OpenMP Advisor: A Compiler Tool for Heterogeneous Architectures

Alok Mishra[1,3(✉)], Abid M. Malik[2], Meifeng Lin[2], and Barbara Chapman[1,3]

[1] Stony Brook University, Stony Brook, NY 11794, USA
almishra@cs.stonybrook.edu, barbara.chapman@stonybrook.edu
[2] Brookhaven National Laboratory, Upton, NY 11973, USA
{amalik,mlin}@bnl.gov
[3] Hewlett Packard Enterprise, Spring, TX 77389, USA
{alok.mishra,barbara.chapman}@hpe.com

Abstract. With the increasing diversity of heterogeneous architecture in the HPC industry, porting a legacy application to run on different architectures is a tough challenge. In this paper, we present OpenMP Advisor, a novel compiler tool that enables code offloading to a GPU with OpenMP using Machine Learning. Although the tool is currently limited to GPUs, it can be extended to support other OpenMP-capable devices. The tool has two modes: Training and Prediction. It analyzes benchmark codes, generates every possible code variant on the target device, runs and gathers data to train an ML-based cost model in the training mode, which predicts the runtime of every code variant in the prediction mode. The main objective behind this tool is to maintain the portability aspect of OpenMP. Our Advisor produced code for several applications on seven architectures with four compilers, and accurately anticipated the top ten options for each application on every architecture. Initial results suggest that this tool can help compiler developers and HPC researchers migrate their legacy codes to the new heterogeneous computing environment.

Keywords: openmp · gpu · machine learning · cost model · compiler

1 Introduction

General Purpose Graphics Processing Units (GPGPUs), initially designed for graphics tasks, have become integral to HPC platforms and general-purpose computing over the last decade, combining their capacity for efficient data parallelism with low power consumption. The vast majority of HPC systems in use today are heterogeneous, with AMD or NVIDIA GPUs delivering high performance per unit of energy consumed. Programming updates are needed to enable efficient utilization of diverse hardware resources, such as GPUs and specialized processors, in order to cater to the rapid change in heterogeneous architecture in the HPC industry.

Many programmers are adapting their code to take advantage of GPUs. Unfortunately, it can be time-consuming and require extensive re-engineering

S. McIntosh-Smith et al. (Eds.): IWOMP 2023, LNCS 14114, pp. 34–48, 2023.
https://doi.org/10.1007/978-3-031-40744-4_3

to maximize a GPU's computational power while minimizing overheads. It will be much harder to develop code for systems with extreme heterogeneity and a large number of devices. Therefore, it is essential to create tools that will relieve the application scientists of the burden of such development. Despite the variety of programming models available, it is still quite challenging to optimize large scale applications consisting of tens-to-hundreds of thousands lines of code. Even when using a directive based programming model such as OpenMP [6], pragmatizing each kernel is a repetitive and complex task. OpenMP offers a variety of options for offloading a kernel to GPUs. However, the application scientist must still figure out all the intricate GPU configurations. To demonstrate the complexity of porting a kernel to emerging exascale hardwares, we use a kernel from the Lattice Quantum Chromodynamics (LQCD) [2] application, which is a computer-friendly numerical framework for QCD. One of LQCD's key computational kernels is the Wilson Dslash operator [15], which is essentially a finite difference operator, to describe the interaction of quarks with gluons. The Wilson Dslash operator, D, in four space-time dimensions is defined by Eq. 1.

$$D^{ij}_{\alpha\beta}(x,y) = \sum_{\mu=1}^{4}[((1-\gamma_\mu))_{\alpha\beta}U^{ij}_{\mu}(x)\delta_{x+\hat{\mu},y} + (1+\gamma_\mu)_{\alpha\beta}U^{\dagger ij}_{\mu}(x+\hat{\mu})\delta_{x-\hat{\mu},y})] \tag{1}$$

Here x and y are the coordinates of the lattice sites, α, β are spin indices, and i, j are color indices. $U_\mu(x)$ is the gluon field variable and is an $SU(3)$ matrix. γ_μ's are 4×4 Dirac matrices that are fixed. The complex fermion fields are represented as one-dimensional arrays of size $L_X \times L_Y \times L_Z \times L_T \times SPINS \times COLORS \times 2$ for the unpreconditioned Dirac operator, where L_X, L_Y, L_Z and L_T are the numbers of lattice sites in the x, y, z and t directions, respectively. The spin and color degrees of freedom, which are commonly 4 and 3, are denoted by the variables $SPINS$ and $COLORS$.

