AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction

Size Zheng
Peking University
Beijing, China
zhengsz@pku.edu.cn

Renze Chen
Peking University
Beijing, China
crz@pku.edu.cn

Anjiang Wei*
Stanford University
United States
anjiang@stanford.edu

Yicheng Jin
Peking University
Beijing, China
yicheng.jin@pku.edu.cn

Qin Han
Peking University
Beijing, China
hanqin@pku.edu.cn

Liqiang Lu
Peking University
Beijing, China
liqianglu@pku.edu.cn

Bingyang Wu
Peking University
Beijing, China
bingyangwu@pku.edu.cn

Xiuhong Li
Sensetime Research &
Shanghai AI Lab
China
lixiuhong@sensetime.com

Shengen Yan
SenseTime Research
Beijing, China
yanshengen@sensetime.com

Yun Liang†
Peking University
Beijing, China
liqianglu@pku.edu.cn

ABSTRACT

Hardware specialization is a promising trend to sustain performance growth. Spatial hardware accelerators that employ specialized and hierarchical computation and memory resources have recently shown high performance gains for tensor applications such as deep learning, scientific computing, and data mining. To harness the power of these hardware accelerators, programmers have to use specialized instructions with certain hardware constraints. However, these hardware accelerators and instructions are quite new and there is a lack of understanding of the hardware abstraction, performance optimization space, and automatic methodologies to explore the space. Existing compilers use hand-tuned computation implementations and optimization templates, resulting in sub-optimal performance and heavy development costs.

In this paper, we propose AMOS, which is an automatic compilation framework for spatial hardware accelerators. Central to this framework is the hardware abstraction that not only clearly specifies the behavior of spatial hardware instructions, but also formally defines the mapping problem from software to hardware. Based on the abstraction, we develop algorithms and performance models to explore various mappings automatically. Finally, we build a compilation framework that uses the hardware abstraction as compiler intermediate representation (IR), explores both compute mappings and memory mappings, and generates high-performance code for different hardware backends. Our experiments show that AMOS achieves more than 2.50× speedup to hand-optimized libraries on Tensor Core, 1.37× speedup to TVM on vector units of Intel CPU for AVX-512, and up to 25.04× speedup to AutoTVM on dot units of Mali GPU. The source code of AMOS is publicly available.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; • **Computer systems organization → Special purpose systems**; • **Hardware →** *Emerging architectures.*

KEYWORDS

spatial accelerators, code generation, mapping, tensor computations

ACM Reference Format:
Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527440

*Work done while the author was a student at Peking University.
†Corresponding author.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527440

1 INTRODUCTION

Recently, we have witnessed the success of domain-specific architecture in various application domains. Among the various hardware

accelerators, spatial architecture that organizes a large number of processing elements in a structural and hierarchical manner is demonstrated to be extraordinarily efficient for various tensor applications in deep learning [6, 15, 20, 50], scientific computing [36], and data mining. The spatial architecture naturally aligns with the tensor computations with regular loop structures and memory access patterns, which are often represented using nested loops. For example, Google TPU [27] and Gemmini [17] specialized for GEMM computations use systolic array architecture. Moreover, the spatial architectures continue to evolve with new tensor applications [12, 35, 61].

Depending on the degree of generality and programmability, hardware accelerators can be designed differently and thus require different mapping strategies. We divide the existing mapping flows for spatial accelerators into two categories: **hardware-aware** and **ISA-aware**. For specific domains, the hardware design and software mapping can be coupled together by providing a domain-specific hardware and software interface [28, 33, 47, 62]. More clearly, the compiler/mapper is aware of the hardware architecture details such as the number of processing elements (PEs) and their interconnection, and then the mapping can be formulated as optimization problems with respect to hardware constraints [24, 39, 66]. We call this **hardware-aware** mapping. This approach achieves very high energy efficiency for specific application domains but sacrifices flexibility. For **ISA-aware** mapping, the hardware accelerators are programmable with ISA, which separates the algorithmic specification from hardware architectural details. These instructions are often exposed as special **intrinsics** and using these intrinsics for tensor computation is called **tensorization** [9]. We focus on the ISA-aware mapping problem in this paper.

While intrinsics provide programmability, ISA-aware mapping is still a challenging task mainly for two reasons. First, there are different ways to compose a mapping using intrinsics. For example, we find that there are 35 different ways to map the 7 loops of a 2D convolution to the 3 dimensions of Tensor Core. The quality of the mapping is apparently critical to the performance as different mappings vary substantially by affecting data locality and parallelism. But existing compilers [9, 10, 52, 58, 67] heavily rely on manual programming with intrinsics to develop libraries or templates, which may miss the optimal mapping choice. Second, different accelerators provide different intrinsics with intricate compute and memory semantics. For example, Tensor Core uses different intrinsics to describe matrix load/store, matrix multiplication, and initialization semantics, while for Mali GPU, a single *arm_dot* intrinsic can work without other explicit load/store intrinsics. Therefore, to support a single algorithm on different accelerators, the programmers practically need to implement and tune the algorithm for each target platform individually. Apparently, a desirable mapping solution is to use a unified approach to expose a search space of feasible mappings that can be explored automatically.

In this paper, we address the ISA-aware mapping problem by providing one layer of abstraction above the intrinsics. This is based on the following insights: 1) although different accelerators use different intrinsics, these intrinsics can be rewritten into an equivalent scalar format, which not only breaks the original opaque intrinsic into an analyzable format but also bridges the gap between high-level tensor programs and low-level intrinsics; 2) though different

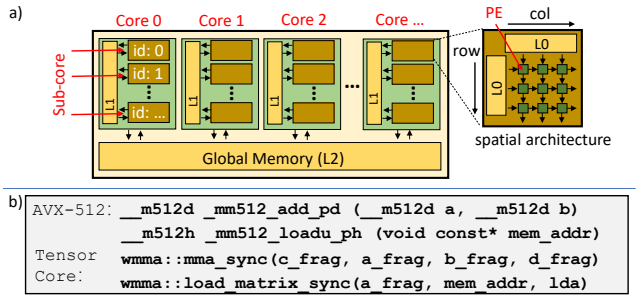


Figure 1: a) Accelerator design example with 3-level hierarchy. b) Examples for different intrinsics.

spatial accelerators vary in architecture design, they bear fundamental similarities in mapping design. For example, their hardware constraints can be uniformly defined as problem size constraints and memory capacity constraints. Based on these insights, we are able to use a unified abstraction to describe different intrinsics of different accelerators and design an automatic solution to mapping generation, validation, and exploration for different hardware.

In detail, the proposed abstraction includes two parts: compute abstraction and memory abstraction, which depict the compute and data access operations within an intrinsic in scalar format. Based on the proposed abstraction, we further develop a two-step mapping generation flow and a validation algorithm to automatically generate valid software-hardware mappings. The generation flow first maps software computations to a virtual hardware accelerator without hardware constraints and then modifies the mapping with respect to the actual physical hardware constraints. However, not all the generated mappings are valid because they may be different from the original semantics. Therefore, we design a validation algorithm based on a binary matrix representation to check the validity of mappings. Moreover, by combining tuning and analytic models together, we can efficiently explore the mapping space to achieve high performance on spatial accelerators. Finally, we implement the techniques introduced above into a compilation framework called AMOS.

We conclude our contributions as follows:

(1) We propose a hardware abstraction to formally define the spatial hardware compute and memory behavior and the mapping from software to hardware.
(2) We propose a fully automatic solution to the ISA-aware mapping problem by designing a two-step mapping generation flow, a novel validation algorithm, and a performance model for exploration.
(3) We propose a compilation framework called AMOS that supports a wide variety of tensor applications on spatial accelerators for both operator level and full network level.

Mapping tensor computations onto spatial accelerators is so esoteric that it is very difficult for the users to understand the hardware mapping (e.g., structure and constraints) and performance optimization space. AMOS solves the mapping problem by formulating an equivalent iteration mapping generation and validation problem. Exploration of mapping space is also necessary as different tensor

computations and input shapes have diverse behaviors and prefer different hardware mappings.

The fundamental improvement of AMOS over prior approaches (hand-tuned libraries [31, 41–43] and template-based compilers [10, 58, 68]) lies in two aspects: hardware abstraction of spatial hardware that expresses the behavior of different intrinsics in a general manner and the formulation and exploration of the complete mapping space. The core hardware abstraction enables AMOS to systematically explore the space by varying the mapping of software iterations to hardware iterations and performing mapping validation. Meanwhile, prior libraries and compilers are fundamentally limited in two aspects. First, prior works such as hand-tuned libraries and template-based compilers can only explore one or a few fixed mappings by manually writing templates or low-level code. Second, even if the templates can be designed manually with a moderate amount of engineering effort, it is still not clear what mappings and how many of them should be designed as there is a lack of understanding of the hardware abstraction of spatial accelerators and their mapping optimization space. Therefore, prior works lead to sub-optimal performance and heavy development costs.

