

Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators

Xuan Yang
xuany@stanford.edu
Stanford University

Jeff Setter
setter@stanford.edu
Stanford University

Steven Bell
sebell@stanford.edu
Stanford University

Priyanka Raina
praina@stanford.edu
Stanford University

Mingyu Gao
gaomy@tsinghua.edu.cn
Tsinghua University

Jing Pu
jingpu@alumni.stanford.edu
Stanford University

Kaidi Cao
kaidicao@stanford.edu
Stanford University

Christos Kozyrakis
kozyraki@stanford.edu
Stanford University, Google

Qiaoyi Liu
joeyliu@stanford.edu
Stanford University

Ankita Nayak
ankitan@stanford.edu
Stanford University

Heonjae Ha
hunjaeha@stanford.edu
Stanford University

Mark Horowitz
horowitz@ee.stanford.edu
Stanford University

Abstract

We show that **DNN accelerator micro-architectures and their program mappings represent specific choices of loop order and hardware parallelism for computing the seven nested loops of DNNs**, which enables us to create a formal taxonomy of all existing dense DNN accelerators. Surprisingly, the loop transformations needed to create these hardware variants can be precisely and concisely represented by Halide's scheduling language. By modifying the Halide compiler to generate hardware, we create a system that can fairly compare these prior accelerators. **As long as proper loop blocking schemes are used, and the hardware can support mapping replicated loops, many different hardware dataflows yield similar energy efficiency with good performance.** This is because the loop blocking can ensure that most data references stay on-chip with good locality and the processing units have high resource utilization. How resources are allocated, especially in the memory system, has a large impact on energy and performance. By optimizing hardware resource allocation while keeping throughput constant, we achieve up to 4.2× energy improvement for Convolutional

Neural Networks (CNNs), 1.6× and 1.8× improvement for Long Short-Term Memories (LSTMs) and multi-layer perceptrons (MLPs), respectively.

• **Computer systems organization** → **Neural networks; Data flow architectures; High-level language architectures**; • **Software and its engineering** → *Compilers*.

neural networks; dataflow; domain specific language

ACM Reference Format:

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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, March 16–20, 2020, Lausanne, Switzerland. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3373376.3378514>

1 Introduction

Deep neural networks (DNNs) have recently displaced classical image processing and machine learning methods due to their state-of-the-art performance on many tasks, particularly in recognition, localization, and detection [18]. As the number of applications for DNNs has grown, so have proposals for DNN accelerators. NeuFlow created a 2D systolic array for convolutional neural networks (CNNs) [13]. The DianNao family was built around customized inner-product units [6, 10]. Eyeriss highlighted the importance of on-chip dataflow on energy efficiency and proposed a row-stationary heuristic [8, 9]. And Google's TPU used a simple yet efficient systolic dataflow on a large 2D processing array [20]. These are just a few of the recent publications on DNN acceleration.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7102-5/20/03...\$15.00

<https://doi.org/10.1145/3373376.3378514>

All these proposals stated the advantages of their approaches over conventional general-purpose baseline platforms, yet the architectures and dataflows differ significantly across these approaches.

To help us understand and further improve these DNN accelerators, we realized that, since all the hardware designs perform the same computation, i.e., a seven-level loop nest for convolution as in Algorithm 1, the space of all accelerators can be formally specified by how they transform (block, reorder, and parallelize) the loop nest. We use this insight to create a formal taxonomy of DNN accelerators that expresses design choices as different loop transformations. For example, to improve data reuse and energy efficiency, the loops can be blocked and reordered to better schedule the computation, such that most data references are captured by the smallest and the most efficient memory buffer. The on-chip dataflow choices can also be represented as parallelizing different loops on multiple hardware processing units, known as spatial loop unrolling.

Prior work, like Timeloop [30], also adopted similar loop-based approaches to represent and analyze the design space for DNN accelerators systematically. We extend this work by showing that the loop transformations we use to specify the micro-architecture and dataflow choices of DNN accelerators are almost a subset of the loop transformation and memory allocation primitives provided by Halide’s scheduling language [34]. Halide is a domain-specific language for image processing applications. It provides all the required facilities to perform these loop transformations to convert a single application into efficient implementations on CPU, GPU, and more recently specialized hardware [32].

To create any possible DNN dataflow and storage hierarchy, we extend Halide, enabling it to create hardware accelerators for dense linear algebra in addition to image processing. The decoupling between algorithm and schedule within the Halide system makes it easy to explore different DNN mappings and hardware by simply changing the schedules in the Halide code. Using this system makes it easy to recreate the previously proposed designs and fairly compare their resulting performance and energy efficiency.

The extended Halide system allows us to create a systematic optimization framework, which can efficiently study the impact of dataflow and underlying hardware architecture design choices. Our results using Halide and the optimization framework show, with proper loop blocking, many dataflow choices can achieve similar and close-to-optimal energy efficiency. This large number of “optimal” solutions results from the fact that operands in most convolutional layers in DNNs have high reuse rates so, as long as properly blocked, most data references occur locally. Tailoring the loop blocking to each architecture is key; the blocking approach is usually more critical than the dataflow choice. For the layers that do not have enough data reuse to exploit, e.g., fully-connected layers that are not batched together, overall

energy is dominated by off-chip main memory accesses, thus the on-chip dataflow still does not have a large impact. From a performance perspective, it is important that the hardware supports unrolling multiple loops onto one spatial dimension of the underlying hardware, named as replication, to achieve good resource utilization.

In fact, energy efficiency is more tightly tied to the design of the hierarchical memory system and how each level in this hierarchy is sized. Since every multiply-add (MAC) involves fetching many operands from a memory (usually a register file, RF) and the cost of each RF fetch is proportional to the RF size, it is most efficient to adopt a relatively small RF. This small first-level memory creates the need for a memory hierarchy, since the size ratio between the adjacent memory levels needs to be in a certain range to balance the total energy cost of accessing data at each level in the memory hierarchy. Using these insights, we created an efficient optimizer for these types of Halide programs to jointly optimize memory system with schedules, which achieves up to 4.2 \times , 1.6 \times , and 1.8 \times energy improvement over the original Eyeriss accelerator for various CNNs, LSTMs, and MLPs respectively.

This paper makes the following contributions:

- Introduces a systematic approach to precisely and concisely describe the design space of DNN accelerators as schedules of loop transformations.
- Shows that both the micro-architectures and dataflow mappings for existing DNN accelerators can be expressed as schedules of a Halide program, and extends the Halide schedule language and the Halide compiler to produce different hardware designs in the space of dense DNN accelerators.
- Creates a tool to optimize the memory hierarchy, which is more important than the choice of dataflow, achieving a 1.8 \times to 4.2 \times energy improvement for CNNs, LSTMs, and MLPs.

The next section briefly reviews DNN accelerators. Then Section 3 describes how these accelerators can be characterized by their loop nest structures. Section 4 introduces Halide, explains how its scheduling language expresses the transformations we need, and shows how we use it to generate different accelerator implementations. To help us rapidly evaluate these designs, Section 5 discusses an analytical model, and how we validated this model. We then use this model to evaluate the energy and performance of different designs in Section 6. Finally Section 7 concludes the paper.

