Ruby: Improving Hardware Efficiency for Tensor Algebra Accelerators Through Imperfect Factorization

Mark Horeni
Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA

Pooria Taheri
Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA

Po-An Tsai
NVIDIA
Westford, MA, USA

Angshuman Parashar
NVIDIA
Westford, MA, USA

Joel Emer
MIT / NVIDIA
Cambridge, MA, USA

Siddharth Joshi
Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA

Abstract*—Finding high-quality mappings of Deep Neural Network (DNN) models onto tensor accelerators is critical for efficiency. State-of-the-art mapping exploration tools use remainderless (i.e., perfect) factorization to allocate hardware resources, through tiling the tensors, based on factors of tensor dimensions. This limits the size of the search space, (i.e., mapspace), but can lead to low resource utilization. We introduce a new mapspace, Ruby, that adds remainders (i.e., imperfect factorization) to expand the mapspace with high-quality mappings for user-defined architectures. This expansion allows us to allocate resources more precisely by generating tile sizes that better conform to hardware resources. However, this mapspace expansion also incurs an increase in the number of unique mappings. Consequently, this paper studies the trade-off between Ruby's mapspace expansion and mapping quality. We propose Ruby-S (Spatial) to only employ imperfect factorization towards improved parallelism. Ruby-S incurs a moderate mapspace expansion while reducing energy-delay product (EDP) up to 50% when implementing ResNet-50 on an Eyeriss-like architecture with an average improvement of 20%. For the most part, this improvement can be attributed to higher compute utilization. EDP on a Simba-like architecture improves up to 40% with an average of 10%. For DeepBench workloads Ruby-S yields improvements of up to 45% with an average improvement of 10% on an Eyeriss-like architecture. Ruby-S is robust to accelerator configurations and improves EDP by 20% on average, with a maximum improvement of 55% when implementing ResNet-50 on different accelerator configurations. Ruby-S mappings form a new Pareto frontier, improving the performance of previous configurations by an average of 30% and 20% for ResNet-50 and DeepBench workloads respectively.

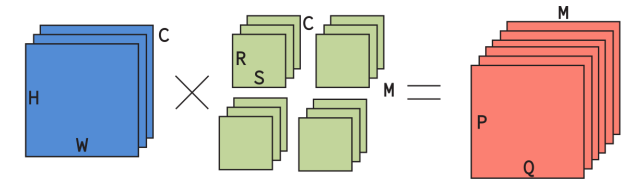
I. Introduction

Tensor algebra (TA) computations are crucial to many workloads that are deployed across computing platforms, ranging from cloud services to constrained edge devices [1–3]. Increasingly, high-performance computing applications like weather modeling, fluid dynamics, and physics simulations are being augmented by algorithms that also benefit from accelerating TA computations, including those employed in machine learning (ML) and Deep Neural Network (DNN) workloads [4–6]. Similarly, many tasks like video rendering, image recognition, and speech processing are run on power constrained mobile devices that also employ TA computations [7–9]. Given this prevalence of TA computations, the design of domain-specific TA accelerators that can efficiently execute different TA kernels [1] has become critical to the next generation of energy-efficient computing platforms. The specialization afforded by TA accelerators opens multiple avenues for optimizations that include data orchestration [10] and scheduling [11], optimizing dataflow and reuse [2, 12], as well as optimizations that adapt the TA kernels to better suit the underlying hardware [13, 14]. These different optimizations can improve the end-to-end performance of TA accelerators, often by orders of magnitude [11, 12, 15–20]. Crucially, these dataflow optimizations can occur at different levels of granularity, as an example optimizations like *operator fusion* employed in *vertical scheduling* tools [11, 21] can be used in conjunction with more fine-grained, per-operation optimizations like loop-reordering. This paper, specifically focuses on fine-grained optimizations to improve the efficiency of mapping tensor operations on TA accelerators.

Consider a tensor operation in a convolutional neural network (CNN), written as a *loopnest* as shown in Fig. 1. This convolution can be broken down into its operand tensors: input feature maps (IFMs), output feature maps (OFMs), and filter parameters. Each such tensor can be represented as a set of nested loops, where the loop bound is determined by the dimension of this tensor. As an example, the number of channels in the OFM (M) also determines the loop bound in the loopnest in Fig. 1.

If we define a mapping as each unique allocation, in space and time, of a tensor operation to the different processing elements (PEs) and memories in the memory hierarchy. There are multiple ways to map this loopnest on different TA

MH, PT, and SJ were supported by Semiconductor Research Corporation (SRC) as part of ASCENT, a JUMP center.




```
for n in [0:N) #Batch
  for m in [0:M) #Output Channels
    for c in [0:C) #Input Channels
      for p in [0:P) #Input Rows
        for q in [0:Q) #Input Columns
          for r in [0:R) #Filter Rows
            for s in [0:S) #Filter Columns
             out[p,q,m]+= in[h+s, w+r,c]
                          ·weights[r,s,c,m]
```

Fig. 1: Convolution operations employed in convolutional neural networks as well as their equivalent loop nest consisting of 7 nested loops.

accelerator architectures (e.g., [2, 3, 17, 22–24]). Different mappings can radically alter the data reuse and access patterns for different TA computations, in turn altering the impact on energy and latency. The vast space of mappings together with the complexity of their interaction with the underlying hardware necessitates automatic mapping discovery. Automatically discovered mappings have been shown to improve accelerator efficiency by up to 20× [16] when compared to naïve mappings. However, automatic mapping generation and discovery must contend with the enormous size of the mapspace (the set of all mappings) which is typically large enough to exclude an exhaustive search. While tools that can rapidly and comprehensively evaluate mappings for fixed architectures have shown great promise [19, 25], adopting similar techniques for user-defined architectures remains challenging due to the large number of mapspaces possible for user-defined architectures. Rapidly generating mapspaces that deliver high-quality mappings for user-defined architectures is critical for evaluating and developing future TA accelerators.

