COMPOFF: A Compiler Cost model using Machine Learning to predict the Cost of OpenMP Offloading

Alok Mishra*, Smeet Chheda*, Carlos Soto†, Abid M. Malik†, Meifeng Lin†, Barbara Chapman*†
*Stony Brook University, Stony Brook, NY - 11794, USA
Email: (almishra,schheda,bchapman)@cs.stonybrook.edu
†Brookhaven National Laboratory, Upton, NY - 11973, USA
Email: (csoto,amalik,mlin,bchapman)@bnl.gov

***Abstract*—The HPC industry is inexorably moving towards an era of extremely heterogeneous architectures, with more devices configured on any given HPC platform and potentially more kinds of devices, some of them highly specialized. Writing a separate code suitable for each target system for a given HPC application is not practical. The better solution is to use directive-based parallel programming models such as OpenMP. OpenMP provides a number of options for offloading a piece of code to devices like GPUs. To select the best option from such options during compilation, most modern compilers use analytical models to estimate the cost of executing the original code and the different 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. In this paper, we present COMPOFF, a cost model that statically estimates the Cost of OpenMP OFFloading using a neural network model. We used six different transformations on a parallel code of Wilson Dslash Operator to support GPU offloading, and we predicted their cost of execution on different GPUs using COMPOFF during compile time. Our results show that this model can predict offloading costs with a root mean squared error in prediction of less than 0.5 seconds. Our preliminary findings indicate that this work will make it much easier and faster for scientists and compiler developers to port legacy HPC applications that use OpenMP to new heterogeneous computing environment.**

***Keywords*-openmp, gpu, machine learning, cost model**

I. Introduction

Since the end of Dennard scaling hardware developers have improved chip performance by configuring a growing number of compute cores. This multi-core processor technology was rapidly adopted by the High Performance Computing (HPC) community. It requires the modification of application codes to exploit the cores, e.g. by inserting pthreads or OpenMP constructs into the source code. In the last decade, General Purpose Graphics Processing Units (GPGPUs) have been attached to the multicore processors on many HPC platforms in order to benefit from their ability to handle large amounts of data parallelism with low power consumption. The trend toward heterogeneous HPC platforms is clearly visible in the most recent TOP500 list; while there are notable exceptions, a large fraction of the systems are heterogeneous with in most cases either NVIDIA GPUs or Intel Xeon Phis delivering high performance per watt. The second-ranked Summit supercomputing cluster is composed of two IBM PowerPC9 multicore CPUs with six NVIDIA V100 GPUs per node and three kinds of memory. Moving forward, we expect HPC platforms to be more diverse and potentially include domain-specific accelerators designed for specialized paradigms like machine learning, neuromorphic computing and ultimately quantum computing, leading to an era of extreme heterogeneity.

The presence of a single type of accelerator on a node already poses a challenge to current programming environments; we are not yet well prepared for a future in which multiple types of heterogeneous accelerators may be configured. Many application developers are adapting their codes to exploit GPUs. Lattice Quantum Chromodynamics [1], an application developed under the US Department of Energy's ECP project, is one such application that can greatly benefit from the use of accelerators such as GPUs. Unfortunately, effectively utilizing GPUs is a time-consuming endeavor that may necessitate re-engineering data structures as well as large regions of code in order to maximize the GPU's computational power while minimizing overheads. It will be far more difficult to develop code for systems with extreme heterogeneity containing different devices. It is therefore imperative to develop techniques that will relieve the application developers of the burden of such development.

Using a directive-based programming model, such as OpenMP, the de-facto programming standard for parallel programming in C/C++ and Fortran, is one way to handle portability across architectures. OpenMP now supports GPU

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).

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00074

offloading and is planning a transition to extreme heterogeneity [2], rendering the underlying hardware transparent and allowing for greater portability. Nevertheless, even with directive-based programming models like OpenMP, optimizing large scale applications with tens to hundreds of thousands of lines of code remains an arduous task.

In this work, we explore state of the art ML techniques to develop COMPOFF (Cost of OpenMP OFFloading), a first of its kind compiler cost modeling tool that employs ML to predict the execution time of an OpenMP kernel on GPUs during compilation. We first discuss some cutting-edge work that is related to and precedes our work in Section II, followed by Section III's discussion of the challenges encountered during offloading using OpenMP. In Section IV, we introduce static program features, that can be used to create a static cost model using Machine Learning, independent of compilers and hardware dependencies, as well as how we synthesize data to train our model. The design of our model is then presented in Section V. Section VI covers the experiments carried out in this paper in order to predict the cost of offloading the Wilson-Dslash stencil operator [3] for Lattice Quantum Chromodynamics application. The results are analysed in Section VII. Finally, we conclude our work with future discussion in Section VIII.

II. Related Work

Compiler engineers are developing a number of frameworks [4], [5], [6] to assist application developers deal with extreme heterogeneity more effectively. These frameworks require analytical cost models to help them make better decisions when selecting the choice of optimization or transformation required by the application. However, developing a cost function is time-consuming, and almost all modern compilers, including LLVM/Clang, use a simple "one-size-fits-all" cost function that does not provide the best performance in the case of diverse architecture. Hand-tuned cost functions are currently popular, but calculating the costs and benefits of a compiler optimization requires a deeper understanding of the underlying hardware. Despite its effectiveness, manually constructing a cost model for a single architecture can take months. Since cost functions are critical and manual tuning is rather laborious, compiler engineers are investigating Machine Learning (ML) techniques as a means of automating this process.