2 Diversity of DNN Accelerators

While independent research efforts often converge to a few common approaches, this does not seem to be the case for DNN acceleration. The NeuFlow architecture was a 2D systolic array for CNNs, where each processing element (PE) communicated only with its neighbors, and data was streamed to and from DRAM [13]. Its successor TeraDeep used a fixed

loop-blocking strategy for CONV layers [16]. The DianNao family was built around customized inner-product units. The first generation used a single level of small buffers [6], while in a later iteration the original unit was surrounded by a large eDRAM that stored the complete data sets [10]. Another version specially built for embedded systems further extended to a 2D PE mesh that supported optimized inter-PE data propagation [12]. More recently, Eyeriss highlighted the importance of such on-chip dataflow for energy efficiency, and proposed using row-stationary dataflow as a heuristic solution [8, 9]. Neurocube and Tetris combined the spatial PE arrays with 3D-stacked DRAM to reduce the main memory access cost [14, 21]. FlexFlow leveraged the complementary effects of different dataflow styles and mixed them on the same PE array to improve resource utilization [27]. To improve its efficiency, Eyeriss V2 [7] designed a flexible interconnect to support different replication schemes. In addition to CNNs, Google’s TPU used a simple systolic dataflow on a large 2D array of PEs, which could also be used for MLPs and LSTMs [20]. Tangram investigates the dataflow optimizations for coarse-grain parallelism to eliminate excessive data duplication in the on-chip buffers for tiled NN accelerators [15]. Song in [39] proposed to reorganize dataflows to improve the data reuse for non-standard convolutional layers in Generative Adversarial Networks (GANs). Other prior work has also implemented architectures that are flexible to support multiple different dataflow types [25, 43]. Beyond dense matrix computations, other designs have explored DNN sparsity and proposed specialized dataflow schedules [4, 17, 31, 46, 49]. Another group of designs have transformed DNN processing into the frequency domain to reduce computations [22, 48].

On FPGA platforms, Zhang et al. [47] adopted the Roofline model to explore loop blocking, but considered only two levels of memory and only minimized off-chip bandwidth rather than total memory energy. Alwani et al. [5] fused the computation of different NN layers, and Li et al. [26] mapped the entire CNN onto an FPGA in a pipelined manner, both to reduce intermediate data writeback. Shen et al. [36, 37] optimized FPGA resource utilization by using a heterogeneous design. Sharma et al. [35] provided hand-optimized templates for generating dataflow architectures.

3 DNN Accelerator Design Space

All the DNN accelerators discussed in Section 2 have demonstrated improvements over general-purpose baselines, and explored how the parameters they experimented with would affect their performance and efficiency. Unfortunately, without an understanding of the global design space, each paper explores a different part of the space — perhaps coupling together independent parameters — so this approach leads to conflicting reports on the “optimal” parameters. To avoid this problem we first wanted to understand the space

Algorithm 1 CONV layer: simple seven nested loops.

```

for  $b = 0$  until  $B$  do
  for  $k = 0$  until  $K$  do
    for  $c = 0$  until  $C$  do
      for  $y = 0$  until  $Y$  do
        for  $x = 0$  until  $X$  do
          for  $f_y = 0$  until  $F_Y$  do
            for  $f_x = 0$  until  $F_X$  do
               $O[b][k][x][y] += I[b][c][x+f_x][y+f_y]$ 
                 $\times W[k][c][f_x][f_y]$ 

```

of all possible dense DNN accelerators. We thought this was possible since each accelerator computes the same result, so the differences must be in the resources available to compute the result, and the way the computation is scheduled to use these resources. To understand how this works, let’s review the computation which needs to be performed.

A DNN is a directed acyclic graph (DAG) of various types of layers. The CONV layer computation is summarized as:

$$O[b][k][x][y] = \sum_{c=0}^{C-1} \sum_{f_y=0}^{F_Y-1} \sum_{f_x=0}^{F_X-1} I[b][c][x+f_x][y+f_y] \times W[k][c][f_x][f_y]$$

and also shown in Algorithm 1 as seven levels of nested loops. The nested loops generate output feature maps (*fmaps*) O , which have K channels of $X \times Y$ images, by processing the input fmaps I of C channels. The fmap data are processed in *batches* (B) to increase parallelism and data reuse. W contains the weights as K 3D stencil filters with size $C \times F_X \times F_Y$. By summarizing this computation by this loop nest, we can express computation beyond the batched CONV layers, including non-batched operations, fully-connected (FC) layers, etc., by setting some loop bounds to 1. For example, the FC layer computes a matrix vector multiplication and can be described using the same nested loops with only C , K , and B loops, while all the other loops bounds are set to 1.

DNNs also include other layer types such as pooling, normalization, and element-wise. However, CONV and FC layers dominate the computation and memory communication, so we focus on them in this paper.

3.1 Design Space Overview

The design parameters that have been widely studied to optimize data reuse and performance of DNNs are *loop blocking* [45] and *dataflow* [8, 24]. However, when exploring such software scheduling decisions, prior designs often used different hardware *resource allocations*. We therefore consider a corresponding three-dimensional design space for DNN accelerators, as shown in Figure 1.

Dataflow: DNN accelerators often exploit parallelism to improve performance by using multiple processing elements

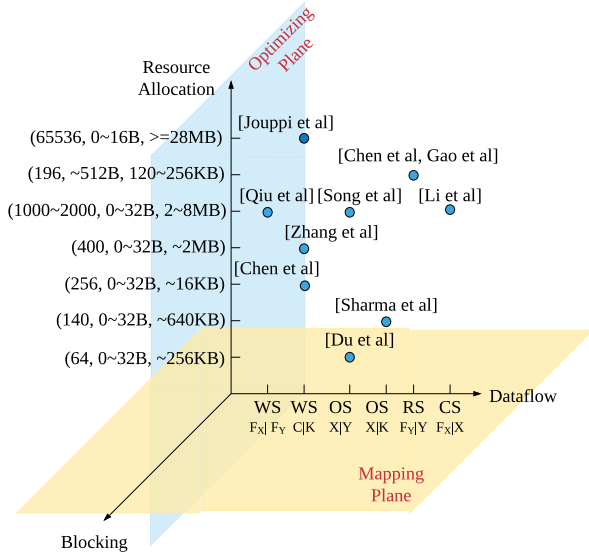


Figure 1. 3D design space for DNN accelerators. The positions of labels or vectors on each axis only represent different choices without specific information about ordering or distance.

(PEs) simultaneously. Essentially it executes one or more loops in Algorithm 1 in parallel through *spatial loop unrolling*. The data access and communication patterns across the multiple PEs are determined by the *dataflow* scheme [8]. Typically, the dataflow is carefully orchestrated so that data accesses to more expensive memories, including the storage in other PEs and the large shared buffers, can be minimized. We will provide a comprehensive dataflow taxonomy in Section 3.2. For now we use stationary characteristics from [8] as the dataflow labels, such as weight stationary (WS), row stationary (RS), etc., to represent the choices in the dataflow dimension in Figure 1. Here we assume dataflow choice is purely a subset of mapping space, which is orthogonal to the underlying architecture. Later we will provide more explanations about this assumption.

Resource Allocation: Hardware resource allocations, such as the dimensions of the PE array and the size of each level in the memory hierarchy, are also essential to the performance and efficiency of the accelerator. They determine the computation throughput, the location of the data, and the energy cost and latency for each memory access. For example, since the energy cost and latency of each data access grow with memory size, an efficient design needs to carefully size each memory level to optimally balance buffering sufficient data for high locality while being as small as possible to minimize fetch energy. In summary, the resource allocation axis needs to cover various hardware choices. Here in Figure 1 we mainly consider the number of MAC units N , and the memory size S_i at each level i , represented as a vector (N, S_1, S_2, \dots) .

Loop Blocking: Assuming a multi-level memory hierarchy (e.g., register files, on-chip SRAM, and off-chip DRAM), we want to schedule the computation to maximize the data reuse in the near, smaller memories to lower overall energy cost. Since all the data fetched — inputs, outputs, and weights — can be potentially reused, an optimal schedule must choose the best data reuse opportunities. The techniques of loop blocking and reordering, which together we refer to as loop blocking [45], transforms the seven nested loops in Algorithm 1 into a much larger number of loops, and generates different data access/reuse patterns for each memory level.

