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.

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 978-1-4503-8069-6/20/11. . . $15.00 https://doi.org/10.1145/nmnnnn.nmnnnn

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

\[
\text{Total Cost} = C_{\text{compute}} \times \Theta_0 + C_{\text{init}} \times \Theta_2 + C_{\text{data}} \times \Theta_3
\]  

\[
C_{\text{compute}} = C_{\text{CGP}} \times \Theta_1 + C_{\text{MRP}} \times \Theta_2
\]  

\[
C_{\text{data}} = C_{\text{init}} \times \Theta_4 \times D + C_{\text{Sim}} \times N + C_{\text{QPI}} \times L + C_{\text{SIM}} \times N \times L + C_{\text{SIM}} \times Q
\]  

Where \(C_{\text{compute}}\) and \(C_{\text{data}}\) are **Compute Time** and **Data Transfer Time** which can be calculated by models based on Equation 2 and 3 respectively. \(C_{\text{data}}\) depends on parameters like total data transferred (D), number of variables transferred (N), speed of the PCIe link (S_d), number of lanes in the PCIe link (N_d) and speed of QPI link between GPUs (S_q). \(C_{\text{init}}\) 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


