CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture

Jinming Zhuang
University of Pittsburgh
jinming.zhuang@pitt.edu

Jason Lau
University of California, Los Angeles
lau@cs.ucla.edu

Hanchen Ye
University of Illinois at Urbana-Champaign
hanchen8@illinois.edu

Zhuoping Yang
University of Pittsburgh
zhuoping.yang@pitt.edu

Yubo Du
University of Pittsburgh
yubo.du@pitt.edu

Jack Lo
Advanced Micro Devices Inc.
jack.lo@amd.com

Kristof Denolf
Advanced Micro Devices Inc.
kristof.denolf@amd.com

Stephen Neuendorffer
Advanced Micro Devices Inc.
stephen.neuendorffer@amd.com

Alex Jones
University of Pittsburgh
akjones@pitt.edu

Jingtong Hu
University of Pittsburgh
jthu@pitt.edu

Deming Chen
University of Illinois at Urbana-Champaign
dchen@illinois.edu

Jason Cong
University of California, Los Angeles
cong@cs.ucla.edu

Peipei Zhou
University of Pittsburgh
peipei.zhou@pitt.edu

ABSTRACT

Dense matrix multiply (MM) serves as one of the most heavily used kernels in deep learning applications. To cope with the high computation demands of these applications, heterogeneous architectures featuring both FPGA and dedicated ASIC accelerators have emerged as promising platforms. For example, the AMD/Xilinx Versal ACAP architecture combines general-purpose CPU cores and programmable logic with AI Engine processors optimized for AI/ML. An array of 400 AI Engine processors executing at 1 GHz can provide up to 6.4 TFLOPs performance for 32-bit floating-point (fp32) data. However, machine learning models often contain both large and small MM operations. While large MM operations can be parallelized efficiently across many cores, small MM operations typically cannot. We observe that executing some small MM layers from the BERT natural language processing model on a large, monolithic MM accelerator in Versal ACAP achieved less than 5% of the theoretical peak performance. Therefore, one key question arises: *How can we design accelerators to fully use the abundant computation resources under limited communication bandwidth for end-to-end applications with multiple MM layers of diverse sizes?*

We identify the biggest system throughput bottleneck resulting from the mismatch of massive computation resources of one monolithic accelerator and the various MM layers of small sizes in the application. To resolve this problem, we propose the CHARM framework to compose **multiple diverse MM accelerator architectures** working concurrently towards different layers within one application. CHARM includes analytical models which guide design space exploration to determine accelerator partitions and layer scheduling. To facilitate the system designs, CHARM automatically generates code, enabling thorough onboard design verification. We deploy the CHARM framework on four different deep learning applications, including BERT, ViT, NCF, MLP, on the AMD/Xilinx Versal ACAP VCK190 evaluation board. Our experiments show that we achieve 1.46 TFLOPs, 1.61 TFLOPs, 1.74 TFLOPs, and 2.94 TFLOPs inference throughput for BERT, ViT, NCF, MLP, respectively, which obtain 5.29×, 32.51×, 1.00× and 1.00× throughput gains compared to one monolithic accelerator.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Hardware** → **Hardware-software codesign**.

KEYWORDS

Heterogeneous Architecture, Domain-Specific Accelerator, Versal ACAP, Mapping Framework, Matrix-Multiply, Deep Learning

ACM Reference Format:

Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, Peipei Zhou. 2023. CHARM: Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3543622.3573210

1 INTRODUCTION

Dense matrix multiply (MM) serves as one of the most heavily used kernels in many deep learning workloads, including BERT [1] for natural language processing, NCF [2] for recommendations, ViT [3] for vision classification, and MLP [4] for multilayer perceptron classification or regression. According to profiling results from Google [5], dense matrix multiply tasks occupied 90% of Neural Network (NN) inference workload in Google's data center in 2017. The increasing complexity of these applications leads to extreme demands for computation and data movement.

According to [6, 7, 8, 9], the off-chip bandwidth has been a bottleneck for both the performance and energy efficiency of a system and a common trend on current platforms is that the off-chip bandwidth does not scale as fast as the computation resources. Therefore,

This work is licensed under a Creative Commons Attribution International 4.0 License.

FPGA '23, February 12–14, 2023, Monterey, CA, USA

© 2023 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9417-8/23/02.
https://doi.org/10.1145/3543622.3573210

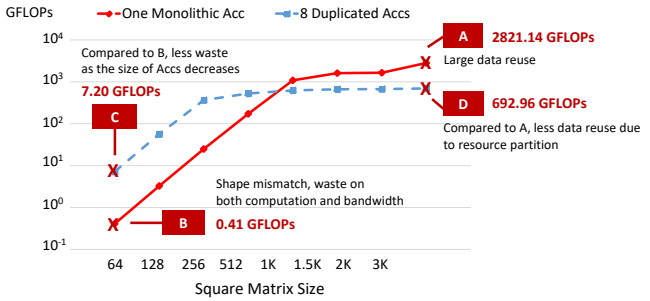

Figure 1: Throughput of square MM under different sizes.

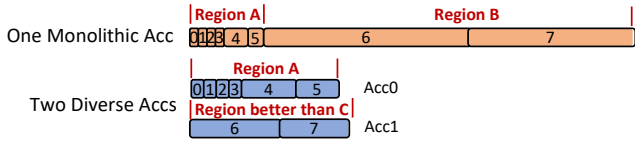

Figure 2: Execution timeline of one monolithic MM design vs. two diverse MM accs design for BERT on VCK190.

the first research question arises: *How to sustain the faster scaling computation with the slower scaling off-chip bandwidth?*

A common solution is to increase data reuse by allocating more on-chip storage within an accelerator (acc). As shown in asymptotic analysis in [9], the total off-chip communication volume in MM scales as $O(\frac{1}{\sqrt{M}})$ where M is the on-chip tile size. If we increase the tile size, we can reduce the total communication volume, therefore reducing the pressure on the off-chip bandwidth.

In this work, we target on the AMD/Xilinx Versal ACAP architecture [10], which combines general-purpose CPU cores and programmable logic (PL) with AI Engine processors (AIE) optimized for AI/ML computation. For example, we implemented an MM accelerator on an AMD/Xilinx VCK190 board using 384 AIEs and over 80% on-chip URAM and BRAM resources. The red line in Figure 1 illustrates the performance of this accelerator. This design operates on a native tile size of 1536×128×1024 and achieves 2.8 TFLOPs throughput when carrying a tiled execution of a large square MM (point A). However, when simply mapping different sizes of MM to such a design, the performance decreases significantly as the square MM size drops below 512, since each tile is padded to the native tile size of the accelerator. For instance, at point B, the performance of such a monolithic design goes to 0.41 GFLOPs, which is 6880× lower than point A. Although padding is a common and simple approach to implementing small MM operations on a large accelerator, padding can waste both computation and bandwidth.

An alternative to padding is implementing multiple accelerators with smaller native tile sizes, potentially executing different tasks on each accelerator in parallel [11]. We apply this approach using eight independent accelerators with a native tile size of 256×128×256, illustrated by the blue dash line in Figure 1. For small square MM operations with size 64, this approach achieves 7.2 GFLOPS at point C, approximately 17× speedup compared to point B.

However, the smaller accelerator size also means less data reuse for large MM, with total throughput almost saturation when the operation size is larger than 256. When the MM size is 3072 (point D), the total throughput from eight duplicate accs is 4.08× smaller than point A in one monolithic design.

These experiments expose two conflicting design goals. Firstly, we want to implement large MM operations with sufficient data reuse to achieve the highest possible performance on the devices. Secondly, we want to implement small MM operations while minimizing computation and communication overheads. Neither of these simple designs seems able to achieve these design goals simultaneously. Therefore, the second research question arises: *How to trade-off between the two design goals for real-world, end-to-end applications where MM layers with large and small sizes coexist?*

To illustrate how these conflicting design goals can affect the performance of practical machine learning models, we consider BERT [1] as a representative workload containing MM layers with both large and small sizes. In a transformer layer of BERT, there are a total of 8 types of MM kernels where Kernels 0-5 are large MMs and Kernels 6 and 7 are batch dots, i.e., small MMs. The detailed shapes can be referred to Table 5. Take Kernel 5 and Kernel 6 as examples, Kernel 5 is an MM with the shape 3072×1024×4096, Kernel 6 is a batch dot with the shape 96×512×512×64, which means there are 96 small independent MMs sized at 512×512×64.