Using these three factors enables us to create a design space that is comprehensive and systematic, enabling us to project existing DNN accelerators as shown in Figure 1. We do not explicitly show the loop blocking schemes for each architecture in this figure, since most prior work did not report their loop blocking strategy, or simply exhaustively searched for an ad-hoc optimized scheme. This figure clearly shows the wide design space current accelerators occupy.

3.2 A Formal Dataflow Taxonomy

Since blocking is by definition operations on loop nests, we next show how the dataflow of an accelerator can also be represented by loop operations. Noticing the connection between dataflow and spatial loop unrolling, we represent the dataflow of an accelerator through the mapping of particular loops to the parallel computation structures, similar to the terminology for the spatial partitioning in Timeloop [30]. In other words, the data communication pattern is determined by *which loops are spatially unrolled* in hardware, and which are not. For example, if the X and Y loops are unrolled onto the 2D array, then each PE produces a single output pixel. This output stationary pattern implies that input pixels will be reused across neighbor PEs as they contribute to multiple output pixels in a convolution, and the filter weights shared by all output pixels must be transferred to all PEs. If we instead unroll the F_X and F_Y loops, we obtain a weight stationary pattern, where the weights stay and are reused within the same PEs, but the inputs and outputs are spatially broadcast or accumulated.

To concisely represent dataflows using spatial loop unrolling schemes on 2D PE arrays, we use the syntax $U | V$, where U and V denote the loops unrolled across the vertical and horizontal dimensions, respectively. Given L -level (excluding those with loop bounds as 1) nested loops in the algorithm and d spatial dimensions in the accelerator, there are $\binom{L}{d}$ possible dataflow choices. For a 2D array and the unblocked algorithm, the number of dataflow types is $\binom{7}{2} = 21$ for a CONV layer, and $\binom{3}{2} = 3$ for a fully-connected layer.

However, the above considered dataflows, which unroll a single loop at each spatial dimension, can potentially result

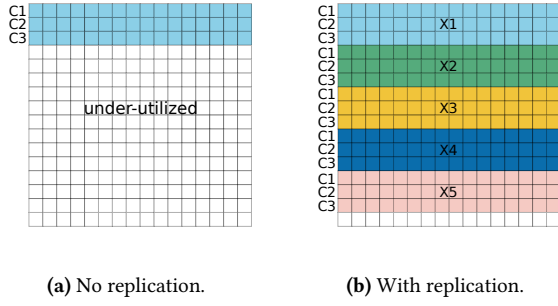


Figure 2. Computation resource utilization can be improved by replication.

Table 1. Common dataflows from [8] expressed using spatially unrolled loops.

Dataflow	Representation
Output stationary	$X Y$
Weight stationary	$F_X F_Y$
Row stationary	$F_Y Y$
Weight stationary	$C K$

in under-utilizing computation resources. For instance, as illustrated in Figure 2a, when unrolling a loop C with size of 3 on the vertical dimension of a 16×16 PE array, only 3 of the 16 rows of PEs are utilized, leaving the remaining PEs idle. To overcome this issue, in addition to unrolling loop C , another loop such as X is also unrolled by a factor of 5, as shown in Figure 2b, improving the utilization ratio from $3/16$ to $15/16$. Therefore, it is of great importance to support unrolling multiple loops onto one spatial dimension. This improvement is called *replication* [7, 8], which processes multiple small loops in parallel to increase resource utilization. Our loop-based taxonomy can also nicely and consistently express it using $U | VW$ or $UW | V$, depending on the replicated dimension. When replication is supported, the number of dataflow choices for a layer further increases.

Note that with replication (i.e. mapping multiple loops onto the same physical dimension), the data communication pattern is no longer uniform: intra-loop data can be communicated among nearest-neighbor PEs, while inter-loop data have to be sent multiple hops away with higher communication cost. Syntactically, we represent this by ordering the loops mapped to the same dimension, where the PEs generated by unrolled loops to the left have shorter communication distances than the loops on the right. Figure 3 shows an example of unrolling both C and K loops onto a 1D array. The eight PEs have been divided into two groups, each working on a output channel K_i . Within each group, different PEs process different input channels C_i . The outputs are only communicated among the nearest PEs, while

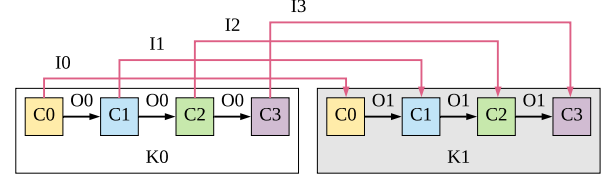


Figure 3. Loops C and K are unrolled onto a 1D array with dataflow CK . Outputs are communicated between adjacent PEs, while inputs are communicated across groups.

the inputs have to transfer from one group to the other with a cost four times that of nearest neighbor communication. As was mentioned, Eyeriss V2 [7], enables flexible replication by providing flexible interconnections between the processing elements. Fortunately, the interconnects provided by some hardware targets, including FPGAs and CGRAs, naturally provide this flexibility.

Advantages: The loop-based approach builds upon the ideas of stationary characteristics [8], and provides a more precise definition of each dataflow while expanding the range of flows that can be described. Table 1 shows several common dataflows in prior work expressed using our taxonomy. As an example, $C | K$ is a widely adopted dataflow (Figure 1, and [5, 36, 37, 40]) due to its flexibility to also map matrix multiplications in MLPs and LSTMs. Even though $C | K$ also keeps the weight stationary in PEs, its data reuse pattern is quite different from the weight-stationary dataflow $F_X | F_Y$ introduced by [8]. Furthermore, more complicated dataflows, such as a hybrid weight and output stationary pattern, are easy to represent in our taxonomy, e.g., $C | KX$, demonstrating its completeness.

Using the loop-based dataflow taxonomy, dataflows and loop blocking schemes can now both be expressed as transformations of the seven nested loops in Algorithm 1. There are many existing approaches to find loop transformations that optimize some cost functions, in order to either realize optimal software implementations on CPUs or GPUs, or generate optimal hardware designs using FPGAs or ASICs. These include general approaches like Polyhedral analysis [28, 51] and studies specific to DNNs [23, 45].

In the next section, we show that these loop transformations can be expressed in Halide’s scheduling language, and we use this language to specify the schedules and hardware resources (both memory and compute units) of any specific DNN accelerator design. Although Halide and polyhedral models both support affine loop transformations, we choose Halide over creating a new system with a polyhedral model, in order to leverage Halide’s compact scheduling primitives and mature compilation framework. This allows us to separate the hard optimization problem – finding the right schedule, from the mechanical transformation and manual effort to implement it.

4 Halide Accelerator Design

Halide [34] is a domain-specific language (DSL), originally designed for image processing but generally applicable to dense loop-structured computation including linear algebra and DNNs. The key idea in Halide is to split the computation to be performed (the *algorithm*) from the order in which it is done (the *schedule*). To allow the user to express the implementation order, Halide provides a compact and elegant language to represent loop transformations. These transformations along with commands that create intermediate storage are sufficient to completely specify a DNN accelerator, and the blocking of the algorithm which most efficiently utilizes it.

Halide algorithm: Halide represents the computation algorithm in pure functional form. The following example shows the Halide algorithm for a CONV layer with $3 \times 5 \times 5$ filter size.

```
1 // To perform a 5 x 5 convolution with 3 channels
2 // RDom(xMin, xExt, yMin, yExt, kMin, kExt)
3 RDom r(-2, 5, -2, 5, 0, 3);
4 output(x, y, k) += input(x + r.x, y + r.y, r.z)
5                     * w(r.x + 2, r.y + 2, r.z, k);
```

The RDom keyword defines a multi-dimensional reduction domain, over which an iterative computation such as a summation is performed. A RDom is defined by the minimum position and extent in each of its dimensions. In the CONV layer example, the RDom covers the width and height of the filters and the number of input fmaps, over which the accumulation will iterate.

