HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation

Abstract—**Tensor computation overwhelms traditional general-purpose computing devices owing to the massive data and operations. It calls for a holistic solution composed of both hardware acceleration and software mapping. Hardware/software (HW/SW) co-design optimizes the hardware and software in concert and produces high-quality solutions. There are two main challenges in the co-design flow. First, multiple methods exist to perform HW/SW partitioning and have different impacts on performance and energy efficiency. Besides, the hardware part must be implemented by the intrinsic functions of accelerators. It is hard for programmers to identify and tell the differences between partitioning methods manually. Second, the overall design space composed of HW/SW partitioning, hardware optimization, and software optimization is huge. Programmers call for efficient approaches to explore the design space.**

To this end, we propose an agile co-design approach HASCO that provides an efficient HW/SW solution to tensor computation. We first lower the target tensor computation and intrinsic functions into tensor syntax trees, based on which HASCO exposes partition methods. We propose different algorithms for exploring the hardware and software design spaces, as they have distinct objectives and evaluation costs. We develop a multi-objective Bayesian optimization algorithm to explore hardware optimization. For software optimization, we use heuristic and Q-learning algorithms. Experiments demonstrate that HASCO achieves 1.25X to 1.44X latency reduction through HW/SW co-design compared with developing the hardware and software separately.

I. INTRODUCTION

Tensor computation is fundamental for many scientific and engineering applications, such as machine learning [1], [5], [39], [46], [65], data mining [40], [51], [56], quantum chemistry [17], [69]. Tensors are data organized in multi-dimensional arrays. Common tensor computations include matricized tensor times Khatri-Rao product (MTTKRP), tensor-times-matrix (TTM), general matrix multiply (GEMM), general matrix-Vector multiplication (GEMV), and convolution. For example, the 2D convolution is one of the most popular tensor operations in deep learning applications. MTTKRP is widely used for tensor factorization in recommendation systems [66].

For tensor computation, it is essential to develop a *holistic* solution that is a combination of *hardware acceleration* and *software mapping*. The conventional general-purpose processors suffer from the increasingly high complexity of tensor computation, which motivates specialized hardware acceleration. Recently, dedicated spatial accelerators implemented on FPGAs and ASICs are shown to be efficient hardware architectures due to the massive parallelism and high energy efficiency [14], [15], [27], [28], [34], [42], [45], [61], [67]. For instance, Google Cloud TPU [34], [54], an ASIC processing neural networks, can reduce the training time by 27X at a 38% lower cost than NVIDIA V100 GPU clusters. On the other side, the success of an end-to-end acceleration solution hinges largely on the software mapping or compilation. For tensor computation accelerator, the software mapping is responsible for splitting a large tensor into small parts and invoking the corresponding hardware execution, as the accelerator can only handle a fixed size of tensor at a time. The mapping is often accomplished by loop transformations, including loop splitting, reordering, tiling, fusion, etc. Software mapping is crucial for performance optimization. For instance, compared with manually calling tensor cores of the V100 GPUs, an optimized software CUTLASS [21] can achieve up to 1.73X performance improvement.

Though dedicated hardware and software optimizations have progressed considerably for tensor computation, they primarily focus on *either the hardware part [26], [35], [37], [60], [78] or the software part [13], [20], [32], [58], [82]*. Optimizing the two parts in isolation inevitably suffers from sub-optimal solutions confined in a local design space. While seemingly appealing, there has been less attention on hardware/software co-design for tensor computation [3], [8], [70]. It is largely because of the high complexity brought by the entangled hardware and software design space. The fundamental questions are: 1) how to define the interface between tensor computation accelerator and software programs, 2) how to navigate the huge design space for each part.

In this paper, specific to tensor computation, we term the HW/SW interface as *tensorize*. A tensorize choice determines how to divide the tensor computation into sub-workloads and map the sub-workloads onto the hardware. The challenge is that **there exist multiple tensorize choices, which have sufficiently distinct impacts for performance and energy efficiency**. For instance, dot product, 1D/2D convolutions, GEMM, and GEMV are all tensorize choices for a 2D convolution. Designers do not yet have the reliable intuition about how to select from these choices, as the selection highly relies on the underlying accelerator. The accelerator designed for tensor computation typically supports one or a set of specific functions, which are termed as *hardware intrinsics*. For instance, the hardware intrinsic of Gemmini accelerators [25] is a GEMM function. The intrinsics of NVDLA accelerators [22] include 2D convolutions, pooling, activation functions, etc. Different tensorize choices built on the same intrinsic are likely to vary a lot in performance due to different levels of data locality, reuse opportunities, and padding styles. Furthermore, the size of the sub-workload can vary dramatically, making the tensorize choices more complicated.

The tensorize interface separates hardware and software so that each can be optimized separately. The hardware part implements the functionalities of the tensorize (one sub-workload), and the software part invokes the hardware parts iteratively for all the sub-workloads. However, both the hardware and software design spaces are huge. Hardware parameters consist of bandwidth, datatype, parallelism, dataflow, etc., determining the detailed implementation of the intrinsic functions. Collectively, **these parameters form a huge design space, which cannot be exhaustively searched**. For example, the legal design space of a GEMM accelerator [25] is $O(10^9)$. Besides, developers need to prune the design space for different scenarios where performance (latency/throughput), power, and area are constrained. On the other hand, **software mapping for various tensor computations highly depends on the target accelerator**. Loop transformations, the primary optimization technique for tensor computations, require deep comprehension of the architecture details. For instance, loop reordering changes data locality, which impacts the efficiency of the memory hierarchy. Loop splitting factors determine the size of a sub-workload, which is restricted by the compute capability and on-chip memory size. Programmers call for efficient approaches to explore the design spaces.

In this paper, we propose HASCO, an agile co-design approach for tensor computation. HASCO jointly optimizes the hardware-software interface (tensorize choice), hardware parameters, and software optimizations. First, we define tensor syntax trees, based on which HASCO exposes tensorize choices using a two-step matching. Then, we develop different algorithms to explore the hardware and software design spaces, as they differ in optimization objectives and evaluation costs. We treat the hardware exploration as a multi-objective problem, where performance, power, and area are optimized. We develop a Bayesian optimization algorithm to find the Pareto set of hardware parameters. For the software, we use heuristic and Q-learning searching algorithms to find optimized software mapping. To sum up, the key contributions of this paper include:

- We propose a co-design approach HASCO to design hardware accelerators and software mapping in concert. HASCO offers one of the deepest exploration for tensor computations.
- We propose efficient algorithms, covering the entire design space of the hardware and software interface (tensorize choice). This interface serves as the base for separating the hardware and software designs.
- We develop heuristic, Q-learning, and Bayesian optimization algorithms to explore the hardware and software design spaces efficiently.

Experiments demonstrate that HASCO achieves 1.25X to 1.44X latency reduction through HW/SW co-design compared with developing the hardware and software separately. HASCO speeds up the hardware design space exploration by 2.5X and achieves a 1.19X hypervolume compared with NSGAII. HASCO also optimizes the software by 3.17X and 1.21X compared to a library implementation and AutoTVM, respectively.

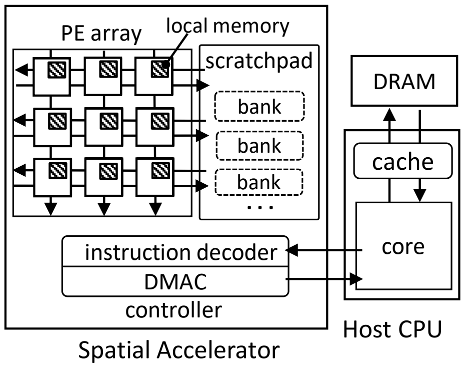


Fig. 1: A system overview of spatial accelerators.

II. Background and Motivation

A. Spatial Accelerators and Generators

Spatial accelerators have been successfully employed to accelerate tensor computation [26], [34], [60], [78], [79]. Figure 1 presents a system overview of the spatial architecture, which includes three basic components: a 1D/2D array of processing elements (PEs), a scratchpad memory, and a controller. In the PE array, hundreds of PE communicate via on-chip interconnections between them and enable massive parallelism. Within a PE, there are an ALU performing computation, registers, and an optional local memory feeding data to the ALU. The scratchpad memory is shared by all the PEs and can be further partitioned into multiple banks to support concurrent data accesses. The controller consists of an instruction decoder and a direct memory access controller (DMAC). The instruction decoder fetches and decodes instructions, which control the behaviors of PEs and the DMAC. Spatial accelerators can support multiple dataflows, which are the ways tensors are distributed to PEs and reused [41]. Recently, generators are proposed to automate the generation of spatial accelerators [22], [25], [72], [74], [77].

