International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

Compiler Design Techniques to Better Utilize GPUs

Parth Venkatesh[1], Nikith Hosangadi[2], P. Samith[3], A. Parkavi[4]
[1,2,3]*Student, Department of CSE, M. S. Ramaiah Institute of Technology, Bengaluru, India*
[4]*Professor, Department of CSE, M. S. Ramaiah Institute of Technology, Bengaluru, India*

***Abstract*: The amount of computational power to handle parallelism in programming has increased since the increase in popularity of utilizing GPUs. GPUs are better utilized for computation that takes a lot of time on CPUs. GPUs do this by parallelizing instructions. Python libraries such as numpy have better computation power than regular libraries because of parallelization procedures. Despite having so many advantages, GPUs have various shortcomings. This paper puts forward a study of approaches that can be used to better utilize GPUs to their full extent. We have put forward various techniques like cache-bypassing. We also put forward few compiler frameworks that would compile code so that the program runs better on GPUs.**

***Keywords*: compiler, cache bypassing, graphics processing unit (GPU), single program multiple data(spmd), single instruction multiple data (simd), GNU compiler collections (gcc)**

1. Introduction

GPUs are better suited for handling programs that require simultaneous computations, as they have the ability to process multiple instructions in parallel more effectively compared to CPUs. But for GPUs to be effective, they have to be optimized. Otherwise we cannot extract the full processing potential of the GPU. In order to overcome this problem, several researchers have put forward different methods to solve the optimization problem, including methods such as redundant multithreading (RMT), hardware saturation, introducing high level languages for GPUs and so on. Implementing reliability in GPUs has also been researched. The researchers have analyzed their proposed methods practically and have obtained results and evidence to show that the methods used improve GPU computation efficiency. But GPUs are not suitable for all programs and cannot be expected to deliver drastic improvements in all programs. Only programs which utilize multiple threads while processing will benefit from the parallelism support GPUs have to offer. Thus the following survey paper focuses on surveying different methods to optimize GPUs and enforce reliability.

2. Literature survey

Munesh Singh Chauhan in his research work describes the growing popularity of GPU computing and the need for a universal compiler base to support the growing number of present day languages. The LLVM (Low Level Virtual Machine) compiler is the most popular compiler base for GPU computing. The main benefit of LLVM compilers is that it features parallel execution such as multithreading. The most popular architecture for GPU computing, CUDA is based on the LLVM infrastructure. The working of CUDA compiling is described in the paper [1].

Gautam Chakrabarti et al. in their work talk about how GPUs have evolved to handle tasks which require a large amount of parallel threads just to make any progress. In the paper they talk about their experiences in developing a compiler for a language called as CUDA C. Optimizations for CUDA architecture are also presented. Optimizations are implemented and analysis is done to see the scope of improvement. They conclude by stating that parallelism is best achieved through GPU computing. [2]

Christopher Dubach et al. in their research work talk about how using low-level languages for GPU programming such as OpenCL and CUDA require explicitly managing numerous low-level details involving synchronization and communication. They found that this extra burden makes GPU programming more prone to errors and difficult to do. Using a high-level programming language can avert these inconveniences while concurrently exploiting the GPU's computational power. A Java compatible language called Lime is presented. Analysis has been done to compare Lime to low-level languages such as those listed above. Optimizations are introduced and tested. [3]

Jack Wadden et al. talk about how to implement reliability for general purpose processing on GPUs. Implementing reliability through hardware has several drawbacks, including expensiveness, requiring dedicated on-chip resources, and lack of portability across different architectures. A software solution called RMT (redundant multithreading) is discussed in this paper. Analysis on several parameters such as overhead and power is carried out [4].

Alberto Magni et. al. discuss about how to realize the full performance potential of GPUs for general purpose computation. This often requires extensive compiler tuning, which is an expensive procedure. To avoid this, hardware saturation is introduced, where an application is executed with a large number of threads such that all the hardware resources are fully utilized. This method is applied to quickly infer the full performance potential. Further analysis is done to test the effectiveness of the proposed method. Thus they conclude by

showing experimental evidence of their research, and strengthening the legitimacy of this method [5].

Reliability for general purpose computation on GPUs has become low in of construction supercomputers nowadays. This is because of introduction of unnecessary multi threads when CPU code is parallelized for running it on GPUs. this paper proposes usage of Wang et. al. and apply it to OpenCL's kernel compiler. This compiler converts the code into IR code. It is then later optimized into LLVM.'s IR. code. Then a compiler on the backend further compiles and optimizes it for running it on gpus. This three-phase architecture of [6] is less prone to introduce unnecessary multi threads into the gpu compiled version of our code.

