International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 04 Issue: 06 | June -2017 www.irjet.net p-ISSN: 2395-0072

Auto conversion of serial C code to CUDA code

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

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

---

Abstract - As the GPU is growing demand of the Game Industry and large scientific computations , efforts have been made to take advantages to gain maximum utilization of GPUs in computation. GPUs follow architecture named CUDA- Compute Unified Device Architecture. And to use GPUs there is a language CUDA C which is extension to C. But CUDA C needs to be learned by the developers. Though GPUs are widely used in Supercomputers today , they are not portable because one has to sit and code the algorithms in CUDA to run them on GPU. So if we can have some middleware that converts the C programs to CUDA, the end user gets transparency. We tried to develop a prototype compiler that is in visual studio and converts the C programs in CUDA C language. The paper describes the Pattern approach to develop a translator for source code to source code translation.

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

Introduction:

Because of the demands of game industry, Graphics Processing Units (GPUs) have evolved from application-specific units for 3D scene rendering into highly parallel and programmable multi pipelined processors that can satisfy extremely high computational requirements at low cost. The fact that the performance of graphic processing units (GPUs) is much bigger than the central processing units (CPUs) of now-a-days [1] is hardly surprising. GPUs were formerly focused on such limited field of computing graphic scenes. Within the course of time, GPUs became very powerful and the area of use dramatically grew. So, we can come together on the term General Purpose GPU (GPGPU) denoting modern graphic accelerators. The driving force of rapid rising of the performance is computer games and the entertainment industry that evolves economic pressure on the developers of GPUs to perform a vast number of floating-point calculations within the unit of time. The research in the field of GPGPU started in late 70's. Today's fastest GPUs can deliver a peak performance in the order of 500 GFLOPS , more than four times the performance of the fastest x86 quad-core processor. This thesis introduces a source to source transformation of c programs to CUDA Architecture. It also finds out dependencies and performs optimization for peak performance gain. Automatic evolution of kernels, independent code finding, Loop unrolling, Memory coalescing and thread scheduling are main part of concerns. IR level optimization and higher level optimizations patterns finding is important issue that may be covered by this thesis. Thesis describes parallelization patterns and CUDA C extensions from C to find out transformation rules.

Introduction to ANTLR3:

ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a vari-ety of target languages. ANTLR has a sophisticated grammar development environment called ANTLRWorks. ANTLR provides environment to develop a compiler that parses the input program. Lexer and Parser code can be generated in C#, C, Java, Python etc.

C2CUDATranslator Flow:

The flow of the translator shows the overall functionality and proper compiler structure. The input is C file which is pre-processed input to translator. The input is given to C Parser which is generated using ANTLR grammar and contains two files lexer and parser. Lexer generates tokens and using ITokens interface the parser rules are parsed and the code is checked using the parser. The symbol table is generated. The preprocessor outlines the kernel by pragma pack. Before starting the kernel region the line with "#pragma kernel start" comes. So the compiler can know that the next statements are of kernel region which will be ported on the GPU. The kernel is finished with the statement "#pragma kernel end". The translator uses

symbol table and converts the kernel region with kernel function.

