Conversion of C code to CUDA C code for faster execution

[1]Tejas Gijare, [2]Vishal Bafna, [3]Chaitanya Subhedar, [4]Aniket Ingale

Computer Engineering Department, Zeal College of Engineering and Research Pune, India

Abstract :- *There is need for a converter that can convert one programming language to other to save time for learning a new programming language especially like CUDA which deals with CPU as well as GPU. Making such a system automated is also important. Also parallelism is the need to save processing power as well as user's time.*

Keywords : - *Parallel Computing, Serial Computing, CUDA, GPU, HPC*

I. INTRODUCTION

In the last decades, there has been great advancement in the region of Parallel Computing. With the introduction of General Purpose Graphical Processing Units, parallel processing capability has become easy and affordable. A typical GPU is a multicore architecture with each core capable of thousands of threads running simultaneously [1]. CUDA is a parallel computing system which is developed by NVIDIA. The GPU remains idle during running of general purpose applications. To increase the system performance, the computing capability of the GPU available can get properly exploit during execution of application outside the graphics domain. Parallel computing is an important computing field in which many computations are carried out simultaneously.

Some of the areas where GPUs have been used broadly for General Purpose computing are: scientific computing [2], Data Analysis [3], image processing [4], animation and simulation [5] [6] and cryptography [7].

But the vast repositories of legacy serial C codes, which are still in used. They are unable to exploit this addition computing power available to them. Manually updating all such codes is tiring and error-prone. Parallelizing even a single C code is not a minor task. The programmer needs to have a entire knowledge of source code being parallelized and should be comfortable with the destination parallel architecture. Also, even though APIs, such as those of CUDA, have appeals many non-graphics programmer to port their applications to GPGPUs, still the process remains very challenging for programmers. In particular, CUDA places on the programmer the burden of packaging GPU codes in separate methods, of explicitly managing transfer the data between the host memory and many different GPU memories, and of manually optimizing the utilization of the GPU memory [8].

Due to the reasons mentioned above, we have attempted the task to develop the Automated Tool to Generate Parallel CUDA code from a Serial C Code. The tool is aimed to design and enabling simple portability of existing serial software to parallel architectures. This should be possible without the user having any knowledge of the algorithm and the architecture.

II. EXISTING SYSTEM

In existing systems the two approaches are used that are the RPA (Rewrite Parallel Algorithm) and MOL (Modify Original Library). RPA redesigns the sequential or parallel algorithms and tools on GPUs. Using these approaches, the users must need to understand the original (sequential or parallel) algorithm on CPU absolutely at first, and then write the CUDA program on GPU directly. The RPA is the porting approach with the high complexity and low correctness. For an existing CPU program, when using the RPA, the programmers also need to understand the functions and path of execution in the program, and then write the CUDA program on GPU. The process of understanding the algorithm/program will increase greatly the time and complexity of the porting process.[9]

The second porting approach, MOL, is to modify an existing CPU program and let it can be operated on GPU. In the MOL, the first step is to understand the source codes of original CPU program and the operational processes. By using profiler tool, the most time consuming libraries (or functions) can be found and then they (a partial program) are modified

greatly (rewritten the procedure or data structures in general) to become CUDA programs (kernel functions).

Ample of work has been done in enhancing the software support for GPGPU programming. The first group extends CUDA support to other programming languages [10], such as PyCUDA for Python, jCUDA for Java and CUDA Fortran 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,[11] propose a compiler framework for translating an OpenMP program to a CUDA program. The main contributions of this type of work include an interpretation of OpenMP semantics under the CUDA model and the set of transformations that optimize global memory accesses.

PGI has released a directive-based Accelerator Programming Model [12] for CPU + Accelerator systems, and the latest PGI Fortran and C compiler supports this model on CUDA-enabled NVIDIA GPUs. Compared to hiCUDA[10], OpenMP is a standard API which are familier with the programmers 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, less the support of the 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.

III. SYSTEM ARCHITECTURE

The main purpose of this tool is to convert a C code to its equivalent CUDA code. Same is illustrated in the flowchart below in Fig 1. The generated output will be a code in CUDA language which can be executed on any machine with a CUDA enabled graphics card.

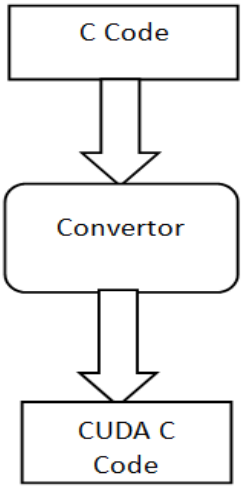

Fig 1: Flowchart showing the input and output with the tool as a black-box

Internally, the tool works in two phases (Fig 2). Open MP is used as the intermediate language between them:

• During the first pass, portions of C code are identified with are not dependent on each other and can be executed in parallel. Open MPPragmas are then inserted for those portions.

• In second pass, aOpen MP compiler produces an equivalent CUDA code consisting of various CUDA functions and kernel (GPU) related operations using Open MP pragmas.

