HeteroSVD: Efficient SVD Accelerator on Versal ACAP with Algorithm-Hardware Co-Design

Xinya Luan[1], Zhe Lin[2†], Kai Shi[1], Jianwang Zhai[1], and Kang Zhao[1†],
[1]Beijing University of Posts and Telecommunications [2]Sun Yat-sen University
{luanxinya, shikai, zhaijw, zhaokang}@bupt.edu.cn, linzh235@mail.sysu.edu.cn

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11132878

***Abstract*—Singular value decomposition (SVD) is a matrix factorization technique widely used in signal processing and recommendation systems, etc. In general, the time complexity of SVD algorithms is cubic to the problem size, making SVD algorithms difficult to meet stringent performance requirements in real-time. However, existing FPGA and GPU solutions fall short of jointly optimizing latency, throughput, and power consumption. To settle this issue, this paper proposes HeteroSVD, a heterogeneous reconfigurable accelerator for SVD computation on the Versal ACAP platform. HeteroSVD introduces a system-level SVD decomposition mechanism and proposes an algorithm-hardware co-design method to optimize SVD ordering jointly and AI engine (AIE)-centric dataflow and placement with Versal. Furthermore, in order to improve the quality of results (QoR) and facilitate micro-architecture selection, we introduce an automatic optimization framework that performs accurate performance modeling and fast design space exploration. Experiment results demonstrate that HeteroSVD reduces the latency by 1.98× over existing FPGA accelerators and outperforms GPU solutions with an improvement of up to 7.22× in latency, 1.77× in throughput, and 13.18× in energy efficiency.**

I. Introduction

In linear algebra, singular value decomposition (SVD) is a commonly used approach for identifying singular values and constructing a set of singular vectors or matrices from a given matrix. It is widely adopted in applications such as wireless communication [1]–[3], recommendation system [4], [5], etc., where data approximation, compression, and denoising are indispensable. Despite its importance and popularity in data processing, SVD is highly computationally intensive. For example, the Hestenes-Jacobi decomposition method, one of the mainstream SVD algorithms, conducts orthogonalization transformation, namely, multiple vector multiplications, for each pair of column vectors, in a single iteration. This leads to $N \times (N-1)/2$ rounds of vector multiplication computation in an iteration for an input matrix with N columns. To make matters worse, SVD algorithms usually demand multiple iterations until results finally converge. The high computation cost of SVD makes efficient implementation extremely difficult, especially under rigorous real-time requirements.

To boost the performance of real-time SVD processing, researchers have presented SVD accelerator architectures based on field-programmable gate array (FPGA) or graphics processing unit (GPU) platforms. FPGAs [6]–[8] are featured with programmable logic and wirings that can realize any boolean function via hardware reconfiguration. As a result, using FPGAs as the substrate, researchers can take advantage of a number of flexible logic blocks and wirings to tailor the control and data paths of SVD implementation, finally achieving low latency per matrix transformation. However, the scarcity of on-chip memory hinders FPGA solutions from achieving high throughput through effective parallelism. Conversely, GPUs [9]–[13] have a large number of well-constructed computation kernels and this facilitates the simultaneous processing of a set of data. Therefore, researchers investigate batch SVD implementation on GPU and demonstrate higher throughput compared with FPGA design. Nevertheless, this is accompanied by high power consumption. Besides this, GPU solutions incur high latency in computing SVD for single and small-scale matrices since the computation kernels are mostly underutilized in this case. In summary, the existing solutions for efficient SVD implementation fail to jointly optimize latency, throughput, and power consumption due to the high computational intensity of SVD and the inherent limitations of the target platforms.

Recently, AMD/Xilinx has introduced the Versal ACAP architecture, which integrates AI engines (AIEs), programmable logic (PL), and processor systems (PS) to address the limitations of GPUs and FPGAs. The AIEs are optimized for vector and matrix multiplication, allowing for high performance and low power consumption. However, due to the complexity of Versal ACAP, there are three key challenges for efficient SVD acceleration on Versal ACAP. First, the problem size of SVD often exceeds the computational power of a single AIE kernel, necessitating careful partitioning of the input matrix for effective processing. Second, as the usage of AIEs increases with the growing design scale, the limited communication bandwidth between AIEs and PL may incur a bottleneck. Lastly, the versatility of AIEs and the flexibility of PL give rise to a vast design space, making it non-trivial to identify the optimal configurations of micro-architecture that can offer high performance and energy efficiency.

To address these challenges, in this work, we propose an acceleration paradigm of SVD on the heterogeneous computing platform Versal ACAP, which is named HeteroSVD. HeteroSVD constructs the hardware of SVD with a specific focus on the system-level optimization through effectively harnessing the AIE and PL resources. It presents the techniques of workload allocation, communication optimization, and algorithm-hardware co-design. To go one step further, HeteroSVD presents an automatic optimization framework to take into account the effect of various micro-architecture parameters, which helps to rapidly identify the optimal hardware solutions under different problem sizes. The main contributions are as follows:

- We present HeteroSVD[‡], a high-performance and scalable hardware architecture of SVD on the Versal ACAP. To the best of our knowledge, this is the first work that seeks to optimize SVD on this novel heterogeneous computing platform.
- We propose a co-design methodology that jointly optimizes the SVD ordering from an algorithm perspective and the AIE-centric dataflow and placement from a hardware perspective, which boosts the overall performance and energy efficiency.
- We introduce an automatic design optimization framework with a performance analysis model and an efficient design space exploration engine, which navigates the rapid tuning of micro-architecture parameters and ultimately leads to a fast exploration of optimal designs given different problem sizes.