State-of-the-art (SoTA) mapping tools [15, 16, 26, 27] address the challenge of generating and searching through a mapspace for high-quality mappings by employing various heuristics. As an example, Timeloop [16] decomposes the tensor mapping problem into a set of independent subproblems: (i) mapspace generation, (ii) mapspace search, and (iii) architecture cost modeling. Alternative approaches may merge these subproblems to achieve greater speed or accuracy in their evaluation of a specific architecture [26], or improve a specific subproblem such as search [18, 28, 29]. Indeed, GAMMA [28] and Mind Mappings [29] improve the search in Timeloop-like mapspaces, with Timeloop's mapspace used as the training set for the neural surrogate in [29]. Consequently, improved mapspace quality, promises to be highly impactful to future TA accelerator designs. This paper focuses on exactly such improvement to Timeloop's mapspace.

Timeloop constrains its mapspace by using index factorization which decomposes a tensor into tiles of sizes determined by the factors of that tensor's shape along each dimension. These decompositions, in turn, determine the loop bounds in the loopnest of a TA computation. We refer to such decompositions as *perfect factorization* and refer to the mappings generated by mappers employing index factorization as perfect-factorization mappings (PFMs). Generating the mapspace through PFMs restricts the mapspace size by limiting the possible tilings of the tensor [15, 16, 18, 28, 29]. However, PFM performance is only optimal when the hardware resources (e.g., number of PEs, available memory) can precisely match the tile size. As we will show, for a set of benchmark workloads, the variety in tensor shapes across TA workloads consistently results in a mismatch between PFM generated tilings and hardware resources resulting in hardware underutilization.

This paper proposes a new mapspace, *Ruby*, to address the underutilization and underperformance that arise when tensor dimensions do not align with the underlying TA accelerator hardware resources. Ruby achieves better alignment by expanding the mapspace to allow for remainders (i.e., *imperfect factorization*) along any of the tensor dimensions, producing higher quality mappings that better utilize the available hardware. Unfortunately, relaxing the constraint of perfect (i.e., remainderless) factorization expands the mapspace tremendously, which we address through Ruby-S, limiting the expansion based on the spatial attributes of the hardware. Ruby-S is able to improve upon mapping quality over a wide range of workloads while also keeping the mapspace expansion manageable.

Our main contributions are:

- developing *Ruby* a new mapspace that addresses the challenge of underutilization of resources seen in SoTA mappers employing PFMs;
- analyzing the trade-off between mapspace expansion and mapping quality and introducing *Ruby-S* to deliver high-quality mappings with manageable mapspace expansion;
- using Ruby-S to analyze the impact of architectural design choices, such as number of PEs, on the performance of various DNN-based tensor computation workloads.

II. Background and Related Work

A. Related Work

Recently proposed tools that optimize the mapping of TA computations onto accelerators build on several established techniques. Some early techniques leveraged linear cost-models and transforms to optimize for TA computations. Wolf and Lam [30] create a linear, cache-aware model that quantifies reuse and locality in TA computations. Building on this model, they developed mutually independent loop-transformations that could increase reuse or data locality. Finally, they developed a means to analytically synthesize compound loop transformations such that the effective space of all possible transformations is pruned and easily searched. Alternative approaches targeted specialized hardware, partitioning algorithms across large computational arrays using an early version of index factorization [31]. Variations of these early

techniques are still employed by contemporary tools, e.g., tools targeting systolic arrays [32]. However, many commonly proposed TA accelerators employ dataflow architectures, with targeted mapspace generation and evaluation tools [2, 22]. Maestro describes the mapspace for a dataflow architecture as being data-centric, formulating the mapspace in terms of the computations allocated across PEs and the scheduling of those computations over time [19]. More recently, TENET offers a different formulation of the mapspace by explicitly encapsulating the dataflow and computation allocation using *relations* for a more comprehensive notion of various mappings [27]. Alternative top-down techniques to evaluate and traverse different mapspaces for user-defined architectures can offer greater flexibility due to their more managable mapspaces. Timeloop is a commonly known mapspace generation and evaluation tool that uses PFMs to generate mapspaces [16]. Timeloop formulated the architecture-kernel interaction as two intertwined problems: (a) mapspace generation and mapping and (b) evaluating mappings on an architectural cost-model. dMazeRunner employs PFMs with additional restrictions to the memory hierarchy and prunes the mapspace using heuristics such as: (a) PE utilization, (b) data access patterns in DRAM, and (c) local data reuse [26]. ZigZag developed a memory-centric approach that allows uneven tiling of tensors [15]. As a consequence of this uneven tiling, different levels in different memory hierarchy must accommodate tensor-specific tile sizes. Ruby expands these tools by expanding the mapspace through imperfect factors similar to ZigZag's expansion of the mapspace with uneven mappings.

Complementing the mapspace generation tools described above, other frameworks target better search techniques to find high quality mappings more easily. COSA reformulates Timeloop's optimization as a mixed-integer-programming problem for dramatically improved search times on Timeloop's mapspace [18]. Mindmappings builds upon Timeloop's mapspace to develop a differentiable surrogate space which encapsulates the architectural performance cost-model [29]. This differentiable space can then be traversed through gradient descent and optimized mappings can be generated through backwards propagation of gradients through the mapspace. GAMMA builds upon Maestro's framework and improves the search by using a genetic algorithm that optimizes for mapping quality [28]. Our proposed mapspace generation framework is orthogonal to these search strategies and can leverage them for improved performance.

