Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine

Prasanth Chatarasi
Georgia Institute of Technology
cprasanth@gatech.edu

Stephen Neuendorffer, Samuel Bayliss, Kees Vissers
Xilinx Research Labs
{stephenn, samuelb, keesv}@xilinx.com

Vivek Sarkar
Georgia Institute of Technology
vsarkar@gatech.edu

***Abstract*—Xilinx's AI Engine is a recent industry example of energy-efficient vector processing that includes novel support for 2D SIMD datapaths and shuffle interconnection network. The current approach to programming the AI Engine relies on a C/C++ API for vector intrinsics. While an advance over assembly-level programming, it requires the programmer to specify a number of low-level operations based on detailed knowledge of the hardware. To address these challenges, we introduce *Vyasa*, a new programming system that extends the Halide DSL compiler to automatically generate code for the AI Engine. We evaluated Vyasa on 36 CONV2D workloads, and achieved geometric means of 7.6 and 24.2 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 75.6% of the peak performance respectively).**

I. Introduction

It is widely recognized that a major disruption is underway in computer hardware as processors strive to extend, and go beyond, the end-game of Moore's Law. As part of the disruption, there is an emphasis on specializing SIMD units to improve energy efficiency benefits for compute-intensive domains. An important specialization, which is referred to as "2D vector SIMD datapath" [1]–[3], is the ability of each vector lane to execute more than one scalar operation and to chain the results from one operation to another. Another specialization replaces expensive data permutation units (e.g., shuffle units) [4], [5] with sophisticated, programmable interconnection networks (a.k.a shuffle networks) between the SIMD datapath and vector register file [3], [6].

A recent industry example with these specializations is the Xilinx Versal AI Engine [7], a high-performance VLIW SIMD core with performance comparable to traditional FPGA solutions for Computer Vision, Deep Learning, and 5G wireless domains, but with 50% less power and up to 8× more compute capacity per silicon area [7]. The Versal AI Engine series VC1902 has a total of 400 AI Engines that can deliver a peak performance of 6.4 TOPS, 25.6 TOPS, and 102.4 TOPS for 32-bit, 16-bit, and 8-bit operands, respectively [8].

Tensor convolution is a widely used mathematical operation in these domains, and it is becoming increasingly important with the rise of its use in image processing workflows [9]–[11] and with the proliferation of deep learning models [12]–[15] in data centers, edge, and mobile devices. There has been a lot of prior work on optimizing tensor convolutions for a variety of target hardware devices such as CPUs [9], [10], [16], GPUs [9], [16], [17], FPGAs [16], [18]–[21], and Dataflow accelerators [20], [22]–[24]. However, while it is desirable to automatically generate optimized code for new high performance processor architectures like the AI Engine from high-level descriptions, doing so can be challenging. This work demonstrates the ability to optimize tensor convolutions for the AI Engine automatically and to obtain near-peak performance for various workloads while using the Halide DSL as a high-level programming model,

Challenges. Achieving peak performance on the AI Engine requires leveraging several architectural features to maximize vector datapath occupancy during program execution. Unlike standard SIMD architectures that operate on 1D vectors, the AI Engine architecture includes 2D vector operations for some datatypes, which conceptually implement the fusion of several 1D vector operations. Unlike other architectures, the AI Engine does not implement direct support for unaligned loads, scalar broadcasts, and data manipulation operations. Instead, the AI Engine architecture includes a novel shuffle network that selects the desired elements of a vector register for a vector operation instead of explicitly shuffling and storing them into another vector register. To effectively leverage these features, the layout of data in memory must match the capabilities of the shuffle network.

Existing AI Engine compilers perform VLIW scheduling but do not perform auto-vectorization. High-performance vector code must be expressed using intrinsic architectural functions for most vector operations, including the shuffle network configuration. Optimizing programs in this way can be time-consuming, even for experts. Simultaneously, there is a wide variety of tensor convolution operators in common use; for instance, deep neural networks may contain regular 2D convolutions, depth-wise convolutions, and point-wise convolutions. Even within the same network, the shape of tensor data can vary radically between the early and late layers in DNN models. We find that no single optimization strategy is an optimal choice for all these scenarios. Reducing the need for manual optimization and quickly adapting to new tensor operations through automatic optimization avoids these problems. The overall goal of our work is *to automate the generation of high-performance vector code for tensor convolutions based on their variations and shapes, while exploiting the unique capabilities of the Xilinx AI Engine without requiring manual effort in development and tuning*. Achieving this goal requires significant loop-level reuse analysis, code transformations, and

data-layout transformations, along with optimized low-level code generation taking into account the shuffle network. Even though our approach is tied to the AI Engine, ideas in the approach are very applicable to other specialized SIMD units with similar architectures. The main technical contributions of this paper are briefly described below:

- We introduce an intermediate representation, *Triplet*, to symbolically capture the loop body of a tensor convolution and also to simplify analyses and transformations to generate optimized code for the AI Engine.
- We propose a novel multi-step compiler approach that includes analyses and transformations to 1) exploit the 2D SIMD datapath by identifying multiple 1D logical vector operations that can be legally fused, 2) realize unaligned loads, scalar broadcasts, data manipulation using the shuffle network, 3) improve memory utilization by exploiting vector register reuse and loop optimizations, and 4) generate code that is more amenable to enabling VLIW instruction scheduling for the AI Engine.
- We created a new tool, Vyasa[1], to implement our multi-step compiler approach. Vyasa is built on the Halide framework [9] and includes extensions needed for the AI Engine that is not supported by Halide. Given a tensor convolution specification in the Halide language and workload sizes, Vyasa generates high-performance C-code with vector intrinsics for the AI Engine.
- We evaluated Vyasa on 36 CONV2D workloads using the in-house cycle-accurate simulator [2]. Our results show geometric means of 7.6 and 24.2 MACs/cycle for 32-bit and 16-bit operands (95.9% and 75.6% of the peak performance respectively). For four of these workloads for which expert-written implementations were available to us, Vyasa achieved a geometric mean performance improvement of 1.10× from extended Halide code that is around 50× smaller than the expert-written C/C++ code.