As shown in Figure 2, when using one monolithic MM accelerator, Kernels 0-5 consume 92% of the total BERT MM computation operations and 12% of the total MM acc time. In contrast, Kernels 6-7 consume 8% of the total operations but take 88% of the total MM acc time. For Kernels 0-5, they lie in Region A (a region that performs similarly to Point A in Figure 1), where the throughput of acc is more than 2082 GFLOPS. For Kernel 6-7, they lie in Region B, where the throughput of acc is only 23.6 GFLOPS. Given there is a large portion of acc execution underutilized in the timeline, the overall MM acc throughput is only 276 GFLOPS. Can we achieve a design for BERT that lies in region A, i.e., good for large MMs, and also in a region better than point C, i.e., good for small MMs with less or no waste computation/bandwidth?

Our answer is "Yes". The key idea is to allocate more portion of the resources to accs dedicated to computing larger MMs and a smaller portion of the resources to other accs to compute smaller MMs at the same time, as shown in Figure 2 where a two-diverse accs system is illustrated. To achieve our design goals, we need to solve these *new challenges*. **First**, we need to achieve high computation utilization for every single acc, i.e., use the smaller acc(s) to reduce the waste for small MMs and use the larger acc(s) to maximize the data reuse for large MMs. **Second**, to maximize overall utilization while maintaining high throughput and low latency, we need to carefully overlap the execution time for these accs by co-optimizing workload and resource partitioning. **Third**, to facilitate the design space explorations (DSE), we need analytical models to optimize the overall throughput under resource and bandwidth constraints. **Fourth**, to reduce the programming efforts for the system implementation, we need automatic code generation. **Fifth**, to resolve the dependency of the kernels within the application graph when running multiple accs we need an accelerator runtime to schedule kernels from different tasks onto the accs.

To answer the research questions, we propose the CHARM architecture and its corresponding automation framework, the CHARM framework. Our contributions are summarized below:

- **CHARM Systematical Design Methodology on Versal**: To achieve high computation and communication efficiency of each acc, in Section 4, we propose a thorough design methodology on Versal heterogeneous platform. We further provide automatic CHARM DSE (CDSE) to find the optimized single acc configuration. To the best of our knowledge, this is the first work that provides a detailed analysis of the systematical data movement and computation on Versal.
- **CHARM Architecture and Framework:** To achieve the design goals of good performance for MMs with both small and large sizes in an application, in Section 5, we propose the CHARM architecture and the CHARM framework to find the optimized design. In the CHARM framework, there are several modules. First, on top of CDSE, we propose the CHARM diverse accelerator composer (CDAC), which features a sort-based two-step search algorithm to find an optimized CHARM design in the polynomial time complexity instead of exponential time complexity. Furthermore, to automate the system implementation, CHARM automatic code generation (CACG) is proposed to generate source code files for AIEs, PL, and host CPU. Lastly, the CHARM runtime system (CRTS) is launched in the host CPU that schedules different kernels to the accs for optimizing both task latency and overall system throughput.
- We deploy the CHARM framework to accelerate four applications on VCK190 in Section 6. Our on-board experiments demonstrate that CHARM achieves 1.46 TFLOPs, 1.61 TFLOPs, 1.74 TFLOPs, and 2.94 TFLOPs inference throughput for BERT, ViT, NCF, MLP, respectively, which obtain 5.29×, 32.51×, 1.00×, and 1.00×, throughput gains compared to one monolithic accelerator.
- **White-Box Open-Source Tools for Versal.** While AMD provides users a block-box IP for NN applications called DPU [11], we open-sourced our tools completely as a white-box with a detailed step-by-step guide to reproduce all of the results presented in this paper and for the other users to learn and leverage in their end-to-end systems. (***https://github.com/arc-research-lab/CHARM***)

2 PRIOR WORK

To achieve high throughput and energy efficiency, NN accelerators usually employ a large number of processing elements (PE) and share a similar memory hierarchy. That is, while the big bulk of data is stored in the off-chip memory, there are multiple levels of on-chip buffers, including the local memory attached to each PE and global shared memory, to further reduce the costly data movement from/to off-chip memory. Several works contribute to NN accelerators by discussing the data reuse opportunities, computation parallelism, and the choice of dataflow.

However, many of the prior works apply a one-size-fits-all monolithic design that cannot efficiently handle layers with huge differences in shapes and sizes (Eyeriss [12, 13], ShiDiannao [14], NPU [15, 16, 17] and others [18, 19, 20, 21]). AutoSA [22] is a polyhedral-based compilation framework that generates monolithic systolic array designs for dense matrices. Sextans and Serpens [23, 24] are general-purpose monolithic accelerators for sparse matrices. [25, 26] analyze layout and pipeline efficiency. Other works like AMD DPU [11], Mocha [27] explore task-level parallelism by allocating multiple duplicate accs on the device without specializing each acc. DNNBuilder [28] designs a dedicated acc for each layer according to the number of operations within the layer. DNNExplorer [29] enhances DNNBuilder by combining dedicated accs for the first several layers and a monolithic acc for the rest of the layers. While it employs multiple accelerators, it lacks a comprehensive exploration of workload assignments. TETRIS [30] and TANGRAM [31] propose multiple dataflow optimizations within and across the NN layers to improve performance and energy efficiency. Although they offer diverse accelerator designs, they lack the DSE and workload assignment for high overall throughput. Herald [32] proposes an architecture with multiple diverse accelerators and explores the workload assignment and resource partition. Still, they choose several existing acc designs from their candidate pool, e.g., ShiDiannao [14], NVDLA [33] without doing DSE for each acc. FPCA [34] and CHARM'12 [35] propose a fully pipelined and dynamically composable coarse-grained reconfigurable architecture and compose loosely coupled accelerators for different kernels within an application via permutation network, which costs high in chip area.

Table 1: Comparison with prior works.

Prior Works	One Mono	Multi Duplicate	Multi Diverse	Workload Assignment	Specializa -tion for Acc
Eyeriss etc. [12]-[26]	✓	×	×	×	×
DPU etc. [11, 27]	✓	✓	×	×	×
DNN Expl. etc. [28, 29]	✓	✓	✓	×	×
Herald [32]	✓	✓	✓	✓	×
CHARM (Ours)	✓	✓	✓	✓	✓

In conclusion, we summarize the differences between our work and prior works in Table 1. Our work is capable of choosing the design from one monolithic, multiple duplicates, and multiple diverse accelerators, and each accelerator is a specialized design considering the different workload assignments, dataflow, and data parallelism strategies that are covered by our DSE.

3 VERSAL ACAP ARCHITECTURE OVERVIEW

In this section, we first introduce the system architecture of AMD/Xilinx Versal ACAP architecture in Section 3.1 and then the memory model of AIE array in Section 3.2.

3.1 Versal ACAP Architecture

Figure 3 illustrates the overall architecture of the VCK190 [36] board and highlights the AIE array on the top. The VCK190 board features (1) the first-generation AIE architecture, which has 8×50 **1 GHz** 7-way VLIW processors supporting vector operations up to 1024 bits [37], (2) ARM processors to run Linux and general-purpose applications, and (3) PL to design application-specific hardware with Digital Signal Processors (DSP) available for integration. The AI engine cores and ARM CPUs can be programmed with C/C++ code, while PL can be programmed using both RTL and C/C++ code using High-Level Synthesis (HLS) [38, 39, 40, 41, 42, 43]. These three components are integrated with I/O peripherals, such as PCIe and

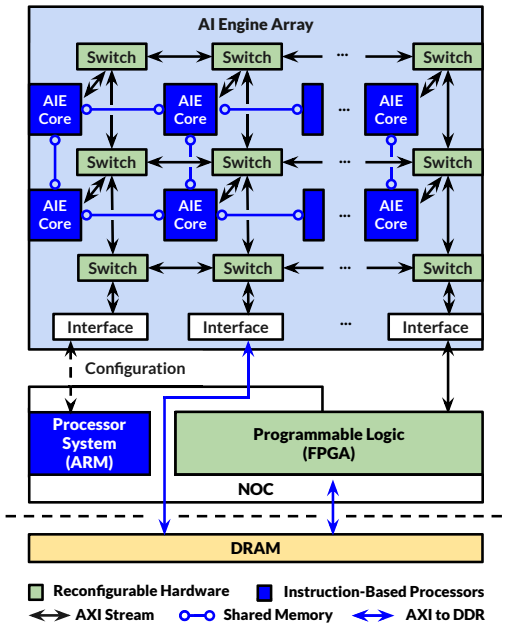

Figure 3: Versal ACAP architecture.

