DNN Dataflow Choice Is Overrated

Xuan Yang[*], Mingyu Gao[*], Jing Pu[*], Ankita Nayak[*], Qiaoyi Liu[*], Steven Emberton Bell[*], Jeff Ou Setter[*],
Kaidi Cao[†], Heonjae Ha[*], Christos Kozyrakis[*] and Mark Horowitz[*]

[*]Stanford University
[†]Tsinghua University

Abstract—**Many DNN accelerators have been proposed and built using different microarchitectures and program mappings. To fairly compare these different approaches, we modified the Halide compiler to produce hardware as well as CPU and GPU code, and show that Halide's existing scheduling language has enough power to represent all existing dense DNN accelerators. Using this system we can show that the specific dataflow chosen for the accelerator is not critical to achieve good efficiency: many different dataflows yield similar energy efficiency with good performance. However, finding the best blocking and resource allocation is critical, and we achieve a $2.6\times$ energy savings over Eyeriss system by reducing the size of the local register file. Adding an additional level in the memory hierarchy saves an additional 25%. Based on these observations, we develop an optimizer that automatically finds the optimal blocking and storage hierarchy. Compared with Eyeriss system, it achieves up to $4.2\times$ energy improvement for Convolutional Neural Networks (CNNs), $1.6\times$ and $1.8\times$ improvement for Long Short-Term Memories (LSTMs) and multi-layer perceptrons (MLPs) respectively.**

I. Introduction

Advances in deep, multi-layer neural networks have led to a resurgence of their use in many application domains [1]. 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. 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) [2]. The DianNao family was built around customized inner-product units [3], [4]. Eyeriss highlighted the importance of on-chip dataflow on energy efficiency, and proposed a row-stationary heuristic [5], [6], while Google's TPU used a simple yet efficient systolic dataflow on a large 2D array of processing elements (PEs) [7]. These are just a few of the recent publications on DNN acceleration. All the publications state the advantages of their approach, yet the architecture and dataflow of the DNN differ significantly between these approaches.

These differences raise the question whether the choice of the microarchitecture and dataflow are really that critical in dense DNN computations, and whether there is fair methodology to compare these different approaches. To answer this, we start with Halide [8], a domain-specific language which has been used to map image processing applications to hardware. We realized that the different DNN microarchitectural choices could be expressed by a Halide schedule, since the overall algorithm being performed in each of the designs was the same. Using this insight, we extended Halide to create hardware for dense linear algebra in addition to image processing. This system allows us to create different DNN mappings and hardware by simply changing the schedule in Halide, and using it we can easily recreate the previously proposed designs. Since we use the same building blocks for all the designs, it also enables us to fairly compare the resulting performance and energy efficiency.

Perhaps unsurprisingly given the diversity of research solutions, our analysis using Halide shows that when properly blocked, the specific dataflow used in the design does not have a significant impact on the overall performance or energy efficiency. There is enough parallelism and locality in most DNN networks that, properly blocked, many schedules work well.

In fact, energy efficiency is more tightly tied to the design of the hierarchical memory system and how each layer is sized. For every operation such as a multiply-add (MAC), at least three register file (RF) accesses are required. Since the cost of each RF fetch is proportional to the RF size, it is most efficient to adopt a relatively small RF. The hierarchy depth also matters, since the size ratio between the adjacent memory levels should be in a certain range to balance the cost of accessing the next larger memory. Using these insights, we created an auto-optimizer for these types of Halide programs, which achieves up to $4.2\times$, $1.6\times$ and $1.8\times$ energy improvement over Eyeriss system for various CNNs, LSTMs and MLPs respectively.

This work makes the following contributions:
- Introduces a systematic approach to concisely describe the design space of DNNs.
- Shows that the dataflow and hardware resource allocation for existing neural network accelerators can be expressed with different schedules of a Halide program.
- Describes modifications to Halide which enable it to generate FPGA and ASIC hardware implementations for DNNs.
- Demonstrates that many dataflow patterns achieve similar energy efficiency and perforamnce, and that the choice of hardware and memory size is more important than the choice of dataflow.
- These insights enable us to create an fast auto-optimizer to tune the memory hierarchy for this application class.

The next section briefly reviews DNN accelerators. Section III describes the design space of DNNs and provide a formal taxonomy for dataflows. Section IV introduces the

Halide language and shows how we extended it to generate different hardware implementations, and Section V discusses how we created an analytical model from the ASIC designs we generated from this Halide framework. We then use this analytic framework to evaluate the energy and performance of different designs in Section VI.

II. DNN Accelerators

The importance of DNNs, combined with their computational intensity, has led many researchers to investigate hardware acceleration. While often this concerted research effort converges to a few common approaches, this doesn't 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 the data were streamed to and from DRAM, but with limited flexibility to customize the memory hierarchy [2]. Its successor TeraDeep used a fixed loop-blocking strategy for CONV layers [9]. The DianNao family was built around customized inner-product units. Its first generation used a single level of small buffers [3], while in a later iteration the original unit was surrounded by large eDRAM that stored the complete sets of data and layer weights [4]. Another version specially built for embedded systems further extended to a 2D PE mesh that supported optimized inter-PE data propagation [10]. More recently, Eyeriss highlighted the importance of on-chip dataflow for energy efficiency, and proposed using row-stationary dataflow as a heuristic solution [5], [6]. Neurocube and Tetris combined the spatial PE arrays with 3D-stacked DRAM to reduce the main memory access cost [11], [12]. Cambricon proposed an ISA for DNN computation [13]. FlexFlow leveraged the complementary effects of different dataflow styles and mixed them on the same PE array to improve resource utilization [14]. Other prior work has also implemented architectures that are flexible to support multiple different dataflow types [15], [16]. Google's TPU also used a simple systolic dataflow on a large 2D array of PEs, which could also be used for MLPs and LSTMs in addition to CNNs [7]. Recently Song et al. proposed to reorganize dataflows to improve the data reuse for non-standard convolutional layers in Generative Adversarial Networks (GANs) [17]. Other designs have explored the sparsity of DNN computation and proposed dataflow schedules specific for sparse NN processing [18], [19], [20], [21], [22]. Another group of designs have transformed CNN computations into frequency domain [23], [24].

On FPGA platforms, Zhang et al. also 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 [25]. Alwani et al. fused the computation of different NN layers to reduce intermediate data writeback [26]. Shen et al. optimized FPGA resource utilization by using a heterogeneous design [27], [28]. And Li et al. mapped the computation of an entire CNN onto an FPGA in a pipelined manner [29]. Sharma et al. provided hand-optimized templates for generating dataflow architectures [30].

III. Design Space

The numerous DNN accelerators discussed in section II have all demonstrated enormous improvements over the general-purpose baselines. However, due to their different hardware architectures, implementation platforms, and design technique choices, it is very difficult to equitably compare the performance and energy efficiency *across* these accelerators. Moreover, the design space of DNN accelerators has not been clearly defined, making it difficult to directly pinpoint the key factors for an efficient design. To help correct these shortcomings, this section describes a systematic approach to describe the DNN accelerator design space, into which we map the previous designs in order to draw fair comparisons between them. The next section reviews the computation in a DNN to introduce the terminology we use in the rest of the paper.

