Using Machine Learning for OpenMP GPU Offloading in LLVM

Alok Mishra*
alok.mishra@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA

Abid M. Malik†
amalik@bnl.gov
Brookhaven National Laboratory
Upton, NY, USA

Barbara Chapman‡
barbara.chapman@stonybrook.edu
Stony Brook University
Stony Brook, NY, USA
Brookhaven National Laboratory
Upton, NY, USA

ABSTRACT

OpenMP 5.0 provides features to exploit the compute power within the node of today's leadership class facilities. Among these features, the GPU offloading directives are key to take advantage of heterogeneity on modern machines. However, these features place the domain scientists with portability challenges, especially for optimizing data movement between a host and a device. Tools that facilitate the usage of such features are becoming important to the scientific community. An important tool for porting legacy codes to newer machines will be compilers that can predict the feasibility of transferring kernels on GPUs and inserts required OpenMP GPU offload features automatically at compile time. In this work, we are exploring a novel approach for the automated handling of OpenMP GPU offloading using machine learning techniques. We aim to develop an end-to-end application framework, from legacy code to GPU offloading, that integrates machine learning techniques into the LLVM compiler.

ACM Reference Format:

Alok Mishra, Abid M. Malik, and Barbara Chapman. 2020. Using Machine Learning for OpenMP GPU Offloading in LLVM. In *Proceedings of Supercomputing '20: ACM Student Research Competition (Supercomputing '20).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

POSTER SUMMARY

Majority of supercomputers today run advanced, large-scale applications efficiently, by leveraging GPU-powered parallel processing across multiple compute nodes. GPUs are now being used by applications on these supercomputing clusters to accelerate compute intensive work ranging from 3D simulations to medical imaging to financial modeling. This acceleration gives a significant boost to their cost savings and efficiency, paving the way for scientific exploration. However, the task of writing a new GPU application or offloading an already parallel computation to GPU and achieve maximum performance can be anywhere from hard to herculean.

Porting or writing applications for GPU requires extensive knowledge of the underlying architecture, the application/algorithm and the interfacing programming model. One approach to reduce this effort of the developers is to use a directive based programming model, like OpenMP, which since version 4.0 have GPU device offloading support.

Nevertheless, even if we use directive based models like OpenMP, it is still quite challenging to optimize large scale applications consisting tens-to-hundreds of thousands lines of code. For example, the Lattice QCD [3] project code uses the Grid library [2], where most of the operations are small, don't have enough computational work to justify a GPU execution, deeply buried in the library specification, or evenly spread throughout the application. Effectively, "pragmatizing" each kernel is a repetitive and complex task. Moreover the performance of an application depends a lot on the GPU architecture, which evolves so quickly, that it is difficult to keep up. There is a need for a compiler optimization that automatically offloads a region of code to GPU, maximizing GPU resource utilization and delivering the best speedup when compared to the original code. One such optimization could use a static cost model which predicts the cost of execution of the code on a given GPU.

Since the efforts to tune the cost function are quite expensive, almost all modern compilers, such as Clang, simply use the "one-size-fits-all" cost function, which does not provide the best performance. Consequently, hand-crafted cost functions are widely used in compiler optimizations. One of the problem of relying on a hand-tuned cost model is that the costs and benefits of a compiler optimization often requires thorough knowledge of the underlying hardware. Though effective, manually developing a cost model can take months or years for a single architecture. Because cost functions are critical and manual tuning is rather arduous for each individual architecture, compiler engineers are exploring how to use machine learning to automate this process.

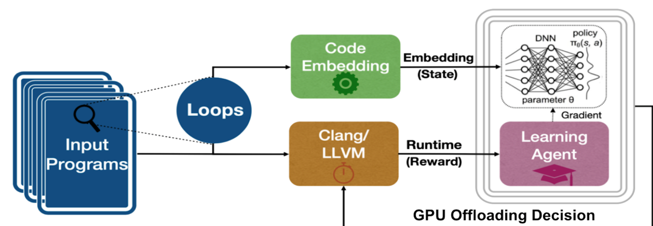

Figure 1: The workflow for ML learning for GPU offloading problem based on [6]

*Graduate Student Author
†Mentor
‡Advisor

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Supercomputing '20, November 17–19, 2020,

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

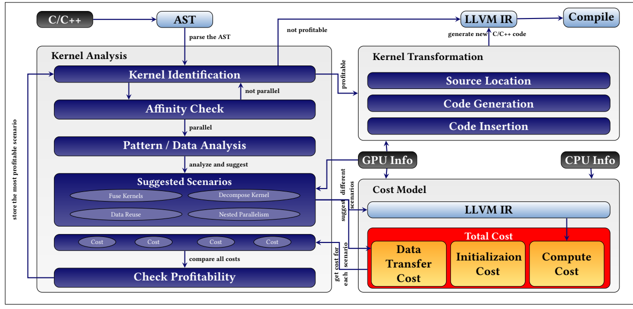

Figure 2: The workflow for Program Transformation for Automatic GPU-offloading based on [8] and [9]