B. Baseline Tensor Algebra Accelerator

To demonstrate the generality of our framework we show results on both an Eyeriss-like [2] and Simba-like [3] architecture. However, for consistency with literature we consider an Eyeriss-like TA accelerator with PEs organized in a 2D grid as our baseline. Unless otherwise noted, we follow the original specifications with each PE comprising input buffers of depth 12, partial sum buffers of depth 16, weight buffers of depth 224, and a 16-bit integer multiply accumulate unit as shown in Fig. 2. Eyeriss specifies a PE array of 14×12

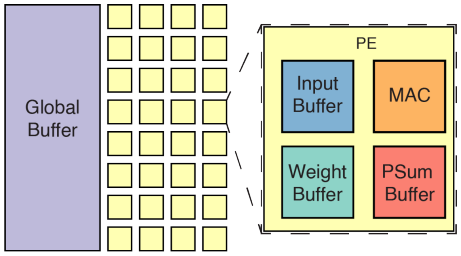


Fig. 2: Eyeriss-Like architecture consisting of *Global Buffer* connected to an array of processing elements, containing memory dedicated to each operand tensor.

```
Off Chip
  for n₁ in [0:N₁) #Batch
    for m₁ in [0:M₁) #Output Channels
      for c₁ in [0:C₁) #Input Channels
        for p₁ in [0:P₁) #Split Input Rows
          for q₁ in [0:Q₁) #Split Input Columns
Global Buffer
            for p₂ in [0:P₂) #Split Input Rows
              for q₂ in [0:Q₂) #Split Input Columns
PE Fanout
                parFor r₁ in [0:R₁) #Filter Rows
                  parFor s₁ in [0:S₁) #Filter Columns
Arithmetic Level
                    out[p,q,m] += in[h+s, w+r,c]
                                  ·weights[r,s,c,m]
```

Fig. 3: An example of a loop ordering broken down by memory hierarchy. In this example, part of the feature map is stored for reuse in a global buffer, while the the filters are spatially allocated across a set of PE's able to perform computation.

and we test with a slightly larger shared global buffer (GLB) of size 128 KiB. For simplicity, we do not model the effect of Eyeriss's run length encoding (RLE). In this architecture, tensors are stored off-chip in DRAM and subsequently inputs and outputs are moved on-chip into the GLB, and then to their dedicated memories in the PE, while model parameters are moved directly into their memories in each PE.

C. Problem Formulation

Consider the tensor operations pictured in Fig. 1. To run such an operation efficiently on a given TA accelerator, entails multiple operations and decision points. First, operands must be fetched from memory to on-chip memories while keeping to the constraints imposed by the memory hierarchy and size. Next, operands must be assigned (or scheduled) to each memory and computational element in the hierarchy, such that they can implement the required computation.

A more comprehensive loop nest could describe the precise hardware allocation of these tensor computations shown in Fig. 1. Such a description would decompose each loop into smaller components revealing the precise tensor allocation and blocking implemented at each level of the memory hierarchy. This results in a set of ordered loops, describing the tiling sizes and access patterns incurred by a given architecture for a given tensor operation. This breakdown is further exemplified in the expanded loopnest shown in Fig. 3, where parts of the feature maps are stored inside a large global buffer and

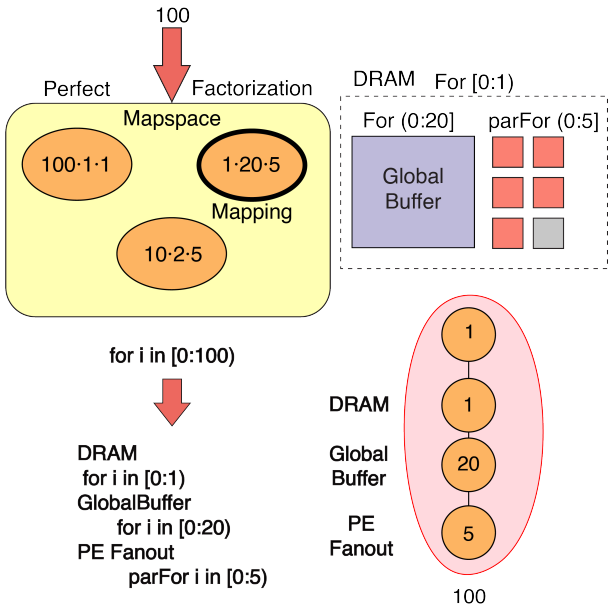


Fig. 4: A sample of the original perfect-factorization-based mapspace. The highlighted example shows a temporal allocation inside a global buffer and spatial allocation across 5 of 6 possible processing elements. Displayed is a depiction of the mapspace and the example, a loopnest, and a tree-like structure to visualize the tiling structure.

while the rows and columns of the filter are spatially allocated computational units. In turn, these different mapping choices affect the scheduling of computation, amount of access counts from a buffer, storage coordination of different tensors, and network traffic of tensors – impacting the energy and latency of implementing TA computations for that architecture. A convolution needs to use the inputs of a feature map multiple times given a single weight. Storing these inputs to be reused multiple times in intermediate buffers can be helpful in reducing traffic from off chip. Fig. 3 highlights the possibility of parts of the feature map being stored in an intermediate buffer enabling optimizations through reuse that could lower both energy and latency. The complex landscape of trade-offs resulting from these tensor assignment and architectural paramterizations (e.g., buffer size) underlies the complexity of the *mapping problem*.

Automated tools that can efficiently traverse these vast map spaces and deliver efficient mappings for a user-defined accelerator play a crucial role in maximizing the accelerator performance [17–19, 25, 26, 28, 29, 33]. Ultimately, an ideal mapping tool aims to rapidly evaluate and deliver the optimal schedule for a TA computation that would optimize the energy, latency, or any combination of these attributes for the accelerator.