The source code of AMOS is publicly available on Github (https://github.com/pku-liang/AMOS). Experiments show that AMOS can achieve more than 2.50× speedup on Tensor Core compared to hand-optimized libraries [41, 42], 1.37× speedup to expert designed templates in TVM [9] using AVX-512, and up to 25.04× speedup to AutoTVM [10] using dot units of Mali GPU. We also evaluate AMOS on new spatial accelerator designs.

2 BACKGROUND AND MOTIVATION

2.1 Spatial Accelerator

Spatial accelerators are usually designed in hierarchy, the innermost level of which is designed in spatial architecture and employs a processing elements (PE) array with dedicated dataflow and connection. Different spatial accelerators have different designs for PE array and support different kinds of computation. Each PE contains compute units for multiply and accumulate computation. For outer levels of hardware, sub-cores and cores are formed, sharing buffers and global memories. The spatial hardware is usually embedded into general-purpose architectures such as CPUs and GPUs and serves as special functional units. In Figure 1 part a) we show an example of 3-level accelerator. The innermost level corresponds to the PE array with spatial architecture. In the second innermost level are sub-cores and shared buffers. And in the outermost level are cores and global memory. Accelerators such as Nvidia GPU [40, 45, 46] with Tensor Core, Intel CPU with AVX-512, Mali GPU [3] with dot unit, and Ascend NPU [32] with cube and vector units all follow this design paradigm.

The interface between compilers and spatial accelerators is called intrinsic. An intrinsic can be viewed as a special instruction designed for the specialized hardware. There are two types of intrinsic: compute intrinsic (for computation instructions) and memory intrinsic (for memory load and store instructions). Figure 1 part b) shows four examples of intrinsics. The first two intrinsics are from AVX-512. *_mm512_add_pd* is a vector addition intrinsic (compute) and *_mm512_loadu_ph* is a vector load intrinsic (memory). The following two intrinsics are from Tensor Core WMMA. *mma_sync* is a matrix multiplication intrinsic (compute) and *load_matrix_sync* is a matrix load intrinsic (memory).

Table 1: State-of-the-art compilers/mappers for spatial accelerators.

Name	Mapping Method
Hardware-Aware Mapping:	
Nowatzki et al. [39]	SMT + Linear Programming
CoSA [24]	Mixed-Integer Programming
SARA [66]	Mixed-Integer Programming
HASCO [60]	Pre-defined SW-HW Interfaces + Tuning
ISA-Aware Mapping:	
AutoTVM [10]	Hand-written Templates + Tuning
Ansor [68]	Generation Rules + Tuning
UNIT [58]	Hand-written Templates
XLA [18]	Templates and Rules
ISA Mapper [52]	Templates and Rules + Tuning
Tiramisu [4]	Polyhedral Model
AKG [67]	Polyhedral Model + Templates
AMOS	**Analyzable Abstraction + Tuning**

Table 2: The complex pattern matching rules in XLA have only succeeded in mapping a few operators to Tensor Core for real DNN models. But our method can map more operators.

Name	Total Ops	XLA Mapped	Failed Example	Our Mapped
ShuffleNet	70	6	depthwise conv	50
ResNet-50	71	15	strided conv	54
MobileNet	30	7	grouped conv	29
Bert	204	42	part of attention	84
MI-LSTM	11	0	linear	9

2.2 Existing Mapping Flow

In Table 1 we divide the state-of-the-art compilers/mappers for spatial accelerators into two categories.

1) Hardware-aware mapping: The compilers/mappers are aware of the detailed PE connection and buffer configuration so they can lower software computations to PEs and memory through domain-specific interfaces [28, 33, 47, 62]. This approach is mainly applied for hardware and software co-design of specific domains. The major objective of these mappers is to optimize hardware resource utilization (parallelism and reuse) under certain hardware constraints. They usually solve the mapping problem by solving an equivalent linear programming problem to avoid brute-force searching in the large schedule space [24, 39, 66]. These compilers/mappers are orthogonal to this paper and we focus on solving the problem of mapping with programmable intrinsics without direct access to the hardware architecture details.

2) ISA-aware mapping: For flexible spatial accelerators, they expose intrinsics to the programmers. To use these intrinsics, the programmers have to write schedule templates [10, 58] and tune for performance within a given schedule space. For example, TVM [9] exposes a *tensorize* interface for users to configure their own intrinsics and the users have to manually invoke intrinsics when implementing the software. Polyhedral compilers such as AKG [67] relies on a combination of polyhedral model and templates to map software onto spatial accelerators. AutoTVM [10] and UNIT [58] use hand-tuned templates with intrinsics to support a narrow range

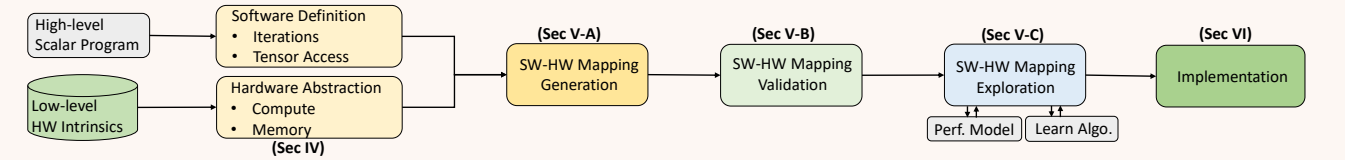


Figure 2: The compilation flow of AMOS.

of operators and accelerators. To support new operators, new templates should be developed manually. The manual design process of templates makes it hard to support new operators and accelerators. Ansor[68] uses combinations of generation rules such as loop tiling, reordering, and unrolling for general-purpose hardware. But this approach falls back to template-based method for spatial accelerators and requires the users to manually rewrite its rules for each intrinsic.

2.3 Motivational Example

To motivate our work, we show that manually designed templates for spatial accelerators can be inefficient for real workloads by profiling the generated code of a state-of-the-art compiler XLA [18] on Nvidia V100 GPU with Tensor Core. We use common DNN models including ShuffleNet [65], ResNet-50 [20], MobileNet [22], Bert [15], and MI-LSTM [59] as the inputs of XLA. XLA uses complex templates tuned by hand to optimize the operators in these models. Especially, some of these templates match special patterns for Tensor Core, and the successfully matched operators will be lowered to low-level hand-tuned libraries such as CuDNN [42] and CUTLASS [43] to invoke Tensor Core intrinsics.

Table 2 lists the profiled results. Although the templates in XLA are well designed, only a few operators in these networks are successfully mapped to Tensor Core. Moreover, we find that it is possible to map many of the remaining operators to Tensor Core. For example, the depthwise convolution [12] and grouped convolution [23] in ShuffleNet are variants of 2D convolution and the multiply-add operations in these operators are amenable for Tensor Core; the linear layers in MI-LSTM [59] can be mapped to Tensor Core, too. If we can map these operators to Tensor Core, better performance can be achieved. However, XLA fails in doing this and falls back to the scalar units of the GPU. This is because the patterns of these operators do not match with the XLA's handwritten templates. For example, for linear layers, XLA expects to map a matrix multiplication to Tensor Core, but when the batch size is 1, linear layers in MI-LSTM are matrix-vector multiplications, which mismatches with the pattern of matrix multiplication. As a result, the achieved Tensor Core utilization is low (for the above 5 models, the average utilization is no more than 10%). In contrast, our framework can successfully map all the operators except for those that are inherently not supported by Tensor Core (e.g., ReLU and MaxPooling). Table 2 also shows the number of our mapped operators on Tensor Core. The data is collected from the experiments conducted in Section 7.4.

The manually designed templates have two limitations. First, these templates rely on explicit intrinsic programming to use the dedicated computation units provided by spatial accelerators. For example, XLA relies on hand-tuned libraries (CuDNN, CUTLASS) to generate code with intrinsics, and the templates of AutoTVM [10] and UNIT [58] use intrinsics explicitly for fixed loop patterns. It is esoteric and time-consuming to develop new templates for most users, and as a result, only a few operators can be mapped to spatial accelerators. Second, the templates use fragile pattern matching rules to map operators to accelerators. A subtle modification of the operator (e.g., change the layout) can result in matching failure, which further reduces the number of operators that can be successfully mapped to spatial accelerators.

Considering the limitations of manual intrinsic programming and templates, we are motivated to design a fully automatic compilation flow without templates.