When we express Eq. 1 in C++, it has four nested `for` loops iterating over L_T, L_Z, L_Y, and L_X (as shown in Code 1.1). When we keep the values of L_T, L_Z, L_Y, and L_X at 16 each, the COMPUTE section of the code has over 5 million variable definitions, 1.2 billion variable references, over 150 million addition/subtraction, 163 million multiplication, and so on. Additionally, this function is called several times throughout the LQCD application. It is a herculean task for an application scientist to understand the physics, transform it

```
#pragma omp target
  for(int i=0; i<N_i; i++) {
    for(int j=0; j<N_j; j++) {
      for(int k=0; k<N_k; k++) {
        for(int l=0; l<N_l; l++) {
          /* ... COMPUTE ... */
  }}}}
```

Code 1.1. Loops of Wilson Dslash Operator

into computer program, analyze the offloadable kernel, and then consider how to parallelize it to execute efficiently on an HPC cluster. To get the best performance out of a GPU, an application scientist needs a thorough understanding of the underlying architecture, algorithm, and interface programming model. Alternately, they could test out various GPU transformations until they find the most effective one. However, none of these strategies is very efficient.

1.1 Our Contribution

This paper presents **OpenMP Advisor**, a first-of-its-kind compiler tool that advises application scientists on various OpenMP code offloading strategies. This tool performs the following tasks to successfully address the challenges of effectively transforming an OpenMP code:

1. detect potentially offloadable kernel;
2. identify data access and modification in kernel;
3. recommend potential OpenMP variants for offloading that kernel to the GPU;
4. evaluate the profitability of each kernel variant via an adaptive cost model;
5. insert pertinent OpenMP directives to perform offloading.

Although the tool is currently limited to GPUs, it is extensible to other OpenMP-capable devices. In the rest of the paper, we first discuss state of the art work that is related to and precedes our work in Sect. 2. Then we define our OpenMP Advisor in Sect. 3. The experiments conducted for this paper are covered in Sect. 4 along with their analysis, and Sect. 5 concludes our work with discussions of our future goals.

2 Related Work

Many studies have looked into how to best manage GPU memory when data movement must be explicitly managed. For instance, Jablin et al. [10] provide a fully automatic system for managing and optimizing CPU-GPU communication for CUDA programs. Also OMPSan [1] performs static analysis on explicit data transfers are already inserted in OpenMP code. However, these studies do not address the issue of data transfer and the use of data reuse on GPU for implicitly managed data between different kernels. In one of our previous work [18] we proposed a technique for statically identifying data used by each kernel and automatically recognizing data reuse opportunities across kernels. In this tool, we make use of this method for data identification and management between the CPU and the GPU.

HPC applications are getting extremely heterogeneous, complicated, and increasingly expensive to analyze. Because of heterogeneity, a tool like OpenMP Advisor is required to help application scientists offload their code to GPUs. Other related research on automatic GPU offloading by Mendonça *et.al.* [16] and Poesia *et.al.* [22] can benefit from our tool by including our technique of data optimization and cost model in their framework, further reducing the challenges of using GPUs for scientific computing. However, developing a cost model

is time-consuming, and almost all modern compilers adopt a simple "one-size-fits-all" cost function that does not perform optimally in the situation of extreme heterogeneous architecture. In order to create a portable static cost model for our OpenMP Advisor tool, we utilize our ML based cost model, COMPOFF [17] which offers a new portable cost model that statically estimates the cost of OpenMP offloading on various architectures.

3 OpenMP Advisor

We design and develop the OpenMP Advisor, a compiler tool which transforms an OpenMP code to effectively offload to a GPU. This tool detects OpenMP kernels, proposes several GPU offloading OpenMP variants, and predicts the runtime of each kernel using a machine learning based cost model. Although the Advisor's initial implementation, as described in this paper, assists application scientists in programming for accelerators like GPUs, it can be expanded to support all OpenMP-capable devices. The tool has two modes: **Training** and **Prediction** mode. In the training mode, the Advisor makes use of data collected from multiple devices and compilers. It takes all benchmark codes as input and generates all possible code variants to run on the target device. Then it collects data from all generated codes to train an ML-based cost model for use in prediction mode. In prediction mode the tool does not need any interaction with the target device. It accepts C/C++ code as input and returns the best code variant that can be used to offload the code to the specified device. The tool can determine the kernels that are best suited for offloading by predicting their runtime using a machine learning-based cost model as defined in Sect. 3.2. The following are the key attributes of the OpenMP Advisor.

1. **Portable** – The Advisor's key feature is its portability across compilers and HPC clusters, as demonstrated in Sect. 4, which included four different compilers and seven HPC clusters with different GPUs.
2. **Static** – Since HPC GPUs are not always available during development, the Advisor performs all of its analysis at compile time and does not require runtime profiling in the prediction mode.
3. **Minimalistic** – The Advisor generates different kernel variants by adding OpenMP directives and clauses to the application's "omp target" regions without changing the kernel's body.
4. **Correctness** – The Advisor ensures that the generated code adheres to the OpenMP programming model, but it does not alter the kernel body or guarantee its correctness. Consequently, despite its ability to predict the optimal scenario for GPU offloading, it generates the top 10 code variants and lets the application scientist select which code to utilize.
5. **Adaptable** – The Advisor is adaptable enough to accept new applications by training the model on a proxy application if collecting real-time data is challenging or impractical for the real world application.