A. DNN Structure and Terminology

A Deep Neural Network (DNN) is a pipeline or directed acyclic graph (DAG) of various types of *layers*. In vision problems, the DNNs used are usually Convolutional Neural Networks (CNNs) composed primarily of convolutional (CONV) layers. These layers organize their neurons into a a set of 2D images with dimensions X and Y, called a feature map (*fmap*). The number of the fmaps in a set is C, which corresponds to the color *channels*. These C fmaps are the input to the CONV layer, and produces K output fmaps through K shift-invariant 3D stencil filters, each with size $C \times F_X \times F_Y$. Typically the dimensions of the filters (F_X and F_Y) are much smaller than the fmap dimensions (X and Y). The $K \times C \times F_X \times F_Y$ filter coefficients are the *weights* of the CONV layer. Finally, a common way to increase parallelism and amortize processing cost is *batching*, where the weights are reused across multiple input images, which adds another dimension b to the fmaps. Thus, the the CONV layer computation can be formally summarized as:

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

where O is the output, I is the input, and W contains the filter weights. The computation of a CONV layer can also be represented as seven levels of nested loops, as depicted in Algorithm 1.

Fully-connected (FC) layers are also commonly present in DNNs such as Multi-Layer Perceptrons (MLPs) and as the last few layers of CNNs. In addition, the most commonly used Recurrent Neural Networks (RNNs), Long Short-Term Memories (LSTMs), also contain FC layers in their basic cell structure. The FC computation can be thought of as an $M \times N$ matrix-vector multiplication, which maps M input values to N output values. Unlike CONV layers, the $M \times N$ unique weights have much larger dimensions than the layer inputs and outputs, and are only reused when applied to data batches. Note that FC

```
1  for b = 0 : B do                                          (Batch)
2  │  for k = 0 : K do                                (Output channel)
3  │  │  for c = 0 : C do                              (Input channel)
4  │  │  │  for y = 0 : Y do                             (fmap height)
5  │  │  │  │  for x = 0 : X do                           (fmap width)
6  │  │  │  │  │  for f_y = -\frac{F_Y-1}{2} : \frac{F_Y-1}{2} do   (Filter height)
7  │  │  │  │  │  │  for f_x = -\frac{F_X-1}{2} : \frac{F_X-1}{2} do  (Filter width)
8  │  │  │  │  │  │  │  O[k][x][y] +=
                        I[c][x + f_x][y + f_y] × W[k][c][f_x][f_y]
```

Algorithm 1: The seven nested loops of the convolutional layer. In this simple example, the loop order is represented by $F_x F_y XYCKB$, from innermost to outermost.

layers can also be described similarly using the nested loops in Algorithm 1, but only with C, K, and B loops, while other loops bounds are all 1.

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

B. Design Space Overview

The design considerations that have been widely studied to optimize data reuse and performance of DNNs are *loop blocking* [31] and *dataflow* [5]. However, when exploring such software scheduling design choices, the hardware *resource allocations* used by prior work are also often different from each other. Therefore, we present a three-dimensional design space, with the three axes being *loop blocking*, *dataflow*, and *resource allocation*. We can decouple these three factors, and independently investigate and optimize each of them.

Loop Blocking: Assuming a multi-level memory hierarchy (e.g., register files, on-chip SRAM, and off-chip DRAM), we want to maximize the data reuse in the near, smaller memories which have lower energy cost, to minimize the data accesses to far, larger memories which have higher energy cost. What makes this scheduling hard is the fact that all the data fetched — input, output, and weights — can be reused multiple times. For example, an input fmap is reused as it is convolved with each filter for different output fmaps, while the filters are reused across all the pixels of an entire input fmap. An optimal schedule must balance these data reuse opportunities. The techniques of loop tiling and reordering, which together we refer to as loop blocking [31], can generate different schemes with different subsets of data buffered in each memory level. Selecting the loop blocking schemes decides the data reuse efficiency in the memory hierarchy.

Dataflow: Hardware accelerators also exploit parallelism to improve performance, by using multiple processing elements (PEs) to execute operations simultaneously. In essence, this flattens one or more loops in Algorithm 1 by executing all of the loop iterations in parallel. We refer to this as *spatial unrolling*, since it is the hardware equivalent of unrolling a

loop in software, but exploits the replication of the hardware in physical space. The spatial loop unrolling also determines the data access and communication patterns within the accelerator, which is called the *dataflow* [5]. The dataflow must be 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 subsection III-C.

Both *loop blocking* and *dataflow* can be regarded as loop schedules, and many algorithms have been proposed for these optimization problems, some general, like Polyhedral analysis [32], [33] and others tailored to CNNs like Eyeriss [5], [6]. However, it is insufficient to consider only the loop schedule when designing an accelerator.

Resource Allocation: The hardware resource allocations, such as 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 many key configurations, including the computation throughput, the location of the data, and the energy cost and latency for each memory access. An efficient design needs to carefully manage such *resource allocation*, the energy cost and latency of each access grow with memory size. The allocation must set the size of each memory level to optimally balance buffering sufficient data to maximize locality and providing each fetched data for lowest energy cost.

C. A Formal Dataflow Taxonomy

Chen et al. presented a taxonomy to group the previous accelerators into dataflow categories based on the data stationary characteristics [5]. Yet their categories — weight stationary, output stationary, no local reuse, and row stationary — do not cover the entire design space. For example, a hybrid stationary pattern, which stores both weights and output values in the PE registers is not represented in their scheme. To enumerate the entire design space we step back to look at the overall computation, and then create a formal way to describe it.

An accelerator's dataflow pattern is defined by the mapping of particular loops to its parallel computation structures. Said another way, 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 two dimensions of a 2D array, then each PE is responsible for producing a single output pixel, which is the output stationary pattern. This pattern implies that input pixels will be reused across neighbor PEs as they contribute to multiple output pixels in a convolution, and the weights for different output fmaps must be either broadcast to the corresponding PEs, or replicated and stored privately inside each PE. If we instead unroll the F_X and F_Y loops onto the 2D array, we obtain a weight stationary pattern, where the weights are reused within the same PEs, but the input data are broadcast or replicated, and the output data are spatially accumulated.

To concisely represent all of the possible unrolling schemes on a 2D array of PEs, we use the syntax $U \mid V$, where
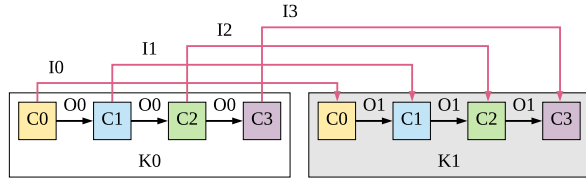
Fig. 1: Mapping 2 unrolled loops C and K to a 1D array using dataflow CK. Outputs are communicated between adjacent PEs, while inputs are communicated across blocks.