B. HW/SW Interface: Tensorize

Here, we use an example of mapping 2D convolutions to accelerators with GEMM hardware intrinsics. A 2D convolution can be noted as $C[k,x,y] = \sum A[c,x+r,y+s] * B[k,c,r,s]$, where tensor B is filters, A is input feature maps, and C is output feature maps. We use a convolution workload from ResNet [29] for example. A convolves B with size $64 \times 64 \times 3 \times 3$ to produce C with size $64 \times 56 \times 56$. Similarly, the GEMM intrinsic is noted as $L[i,j] = \sum M[i,k] * N[k,j]$.

In Listing 1, *Conv_workload* and *GEMM_intrin* represent the workload and the GEMM intrinsic, respectively. The hardware intrinsic can only process a fixed size GEMM computation (16×16 here). The intrinsic size is determined by the PE array shape of the accelerator. Directly calling the intrinsic function from the host CPU is inefficient, as the data

Listing 1 Mapping 2D convolutions to GEMM accelerators.

```
def Conv_workload(A, B, C, ...):
  for y in range(0, 56):
  for r in range(0, 3):
  for s in range(0, 3):                        Software
  for k1 in range(0, 64, 32):                  Program
  for x1 in range(0, 56, 32):
  for c1 in range(0, 64, 8):                     HW/SW
    tensorized_GEMM(A, B, C, ...)                Interface

def tensorized_GEMM(A, B, C, ...):
  Tensor sA, sB, sC
  sA = A[c1:c1+8, x1+r:x1+r+32, y+s]
  sB = B[k1:k1+32, c1:c1+8, r, s]
  for k2 in range(0, 32, 16):
  for x2 in range(0, 32, 16):                    Hardware
  for c2 in range(0, 8):                         Accel.
    M = sA[c2, x2:x2+16]
    N = sB[k2:k2+16, c2]
    L = GEMM_intrin(M, N, ...)   —  Intrinsic
    sC[k2:k2+16, x2:x2+16] += L
  C[k1:k1+32, x1:x1+32, y] += sC
```

movements between the host and the accelerator would be frequent and short. Hence, we employ *tensorized_GEMM* as the HW/SW interface. *tensorized_GEMM* transfers data in a burst mode and invokes the GEMM intrinsic multiple times. It processes a fixed but larger GEMM sub-workload compared with the intrinsic. The sub-workload size is mainly constrained by the scratchpad memory size and the burst length of the accelerator.

As we can express a sub-workload as the sub-loops of tensor computation, we achieve tensorization by loop splitting and reordering. In this example, the `k` loop is split into two sub-loops represented by `k1` and `k2`. Similar splitting is applied to the `x` and `c` loops. After reordering the loops, `k2`, `x2`, and `c2` determine the size of the tensorized sub-workload. The six outer-most loops form the software program launching the sub-workloads. The software program can be further transformed and optimized.

In the *tensorized_GEMM* interface, `sA`, `sB`, and `sC` are buffers in the scratchpad memory. The interface loads a subset of `A` and `B` into the scratchpad memory, performs GEMM computation, and then stores the result of `C` back to the DRAM. Specifically, it distributes the scratchpad buffers to PEs' local memories (`M`, `N`, and `L`) and calls the intrinsic *GEMM_intrin* 32 times. The `k2`, `x2`, and `c2` loops determine how the data are organized and computed. Their order needs to match the accelerator's dataflow, and their strides must be identical to the PE array shape. This interface is highly architecture-specific and must be carefully programmed. Our co-design flow can automatically infer it once the tensorize choice and the hardware intrinsic are determined.

C. Motivational Case Study

The hardware parameters and software optimizations are hard to determine but vital to performance. In this case study, we prototype two GEMM accelerators (`GA_L` and `GA_S`) on FPGAs with different parameters. `GA_L` has a 16 × 16 PE array and a 256 KB scratchpad memory, while `GA_S` has a 8 × 8 PE array and a 128 KB scratchpad. We apply the same set of optimized programs to both architectures.

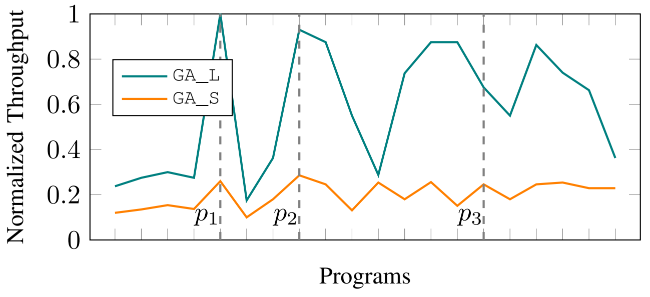


Fig. 2: Normalized throughput when running optimized programs on the two GEMM accelerators.

Figure 2 gives the results. The X-axis represents different software programs, and the Y-axis is the normalized throughput. First, we find that **software optimizations have a huge impact on the final performance**. We highlight three programs `p1` to `p3` in the figure. Programs `p1` and `p2` have the same on-chip computation but different loop orders. Program `p3` has the same loop order with `p1` but more on-chip computation. `p1` achieves the peak performance for `GA_L`, which means loop orders and tensorization all matter. Besides, more on-chip computation does not necessarily result in higher performance (`p3` v.s. `p1`). Furthermore, accelerators prefer different optimizations. As shown, `p2` instead of `p1` achieves the peak performance on `GA_S`. Second, **the design space exhibits complex trade-offs**. `GA_L` has a 4X larger PE array and a 2X larger scratchpad memory than `GA_S`. For our FPGA prototypes, `GA_L` consumes 2.58X area and 1.49X power and achieves 4.27X peak throughput (122.33 MOPS v.s. 28.68 MOPS) compared with `GA_S`. Floor-planned ASIC designs also demonstrate a complex relation [25]. It is hard to choose accelerator parameters to meet the constraints of different scenarios. Also, this example only involves one hardware intrinsic and one workload. Programmers would face greater challenges given various intrinsics and tensor computations. To this end, we propose `HASCO` to explore the hardware and software design spaces in concert.

III. HASCO

Figure 3 presents the workflow of `HASCO`. The user specifies the target computation workload, the hardware generation method, and constraints in the input description to define the co-design process. `HASCO` automatically performs co-design processes, explores design spaces, and outputs solutions to the tensor computation. We divide a co-design process into three steps, as shown in Figure 3:

Step 1: HW/SW Partitioning. `HASCO` first identifies tensorize choices from tensor syntax trees. Each choice represents a HW/SW partition method. All the choices form a partition

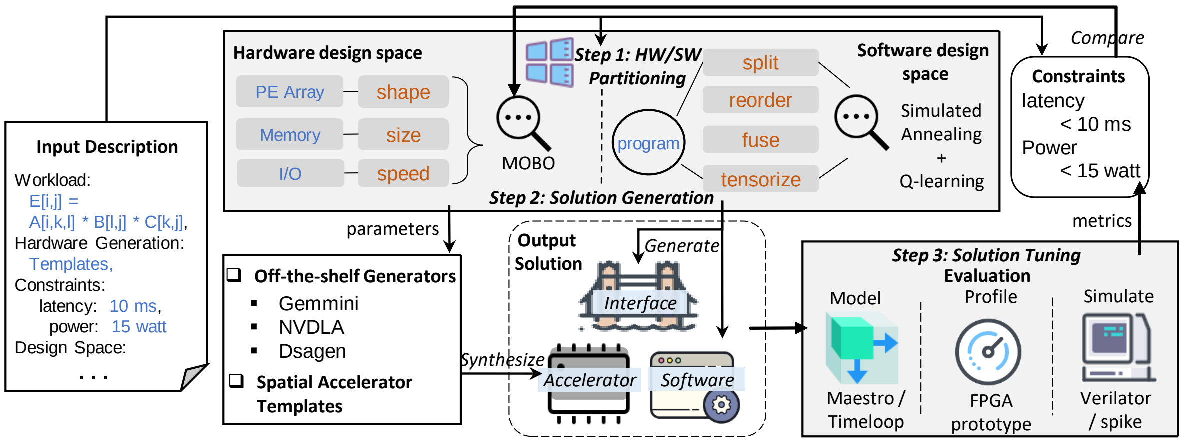


Fig. 3: The workflow of HASCO.

space, which would be explored with software design space in concert.

Step 2: Solution Generation. HASCO generates both the accelerator and software programs through design space exploration (DSE). The hardware DSE and software DSE have distinct objectives and evaluation costs. The hardware DSE concerns multiple objectives like power and area in addition to performance. Each point in the hardware design space corresponds to an accelerator instance. Evaluating design points may require prototyping accelerators, which is a lengthy and expensive process. The software DSE is usually performance-driven and can be fast if we fix the accelerator.

HASCO explores the hardware design space with a Multi-objective Bayesian Optimization (MOBO) algorithm and obtains the Pareto optimal accelerator parameters, as introduced in Section V-B. With the parameters, HASCO instantiates and synthesizes accelerators from off-the-shelf generators or our spatial accelerator templates. For the hardware primitives proposed in Section V-A, we develop templates in a hardware design language, Chisel [7]. Then, for the workload and the accelerator, HASCO explores software optimizations through heuristic and Q-learning algorithms, as detailed in Section VI. Also, it automatically generates the HW/SW interfaces. The accelerator, software programs, and interfaces work together as a holistic solution.