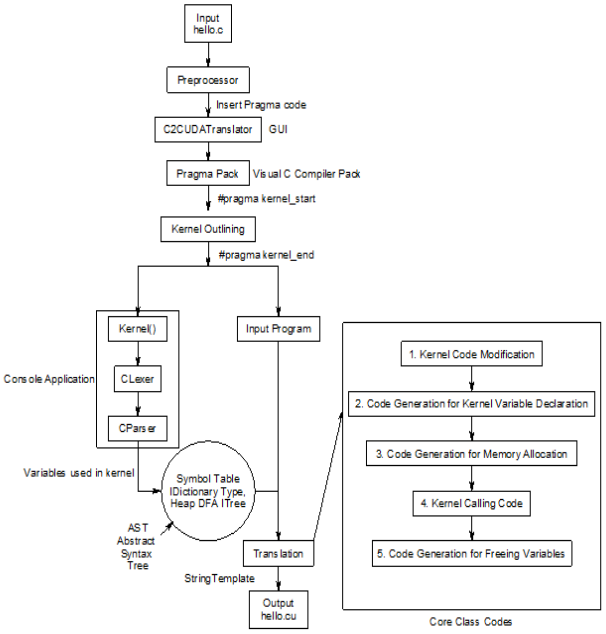

Figure 1: Flow of the C2CUDA Translator

For kernel code writing the translator performs 5 steps shown in figure above. They are nothing but the proposed patterns in previous sections. The translator generates codes for memory allocation on both host and devices and auto generates required pointers and variables. After that it ports the kernel region in kernel function. After parallelization is added it will be ported on many threads and blocks.
Finally, the CUDA File code is generated and extra functions in .c file are as it is in .cu file.

Compiler Style:

For kernel region compiler has a unique style. There are some projects that start but never reach to the end. The algorithms for parallelization are more and time to add all of them are not enough for thesis. So I tried to implement a mechanism in which one can add more algorithms if one wants to. Here compiler generates patterns for the input program. By the time we can see there are N numbers of programs and infinite. New programs and algorithms will be introduced to the CUDA by the time. So there will be N number of patterns. Patterns are the heart of the compiler. Pattern describes the mechanism for the compiler like virus signature does in Antivirus software. New virus are always generated and corresponding signatures are also made in antivirus. Similarly new patterns can be added later and so on.

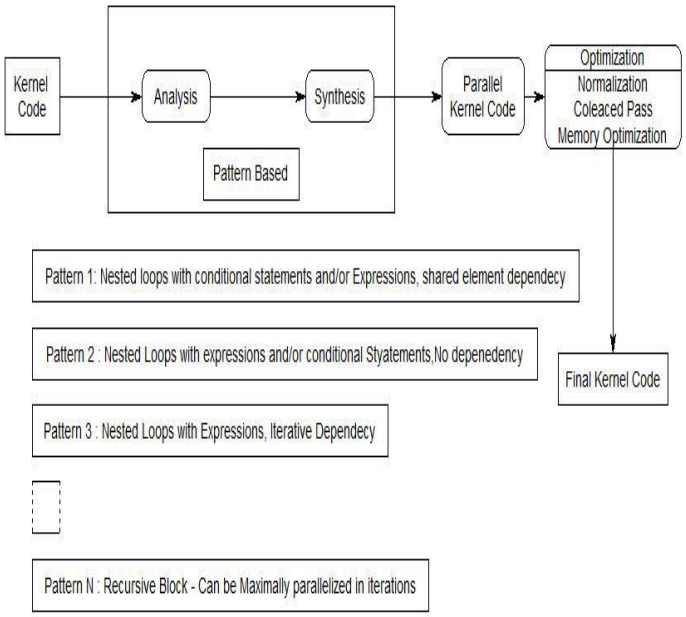

Figure 2: Compiler Style

Results and Evaluations:

The compiler is tested with parboil benchmark suite. The graph shows the comparison between CUDA BASE programs, which are handwritten and fully optimized and C2CUDATranslator converted programs.

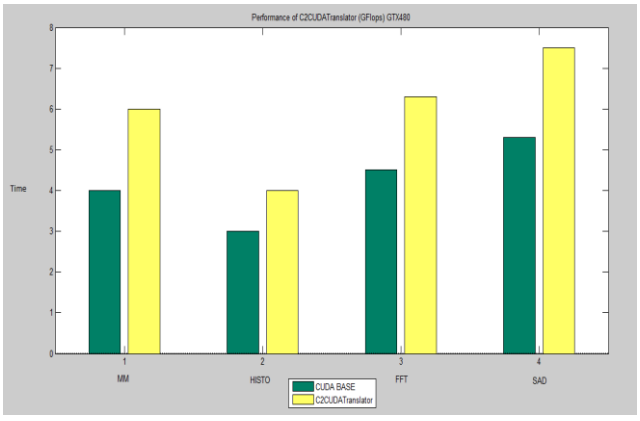

Figure 3: Evaluation of C2CUDATranslator

Conclusion:

The C2CUDATranslator saves 95% of the development translation time. This can also be used as a framework for future translator development for other developers.

References:

[1] 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 703712.

[2] 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. 7380.

[3] 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.

[4] 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.

[5]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.