2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-2292-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/HIPC50609.2020.00016

Towards High Performance, Portability, and Productivity: Lightweight Augmented Neural Networks for Performance Prediction

Ajitesh Srivastava*§ Naifeng Zhang*§ Rajgopal Kannan† and Viktor K. Prasanna*
*University of Southern California
{ajiteshs,naifengz,prasanna}@usc.edu
†US Army Research Lab-West
rajgopal.kannan.civ@mail.mil

***Abstract*—Writing high-performance code requires significant expertise in the programming language, compiler optimizations, and hardware knowledge. This often leads to poor productivity and portability and is inconvenient for a non-programmer domain-specialist such as a Physicist. More desirable is a high-level language where the domain-specialist simply specifies the workload in terms of high-level operations (e.g., matrix-multiply(A, B)), and the compiler identifies the best implementation fully utilizing the heterogeneous platform. For creating a compiler that supports productivity, portability, and performance simultaneously, it is crucial to predict the performance of various available implementations (variants) of the dominant operations (kernels) contained in the workload on various hardware to decide (a) which variant should be chosen for each kernel in the workload, and (b) on which hardware resource the variant should run. To enable the performance prediction, we propose lightweight augmented neural networks for arbitrary combinations of kernel-variant-hardware. A key innovation is utilizing the mathematical complexity of the kernels as a feature to achieve higher accuracy. These models are compact to reduce training time and allow fast inference during compile-time and run-time. Using models with less than 75 parameters, and only 250 training data instances, we are able to obtain accurate performance predictions, significantly outperforming traditional feed-forward neural networks on 48 kernel-variant-hardware combinations. We further demonstrate that our variant-selection approach can be used in Halide implementations to obtain up to 1.7x speedup over Halide auto-scheduler.**

***Index Terms*—Lightweight augmented neural networks, Performance prediction, Productivity, Portability, Compiler, Heterogeneous platforms**

I. Introduction

With various heterogeneous technologies emerging today, there have been unprecedented opportunities for accelerating applications. Application-specific integrated circuits (ASICs) [1] provide highly specialized implementations but require expertise in implementation and are specialized for one application. On the other hand, CPUs, GPUs, and FPGAs provide more flexibility and are easier to program, but are much slower compared to ASICs. Providing the flexibility in applications and ease of implementation while reaching the speedup offered by ASICs has been the focus of many recent works [2], [3]. However, even writing a CPU/GPU code to get the most out of available hardware requires programming expertise, hardware knowledge, and time. Further, that optimized code may not be "portable", i.e., working well on a different platform. Finally, a domain-specialist such as a physicist is expected to know the operations involved in their workload, but not the details of their highly-optimized implementations. This is important for "productivity", i.e., implementing the desired workflow with few lines of code, not worrying about the code optimizations.

With the objective of achieving high performance, portability, and productivity, we are building a compiler that executes a high-level domain-specific language on heterogeneous platforms aligned with recent DARPA projects [4]. The user will write a high-level code that can be broken down into high-level operations (matrix multiplication, convolution, etc.) which we call kernels. The user only specifies the operation with the inputs such as `matrix-multiply(A, B)` without worrying about the optimized implementation of the actual multiplication, thus enabling high **productivity**. It is the compiler's job to automatically identify how to best execute this code by distributing the kernels among the available hardware configurations on the platform.

In order to identify a high **performance** execution plan, the compiler should be able to predict the performance of a kernel on various hardware resources. This enables the following decisions: *(i) Variant-Selection:* A compiler may have several variants in its library implementing the kernel on the same hardware with potentially different performances, e.g., Boost library vs Eigen library for matrix multiplication. The variation may also come from setting certain parameters in the implementation that affect the runtimes, such as compilation flags and other tunable parameters of the implementation. Given the input, which variant should be selected? *(ii) Mapping to hardware:* The workload is a collection of possibly interdependent kernels. Each kernel can be mapped to various available hardware resources (CPUs, GPUs, etc.). For each kernel-hardware pair, there may be a different kernel variant that is optimal. Having accurate kernel performance

§Equal contribution

978-1-6654-2292-5/20/$31.00 ©2020 IEEE
DOI 10.1109/HiPC50609.2020.00016

models is crucial for these decisions. We acknowledge that our approach to designing this compiler is not suited for compiling arbitrary low-level code as we rely on already available implementations of certain kernels. However, the kernels chosen in the paper dominate the runtime of many workflows including machine learning. In fact, our chosen kernels cover >80% of the workflows [5] in the DARPA SDH program [4], [6]. We emphasize that predicting the execution time is more useful than simply knowing the better variant or hardware resource for individual kernels. For instance, suppose we want to execute two matrix multiplications that do not have any data dependencies on a platform containing a CPU and a GPU. The first one involves matrices of size 100 and the second of size 10000. While the first multiplication alone may be faster on GPU, it should still be scheduled on the CPU so that the GPU is available for the second which is the larger multiplication.

To enable **portability**, the compiler must support learning performance models of execution times $T(K_i, H_j)$ on arbitrary platforms, where K_i is an arbitrary kernel implemented on an arbitrary hardware H_j. We do not assume any access to hardware profilers or details of the kernel implementation. The kernel implementations on various hardware are treated as black-boxes and we can only manipulate the inputs to the implementations. This makes our approach easily extensible when a new implementation of a kernel is added to the library. These performance models can be trained during compiler installation by generating benchmark datasets for each kernel (along with its variants) on the available hardware. To make this feasible, the models must be lightweight so that they can learn quickly with small training data without overfitting. Once the models are trained, the compiler will be ready for scheduling kernels at compile-time. The prediction may also be needed at runtime since the exact input to the kernel may not be known at compile-time, and therefore, the mapping decisions (which variant to select and where to run) will have to be made dynamically at runtime. Making the models compact is necessary to ensure that they do not constitute a significant portion of the runtime. We build performance models for various ubiquitous kernels [4] found in common workflows including (i) Matrix-Matrix Multiplication, (ii) Matrix-Vector Multiplication, (iii) Matrix Convolution, (iv) Max-Pooling, (v) Blur filter, and (vi) FFT. We propose a novel approach called Augmented Neural Network (**NN+C**) which is extremely lightweight and utilizes the time complexity function to perform execution time prediction.

Key Contributions: Our key contributions are as follows.