3 AMOS OVERVIEW

In this section, we present the overall workflow of AMOS. In Figure 2, the input of AMOS is a high-level program written in the domain-specific language (DSL) of existing compilers [9, 67]. The DSL defines loops and tensors in a high-level language such as Python. The loop structure and tensor access indices of each tensor are also written in the DSL. Figure 3 part a) shows an example for 2D convolution. AMOS represents hardware intrinsics using our proposed hardware abstraction for both compute and memory. We will explain the abstraction in Section 4. Then AMOS uses the software and hardware information to generate different software-hardware mappings. One software-hardware mapping is composed of compute mapping and memory mapping. The compute mapping specifies which operations defined in software should be mapped to the target spatial hardware and implemented through corresponding compute intrinsics. The memory mapping specifies how to load data from global/shared memory and store the data in on-chip registers through memory intrinsics. The spatial hardware enforces constraints such as fixed computation problem size and memory capacity, which are reflected by the intrinsics. The details about mapping generation under these constraints are illustrated in Section 5.1. In addition to hardware constraints, preserving correct semantics in mapping is also necessary. We use an algorithm introduced in Section 5.2 to verify the validity of generated mappings. After that, we combine tuning methods and analytical models in Section 5.3 to choose a high-performance mapping from the valid mapping candidates. At last, we explain the implementation of AMOS and code generation in Section 6.

AMOS is fundamentally different from previous work [10, 18, 58] in that it does not rely on hand-tuned templates and libraries and provides a fully automatic compilation flow for mapping tensor

programs to spatial accelerators through automatic mapping generation.

4 HARDWARE ABSTRACTION

In this section, we introduce hardware abstraction. The main idea is to reform the low-level intrinsics into equivalent high-level scalar-based representations. We divide the abstraction into two parts: 1) hardware compute abstraction; 2) hardware memory abstraction. The goal of this abstraction is to formally define the behavior of spatial hardware accelerators so that AMOS can automatically analyze the intrinsics of different spatial accelerators.

4.1 Compute Abstraction

Def 4.1. ***Compute Abstraction*** *is a statement that specifies the operands, arithmetic operation among operands, and data access indices of operands for one hardware compute intrinsic. The range of the indices should also be represented in the abstraction. Assume that we have M data sources and the m-th data source Src_m is a D_m-dim array. The output Dst is an N-dim array. The indices of output array are represented as $\vec{i} = [i_1, i_2, .., i_N]$, and the indices of the m-th source array are represented as $\vec{j}^m = [j_1^m, j_2^m, ..., j_{D_m}^m]$. The statement is written as:*

$$Dst[\vec{i}] = \mathcal{F}(Src_1[\vec{j}^1], Src_2[\vec{j}^2], ..., Src_M[\vec{j}^M]),$$
$$s.t.\ A\vec{i} + \sum_m B^m \vec{j}^m + C < 0$$

Explanation: The function $\mathcal{F}$ represents arithmetic operations such as addition, multiplication, or multiply-add operation. The matrices A, B^m $(1 \le m \le M)$, and C are used to express the range of indices in an affine manner. The constant matrix C is necessary because we need to express the boundaries of the indices. For example, to represent a Tensor Core *mma_sync* intrinsic, which can be used to compute a matrix multiplication for a special shape $32 \times 8 \times 16$, we can write the abstraction as

$$\mathrm{Dst}[i_1, i_2] = \text{multiply-add}(\mathrm{Src}_1[i_1, r_1], \mathrm{Src}_2[r_1, i_2]),$$
$$\text{s.t.} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \end{bmatrix} + \begin{bmatrix} -32 \\ -8 \\ -16 \end{bmatrix} < 0 \tag{1}$$

4.2 Memory Abstraction

Def 4.2. ***Memory Abstraction*** *is a list of statements. Each statement in the list specifies the scope, operands, and memory access indices. Similar to the symbols we have used in compute abstraction, we use Dst to denote the output results and Src_m to denote the input data arrays. Prefixes $global$, $shared$, and reg represent global memory, shared buffers, and registers, respectively. $\vec{i}$, $\vec{j^m}$, $\vec{k}$, and $\vec{l^m}$ are indices for the data arrays in different scopes.*

$$reg.Src_1[\vec{j^1}] = shared.Src_1[\vec{l^1}]$$
$$...$$
$$reg.Src_M[\vec{j^M}] = shared.Src_M[\vec{l^M}]$$
$$global.Dst[\vec{k}] = reg[\vec{i}]$$

Explanation: The memory abstraction shows that input data is from shared buffers and the output data is stored in global memory. Data load from global memory to shared buffers for inputs is not shown for simplicity. The indices for the same operand within different memory scopes can be different. For example, the intrinsic *load_matrix_sync* that loads data from shared memory to Tensor Core registers and the intrinsic *store_matrix_sync* that stores data from Tensor Core registers to global memory can be expressed together as

$$\begin{aligned} &reg.\mathrm{Src}_1[i_1, r_1] = shared.\mathrm{Src}_1[addr_a + i_1 * stride_a + r_1] \\ &reg.\mathrm{Src}_2[r_1, i_2] = shared.\mathrm{Src}_2[addr_b + r_1 * stride_b + i_2] \\ &global.\mathrm{Dst}[addr_c + i_1 * stride_c + i_2] = reg.\mathrm{Dst}[i_1, i_2] \end{aligned} \tag{2}$$

where $addr_a$, $addr_b$, and $addr_c$ are base memory addresses and $stride_a$, $stride_b$, and $stride_c$ are memory access strides. These parameters should be set by the compilers when mapping software to hardware. These parameters are useful because the intrinsics *load_matrix_sync* and *store_matrix_sync* require them as inputs.

Both compute and memory abstractions clearly model the behavior of hardware intrinsics and break the opaque intrinsics into a series of equivalent scalar operations for holistic analysis. The various intrinsics in existing accelerators [27, 32, 37] are actually different SIMD-like (single instruction multiple data, reduction is also allowed [11, 58]) operations, and SIMD-like operations can be naturally decomposed into equivalent scalar operations, which can be perfectly described by our abstraction. Therefore, the compute and memory abstraction can be used for different spatial accelerators.

4.3 Software-Hardware Mapping

In the following, we introduce the mapping from software to hardware. First, we introduce the concept of **software iterations**. Tensor computations can be represented as perfectly nested loops. Software iterations are all the loop instances within the loop nest [5, 56]. For example, in Figure 3 part a) we show a 2D convolution example, and the software iterations are listed in part b).

Similarly, we can define **intrinsic iterations** for intrinsics based on the previously introduced compute abstraction. Intrinsic iterations are the scalar operation instances within the indices range of the compute abstraction. For example, there are three iterations (i_1, i_2, r_1) in Tensor Core intrinsic as shown in Equation 1. Given the definitions of software iterations, intrinsic iterations, compute abstraction, and memory abstraction, we define the software-hardware mapping as follows

Def 4.3. ***Software-Hardware Mapping*** *is composed of two parts: compute mapping and memory mapping. Compute mapping is to assign each software iteration to an intrinsic iteration. Memory mapping is to assign each software iteration to a memory address (base address and strides) for each operand in the memory abstraction.*

$$\Theta = <\ Compute\text{-}Map : \mathbb{C},\ \ Memory\text{-}Map : \mathbb{M}\ >$$
$$\mathbb{C} = \{Software[\vec{s}]\ \rightarrow\ Intrinsic[\ \vec{i}, \vec{j^1}, ..., \vec{j^M}]\}$$
$$\mathbb{M} = \{Software[\vec{s}] \rightarrow operand[\vec{addr}]\}$$
$$for\ each\ operand\ in\ memory\ abstraction$$

Explanation: To map software to spatial accelerators through intrinsics, we have to know how each scalar operation defined in software is implemented by corresponding compute/memory intrinsics. Based on the compute/memory abstraction, the original intrinsics are split into a series of scalar operations so that the compilers can establish a mapping relationship between software iterations and intrinsic iterations. Apart from the mapping for computation, the