DRAM controllers, into a heterogeneous SoC with a Network-on-Chip (NoC). The VCK190 board is equipped with one DDR4-DIMM off-chip memory with a 25.6 GB/s peak bandwidth.

3.2 AIE Memory Model

Each AIE processor tile contains 32 KB of data and is capable of sharing data with the adjacent AIEs in four directions (AIE↔neighbor AIE). In addition to local memory shared with adjacent tiles, each AIE tile also connects to an AXI-Stream (AXIS) switch network, which enables non-local communication between AIE processors (AIE↔non-local) and communication with the PL through the PLIOs in the 39 interface tiles (PL↔AIEs). The VCK190 board provides 1.2 TB/s (PL↔AIEs) / 0.9 TB/s (AIEs↔PL) bandwidth between PL and AIEs, which is 46× more than the bandwidth between DDR4 and PL. The AXIS switches support both circuit-switched and packet-switched connections between ports. Circuit-switched connections provide dedicated, deterministic communication and support broadcast, where data from a single input channel is transmitted to multiple output channels simultaneously. Packet-switched connections allow data from an input channel to be dynamically routed to different destinations based on a destination header at the start of each packet. This enables data flows to be time-multiplexed on a single routing path. One situation in which we can use packet-switched connections happens when the computation-to-communication (CTC) ratio of an AIE is more than one. During the computation of AIE 0, the port assigned to this AIE is idle and thus can be used to transfer data to another AIE, say AIE 1, by assigning a different header that matches the destination ID of AIE 1.

4 CHARM SINGLE ACCELERATOR DESIGN

In this section, we describe the dataflow and mapping strategy for a single MM acc using hundreds AIEs in Section 4.1. Then, in Section 4.2, we present the data reuse optimizations to balance the

```
1   // Off-Chip <-> On-Chip Time Loop
2   for(int i.0=0;i.0<TX;i.0++)   // TX=M/(TI*A*X)
3   for(int j.0=0;j.0<TZ;j.0++)     // TZ=N/(TJ*C*Z)
4   for(int k.0=0;k.0<TY;k.0++)       // TY=K/(TK*B*Y)
5     copyDataFromOffChipOnChip(...)
6     // PL On-chip Buffer Reuse Time Loop
7     for(int i.1=0;i.1<X;i.1++)   // X
8     for(int j.1=0;j.1<Z;j.1++)     // Z
9     for(int k.1=0;k.1<Y;k.1++)       // Y
10      copyDataFromOnChiptoAIE(...)
11      // AIE Array Spatial Loop
12      for(int i.2=0;i.2<A;i.2++)   // A
13      for(int j.2=0;j.2<C;j.2++)     // C
14      for(int k.2=0;k.2<B;k.2++)       // B
15          // Single AIE 2D-SIMD Vectorization Loop
16          for(int i.3=0;i.3<TI;i.3++)
17          for(int j.3=0;j.3<TJ;j.3++)
18          for(int k.3=0;k.3<TK;k.3++)
19              ...
20              2D-SIMD(i.3,j.3,k.3);
```

Listing 1: Pseudocode of MM loop tiling and dataflow.

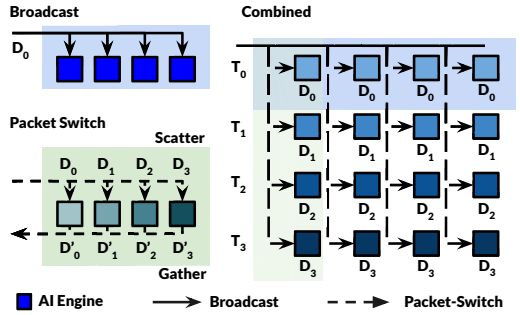

Figure 4: Combining broadcast circuit-switched and packet-switched connections to reduce required I/O to AIE array.

massive computation parallelism and communication among AIEs and between PL↔AIEs and PL↔DDR.

4.1 Dataflow and Mapping Strategy of a Single Matrix Multiply Accelerator

Listing 1 depicts the overall four-level tiling and mapping strategy for a basic dense matrix-matrix multiply. The innermost loop tiling (Lines 16-20) implements MM on a single AIE core and exploits instruction-level parallelism and data-level parallelism by issuing fully pipelined 2D-SIMD (vector-matrix multiplication) instructions. Each AIE stores a (TI×TK) LHS and a (TK×TJ) RHS matrix and computes a (TI×TJ) output matrix in its local memory. The second-innermost loop tile (Lines 12-14) represents the spatial distribution of execution across different AIE cores in the AIE array. These loops are fully unrolled and computed on (A×B×C) AIE cores in a parallel fashion. The spatial distribution also corresponds to the number of required IOs, which will be discussed in Section 4.2. The third-innermost time loop tile (Lines 7-9) represents the sequential processing of data stored in PL on-chip memories. The data from on-chip PL buffers are fed into the AIE array (X×Y×Z) times, and the intermediate partial sum from the AIE array is accumulated on PL. The outermost loop (Lines 2-4) represents the temporal processing of data stored in off-chip memory, enabling the processing of large matrices that do not fit in on-chip memory. The loop boundary can be decided by the overall input matrix size (M, K, N).

4.2 Data Reuse in Multiple Levels

When designing each acc, we adopt a bottom-up strategy and explore data reuse at each level. Firstly, at a single AIE level, we make full use of the seven-way VLIW capability of the AIE vector processor to achieve fully pipelined MAC operations by reusing the AIE local register and the local memory.

Secondly, at the PL↔AIEs level, when feeding data to tens or hundreds of AIEs, as the number of PLIOs connecting the AIE array and PL is much smaller than the total number of AIE cores, we reduce the number of required PLIO by exploring the data broadcast and packet-switch (described in Section 3.2) mechanism. Figure 4 shows how we reuse one single PLIO port by combing broadcast with packet-switch. Assume that we have a 4×4 AIE array that calculates an MM with size 1×4×4 (1 MAC/AIE), and it takes one cycle for one AIE to get the left-hand-side (LHS) and the right-hand-side (RHS) operands and four cycles to finish one multiplication which makes the CTC ratio equal to 4. By leveraging the data reuse opportunity in MM (e.g., the row of LHS can be reused by different columns of RHS), we can broadcast the first data from LHS to the first row of AIE arrays at Time 0 utilizing one PLIO port as shown in solid lines. At Time 1, by specifying a different destination header, we can transfer the second data of LHS to the second row of the AIE array by reusing the same PLIO port. At Time 2 and Time 3, the third and fourth data of LHS are sent to the third and fourth rows of AIEs. At Time 4, the first row of AIEs finishes the computation, and the PLIO completes the data transfer to the fourth row of AIEs. Therefore, in this case, we can use one PLIO port to send LHS data to 16 AIEs without any performance degradation.

Thirdly, in PL↔DDR, we further allocate three sets of on-chip buffers for each acc to store the LHS, the RHS, and the output matrices so that a tile of LHS with size (X×A×TI) × (Y×B×TK) can be reused on-chip for (Z×TJ) times. The buffer size and reuse rate for RHS and output matrices can be calculated in the same way. Besides, the double-buffering technique is applied to three buffers to overlap the off-chip data movement with the computation. By greatly exploring the data reuse opportunities at multiple levels, our system can sustain high computation efficiency under limited off-chip bandwidth, i.e., 25.6 GB/s of DIMM-DDR4 on VCK190.