U and V denote the loops which are unrolled across the vertical and horizontal dimensions, respectively. Table I shows several common dataflows expressed as unrolled loops and the corresponding terminology in prior work. Understood this way, there are $\binom{L}{d}$ possible dataflow patterns, with L being the number of loops and d being the number of spatial dimensions. Given a 2D spatial array, the number of dataflow types for CONV layers is $\binom{7}{2} = 21$; for fully-connected layers, the number is $\binom{3}{2} = 3$.

Dataflow representation	Common name
$X \mid Y$	Output stationary
$F_X \mid F_Y$	Weight stationary
$F_Y \mid Y$	Row stationary
$C \mid K$	Weight stationary

TABLE I: Dataflow examples from Chen et al. [5] expressed by unrolling certain loops in the algorithm.

Of course, it is also possible to unroll a loop only partially, or to unroll multiple loops onto one hardware dimension. Breaking a large loop tile (*folding*) is useful when the loop is larger than the array size, and processing multiple tiles in parallel (*replication*) increases utilization when handling small tiles. When replication is supported, the number of dataflow types increases to $\binom{7}{x}$ for a CONV layer mapping to a 2D array, where $x >= 2$, thus the dataflow design space becomes even larger.

With replication, the data communication pattern is no longer uniform: only one type of data can be communicated among nearest-neighbor PEs, while other data types have to be sent to multiple hops away, with a 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 1 shows an example of mapping unrolled C and K loops onto a 1D array. The 8 PEs have been divided into 2 sets, each set works on a output channel dimension K. The weights for the corresponding output channel is either broadcast to or replicated inside each PE. Within each group, every PE processes an input channel, and relay the output to the next neighbor PE. Thus, the outputs are only communicated among the nearest PEs, while the inputs have to transfer from one group to the next group. Assuming the nearest neighbor communication cost is a unit

cost 1, then for this example, each input access at the array level consumes a cost of 4.

Dataflows	Previous design
$X \mid Y$	[10], [17]
$X \mid K$	[30]
$F_X \mid F_Y$	[34], [17]
$C \mid K$	[3], [7], [25], [26], [27], [28], [35]
$F_Y \mid Y$	[5], [6], [12]
$F_X \mid X$	[29]

TABLE II: Categorization of prior DNN accelerators using our dataflow taxonomy.

This dataflow taxonomy opens up a much broader design space, and allows us to optimize dataflow and resource allocation independently. Table II categorizes existing DNN accelerators based on this taxonomy. Converting a convolution to matrix multiplication is widely used by previous work, as it provides the flexibility to map MLPs and LSTMs in addition to CNNs, thus a lot of accelerator designs have adopted the dataflow $C \mid K$. Even though this dataflow keeps the weight stationary, the data reuse pattern is different from the weight-stationary pattern introduced in Eyeriss, which unrolls F_X and F_Y. Using this formal loop-based taxonomy, we can clearly distinguish these and many others which are not covered by the Eyeriss taxonomy.

Through this dataflow taxonomy, various dataflows and loop blockings of the previous work can be expressed and represented by the transformations of the seven nested loops. This characteristics aligns nicely with the capability of *Halide*, an image processing domain-specific language (DSL), since it provides a compact and elegant representation of these kinds of loop transforms. The next section shows how we use the Halide scheduling language to specify the locality, dataflow, and resource configuration.

IV. HARDWARE GENERATION

A. Halide

Halide [8] 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*), and to provide a compact language for specifying schedules, enabling developers to rapidly explore the space of possible schedules without fear of accidentally breaking the correctness of the computation.

Halide algorithm: To achieve this split, Halide represents the computation in pure functional form. As an example, Figure 2a shows the Halide algorithm for a CONV layer. The `RDom` keyword defines a multi-dimensional reduction domain, over which an iterative computation such as a summation is performed. The `RDom` is defined by the minimum position and extent of each dimension of the domain. In the CONV layer example, the `RDom` covers the width and height of the filters and the number of input fmaps, so the accumulation will

```
1  // Define (minX, extentX, minY, extentY, minK, extentK)
2  RDom r(-2, 5, -2, 5, 0, 3);
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);
```

(a) The Halide algorithm for a $5 \times 5 \times 3$ convolution.

```
1  d = output.in()
2
3  output.tile(x, y, xo, yo, xi, yi, 28, 28)
4        .reorder(xi, yi, xo, yo)
5        .accelerate({input, w});
6
7  input.in().compute_at(output, xo);
8  w.in().compute_at(output, xo);
9  output.compute_at(d, xo);
```

(b) An example Halide schedule for the convolution above.

```
1  for (k, 0, 64)
2
3    for (yo, 0, 4)
4      for (xo, 0, 4)
5        // Allocate local buffer for output.
6        alloc obuf[28, 28, 1]
7
8        // Allocate local buffer for input.
9        alloc ibuf[28 + 5 - 1, 28 + 5 - 1, 3]
10       // Copy input to buffer.
11       ibuf[...] = input[...]
12
13       // Allocate local buffer for w.
14       alloc wbuf[5, 5, 3, 1]
15       // Copy w to buffer.
16       wbuf[...] = w[...]
17
18       for (yi, 0, 28)
19         for (xi, 0, 28)
20
21           for (r.z, 0, 3)
22             for (r.y, -2, 5)
23               for (r.x, -2, 5)
24                 obuf(xi, yi, 0) +=
25                   ibuf(xi + r.x, yi + r.y, r.z)
26                   * wbuf(r.x + 2, r.y + 2, r.z, 0)
27
28       // Copy buffer to output.
29       output[...] = obuf[...]
```

(c) Pseudocode for the implementation generated by Halide.

Fig. 2: Halide algorithm and schedule, and the corresponding implementation, for a CONV layer with 64 output fmaps of size 112×112, 3 input fmaps, and filters of size 5×5.

iterate over these three dimensions, as shown in the innermost three loops in Figure 2c.

Halide schedule: While the algorithm defines the functionality of the computation, it does not define the ordering of parallel operations or the data storage. These are controlled using Halide schedules, which consist of *scheduling primitives* applied to various stages of the algorithm. The basic Halide scheduling primitives are based on loop transformations, such as loop splitting, reordering, and unrolling. Such operations are the same choices that form the loop blocking and dataflow axes of our design space, meaning that it is possible to use Halide's existing scheduling language to rapidly explore large parts of the DNN design space. Figure 2b shows an example schedule for the CONV layer algorithm in Figure 2a, from which the Halide compiler will generate an implementation similar to Figure 2c. In lines 3 and 4, tile and reorder perform loop

splitting and interchanging, respectively. Each of the original x and y loops is split into an inner loop of size 28 and an outer loop, effectively blocking the fmaps into 28×28 tiles. The primitives in and compute_at introduce additional levels of local data buffering. As in Figure 2c, three local buffers are allocated inside the loop xo, for input, output, and weight data, respectively. The compiler automatically determines the necessary buffer sizes.

