# DeepFrack: A Comprehensive Framework for Layer Fusion, Face Tiling, and Efficient Mapping in DNN Hardware Accelerators

Tom Glint
*IIT Gandhinagar*, India
tom.issac@iitgn.ac.in

Mithil Pechimuthu
*IIT Gandhinagar*, India
pechimuthumithil@iitgn.ac.in

Joycee Mekie
*IIT Gandhinagar*, India
joycee@iitgn.ac.in

***Abstract*—DeepFrack is a novel framework developed for enhancing energy efficiency and reducing latency in deep learning workloads executed on hardware accelerators. By optimally fusing layers and implementing an asymmetric tiling strategy, DeepFrack addresses the limitations of traditional layer-by-layer scheduling. The computational efficiency of our method is underscored by significant performance improvements seen across various deep neural network architectures such as AlexNet, VGG, and ResNets when run on Eyeriss and Simba accelerators. The reduction in latency (30% to 40%) and energy consumption (30% to 50%) are further enhanced by the efficient usage of the on-chip buffer and reduction of external memory bandwidth bottleneck. This work contributes to the ongoing efforts in designing more efficient hardware accelerators for machine learning workloads.**

***Index Terms*—Fused Layer Scheduling, Deep Neural Network, Optimal Mapping**

## I. Introduction

Machine Learning (ML) has seen rapid advancements, contributing to progress in vision, natural language processing, and various predictive tasks [1]. This is fueled by computational capabilities expanding across platforms from data centers to edge devices [2]. Specialized hardware accelerators further enhance ML efficiency by utilizing dedicated computing clusters, optimizing data sharing and reuse, outperforming conventional cores [2].

However, escalating operational demands from ML applications demand superior throughput and energy-efficient hardware accelerators. This urgency arises from slower global energy production rates compared to computational demands, potentially leading to an energy deficit by the 2030s [3]. Coupled with ML's penetration into power-limited devices and the evolving nature of ML algorithms, this emphasizes the need for frequent accelerator redesigns [1], [2], [4].

Designing these accelerators demands a thorough understanding of the unique workload processing capacities and data flows. Only one mapping proves optimal for a specific accelerator-workload combination [4]–[6]. Optimal mappings for one architecture might not suit another, underlining the importance of considering the mapping in the design phase.

Deep Neural Networks (DNNs) consist of layers like convolutional, ReLU, and pooling [1]. Typically, hardware accelerators process these layers sequentially, causing roughly half of the energy consumption in DRAM communication [4]. A modern approach to reduce DRAM energy is by fusing multiple layers [7]–[9]. Yet, current tools like LoopTree [8], DeFines [9], and Stream [7] fail to optimally fuse workloads and determine the best partitioning strategy for fused layers.

In response, we introduce DeepFrack, an algorithmic framework to find the optimal layer fusion strategy based on hardware and optimization criteria. DeepFrack also determines data flow and partitioning for execution in fused layer sets. The tool supports varied layer types, from 2D and 3D convolution to ReLU and pooling.

Our experiments with DeepFrack show an energy consumption and latency reduction between 20% and 60% on state-of-the-art DNN accelerators by optimally scheduling fused layers compared to individual layer processing. This paper elaborates on the employed algorithm for optimal fused layer mapping and includes case studies on the Simba and Eyeriss architectures [10], [11]. The framework's open-source version is available at https://github.com/TomGlint/DeepFrack.

This work is supported through grants received from Science and Engineering Research Board (SERB), Government of India, under SERB-SUPRA grant SPR/2020/000450, funding received from Ministry of Electronics and IT (MEITY) under C2S and Semiconductor Research Corporation (SRC) through contracts 2020-IR-2980

## II. Motivation

As deep neural network models continue to evolve and improve, they hold tremendous potential for making our lives more efficient. However, their real-world deployment requires the fulfillment of specific quality of service criteria and adherence to operational constraints. For instance, data center deployments might prioritize high throughput and energy efficiency due to the feasibility of batching services. On the other hand, mobile applications, where live user interactions occur, have to adhere to latency, energy, and power constraints with limited opportunities for workload batching.

Regardless of the application context, the compute capability required to process these evolving workloads continues to escalate. To keep pace with these developments, there is a pressing need to design hardware accelerators that align with the time and operational characteristics of these workloads. Moreover, striking a balance between efficiency, latency, and throughput in these designs is paramount [2], [4].