We used the LLVM compiler project [14] to develop the OpenMP Advisor. Despite the fact that LLVM's strength lies in the LLVM-IRs, our requirement is

to generate and return C/C++ code to the scientist. To do so, we need to be able to accurately insert OpenMP directives into a C/C++ file, which the LLVM-IR cannot guarantee. On the other hand Clang's AST closely resembles both the written C++ code and the C++ standard. Clang has a one-to-one mapping of each token to the AST node and an excellent source location retention for all AST nodes. Clang's AST is the best option for accurate source code information and inserting OpenMP directives into C/C++ files. Hence, we implemented the Advisor in Clang compiler (ver 14.0.0). Both the Training and Prediction modes have three major modules, which are explained in the following subsections - *Kernel Analysis*, *Cost Model* and *Code Transformation*.

3.1 Kernel Analysis

This is the first module which the Advisor calls in both the Prediction and Training mode. As the name implies, this module analyzes an OpenMP kernel. Overall, this module takes as input a C/C++ source file, analyzes it, and outputs all possible GPU offloading variants. This module is responsible for three tasks: *Identifying Kernels, Data Analysis* and *Variant Generation*.

Identifying Kernels — As the project's scope doesn't include automatically parallelizing the code, the user's input serves to identify a target region. An application scientist only needs to use the "`omp target` directive" to mark the region. We parse the clang AST to search for the `OMPTargetDirective` node, which is a subclass of the `OMPExecutableDirective` class and an instance of the `Stmt` class. We override the `OMPAdvisorVisitor` class's `VisitStmt` method and check each visited `Stmt` to see if they are the `OMPTargetDirective` node. Once a kernel is identified, we assign it a unique id and create an instance of the class "KernelInformation", in which we store information like, unique_id, start and end locations of the kernel, function from which the kernel is called, whether the kernel is called from within a loop and the number of nested `for` loops.

Data Analysis — The next step is to determine what data the kernel uses. We need to carefully manage data transfer between the CPU and GPU due to the high cost of transfers. We reuse our work on *Data Reuse Analysis* [18] to identify and utilize GPU data and improve overall execution time, since OpenMP doesn't specify how data should be handled in implicit data transfer. We use the Clang AST to implement "*live variable analysis*" for each kernel, concentrating only on variables used within a kernel. Our current approach only maps data between the CPU and GPU before and after the kernel. Managing data transfer during kernel execution is a future task. Before the variables are stored in the `KernelInformation` object, we classify them into five groups, based on how they are accessed before, during, or after the kernel:

- ***alloc***: Variables *assigned within* the kernel for the first time. Data need to be mapped, but no data transfer is required.
- ***to***: Variables *assigned before* but were only *accessed* within the kernel, not modified. Only host to device transfer of data is required.

- ***from***: Variables *assigned within* and *accessed after* the kernel definition. Only device to host transfer of data is required.
- ***tofrom***: Variables *assigned before*, *updated within* and *accessed after* the kernel definition. Data must be transferred both to and from the host and device.
- ***private***: Variables that are defined and used *only within* the kernel. No data transfer is required.

Data labeled *alloc*, *to*, and *tofrom* are mapped in "`omp target enter data`" directives before the kernel, while data labeled *from* and *tofrom* are mapped in "`omp target exit data`" directives after the kernel.

Variant Generation — Finally, we generate a number of different kernel variants that can be used to offload the kernel to the GPU. We'll start by counting how many nested collapsible for loops are there. In the current implementation, we can check up to four levels of collapsing the for loops. We chose four nested loops (similar to Code 1.2) because the Wilson Dslash kernel has four for loops. Each of these for loops is given a unique Loop number ranging from $0-3$. Loop 0 is always expected to be distributed across all teams on the GPU.

```
#pragma omp target teams distribute collapse(1)
  for (int i = 0; i < N_i; i++) {  ¡= Loop 0
#pragma omp parallel for collapse(3) schedule(static)
    for (int j = 0; j < N_j; j++) {  ¡= Loop 1
      for (int k = 0; k < N_k; k++) {  ¡= Loop 2
        for (int l = 0; l < N_l; l++) {    ¡= Loop 3
          /* ... COMPUTE ... */
  }}}}
```

Code 1.2. A variant of four nested "`for`" loops for GPU offloading

The variants are generated based on the collapse values used in `distribute` and `parallel for` directive, position of the `parallel for` directive, loop iteration's scheduling type and host-device data transfer. The total number of `for` loops and the position of the `parallel for` directive determine the maximum value of `collapse` that can be used in the `teams distribute` and `parallel for` directives. Suppose there are four `for` loops, as in Code 1.2. If the `parallel for` directive is at Loop 0, the "`teams distribute parallel for`" directive will be combined and thus the `collapse` clause for `distribute` directive doesn't exist. If the `parallel for` directive is located on Loop x (where $1 \leq x \leq 3$), then the maximum possible value of collapse for the `teams distribute` directive is x. While the maximum possible value of `collapse` for the `parallel for` directive is $(NUM - x)$, where NUM is the total number of for loops. The scheduling type of the loop iteration could be one of `static`, `dynamic` or `guided`. Using different permutations of these parameters, we could generate a variety of GPU offloading code variants. Once all of the variants have been generated, we use our static cost model to predict the runtime of each of these generated kernels.

