G^{2}PM: Performance Modeling for ACAP Architecture with Dual-Tiered Graph Representation Learning

Tuo Dai, Bizhao Shi, and Guojie Luo
School of Computer Science, Peking University
Center for Energy-efficient Computing and Applications, Peking University
Beijing, China
{daitoto,bshi,gluo}@pku.edu.cn

ABSTRACT

Performance estimation is a crucial component in the optimization processes of accelerator development on the Versal ACAP architecture. However, existing approaches present limitations - they are either too slow to facilitate efficient iterations, or they lack the necessary accuracy due to the specific AIE array architecture and two-level programming model of Versal ACAP. To tackle this challenge, we propose G^{2}PM, a performance modeling technique based on a hierarchical graph representation centered on the AIE array. More specifically, we employ a hierarchical graph neural network to identify features of both kernel programs and dataflow programs, taking into account the hardware and software characteristics of the Versal ACAP architecture. In our evaluations, our method demonstrates significant improvements, achieving a mean error rate of less than 1.6% and providing a speed-up factor of 4165× compared to the simulation-based method.

KEYWORDS

Modeling, Graph Neural Network, Versal ACAP

ACM Reference Format:
Tuo Dai, Bizhao Shi, and Guojie Luo. 2024. G^{2}PM: Performance Modeling for ACAP Architecture with Dual-Tiered Graph Representation Learning. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655898

1 INTRODUCTION

Recently, the AMD/Xilinx Versal Adaptive Compute Acceleration Platforms (ACAP) [2] have attracted wide attention because of their high-performance heterogeneous computing resources and flexible software programmability. There have already been several application acceleration and programming frameworks [3, 6, 19] targeting the Versal ACAP architectures. With the increasing trend of automated design flows, an ACAP-specialized performance modeling approach plays an essential role in guiding optimization, such as performance tuning and workload assignment.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3655898

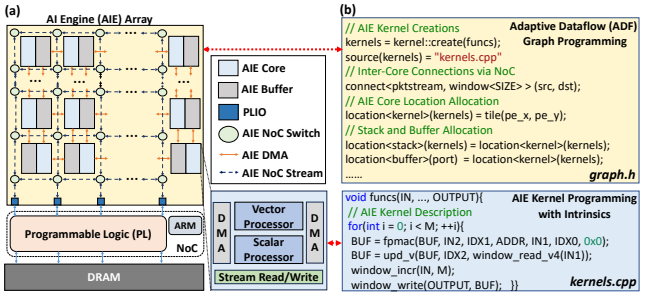

Figure 1: (a) Versal ACAP architecture; (b) Two-level programming model of the AIE array.

A typical Versal ACAP architecture is depicted in Figure 1(a), where the heterogeneous resources encompass ARM cores, Programmable Logic (PL), and an AI Engine (AIE) array. The most distinctive aspect of this architecture – the AIE array, is specifically designed for high-performance AI and Digital Signal Processing (DSP) applications. It organizes hundreds of AIE cores in a mesh-connected array through a network-on-chip (NoC). Each AIE core executes vectorized computations in a very-long-instruction-word (VLIW) [8] style, while the NoC facilitates efficient data transfer between cores. Besides, as illustrated in Figure 1(b), a two-level programming model, consisting of kernel programs and dataflow programs, is proposed for developing operators on the AIE array.

To leverage the exceptional computational capabilities of AIEs, numerous prior studies [3, 6, 19, 20] have proposed various development methods. These methods involve design space exploration and compilation optimization processes, which require efficient and accurate performance estimation. The works [19, 20] proposed an analytical performance model specific to the dataflow designed in their work, but this specialized model does not capture the details in the AIE kernel programs, making it difficult to apply to other designs. The work [13] proposed an analytical model based on the compilation transformation but does not explicitly model the NoC connections. Others [3, 6] utilized simulation-based methods, which can be quite accurate but are also time-consuming and may not be feasible in certain situations. To modeling the AIEs accurately, there are two key features to be aware of when modeling performance for the ACAP architecture: 1) **AIE instruction set architecture (ISA):** The ISA of AIE cores differs from previous processors, which adds to the complexity of modeling. The model should take both intrinsics and VLIW into account to ensure accuracy; 2) **NoC Architecture:** Data transfer speeds can vary significantly among different connection methods on the NoC, leading

to diverse performance estimations for memory-bound applications. The model should incorporate these hardware features of the ACAP.