II. Background

In this section, we start with an overview of tensor convolutions, and then we briefly summarize the key architectural features of the Xilinx Versal AI Engine.

A. Tensor Convolutions

Convolution is a mathematical operation which computes the amount of overlap of a function g as it is shifted over another function f. In this section, we restrict our attention to describing CONV2D, a popular convolution operator widely used in Deep learning [12]–[15], [25], [26] and Computer Vision [9]–[11], [27]. In these domains, the function `f` and `g` are referred to as the "input" tensor (a.k.a image/activations) and "weight" tensor (a.k.a filters/kernels), respectively. The CONV2D deals with three four-dimensional tensors, i.e., Output (O), Weight (W), and Input (I), whose dimensions are described below.

Tensor	Dim1	Dim2	Dim3	Dim4
Output (O)	Width (X)	Height (Y)	Channels (K)	Batch (N)
Weight (W)	Width (R)	Height (S)	Channels (C)	Batch (K)
Input (I)	Width (X')	Height (Y')	Channels (C)	Batch (N)

The mathematical expression of the CONV2D operations is shown below, where f refers to stride factor.

$$O(x,\ y,\ k,\ n) = \sum_{c}^{C}\sum_{s}^{S}\sum_{r}^{R} W(r,\ s,\ c,\ k) \times I(x \times f + r,\ y \times f + s,\ c,\ n)$$

Specialized versions of CONV2D are often interesting. For example, image processing pipelines such as Blur detection, Harris corner detection algorithms in compute vision domain often involve CONV2D's operating over two-dimensional tensors only (number of channels and batch size are set to one). In Convolutional Neural networks (CNNs), the number of channels is often large, particularly in later layers. Other types of convolution layers are special cases of CONV2D, such as fully-connected, point-wise, depth-wise separable, and spatially separable convolutions. These variations can be viewed as constraints on the regular CONV2D as follows:

Operator	Constraints on CONV2D
Point-wise (PW)	Filter width = Filter height = 1
Fully-connected (FC)	Filter width = Input width Filter height = Input height
Spatially separable (SS)	Filter width = 1 or Filter height = 1
Depth-wise separable (DS)	Input channels = Filter channels = 1

B. Xilinx AI Engine

Driven by the performance and energy efficiency requirements of many computing applications, Xilinx introduced Versal Advanced Compute Acceleration Platform (ACAP) [8], [28], a fully software-programmable, heterogeneous compute platform. The Versal platform consists of three types of programmable processors – Scalar Engines (CPUs), Adaptable Engines (Programmable Logic), and an array of Intelligent Engines (AI Engines) [8]. In this work, we focus on AI Engines, which are specialized SIMD and VLIW high-performance processors for compute-intensive applications.

An AI Engine includes a 2D SIMD datapath for fixed-point vector operations (our focus), a 1D SIMD datapath for floating-point vector operations, and a scalar unit for scalar operations. Each AI Engine also has access to 128KB scratch-pad (a.k.a data/local) memory, a 16KB program memory, and a 256B vector register file (a total of 16 registers with each size being 128 bits). These high-performance AI Engines are programmed using the C/C++ programming language with optional pragmas. A simplified overview of the key architectural features of the AI Engine core is shown in fig. 1.

1) Two-dimensional SIMD Datapath. The fixed point vector unit of the AI Engine is a two-dimensional SIMD datapath, and vector operations on the 2D SIMD datapath

[1]Vyasa means "compiler" in the Sanskrit language, and also refers to the sage who first compiled the Mahabharata.

[2]Since the AI Engine architecture was developed for real-time processing applications which require deterministic performance, the simulator results are reliably correlated with the actual performance of the AI Engine hardware.

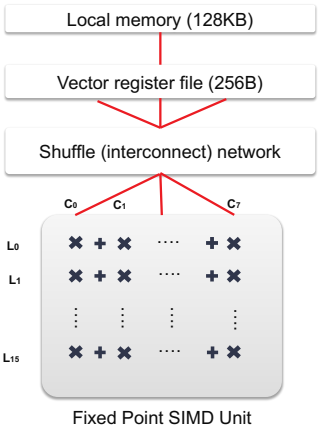

Fig. 1. A pictorial overview of the key architectural features of the Xilinx AI Engine, i.e., 2D vector SIMD datapath and shuffle network.

are described using *lanes/rows* and *columns*. The number of lanes corresponds to the number of output values generated from the vector operation. The number of columns is the number of operations that are done per output lane, with each of the results being reduced together. This technique of executing back to back dependent scalar operations along a vector lane is popularly known as operation chaining [1] and can improve energy efficiency by not writing intermediate values back to the register file. Furthermore, the number of columns is dependent on the operand precision. Operations on 32-bit types are organized as 8 lanes with 1 column, without internal reduction. However, the operations on 16-bit types are organized as either 16 lanes with 2 columns or 8 lanes with 4 columns. Also, the operations on 8-bit types are organized as 16 lanes with 8 columns. As a result, the 2D datapath can perform either 8 MACs on 32-bit inputs, 32 MACs on 16-bit input, or 128 MACs on 8-bit input per cycle. Currently, the AI Engine compilers do not advertise support for auto-vectorization, and application programmers write vectorized code explicitly using architecture-specific vector intrinsics.

2) Shuffle network. A novelty of the AI Engine architecture is its *shuffle network*, a flexible interconnection network between the 2D SIMD datapath and vector register file to allow flexible data selection from the input vector registers for the multipliers of each lane and column of the SIMD datapath. The ability to configure the shuffle network for each vector operation is exposed to programmers via the vector intrinsic functions arguments. The granularity of data selection using the shuffle network on the vector registers is 32b, and so the network allows full flexibility for making data selection, replication, and permutation on vectors of 32b data types. However, for data types of smaller sizes such as 16b and 8b data types, the shuffle network imposes further constraints on data selection.

