and software or hardware design[11,12]. There are some systems available in that programmer have to specify manually the region of code to make parallel.

2. EXISTING SYSTEM

Amlpe of work has been done in enhancing the software support for GPGPU programming. The first group extends CUDA support to other programming languages [1], such as PyCUDA for Python, jCUDA[26] for Java and CUDA Fortran [19] to be jointly developed by PGI and NVIDIA.

The second group of related work provides high level abstraction of CUDA programming terms of compiler directives,[10] propose a compiler framework for translating an OpenMP program to a CUDA program. The main contributions of this work include an interpretation of OpenMP semantics under the CUDA model and a set of transformations that optimize global memory accesses.

PGI has released a directive-based Accelerator Programming Model [11] for CPU + Accelerator systems, and the latest PGI Fortran and C compiler supports this model on CUDA-enabled NVIDIA GPUs. Compared to hiCUDA[1], OpenMP is a standard API that many programmers are already familiar with and many existing applications are programmed in OpenMP. However, both the OpenMP and the Accelerator model are not specific to the CUDA architecture, and therefore, lack the support of important concepts like shared memory and thread block. Creating an abstraction that closely matches the CUDA model is exactly the reason to design a new and simpler set of directives.

The third category of work that is related to proposed system by S. Ryoo et al. [3] which focuses on helping the programmer to optimize a CUDA application.

As per research mentioned in the paper of Tian Yi David Han and Tarek S.Abdelrahman[1] authors mentioned that, GPUs can be used to accelerate applications outside the graphics domain. It is hard to write program in CUDA for average programmer. CUDA puts load on the programmer

- To package GPU code in separate functions called kernel.
- Need to explicitly manage data transfer between host memory and GPU memory.
- Manual optimization of GPU memory is required.

Authors have designed hiCUDA[1], a high level directive based language for CUDA programming.

The hiCUDA presents programmers with a computation model and a data model. Computation Model allows the programmer to identify code regions that are intended to be executed on the GPU and specify how they are to be executed in parallel. hiCUDA provides four directives in its computational model kernel, loop_partition, singular and barrier. Data Model allows programmers to allocate memory on the GPU and move data back and forth between the host memory and GPU memory. hiCUDA provides four directives in its data model: Global, constant, texture and shared.The global directive says the data gets stored in global memory of GPU. It takes more time to access the data. Constant and Texture directives indicates that the variables which are declared with these directives

3. SYSTEM ARCHITECTURE

The implemeted systems block diagram is shown in Fig.1. It contains input, output and basic system block. This implementation is based on techniques prposed by Tian Yi David Hans work[1,17]. The input to this system is a sequential C program and output is an equivalent parallel program in CUDA for given sequential C program.The proposed system provides an automated tool to convert sequential C program to the parallel CUDA program.