II. Background

A. Block Hestenes-Jacobi Algorithm for SVD

The SVD problem can be factorized as:

$$\boldsymbol{A}_{m\times n} = \boldsymbol{U}_{m\times n}\boldsymbol{\Sigma}_{n\times n}\boldsymbol{V}^{\boldsymbol{T}}_{n\times n}, \tag{1}$$

†Corresponding author.
‡Our method is open-source at https://github.com/zhaokang-lab/HeteroSVD.

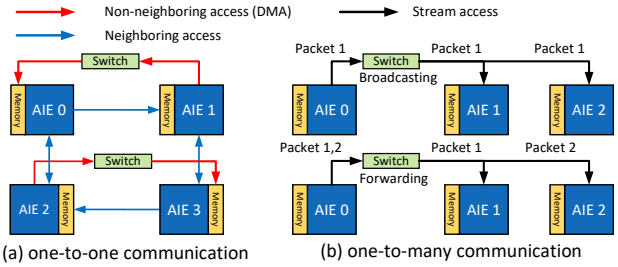

Fig. 1 Various AIE communication mechanisms.

where both U and V are unitary matrices and Σ is a diagonal matrix, and the diagonal elements of Σ are called singular value of A. The key steps of Hestenes-Jacobi [14] denoted as orthogonalization:

$$\begin{aligned} \boldsymbol{B} &= \boldsymbol{AV} = \boldsymbol{U\Sigma}, \\ \boldsymbol{V} &= \boldsymbol{J_1 J_2 J_3 J_4} \cdots . \end{aligned} \tag{2}$$

where the rotation matrix [15] $\boldsymbol{J}$ comprised of c and s is shown as:

$$[\boldsymbol{B}_i, \boldsymbol{B}_j] = [\boldsymbol{A}_i, \boldsymbol{A}_j] \cdot \begin{bmatrix} c & -s \\ s & c \end{bmatrix}, where\ \boldsymbol{B}_i^T \cdot \boldsymbol{B}_j = \boldsymbol{0}, \tag{3}$$

$$c = \frac{1}{\sqrt{1+t^2}}, \text{ and } \mathrm{s} = a_i^T a_j \frac{tc}{|a_i^T a_j|}, \tag{4}$$

$$t = \frac{\mathrm{sign}(\tau)}{|\tau| + \sqrt{1+\tau^2}}, \text{ and } \tau = \frac{a_j^T a_j - a_i^T a_i}{2|a_i^T a_j|}. \tag{5}$$

After multiple orthogonalizations for all column pairs, the convergence rate is less than the specified precision:

$$\frac{\boldsymbol{B}_i^T \boldsymbol{B}_j}{\sqrt{(\boldsymbol{B}_i^T \boldsymbol{B}_i)(\boldsymbol{B}_j^T \boldsymbol{B}_j)}} < precision. \tag{6}$$

Finally, normalization is performed to calculate Σ and U by:

$$\boldsymbol{\Sigma} = \sqrt{\boldsymbol{B^T B}}, \quad \boldsymbol{U} = \boldsymbol{B}/\boldsymbol{\Sigma}. \tag{7}$$

To solve the SVD of large matrices with limited resources, the block Jacobi algorithm is proposed. It divides a large-scale matrix into manageable sub-matrix blocks, enumerates block pairs, and further orthogonalizes all column pairs in a specific sequence such as ring [16] and round-robin [17].

B. Versal ACAP

Versal ACAP is a high-performance heterogeneous computation architecture consisting of central processing units (CPUs), PL, and AIEs, which are connected by a high-bandwidth network on chip (NoC). Each AIE comprises an AI-oriented computation core, a tile interconnect module (denoted as a switch), and a memory module with four banks of 8KB. Additionally, there are multiple interfaces called PLIO between the PL and AIEs, supporting a bandwidth of 24 GB/s from AIE to PL and 32 GB/s in the reverse direction.

Fig. 1 illustrates different mechanisms of data movement between AIEs. Fig. 1(a) depicts one-to-one communication where all AIEs are placed next to each other. The blue arrows indicate that the source AIE can access the memory of the target AIE, because the computation core and memory are adjacent, with no other computation cores or memory blocking them in between, thus considering them neighbors. In contrast, non-neighboring AIEs' communication is supported by the direct memory access (DMA) mechanism, which requires twice the memory resources and has a lower data transmission rate. Fig. 1(b) depicts two types of one-to-many communication mechanisms, including statically broadcasting packets from one AIE to several specific destinations and dynamically forwarding packets to different

Algorithm 1: SVD Algorithm in HeteroSVD.

Input: $A_{m\times n}$
Output: U,Σ

1 $convergence\_rate = 0$;
2 **while** $convergence\_rate > precision$ **do**
3 // AIE-PL communication
4 **for** *each block pair*(A_u, A_v) *in* $[A_1, A_2, \cdots, A_p]$ **do**
5 send A_u, A_v to orth-AIEs;
6 // AIE interconnection
7 **for** *each column pair*(A_i, A_j) *in block pair*(A_u, A_v) **do**
8 // AIE computation of orthogonalization (orth-AIE)
9 calculate $conv(A_i, A_j)$;
10 update $conv(A_u, A_v)$;
11 calculate t, τ, c, s;
12 update(A_i, A_j);
13 **end**
14 receive A_u, A_v, $conv_(A_u, A_v)$ from orth-AIEs;
15 update convergence_rate;
16 **end**
17 **end**
18 // AIE-PL communication
19 **for** *each block*(A_i) *in* $[A_1, A_2, \cdots, A_p]$ **do**
20 send A_i to norm-AIEs;
21 **for** *each column*(A_j) *in block*(A_i) **do**
22 // AIE computation of normalization (norm-AIE)
23 calculate Σ_j, U_j;
24 **end**
25 receive Σ_i, U_i from norm-AIEs;
26 **end**