3) VLIW capabilities. The AI Engine is a very long instruction word (VLIW) architecture with up to six issue slots per instruction and one instruction per clock. Each VLIW instruction includes up to two scalar operations, two vector load operations, one vector store operation, and one fixed/floating-point vector operation. Currently, the AI Engine compilers support automatic software pipelining [29] of innermost loops to exploit instruction-level parallelism.

III. Our Approach

In this section, we introduce our approach to generating a high-performance vector code for a given high-level specification of tensor convolution and its workload sizes that fit into a single AI Engine's data memory. These vector codes are intended to execute on a single AI Engine and will be integrated by a high-level compiler to run larger tensor convolutions across multiple AI Engines. Our approach is summarized in fig. 2, and it is implemented in a tool called Vyasa which is developed as an extension to the Halide [9] framework.

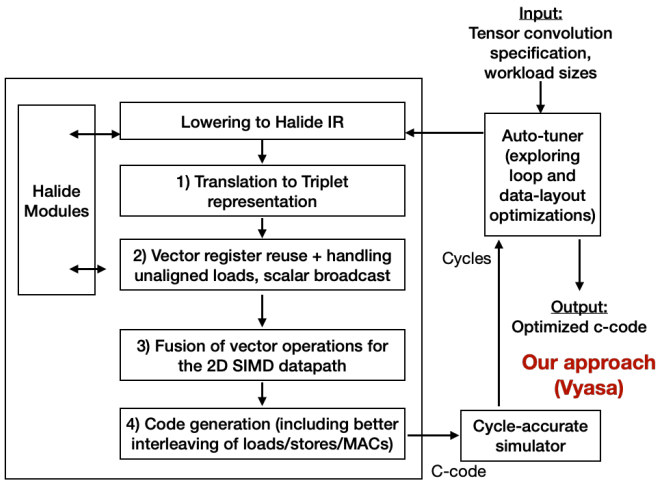

Fig. 2. Workflow of our approach (Vyasa) which is implemented as an extension to the Halide framework [9].

Our approach begins with an auto-tuner taking the specification of a tensor convolution in the Halide language and corresponding workload sizes. Then, the auto-tuner iterates through each possible schedule in the space of loop transformations and data-layouts and invokes our *multi-step compiler approach* to generate vector c-code corresponding to the schedule. Then, our approach evaluates the generated code using a cycle-accurate simulation of the AI Engine and chooses the best one among all schedules to finally emit as the performant output code. Now, we briefly describe various steps involved in our multi-step compiler approach.

A. Translating into Triplet Representation

Tensor convolutions are often specified as multi-dimensional perfectly nested loops, where each statement of the loop body has two aspects – 1) A group of multiply-and-accumulate (MAC) operations over input and weight tensors, and 2) An update (reduction) operation to the output tensor. Since each statement in the loop body performs a reduction and the reduction is commutative, each statement's order doesn't impact its correctness. Hence, a representation holding information about the two significant aspects described above

```
Buffer<int16> I(X',Y'); Buffer<int16> W(4,3);
Var x, y; RDom r(4, 3); Func O; //output

//(a) Description of the convolution computation
O(x,y) += W(r.x, r.y) * I(x+r.x, y+r.y);

//(b) A sample schedule: Unrolling reduction loops
//Vectorizing loop corresponding to image width
O.update().unroll(r.x, 4).unroll(r.y,3)
     .vectorize(x, 16);

//(c) Intermediate code after lowering
for y:
 for x: (vectorized)
   O(x:x+15,y) += W(0,0) * I(x:x+15,y);
   O(x:x+15,y) += W(1,0) * I(x+1:x+16,y);
   O(x:x+15,y) += W(2,0) * I(x+2:x+17,y);
   O(x:x+15,y) += W(3,0) * I(x+3:x+18,y);
   ......
```

Fig. 3. Algorithmic description of the convolution of a 4x3 filter over an input 2D image in the Halide language [9]. A(a:b,c) is a short hand vector notation for denoting a contiguous slice from A(a,c) to A(b,c) in one direction.

for each statement is sufficient to precisely capture the body. We call this representation a "triplet" since it symbolically holds information about the accesses of two operands of each multiplication and the update operand of each statement.

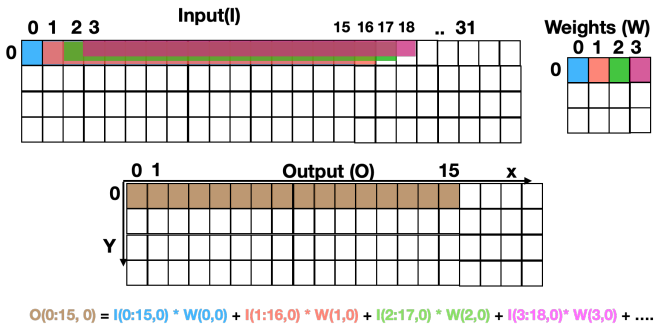

Fig. 4. A pictorial overview of the convolution of 4x3 filter based on the schedule described in fig. 3(b) at the loop iterations x = 0 and y = 0.

We consider the convolution of a filter with size 4x3 on a 2D input image as a running example, whose specification is shown in fig. 3(a). A sample schedule is also shown in fig. 3(b), and it refers to unrolling loops corresponding to filter dimensions (`r.x, r.y`) and vectorizing the `loop-x` with vector length as 16. The pictorial overview of the computation at the loop iteration x = 0, y = 0 is shown in fig. 4. The first step in our approach after lowering the specification and its schedule into Halide IR is to translate the loop body into our triplet representation. For instance, the triplet representation of the loop body in fig. 3(c) is shown in Table I, where each row symbolically captures the access patterns of multiplication operands and update operands of a statement.