- We propose novel lightweight neural network models for kernel performance prediction on CPUs and GPUs.
- We demonstrate that the lightweight models are portable to more than 48 kernel-variant-hardware combinations. Results from 48 combinations have been discussed, which include 4 linear algebra kernels, each of which has 2 variants on each of 3 CPUs and 2 variants on each of 2 GPUs. In addition, we also consider 8 Halide [7] implementations covering Blur filter and FFT kernels on various CPUs and GPUs. To the best of our knowledge, no existing work has demonstrated one approach that is as portable as ours working for a variety of implementations (C++ Eigen, C++ Boost, CUDA, and Halide) on various CPUs and GPUs.
- We demonstrate that our models achieve high accuracy even with a small training set in a short amount of training time outperforming traditional feed-forward networks for all 48 kernel-variant-hardware combinations.
- We demonstrate that our performance models can be used to identify the best implementation of a kernel where thousands of variants exist with significantly different runtimes. Specifically, for Halide implementation of Blur filter, our approach results in up to $1.7\times$ speed up over Halide auto-scheduler.

II. Related Work

Most existing works focus on predicting the performance of the whole specific workload. Huang et. al. [8] use sparse polynomial regression to predict the execution time of arbitrary programs. In [9], a neural network is used to predict the execution time of a workload. On the other hand, [10] proposes feature selection from workloads to identify similar applications for which the runtimes are known and then predicting the runtime for the given application using mean or linear regression. These approaches are limited to one or similar applications and will require retraining for every application, and thus are not scalable. Further, it is not clear what type of workloads will result in good predictions and whether a similar approach can be ported to other hardware. Instead, we perform predictions at coarse-level building blocks of a program on various hardware. If a compiler can predict performance at coarse-level operations (kernels such as matrix multiplication) on available hardware, it can make mapping decisions accordingly. For this, we consider four kernels that are dominant in many other workloads. Therefore, instead of being tied to a particular workflow, our approach applies to many, such as the entire class of deep learning workloads.

Other existing works [11], [12] rely on the instruction set architecture or hardware-specific metrics, which can potentially be used to predict kernel (instead of workload) performance. However, this would require explicit knowledge of the hardware and corresponding profilers, and thus will reduce portability. Our approach enables a black-box treatment of the kernels and allows prediction without knowing the specific architecture or implementation details. Table I summarizes the works closest to ours. Although we do not compare our approach against the above-mentioned works quantitatively, as they are for different objectives, we do show comparison against their chosen machine learning models (neural networks and linear regression) and that our lightweight augmented neural networks achieve superior accuracy. Finally, our work is different from [13] as they focus on performance prediction of hardware using hardware profiling instead of the performance of dominant operations.

Table I: Distinction from related works

Approach	Workload Coverage	Portability
Workload-specific [8]–[10]	Low	N/A
ISA/Hardware specific [11], [12]	High	Low/Medium
Our work	Medium	High

III. Proposed Approach

Problem Definition: For each operation on an arbitrary platform with arbitrary implementations, given corresponding inputs, find a lightweight model that accurately predicts the execution time using a small amount of training time.

To solve this problem, we propose the Augmented Neural Network (NN+C). The key idea of NN+C is utilizing known mathematical function $f(K, H)$ as an extra input to NN. For example, in Matrix-Matrix Multiplication ($A_{m,n} \times B_{n,k}$), besides using basic features such as matrix dimensions, matrix density as inputs, we calculate the number of total operations during Matrix-Matrix Multiplication. That is, $f(K, H) = m \times n \times k$. $f(K, H)$ for Matrix-Vector Multiplication, Matrix Convolution, and Max-Pooling is also calculated similarly. The lightweight aspect enables fast decision making during compile-time as well as run-time. These augmented neural networks provide the flexibility to incorporate any tunable parameter available for the kernel and the hardware.

A. Neural Network Structure

The structure of NN+C is shown in Figure 1. Inputs to the neural network are

1) known mathematical function $f(K, H)$
2) kernel parameters K_i, such as input matrix dimension and matrix density
3) hardware/code-optimization parameters H_j, for example, how many threads are used in the multi-threaded implementation and other controllable features that may affect the runtime such as compilation flags

Our augmented neural network contains at most two hidden layers. The output layer has one node, which is the predicted execution time. The number of nodes in hidden layers varies given different kernels and different inputs, resulting in different models for each kernel. Further, models for a given kernel differ for CPU and GPU due to different inputs: in CPU we use multi-threading and take the number of threads as input. Thus, for example, for four kernels mentioned above and two hardware configurations, this results in eight different neural network structures. However, the structure of the models remains the same irrespective of the implementation of the kernel (e.g., different software libraries), and the type of CPU or the type of GPU (e.g., Intel or AMD). In this case, only the weights in the neural network that are learned during training will change.

B. Model Inputs

a) Matrix-Matrix Multiplication($A_{m,n} \times B_{n,k}$): Inputs are the dimensions of the matrices m, n, and k, densities of matrix A ($d_1 = \frac{\text{number of non-zero entries}}{m \times n}$) and of matrix B (d_2), and the number of threads we utilize during multi-threading on CPU, N_{thd}, which is an extra input for operations on CPU and not present for GPU. We augment the neural network with $c = f(K, H)$, which is approximately the total number of operations in the kernels. In this case, $c = m \times n \times k$.

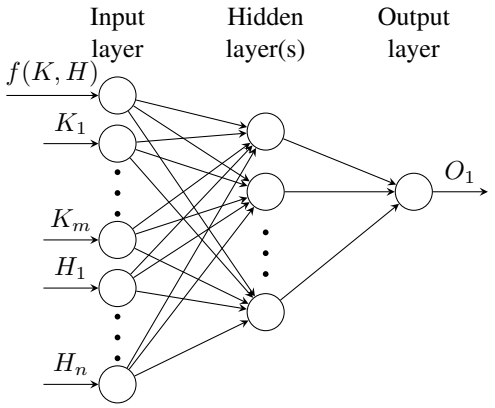


Figure 1: Structure of Augmented Neural Network (NN+C)

b) Matrix-Vector Multiplication ($A_{m,n} \times B_{n,1}$): Inputs are m, n, d, c, N_{thd} as defined above. m and n are dimensions of matrix A. d is the density of matrix A and the density of vector B is set as 1. c is the number of operations, $c = m \times n$. N_{thd} is the number of threads.

c) Matrix Convolution($A_{m,n} * B_{r,r}$): Inputs are m, n, d, c, N_{thd} as defined above, and r is the dimension of square matrix B. d is the density of matrix A and the density of square matrix B is set as 1. The number of operations is given by $c = (m - r + 1) \times (n - r + 1) \times r^2$. N_{thd} is the number of threads.

d) Max-Pooling ($A_{m,n} * B_{s,s}$): Inputs are m, n, d, c, N_{thd} as defined above, and s is the dimension of square matrix B. d is the density of matrix A and the density of square matrix B is set as 1. The number of operations is given by $c = \lceil \frac{n}{s} \rceil \times \lceil \frac{m}{s} \rceil \times s^2$. N_{thd} is the number of threads.