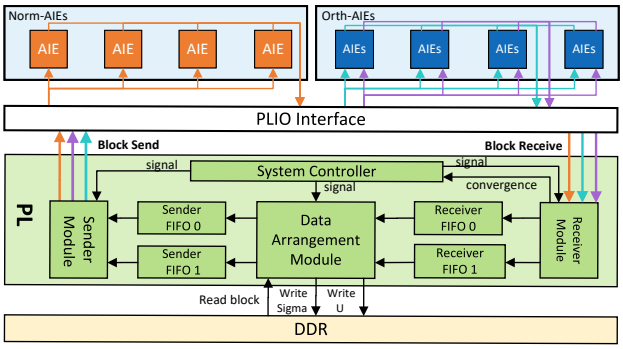

Fig. 2 HeteroSVD architecture.

destinations according to the packet header. Both utilize stream access for communication, which has a speed comparable to that of DMA.

III. HeteroSVD Accelerator Design

A. Overview

SVD Decomposition. Algorithm 1 provides the SVD algorithm that is restructured for HeteroSVD. The key idea behind Algorithm 1 is to split a large-scale matrix into smaller blocks, each of which is further divided into a set of column vectors. Subsequently, column pairs are constructed by grouping any two column vectors from two blocks, which are then sent to the AIE array for result computation. The AIEs are responsible for performing two core types of operations on each pair of column vectors, namely, orthogonalization (lines 7-13) and normalization (lines 21-24).

Overall Architecture. HeteroSVD orchestrates PL and AIE to facilitate the system-level hardware optimization, as depicted in Fig. 2. To start with, the data arrangement module reads the holistic matrix $A_{m\times n}$ with the size of $m \times n$ from DDR and splits $A_{m\times n}$ into smaller ones with the size of $m \times k$. Subsequently, it reorders the incoming data from DDR or the receiver FIFOs in a round-robin manner and passes block pairs to two sender FIFOs respectively. Next, the sender module leverages the dynamic forwarding mechanism and packs each column into packets with headers to ensure correct routing to the respective AIEs. After AIEs finish processing, the receiver

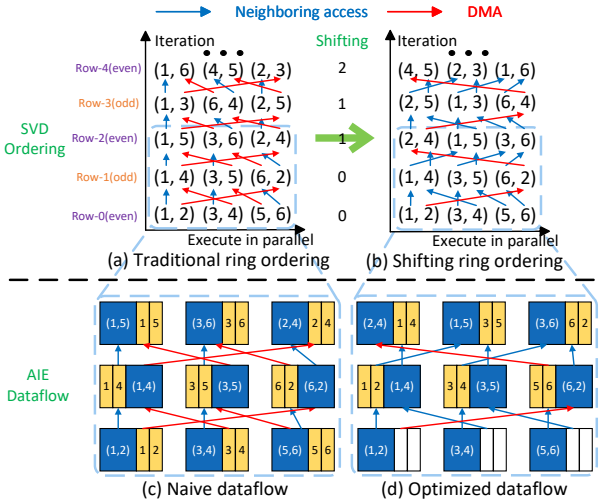

Fig. 3 The comparison of ordering and dataflow between traditional ring ordering and HeteroSVD method.

module reunites packets from AIEs, sorts them into columns, stores them in the corresponding receiver FIFOs, and then sends the deduced convergence rate to the system module. If the convergence rate is less than the user-specified precision, the system module will terminate the orthogonalization stage and proceed into the normalization stage. Finally, the data arrangement module stores the results of Σ and U into DDR and releases a signal of completion.

Note that naive SVD computation may lead to a complicated data transmission scenario between AIEs, which can severely impede communication efficiency. In light of this problem, we propose an algorithm-hardware co-design strategy to jointly optimize both the SVD ordering from an algorithm perspective and the AIE-centric dataflow from a hardware perspective. Finally, we present the method of AIE placement that instantiates the SVD on Versal ACAP.

B. Algorithm-Hardware Co-Design

SVD ordering refers to the practice of determining the processing order of orthogonalizing different column pairs. Unfortunately, the traditional SVD ordering, such as the widely used ring ordering [16], does not fit in well with AIE's inherent characteristics. Specifically, the AIEs are arranged in rows in the AIE array, with neighboring rows having different topologies: *in even rows, each computation core is located on the left side of its internal memory, whereas in odd rows, the relative position of each computation core and its memory gets reversed*. The traditional ordering of SVD algorithms maintains a monolithic data movement pattern across different iterations, which is inconsistent with the AIE alignment in different rows. This discrepancy makes it necessary to use a large number of DMA transmissions, a type of less efficient data communication between non-neighboring AIEs that leads to reduced transmission speed and increased memory usage, as discussed in Section II-B. Fig. 3 (a) and (c) illustrate this issue with matrix $A_{m\times 6}$ as an example, where the number in parentheses denotes the column index, the red arrow indicates DMA transmission and the blue represents neighboring access.

HeteroSVD proposes to co-optimize the SVD algorithm and the Versal AIE array, which can significantly reduce the number of DMA transmissions from $2k(k-1)$ to $2(k-1)$ for a matrix with the size of $m \times 2k$, as shown in Fig. 3 (b) and (d). Specifically, we propose the shifting ring ordering method to augment the SVD algorithm, and the AIE dataflow that reduces communication and memory usage.