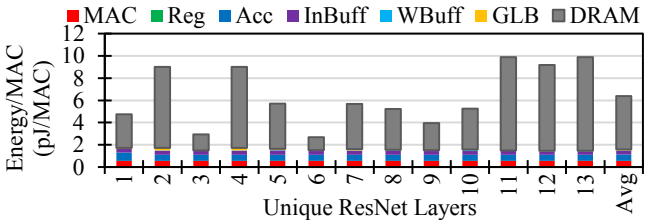


Fig. 1. Amortized cost per compute at the system level for optimal layer by layer scheduling of ResNet on Simba

One critical factor affecting the energy efficiency during neural network inference is the scheduling of the neural network workload on the hardware. Figure 1 illustrates the system-level energy distribution while processing different layers of ResNet on Simba, using optimal layer-by-layer scheduling. Remarkably, more than 50% of the system-level energy is consumed by DRAM access or update [4]. This energy expenditure could be significantly reduced if outputs were forwarded as inputs to the subsequent layer at the chip level, instead of being stored in DRAM for later retrieval.

However, it's impractical to have a large buffer at the chip level, capable of storing an entire layer-level output,

979-8-3503-4859-0/DATE24/© 2024 EDAA

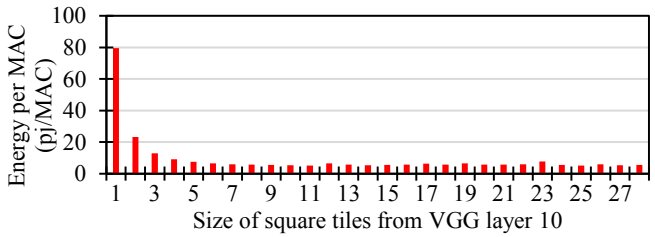


Fig. 2. Amortized cost per compute at the system level for processing different tile sizes belonging to the same convolution layer

due to high area, energy, and access latency costs associated with large buffers [4]. Consequently, each layer is broken down into smaller tiles, which are processed individually, and their outputs are forwarded internally for processing the corresponding tile in the next layer. As shown in Figure 2, executing larger tiles can reduce energy per computation to a certain degree, after which it plateaus.

Given these factors, several critical questions arise in the quest to ensure low latency and energy-efficient workload inference. These include determining the number of layers that can be fused together given a specific hardware configuration, deciding which layers should be fused together to form the neural network given the hardware limitations, understanding how to best tile the face of the fused layer, and strategizing about the caching or fetching of layer weights from DRAM.

Current state-of-the-art tools such as DeFines [9], LoopTree [8] and Stream [7], which support layer fusion or data forwarding during workload execution, do not provide definitive answers to these questions. Instead, they require the user to specify the configuration, including fused layers and/or tiling options.

Our work seeks to address this gap by providing answers to these questions for a given workload and hardware design combination. We believe that this approach will empower researchers to more effectively automate and optimize the design of hardware accelerators capable of executing fused layers.

## III. DeepFrack Overview

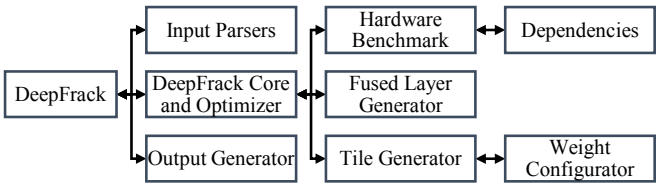


Fig. 3. Overview of DeepFrack framework, depth first traversal of the nodes from left to right and top-to bottom reveals the interaction between different modules in the framework

DeepFrack is a framework that optimizes the scheduling of fused layers in deep neural networks given a specific hardware accelerator. It provides detailed insights, including execution time, energy usage, amortized cost per compute operation, and hardware resource utilization (buffers, computes, bandwidth of the links). Additionally, DeepFrack generates comprehensive reports detailing aspects such as which layers have been fused, how faces are tiled under each subset of fused layers, and the mapping within each tile.

Primarily, DeepFrack accepts two types of input: the workload definition and the architectural definition. These definitions are provided as YAML files in a format that aligns with the community standard, used by platforms like Timeloop [4] and Accelergy [12]. DeepFrack's format has been extended to include ordering and residual layer support.