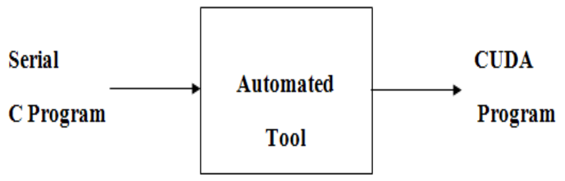

Fig. 1: System Block Diagram

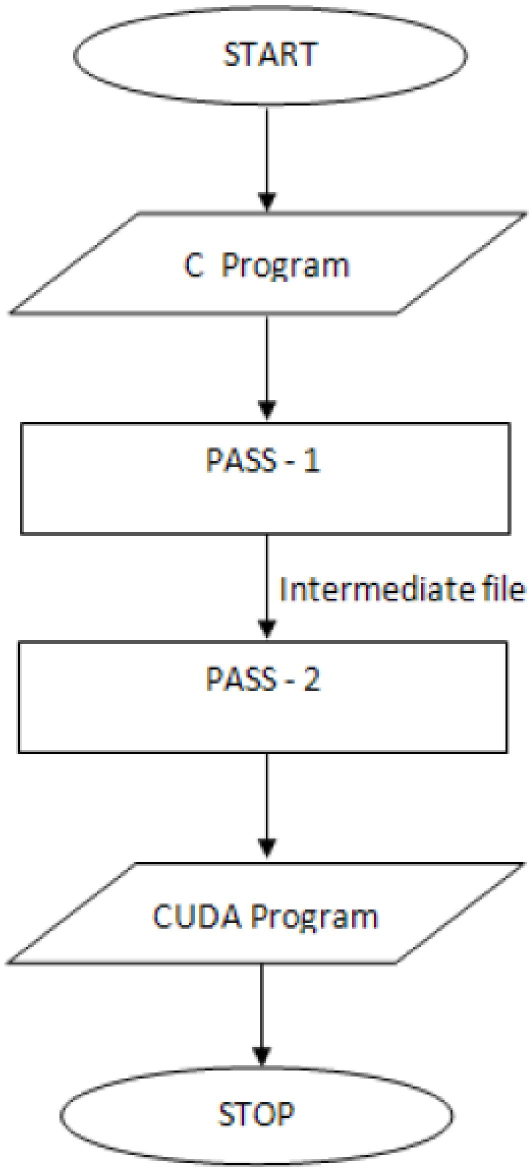

Fig.2: Process to convert C program to CUDA C Program

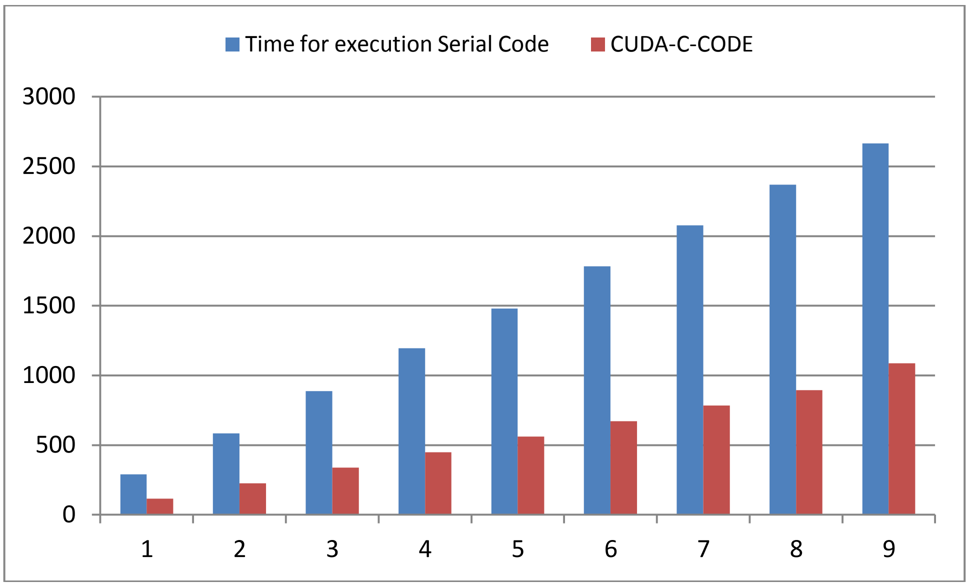

Fig. 3: Graphical representaion comparing time required on CUDA C code as compared to conventional serial C code with nine input sizes indicating low groth in required time for parallel Program

An automated tool works in two passes: pass1 and pass2. In first pass, it analyses the C program, identifies parallelizable portions in the given C program and mark them with appropriate start and end indicators. Finally this pass generates intermediate file containing source code along with parallel block of code marked with #pragma kernel_start and #pragma kernel_end. In second pass, it creates resultant .cu file to store generated code. The tool reads intermediate file line by line. If parallelizable portion of code found, it configures the grid required to execute the code on the GPU cores. It maintains this code in separate function called kernel function. These functions are written in a resultant file and calls to these functions are make at appropriate places in main function. Other code is just normally copied to the resultant file operation. At the end of pass2, resultant .cu file contains parallel CUDA program equivalent to given C program. The Fig.2 shows detailed process of the conversion of C program to parallel CUDA program.

4. RESULT ANALYSIS

Initially the vector addition program was tested on above mentioned configuration. For testing, the C programs are converted in parallel CUDA programs. The resulting CUDA enabled parallel program was executed on GPU with variable input size. The time requred for execusion of these program were noted and were compared with serial execution of the same programs with same input size. The system is implemented using the below mentioned experimental platforms setup for execution of the program.