Shifting Ring Ordering. To accommodate the AIE array's asymmetric characteristics, we propose the shifting ring ordering mecha-

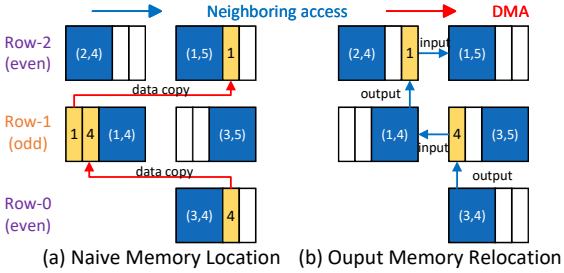

Fig. 4 Dataflow Comparison between different memory location strategies.

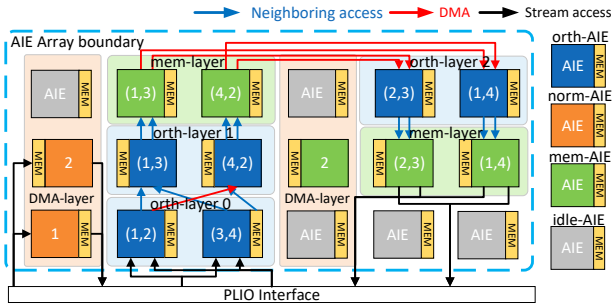

Fig. 5 AIE placement with three types of AIE.

nism for SVD. The main idea is to dynamically change the pattern of data movements in an effort to align with the topology of each AIE row, as depicted in Fig. 3 (b). For row-i, we shift the column pairs to the right by a step size of $\lfloor \frac{i}{2} \rfloor$ in a cyclic manner. For instance, in Fig. 3 (b), all the column pairs in row-2 are shifted by one step cyclically: the right-most column pair (2,4) becomes the left-most, the left-most column pair (1,5) shifts to the middle, and so on. By ordering in this way, the original straight and leftward movements from odd rows to even rows are respectively transformed into rightward and straight movements, while the movements from even rows to odd rows remain unchanged. This method synchronizes the data movement with the topologies of the AIE array.

AIE-Centric Dataflow. To optimize the inter-AIE communication, we relocate the position of output storage to convert the non-neighboring DMA transmission into direct neighboring access. As illustrated in Fig. 4 (a), using the naive memory configuration, each AIE's output is stored in its own memory, necessitating DMA to copy the data due to the lack of a direct interface between the source AIE and target AIE. This means that the orthogonalization operation requires a memory block twice the size of the result to store it, and an expensive DMA communication is engaged as well. To address this issue, we subtly assign each AIE's output to the subsequent AIE's memory in the next row as shown in Fig. 4 (b). This allows the target AIE to communicate with the source AIE through two neighboring accesses via the intermediate AIE memory. Untimately, except for the long-distance communication between the first and the last AIE columns, all data movements from the shifting ring ordering can be achieved with neighboring direct access, as shown in Fig. 3 (d).

C. AIE Placement

AIE placement means the instantiation of AIEs and the corresponding interconnect to implement the SVD algorithm on hardware. However, this problem is challenging. The complete SVD algorithm comprises two phases: orthogonalization and normalization. It is essential to wisely assign AIEs to complete both phases. In particular, the shifting ring ordering for orthogonalizing matrix $A_{m\times 2k}$ requires an array of $(2k-1)\times k$ AIEs, which is incompatible with the 8×50

size of the AIE array on our target board. To this end, we propose an AIE placement strategy, and Fig. 5 illustrate it with matrix $A_{m\times 4}$ as an example.

We define three types of AIE, namely orth-AIE, norm-AIE, and mem-AIE, which are designed for orthogonalization, normalization, and the storage of intermediate data, respectively. For the orthogonalization stage, we partition the shifting ring ordering of size of $(2k-1)\times k$ into $2k-1$ orth-layers, each consisting of k orth-AIEs. Then we place the $2k-1$ orth-layers in a row-wise manner. Notably, at the boundary of the AIE array, namely the first row and the last row, we place a mem-layer instead of an orth-layer as the maximum row limitation of the AIE array prevents a subsequent row from storing the current layer's output. Consequently, we choose another k columns to continue placing the remaining orth-layers, albeit at the expense of some unavoidable DMA transmissions. Moreover, we assign the columns on both sides of the orth-layers as DMA-layers, allocating mem-AIEs to store the copied data due to the DMA mechanism, This is necessary because the orth-AIEs with DMA have already stored data from the previous orth-layers and lack sufficient memory to accommodate the additional copied data. For the normalization stage, we allocate the norm-AIE within the remaining idle-AIE.

We further establish a dynamic forwarding rule to ensure that the block pair from PL is correctly routed to the placed AIEs. Specifically, we designate that odd and even columns are sourced from different blocks within the block pair, utilizing four PLIOs for their respective transmission. Different columns from a single block are routed to different AIEs as shown in Fig. 5. For the norm-AIE, we only use two PLIOs, as the two blocks in the block pair are transmitted sequentially between the PL and AIE.

IV. Automatic Design Optimization Framework

In this section, we first describe the micro-architecture parameters derived from two levels of parallelism. After that, we discuss our performance modeling and design space exploration (DSE) methods.