To address the challenges associated with performance modeling methods, we introduce G^{2}PM, a performance model for the ACAP architecture that utilizes a dual-tiered graph representation. This model is constructed based on the hardware features and two-level programming model of AIEs, using a dual-tiered graph neural network centered on the AIE array. In the AIE core tier, we design a graph convolution network (GCN) to learn the features of the AIE ISA. This network represents the computation and memory assessment within a single AIE core. It takes the control dataflow graph (CDFG) representation of the original AIE kernel program as input and calculates the feature vectors for the cycles spent in one core. In the AIE array tier, we propose a graph attention network (GAT) to model the features of the NoC, representing the data transfer across the entire array. By integrating these two tiers of graph representation, G^{2}PM is able to accurately model both the AIE ISA and the NoC architecture of the Versal ACAP for general applications on AIEs. In the evaluation section, we demonstrate the accuracy and efficiency of G^{2}PM by comparing it with other performance models across multiple applications and various scales of designs. Additionally, we highlight the ability of our model to integrate with mature development frameworks.

We summarize our contributions as follows:

- We propose G^{2}PM, a dual-tiered graph representation-based performance model for the ACAP architecture, and introduce a two-stage training method to effectively train this model.
- We evaluate G^{2}PM using a diverse dataset, which achieves a mean error rate of less than 1.6% and provides a speed-up factor of 4165× when compared to traditional simulation-based methods.
- We illustrate the potential use cases of G^{2}PM in the development flows for ACAP, further demonstrating the practicality and applicability of our proposed model.

2 BACKGROUND

2.1 Versal ACAP Architecture

To construct a performance model for the Versal ACAP architecture, it is essential to first understand its hardware features and programming model of the AIE array.

2.1.1 Hardware Features. Figure 1(a) depicts the detailed architecture of the VC1902 [1], an evaluation kit for the Versal ACAP, which comprises the CPU, Programmable Logic (PL), and AIE components. The AIE array in the VCK5000 consists of an 8×50 grid of AIE cores, with each core possessing the capability to execute 128 multiplication-accumulations (MACs) of int8 data type per cycle at a frequency of 1GHz or higher. In addition, the AIE cores operate in a Single-Instruction-Multiple-Data (SIMD) mode, following a VLIW pattern. Furthermore, the AIE cores are equipped with capabilities for both memory access and data streaming, which are used for data load and store operations. These functionalities introduce a unique aspect to performance modeling, differentiating the AIE cores from previous processor architectures. Figure 1 also shows the NoC architecture, which encompasses various data transfer methods. The most efficient method involves shared memory between neighboring cores, enabling data transfer rates of up to ten terabytes per second. Meanwhile, this rate decreases to approximately one terabyte per second when data transfer occurs between cores that are more than one unit apart. As a result, these distinctive features of the NoC for data transfer introduce a new challenge in performance modeling for the ACAP architecture.

2.1.2 Model. The programming model [2] designed for the AIE array by AMD/Xilinx comprises two components: data-flow graph programs for the AIE array and kernel programs for individual AIE cores, as illustrated in Figure 1(b). On one hand, the graph for the AIE array illustrates the dataflow among the AIE cores and between the AIE and I/O ports. This information, combined with the unique features of the NoC architecture, determines the time cost of data transfer within the architecture. On the other hand, the kernel programs represent instructions as intrinsics, accompanied by certain pragmas serving as hints for loop optimization. Different combinations and sequences of intrinsics can result in varying cycle costs within a single core. Consequently, an accurate performance model must take into account the AIE ISA.

2.2 Performance Modeling

While the most accurate method for performance estimation is simulation-based, it can be time-consuming. AMD/Xilinx provides an AIE simulator [2] that can generate detailed performance metrics for AIEs, including cycle counts on AIE cores and data transfer information on NoC. However, the AIE simulator can take a significant amount of time to generate these estimations for large-scale designs - approximately 1.2 hours for a design with 100 cores, for instance. Moreover, the compilation time for AIE simulator is also lengthy, making it impractical for an iterative development flow.