[7] introduces us to a new framework for python code. This code has now been implemented into a python library. The library contains its own version of various other libraries such as numpy. It mimics the formulations but is much faster. We are talking about theano. Theano, when introduced, was a powerful python code compiler. It takes in code written in a particular format. There have to be four segments of code. The first segment has declaration of symbolic variables. This is to be followed by building a symbolic expression graph. Then comes the section where theano functions are written. The last section is where these functions are used and called. This format can be seen by most machine learning and deep learning programs.

This format of code is efficiently compiled into parallel instructions from GPUs. However, programs in all languages cannot be written in this format. Therefore [8] introduces us with kernelgen. KernelGen is another compiler platform that would help to accelerate numerical model that are to run on GPUs. The optimized code runs on an entirely different address space. The address space is supposed to contain all the lain instructions. Functions are kept in secondary memory. The start addresses of functions are stored in registers. When a function is called. GPU fetches address of the function and stores in primary scratchpad memory. The address is then stored in registers. Subsequent calls to the functions are faster.

In [9], the authors propose another compiler called ispc. Ispc compiler stands for intel SPMD program compiler. So basically this compiler has its own approach of parallelizing plain instructions. The parallelization follows a pipelined technique. The instructions are divided into certain phases such as fetch, decode, etc. Then instead of running one instruction per core, one phase is done per core. This one core always does fetch of instructions and other core always does decoding part. This makes one core just for executing the instructions.

[10] deals with a pretty common problem that every piece of code running on GPUs faces. This paper talks about efficient usage of the cache and on-board scratch pad memory. There must be a balance in the amount of instructions that search cache and instructions that search data in on-board scratchpad memory. It provides us with a way we can bypass the cache fetches.

In [11] they explain how Graphics processing units (GPUs) successfully accelerate regularly structured and data intensive applications. GPU's are used to increase the speed of computations which exhibit basic patterns of parallelism. Dynamic parallelism (DP) which was introduced by Nvidia allows the launch of kernels directly from GPU threads which further enables nested parallelism at runtime. Poor performance can be the direct result of the poor or improper use of DP. However, we can improve performance by the use of three workload consolidation schemes and by implementing the DP-based codes in a /directive-based compiler. The improvements include reduction of runtime overhead, improvement in GPU utilization. DP, however, has its own set of limitations which include kernel launch overhead, kernel buffering overhead and synchronization overhead.

Although GPU's enable a drastic improvement in performance in data intensive parallel applications, the execution of these applications on the GPU's requires certain low level operations such as handling memory allocations, optimizing kernels by making use of the right memory types on the GPU and using low level programming models to write GPU kernels. This in turn is a herculean task for a large number of programmers as they use high level programming languages and only expert programmers are able to successfully take full advantage of the computational powers of GPU's. While GPU's do have benefits, the use of high-level programming languages provides productivity benefits. As per [12], one such language is Java, therefore a compiler which can generate GPU code which is optimized from subsequent pure Java programs is essential. The compiler can make use of the parallel streams API's which are offered in Java 8 to write lambda expressions, the compiler would thus convert Java 8 code into GPU code and generate runtime calls automatically which would further handle the above-mentioned low-level operations. The benefits of the compiler include the increase of the memory bandwidth, increase of memory efficiency and lastly eliminate redundant data transfers between the host machine and the GPU.

The recent developments in GPU design and the programmability of GPU's allow for general-purpose computation on a GPU(GPGPU). Although multiple libraries, tools and languages are proposed to enable GPGPU programming the irregular programming model of the GPU is a hindrance to well written GPGPU programs. To overcome this, authors in [13] propose that code fragments which are meant to be executed on the GPU are labelled using compiler directives, further these labelled code fragments are converted into C code(ISO-complaint) which in turn contains the necessary OpenGl and Cg application program interfaces (API'S). The code can then be compiled into executable code using a native C compiler. By following the above-mentioned steps, a suitable compiler could be generated which would produce significant improvements in performance for data intensive parallel programs

Graph algorithms in themselves are very challenging because

they tend to be dense array programs which have affine loop nests. High-performance implementations of these algorithms on the GPU are even more challenging. However, by making use of throughput optimizations and polyhedral compilation techniques we can generate high quality Compute Unified Device Architecture (CUDA) code. The throughput optimizations cannot be implemented by hand due to their large implementation space. The throughput optimizations include kernel launch throughput, fine grained synchronization throughput and the graph traversal throughput. These throughputs could cause bottlenecks in GPU performance therefore limiting GPU's. Furthermore, by keeping track of certain focal points which are the nodes in the program's control flow graph which are sure to be visited by each and every thread we can perform more optimizations. This has been put forth in [14].