Early work exploiting ML in compilers, like [7], primarily explored its use to help optimize sequential programs. However, with the proliferation of multi-core platforms and, more recently, heterogeneous systems, its application to the task of optimizing parallel programs has received significant attention in the last decade. A decision-tree-based approach has been developed to predict the scheduling policy for an OpenMP parallel region [8]. Furthermore, [9] optimizes OpenMP programs for scheduling policies and thread count using ML techniques. ML has been used to determine the optimum degree of parallelism for transactional memory [10] and hardware resource allocation [11]. The work presented in [12] and [13] applies ML to complex parallel programs and divide them among the available multi-core resources. The Petabricks project [14] employs a genetic search to tune algorithmic choices at compile time to reduce searching overheads. Tree and graph-based features have also been used by Malik et.al. [15] who present a unique graph-based approach for feature representation. ML techniques were used to build classifiers to determine whether to offload OpenCL code [16], and to select a clock frequency at which the processor should operate [17]. The work was reported to be extremely accurate, but the benefits could not be quantified because no modified code was generated. The results of prior efforts from applying ML on compiler optimizations are encouraging. However, new feature engineering practices that can help ML learn more about a code and its computational needs must be investigated.

III. OpenMP

The OpenMP API 4.0 specification (released in 2013 [18]) includes a collection of directives that tell the compiler when to offload a block of code to devices, like GPU, FPGA etc. However, achieving scalable performance on large parallel machines still necessitates significant effort in performance tuning, particularly in terms of cache management and locality, data and work sharing, and synchronizations.

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    float tmp = 0.0f;
    for (int k = 0; k < n; k++) {
      tmp += A[i*n+k] + B[k*n+j];
    }
    C[i*n+j] = tmp;
  }
}
```

Code 1: Sequential Matrix Multiplication program

```
#pragma omp target
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    float tmp = 0.0f;
    for (int k = 0; k < n; k++) {
      tmp += A[i*n+k] + B[k*n+j];
    }
    C[i*n+j] = tmp;
  }
}
```

Code 2: Matrix Multiplication on GPU using OpenMP

When each architecture supports either a distinct native language or alternative optimization methods for the same language, program portability becomes a major problem. This is especially important for users whose programs must run smoothly on systems with diverse node architectures, such as manycore vs. GPU-accelerated nodes, or when

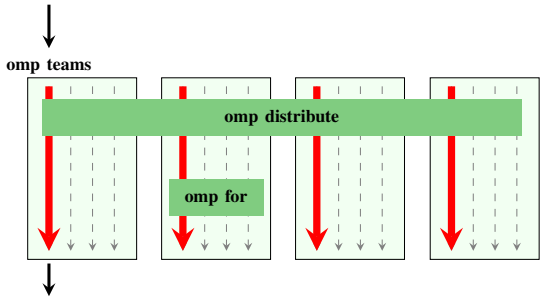


Figure 1: Teams distribute parallel for

dealing with multiple device memory accesses to the same data. In the future, the ultimate level of portability would be if we can have developers write a sequential code, like matrix multiplication in Code 1, and then build this code using any compiler on any platform and expect it to exploit all hardware and accelerator parallelism effectively.

Unfortunately, programming languages and compilers are far away from a state where they can handle portability without programmer intervention. Usually, a base language lacks all of the features that a software developer needs, and they rely on libraries to meet their performance goal. Architecture-specific libraries, like CUDA, can be used to extract all of NVIDIA GPU's performance to achieve accelerator parallelism, but portability suffers as a result. Directive-based programming languages, like OpenMP (e.g. Code 2), serve as a middle ground, providing a portable way to augment the base languages, while filling in the performance gaps required to support a specific architecture.

A. GPU Offloading in OpenMP

OpenMP provides additional information to the compilers, which bridges a gap from serial programming languages to parallel programming languages. GPUs are highly parallel, and the programmer should fully utilize it's parallel capacity to extract maximum performance. Simply parallelizing a code using *"omp parallel for"* will parallelize it for CPUs, but will not offload the computation to a GPU. OpenMP device offloading consists of two major components:

- Data mapping between host and device
- Offloading computations from host to device.

All data is initially stored in CPU memory, and GPUs have no access to it. To access host data, the GPU must first move the data from the host (CPU) to the device (GPU) using the `data map` clause, and then move the data back from the device to the host once the computation on the GPU is complete. The `omp parallel for` creates a single contention group of threads with shared memory and the ability to coordinate and synchronize, while the `omp target` directive marks a code section for offloading. But GPUs are not parallel worksharing machines.

On a platform like GPU, we expect a high degree of coarse grain parallelism across the entire device. This structure limits the degree of parallelism that a GPU can exploit. Instead, programmers can use multiple options provided by OpenMP directives, like *teams distribute*, which exposes coarse grained, scalable parallelism to the entire GPU. OpenMP teams and distribute are directives that spawn additional level of parallelism, as shown in Figure 1. Only one team and one member thread are active at the start of a target region. If we want to have multiple teams, we use the directive *teams distribute* first, which distributes the entire loop iteration space among all teams. Furthermore, if there are more nested parallel loops, we use the *parallel for* directive upon them to distribute the iterations of the nested loops among threads within a team. When there is only one level of parallel loops, or when the outer loop has enough parallelism, we use the combined directive *teams distribute parallel for* to distribute the iteration space of one loop among teams and threads within a team.

Threads are organized into groups using `teams`, and `distribute` allows to schedule a group of teams to run a loop. As there is no synchronization primitive to act as a barrier between threads belonging to different teams, `teams` appear to be similar to *CUDA threadblocks* on NVIDIA GPUs. As is usual, `parallel for` is used to parallelize the threads within each team. Coarse grained parallelism may be combined (Code 3) or split (Code 4). There could be other transformation as well which we could use to fully utilize the parallelism on a GPU. For instance, in Code 5 and 6 the programmer could swap the outer (iterating over `i`) and inner (iterating over `j`) loops and apply combined