While the algorithm defines the functionality of the computation, it does not specify the ordering of parallel operations or data accesses. These are further controlled using Halide schedules, which consist of *scheduling primitives* applied to various stages of the algorithm. These primitives can express the required loop blocking, resource allocation, and, with minor extensions, dataflow choices as well.

4.1 Halide Schedules

With only small extensions to the current Halide scheduling language, we can explore all hardware architectures and software scheduling choices in the design space introduced in Section 3. Table 2 summarizes how the scheduling primitives control each of the three design space dimensions. Listing 1 shows an example Halide schedule for the above Halide algorithm of a CONV layer.

The algorithm and schedule provide a user-facing language to construct hardware. Figure 4 pictorially shows this example schedule in details. From left to right, we iteratively apply three sets of scheduling primitives to achieve the final accelerator structure. Listing 2 presents the intermediate representation (IR) generated by Halide, corresponding to

Table 2. Halide scheduling primitives that control each dimension of the 3D design space.

Dimensions	Scheduling primitives
Loop blocking	split, reorder
Resource allocation	in, compute_at
Dataflow	unroll, systolic
Overall scope	accelerate

```
1 Var xo, yo, xi, yi;
2 output.update().split(x, xo, xi, 8)
3                   .split(y, yo, yi, 8)
4                   .reorder(xi, yi, xo, yo);
5 input.in().compute_at(output, xo);
6 w.in().compute_at(output, xo);
7 output.accelerate({input, w});
8 output.update().unroll(xi, 4);
9 output.update().systolic({xi});
```

Listing 1. An example Halide schedule for the CONV layer algorithm.

```
1 // To generate output of size 16 x 16 x 64
2 for (k, 0, 64)
3   for (yo, 0, 2)
4     for (xo, 0, 2)
5       // Allocate local buffer for input.
6       alloc ibuf[8 + 5 - 2, 8 + 5 - 2, 3]
7       // Copy input to buffer.
8       ibuf[...] = input[...]
9       // Allocate local buffer for w.
10      alloc wbuf[5, 5, 3, 1]
11      // Copy w to buffer.
12      wbuf[...] = w[...]
13      for (yi, 0, 8)
14        for (xi, 0, 8)
15          for (r.z, 0, 3)
16            for (r.y, -2, 5)
17              for (r.x, -2, 5)
18                output(xi, yi, 0) +=
19                  ibuf(xi + r.x, yi + r.y, r.z)
20                  * wbuf(r.x + 2, r.y + 2, r.z, 0)
```

Listing 2. Intermediate representation generated by Halide after using in and compute_at combined with split and reorder, corresponding to the third phase after step (2) in Figure 4.

the third phase in Figure 4. The rest of this section describes the three transformations in that figure.

Loop blocking: The existing Halide scheduling primitives are primarily designed for loop transformations on general-purpose processors, but the syntax and semantics also support loop blocking on accelerators thanks to the same underlying principles. Lines 2–4 of Listing 1 use split to break the x and y loops into two levels, where the inner loops

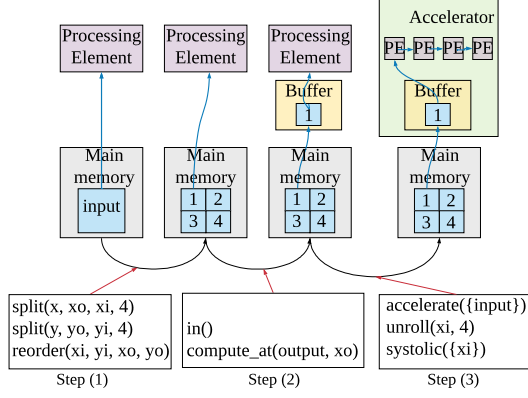
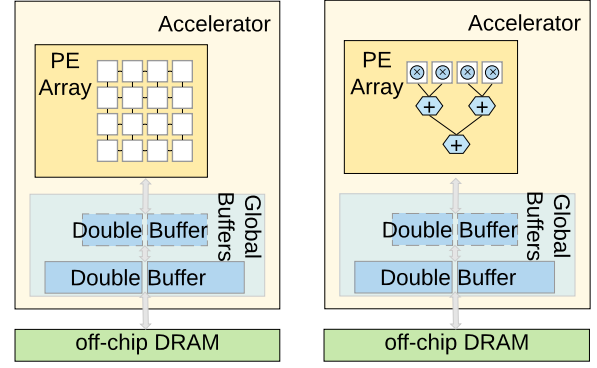


Figure 4. The initial design fetched the data as one large block from memory. After the split and reorder, the data is broken into 4 smaller tiles. Next a local buffer for one tile is allocated, and finally a 4 PE systolic array is implemented to process the data.

have 8 iterations. `split` can also be applied repeatedly to create more levels. `reorder` interchanges the loops, setting the order of computation and data access. These two primitives can realize different *loop blocking* schemes, splitting the data into multiple smaller subtiles (4 tiles of 8×8 in this example) that are processed in a certain order (x then y). Step (1) in Figure 4 visually shows the four tiles that are created due to the new loops (lines 3–4 and 13–14 in Listing 2).

Resource allocation of memory hierarchy: After splitting the data, we need to allocate SRAM buffer resources so that each data subtile can be cached on-chip while being processed to reduce their access cost. The existing primitives `in` and `compute_at` (lines 5–6 of Listing 1) together introduce additional memory levels, and specify at which loop iteration to fetch which subtiles into the buffers (Figure 4). The compiler combines this information with loop sizes, and instantiates the correct number of memory levels with appropriate size and data layout for each buffer. As shown by lines 5–12 in Listing 2, by calling `in` and `compute_at` together for both input and w , two local buffers with required sizes are allocated within loop x_0 for input and weight data respectively. This allows us to explore different *resource allocation* choices. For each memory level, we use a *double buffer* design (Figure 5), which enables overlapping computation and data fetch: during the computation of the current tile, the next tile is loaded into the alternate buffer.

Dataflow and PE array micro-architecture: While the previous two dimensions can be covered using existing Halide primitives, expressing *dataflow* requires extensions to support the complex on-chip data propagation patterns uniquely appearing in accelerators. First, like Pu et al. [32], we overload the existing `unroll` primitive to specify spatial loop unrolling onto the PE array. As discussed in Section 3.2,



(a) Systolic array.

(b) Reduction tree.

Figure 5. DNN accelerator architecture consisting of a PE array and a memory hierarchy.

given a 1D dataflow U or 2D dataflow $U \mid V$, we spatially unroll the loops U and V on each physical dimension, respectively. For example, in Figure 4 step (3), the loop x_i is unrolled to execute in parallel on 4 systolic PEs.

Second, we support various types of PE array micro-architectures. We introduce a new primitive, `systolic`, which realizes a systolic PE array as shown in Figure 5a, and allows direct inter-PE data communication without always fetching data from higher-level data buffers [8, 20]. Combined with different `unroll` primitives, we can realize different dataflows on the PE array. Figure 6a maps the $F_Y \mid Y$ dataflow used in Eyeriss [8], by unrolling the F_Y and Y loops. It transfers multiple rows of filter weights horizontally, and accumulates multiple columns of output fmaps vertically. Alternatively, Figure 6b performs matrix multiplications using dataflow $C \mid K$, which is used by a large group of designs including Google’s TPU [20].

Without applying `systolic`, the PEs are by default organized into reduction tree structures [6], as shown in Figure 5b. Figure 6c provides one example dataflow on a 1D reduction tree, which unrolls the loop C to multiply input pixels from different input fmaps with their corresponding weights, and accumulates the products into a single output pixel in an output fmap.