IV. Experiments

A. Platforms and Optimizations

To demonstrate portability of our models we conducted our experiments on five platforms: Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (**Xeon**), Intel(R) Core i7-8750H CPU @ 2.20GHz (**I7**), Intel(R) Core i5-7360U CPU @ 2.30GHz (**I5**), NVIDIA Tesla K40c (**Tesla**) and NVIDIA Quadro K420 (**Quadro**).

To perform the kernel operations on CPU, we used the Eigen library and the Boost library in C++. Eigen/Dense, Eigen/Sparse, uBLAS/matrix, and uBLAS/matrix_sparse are used to optimize dense and sparse matrix in each kernel. Multi-threading was also used in Eigen to vary the number of threads. However, it is difficult to vary the number of threads without heavily changing the code structure in the Boost library. Owing to our black-box approach, we used a single thread in the Boost library. Among our platforms, Xeon

has 16 cores, 32 threads; I7 has 12 cores, 24 threads; and I5 has 2 cores, 4 threads. For all operations on GPU, we used two implementations of CUDA to optimize, one through global memory and one through shared memory. This results in 10 implementations of each kernel: 2 variants on each of 3 CPUs and 2 variants on each of 2 GPUs. We published our code for reproduciblility[1].

B. Datasets

We measured the performance of four kernels on each platform: Matrix-Matrix Multiplication (**MM**), Matrix-Vector Multiplication (**MV**), Matrix Convolution (**MC**) and Max-Pooling (**MP**). Evaluations on Halide kernels (Blur filter and FFT) are discussed in Section V in the context of selecting the best variant for a given kernel. Discounting the Halide kernels, for each kernel-variant-hardware combination (there are 40 such combinations), we generated 500 instances of data, where 250 instances were used to train the model and 250 instances to test. Each data instance was generated randomly with ranges of parameters as described in Table II. While the experiments may be conducted with a different set of ranges, we chose these ranges as they are common sizes for deep learning workflows. Since we use multi-threading on CPU, all operations on CPU take an extra input N_{thd}, which is randomly generated between 1 to the maximum threads supported by the given platforms.

Table II: Parameters for data generation

Matrix-Matrix Multiplication	$m, n, k \in \{1, 2, 3, \ldots, 1024\}$ $d_1 \in \{1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^{\log_2 m \times n}}\}$ $d_2 \in \{1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^{\log_2 n \times k}}\}$
Matrix-Vector Multiplication	$m, n \in \{1, 2, 3, \ldots, 1024\}$ $d \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots, \frac{1}{2^{\log_2 m \times n}}\}$
Matrix Convolution	$r \in \{3, 5, 7\}$ $m, n \in \{r, r+1, r+2, \ldots, 1024\}$ $d \in \{1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^{\log_2 m \times n}}\}$
Max-Pooling	$r \in \{2, 3, 4, 5\}$ $s \in \{1, 2\}$ $m, n \in \{r, r+1, r+2, \ldots, 1024\}$ $d \in \{1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^{\log_2 m \times n}}\}$

C. Models

Our augmented neural networks are built under the TensorFlow framework. Each model is kept under 75 parameters to maintain lightweight and a short training time. All models have at most 3 dense layers and use ReLU as the activation function. We use Adam as the optimizer [14], with learning rate varying between 0.01, 0.0001, and 0.0001. The loss function is chosen to be mean squared error. Each epoch included training with a full batch. The number of parameters of each model as well as its average training time is shown in Table III.

[1]https://github.com/Naifeng/Augmented-Neural-Network

Table III: Number of parameters and average training times

	MM	MV	MC	MP
CPU	64, 19s	50, 18s	73, 6s	73, 6s
GPU	41, 19s	73, 6s	50, 8s	73, 7s

D. Baselines

We compare our method against four baselines: (1) Neural Network (**NN**). NN has the same implementation as NN+C except that NN does not take the mathematical complexity as an extra input. (2) Constant (**C**). In C, we only take the mathematical complexity as input and predict execution time using linear regression. (3) Augmented Linear Regression (**LR+C**). We take the same inputs as NN+C but use linear regression in LR+C. (4) Augmented Non-Linear Regression (**NLR+C**). In NLR+C we take the same inputs as NN+C but use the random forest regression [15]. Random forest based regression has been demonstrated to be competitive in performance prediction [16], [17].

E. Evaluation Metrics

We use mean absolute percentage error (MAPE) to evaluate the predictions $\{\hat{t}_1, \hat{t}_1, \ldots, \hat{t}_N\}$ obtained by the baselines and our models w.r.t. the ground truth $\{t_1, t_2, \ldots, t_N\}$:

$$MAPE = \frac{100}{N} \sum_i \frac{|t_i - \hat{t}_i|}{t_i}. \quad (1)$$

MAPE was used in many existing works on performance prediction [8], [10]–[12]. By the definition of MAPE, a small misprediction ($|t_i - \hat{t}_i|$) might lead to a exceptionally high MAPE (up to 5000%) if the true runtime t_i is minute. Those extreme MAPE values skew the average despite most of the predictions being accurate. Thus, we introduce a threshold at the 30% of the testing data, ranking from the lowest runtime to the highest runtime. Overall, the average runtime of testing data below the threshold is 13% of the average runtime of all the testing data, but these low runtime data instances contribute approximately 80% of the overall MAPE. For example, when analyzing NN+C's performance on MC on GPU, MAPE given by the data instances below the threshold is 128% and the MAPE given by the data instances above the threshold is 15%, whereas the overall MAPE is 49%. Therefore, in reporting MAPE, we drop 30% of testing data with the lowest runtime for a more precise assessment of models' performance.

V. Demonstration of Variant-Selection

As a crucial application of our performance prediction approach, we demonstrate that it can be used to pick the best variant for a given kernel, i.e., picking the best available code among several options. Possible scenarios include choosing between a CPU and a GPU implementation and identifying compilation flags that will be best suited for the kernel. To show the variant selection capability of our approach, we choose a scenario where the number of variants can be extremely high. Further, we choose two kernels different than the four discussed thus far to show the generalizability of our approach.