5 CHARM ARCHITECTURE AND CHARM FRAMEWORK TO COMPOSE MULTIPLE DIVERSE ACCELERATORS

In this section, we introduce the CHARM architecture in Section 5.1 and CHARM framework overview in Section 5.2. We then discuss each module within the framework from Section 5.3 to Section 5.6.

5.1 CHARM Architecture

Figure 5 illustrates the CHARM architecture with one or more diverse MM accs in the system and other kernel accs for non-MM kernels within an end-to-end deep learning application. We partition the AIE array for multiple MM accs (two in this example). For each MM acc, we design a specialized DMA module that contains the data transferring control logic and on-chip buffer according to the tiling strategy. The different AIE partitions communicate with their corresponding DMA modules through the PLIO interface and NOC. We refer to the AIE array, its corresponding PLIO,

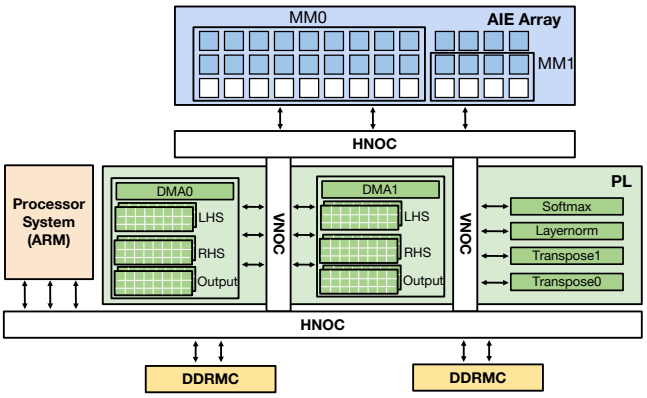

Figure 5: System architecture of multiple diverse MM accs and other non-MM accs.

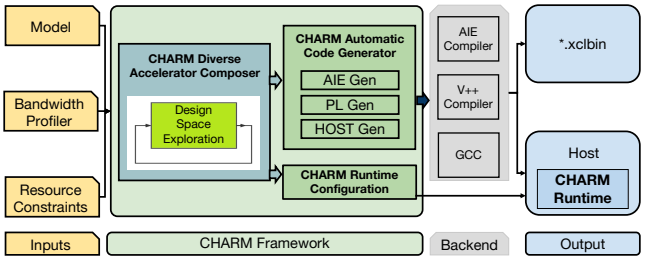

Figure 6: CHARM framework overview.

and the DMA module as one MM acc design. For each non-MM kernel, e.g., transpose, softmax, and layer normalization in BERT, ViT models, we design one acc for each type of kernel on the PL side. Each non-MM acc contains DMA, computation logic, and local buffers. For these communication-bound kernels, the design goal is to achieve near-peak off-chip bandwidth. When running these kernels, as they consume all the off-chip bandwidth, we choose to sequentially launch these non-MM and communication-bound kernels before or after MM acc(s).

5.2 CHARM Framework Overview

We illustrate the proposed CHARM framework overview in Figure 6. The CHARM framework takes the application model, platform off-chip bandwidth profiling, and platform hardware resource constraints as input, performs automated optimization and code generation, and launches backend compilers to generate the ready-to-run binaries as output. There are several modules in the CHARM framework: (1) On top of CDSE, CDAC finds the optimal design with the highest throughput and outputs the configurable design parameters for each acc. It also generates a runtime config file that specifies which acc should be called for a certain kernel. (2) CACG takes the configurable parameters generated from CDAC as input, implements the design, and generates all the needed source code files for AIEs, PL, and host CPUs. CHARM calls the corresponding backend tools to generate both the hardware bitstream and host binaries. (3) CRTS takes the runtime config files from CDAC and kernel dependency graph as input and schedules the kernels in the task pools onto available accs.

5.3 CHARM Design Space Exploration (CDSE) for a Single Acc

DSE Configurable parameters: A, B, C, X, Y, Z. In order to attain optimized throughput for each diverse accelerator, we design CDSE, which takes matrix sizes (M, K, and N), optional user-specified hardware constraints, and hardware platform off-chip characterization database as input and perform an analytical model-based search. During CDSE, we set the single AIE workloads to 32×32×32, i.e., TI=TK=TJ=32. We achieve up to 95% of kernel efficiency for MM, utilize 75% of the AIE local memory in this design point, and obtain the CTC ratio of 4. The outputs of CDSE are the configurable parameters, including **A, B, C, X, Y, Z** that meet all the hardware constraints. The parameters A, B, C determine the number of AIE and PLIO used in the AIE array. X, Y, Z, A, B, C together with prefixed parameters **TX, TY, TZ** decide the number of utilized on-chip buffers. This optimization problem can be formulated as an integer programming (IP) optimization problem shown as below. AIE_{num}, $PLIO_{in}$, $PLIO_{out}$ and On_chip_{RAM} represent the user-specified hardware constraints:

$$\max \mathit{Throughput} = M \cdot K \cdot N \cdot 2/\mathit{TIME} \tag{1}$$

$$\text{s.t.} A \times B \times C \leq AIE_{num} \tag{2}$$

$$\begin{aligned} Port_{in} &\leq PLIO_{in}, \\ Port_{out} &\leq PLIO_{out} \end{aligned} \tag{3}$$

$$\mathit{Buff} \leq On_chip_{RAM} \tag{4}$$

AIE-Array Tiling Selection. Since {A, B, C} are fully unrolled and mapped to the AIE Array, the multiplication of the unroll factors A, B, and C should be less than or equal to the total number of AIEs in Equation 2. The number of packet-switch ports is determined by {A, B, C} and the I/O reuse mechanism described in Section 4.2. They should meet the input and output PLIO resource constraints. The input and output PLIO numbers can be obtained by:

$$\begin{aligned} Port_{in} &= \lceil A \cdot B/CTC \rceil \\ &+ \lceil C \cdot B/CTC \rceil \\ Port_{out} &= \lceil A \cdot C/CTC \rceil \end{aligned} \tag{5}$$

PL Tiling Selection. On-chip PL buffers are allocated in order to amortize the 46x bandwidth gap between off-chip to PL and PL to AIE-Array by increasing the data reuse rate. Equation 6 shows the size of LHS, RHS, output buffers, and their off-chip to on-chip communication time. BPD refers to bytes per data and BW_L,R,O are the off-chip bandwidth measured from bandwidth profiling.

$$\begin{aligned} \mathit{Buff}_L &= (X \cdot A \cdot TI) \cdot (Y \cdot B \cdot TK) \cdot BPD \\ \mathit{Buff}_R &= Y \cdot Z \cdot B \cdot C \cdot TK \cdot TJ \cdot BPD \\ \mathit{Buff}_O &= X \cdot Z \cdot A \cdot C \cdot TI \cdot TJ \cdot BPD \\ \mathit{Buff} &= 2 \cdot (\mathit{Buff}_L + \mathit{Buff}_R + \mathit{Buff}_O) \\ \mathit{Time}_{L,R,O} &= \mathit{Buff}_{L,R,O}/BW_{L,R,O} \end{aligned} \tag{6}$$

Performance Modeling. To calculate the overall execution time, the scheduling of data communication between off-chip to on-chip and the AIE array computation should be considered. The computation time for all the on-chip time loops, i.e., Line 6 in Listing 1, can be defined by Equation 7 in which MAC represents the theatrical MAC operation that one AIE engine can do in one cycle, and Eff refers to the real efficiency that the computation kernel achieves. We consider both single AIE and AIE array pipeline efficiency (PL↔AIE) here and assign the overall efficiency to 80%. For the off-chip to on-chip scheduling, as described in Listing 1, the loop order of the outermost loop is TY→TZ→TX, thus, the memory access time for LHS and RHS will happen TX×TX×TZ times in total. The overall execution TIME can be calculated by Equation 8. This is an equation for illustration purposes where we leave out the details on the formulation of time spent storing the output and prologue and epilogue time in the pipeline.

$$\mathit{Time_comp} = (X \cdot Y \cdot Z \cdot TI \cdot TK \cdot TJ/MAC)/Eff \tag{7}$$

$$\begin{aligned} \mathit{TIME} = max([\mathit{Time}_L, \mathit{Time}_R, \mathit{Time_comp}]) \\ \cdot (TX \cdot TY \cdot TZ) \end{aligned} \tag{8}$$

For any specific shape(s), all the possible configurable parameters will be evaluated in an exhaustive fashion. After CDSE, top-ranked optimized design points will be reported.