```
#pragma omp target teams distribute parallel for
 for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    float tmp = 0.0f;
    for (int k = 0; k < n; k++)
      tmp += A[i * n + k] + B[k * n + j];
    C[i * n + j] = tmp;
  }
 }
```

Code 3: Combined parallelism

```
#pragma omp target teams distribute
 for (int i = 0; i < n; i++) {
#pragma omp parallel for
   for (int j = 0; j < n; j++) {
     float tmp = 0.0f;
     for (int k = 0; k < n; k++)
       tmp += A[i * n + k] + B[k * n + j];
     C[i * n + j] = tmp;
   }
 }
```

Code 4: Split parallelism

```
#pragma omp target teams distribute parallel for
  for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++) {
      float tmp = 0.0f;
      for (int k = 0; k < n; k++)
        tmp += A[i * n + k] + B[k * n + j];
      C[i * n + j] = tmp;
    }
  }
```

Code 5: Swap Combined parallelism

```
#pragma omp target teams distribute
  for (int j = 0; j < n; j++) {
#pragma omp parallel for
    for (int i = 0; i < n; i++) {
      float tmp = 0.0f;
      for (int k = 0; k < n; k++)
        tmp += A[i * n + k] + B[k * n + j];
      C[i * n + j] = tmp;
    }
  }
```

Code 6: Swap Split parallelism

```
#pragma omp target teams distribute parallel for \
                                  collapse(2)
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      float tmp = 0.0f;
      for (int k = 0; k < n; k++)
        tmp += A[i * n + k] + B[k * n + j];
      C[i * n + j] = tmp;
    }
  }
```

Code 7: Collapse Loop

