CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture

JINMING ZHUANG, University of Pittsburgh, USA
JASON LAU, University of California, Los Angeles, USA
HANCHEN YE, University of Illinois at Urbana-Champaign, USA
ZHUOPING YANG, University of Pittsburgh, USA
SHIXIN JI, University of Pittsburgh, USA
JACK LO, Advanced Micro Devices Inc, USA
KRISTOF DENOLF, Advanced Micro Devices Inc, USA
STEPHEN NEUENDORFFER , Advanced Micro Devices Inc, USA
ALEX JONES, University of Pittsburgh, USA
JINGTONG HU, University of Pittsburgh, USA
YIYU SHI, University of Notre Dame, USA
DEMING CHEN, University of Illinois at Urbana-Champaign, USA
JASON CONG, University of California, Los Angeles, USA
PEIPEI ZHOU, University of Pittsburgh, USA

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

Authors' addresses: Jinming Zhuang, jinming.zhuang@pitt.edu, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15261; Jason Lau, lau@cs.ucla.edu, University of California, Los Angeles, 404 Westwood Plaza, Los Angeles, California, USA, 90095; Hanchen Ye, hanchen8@illinois.edu, University of Illinois at Urbana-Champaign, 1308 W MAIN ST, Urbana, Illinois, USA, 61820; Zhuoping Yang, zhuoping.yang@pitt.edu, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15261; Shixin Ji, shixin.ji@pitt.edu, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15261; Jack Lo, jack.lo@amd.com, Advanced Micro Devices Inc, San Jose, California, USA; Kristof Denolf, kristof.denolf@amd.com, Advanced Micro Devices Inc, San Jose, California, USA; Stephen Neuendorffer , stephen.neuendorffer@amd.com, Advanced Micro Devices Inc, San Jose, California, USA; Alex Jones, akjones@pitt.edu, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15261; Jingtong Hu, jthu@pitt.edu, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15261; Yiyu Shi, yshi4@nd.edu, University of Notre Dame, 325D Cushing Hall of Engineering, Notre Dame, Indiana, USA, 46556; Deming Chen, dchen@illinois.edu, University of Illinois at Urbana-Champaign, 1308 W MAIN ST, Urbana, Illinois, USA, 61820; Jason Cong, cong@cs.ucla.edu, University of California, Los Angeles, 404 Westwood Plaza, Los Angeles, California, USA, 90095; Peipei Zhou, University of Pittsburgh, 3700 O'Hara Street, Pittsburgh, Pennsylvania, USA, 15261, peipei.zhou@pitt.edu.

Manuscript submitted to ACM

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 between 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 on different layers within one application. CHARM includes analytical models which guide design space exploration to determine accelerator partitions and layer scheduling. To facilitate system designs, CHARM automatically generates code, enabling thorough onboard design verification. We deploy the CHARM framework on four different deep learning applications in FP32, INT16, and INT8 data types, including BERT, ViT, NCF, and 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, and MLP in FP32 data type, respectively, which obtain 5.29×, 32.51×, 1.00×, and 1.00× throughput gains compared to one monolithic accelerator. CHARM achieves the maximum throughput of 1.91 TOPS, 1.18 TOPS, 4.06 TOPS, and 5.81 TOPS in the INT16 data type for the four applications. The maximum throughput achieved by CHARM in the INT8 data type is 3.65 TOPS, 1.28 TOPS, 10.19 TOPS, and 21.58 TOPS, respectively. We have open-sourced our tools, including detailed step-by-step guides to reproduce all the results presented in this paper and to enable other users to learn and leverage CHARM framework and tools in their end-to-end systems: ***https://github.com/arc-research-lab/CHARM***.

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

Additional Key Words and Phrases: Heterogeneous Architecture, Domain-Specific Accelerator, Versal ACAP, Mapping Framework, Matrix-Multiply, Deep Learning

ACM Reference Format:
Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Shixin Ji, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Yiyu Shi, Deming Chen, Jason Cong, and Peipei Zhou. xxxx. CHARM 2.0: Composing Heterogeneous Accelerators for Deep Learning on Versal ACAP Architecture. *ACM Trans. Reconfig. Technol. Syst.* xx, x, Article xxx (x xxxx), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Dense matrix multiplication (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 multiplication tasks occupied 90% of the 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 [6–9], 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 computational resources. Therefore, 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 the 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, thereby reducing the pressure on the off-chip bandwidth.

In this work, we target 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

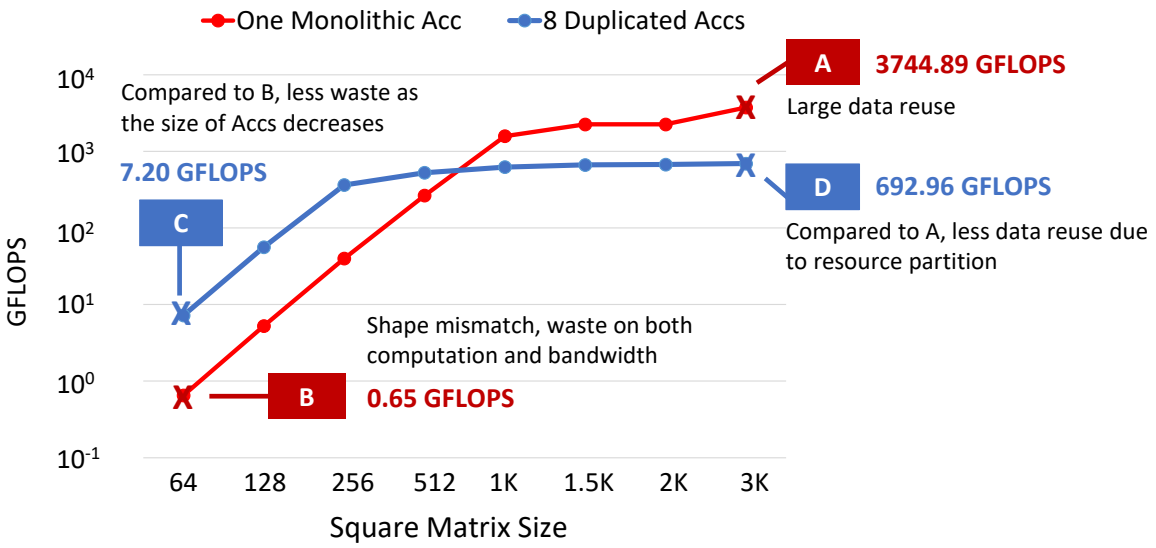


Fig. 1. Throughput of square MM under different sizes.

BRAM resources. The red line in Figure 1 illustrates the performance of this accelerator. This design operates on a native tile size of $1536 \times 128 \times 1024$ and achieves 3.7 TFLOPS throughput when carrying out 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 drops to 0.65 GFLOPS, which is 5760× lower than at point A. Although padding is a common and simple approach to implementing small MM operations on a large accelerator, it can waste both computation and bandwidth. More specifically, in point B, when using an accelerator that computes with a tile size of 1536×128×1024, padding the matrix from 64×64×64 to 1536×128×1024 leads to a 768× inefficiency due to excessive invalid computation. Additionally, in point B, since there is only one tile, there is no overlap between communication and computation, which leads to another 7.5× gap. Together, this explains the inefficiency caused by the mismatch between the original kernel shape and the monolithic accelerator's shape size.

An alternative to padding is to implement 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, as illustrated by the blue dashed line in Figure 1. For small square MM operations with size 64, this approach achieves 7.2 GFLOPS at point C, approximately an 11× speedup compared to point B.

However, the smaller accelerator size also means less data reuse for large MM, with total throughput nearly saturating when the operation size is larger than 256. When the MM size is 3072 (point D), the total throughput from eight duplicate accs is 5.4× smaller than that at 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 does one trade off between the two design goals for real-world, end-to-end applications where MM layers of large and small sizes coexist?*

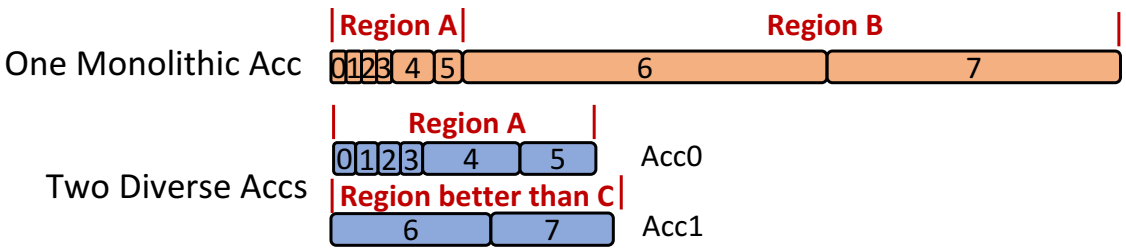


Fig. 2. Execution timeline of one monolithic MM design vs. two diverse MM accs design for BERT on VCK190.

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 of 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 in Table 6. 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 a single 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 Kernels 6-7, they lie in Region B, where the throughput of acc is only 23.6 GFLOPS. Given that 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 wasted computation/bandwidth?