Step 3: Solution Tuning. HASCO evaluates the solution metrics through mathematical models [41], [57], [63], [64] and runtime profiling. If the metrics violate the user constraints, HASCO goes to the second step to find another solution. The metrics would drive the hardware DSE.

IV. HW/SW Partitioning

A. Tensorize Choices

Given the tensor computation and the hardware intrinsic, tensorize choices can be substantial. Take implementing the GEMM computation (L = M × N) with a GEMV intrinsic as an example. Figure 4 gives four tensorize choices. Naturally, we can treat columns or rows of N as the vectors in GEMVs, as the #1 and #2 choices illustrate. However, the choice #2 is

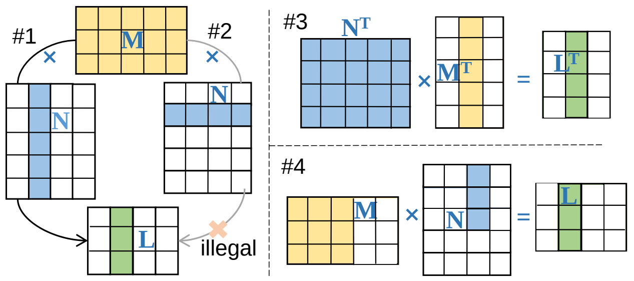


Fig. 4: Four tensorize choices when implementing GEMM with a GEMV intrinsic. The squares are data, and the colored ones form GEMVs.

illegal as it outputs incorrect results. Treating rows of M as the vectors like the choice #3 is also legal if matrix transpositions are allowed. We can further tile M and N into sub-matrices and perform GEMVs, as the choice #4 does. Though the three choices in the figure are legal, they differ in data padding, reuse, and locality. Many other tensorize choices exist for this simple example. As tensor dimensions increase, it is hard for programmers to identify and differentiate all tensorize choices.

We formulate the tensorize problem. Let the target tensor computation be a function $F : \langle I_1, I_2, \dots \rangle \rightarrow O$, where I and O represent input and output tensors, respectively. Let the accelerator intrinsic be another function G. A tensorize choice is to find a set $\{i_j | i_j$ is a sub-tensor of $I_j\}$ and two functions T and S, s.t.

$$\begin{gathered} T = G^n \\ F(I_1, I_2, \dots) = S(T(i_1, i_2, \dots)) \end{gathered} \tag{1}$$

The function T is the tensorize interface, which invokes the hardware intrinsic n times. S is the software program running on the host CPU. All legal T functions form a partition space.

B. Partition Space Generation

We find all legal tensorize choices with a two-step approach. We first define a data structure, tensor syntax trees (TST), to abstract tensor information from a function. In a tensor syntax tree, each node denotes a construct occurring in the function

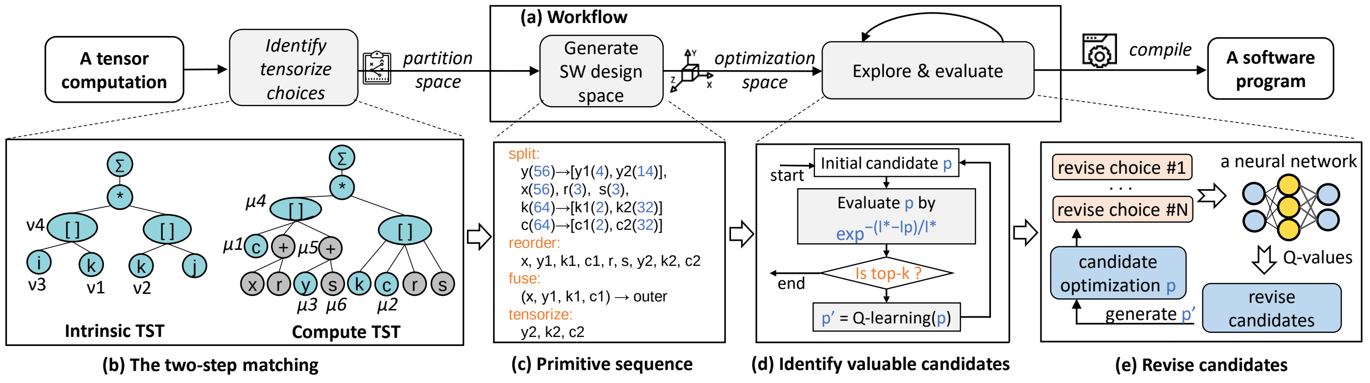


Fig. 5: Schedule a 2D convolution on GEMM accelerators. (a) Software optimization flow. (b) The two-step matching. (c) A primitive sequence representing an optimization for the convolution. (d) The heuristic algorithm. (e) The Q-learning algorithm.

notation. Specifically, each leaf node is the index variable and the range of a tensor dimension, and the intermediate nodes are operations, such as indexing, sum, add, and multiply. The order of leaf nodes indicates the data layout of a tensor. Figure 5(b) illustrates the TSTs of the GEMM intrinsic and the 2D convolution. The intrinsic tree has four leaf nodes representing the four indexes in the notation $\sum M[i,k] * N[k,j]$. The compute tree has 9 leaf nodes corresponding to indexes in $\sum A[c,x+r,y+s] * B[k,c,r,s]$. TSTs explicitly show the dimensions involved in an operation along with the relations between dimensions.

`HASCO` lowers both tensor computations and intrinsics into TSTs, and performs the two-step approach: index matching and structure matching. ***In the index matching***, `HASCO` enumerates the leaf nodes subsets of the compute tree. Given an intrinsic tree Q, a potential leaf subset P must: ① have the same number of leaf nodes as Q does, ② ensure a bijective mapping from each leaf node $\nu \in Q$ to a node $\mu \in P$. For instance, nodes $\nu_1, \nu_2 \in Q$ representing index k in Figure 5(b). If $\nu_1 \leftrightarrow \mu_1$ and $\nu_2 \leftrightarrow \mu_2$, then μ_1, μ_2 must represent the same index (c in this case). ***In the structure matching***, `HASCO` finds the lowest common ancestors (LCAs) of every two nodes in the subset P to match the intermediate nodes of the intrinsic tree Q. In the figure, node μ_4 is the LCA of μ_3 and μ_1. If $\mu_1 \leftrightarrow \nu_1$ and $\mu_3 \leftrightarrow \nu_3$ are determined in the index matching, we require μ_4 to represent the same operation with the LCA of ν_1 and ν_3 (node ν_4 in the figure). Another mapping $\mu_6 \leftrightarrow \nu_1$ that maps index s to k can also pass the index matching. However, node μ_5, the LCA of μ_3 and μ_6, and μ_4 represent different operations, leading to an illegal matching.

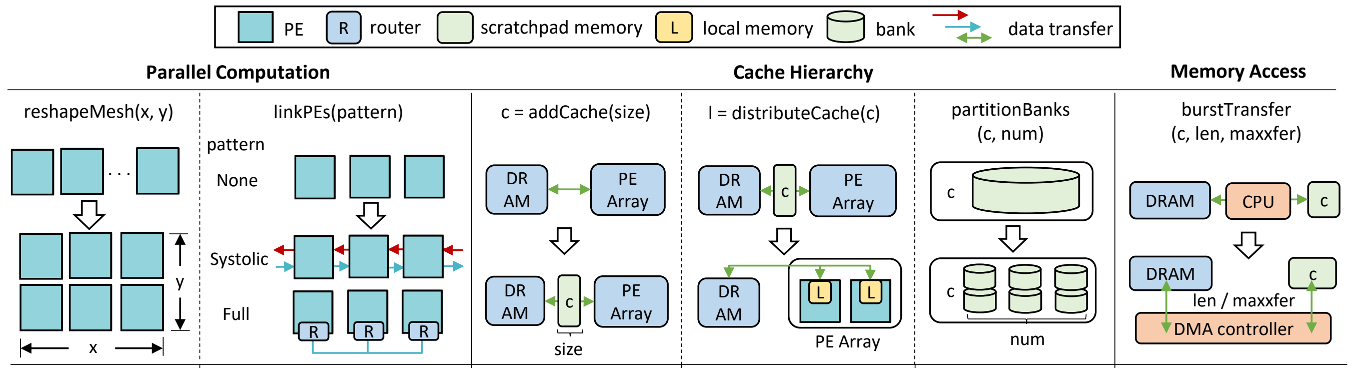
The two-step matching exams whether the intrinsic can implement the sub-workload formed by the leaf subset. It does not restrict the order or range of the matched leaf nodes and allows more tensorize choices. For instance, the order of μ_1 and μ_3 in the compute tree differs from the order of ν_1 and ν_3 in the intrinsic tree. In addition, μ_1 and μ_3 are non-adjacent dimensions. Different node orders allow tensorize choices with data arrangements, like the matrix transpositions of the choice #3 in Figure 4. Furthermore, the matching does not decide the range of each node, such that the size of the sub-workload is flexible. It enables tensorize choices with data tiling, such as the choice #4 in Figure 4. Through the matching, `HASCO` can find all legal tensorize choices forming a HW/SW partition space. We treat this space as a part of the software design space and explore them in concert.