We consider the Blur filter (**Blur**) kernel and Fast Fourier transform (**FFT**) kernel implemented in Halide [7]. A Halide code decouples the functional program from its execution "schedule" that determines various aspects of the execution such as the ordering of the loops, degree of unrolling loops, and vectorization strategy. The schedule description can be considered as a combination of shape (feature space) and parameters. For instance,

```
blur_y.tile(x, y, xi, yi, 128, 256)
```

defines two dimensions of the shape and the parameters 128 and 256 are the tunable parameters along these dimensions. Changing the schedule does not affect the output of the code, but it may significantly affect the runtime. Therefore, each schedule generates a variant of the same kernel, and our task is to identify the best variant to use.

A. Model Inputs

We train our compact augmented neural networks with inputs representing the schedule features. This allows us to quickly estimate runtimes of the code with various schedule parameters without actually executing the code. Halide provides an auto-scheduler (Mullapudi2016 [18]) that attempts to identify the best schedule itself. (At the time of writing, the newest Halide auto-scheduler (Adams2019 [19]) has not been included in the stable release [20]) We run the auto-scheduler to identify the shape/feature space and ignore the suggested parameters. Within this feature space we generate candidate schedules $S = \{s_1, s_2, s_3, \ldots, s_N\}$, where each s_i is a vector of parameters, and find $s = \arg\min_i P(s_i)$, where $P(s_i)$ represents the predicted runtime given by schedule s_i. For kernels that Halide auto-scheduler are not applicable to, we identify the feature space based on the provided manually written implementation.

Input data dimensions n and augmented constant are also fed into the neural network. We augment n^2 for Blur and $n \log_2 n$ for FFT to corresponding variant-selection models given the complexity of Halide implementation of both kernels [21].

B. Platforms and Optimizations

We conducted variant-selection experiments on five platforms: Xeon, I7, I5, Tesla, and Quadro. We used Halide to implement the kernel operations. More experiment settings of variant-selection can be found within our published code[2].

C. Datasets

We evaluated two kernels: Blur and FFT. The performance of Blur is measured on five platforms. Given that there is no existing GPU schedule of FFT provided by Halide, we only conducted experiments of FFT on three CPU platforms. To demonstrate that our models are able to identify the best implementation among numerous existing variants, we generated thousands of data instances and restricted the training set to consist of 250 instances to maintain portability.

[2]https://github.com/Naifeng/Variant-Selection

The following is a piece of code from the implementation of Halide Blur on CPU. Each of s_1, s_2, s_3 and s_4 resides in a `.split()` function and serves as a split factor. The inner loop runs from zero to the split factor and the outer loop runs from zero to the extent required by the first argument divided by the split factor [7]. Thus, a combination of $\{s_1, s_2, s_3, s_4\}$ defines a candidate schedule and different schedules have significantly different runtimes. We varied each parameter extensively to generate a candidate set. The schedule given by Halide auto-scheduler is $\{8, 256, 128, 8\}$.

```
{
    Var x = blur_x.args()[0];
    blur_x
        .compute_at(blur_y, x_o)
        .split(x, x_vo, x_vi, s1)
        .vectorize(x_vi);
}
{
    Var x = blur_y.args()[0];
    Var y = blur_y.args()[1];
    blur_y
        .compute_root()
        .split(x, x_o, x_i, s2)
        .split(y, y_o, y_i, s3)
        .reorder(x_i, y_i, x_o, y_o)
        .split(x_i, x_i_vo, x_i_vi, s4)
        .vectorize(x_i_vi)
        .parallel(y_o)
        .parallel(x_o);
}
```

According to Halide implementation rules and current supports (e.g., Halide only supports limited input dimensions for FFT), we varied parameters as described in Table IV. For Blur and FFT on CPU, we generated 1000 data instances for each input data dimension, resulting in 6000 instances and 4000 instances, respectively. For Blur on GPU, we exhaustively generated all possible combinations, that is, 1176 instances.

Table IV: Parameters for data generation

Blur (CPU)	$n \in \{2^{10}, 2^{11}, 2^{12}, \ldots, 2^{15}\}$ $s_1, s_2 \in \{2, 4, 8, \ldots, 1024\}$ $s_3 \in \{2, 4, 8, \ldots, s_2\}$ $s_4 \in \{2, 4, 8, \ldots, s_3\}$
Blur (GPU)	$n \in \{2^{10}, 2^{11}, 2^{12}, \ldots, 2^{15}\}$ $s_1 \in \{2, 4, 8, 16\}$ $s_2, s_3 \in \{1, 2, 4, \ldots, 64\}$
FFT (CPU)	$n \in \{2^4, 2^5, 2^6, 2^7\}$ $s_1 \in \{2, 4, 8, \ldots, 2^{n-1}\}$ $s_2, s_3, s_4, s_5, s_6 \in \{2, 4, 8, \ldots, 2^n\}$

D. Models

Augmented neural networks used for variant-selection is the same as models described in Section IV-C except that all models used for variant-selection have exact 3 dense layers. The number of parameters of each model as well as its average training time is shown in Table V.