Therefore, to model the performance of designs on the ACAP architecture more quickly, analytical methods [13, 19] have been proposed. These models make certain assumptions, such as a decay rate for AIE cores, while overlooking the detailed kernel programs. Moreover, they only take into account data transfer speed between neighboring cores for NoC architecture, while ignoring the different speed between non-neighboring cores. Thus, these models are limited in their applicability. They are effective only for designs that follow specific patterns in AIE cores and AIE arrays, and cannot be applied to other scenarios or design configurations. This limitation underscores the need for more flexible and comprehensive modeling approaches.

In an effort to construct an accurate and efficient performance model, learning-based methods have been explored. Representation learning techniques, as demonstrated in studies such as IR2Vec [18] and CodeBERT [7], show immense potential for modeling programs across diverse processor architectures. However, these methods overlook the dataflow interconnections at the graph level. On the other hand, graph-based learning techniques [9, 10, 16, 17] for data flow graphs (DFG) have demonstrated the ability to capture information in complex graphs. Therefore, applying graph representation learning to both AIE cores and NoC dataflow presents a promising avenue for modeling ACAP architectures.

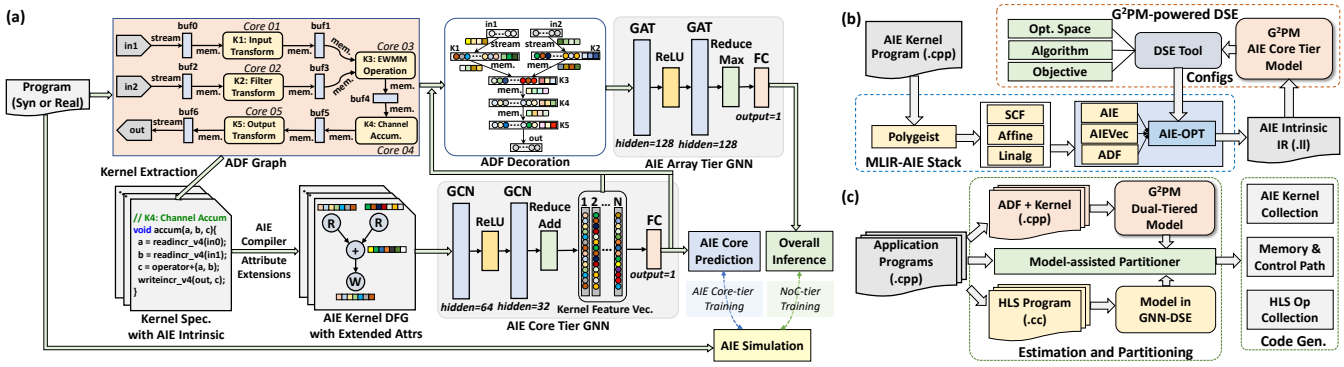

Figure 2: G²PM and downstream use cases: (a) architecture and workflow; (b) G²PM-enhanced compilation optimization with MLIR-AIE; (c) G²PM-assisted workload partitioning.

3 G²PM PERFORMANCE MODEL

3.1 Overview

Building on the analysis presented in the background section, we propose G²PM, a performance model with dual-tiered graph representation, as illustrated in Figure 2(a). A general design for the AIE array in the ACAP takes the form of an ADF graph, along with kernel programs for each AIE core. The ADF graph describes the dataflow in the AIE array tier, which performs dynamically, while the kernel program in the AIE core tier remains static once it is programmed to the core. These different workflows provide the insight necessary to design the dual-tiered performance model. The following subsections will provide a detailed explanation of our model's design and the training process.

3.2 AIE Core-Tier Model

The AIE core-tier model is specifically designed with an understanding of the AIE ISA. For AIE kernel programs, we employ a subnetwork with a graph representation. As illustrated in Figure 2 (a), we utilize graph convolution (GCN) layers to learn the features of individual kernel programs. The control data flow graph (CDFG) of the program and the features of its nodes serve as the input for the AIE core-tier model, with details outlined in Table 1. When designing the node features, we consider both the data type and intrinsic type to identify the capabilities of the AIE vector processor. Concurrently, we differentiate between types of load and store operations due to their distinct performance characteristics within the AIE core, particularly in terms of memory access and data streaming.