Our answer is “Yes". The key idea is to allocate a larger portion of the resources to accs dedicated to computing larger MMs and a smaller portion of the resources to other accs for computing smaller MMs simultaneously, 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 times for these accs by co-optimizing workload and resource partitioning. **Third**, to facilitate the design space exploration (DSE), we need analytical models to optimize the overall throughput under resource and bandwidth constraints. **Fourth**, to reduce the programming effort for 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 for each acc, in Section 4, we propose a thorough design methodology on the Versal heterogeneous platform. We further provide an automatic CHARM DSE (CDSE) to find the optimized single acc configuration.

- **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 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 the host CPU. Lastly, the greedy-algorithm-based CHARM Runtime Scheduler (CRTS) is launched in the host CPU that schedules different kernels to the accs for optimizing both task latency and overall system throughput. A mixed-integer-programming (MIP) mathematical formulation is proposed to prove the optimality and search time efficiency of the greedy-algorithm-based CRTS.
- We deploy the CHARM framework to accelerate four applications with multiple data types on the 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 FP32 inference throughput for BERT, ViT, NCF, and MLP applications, which obtain 5.29×, 32.51×, 1.00×, and 1.00× throughput gains compared to one monolithic accelerator. CHARM achieves the maximum throughput of 1.91 TOPS, 1.18 TOPS, 4.06 TOPS, and 5.81 TOPS in the INT16 data type for the four applications. The maximum throughput achieved by CHARM in the INT8 data type is 3.65 TOPS, 1.28 TOPS, 10.19 TOPS, and 21.58 TOPS respectively.
- **White-Box Open-Source Tools for Versal.** While AMD provides users with a black-box IP for NN applications called DPU [11], we have open-sourced our tools completely as a white-box, including a detailed step-by-step guide to reproduce all the results presented in this paper and to enable other users to learn and leverage them 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 (PEs) and share a similar memory hierarchy. That is, while the 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 [15–17] and others [18–21]). 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

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

Table 1. Comparison with prior works.

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.

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 of three options: monolithic, multiple duplicates, and multiple diverse accelerators. Each accelerator is a specialized design that considers different workload assignments, dataflow, and data parallelism strategies covered by our DSE.

3 VERSAL ACAP ARCHITECTURE OVERVIEW

In this section, we first use VCK190 as a representative example to illustrate the system architecture of AMD/Xilinx Versal ACAP architecture in Section 3.1. We then elaborate on the tile structure of a single AIE, the connections between AIEs↔AIEs and PL↔AIEs in Sections 3.2, 3.3 and 4.2, respectively.

3.1 Versal ACAP Architecture

Figure 3 illustrates the overall architecture of the VCK190 [36] and highlights the AIE array at 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 for running Linux and general-purpose applications, and (3) PL for designing 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 the PL can be programmed using both RTL and C/C++ code with High-Level Synthesis (HLS) [38–43]. These three components are integrated with I/O peripherals, such as PCIe and DRAM controllers, into a heterogeneous SoC with a Network-on-Chip (NoC). The VCK190 board is equipped with one DDR4-DIMM off-chip memory module with a peak bandwidth of 25.6 GB/s.

3.2 The Tile Architecture of a Single AIE

Figure 4 illustrates the tile architecture of the AIE, which mainly consists of the AIE core, AIE local memory, and wire connections. Each AIE core is a vector processor that supports up to 7-way VLIW instructions, including one vector, one scalar, two load, one store, and two move instructions. The vector instruction supports up to 8, 32, and 128 MACs/cycle in FP32, INT16, and INT8 data types, respectively. The scalar instruction supports general-purpose data processing, e.g., non-linear functions. While the load and store instructions are responsible for transferring data between local memory and vector registers, the move instruction handles data movement between the 2048-bit vector registers and 3072-bit

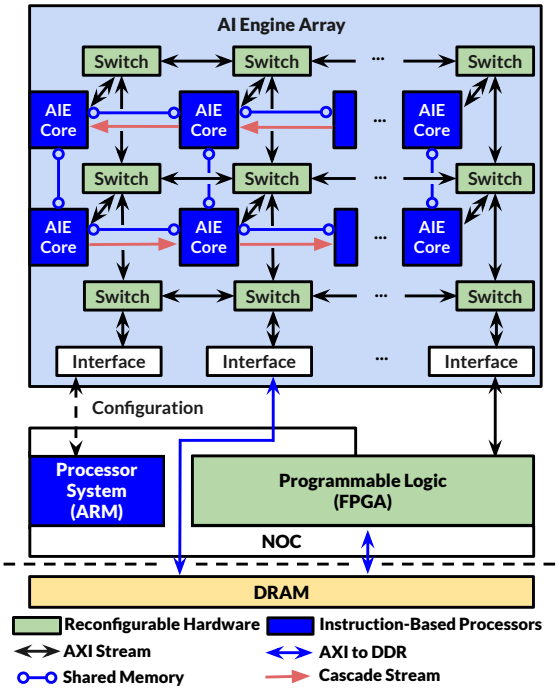


Fig. 3. Versal ACAP architecture.

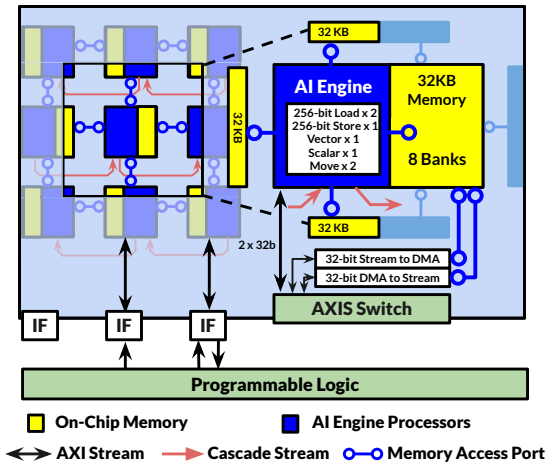


Fig. 4. Versal ACAP tile structure of AIE.

accumulation registers within each AIE core. Additionally, an AIE processor core owns a 32 KB local memory, which consists of eight memory banks. The AIE core can access multiple banks from four ports at the same time if there are no bank conflicts.

3.3 Data Transmission within AIE Array (AIEs↔AIEs)

In terms of data communication between AIEs, there are mainly three kinds of connections: the shared memory, the cascade stream, and the AXI Stream. A tile of data can be transferred between neighboring AIEs through the shared memory in a ping-pong buffer manner with two 256-bit load and one 256-bit store instructions, as mentioned above. On the other hand, the cascade connection provides a 384-bit fine-grained stream access that can transfer data between accumulator registers in neighboring AIEs. Unlike the shared memory, the cascade stream is a single-direction connection alternating between rows. In addition to the neighboring transfer, the data movement between non-neighboring AIEs is through the AXIS stream network, connected by the switch boxes with a 32-bit width per wire. Each AIE core has two input and two output connections from/to the switch. Each switch has six output ports to its north neighbor, thus six input ports from its south neighbor. For the rest of the directions, the switch has four I/O ports with its neighbor.

3.4 Data Transmission between AIE and PL (PL↔AIEs)

Communication between AIE and PL is through the PLIOs in the 39 interface tiles. On the PL domain, each interface tile has eight 64-bit input channels and six 64-bit output channels running at the PL clock frequency. The connections can

also be configured as 128-bit to catch up with the speed of the AIE side, with the number of channels being halved. On the AIE domain, there are eight 32-bit input channels and six 32-bit output channels running at 1 GHz. Inside the AIE array, the ports from the PLIO are also connected throughout the AXI stream mesh through the AXI switches. The AXIS switches can be reconfigured in two ways, i.e., circuit-switched and packet-switched. 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.

3.5 Unveil the Optimization Design Space for Versal Architecture

The heterogeneity of the Versal architecture provides abundant potential optimizations to achieve high performance for deep learning applications. In this section, we will unveil the optimization design space, which motivates the creation of the CHARM framework for automatic mapping. (1) The AIEs in Versal are different from the traditional MAC array composed of DSPs because of their VLIW characteristics. Simply abstracting them by the number of maximum MACs each core could do is no longer enough. Rather, the data locality of local memory as well as the data reuse in the register are of great significance to AIE performance. (2) The high-speed AXIS network is flexible so that more choices of parallelism and dataflow can be explored in the AIE array. For example, many previous works [22, 44, 45] explore the 2D-array parallelism with 1D-SIMD capability, whereas 3D-array parallelism with 3D-SIMD instruction is covered in the CHARM framework. (3) Good orchestration between AIE and PL is required in Versal to sustain the high throughput of the AIE array, which hasn't been systematically explored by other works. Thus, we propose the CHARM framework, which takes an in-depth analysis of the Versal architecture, abstracts an accurate architecture modeling, and designs an automatic framework for deep learning mapping.

4 CHARM SINGLE ACCELERATOR DESIGN

High-throughput and low-latency are two significant evaluation criteria for deploying deep learning models on computing platforms. Many previous solutions including Eyeriss [12, 13], ShiDiannao [14], NPU [15–17] allocated hundreds of MAC arrays to a monolithic design which achieves good efficiency for applications with uniform large layers. In order to achieve high performance for those applications, we describe the dataflow and mapping strategy to sustain the throughput of a single MM acc with hundreds of AIEs in Section 4.1. In Section 4.2, we present our proposed optimization techniques to achieve high single AIE efficiency. Then in Section 4.2 and 4.2, we demonstrate the IO and data reuse to balance the massive computation parallelism and communication among AIEs, 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 basic dense matrix-matrix multiplication. 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 3D-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

```
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: Determine On-Chip SRAM Allocation
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: Determine AIE and PLIO Utilization
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             // Determine AIE Local Mem and VLIW Scheduling
17             for (int i.3=0; i.3<TI; i.3++)
18             for (int j.3=0; j.3<TJ; j.3++)
19             for (int k.3=0; k.3<TK; k.3++)
20                 ...
21                 2D-SIMD(i.3, j.3, k.3);
```

Listing 1. Pseudocode for MM Loop Tiling and Dataflow.