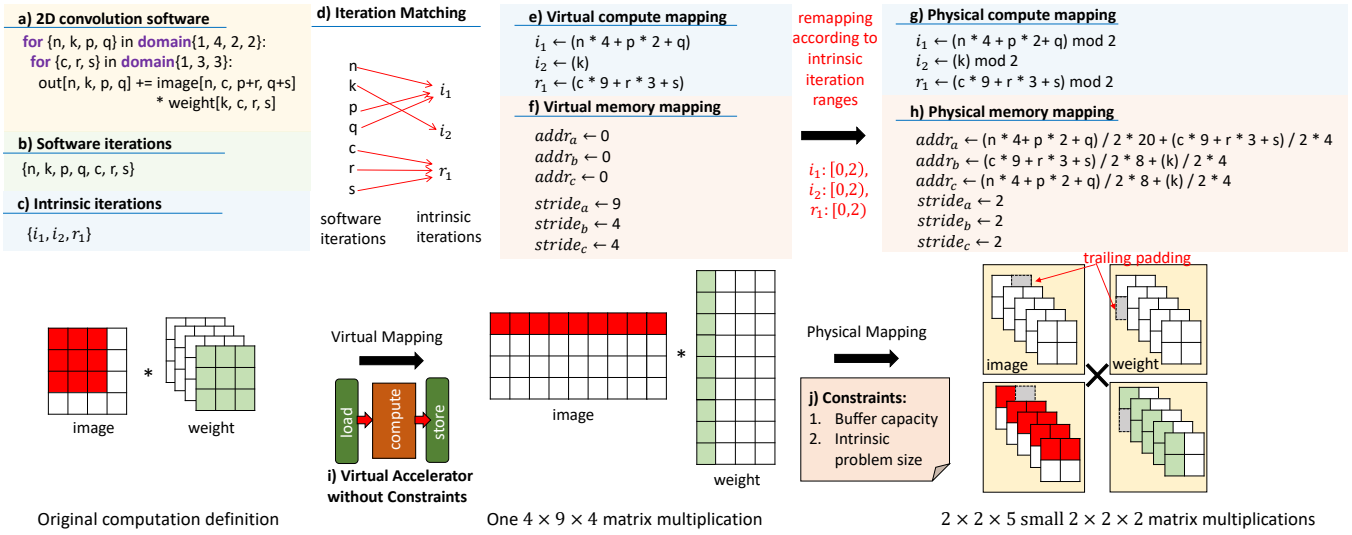


Figure 3: Mapping generation flow. This example maps a small 2D convolution to Tensor Core (simplified to $2 \times 2 \times 2$ multiplication). Part a): Software description. Part b): software iterations for 2D convolution. Part c): intrinsic iterations for Tensor Core. Part d) Iteration matching between software iterations and intrinsic iterations. Part e) and f): virtual compute and memory mapping without consideration for constraints of intrinsics. Part g) and h): physical compute and memory mapping with consideration for intrinsic constraints. Part i): virtual accelerator illustration. Part j): constraints in mapping.

mapping for memory access is also feasible. In memory abstraction, the access indices for each operand within different memory scopes are explicitly specified. If we can associate each software iteration with an array of access indices for each operand, then the memory access addresses can be automatically generated. In the following, we will explain how to generate software-hardware mappings and select a high-performance valid mapping for code generation with intrinsics automatically.

5 MAPPING GENERATION, VALIDATION, AND EXPLORATION

5.1 Software-Hardware Mapping Generation

To generate the mapping defined Section 4.3, we propose a two-step approach. First, we map software iterations to a virtual accelerator that has only one load engine, one compute engine, and one store engine as illustrated in Figure 3 part i). The memory capacity and computation ability are not modeled at this step. Second, we add the constraints of hardware and intrinsics back and modify the previous mapping to output a physical mapping that can be directly implemented by hardware intrinsics. To explain the mapping generation flow, we use Figure 3 as a running example. Although this example uses 2D convolution and a simplified Tensor Core (designed for $2 \times 2 \times 2$ matrix multiplication) for illustration, our flow is general and can be used for other workloads and intrinsics.

For the first step, AMOS assumes the virtual accelerator can load data of arbitrary volume into on-chip registers and has enough hardware resources to perform all the computations at once. So the main problem for this step is how to place software-defined scalar operations into corresponding intrinsic-provided computation slots. In other words, it is to match between software iterations and intrinsic iterations. In Figure 3 part d) we show a matching example that matches software iterations n, p, q with intrinsic iteration i_1, software iterations k with intrinsic iteration i_2, and software iterations c, r, s with intrinsic iteration r_1. According to this matching, the original 2D convolution (batch=1, input channel=1, output channel=4, height=4, width=4, kernel size=3) is reformed to an equivalent matrix multiplication (height=4, reduction length=9, width=4). And the virtual accelerator is expected to load two matrices (of shape 4×9 and 9×4) into registers, complete the $4 \times 9 \times 4$ matrix multiplication simultaneously, and store the output matrix to global memory. The base addresses in memory mapping are set to zeros and the strides are set according to the matrix shape as shown in Figure 3 part e) and f) because all the data is assumed to reside in registers. However, virtual mapping is unrealistic because real accelerators are constrained in terms of buffer capacity and computation resources. So we need to take these constraints into consideration in the next step.

For the second step, two kinds of constraints are considered: intrinsic problem size and memory capacity. The hardware accelerator can only compute a fixed size of results at a time, which is called problem size constraint. AMOS limits the range of matched software iterations to the intrinsic problem size by modulo operations. The problem size of an intrinsic can be extracted from its indices range, which is represented in the compute abstraction. In Figure 3 part g), the matched software iterations are restricted by factors 2 (*mod 2*) because the problem size of the example Tensor Core is $2 \times 2 \times 2$.

Algorithm 1: Mapping Validation Algorithm

input : Software Access Matrix: X
input : Iteration Matching Matrix: Y
input : Intrinsic Access Matrix: Z
output: validity (True or False)

1: // $\star$ is binary matrix multiplication;
2: $X' := Z \star Y$; // get software access relationship;
3: $Z' := X \star Y^T$; // get hardware access relationship;
4: **return** $(X' = X)$ and $(Z' = Z)$

For the constraint of memory capacity, each register fragment in a spatial accelerator is only able to hold a limited size of data. Therefore, AMOS splits the whole input/output data into small slices and load/store these slices of data multiple times, which means that the base addresses and strides in virtual memory mapping should be updated correspondingly. To do this, AMOS uses the remaining software iterations that are not mapped to intrinsic iterations to locate the index of sub-slices of data and generates the base addresses for physical memory mapping. As for the strides, they are no longer set by the shape of the total data, but by the register capacity. In the example of Figure 3, one fragment of Tensor Core could only hold one 2×2 matrix for each operand (image, weight, and out). In part h), $(n * 4 + p * 2 + q)/2$ corresponds to the iterations that are not mapped to intrinsic iteration i_1 and $(c*9+r*3+s)/2$ corresponds to the iterations that are not mapped to intrinsic iteration r_1. So these two remaining iterations are used to locate the index of sub-matrices of the image (by setting the value of $addr_a$). Note that we write the base addresses in a flattened manner $(n*4+p*2+q)/2*20+(c*9+r*3+s)/2*4$, where 4 is the number of elements within a sub-matrix and $20 = 5 \times 4$ means that 5 sub-matrices are grouped together (as shown in Figure 3). Finally, AMOS should handle the trailing padding problem. In this example, a 4×9 matrix is split into 2×5 small 2×2 sub-matrices, so there exist trailing data in sub-matrices. AMOS will pad zeros to these trailing sub-matrices during mapping generation.

5.2 Software-Hardware Mapping Validation

However, some of the generated mappings may not be valid, and AMOS uses a validation algorithm to ensure that only the valid mapping candidates are considered. For example, if we map software iterations n, k to the same intrinsic iteration i_1 in Figure 3 part d), we will get a wrong mapping even though the hardware related constraints are satisfied. The reason is that software iterations n and k have different semantics in the original 2D convolution: n is the common index of output and image but never appears in weight, while k is the common index of output and weight but never appears in the image. So they should not be mapped to the same intrinsic iteration.

Our validation algorithm is shown in Algorithm 1. We first explain the inputs of the algorithm. Access matrix is a binary-value matrix that describes the data access relationship between indices and tensors (or data arrays). Each row of the matrix represents a tensor and each column represents an index. If an index at column col is used to access the data of a tensor at row row, then the corresponding value at position (row, col) of the access matrix is set to 1, otherwise, it is 0. As for the matching matrix, it is a binary-value matrix that describes the matching relationship between software iterations and intrinsic iterations. For example, in Figure 4 we show the access matrix of 2D convolution, Tensor Core compute intrinsic, and the matching matrix between them. The matching relationship reflected in this matching matrix is the same as shown in Figure 3 part d). These binary matrices carry important access and matching information that is necessary for validation.

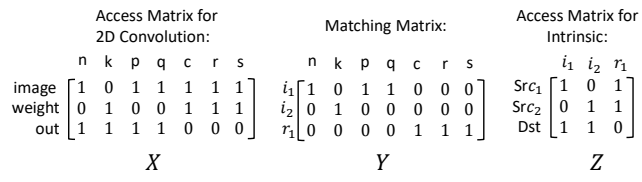


Figure 4: Example of access matrix and matching matrix.

Table 3: a) Schedule optimizations used by AMOS. b) Symbols used in performance model.

a) Optimization	Explanation
tile	split and reorder loop nests
fuse	fuse two sub-loops into a large loop
bind	bind loops to parallel cores
parallel	use multi-threading
cache	use shared buffers
unroll/vectorize	unroll/vectorize to increase parallelism
b) Symbol	**Meaning**
$\mathcal{L}_l$	Latency of computation on level l
$\mathcal{R}_l$	Latency of data load on level l
$\mathcal{W}_l$	Latency of data store on level l
$\vec{S}_l$	sequential loops (not bound to threads) for level l
DataIn_l	amount of data read for the l-level memory
DataOut_l	amount of data write from the l-level memory
in_bw_l	bandwidth of data load of level l memory
out_bw_l	bandwidth of data store of level l memory

Algorithm 1 validates the mappings by checking the software access relationship and the hardware access relationship. Software access relationship defines which software iterations access which hardware memory (registers); and hardware access relationship defines which intrinsic iterations access which software-defined tensors. To do this, the algorithm first uses intrinsic access matrix Z and matching matrix Y to calculate the software access relationship by $Z \star Y$, which is represented by another binary matrix X'. We use $\star$ to represent matrix multiplication operation between binary-value matrices. If X' is the same as X, then it means that for every pair of input/output tensor (in software definition) and input/output operand (in intrinsic), the access behavior remains the same for all the indices, otherwise, the semantics are not preserved. Similarly, the algorithm then calculates the hardware access relationship by $X \star Y^T$, where X is the software access matrix. The result Z' is also a binary matrix and is expected to be the same as intrinsic access matrix Z if the matching matrix Y is valid. Algorithm 1 is efficient and accurate for validating the software-hardware mapping. AMOS leverages this algorithm to guarantee that the generated mappings preserve original semantics in software. For the example in Figure 3, AMOS finds that only 35 mappings are valid out of $3^7 = 21087$ possible candidates.