- Processor: Intel Core i5-4200U, 1.60 GHZ X 4
- RAM: 4 GB
- OS: 64 bit, Ubuntu 14.04 LTS
- NVIDIA graphics card: GeForce 840 GT, 384 CUDA Cores
- CUDA 6.5 (SDK, Toolkit, Drivers)

Table 1:Experimental results presenting time required for excecution of Vector Addition program with conventional C code and parallel CUDA code with varied input size

Sr. No.	Input Size	T_c	T_p	%SP
1	*10000*	*290.00*	*115.10*	60.31
2	*20000*	585.00	225.72	61.41
3	*30000*	888.00	337.63	61.98
4	*40000*	1194.00	448.44	62.44
5	*50000*	1480.00	560.44	62.13
6	*60000*	1783.00	671.16	62.36
7	*70000*	2076.00	784.44	62.21
8	*80000*	2368.00	893.47	62.27
9	*90000*	2664.00	1086.00	59.23
Average		1480.89	569.16	61.59

Table 1 presents experimental results for computation of initialization and addition operation of vector addition program in serial and parallel for given input sizes and it shows time required for serial code and CUDA code in microseconds. Let T_c be a time required by conventional serial program to exceute and T_p be a time required by Parallel program to exceute. Table1 presents thse reults. Here it is observed that computation time taken by serial program more as compred to parallel for the same vector addition program. Let the % speedup be represented as % SP. The % speedup obtained is calculted by equation 1.

$$\%SP = (((T_c - T_p)/ T_c)*100) \qquad (1)$$

Results table show that speedup is around 60% in all cases irrespective of input size for vector addition program. The results are graphically represented in Fig. 3.

5. CONCLUSION

The implemeted system is an automated tool which generates an equivalent parallel CUDA program for given sequential program in C language without manual involvement of user in the parallelization process. It is observed that the generated parallel CUDA programs are correct. The validation of the correctness of the output for parallel converted CUDA programs were validated by testing the results of the program. It was found that the outputs were correct. This tool provides simple and user friendly user interface. The user doesn't required to have knowledge of CUDA programming to use the tool for conversion of serial C programs to parallel CUDA programs.

To validate the the working of this sytem serial C programs was converted to parallel CUDA programs and were tested on NVIDIA graphics cards. As per results obtained it is observed that the computation time required to perform same computation serially on CPU takes more time than on GPU in parallel. Generally GPU's are used for executing graphics applications. The system helps to utilize the NVIDIA graphics cards available in the computer system to accelerate the general purpose applications and thus enhances the system performance by reducing the time complexity of an application. A programmer who doesn't have knowledge of hi-CUDA and CUDA programming can also take benefits of CUDA programming. On average around 60% speedup is obtained.

6. REFERENCES

[1] Tian Yi David Han, Tarek S. Abdelrahman, hiCUDA: High- Level GPGPU Programming, IEEE Transactions on Parallel and Distributed Systems, Vol. 22, No. 1, January 2011.

[2] I. Buck et al., GPUs: Stream Computing on Graphics Hardware, Proc. ACM SIGGRAPH, 2004.

[3] S. Ryoo et al., Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA, Proc. Symp. Principles and Practice of Parallel Programming, pp.73-82, 2008.

[4] C. Liao et al., Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization, Proc. Int'l Workshop Languages and Compilers for Parallel Computing, Oct. 2009.

[5] NVIDIA,NVIDIA GeForce 8800 GPU Architecture Overview, Nov. 2006

[6] NVIDIA, NVIDIA CUDA C Programming Guide v4.2, CUDA-C Programming Guide.pdf, April. 2012.

[7] S.Z. Ueng et al., CUDA-lite: Reducing GPU Programming Complexity, Proc. Int'l Workshop Languages and Compilers for Parallel Computing, pp. 1-15, 2008.

[8] J. Fabri, Automatic Storage Optimization, Proc. Symp. Compiler Construction, pp. 83-91, 1979.

[9] The Portland Group, CUDA Fortran Programming Guide and Reference, Release 2012.

[10] S. Lee, S.J. Min, and R. Eigenmann, OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization, Proc. Symp. Principles and Practice of Parallel Programming, 2009.

[11] The Portland Group, PGI Fortran and C Accelerator Programming Model, Dec 2008.

[12] C.K. Luk, S. Hong, and H. Kim, Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping, Proc. Int'l Symp. Microarchitecture, pp.45-55, 2009.

[13] M.M. Baskaran et al., A Compiler Framework for Optimization of affine loop Nests For GPGPUs, 2008.

[14] Leonardo Dagum and Ramesh Menon, OpenMP: An industry standard API for shared memory programming, IEEE Computational Science and Engineering, January-March 1998.

[15] Stone, J.E., Gohara, D., Guochun Shi, OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems, Computing in Science and Engineering, Vol. 12, Issue 3, pp. 66-73, May 2010.

[16] NVIDIA, CUDA Programming Model Overview, NVIDIA Corporation, 2008

[17] Tian Yi David Han, Directive-Based General-Purpose GPU Programming, master's thesis, Univ. of Toronto, Sept. 2009.