```
#pragma omp target teams distribute parallel for \
                                  collapse(2)
  for (int j = 0; j < n; j++) {
    for (int i = 0; i < n; i++) {
      float tmp = 0.0f;
      for (int k = 0; k < n; k++)
        tmp += A[i * n + k] + B[k * n + j];
      C[i * n + j] = tmp;
    }
  }
```

Code 8: Swap Collapse Loop

and split parallelism on them as well. A programmer could also collapse the two loops as shown in Code 7 and 8 to exploit its full degree of parallelism. The implementation of collapse varies depending on the compiler, as does how it affects the outcome. But, how do we decide which of the these transformation will be better for our kernel and architecture? One possible approach is to take such decision during compile time using some static cost model.

IV. Machine Learning in Compilers

Despite their efficacy, designing hand tuned cost models for each architecture is rather costly and time consuming. To automate the process, compiler developers are turning to machine learning techniques. Identifying the feature set that can be used to train an ML model is the first step toward using ML in compilers. Since the goal of this work is to maintain portability using OpenMP, we must look for features that are platform and compiler independent. This means we should only consider features from the original source code written by the programmer without applying any compiler optimizations on them. Furthermore, because we need to make predictions at compile time, these features should be static in nature. For the purpose of this research, we only consider the six transformations explained in Section III and shown in Code 3 – 8.

After studying several GPU cost models ([19], [20], [21]), we conclude that there are three major factors which affect the cost of execution of a kernel on a GPU - level of parallelism, memory access, and computation to be performed by

Types	Feature	Description
Parallelism	Outer	Outer Loop of Iteration
	Inner	Inner Loop of Iteration (Set to 0 when there is no nested loop)
	Schedule	static, dynamic, guided, auto, runtime
Memory	MemTo	Total memory transferred to GPU in bytes
	MemFrom	Total memory transferred from GPU in bytes
	VarDecl	Total number of variable declaration
	RefVar	Total number of variable referenced
	IntLiteral	Total number of integer constant referred
	FloatLiteral	Total number of floating point constant referred
	IntAssign	Total integer assignment
	FloatAssign	Total floating point assignment
Computation	IntAddSub	Total integer addition and subtraction
	FloatAddSub	Total floating point addition and subtraction
	IntMult	Total integer multiplication
	FloatMult	Total floating point multiplication
	IntDiv	Total integer division
	FloatDiv	Total floating point division
	Others	Total logical, relational and bitwise operations

Table I: Static Kernel features independent of architectures and compilers

the kernel. To train a model which could predict the cost of execution of above transformation, we extract static features from kernels, which can be grouped in accordance to these factors. A list of features that we consider in this paper can be found in Table I. Here we group the features based on level of parallelism, memory access and computation.

1) Level of parallelism.: To check for the level of parallelism we can look into the number of iterations of the *for loop*, the number of available threads and how OpenMP schedules the iterations between the threads. Usually, the number of available threads on a GPU and the scheduling-type used by OpenMP are present at the time of compilation, but the number of iterations are usually a variable. For this current study we make sure that the number of iteration in the *for loop* are also constant. Predicting the cost at runtime using variable iterations is part of the future work. The level of parallelism can be increased by collapsing the nested loops. In this work, we restrict the nesting of loops to two levels only, referring to them as the outer and inner loops.

2) Memory access.: In Section III, we discussed how the memory of the CPU and GPU differs and how data must be synced between them for the kernel to function correctly. Data movement between CPU and GPU (to and from) is a big factor which affects the execution time of a kernel. Another factor which affects the execution time of a kernel is how many times memory locations are declared, referred and written to.

3) Computation.: Finally we count the number of operations that occur in a kernel. One primary reason why a generic cost model with a "one size-fits-all" cost function (like [22]) does not deliver the best results is that it considers the cost of running all operations to be equal. But, in practice this is not true. For instance, a multiplication operation will take more time than an addition or logical operation. Also, floating point multiplication, might take different time than integer multiplication. Hence, we need to extract the number of times each of the operations are called. Still there are some operations which we can assume to have same cost of execution. For instance, we combine addition and subtraction which has the same cost of execution.

A. Data Collection

The absence of publicly available data is the most significant challenge that any ML engineer faces for compiler problems. We faced the same problem as well. So, the first step in developing a machine learning-based cost model was to create a dataset that could be used to train our model. And we needed a dataset which would cover all the features, defined in our feature set. To cover all the features, we wrote mini benchmark applications like Matrix Multiplication, Laplace Equation, SAXPY, Gauss Seidel Method and use some benchmarks from the Rodinia benchmark suite, like the BFS, LU-Decomposition, Particle Filter, etc. We created over 10,000 different kernels by varying the levels of parallelism and data used for all these benchmark applications. Then, we extracted and collected data on the feature set defined in Table I as well as the execution time of each of these kernels.