D. Generating Mappings for an Architecture

Optimized mappings that are tailored to an architecture, necessarily depend on both the hardware architecture and the workload. Typically mappings are generated through loopnest descriptions of the workload in a two-step process. The first step modifies this loopnest description by generating tiled tensors (i.e., memory blocking) for each level of the memory hierarchy provided in the architectural description. Since the initial mapspace only accounts for the memory hierarchy and not other architectural parameters (e.g., memory size) it includes invalid mappings that are filtered out in the second step. We first examine how the task of distributing 100 elements, from a tensor stored in DRAM across different PEs through an intermediate GLB, can be mapped on to the toy Eyeriss-like architecture in Fig. 4. This architecture is configured with 6 PEs, without local storage, arranged in a 3×2 grid and a GLB of size 1 KiB. Consider the highlighted mapping $(1 \cdot 20 \cdot 5)$, where 100 elements are stored in the GLB, followed by 20 iterations of distributing 5 elements over 5 PEs, with one inactive PE. Note, that because each iteration distributes 5 elements across the PEs and we take 20 iterations to distribute all the tensor elements, the GLB must contain all 100 elements. Alternatively, the mapspace also includes a different mapping which would not utilize the parallelism of mulitple PEs and instead access the DRAM 100 times $(100 \cdot 1 \cdot 1)$.

The loop bounds determined for tensors at each level of the memory hierarchy correspond to the tiled tensor that will be transferred across levels of memory hierarchy. Mapping tools optimize these tile sizes at each level of the memory hierarchy towards their objective such as minimizing energy, delay, or another target objective. To extend our previous single dimensional example to the multiple dimensions encountered in practical TA workloads, we treat each factor as a single dimension of a tiled multidimensional tensor, with tiling determined for each level of the memory hierarchy. This formulation also defines a logical memory hierarchy that is different from the physical memory hierarchy, inclusive of both *spatial* levels and *temporal* levels. Here, we define a spatial level in the memory hierarchy to represent parallelism or vector operations, e.g., fanout between two physical memories or multiply-accumulate (MAC) units — represented by a *parFor*. In our highlighted example we have a spatial level of size 5 that represents a fanout from the GLB to the PEs of size 5. A temporal level corresponds to multiple accesses to a physical memory — represented by a *for*. In our example the GLB has a temporal level of 20, consequently we iterate over GLB 20 times, with each such iteration accessing 5 tiled elements.

Further improvements can be derived from additional optimizations employed by SoTA tools. One such is *bypassing*, which allows tensors or some elements of tensors to skip accessing levels of the memory hierarchy [15]. Similarly, loopnests such as the ones in Fig. 1, can be reordered while maintaining semantics, e.g., the ordering between input and output channels can be swapped. The unique mappings corresponding to such loop reorderings express the access order for these tensors. Permuting the access order impacts the reuse seen by these different tensors, e.g., IFM or OFM memories might see different reuse depending on input- or output-stationary loop orderings. The complex landscape of available optimizations and their myriad interactions with the underlying hardware motivates the development of tools that can rapidly

generate these mapspaces (e.g., previously mentioned [15, 16, 19, 26, 27]) and efficiently search them [18, 28, 29]).

E. Perfect Factorization

Because all valid mappings are a subset of the initially generated mapspace, creating a mapspace rich with high quality mappings is crucial to properly evaluate a tensor accelerator's performance on different workloads. We consider a SoTA mapspace generation tool Timeloop [16] as our baseline PFM generator and use Timeloop as the defacto PFM generator in subsequent references. Timeloop employs index factorization to generate the tiling loop bounds based on **perfect factorization** of tensor dimension sizes. All valid mappings generated by Timeloop, our baseline PFM, solve this recursive equation per tensor dimension:

$$L_n = L_{n+1} P_n. \tag{1}$$

This equation formulates the total number of tiles at the current level of the memory hierarchy (L_n) as the product of the total number of tiles at the level immediately higher in the memory hierarchy (L_{n+1}) and number of sub-tiles created at the current level (P_n), where n denotes the level of the memory hierarchy. To elaborate, consider our example from Fig. 4. If there are 20 tiles at the GLB ($L_1 = 20$), and we further divide each tile into 5 sub-tiles ($P_0 = 5$) then the total number of tiles that will need to be allocated to the PEs will be 100 i.e., $L_0 = 100$. Since this is the lowest level of the memory hierarchy, L_0 must equal the size of the computation. We also call P_n the *tiling factor* for that level of the memory hierarchy. The recursion is initiated by setting $L_{n+1} = 1$ for the highest level of memory hierarchy. Under this formulation, P_n **must be the product of prime factors** of the tensor dimension being evaluated. This can be represented as a chain of products as shown in the bottom of Fig. 4 and Fig. 6 (a), where the loop bounds of each node are multiplied to get a final dimension size D. An example mapping generated by such a PFM is shown in Fig. 4, where the highlighted mapping represents the following sequence of recursions: $L_3 = 1$ (base case), this is followed by choosing a P_n in the domain of eq (1), in this example we choose $P_2 = 1$, resulting in $L_2 = L_3 P_2 = 1$. Next, we choose $P_1 = 20$ resulting in $L_1 = L_2 P_1 = 20$. Similarly, solving the entire recursion results in $L_0 = 100$, with our final tiling factor choice of $P_0 = 5$.