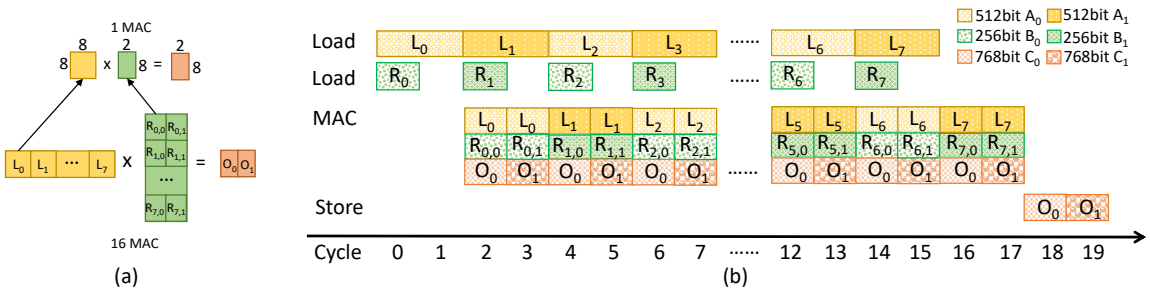


Fig. 5. Single AIE VLIW scheduling.

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 I/Os, 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 is fed into the AIE array (X×Y×Z) times, and the intermediate partial sum from the AIE array is accumulated on the 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 determined by the overall input matrix size (M, K, N).

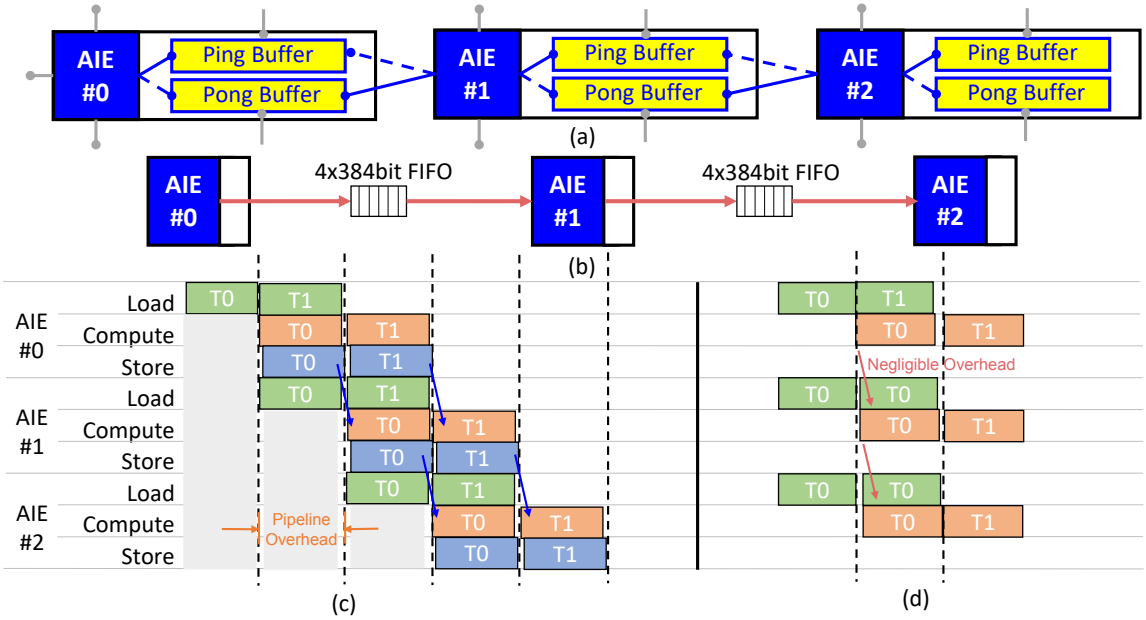


Fig. 6. Data transfer between AIEs: shared memory VS. cascade stream.

4.2 Level 0: Efficient VLIW Scheduling in A Single AIE

Firstly, at the single AIE level, in order to sustain the high throughput of a single AIE, the data orchestration between the AIE local memory and AIE registers under the bandwidth constraints serves as the key point. More specifically, we optimize the single AIE kernel by applying the following techniques: ① We separate the workload of a single AIE ($TI \times TK \times TJ$) into small partitions by **Unrolling Multiple VLIW Instructions** in the innermost loop. ② The **Double Register** technique is utilized to overlap the time spent on register loading with the MAC operations. ③ To balance the computation and communication, **Register Reuse** is mathematically analyzed. ④ We apply the **Output Stationary Dataflow**, meaning that we set k.3 as the innermost loop in the AIE design.

Among the data types supported in the current AIE, due to the high parallelism (128 MACs/Cycle/AIE), the INT8 data type is the most difficult one to balance in terms of computation and communication. Thus, we illustrate the designed VLIW scheduling for INT8 in Figure 5. Note that other data types share a similar methodology but are more prone to design. In this example, we pack 16 $8 \times 8 \times 2$ INT8 MAC operations together to form a small partition with size $8 \times 64 \times 4$. This provides the AIE compiler more opportunities to launch software pipelining and thus is prone to get the back-to-back issued MAC instructions (①). Two 512-bit vector registers A_0 and A_1 are allocated as double registers for the LHS matrix. Each of them is capable of holding 8×8 8-bit LHS operands. B_0 and B_1 represent two 256-bit registers, with each one consisting of two 8×2 8-bit sub-registers denoted by $B_{x,0}$ and $B_{x,1}$ (②). By doing so, at Cycles 2-3 while the MAC operation depends on the data stored in registers A0 and B0, it can pre-load the data to registers A1 and B1 simultaneously for MAC operations that will execute in Cycles 4-5. This register overlapping process comes along with careful consideration for the size of registers and local memory access bandwidth (③). $8 \times 8 \times 2$ MAC operation is manually constructed instead of the original $16 \times 8 \times 1$ one to guarantee that the size of registers is under 2048 bits after double buffering. Registers A_0 and A_1 are both reused twice, which allows their ping or pong buffer to be filled by two

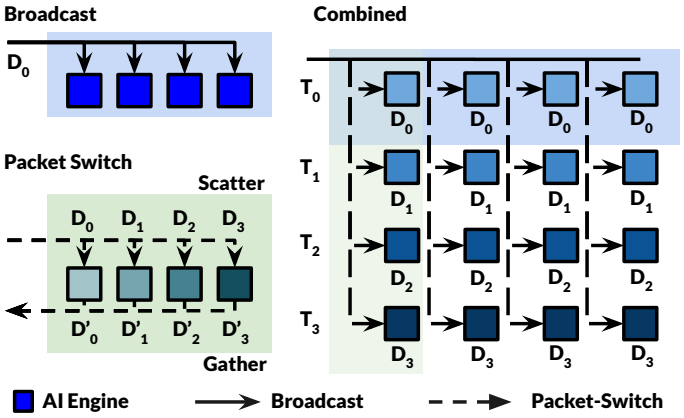


Fig. 7. Combining broadcast circuit-switched and packet-switched connections to reduce required I/O for AIE array.

256-bit load instructions in two cycles. Finally, to avoid frequent register eviction, we only store data from 2 768-bit accumulation registers (C_0 and C_1) back to local memory every 16 cycles, as shown in Cycles 18-19 (④). This hugely reduces the temporary results eviction and reloading between the local memory and registers, thus saving precious load/store bandwidth. During VLIW scheduling, we also explore other dataflows, e.g., LHS/RHS-stationary, by changing the loop permutation. However, because of intermediate results access, these dataflows incur three simultaneous read operations from local memory to the registers, whereas the VLIW is capable of packing only two read operations at the same time, leading to less efficient VLIW packing.

4.3 Level 1: IO Connections Within the AIE Array and Between AIE and PL

Secondly, at the AIE array level, we evenly partition the workloads among $A \times B \times C$ AIEs, which form a rectangular physical layout in the 8×50 AIE array. Shared memory or cascade streams are applied to transmit the output matrices between neighboring AIEs, while the packet-switch and broadcast mechanisms are utilized to transfer the LHS and RHS matrices. **Data Transfer Between AIEs: Shared Memory VS. Cascade Stream (AIE↔AIE).** To leverage the high bandwidth of intra-AIE communications, we explore the data forwarding for the output temporary data in two granularities, i.e., the tile-level shared memory and register-level cascade stream. The graph-level execution models of these two granularities are shown in Figure 6(a) and (b). The AIEs connected by the shared memory apply a ping-pong buffer manner. During processing, while AIE 1 is calculating the data in the ping buffer of AIE 0, AIE 0 is storing the result in its pong buffer. The access pattern alternates during each time step. Because of the mechanism of the ping-pong buffer, the dependency between the adjacent AIEs leads to the tile-level overhead as shown in Figure 6(c). More specifically, AIE 1 can start computing Tile 0 (T0) only after AIE 0 stores all the data of T0 in the local buffer. On the other hand, the cascade stream provides a dedicated 384-bit connection with a four-depth 384-bit FIFO between the neighboring AIEs. In this situation, the first result of the previous AIE triggers the execution of the latter one. By

applying the same execution pattern for adjacent AIEs, we create a fine-grained pipeline without cascade stall and thus lead to negligible overhead as shown in Figure 6(d).