The model's backbone structure leverages two GCN layers to propagate features along the dataflow dependence. Each node in the DFG represents a specific operation or instruction in the program, with edges indicating data dependencies between these operations. The GCN layers propagate information across this graph, thereby capturing the complex patterns of data dependencies within the program. In our model, the features propagated by the GCN layers denote the running cycles of the program's operations. This modeling approach enables our model to effectively capture the temporal dynamics of the program's execution. The dual GCN layers in our model allow it to discern both local and more global patterns in the data dependencies. The first GCN layer primarily captures local dependencies between directly connected nodes, while the second GCN layer can capture more global dependencies through multi-step information propagation in the graph. Regarding the aggregation function, we chose the sum aggregation function. This decision is informed by the VLIW processor architecture, which enables simultaneous execution of multiple instructions. Consequently, the sum of individual node features serves as an appropriate representation. Thus, the output of AIE core-tier model, $\boldsymbol{y}^n$, for the n-th core can be represented as:

$$\boldsymbol{y}^n = \sum_{i=1}^{|V_n|} (GCN(ReLU(GCN(\boldsymbol{V}_n^i, \boldsymbol{E}_n, \boldsymbol{W}_1)), \boldsymbol{E}_n, \boldsymbol{W}_2)) \quad (1)$$

where $\boldsymbol{V}_n^i$, $\boldsymbol{E}_n$, $\boldsymbol{W}_1$, $\boldsymbol{W}_2$ donate the i-th node in the CDFG of the n-th core program, edges in the CDFG, the weights of the first GCN layer, and the weights of the second GCN layer, respectively.

Table 1: Input Attributes for Dual-Tiered GNN Model.

Tier	Basic	Attributes
AIE Core	CDFG	node: {*opcode, data type, bitwidth vector/scalar, offset, stream/memory*}
AIE Array	DFG	node: { *kernel vector, type, location, cycles* } edge: { *type, size, #channels, bitwidth* }

3.3 AIE Array-Tier Model

The AIE array connected by the NoC architecture presents unique challenges and opportunities for modeling. Thus, we propose an AIE array-tier network with a graph representation. In the modeling process, we construct a graph representation of the AIE array, where each node corresponds to an AIE core, and the edges represent the data transfer connections between these cores. The locations of the nodes within the AIE array, along with the data transfer connections, serve as input to our model. Given the differing bandwidths between neighboring and non-neighboring cores, we introduce additional edge features to our graph representation as shown in Table 1. These edge features enable us to model the impact of data transfer connections on the performance of the AIE array. We integrate kernel feature vectors, generated by the AIE core-tier model, with these features to form the ADF decoration, which serves as the graph representation for the AIE array-tier

model. These feature vectors encapsulate the performance characteristics of individual AIE cores. By integrating these features into our AIE array-tier model, we can capture the impact of individual cores on the AIE array's overall performance.

We utilize the Graph Attention Network (GAT) as the backbone of our model to model the running cycles across the entire array. The GAT layers concentrate on the most critical nodes in the neighborhood, thereby capturing the most pertinent information. The first GAT layer is dedicated to local dependencies. It updates each node's feature vector based on the feature vectors of its immediate neighbors in the graph. This process enables the model to capture the impact of the local configuration of the AIE array on the program's performance. In contrast, the second GAT layer focuses on global dependencies. It updates each node's feature vector based on the feature vectors of all other nodes in the graph, with weights determined by the attention mechanism. This approach allows the model to capture the impact of the AIE array's global configuration on the program's performance. Lastly, we employ a global max pooling function as our aggregation function. This function allows us to extract global features from our graph representation, which represent the overall data flow within the AIE array. By using a global max pooling function, we ensure that our model gives precedence to the most significant features of the data flow. Therefore, the features extracted by the AIE array-tier model, $\boldsymbol{f}$, can be represented as:

$$\boldsymbol{f} = \max_{i=1}^{|U|}(GAT(ReLU(GAT(Concat(\boldsymbol{y}^i, \boldsymbol{U}^i), \boldsymbol{D}, \boldsymbol{C}, \boldsymbol{A}_1), \boldsymbol{D}, \boldsymbol{C}, \boldsymbol{A}_2)) \tag{2}$$

where $\boldsymbol{U}^i$, $\boldsymbol{D}$, $\boldsymbol{C}$, $\boldsymbol{A}_j$ donate the i-th core in the DFG of the ADF graph, the edges of the DFG, the features of edges, and the weights of the j-th GAT layers, respectively.