3.2 Cost Model

A compile-time cost model is required to select the best option from all the variants generated by the Kernel Analysis module. Most modern compilers employ analytical models to calculate the cost of executing the original code as well as the various offloading code variants. Building such an analytical model for compilers is a difficult task that necessitates a lot of effort on the part of a compiler engineer. Recently, machine learning techniques have been successfully applied to build cost models for a variety of compiler optimization problems. For our tool we extended our previous work on COMPOFF [17] to be used as our cost model. COMPOFF is a machine learning based compiler cost model that statically estimates the cost of OpenMP offloading using an artificial neural network model. Their results show that this model can predict offloading costs with an average accuracy greater than 95%. The major limitation of COMPOFF was that they had to train a separate model for each variant. In our work, we add more training data and extend it to train a single cost model for all variants.

As soon as we know the prediction for the generated variant, we store it in the instance of the `KernelInformation` class so that the Kernel Transformation module can use it. But the biggest challenge in implementing an ML based cost model is the lack of available training data. To overcome this problem, we wrote additional benchmark applications (like the Pearson's Correlation Coefficient (correlation), Covariance (covariance), Laplace's Equation (laplace), Matrix-Matrix Multiplication (mm), Matrix-Vector Multiplication (mv), Matrix Transpose (mt)) and adopted some benchmarks from the Rodinia benchmark suite [4] (like the Breadth First Search (bfs), Gaussian Elimination (gauss), K-Nearest Neighbor (knn) and Particle Filter (particle)). The goal is to include a broader class of benchmarks that would cover the spectrum of statistical simulation, linear algebra, data mining, etc. We also developed a proxy app that has same number of loops and performs similar computation to our target app, the Wilson Dslash operator. Whenever it is difficult to collect data on real applications, proxy apps help us collect more data. More applications from various other domains will be added to this repository in the future.

3.3 Kernel Transformation

In the Kernel Transformation module we need to actually transform the original source code based on the analysis and predictions from the previous modules. For the given kernel, we generate every possible code variation in the Training mode. However, before we can generate code in Prediction mode, we must first address another crucial question. Which code should we generate? Should we only generated code for the fastest kernel? Regrettably, once the directives are in place, neither the Advisor nor OpenMP validate the kernel's correctness. This is in line with the OpenMP philosophy as well. As a result, we can only guarantee the correctness of the generated OpenMP directive in our framework.

So how can we overcome this problem? We could generate code for every possible variation, as we do during training, and let the user choose which one

```
0: // Predicted Runtime: 1.2 s
1: #pragma omp target enter data map(...)
2: #pragma omp target teams distribute collapse(2)
    for (int i = 0; i < N_i; i++) {
3:
      for (int j = 0; j < N_j; j++) {
4: #pragma omp parallel for schedule(dynamic)
        for (int k = 0; k < N_k; k++) {
5:
          for (int l = 0; l < N_l; l++) {
            /* ... COMPUTE ... */
    }}}}
6: #pragma omp target exit data map(...)
```

Code 1.3. Location of the seven generated code.

to use. But this means that users will be overwhelmed with information. Alternatively, we could ask the user to provide a number for the maximum number of codes to generate. The predicted runtime can be put as a comment before the kernel in every piece of code. The application scientist will then have more power to accept or reject the generated code. We will be able to produce a single code and provide it to the user once the issue of validating an OpenMP code for correctness is resolved. Until then, our Advisor will be able to generate the top best variants as specified by the application scientist. Regardless, we need to write a module to modify the existing source code and generate a new code. Clang provides the `Rewriter` [5] interface, whose primary function is to route high-level requests to the involved low-level `RewriteBuffers`. A `Rewriter` assists us in managing the code rewriting task. In the `Rewriter` interface we can set the `SourceManager` object which handles loading and caching of source files into memory. The `SourceManager` can be queried to obtain information about `SourceLocation` objects, which can then be converted into spelling or expansion locations. The `Rewriter` is a critical component of our plugin's kernel transformation module. The strategy used here is to meticulously alter the original code at crucial locations to carry out the transformation rather than handling every possible AST node to spit back code from the AST. For this we use the `Rewriter` interface, an advanced buffer manager that effectively manipulates the source using a rope data structure. For each `SourceManager`, the `Rewriter` also stores the low-level `RewriteBuffer`. Together with Clang's excellent retention of source location for all AST nodes, `Rewriter` makes it possible to remove and insert code very precisely in the `RewriteBuffer`. When the update is finished, we can dump the `RewriteBuffer` into a file to obtain the updated source code.