Data Transfer Between AIEs and PL: Packet-Switched & Broadcast (AIE↔PL). At the AIE↔PL level, when feeding data to tens or hundreds of AIEs, since 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 PLIOs by exploring the data broadcast and packet-switch mechanisms (described in Section 3.3). Figure 7 shows how we reuse a single PLIO port by combining broadcast with packet switching. Assume that we have a 4×4 AIE array that calculates an MM of size 1×4×4 (1 MAC/AIE). 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 Times 2 and 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.

4.4 Levels 2 & 3: On-Chip Data Reuse and Off-Chip DDR Access (PL↔DDR)

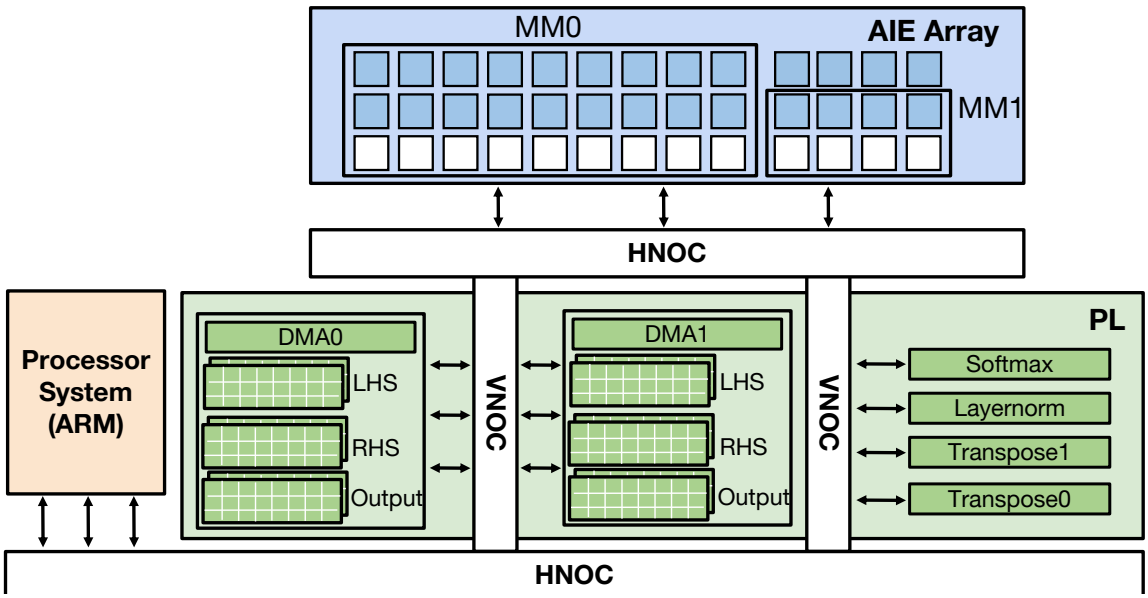
Thirdly, at the PL↔DDR level, we further allocate three sets of on-chip buffers for each acc to store the LHS, the RHS, and the output matrices, so that a tile of LHS with size (X×A×TI) × (Y×B×TK) can be reused on-chip for (Z×C×TJ) times. The buffer size and reuse rate for RHS and output matrices can 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 data reuse opportunities at multiple levels, our system can sustain high computational 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 provide an overview of the CHARM framework in Section 5.2. We then discuss each module within the framework from Section 5.3 to Section 5.5.

5.1 CHARM Architecture

Figure 8 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 an on-chip buffer according to the tiling strategy. The different AIE partitions communicate with their corresponding DMA modules through the PLIO interface and the NOC. We refer to the AIE array, its corresponding PLIO, and the DMA module collectively as one MM acc design. For each non-MM kernel, e.g., transpose, softmax, and layer normalization in BERT and 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, since 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).



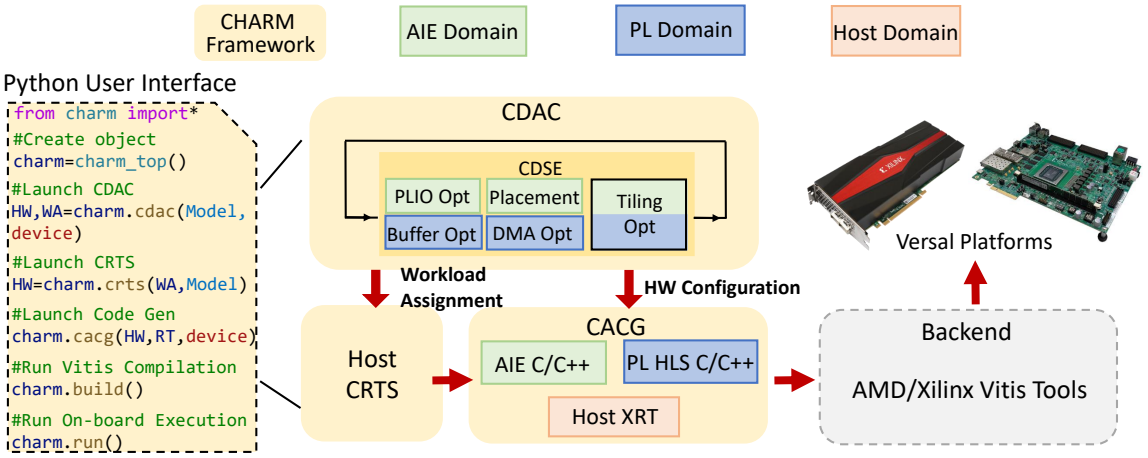


Fig. 9. CHARM framework overview.

5.2 Python-based CHARM Automatic Framework Overview

In order to ease the programming effort, we propose a Python-based automatic CHARM framework shown in Figure 9. The CHARM framework takes the application models, 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 workload assignment strategy as well as the configurable design parameters of each acc. During this process, CDSE optimizes the placement of the local memory, AIE core, and PLIO ports. It also covers the AIE array tiling and PLIO optimization on the AIE side. On the PL side, it determines the tiling, buffer allocation, and DMA optimization. (2) CRTS takes the layer dependency graph and the workload assignment strategy from CDAC as inputs. It provides the scheduling strategy of the layers in the application onto the available accs. (3) CACG takes the configurable parameters generated from CDAC and the runtime scheduling from CRTS as input. Based on the CDAC and CRTS, it 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. All the components of the CHARM framework are abstracted into Python APIs as shown in the Python user interface, which greatly reduces the time for on-board deployment on Versal ACAP.

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

DSE Configurable parameters: A, B, C, X, Y, Z in Listing 1. 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 a hardware platform off-chip characterization database as inputs and performs an analytical model-based search. During CDSE, we set the single AIE workloads to 32×32×32, 48×48×48, and 64×64×64 for FP32, INT16, and INT8 respectively, i.e., TI=TK=TJ=32, 48, or 64. We achieve over 90% of kernel efficiency for MM, utilize over 75% of the AIE local memory in this design point, and obtain the CTC ratios of 4, 3, and 2 in FP32, INT16, and INT8 data types, respectively. 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 numbers of AIE and PLIO used in the AIE array. X, Y, Z, A, B, C together with pre-fixed 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 as shown below. AIE_{num}, $PLIO_{in}$, $PLIO_{out}$ and On_chip_{RAM} represent the user-specified hardware constraints:

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

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

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

$$Buff \leq On\_chip_{RAM} \tag{4}$$

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 is described in Section 4.2. They should meet the input and output PLIO resource constraints. The input and output PLIO numbers can be obtained as follows:

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

PL Tiling Selection. On-chip PL buffers are allocated to amortize the 46× bandwidth gap from off-chip to PL and from PL to AIE-Array by increasing the data reuse rate. Equation 6 shows the sizes of the LHS, RHS, and output buffers,

as well as their off-chip to on-chip communication times. BPD refers to bytes per data, and BW_L,R,O represents the off-chip bandwidths measured from bandwidth profiling.

$$\begin{aligned} Buff_L &= (X \cdot A \cdot TI) \cdot (Y \cdot B \cdot TK) \cdot BPD \\ Buff_R &= Y \cdot Z \cdot B \cdot C \cdot TK \cdot TJ \cdot BPD \\ Buff_O &= X \cdot Z \cdot A \cdot C \cdot TI \cdot TJ \cdot BPD \\ Buff &= 2 \cdot (Buff_L + Buff_R + Buff_O) \\ Time_{L,R,O} &= Buff_{L,R,O} / BW_{L,R,O} \end{aligned} \tag{6}$$

Performance Modeling. To calculate the overall execution time, the scheduling of data communication from 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 theoretical MAC operation that one AIE engine can perform in one cycle, and Eff refers to the actual efficiency that the computation kernel achieves. We consider both single AIE and AIE array pipeline efficiency (PL↔AIE) here and assign the overall efficiency as 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.