Using the two micro-architectures in Figure 5 as building blocks, we can generate a variety of accelerator designs by composition. They can be described by applying `unroll` at different loop levels, and calling `systolic` at the corresponding unrolled loops. For instance, we can have a reduction tree of PEs acting as a node in a systolic array, or multi-level reduction trees, effectively supporting a wide range of designs including ARM ML processor [1] and NVDLA [2]. We could additionally introduce new primitives similar to `systolic`, to support other PE array micro-architectures if desired.

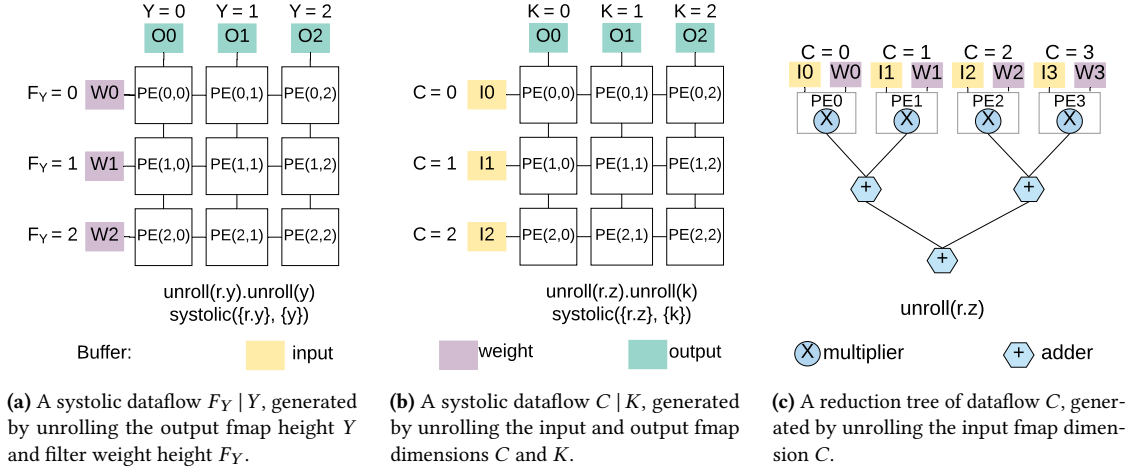


Figure 6. Different PE array micro-architectures generated from Halide scheduling primitives.

Accelerator scope: Finally, we introduce an additional primitive, `accelerate`, which defines the scope of the hardware accelerator and the interface to the rest of the system, in a similar manner to Pu et al. [32].

4.2 Implementation

Given this concise and concrete expression for DNN accelerators using Halide schedules, we extended the Halide compiler to generate hardware from these descriptions. As a result, different DNN accelerator designs and mapping schemes can be realized by simply changing the Halide schedule associated with the same Halide algorithm. We can also use it to easily recreate previously proposed designs for a fair comparison (see Section 6).

Our implementation of the Halide toolchain is built on top of Pu’s work [32], which was designed for generating hardware for image processing pipelines. To create our system, we needed to extend this work in three important ways. The first was to add support for systolic arrays, which was challenging due to the diversity of the PE connectivity and the complexity of creating a state machine for the control logic inside each PE. Next, hardware virtualization was added to map different layers in a DNN (represented as different Halide functions) onto the same hardware module. Otherwise, naïvely instantiating spatially separated modules for each layer in large DNNs would consume unrealistically large silicon area. Finally, we extended Pu’s work to generate ASIC as well as FPGA implementations.

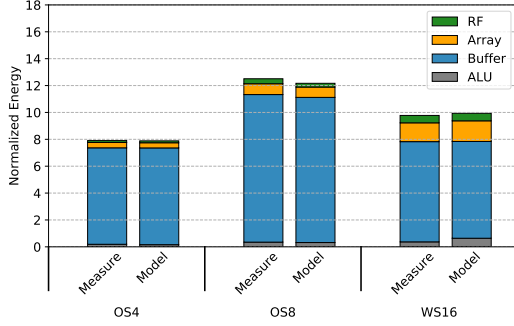
5 Methodology

Our Halide-based accelerator design flow in Section 4 supports both FPGA and ASIC backends. The FPGA results

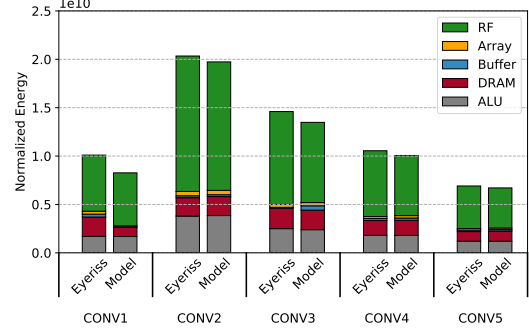
enabled us to validate that the system was functional and produced efficient designs: it achieved similar GOPs and DSP utilization when compared against manually optimized designs [33, 40].

ASIC Hardware synthesis toolchain: Our extended Halide compiler generates C++ code specialized for Catapult High-Level Synthesis, which is then compiled to RTL designs in Verilog. We synthesize the RTL designs in a 28 nm technology using Synopsys Design Compiler. Standard cells and memory models from commercial vendors are used for power, performance, and area analysis. We use 16-bit arithmetic for inference tasks throughout this paper. All of our ASIC designs achieve 400 MHz frequency with no timing violations. For power analysis, the appropriate switching activities are set on all the primary ports and propagated through the design using the design tools.

Analysis framework: To allow for rapid design exploration, we also developed an analytical model to estimate the performance and energy efficiency of the ASIC DNN accelerators. We use CACTI 6.5 [29] to model SRAM arrays and tune its parameters to match our 28 nm commercial memory library. For small arrays and register files (RFs), we use the Cadence XtensaProcessor Generator [3] to extract energy numbers based on our standard cell library. Table 3 shows the energy cost of accessing memories with different sizes. Note that our energy ratios between memories and MAC are larger than those reported in Eyeriss [8]. There are several reasons: we use a 28 nm technology instead of 65 nm; our memory is highly banked with higher energy cost; and our MAC units consume lower energy as their activity factors are relatively low with data stationary patterns. Nevertheless, our analytical framework works with different technology processes, and it is easy to supply new cost models to study



(a) Energy breakdown comparison between actual synthesized designs and the analytical model.



(b) Energy breakdown comparison between reported Eyeriss model and our model.

Figure 7. Validation of the analytical model against post-synthesis results and previous published model [8].

Table 3. Energy per 16-bit access with various register file (RF) and SRAM sizes, and for a MAC operation, one hop communication cost and a DRAM access.

RF Size	Energy (pJ)	SRAM Size	Energy (pJ)
16 B	0.03	32 KB	6
32 B	0.06	64 KB	9
64 B	0.12	128 KB	13.5
128 B	0.24	256 KB	20.25
256 B	0.48	512 KB	30.375
512 B	0.96		
MAC	0.075	DRAM	200
Hop	0.035		

more advanced technologies. Also, many of our observations in Section 6 are technology-independent.

To compute the overall memory energy in an L -level hierarchy, we adopt a model similar to [8] and [30]:

$$E = \sum_{i=1}^L \#acc_i \times e_i \quad \text{where} \quad \#acc_i = \prod_{j=i}^L RT_j$$

Here e_i is the energy of accessing the i th level once. The total numbers of accesses are affected by data reuses RT_i at different memory levels. It is defined as the number of times the data are accessed by its immediate lower-cost (child) level during its lifetime in this level. To support direct inter-PE communication in systolic arrays, we treat neighbor PEs as an additional level in the hierarchy. We distinguish the cost for different communication distances (Figure 3) which is an improvement over [8].

Since all designs today are power constrained, finding the optimal accelerator design now becomes an optimization problem of minimizing E over the 3D design space, similar to Yang’s work [44]. e_i is determined by the resource

Table 4. ASIC designs for model validation.