This suffices for mapping DNNs to existing hardware, but our aim is to generate our own accelerator designs from Halide and fully map the design space. To do this, we introduce two additional scheduling primitives and a template architecture for a DNN accelerator. Based on the Halide schedule we can extract a complete description of the loop blocking, dataflow, and resource allocation, and then apply these parameters to the template to create a specific architecture.

Our first new primitive is 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. [36]. The second new primitive is systolic, which controls the data access patterns inside the compute kernel. It allows data to be read from neighboring PEs in a systolic manner [5], [7], rather than always accessing the higher-level data buffers outside of the loops. Halide's loop splitting and reordering primitives retain their functionality, and we overload the existing unroll primitive to specify spatial hardware unrolling onto the PE array (see subsection III-C).

With these 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 III. Table III summarizes how the scheduling primitives control each of the three dimensions of the design space, and the following section describes how these are applied to our architectural template.

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

TABLE III: The scheduling primitives mapped to each axis of the 3D design space.

B. Architectural Template for DNNs

Like most DNN accelerators, our architectural template uses a grid of PEs to perform the multiply-add operations in DNN layers, and uses a hierarchy of memories to optimize the data locality, as shown in Figure 3.

Micro-architecture of the PE array: The PE array contains a large grid of processing elements, each of which contains an arithmetic core and a register files (regfile) to maximize local data reuse.

If the systolic primitive has been applied, then the PEs are connected together into a systolic array, which enables inter-PE communication and reduces the number of accesses
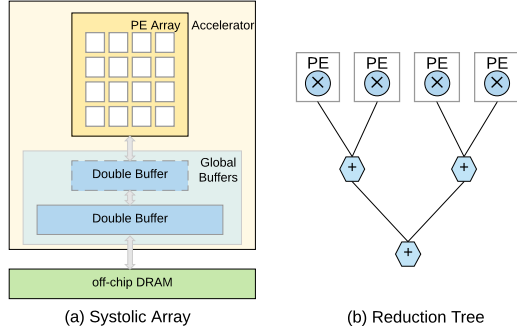
(a) Systolic Array   (b) Reduction Tree

Fig. 3: A DNN accelerator template that consists of a PE array and a memory hierarchy.

to the next-level global buffers to save energy. For example, in Figure 4b, we unroll the F_Y and Y loops to realize dataflow $F_Y \mid Y$ used in Eyeriss [5], which transfers multiple rows of filter weights horizontally, and accumulates multiple rows of output fmaps vertically. On the other hand, Figure 4c performs a matrix multiplication using dataflow $C \mid K$, which is used by a large group of designs including Google's TPU [7].

If `systolic` is not used, the PEs are linked into one or more reduction trees [3], depending on the computation pattern, as shown in Figure 3. Figure 4a also presents an example for this micro-architecture, where dataflow C can be implemented using a 1D reduction tree of PEs, by unrolling the loop C to make each input from different input fmaps multiply its corresponding weight. Then the products are accumulated into a single output element in an output fmap.

Double buffers in memory hierarchy: In the prior work generating hardware from Halide, intermediate data was stored in a line buffer [36]. This works well for stencil-based image processing applications where only a few lines of the image need to be buffered, but is insufficient for the more complex reuse patterns in DNNs.

Instead, our template uses a multi-level hierarchy of *double buffers*, each of which stores subsets of the input, output, and weight data in DNN layer processing (Figure 3). Double buffers allow the hardware to overlap computation and memory loads by supplying operands out of one buffer while performing load/store operations on the other.

The Halide scheduling primitives `in` and `compute_at` determine which blocks of data are stored at which loop levels. By combining these primitives with the information about loop sizes, the compiler is able to instantiate the correct number of storage levels and configure each level with the appropriate size and data layout.

C. Flexible Hardware Generation

As we have described it so far, our Halide to hardware design flow generates a hardware module for each Halide function, i.e., for each layer in the DNN. For large DNNs, instantiating the entire network in hardware requires unrealistically large silicon area. Since the computation patterns of different layers (CONV and FC) in a DNN are similar, it is

often more efficient to time-multiplex all the layers onto a small number of accelerator instances.

To support this, the generated hardware must be sufficiently configurable to support all the layers that will be mapped to it. To create these flexible modules, we adopt a two-step approach. First, we generate the programmable accelerator using a parameterized Halide function that represents an abstract CONV layer, then we schedule each layer of the network onto the parameterized hardware.

The complete design flow is shown in Figure 5. It takes two inputs: A complete DNN specified as a Halide algorithm and schedule, and a parameterized Halide algorithm and schedule for an abstract CONV layer. The compiler uses the latter input to generate the hardware, and then maps the complete DNN to the hardware module it has just created.

The generation process begins with an analysis pass on the parameterized input code to extract the architecture template. A transformation pass then operates on the section marked for acceleration to produce a dataflow intermediate representation suitable for High-Level Synthesis (HLS). After optimizations such as constant propagation and common sub-expression elimination, the dataflow Intermediate Representation (IR) is passed to Vivado HLS and Catapult HLS, which generate hardware designs targeted to FPGA and ASIC respectively. We use two different dataflow IR transformations for the two backends, since the HLS directives and code structures are different between the two HLS tools.

Once the parameterized hardware has been created, the compiler reads in the full DNN, extracts the configuration for each layer, and uses another IR transformation to emit function calls to the configured hardware. The arguments of the function calls act as the configuration bits to the hardware.

V. METHODOLOGY

While the Halide hardware design generation flow from section IV can support both ASIC and FPGA backends, we focus on the ASIC platform in this paper. The Halide to hardware framework generates C++ code specialized for Catapult High-Level Synthesis, which then we compile to Verilog using Catapult. The results in this paper were generated by synthesizing the resulting RTL design into a 28 nm technology using Synopsys Design Compiler using topo mode so it places the cells. The wire length and capacitance is extracted from this placement. Standard cells and memory models from a commercial vendor were used for power, performance, and area analysis of the gate-level netlist. All of our ASIC designs achieve 400 MHz operation with no timing violations. For power analysis, the appropriate switching activities are set on all the primary ports and propagated through the design by the design tools.

A. 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 to model the SRAM arrays and tune its parameters to match
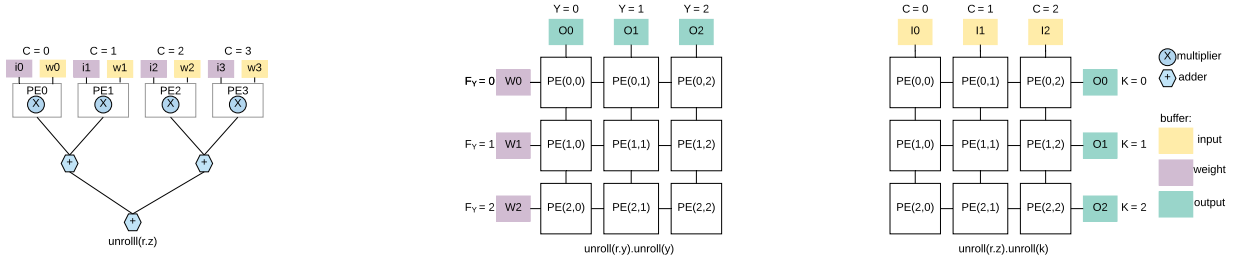
(a) A reduction tree of dataflow C, generated by unrolling the input fmap dimension C.