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

$$TIME = max([Time_L, Time_R, Time\_comp]) \cdot (TX \cdot TY \cdot TZ) \tag{8}$$

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 optimized overall throughput when mapping diverse sizes of MM kernels onto 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 into multiple groups. The number of groups equals the number of diverse accs, which is a hyperparameter in CDAC. After the workload partitioning, 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 resources, we search for 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, redo the resource partition, and perform a configurable parameter search, which further optimizes the system throughput for 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 ensure they work concurrently with similar execution times. However, mapping an application with **n** kernels to **num** accs suffers from 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

Algorithm 1 Diverse Accelerator Composing Algorithm

Input: layer[n], bw, hw_sr, num, ubound

▷ **layer[n]** represents 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** contain 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($C\binom{n-1}{num-1}$) **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 $Acc[:], Cycle[:]$

sort-based algorithm to partition the workload with reduced time complexity as $O(\binom{n-1}{\text{num}-1}) = C^{n-1}_{\text{num}-1}$. 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 $C\binom{8-1}{2-1} = 7$ grouping design choices.

2nd Step: Hardware Resource Partitioning. For each workload assignment, we perform DSE to find the optimized configurable acc parameters under the partitioned hardware resource constraints, 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 each 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 allocations of others (Lines 13-14). If a better result is found, we update the global optimal execution cycles and the corresponding acc configuration settings (Lines 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 off-chip bandwidth partitioning for future work.

5.5 CHARM Runtime Scheduler (CRTS)

While the two-step CDAC algorithm provides the solution for workload assignment and hardware partitioning, the CHARM runtime scheduler (CDAC) is proposed to resolve the dependencies among the layers in the application graphs. By exploring the task-level parallelism, the accelerators are able to achieve high throughput simultaneously. In this section, we first mathematically formulate the mixed-integer-programming (MIP) based problem, which provides the optimal solutions. Then, to improve the search efficiency, we explain a scalable greedy scheduling algorithm.

Mixed-Integer-Programming Formulation. Based on the workload assignment and hardware partitioning strategy, CDAC provides the design with maximum throughput, considering there is no dependency in the application graph. Thus, batch-level pipelining and the runtime scheduler are proposed to achieve optimized throughput and resolve the dependency. We mathematically formulate the problem by MIP, as shown in Equations 9–13, which serves as the oracle solution. In the MIP formulation, we consider the following factors and variables:

- num_bat refers to the number of batches needed to sustain the pipeline, which will be exhaustively searched from 1 to the user-specified upper bound.
- G refers to the application graph, which consists of all the layers.
- $D_{n,m}$ is the binary dependency matrix where $D_{n,m} = 1$ means that layer m depends on layer n.
- ACC refers to all the accelerators in the system.
- $A_{n,acc}$ is the assignment strategy of each layer on every accelerator.
- $T_{n,acc}$ is the execution time matrix of each layer on every accelerator.
- ST_n denotes the start time of each layer.
- ET_n denotes the end time of each layer.

We maximize the throughput of the system under a certain batch by calculating the maximum end time (T_{max}) of all the layers, as illustrated in Equations 9 and 11. ET_n can be formulated by adding the start time ST_n to the corresponding execution time $T_{n,acc} \times A_{n,acc}$, as shown in Equation 10. The dependency among the layers in the graph is maintained during execution by Equation 12. Equation 13 guarantees that at each time, one accelerator will only process one layer in the graph.

$$maximize\ num\_bat/T_{max} \tag{9}$$

$$s.t.\ ET_n = ST_n + T_{n,acc} \times A_{n,acc},\ \forall n \in (G), \forall acc \in (ACC) \tag{10}$$

$$T_{max} = \max(ET_n),\ \forall n \in (G) \tag{11}$$

$$ET_m \geq ST_n,\ D_{n,m} = 1, \forall (n) \in G, \forall (m) \in G \tag{12}$$

$$\begin{gathered} ET_m \geq ST_n\ or\ ET_n \geq ST_m \\ D_{n,m} = 0, A_{m,acc} = A_{n,acc},\ \forall (n) \in G, \forall (m) \in G, \forall acc \in ACC \end{gathered} \tag{13}$$

Scalable Greedy Algorithm. The proposed MIP-based formulation provides the optimal scheduling solution under the workload assignment strategy and the dependency constraints. However, the large design space also leads to a very long search time. We then propose a greedy-algorithm-based fast runtime scheduler that can resolve the dependencies while achieving high throughput under a different number of batches. Algorithm 2 lists the proposed scheduling algorithm. It takes the dependency graph, the number of accelerators, and the layer assignment configuration file generated by CDAC as input. There are two parallel processes in the greedy CRTS.

The first process continuously tracks to check if there are any idle acc to which we can assign tasks (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 layers 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 changes the status of the acc (Line 14) to idle.