B. Shuffle Interconnection Network

Our approach leverages the AI Engine's unique shuffle network to exploit temporal locality via vector register reuse and spatial locality via adjacent scalar operands in memory.

TABLE I
TRIPLET REPRESENTATION OF THE LOOP BODY IN FIG. 3(C)

Update Operation Operand	MAC Operations	
	Operand1	Operand2
`O(x:x+15, y)`	`W(0, 0)`	`I(x:x+15, y)`
`O(x:x+15, y)`	`W(1, 0)`	`I(x+1:x+16, y)`
`O(x:x+15, y)`	`W(2, 0)`	`I(x+2:x+17, y)`
`O(x:x+15, y)`	`W(3, 0)`	`I(x+3:x+18, y)`
..	..	..

During this process, unaligned vector loads and scalar broadcast operations, which are common in vectorization of tensor convolutions, are indirectly addressed.

1) Exploiting vector register reuse. Tensor convolutions often exhibit significant data reuse between vector loads. For instance, the two vector loads `I(x:x+15,y)` and `I(x+1:x+16,y)` have 15 data elements in common. Exploiting vector register reuse by reusing those common elements instead of fetching again from memory is essential to reduce memory traffic and achieve better efficiency. The AI Engine architecture provides support for grouping vector registers into a larger vector register. Leveraging this, our approach groups individual vector loads having the reuse and constructs a larger aligned vector load that subsumes the individual vector loads having reuse. During vector operations, appropriate data elements are then selected from vector registers using the architectural shuffle network.

Our approach identifies opportunities for vector register reuse by constructing a *reuse graph*, an undirected graph where each node denotes a vector load, and an edge between two nodes denotes the presence of common elements. The reuse graph corresponding to the vector loads of the tensor `I` in table I is shown in fig. 5, for instance, nodes `I(x:x+15,y)` and `I(x+1:x+16,y)` corresponds to two vector loads and the edge between them denotes the presence of common elements/reuse.

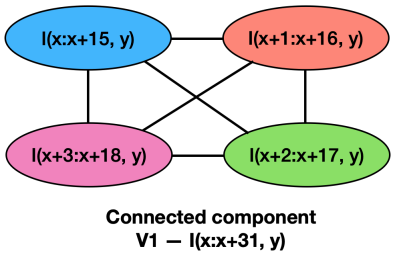

Fig. 5. Reuse graph involving vector loads of the tensor `I` in Table I

Each connected component in the reuse graph can be viewed as a larger vector load that subsumes individual vector loads. For instance, the connected component `I(x:x+31,y)` in fig. 5 represents a larger vector load subsuming the vector loads `I(x:x+15,y)`, `I(x+1:x+16,y)`, `I(x+2:x+17,y)`, and `I(x+3:x+18,y)`. The larger vector load is also padded to ensure that vector load instructions are always aligned on vector boundaries.

2) Exploiting Spatial Locality. Tensor convolutions often exhibit significant spatial locality between scalar operands, e.g., the scalar operands `W(0,0)` and `W(1,0)` in Table I are contiguous in the memory and also require broadcasting for vector operations. Similar to our approach in exploiting vector register reuse, we construct another reuse graph to identify scalar operands adjacent in data memory and can be subsumed as part of a single vector load. For instance, the operands `W(0,0)`, `W(1,0)`, `(2,0)`, `W(3,0)` can be fetched from memory using a single aligned vector load (say V2) of `W(0:7, 0)`. Individual scalar elements can be extracted from a vector register and broadcast to different lanes of a SIMD operation using the architectural shuffle network without explicit storage in a vector register.

C. 2D Vector SIMD Datapath

A key distinguishing feature of the AI Engine relative to the traditional SIMD units is the presence of a two-dimensional SIMD datapath that performs a reduction across all columns of a SIMD lane. A single logical 1D vector operation can occupy a single column of 2D datapath, but the vector operations on the 2D SIMD datapath require using all the columns of the datapath and don't allow partial utilization. Hence, our approach identifies and logically groups (fusing) all 1D logical vector operations that contribute to the same output through accumulation/reduction and use the same set of vector register operands. The identification is made by searching in the triplet representation for operations having the same update operand and the same set of vector registers as multiplication operands. Finally, our approach partitions the logical groups based on the number of columns available for the given operand type and the constraints imposed by the shuffle network on the data selection over vector register operands. If the data selection required for the operands of fused vector operations is incompatible with the constraints of the shuffle network, then our approach generates a compilation error and prunes that candidate code variant. There are four valid fusible logical 1D operations for each row of the filter in Table I; our approach groups them into two fused vector operations whose pictorial overview is described in fig. 6.

D. Code Generation

Our approach extends the code generation capabilities in the Halide [9] by implementing a code generator for the triplet representation to generate explicitly vectorized code using AI Engine intrinsic functions. A naive approach to code generation can be implemented by first emitting all vector loads, followed by all vector MAC operation, and then finally, all vector stores. However, this naive approach results in variables (loads) having large live ranges, possibly leading to register spills and preventing software pipelining. Optimization of memory accesses can be challenging for the downstream compilers only given the generated intrinsic code. Hence, our approach also reorders memory accesses and interleaves them with vector MAC operations during the code generation process to reduce each variable's live range. This process is relatively easy given the information about memory access patterns in Halide and helps the downstream compilers to improve the packing of stores, loads, and vector MACs into VLIW instructions.