Name	Dataflow	PE Array	RF	SRAM
OS4	X	1D, 4	32 B	32 KB
OS8	X	1D, 8	64 B	64 KB
WS16	$C K$	2D, 4×4	64 B	32 KB

allocation (Table 3), and RT_i can be directly calculated from the dataflow and loop blocking schemes. In this work, we simply perform a conservatively pruned search over the full design space guided by domain-specific knowledge. This analytical model is available at <https://github.com/xuanyoya/Interstellar-CNN-scheduler>.

Framework validation: We have thoroughly validated the accuracy of our model by comparing its results to complete designs generated by our synthesis toolchain. Table 4 shows three example designs we have generated in ASIC platforms, and Figure 7 shows the energy comparison between our analytic model and post-synthesis results. The resulting errors are less than 2%. Furthermore, our framework is also able to reproduce the results from [8] with small differences.

6 Results

Using our dataflow taxonomy and the ability to rapidly generate and evaluate large numbers of accelerator designs with Halide, we first explore different dataflow and loop blocking choices, and then consider hardware resource optimizations. At the end we leverage the characteristics of these results to introduce an efficient optimizer for DNN accelerators.

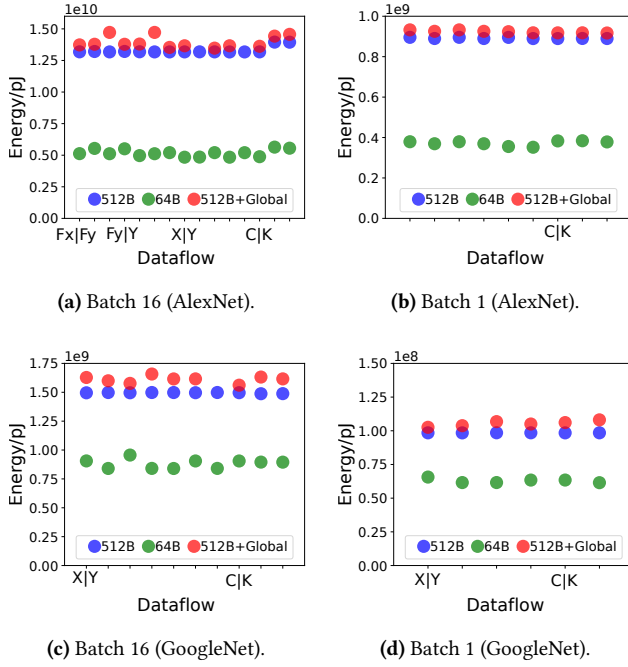


Figure 8. Design space of dataflow for AlexNet CONV3 and GoogLeNet 4C3R layers. The Y-axis is the energy consumed to execute the entire batch. Different dataflows are shown horizontally, with only the most common choices labeled for clarity. All dataflows use replication and the optimal loop blocking schemes. Different colors represent different hardware resource allocations.

6.1 Impact of Dataflow and Loop Blocking

Figure 8 compares the energy efficiency of different dataflow choices. We use the CONV3 layer in AlexNet and 1×1 reduction layer 4C3R in GoogLeNet Inception (4c) module as examples. The other layers have also been investigated, and share a similar trend. More DNNs will be studied in Section 6.3. We use the optimal loop blocking scheme for each dataflow. We can see that when optimized loop blocking schemes are applied, many different dataflows achieve similar and close-to-optimal energy efficiency on the same hardware configuration. We have evaluated three different hardware configurations: the blue one is the same as Eyeriss [8], with 512 B register file (RF), 128 KB SRAM buffer, and 16×16 PE array per PE; the red one uses a different array bus design, which disables inter-PE communication and broadcasts all data from the global buffer; the green one uses a smaller 64 B RF to lower its access energy (see Section 6.2). Figure 8b and 8d also show the cases with batch size 1, which is most commonly used in mobile systems; the conclusion is consistent across different batch sizes. The small influence of dataflow remains true over a wide set of experiments including different layer types, different PE array structures and

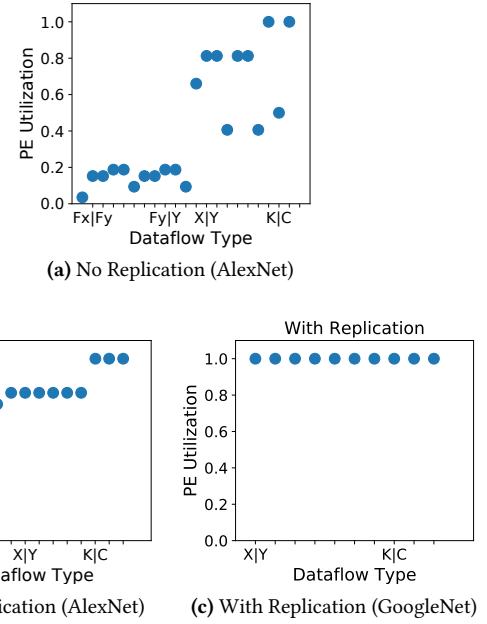


Figure 9. PE array utilization for the energy-optimal dataflow choices on AlexNet CONV3 layer with and without replication, and GoogLeNet 4C3R layer with replication.

sizes, different memory configurations, and different energy cost models.

On the other hand, in Figure 9 we observe that the PE array utilization (active PE ratio for each run), and therefore the computation throughput, is more sensitive to different dataflow choices than the energy efficiency for some convolution layers. Without replication (Figure 9a), the overall utilization can vary significantly and stay low for many dataflow choices. However, using proper replication substantially improves the utilization and eliminates most of the differences among all dataflows (Figure 9b and 9c). These results also imply that accelerators that support a diversity of replication schemes, such as CGRAs and Eyeriss V2 [7], will generally achieve higher overall utilization compared to the ones with fixed interconnects. Here, the impact of the interconnect bandwidth to evaluate the overall performance is not included in the analysis, as prior works [7, 24] have already studied such impact. Figure 9b also demonstrates, for the CONV3 layer in AlexNet, that the C|K dataflow achieves 20% higher utilization than the others such as Fy|Y. This is because the channel dimensions C and K are typically the largest in most CONV and FC layers, so it is easier to unroll them onto a fixed-sized PE array with small fragmentation. For this reason, we will use the C|K dataflow in the rest of this paper.

Not all the computational layers have as much data sharing to exploit. Weight sharing in FC layers only comes from

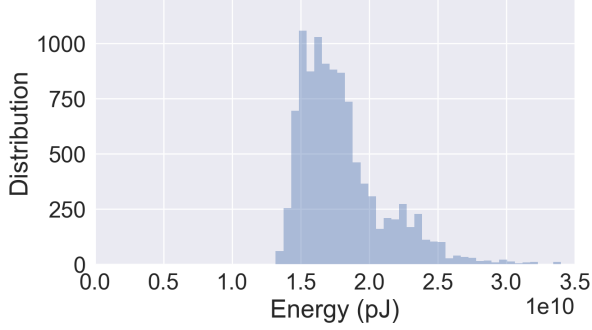


Figure 10. Design space of loop blocking for AlexNet CONV3 using dataflow $C|K$ with 512 B RF per PE.

batching, and some applications limit the batch size to be small, even one. Interestingly, even in these computations the dataflow does not have a large influence on performance or energy. For computations with limited reuse, the data must come from the off-chip DRAM, or the last level on-die storage, if it is large enough. The storage properties at this level will limit the device’s energy and performance, so for this class of application, the design of the computation units is less important.

Instead of dataflow choices, Figure 10 shows the design space of loop blocking for AlexNet CONV3, using a 512 B RF, corresponding to the blue configuration in Figure 8a. The energy variance of different blocking schemes is much more significant than that of dataflow, and only 30% of the schemes fall within $1.25\times$ of the minimum energy. This indicates that loop blocking has a large impact on energy efficiency.