III. The Ruby Framework

Reexamining Fig. 4, the mismatch between the number of PEs (6) and the tensor dimension (100) leads to an underutilization of PEs when employing PFMs ($100 = 5 \times 5 \times 2 \times 2$). However, expanding this mapspace through **imperfect factorization** can improve PE utilization, as shown by the imperfect factorization based mappings in Fig. 5. From Fig. 5, consider the highlighted mapping which ensures that all PEs are utilized for 16 cycles and only 2 PEs are unutilized for the last cycle. This saves 3 cycles when compared to the best mapping generated using perfect factorization, which would utilize 5 PEs over 20 cycles.

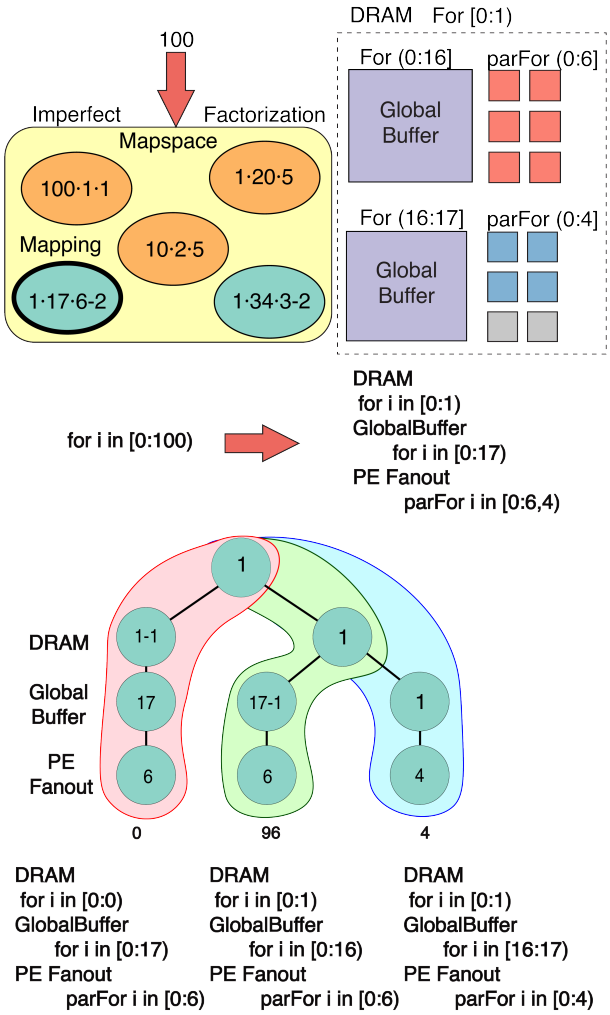


Fig. 5: A sample of the new mapspace with only spatial factors and remainders along with a highlighted example with temporal allocation inside the global buffer, and spatial allocation using all of the PEs for the first set of cycles, and fewer PEs for the last set. This saves 3 cycles in the toy example.

We reexamine the PFMs in eq (1) and Fig. 4 to understand how Ruby can automatically generate imperfect factorization based mapspaces. The product of the tiling factors (P_n) at each level should equal the dimension (D) of the given tensor (e.g., input channel C in a CNN). In other words, $D = \prod P_i$, which is also represented graphically in Fig. 6 (a). Based on the observation that $a \cdot b = (a-1) \cdot b + b$, we can ammend the previous equation to now include a remainder by substituting P_n for a and the remaining product for b. In other words, we can reformulate our equation as follows:

$$\begin{aligned} D &= \prod_{i=0}^{n} P_i, \\ &= (P_n - 1) \prod_{i=0}^{n-1} P_i + 1 \prod_{i=0}^{n-1} P_i. \end{aligned} \tag{2}$$

This reformulation is also shown graphically in Fig. 6 (b),

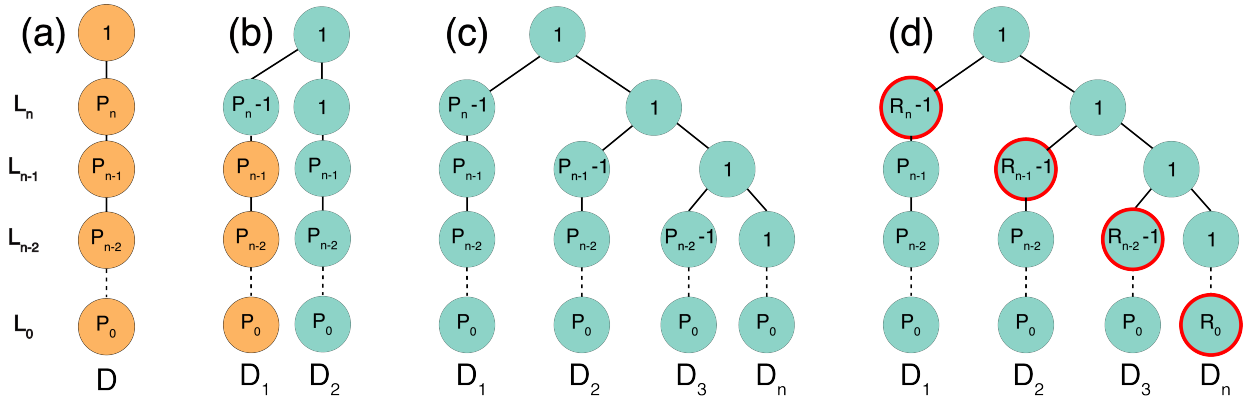


Fig. 6: **(a)** A graphical representation of eq (1) where D is the dimension size. **(b)** A transition step to the new representation. **(c)** The continuation of the recursion, equivalent to (a). **(d)** replacing the first leftmost leaf with R_n to add in the concept of remainders which leads us to eq (5). Note that the final leaf doesn't have a -1, seen in nodes containing R_0 and P_0.