3.4 Two-stage Training

The G^{2}PM consists of two tier models designed to learn different levels of information. To effectively train this dual-tiered GNN model, we propose a two-stage training method. This approach allows the core-tier model to focus on the AIE kernel performance, while the AIE array-tier model concentrates on the array performance.

The first stage involves training the AIE core-tier model, which is designed to capture the characteristics of individual AIE cores. We construct the training data using AIE kernel programs and detailed performance information of individual cores. The input to this tier model is the CDFG f AIE programs, which are transformed into LLVM IR by the AIE Clang compiler. Incorporating the node features defined in Table 1, we build the CDFG into the vector of the training dataset. For the label field in the training, we use the AIE simulator to generate detailed cycle costs for each AIE core and associate them with the corresponding CDFG. We also add a fully connected (FC) layer for this AIE performance prediction, adopting the Mean Squared Error (MSE) loss function for this regression task. In this stage, we train the AIE core-tier model to learn the features of AIE kernel programs, which represent the performance of running cycles.

The second stage involves training the AIE array-tier model, designed to capture the global characteristics of the AIE array and the interactions between cores. The input to this tier model includes the feature vectors of the cores generated by the pre-trained core-tier model, the detailed features of nodes, and the connection edges of the AIE array, as listed in Table 1. We construct the training data using the ADF graph description, node placement and routing information, AIE kernel programs, and performance of the whole array as labels. The labels used in this training stage are the total running cycles obtained from the AIE simulator for the corresponding input. In the training process, we first feed the AIE kernel programs to the AIE core-tier model with frozen parameters to get the feature vectors of cores. Then, the model propagates the input vector in the GAT layers and outputs the cycle results via an FC layer. Here, we choose Mean Squared Error (MSE) as the loss function also. MSE quantifies the difference between the predicted and actual performance, and we utilize backpropagation to update the parameters in the array-tier model.

By training the dual-tiered GNN model in two stages, we can ensure that each tier is adequately trained to capture the characteristics it is designed to model.

Table 2: Design space of optimization with AIE-OPT.

Pass	Factor	Type	Range
affine-loop-fusion	fusion-compute-tolerance	Int	0-10
	fusion-local-buf-threshold	Int	0-4096
	mode	Enum	All, In, Mixed
affine-loop-tile	cache-size	Int	0-4096
	size	Int	0-1024
affine-loop-unroll	cleanup-unroll	Bool	T/F
	unroll-factor	Int	1-128
	unroll-full	Bool	T/F
	unroll-num-reps	Int	1-7
affine-super-vectorize	vectorize-reductions	Bool	T/F
	virtual-vector-size	Int	1-32
aie-vectorize	dup-factor	Int	1-32
	shift	Int	1-128

4 ILLUSTRATIVE USE CASES OF G^{2}PM

In this section, we will illustrate the compatibility and applicability of G^{2}PM through two examples. Kernel program optimization and heterogeneous workload partitioning are two prevalent processes in the application development flow for ACAP architecture.

4.1 Compilation Optimization with MLIR-AIE

MLIR-AIE [3] is an open-source compilation infrastructure specifically designed for developing applications on AIEs. The development flow of MLIR-AIE is illustrated in Figure 2(b). MLIR-AIE introduces several dialects and optimization passes that can optimize kernel programs by adjusting various parameters. Using the automatic exploration framework outlined in [5], we can construct an efficient Design Space Exploration (DSE) tool using MLIR-AIE, further enhanced by G^{2}PM. The objective of this tool is to identify the combination of passes and factors that minimizes the running cycles for a given kernel program. G^{2}PM plays a crucial role in providing accurate and efficient performance estimation for kernel programs using the AIE core-tier model. As mentioned in Figure 2(b), the design space for optimization is detailed in Table 2. The exploration algorithms from OpenTuner [5] are used, with the objective being to minimize the number of running cycles. The detailed workflow of exploring the best pass combination of AIE-OPT in MLIR-AIE can be summarized as follows: Given the original kernel program

as input, the DSE tool enhanced by G^{2}PM generates passes and factors iteratively. After working with AIE-OPT and the tuning algorithms from OpenTuner, we can arrive at the final configuration after thousands of iterations.