B. Wilson Dslash Operator

In addition to the benchmark applications, we employ a kernel that represents the Wilson Dslash Operator [3] found in the Lattice Quantum Chromodynamics (LQCD) application. QCD is the theory of the strong nuclear force, one of nature's fundamental forces that holds quarks together to form protons and neutrons, which then form the nuclei of atoms. LQCD is a computer-friendly numerical framework for QCD, that employs the Wilson Dslash kernel, which is essentially a finite difference operator.

The Wilson Dslash operator, D, in four space-time dimensions is defined as

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

where 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. For the unpreconditioned Dirac operator, the complex fermion fields are represented by one-dimensional arrays of size $L_X \times L_Y \times L_Z \times L_T \times SPINS \times COLORS \times 2$ 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. $SPINS$ and $COLORS$ are the numbers of spin and color degrees of freedom, typically 4 and 3 respectively.

The above equation (Eq 1), when represented in a C++ code has four nested `for` loops iterating over L_T, L_Z, L_Y, L_X in that order. We modified the source code for this study to offload this kernel to the GPU using all six transformations described in Section III. We run this operator for different values of L_T and L_Z (the two outer loops) to determine the runtime on Summit [23] and Seawulf [24] HPC clusters. On both clusters, we collected approximately 40 data points for each transformation. We used some of these data points to train our model and kept the rest to evaluate our model's prediction accuracy, as described later in Section VI.

V. Training Models

The available data and the target execution time predictions leans towards to a supervised machine learning (ML) approach, which involves learning a feature representation and fitting a predictive model using training data containing ground-truth targets. This can be accomplished with a number of machine learning model types, all designed to ultimately perform a regression task. There are several ways of performing regression using the collected data. In

all cases, the available data – populated parameters and corresponding measured execution time – is divided into disjoint training and testing sets, with model's parameters determined by fitting on the training set, and the model's performance determined separately on the hold-out test set. This split allows for the evaluation of the models' generalized performance on previously unseen data.

A. Why Regression?

Since the runtime is a positive real-valued variable, the prediction task for all ML models is a regression problem, as opposed to classification, which has a limited set of possible outcomes. The rationale for using machine learning regression models to predict the cost of OpenMP offloading is the reasonable expectation that when well-selected and measurable kernel parameters are combined, they will be strongly predictive of kernel execution time, but the nature of this predictive correlation may be complex. ML approaches are well suited to determining these correlations in a data-driven manner – without exhaustive reasoning and debate about the relative importance of and relationship between particular features.

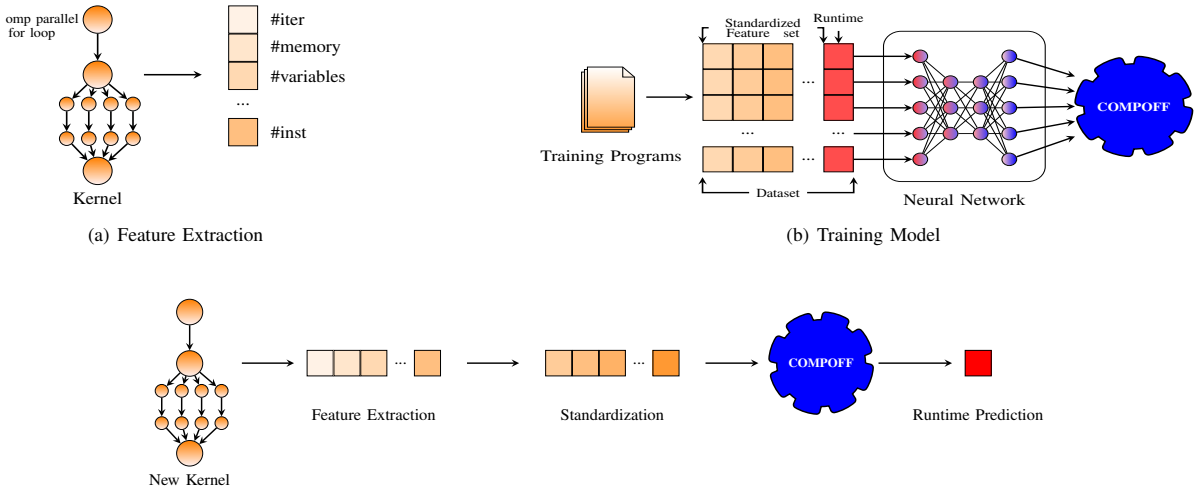
The standard linear regression approach to this problem determines the specific linear combination of all input parameters that best fits the target values. Although the restriction to linear relationships is constraining in practice, the easy-to-interpret weights and biases among features make this a useful and accessible baseline. One addition we made in this work was the extension of input parameters by appending a log-transformed duplicate set, thus enabling approximate representation of multiplicative and rational relationships (as sums and differences of logs) as well as linear ones.

B. Neural Network

Linear regression applies a linear fit across all input in the original parameter space, and the data must have a linear relationship in order for this to work. This condition is violated in the data we collect – some features are strongly correlated, while relationships for others cannot be determined in the original parameter space. In order to find a better fitting non-linear data relationship, stronger ML models can map inputs to a different parameter space (e.g. in a higher dimension). Support Vector Machines (SVMs) and Artificial Neural Networks (ANNs) are examples of ML techniques that can learn complex feature relationships that are impossible to capture with linear regression while maintaining efficiency at inference time. We chose neural networks over SVMs because they can capture even more complex data relationships while being easier to parameterize and converge using gradient-based optimization algorithms (e.g., Stochastic Gradient Descent).

ANNs are a very diverse family of models; the class we employ is fully-connected feed-forward networks, also called Multi-Layer Perceptrons (MLPs). An MLP is essentially a stacked series of linear regression layers. The non-linearity is added by activation functions such as ReLU [25], sigmoid, tanh, functions etc, added on top of each linear layer. The key hyperparameters of MLPs are the number and size of the hidden layers which build up increasingly complex data representations before a final regression layer is applied to perform the ultimate prediction.