A. Micro-Architecture Parameters

HeteroSVD has two levels of parallelism that expand its design space, as shown in Fig. 6. The first parallelism is AIE-level parallelism (denoted as P_{eng}), which determines the number of AIEs used in parallel to execute the SVD computation for a single task. The second parallelism is task-level parallelism (denoted as P_{task}), which represents the number of tasks processed simultaneously in the computing system. These two parallelism factors, together with the given operating frequency of PL, determine the system-level performance and guide the selection of parameters in the HeteroSVD design, making them first-order micro-architecture parameters, as listed in TABLE I. Additionally, second-order micro-architecture parameters include the number of PLIOs and the counts of AIEs designated as orth-AIE, norm-AIE, and mem-AIE.

In summary, by specifying P_{eng}, P_{task}, and PL frequency, we can determine the architecture of HeteroSVD. However, searching for the optimal and feasible HeteroSVD architecture by enumerating all possible micro-architecture parameters requires the full invocation of the EDA front-end, back-end, and cycle-accurate simulation flow, which is extremely time-consuming. For instance, running a single design point for an SVD of a 128×128 matrix would take more than seven hours. Considering a design space of (P_{eng}, P_{task}) that leads to 286 design points, the cost of searching for optimal architecture is unaffordable. Therefore, HeteroSVD proposes an automation design approach with a performance model and a design space exploration flow to construct the optimal HeteroSVD structure of a given problem size within minutes.

TABLE I HeteroSVD architecture parameters at different hierarchies.

Hierarchy	Parameter	Range or Value
First order	P_{eng}	$n \in [1, 11]$
	P_{task}	$k \in [1, 26]$
	PL frequency (MHz)	-
Second order	the number of orth-AIE	$n(2n-1)k$
	the number of norm-AIE	nk
	the number of mem-AIE	-
	the number of PLIO	$6k$

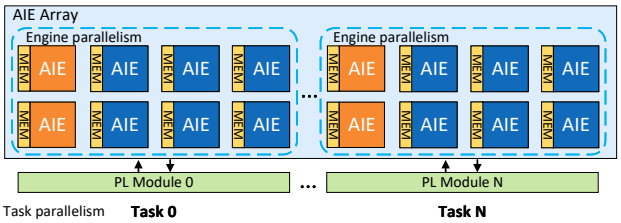

Fig. 6 Engine parallelism and task parallelism.

B. Performance Model

To efficiently evaluate the performance with different AIE-level and task-level parallelisms, we first present a performance model to estimate the latency and throughput of the overall computing system. The complete pipeline of the orthogonalization is shown in Fig. 7. This pipeline can be further decomposed into data sending (Tx) from PL to AIE, data receiving (Rx) of PL from AIE, and execution of orthogonalization (orth-AIEs). And since orth-AIEs have two inputs and output, we use left(L) and right(R) to distinguish them.

The execution time of Tx and Rx is determined by PL frequency and PL bandwidth:

$$t_{Tx,Rx} = \frac{databits}{bandwidth \cdot frequency}, \tag{8}$$

and the AIE time is estimated by the AIE simulator in advance. Besides, there are three main types of latency.

AIE-wait latency ($t_{AIEwait}$) arises from the imbalance between the execution time of AIE and the time taken for data transmission:

$$t_{AIEwait} = max(t_{orth} - P_{eng} \cdot t_{Tx}, 0). \tag{9}$$

Specifically, when multiple packets are transmitted serially to different AIEs through a single PLIO, the incoming data must wait for the completion of AIE's last execution.

Algorithm latency (t_{algo}) results from the data dependency between the second Rx-R and the first Tx-R in the round-robin algorithm:

$$t_{algo} = t_{Tx} + t_{AIEwait}. \tag{10}$$

The lack of sending data causes data-wait latency ($t_{datawait}$):

$$\begin{aligned} t_{datawait} = max(-(num-1)\cdot(t_2 + t_{Tx}) \\ + t_{AIEs} + t_{Rx} + t_{algo}, 0), \end{aligned} \tag{11}$$

where num and t_{AIEs} represent the number of block data pairs and the total AIEs time between Tx and Rx.

Beyond these latencies, there is also latency caused by HLS (t_{hls}) and DDR (t_{DDR}). HLS [18] incurs additional cycles when switching between loops, thus, t_{hls} can be calculated based on the loop structure in the code. The inability to load block pairs from DDR simultaneously results in significant latency during the first iteration:

$$t_{DDR} = num \cdot t_{Tx}. \tag{12}$$

Then we can calculate the time required for one iteration:

$$\begin{aligned} t_{blocks} &= num \cdot (t_{Tx} + t_{AIEwait}) + t_{algo} + t_{datawait}, \\ t_{iter} &= (num-1)\cdot t_{blocks} + AIE_{total} + t_{Rx}. \end{aligned} \tag{13}$$

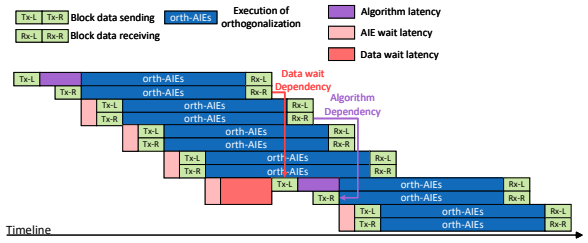

Fig. 7 HeteroSVD pipeline model at orthogonalization stage

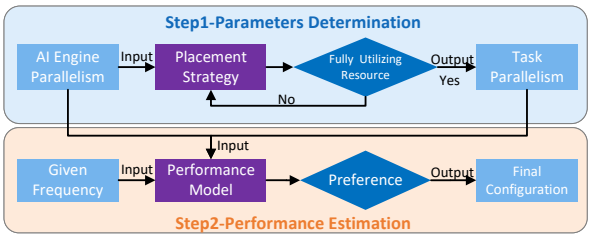

Fig. 8 DSE flow with two stages.