As shown in Figure 3, the input parser interprets user inputs, while the output generator creates a detailed report. DeepFrack's core algorithms determine the optimal fused layer mapping in several steps:

1) **Benchmarking Tiles:** Tiles of all possible sizes, which may be used to construct the face of each layer, are benchmarked on the specified hardware. During this process, it's assumed that input and weight can come from either the external DRAM or within the modeled chip. Similarly, output can be stored back either within the chip or into the external DRAM. All these scenarios are considered when benchmarking, and the resultant data on the latency and energy cost of executing different tile sizes are saved internally for use in later stages of framework. DeepFrack leverages models of Timeloop [4], Accelergy [12], and Cacti [13] via the dependency manager for this benchmarking process.
2) **Layer Fusion:** The framework identifies all possible combinations of layers that could be fused using the layer fusion generator.
3) **Face Tiling:** For each potential set of fused layers, the tile generator finds the optimal way to tile the face of the fused layer. It does this by examining all possible face combinations made up of square tiles.
4) **Weight Caching:** While evaluating each possible face of the fused subset of layers, the framework determines whether to store each layer's weights on the chip or to refetch it every time a new tile from the same layer is computed.
5) **Optimal Selection:** Lastly, the DeepFrack core selects the best combination of fused layers that align with the optimization criteria, employing memoization to carefully evaluate potential layer fusions.

Through these steps, DeepFrack empowers users to systematically and efficiently optimize the design and utilization of hardware accelerators for deep neural networks.

## IV. Memoized DeepFrack Scheduling Algorithm

### A. Background on problem subset

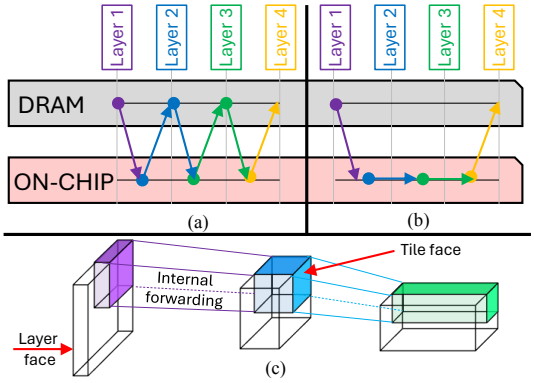


Fig. 4. Depiction of data flow in (a) layer by layer and (b) fused layer processing. (c) Tile level forwarding

**Layer fusion:** As depicted in Figure 4, there is a difference in the data flow between layer-by-layer execution and fused-layer execution. However, it is important to consider that the on-chip buffer may not have sufficient capacity to store a pair

of input and output at a given time. Therefore, only a small tile from the larger face of the layer can be processed at a time in a fused manner.

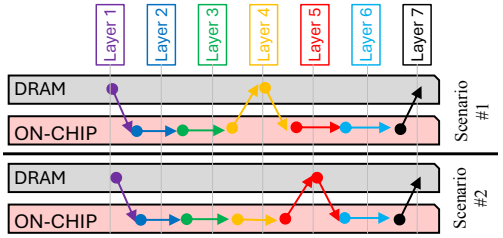


Fig. 5. Depiction of multiple possibilities in fusing layers

**Fused Layer combinations:** As illustrated in Figure 5, various combinations of workload layers can be fused together to form sets of fused layers that constitute the total workload. However, fusing all layers of the workload into a single large fused layer may not be feasible or optimal, considering that weights of all these fused layers cannot be effectively cached given the limited resources typically found inside a hardware accelerator.

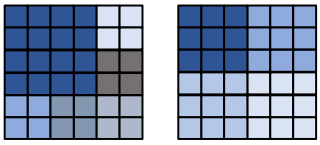

Fig. 6. Multiple possibilities in tiling of the same fused layer face. A face can be constructed out of homogeneous or heterogeneous tile sizes.

**Tile Faces:** As depicted in Figure 6, the face (width and height) of a layer can be formed from multiple tiles, with different combinations of tiles capable of forming the same face. However, as we discussed in Section II, the energy required for computing each operation varies between different tile sizes belonging to the same layer. Therefore, there exists at least one optimal arrangement or construction of the total face with smaller tiles. In this work, we only consider square tiles, even though the input may have different widths and heights. Further, we consider tiles of size larger than width=height=1. Moreover, heterogeneous tiles sizes that could form a face allows splitting dimensions of the input layer for efficient processing such that it is better aligned with the DNN accelerators' spatial organization.