Table V: Number of parameters and average training times

	Blur	FFT
CPU	71, 18s	67, 12s
GPU	66, 7s	N/A

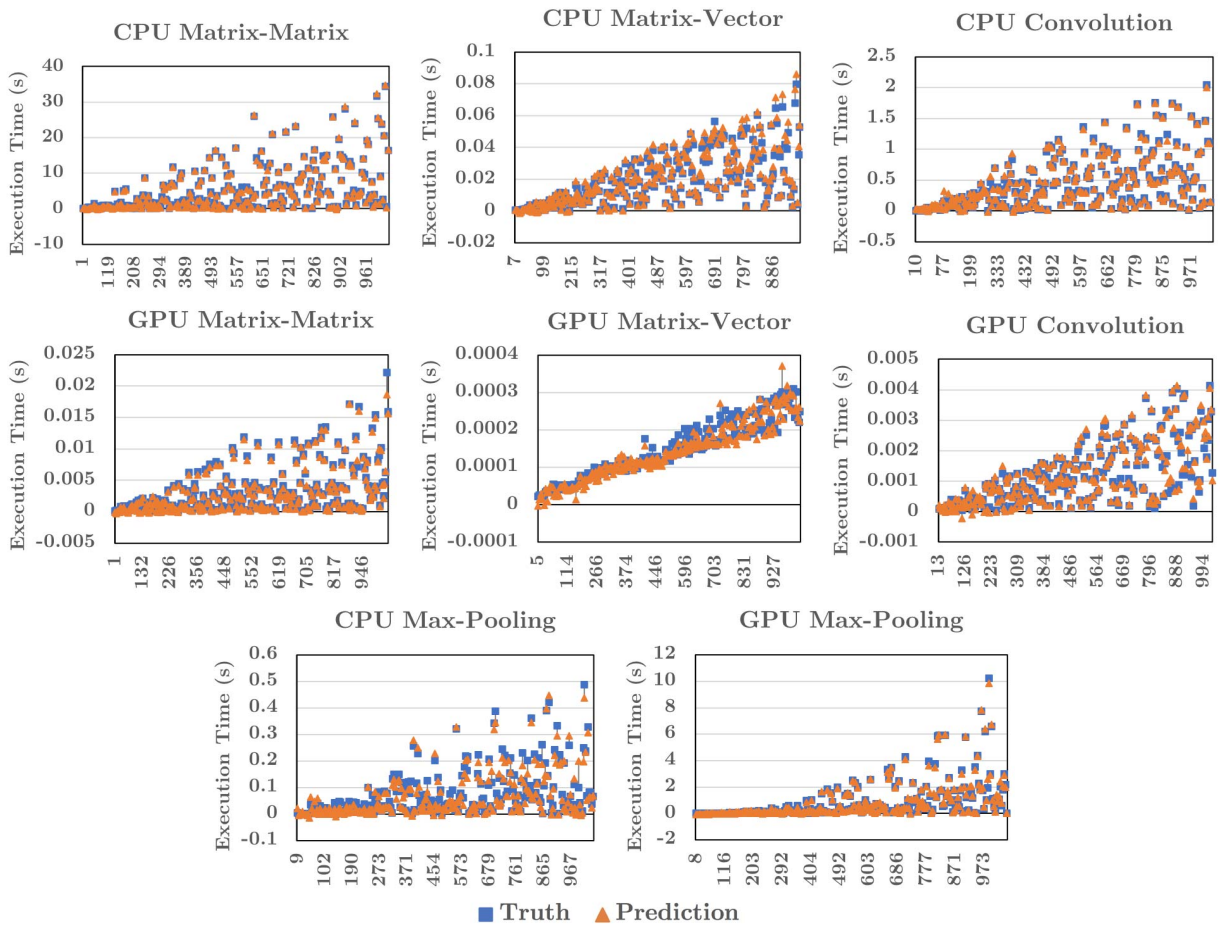


Figure 2: Performance predictions of four kernels using NN+C

E. Baselines

We compare our method against four baselines identical to baselines described in Section IV-D: (1) NN, (2) C, (3) LR+C, and (4) NLR+C. In addition, for Blur on CPU, we compare our variant-selection approach with the Halide auto-scheduler to show the overall improvement. For Blur on GPU, due to the fact that Halide does not have a stable auto-scheduler to generate a GPU schedule, we compare our variant-selection results with the average runtime among the runtime of all candidate schedules. Similarly, since current Halide auto-scheduler is not capable of scheduling a complicated pipeline such as FFT, we compare our results with the average runtime as well as the minimum runtime among the runtime of all candidate schedules for FFT on CPU.

F. Evaluation Metrics

We use MAPE and Spearman's rank correlation coefficient (ρ) to evaluate the predictions $\{\hat{t}_1, \hat{t}_1, \ldots, \hat{t}_N\}$ obtained by the baselines and our models w.r.t. the ground truth $\{t_1, t_2, \ldots, t_N\}$. MAPE is defined in Equation (1). ρ is defined as

$$\rho = 1 - \frac{6 \sum_{i=1}^{N} d_i^2}{N\,(N^2 - 1)} \tag{2}$$

where $d_i = |\mathrm{rank}(t_i) - \mathrm{rank}(\hat{t}_i)|$. $\mathrm{rank}(t_i)$ is the rank of t_i among $\{t_1, t_2, \ldots, t_N\}$, ranking from the lowest value to the highest value. $\mathrm{rank}(\hat{t}_i)$ is the rank of $\hat{t}_i$ among $\{\hat{t}_1, \hat{t}_1, \ldots, \hat{t}_N\}$, ranking from the lowest value to the highest value. ρ ranges from -1 to 1. ρ of 1 indicates a perfect positive correlation of two variables' ranks and ρ of -1 indicates a perfect negative correlation of two variables' ranks. The closer ρ is to zero, the weaker the correlation between the ranks.

VI. Results

A. Kernel Performance Prediction

Figure 2 shows a visualization of using NN+C to predict kernel performance on two platforms. We choose the results of I5 and Tesla to represent the results on CPU and GPU, respectively. We pick matrix dimension as x-axis, plotting against execution time in seconds to visualize prediction. A line joining two points in the plot indicates the corresponding prediction and ground truth. Note that very few points have a significant misprediction. Figure 3 shows the prediction percentage error distributions across execution times using NN+C and compares it against using NN. Similarly, we choose the results of I5 and Tesla to represent the results on