Algorithm 2 Runtime Scheduling 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 ¬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] ≠ ∅ ∧ task_pool[t][l].valid() then
8:                      Acc[acc].assign(task_pool[t][l])
9:                      Continue line 2
10: while (1) do                                   ▷ Update the task_pool according to the 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)
```

5.6 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 factors of a single AIE (TI, TK, TJ) and an AIE Array (A, B, C) as inputs and instantiates the corresponding number of AIE cores. It leverages the C++-based Adaptive Data Flow (ADF) Graph API [46] to build connections among AIE cores through the AXI network and connections between the 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 transfer modules for sending/receiving data to/from the AIE array. HostGen emits host code based on the Xilinx Runtime Library (XRT) API.

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 into a host program that runs on the ARM CPU for kernel scheduling and system controls. In terms of usability, the hardware configurations can be automatically handled by code generation and compilation. We also provide separate Python APIs as shown in Figure 9 for (1) launching DSE, (2) Jinja2-template-based code generation, and (3) backend compilation, so that users have more flexibility in applying our frameworks.

6 EXPERIMENT RESULTS

In this section, we first analyze the single AIE efficiency and the single MM acc throughput from Section 6.1 to Section 6.3. In Section 6.5, 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 were conducted on the VCK190 with 230 MHz on the PL and 1 GHz on the 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 [47], and the AMD/Xilinx Board Utility tool, Xbutil [48].

6.1 Single AIE Kernel Efficiency Comparison

In this section, we showcase the efficiency of our single AIE MM computation in different matrix sizes for FP32, INT16, and INT8. We leverage the AIE intrinsics [49] to program the single kernel design and obtain the execution cycle of our single AIE design by simulating it on the Versal ACAP AI Engine System C simulator [50], 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 the MM size equals 32×32×32. Compared to the AIE dense MM kernel efficiency reported in H-GCN [51], our single kernel achieves a 2.26× average efficiency gain for FP32.

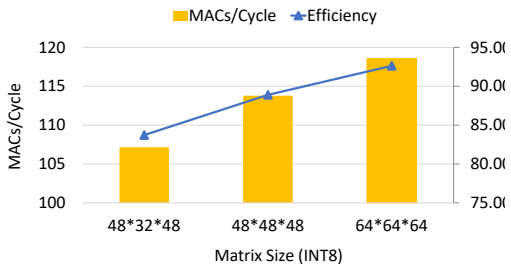
We test multiple combinations of matrix sizes for INT16 and INT8 and list three that balance each dimension well in Figure 10 and Figure 11. In INT8 and INT16 data types, CHARM achieves up to 92.62% and 98.04% single AIE efficiency. For the whole system design, we choose 32×32×32, 48×48×48, 64×64×64 as our single kernel design for FP32, INT16, and INT8 data types respectively, as these shapes achieve 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. CHARM achieves high single AIE efficiency in all the presented data types by applying the optimization techniques described in Section 4.2. More specifically, CHARM takes into consideration, (1) the different ways of constructing the 3D-SIMD MAC instructions, (2) the data in AIE core reuse with a limited number of vector registers, (3) the data reuse and layout of the AIE local memory. Finally, for the example in Section 4.2, according to the profiling results from the AIE compiler, the manually unrolled 16 8×8×2 INT8 MAC operations in the innermost loop can be done in 16 cycles, which means it's perfectly pipelined.

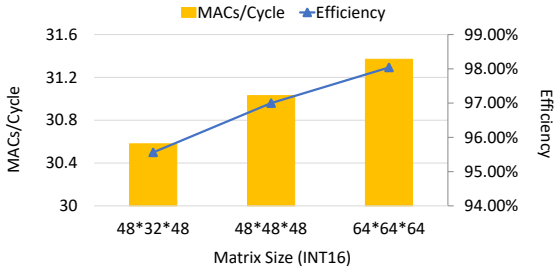
6.2 Performance of Square MMs on One Monolithic Accelerator

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 1 GHz and the PL side at 230 MHz. As shown in Figure 12, the throughput of the one-acc monolithic design increases as the square MM size increases. While it achieves 4.2 TFLOPS at size 6144, the

Table 2. Single AIE MM comparison under FP32 data type.

Size: M x K x N	H-GCN[51] MACs/Cyc	H-GCN[51] Eff	CHARM (this work) MACs/Cyc	CHARM (this work) Eff	CHARM (this work) Eff gain
$16 \times 16 \times 16$	2.34	29.30%	6.18	77.22%	***2.64x***
$32 \times 32 \times 32$	3.64	45.50%	7.57	94.70%	***2.08x***





pe.

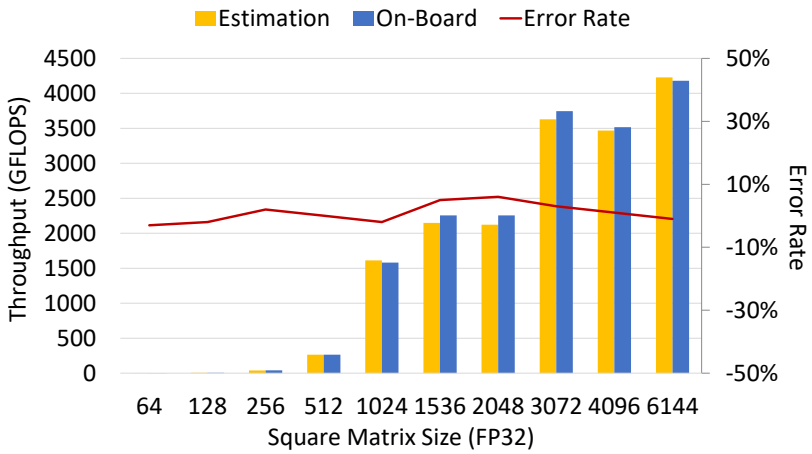


Fig. 12. Performance comparison in GFLOPS between on-board measurements and CDSE analytical modeling estimations for different matrix sizes. The error rates, shown in percentages, indicate that CDSE achieves high prediction accuracy.

throughput at size 64 is only 0.65 GFLOPS. CHARM CDSE is capable of precisely estimating the on-board execution time with an average estimation error rate of only 2.6%.

We compare the throughput of the MM application implemented on AMD U250 FPGA using the state-of-the-art systolic-array-based framework AutoSA [22] for FP32, INT16, and INT8 data types. We reimplement the design with the best configurations reported in the paper and measure the corresponding power through the AMD/Xilinx Xbutil tool. While AutoSA runs at 300 MHz, 250 MHz, and 300 MHz for FP32, INT16, and INT8 data types, the PL and AIE of CHARM run at 230 MHz and 1 GHz in all data types. As shown in Table 3, the proposed CHARM on Versal VCK190

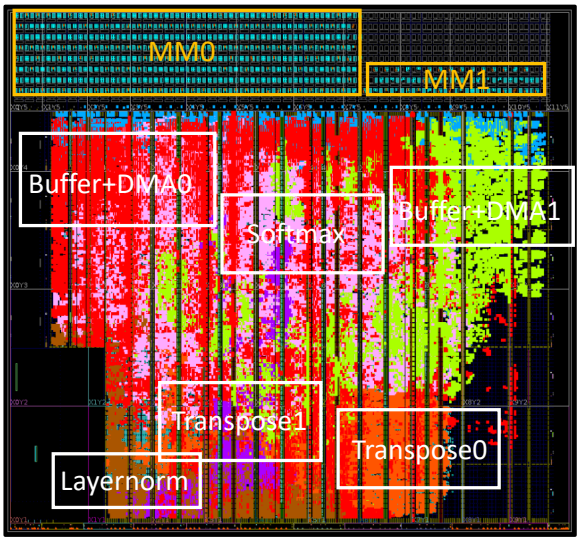


Fig. 13. System implementation layout of the two-diverse MM accs and four non-MM accs for BERT.

Table 3. MM performance and energy efficiency comparison between AutoSA and CHARM.

Data Type	AutoSA [22] Float32	AutoSA [22] INT16	AutoSA [22] INT8	CHARM(Ours) Float32	CHARM(Ours) INT16	CHARM(Ours) INT8
Frequency(Hz)	300M	300M	300M	PL:230M/AIE:1G	PL:230M/AIE:1G	PL:230M/AIE:1G
DSP/AIE	DSP:8333	DSP:8205	DSP:6157	AIE:384	AIE:288	AIE:192
GOPS	930	3410	6950	**4179 (4.5x)**	**10034 (2.9x)**	**31307 (4.5x)**
Power(W)	96.8	84.9	91.2	61.9	64.8	62.7
Eng.Eff(GOPS/W)	9.6	40.2	76.2	**67.9 (7.1x)**	**154.3 (3.8x)**	**499.2 (6.6x)**

Table 4. Performance comparison between shared memory and cascade stream connections in AIE array under FP32, INT16, and INT8 data types.

Data Type	# of AIEs	Matrix Size	Shared Memory	Cascade Stream	Performance Gain
FP32	384	6k*6k*6k	3.4 TFLOPS	**4.2 TFLOPS**	**1.23x**
INT16	288	6.75k*9k*9k	7.5 TOPS	**10.0 TOPS**	**1.33x**
INT8	192	12k*12k*12k	28.2 TOPS	**31.3 TOPS**	**1.11x**

single MM acc achieves 4179 GFLOPS, 10034 GOPS, and 31307 GOPS for FP32, INT16 and INT8, which is 4.5×, 2.9×, and 4.5× faster than AutoSA on U250. In terms of energy efficiency, CHARM achieves 7.1×, 3.8×, and 6.6× gains respectively.

6.3 Performance Comparison Between Shared Memory and Cascade Stream Connections Within an AIE Array

We compare the shared memory with the cascade stream connections when forwarding the output matrix within the AIE array in different data types including FP32, INT16, and INT8. The number of AIEs and size of the matrices are set to

Table 5. Lines of code comparison for a matrix-multiple example between using CHARM framework and expert manual design.

CHARM APIs	ADF Graph	AIE Intrinsic	PL HLS	XRT Host
16 LoC	800+	600+	1000+	350+

the same configuration in each data type as shown in Table 4. As analyzed in Section 4.2, by applying the register-level fine-grained data transmission manner, i.e., cascade stream, it achieves higher throughput due to the smaller overhead. The designs with cascade stream achieve 1.23×, 1.33×, and 1.11× gains on throughput over the one utilizing shared memory as the intra-AIE connection. Note that the listed results don't indicate that the cascade stream will always outperform the shared memory connection for any applications. It is because of our careful design on a single AIE for MM that makes the behavior of different AIEs remain almost the same. Thus, our design avoids the cascade FIFO stall and achieves high efficiency.

6.4 Programming Abstraction Analysis of the CHARM Framework

As described in Section 5.6, deploying matrix-multiply based applications on Versal Architecture involves abundant programming effort to optimize C++-based intrinsics and ADF graph APIs for AIEs, HLS C/C++ code for PL, and XRT API-based host code for ARM CPU. The CHARM framework provides architecture abstraction for accurately modeling Versal, automatic DSE for application mapping, and Jinja2-template-based code generation. CHARM helps to relieve the burden on domain-specific experts in other areas for accelerating their problems on Versal. Take a simple matrix-multiply in FP32 as an example. As shown in Table 5, by using the APIs for CDAC, CDSE, CRTS, and CACG, users only need to write 16 lines of code (LoC), so that the CHARM framework can automatically generate an overall 170× optimized source code for the ARM host, PL and AIEs.

6.5 End-to-End Applications