(b) A systolic architecture of dataflow $F_Y \mid Y$, generated by unrolling the output fmap height Y and filter weight height F_Y.

(c) A systolic architecture of dataflow $C \mid K$, generated by unrolling the input and output fmap dimensions C and K.

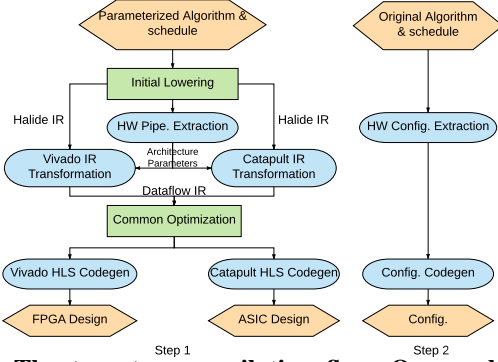**Fig. 4: Different PE array micro-architectures generated from Halide scheduling primitives.**

Fig. 5: The two-step compilation flow. Orange hexagons are inputs and outputs of the compiler. Green rectangles are existing Halide compilation passes. Blue round boxes are newly added to the Halide system.

Reg. 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			
			DRAM	200
MAC	0.075			

TABLE IV: Energy per 16-bit access for various register file and SRAM sizes.

our 28 nm commercial memory library [37]. For small arrays and register files, we use the Cadence XtensaProcessor Generator to extract energy numbers based on our standard cell library [38]. Table IV shows the energy cost of accessing memories of different sizes. Note that our energy ratios between memory and MAC are larger than those reported in Eyeriss [5]. There are several possible reasons: we use 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.

To compute the overall memory energy in an n-level mem-

ory hierarchy, we adopt a model similar to Eyeriss [5]:

$$E = \sum_{i=1}^{n} \#\text{acc}_i \times e_i \quad \text{where} \quad \#\text{acc}_i = \prod_{j=i}^{n} \text{RT}_j \quad (2)$$

In this equation, e_i is the energy cost of accessing level i once, as shown in Table IV. The total number of accesses is distributed into reuses at different levels in the memory hierarchy. RT_i represents the reuse occurances at level i, which is defined as the number of times the data at the current level is accessed by its immediate lower-cost (child) level during its lifetime in this level. RT_i can be calculated based on the dataflow and loop blocking schemes, as it is a function of the loop orders, tiled loop sizes, and loop unroll factors. Thus, finding the optimal schedule becomes an optimization problem of minimizing E in Equation 2.

When there are multiple PEs in the accelerator, data can be replicated in the local storage (register file) of each PE to enable fast and cheap data access. We also support inter-PE communication to allow the data to be fetched from a neighbor PE rather than the global buffer. We include the cost for such inter-PE access, and consider the data duplication overhead on the register file capacity. Note that for the inter-PE communication cost, where Eyeriss treats all the data communication at the array level equally [5], we distinguish different access patterns based on their communication distances, as explained in Section III-C (see Figure 1). The per-hop communication cost is estimated as the energy consumed by the wires and registers used when propagating data.

B. Framework Validation

We validated the accuracy of our model by comparing its results to complete designs generated by our synthesis system. Table V shows three example designs we have generated in ASIC platforms using the design flow introduced earlier, and Figure 6 show the energy estimates from our analytic model, and post-synthesis. The resulting errors are less than 2%. In addition, we also validated the model against the reported Eyeriss design [5], by using the energy parameters and comparing the total energy consumption for CONV layers in AlexNet. The resulting difference is less than 10%, except the CONV1 layer. This exception is due to the fact that CONV1 has a much larger convolution window size, so the
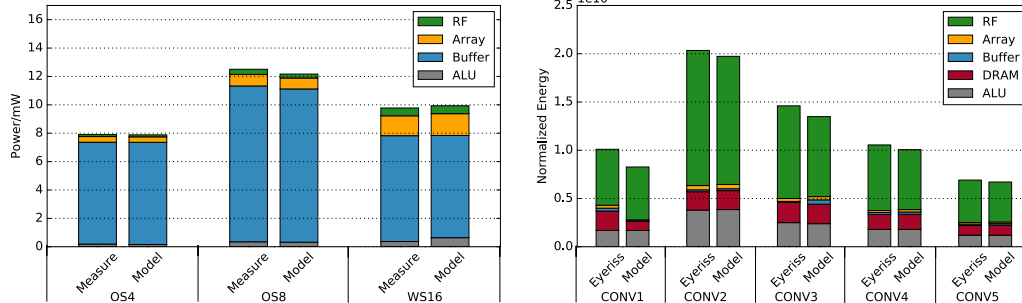
Fig. 6: Analytical model validation. Left: energy breakdown comparison between actual synthesized designs and the analytical model. Right: energy breakdown comparison between reported Eyeriss model and our model.

error caused by the fragmentation is amplified. Such large windows are relatively rare, so our model remains sufficiently accurate on complete networks.

Name	Dataflow	Dimension	PE Number	RF Size	Mem Size
OS4	X	1D	4	32 B	32 KB
OS8	X	1D	8	64 B	64 KB
WS16	$C \mid K$	2D	16	64 B	32 KB

TABLE V: Generated ASIC designs used for validation.

VI. RESULTS

Using our dataflow taxonomy and the ability to rapidly generate and evaluate many accelerator designs with Halide, this section maps out the important features of each dimension of the design space. We begin by exploring dataflow, and then consider resource allocation. Using the insights from these explorations, we draw some general observations and finally introduce an automated scheduler for DNNs.

A. Design Space of Dataflow

In this section we use the layers from AlexNet [39], MobileNet [40] and GoogleNet [41] as our benchmark; we evaluate other DNNs in the following sections. Figure 7 illustrates the complete dataflow design spaces for AlexNet CONV3, MobileNet CONV14 and CONV15 layers, which are representative of all convolution layers in AlexNet, and all depthwise and pointwise layers in MobileNet respectively. The CONV layers of GoogleNet also have similar dataflow space as Figure 7. For each dataflow, the loop blocking scheme is optimized to minimize the energy based on the analysis framework in Section V-A, and the utilization ratio is constrained to be higher than 75%. The setting on the utilization ratio limits the performance degradation allowed. The blue configuration uses the same hardware as Eyeriss [5], which has a 512 B register file (RF) per PE, 128 KB global buffer and 16×16 PE array, with 16-bit precision. The red configuration treats all the communication to be global, with cost independent of the transfer distance, which slightly increases the communication costs compared to the blue configuration. These points should amplify differences caused by communication. To lower the total energy required, the green configuration uses a smaller, 64 byte or 16 byte register file.

Figure 8 redraws this same data creating energy histograms for each of the three design configurations used in Figure 7a. This data shows that different dataflows all achieve similar and close-to-optimal energy efficiency, with only a few outliers.

This result is not sensitive to memory/communication model used. It remains true for a variety of scenarios, including using different layers, different NNs, different spatial array organizations, different PE array sizes, different models for communication cost estimation, and different memory configurations. For example, rather than building a 2-D PE array, we created a 256 PE 1-D systolic array. This design was only up to 0.4% worse than the 2-D array.

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

In hindsight, this result is not surprising since DNNs exhibit enormous data reuse opportunities in their computation. Regardless of the dataflow used, as long as high data reuse is achieved through proper loop blocking schemes, the resulting energy efficiency should be good. This is further illustrated in Figure 9, which shows the energy breakdown of the optimal dataflow for different hardware resource allocations with 512B RF. Most of the energy is consumed in the PE register fetch and DRAM levels, rather than in the intermediate storage. By optimally blocking the computation, nearly all accesses (98%) occur at the RF level, which leads to the RF being the dominant energy component. Additionally, most of the DRAM energy is inevitable, since the inputs, weights and outputs have to be fetched at least once from off-chip. On the other hand, the on-chip communication cost is generally only a small portion of the total energy, and therefore different dataflows do not substantially impact the overall energy efficiency.

Another interesting result of Figure 9 is that the total energy is dominated by the RF level for all three dataflows. This suggests that the resources are not optimally allocated, which we investigate in more detail later.

In addition to energy efficiency, it also is instructive to examine the PE array utilization as a function of dataflow, shown in Figure 10. The left shows utilization for the dataflow choices from Figure 7a that use replication, while the right side shows those which do not. Note since we constrain the

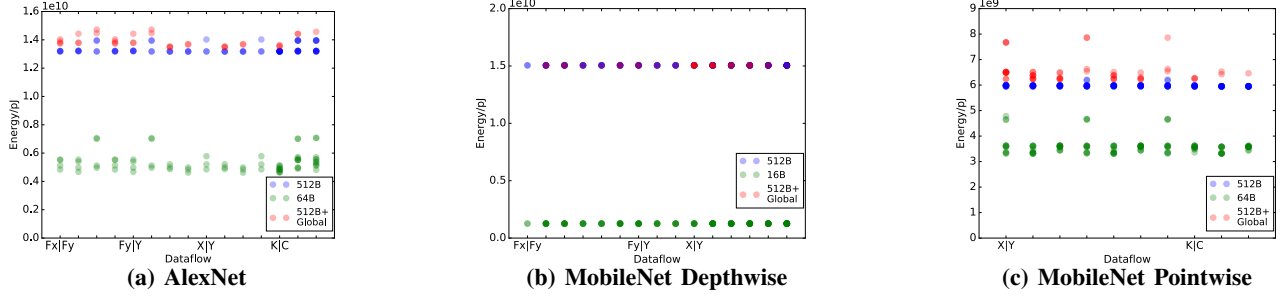**(a) AlexNet**   **(b) MobileNet Depthwise**   **(c) MobileNet Pointwise**

Fig. 7: Design space of dataflow for AlexNet CONV3 and MobileNet CONV14 (Depthwise) and CONV15 (Pointwise). Different dataflows are shown horizontally (for clarity, only the most common choices are labeled). Each color represents a different hardware resource allocation. Dots corresponding to the same X coordinate are designs with the same dataflow but different replications.
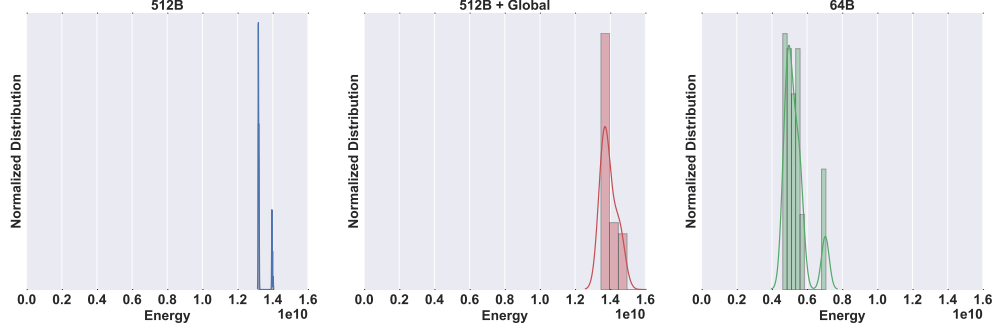

Fig. 8: Energy efficiency distribution across the dataflow design space for the designs in Figure 7a. The leftmost figure uses a 512 B RF, the figure in middle treats all communication as global broadcast, the rightmost figure uses a 64 B RF.
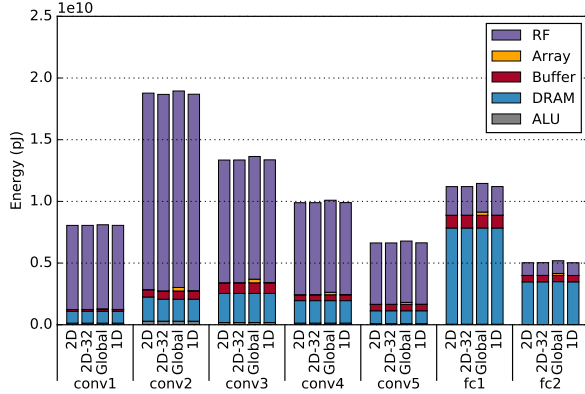

Fig. 9: Energy breakdown of the optimal dataflows with different hardware configurations. 2D and global refer to the best blue and red points in Figure 7a. 2D-32 and 1D change the PE array to 32×32, and 1D with 256 PEs respectively. The communication energy is negligible for all of the hardware configurations. However, the 512B register file (RF) energy always dominates the overall energy for CONV layers.
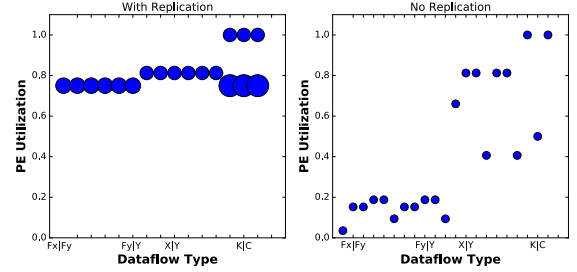

Fig. 10: PE array utilization for all energy-optimal dataflow choices with and without replication. The area of each point is proportional to the number of designs with this utilization.

sensitive to different dataflow choices in the design space than the energy efficiency. Therefore, from a performance perspective, optimizing replication is still useful. The dataflow $C \mid K$ achieves the best utilization, so we will use it following experiments on resource allocation.

B. Impact of Hardware Resource Allocation

Figure 11 shows the energy impact of the memory resource allocation. The energy is accumulated across all layers (including FC layers) in AlexNet, and contains both computation and memory access portions. This figure indicates that by using a smaller register file size, like 32 B or 64 B, the total energy efficiency can be improved by $2.6\times$. If we also increase
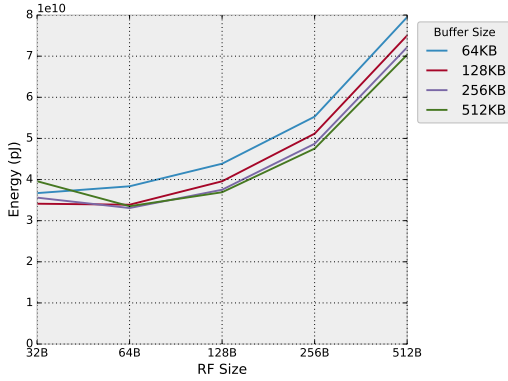
utilization to be higher than 75% when exploring the dataflow design space, all the data points show in this figure have utilization no less than that threshold. Comparing Figure 10 and Figure 7a, we observe that the utilization is much more

Fig. 11: Memory hierarchy exploration with dataflow $C \mid K$. Lines with different colors correspond to different global buffer sizes. Along the X-Axis are different PE register file (RF) sizes.
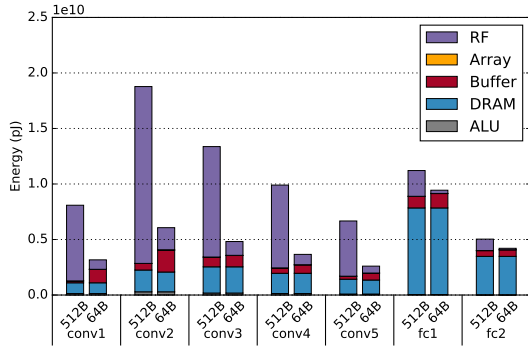

Fig. 12: Energy comparison between 512 B and 64 B register file (RF) sizes with the same dataflow schemes. Using a 64 B RF reduces the overall energy significantly.

the global buffer size, the energy efficiency can be further improved. However, when global buffer size grows beyond 256 KB, the benefit becomes negligible. Given the significant area cost, it is not always necessary to use large global buffers.

We further look at the energy breakdown of using a 64 B register file, shown in Figure 12. Compared with a 512 B register file, the register file energy decreases dramatically for all the CONV layers when using 64 B register files, due to the much lower energy cost per access for the smaller register file. At the same time, more accesses go to the inter-PE array level and the global buffer, since the smaller register file captures less data reuse inside each PE. But reducing the register file size has almost no impact on DRAM energy, as the data are still efficiently reused in the global buffer. Overall, a small register file 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 output-stationary and weight-stationary designs do not perform well, as discussed by Chen et al. [5]. Those designs cannot capture sufficient

reuse at the register file level, and result in significant energy consumption at the DRAM level, which dominates the overall system energy.

However, there is also an exception for Observation 2. When DRAM dominates the total energy but the total number of DRAM accesses is already minimized (fetching the input data once and writing back output once), the DNN is memory bound, and based on Ahmdal's law, little further optimization can be achieved for the memory hierarchy. This scenario is particularly likely for MLPs and LSTMs that contain many
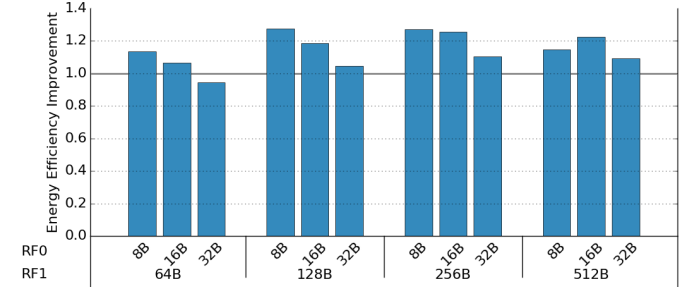

Fig. 13: Overall energy efficiency improvement by adding another level of register file into the memory hierarchy. This improvement is calculated by dividing the overall energy of using two levels of RF by the optimal energy of using single level of RF. Bars that exceed 1 have energy improvement against the previous optimal design when using a single level of RF.

By rearranging the memory sizes in the current hierarchy, we reap a significant efficiency improvement. We next explore whether changing the hierarchy itself, such as adding another level, can further improve the energy efficiency. As shown in Figure 11, the overall energy efficiency is most sensitive to the register file level. Hence we add another level of private register file into the PEs, and plot the resulting impact on energy in Figure 13. We again use the $C \mid K$ dataflow, but other dataflows have a similar trend. We normalize the total energy against that of using one-level register file with the optimal size (64 B).

The energy reduction for the CONV layers in the network, is more than 30%. This reduction leads to an overall efficiency improvement of approximately 25%. This reduction comes from the fact that the FC layers, which are included in the total energy, have its locality exploited in the original memory hierarchy, and almost all the data (input, weights, and output) are only accessed from the main memory once. As a result additional levels of memory hierarchy don't improve the efficiency of theFC layer. We can expect a slightly higher efficiency improvement for ResNet [42] or GoogLeNet [41], as they are mostly composed of CONV layers.

Figure 13 also shows that memory hierarchies for DNN applications follow similar types of sizing rules that are used for multi-level caches, but perfect prefetching allows them to use larger size ratios.

Observation 3: *The ratio of the on-chip storage sizes between the adjacent levels should be 8 or larger, and grows*
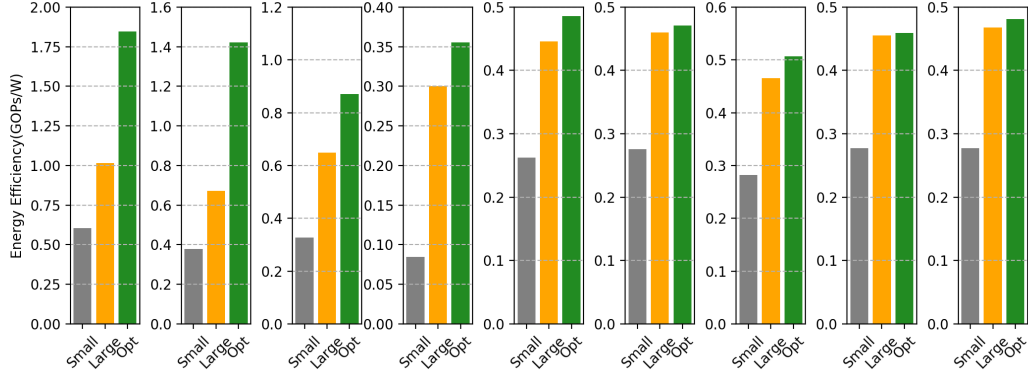
Fig. 14: Overall energy efficiency improvement by using the auto-optimizer. Consider various CNNs, LSTMs, and MLPs.

as you move up the hierarchy. The optimal size of a memory level doesn't depend on levels that are "far" away from it.

In an optimally-sized memory hierarchy, each memory level should shield most of the references it receives from the next level in the hierarchy. Since the energy cost of an access grows slowly with size, this leads to large changes in memory size. The optimal size for the 2-level registers is 8 B/128 B, and 8 or 16B/256B, which both have min ratio of 16. Sixteen of these units create a 4 kB memory, which fetches data from a 256 kB or 512 kB buffer, which is a scale up of 64 to 128. The size of this buffer doesn't depend much on the configuration of the register hierarchy, and mainly depends on shielding most of the DRAM references. Comparing Figure 11 with Figure 13, we find that the optimal global buffer sizes are both 512 KB, the same regardless of different numbers of hierarchy levels.

C. Auto-optimizer

With the tremendous number of hardware and software optimization choices for DNNs, exhaustive search for the optimal designs is usually infeasible. However, using the observations above, we can speed up the optimization process by pruning the search space down and evaluating only the remaining candidates using the framework introduced in Section V-A.

We propose the following detailed approach for the optimization. First, according to **Observation 1**, dataflow does not matter as long as proper loop blocking schemes are used, so we fix the dataflow to be $C \mid K$. Next, we search the optimal size of each memory level within a three-level hierarchy (PE register file, global buffer, and DRAM), where we will only explore a subset of configurations that satisfy **Observations 2 and 3**. Finally, we evaluate the benefit of adding another level of register file. Based on the characteristics of caches, we use the same global buffer size as the optimal one reported by the previous step, and search the optimal sizes of the two levels of register files using **Observations 2 and 3**.

Using this approach, we develop an auto-optimizer that is closely coupled with the Halide framework. We extract the required parameters from the Halide algorithms such as the DNN configurations, and send them to the auto-optimizer. Then we provide the energy cost model of the CMOS technology, as

Network	Embedding size	Batch size
LSTM-M	500	128
LSTM-L	1000	128
RHN	1500	128

Network	Layer	C	K	Batch
MLP-M	FC1	784	1000	128
	FC2	1000	500	128
	FC3	500	250	128
MLP-L	FC1	784	1500	128
	FC2	1500	1000	128
	FC3	1000	500	128

TABLE VI: Benchmarks for auto-optimizer. For CNN, we use AlexNet, VGG-16, GoogleNet and MobileNet.

well as other constraints such as total chip area. The auto-optimizer explores different architecture specifications such as memory and computation core configurations, to optimize for performance and energy efficiency.

Auto-optimizer Results: We use four CNNs, three LSTMs, and two MLPs as benchmarks to demonstrate the effectiveness of our auto-optimizer. The CNNs we use are AlexNet, VGG-16 [43], MobileNet [40] and GoogleNet [41] with batch size 16. The LSTM-M and LSTM-L are proposed by Google, used for sequence to sequence learning [44]. We also evaluate different embedding sizes for this LSTM. In addition the Recurrent Highway Network (RHN) [45] is also evaluated. The MLPs are from [46] with batch size 128. The layer configurations of the benchmarks are shown in Table VI. All DNNs evaluated use 16-bit precision. We use two baselines, both using dataflow $C \mid K$. The smaller chip uses a memory hierarchy similar to Eyeriss [5], and 16×16 PE array, whose area and power budgets are suitable for mobile platforms. The larger chip uses 32×32 PE array, with 8 B register per PE, 64 KB for first-level global buffer and a 28 MB second-level global buffer. Such a chip is similar to cloud-based accelerators such as TPU [7].

Figure 14 demonstrates the energy efficiency gain achieved by the auto-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

MobilelNet, up to $1.6\times$ for LSTM layers, 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 double buffer. This hardware configuration is shared by all the layers in the DNNs, as discussed in Section IV-C. Different from Eyeriss, the overall system energy consumption is not dominated by the register file. The energy efficiency for the five benchmarks are 1.85, 1.42, 0.87, 0.35, 0.49, 0.47, 0.5, 0.46, 0.48 GOPs/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 second-level global 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 in the DNNs.

Finally, we also validate our system by using the FPGA backend to generate a hardware implementation on a Xilinx ZCU102 board containing a Zynq XCZU9EG SoC. The generated accelerator design is comparable to or better than the prior results as shown in Table VII and Table VIII.

DSP	BRAM	LUT	FF	Freq
1030 (40%)	1670 (91%)	39330 (7%)	274080 (5%)	200 MHz

TABLE VII: FPGA resource utilization.

	[34]	[35]	Ours
Device	Zynq XC7Z045	Stratix GSD8	Zynq XCZU9EG
DSP #	780	1963	1030
GOPS	137	117.8	185

TABLE VIII: Comparison with prior FPGA work.

VII. Conclusion

We leverage Halide DSL to express various dataflows, blocking, and hardware resource for DNNs, and modify the Halide compiler to generate hardware implementations for both FPGA and ASIC platforms. Using this system, we are able to evaluate prior designs for DNNs. We find that many dataflows can achieve similar and close-to-optimal energy efficiency, while what is more critical is to properly design the memory hierarchy. Based on these insights, we develop an auto-optimizer that can find the optimal Halide schedules for generating efficient hardware implementations.

References

[1] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, 2006.

[2] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *2011 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2011.

[3] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[4] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *47th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2014.

[5] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[6] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.

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

[8] 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," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462176

[9] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2014.

[10] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[11] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[12] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. 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.

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

[14] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *23rd IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[15] H. Kwon, A. Samajdar, and T. 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*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173176

[16] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062207

[17] M. Song, J. Zhang, H. Chen, and T. Li, "Towards efficient microarchitectural design for accelerating unsupervised GAN-based deep learning," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[19] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[20] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[21] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[22] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[23] C. Zhang and V. 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*, ser. FPGA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021727

[24] J. H. Ko, B. Mudassar, T. Na, and S. 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*, ser. DAC '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062228

[25] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

[26] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[27] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial CNN accelerators," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016.

[28] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2017.

[29] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. 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)*, Aug 2016.

[30] 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)*, Oct 2016.

[31] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A systematic approach to blocking convolutional neural networks," *arXiv preprint arXiv:1606.04209*, 2016.

[32] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694364

[33] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435271

[34] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. 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*. ACM, 2016.

[35] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. 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*. ACM, 2016.

[36] 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 Trans. Archit. Code Optim.*, vol. 14, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3107953

[37] N. Muralimanohar, R. Balasubramonian, and N. 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*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.30

[38] Cadence Design Systems, Inc., "Tensilica customizable processor IP," http://ip.cadence.com/ipportfolio/tensilica-ip.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *25th International Conference on Neural Information Processing Systems (NIPS)*, 2012.

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

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *arXiv preprint arXiv:1409.4842*, 2014.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[44] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[45] J. G. Zilly, R. K. Srivastava, J. Koutník, and J. Schmidhuber, "Recurrent highway networks," *CoRR*, vol. abs/1607.03474, 2016. [Online]. Available: http://arxiv.org/abs/1607.03474

[46] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. 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)*, 2016.