Finally, we create a vector of seven strings. The location of these seven strings are shown in Code 1.3. The first string (at #0) is always the comment that maintains the text – "`Predicted Runtime: ## s`". As the text suggests `##` is the predicted runtime for this particular kernel variant. This string is always placed before the kernel's start location. Then comes the `target enter data` construct (at #1). This directive handles what memory on the GPU needs to be created for the kernel and what data needs to be sent to the GPU before execution. This string is always placed right after the comments string. The next string (at #2) contains the OpenMP directive which specifies that this is the kernel to offload to the target. To gain maximum performance out of the GPU, we should always distribute

Table 1. Clusters and Compilers used in experiments

Cluster	GPU	Compiler	Version
Summit [20]	NVIDIA Tesla V100	LLVM/clang (nvptx)	13.0.0
		GNU/gcc (nvptx-none)	9.1.0
Corona [13]	AMD Radeon Instinct MI50	LLVM/clang (rocm-5.3)	15.0.0
Ookami [3]	NVIDIA Tesla V100	LLVM/clang (nvptx)	14.0.0
Wombat [21]	NVIDIA Tesla A100	LLVM/clang (nvptx)	15.0.0
Seawulf [23]	NVIDIA Tesla K80	LLVM/clang (nvptx)	12.0.0
		NVIDIA/NVC	21.7
Intel DevCloud [9]	Intel Xeon E-2176 P630	Intel/icpx	2021.1.2
Exxact	NVIDIA GeForce RTX 2080	LLVM/clang (nvptx)	14.0.0

the kernel across all the teams available in the GPU. Hence this string always contain the directive – "`#pragma omp target teams distribute`". The variant determines whether this directive contains any other clauses such as `collapse` or `map`, and what will be the values of the clause. This string is always placed immediately before the kernel's start location, but after the `target enter data` string. The remaining strings (#3, #4 and #5 if required by the variant) are placed just before the start location of their nested `for` loop. If these strings are not needed by a variant, they are left empty and no code is inserted in their location. The last string (at #6) is the `target exit data` construct, which identifies the data that must be released from the GPU or returned to the CPU. If not empty, each of these strings is always terminated by a new line. Once these seven strings are in their proper location, the code is dumped into a new C++ file and returned to the application scientists, who can choose to accept or reject the best code based on the kernel runtime provided in the comment.

4 Experiments and Evaluations

We used several clusters (on each using a single GPU) and compilers, as shown in Table 1, to perform multiple experiments and evaluate our tool. For the purposes of this study, we only use one GPU per node on the cluster. The management of multiple GPUs is left for future research. The three modules explained in Sect. 3 need different experiments.

4.1 Experiment 1 - Data Analysis

First, we test our Advisor against all benchmark applications to determine whether or not data is correctly identified and generated. In order to conduct this experiment, we made use of our Advisor to generate code that used the correct data between the host and the device. Additionally, we manually altered each benchmark algorithm to map all data to and from the GPU. We executed

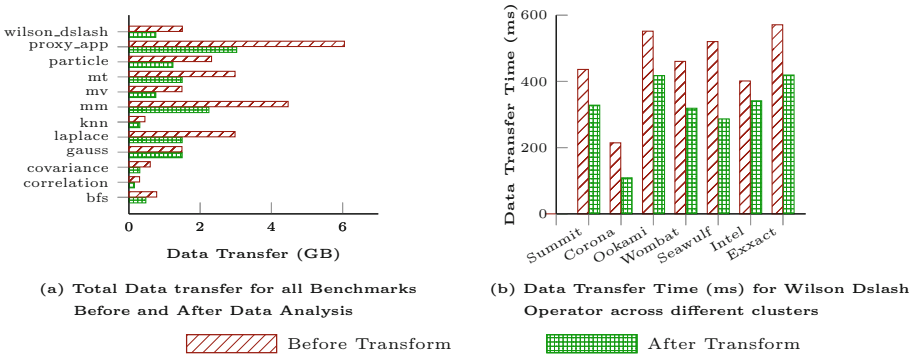

Fig. 1. Data transfer Before and After Code Transformation

all the codes on each cluster, from Table 1, and collected data about the volume and the duration of the data transfer. We found that the Advisor improved the data management in all cases. Figure 1(a) shows the amount of data transferred (in GB) between the CPU and the GPU before and after transformation for all benchmark applications. After applying our transformation, we can clearly see that the amount of data transfer has indeed been considerably reduced. Reduced data transfer has an impact on all applications' data transfer times. Along with reduced data transfer, the interconnecting bus between CPU and GPU (its version and number of lanes), affects the data transfer times of all applications. On all the available clusters, we ran these applications and collected the data transfer times. Figure 1(b) shows the data transfer time for the Wilson Dslash Operator across different clusters.