Figure 2 (a) and (b), gives a basic idea of the flow of training our model. When we want to use this model we



(a) Feature Extraction

(b) Training Model

(c) Prediction

Figure 2: Overview of the ML Model

extract the required features from the new kernel, append a log-transformed duplicate set, standardize these features using the mean (μ) and standard deviation (σ) from the original model, and pass the standardized data to our model, which returns the prediction of the kernel's execution time (as shown in Figure 2 (c)).

VI. Experiments and Evaluations

Section IV already discussed multiple code level transformations which we can use to offload an OpenMP parallel loop to the GPU. As a proof of concept for this paper, we only consider these six transformations, and create a model for each of them individually. We experiment on two different HPC clusters, each with an LLVM/Clang 13.0 compiler that supports GPU offloading:

1) Summit Supercomputer with NVIDIA V100 GPUs
2) Seawulf cluster with NVIDIA K80 GPUs

Despite the fact that each node on both clusters is connected to multiple GPUs, we only consider single GPU for the purposes of this research. Dealing with multiple GPUs will be left for future research. Each of these transformations has data in our dataset, with the kernel built on both clusters. We trained a total of twelve models, one for each of the six transformation, on both clusters.

COMPOFF employs an MLP model with six layers – 1 input, 4 hidden and 1 output neurons. Rather than selecting an arbitrary number of neurons in each hidden layer or perform an exhaustive grid search, we set the number of neurons on multiples of the number of input features (number of neurons in the first layer). Therefore, with 33 input features, the first, second, third and fourth hidden layers have 66, 132, 66 and 33 neurons respectively. Glorot initialization, described in [26], is used to set the weights of linear layers. The bias is initialized to 0. In all runs, the batch size for training data is set to 16.

We experiment with SGD (Stochastic Gradient Descent), Adam [27] and RMSprop [28] as the underlying optimization algorithm. We choose the RMSprop optimization algorithm with an initial learning rate of 0.01 that is stepped down every 30 epochs by a factor of 0.1 and weight decay of 0.0001 for 150 epochs because it gives us optimal performance on transformations for both HPC clusters.

The objective function to train all models is based on the Mean Square Error loss function defined by:

$$l(\bar{y_i}, y_i) = (\bar{y_i} - y_i)^2$$

where $\bar{y_i}$ and y_i represent the predicted and ground truth runtime, respectively.

Each training dataset is divided into train (72%), validation (8%) and testing sets (20%). The model **does not learn** from the validation and testing sets. Z-score standardization is the only augmentation applied to training and validation data. We use the training set to train the model and the unseen validation set to validate its growth. The computed statistics are then applied to testing data. For our second set of experiments with Wilson Dslash Operator data, we add 20 data points to the original dataset and hold the remaining 20 data points for evaluation. We calculate Root Mean Square Error (R.M.S.E.), as the standard deviation of the prediction errors. The lower this value, the more accurate our model is. In addition, we calculate the Mean Absolute Percentage Error (M.A.P.E.) to determine the accuracy of our model.

VII. Results and Analysis

A good statistical validation technique provides us with a thorough measure of our model's performance across the entire dataset. The validation set is selected at random for each data file. The validation set helps us in understanding the models progress. Once a training epoch is complete, we freeze the network and compute the R.M.S.E. and M.A.P.E. on the validation set. These values are expected to decrease as we train and then increase eventually due to overfitting.

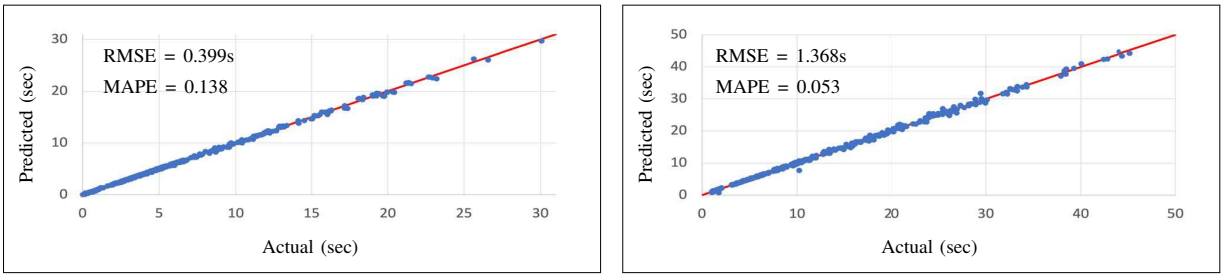
Following the training protocol outlined in the previous section, we investigate the models' fit across multiple data files. On regular microbenchmark data, we can see that a simple MLP performs impressively. The results are shown in Table II. Due to the small number of data points collected for the Wilson Dslash Operator, we can see that the model can adapt to the new application relatively well in some cases but not so well in others, as shown in Table III. The new application's few data points are clustered further apart

Transformation	Cluster	R.M.S.E. (sec)	M.A.P.E.
Collapse	Seawulf	0.279	0.030
	Summit	0.090	0.211
Combined	Seawulf	1.368	0.053
	Summit	0.399	0.138
Split	Seawulf	0.420	0.044
	Summit	0.112	0.227
Swap Collapse	Seawulf	1.242	0.055
	Summit	0.128	0.120
Swap Combined	Seawulf	0.669	0.027
	Summit	0.276	0.062
Swap Split	Seawulf	0.919	0.059
	Summit	0.454	0.284

Table II: Evaluation results on the microbenchmark data files for respective transform and cluster

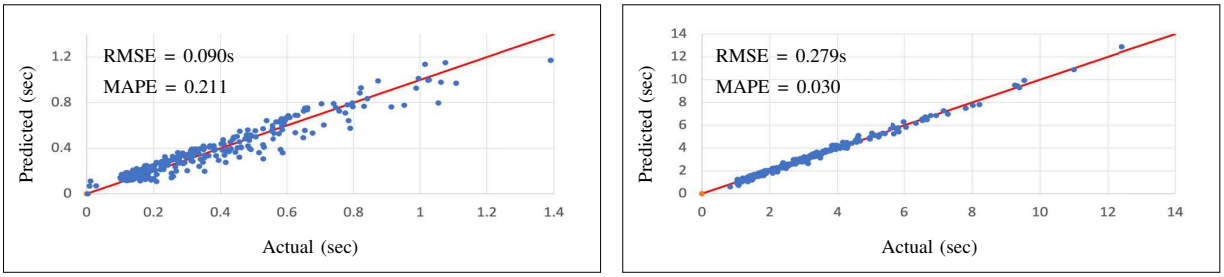
Transformation	Cluster	R.M.S.E.(sec)	M.A.P.E.
Collapse	Seawulf	0.143	0.576
	Summit	0.067	4.201
Combined	Seawulf	1.500	0.323
	Summit	0.333	0.259
Split	Seawulf	0.384	1.044
	Summit	0.041	0.584
Swap Collapse	Seawulf	8.972	3.305
	Summit	0.512	2.871
Swap Combined	Seawulf	1.771	3.321
	Summit	0.584	2.686
Swap Split	Seawulf	2.668	7.083
	Summit	0.032	0.515

Table III: Evaluation results on Wilson Dslash kernel data



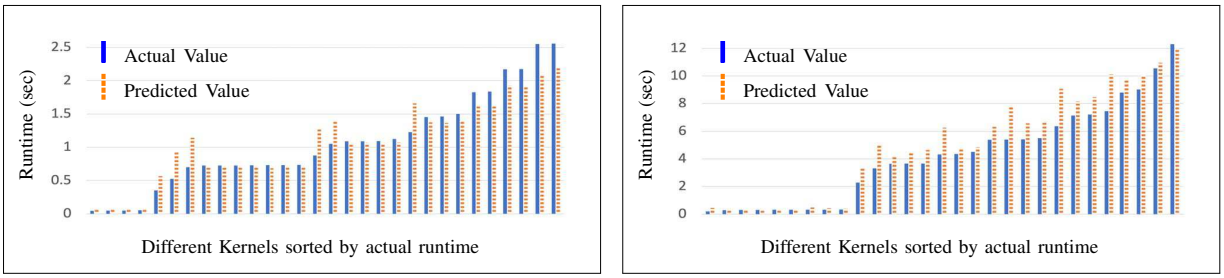