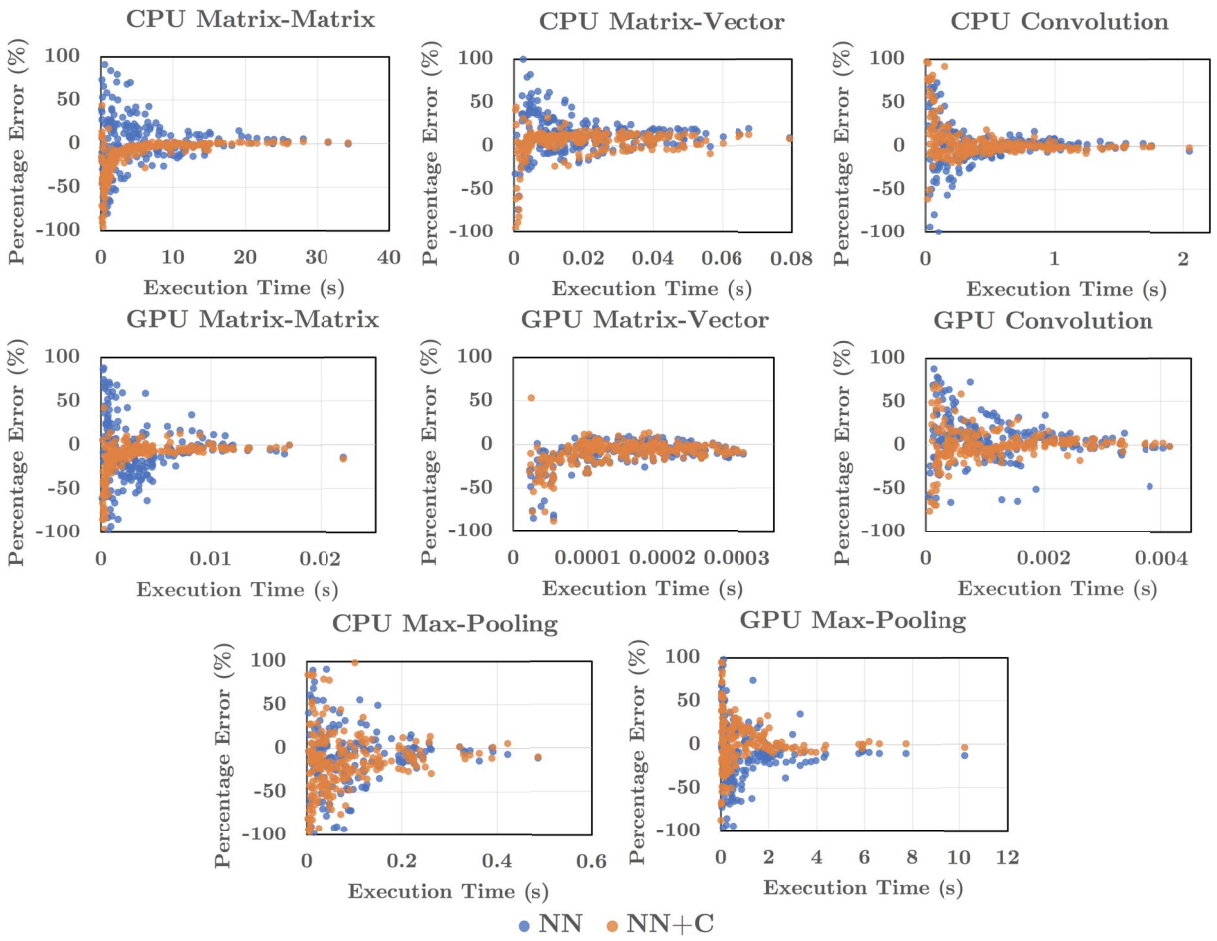


Figure 3: Percentage error distributions of four kernels using NN and NN+C

Table VI: Prediction MAPE of Matrix-Matrix Multiplication

	CPU						GPU			
	Eigen			Boost			$CUDA_{Global\ Memory}$		$CUDA_{Shared\ Memory}$	
	Xeon	I7	I5	Xeon	I7	I5	Tesla	Quadro	Tesla	Quadro
NN+C	**14%**	**23%**	**8%**	**7%**	**27%**	**6%**	**7%**	**5%**	**8%**	**8%**
NN	29%	31%	26%	20%	35%	19%	23%	13%	18%	16%
C	39%	34%	28%	8%	34%	7%	9%	9%	10%	10%
LR+C	44%	31%	33%	8%	34%	7%	8%	8%	**8%**	**8%**
NLR+C	23%	24%	**8%**	9%	33%	7%	10%	10%	18%	19%

Table VII: Prediction MAPE of Matrix-Vector Multiplication

	CPU						GPU			
	Eigen			Boost			$CUDA_{Global\ Memory}$		$CUDA_{Shared\ Memory}$	
	Xeon	I7	I5	Xeon	I7	I5	Tesla	Quadro	Tesla	Quadro
NN+C	**21%**	**21%**	**25%**	**11%**	**8%**	**9%**	**7%**	**7%**	**7%**	**6%**
NN	22%	24%	29%	14%	11%	12%	**7%**	8%	**7%**	9%
C	**21%**	22%	**25%**	12%	**8%**	**9%**	23%	23%	11%	10%
LR+C	**21%**	22%	26%	12%	**8%**	**9%**	**7%**	**7%**	**7%**	**6%**
NLR+C	26%	25%	27%	12%	**8%**	**9%**	29%	28%	21%	22%

CPU and GPU, respectively, and we discard data instance whose absolute percentage error is beyond 100% in plotting. We observe that the error of NN+C has a smaller spread compared to that of NN. Data instances with a low execution time is more likely to yield higher percentage error than data instances with a high execution time. Tables VI, VII, VIII, and IX quantify these results using MAPE.

Table VIII: Prediction MAPE of Matrix Convolution

	CPU						GPU			
	Eigen			Boost			$CUDA_{Global\ Memory}$		$CUDA_{Shared\ Memory}$	
	Xeon	I7	I5	Xeon	I7	I5	Tesla	Quadro	Tesla	Quadro
NN+C	**8%**	**21%**	**4%**	**30%**	**20%**	**13%**	**10%**	**15%**	**17%**	**19%**
NN	9%	22%	7%	50%	30%	30%	16%	**15%**	22%	**19%**
C	27%	40%	22%	48%	44%	40%	30%	30%	42%	42%
LR+C	15%	32%	13%	46%	38%	37%	15%	**15%**	29%	30%
NLR+C	18%	32%	7%	**30%**	32%	24%	17%	17%	21%	21%

Table IX: Prediction MAPE of Max-Pooling

	CPU						GPU			
	Eigen			Boost			$CUDA_{Global\ Memory}$		$CUDA_{Shared\ Memory}$	
	Xeon	I7	I5	Xeon	I7	I5	Tesla	Quadro	Tesla	Quadro
NN+C	**23%**	**13%**	**14%**	**27%**	**12%**	**14%**	**14%**	**8%**	**25%**	**27%**
NN	32%	20%	22%	36%	20%	34%	32%	32%	40%	47%
C	67%	37%	43%	81%	41%	47%	93%	95%	40%	28%
LR+C	50%	26%	27%	63%	31%	33%	75%	77%	40%	28%
NLR+C	25%	**13%**	17%	**27%**	13%	18%	**14%**	14%	31%	29%

For all five kernels using any implementation, NN+C produces the lowest MAPE in predictions. Ranking from the highest accuracy (lowest MAPE) on average to the lowest is (1) NN+C, (2) NLR+C, (3) NN, (4) LR+C, and (5) C. On average, NN+C outperforms traditional NN by a margin of 8% and outperforms the second-best approach NLR+C by 5%. LR+C has a good prediction for kernels on GPU. Performance of C on all platforms is worst among all kernels expect MV. Overall, NN+C predicts more accurately for kernels on GPU than those on CPU, achieving on average a low MAPE of 12% and 16%, respectively.

We report the aggregated average of MAPE for the four kernels and the two hardware classes (CPU, GPU) in Table X. For each kernel, MAPE was aggregated over all hardware and variants. For each hardware class, MAPE was aggregate over all the kernels, variants, and specific devices. We show the comparison of NN+C against traditional NN. NN+C significantly outperforms NN in almost all cases. In fact, for MM, MAPE for NN+C is less than half of that of NN, suggesting that the traditional neural network is far inferior than our augmented neural network for some specific kernels.