V. Hardware Generation

A. Hardware Primitives and Design Space

The general hardware design space is composed of accelerator parameters, including the PE array shape, memory size, transfer speed, etc. We provide hardware primitives to prune the design space further, as shown in Figure 6. The primitives describe three aspects of spatial accelerators: computation parallelism (*reshapeArray* and *linkPEs*), on-chip cache hierarchy (*addCache*, *distributeCache*, and *partitionBanks*), and off-chip memory access (*burstTransfer*). *reshapeArray* can reshape the 2D PE array and change the intrinsic size. Parallel computation relies on the massive communications between PEs. We abstract common interconnection patterns and use *linkPEs* to specify them. `None` pattern means no data dependency is between PEs. `Systolic` pattern lets PEs receive data from its upstream neighbors and pass results downstream. `Full` pattern is to enable global PE communications with routers. Cache configurations (size, bank number, and distribution) also impact spatial accelerators' overall performance. Programmers can use *addCache* to embed a scratchpad memory shared by all PEs. A scratchpad memory can be partitioned into multiple banks via *partitionBanks* to support concurrent accesses from PEs. It can be further distributed into each PE to form private local memories through *distributeCache*. Last, to fasten off-chip memory accesses, we can use *burstTransfer* to define a DMA controller between a cache and the DRAM to transfer data in the burst mode.

Listing 2 gives an example where we define and prune the design space for a GEMM accelerator. In the code, we first define a design space by specifying the hardware intrinsic (GEMM) and how the accelerator should be generated (from templates). We then prune the PE array shape and scratchpad configurations from the space. The PEs are organized as a 16×16 array (Line 3) and share a 256 KB scratchpad memory



reshapeMesh(x, y)	linkPEs(pattern)	c = addCache(size)	l = distributeCache(c)	partitionBanks (c, num)	burstTransfer (c, len, maxxfer)
Organize PEs into a 2D array with size `x×y`. The array becomes 1D if `x` or `y` is 1.	Specify interconnect `pattern` for PEs, which can be `None`, `Systolic`, and `Full`.	Embed a scratchpad memory `c` shared by all PEs. The cache size is `size` bytes.	Distribute the memory `c` into all PEs, forming local memory `l` within each PE.	Partition a memory `c` into `num` banks to enable concurrent accesses.	Use a DMAC to transfer data in the memory `c`. The burst length is `len`. The maximal transfer size is `maxxfer`.

Fig. 6: Hardware primitives used in HASCO.

(Line 5). The scratchpad is partitioned into four banks to improve data parallelism (Line 6). Uninvolved accelerator parameters like local memory size and burst length still need to be explored.

Listing 2 Define and prune a hardware design space.

```
1 arch = createHWSpace(method = "Templates",
  ↪  intrinsic = L[i, j]: M[i, k] * N[k , j])
2 # prune the array shape
3 arch.reshapeArray(16, 16)
4 # prune the scratchpad size and bank
5 scratchpad = arch.addCache(256 * 1024)
6 arch.partitionBanks(scratchpad, 4)
```

B. Accelerator Parameter Exploration

Next, HASCO starts to explore the accelerator parameters and optimize latency, power, area, and other solution metrics. As the correlations between the parameters and the metrics are complex, we treat the exploration as a black-box optimization problem and formulate it as:

$$\begin{aligned} y &= f(w; x) \\ \chi &= \arg\max_{x \in \mathbb{X}} f(w; x) \end{aligned} \tag{2}$$

where w denotes the target computation workloads, x denotes the accelerator parameters, and y denotes solution metrics. $\mathbb{X}$ is the hardware design space. f is a collection of objective functions that characterize the relationship between the accelerator parameters x, workloads w, and metrics y. $\arg\max$ is to find the parameters x that maximizes y. As solution metrics of interests can be multi-dimensional, the problem is a case of multi-objective optimization. Then $\arg\max$ is to find the Pareto optimal set χ over different metrics.

To solve Equation 2 and find the Pareto set χ of accelerator parameters, we develop a multi-objective Bayesian optimization (MOBO) algorithm [43], [52], [53] in HASCO. Compared with other black-box optimization methods, Bayesian optimization attempts to find the global optimum in a few steps. It incorporates prior information about the objective function f into a surrogate model, which gives the posterior distribution of f. Then Bayesian optimization determines the most promising x that maximizes f with the posterior and an acquisition function. As the optimization proceeds, the prior information about f and the surrogate model keep updating, leading to better posterior distributions and χ.

Algorithm 1 Pseudo-code for the MOBO Algorithm

```
Input $\mathbb{X}$, $f$, $w$, $N$, $\mathbb{M}$, $ac$
 1: Init the prior: $\mathbb{D} \leftarrow$ sample($f$, $\mathbb{X}$)
 2: for $i \leftarrow |\mathbb{D}|$ to $N$ do
 3:     Update the surrogate model $\mathbb{M}$ to fit $\mathbb{D}$
 4:     Calculate the posterior $p(y|x, \mathbb{D})$ with $\mathbb{M}$
 5:     Acquire a promising $x_i$:
        $x_i \leftarrow ac(\mathbb{X}, p(y|x, \mathbb{D}))$
 6:     Evaluate $x_i$: $y_i \leftarrow f(w; x_i)$
 7:     Update the prior: $\mathbb{D} \leftarrow \mathbb{D} \cup (x_i, y_i)$
 8:     Calculate the Pareto set: $\chi \leftarrow$ Pareto set of $\mathbb{D}$
 9: end for
10: Return the current Pareto set $\chi$
```

Algorithm 1 gives the overall procedure of the MOBO algorithm. It first samples and evaluates design points to build a training dataset $\mathbb{D}$ incorporating prior information (Line 1). Then it explores the design space iteratively till the maximal trial number N is reached. At each iteration, it updates the surrogate model $\mathbb{M}$ and computes the posterior distribution $p(y|x, \mathbb{D})$ (Line 3-4). Based on the surrogate model, it selects the design point x_i with the acquisition function ac and evaluates x_i (Line 5-6). Last, it updates the prior dataset with the newly explored design point and calculates the Pareto set χ (Line 7-8). The Pareto set can help us to achieve better trade-offs among different performance constraints in changing scenarios. In practice, we use a Gaussian Process (GP) [62] as the surrogate model and use the hypervolume-based probability of improvement [6] as the acquisition function. The GP model explicitly describes the relation between parameters and metrics is cheap to evaluate.

VI. Software and Interface Generation

Figure 5(a) shows the workflow of the software generation. Given the computation, HASCO builds a software design space with software primitives and explores it using heuristic and Q-learning methods. HASCO also generates interfaces specialized for the target accelerator.

A. Software Primitives and Design Space

We use a set of software primitives, including partitioning (*tensorize*), reordering (*reorder*), splitting (*split*), fusion (*fuse*), etc. Specially, the *tensorize* primitive uses loops to express tensorized sub-workloads. It represents the function T in Equation 1, and the other primitives compose the function S. All the combinations of these primitives form a software design space. Formally, a sequence of software primitives form the skeleton of an optimization, and by setting the factor of each primitive in the sequence we get a concrete optimization. In Figure 5(c), we show a sequence example of optimizing convolutions for GEMM accelerators. The primitive sequence is [*split*, *reorder*, *fuse*, *tensorize*]. It means we first split the `y`, `k`, and `c` loops into six sub-loops and interchange all loops in a specified order. Then we fuse the four outer-most loops into one loop and specify the three inner-most loops denoted by `y2`, `k2`, and `c2` as a tensorized sub-workload.

B. Software Optimization and Generation

Finding the optimal software optimization is an open problem and calls for efficient DSE algorithms. To guarantee the quality of the exploration results, we initialize plenty of candidate optimizations before the exploration starts by randomly generating primitive sequences and factors. Then, we incrementally revise the candidate optimizations to generate new candidates. The revise process may repeat for hundreds of rounds till we find better optimizations. The best optimization would be translated into the final software program by code generation tools [12]. As the exploration proceeds, there can be a great number of candidates in hand, making it time-consuming to revise all of the candidates. Also, to revise each candidate, we have many choices: change the combination of the primitive sequence or change one primitive factor. Exhaustively trying out all the possible revise choices is inefficient.