4.2 Workload Partitioning on ACAP

Heterogeneous workload partitioning is a critical task when developing applications for the Versal ACAP architecture due to the presence of multiple heterogeneous hardware targets within the architecture. The PL fabric and AIE arrays exhibit different performances across various operators. As illustrated in Figure 2(c), a model-assisted partitioner uses performance estimates for different hardware targets as input. It generates the optimal partition scheme for specific applications, taking into account the time cost of data transfer simultaneously. The partitioning problem can be formulated simply as:

$$\underset{\boldsymbol{P}}{Minimize} \sum_{op} Perf(\boldsymbol{P}, op) + Tran(\boldsymbol{P}) \tag{3}$$

where $\boldsymbol{P}$ denotes the partition results of the operators, $Perf$ denotes the time cost of a certain operator after partitioning, and $Tran$ denotes the time cost of data transfer.

In this process, G^{2}PM is used to generate performance estimates for operators on the AIEs, while the model in GNN-DSE [17] generates estimates for the PL target. The use of these models can significantly reduce the compilation and synthesis time needed to obtain performance estimates. The partitioner then applies a greedy algorithm using these estimation results to partition the application's workloads onto suitable targets (either the PL or AIEs). By using this approach, the partitioner can create an optimal partition scheme that efficiently leverages the capabilities of both the PL fabric and the AIE arrays. This leads to improved performance and productivity when developing applications for the Versal ACAP architecture.

5 EVALUATION RESULTS

5.1 Dataset Generation and Evaluation Setup

We constructed our dataset, as detailed in Table 3, from diverse sources to ensure a comprehensive evaluation of our methodology. We collected programs from various domains including Matrix multiplication (MM), Monvolution (Conv), Fast Fourier Transformation (FFT), and Finite Impulse Response filter (FIR), each with different scales and computational requirements. Additionally, we adjusted the scales in both ADF graphs and kernel programs to augment the real-world program dataset. These programs are representative of the typical workloads that an ACAP architecture might encounter in real-world applications. To increase the total number of datasets for training and testing, we also have generated synthetic programs using heuristic random algorithms, which involved generating ADF graphs and kernel programs.

For all programs in the dataset, we utilized the AIE compiler and AIE simulator in AMD/Xilinx Vitis 2022.1 to generate the LLVM IR and simulation results, which served as the ground truth. During the compiling process, we set the hardware target as Versal ACAP VCK5000 and the frequency for AIE to 1.25GHz. After removing the failed instances, we collected 1636 programs from real-world sources and 8000 programs from synthetic sources. In our dataset, the number of nodes (and edges) ranges from 5 (9) to 460800 (720460) in the total of ADF graphs and DFG of AIE cores.

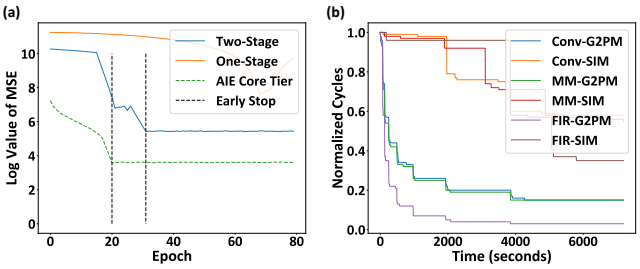

Figure 3: (a) Loss on evaluation set during training epochs; (b) Best results by time of DSE with G^{2}PM and AIE Sim.

We developed G^{2}PM using PyTorch [15] and utilized an Nvidia RTX 3080Ti for training and testing. The features of the GCN and GAT layers were set to 64, 32, 128, and 128, respectively. We allocated 80% of all data for training, 10% for validation, and 10% for testing. During the training process, we used the Adam Optimizer [12] with a learning rate of 0.001.