4.2 Experiment 2 - Code Generation

In the second experiment, we use our Advisor to generate every possible code combination using each of the Benchmark applications, as discussed in Sect. 3.1. We used the compilers listed in Table 1 to compile all of these codes for various clusters. Some compilers (NVIDIA/nvc on Seawulf and LLVM/Clang 15 on Wombat) do not support dynamic or guided scheduling on a GPU, resulting in compilation failure. Apart from that, all of the codes successfully compiled and ran on their respective clusters. We collected the runtime of each of the kernels in this experiment, to be used by our cost model. We collected the data for the Intel architecture a while ago, and we don't currently have access to the cluster to conduct new experiments. As a result we had very limited data for Wilson Dslash Operator and no data for our proxy_app on the Intel architecture. We were unable to gather many data points for the Exxact machine (with NVIDIA GeForce GPUs) due to the unavailability of compute nodes. Both these clusters has only around 2,000 data points each. Seawulf has NVIDIA K80 GPUs, which is the slowest of the GPUs we're using in our experiment. So each kernel runs longer on Seawulf than it would on any other cluster. On the other hand, most variants of kernels failed to compile on Wombat due to their compilers not

supporting dynamic and guided scheduling on GPUs. Due to these reasons, we could only collect around 3,000 data points on Seawulf and Wombat. All our kernels compiled and ran successfully on Summit, Corona and Ookami and we were able to collect over 10,000 data points on each of these architectures.

4.3 Experiment 3 - Cost Model

To build our cost model, we extended our COMPOFF cost model from six variants to all 84 variants. We build our cost model in the testing mode and then use it to predict the runtime in the prediction mode. Our cost model utilizes an MLP model with six layers: one input layer, four hidden layers, and one output layer. We set the number of neurons on multiples of the number of input features rather than choosing a random number of neurons in each hidden layer or conducting an exhaustive grid search (number of neurons in the first layer). As a result, the first, second, third, and fourth hidden layers, with 33 input features, have 66, 132, 66, and 33 neurons, respectively. The weights of linear layers are set using the glorot initialization method, which is described in [7]. The bias is set to 0, and the batch size for training data is set to 16 in all runs.

$$RMSE(\bar{y}, y) = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(\bar{y_i} - y_i)^2} \tag{2}$$

$$NRMSE(\bar{y}, y) = \frac{RMSE(\bar{y}, y)}{(y_{max} - y_{min})} \tag{3}$$

As the underlying optimization algorithm, we evaluate SGD (Stochastic Gradient Descent), Adam [12] and RMSprop [8]. We chose the RMSprop optimization algorithm as the underlying optimization algorithm, with an initial learning rate of 0.01 that is stepped down by a factor of 0.1 every 30 epochs and weight decay of 0.0001 for 150 epochs. We use the Root Mean Square Error (RMSE) loss function defined in Eq. 2, where $\bar{y_i}$ and y_i represent the predicted and ground truth runtimes, respectively.

We split the dataset used by all benchmark applications into two parts: training (80%) and validation(20%). The validation set do not occur in any learning for the model. The only augmentation applied to the training and validation data is Z-score standardization. The model is trained using the training set, and after that, testing data are given to the model to test its performance. In order to determine the standard deviation of the prediction errors, we compute the RMSE. The lower this value, the better our model. However, what constitutes a good RMSE is determined by the range of data on which we are computing RMSE. One way to determine whether an RMSE value is "good" is to normalize it using the formula shown in Eq. 3. This yields a value between 0 and 1, with values closer to 0 indicating better fitting models. Having a model with a normalized RMSE of less than 0.1 is considered successful in this study.

We observed strong correlation between actual and predicted data in Fig. 2, indicating that a simple MLP performs admirably in all our applications. It was anticipated that Intel and Exxact's model would perform the worst because of the lack of data. However, the model for Exxact performed better than Intel's due to the availability of more data for the proxy_app. Both Wombat and Seawulf

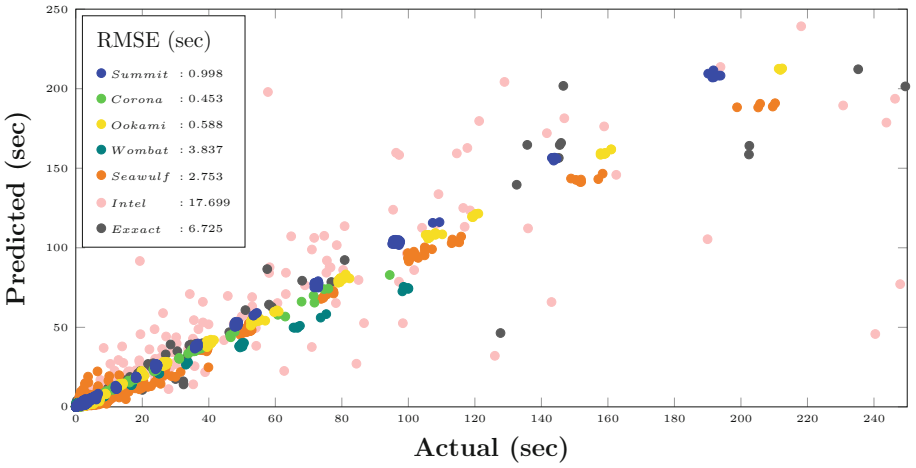

Fig. 2. Validation of Cost Model on different clusters

performed moderately well when compared to other models trained on a larger dataset. It is still an open question that how much data is enough data to train an ML model. We have observed from Fig. 2, however, that if we have more than 10,000 data points for our model, we will be able to train a model that is much more acceptable.