Table X: Aggregated MAPE of NN+C vs. NN

	MM	MV	MC	MP	CPU	GPU
NN+C	11%	12%	16%	18%	16%	12%
NN	23%	14%	22%	32%	24%	20%

a) Unconstrained Augmented Neural Networks: To enable fast inference, our models are kept extremely lightweight – less than 75 weights. Also, we only generate 500 data instances for each kernel-variant-hardware combination, out of which 250 are used to train our models. In order to assess how much of the performance is compromised due to these restrictions, we build similar NN+C models with more parameters and generate a larger dataset with 5000 data instances (2500 instances are used to train and 2500 instances to test). Figure 4 illustrates the comparison between lightweight models and unconstrained models in terms of error. Overall, MAPE achieved by lightweight NN+C is 14% and by unconstrained NN+C is 9%. Specifically, on CPU, using unconstrained NN+C, MM, MV, MC, and MP have a decrease on average MAPE of 5%, 2%, 8%, and 12%, respectively. On GPU, using unconstrained NN+C, MM, MV, MC, and MP have a decrease on average MAPE of 1.5%, 1%, 3%, and 2.5%, respectively. However, accuracy comes at the cost of increased model size and the overall time as summarized in Table XI.

Table XI: Preparing time increase and model size increase

	MM	MV	MC	MP
CPU	9.31x, 2.13x	2.30x, 2.12x	11.59x, 2.21x	10.24x, 2.48x
GPU	5.08x, 8.80x	7.07x, 2.34x	4.28x, 2.12x	3.35x, 2.62x

The preparing time (training data generation time plus model training time) on average of lightweight NN+C on CPU is 104s and that of unconstrained NN+C is 1040s, which is 10x of lightweight NN+C. The preparing time on average of lightweight NN+C on GPU is 20s and that of unconstrained NN+C is 97s, 4.85x of lightweight NN+C. With lightweight NN+C, in addition to the preparing time loss, model size is significantly reduced. Model size reduction is most evident in MM on GPU. Comparing to unconstrained NN+C, lightweight NN+C model is 8.80x smaller. The rest of the models are downsized by 2.29x on average.

If the training and inference time is not constrained, then one can use our unconstrained (larger and more accurate) augmented models. However, we envision that lightweight models may be necessary due to the following reasons: (a) at compile-time many kernels need to be evaluated: consider VGG16 inference that requires >1M 2D-Convolutions. At a given layer, there can be >100K 2D-Convolutions, each of which may have different execution times not only due to heterogeneous hardware but also due to different sparsity. This number of convolutions will multiply with the factor of the number of parallel image classification pipelines. (b) Some

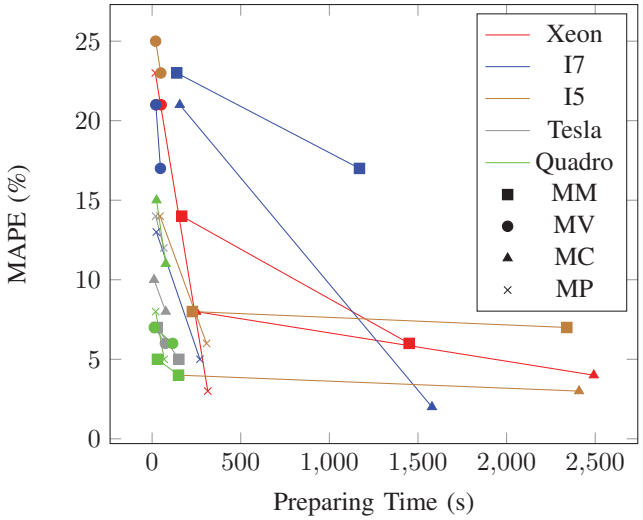


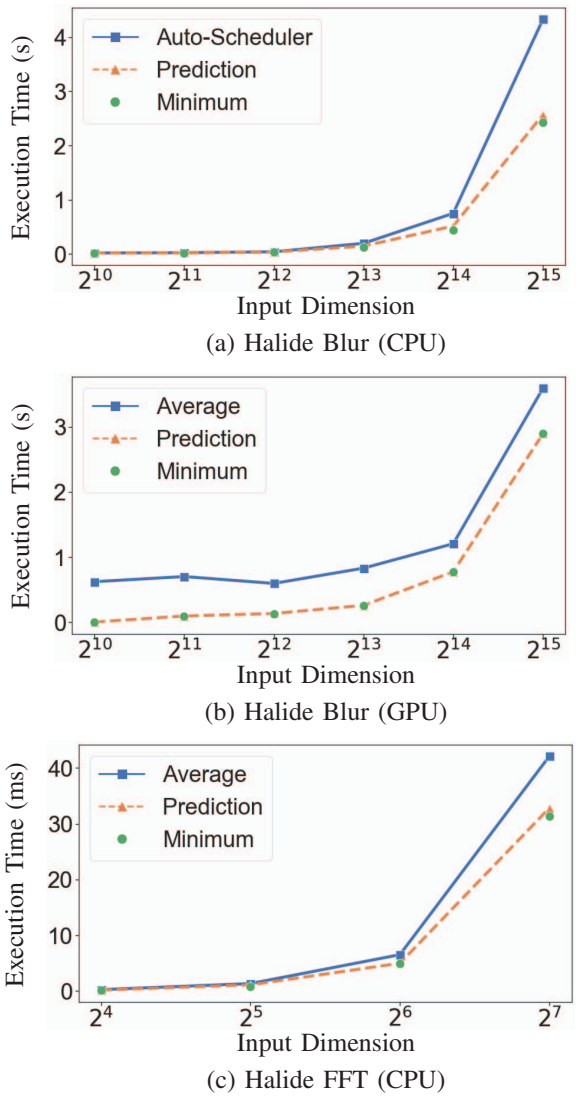
Figure 4: Performance comparison between Lightweight Models and Unconstrained Models

decisions may have to be made at runtime: some kernels may be only instantiated at runtime, which is the only time performance prediction inference has to be performed. In such scenarios, the inference time should be as minimal as possible to avoid an significant impact on the total runtime.