Norm-AIE's pipeline resembles orth-AIE's but with single input/output and no algorithm latency. We define the normalization time as t_{norm}. The task time can be computed by summing the execution times of orthogonalization, normalization, and additional latency. The system time can then be derived using:

$$\begin{aligned} t_{task} &= t_{DDR} + ITER \cdot t_{iter} + t_{norm} + t_{hls}, \\ t_{sys} &= \lceil \frac{num_{task}}{P_{task}} \rceil \cdot t_{task}, \end{aligned} \tag{14}$$

where ITER represents the specified iterations and num_{task} represents the number of tasks.

C. Design Space Exploration (DSE)

Problem Formulation. In general, the goal of our architecture optimization framework is to increase the performance of the SVD computing system. Specifically, given the size of the matrix ($M \times N$) and batch size (B), we need to properly select the first-order parameters listed in TABLE I, i.e., the parallelism factor of AIEs (P_{eng}) and tasks (P_{task}), and the practical frequency ($Freq$) of PL, while keeping all types of resources, including AIE kernel, PLIO, BRAM and URAM, under their budgets. To this end, we formally define our problem of design space exploration as follows:

$$\begin{aligned} \min \quad & runtime(P_{eng}, P_{task}, Freq), \\ \text{s.t.} \quad & Resource_i(P_{eng}, P_{task}), \leq C_i \\ & i \in \{AIE,\ PLIO, BRAM, URAM\}. \end{aligned} \tag{15}$$

DSE Flow. Fig. 8 illustrates the design exploration flow to search for the optimal HeteroSVD architecture. In the first stage, we enumerate the engine parallelism and maximize task parallelism by fully utilizing resource according to our placement strategy. Equation (16) summarized all constraints C_i, where $i \in \{AIE, PLIO, BRAM, URAM\}$, represent the number of hardware resource limits. The actual usage of AIE resource, num_{orth}, num_{norm}, num_{mem}, and num_{PLIO}, can be determined after placement while the resource usage of PL memory, including num_{BRAM} and num_{URAM} can be estimated by the array size of PL design. In the second stage, with engine parallelism, task parallelism, and the user-specified frequency as input, we utilize our performance model to estimate the overall computing system's performance. Furthermore, We determine the optimal HeteroSVD architecture based on the demand of latency or throughput.

$$\begin{aligned} num_{orth} + num_{norm} + num_{mem} &\leq C_{AIE}, \\ num_{PLIO} &\leq C_{PLIO}, \\ num_{BRAM} &\leq C_{BRAM}, \\ num_{URAM} &\leq C_{URAM}. \end{aligned} \tag{16}$$

V. Experiment Results

A. Experimental Setup

Our experiments are conducted on AMD/Xilinx Versal AI Core Series VCK190 [19] evaluation board. We use the Vitis 2023.2 toolkit to build up the whole HeteroSVD acceleration system. The AIEs operate at 1.25 GHz, while the frequency of the PL design depends on the specific system configurations. We use the AMD BEAM tool [20], to measure the exact power of the system while running on the VCK190 board. We compare our SVD acceleration system with the SOTA SVD accelerators implemented on AMD/Xilinx XC7V690T FPGA and GeForce RTX 3090 GPU.

B. Performance and Energy Efficiency

We compare HeteroSVD, with the state-of-the-art high-performance hardware implementations on FPGA [6] and GPU [11], both of which offer better latency or throughput compared to other implementations on the same platform. To enable a fair comparison, we perform six iterations per matrix and configure the FPGA accelerator to its maximum task parallelism at an achievable peak frequency of 200 MHz. Generally speaking, the GPUs perform better with large-scale data, while the FPGAs are featured with low-latency data processing. TABLE II presents the latency comparison and resource utilization between HeteroSVD and FPGA. The experimental results show that the HeteroSVD solution supports more flexible architecture and significantly outperforms the FPGA solution regarding latency in all matrix sizes, achieving 1.27× – 1.98× speedup over FPGA. Meanwhile, HeteroSVD consumes fewer computational resources than FPGA, indicating that HeteroSVD has greater potential for parallelism in handling multiple tasks.

Furthermore, we evaluate the latency, throughput, and energy efficiency for batch data processing of SVD between HeteroSVD and GPU as listed in TABLE III. We obtain the optimal micro-architecture configuration from HeteroSVD's DSE flow. In addition, we perform multiple iterations of SVD until the results converge at a rate of 10^{-6} for both HeteroSVD and GPU solution. HeteroSVD achieves 1.15× – 7.22× latency speedup over GPU for small matrix. Regarding throughput, HeteroSVD shows up to 1.77× speedup with a small matrix size. However, as the matrix size increases, the GPU solution outperforms our HeteroSVD solution in throughput. Fig. 9 illustrates the underlying mechanism. As the design size increases, the GPU solution exhibits higher utilization of its computation cores and memory, significantly boosting its performance. In contrast, due to limitations in PL memory, HeteroSVD experiences reduced task parallelism, leading to lower overall utilization. Moreover, larger design sizes increase the complexity of the PL, which reduces the maximum achievable frequency and further diminishes HeteroSVD's latency and throughput. Therefore, with adequate RAM resources and optimized operating frequency, we believe that HeteroSVD has the potential to outperform GPU solutions with similar core utilization.