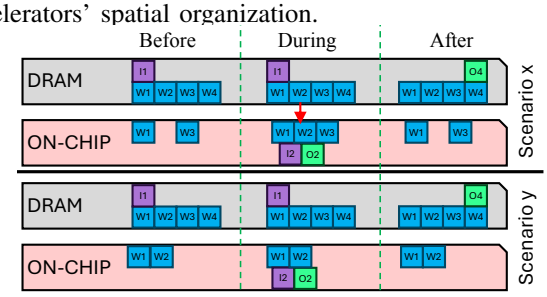


Fig. 7. Depiction of two of the many possibilities in weight caching during execution of the same fused tile with four layers involved. The caching is applicable when processing the second tile on-wards in processing the face and 'before' phase shows the static caching just before the second tile in a stack is processed. The 'during' phase shows the instantaneous state when data belonging to the second layer is processed in the tile. 'After' phase show the state at the end when a stacked tile is processed.

**Weight Caching:** Figure 7 illustrates the possibilities of weight caching for the fused layer tile made up of individual layers at the chip level. If the weights are not cached, then the same weight data from the DRAM needs to be fetched when processing the next tile belonging to the same layer. However, skipping the storage might allow for deeper fused layers.

### *B. Optimal Weight Caching*

Algorithm 1 determines the optimal way to cache the weights of each layer within a fused set of layers, given the set tiles that constitute the face of a fused subset of layers. In doing so, it ensures that the internal buffers have sufficient storage capacity to hold all the cached items, including the forwarded input and output and the combination of weights that it aims to store. For weights that are cached, subsequent tiles that use the same data do not need to refetch the weight data from the DRAM. As this algorithm has exponential complexity, it is parameterized to never exceed $2^{20}$ by merging caching policy selection of adjacent layers for fused subsets with more than 20 layers. The algorithm then selects the optimal combination of weights to be cached that meet the optimization criteria. Furthermore, it provides the list of latency, energy, and the weight caching combination to the caller. In case it is not possible to fuse the layers with the given set of tiles due to resource limitations, the algorithm returns a null value. Further, within the algorithm, `getLayerCost` is called to return the stored performance metrics of a single non-fused tile with a specified data flow from the **benchmarking phase**. `getTotalCost` accumulates the performance results based on weight caching pattern selected for the specified set of tiles belonging to consecutive layers.

### *C. Optimal tiling of fused subset*

In the sections II, we observed that the system level energy per compute varies with tile size. Hence, this problem presents two challenges. First, we need to identify all possible sets of tiles that will completely cover a fused subset face. Second, among these, there might be a tiling configuration that allows for fused operation for the specified depth of the fused layer and optimizes the user-specified criteria.

Algorithm 2 finds the optimal way to tile a given fused subset of layers for the given hardware and also reports the performance metrics associated with processing the subset of workload. The function `getNextTileSet` returns a list of tiles that will completely cover the face. The algorithm for getNextTileSet is omitted for brevity. We call this function iteratively and evaluate if this combination of tiles optimizes the user-given criteria.

To make the solution tractable, we introduce two parameterizations. First, the `getNextTileSet` (algorithm not discussed for brevity) only returns lists of tiles, which are made up of tile sizes greater than 1. Including a tile size of 1 would produce a large number of combinations that cover the face. From our previous discussion, we know that a tile size of 1 has high energy per compute due to poor data reuse and data sharing. Thus, we avoid seeking the optimality of such trivial and sub-optimal combinations.

Second, the face of the fused layer can be covered using tiles of various face shapes. However, finding all such faces is not computationally tractable. Therefore, we limit ourselves to tiles which are square in shape, although the input face need not be square. However, the each of the tiles forming the input shape can be differently sized squares.