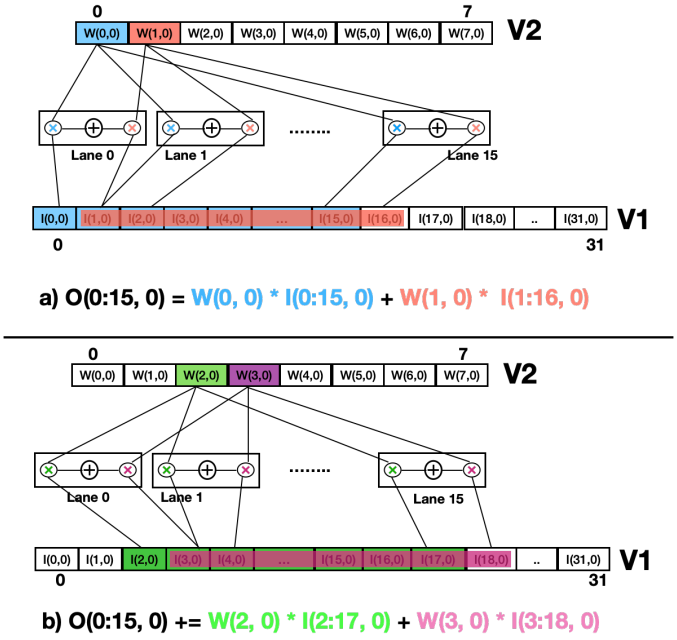

Fig. 6. An overview of the two fused vector operations (a and b) over the vector registers V1, V2 for input and weights, respectively of the running example at iterations x=0 and y=0. The shuffle network of the AI Engine helps each multiplier of the 16 lanes and 2 columns of the 2D datapath to choose required elements from the vector registers.

E. Auto-tuner

Steps 1-5 in our multi-step compiler approach generates the vectorized code for a given specification of tensor convolution, a schedule from the auto-tuner, and workload sizes. The auto-tuning functionality of the Halide framework supports only multi-staged pipelines [30], [31], but our focus is only on a single-stage involving a tensor convolution. To find the best schedule for the operation, we implemented a custom auto-tuner in our approach to explore all possible schedules involving loop interchange, loop unroll and jam, choice of the loop for vectorization, and data-layout choices such as dimension permutation and data tiling. Our approach applies the following pruning strategies: 1) Unrolling of reduction loops to avoid memory traffic in writing and reading intermediate (partial) results, and 2) applying bounds on the unroll and jam factors to avoid spending compilation time on large code since AI Engine has only 16KB of program memory per core. Our auto-tuner evaluates each point in the pruned search space by generating the vectorized C-code, compiling with the AI Engine compiler, and executing it using the in-house cycle-accurate architecture simulator. With performance as the primary optimization goal, our approach obtained a geometric mean performance improvement of 1.10× fewer cycles than the expert-written and tuned codes available for

four workloads, showing that automatic exploration can find useful design points which are not obvious to humans.

IV. Experiments

We evaluated our approach over a total of 36 workloads involving variations of CONV2D over two operand precisions (32-bit and 16-bit) on a single AI Engine. Each workload represents a unique combination of a convolution operation, tensor shapes, and operand precision. The configuration is shown in Table II and includes a 128KB local memory pre-loaded with all the data required for the evaluation of each workload. This accurately reflects real system performance where local memory is usually double-buffered and loaded by separate data movement engines. The configuration also includes a vector register file of size 256B (a total of 16 registers with each size as 128 bits) in between the SIMD datapath and the local memory. We used the AI Engine's cycle-accurate simulator to evaluate the functionality and performance of our generated codes. We define the performance (MACs/Cycle) of convolution implementation as the total number of MAC operations in the convolution divided by the total number of execution cycles taken by the implementation.

TABLE II
THE AI ENGINE CONFIGURATION USED IN OUR EVALUATION.

Parameter	32-bit	16-bit
2D SIMD data path	8 x 1	16 x 2
Peak compute	8 MACs/cycle	32 MACs/cycle
Scratchpad memory	128 KB @ 96B/cycle	
Scratchpad memory ports	32B 2 read and 1 write	
Vector register file	256 B	

A. CONV2D in Computer Vision (2D Tensors)

In the following experiments, we compare two experimental variants: 1) Code written by an expert (for 3×3 and 5×5 filters) available as part of the Xilinx's AI Engine compiler infrastructure, 2) Code generated by our approach leveraging the auto-tuner. Both codes are designed to produce a 256×16 tile of a larger image. We observe from fig. 7 that our approach achieved a geometric mean performance improvement of 1.10× from the Halide codes compared with the available expert-written codes.

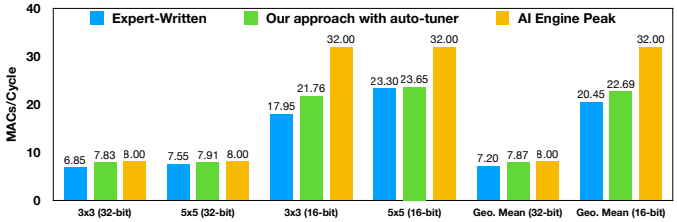

Fig. 7. Comparison of our approach with auto-tuner against the available expert-written codes for CONV2D operation with 3×3 and 5×5 filters.

The auto-tuner of our approach was able to find better schedules than used in the expert-written codes (roof-line graphs for the workloads is shown in fig. 8), including non-trivial unroll and jam factors along the image height (`loop-y`) dimension for better reuse. These factors also enabled more opportunities in the loop body for the downstream compilers to perform better software pipelining. Furthermore, since workload sizes are also expressed in the Halide codes, our approach annotated the loops of generated codes with loop bound pragmas enabling the downstream compilers to estimate loop pre-amble and post-amble overheads accurately and generate better VLIW code. Such overheads can be significant, particularly for tiled inner loops executed many times.