where it is depicted as a branch on the tree. In this case the two branches are constructed such that the sum of their tiling factors equals D. Expanding upon the second term in eq (2) and generalizing, we derive:

$$D = (P_n - 1)\prod_{i=0}^{n-1} P_i + (P_{n-1} - 1)\prod_{i=0}^{n-2} P_i + \ldots + (P_{n-j} - 1)\prod_{i=0}^{n-j-1} P_i + \ldots + (P_0 - 1) + 1\prod_{i=0}^{-1} P_i. \quad (3)$$

In eq (3), the empty product $\prod_{i=0}^{-1}$ reduces to 1, leading to the more succinct:

$$D{=}1 + \sum_{j=0}^{n}(P_{n-j} - 1)\prod_{i=0}^{n-j-1} P_i. \quad (4)$$

Since $P_n - 1$, $P_{n-1} - 1$, ... are not prime factor of D, we refer to them as the remainder terms in this formulation, denoted by R_n (see Fig. 6). One example of this transformation with remainders applied to our mapping example from earlier is shown in Fig. 5. Each branch there corresponds to a split in the evaluation of the number of tiles or subtiles for that spatial or temporal level. Reexamining eq (1), we can apply the same principles used to derive eq (4) to determine a new recursion that allows the generation of imperfect factors:

$$L_n = L_{n+1}P_n + R_n - 1. \quad (5)$$

The inclusion of the -1 above modifies the base case to be $L_{n+1} = 0$ meaning now L_n represents the total number of tiles created at the n^{th} level of the memory hierarchy in all but the last iterations. R_n denotes the remainder term, which represents the tiling allocated on the last iteration for the n^{th} level in the hierarchy. Setting $R_n = P_n$ generates mappings identical to those produced by eq (1). Thus, while Ruby generates a superset of the PFM, the inclusion of a remainder term also results in additional mappings, the quality of which will be studied in this paper.

Similar to our previous example, Fig 5 shows mappings produced using both perfect and imperfect factorization. As the tree-like representation shows, there are distinct paths that correspond to each level in the memory hierarchy. The first path corresponds to the DRAM which has a tiling factor of 1 corresponding to a single tile which is transferred over to the GLB. The reformulation from eq (5) results in the product of the P_i in that branch being 0. The second branch, corresponds to the different iterations resulting from imperfect factorization at the GLB. Here, 17 tiles are created for further processing at the PE level with the first 16 being allocated to 6 PEs and the final being allocated to 4 PEs. The products along each branch sum up to 100 which is the original dimension being mapped.

This new example (Fig. 5) starts with the base case of $L_3 = 0$, and is followed by choosing P_2 still equal to 1, with $R_2 = 1$ representing the single iteration accessing the tiled tensor in DRAM. This results in $L_2 = 0 \cdot 1 + 1 - 1 = 0$. With the choice of $P_1 = 17$ and and $R_1 = 17$, we can calculate $L_1 = (17 \cdot 0) + 17 - 1$, resulting in 16 iterations. Finally, at the PE level $L_0 = (6 \cdot 16) + 4 - 1 = 99$, which together with the last loop iteration (1), results in $L_0 + 1 = D = 100$.

A. *Evaluating Ruby's Mapspace Expansion*

As shown in Fig. 6, transitioning from the PFM (Fig. 6 (a)) to the imperfect factorization mapspace generated by Ruby results in a much larger mapspace. However, simply expanding the mapspace is not sufficient, since this can make the search for a high-quality mapping intractable. Thus, it is worth examining how the expanded mapspace fares in terms of the density of high-quality mappings and investigate how constraining Ruby might ease search for improved mappings.

Timeloop's mapspace generator was modified to include all possible remainder combinations for these limited problems. The reduced problem size enables this exhaustive enumeration, which would be intractable for TA computations of interest

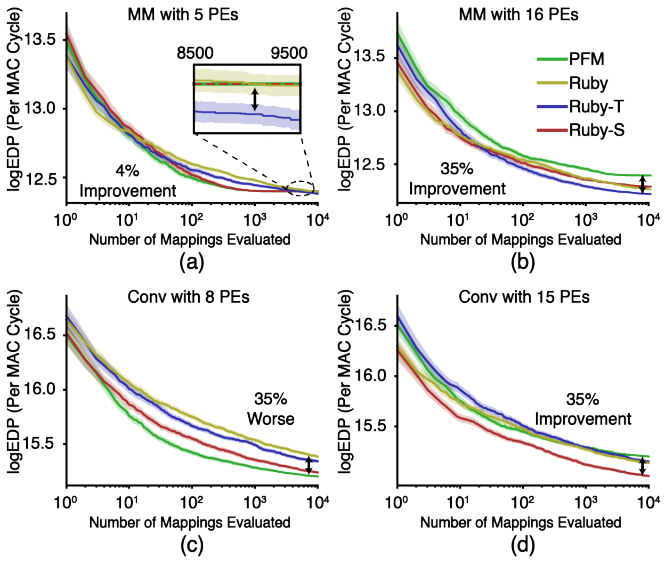


Fig. 7: Comparison of all the mapspaces over evaluating the first 10,000 mappings with different amounts of PEs and different layer types.

evaluated on realistic architectures. We refer to the entirety of the mapspace generated through imperfect factorization as *Ruby* (unconstrained mapspace). We also consider two natural constraints: a) imperfect factorization only at the spatial levels i.e., *Ruby-S* (-S for Spatially constrained) and b) only expanding the mapspace to include imperfect factorization on temporal levels i.e., *Ruby-T* (-T for the Temporally constrained).