As for energy efficiency, HeteroSVD demonstrates 4.36× – 13.18× improvement over GPU solutions. In summary, the tremendous gains justify that HeteroSVD is a highly competitive alternative to both FPGA and GPU in achieving high-performance yet energy-efficient SVD accelerators.

TABLE II Latency comparison and resource utilization between HeteroSVD and FPGA.

Matrix Size	FPGA [6]				HeteroSVD				HeteroSVD
	Latency(s)	LUT	BRAM	DSP	Latency(s)	LUT	URAM	AIE	Speedup
128×128	0.0014	212K (30.6%)	519.5 (31.4%)	1602(44.5%)	0.0011	15.1K(1.68%)	4(0.86%)	128(32%)	1.27×
256×256	0.0113				0.0057	15.2K (1.69%)	20 (4.32%)		1.98×
512×512	0.0829				0.0435	15.5K (1.72%)	64 (13.82%)		1.90×
1024×1024	0.6119				0.3415	15.7K (1.75%)	244 (52.70%)		1.79×

TABLE III Latency, throughput and energy efficiency comparison between HeteroSVD and GPU.

Matrix Size	GPU [11] (270W)			HeteroSVD (<39W)			HeteroSVD vs GPU		
	Latency (s)	Throughput (Tasks/s)	Energy Efficiency (Tasks/s/Watt)	Latency (s)	Throughput (Tasks/s)	Energy Efficiency (Tasks/s/Watt)	Latency Speedup	Throughput Speedup	EE Gain
128×128	0.0166	1351.35	5.005	0.0023	2389.69	65.940	7.22×	1.77×	13.18×
256×256	0.0429	217.39	0.805	0.0130	239.48	6.251	3.30×	1.10×	7.76×
512×512	0.1237	27.55	0.102	0.1076	24.42	0.663	1.15×	0.89×	6.50×
1024×1024	0.6857	3.52	0.013	0.7937	1.27	0.057	0.86×	0.36×	4.36×

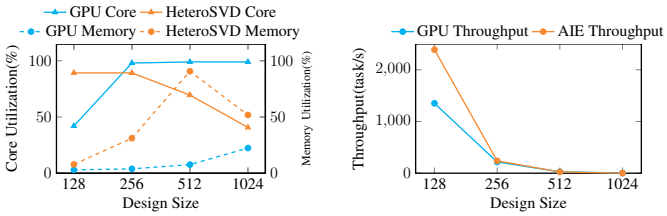

Fig. 9 The throughput and utilization of core and memory comparison with different design sizes between GPU and HeteroSVD

TABLE IV The processing time (ms) of SVD single iteration from the performance model and on-board measurement under a frequency of 208.3 MHz.

Matrix Size	P_{eng}	On-board Meas.	Perf. Model	Error
128×128	2	0.993	1.022	2.92%
256×256		6.151	6.338	3.03%
512×512		43.229	42.020	2.80%
128×128	4	0.395	0.391	1.03%
256×256		2.853	2.806	1.66%
512×512		21.584	21.265	1.48%
128×128	8	0.214	0.219	2.57%
256×256		1.475	1.476	0.05%
512×512		10.965	10.903	0.56%

TABLE V The processing time (ms) of SVD with one iteration from the performance model and on-board measurement under various application scenarios.

Matrix Size	Batch	Freq.(MHz)	P_{eng}	P_{task}	On-board Meas.	Perf. Model	Error
128×128	1	450	8	1	0.357	0.384	7.52%
256×256	1	420	8	1	1.202	1.120	6.82%
512×512	1	350	8	1	7.815	7.510	3.90%
1024×1024	1	310	8	1	58.885	58.255	1.02%
128×128	100	330	4	9	6.099	6.412	5.12%
256×256	100	310	4	9	27.836	26.623	4.36%
512×512	100	310	4	7	238.002	224.301	5.76%
1024×1024	100	310	8	1	5872.181	5878.970	0.12%

C. Effectiveness of DSE

Accuracy of Performance Model. The accuracy of the performance model is key to developing a high-quality design space exploration approach. Therefore, in this experiment, we first evaluate the effectiveness of our performance model. We apply different factors of engine parallelism and the input matrix size and examine the processing time of HeteroSVD in terms of a single iteration. The frequency of the PL side is fixed at 208.3 MHz. TABLE IV shows the results of the HeteroSVD performance model and the on-board measurement. We can observe that the maximum error of our model is 3.03% and the average error is 1.78%, which shows the efficacy of our performance model in the evaluation of latency.

To further validate the generalization ability of the performance model across different application scenarios, we use the HeteroSVD DSE approach to identify the optimal configuration with minimum execution time. Then, we get the latency of processing a single matrix and the throughput of processing 100 matrices from both the model and on-board measurement. As shown in TABLE V, our performance model demonstrates a maximum error of 7.52% and an average error of 4.33%. This highly accurate performance model ensures the effectiveness of our DSE flow with design point enumeration.

TABLE VI Latency(ms), throughput(task/s) and power(W) comparison between different design points. The matrix size is 256×256 and the PL frequency is reported in 208.3MHz

P_{eng}	P_{task}	AIE	URAM	Latency	Throughput	Power
2	26	293(73.25%)	416(89.85%)	35.689	707.501	44.16
4	9	357(89.25%)	144(31.10%)	19.303	508.436	34.63
6	4	366(91.50%)	120(25.92%)	13.117	306.876	30.79
8	2	322(80.50%)	32(6.91%)	9.247	219.257	26.06

Discussion on DSE Results. TABLE VI evaluates how micro-architecture parameters influence latency, throughput, and power consumption of SVD implementation, with each design point executing six iterations. Increasing P_{eng} allows for more AIEs to be allocated to a single SVD, effectively reducing latency. However, a design point with high P_{eng} is not suitable for scenarios with strict throughput requirements, as it restricts task parallelism. Conversely, higher task parallelism (P_{task}) improves throughput but increases URAM usage, leading to significant power consumption. In summary, we favor lower P_{eng} and higher P_{task} for high throughput, while the opposite configuration is preferable for minimizing latency. Moreover, a comprehensive consideration of micro-architecture parameters is essential for effectively managing power consumption.