`isOptimal` keeps track of the best combination encountered so far and helps update optimalTileFace when a better

**Algorithm 1:** Finds the optimal way to cache weights

**Result:** optimal latency, energy and weight cache pattern for the tile sizes given as input or null

**Function** `getOptimalTileCostList` ($optimization$, $startLayer$, $depth$, $tiles$) **:**
- $maxTile \leftarrow \max(tiles)$
- $maxInput \leftarrow$ getMaxInput($startLayer, depth$)
- $maxOutput \leftarrow$ getMaxOutput($startLayer, depth$)
- **for** $i \leftarrow startLayer$ **to** $startLayer + depth$ **do**
  - $filterVolume[i] \leftarrow$ calcFilterVolume($i$)
- $layersGrouped \leftarrow 1$
- **if** $depth > 20$ **then**
  - $layersGrouped \leftarrow \lceil$ depth$/20 \rceil$
- **for** $combo \leftarrow 0$ **to** $2^{\lceil depth/layersGrouped \rceil} - 1$ **do**
  - $cachePattern \leftarrow$ genPattern($combo, depth, layersGrouped$)
  - **if** *canFit*($maxInput, maxOutput, cachePattern, filterVolume$) **then**
    - $latency, energy \leftarrow 0, 0$
    - **for** $i \leftarrow startLayer$ **to** $startLayer + depth$ **do**
      - $inputSrc \leftarrow$ DRAM if $i = startLayer$, else INTERNAL
      - $outputDest \leftarrow$ DRAM if $i = startLayer + depth$, else INTERNAL
      - $weightSrc \leftarrow$ ONCHIP if $cachePattern[\lfloor$ i - startLayer$/layersGrouped \rfloor] = 1$, else DRAM
      - $layerLatency, layerEnergy \leftarrow$ getLayerCost($i, maxTile, inputSrc, weightSrc, outputDest$)
      - $latency \leftarrow latency + layerLatency$
      - $energy \leftarrow energy + layerEnergy$
    - **if** *isOptimal*($optimization, latency, energy$) **then**
      - $optimalPattern \leftarrow cachePattern$
- **if** $optimalPattern = null$ **then**
  - **return** *null*
- **else**
  - **for** $tileSize \in tiles$ **do**
    - $totalLatency[tileSize], totalEnergy[tileSize] \leftarrow$ getTotalCost($tileSize$, $optimalPattern$, $startLayer$, $depth$, $layersGrouped$)
  - **return** $totalLatency, totalEnergy, optimalPattern$

combination is encountered in the search. `getTotal` returns sum total of processing all Tiles in the current selection, based on data from benchmarking phase and the data flow of each tile selected from Algorithm 1. This function, which finds the optimal tiling list, returns the optimal list and the energy and latency of computing each tile if it is possible to create a tiled version of the face with the given resources at the chip level. If not, null is returned to signify that a fused subset of specified layers is not possible with the given hardware.

### *D. Optimal subset of fused layers*

Algorithm 3 finds the optimal way to combine layers into subsets. The output is an ordered list of these fused subsets, each complete with optimal tiling information and their associated performance metrics.

**Algorithm 2:** Finds optimum tile face of fused layer

**Result:** Total latency, total energy, and tile face or null

**Function** `getOptimalTileFace` ($optimization$, $startLayer$, $depth$) **:**
- $faceSize \leftarrow$ getFaceSize($startLayer, depth$)
- $optimalTileFace \leftarrow$ null
- $optimalLatency, optimalEnergy \leftarrow \infty, \infty$
- $tiles \leftarrow$ getNextTileSet($faceSize$)
- **while** $tiles \neq null$ **do**
  - $tileCostList \leftarrow$ getOptimalTileCostList($optimization, startLayer, depth, tiles$)
  - **if** $tileCostList = null$ **then**
    - **return** *null*
  - $latency, energy \leftarrow$ getTotal($tileCostList$)
  - **if** *isOptimal*($optimization, latency, energy, optimalLatency, optimalEnergy$) **then**
    - $optimalTileFace \leftarrow tiles$
    - $optimalLatency, optimalEnergy \leftarrow latency, energy$
  - $tiles \leftarrow$ getNextTileSet($faceSize$)
- **if** $optimalTileFace = null$ **then**
  - **return** *null*
- **else**
  - **return** $optimalLatency, optimalEnergy, optimalTileFace$