Observation 1: *With the same hardware resources, many different dataflows are able to achieve similar and close-to-optimal energy efficiency, as long as proper loop blocking and replication are used.*

In hindsight, this result is not surprising. When the DNNs exhibit enormous data reuse opportunities, as long as high data reuse is achieved through proper loop blocking schemes, the resulting energy efficiency should be good. When the reuse is limited, the performance is limited by the bandwidth of the last level in the memory hierarchy instead of the PE array. These two situations are further illustrated in Figure 11, where the left bars with 512 B RF show the energy breakdown of the optimal dataflow for the blue configuration in Figure 8a. For CONV layers with high reuse, most energy is consumed in the RF level rather than the array buses or intermediate buffers. By optimally blocking the computation, nearly all accesses (98%) occur at the RF level, making it the dominant energy component. For FC layers with limited reuse, most of the DRAM energy is inevitable, since the data have to be fetched at least once from off-chip (compulsory misses). On the other hand, the on-chip communication is

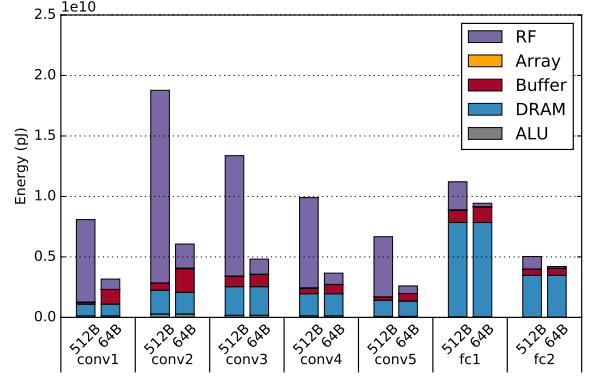


Figure 11. Energy breakdown comparison between 512 B and 64 B RF sizes with the same dataflow. Using a 64 B RF reduces the overall energy significantly.

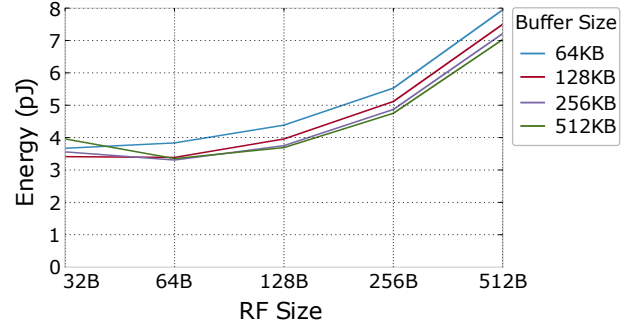


Figure 12. Memory hierarchy exploration with dataflow $C|K$. Different RF sizes per PE are shown horizontally. Lines with different colors correspond to different SRAM buffer sizes.

generally only a small portion of the total energy, and therefore blocking choice has a more substantial impact on the overall energy efficiency than dataflow choice.

6.2 Impact of Hardware Resource Allocation

Another interesting result from Figure 11 is that the total energy is always dominated by the RF level with a 512 B RF. This result indicates that this resource allocation may be suboptimal. Figure 12 shows the impact of memory resource allocation on energy efficiency. The energy is accumulated across all layers (including FC layers) in AlexNet, and contains both computation and memory access portions. It indicates that using a smaller RF size such as 32 or 64 B can improve the total energy efficiency by up to $2.6\times$. If we also increase the global SRAM buffer size, the energy efficiency can further improve. However, when SRAM buffer size grows beyond 256 KB, the benefit becomes negligible.

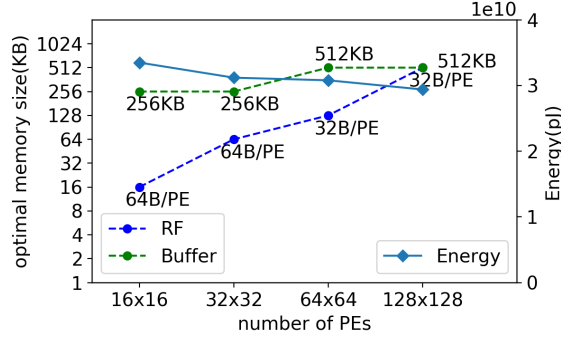


Figure 13. The optimal memory resource allocation and the corresponding total energy when varying PE array size.

Given the significant area cost, it is not always necessary to use large global buffers.

The right bars in Figure 11, which give the energy breakdown of using a 64 B RF, illustrate that the energy decreases dramatically for all the CONV layers due to the much lower energy cost per access of the smaller RF. At the same time, more accesses go to the inter-PE array level and the global buffer, since the smaller RF captures less data reuse inside each PE. But reducing the RF size has almost no impact on the DRAM energy, as the data are still efficiently reused in the global buffer. Overall, a smaller RF achieves significantly better energy efficiency, with a more balanced energy breakdown among different memory hierarchy levels.

Observation 2: *The total energy of an efficient system should not be dominated by any individual level in the memory hierarchy.*

Observation 2 also explains why some output-stationary and weight-stationary designs do not perform well, as discussed by [8]. Those designs cannot capture sufficient reuse at the RF level, and result in high energy consumption at the DRAM level, which dominates the overall energy.

However, there is an exception for Observation 2. When DRAM dominates the total energy but the number of DRAM accesses is already minimized (fetching the input once and writing back output once), the DNN is memory bound, and based on Amdahl’s law, little further optimization can be achieved for the memory hierarchy. This is the case particularly for a batch size of 1, and MLPs and LSTMs that contain many FC layers.

We also investigate whether changing the hierarchy itself can further improve the energy efficiency. Due to the dominant role of the RF level, we add another level of private register file. The largest energy efficiency improvement is obtained when sizing each memory level is based on the rule that the ratio of the on-chip storage sizes between the adjacent levels should be around 4 to 16. Specifically, the overall energy efficiency of AlexNet is improved by 25%, by

choosing 16 B and 256 B to be the two level register file sizes, and 256 KB to be the global buffer size.

Figure 13 depicts the optimal memory resource allocation and the corresponding total energy for AlexNet when varying PE array size. We use only one level of RF here. These correspond to the optimal points on the optimizing plane shown in Figure 1. With increasing numbers of PEs, the optimal memory size at each level grows sub-linearly. Ideally we would like to keep the same amount of data reuse with constant storage capacity for each PE, which would lead to linearly increased memory size. However, the access cost of each memory level grows with its size (Table 3), which slows down the optimal capacity scaling to sub-linear. Between the RF and the SRAM buffer, the data reuse in RF is more critical. So the RF level has a stronger trend to keep constant capacity per PE. But it is eventually bounded by the size of the next-level, i.e., the SRAM buffer.

Also we notice that the total energy reduces slightly with the increasing number of PEs. This indicates that larger arrays will have a significant effect on throughput and a small change in energy efficiency. The energy improvement is achieved by buffering more data on chip for reuse, and since most communication is nearest neighbor, the larger die does not increase communication costs significantly.

6.3 An Efficient Optimizer

With the large number of hardware and software choices for DNN accelerators, exhaustive search for the optimal designs is usually infeasible. Instead, using the observations above, we can speed up the optimization process by pruning the search space and evaluating only a small number of candidates using the framework from Section 5.

We developed an auto-optimizer that efficiently finds energy efficient accelerator designs for given DNNs. The optimizer takes as input the DNN topology, the energy cost model, and various constraints such as the total chip area. First, according to **Observation 1**, we fix the dataflow to be $C \mid K$, and only search the design points on the optimizing plane in Figure 1. Next, we only evaluate a subset of hardware configurations with the optimal size of each memory level satisfying **Observation 2**, leveraging the rule that the ratio of the on-chip storage sizes and the adjacent levels should be around 4 to 16. It outputs an optimized design with corresponding Halide schedule primitives, which can then be fed into our hardware synthesis toolchain.