We apply the CHARM framework to four applications: BERT, ViT, NCF, and MLP. All the shapes of the MM kernels in these models are listed in Table 6. 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 remains 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 the designs of the same application use the same non-MM kernels. Table 8 reports the on-board throughput and power consumption in 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, and MLP in the FP32 data type. Table 7 shows the time breakdown for MM, layernorm, softmax, and transpose in the FP32 data type for each end-to-end application. We highlight the best design(s) for each application in Table 8. 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 both BERT and ViT have both large and small MMs, whereas MLP has only large MMs. NCF also has both large and small MMs. However, small MMs with size less than $3072 \times 256 \times 128$ consume less than 0.4% 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

Table 6. 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	3072	N/A
	1	3072	1024	1024	N/A
	1	3072	1024	4096	N/A
	1	3072	4096	1024	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 7. 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	119ms
Layernorm	4.5ms	4.5ms	0	0
Softmax	18.7ms	2.3ms	0	0
Transpose	5.2ms	5.2ms	0	0

monolithic design, the customization of using one specialized acc design for a specific application provided by CHARM gives 2.13× and 5.08× gains in energy efficiency (GFLOPS/W), respectively. The additional design spaces explored by

Table 8. On-board throughput and power comparisons of different MM accs configurations for BERT, ViT, NCF, MLP under FP32 data type.

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_spe	***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 9. 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%)

Table 10. On-board throughput and power comparisons of different MM accs configurations for BERT, ViT, NCF, MLP under INT16 data type.

App	CHARM Cfg	LUT	BRAM	URAM	DSP	AIE	GOPS	Power(W)	GOPS/W (Ratio)
BERT	One_mono	111626(12.41%)	885 (91.52%)	384 (82.94%)	91 (4.62%)	288 (72%)	215.0	31.4	6.9 (1x)
	One_spe	98694(10.97%)	237 (24.51%)	336 (72.57%)	101 (5.13%)	216 (54%)	883.7	30.7	28.8 (4.2x)
	Two_diverse	***75451(8.38%)***	***602 (62.25%)***	***448 (96.76%)***	***160 (8.13%)***	***120 (30%)***	***1913.1***	***32.0***	***59.8 (8.7x)***
	8_duplicate	198572(22.07%)	872 (90.18%)	384 (82.94%)	744 (37.80%)	192 (48%)	1371.6	35.1	39.1 (5.7x)
ViT	One_mono	111626(12.41%)	885 (91.52%)	384 (82.94%)	91 (4.62%)	288 (72%)	34.2	29.3	1.2 (1x)
	One_spe	43928(4.88%)	429 (44.36%)	0 (00.00%)	77 (3.91%)	120 (30%)	461.0	22.5	20.5 (17.6x)
	Two_diverse	***72205(8.02%)***	***602 (62.25%)***	***416 (89.85%)***	***160 (8.13%)***	***108 (27%)***	***1180.1***	***26.6***	***44.4 (38.0x)***
	8_duplicate	198572(22.07%)	872 (90.18%)	384 (82.94%)	744 (37.80%)	192 (48%)	747.6	32.8	22.8 (19.6x)
NCF	One_mono	111626(12.41%)	885 (91.52%)	384 (82.94%)	91 (4.62%)	288 (72%)	1638.3	40.4	40.5 (1x)
	One_spe	125337(13.93%)	481 (49.74%)	384 (82.94%)	85 (4.32%)	288 (72%)	2775.4	34.7	80.0 (2.0x)
	Two_diverse	***126478(14.06%)***	***826 (85.42%)***	***384 (82.94%)***	***162 (8.23%)***	***288 (72%)***	***4056.9***	***47.1***	***86.2 (2.1x)***
	8_duplicate	198572(22.07%)	872 (90.18%)	384 (82.94%)	744 (37.80%)	192 (48%)	2432.9	38.4	63.4 (1.6x)
MLP	One_mono	111626(12.41%)	885 (91.52%)	384 (82.94%)	91 (4.62%)	288 (72%)	4855.5	53.0	91.6 (1x)
	One_spe	***127847(14.21%)***	***481 (49.74%)***	***384 (82.94%)***	***85 (4.32%)***	***288 (72%)***	***5807.6***	***52.0***	***111.7 (1.2x)***
	Two_diverse	126478(14.06%)	826 (85.42%)	384 (82.94%)	162 (8.23%)	288 (72%)	5159.1	50.8	101.5 (1.1x)
	8_duplicate	198572(22.07%)	872 (90.18%)	384 (82.94%)	744 (37.80%)	192 (48%)	2738.3	35.2	69.8 (0.8x)

using more than one-acc, with heterogeneous and diverse-shaped accs provided by the CHARM framework, give us 2.25× and 5.24× extra energy efficiency gains for BERT and ViT, respectively.

The resource utilization, performance, and energy efficiency results for INT16 and INT8 data types are demonstrated in Tables 10 and 11. CHARM achieves maximum throughputs of 1.91 TOPS, 1.18 TOPS, 4.06 TOPS, and 5.81 TOPS in

Table 11. On-board throughput and power comparisons of different MM accs configurations for BERT, ViT, NCF, MLP under INT8 data type.

App	CHARM Cfg	LUT	BRAM	URAM	DSP	AIE	GOPS	Power(W)	GOPS/W (Ratio)
BERT	One_mono	115628(12.85%)	662 (68.46%)	388 (83.80%)	71 (3.61%)	192 (48%)	480.3	33.3	14.4 (1x)
	One_spe	80277(8.92%)	813 (84.07%)	64 (13.82%)	55 (2.79%)	128 (32%)	2369.2	26.0	91.1 (6.3x)
	Two_diverse	***104609(11.63%)***	***730 (75.49%)***	***240 (51.84%)***	***160 (8.13%)***	***120 (30%)***	***3649.2***	***28.0***	***130.2 (9.0x)***
	8_duplicate	161336(17.93%)	952 (98.45%)	320 (69.11%)	536 (27.24%)	96 (24%)	2292.2	29.0	79.0 (5.5x)
ViT	One_mono	115628(12.85%)	662 (68.46%)	388 (83.80%)	71 (3.61%)	192 (48%)	75.8	30.3	2.5 (1x)
	One_spe	51685(5.74%)	557 (57.60%)	0 (00.00%)	55 (2.79%)	64 (16%)	696.0	20.3	34.2 (13.7x)
	Two_diverse	***95326(10.59%)***	***634 (65.56%)***	***216 (46.65%)***	***100 (5.08%)***	***108 (27%)***	***1278.8***	***22.5***	***56.9 (22.8x)***
	8_duplicate	161336(17.93%)	952 (98.45%)	320 (69.11%)	536 (27.24%)	96 (24%)	942.7	27.0	22.8 (34.9x)
NCF	One_mono	115628(12.85%)	662 (68.46%)	388 (83.80%)	71 (3.61%)	192 (48%)	4062.7	41.0	99.1 (1x)
	One_spe	108102(12.01%)	813 (84.07%)	256 (55.29%)	67 (3.40%)	192 (48%)	4392.3	32.4	135.5 (1.4x)
	Two_diverse	***120120(13.35%)***	***858 (88.73%)***	***384 (82.94%)***	***122 (6.20%)***	***160 (40%)***	***10187.9***	***39.9***	***255.1 (2.6x)***
	8_duplicate	161336(17.93%)	952 (98.45%)	320 (69.11%)	536 (27.24%)	96 (24%)	4203.9	30.7	136.8 (1.4x)
MLP	One_mono	115628(12.85%)	662 (68.46%)	388 (83.80%)	71 (3.61%)	192 (48%)	11283.4	54.7	206.1 (1x)
	One_spe	***116320(12.93%)***	***669 (69.18%)***	***384 (82.94%)***	***67 (3.40%)***	***192 (48%)***	***21579.1***	***53.9***	***400.5 (1.9x)***
	Two_diverse	119449(13.27%)	826 (85.42%)	416 (89.85%)	134 (6.81%)	176 (44%)	19319.0	48.5	398.4 (1.9x)
	8_duplicate	161336(17.93%)	952 (98.45%)	320 (69.11%)	536 (27.24%)	96 (24%)	4799.5	31.0	154.7 (0.8x)

the INT16 data type for the four applications. The maximum throughput achieved by CHARM in the INT8 data type is 3.65 TOPS, 1.28 TOPS, 10.19 TOPS, and 21.58 TOPS, respectively. In these data types, the two diverse MM ACCs designs are the best for BERT, ViT, and NCF applications. The one specialized design is the best configuration for the MLP application. Take the INT8 data type as an example. Compared with the monolithic design, our two-diverse acc designs provided by CHARM have 7.6× and 16.9× improvements in throughput and 9.0× and 22.8× improvements in energy efficiency for BERT and ViT applications, respectively, because of the dedicated optimization for small and large layers. The specialized accelerator works best for the MLP application, providing 1.9× gains in both throughput and energy efficiency. The results of these three applications share the same trend as the FP32 data type. For the NCF application, the higher parallelism of INT8 data types requires a much larger tile size, e.g., 3072×256×2048, to sustain the throughput of the system under the same off-chip bandwidth. Thus, it adds padding to all layers except the first layer, which has a size of 3072×4096×2048. By designing two customized accs for the first layer and the other layers, and by carefully overlapping the execution of these two groups, the diverse acc design outperforms the monolithic design by 2.7x and 2.6x in throughput and energy efficiency, respectively. The experiments with the INT16 data type show similar trends. The two diverse design is the best configuration among all the configurations for BERT, ViT, and NCF applications for the INT16 data type, which achieves 8.7×, 38.0×, and 2.1× energy efficiency gain compared with the one monolithic design. The one specialized acc design stands out for the MLP application, which has a 1.2× gain in energy efficiency. **These gains demonstrate the generality of CHARM in applying to different data types when composing heterogeneous accelerators.**

We show the implementation layout of the two-diverse MM acc design in the FP32 data type, i.e., the best design for BERT, in Figure 13. This is also the layout corresponding to Figure 8, which contains two MM accs and four non-MM communication-bound accs. The hardware resource utilization for each acc is reported in Table 9. 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.

CHARM DSE Efficiency. We use CHARM to perform a sort-based two-step search algorithm in CDAC and depict design throughput compared to the exhaustive search over the search iteration number in Figure 14. For BERT, compared

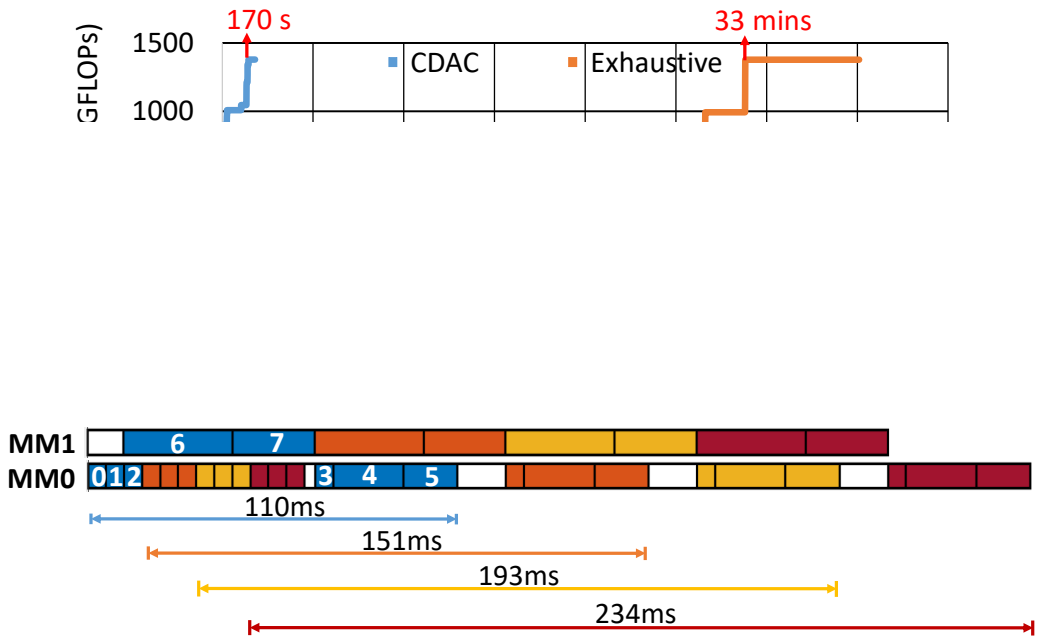


Fig. 15. Timeline of four tasks scheduled on 2 accs for BERT.