B. Variant-Selection

Figure 5 shows the comparison of execution times for varying input sizes of two kernels. Our predicted best schedule (Prediction) produces a runtime close to the true best schedule (Minimum) within the candidate set in all cases. Further, Figure 5(a) shows that using our predictions, we were able to outperform Halide auto-scheduler, getting up to $1.7\times$ speedup in kernel Blur on CPU. As for Blur on GPU, we obtained up to $223.5\times$ speedup compared to a randomly selected schedule on a small input size (2^{10}), see Figure 5(b), and among all input sizes, we were able to obtain a speedup of at least $1.24\times$. Shown in Figure 5(c), we obtained up to $1.5\times$ speedup compared to a randomly selected schedule of Halide FFT on CPU.

Note that MAPE varies among different training processes and train-test splits. The MAPE value shown in Table XII is the best (lowest) MAPE obtained by our methods and the baselines on the Halide kernels. The table also shows the Spearman's rank correlation coefficient. Since the main objective is to select the best variant which requires the ability to correctly rank the variants, this is the primary metric of comparison. We observe that our approach NN+C obtains the highest rank correlation in the majority of the cases. NLR+C has a higher rank correlation for Halide Blur while having much worse MAPE. On the other hand, for Halide FFT, NLR+C obtains identical rank coefficients with NN+C and better MAPEs. However, NLR+C has a much higher inference time, and therefore likely to hinder the execution of the actual application if used by the runtime component of a compiler,



(a) Halide Blur (CPU)

(b) Halide Blur (GPU)

(c) Halide FFT (CPU)

Figure 5: Runtime comparison of variants obtained from baseline and our approach

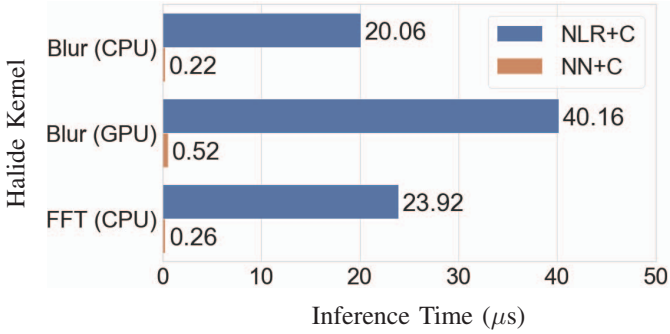


Figure 6: Inference time comparison of NN+C and NLR+C

as we argued in the end of Section VI-A. Figure 6 shows the comparison of inference times between our lightweight

Table XII: Prediction MAPE and Spearman's coefficient

	Halide Blur					Halide FFT		
	CPU			GPU		CPU		
	Xeon	I7	I5	Tesla	Quadro	Xeon	I7	I5
NN+C	**50%**, 0.91	**23%**, **0.95**	**28%**, **0.97**	**8%**, **0.99**	**22%**, **0.97**	8%, **0.99**	14%, **0.97**	3%, **1**
NN	72%, 0.87	25%, 0.94	40%, 0.91	12%, 0.98	29%, 0.94	11%, 0.98	19%, 0.95	17%, 0.95
C	1140%, 0.76	59%, 0.82	167%, 0.93	84%, 0.73	64%, 0.85	66%, 0.86	33%, 0.97	30%, 0.97
LR+C	1687%, 0.66	93%, 0.82	411%, 0.84	106%, 0.89	62%, 0.87	44%, 0.97	32%, 0.92	26%, 0.90
NLR+C	150%, **0.93**	39%, 0.94	43%, **0.97**	10%, **0.99**	23%, **0.97**	**3%**, **0.99**	**11%**, **0.97**	**2%**, **1**

NN+C and NLR+C. It is noteworthy that the inference time of NLR+C is more than $75\times$ of that of our approach.

VII. Conclusion

We have proposed a novel lightweight augmented neural network (NN+C), to predict kernel performance on CPUs and GPUs. Our approach is designed in support of creating compilers with high productivity, portability, and performance. To show that our models are portable to different platforms with different implementations, we have evaluated our model on several CPUs and GPUs with multiple optimizations, resulting in a total of 48 kernel-variant-hardware combinations. To the best of our knowledge, no existing work has demonstrated one approach that is as portable as ours working for a variety of implementations (C++ Eigen, C++ Boost, CUDA, and Halide) on various CPUs and GPUs. Our models significantly outperformed the baselines including standard neural network. We have shown that our approach can be used to identify the best variants even when the number of variants is extremely high, by demonstrating a $1.7\times$ speedup over Halide auto-scheduler. In future work, we will build prediction models for other popular kernels. These models will be used to perform optimized mapping of kernels in workflows for various heterogeneous platforms.

Acknowledgement

This work is supported by the Defense Advanced Research Projects Agency (DARPA) under BAA number HR0011-20-9-0019 and by the National Science Foundation Award number 1911229. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] M. M. Vai, W. S. Song, and B. M. Tyrrell, "Application-specific integrated circuits," in *High Performance Embedded Computing Handbook* (D. R. Martinez, R. A. Bond, and M. M. Vai, eds.), ch. 9, pp. 191–215, A Systems Perspective, 2008.

[2] I. Kuon and J. Rose, *Quantifying and Exploring the Gap Between FPGAs and ASICs*. Springer Publishing Company, Incorporated, 1st ed., 2009.

[3] B. Zahiri, "Structured asics: opportunities and challenges," in *Proceedings 21st International Conference on Computer Design*, pp. 404–409, Oct 2003.

[4] "Software defined hardware (sdh)." https://www.darpa.mil/program/software-defined-hardware.

[5] H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Design and implementation of knowledge base for runtime management of software defined hardware," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2019.

[6] "Darpa looks to propel parallelism." https://www.hpcwire.com/2019/09/04/darpa-looks-to-propel-parallelism/.

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[8] L. Huang, J. Jia, B. Yu, B. gon Chun, P. Maniatis, and M. Naik, "Predicting execution time of computer programs using sparse polynomial regression," in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 883–891, Curran Associates, Inc., 2010.

[9] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *European Conference on Parallel Processing*, pp. 196–205, Springer, 2005.

[10] W. Smith, I. Foster, and V. Taylor, "Predicting application run times using historical information," in *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 122–142, Springer, 1998.

[11] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," *arXiv preprint arXiv:1808.07412*, 2018.

[12] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37–56, 2017.

[13] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 564–576, IEEE, 2015.

[14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[15] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[16] C. Dahinden and M. Ethz, "An improved random forests approach with application to the performance prediction challenge datasets," *Hands-on Pattern Recognition, Challenges in Machine Learning*, vol. 1, pp. 223–230, 2011.

[17] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

[18] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–11, 2016.

[19] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[20] Halide, "Release halide 10.0.0." https://github.com/halide/Halide/releases/tag/v10.0.0, Sep 2020.

[21] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.