(a) Summit (b) Seawulf

Figure 3: Validation of Combined Offload on Summit and Seawulf



(a) Summit (b) Seawulf

Figure 4: Validation of Collapse Offload on Summit and Seawulf



(a) Summit (b) Seawulf

Figure 5: Wilson Dslash operator prediction on Summit and Seawulf

from the data collected by the microbenchmark application, resulting in a data imbalance. This is commonly referred to as the long-tail problem seen in many real-world applications, and is not in the scope of this work.

The goal of this work is to show that ML regression models can be used to predict the cost of OpenMP offloading for different kernels, and it is evidenced by the strong correlation between actual and predicted data in Figures 3 and 4. When looking at Figure 4(a), it may appear that the predictions on Summit are dispersed. However, when the scale of the steps on its y-axis and the R.M.S.E. values are taken into account, we can see that the actual and predicted data are strikingly similar. Furthermore, we see the model perform well on a wide range of data (from 100s of ms to a few hundred seconds), implying that a single model can be trained to predict high variance data with careful finetuning, hyperparameter setup, and a large amount of data. Because the R.MS.E. for these predictions is so low, we can ignore the higher M.A.P.E. values observed in some cases. This is also observed in the prediction for the Wilson Dslash operator data as evident in Figure 5 (the x-axis represents different Wilson Dslash kernels, sorted by actual execution time). The blue bar in the figure represent the actual runtime of the kernel, while the orange dashed bar represents the prediction by COMPOFF. We can see that in all cases, the prediction is close enough to the actual runtime.

VIII. Conclusion and Future Work

This work is a proof of concept that an ML model can be trained and used by compiler developers to make better decisions in offloading a kernel to a GPU using OpenMP. Our findings show that this model can predict offloading costs of several benchmark application and one real application (Wilson Dslash operator) with a Root Mean Squared Error of less than 0.5 seconds. It has some limitations though, biggest of which is the lack of an exhaustive dataset. The current dataset that we collected for the purpose of this work is sufficient for a proof of concept. But to create a portable cost model across multiple applications and architectures, we need to train the ML model extensively across several platforms, compilers and applications.

Even if we have an exhaustive dataset across various environments, another big challenge is runtime parameters tuning. It is observed that the same kernel, when run on the same platform multiple times show different execution time. It is seen that the execution time differ by a couple of seconds. Therefore, in terms of percentage variation, for smaller kernels which run for only a second or so, this variation could be huge. This brings in a lot of noise in our training set for smaller kernels. However, in large kernels, a couple of seconds of variation in the prediction is usually not a big issue and can be ignored. In actual practice, if a kernel is small and does not have enough computational work, a GPU execution is not justified. Therefore, currently, we leave the decision of how to interpret the prediction for smaller kernel upon the users of COMPOFF.

Other parameters that we have not considered in this work and that may affect the execution time of a kernel are:

- **Data affinity** – Data affinity is not yet supported by OpenMP and hence we are also not considering it.
- **Data reuse** – We have previously demonstrated [29] that reusing data between multiple kernels improves the overall execution time of an application. Currently, this work does not consider data reuse between multiple kernels, and each of the kernels are considered mutually independent with respect to data reusability.
- **Unified memory** – Previous studies [30], [31] have shown that using unified memory between CPU and GPU, an application can significantly improve OpenMP GPU offloading performance. However, we haven't yet considered unified memory in this project.
- **Data overfitting** occurs when a statistical model fits exactly against its training data, but fails against unseen data, defeating its purpose. Even though this is partially mitigated by the use of validation sets, regularization and dropout layers, it is not a foolproof method.
- **Constants** – As with most compile time cost models, COMPOFF can only predict the execution time of kernels whose data size and level of parallelism are available during compile time.

For our future work, we plan to apply ML techniques to aforementioned challenges. We also plan to work on improving feature selection and representation for parallel programming models. Recently, Graph Neural Network is gaining popularity for its performance for learning graph representation. We plan to use it in the future to learn characteristics of parallel code.

References

[1] R. Brower, N. Christ, C. DeTar, R. Edwards, and P. Mackenzie, "Lattice qcd application development within the us doe exascale computing project," in *EPJ web of conferences*, vol. 175. EDP Sciences, 2018, p. 09010.

[2] M. A. Heroux, R. Thakur, L. McInnes, J. S. Vetter, X. S. Li, J. Aherns, T. Munson, and K. Mohror, "Ecp software technology capability assessment report," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2020.

[3] M. Lin, "Optimization of the domain wall dslash kernel in columbia physics system," in *Proceedings of the 34th annual International Symposium on Lattice Field Theory (LATTICE2016). 24-30 July 2016. University of Southampton*, 2016, p. 269.