This work explores a novel machine learning approach for the automated handling of OpenMP GPU offloading by estimating a given kernel's execution cost. Machine learning models need training data for building a model. Currently, we are preparing the training data sets by modifying applications from common benchmarks like SPEC and Rodinia Benchmark Suites [4]. We generate several variants of each application by varying parameters like input size, loop iteration or common optimizations like loop unrolling, interleaving, etc. For this, we studied existing cost models [1, 5–7, 10–13], and prepared a list of common parameters/features which effect kernel execution time. These parameters, which are input to machine learning model, can be divided into three categories:

(1) *Common GPU Parameters* (CGP) – threads per wrap, frequency, memory bandwidth, etc.
(2) *Machine Resources Parameters* (MRP) – threads per block, number of blocks, active SMs, etc.
(3) *Source Code Analysis Parameters* (SCAP) – computation instructions, memory instructions, etc.

We model the problem as "Regression Problem" and "Classification Problem". For the regression problem, we estimate the **Total Cost** of execution using Equation 1.

$$Total_Cost = \Theta_0 \times C_{compute} + \Theta_1 \times C_{data} + \Theta_2 \times C_{init} \quad (1)$$

$$C_{compute} = \Theta_{c_0} \times CGP + \Theta_{c_1} \times MRP + \Theta_{c_2} \times SCAP \quad (2)$$

$$C_{data} = \Theta_{d0} \times D + \Theta_{d1} \times N + \Theta_{d2} \times S_L + \Theta_{d3} \times N_L + \Theta_{d4} \times S_q \quad (3)$$

Where $C_{compute}$ and C_{data} are **Compute Time** and **Data Transfer Time** which can be calculated by models based on Equation 2 and 3 respectively. C_{data} depends on parameters like total data transferred (D), number of variables transferred (N), speed of the PCIe link (S_L), number of lanes in the PCIe link (N_L) and speed of QPI link between CPUs (S_q). C_{init} is the **Initialization Time** which has impact only once for an application, when the first target region is encountered. For small programs, it can have a significant impact and can be calculated experimentally. We use machine learning to find coefficients in the afore-mentioned equations. We also use classification techniques for the OpenMP GPU offloading problem. We use reinforcement learning to train a machine learning model to decide whether it is profitable to move a kernel to GPU device or not.

Figure 1 shows the basic workflow of our framework using LLVM. Code embedding converts a kernel in a format that can be used by a machine learning technique. Based on the input, the model makes a decision for a kernel whether or not to offload, which is then used by the LLVM framework and evaluated at runtime. The framework automatically insert OpenMP directives to support GPU offloading using Clang based tools presented in the work [8] and [9]. The generic workflow of these tools and how they use the cost model to make decision is depicted by Figure 2. The source-to-source tool automatically identifies and insert the pertinent, optimized OpenMP GPU offloading directives. The performance result is given to a learning agent for evaluation. Based on the evaluation results, the weights of the model are adjusted. The process is repeated until the performance is acceptable.

Machine learning is not a remedy to cure all problems, but it definitely opens up the possibility of much greater creativity and new research areas in the field of compilers. Our goal is to provide the first general solution to develop an end-to-end application framework, from legacy code to GPU offloading, that integrates machine learning techniques into the LLVM compiler.

REFERENCES

[1] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 105–114.
[2] Peter Boyle, Azusa Yamaguchi, Guido Cossu, and Antonin Portelli. 2015. Grid: A next generation data parallel C++ QCD library. *arXiv preprint arXiv:1512.03487* (2015).
[3] Richard Brower, Norman Christ, Carleton DeTar, Robert Edwards, and Paul Mackenzie. 2018. Lattice QCD application development within the US DOE Exascale computing project. In *EPJ web of conferences*, Vol. 175. EDP Sciences, 09010.
[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.
[5] Xi E Chen and Tor M Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 329–340.
[6] Ameer Haj-Ali, Nesreen K. Ahmed, Theodore L. Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: end-to-end vectorization with deep reinforcement learning. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*. ACM, 242–255. https://doi.org/10.1145/3368826.3377928
[7] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*. 152–163.
[8] Alok Mishra, Martin Kong, and Barbara Chapman. 2019. Kernel fusion/decomposition for automatic GPU-offloading. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 283–284.
[9] Alok Mishra, Abid M. Malik, and Barbara Chapman. 2020. Data Transfer and Reuse Analysis Tool for GPU-offloading using OpenMP. In *Proceedings of the International Workshop on OpenMP (IWOMP2020)*.
[10] Yuhan Peng, Max Grossman, and Vivek Sarkar. 2016. Static cost estimation for data layout selection on GPUs. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 76–86.
[11] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 11–22.
[12] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.
[13] Yao Zhang and John D Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*. IEEE, 382–393.