**Algorithm 3:** Finds the set of optimal fused layer set

**Result:** Optimal set of fused layers, optimal tile faces, total latency, total energy

**Function** `getOptimalFusedLayerSet` ($optimization$, $totalDepth$) **:**
- $costMatrix \leftarrow$ array($totalDepth$, $totalDepth$)
- **for** $initialLayer \leftarrow 1$ **to** $totalDepth$ **do**
  - **for** $finalLayer \leftarrow initialLayer$ **to** $totalDepth$ **do**
    - $costMatrix[initialLayer, finalLayer] \leftarrow$ getOptimalTileFace($optimization, initialLayer, finalLayer - initialLayer$)
- $optimalSet \leftarrow$ null
- $optimalLatency, optimalEnergy \leftarrow \infty, \infty$
- $fusedSet \leftarrow$ getNextFusedSet($totalDepth$)
- **while** $fusedSet \neq null$ **do**
  - $latency, energy \leftarrow 0, 0$
  - **for** $layer \in fusedSet$ **do**
    - $latencyLayer, energyLayer, \_ \leftarrow costMatrix[layer.start, layer.end]$
    - $latency \leftarrow latency + latencyLayer$
    - $energy \leftarrow energy + energyLayer$
  - **if** *isOptimal*($optimization, latency, energy, optimalLatency, optimalEnergy$) **then**
    - $optimalSet \leftarrow fusedSet$
    - $optimalLatency, optimalEnergy \leftarrow latency, energy$
  - $fusedSet \leftarrow$ getNextFusedSet($totalDepth$)
- **if** $optimalSet = null$ **then**
  - **return** *null*
- **else**
  - $optimalTileFaces \leftarrow$ getTileFacesFromSet($optimalSet, costMatrix$)
  - **return** $optimalSet, optimalTileFaces, optimalLatency, optimalEnergy$

In order to avoid unnecessary repetition of performance calculations, the algorithm constructs a matrix that stores the performance data for all the layers. This matrix includes information for each layer from the first to the last, as well as their potential combinations or "fused possibilities". The depth of these fused possibilities ranges from 1 to $finalLayer - currentLayer$, and there are $\mathcal{O}(n^2)$ such possibilities in total.

To illustrate, let's consider a workload of 12 layers. If we have fused subsets of layers of lengths [3,7,2] and [1,2,7,2], the cost for the fused subset of length 7 and 2 would not need to be recalculated, because it's already stored in our matrix.

The challenging part of this process is determining all the unique ways the layers can be fused. The time complexity of generating all partitions of a number $n$ is represented by the partition function $p(n)$, which can be approximated by the Hardy-Ramanujan formula [14] as $\mathcal{O}(p(n)) \approx \frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4n\sqrt{3}}$

Calculating all permutations of unique partitions involves examining the number of partitions and the size of each partition. The number of permutations for a partition of size $k$ can be computed in $\mathcal{O}(k)$ time using the multinomial coefficient, making the overall time complexity represented by $\mathcal{O}(n \cdot p(n))$

However, this becomes computationally infeasible for values of $n$ greater than 20. To manage this, the function `getNextFusedSet` (algorithm not discussed for brevity) internally groups adjacent layers together to keep the total below 20. The output from this function retains the individual layers for clarity. Finally, the algorithm explores all the unique fusing possibilities and identifies the one that meets the given optimization criteria. So the total time complexity is $\mathcal{O}(n^2)2^y + \mathcal{O}(y \cdot p(y))$ where $y = \lceil n/\lceil (n/20) \rceil \rceil$. Further, it should be noted that the layers presented to Algorithm 3, is in topologically sorted order. However, the algorithm is aware of multiple outgoing and incoming edges to a layer. Further, if both the ends of the edge appear within a stack for evaluation, caching the data represented by the edge within the chip is also considered. This allows for generating schedules for DNNs like inception network. However, all topological sort combinations are not automatically considered, and if the user wants to compare all or specific combinations, they may be sequentially evaluated and the best one may be picked from the output list. Regarding the wall clock time for generating the optimal schedule, $\sim$ 6 hours is required for the benchmark operation for all the networks used here on a Ryzen 5950 processor-based system. Further, for the rest of the algorithm, 5 minutes, 13 hours, 16 hours, and 22 hours (upper bound) are required for AlexNet, VGG, ResNet-18, and ResNet-50.

## V. Results

### A. Hardware model validation

This work utilizes Timeloop and Accelergy internally, primarily due to their flexibility in modeling arbitrarily deep hardware accelerator architectures, and their proficiency in finding optimal mapping for individual layers. These dependencies also allow for data residency at various levels, thus providing comprehensive hardware modeling capabilities. The specific models we have chosen to work with in this paper are Eyeriss and Simba, both of which represent state-of-the-art DNN hardware accelerator architectures of their time. Our choice of these models, and indeed our use of Timeloop and Accelergy, is further validated by their successful hardware matching demonstrated in [4]. An additional advantage of using Timeloop and Accelergy is the ease of interfacing and invocation, facilitated by their use of YAML and Python. This ensures that our work can be easily adapted by the community for a variety of hardware models. Further, the hardware models used in this work are openly available in the Accelergy code base.