4.4 Experiment 4 - Prediction

Finally, for our final set of experiments we use our Advisor to predict the top 10 best variants for the Wilson Dslash Operator. Once the top 10 variants are identified we use the Code Transformation module to generate those 10 code variants and return them back to the user. The Advisor takes as input the base

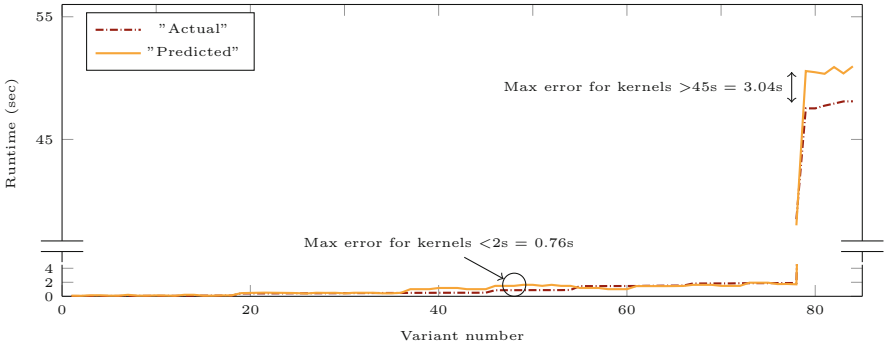

Fig. 3. Wilson Dslash operator's Actual and Predicted Runtimes (in sec) on Summit for all variants sorted by runtime.

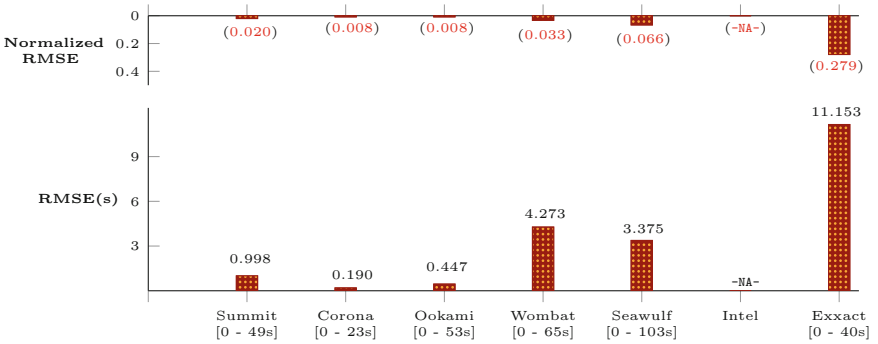

Fig. 4. RMSE and Normalized RMSE for runtime prediction of Wilson Dslash operator on different clusters (runtime range for each cluster is mentioned below their name).

Wilson Dslash kernel, where L_X, L_Y, L_Z, L_T are set at $32, 32, 32, 16$ each, and generates the top 10 best kernels as predicted by the cost model. As shown in Fig. 3, we plot the actual and predicted runtimes of all the 84 generated variants (sorted by actual runtime) of one such kernel when run on the Summit supercomputer. We can clearly see a strong correlation between the actual and predicted runtime for all the variants. The same correlation can be found in almost all kernels across all clusters. In Fig. 4, we display the Wilson Dslash operator's RMSE and normalized RMSE for each cluster. The range of runtimes (in seconds) for each cluster is mentioned below their name in the plot. We currently do not have access to the Intel cluster to conduct new experiments, and the Intel dataset contained very few data from the targeted Wilson Dslash kernel and none from our proxy_app. So, even if we make a prediction using this model, there is no way to validate it. Consequently, we did not conduct this experiment on the Intel architecture and the result is marked as $-NA-$. As expected on Exxact, the target kernel's RMSE increased significantly (11.153s) due to less data in its dataset. Even with a normalized RMSE of 0.279, it fell short of our expectation of 0.1. Nonetheless, this model demonstrated some correlation between the actual and predicted data. In contrast, Wombat and Seawulf performed reasonably well and were able to predict the top 10 kernel variants despite having an RMSE of 4.273s and 3.375s, respectively. However, with 0.033 and 0.066, respectively, their normalized RMSE was well within our expectation. As per our observation, their RMSE can also be improved by adding more data for these clusters. Finally, as shown in Fig. 4, the RMSE rates for Summit, Corona, and Ookami are less than one second each, and they were able to accurately predict the top ten kernel variants.