5.3 Exploration of Mapping and Schedule

So far, we can select the valid software-hardware mappings. However, it is hard to know the performance of these mappings when optimized with optimization schedules such as tiling and parallelization. We show the schedule optimizations AMOS considers in Table 3 part a). These different optimization schedules combined with different software-hardware mappings vary in performance as they are different in resource utilization (compute cores and memory units). The combined search space of schedules and mappings is large (usually more than 10^5). This huge space makes exhaustive exploration infeasible. We thus adopt a combination of performance model and tuning techniques for efficient schedule and mapping exploration. Considering the spatial accelerators are designed in hierarchy, our performance model is also built level by level (level 0 corresponds to intrinsics). The explanation of used symbols is listed in Table 3 part b). The performance model can be written as

$$\text{Perf} = \mathcal{L}_{L-1}, \quad L \text{ is the number of levels of hardware}$$

$$\mathcal{L}_l = \begin{cases} (\prod \vec{S}_l) \times \max(\mathcal{L}_{l-1}, \mathcal{R}_{l-1}, \mathcal{W}_{l-1}), & l > 0 \\ (\prod \vec{S}_l) \times \text{latency\_of\_intrinsic}, & l = 0 \end{cases}$$

$$\mathcal{R}_l = \frac{\text{DataIn}_l}{\text{in\_bw}_l} \qquad \mathcal{W}_l = \frac{\text{DataOut}_l}{\text{in\_bw}_l}$$

The maximum of compute latency, read latency, and store latency dominates the overall latency (fully pipelined execution), and the compute latency is either the latency of inner level computation (level $l-1$) or the latency of intrinsic (if $l = 0$). The latency of intrinsic is a fixed value, which can be estimated through hardware models such as Maestro [28] and TENET [33]. The final latency is multiplied by the trip counts of sequential loops because these loops are not bound to parallel cores and execute in sequential order. *DataIn* and *DataOut* can be calculated by inferring the size of buffers used in computation. Compilers such LLVM [29] and TVM [9] have already provided mature tools for bound inference and we omit the details here.

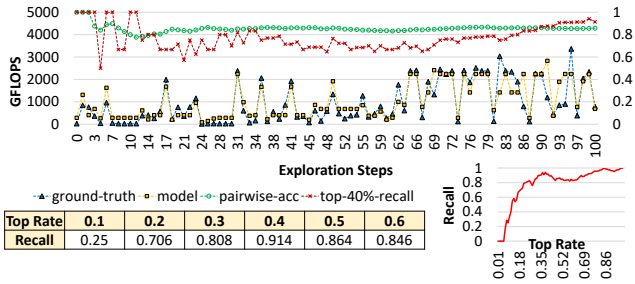
We integrate this performance model with existing optimization frameworks [10, 68, 70] to explore the combined search space of mappings and optimization schedules. We choose to use genetic algorithms in tuning engine as it is reported to produce the state-of-the-art performance [68]. In details, at the beginning of tuning, AMOS enumerates all the possible mappings through our mapping generation and validation process and randomly assigns different schedule parameters to each mapping candidate. Then these mappings along with their schedules are evaluated by our performance model. According to the evaluated results, AMOS chooses a set of good mappings and mutates their schedules to produce new choices to evaluate. This process may repeat thousands of times, and the final result is a pair of good mapping and schedule parameters, which can be used for final code generation.

6 IMPLEMENTATION OF AMOS

The hardware abstraction is implemented as compiler IR on top of some basic IR nodes in Table 4. We add two new IR nodes (*Compute* and *Memory*). *Compute* and *Memory* nodes can express all the information required by our compute and memory abstraction introduced in previous sections. A *Compute* node represents a small loop nest that matches with a compute intrinsic. Similarly, *Memory* nodes represent memory load/store operations and correspond to memory intrinsics. These IR nodes have specific attributes as shown in Table 4. For *Compute* node, the first attribute is the destination buffer to store the computation results, which is represented by a *Tensor* node. The second attribute is an expression describing the concrete arithmetic operations (addition, multiplication, etc.). We use an *Expr* node to represent the expression. The third attribute of the *Compute* node is used to represent the intrinsic iterations as illustrated in compute abstraction. We use an *Array* of *Expr* nodes to represent this attribute. As for *Memory* node, the first attribute is the destination buffer of memory access operations. The second attribute is a *String* node that is used to encode buffer scope information (global, shared, or register). The third attribute is a *BufferLoad* node used to represent the source buffer and load indices. AMOS uses the IR to express hardware intrinsics and reuse the tensorize interface of TVM[9] to insert these nodes into the compiler abstract syntax tree during lowering and code generation.

Table 4: IR nodes used by AMOS.

Basic IR Node	Explanation
Expr	represent arithmetic expressions (+,-,*,/, etc.)
BufferLoad	multi-dim load operation from a buffer
Tensor	represent an n-dim data buffer
Array	pack a list of IR nodes into an array
String	represent a string data node
New IR Node	**Semantics**
Compute	Compute(Tensor, Expr, Array<Expr>)
Memory	Memory(Tensor, String, BufferLoad)



Top Rate	0.1	0.2	0.3	0.4	0.5	0.6
Recall	0.25	0.706	0.808	0.914	0.864	0.846

Figure 5: Model validation results on Tensor Core GPU using 2D convolution workload.

7 EXPERIMENTAL RESULTS

7.1 Evaluation Setup

We first evaluate AMOS using three commercial spatial accelerators and their intrinsics include *mma_sync* in TensorCore GPU (V100 [46] and A100 [40]), *_mm512_dpbusds_epi32* in Intel CPU with AVX-512 (Xeon(R) Silver 4110), and *arm_dot* in Mali Bifrost GPU (G76 [3]). In addition, we also evaluate AMOS using new spatial accelerator designs and intrinsics.

For evaluation, we first validate the accuracy of AMOS's performance model. Then we evaluate AMOS on Tensor Core GPUs for single operators and entire DNN networks. After that, we evaluate AMOS on AVX-512 CPU and Mali GPU for convolution. At

Table 5: SW-HW mappings found by AMOS for each C2D layer in ResNet-18 on A100 GPU. n, k, p, q, c, r, s are batch, output channel, height, width, input channel, kernel height, and kernel width. We only show the compute mappings due to space limit.

Layer	n	c	k	p	q	r	s	stride	SW-HW mapping (compute mapping only)
C0	16	3	64	112	112	7	7	2	$[i_1, i_2, r_1] \leftarrow [(n*112+q) \bmod 16, k \bmod 16, (c*49+r*7+s) \bmod 16]$
C1	16	64	64	56	56	3	3	1	$[i_1, i_2, r_1] \leftarrow [(n*56+q) \bmod 16, k \bmod 16, (c*3+r) \bmod 16]$
C2	16	64	64	56	56	1	1	1	$[i_1, i_2, r_1] \leftarrow [(p*56+q) \bmod 16, k \bmod 16, c \bmod 16]$
C3	16	64	128	28	28	3	3	2	$[i_1, i_2, r_1] \leftarrow [(n*784+p*28+q) \bmod 16, k \bmod 16, (c*3+s) \bmod 16]$
C4	16	64	128	28	28	1	1	2	$[i_1, i_2, r_1] \leftarrow [(p*28+q) \bmod 16, k \bmod 16, c \bmod 16]$
C5	16	128	128	28	28	3	3	1	$[i_1, i_2, r_1] \leftarrow [(p*28+q) \bmod 16, k \bmod 16, c \bmod 16]$
C6	16	128	256	14	14	3	3	2	$[i_1, i_2, r_1] \leftarrow [n \bmod 16, k \bmod 16, (c*3+s) \bmod 16]$
C7	16	128	256	14	14	1	1	2	$[i_1, i_2, r_1] \leftarrow [(n*196+p*14+q) \bmod 16, k \bmod 16, c \bmod 16]$
C8	16	256	256	14	14	3	3	1	$[i_1, i_2, r_1] \leftarrow [(p*14+q) \bmod 16, k \bmod 16, c \bmod 16]$
C9	16	256	512	7	7	3	3	2	$[i_1, i_2, r_1] \leftarrow [(n*49+p*7+q) \bmod 16, k \bmod 16, (c*9+r*3+s) \bmod 16]$
C10	16	256	512	7	7	1	1	2	$[i_1, i_2, r_1] \leftarrow [(n*49+p*7+q) \bmod 16, k \bmod 16, c \bmod 16]$
C11	16	512	512	7	7	3	3	1	$[i_1, i_2, r_1] \leftarrow [n \bmod 16, k \bmod 16, (c*9+r*3+s) \bmod 16]$