We use four CNNs, three LSTMs, and two MLPs as benchmarks to demonstrate the effectiveness of our efficient optimizer. All DNNs evaluated use 16-bit precision. The CNNs are AlexNet, VGG-16 [38], MobileNet [19], and GoogleNet [42] with batch size 16. The LSTM-M and LSTM-L are proposed by Google for sequence-to-sequence learning [41] with embedding sizes 500 and 1000. We also study the Recurrent Highway Network (RHN) [50]. The MLPs are from [11] with

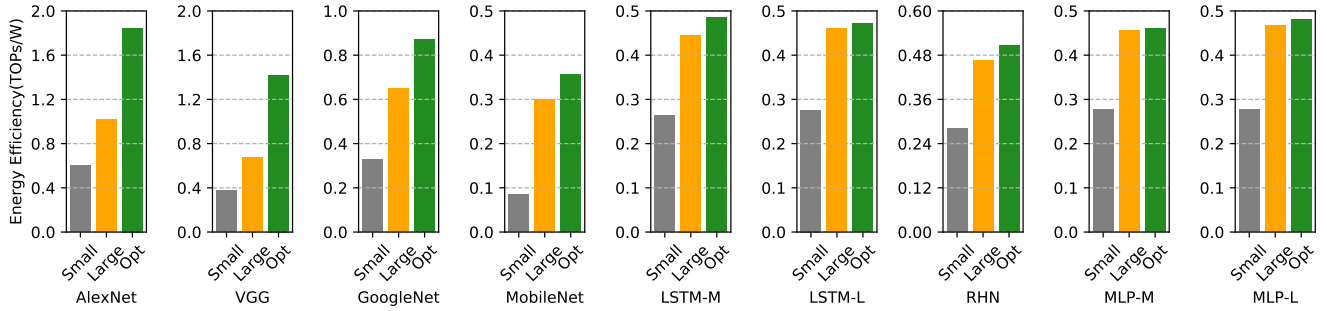


Figure 14. Overall energy efficiency improvement by using the auto-optimizer.

batch size 128. We use two baselines, both using dataflow $C \mid K$, which are the two left columns in Figure 14. The smaller chip uses a memory hierarchy similar to Eyeriss [8], and 16×16 PE array, whose area and power budgets are suitable for mobile platforms. The larger chip uses 128×128 PE array with a 8 B register per PE, 64 KB for the first-level global buffer, and a 28 MB second-level global buffer, similar to cloud-based accelerators such as TPU [20].

Figure 14 demonstrates the energy efficiency gain achieved by the efficient optimizer. We can improve the energy efficiency by up to $3.5\times$, $2.7\times$, and $4.2\times$ for VGG-16, GoogleNet and MobileNet, up to $1.6\times$ for LSTMs, and up to $1.8\times$ for MLPs. The optimal memory hierarchy uses 16 B and 128 B for the first-level and second-level register files, with a 256 KB global SRAM double buffer. This hardware configuration is shared by all the layers in the DNNs. The overall system energy consumption is not dominated by the RF level. The energy efficiency for the nine benchmarks are 1.85, 1.42, 0.87, 0.35, 0.49, 0.47, 0.5, 0.46, and 0.48 TOPs/W, respectively. Notice that even though the larger system has a smaller RF size, its energy is better than the smaller system. This is because with a much larger global SRAM buffer, it can store all the input and output data and the layer weights, and the accesses to DRAM are eliminated when switching to the next layer.

7 Conclusion

To help elucidate factors that matter for DNN accelerators, we realized that **Halide’s scheduling language was almost rich enough to describe the design space of all possible accelerators**. By extending the scheduling language to include local communication, and creating a hardware backend, we were able to generate and hence fairly compare all proposed dense DNN accelerators. The results show that as long as the data reuse and the resource utilization are maximized by proper loop blocking and mapping replication, many hardware dataflow choices will be near optimal. Not surprisingly, optimizing the memory hierarchy had a large influence on energy efficiency, with smaller local registerfiles and deeper memory hierarchies providing the best performance.

8 Acknowledgments

The authors want to thank Kayvon Fatahalian, Jonathan Ragan-Kelley, Andrew Adams and Stephen Richardson for the insightful discussion, Keyi Zhang for his help on LaTeX formatting, and the anonymous reviewers for their valuable comments. This work was supported by the DSSoC DARPA grant, the Stanford AHA Agile Hardware Center and Affiliates Program, the Stanford SystemX Alliance, the Stanford Platform Lab, and SRC Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

References

- [1] ARM ML Processor. <https://developer.arm.com/products/processors/machine-learning/arm-ml-processor/>.
- [2] NVDLA. <http://nvdla.org/>.
- [3] Tensilica customizable processor IP. <http://ip.cadence.com/ipportfolio/tensilica-ip>.
- [4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016.
- [5] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.
- [6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.
- [7] Y. Chen, T. Yang, J. Emer, and V. Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, June 2019.
- [8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.
- [9] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, 2016.
- [10] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam.

- DaDianNao: A machine-learning supercomputer. In *47th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 609–622, 2014.
- [11] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *43rd International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.
- [12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015.
- [13] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *2011 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 109–116, 2011.
- [14] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and efficient neural network acceleration with 3D memory. In *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [15] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, pages 807–820, New York, NY, USA, 2019. ACM.
- [16] Vinayak Gokhale, Jonghoon Jin, Aysegül Dundar, Berin Martini, and Eugenio Culurciello. A 240 G-ops/s mobile coprocessor for deep neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 696–701, 2014.
- [17] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.
- [18] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
- [19] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.
- [20] Norman P. Jouppi, Cliff Young, Nishant Patil, David 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. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2017.
- [21] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuro-morphic architecture with high-density 3D memory. In *43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, 2016.
- [22] Jong Hwan Ko, Burhan Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation. In *Proceedings of the 54th Annual Design Automation Conference 2017, DAC '17*, pages 59:1–59:6, New York, NY, USA, 2017. ACM.
- [23] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.
- [24] Hyoukjun Kwon, Michael Pellauer, and Tushar Krishna. MAESTRO: an open-source infrastructure for modeling dataflows within deep learning accelerators. *CoRR*, abs/1805.02566, 2018.
- [25] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*, pages 461–475, New York, NY, USA, 2018. ACM.
- [26] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016.
- [27] Wenyuan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564, 2017.
- [28] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 429–443, New York, NY, USA, 2015. ACM.
- [29] Naveen Muralimanohar, Rajeev Balasubramanian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40*, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.
- [30] A. Parashar, P. Raina, Y. S. Shao, Y. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, March 2019.
- [31] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Pugliese, Rangharajan Venkatesan, Bruce Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.
- [32] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Trans. Archit. Code Optim.*, 14(3):26:1–26:25, August 2017.
- [33] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

- [34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.
- [35] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.
- [36] Yongming Shen, Michael Ferdman, and Peter Milder. Overcoming resource underutilization in spatial CNN accelerators. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.
- [37] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2017.
- [38] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [39] Mingcong Song, Jiaqi Zhang, Huixiang Chen, and Tao Li. Towards efficient microarchitectural design for accelerating unsupervised GAN-based deep learning. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 66–77. IEEE, 2018.
- [40] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25. ACM, 2016.
- [41] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.
- [42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.
- [43] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017, DAC '17*, pages 29:1–29:6, New York, NY, USA, 2017. ACM.
- [44] Xuan Yang. *A Systematic Framework to Analyze the Design Space of DNN Accelerators*. PhD thesis, Stanford University, 2019.
- [45] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. A systematic approach to blocking convolutional neural networks. *arXiv preprint arXiv:1606.04209*, 2016.
- [46] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.
- [47] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.
- [48] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17*, pages 35–44, New York, NY, USA, 2017. ACM.
- [49] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An accelerator for sparse neural networks. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.
- [50] Julian G. Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *CoRR*, abs/1607.03474, 2016.
- [51] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13*, pages 9–18, New York, NY, USA, 2013. ACM.