5 Conclusion and Future Work

In this paper, we introduced the OpenMP Advisor, a compiler tool that advises application scientists on various OpenMP code offloading strategies. Although the tool is currently restricted to GPUs, it can be extended to support other

OpenMP-capable devices. Using our Advisor, we were able to generate code of multiple applications for seven different architectures, and correctly predict the top ten best variants for each application including a real world application (the Wilson Dslash operator) on every architecture. Preliminary findings indicate that this tool can assist compiler developers and HPC application scientists in porting their legacy HPC codes to the upcoming heterogeneous computing environment. As a next step, we will extend our tool to 1) Data synchronization between host and device during kernel execution 2) Offload computation to multiple GPUs [11] via tasks 3) Predict the best variants for a variety of data sizes, and then use the OpenMP metadirective [19] directive to generate multiple directive variants for each range and 4) Extend the Advisor to other directive-based models, such as OpenACC. This tool is a first-of-its-kind attempt to build a framework for automatic GPU offloading using OpenMP and machine learning; as a result, there is plenty of room for improvement.

Acknowledgement. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is also based upon work supported by the National Science Foundation under grant no. CCF-2113996. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the SeaWulf computing system, which was made possible by a $1.4M National Science Foundation grant (#1531492).

References

1. Barua, P., Shirako, J., Tsang, W., Paudel, J., Chen, W., Sarkar, V.: OMPSan: static verification of OpenMP's data mapping constructs. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 3–18. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_1
2. Brower, R., Christ, N., DeTar, C., Edwards, R., Mackenzie, P.: Lattice QCD application development within the us doe exascale computing project. EPJ Web Conf. **175**, 09010 (2018). https://doi.org/10.1051/epjconf/201817509010
3. Burford, A., et al.: Ookami: deployment and initial experiences. In: Practice and Experience in Advanced Research Computing, pp. 1–8. ACM, New York (2021)
4. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE international symposium on workload characterization (IISWC), pp. 44–54. IEEE (2009)
5. Clang: Clang Rewriter class reference (2021). https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html
6. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
7. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, pp. 249–256. JMLR Workshop and Conference Proceedings (2010)

8. Hinton, G., Srivastava, N., Swersky, K.: Neural networks for machine learning. Lecture 6a Overview of Mini-batch Gradient Descent, vol. 14, no. 8, p. 2 (2012)
9. Intel: Intel Developer Cloud (2021). https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html
10. Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I.: Automatic CPU-GPU communication management and optimization. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 142–151 (2011)
11. Kale, V., Lu, W., Curtis, A., Malik, A.M., Chapman, B., Hernandez, O.: Toward supporting multi-GPU targets via taskloop and user-defined schedules. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) IWOMP 2020. LNCS, vol. 12295, pp. 295–309. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_19
12. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015, Conference Track Proceedings (2015). http://arxiv.org/abs/1412.6980
13. Laboratory, L.L.N.: LLNL - Corona (2019). https://hpc.llnl.gov/hardware/compute-platforms/corona
14. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, pp. 75–86. IEEE (2004)
15. Lin, M.: Optimization of the domain wall dslash kernel in columbia physics system, p. 269 (2016)
16. Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., Pereira, F.M.Q.: DawnCC: automatic annotation for data parallelism and offloading. ACM Trans. Archit. Code Optimiz. (TACO) **14**(2), 13 (2017)
17. Mishra, A., Chheda, S., Soto, C., Malik, A.M., Lin, M., Chapman, B.: COMPOFF: a compiler cost model using machine learning to predict the cost of openmp offloading. In: 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 30 May - 3 June 2022. IEEE (2022)
18. Mishra, A., Malik, A.M., Chapman, B.: Data transfer and reuse analysis tool for GPU-offloading using openMP. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) IWOMP 2020. LNCS, vol. 12295, pp. 280–294. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_18
19. Mishra, A., Malik, A.M., Chapman, B.: Extending the LLVM/clang framework for openMP metadirective support. In: 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar), pp. 33–44. IEEE (2020)
20. ORNL: Oak Ridge Leadership Computing Facility - Summit supercomputing cluster (2017). https://www.olcf.ornl.gov/summit/
21. ORNL: Oak Ridge Leadership Computing Facility - Wombat cluster (2020). https://www.olcf.ornl.gov/olcf-resources/compute-systems/wombat/
22. Poesia, G., Guimarães, B., Ferracioli, F., Pereira, F.M.Q.: Static placement of computation on heterogeneous devices. In: Proceedings of the ACM on Programming Languages 1(OOPSLA), pp. 1–28 (2017)
23. Stony Brook University: Seawulf, computational cluster at stony brook university (2019). https://it.stonybrook.edu/help/kb/understanding-seawulf