### B. Latency and Energy

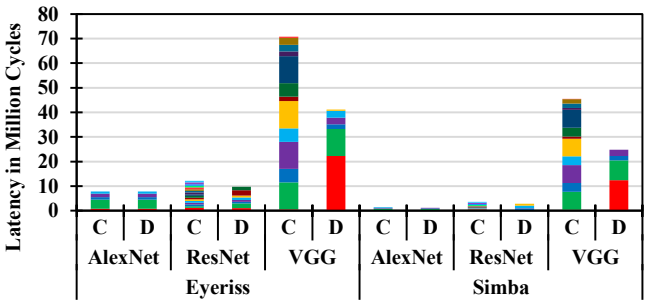


Fig. 8. Latency obtained with Classical (C) and DeepFrack (D) scheduling. Each block in a stack show latency of layers or fused layers (in case of DeepFrack). Similar annotation for further figures

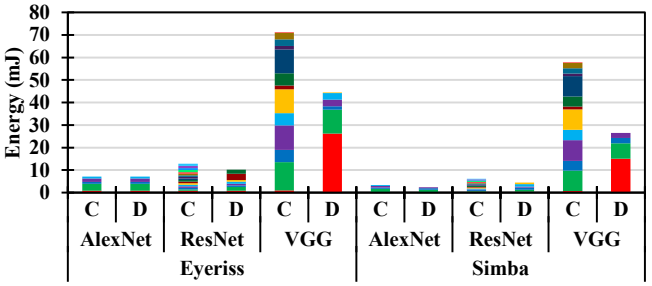


Fig. 9. Energy obtained with Classical (C) and DeepFrack (D) scheduling.

Figures 8 and 9 compare the latency and energy consumption between the DeepFrack method and traditional layer-by-layer (optimal) scheduling. The comparisons are made across AlexNet, VGG, and ResNet architectures when run on Eyeriss and Simba DNN accelerators.

We notice significant improvements in terms of speed when using the DeepFrack approach. The accelerators, Eyeriss and Simba, demonstrated speedups of 1.54× and 1.75×, respectively. This is largely due to the superior asymmetric scheduling of different tile sizes, which also helps alleviate the bottleneck created by the external memory bandwidth. While the latency improvements are noteworthy, the most substantial gains are seen in energy consumption. Eyeriss and Simba accelerators exhibited an average reduction of 30% and 50% energy, respectively, when using DeepFrack.

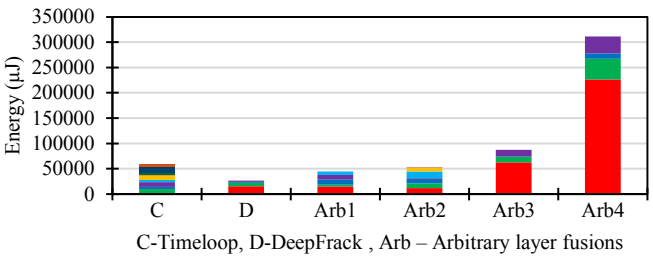


Fig. 10. Comparing VGG schedule for Timeloop, DeepFrack and arbitrary layer fusion

Up on analysis of the detailed statistics, it is observed that for Eyeriss, the energy reduction mostly came from a better asynchronous mapping of tiles, and the proposed scheduling only provided a 2% reduction in DRAM access energy. However, for Simba, the DRAM access energy was reduced by 51% on top of the on-chip energy being reduced by 53%. Further, layers that were fused with the most depth are in the initial layers due to smaller weight volume with high data reuse, whereas, towards the final layers, the depth of fused layers is reduced.

Further, we perform sensitivity analysis on VGG layer fusion to show the impact of arbitrary layer fusion on Simba's system energy, as shown in Fig. 10; random fusion choices can lead to increased power even compared optimal layer by layer scheduling.

## VI. Other Related Works

Layer fusion and related optimizations have been a topic of interest in earlier research endeavors. However, many studies fail to encompass the three-fold criteria of our approach: (1) Deciding on the layers to fuse, (2) Determining the tiling strategy for the fused layer, and (3) Selecting the data caching strategy during sequential tile computation. Distinctively, our methodology is universally adaptable, not confined to any specific hardware or data flow. The primary objective is to craft a flexible framework that paves the way for future hardware explorations. Such a platform facilitates hardware design evaluations, analyzing each workload based on its optimal fused layer scheduling.