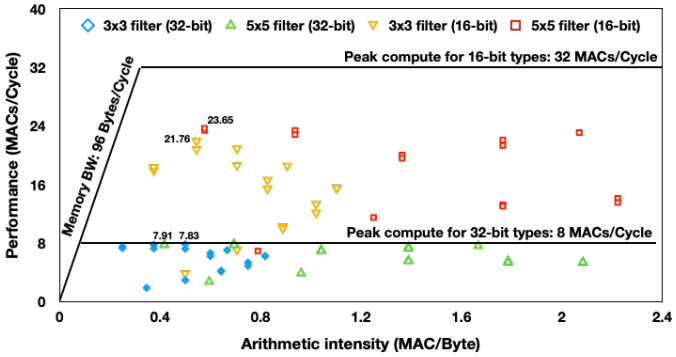

Fig. 8. Roof-line graphs of four workloads considered in fig. 7, where each data point is a schedule explored by the auto-tuner.

In the case of the 3x3 and 5x5 filters with 16-bit operands, the total number of fusible logical 1D vector multiplications corresponding to each row of the filters is odd. Hence, our approach padded the filters with an additional column to generate an even number of fusible 1D operations and map onto the two columns present in the 2D SIMD datapath for 16-bit types. Expert-written codes used a slightly different optimization strategy, fusing some logical 1D vector multiplications corresponding to different rows of the filters to reduce unused vector lanes due to padding. This was accomplished by carefully merging the required input image data from different rows into a single vector register in a way that is not currently accessible in our automated approach. However, we see that the code generated using our approach can perform better than the expert-written code by leveraging non-trivial unroll and jam factors.

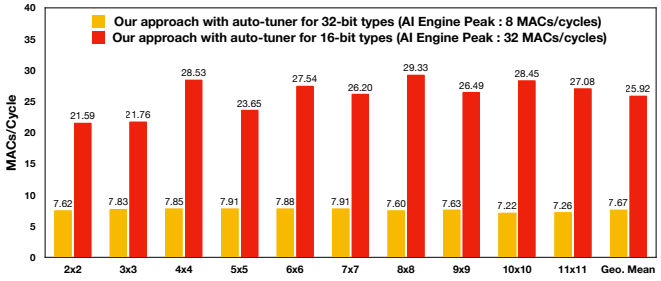

Fig. 9. Performance of our approach generated codes for CONV2D workloads in Table III over filter sizes from 2 to 11.

TABLE III
CONV2D WORKLOADS OVER 2D TENSORS USED IN OUR EVALUATION AND OPTIMAL SCHEDULES FROM AUTO-TUNER

Output (O) size	Weight (W) size	Input (I) size	#MACs	Optimal schedule from auto-tuner				
				Unroll and Jam factors				Loop order
				32-bit		16-bit		
				x	y	x	y	
256 x 16	2 x 2	264 x 17	16384	1	4	1	8	xy
	3 x 3	264 x 18	36864	1	4	1	2	xy
	4 x 4	264 x 19	65536	1	2	1	1	xy
	5 x 5	264 x 20	102400	1	2	1	1	xy
	6 x 6	264 x 21	147456	1	1	1	1	xy
	7 x 7	264 x 22	200704	1	1	1	1	xy
	8 x 8	264 x 23	262144	1	4	1	1	xy
	9 x 9	264 x 24	331776	1	4	1	1	xy
	10 x 10	264 x 25	409600	1	4	1	4	xy
	11 x 11	264 x 26	495616	1	4	1	4	xy

In addition to 3x3 and 5x5 filters, we have evaluated other filter sizes commonly used in computer vision. Table III presents those workload sizes, total MAC operations involved in each workload, and optimal schedules reported by the auto-tuner. We padded each non-even sized 16-bit filter with an additional column for evaluation, and report achieved performance (MACs/cycle) in fig. 9. Our approach made a geometric mean performance of 7.67 and 25.92 MACs/cycle for 32-bit and 16-bit types, respectively, for the workloads in Table III. The auto-tuner chose the `loop-x` for vectorization for all the workloads because it has more reuse opportunities and has a larger number of iterations than the `loop-y`. The optimal unroll and jam factors are not the same for all the workloads and vary for different precisions of the same filter size. Even though increasing unroll and jam factors improve the reuse opportunities, it often resulted in register spills after a threshold and interfered with software pipelining of inner loops. Furthermore, larger unroll and jam factors along the `loop-x` resulted in larger connected components of the reuse graph and required a larger vector register than the maximum possible (e.g., 1024b for 32-bit operands) in the hardware.

B. CONV2D in Deep Learning

We considered a wide variety of CONV2D operations in the deep learning domain such as regular (REG) CONV2D over various filter sizes, point-wise (PW), spatially separable (SS), depth-wise separable (DS), and fully-connected (FC) operations. Table IV presents those workload sizes (with unit batch size, i.e., N = 1), total MAC operations involved in each workload, and optimal schedules reported by the auto-tuner. Since the memory footprint of typical CONV2D operations don't fit into the local memory, we chose the similar tensor memory footprint used in section IV-A. For these workloads, our approach achieved a geometric mean performance of 7.67 and 22.53 MACs/cycle for 32-bit and 16-bit types respectively, as shown in fig. 10.

The auto-tuner identified vectorization along `loop-x` (i.e., output width) to be beneficial for REG-5x5 and REG-7x7 workloads, because there exist more opportunities for vector

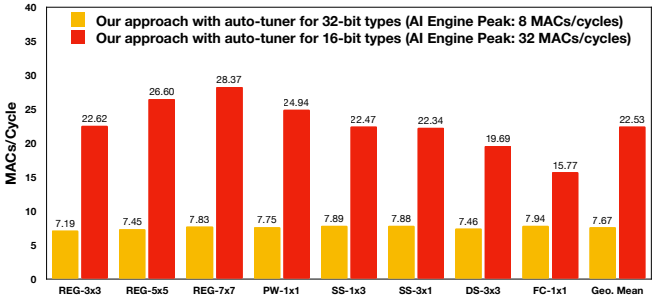

Fig. 10. Performance of our approach generated codes for a wide variety of CONV2D workloads in Deep Learning models.

register reuse (convolutional reuse) in `loop-x` with larger kernels sizes. But, for workloads such as PW (REG-1x1) and FC that have little to no convolutional reuse in `loop-x`, vectorization was performed on `loop-k` (i.e., output channels).