We compare the quality of mappings produced by the PFM mapper in [16] against Ruby and its constrained variants for a two-level memory hierarchy toy architecture with each linear-PE allocated a 1 KiB scratchpad buffer. We study how different architecture configurations impact the mapping quality. In this study, we only evaluate the first $10,000$ generated mappings over 100 runs to average out the effect of the stochastic search algorithm (Timeloop's *Random Sampling*). The figures in Fig. 7, show the energy-delay product (EDP) achieved for the best mapping evaluated so far against the number of evaluated mappings. Because the problem size is constrained, we were able to evaluate all the mappings generated by Timeloop's mapper, while this was not true for Ruby and its variants. Despite that, the improvement to the schedule offered through a limited exploration of Ruby's mapspaces reaffirms its higher quality mappings when compared to those generated by PFM. Both scenarios with a mismatch between the problem size and the underlying hardware highlight this improvement for Ruby. When this mismatch is between the number of PE's and the problem size, Ruby-S's heuristic to improve utilization delivers improved mappings. Similarly, a mismatch between memory capacity and problem size showcases Ruby-T's improved reuse, lowering energy consumption. Despite the increase in mapspace size for Ruby and its variants, the improved mapping quality delivers schedules with consistently lower EDP (the targeted optimization metric).

In Fig. 7 (a) we examine how PFM compares to Ruby and its variants for a matrix multiplication problem over two tensors of size 100×100 for a design with 5 PEs. Due to the simplicity of this mapspace, fewer than $10,000$ PFM mappings exist and the entire mapspace is evaluated. Similarly, Ruby-S also generates fewer than $10,000$ mappings and converges to the results generated by the PFM. However, Ruby and Ruby-T incur a dramatic expansion of the mapspace, exceeding the $10,000$ limit. At the same time, results indicate that the larger unconstrained mapspace of Ruby can deliver better mappings if all the mappings can be evaluated as evidenced by the cross-over point between Ruby and the PFM (see inset).

Next, we introduce a mismatch between the tensor dimensions and the hardware resources by increasing the number of PEs to 16. Fig. 7 (b) shows the mappings generated for the same matrix-multiplication workload for this new configuration. Ruby-generated mappings outperform alternatives by better exploiting the mismatch between the tensor dimensions and hardware resources. The temporal reuse employed by Ruby and Ruby-T proves to be more impactful than the spatial utilization improvements through Ruby-S.

To investigate our proposed mapspace on a more complex problem, we examine how to map the convolution of a $3 \times 3 \times 64$ filter with a $28 \times 28 \times 64$ image onto the same toy architecture configured with 8 PEs. We impose an additional constraint that only C and M be mapped onto the PEs. The resulting mapspace (Fig. 7 (c)) shows that the PFM delivers high quality mappings due to the alignment between the hardware resources and the tensor dimension, bolstered by the PFMs limited mapspace. Ruby-S is able to approach this result due to its more constrained mapspace while Ruby and Ruby-T are uncompetitive. We examine the mapping quality under misalignment between hardware resources and the tensor dimension. Again, we modify the design to now have a misalignment between the hardware resources (15 PEs) and the tensor dimension to examine the mapping quality. The results in Fig. 7 (d), show that Ruby-S's moderate mapspace can take advantage of the mismatch between resources and the tensor dimensions to outperform PFMs while simultaneously easing search compared to the other examined imperfect factorization mapspaces.

While our previous results demonstrated that Ruby produces high quality mappings, just mapping quality is insufficient. So, we also evaluate how the different mapspaces scale with workloads to study the tractability of search in these mapspaces. Table I summarizes these results for mapping a tensor of rank 1 onto the previous architecture with 9 PEs. For the different tensor sizes, ranging from 3 – 4096, we generate the possible PFM combinations using eq (1) and further select only those mappings which are valid. For Ruby we employ eq (5), however the large mapspace renders further filtering unfeasible. However, the spatial constraints imposed on Ruby-S, prune mapspace branches that exceed a spatial factor of 9 at the PE level (e.g., as seen in Fig. 5) resulting in a more manageable mapspace. Mapspaces generated by Ruby and Ruby-T grow dramatically with increasing tensor size due to fewer constraints on them. Ruby-S offers a favorable trade-

TABLE I: Table of evaluating a single dimensional tensor over 2 levels of memory hierarchy, and a spatial fanout of 9 between these two levels.

Dimension Size	**3**	**64**	**100**	**1000**	**4096**
PFM (9)	3	22	24	52	43
Ruby-S (9)	7	50	61	117	83
Ruby-T	7	157	247	2392	10040
Ruby	11	414	690	5713	30612

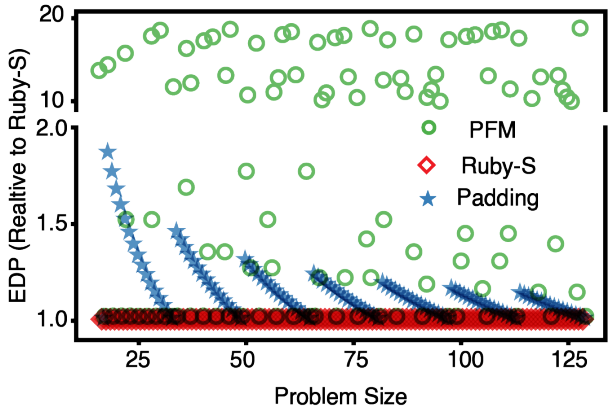


Fig. 8: Sweeping over different dimension sizes on a toy architecture (lower is better)

off between improved mappings and the mapspace expansion.