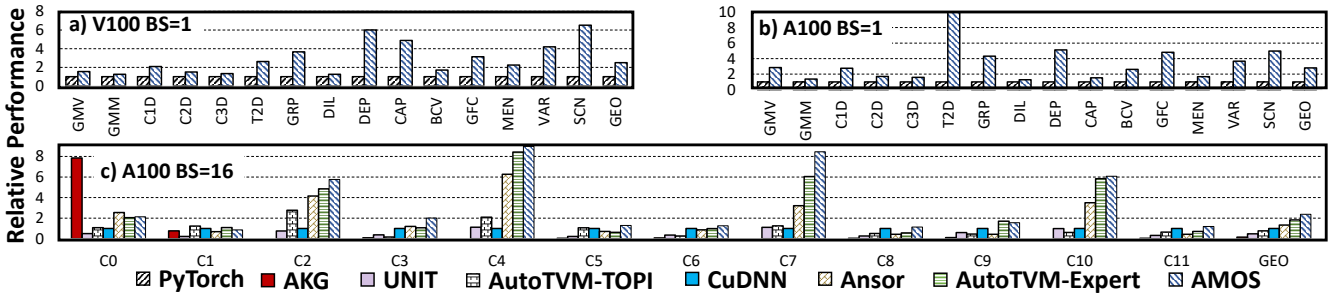


Figure 6: Part a) and b): single operator performance relative to PyTorch. Part c): performance comparison for C2D on A100 relative to CuDNN.

last, we show the results of AMOS on new hardware architectures. We compare to PyTorch [48], CuDNN [42], Ansor [68], AutoTVM [10], UNIT [58], and AKG [67] for performance. For benchmarks, we use DNNs from both image processing and natural language processing. They include ShuffleNet [65], ResNet-18 and ResNet-50 [20], MobileNet-V1 [22], Bert [15] (base configuration), and MI-LSTM [59]. AMOS is fully automatic and the only inputs are software descriptions in DSL, while other compilers need templates as additional inputs. We will explain the details in Section 7.3.

7.2 Model Validation

We validate AMOS's performance model using Tensor Core GPU. We set the configurations of V100 GPU according to the specifications [26, 46], including the number of SM, the number of sub-core within one SM, memory size, and evaluated bandwidth. We use 2D convolution layers from ResNet-18 and compare the ground-truth performance with model predicted performance for different exploration steps in Figure 5. We also show the pairwise (rank) accuracy (green line) and recall value for top-40% mappings (red line) in the figure. For each exploration step, the predicted performance is close to real performance in trend (not in absolute value). The pairwise accuracy shows that AMOS can predict the relative performance of explored mappings with good precision (the overall accuracy is 85.69%. The top-40% recall value shows that AMOS can retrieve the top 40% mappings with a high probability (overall recall is 91.4%). We also show the recall results under different top rates in Figure 5. The results indicate that AMOS can retrieve promising mappings during exploration with a probability of more than 80% for top rate lager than 30%. Based on this performance model, AMOS can successfully filter out the inferior mappings and only select the promising mappings for exploration.

7.3 Evaluation for Operators on Tensor Core

For single operators, we consider GEMV (GMV), GEMM (GEM), 1D convolution (C1D), 2D convolution (C2D), 3D convolution (C3D), transposed 2D convolution (T2D), group convolution [23] (GRP), dilated convolution (DIL), depthwise convolution [12] (DEP), capsule convolution [21] (CAP), batched convolution [61] (BCV), grouped fully-connected layer [35] (GFC), matrix mean and variance (MEN and VAR), and scan computation [13] (SCN). We test 113 different configurations (7-8 for each operator on average) for all the operators and report the geometric mean speedup to baselines. All the configurations are extracted from real-world networks [7, 15, 20, 21, 35, 50, 61].

First, we compare AMOS with PyTorch. PyTorch uses hand-optimized libraries such as CuDNN [42], CuBlas [41], and CUTLASS [43] to support different kinds of operators. We show the results of all the operators for batch 1 on V100 and A100 GPU in Figure 6 part a) and b). AMOS consistently exceeds PyTorch for all the operators and achieves 2.50× and 2.80× geometric mean speedup on V100 and A100, respectively. AMOS generates high-performance code on different platforms via a combined exploration of both

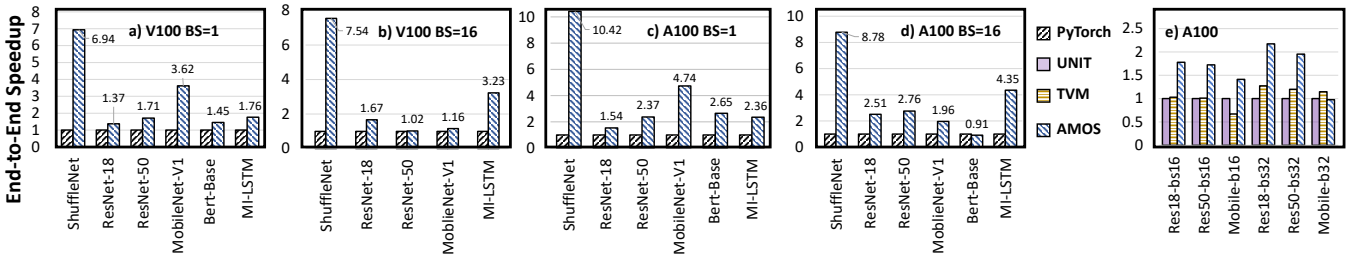


Figure 7: Part a) to d): performance of networks on V100 and A100 relative to PyTorch. Part e): performance of TVM and AMOS on A100 relative to UNIT.

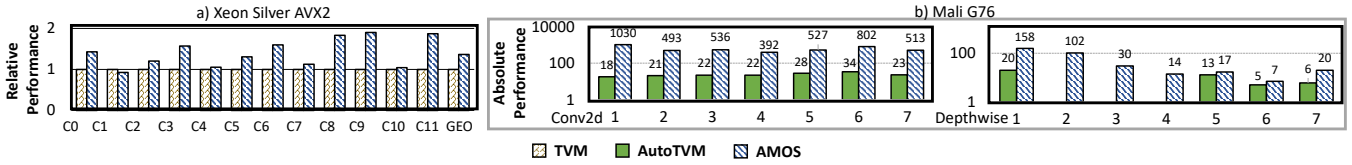


Figure 8: Part a): performance on Intel Xeon(R) Silver 4110 CPU relative to TVM. Part b): performance comparison to AutoTVM on Mali G76 GPU. We show absolute performance in logarithmetic values.

mappings and schedules. AMOS achieves speedup because of the comprehensive software-hardware mapping exploration, while PyTorch only uses fixed mappings implemented in the hand-optimized libraries, which gives a sub-optimal performance.

Second, we compare to state-of-the-art compilers. We use C2D in NCHW layout for performance evaluation. We use all the convolution layers from ResNet-18 [20] (totally 12 different configurations) and use C0-C11 to represent them (see Table 5). The results are shown in Figure 6 part c). AMOS achieves the best average performance among all the compilers (2.38× to CuDNN, 1.79× to Ansor, 1.30× to AutoTVM-Expert, 4.96× to UNIT). AKG[67] only maps a few layers to Tensor Core because its polyhedral model is not designed for intrinsics and fails to recognize opportunities to map convolutions to Tensor Core. Ansor[68] has no code generation rules for Tensor Core. So it can't use Tensor Core for all the layers. When these compilers fail to use Tensor Core, they will use CUDA Core instead. But different compilers use different optimization techniques. Therefore, their performance on CUDA Core is also different. UNIT's template always maps the height and width dimensions of C2D to Tensor Core intrinsic and ignores the batch dimension, which suffers from low parallelism and thus is much slower than AMOS. AutoTVM [10] fails in recognizing opportunities to use Tensor Core intrinsics because its hand-written templates are only designed for NHWC/HWNC layouts. NCHW layout is widely adopted by frameworks such as PyTorch, but NHWC is also a desirable layout for certain cases[44]. AMOS is not restricted to any specific layout. Therefore, we also compare C0 with NHWC layout to AutoTVM and the speedup is 2.83×. We also evaluate AutoTVM by manually adding a new template for NCHW layout with FP16 precision (denoted as AutoTVM-Expert). But the overall performance is still inferior to AMOS because the template adopts a fixed mapping strategy, while AMOS can systematically explore a large space of mappings for C2D.