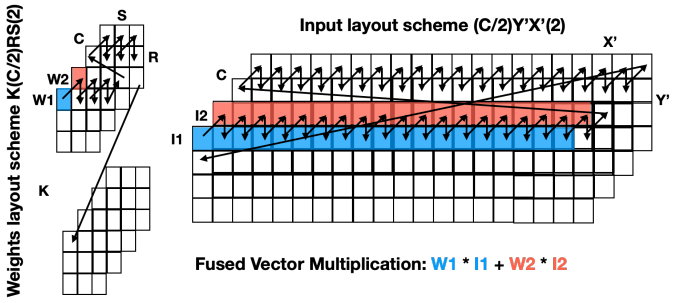

Fig. 11. Data-layouts of input and weight tensors of the 16-bit REG-3x3 workload, to enable the fusion of 1D logical vector multiplications along the channels, thereby avoiding the padding required for weights.

In these workloads, there exists an even number of fusible 1D vector multiplications along the filter channels. Hence our approach didn't require any padding to the filters, except for the depth-wise CONV2D workload, which has only one channel. However, the data-layouts of these workload tensors need to be modified to support the fusion of 1D logical vector multiplications along the channels. An example data-layout for the input and weights of the 16-bit REG-3x3 workload identified by our approach for the fusion along channels is shown in fig. 11, where the data-layout scheme for the input tensor (C/2)Y'X'(2) refers to first laying out a block of two channels followed by width, height, and remaining channels.

Along with the advantages of avoiding padding, data-layouts can be used for exploring better schedules as well. Such data-layout schemes over the workload tensors should respect two constraints: 1) The required number of data elements of each operand of the fused vector multiplication should fit into logically the maximum vector register size, and 2) The required data selection parameters over the vector register should respect the shuffle network constraints. Our auto-tuner was able to automatically explore a variety of such valid data-layout schemes in our evaluation. Although the architecture can support the resulting data-layouts, they can be rather complex and non-intuitive (e.g., (K/16)SR(C/2)(16)(2)

TABLE IV
CONV2D WORKLOADS IN DEEP LEARNING DOMAIN USED IN OUR EVALUATION (VARIABLES DESCRIBED IN SECTION II) AND OPTIMAL SCHEDULES.

CONV type	Output (O) size (XxYxK)	Filter (F) size (RxSxCxK)	Input (I) size (X'xY'xC)	#MACs	Precision	Optimal schedules from the auto-tuner								
						Data layouts			Vector loop	SW loop	Unroll and Jam factors			Loop order
						O	W	I			x	y	k	
(REG)	128x2x16	3x3x8x16	144x4x8	294912	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	x	1	2	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	1	1	yxk
		5x5x8x16	144x6x8	819200	32-bit	KYX	KCSR	CY'X'	x	x	1	1	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	2	1	kyx
		7x7x8x16	144x8x8	1605632	32-bit	KYX	KCSR	CY'X'	x	x	1	2	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	2	1	kyx
(PW)		1x1x8x16	144x2x8	32768	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	x	1	2	1	kyx
					16-bit	YXK	(K/16)SR(C/2)(16)(2)	Y'X'C	k	k	1	2	1	xyk
(SS)		1x3x8x16	144x4x8	98304	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	p	1	2	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	2	1	kyx
		3x1x8x16	144x2x8	98304	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	x	1	1	1	kyx
					16-bit	YXK	(K/16)SR(C/2)(16)(2)	Y'X'C	k	k	1	2	1	xyk
(DS)		3x3x16x16	144x4x16	36864	32-bit	KYX	KCSR	CY'X'	x	x	1	2	1	kyx
					16-bit	KYX	KCSR	CY'X'	x	x	1	2	1	kyx
(FC)	4096x1x1	1x1x8x4096	16x1x8	32768	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	k	1	1	1	kyx
					16-bit	YXK	(K/16)SR(C/2)(16)(2)	Y'X'C	k	k	1	1	1	xyk

for weights in 16-bit PW-1x1). Manually identifying such a data layout and writing the corresponding intrinsic-based code is exceptionally challenging and error-prone, even for experts, thereby demonstrating the benefits of our automatic approach.

The FC workload in our evaluation has lower arithmetic intensity (because it lacks convolutional reuse) and lies on the left side of the inflection point of the roof-line graph of the AI engine, indicating memory-bound execution. The workload peak performance based on its arithmetic intensity is 21.22 MACs/cycle, and our approach achieved 75% of the peak.

Overall, our evaluation over all the workloads shows geometric means of 7.6 and 24.2 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 75.6% of the peak performance respectively). This difference in efficiency is not surprising, since it is more challenging to utilize two columns in the SIMD data path in the case of 16-bit operands, compared to a single column in the case of 32-bit operands. However, the absolute performance in the 16-bit case is still significantly higher than the 32-bit case, despite a lower efficiency.

V. RELATED WORK

The Halide framework [9] for image processing pipelines have been shown to improve the productivity of application programmers, while generating high-performance code for a variety of architectures, including CPUs, GPUs, and FPGAs. Recently, Vocke et al. [32] extended the Halide framework to support specialized Digital Signal Processors (DSPs) of the Intel Imaging Processing Units (IPUs). Furthermore, Halide has the support for the Hexagon Vector eXtensions (HVX) on the Qualcomm Hexagon DSP processors. However, none of the above prior work addresses the 2D SIMD datapaths and shuffle network, which are unique to the AI Engine. To the best of our knowledge, the only prior work on auto-vectorizing for a 2D SIMD datapath is the work by Dasika et al. [2] for the PEPSC's architecture, where the authors have proposed a greedy compiler approach to identify fusible vector operations.