We determine what and how to revise in two steps. The first step is to find valuable candidates among all candidate optimizations, and the next step selects the most promising revise choice from all possible choices. There are a number of algorithms for implementing the two steps, such as the random algorithm, dynamic programming, and machine learning algorithm. Especially, the two steps cater to the exploration and exploitation in reinforcement learning [48], [68]. Thus, we use a heuristic algorithm and a Q-learning algorithm to implement the two steps, respectively, as shown in Figure 5(d) and (e). ***To identify valuable candidates***, we measure and maintain the latency of each candidate optimization p as l_p, and the lowest latency in history is l^*. Then, the value of p is measured by $exp^{-(l^*-l_p)/l^*}$. The higher the value is, the better the candidate is. We choose the top-k candidates as valuable candidates, where k is a mutable value. ***To revise candidates***, we use Q-learning to generate a new candidate p' for a valuable candidate p. In Q-learning, we use a Q-value to indicate how good each revise choice is. We apply the revise choice with the highest Q-value to p to generate p'. Specially, we use the DQN [49] algorithm to train a 4-layer fully-connected neural network, which predicts Q-values.

TABLE I: Benchmark Tensor Computations.

Tensor Computation	Notation	Test Cases	Compute Complexity
MTTKRP	$D[i,j] = \sum A[i,k,l] * B[l,j] * C[k,j]$	10	255M - 5.9G
TTM	$C[i,j,k] = \sum A[i,j,l] * B[l,k]$	10	16M - 8.6G
2D Conv.	$C[k,x,y] = \sum A[c,x+r,y+s] * B[k,c,r,s]$	10 + CNNs	87M - 3.7G
GEMM	$L[i,j] = \sum M[i,k] * N[k,j]$	10	16K - 4.3G

C. Interface Generation

Hardware intrinsic is only a function abstraction and needs to be implemented as accelerator instructions. Different accelerators can have the same hardware intrinsic. For instance, both Gemmini and VTA [50] use GEMM as the intrinsic function. As spatial accelerators expose their ISAs with a granularity that differs from design to design, we should specialize the tensorize interface calling the intrinsics for the target accelerator.

HASCO automatically translates the interface into instructions once the software DSE makes the tensorize choice. There are two basic types of instructions in spatial accelerators: the data movement instructions move data between the scratchpad memory and the DRAM, and the compute instructions invoke computations on the PE array. Such ISAs suggest the interfaces should explicitly manage scratchpad data and call the intrinsic function. HASCO inserts the data movement instructions before and after the intrinsic call to prepare the scratchpad. Then it replaces the intrinsic call with the compute instructions. For instance, the GEMM intrinsic is replaced with the *compute_accumulated* instruction of Gemmini, which controls the PE array to perform 16×16 multiply-add operations.

VII. Experiments

A. Experimental Setup

Benchmarks. We use a set of tensor computations as our benchmarks, as shown in Table I. MTTKRP and TTM are core computations in tensor decomposition. GEMM and 2D convolution are used in convolutional neural networks (CNNs). We also collect workloads from modern CNNs as the 2D convolutions in Table II, including ResNet-50 [29], MobileNet [31], and Xception [18].

Hardware. In our evaluation, we use four hardware intrinsics: dot product (DOT: $C = \sum A[i] * B[i]$), GEMV ($C[i] = \sum A[i,j] * B[j]$), GEMM, and 2D convolution (CONV2D). We employ Gemmini [25] to generate GEMM accelerators. We use the Rocket Chip generator [4] and our Chisel templates to build accelerators with the other intrinsics. For simplicity, we refer to accelerators with GEMM and CONV2D intrinsics as GEMMCore and ConvCore, respectively.

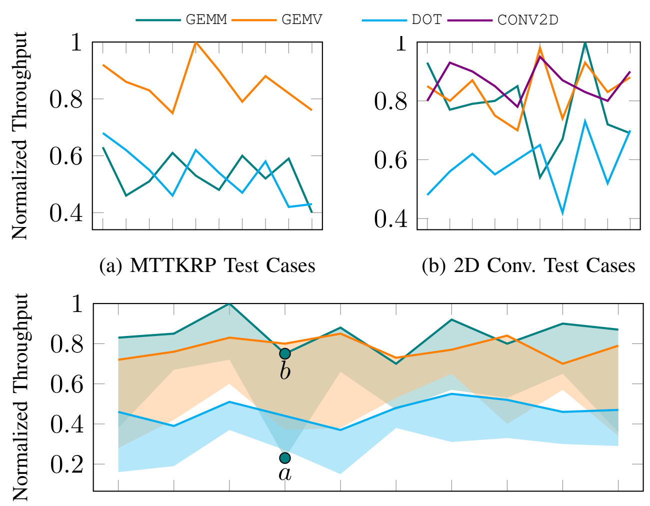


Fig. 7: Normalized throughput results of different tensor computations and hardware intrinsics.

Methodology. We first analyze different intrinsics and tensorize choices. Then we demonstrate the efficiency of our hardware DSE with comparisons and detailed analysis. For the software, we compare `HASCO` with AutoTVM [13] and an accelerator library [25]. The library implements hand-tuned computations, such as matrix multiplication of any size, CNNs, and non-linear activations. It carefully splits and reorders loops in the computations and calls the GEMM intrinsic. Last, we discuss the overall benefits brought by co-design.

Metrics. In the hardware DSE evaluation, we use the models provided by Maestro [41], which evaluate the latency, power, and area of accelerators according to compute and memory transactions. We synthesize the accelerators with Xilinx Vivado tools [76] in the software evaluation and prototype them as Rocket Chip SoCs on a Xilinx VU9P FPGA board. The SoC frequencies are all set as 200 MHz. We time the latency, calculate the throughput, and evaluate the chip power with Vivado tools.

B. Tensorize Choice and Hardware Intrinsic

Here, we compare the four hardware intrinsics (DOT, GEMV, GEMM, and CONV2D) when optimizing the benchmarks' throughput. Specifically, we specify an array of 64 PEs and a 256 KB scratchpad memory with our hardware primitives for all accelerators and give them different intrinsic functions. With tensor syntax trees and the two-step matching, `HASCO` can divide all the tensor computations into GEMM, GEMV, and DOT sub-workloads. Only 2D convolutions can be tiled into CONV2D sub-workloads.

Figure 7 compares the throughput results of the four intrinsics. The X-axis represents the test cases of each tensor computation, and the Y-axis is the normalized throughput. We draw two conclusions. First, **different tensor computations prefer different hardware intrinsics**. In general, an intrinsic is more efficient if it is dedicated to the tensor computation. As the figure shows, in general, TTM and GEMM prefer the GEMM intrinsic, and 2D convolution prefers the CONV2D intrinsic. Dedicated accelerators provide more data reuse opportunities and achieve higher performance. Though the DOT intrinsic is the most general, it reuses no tensor data and achieves low performance. MTTKRP is an exception, which prefers the GEMV intrinsic instead of GEMM. To illustrate the reason clearly, we treat MTTKRP as two stages: $E[i,k,j] = \sum A[i,k,l] * B[l,j]$ and $D[i,j] = \sum E[i,k,j] * C[k,j]$. Only the first $A \times B$ stage can be divided into GEMM sub-workloads and accelerated by the GEMM intrinsic. Nevertheless, `HASCO` can find GEMV sub-workloads in the two stages from tensor syntax trees. In other words, the GEMM intrinsic accelerates three loops represented by `i/k`, `l`, and `j` in MTTKRP, while the GEMV intrinsic benefits four loops represented by `i`, `k`, `l`, and `j`. In addition, an intrinsic shows different efficiencies when accelerating the same tensor computation.

Second, **different tensorize choices have different impacts on the target metrics** (throughput in this case). For each case, `HASCO` can find a great number of tensorize choices and explore them efficiently. In Figure 7(c), we use a colored area to represent the throughput range of the tensorize choices for the same intrinsic. Data reuse, locality, and padding all contribute to the throughput variance. To illustrate the variance clearly, we mark two tensorize choices a and b in the figure. Choice a divides tensor A in TTM along the last two dimensions `j` and `l` so that the sub-tensor can be accessed continuously in the DRAM. Choice b divides A along the `i` and `l` dimensions, leading to non-continuous data access. Besides, the tensorize interface of a calls the GEMM intrinsic exactly 64 times, while the tensorize interface of b requires data padding before calling the intrinsic. As a result, the throughput results of the two choices have a 3.26X difference. Overall, co-design is vital to solving tensor computations as both the hardware intrinsic and tensorize choice are hard to determine.

C. Hardware DSE Evaluation

Ground Truth. We first collect metrics of ConvCore accelerators with models as the ground truth of the hardware DSE. We limit the design space of this ground truth experiment by decreasing workloads and accelerator parameters. The workload consists of six convolutions from Xception, whose computations range from 86.7 MOPs to 454.2 MOPs. The parameters to be explored are the PE array shape and bank number of the scratchpad memory. All software programs are generated by `HASCO`.

Figure 8 illustrates the ground truth data. When designing accelerators, the correlations between the latency, power, and area data are not trivial. Figure 8(c) shows a positive correlation between the normalized power and area data, as a larger design spends more energy on computations and PE communications. However, the normalized power and area data can vary dramatically under the same latency constraint, as Figure 8(a) and (b) show. For instance, the power data range from 207.46 mW to 25136.7 mW under the 0.05 normalized latency constraint, leading to a 121.16X difference. Hence,

TABLE II: Pareto solutions of the random search, NSGAII, and MOBO methods. L: latency. P: power.

Workloads & Constraints	Intrinsic	Latency (cycles)			Power (mW)			Area(μm^2)		
		Random	NSGAII	MOBO	Random	NSGAII	MOBO	Random	NSGAII	MOBO
ResNet: L $\leq$2E9, P$\leq$1E4	GEMM	3.972E8	3.528E8	3.528E8	3293.5	3099.8	3099.8	9.470E7	7.005E7	7.005E7
	CONV2D	4.439E8	3.863E8	3.411E8	3298.65	2989.44	2403.87	7.469E7	6.551E7	5.438E7
MobileNet: L $\leq$ 1E10, P$\leq$1E4	GEMM	3.062E9	1.923E9	1.923E9	4068.14	3874.18	3487.17	1.933E8	1.686E8	1.193E8
	CONV2D	2.135E9	2.335E9	1.813E9	3811.98	3589.21	3589.21	1.304E8	1.229E8	1.229E8
Xception: L $\leq$1E11, P$\leq$1E4	GEMM	2.286E10	2.286E10	2.286E10	4874.48	4355.45	3874.18	2.716E8	2.425E8	1.686E8
	CONV2D	2.375E10	2.177E10	2.177E10	4013.59	4456.74	4013.59	1.756E8	1.943E8	1.756E8

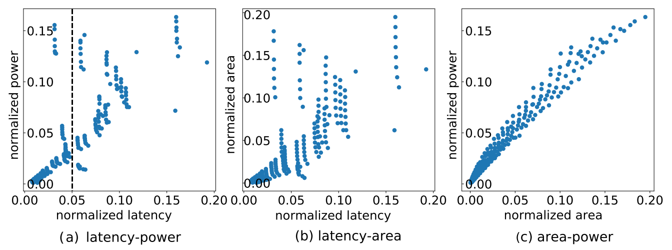


Fig. 8: Correlations between latency, power, and area data collected with Maestro [41].

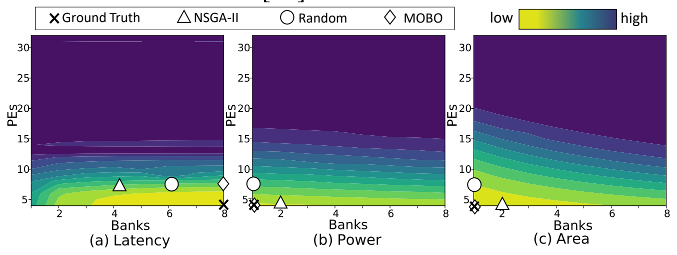


Fig. 9: Correlations between the ground truth data and accelerator parameters.

finding the Pareto solutions to tensor computations is vital to energy-efficient designs.

We further analyze how the accelerator parameters impact the solution metrics. Figure 9 illustrates the correlations between the ground truth data and the parameters. The X-axis represents the bank numbers ranging from one to eight, and the Y-axis represents the PE array shape ranging from 4×4 to 32×32. The color of each point indicates a normalized latency, power, or area value. As Figure 9(b) and (c) show, power and area data increase as the PE number and the bank number increase. This observation is natural, for the PEs and scratchpad consume more energy and areas. Normally, the latency data show negative correlations with the PE number and the bank number. As the PEs and banks become over-provisioned, the contour color would remain the same. However, in this case, the latency increases when the generated convolution accelerators have more PEs and banks, as Figure 9(a) shows. The reason is that the convolutions used in this ground truth experiment have small computations and limited parallelism. Small PE arrays are enough to process them efficiently. As the PE number keeps growing, the latency of one intrinsic call also increases. Besides, more data are padded to fill the PE array, leading to wasted computations and an increase in overall latency.

Comparisons. We compare the MOBO method used in HASCO with the random search and the NSGAII genetic

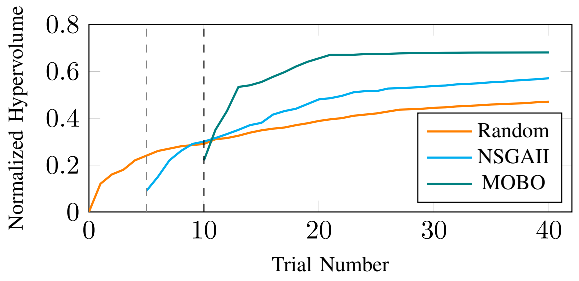


Fig. 10: Normalized hypervolume improvements of random search, NSGAII, and MOBO.

algorithm [23]. We use HASCO to generate software and use the three methods to optimize the latency, power, and area of ConvCore simultaneously within 20 trials[1]. We mark the final solutions in Figure 9. MOBO can find the Pareto optimal set. The random search achieves 74.8% latency, 43.8% power, and 41.6% area compared with the Pareto optimal set. NSGAII achieves 80.5% latency, 95.2% power, and 62.2% area.

We then use more CNNs and intrinsics (GEMM and CONV2D) for comparisons. In the evaluations[2], we constrain the latency and power and use the three methods to find the Pareto solutions. Table II lists the constraints and results. MOBO always outperforms the random search and NSGAII in our evaluations. It achieves 1.215X average latency improvement, 1.154X average power reduction, and 1.336X average area reduction compared with the random search. In some cases using the GEMM intrinsic, NSGAII can give the same solutions as MOBO does. However, in cases using the CONV2D intrinsic, NSGAII fails to converge to good parameters as there are more tensorize choices and a larger hardware design space. MOBO converges rapidly instead.

We calculate the hypervolume for the case using ResNet and the GEMM intrinsic to show the three methods' convergence. In multi-objective optimizations, hypervolume is an important indicator measuring the size of the space dominated by a set of design points. The closer the design points are to the Pareto front, and the more likely they are distributed along the Pareto front, the larger the hypervolume becomes. As Figure 10 shows, MOBO quickly improves its hypervolume after the initialization phase. It has surpassed the final results for both the NSGAII and random search algorithms at trial 16. The reason is that MOBO reduces the number of redundant

[1]MOBO uses five samples as its prior and iterates 15 times.

[2]The maximal trial number of all methods is set as 40. The population size of NSGAII is 5. The sample size of MOBO is 10.

TABLE III: HASCO results when scaling the power constraints.

Scenario	CNNs	Baseline-GEMMCore: separated designs				HASCO-GEMMCore: co-design				HASCO-ConvCore: co-design			
		PEs	Memory (KB)	Banks	latency (ms)	PEs	Memory (KB)	Banks	latency (ms)	PEs	Memory (KB)	Banks	latency (ms)
Edge (power: 2 W)	ResNet	64	256	4	12321.8	64	256	6	8547.8	144	320	8	4673.7
	MobileNet	64	256	4	56457.6	64	512	8	42977.1	121	512	6	24273.5
	Xception	64	256	4	707105.1	64	512	8	544023.8	144	512	8	318485.6
Cloud (power: 20 W)	ResNet	4096	1024	4	260.5	4096	1024	8	197.2	4096	1536	8	195.3
	MobileNet	4096	1024	4	1456.9	4096	1024	8	1020.5	4096	1024	8	901.5
	Xception	4096	1024	4	15706.3	4096	1024	8	12548.9	4096	1536	8	11594.4

evaluations by building a statistical model based on earlier observations.

Notably, each trial takes minutes to hours in modeling, simulating, implementing, and profiling accelerators. MOBO achieves **a 1.19X hypervolume improvement** compared with NSGAII, meaning MOBO finds more design points close to the Pareto front. MOBO uses **2.5X fewer trials** to achieve the final hypervolume of NSGAII, significantly reducing the co-design cost.

D. Software DSE Evaluation

We first demonstrate the software quality by comparing HASCO with the library proposed in [25]. We prototype a GEMMCore on the FPGA, which has a 16 $\times$ 16 PE array and a 256 KB scratchpad. Then we use the library to run ResNet on the accelerator. The library converts 2D convolutions to GEMMs and invokes the GEMM intrinsic. Specifically, it always unfolds the operand tensors into matrices (*im2col*), performs GEMMs, and folds the result matrix back to a tensor (*col2im*) [33]. GEMMs converted from convolutions are divided into sub-workloads by loop splitting. The split factors depend on the array shape and scratchpad size.

We use HASCO to optimize software for ResNet and the accelerator. The HASCO-generated software outperforms the library by more than 2X in 18 cases out of ResNet's 53 convolution workloads and provides **a 3.17X average latency reduction**. We illustrate the first 20 cases in Figure 11. As the library converts 2D convolutions to GEMMs, the convolution becomes $C[k, x \times y] = \sum A[c \times r \times s, x \times y] * B[k, c \times r \times s]$. This conversion can be omitted only if dimensions r and s are reduced. It is an algorithm-level optimization beyond the scope of this paper. Though the conversion is a natural way to call GEMM intrinsics, it introduces significant latency overheads. As Figure 11 shows, once the *im2col* and *col2im* are performed, their overhead dominates the overall latency of the workload. Besides, the conversion requires a much larger DRAM region to store the intermediate matrices.

In contrast, HASCO customizes the tensorize interface for each GEMM workload. Instead of converting convolutions to GEMMs, it directly partitions a convolution workload along different dimensions according to operand tensors' shapes. By analyzing tensor syntax trees, HASCO determines k, x/y, and another dimension should be partitioned. For convolutions where dimension c is large, HASCO would choose c as the last partition dimension to provide enough data parallelism. Otherwise, dimension r/s would be partitioned to reduce wasted computations as much as possible.

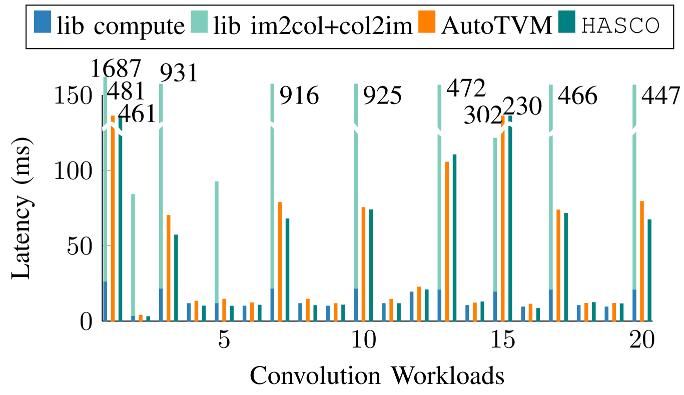


Fig. 11: Comparisons between ResNet software.

For a fair comparison, we also use AutoTVM [13] to optimize software by directly partitioning convolutions. AutoTVM requires users to manually make tensorize choices and write primitive templates for each tensor computation. Besides, it only optimizes the size of tensorized sub-workloads. For ResNet and the GEMMCore, **HASCO outperforms AutoTVM by 1.21X**. The improvement is because HASCO systematically explores tensorize choices and software primitives, while AutoTVM relies on static templates and fixed tensorize choices.

E. Overall Solution Analysis

Last, we evaluate our hardware and software DSE algorithms together and demonstrate the overall benefits brought by co-design. We scale the power constraints to simulate cloud (20 W) and edge (2 W) scenarios, respectively. Under the constraints, we use HASCO to generate GEMMCore accelerators and software in 20 co-design iterations. Table III lists the accelerator parameters and latency results.

The baseline solution employs the traditional methodology, which decouples the hardware and software development. For the baseline hardware, we employ two GEMMCore accelerators with the classic parameters listed in Table III for the two scenarios, respectively. If we use the library [25] as our baseline software, HASCO can achieve a 2.14X average latency reduction. For fair comparisons and improvement breakdowns, we use AutoTVM to generate the baseline software. As the table shows, HASCO solutions achieve **1.25X to 1.44X latency reduction** in the two scenarios compared with the baseline solutions. The hardware DSE of HASCO provides 29.53% of the latency reduction. As the table shows, the GEMMCore accelerators generated by HASCO tends to use more scratchpad memories and banks, which enable larger tensorized sub-workloads and more data reuse. For each scenario, the HASCO accelerator uses the same number of PEs as the baseline. The reason is GEMMCore constrains its PE array shape to be $2^n \times 2^n$. Under this PE constraint and the power constraint, MOBO converges to the optimal PE array shape. The software optimization of HASCO provides the rest 70.47% latency reduction.

We also use HASCO to co-design ConvCore accelerators and the software. The results are also given in Table III. The HASCO-ConvCore solutions further reduces the latency by 1.42X on average compared with the HASCO-GEMMCore solutions. The improvement is two-fold. For one thing, the CONV2D intrinsic is dedicated to convolutions and provides more data reuse opportunities than the GEMM intrinsic. For another, unlike GEMMCore, ConvCore does not restrict the PE array shape, giving HASCO opportunities to use more PEs under the same power constraint.

VIII. Related Works

Table IV lists the related works of HASCO:

Hardware Acceleration. Many hardware accelerators have been proposed for DNNs and tensor computations. Previous works [14]–[16], [24], [27], [28], [34], [44], [45], [60], [78]–[81], [83] target the most common computations in DNNs, including convolutions and matrix multiplications. Previous works [26], [30], [42], [55], [61], [67] propose flexible architectures for more general tensor computations. All these works generate chips with fixed powers and areas and cannot be deployed in various scenarios. Recently, hardware generators are proposed to provide more efficiency. Gemmini [25] generates systolic array accelerators for matrix multiplications. NVDLA [22] generates deep learning inference accelerators scaled across a wide range of IoT devices. MAGNet [72] and AutoDNNchip [77] are generator infrastructures generating DNN accelerators. MAGNet is based on highly configurable PEs and supports multiple dataflows. AutoDNNchip instantiates IPs to generate accelerators rapidly. DSAGEN [74] extracts information about parallelism and concurrency from the target workload and generates specialized spatial accelerators from scratch. VTA [50] designs parametrizable architecture, where memories, datatypes, and sizes of the GEMM intrinsic can be customized. It relies on AutoTVM [13] to optimize software, meaning users make tensorize choices. HASCO leverages off-the-shelf hardware generators.

Design Space Exploration. For software optimizations, loop transformations [9], [10], [47], [59], [73], [75] have been studied for decades. However, traditional optimization flows rely on programmers to determine loop transformations. Recently, the advance in machine learning enables automated loop transformations. Halide auto-scheduler [2] uses tree searching and random programs in the exploration and mainly targets image processing. PlaidML [19] and Tensor Comprehensions [71] use an analytical model and polyhedral models for software DSE, respectively. Halide, PlaidML, and Tensor Comprehensions only support limited heterogeneous hardware platforms. AutoTVM [13] uses XGBoost [11] for exploration

TABLE IV: Related works.

Domain	Related Works
Architecture	DNN: Eyeriss [15], Diannao family [14], [16], [24], [44], Cambricon family [45], [80], [81], [83], TPU [34], EIE [28], ESE [27], Thinker family [78], [79], Plasticine [60]
	Tensor: DySER [26], MAERI [42], ExTensor [30], Tensaurus [67], OuterSPACE [55], SIGMA [61]
Generators	Gemmini [25], NVDLA [22], MAGNet [72], AutoDNNchip [77], DSAGEN [74], VTA [50]
DSE	HW: MAGNet [72], AutoDNNchip [77], DSAGEN [74], ConfuciuX [36]
	SW: Halide auto-scheduler [2], AutoTVM [13], FlexTensor [82], PlaidML [19], TACO [38], Tensor Comprehensions [71]

and supports a wider range of hardware. It requires programmers to develop primitive templates and make tensorize choices. FlexTensor [82] proposes a fully-automatic method to optimize programs. However, it only supports general programming platforms. HASCO targets spatial accelerators with different hardware intrinsics and tensorize choices.

For hardware design space exploration, many hardware generators design DSE methods for their accelerators, such as [72], [74], [77]. For instance, AutoDNNchip [77] also builds models to predict accelerator metrics based on DNN parameters. It relies on design space pruning to enable fast exploration. DSAGEN [74] iteratively optimizes a single objective until the objective converges. ConfuciuX [36] uses reinforcement learning and genetic algorithms to search the number of PEs assigned to different DNN layers. It leverages accelerators in a fine-grained way, which requires architecture supports. Besides, it optimizes one objective at a time. Compared with these works, HASCO provides a multi-objective DSE approach serving a class of spatial accelerators. More importantly, previous DSE works decouple the hardware and hardware spaces and target one space, while HASCO co-explore the two spaces in concert.

IX. Conclusion

Though HW/SW co-design can generate high-quality solutions to tensor computation, it faces two fundamental challenges. First, the functionality gap between tensor computations and accelerator intrinsics calls for a tensorize interface setting the hardware and software apart. However, it is hard to identify and differentiate substantial tensorize choices. Second, the overall design space composed of tensorize choices, software optimizations, and accelerator parameters is too huge to be efficiently explored. In this work, we propose HASCO as an agile co-design approach. HASCO automatically identifies tensorize choices from tensor syntax trees. It uses heuristic and Q-learning algorithms for software optimization. It uses multi-objective Bayesian optimization to explore hardware parameters. Putting these techniques together, HASCO provides significant improvements in solution quality and DSE efficiency.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in Symposium on Operating Systems Design and Implementation, 2016.

[2] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to optimize halide with tree search and random programs," ACM Trans. Graph., vol. 38, no. 4, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3306346.3322967

[3] S. Alkalay, H. Angepat, A. Caulfield, E. Chung, O. Firestein, M. Haselman, S. Heil, K. Holohan, M. Humphrey, T. Juhasz et al., "Agile co-design for a reconfigurable datacenter," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016, pp. 15–15.

[4] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton et al., "Chipyard: Integrated design, simulation, and implementation framework for custom socs," IEEE Micro, 2020.

[5] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky, "Tensor decompositions for learning latent variable models," The Journal of Machine Learning Research, 2014.

[6] A. Auger, J. Bader, D. Brockhoff, and E. Zitzler, "Hypervolume-based multiobjective optimization: Theoretical foundations and practical implications," Theoretical Computer Science, vol. 425, pp. 75–103, 2012.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in DAC Design Automation Conference 2012. IEEE, 2012, pp. 1212–1221.

[8] F. Balarin, P. Giusto, A. Jurecska, M. Chiodo, C. Passerone, E. Sentovich, H. Hsieh, L. Lavagno, B. Tabbara, A. Sangiovanni-Vincentelli et al., Hardware-software co-design of embedded systems: the POLIS approach. Springer Science & Business Media, 1997.

[9] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004., 2004, pp. 7–16.

[10] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008, pp. 101–113.

[11] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, 2016, pp. 785–794.

[12] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: End-to-end optimization stack for deep learning," in SysML Conference, 2018.

[13] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in Advances in Neural Information Processing Systems, 2018, pp. 3389–3400.

[14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," ACM Sigplan Notices, 2014.

[15] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE Journal of Solid-State Circuits, 2016.

[16] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in Proceedings of the International Symposium on Microarchitecture, 2014.

[17] S. R. Chinnamsetty, M. Espig, B. N. Khoromskij, W. Hackbusch, and H.-J. Flad, "Tensor product approximation with optimal rank in quantum chemistry," The Journal of chemical physics, vol. 127, no. 8, p. 084110, 2007.

[18] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 1251–1258.

[19] I. Corporation. (2020) Plaidml. [Online]. Available: https://ai.intel.com/plaidml

[20] N. Corporation. (2020) Nvidia cudnn. [Online]. Available: https://developer.nvidia.com/cudnn

[21] N. Corporation. (2020) Nvidia cutlass. [Online]. Available: https://github.com/NVIDIA/cutlass

[22] N. Corporation. (2020) Nvidia deep learning accelerator (nvdla). [Online]. Available: http://nvdla.org/

[23] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182–197, 2002.

[24] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in ACM SIGARCH Computer Architecture News, 2015.

[25] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister et al., "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," arXiv preprint arXiv:1911.09925, 2019.

[26] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," IEEE Micro, vol. 32, no. 5, pp. 38–51, 2012.

[27] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, and Y. Wang, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in Proceedings of the International Symposium on Field Programmable Gate Arrays, 2017.

[28] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in Proceedings of the International Symposium on Computer Architecture. IEEE, 2016.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[30] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in Proceedings of the International Symposium on Microarchitecture, 2019.

[31] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," CoRR, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[32] Intel Corporation. (2020) Intel mkl-dnn. [Online]. Available: https://software.intel.com/mkl

[33] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in Proceedings of the 22nd ACM international conference on Multimedia, 2014, pp. 675–678.

[34] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 1–12.

[35] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," Communications of the ACM, vol. 61, no. 9, pp. 50–59, 2018.

[36] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning," in Proceedings of the 53nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020. ACM, 2020.

[37] D. Y. Kim, J. M. Kim, H. Jang, J. Jeong, and J. W. Lee, "A neural network accelerator for mobile application processors," IEEE Transactions on Consumer Electronics, vol. 61, no. 4, pp. 555–563, 2015.

[38] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, p. 77, 2017.

[39] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," SIAM review, 2009.

[40] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in 2008 Eighth IEEE international conference on data mining. IEEE, 2008, pp. 363–372.

[41] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in Proceedings of the International Symposium on Microarchitecture, 2019, pp. 754–768.

[42] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in ACM SIGPLAN Notices, vol. 53. ACM, 2018, pp. 461–475.

[43] M. Laumanns and J. Ocenasek, "Bayesian optimization algorithms for multi-objective optimization," in International Conference on Parallel Problem Solving from Nature. Springer, 2002, pp. 298–307.

[44] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in ACM SIGARCH Computer Architecture News, 2015.

[45] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). IEEE, 2016, pp. 393–405.

[46] M. Mahmoud, I. Edo, A. H. Zadeh, O. M. Awad, G. Pekhimenko, J. Albericio, and A. Moshovos, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020, pp. 781–795.

[47] K. S. Mckinley, S. Carr, and C. Tseng, "Improving data locality with loop transformations," ACM Transactions on Programming Languages and Systems, vol. 18, no. 4, pp. 424–453, 1996.

[48] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[49] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," nature, vol. 518, no. 7540, pp. 529–533, 2015.

[50] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardware–software blueprint for flexible deep learning specialization," IEEE Micro, vol. 39, no. 5, pp. 8–16, 2019.

[51] M. Mørup, "Applications of tensor (multiway array) factorizations and decompositions in data mining," Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 1, no. 1, pp. 24–40, 2011.

[52] L. Nardi, D. Koeplinger, and K. Olukotun, "Practical design space exploration," in 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2019, pp. 347–358.

[53] L. Nardi, A. Souza, D. Koeplinger, and K. Olukotun, "Hypermapper: a practical design space exploration framework," in 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2019, pp. 425–426.

[54] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. P. Jouppi, and D. Patterson, "Google's training chips revealed: Tpuv2 and tpuv3," in 2020 IEEE Hot Chips 32 Symposium (HCS). IEEE Computer Society, 2020, pp. 1–70.

[55] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in Proceedings of the International Symposium on High Performance Computer Architecture, 2018.

[56] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Tensors for data mining and data fusion: Models, applications, and scalable algorithms," ACM Transactions on Intelligent Systems and Technology (TIST), vol. 8, no. 2, pp. 1–44, 2016.

[57] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019, pp. 304–315.

[58] A. H. plc. (2020) Arm compute library. [Online]. Available: https://www.arm.com/why-arm/technologies/compute-library

[59] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop transformations: convexity, pruning and optimization," ACM SIGPLAN Notices, vol. 46, no. 1, pp. 549–562, 2011.

[60] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017, pp. 389–402.

[61] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in Proceedings of the International Symposium on High Performance Computer Architecture, 2020.

[62] C. E. Rasmussen, "Gaussian processes in machine learning," in Summer School on Machine Learning. Springer, 2003, pp. 63–71.

[63] Y. S. Shao, B. Reagen, G. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014, pp. 97–108.

[64] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, pp. 1–12.

[65] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in Proceedings of the Workshop on Irregular Applications: Architectures and Algorithms, 2015.

[66] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in 2015 IEEE International Parallel and Distributed Processing Symposium. IEEE, 2015, pp. 61–70.

[67] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2020, pp. 689–702.

[68] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.

[69] S. Szalay, M. Pfeffer, V. Murg, G. Barcza, F. Verstraete, R. Schneider, and Ö. Legeza, "Tensor product methods and entanglement optimization for ab initio quantum chemistry," International Journal of Quantum Chemistry, vol. 115, no. 19, pp. 1342–1391, 2015.

[70] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference," in 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020, pp. 1–6.

[71] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. (2018) Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions.

[72] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. R. Pinckney, P. Raina et al., "Magnet: A modular accelerator generator for neural networks." in ICCAD, 2019, pp. 1–8.

[73] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, no. 4, pp. 1–23, 2013.

[74] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 268–281.

[75] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29. IEEE, 1996, pp. 274–286.

[76] Xilinx.com. (2020) Vivado design suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[77] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin, "Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics," in The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2020, pp. 40–50.

[78] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, S. Zheng, T. Lu, J. Gu, L. Liu, and S. Wei, “A high energy efficient reconfigurable hybrid neural network processor for deep learning applications,” IEEE Journal of Solid-State Circuits, vol. 53, no. 4, pp. 968–982, 2017.

[79] S. Yin, P. Ouyang, J. Yang, T. Lu, X. Li, L. Liu, and S. Wei, “An energy-efficient reconfigurable processor for binary-and ternary-weight neural networks with flexible data bit width,” IEEE Journal of Solid-State Circuits, vol. 54, no. 4, pp. 1120–1136, 2018.

[80] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, “Cambricon-x: An accelerator for sparse neural networks,” in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2016, pp. 1–12.

[81] Y. Zhao, Z. Du, Q. Guo, S. Liu, L. Li, Z. Xu, T. Chen, and Y. Chen, “Cambricon-f: machine learning computers with fractal von neumann architecture,” in 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2019, pp. 788–801.

[82] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, “Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system,” in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 859–873.

[83] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, “Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach,” in The International Symposium on Microarchitecture. IEEE, 2018.