5.2 Model Evaluation

5.2.1 Targeted on the AIE Array. To evaluate the accuracy and efficiency of G^{2}PM, we select several representative prior methods, applied them to the same benchmark in our test dataset, and present the results in Table 4. CHARM [19] proposes an analytical model for Matrix Multiplication (MM) designs on the ACAP architecture. Although it achieves a 4.9% error rate in the MM set, its application to other domains is limited. CDS [13] presents an analytical model within the compilation framework. Despite its quick computation time, it yields an average error of 19.5%. We also constructed a Network-on-Chip (NoC) model [14] based on the ACAP's hardware settings and generated performance estimates incorporation with the AIE simulator. When compared to these methods and the AIE simulator, G^{2}PM outperforms them significantly, achieving a speed-up of 4165× over the AIE simulator and maintaining an average error of just 1.6%. This makes G^{2}PM the most accurate method among those tested.

5.2.2 Targeted on Single AIE Core. The AIE core-tier model within G^{2}PM is capable of predicting the performance of a single AIE kernel. To evaluate this model, we used an AIE core dataset and compared it with other methods of single program performance estimation. Our model significantly outperforms both the analytic method, CDS [13], and the learning-based methods, IR2Vec [18] and GraphCodeBERT [9], in terms of accuracy, achieving an error rate of just 1.4%. Furthermore, in terms of efficiency, our model achieves a speed-up of 1390× over the AIE simulator.

5.2.3 Training Process. We conducted a comparison between our two-stage training method and the end-to-end one-stage training method, using the same settings for the model and optimizer. As depicted in Figure 3(a), the loss on the evaluation dataset converges much more quickly in the two-stage training process than in the one-stage process. Furthermore, the minimum value of loss achieved in the two-stage training is also smaller. The loss of the AIE-tier core model, as shown in the figure, indicates that it takes

Table 3: Dataset used for training and evaluation.

Source	Domain	#Programs	#Total Nodes (*min*/*max*/*avg*)	#Total Edges (*min*/*max*/*avg*)	#ADF Nodes (*min*/*max*/*avg*)	#ADF Edges (*min*/*max*/*avg*)	#Core Nodes (*min*/*max*/*avg*)	#Core Edges (*min*/*max*/*avg*)
Real	MM	576	9/204800/25840	9/400400/60000	1/400/64.6	2/1000/98.5	8/1024/388.2	7/1566/466.8
	Conv	288	17/460800/65560	16/672040/78800	1/400/80.5	2/1000/100.4	16/2718/656.5	14/3360/788.3
	FFT	288	139/164960/76800	166/720460/160330	5/320/48.0	7/456/80.4	134/3568/1698.2	159/3338/2003.9
	FIR	384	11/423780/106770	11/434200/108720	1/400/88.4	2/900/176.5	10/2066/520.5	9/2397/603.8
	Others	100	5/200300/16300	5/226400/17600	1/400/32.6	2/940/72.9	4/1576/498.9	3/1702/533.8
Synthetic	-	8000	11/320000/19440	12/464000/22550	1/400/38.8	2/1200/97.6	10/3000/500.0	10/3540/580.0

Table 4: Results of model evaluation on the test dataset.

Target Tier	Method	MRPE	Time (s)	Speed-up
AIE Core	AIE Sim.	-	194.6	1.0×
	CDS [13]	12.3%	0.07	2780×
	IR2VEC [18]	9.8%	0.87	223.7×
	GraphCodeBERT [9]	5.3%	1.25	155.6×
	AIE Core Tier in G^{2}PM	**1.4%**	**0.14**	**1390×**
AIE Array	AIE Sim.	-	1416	1.0×
	CHARM* [19]	4.9%	0.09	15733×
	CDS [13]	19.5%	0.89	1591×
	AIE Sim. + NoC Model [14]	15.7%	583.9	2.43×
	Dual Tier in G^{2}PM	**1.6%**	**0.34**	**4165×**

*CHARM is only tested on MM.

Table 5: Results of latency estimation (ms) and error (%) on DDPM in the model-assisted partitioner.

Operators	AIE (Est./ Error)	PL (Est./ Error)	Vitis-AI (ms)	Partitioner (ms)
UNet	2780/+0.72	14734/-0.59	-	-
Time Embedding	95.5/-0.61	9.03/-0.73	-	-
Layer Activation	121/-1.10	258/+0.44	-	-
Prior Mean	716/+0.98	228/-0.36	-	-
Total	-	-	5480	**3870**

approximately 8 hours to train the entire G^{2}PM, with the AIE-tier model specifically requiring about 1.4 hours.

5.3 Evaluation of Use Cases

5.3.1 G^{2}PM-powered DSE. We evaluate the task of compilation optimization with MLIR-AIE across three benchmarks: MM of size $32 \times 32 \times 32$, Conv of size $64 \times 64 \times 3 \times 3$, and FIR of size 12288×7. Additionally, we employ the AIE simulator to generate the AIE core performance estimation as a baseline. As illustrated in Figure 3(b), the DSE powered by G^{2}PM yields better optimization results than using the AIE simulator, due to its efficient and accurate estimation feedback. Specifically, the time cost per iteration in the DSE process is reduced to 1/41 when using G^{2}PM.

5.3.2 G^{2}PM-assisted Workload Partitioning. To evaluate the practicality of the model-assisted workload partitioner, as depicted in Figure 2(c), we selected the Denoising Diffusion Probabilistic Model [11] (DDPM) as an example and used the designs in Vitis-AI [4] on VCK5000 as the baseline. As illustrated in Table 5, our G^{2}PM model provides an accurate estimation of kernel latency on AIEs and PL, respectively, aiding in the development of a superior design with the partitioner. It is worth noting that there are other modules, such as memory access, in the final design. Additionally, our model for performance estimation significantly reduces compilation time compared to the tools in Vitis, achieving a specific speed-up of more than 240×.

6 CONCLUSION

In this paper, we present G^{2}PM, a dual-tiered graph representation learning-based performance model for ACAP architecture. Our model is trained using a two-stage training method, utilizing data from both AIE core and AIE array to predict the performance of AIE core programs and AIE array designs. Furthermore, we introduce two potential downstream tasks, which are integral parts of typical development flows, that are enabled by G^{2}PM. Through comprehensive evaluation, we demonstrate the efficiency and accuracy of G^{2}PM, as well as the practicality of the downstream tool.

ACKNOWLEDGMENT

This work was partly supported by the National Natural Science Foundation of China (Grant No. 62090021) and the National Key R&D Program of China (Grant No. 2022YFB4500500).

REFERENCES

[1] Sagheer Ahmad et al. 2019. Xilinx First 7nm Device: Versal AI Core (VC1902). In *IEEE Hot Chips 31 Symposium (HCS)*.
[2] AMD/Xilinx. 2020. *AI Engine Development.* https://docs.xilinx.com/p/ai-engine-development
[3] AMD/Xilinx. 2022. *MLIR-based AIEngine toolchain.* https://github.com/Xilinx/mlir-aie
[4] AMD/Xilinx. 2022. Vitis AI Library User Guide.
[5] Jason Ansel et al. 2014. OpenTuner: An extensible framework for program autotuning. In *PACT*.
[6] Prasanth Chatarasi et al. 2020. Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine. In *HPEC*.
[7] Zhangyin Feng et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*.
[8] Joseph A. Fisher. 1983. Very Long Instruction Word Architectures and the ELI-512. In *ISCA*.
[9] Daya Guo et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.
[10] William L. Hamilton et al. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.*
[11] Jonathan Ho et al. 2020. Denoising Diffusion Probabilistic Models. In *NeurIPS*.
[12] Diederik P. Kingma et al. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.
[13] Zhijing Li et al. 2022. Compiler-Driven Simulation of Reconfigurable Hardware Accelerators. In *HPCA*.
[14] Oumaima Matoussi. 2021. NoC Performance Model for Efficient Network Latency Estimation. In *DATE*.
[15] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.
[16] Nilton Luiz Queiroz et al. 2023. A graph-based model for build optimization sequences. *Journal of Computer Languages*.
[17] Atefeh Sohrabizadeh et al. 2022. Automated Accelerator Optimization Aided by Graph Neural Networks. In *DAC*.
[18] S. VenkataKeerthy et al. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.*
[19] Jinming Zhuang et al. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *FPGA*.
[20] Jinming Zhuang et al. 2023. High Performance, Low Power Matrix Multiply Design on ACAP: from Architecture, Design Challenges and DSE Perspectives. In *DAC*.