5.4 CHARM Diverse Acc. Composer (CDAC)

Two-step search algorithm in CDAC. To achieve overall optimized throughput when mapping diverse sizes of MM kernels on multiple accs, we propose a sort-based two-step algorithm in CDAC. In the first step of CDAC, we partition the MM kernels of different workloads within an input model to multiple groups. The number of groups equals the number of diverse accs, which is a hyperparameter in CDAC. After the workload partition, in the second step, we generate a resource partition candidate that specifies the resource budget for each accelerator to be proportional to the total number of operations from the assigned MM kernel(s). Under the assigned workload and assigned resource, we search all valid candidates of configurable parameters (A,B,C,X,Y,Z) for each accelerator. We then fine-tune the memory resource partition to generate more resource partition candidates. After the memory fine-tuning, we generate a new workload partition and redo the resource partition and configurable parameter search which further optimize the system throughput of all the accs. We discuss the details of each step as follows.

1st Step: Workload Assignment. To improve the overall throughput of the diverse acc architecture, we need to properly assign the MM kernels to the accs and make them work concurrently with a similar execution time. However, mapping an application with **n** kernels to **num** accs suffers from the exponential time complexity as the total mapping search space scales as $O(num^n)$. To better scale larger models that contain more kernels, i.e., a larger n, we propose a sort-based algorithm to partition the workload with reduced time complexity as $O(\binom{n-1}{num-1}) = C_{num-1}^{n-1}$. As shown in Algorithm 1, CDAC first sorts the different shapes of the MM kernels by their number of operations (Line 4) so that MMs with larger and smaller sizes can be properly divided. Then we divide the sorted MM kernels into n groups (Lines 5-6). For example, if there are eight different shapes of kernels that need to be mapped to num=2 accs, after sorting the kernels, we put one separator between any two kernels to separate all kernels into two groups. In total, it gives us $C_{2-1}^{8-1} = 7$ grouping design choices.

2nd Step: Hardware Resource Partitioning. For each workload assignment, we perform DSE to find the optimized acc configurable parameters under the partitioned hardware resource constraints,

Algorithm 1 Diverse Accelerator Composer Algorithm

Input: layer[n], bw, hw_sr, num, ubound
▷ **layer[n]** are n layers in an application. **bw** refers to bandwidth, **hw_sr** includes the AIE, PLIO, RAM resources, **num** refers to the number of accs, **ubound** is the hyperparameter for memory tuning
Output: Workload[num], final_Acc[num]
▷ **Workload** and **final_Acc** contains the workload assignment and the hardware configuration for each acc respectively

1: $BW \leftarrow bw/num_acc$
2: $HW.RAM[:] \leftarrow hw_sr.ram/num_acc$
3: $final_cycle \leftarrow inf$
4: $layer_sort[:] \leftarrow \text{sort}(layer)$
5: **for** sche in range($C(^{n-1}_{num-1})$) **do**
6: $partition[:] \leftarrow \text{partition}(layer_sort[:], num, sche)$ ▷ 1st step
7: $op_portion[:] \leftarrow \text{cnt}(partition[:])$ ▷ 2nd step
8: update($HW.AIE[:], HW.PLIO[:], op_portion[:]$)
9: $Acc[:], cycle[:] \leftarrow \text{Acc_search}(HW, BW, partition[:])$
10: ▷ Sequentially launch CDSE
11: **while** $tune_cnt \neq ubound$ **do** ▷ Memory tuning
12: $index \leftarrow \max(cycle[:])$
13: update($HW.RAM[:], index$) ▷ Increase the memory of the slowest acc
14: $Acc[:], cycle[:] \leftarrow \text{Acc_search}(HW, BW, partition[:])$
15: **if** $\max(cycle[:]) < final_cycle$ **then** ▷ Update optimal point
16: $final_cycle \leftarrow \max(cycle[:])$
17: $final_Acc[:] \leftarrow Acc[:]$
18: $Workload[:] \leftarrow partition[:]$
19: $tune_cnt$++
20: Define Acc_search($HW, BW, partition[:]$) :
21: **for** acc in range(num) **do**
22: CDSE($partition[acc], HW[acc], BW[acc]$)
23: return $Acc[:], Cycle[;]$

including the number of AIEs, PLIO, on-chip RAM, and off-chip bandwidth. To minimize the maximum execution time of all the accs, CDAC assigns the number of AIEs and PLIO constraints proportional to the total number of operations assigned to the acc (Lines 7-8). For the number of on-chip RAMs, we first evenly distribute it (Line 2). After sequentially launching CDSE to find the configuration of every acc once (Lines 9-10), we apply a memory fine-tuning step to optimize the memory allocation. It finds the index of the acc that consumes the most time (Line 12) and then tries to explore a better configuration by increasing the memory allocation of this acc while decreasing the memory allocation of others'(Line 13-14). If a better result is found, we update the global optimal execution cycles and corresponding acc configuration settings (Line 15-18). Note that, in the current model, we assume each acc evenly occupies the off-chip bandwidth (Line 1) and leave the discussion of the off-chip bandwidth partition for future work.

5.5 CHARM Auto. Code Generation (CACG)

After finding the hardware design parameters of optimized designs from CDAC, we implement CACG, including AIEGen, PLGen, and HostGen, to generate the corresponding source code for AIEs, PL, and host CPU. AIEGen takes the tiling factor of a single AIE (TI,TK,TJ) and AIE Array (A,B,C) as input and instantiates the corresponding number of AIE cores. It leverages the C++-based Adaptive Data Flow (ADF) Graph API [44] to build connections among AIE cores through the AXI network and connections between AIE Array and PL through PLIOs. Using the PL level (X,Y,Z) design parameters, PLGen generates HLS C/C++ code that allocates on-chip buffers on the PL side and implements the data transferring modules for sending/receiving data to/from the AIE array. HostGen emits the Xilinx runtime library (XRT) API-based host code.

After code generation, CHARM launches the vendor tools, including the AIE compiler and the V++ compiler to generate the output object files `libadf.a` and `kernel.xo` which are linked into one `xclbin`, i.e., the hardware bitstream of the design. The GCC compiler compiles XRT-API-based host code to host program runs on the ARM CPU for kernel scheduling and system controls.

5.6 CHARM Runtime Scheduler (CRTS)

Algorithm 2 Runtime Scheduler Algorithm

Input: Graph, num, task_pool[task][layer]
Output: Runtime scheduling for each accelerator

1: **while** (1) **do** ▷ Assign ready tasks to corresponding Accs
2: **for** acc in range(num) **do**
3: **if** $\neg Acc[acc]$.idle() **then**
4: **Continue**
5: **for** t in range($tasks$) **do**
6: **for** l in range($layer$) **do**
7: **if** $task_pool[t][l] \neq \emptyset \wedge task_pool[t][l]$.valid() **then**
8: $Acc[acc]$.assign($task_pool[t][l]$)
9: **Continue** line 2
10: **while** (1) **do** ▷ Update task_pool according to dependency graph
11: **for** acc in range(num) **do**
12: **if** $Acc[acc]$.finish() **then**
13: $task_pool$.update($Graph$)
14: $Acc[acc]$.update($idle$)

To achieve high throughput while maintaining relatively low latency under dependency constraints within each task, we propose CRTS that runs on the ARM CPU during runtime. Algorithm 2 lists the scheduler algorithm. It takes the dependency graph, number of accelerators, and layer assignment configuration file generated by CDAC as input. There are two parallel processes in CRTS.