AMOS finally chooses 8 different types of mappings for the 12 convolution layers (shown in Table 5). For example, the mapping for C7 maps iterations n, p, q to i_1, and maps iteration c to r_1. Other compilers can not use this mapping because their templates are fixed. For example, the template of UNIT only maps iterations p, q to i_1. AMOS' mapping for C9 is the same as that shown in Figure 3 part d); mapping for C5 is the same as that used by UNIT's template. All these mappings are automatically generated by AMOS.

7.4 Evaluation for Networks

AMOS can enable more operators to be mapped to Tensor Core and find better mappings for these operators compared to hand-tuned libraries and templates. We use batch size 1 and 16 for performance comparison of full networks. The results are shown in Figure 7 part a) to d). Bert of batch size 16 on V100 is not shown due to memory limit exceeding. AMOS exceeds PyTorch for all the benchmarks except for Bert with batch size 16 on A100 (speedup ranges from 0.91× to 10.42×). Bert is mainly composed of GEMM. And GEMM is highly optimized in libraries [41, 43] for decades. Even so, for batch size 16, AMOS still achieves more than 90% the performance of libraries. The significant speedup of ShuffleNet comes from the acceleration for group convolution (GRP) and depthwise convolution (DEP), which are not well supported by hand-tuned libraries on Tensor Core.

We also compare to UNIT and TVM for the whole network with different batch sizes. We use ResNet-18 ,ResNet-50, and MobileNet-V1 for comparison. Figure 7 part e) shows the results on A100. For most of the test cases, AMOS gets the best performance equally. For ResNet, the speedup arises from both normal convolutions and strided convolutions. The strided convolutions are hard to map to

Table 6: Number of feasible mappings on Tensor Core.

GMV	GMM	C1D	C2D	C3D	T2D	GRP	DIL
1	1	6	35	180	7	35	35
DEP	**CAP**	**BCV**	**GFC**	**MEN**	**VAR**	**SCN**	
11	105	11	1	1	1	1	

Tensor Core because the strides in tensor access index make it hard to generate memory access addresses during compilation and TVM doesn't use Tensor Core intrinsics for them. And UNIT's template doesn't consider batch dimension during mapping, its performance is inferior to TVM and AMOS. The results show that the mapping generation logic of AMOS can handle different sophisticated operators and leverage Tensor Core to get high performance.

7.5 Evaluation On Other Accelerators

Vector Units in Intel CPU: On Intel CPU, we add hardware abstraction for AVX-512 VNNI intrinsics (for matrix-vector multiplication) and use AMOS to generate code for C2D. We compare with TVM and TVM uses a hand-written template to generate VNNI intrinsics. Figure 8 part a) shows the results. C0-C11 refers to C2D layers in Table 5. AMOS outperforms TVM for all the cases except for C2. On average, the speedup to TVM is 1.37×.

Dot Units in Mali GPU: On Mali GPU (Bifrost architecture [3]), We add hardware abstraction for dot intrinsics in AMOS and compare to AutoTVM. The configurations of C2D and DEP are from MobileNet-V2 [51] (totally 7 depthwise layers). AutoTVM uses a hand-written template for Bifrost architecture. The experiment results show that the templates of AutoTVM are less optimized for dot intrinsic and for some layers of DEP (layer 2, 3, and 4), AutoTVM even can't generate code due to internal errors. As a result, the final performance of AutoTVM is much lower than AMOS. The absolute performance (GOPS) is shown in Figure 8 part b). The speedup of AMOS is up to 25.04×.

New Accelerators: To show the generality of AMOS, we use three virtual spatial accelerators, which offer different intrinsics for compute and memory. We choose three intrinsics for AXPY, GEMV, and CONV because they correspond to the three levels of BLAS [16] operations (GEMM is demonstrated in Tensor Core, so we use a more complex pointwise convolution for level 3). We add the hardware abstractions for these intrinsics in AMOS. We use C3D as input software. AMOS can find 15, 7, and 31 different types of mappings for the AXPY accelerator, GEMV accelerator, and CONV accelerator, respectively. For example, one mapping for the CONV accelerator maps output channel, height, width, and input channel to the convolution units. This experiment shows that AMOS is applicable to a new generation of accelerators with different intrinsics and proves the generality of our proposed techniques in this paper.

7.6 Discussion

The improvement over state-of-the-arts are from two folds: 1) AMOS enables systematic exploration of the mapping space; 2) AMOS allows flexible mapping for better performance.

First, we show the number of feasible mappings found by AMOS for different workloads in Table 6. These different mappings are different in how the iterations are mapped to Tensor Core in hardware abstraction. For example, AMOS can generate 180 different mappings for C3D, some of which require sophisticated transformation. For example, some mappings transform the 3D convolution to 2D matrix multiplication by mapping channel dimensions and image height/width/depth dimensions to Tensor Core, while other mappings first transform the 3D convolution to a series of 2D convolutions and then transform these 2D convolutions to 2D matrix multiplications to map to Tensor Core.

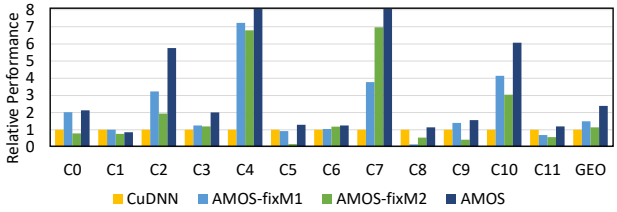


Figure 9: Comparison of CuDNN, AMOS-fixM1, AMOS-fixM2, and AMOS.

Second, AMOS enables flexible mappings for different input shapes, but hand-tuned libraries and template-based compilers always resort to fixed mappings. To illustrate this, we use the C2D layers listed in Table 5 and compare AMOS with hand-tuned CuDNN library and fixed mappings. We use AMOS to generate two fixed but representative mappings: AMOS-fixM1 (use a fixed im2col mapping) and AMOS-fixM2 (use a fixed fuse_hw mapping). AMOS-fixM1 maps iterations c, r, s together to Tensor Core dimension r_1 and maps iterations n, p, q to Tensor Core dimension i_1. AMOS-fixM2 only maps iteration c to r_1 and maps p, q to i_1. The im2col mapping is widely used in CuDNN and the fuse_hw is used in UNIT[58]. For a fair comparison, AMOS-fixM1 and AMOS-fixM2 have the same schedule exploration (e.g., tiling factors) capability as AMOS but use fixed templates to map the iterations to Tensor Core[10, 58].

Overall, compared to AMOS, the performance of AMOS-fixM1 and AMOS-fixM2 drops by 36.8% and 31.9%, respectively. For example, for layer C3, the AMOS-fixM1 and AMOS-fixM2 are inferior to AMOS because they launch too many threadblocks, which results in high wave counts per SM. As for CuDNN, its parallelism is low because of low threadblocks numbers. Moreover, CuDNN maps all the iterations c, r, s to Tensor Core, which requires a large slice of shared memory to store input data. AMOS only maps c, s to Tensor Core and splits the matrix multiplication within one warp into three steps, where shared memory can be reused and the resource pressure is alleviated. As a result, the occupancy of AMOS is higher than CuDNN (3.66×).

8 RELATED WORK

Using Accelerators with Hand-optimized Libraries. Nvidia GPU libraries such as CuDNN [42], CuBlas [41], and CUTLASS [43] are developed to use Tensor Core for GEMM and convolution. On CPUs, oneDNN [25] leverages special instruction sets such as AVX-512 to perform high-performance computation. These libraries are implemented by manual optimization and tuning, which require months or even years to develop. Deep learning frameworks such as

PyTorch [48], TensorFlow [1], and MXNet [8] rely on these libraries to deploy DNNs.

Hardware-aware Mapping: Recent compilers/mappers such as Timeloop [47], dMazeRunner [14], Triton [54], CoSA [24], SARA [66], and HASCO [60] can map software to spatial hardware with consideration for hardware constraints such as PE connection and memory capacity. These compilers leverage the hardware information to analyze the quality of mappings and usually formulate the mapping problem as a linear programming problem. They are suitable for hardware visible scenarios where hardware architectures are fully exposed to compilers. In this paper, we focus on another scenario where hardware details are not fully exposed and only programmable intrinsics are available (ISA-aware).

ISA-aware Mapping: For accelerators that only expose intrinsics to compilers, the mapping problem is called ISA-aware mapping. Halide [49] introduces the concept of *compute* and *schedule* to represent software and optimization, while TVM [9] generalizes the concept and allows users to use *tensorize* primitive to lower part of software to spatial accelerators manually. Automatic schedulers such as Halide Scheduler [2, 30, 38], FlexTensor [70], ProTuner [19], ALT [63], Rammer [34], NeoFlow [69], and Ansor [68] focus on general-purpose hardware and ignore the mapping problem for spatial accelerators such as Tensor Core. Polyhedral model is widely used in compilers for constrained optimization [5, 56, 57]. Recent works such as Tensor Comprehensions [55], Stripe [64], Tiramisu [4], PolyDL [53], and AKG [67] use polyhedral model to generate code for deep learning models. To generate code for spatial accelerators with intrinsic, existing compilers rely on hand-written templates. Typical examples include XLA [18], AutoTVM [10], ISA Mapper [52], and UNIT [58]. Their templates are hard to develop, which limits the range of supported operators for real applications. A recent compiler VeGen [11] can automatically generate templates for intrinsics on Intel CPU for AVX-512, but it doesn't consider other spatial accelerators and the approach is hard to generalize. In this paper, the proposed framework AMOS can automatically explore various valid mappings for different spatial accelerators without templates or libraries and achieve high performance.