High performance code for platforms such as multicore machines, GPU's and distributed machines could unlock the true potential of these platforms. As per authors in [15], the high-performance code can be generated using polyhedral frameworks which use a scheduling language with special commands to deal with complexities arising due to these systems. The scheduling language allows us precise control of optimizations. The applications are limitless but can be used for fields such as deep learning and image processing. The polyhedral framework thoroughly separates the algorithms from the data layouts, the loop transformations and the communications. By doing so we can easily target multiple hardware architectures for the same algorithm. We observe a substantial improvement in performance and the framework outperforms any existing libraries and compilers on different platforms and hardware architectures such as GPUs multicore CPUs and other distributed machines.

3. Discussion

In our thorough literature survey, we observed that the common goal for most of the papers is optimizing GPU compilers to achieve maximum throughput and efficiency. Several related but different approaches were taken by different researchers. The majority of the papers focused on parallelism, CUDA computing, and using high level language compilers for GPUs. Optimizing GPU compilers is attempted through several methods, such as introducing a high level language GPU compiler, hardware saturation, and so on. The most effective method according to us is introducing a specially made high level language for GPUs. This conclusion was reached after considering the fact that using low level languages to run programs on GPUs requires specifying several low level functions and parameters, whereas this can be avoided in high level language compilers. Maintaining reliability in GPU compilers is also of concern, and thus research has been done in the field of GPU reliability. Redundant multithreading is a reliability technique where the program is run redundantly on many threads so that the failure of one thread doesn't end the program, it can just be continued on the other threads. Handling tasks which require a large amount of resources, like graph algorithms, and other tasks which require parallelism, are best handled using GPUs. CUDA architecture allows breakthroughs in several trending fields such as machine learning, deep learning and so on.

4. Conclusion

We conclude our survey by acknowledging the importance of the GPU optimization concept and making GPU programming reliable. Several methods were proposed for optimization and reliability, each with their own advantages and disadvantages. Optimization techniques such as hardware saturation, designing a high level language for GPUs, and different compilers to better parallelize instructions were researched upon. Reliability techniques such as redundant multithreading and using IR compiler code to prevent creating unnecessary threads from wasting resources were also examined. This survey paper was an attempt to aggregate several optimization and reliability techniques which can be applied to GPU computing to improve its efficiency. We aim to research more into these techniques and attempt to develop our own techniques to solve the same problem of optimization and reliability in GPU computing.

References

[1] Munesh Singh Chauhan. "Analysis of LLVM Parallel Compiler on GPU Running on CUDA Framework" The 1st National Symposium on Frontiers in Information Technology: Business Intelligence and Analytics, At Rustaq, Oman, 2015.

[2] Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, Jian-Zhong Wang. "CUDA: Compiling and optimizing for a GPU platform." International Conference on Computational Science, ICCS 2012.

[3] Christophe Dubach, Perry Cheng, Rodric Rabbah,David F. Bacon, Stephen J. Fink "Compiling a High-Level Language for GPUs (via Language Support for Architectures and Compilers)" University of Edinburgh.

[4] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan and K. Skadron, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Minneapolis, MN, 2014, pp. 73-84.

[5] Alberto Magni, Christophe Dubach, Michael O'Boyle. "Exploiting GPU Hardware Saturation for Fast Compiler Optimization." Proceeding

[6] GPGPU-7 Proceedings of Workshop on General Purpose Processing Using GPUs, Pages 99, Salt Lake City, UT, USA, March 01 - 01, 2014.

[7] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins,Joseph Turian, David Warde-Farley, Yoshua Bengio. "Theano: A CPU and GPU Math Compiler in Python"., proc. Of the 9th python in science conf. (SCIPY 2010).

[8] D. Mikushin, N. Likhogrud, E. Z. Zhang and C. Bergström, "KernelGen -- The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs," *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, Phoenix, AZ, 2014, pp. 1011-1020.

[9] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," *2012 Innovative Parallel Computing (InPar)*, San Jose, CA, 2012, pp. 1-13.

[10] X. Xie, Y. Liang, G. Sun and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, 2013, pp. 516-523.

[11] Hancheng Wu, and Da Li, "Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU," IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016.
[12] K. Ishizaki, A. Hayashi, G. Koblents and V. Sarkar, "Compiling and Optimizing Java 8 Programs for GPU Execution," *2015 International Conference on Parallel Architecture and Compilation (PACT)*, San Francisco, CA, 2015, pp. 419-431.
[13] Yu-Te Lin, Peng-Sheng Chen. "Compiler support for general-purpose computation on GPUs" In The Journal of Supercomputing, vol. 50, no. 1, October 2009.
[14] Sreepathi Pai, and Keshav Pingali. "A Compiler for Throughput Optimization of Graph Algorithms on GPUs", Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.
[15] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, Saman Amarasinghe. "Tiramisu: a polyhedral compiler for expressing fast and portable code". CGO 2019 Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization.