

Wire-Aware Architecture and Dataflow for CNN Accelerators

Sumanth Gudaparthi
University of Utah
Salt Lake City, Utah
sgudapar@cs.utah.edu

Surya Narayanan
University of Utah
Salt Lake City, Utah
surya@cs.utah.edu

Rajeev Balasubramonian
University of Utah
Salt Lake City, Utah
rajeev@cs.utah.edu

Edouard Giacomin
University of Utah
Salt Lake City, Utah
edouard.giacomin@utah.edu

Hari Kambalasubramanyam
University of Utah
Salt Lake City, Utah
hari.kambalasubramanyam@utah.edu

Pierre-Emmanuel Gaillardon
University of Utah
Salt Lake City, Utah
pierre-emmanuel.gaillardon@utah.edu

ABSTRACT

In spite of several recent advancements, data movement in modern CNN accelerators remains a significant bottleneck. Architectures like Eyeriss implement large scratchpads within individual processing elements, while architectures like TPU v1 implement large systolic arrays and large monolithic caches. Several data movements in these prior works are therefore across long wires, and account for much of the energy consumption. In this work, we design a new wire-aware CNN accelerator, WAX, that employs a deep and distributed memory hierarchy, thus enabling data movement over short wires in the common case. An array of computational units, each with a small set of registers, is placed adjacent to a subarray of a large cache to form a single tile. Shift operations among these registers allow for high reuse with little wire traversal overhead. This approach optimizes the common case, where register fetches and access to a few-kilobyte buffer can be performed at very low cost. Operations beyond the tile require traversal over the cache's H-tree interconnect, but represent the uncommon case. For high reuse of operands, we introduce a family of new data mappings and dataflows. The best dataflow, WAXFlow-3, achieves a 2× improvement in performance and a 2.6-4.4× reduction in energy, relative to Eyeriss. As more WAX tiles are added, performance scales well until 128 tiles.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks.**

KEYWORDS

CNN, DNN, neural networks, accelerator, near memory

ACM Reference Format:

Sumanth Gudaparthi, Surya Narayanan, Rajeev Balasubramonian, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. 2019. Wire-Aware Architecture and Dataflow for CNN Accelerators. In

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 ACM 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.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6938-1/19/10...\$15.00

<https://doi.org/10.1145/3352460.3358316>

The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3352460.3358316>

1 INTRODUCTION

Several neural network accelerators have emerged in recent years, e.g., [9, 11, 12, 28, 38, 39]. Many of these accelerators expend significant energy fetching operands from various levels of the memory hierarchy. For example, the Eyeriss architecture and its row-stationary dataflow require non-trivial storage for scratchpads and registers per processing element (PE) to maximize reuse [11]. Therefore, the many intra-PE and inter-PE accesses in Eyeriss require data movement across large register files. Many accelerators also access large monolithic buffers/caches as the next level of their hierarchy, e.g., Eyeriss has a 108 KB global buffer, while Google TPU v1 has a 24 MB input buffer [24]. Both architectures also implement a large grid of systolic PEs, further increasing the wire lengths between cached data and the many PEs. In this paper, we re-visit the design of PEs and memory hierarchy for CNN accelerators, with a focus on reducing these long and frequently traversed wire lengths.

It is well known that data movement is orders of magnitude more expensive than the cost of compute. At 28 nm, a 64-bit floating-point multiply-add consumes 20 pJ; transmitting the corresponding operand bits across the chip length consumes 15× more; accessing a 1 MB cache consumes 50× more; and fetching those bits from off-chip LPDDR consumes 500× more [26, 27, 32]. Since this initial comparison from 2011, DNN accelerators have switched to using 8-bit fixed-point [24] or 16-bit flexpoint [29] arithmetic, which helps lower compute energy by an order of magnitude [24]. Recently, technologies like HBM have helped reduce memory energy per bit by an order of magnitude [36]. Meanwhile, on-chip wiring and on-chip caches have not benefited much from technology steps [6, 21]. In response to the relative shift in bottlenecks, this work targets low on-chip wire traversal.

We create a new wire aware accelerator WAX, that implements a deep and distributed memory hierarchy to favor short wires. Such an approach has also been leveraged in the first designs from the startup, Graphcore [18]. We implement an array of PEs beside each cache subarray. Each PE is assigned less than a handful of registers. The registers have shift capabilities to implement an efficient version of systolic dataflow. Each PE therefore uses minimal wiring

to access its few registers, its adjacent register, and a small (few-KB) cache subarray. Data movement within this basic WAX tile has thus been kept to a minimum. Large layers of CNNs map to several tiles and aggregate the results produced by each tile. To increase the computational power of the WAX tile, we introduce a novel family of dataflows that perform a large slice of computation with high reuse and with data movement largely confined within a tile. We explore how the dataflows can be adapted to reduce problematic partial sum updates in the subarray. While this reduces reuse for other data structures and requires more adders, we show that the trade-off is worthwhile.

Our analysis shows that the additional WAX components contribute 46% of the tile area. Our best design reduces energy by 2.6–4.4 \times , relative to Eyeriss. WAX also consumes less area and hence less clock distribution power by eliminating the many large register files in Eyeriss. We show that our best dataflow (WAXFlow-3) enables higher overlap of computation with operand loading into subarrays – this leads to higher compute utilization and throughput than Eyeriss. As we scale the design to several tiles, the computational throughput increases until 128 tiles. A WAX tile can therefore form the basis for both, an energy-efficient edge device and a throughput/latency-oriented server.

2 BACKGROUND

We first describe two designs, one commercial and one academic, that highlight the extent of data movement in state-of-the-art architectures.

Eyeriss

Eyeriss [11] uses a monolithic grid of processing elements (PEs). Each PE has scratchpads and register files that together store about half a kilo-byte of operands. The filter scratchpad has 224 entries and is implemented in SRAM, while the partial sums and activations are stored in 24- and 12-entry register files respectively. Each PE performs operations for an entire row before passing partial results to neighboring PEs (a “row-stationary” dataflow). To increase reuse, the PE combines a set of input features with a number of different kernels to produce partial sums for many output features. The grid of PEs is fed with data from a monolithic 108 KB global buffer, and from off-chip DRAM.

In Eyeriss, the grid of PEs occupies nearly 80% of the chip area. One of the reasons for the large area of the PEs is that 61% of PE area is used for the half-kilobyte scratchpad and register files per PE. As a result, the systolic dataflow among PEs requires traversal over wires that span relatively long distances. The mid-size register files per PE are also problematic as they lead to long wires with high load.

While the grid of PEs and the row-stationary dataflow of Eyeriss are tailored for convolutions, such accelerators are also expected to execute fully-connected classifier layers of CNNs. Such layers exhibit limited reuse, but still pay the price of long wires that span many PEs and large scratchpads/register files.

Google TPU

The Google TPU v1 is a commercial example of a large-scale inference processor, capable of 92 TOPs peak throughput while operating at 40 W. The TPU core is composed of a 256 \times 256 grid of 8-bit MAC units. Operands move between the MACs using a

systolic dataflow. This allows, for example, an input operand to be multiplied by the many weights in one convolutional kernel, and by the weights in multiple kernels. Each MAC is fed by a few registers. While the MACs are working on one computation, the registers are pre-loaded with operands required by the next computation (a form of double-buffering). Weights are fetched from off-chip memory (DDR for TPU v1 and HBM for TPU v2) into a FIFO. Input/output feature maps are stored in a large 24MB buffer.

What is notable in the TPU design is that there is a monolithic grid of MACs that occupies 24% of the chip’s area [24]. Further, all input and output feature maps are fetched from a 24 MB cache, which too occupies 29% of the chip’s area. As a result, most operands must traverse the length or width of the large grid of MACs, as well as navigate a large H-Tree within the cache.

Wire Traversal

Our proposed approach is motivated by the premise that short-wire traversal is far more efficient than long-wire traversal. We quantify that premise here.

While a large scratchpad or register file in an Eyeriss PE promotes a high degree of reuse, it also increases the cost of every scratchpad/register access, it increases the distance to an adjacent PE, and it increases the distance to the global buffer. Figure 1c shows the breakdown of energy in the baseline Eyeriss while executing the CONV1 layer of AlexNet [30]. Nearly 43% of the total energy of Eyeriss is consumed by scratchpads and register files. Our hypothesis is that less storage per PE helps shorten distances and reduce data movement energy, especially if efficient dataflows can be constructed for this new hierarchy. We also implement a deeper hierarchy where a few kilo-bytes of the global buffer are adjacent to the PEs, while the rest of the global buffer is one or more hops away.

To understand the relative energy for these various structures and wire lengths, we summarize some of the key data points here. First consider the energy difference between a 54 KB global buffer (corresponding to an 8-bit version of the Eyeriss architecture) and a 6 KB subarray employed in the proposed WAX architecture: according to CACTI 6.5 [34] at 28 nm, the smaller subarray consumes 1.4 \times less energy.

Similarly, consider the energy gap between a 224-byte SRAM scratchpad (similar to the filter scratchpad in Eyeriss) and register files with fewer than 4 entries: the register access consumes orders of magnitude less energy (see Figure 1). Scratchpad energy is estimated by CACTI and register energy methodology is detailed below.

We use the following methodology to implement and synthesize varying wiring load configurations using a 28 nm Fully Depleted Silicon On Insulator (FDSOI) technology node. Verilog code has been written to model the behavior of varying size registers and then synthesized using Synopsys Design Compiler, a Low Leakage (LL) library, and clock frequency of 200 MHz. We used Innovus to perform the backend flow; the register netlists were then back-annotated with the SPEF parasitics file obtained from Innovus. This is done to get accurate post layout metrics by taking the parasitics into account through SPICE simulations. Our register file energy estimates are similar to those reported by Balfour et al. [7].

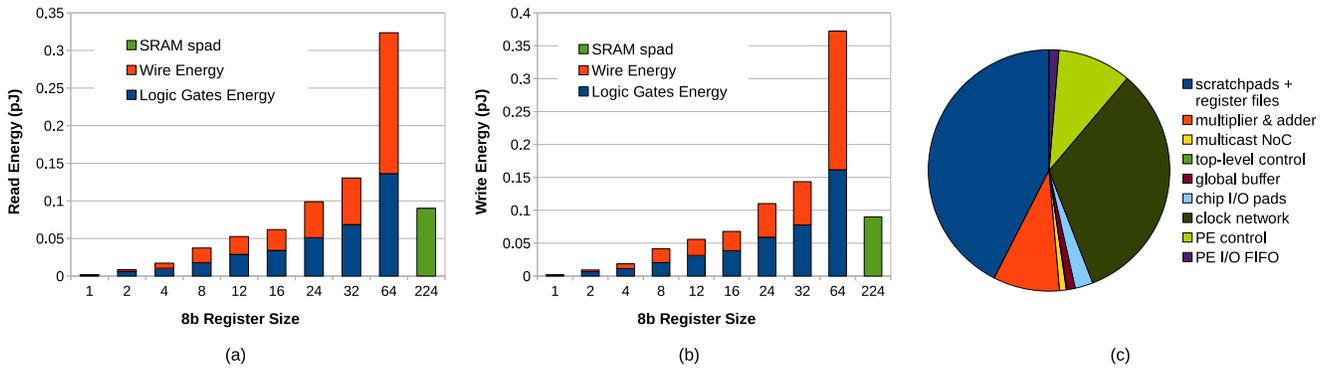


Figure 1: Read (a) and Write (b) energy for register files and a 224-entry SRAM scratchpad. (c) Eyeriss energy breakdown.

Figures 1a and 1b show the read and write energy consumed by an 8-bit register file with varied register sizes. The energy consumed by the register file increases more than linearly with the number of registers. For the single register, most of the energy is consumed by the logic gates themselves, as the wires are relatively small. For larger register files, the overall energy increases due to two factors: (i) the increasing number of rows leads to more complex read and write decoders, (ii) more flip-flops share the same signals (such as the write or address signals), leading to higher load and larger parasitics.

These data points therefore serve as a rough guideline for the design of a wire-aware accelerator. To the greatest extent possible, we want to (i) replace 54 KB buffer accesses with 6 KB buffer accesses (1.4× energy reduction), (ii) replace 224-byte scratchpad access with single register access (46× energy reduction), and (iii) replace 12- and 24-entry register file access with single register access (28× and 51× energy reduction).

Another key consideration is the power for the clock tree. As seen in Figure 1c, the clock tree accounts for 33% of total power in Eyeriss. In architectures like Eyeriss and the Google TPU v1, where the SRAM buffer and the systolic array are separate, the clock tree must primarily span the systolic array. If we employ a tiled architecture with interspersed SRAM and compute all across the chip, it is possible that the larger clock tree may offset the advantage of lower/localized data movement. A wire-aware accelerator must therefore also consider the impact on area and the clock distribution network. By modeling the layout and the clock tree, we show that the proposed accelerator consumes less clock power than the baseline Eyeriss (Section 4). This is primarily achieved by eliminating the large register files per PE.

3 PROPOSED ARCHITECTURE

In this work, we only focus on inference and 8-bit operands, similar to the Google TPU v1. The basic ideas apply to other operand sizes as well as to the forward/backward passes in training.

3.1 A Wire-Aware Accelerator (WAX)

Our goal is to reduce data movement in the common case by designing a new memory hierarchy that is deeper and distributed, and that achieves high data reuse while requiring low storage per

PE. Figure 2 shows an overview of the proposed WAX architecture. Conventional large caches are typically partitioned into several subarrays (a few KB in size), connected with an H-Tree network. We propose placing a neural array next to each subarray, forming a single WAX tile.

The neural array has three wide registers, W , A , and P , that maintain weights, input activations, and partial sums respectively. These registers are as wide as a subarray row. One of the registers, A , has shifting capabilities. These three registers provide input operands to an array of MACs, with the computation results going to P or directly back to the subarray.

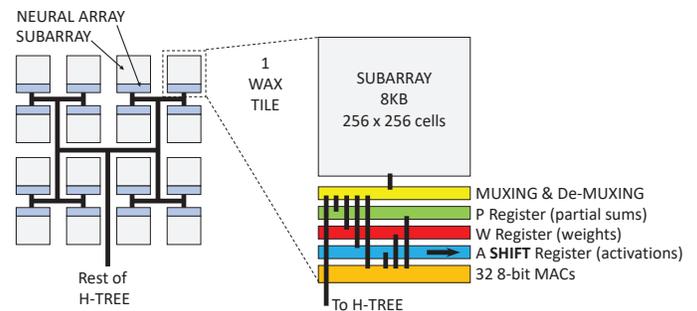


Figure 2: WAX architecture overview.

This design has two key features. First, reuse and systolic dataflow are achieved by using a shift register. This ensures that operands are moving over very short wires. These distances are further kept short because each “processing element” or PE in our design has only one MAC and three 8-bit registers, which is much more compact than a PE in Eyeriss or TPU.

Second, the next level of the hierarchy is an adjacent subarray of size say 8 KB. This is a much cheaper access than TPU or Eyeriss where large H-Trees are traversed for reads/writes to the 24 MB or 108 KB buffer respectively.

Both of these features drive home our central principle: implement a deep hierarchy so that the common case is not impeded by data movement across large data structures. Few registers per MAC enable low-energy dataflow between MACs. But since each MAC has fewer operands, it must fetch operands from the next

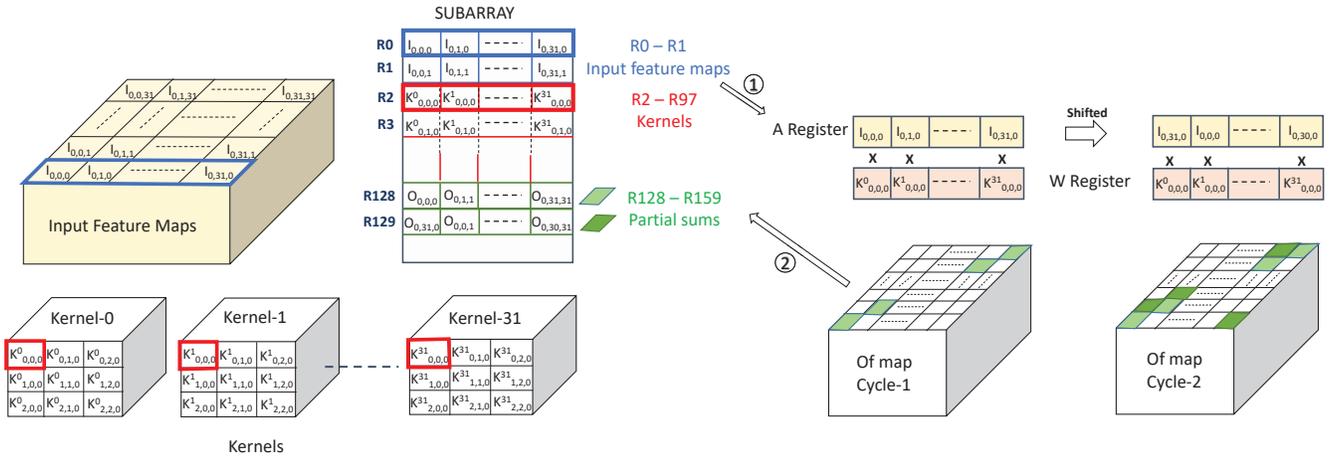


Figure 3: Data mapping and computation order in WAXFlow-1. ① An Activation and a Kernel row are read from the subarray into the register file. ② Partial sums are written back to the subarray.

level of the hierarchy more often (than say Eyeriss or TPU). This is why it is vital that the next level of the hierarchy be a compact 8 KB subarray. When dealing with large network layers, the computation must be spread across multiple subarrays, followed by an aggregation step that uses the smaller branches of the H-Tree. Thus, in the common case, each computation and data movement is localized and unimpeded by chip resources working on other computations.

The accelerator resembles a large cache, but with MAC units scattered across all subarrays. A single WAX tile may be composed of an 8 KB subarray, with 32 8-bit MACs, and 3 8-bit registers per MAC. We assume that the SRAM subarray has a single read/write port. Subarray read, MAC, and subarray write take a cycle each and are pipelined. In addition to the overhead of the MACs and $W/A/P$ registers, muxing/de-muxing is required at the H-Tree/subarray interface. The area overhead of this tile is quantified in Section 5.

3.2 Efficient Dataflow for WAX (WAXFlow 1)

A WAX tile essentially represents a small unit of neural network computation, where an array of 32 MACs can be fed with data from an 8 KB subarray. We’ll first discuss how data can be mapped to a WAX tile and how computation can be structured to maximize reuse and efficiency within a tile. We will then discuss how a large neural network layer may be partitioned across multiple tiles.

We describe our proposed dataflow, *WAXFlow 1*, by walking through a simple example of a convolutional layer. The steps are also explained in the accompanying Figure 3. The example convolutional layer has 32 input feature maps, each of size 32×32 ; we assume 32 kernels, each of size $3 \times 3 \times 32$. An 8 KB subarray can have 256 rows, each with 32 8-bit operands.

We first fill the subarray with 1 row of input feature maps, as shown by the blue box in row R0 in Figure 3. We then place the first element of 32 kernels (shown by the red boxes) in row R2 of the subarray. Similarly, other elements of the kernel are placed in other rows of the subarray. Finally, some rows of the subarray are used for partial sums.

Now consider the following computation order. In the first step, the first row of input feature maps (R0) is read into the activation register A and the first row of kernel weights (R2) is read into weight register W . The pair-wise multiplications of the A and W registers produce partial sums for the first green-shaded diagonal of the output feature maps. This is written into row R128 of the subarray. We refer to this 1-cycle operation as a *Diagonal Pass*.

The activation register then performs a right-shift (with wraparound). Another pair-wise multiplication of A and W is performed to yield the next right-shifted diagonal of partial sums. This process is repeated for a total of 32 times (for this example), yielding partial sums for the entire top slice of the output feature maps, and saved in rows R128-159. Note that a row of weights and a row of input activations read from the subarray are reused 32 times; this is enabled with a relatively inexpensive right-shift within the A register. These 32 cycles represent one *WAXFlow slice*.

To perform the next slice, a new row of kernel weights (R3) is read into the W register. The A register is unchanged, i.e., it exhibits more reuse. The computations performed in this next slice continue to add to the same green partial sums computed in the first slice. The computation thus proceeds one slice at a time, gradually bringing in rows of activations and rows of weights into A and W registers to produce the remaining partial sums for the top slice of the output feature maps.

This initial description shows that a basic slice in WAXFlow and its subsequent slices exhibit very high reuse of both activations and weights. Each slice is performed with a single read and write out of a small adjacent subarray and multiple shift operations – both engage very short wires. Subsequent slices are performed by reading 1 or 2 additional rows out of the subarray. The only drawback here is that partial sums are being read and written from/to the subarray and not a register. Even though the subarray is small, this is a non-trivial overhead that we alleviate later with alternative dataflows (Section 3.3). The subarray access energy per byte is comparable to Eyeriss’s partial sum scratchpad energy to access one byte of data. Thus, WAX and WAXFlow-1 have been tailored to

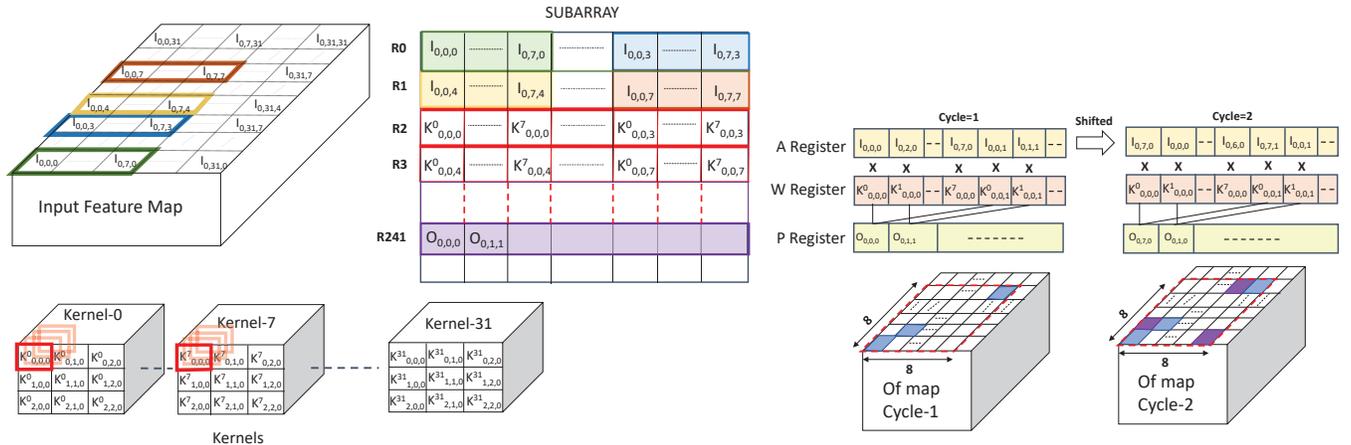


Figure 4: Data mapping and computation order in WAXFlow-2.

reduce overheads for the common case (the activation and kernel reads), while achieving moderate overheads for the less common case (the partial sum updates).

The rest of this sub-section walks through the rest of this example, defining each step of the computation and the required cycles for our example. To recap, a diagonal pass takes a single cycle and a slice pass takes 32 cycles. By the end of a slice pass, all neurons in the top layer of the output feature maps have performed 1 of their 288 multiplications – note that each kernel has size $3 \times 3 \times 32$. Next, row R3 is read into the W register and a slice pass is performed, followed by a slice pass on row R4. At this point, after 96 cycles, the X -dimension of the kernels have been processed and reuse of row R0 has been maximized. We refer to this as an *X-Accumulate Pass*. Row R0 can now be discarded.

The next row of input feature maps is loaded from a remote tile to row R1 of the subarray to perform the next operation. The loading of input feature maps at the start of every *X-Accumulate Pass* cannot be overlapped with computation because the subarray is busy performing partial sum writes to the subarray in every cycle. We overcome this drawback with better dataflows in the next sub-section. Row R1 is moved into the A register, i.e., the first row of the second input feature map. We also bring R5 into the W register, representing kernel element K_{001} of all 32 kernels. This is the start of a second *X-Accumulate Pass*. Such *X-Accumulate Passes* are repeated 32 times, dealing with the entire top slice of the input feature maps. These 32 *X-Accumulate Passes* are called a single *Z-Accumulate Pass*.

A *Z-Accumulate Pass* has consumed $96 \times 32 = 3K$ cycles and performed 96 of the 288 MACs required for each top slice output neuron. Similarly, 192 other MACs have to be performed for each top slice output neuron; this is done by engaging two more tiles in parallel. In other words, three *Z-Accumulate Passes* are performed in parallel on three tiles; those partial sums are then accumulated in a *Y-Accumulate Pass* to yield the final output neurons for the top slice. The H -tree is used to move partial sums from one tile to its adjacent tile; given the 64-bit link into a tile, this accumulation takes 128 cycles. For larger kernels, many parallel *Z-Accumulate Passes* are required and the *Y-Accumulate Pass* would involve a tree of

reductions. In this example involving three tiles, only two sequential *Y-Accumulate Passes* are required. To get ready for the next set of computations in this layer, the output neurons are copied to an *Output Tile*. We have thus processed an entire top slice of output neurons in 3,488 cycles, involving 3 parallel *Z-Accumulate Passes*, 2 sequential *Y-Accumulate Passes*, input loading, and 1 output copy. To compute the next slice of output neurons, only the rows of input feature maps have to be replaced with a new set of input feature maps. These new input feature maps are fetched from the *Output Tile* that was produced by the previous layer. The weights remain in place and exhibit reuse within the subarray. In our example, processing all 30 slices of the output feature map takes about 101K cycles.

3.3 Increasing Reuse for Partial Sums

WAXFlow-1

The WAXFlow-1 algorithm described in the previous sub-section reuses a row of kernel weights for 32 consecutive cycles. The same weights are reused again every 3.4K cycles. A row of input activations is reused for 96 consecutive cycles before it is discarded. We thus see high reuse for activations and weights. Meanwhile, each partial sum is re-visited once every 32 cycles (96 updates in 3K cycles).

This means that partial sums are accessed from the subarray every cycle, causing a significant energy overhead. Table 1 shows the number of accesses to the subarray and registers for WAXFlow-1 in one slice (32 cycles). While activations and filter weights together contribute less than 2 subarray accesses, partial sums cause 64 subarray accesses in one slice.

WAXFlow-2

Overall energy is usually reduced when accesses to these data structures are balanced. For example, if an alternative dataflow can reduce pass accesses by $4\times$ at the cost of increasing activation and filter accesses by $4\times$, that can result in overall fewer subarray accesses. That is precisely the goal of a new dataflow, WAXFlow-2.

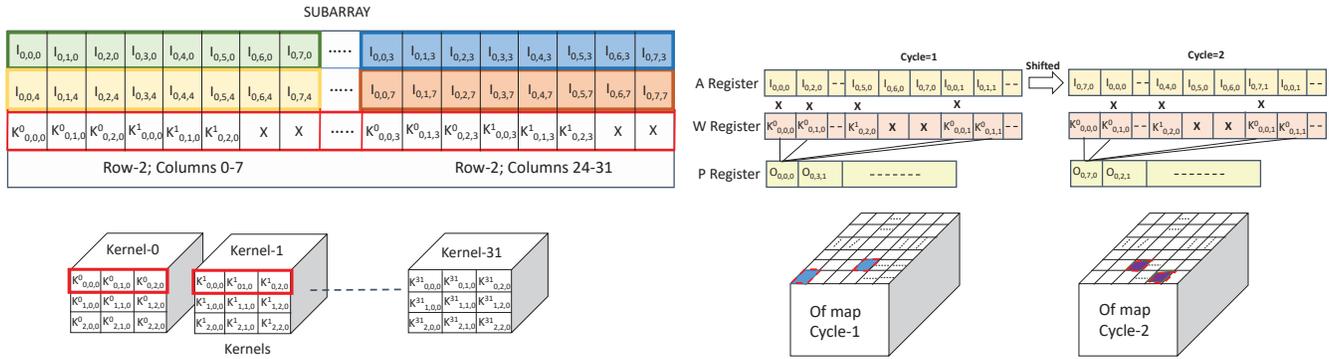


Figure 5: Data mapping and computation order in WAXFlow-3.

First, we modify the data mapping – see Figure 4. Each row of the subarray is split into P partitions. Each partition has input feature maps corresponding to different channels. The wraparound shift operation in the activation register A is performed locally for each partition. As a result, a WAXFlow-2 slice only consumes $32/P$ cycles. With a design space exploration, we find that energy is minimized with $P = 4$.

We now walk through the example in Figure 4. The first row of activations, R_0 , contains the first 8 ifmap elements from four channels. The first filter row, R_2 , is also partitioned into four channels, as shown in the figure. After the pair-wise multiplications of R_0 and R_2 in the first cycle, the results of the 0th, 8th, 16th, and 24th multiplier are added together, since they all contribute to the very first element of the output feature map. Similarly, the 1st, 9th, 17th, and 25th multiplier results are added, yielding the next element of the ofmap. Thus, this cycle produces partial sums for the eight diagonal elements (shown in blue in Figure 4) of the top slice. These 8 elements are saved in the P register (but not written back to the subarray).

In the next cycle, the A register first performs a shift. Note that the shift is performed within each channel, so the wraparound happens for every eight elements, as shown in Figure 4. As in the first cycle, the results of the multiplications are added to produce eight new partial sums that are stored in different entries in the P register.

After 4 cycles, the P registers contain 32 partial sums that can now be written into a row of the subarray. Note that subarray write is performed only in cycles after psum aggregation has completed. With a new data mapping and by introducing eight 4-input adders, we have reduced the psum read/write activity by $4\times$. After 8 cycles, the channels in the A registers have undergone a full shift and we are ready to load new rows into the A and P registers. Thus, the subarray reads for activations and filters have increased by $4\times$. As summarized in Table 1, this is a worthwhile trade-off. The number of MAC operations per subarray access has increased from 15 in WAXFlow-1 to 45 in WAXFlow-2 (Table 1). On the other hand, due to new accesses to the P register, the MAC operations per register has decreased. Since subarray accesses consume much more energy than register accesses (Table 4), this results in an overall significant energy reduction.

In WAXFlow-1, the subarray is busy dealing with partial sums in every cycle. Therefore, some of the data movement – fetching the next row of ifmaps, performing the Y-Accumulate Pass, output copy – cannot be overlapped with MAC computations. However, in WAXFlow-2, the partial sums result in subarray accesses only once every 4 cycles. Because of these subarray idle cycles, some of the other data movement can be overlapped with slice computation. Thus, WAXFlow-2 is better in terms of both latency and energy.

WAXFlow-3

We saw that WAXFlow-2 introduced a few adders so that some intra-cycle aggregation can be performed, thus reducing the number of psum updates in the subarray. We now try to further that opportunity so that psum accesses in the subarray can be further reduced.

Figure 5 shows the new data mapping and computation structure for WAXFlow-3. As with WAXFlow-2, the subarrays are split into 4 partitions. The ifmap is also organized the same way in row R_0 . WAXFlow-2 filled a partition in a kernel row with elements from 8 different kernels (Figure 4); there was therefore no opportunity to aggregate within a partition. But for WAXFlow-3, a row of weights from a single kernel is placed together in one kernel row partition. In our example, a kernel row only has three elements; therefore, there is room to place three elements from two kernels, with the last bytes of the partition left empty.

With the above data mapping, the multiplications performed in a cycle first undergo an intra-partition aggregation, followed by an inter-partition aggregation. Thus, a single cycle only produces 2 partial sums. It takes 16 cycles to fully populate the P register, after which it is written into the subarray. With this approach, the partial sums contribute only 2 subarray reads and 2 subarray writes every 32 cycles (Table 1). The number of activation and filter accesses are unchanged; *the key trade-off is that we have introduced another layer of adders to enable more partial-sum increments before a subarray write* (see the adder details in Figure 7). As seen in Table 1, there is another significant jump in MAC operations per subarray access, and a minor increase in MAC operations per register access.

One other trade-off in WAXFlow-3 is that because two of the elements in every kernel partition are empty, the MACs are only 75% utilized. This is because the kernel dimensions are $3\times 3\times 32$. If the row size is a multiple of 3, the kernel partition need not have

Hierarchy		WAXFlow 1	WAXFlow 2	WAXFlow 3
Subarray	Activation	0.33R + 0.33W	1.33R + 1.33W	1.33R + 1.33W
	Filter weights	1R	4R	4R
	Partial sums	32R + 32W	8R + 8W	2R + 2W
	MAC/subarray access	15.6	45.17	96
	Subarray Energy (pJ)	136.75	47.21	22.22
Register File	Activation	32R + 32.33W	32R + 33.33W	32R + 33.33W
	Filter weights	32R + 1W	32R + 4W	32R + 4W
	Partial sums	–	8R + 8W	2R + 2W
	MAC/Register file access	10.52	8.72	9.76
	Register file Energy (pJ)	4.6	5.54	4.97
Total Energy(pJ)		141.35	52.75	27.19

Table 1: Number of accesses for subarray and register file for different WAX dataflows when executed for 32 cycles.

empty slots. Since a kernel dimension of 3 is common in DNNs, we modify our WAX tile configuration so it is in tune with WAXFlow-3 and the common case in DNNs. We adjust the width of a tile from 32 to 24, i.e., a subarray row is 24 bytes, the subarray capacity is 6 KB, the tile has 24 MACs, etc. The design details for this model are also shown in Figure 7. Feature map size has no effect on the MAC utilization. Depending on the feature map size, we either split a feature map row into multiple rows of the subarray, or activations from multiple rows of the feature map are placed in one row of subarray. There is an effect on performance for certain kernel dimensions even after the adjusted tile size. Only WaxFlow-3 imposes constraints that may occasionally lead to upto 33% compute under-utilization in CONV layers where the kernel X-dimension is of the form $3N+2$. Other convolutional layers and all FC layers exhibit 100% utilization, except in the very last accumulate pass where there may not be enough computation left.

Note again that the many idle cycles for the subarray in WAXFlow-3 allow further overlap of data movement and computation. The energy numbers in Table 1 emphasize the benefits in upgrading the dataflow from WAXFlow-1 to WAXFlow-3. In all the three dataflows, filter weights once loaded remain stationary in the subarray until all of them are fully exploited. In case of activations, the subarray is only used to buffer the next row of activations fetched from the remote subarray. Hence in Table 1, the number of remote subarray accesses for activations is 0.33R for WAXFlow-1, and 1.33R for WAXFlow-2 and WAXFlow-3.

In the baseline Eyeriss architecture, partial sums are written to the scratchpad after every multiplication operation, i.e., every MAC operation requires one read and one write for the partial sum. Meanwhile, in WAXFlow-2 and WAXFlow-3, a set of adders is used to accumulate multiple multiplications before updating the partial sum. At 100% utilization, WAXFlow-2 reduces the number of partial sum register accesses by 4 \times and WAXFlow-3 reduces the number by 12 \times , relative to WAXFlow-1. Scratchpad access energy, as discussed in Section 2, is the dominant energy contributor in Eyeriss, with half the scratchpad energy attributed to partial sum accesses. Thus, by using smaller register files and introducing adders in each tile, we target the number of partial sum updates and the cost of each update.

Fully Connected Dataflow For executing fully connected (FC) layers on WAXFlow-3, a slightly different data mapping is followed.

We disable the shift operation performed by A register so that it emulates a static register file (similar to W/P registers). This is because the nature of FC layers allows for activation reuse but not kernel reuse making the shift operation pointless. Each kernel row in the subarray is comprised of weights corresponding to a particular output neuron, whereas the activation row has inputs corresponding to those weights. In the first cycle, the activation row is fetched and stored in the A register. In the next cycle, the first kernel row is fetched and stored in the W register. Pair-wise multiplications are performed on A and W registers generating 24 psums. As all the kernels in a row (24 in WAXFlow-3) correspond to the same output neuron, the resulting 24 psums can be accumulated into one value and stored in the P register. While the MAC operation is being performed, the next kernel row is prefetched into the W register. An activation row fetched into the A register is reused across all available kernel rows (say N) in the subarray. Once the activation row is utilized across the available kernel rows, we will have psums computed for N output neurons. Multiple subarrays work in parallel to generate the remaining psums for the same N output neurons. This iteration repeats until all the output neurons are computed.

4 METHODOLOGY

For most of this evaluation, we compare the WAX architecture to Eyeriss. For fairness, we attempt iso-resource comparisons as far as possible, where the resource may be area, MACs, or cache capacity.

In order to get accurate area and energy values, we modeled WAX (with 4 banks) and Eyeriss in Verilog, synthesized it using Synopsys Design Compiler and used Innovus for the Place & Route, using a commercial 28 nm FDSOI technology node (typical-typical process corner, 1V, 25C, 10 metal layers). Thus, the results take into account layout effects such as wire length, clock tree synthesis, and parasitics. During floorplanning, we constrained the WAX tile width to be the same as the SRAM subarray in order to have both blocks aligned, as shown in the Figure 6. We ensured that the 192 input pins of the WAX tile are placed on top of the block, to be aligned with the SRAM outputs. As the WAX tile is fully digital, there is not a strong need to perfectly pitch match each column. Since we have not initiated a fabrication effort for the chip, we were not able to gain access to memory compilers from the foundry for the targeted 28nm FDSOI technology node or to the lib/lef/layout

files for the SRAM array. To model the energy and area of SRAM subarrays and the H-tree interconnects, we use CACTI 6.5 [34] at 32 nm, and scale it to 28 nm process. In order to properly account for the layout effects (CTS and wire length mainly), we use the area extracted from CACTI to define blackboxes with routing and placement blockages to account for the SRAM’s area and placement during the backend flow for WAX and Eyeriss. We thus consider the whole area for both architectures and not just the logic part. Since we are not modifying the subarray itself, we anticipate that the relative metrics from CACTI for baseline and proposed are sufficiently accurate. Similar to the Eyeriss analysis, we assume a low leakage LP process. The layout of Eyeriss and WAX are shown in Figure 6. WAX occupies a significantly lower area than Eyeriss; this is primarily because Eyeriss has large register files per PE (see area summarized in Table 2). A side-effect is that the clock distribution power for WAX is lower than that for Eyeriss even though compute is not localized to a region of the chip. The clock tree synthesis performed with Innovus added 25-30% to total chip power; the clock tree in WAX and Eyeriss account for 8 mW and 27 mW. Although Eyeriss’ area is 1.6× higher than WAX area, the clock network has to travel to larger register files in Eyeriss resulting in a larger clock network.

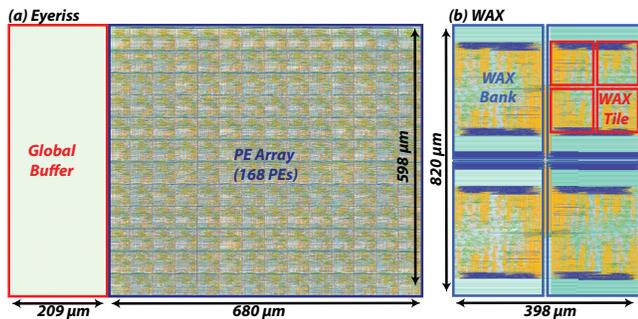


Figure 6: Layouts of (a) Eyeriss and (b) WAX

To evaluate the performance and energy of Eyeriss and WAX, and their respective dataflows, we developed a simulator that captures the latencies, resource contention, and access counts for the various components in the architectures. To get total energy, the access counts were multiplied by the energy per component derived from the circuit models. We assumed a low-power DRAM interface with 4 pJ/bit, similar to baseline HBM [36].

While the original Eyeriss work assumed 16-bit operands, we consider an 8-bit version of Eyeriss in our analysis. All the Eyeriss resources (registers, bus widths, buffer) are accordingly scaled down and summarized in Table 2. Note that the global buffer is now 54 KB and the register storage per PE is 260 bytes.

The overall Eyeriss chip is modeled to have on-chip storage (global buffer + scratchpads) of 96.7 KB, 168 MACs, and a 72-bit bus connecting the PE array and the global buffer. For an iso-resource comparison, we model WAX with 96 KB of SRAM storage, 168 MACs, and bus width of 72 is shared across the banks. The 96 KB SRAM in WAX is organized into 4 banks, and each bank has four 6 KB subarrays. A WAX tile is made up of one 6 KB subarray, an array of 24 MACs, and three 1-byte registers (A , W , and P). We

PE	
Number of PEs	168
Arithmetic precision	8-bit fixed point
GLB	
SRAM Memory Size	54KB
Bus Width	72 (Feature map: 32 Filter weight: 32 Partial sum: 8)
Scratchpads/PE	
Feature Map	12 x 8-b ($386 \mu m^2$)
Filter Weight	224 x 8-b ($524 \mu m^2$)
Partial Sum	24 x 8-b ($759 \mu m^2$)
Total spad size (168 PEs)	42.65 KB
Total area	$0.53 mm^2$

Table 2: Eyeriss reconfigured parameters.

WAX Architecture	
Number of Banks	4 (16 subarrays)
Subarrays with MAC units	7
Subarrays used as Output Tile (inactive MAC units)	9
WAX MAC Configuration	
Activation register	1 x 8-bit
Filter weight register	1 x 8-bit
Partial sum register	1 x 8-bit
Total area	$0.318 mm^2$

Table 3: WAX parameters.

assume 16-b fixed-point adders with output truncated to 8b. After place and route, we estimated that the MAC/registers/control added to each tile account for 46% of the tile area. While a significant overhead, the overall WAX chip area is 1.6× lower than that of Eyeriss. Seven such WAX tiles are implemented (totaling 168 MACs), and the remaining nine 6 KB subarrays are used as Output Tiles to store the output neurons of a CNN layer. The Output Tile is also used to store the partial sums, and prefetch the weights, before loading them to the individual subarrays. For iso-resource analysis, we assume both the architectures run at 200 MHz. The above WAX parameters are summarized in Table 3.

Each cycle, we assume that 72 bits of data can be loaded from off-chip to one of the banks in WAX. The 72-bit H-tree splits so that only an 18-bit bus feeds each subarray in a bank. We introduce additional mux-ing at this split point so that data received from an adjacent subarray can be steered either to the central controller or to the other adjacent subarray (to implement subarray-to-subarray transfers). At a time, 4 24B rows can be loaded into 4 subarrays in 11 cycles. Moving a row of data from one subarray to the adjacent subarray also takes 11 cycles. The proposed architecture considers no interconnect between individual banks. Hence, to fetch data from the output tile, it takes 1 cycle to read the data to the central controller and 1 more cycle to write it back to the subarray.

As workloads, we execute three popular state-of-the-art CNNs: VGG-16 [41], ResNet-34 [20], and MobileNet [22]. VGG-16 is a 16 layer deep neural network with 13 convolution layers and 3 fully

connected layers. ResNet-34 is a 34 layer deep neural network with 33 convolution layers and 1 fully connected network. MobileNet is a depthwise separable convolution architecture with depthwise and pointwise layers. Counting depthwise and pointwise as separate layers, MobileNet has 28 layers.

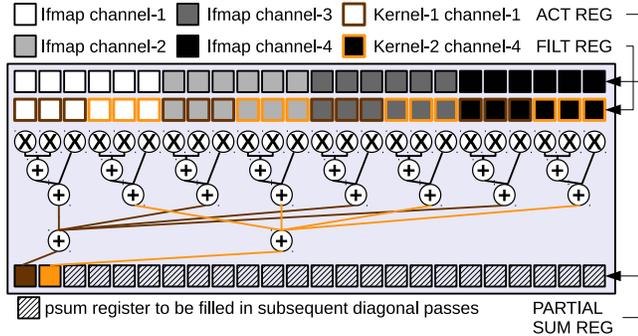


Figure 7: The peripheral logic components of WAX. Color of the box denotes the channel and color of the border denotes the kernel. As explained in Section 3, partial-sums corresponding to the same kernel but different channels are accumulated together to get two partial-sums (brown and yellow).

5 RESULTS

The data in Table 1 has already highlighted the clear benefits of WAXFlow-3 over WAXFlow-1 and -2. Therefore, all results in this section will only focus on WAXFlow-3.

Performance analysis

We first analyze the performance of WAX, relative to Eyeriss. Figure 8b shows the time for each convolutional layer in WAX, while Figure 8a shows time normalized against Eyeriss. To show behavior across convolution layers, this figure includes a breakdown for all layers of VGG16.

Since we are comparing iso-resource configurations, both Eyeriss and WAX are capable of roughly the same peak throughput. Therefore, all performance differences are caused by under-utilization because of how computations map to PEs or because of time to load various structures. We observe that the latter cause is dominant. In Eyeriss, data movement and computations in PEs cannot be overlapped; it therefore spends a non-trivial amount of time fetching kernels and feature maps to the scratchpads before the MACs can execute; it also must move partial sums between PEs and GLB after every processing pass.

On the other hand, with the WAXFlow-3 dataflow introduced in Section 3.3, WAX spends a few consecutive cycles where the MACs read/write only the registers and do not read/write the subarray. This provides an opportunity to load the next rows of activations or weights in the subarray while the MACs are executing. The ability of WAXFlow to leave the subarray idle every few cycles is therefore key to a better overlap of computation and data loading. Across all the layers in VGG16, we see that WAX requires half the time required by Eyeriss. The breakdown in Figure 8c shows that the data movement for partial-sum accumulation in WAX cannot be completely hidden and increases for later layers. While WAXFlow

is 2× faster than Eyeriss on VGG16 and ResNet, it is 3× faster on MobileNet (not shown in the figure). This is primarily because of use of 1×1 filters that exhibit lower reuse and make GLB fetches more of a bottleneck. This is also an example where WAXFlow-3 provides no advantage over WAXFlow-2 because of the filter dimensions. For ResNet-34 and MobileNet, WAX gives a throughput of 58 and 42.6 TOPS and Eyeriss gives a throughput of 24.3 and 11.2 TOPS.

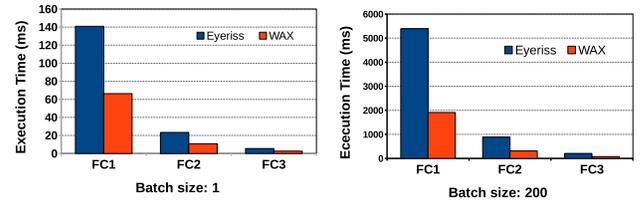


Figure 9: Execution time comparison for Eyeriss and WAX for each fully connected layer in VGG16 at batch sizes of 1 and 200.

Figure 9 shows the time for each fully connected layer in VGG16 for WAX and Eyeriss for different batch sizes. In both cases, WAX is about 2.8× faster. While both WAX and Eyeriss have the same total bus bandwidth, Eyeriss statically allocates its PE bus bandwidth across ifmaps, weights, and psums. Since fully-connected layers are entirely limited by the bandwidth available for weight transfers, Eyeriss takes longer to move weights into PEs.

Energy analysis

We next compare the energy consumed by WAX and Eyeriss. We conservatively assume worst-case wiring distance for all three registers. Table 4 summarizes the energy consumed by each individual operation in both architectures.

Eyeriss	
Hierarchy	Energy (pJ)
Global Buffer Access (9 Bytes)	3.575
Feature Map Register File (1 Byte)	0.055
Filter Weight SRAM Scratchpad (1 Byte)	0.09
Partial Sum Register File (1 Byte)	0.099
8-bit Multiply and Add	0.046
WAX	
Hierarchy	Energy (pJ)
Remote Sub-Array Access (24 Bytes)	21.805
Local Sub-Array Access (24 Bytes)	2.0825
Register File Access (1 Byte) (Feature Map/ Filter Weight/ Partial Sum)	0.00195
8-bit Multiply and Add	0.046

Table 4: Access energy breakdown in Eyeriss and WAX.

Figure 10 shows a breakdown of where energy is dissipated in WAX and Eyeriss. We see that the scratchpad and register file energy in Eyeriss is dominant (consistent with the energy breakdowns in the original Eyeriss work). On the other hand, local subarray access (SA) is the dominant contributor for WAX. Without

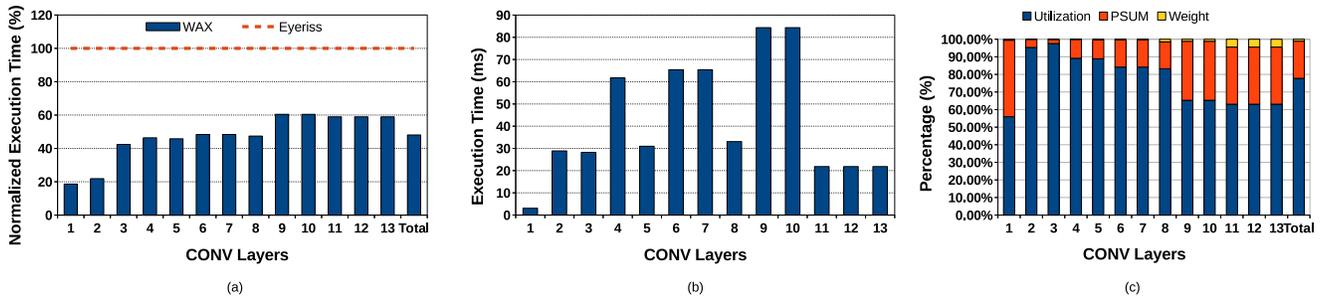


Figure 8: WAX execution time for various convolutional layers in VGG16. (a) Execution time in WAX normalized to Eyeriss, (b) Execution time in WAX (c) Breakdown of execution time in WAX.

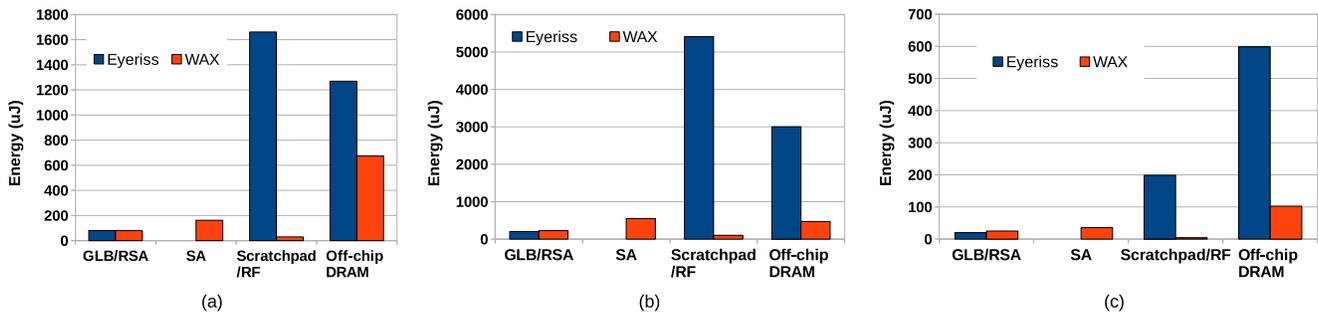


Figure 10: Energy comparison of WAX and Eyeriss for each component on CONV layers of (a) ResNet (b) VGG16 (c) MobileNet. GLB = global buffer; RSA = remote subarray access; SA = local subarray access; RF = register file.

the limited partial-sum updates enabled by WAXFlow-3, this component would have been far greater. Overall, there is a significant benefit from trading more subarray energy for much lower energy in registers and scratchpads. By offering a larger SRAM capacity (in lieu of scratchpads per PE), WAX also reduces the off-chip DRAM accesses. WAX is $2.6\times$ more energy efficient than Eyeriss for ResNet and VGG16, and $4.4\times$ better for MobileNet. Because of its lower reuse, MobileNet has more remote subarray accesses in WAX, but also fewer DRAM accesses in WAX, relative to Eyeriss. The *depthwise* layers of MobileNet yield lower improvements because of their filter dimension and stride, but they contribute less to overall power than the *pointwise* layers. On ResNet and MobileNet, WAX yields a throughput per watt of 18.8 and 12.2 TOPS/W while Eyeriss gives 7.2 and 2.8 TOPS/W.

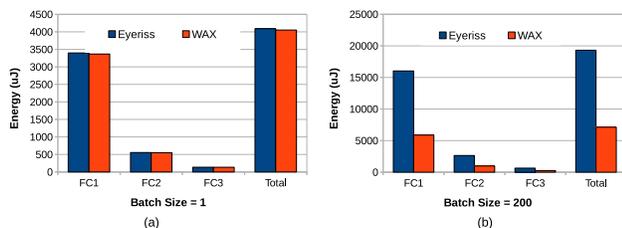


Figure 11: Energy comparison for Eyeriss and WAX for each fully connected layer in VGG-16 at batch sizes of 1 and 200.

Figure 12 shows how each component energy can be broken down across activations, filters, and partial sums, for a representative workload ResNet. The energy breakdown across all three operands in Eyeriss is not balanced, with the partial sum energy being the highest, followed by filter scratchpad energy. Thanks to the better dataflows introduced in Section 3, roughly an equal amount of energy is dissipated in all three operands in WAX. This highlights that the various forms of reuse, that were unbalanced in WAXFlow-1, have been balanced in WAXFlow-3. Weights and partial sums are read repeatedly out of the local subarray, so their energy is dominated by local subarray access. Meanwhile, activations have to be fetched from a remote tile and are not repeatedly read out of the subarray, so the remote fetch dominates activation energy. Partial sum access is much cheaper in WAX than Eyeriss for two reasons. One is the the small register file used for partial sum accumulation and second is the layer of adders that accumulate results in a cycle before updating the register. WAX reduces both DRAM energy and on-chip energy. While DRAM energy is a significant contributor for a small Eyeriss-like chips, it will be a smaller contributor in larger TPU-like chips with higher on-chip reuse.

Figure 13 shows the layer-wise breakdown for each component while executing ResNet on WAX. For deeper layers, the number of activations reduces and the number of kernels increases; this causes an increase in remote subarray access because kernel weights fetched from the remote subarray see limited reuse and activation rows have to be fetched for each row of kernel weights.

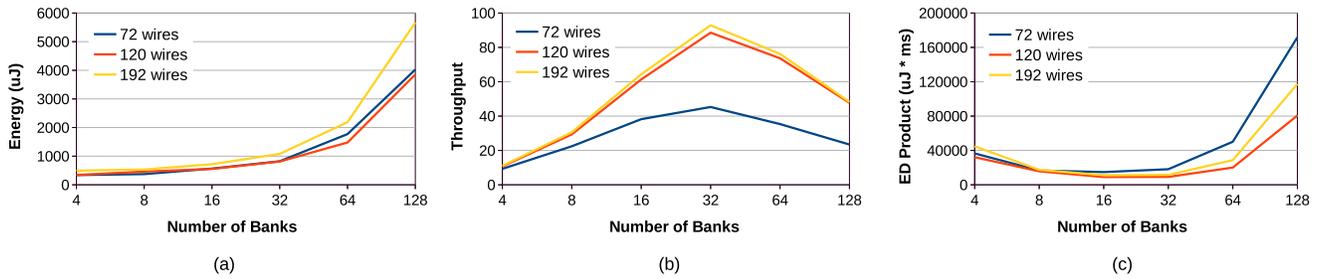


Figure 14: Effect of scaling the size of WAX on convolutional layers in ResNet. (a) Energy with increase in the number of banks in WAX, (b) Throughput, (c) Energy delay product.

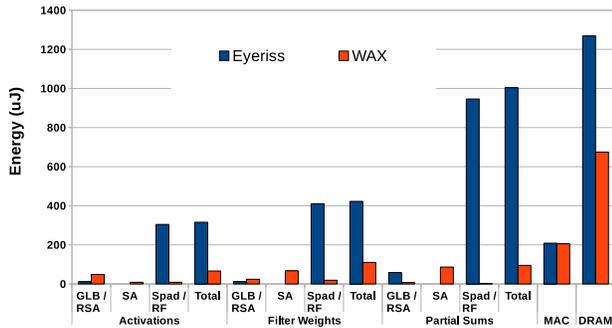


Figure 12: Energy breakdown of activations, weights, and partial sums for WAX and Eyeriss at each level of the hierarchy for convolutional layers of ResNet.

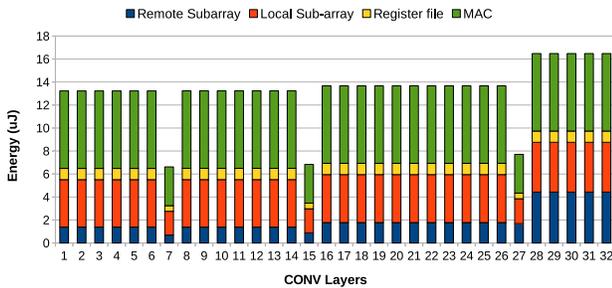


Figure 13: Energy breakdown of each component in WAX for convolutional layers of ResNet.

Figure 11 shows the energy comparison for fully-connected networks at a batch size of 1 and 200. At small batch size, WAXFlow consumes almost the same energy. Although the remote subarray accesses are more expensive than the GLB access in Eyeriss, there is more activation reuse in WAX. At large batch sizes, this overhead is masked by the other energy benefits of WAX and it is nearly 2.7× more energy-efficient.

Figure 14 shows the impact of adding more banks (and hence more MACs) on WAX throughput and Energy consumption. Figure 14b represents the throughput as images per second for each combination of banks and wires. We assume H-Tree bus widths of 72, 120, and 192 for our design space exploration. For all cases, we reserve 8 tiles for remote subarray access. We observe that a

bus width of 120 gives us the best of both energy and throughput. Throughput scales well until 32 banks (128 tiles) and then starts to reduce because of network bottlenecks from replicating ifmaps across multiple subarrays and because of the sequential nature and large size of the H-Tree. To improve scalability, it will be necessary to support an interconnect, say a grid, with higher parallelism and efficient nearest-neighbor communication. Throughput/area peaks at 16 tiles (206 GOPS/mm²) and is higher than that of the Google TPU v1 [24].

6 RELATED WORK

For much of this study, we have used Eyeriss as the baseline and introduced the following key differences. Eyeriss implements a “primitive” per PE that exploits activation and kernel reuse within a row; WAX has significantly more MACs and much fewer registers per tile to reduce wiring overheads and further increase reuse. We also introduce a new data mapping and a shift register per tile that results in a different computation order and reuse pattern.

Similar to WAX, the Neural Cache architecture [3, 16] also tries to move neural computations closer to data in cache subarrays. It does this by introducing in-cache operators that read two rows, perform bit-wise operations and write the result back into the cache. Because of bit-wise operators, it takes many cycles to perform each MAC. Our approach is focused on low energy per operation by reducing wiring overheads and maximizing reuse, while Neural Cache involves many SRAM subarray accesses for each computation.

Some accelerators can leverage analog dot-product operations within resistive crossbars to achieve very low data movement [12, 35, 38]. While promising, such analog elements are likely further down the technology roadmap.

The early DaDianNao [9] and ShiDianNao [15] architectures also focused on near-data processing. DaDianNao used a tiled architecture and placed neural functional units and eDRAM banks within a tile. However, the eDRAM banks were hundreds of kilobytes in size and extensive wiring was required between the eDRAM banks and the MAC units. ShiDianNao later added support for data shuffling and reuse. WAX moves computation into small subarrays and achieves reuse with simple shift registers. Recent commercial efforts by Graphcore [18] and Cerebras [8] have also adopted tiled architectures with compute and SRAM per tile that exploit locality for low data movement.

Like TPU and Eyeriss, the Tesla FSD [42], an IBM core [17], and ScaleDeep [43] are architectures that also implement monolithic systolic arrays fed by large buffers, although with smaller storage units per PE than Eyeriss. Scaleddeep is designed for training and therefore maintains large buffers to store activations during the forward pass. It also implements finer-grained tiles than TPU and Eyeriss.

In the context of GPUs, NUMA and modular chip designs [1, 2, 5, 33] employ distributed GPUs, each with their own local memory, and communicate with each other over short interconnects. They target GPU scaling from a performance standpoint in the post Moore’s law era. Unlike multi-module GPUs, WAX uses deeper hierarchies and distributes computational units across memory at a finer granularity to reduce wire energy.

Several recent works [4, 13, 14, 19, 23, 25, 31, 37, 39, 40] have observed that DNNs exhibit high levels of sparsity, and weights and activations can often be quantized to fewer bits. Eyeriss v2 [10] proposes an architecture that is designed to exploit sparsity in weights and activations to improve throughput and energy efficiency. Eyeriss v2 also uses a flexible NoC to accommodate for varied bandwidth requirements. Both of these are orthogonal approaches that are likely compatible with WAX. As with other sparsity techniques, each tile will require index generation logic to correctly steer partial sums. We leave integration of these techniques in WAX as future work. At a minimum, specific datapaths in WAX can be gated off to save energy by estimating bit widths. To increase throughput when dealing with lower bit widths, configurable MACs, datapaths, shift registers will have to be designed.

7 CONCLUSIONS

In this work, we design a CNN accelerator that pushes the boundaries of near-data execution and short-wire data movement. WAX does this with a deep hierarchy with relatively low resource counts in early layers of the hierarchy. A few-entry register file, a shift operation among an array of registers, and a small adjacent subarray efficiently provide the operands for MAC operations. Various datapaths are considered and we define WAXFlow-3 that balances reuse of various data structures and reduces the expensive accesses (local and remote subarrays). Because of WAX’s ability to perform compute while simultaneously loading the subarray, it has high compute utilization and improves performance by 2 \times , relative to Eyeriss. In terms of energy, WAX yields 2.6-4.4 \times improvement, relative to Eyeriss. By removing the large collection of bulky register files per PE in Eyeriss, the overall chip area is reduced, thus also reducing clock distribution power. The architecture is scalable; as tiles are increased, compute and storage increase in proportion and WAX is able to increase throughput until 128 tiles. The WAX tile can therefore serve as an efficient primitive for a range of edge and server accelerators.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for many helpful suggestions. This work was supported in parts by NSF grant CNS-1718834, Google, and NSF CAREER award 1751064.

REFERENCES

- [1] 2018. NVIDIA DGX-1. <https://www.nvidia.com/en-us/data-center/dgx-1/>.
- [2] 2018. NVIDIA HGX-2. <https://www.nvidia.com/en-us/data-center/hgx/>.
- [3] Shaizeen Aga, Supreet Jeloka, Arun Subramanian, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *Proceedings of HPCA-23*.
- [4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Jerger, and Andreas Moshovos. 2016. Cnvlutin: Zero-Neuron-Free Deep Convolutional Neural Network Computing. In *Proceedings of ISCA-43*.
- [5] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 320–332.
- [6] R. Balasubramonian, A.B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM TACO* 14(2) (2017).
- [7] James Balfour, Richard Harting, and William Dally. 2009. Operand Registers and Explicit Operand Forwarding. *IEEE Computer Architecture Letters* (2009).
- [8] Cerebras. 2019. Cerebras Wafer Scale Engine: An Introduction. <https://www.cerebras.net/wp-content/uploads/2019/08/Cerebras-Wafer-Scale-Engine-Whitepaper.pdf>.
- [9] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of MICRO-47*.
- [10] Y. Chen, T. Yang, J. Emer, and V. Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2019).
- [11] Y-H. Chen, T. Krishna, J. Emer, and V. Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52(1) (2016).
- [12] Ping Chi, Shuangchen Li, Ziyang Qi, Peng Gu, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *Proceedings of ISCA-43*.
- [13] M. Courbariaux and Y. Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv preprint 1602.02830*.
- [14] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830* (2016).
- [15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xi-aobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of ISCA-42*.
- [16] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramanian, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *Proceedings of ISCA-45*.
- [17] Bruce Fleischer, Sunil Shukla, Matthew Ziegler, Joel Silberman, Jinwook Oh, Vijalakshmi Srinivasan, Jungwook Choi, Silvia Mueller, Ankur Agrawal, Tina Babinsky, et al. 2018. A Scalable Multi-TeraOPS Deep Learning Processor Core for AI Training and Inference. In *2018 IEEE Symposium on VLSI Circuits*. 35–36.
- [18] Graphcore. 2017. Intelligence Processing Unit. <https://cdn2.hubspot.net/hubfs/729091/NIPS2017/NIPS%2017%20-%20IPU.pdf>.
- [19] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of ICML-32*.
- [20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).
- [21] R. Ho. 2003. *On-Chip Wires: Scaling and Efficiency*. Ph.D. Dissertation. Stanford University.
- [22] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017).
- [23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061* (2016).
- [24] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datcenter Performance Analysis of a Tensor Processing Unit. (2017).
- [25] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-Serial Deep Neural Network Computing. In *Proceedings of MICRO-49*.
- [26] S. Keckler. 2011. Life After Dennard and How I Learned to Love the Picojoule. Keynote at MICRO.

- [27] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 5 (2011).
- [28] Duckhwan Kim, Jae Ha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *Proceedings of ISCA-43*.
- [29] U. Koster, T. Webb, X. Wang, M. Nassar, A. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloass, R. Pai, and N. Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. *arXiv preprint arXiv:1711.02213* (2017).
- [30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of NIPS*.
- [31] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).
- [32] K. T. Malladi, F. A. Nothhaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz. 2012. Towards Energy-Proportional Datacenter Memory with Mobile DRAM. In *Proceedings of ISCA*.
- [33] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 123–135.
- [34] Naveen Muralimanohar et al. 2007. *CACTI 6.0: A Tool to Understand Large Caches*. Technical Report. University of Utah.
- [35] A. Nag, R. Balasubramonian, V. Srikumar, R. Walker, A. Shafiee, J. Strachan, and N. Muralimanohar. 2018. Newton: Gravitating Towards the Physical Limits of Crossbar Acceleration. *IEEE Micro Special Issue on Memristor-Based Computing* (2018).
- [36] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. Keckler, and W. Dally. 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *Proceedings of MICRO*.
- [37] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S.W. Keckler, and W.J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. (2017).
- [38] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R.S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *Proceedings of ISCA*.
- [39] S.Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of ISCA*.
- [40] S.Han, H. Mao, and W. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding. In *Proceedings of ICLR*.
- [41] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).
- [42] Tesla. 2019. Tesla Autonomy Day. <https://www.youtube.com/watch?v=Ucp0TTmvqOE>.
- [43] S. Venkataramani, A. Ranjan, S. Avancha, A. Jagannathan, A. Raghunathan, S. Banerjee, D. Das, A. Durg, D. Nagaraj, B. Kaul, and P. Dubey. 2017. SCALEDDEP: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. (2017).