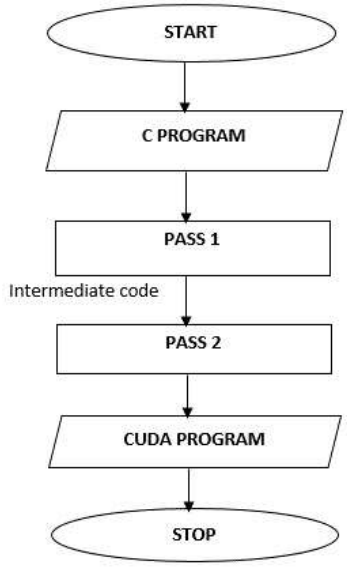

Fig 2: Internal phases

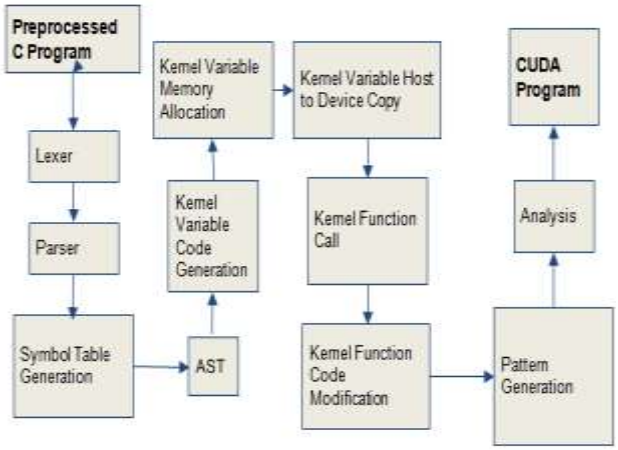

Fig 3: Proposed Work

Initially, the serial C code is passed to the Convertor and then forwarded to following blocks:

- Lexer: The serial C code is tokenized and these tokens are passed to the Parser.
- Parser: The parser generator used is ANTLR which is Another Tool for Language Recognition which is parser generator and checks the tokens against the patterns and grammar.
- STG: Symbol Table Generation block stores all identifiers along with their classification.
- AST: Abstract Syntax Tree generates a Tree according to the representation of the structure of source code written in a computer language. The syntax is "abstract" in not representing every detail appearing in the real syntax.
- Kernel Code: The next few blocks is used to insert the CUDA methods and packages.
- Analysis: The code is further given for complexity analysis and used for performance tuning.

IV. MATHEMATICAL MODEL

System S= {input, output, functions, success, failure}

where,

- Input:C program code.
- Output: CUDA C code.
- Functions={f1,f2,f3}

where,

f1=__global__

f2= Cuda Memory functions.

f3= Cuda Copy functions.

- Success: Successfully generates CUDA code which is feasible to work on GPU.
- Failure: Fails to generate a related CUDA code.

V. CONCLUSION

So from the above analysis it is concluded that there is a need for converter which allows conversion of one source code to another source code. There is huge amount of overhead to learn kernel code management in multiprocessor environment so this converter is useful for converting a source code.

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] E. Alerstam, T. Svensson and S. Andersson-Engels, "Parallel computing with graphics processing units for high speed Monte Carlo simulation of photon migration" , J. Biomedical Optics **13**, 060504 (2008).

[3] Larsen E. S., Mcallister D., ―Fast matrix multiplies using graphics hardware‖, Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, Nov. 2001, pp. 55.

[4] Vladimir Glavtchev, Pinar Muyan-Ozcelik, Jeffrey M. Ota, John D. Owens, "Feature-Based Speed Limit Sign Detection Using a Graphics Processing Unit", IEEE Intelligent Vehicles, 2011.

[5] Purcell T. J., Buck I., Mark W. R., Hanrahan P., ―Ray tracing on programmable graphics hardware‖, ACM Transactions on Graphics 21, 3 (July 2002), pp 703–712.

[6] Knott D., Pai D. K., ―CInDeR: Collision and interference detection in real-time using graphics hardware‖, Proceedings of the 2003 Conference on Graphics Interface, June 2003, pp. 73–80.

[7] Svetlin A. Manavski, "Cuda compatible GPU as an efficient hardware accelerator for AES cryptography" Proc. IEEE International Conference on Signal Processing and Communication, ICSPC 2007, (Dubai, United Arab Emirates), November 2007, pp.65-68.

[8] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming", IEEE Transactions on Parallel and Distributed Systems, Jan. 2011, vol. 22, no. 1, pp. 78-90.

[9]Yu Liu, M. Huang, B. Huang, H.-L. A Huang, and T.Lee, "GPU-Accelerated Longwave Radiation Scheme of the Rapid 1508 Radiative Transfer Model for General Circulation Models (RRTMG)" IEEE J. Sel. Top. Appl. Earth Observ. Remote Sens., vol. 7, pp. 3660-3667, Aug, 2014.

[10] 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.

[11] 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.

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