[4] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. Pereira, "Dawncc: automatic annotation for data parallelism and offloading," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, p. 13, 2017.

[5] A. Mishra, M. Kong, and B. Chapman, "Kernel fusion/decomposition for automatic gpu-offloading," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 283–284.

[6] G. Poesia, B. Guimarães, F. Ferracioli, and F. M. Q. Pereira, "Static placement of computation on heterogeneous devices," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[7] J. Monsalve, K. Harms, K. Kumaran, and G. Gao, "Sequential codelet model of program execution. A super-codelet model based on the hierarchical turing machine," in *IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware, IPDRM@SC 2019, Denver, CO, USA, November 22, 2019*. IEEE, 2019, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IPDRM49579.2019.00005

[8] M. Mirka, G. Sassatelli, and A. Gamatié, "Online learning for dynamic control of openmp workloads," in *9th International Conference on Modern Circuits and Systems Technologies, MOCAST 2020, Bremen, Germany, September 7-9, 2020*. IEEE, 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109/MOCAST49295.2020.9200292

[9] N. Denoyelle, B. Goglin, E. Jeannot, and T. Ropars, "Data and thread placement in NUMA architectures: A statistical learning approach," in *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*. ACM, 2019, pp. 39:1–39:10. [Online]. Available: https://doi.org/10.1145/3337821.3337893

[10] Y. Xiao, T. Jeyakumaran, E. Atoofian, and A. Jannesari, "Improving performance of transactional memory through machine learning," *Concurr. Comput. Pract. Exp.*, vol. 30, no. 10, 2018. [Online]. Available: https://doi.org/10.1002/cpe.4397

[11] R. M. Jenevein, D. DeGroot, and G. J. Lipovski, "A hardware support mechanism for scheduling resources in a parallel machine environment," in *Proceedings of the 8th Annual Symposium on Computer Architecture, Minneapolis, MN, USA, May 1981*, R. Y. Kain and W. R. Franta, Eds. IEEE Computer Society, 1981, pp. 57–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=801866

[12] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva, "Code mapping in heterogeneous platforms using deep learning and LLVM-IR," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 170. [Online]. Available: https://doi.org/10.1145/3316781.3317789

[13] H. Jordan, P. Thoman, J. J. D. Barrionuevo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, J. K. Hollingsworth, Ed. IEEE/ACM, 2012, p. 10. [Online]. Available: https://doi.org/10.1109/SC.2012.7

[14] S. P. Amarasinghe, "Petabricks: a language and compiler based on autotuning," in *High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26, 2011. Proceedings*, M. Katevenis, M. Martonosi, C. Kozyrakis, and O. Temam, Eds. ACM, 2011, p. 3. [Online]. Available: https://doi.org/10.1145/1944862.1944865

[15] A. M. Malik, "Automatic static feature generation for compiler optimization problems," in *AI 2011: Advances in Artificial Intelligence - 24th Australasian Joint Conference, Perth, Australia, December 5-8, 2011. Proceedings*, ser. Lecture Notes in Computer Science, D. Wang and M. Reynolds, Eds., vol. 7106. Springer, 2011, pp. 769–778. [Online]. Available: https://doi.org/10.1007/978-3-642-25832-9_78

[16] S. Dublish, V. Nagarajan, and N. P. Topham, "Poise: Balancing thread-level parallelism and memory system performance in gpus using machine learning," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 492–505. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00061

[17] A. Iranfar, W. S. de Souza, M. Zapater, K. Olcoz, S. X. de Souza, and D. Atienza, "A machine learning-based framework for throughput estimation of time-varying applications in multi-core servers," in *27th IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2019, Cuzco, Peru, October 6-9, 2019*. IEEE, 2019, pp. 211–216. [Online]. Available: https://doi.org/10.1109/VLSI-SoC.2019.8920309

[18] A. OpenMP, "Openmp application program interface version 4.0," 2013.

[19] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 329–340.

[20] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010, pp. 105–114.

[21] A. Haj-Ali, N. K. Ahmed, T. L. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "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, 2020, pp. 242–255. [Online]. Available: https://doi.org/10.1145/3368826.3377928

[22] (2021) LLVM Cost Model. [Online]. Available: https://llvm.org/doxygen/CostModel_8cpp.html

[23] Oak Ridge Leadership Computing Facility - Summit supercomputing cluster. [Online]. Available: https://www.olcf.ornl.gov/summit/

[24] Stony Brook University Seawulf Cluster. [Online]. Available: https://it.stonybrook.edu/help/kb/understanding-seawulf

[25] K. Fukushima, "Visual feature extraction by a multilayered network of analog threshold elements," *IEEE Transactions on Systems Science and Cybernetics*, vol. 5, no. 4, pp. 322–333, 1969.

[26] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[28] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," *Cited on*, vol. 14, no. 8, p. 2, 2012.

[29] A. Mishra, A. M. Malik, and B. Chapman, "Data transfer and reuse analysis tool for gpu-offloading using openmp," in *International Workshop on OpenMP*. Springer, 2020, pp. 280–294.

[30] L. Li, H. Finkel, M. Kong, and B. Chapman, "Manage openmp gpu data environment under unified address space," in *International Workshop on OpenMP*. Springer, 2018, pp. 69–81.

[31] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman, "Benchmarking and evaluating unified memory for OpenMP GPU offloading," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2017.