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.

Explore the Latency-Throughput Tradeoff in CHARM. As shown in Figure 15, we map four concurrent tasks onto 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 110 ms to finish the 1st task and 234 ms to finish the 4th task. For a one-acc specialized design, the latency of each task is 162.6 ms. Therefore, we have a design tradeoff, i.e., one specialized acc design can process fine-grained tasks, whereas a two-diverse accs design requires coarse-grained tasks to fill the pipeline of the two accs. Compared to the specialized acc design, with 0.67×, 0.92×, 1.18×, and 1.43× latency for different tasks, we gain 2.8× overall throughput in return. This illustrates that the CHARM framework allows explorations of the latency-throughput tradeoff, and users can specify targets to let CHARM generate designs that optimize throughput while meeting the latency requirement, or vice versa. One thing to mention is that the CRTS phase can also be combined with the CDAC phase for better latency but similar throughput at the cost of more search time. In this framework, we propose a solution that targets throughput-oriented scenarios first, assuming an infinite number of tasks to be processed. Therefore, in the CDAC phase, we optimize the maximum total execution time of multiple layers assigned to the corresponding accelerator, leading to nearly the same execution time on each accelerator. In the CRTS phase, with enough tasks, the pipeline in Figure 15 can be fully filled. The throughput under different numbers of tasks are shown in Table 12.

CHARM RTS Efficiency. We compare the optimized results and search time of two proposed runtime scheduling algorithms, including the MIP and greedy algorithms, under different tasks shown in Table 12. We implement the two scheduling algorithms on an Intel Xeon Gold 6244 CPU using Gurobi 10.0 [52] and Python 3.7. In this experiment,

Table 12. Runtime scheduling search time comparison between MIP and greedy algorithm.

# of Tasks	1	2	3	4	5	6	7	8
MIP	0.32s	0.81s	1.85s	3.72s	24.35s	638.97s	>1000s	>1000s
Greedy	0.31ms	0.39ms	0.49ms	0.64ms	0.82ms	0.95ms	1.10ms	1.26ms
GFLOPS	790.2	1039.9	1162.4	1235.1	1283.2	1317.5	1343.1	1362.9

the number of tasks is explored from 1 to 8. The greedy-algorithm-based scheduler finds the solution with the same optimal value provided by MIP formulation under all these eight cases. However, because of the design space pruning, the proposed greedy-algorithm-based scheduler requires much less search time. For example, when the task size equals eight, we reduce the search time from >1000 s to 1.26 ms, which has a $7.9\times10^5\times$ reduction. The gain in search time will be more obvious as the search space grows.

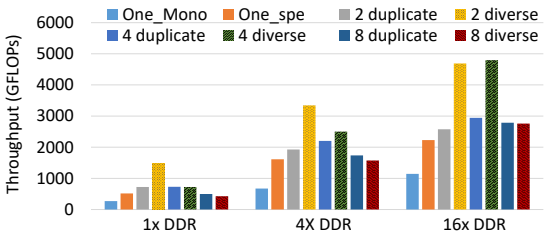


Fig. 16. Throughput comparison under different off-chip bandwidth configurations from CHARM for BERT.

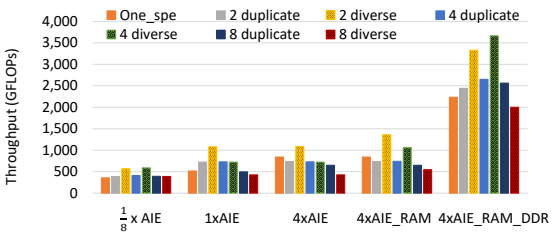


Fig. 17. Throughput comparison under different AIE, local storage, and off-chip configurations from CHARM for BERT.

By leveraging the strong modeling capabilities provided by the CHARM framework, we explore performance under different hardware architecture changes (number of AIEs, on-chip storage size, off-chip bandwidth) to conduct pre-silicon architecture explorations and provide architectural 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 understand the following research questions:

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

A1: Yes. Versal would benefit from a 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 16, 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 at 4.8 TFLOPS, which is constrained by single kernel computation efficiency (95%) and PL↔AIE efficiency (85%). Another observation from Figure 16 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 17 implies that as computation and communication parallelism increase further in the future, there will be 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, the on-chip RAM, and the 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 the PL is equipped with DSPs and has about 1/8 of the theoretical FP32 peak performance of the Versal ACAP. As shown in the first group in Figure 17, 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 massive parallelism and the small MM size is mitigated. On the other hand, as shown in the last group in Figure 17, 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 yield significant improvement, whereas increasing all the resources as a whole does.

Q3: Can CHARM be ported to other architectures?

A3: We can port this to other similar architectures that have a large number of tensor cores (hundreds), where each tensor core requires high parallelism (matrix-matrix computation) to sustain the throughput.

The GPU V100 has 640 tensor cores, operating at 1.245 GHz. Each tensor core can achieve high efficiency only when the matrix size exceeds 32×32×32. When the matrix size is 128×128×128, using 4×4×4 tensor cores (#tensor cores utilization = 10%), each computes 32×32×32 (each core utilization = 100%), or using 8×8×8 tensor cores (# tensor cores utilization = 80%), each computing 16×16×16 (each core utilization = 12.5%), when the matrix size is small, the overall utilization is 10%, which is less than 100%. Only when the matrix is $256 \times 256 \times 256$, the overall utilization (# tensor cores utilization × each tensor core utilization) is near 100%. However, the shape of multi-head attention in the transformer model is far less than $256 \times 256 \times 256$. When mapping such models onto a similar architecture, we can use the CHARM methodology to create diverse accelerators; that is, map multiple MMs and use partial resources for each MM, thereby increasing the overall utilization. When mapping the Transformers onto a GPU V100, if we have a micro-architecture feature that allows us to partition the GPU tensor cores into groups, we can potentially achieve a 1.5×–2× throughput gain when using 4 diverse accelerators compared to 1 accelerator, which is now what the GPU programming model is.

Q4: Is it possible to implement Softmax in the AIE array? Can FlashAttention [53, 54] be enabled to reduce data movement?

A4: The optimization in FlashAttention fuses Softmax with consecutive matrix multiplies to reduce off-chip access and thus is capable of improving our overall latency. We have two ways to enable fused MM+softmax: (1) Fuse Softmax in AIE with MM; (2) Fuse Softmax in FPGA using fine-grained with AIE. When using (1), it incurs 30% extra cycles. When using (2), by using some LUTs in the PL part, there are no extra cycles.

8 CONCLUSION

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 MM-based applications including BERT, ViT, NCF, and MLP in FP32, INT16, and INT8 data types. CHARM is capable of automatically generating high-throughput and energy-efficient end-to-end application solutions. Our experiments show that CHARM achieves 1.46 TFLOPS, 1.61 TFLOPS, 1.74 TFLOPS, and 2.94 TFLOPS inference throughput for BERT, ViT, NCF, and MLP in FP32 data type. in INT16 data type, CHARM has the maximum throughput of 1.91 TOPS, 1.18 TOPS, 4.06 TOPS, and 5.81 TOPS for the four

applications. The maximum throughput achieved by CHARM in the INT8 data type is 3.65 TOPS, 1.28 TOPS, 10.19 TOPS, and 21.58 TOPS respectively.

9 ACKNOWLEDGEMENT AND DISCLOSURE

We acknowledge the support from NSF awards #2213701, #2217003, #2133267, #2122320, #2324864, #2328972, the support from CRISP, one of six SRC JUMP centers, and the support from the CDSC industry partners (cdsc.ucla.edu). 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 Pittsburgh Supercomputing Center. Jason Cong has a financial interest in AMD.

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] Jason Cong and Jie Wang. Polysa: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.
[45] Suhail Basalama, Jie Wang, and Jason Cong. A comprehensive automated exploration framework for systolic array designs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.
[46] AMD/Xilinx. Adaptive Data Flow API.
[47] AMD/Xilinx. Board evaluation and management Tool.
[48] AMD/Xilinx. Xilinx Board Utility (Xbutil).
[49] AMD/Xilinx. AI Engine API and Intrinsics User Guide.
[50] AMD/Xilinx. Versal™ ACAP AI Engine System C simulator.
[51] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Lit, and Dingwen Tao. H-GCN: A graph convolutional network accelerator on versal acap architecture. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 200–208. IEEE, 2022.
[52] Gurobi. Website. https://www.gurobi.com/.
[53] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.
[54] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

Received xxxx; revised xxxx; accepted xxxx