The first process keeps tracking to check if there are any idle accs we can assign tasks to (Lines 2-3). CRTS traverses the layers assigned to this acc following a first-in-first-out principle (Lines 5-6). If the layer is still in the task pool, it means that it has not been issued. Suppose all the preceding layer(s) of the current layer have been executed, i.e., dependency resolved. In that case, CRTS assigns this valid layer to the corresponding acc (Lines 7-8) and continues to track other accs (Line 9). The second process keeps track of the status of every acc to see if it has finished the workload (Lines 12-13) and updates the task pool according to the dependency graph, as well as changing the status of the acc (Line 14) to idle.

6 EXPERIMENT RESULTS

In this section, we first illustrate the single AIE efficiency and single MM acc throughput in Section 6.1 and 6.2. In Section 6.3, we implement different CHARM designs, including one monolithic MM acc, one specialized MM acc, two-diverse MM accs, and eight-duplicate MM accs for four applications: BERT, ViT, NCF, and MLP. All the experiments are conducted on VCK190 with 230MHz on PL and 1GHz on AIE. AMD/Xilinx Vitis version 2021.1 is used as the compilation backend tool. When measuring the power consumption, we iterate each application for more than 60s and report the average value by employing the board evaluation and management tool, AMD/Xilinx BEAM [45].

6.1 Single AIE Kernel Efficiency Comparison

In this section, we showcase our single AIE MM computation efficiency under different matrix sizes for fp32. We leverage the AIE intrinsics [46] to program the single kernel design and obtain the

Table 2: Single AIE MM comparison under fp32 data type.

	H-GCN[48]		CHARM (this work)		
Size: M x K x N	MACs/Cyc	Eff	MACs/Cyc	Eff	Eff gain
16 × 16 × 16	2.34	29.30%	6.18	77.22%	***2.64x***
32 × 32 × 32	3.64	45.50%	7.57	94.70%	***2.08x***
64 × 64 × 8	3.64	45.50%	7.54	94.29%	***2.07x***

Table 3: Performance comparison in GFLOPS between on-board measurements and CDSE analytical modeling estimations under different matrix sizes. The error rates in percentage show that CDSE achieves a high prediction accuracy.

Square MM size	On-board	Estimation	Error	Power(W)
64	0.41	0.40	-2%	32.58
128	3.36	3.22	-4%	32.86
256	25.58	25.79	1%	34.66
512	176.24	178.42	1%	37.95
1024	1103.46	1123.81	2%	41.78
1536	1633.13	1649.01	1%	46.02
2048	1672.76	1688.17	1%	47.87
3072	2850.13	2895.90	2%	50.65
4096	2718.42	2773.26	2%	51.97
6144	3277.99	3363.89	3%	53.57

execution cycle of our single AIE design by simulating on the Versal ACAP AI Engine System C simulator [47], a cycle-accurate architecture simulator. As shown in Table 2, our single AIE can achieve up to 7.57 MACs/cycle and 94.70% peak performance when MM size equals 32×32×32. Compared to the AIE dense MM kernel efficiency reported in H-GCN [48], our single kernel obtains 2.26× average efficiency gain. For the whole system design, we choose 32×32×32 as our single kernel as it achieves high computation efficiency and the total size of LHS, RHS and output matrices are within 16 KB so that they fit in the AIE local memory and can be double buffered.

6.2 Performance for Square MMs on One Monolithic Accelerator

We evaluate the throughput of one monolithic acc design and compare the performance between the modeling estimation from CDSE and the on-board measurement. We build the monolithic design by using 384 AIEs and over 83% of on-chip RAM utilization with the AIE running at 1GHz and the PL side at 230MHz. As shown in Table 3, the throughput of the one-acc monolithic design rises as the square MM size increases. While it achieves 3.27 TFLOP/s at size 6144, the throughput at size 64 is only 0.41 GFLOPs. CHARM CDSE is capable of precisely estimating the on-board execution time with an average estimation error rate of only 2.9%.

We compare the throughput of the same MM application implemented only in the PL part of VCK190 using the state-of-the-art systolic-array-based framework AutoSA [22] for fp32 data type. The PL side of VCK190 is featured with 1968 DSP58 IPs. Instead of using five DSP48 to calculate the floating point multiplication in the previous generation board, it only consumes one DSP58.

As shown in Table 4, the CHARM single MM acc achieves 3.27 TFLOPs throughput, 5.54× throughput and 1.93× energy efficiency gains compared to the PL-only design on VCK190.

Table 4: Comparison between PL only design and PL RAM + AIE design in CHARM on VCK190.

Data Type	PL [22] Float32	CHARM Float32
Frequency	PL:200MHz	PL:230MHz
URAM	0	384
BRAM	923	764
DSP/AIE	DSP58:1536	AIE:384
TFLOPs	0.59 (1x)	3.27 (5.54x)
Power(W)	18.60	53.40
Energy Eff	1x	1.93x

Table 5: MM sizes in BERT, ViT, NCF, MLP.

Model	# of layer	M	K	N	batch dot size
BERT	4	3072	1024	1024	N/A
	1	3072	4096	1024	N/A
	1	3072	1024	4096	N/A
	1	512	64	512	96
	1	512	512	64	96
ViT	1	3072	3024	1024	N/A
	1	3072	1024	1024	N/A
	1	3072	1024	4096	N/A
	1	3072	4096	1024	N/A
	1	3072	1024	3048	N/A
	2	64	64	64	768
NCF	1	3072	4096	2048	N/A
	1	3072	2048	1024	N/A
	1	3072	1024	512	N/A
	1	3072	512	256	N/A
	1	3072	256	128	N/A
	1	3072	128	64	N/A
	1	3072	64	32	N/A
	1	3072	32	16	N/A
	1	3072	32	1	N/A
MLP	1	3072	2048	4096	N/A
	2	3072	4096	4096	N/A
	1	3072	4096	1024	N/A

Table 6: Time breakdown for different types of kernels in the end-to-end solutions that achieves the highest throughput for BERT, ViT, NCF and MLP.

Kernel	BERT	ViT	NCF	MLP
MM	57.2ms	57.7ms	40.4ms	11.9ms
Layernorm	4.5ms	4.5ms	0	0
Softmax	18.7ms	2.3ms	0	0
Transpose	5.2ms	5.2ms	0	0

6.3 End-to-End Performance

We apply the CHARM framework to four applications, BERT, ViT, NCF, MLP. All the shapes of the MM kernels in these models are listed in Table 5. We explore the number of accs from 1 to 8 and showcase the representative CHARM designs, including one monolithic MM acc, one specialized MM acc, two-diverse MM accs, and eight-duplicate MM accs, for each application. The one monolithic MM design is described in section 6.2, which stays the same for all four applications. It is set as the baseline design for comparisons. All the other MM acc designs are customized for each application and are designed and implemented using the CHARM framework. All

Table 7: On-board throughput and power comparisons under different MM accs configurations for BERT, ViT, NCF, MLP.