The study by Yoon et al. [15] focuses on merging successive layers for specific data flows but overlooks tiling and caching aspects. Research in [16] primarily examines storage capacity in FPGA-centric designs, yet does not fully explore the impact on operational costs in accelerators. Chimera [17] addresses both storage and reuse but is tailored for CPUs, GPUs, and specific NPUs, lacking in modeling generic accelerators for comprehensive comparisons.

Additionally, operator fusion, investigated in studies such as Zhao et al. [18], Zheng et al. [19], Cai et al. [20], and Liu et al. [21], integrates MAC operations across layers. However, it often misses critical aspects of hardware accelerators, like shared reads or vector MAC accumulations. This oversight can significantly affect energy efficiency, as illustrated in Fig. 2.

## VII. Conclusion

In this paper, we introduced DeepFrack, an innovative framework that combines an optimal tiling strategy and layer fusion for deep learning workloads, with a focus on asymmetric tile sizes. Our approach leverages a computationally efficient method to solve the scheduling problem, significantly reducing the latency and energy consumption in different DNN architectures run on state-of-the-art accelerators. The results showed promising improvements in both speed and energy efficiency, demonstrating the potential of DeepFrack in practical deep learning applications. Notably, the ability of the framework to alleviate the external memory bandwidth bottleneck opens up new opportunities for efficient hardware design in DNN accelerators.

We believe DeepFrack presents a paradigm shift towards more energy-efficient deep learning computations. Our findings point towards future research directions to improve hardware-software co-design principles for DNNs, addressing some of the key challenges faced in the era of ubiquitous AI. With the continuous advancement of deep learning and the need for efficient computations, frameworks like DeepFrack will play a crucial role in shaping the future of AI hardware accelerators.

## References


[1] V. J. Reddi *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[2] Y. Chen *et al.*, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[3] H. Fuchs *et al.*, "Comparing datasets of volume servers to illuminate their energy use in data centers," *Energy Efficiency*, vol. 13, pp. 379–392, 2020.

[4] A. Parashar *et al.*, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2019, pp. 304–315.

[5] L. Mei *et al.*, "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021.

[6] L. Mei *et al.*, "A uniform latency model for dnn accelerators with diverse architectures and dataflows," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022.

[7] A. Symons *et al.*, "Towards heterogeneous multi-core accelerators exploiting fine-grained scheduling of layer-fused deep neural networks," *arXiv preprint arXiv:2212.10612*, 2022.

[8] M. Gilbert *et al.*, "Looptree: Enabling exploration of fused-layer dataflow accelerators," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 316–318.

[9] L. Mei *et al.*, "Defines: Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 570–583.

[10] Y. S. Shao *et al.*, "Simba: scaling deep-learning inference with chiplet-based architecture," *Communications of the ACM*, 2021.

[11] Y.-H. Chen *et al.*, "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.

[12] Y. N. Wu *et al.*, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019.

[13] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[14] G. Almkvist *et al.*, "A hardy-ramanujan formula for restricted partitions," *Journal of Number Theory*, vol. 38, no. 2, pp. 135–144, 1991.

[15] M. Yoon *et al.*, "Architecture-aware optimization of layer fusion for latency-optimal cnn inference," in *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2023, pp. 1–4.

[16] M. Alwani *et al.*, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[17] S. Zheng *et al.*, "Chimera: An analytical optimizing framework for effective compute-intensive operators fusion," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1113–1126.

[18] J. Zhao *et al.*, "Apollo: Automatic partition-based operator fusion through layer by layer optimization," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 1–19, 2022.

[19] G. Zheng *et al.*, "Operator fusion scheduling optimization for tvm deep learning compilers," in *2023 3rd International Symposium on Computer Technology and Information Science (ISCTIS)*. IEEE, 2023.

[20] X. Cai *et al.*, "Optimus: An operator fusion framework for deep neural networks," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 1, pp. 1–26, 2022.

[21] Z. Liu *et al.*, "Dlfusion: An auto-tuning compiler for layer fusion on deep neural network accelerator," in *2020 IEEE ISPA/BDCloud/SocialCom/SustainCom*. IEEE, 2020, pp. 118–127.