VI. Conclusion

In this paper, we propose HeteroSVD, an optimized hardware accelerator of block-Jacobi SVD based on the novel heterogeneous computing platform, Versal ACAP. For HeteroSVD accelerator design, we decompose the computation of SVD and propose the algorithm-hardware co-design for SVD ordering, AIE dataflow, and AIE placement. To further optimize the design with a set of micro-architecture parameters, we introduce a design space exploration flow to expedite the search for the best hardware solutions at different problem sizes.

Acknowledgment

This work is supported by the National Key R&D Program of China (2022YFB2901100), the National Natural Science Foundation of China (No. 62404257, 62404021), the Beijing Natural Science Foundation (No. 4244107, QY24216, QY24204), and the Guangdong Basic and Applied Basic Research Foundation (2023A1515110769).

REFERENCES

[1] J. Löfgren, S. Mehmood, N. Khan, B. Masood, M. I. Z. Awan, I. Khan, N. A. Chisty, and P. Nilsson, "Hardware implementation of an SVD based MIMO OFDM channel estimator," in *2009 NORCHIP*, 2009, pp. 1–4.

[2] M. V. Athi, S. R. Zekavat, and A. A. Struthers, "Real-Time Signal Processing of Massive Sensor Arrays via a Parallel Fast Converging SVD Algorithm: Latency, Throughput, and Resource Analysis," *IEEE Sensors Journal*, vol. 16, no. 8, pp. 2519–2526, 2016.

[3] G. Lebrun, J. Gao, and M. Faulkner, "MIMO transmission over a time-varying channel using SVD," *IEEE Transactions on Wireless Communications*, vol. 4, no. 2, pp. 757–764, 2005.

[4] W. Wu, L. Zhao, Q. Wu, X. Wang, T. Tian, and X. Jin, "An SSD-Based Accelerator for Singular Value Decomposition Recommendation Algorithm on Edge," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–5.

[5] X. Guan, C.-T. Li, and Y. Guan, "Matrix factorization with rating completion: An enhanced SVD model for collaborative filtering recommender systems," *IEEE Access*, vol. 5, pp. 27 668–27 678, 2017.

[6] T. Hu, X. Li, X. Yu, S. Ren, L. Yan, X. Bai, Z. Xu, and S. Zhu, "A Novel Fully Hardware-Implemented SVD Solver Based on Ultra-Parallel BCV Jacobi Algorithm," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 12, pp. 5114–5118, 2022.

[7] X. Wang and J. Zambreno, "An FPGA Implementation of the Hestenes-Jacobi Algorithm for Singular Value Decomposition," in *IEEE International Parallel & Distributed Processing Symposium Workshops*, 2014, pp. 220–227.

[8] Y. Wang, J.-J. Lee, Y. Ding, and P. Li, "A Scalable FPGA Engine for Parallel Acceleration of Singular Value Decomposition," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*, 2020, pp. 370–376.

[9] C. Toolkit, Cuda toolkit documentation, 2018. [Online]. Available: http://docs.nvidia.com/cuda/index.html

[10] R. Huang, T. Yu, S. Liu, X. Zhang, and Y. Zhao, "A Batched Jacobi SVD Algorithm on GPUs and Its Application to Quantum Lattice Systems," in *Parallel and Distributed Computing, Applications and Technologies*, H. Shen, Y. Sang, Y. Zhang, N. Xiao, H. R. Arabnia, G. Fox, A. Gupta, and M. Malek, Eds., 2022, pp. 69–80.

[11] J. Xiao, Y. Pang, Q. Xue, C. Shui, K. Meng, H. Ma, M. Li, X. Zhang, and G. Tan, "W-Cycle SVD: A Multilevel Algorithm for Batched SVD on GPUs," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–16.

[12] S. Lahabar and P. Narayanan, "Singular value decomposition on GPU using CUDA," in *IEEE international symposium on parallel & distributed processing*, 2009, pp. 1–10.

[13] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes, "Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression," *Parallel Computing*, vol. 74, pp. 19–33, 2018.

[14] M. R. Hestenes, "Inversion of Matrices by Biorthogonalization and Related Results," *Journal of the Society for Industrial and Applied Mathematics*, vol. 6, no. 1, pp. 51–90, 1958.

[15] N. L. Bihan and S. J. Sangwine, "Computing the SVD of a quaternion matrix," 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:18661147

[16] "A Parallel Ring Ordering Algorithm for Efficient One-Sided Jacobi SVD Computations," *Journal of Parallel and Distributed Computing*, vol. 42, no. 1, pp. 1–10, 1997.

[17] R. P. Brent and F. T. Luk, "The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 69–84, 1985.

[18] AMD Vitis High-Level Synthesis User Guide (UG1399). [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction

[19] AMD VCK190 board. [Online]. Available: https://china.xilinx.com/products/boards-and-kits/vck190.html

[20] AMD BEAM Tool for VCK190 Evaluation Kit. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2549186622/Power+Management+-+Getting+Started#Tools-to-Estimate-and-Measure-Power