Exploiting vector register reuse (including partial reuse) on SIMD units is a vital optimization to achieve high-performance, and prior work exploited the reuse by shuffling the vector registers using the data manipulation/shuffle units [33]–[35]. However, our approach constructs a larger vector load covering the loads having the reuse and uses the AI Engine's unique shuffle network to select the desired elements. Furthermore, our approach uses the shuffle network to address the unaligned vector loads and scalar broadcasts without requiring any additional hardware support.

The vector codes generated by our approach can be viewed as high-performance primitives for a single AI Engine. These primitives will be composed and integrated by a high-level compiler to run larger tensor convolutions across multiple AI Engines. Some of the prior works that have followed the similar strategy of automating the library/primitive development for the performance-critical kernels are SPIRAL [36] for the domain of linear transforms, ATLAS [37] for the basic linear algebra subroutines (BLAS), and FFTW [38] for the discrete Fourier transforms.

VI. CONCLUSIONS & FUTURE WORK

In this work, we introduced Vyasa, a high-level programming system built on the Halide framework, to generate high-performance vector codes for the tensor convolutions onto the Xilinx Versal AI Engine. Our proposed multi-step compiler approach leverages the AI Engine's unique capabilities of the 2D SIMD datapath and the shuffle interconnection networks to achieve close to the peak performance for various workloads. Manually identifying best schedules and writing the corresponding intrinsic-based code is exceptionally challenging and error-prone, even for experts, thereby demonstrating the benefits of our automatic approach. Our results show a geometric mean of 7.6 and 24.2 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 75.6% of the peak performance respectively). In the future, we plan to extend our system to other computationally expensive linear algebra kernels. We also plan to integrate the generated high-performance codes into a high-level compiler to run larger tensor convolutions across multiple AI Engines.

VII. Acknowledgments

The authors would like to acknowledge Kristof Denolf and Jack Lo from Xilinx Research Labs for their constructive suggestions on the paper and their help with understanding the performance intricacies of the AI Engine along with a brief description of their manually written kernels.

References

[1] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A High-Performance DSP Architecture for Software-Defined Radio," *IEEE Micro*, vol. 27, no. 1, pp. 114–123, Jan 2007.

[2] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke, "PEPSC: A Power-Efficient Processor for Scientific Computing," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 101–110.

[3] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "AnySP: Anytime Anywhere Anyway Signal Processing," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 128–139, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1555815.1555773

[4] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 265–274. [Online]. Available: https://doi.org/10.1145/1995896.1995938

[5] F. Franchetti and M. Püschel, "Generating SIMD Vectorized Permutations," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 116–131.

[6] S. Seo, M. Woh, S. Mahlke, T. Mudge, S. Vijay, and C. Chakrabarti, "Customizing Wide-SIMD Architectures for H.264," in *Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, ser. SAMOS'09. IEEE Press, 2009, p. 172–179.

[7] *Xilinx AI Engines and Their Applications*, Xilinx, 10 2018, v1.0.2. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf

[8] *Versal: The First Adaptive Compute Acceleration Platform (ACAP)*, Xilinx, 9 2019, v1.0.1. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf

[9] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[10] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic Optimization for Image Processing Pipelines." in *ASPLOS*. ACM, 2015, pp. 429–443.

[11] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–25, 2017.

[12] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[13] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[14] A. Toshev and C. Szegedy, "Deeppose: Human pose estimation via deep neural networks," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[15] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *PAMI*, vol. 35, no. 8, pp. 1915–1929, 2013.

[16] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 579–594.

[17] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning." *CoRR*, vol. abs/1410.0759, 2014.

[18] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[19] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017, pp. 45–54.

[20] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, "mRNA: Enabling Efficient Mapping Space Exploration on a Reconfigurable Neural Accelerator," in *Proceedings of 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019.

[21] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[22] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, T. Krishna, and V. Sarkar, "MARVEL: A Decoupled Model-driven Approach for Efficiently Mapping Convolutions on Spatial DNN Accelerators," *CoRR*, vol. abs/2002.07752, 2020. [Online]. Available: https://arxiv.org/abs/2002.07752

[23] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, "Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning," 2018.

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[25] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[27] L. Shapiro and G. Stockman, *Computer Vision*. Upper Saddle River, NJ: Prentice-Hall, 2001.

[28] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: VersalTM architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.

[29] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 318–328. [Online]. Available: https://doi.org/10.1145/53990.54022

[30] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically Scheduling Halide Image Processing Pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, Jul. 2016. [Online]. Available: https://doi.org/10.1145/2897824.2925952

[31] 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

[32] S. Vocke, H. Corporaal, R. Jordans, R. Corvino, and R. Nas, "Extending Halide to Improve Software Development for Imaging DSPs," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3106343

[33] K. A. Stock, "Vectorization and Register Reuse in High Performance Computing," Ph.D. dissertation, The Ohio State University, 2014.

[34] L. Wang, Z. ChunYan, Y. Huang, and J. Xu, "Partial Elements Reuse of Vector Register in SIMD Mathematical Functions," *International Journal of Advancements in Computing Technology*, vol. 4, pp. 327–335, 01 2012.

[35] X. Jinchen, G. Shaozhong, and W. Lei, "Optimization Technology in SIMD Mathematical Functions Based on Vector Register Reuse," in *2012 IEEE 14th International Conference on High Performance Com-*

puting and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 1102–1107.

[36] F. Franchetti, Y. Voronenko, P. A. Milder, S. Chellappa, M. R. Telgarsky, Hao Shen, P. D'Alberto, F. de Mesmay, J. C. Hoe, J. M. F. Moura, and M. Püschel, "Domain-specific library generation for parallel software and hardware platforms," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–5.

[37] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *PARALLEL COMPUTING*, vol. 27, 2001.

[38] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, Feb 2005.