App	CHARM cfg	LUT	BRAM	URAM	DSP	AIE	GFLOPS	Power(W)	GFLOPS/W (Ratio)
BERT	One_mono	103959(11.55%)	764 (79.01%)	384 (82.94%)	165 (8.38%)	384 (96%)	276.8	37.0	7.48 (1x)
	One_spe	90351(10.04%)	515 (53.26%)	64 (13.82%)	117 (5.95%)	256 (64%)	515.4	32.4	15.91 (2.13x)
	Two_diverse	***343774(38.20%)***	***534 (55.22%)***	***272 (58.75%)***	***442 (22.46%)***	***288 (72%)***	***1464.2***	***40.7***	***35.98 (4.81x)***
	8_duplicate	222956(24.78%)	664 (68.67%)	384 (82.94%)	488 (24.80%)	256 (64%)	534.2	34.2	15.62 (2.09x)
ViT	One_mono	103959(11.55%)	764 (79.01%)	384 (82.94%)	165 (8.38%)	384 (96%)	49.5	32.4	1.53 (1x)
	One_spe	76661(8.52%)	275 (28.44%)	64 (13.82%)	187 (9.50%)	256 (66%)	217.1	28.0	7.75 (5.08x)
	Two_diverse	***240563(26.73%)***	***590 (61.01%)***	***320 (69.11%)***	***299 (15.19%)***	***264 (72%)***	***1609.0***	***39.6***	***40.63 (26.60x)***
	8_duplicate	222956(24.78%)	664 (68.67%)	384 (82.94%)	488 (24.80%)	256 (64%)	382.2	32.8	11.65 (7.63x)
NCF	***One_mono***	***103959(11.55%)***	***764 (79.01%)***	***384 (82.94%)***	***165 (8.38%)***	***384 (96%)***	***1736.0***	***45.2***	***38.41 (1x)***
	One_spe	***103959(11.55%)***	***764 (79.01%)***	***384 (82.94%)***	***165 (8.38%)***	***384 (96%)***	***1736.0***	***45.2***	***38.41 (1.00x)***
	Two_diverse	161597(17.96%)	790 (81.70%)	352 (76.03%)	326 (16.57%)	384 (96%)	1730.9	45.1	38.38 (0.99x)
	8_duplicate	222956(24.78%)	664 (68.67%)	384 (82.94%)	488 (24.80%)	256 (64%)	671.0	35.0	19.17 (0.50x)
MLP	***One_mono***	***103959(11.55%)***	***764 (79.01%)***	***384 (82.94%)***	***165 (8.38%)***	***384 (96%)***	***2936.7***	***51.4***	***57.13 (1x)***
	One_mono	***103959(11.55%)***	***764 (79.01%)***	***384 (82.94%)***	***165 (8.38%)***	***384 (96%)***	***2936.7***	***51.4***	***57.13 (1.00x)***
	Two_diverse	148158(16.46%)	919 (95.04%)	448 (96.76%)	344 (17.48%)	384 (96%)	2386.1	48.8	48.90 (0.86x)
	8_duplicate	222956(24.78%)	664 (68.67%)	384 (82.94%)	488 (24.80%)	256 (64%)	696.0	35.2	19.77 (0.35x)

Table 8: Resource utilization for each acc in the design for BERT with two MM diverse accs, four non-MM accs.

Type	REG	LUTLogic	LUTMem	BRAM	URAM	DSP	AIE
MM0+DMA0+buffer	96790 (5.55%)	91034 (10.41%)	835 (0.19%)	515 (53.26%)	256(55.29%)	246(12.50%)	256(64%)
MM1+DMA1+buffer	62415 (3.58%)	94739 (10.83%)	37668 (8.48%)	19 (1.96%)	16 (3.46%)	196(9.96%)	32 (8%)
Layernorm	45101 (2.58%)	33939 (3.88%)	4234 (0.95%)	15 (1.55%)	90 (19.44%)	129(6.55%)	0 (0%)
Softmax	34270 (1.96%)	33623 (3.84%)	2854 (0.64%)	243 (25.13%)	0 (0%)	151(7.67%)	0 (0%)
Transpose0	14217 (0.81%)	6926 (0.79%)	1097 (0.25%)	15 (1.55%)	0 (0%)	94 (4.78%)	0 (0%)
Transpose1	33967 (1.95%)	58510 (6.69%)	32512 (7.32%)	15 (1.55%)	0 (0%)	19 (0.97%)	0 (0%)

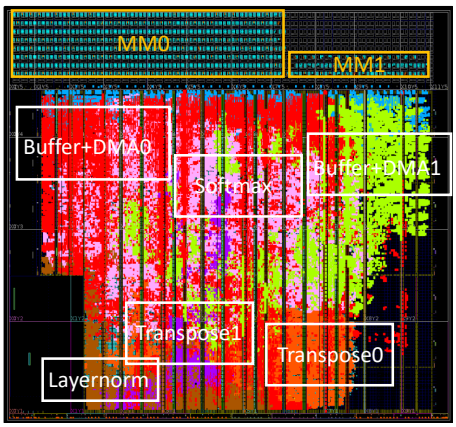

Figure 7: System implementation layout of the two-diverse MM accs and four non-MM accs for BERT.

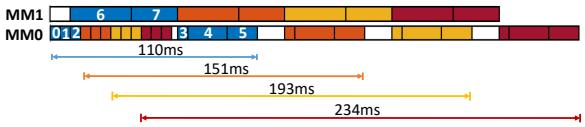

Figure 8: Timeline of four tasks scheduled on 2 accs for BERT.

the designs of the same application use the same non-MM kernels. Table 7 reports the on-board throughput and power consumption under different acc configurations for all the four applications.

CHARM achieves 1.46 TFLOPs, 1.61 TFLOPs, 1.74 TFLOPs, and 2.94 TFLOPs maximum throughput for the MMs in BERT, ViT, NCF, MLP. Table 6 shows the time breakdown for the MM, the layernorm, the softmax, and the transpose for each end-to-end application. We highlight the best design(s) for each application in Table 7. For BERT and ViT, the two-diverse MM accs designs are the best, whereas for NCF and MLP, one-acc designs are the best. This is because BERT and ViT have both large and small MMs whereas MLP only has large MMs. NCF also has both large and small MMs. However, small MMs consume less than 0.8% of the total computation, and designs favoring the large MMs stand out as the best. The eight-duplicate designs are inferior for all the applications due to insufficient data reuse for each acc.

For BERT and ViT, when compared to one monolithic design, the customization of using one specialized acc design for a specific application provided by CHARM gives 2.13×, 5.08× gain on energy efficiency (GFLOPs/W), respectively. The additional design spaces explored by using more than one-acc, with heterogeneous and diverse shaped accs provided by CHARM framework, give us 2.25×, 5.24× extra energy efficiency gains for BERT and ViT, respectively. **These gains demonstrate the innovative design methodology of CHARM, i.e., composing heterogeneous accelerators.**

We show the implementation layout of the two-diverse MM acc design, i.e., the best design for BERT, in Figure 7. This is also the layout corresponding to Figure 5 that contains two MM accs and four non-MM communication-bound accs. The hardware resource utilization for each acc is reported in Table 8. The MM acc 0 provides high data reuse and computation efficiency when calculating large MMs by utilizing 256 AIEs, 53.26% BRAM, and 55.29% URAM. The MM acc 1 utilizes 32 AIEs, 1.96% BRAM, and 3.46% URAM which provides the needed computation and communication without resource over-provisioning for small MMs in BERT.

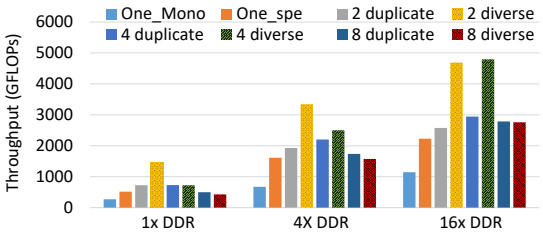

Figure 9: Throughput comparison under different off-chip bandwidth configurations from CHARM for BERT.

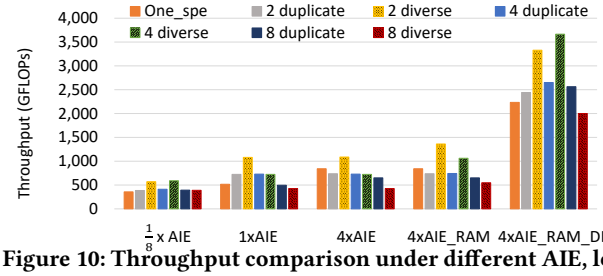

Figure 10: Throughput comparison under different AIE, local storage, and off-chip configurations from CHARM for BERT.

Explore latency-throughput tradeoff in CHARM. As shown in Figure 8, we map four concurrent tasks on the BERT design with two-diverse accs. Each task has eight MM kernels and there are dependency edges, including 0→6, 1→6, 6→7, 2→7, 7→3→4→5, where x→y means y depends on x. The other non-MM communication-bound kernels are not shown in the figure for illustrative simplicity. It takes 110ms to finish the 1st task and 234ms to finish the 4th task. For one-acc specialized design, each task latency is 162.6ms. Therefore, we have a design tradeoff, i.e., one specialized acc design can process fine-grained tasks whereas two-diverse accs design requires coarse-grained tasks to fill the pipeline of the two accs. Comparing to the one specialized Acc design, with 0.67×, 0.92×, 1.18×, 1.43× latency for different tasks, we gain 2.8× overall throughput in return. This illustrates that the CHARM framework allows explorations on the latency-throughput tradeoff and users can specify targets to let CHARM generate the designs that optimize throughput while meeting the latency requirement or vice versa.

CHARM DSE Efficiency. We use CHARM to perform a sort-based two-step search algorithm in CDAC. For BERT, compared to the exhaustive search, CDAC finds the optimal solution in 170 seconds whereas the exhaustive search takes 33 mins (#search iterations: 2M vs. 58M) with MATLAB R2021b on an Intel Core i9-10900X CPU.

7 DISCUSSION OF ARCHITECTURE INSIGHT AND MAPPING INSIGHT

By leveraging the strong modeling capability provided by the CHARM framework, we explore performance under different hardware architecture changes (number of AIEs, on-chip storage size, off-chip bandwidth) to do pre-silicon architecture explorations and provide architecture design insights that could be helpful for future generation devices. Here, we leverage the CHARM modeling to report throughput for different acc configurations including 1-, 2-, 4-, and 8-accs in the system. For each design (except one-acc), we have two variants, duplicate accs or diverse accs. The explorations help us to understand the following research questions:

Q1: Can we benefit from higher off-chip bandwidth?

A1: Yes. Versal needs higher off-chip bandwidth.

We first explore the performance, assuming the platform has more off-chip bandwidth. We increase the DDR bandwidth by 4× to simulate multiple DDR banks and by 16× to simulate the case when we have a high bandwidth memory (HBM). As shown in Figure 9, the throughput from the best design for BERT in each bandwidth configuration rises from 1.48 TFLOPs to 3.34 TFLOPs with 4× bandwidth and to 4.80 TFLOPs with 16× bandwidth. The improvement from 1× to 4× DDR is within expectation and implies that the designs for BERT are bounded by off-chip bandwidth. The maximum throughput for 16× is bounded by the system computation throughput as 4.8 TFLOPs, which is constrained by single kernel computation efficiency (95%) and PL↔AIE efficiency(85%). Another observation from Figure 9 is that the throughput improvement of multiple accs is larger than that of the single acc since when the number of accs increases, each acc has less data reuse and tends to be more bounded by the off-chip bandwidth.

Q2: Can we leverage CHARM in future architectures?

A2: Yes. The last group in Figure 10 implies that as the computation and communication parallelism further increases in the future, there is a need for more heterogeneous accelerator architectures and CHARM can serve as one of the most promising solutions.

We explore the performance by varying the number of AIEs, on-chip RAM, and off-chip bandwidth. We reduce the number of AIEs to 1/8 of the current AIE array size to simulate the computation capacity of the previous generation FPGA where only PL is equipped with DSPs and has about 1/8 of the theoretical fp32 peak performance of Versal ACAP. As shown in the first group in Figure 10, the performance difference between the minimum and the maximum under different acc configurations is less than 40%. As the computation parallelism is reduced to 1/8, the waste resulting from the inconsistency between the massive parallelism and the small MM size is mitigated. On the other hand, as shown in the last group in Figure 10, 4-diverse acc stands out as the best when we increase AIE, on-chip storage, and off-chip bandwidth all by 4×. Simply increasing AIEs does not give significant improvement whereas increasing all the resources as a whole does.

8 CONCLUSION AND ACKNOWLEDGEMENT

In this paper, we propose the CHARM architecture and the CHARM framework to provide a novel system-level design methodology for composing heterogeneous accelerators for different MMs within an application and generating end-to-end application solutions. We will explore and extend CHARM for more applications and more data types in our future work.

We acknowledge the support from the University of Pittsburgh New Faculty Start-up Grant, NSF awards #2213701, #2217003 and the support from CRISP, one of six SRC JUMP centers. We thank all the reviewers for their valuable feedback and Marci Baun for helping edit the paper. We thank AMD/Xilinx for FPGA and software donation, and support from the AMD/Xilinx Center of Excellence at UIUC, the AMD/Xilinx Heterogeneous Accelerated Compute Cluster at UCLA, and the Center for Research Computing (CRC) at University of Pittsburgh.

REFERENCES

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[3] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[4] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking TPU, GPU, and CPU platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019.

[5] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[6] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems*, pages 171–243. Springer, 2023.

[7] Geraldo F Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks. *IEEE Access*, 9:134457–134502, 2021.

[8] Hasan Hassan, Minesh Patel, Jeremie S Kim, A Giray Yaglikci, Nandita Vijaykumar, Nika Mansouri Ghiasi, Saugata Ghose, and Onur Mutlu. Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 129–142, 2019.

[9] Jim Demmel. Communication avoiding algorithms. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1942–2000. IEEE, 2012.

[10] AMD/Xilinx. Versal Adaptive Compute Acceleration Platform.

[11] AMD. IP Overlays of Deep learning Processing Unit , 2022.

[12] Yu-Hsin Chen et al. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 2016.

[13] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[15] Eriko Nurvitadhi, Dongup Kwon, Ali Jafari, Andrew Boutros, Jaewoong Sim, Phillip Tomson, Huseyin Sumbul, Gregory Chen, Phil Knag, Raghavan Kumar, et al. Why compete when you can work together: Fpga-asic integration for persistent rnns. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 199–207. IEEE, 2019.

[16] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C Hoe, Vaughn Betz, and Martin Langhammer. Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 10–19. IEEE, 2020.

[17] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[18] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. FBLAS: Streaming linear algebra on FPGA. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE, 2020.

[19] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible communication avoiding matrix multiplication on fpga with high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 244–254, 2020.

[20] Chen Zhang et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proc. of FPGA*, pages 161–170. ACM, 2015.

[21] Duncan J. M. Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H. W. Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 107–116. Association for Computing Machinery, Feb 2018.

[22] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 93–104. Association for Computing Machinery, Feb 2021.

[23] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 65–77, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 211–216, 2022.

[25] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. Latte: Locality Aware Transformation for High-Level Synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 125–128, 2018.

[26] Peipei Zhou, Hyunseok Park, Zhenman Fang, Jason Cong, and André DeHon. Energy Efficiency of Full Pipelining: A Case Study for Matrix Multiplication. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 172–175, 2016.

[27] Peipei Zhou, Jiayi Sheng, Cody Hao Yu, Peng Wei, Jie Wang, Di Wu, and Jason Cong. MOCHA: Multinode Cost Optimization in Heterogeneous Clouds with Accelerators. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 273–279, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Xiaofan Zhang et al. Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas. In *Proc. ICCAD*, page 56. ACM, 2018.

[29] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[30] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, 2017.

[31] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 807–820, 2019.

[32] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83. IEEE, 2021.

[33] Nvidia. Website. http://nvdla.org/.

[34] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A Fully Pipelined and Dynamically Composable Architecture of CGRA. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, 2014.

[35] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, page 379–384, New York, NY, USA, 2012. Association for Computing Machinery.

[36] AMD/Xilinx. Versal AI Core Series VCK190 Evaluation Kit, 2022.

[37] AMD/Xilinx. AI Engine Technology, 2022.

[38] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[39] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. FPGA HLS Today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[40] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *2009 IEEE 7th Symposium on Application Specific Processors*, pages 35–42. IEEE, 2009.

[41] Alexandros Papakonstantinou, Yun Liang, John A Stratton, Karthik Gururaj, Deming Chen, Wen-Mei W Hwu, and Jason Cong. Multilevel granularity parallelism synthesis on fpgas. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 178–185. IEEE, 2011.

[42] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N Do, and Deming Chen. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012, 2012.

[43] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. Extending high-level synthesis for task-parallel programs. In *2021 IEEE 29th Annual*

International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 204–213, 2021.
[44] AMD/Xilinx. Adaptive Data Flow API.
[45] AMD/Xilinx. Board evaluation and management Tool.
[46] AMD/Xilinx. AI Engine API and Intrinsics User Guide.
[47] AMD/Xilinx. Versal™ ACAP AI Engine System C simulator.
[48] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. H-GCN: A graph convolutional network accelerator on versal acap architecture. *arXiv preprint arXiv:2206.13734*, 2022.