9 CONCLUSION

Spatial accelerators enable additional performance improvement possibilities for tensor programs. However, the intrinsics exposed by these accelerators are hard to use. Existing approaches including hand-tuned libraries and template-based compilers rely on fixed manually written templates to optimize code, which leads to sub-optimal performance and heavy development cost. In this paper, we propose AMOS, which is a compilation and optimization framework for spatial hardware accelerators. AMOS proposes a novel hardware abstraction to represent the intrinsics of the accelerators. This enables systematic exploration of mapping space and flexible mapping with better performance for various tensor kernels. In experiments, AMOS achieves significant speedup on Tensor Core, AVX-512 CPU, and Mali GPU compared to the state of the art.

ACKNOWLEDGMENTS

We would first like to thank the anonymous reviewers for their suggestions. We would also like to thank Kim Hazelwood for her thoughtful suggestions during the revision process. This work is supported in part by the National Natural Science Foundation of China (NSFC) under grant No. U21B2017 and in part by the Science and Technology Commission of Shanghai Municipality, China (Grant No. 20DZ1100800). This work is also funded by Project 2020BD024 supported by PKU-Baidu Fund.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (2019), 121:1–121:12. https://doi.org/10.1145/3306346.3322967

[3] ARM. 2022. *Mali Bifrost architecture white paper.* https://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.22-Monday-Epub/HC28.22.10-GPU-HPC-Epub/HC28.22.110-Bifrost-JemDavies-ARM-v04-9.pdf

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2018. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *CoRR* abs/1804.10694 (2018). arXiv:1804.10694 http://arxiv.org/abs/1804.10694

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 101–113. https://doi.org/10.1145/1375581.1375595

[6] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[7] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. 2018. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 4 (2018), 834–848. https://doi.org/10.1109/TPAMI.2017.2699184

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[10] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 3393–3404. http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs

[11] Yishen Chen, Charith Mendis, Michael Carbin, and Saman P. Amarasinghe. 2021. VeGen: a vectorizer generator for SIMD and beyond. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery Berger, and Christos Kozyrakis (Eds.). ACM, 902–914. https://doi.org/10.1145/3445814.3446692

[12] François Chollet. 2016. Xception: Deep Learning with Depthwise Separable Convolutions. *CoRR* abs/1610.02357 (2016). arXiv:1610.02357 http://arxiv.org/abs/1610.02357

[13] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*. 46–57.

[14] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. 2019. dMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators. *ACM Trans. Embed. Comput. Syst.* 18, 5s (2019), 70:1–70:27. https://doi.org/10.1145/3358198

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[16] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (1990), 1–17.

[17] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert J. Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. 2019. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *CoRR* abs/1911.09925 (2019). arXiv:1911.09925 http://arxiv.org/abs/1911.09925

[18] Google. 2022. *XLA: Domain-specific compiler for linear algebra to optimize tensorflow computations.* https://www.tensorflow.org/xla/jit

[19] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanovic, and Ion Stoica. 2020. ProTuner: Tuning Programs with Monte Carlo Tree Search. *CoRR* abs/2005.13685 (2020). arXiv:2005.13685 https://arxiv.org/abs/2005.13685

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[21] Geoffrey E. Hinton, Sara Sabour, and Nicholas Frosst. 2018. Matrix capsules with EM routing. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HJWLfGWRb

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[23] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[24] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 554–566. https://doi.org/10.1109/ISCA52012.2021.00050

[25] Intel. 2022. *oneAPI Deep Neural Network Library.* https://github.com/oneapi-src/oneDNN

[26] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826

[27] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12. https://doi.org/10.1145/3079856.3080246

[28] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. *IEEE Micro* 40, 3 (2020), 20–29. https://doi.org/10.1109/MM.2020.2985963

[29] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[30] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.* 37, 4 (2018), 139:1–139:13. https://doi.org/10.1145/3197517.3201383

[31] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 229–241.

[32] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 789–801. https://doi.org/10.1109/HPCA51647.2021.00071

[33] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 720–733. https://doi.org/10.1109/ISCA52012.2021.00062

[34] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 881–897.

[35] Ningning Ma, Xiangyu Zhang, Jiawei Huang, and Jian Sun. 2020. Weightnet: Revisiting the design space of weight networks. *arXiv preprint arXiv:2007.11823* (2020).

[36] Svetlin A Manavski and Giorgio Valle. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics* 9, S2 (2008), S10.

[37] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. *CoRR* abs/1803.04014 (2018). arXiv:1803.04014 http://arxiv.org/abs/1803.04014

[38] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (2016), 83:1–83:11. https://doi.org/10.1145/2897824.2925952

[39] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2014. A Scheduling Framework for Spatial Architectures Across Multiple Constraint-Solving Theories. *ACM Trans. Program. Lang. Syst.* 37, 1 (2014), 2:1–2:30. https://doi.org/10.1145/2658993

[40] Nvidia. 2022. *Ampere architecture white paper.* https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[41] Nvidia. 2022. *CuBLAS.* https://developer.nvidia.com/cublas

[42] Nvidia. 2022. *CuDNN.* https://developer.nvidia.com/cudnn

[43] Nvidia. 2022. *CUTLASS.* https://github.com/NVIDIA/cutlass

[44] Nvidia. 2022. *Deep Learning Performance Guide.* https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html

[45] Nvidia. 2022. *Turing architecture white paper.* https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[46] Nvidia. 2022. *Volta architecture white paper.* https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[47] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel S. Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*. IEEE, 304–315. https://doi.org/10.1109/ISPASS.2019.00042

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. 8024–8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

[49] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 519–530. https://doi.org/10.1145/2491956.2462176

[50] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 91–99. http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks

[51] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[52] Matthew Sotoudeh, Anand Venkat, Michael J. Anderson, Evangelos Georganas, Alexander Heinecke, and Jason Knight. 2019. ISA mapper: a compute and hardware agnostic deep learning compiler. In *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*, Francesca Palumbo, Michela Becchi, Martin Schulz, and Kento Sato (Eds.). ACM, 164–173. https://doi.org/10.1145/3310273.3321559

[53] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagandeep Goyal, and Ramakrishna Upadrasta. 2021. PolyDL: Polyhedral Optimizations for Creation of High-performance DL Primitives. *ACM Trans. Archit. Code Optim.* 18, 1 (2021), 11:1–11:27. https://doi.org/10.1145/3433103

[54] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973

[55] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[56] Sven Verdoolaege. 2010. *isl*: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6327)*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[57] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[58] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 77–89. https://doi.org/10.1109/CGO51591.2021.9370330

[59] Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan Salakhutdinov. 2016. On Multiplicative Integration with Recurrent Neural Networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 2856–2864. http://papers.nips.cc/paper/6215-on-multiplicative-integration-with-recurrent-neural-networks

[60] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. 2021. HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 1055–1068. https://doi.org/10.1109/ISCA52012.2021.00086

[61] Brandon Yang, Gabriel Bender, Quoc V Le, and Jiquan Ngiam. 2019. Condconv: Conditionally parameterized convolutions for efficient inference. In *Advances in Neural Information Processing Systems*. 1307–1318.

[62] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 369–383. https://doi.org/10.1145/3373376.3378514

[63] Xi Zeng, Tian Zhi, Zidong Du, Qi Guo, Ninghui Sun, and Yunji Chen. 2020. ALT: Optimizing Tensor Compilation in Deep Learning Compilers with Active Learning. In *38th IEEE International Conference on Computer Design, ICCD 2020, Hartford, CT, USA, October 18-21, 2020*. IEEE, 623–630. https://doi.org/10.1109/ICCD50377.2020.00108

[64] Tim Zerrell and Jeremy Bruestle. 2019. Stripe: Tensor Compilation via the Nested Polyhedral Model. *CoRR* abs/1903.06498 (2019). arXiv:1903.06498 http://arxiv.org/abs/1903.06498

[65] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *CoRR* abs/1707.01083 (2017). arXiv:1707.01083 http://arxiv.org/abs/1707.01083

[66] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 1041–1054. https://doi.org/10.1109/ISCA52012.2021.00085

[67] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1233–1248. https://doi.org/10.1145/3453483.3454106

[68] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[69] Size Zheng, Renze Chen, Yicheng Jin, Anjiang Wei, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2021. NeoFlow: A Flexible Framework for Enabling Efficient Compilation for High Performance DNN Training. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[70] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 859–873. https://doi.org/10.1145/3373376.3378508