B. Evaluating Ruby-S

Since Ruby-S improves mappings through higher parallelism, we explore alternative strategies that can improve PFM performance. We compare Ruby-S to one such commonly employed strategy which pads tensor dimensions to better align the tensor dimension with the available hardware resources. The results of this comparison, normalized to Ruby-S, are shown in Fig. 8. We generate mappings for allocation of a single tensor, across 16 PEs in a linear array in our toy architecture. The padding strategy employed, pads the tensor up to the nearest number divisible by 16 (the size of the PE array). We do not use any datapath gating or skip memory reads to benefit from the sparsity, consequently performing ineffectual calculations due to the padding. On architectures that could effectively exploit single-operand, fine-grained sparsity, with negligible overhead, padding can deliver mappings that perform comparably with those generated by Ruby-S. Varying the problem size highlights the misalignment problem for PFMs, where at size $D = 127$ — a prime number, the computations cannot be parallelized beyond one of the available 16 PEs. This results in multiple fetches from main memory, leading to a worse EDP. However, padding by a single element changes the problem size to 128, which can easily be parallelized across 16 PEs over 8 cycles. This only incurs a minor cost due to the single ineffectual computation. At other problem sizes though, padding proves to be suboptimal, e.g., for $D = 113$, Ruby is able to maximize utilization while padding still incurs a 20% overhead in EDP since $\approx 12\%$ of the computations and memory accesses are due to zeroes introduced by padding.

C. Hardware Overhead of Using Ruby

Unlike conventional architectures which provide a hardware interface through an instruction set architecture (ISA), TA accelerators often expose unique configurable hardware settings. Among them, loop bounds and strides are typically implemented through pattern generators implemented as finite state machines. A minor augmentation to such a state machine can accommodate the requirement for a different final loop. This static configuration adds no extra penalty in terms of complexity, energy, or cycles in our performance evaluation.

IV. Ruby-S Mapping Peformance

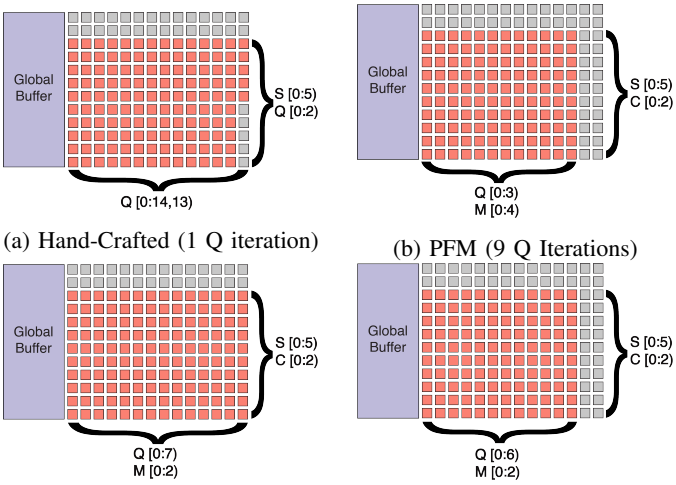
A. Evaluation Setup

We expand on our previous results, to show that Ruby-S also outperforms PFM mapspace generation on realistic architectures, such as our baseline [2]. This baseline and other evaluated models are based on the designs provided in the *Timeloop+Accelergy Exercises* repository[1]. This includes files for modeling both an Eyeriss-like and Simba-like architecture. For our baseline Eyeriss-like architecture we, constrain the mapspace to generate mappings that conform to the data access patterns amenable to row-stationary dataflows. Additionally, in subsequent simulations, we direct the mapping evaluation to optimize for EDP and use the same Timeloop+Accelergy framework employed by previous work [18, 29, 34]. While Timeloop can evaluate for other optimization functions such as energy or delay separately, EDP encapsulates the benefits and drawbacks of improved PE utilization, balancing the latency reduction with the increased energy. We employ the same evaluation methodology as prior work [18, 29, 34], energy is evaluated using Accelergy through plugins that evaluate access counts generated from Timeloop to obtain energy for large memories through Cacti [35] and smaller components such as address generators and register files using numbers included in Aladdin [36]. Delay is represented in terms of cycles, normalized to the delay incurred for a MAC. To disentangle mapspace generation from the search heuristics we only employ Timeloop's random sampling based search across all our tests. We specify the terminating conditions for the search to be 3000 consecutive valid mappings that do not improve EDP across 24 threads.

B. Evaluating Ruby-S against Handcrafted Mappings

Although PFMs can be used to generate and evaluate a vast mapspace, generally outperforming humans there are still edge cases where handcrafted mapspaces exceed PFM quality. One such case occurs for layer 2 of AlexNet (IFM: $27 \times 27 \times 48$ and weights: $5 \times 5 \times 96$), where the *Strip Mining* algorithm as described in [2] outperforms PFMs. Consequently, we evaluate whether the improved mapspace generated by Ruby-S outperforms these known cases where hand-crafted mapping quality for this edge case. The handcrafted mapping shown in Fig. 9 (a) maps an entire row (Q) of the OFM to the PE array, completely evaluating the row before proceeding with

[1]https://github.com/Accelergy-Project/timeloop-accelergy-exercises



(a) Hand-Crafted (1 Q iteration)

(b) PFM (9 Q Iterations)

(c) Ruby-S (Q Iterations 1-3)

(d) Ruby-S (Q Iteration 4)

Fig. 9: Comparison of different mappings for layer two of AlexNet on our baseline architecture.

the computations required for the next row. Computing the next row incurs accesses that fetch new inputs and parameters from the GLB. The PFM, shown in Fig. 9 (b), factors the OFM dimension (Q $D = 27$) into 3 and 9 to generate its mapping. This splits Q into 9 groups of size 3, replicated $4\times$ over the output channel dimension (M). This incurs 9 accesses to each row of Q to complete the partial sum. These mappings generated by PFMs underutilize the PEs. While the handcrafted mapping achieves 85% utilization, the PFM only achieves 71% utilization, increasing the latency and consequently the EDP to evaluate that layer. Ruby-S generates a new mapping that creates 4 groups along the Q dimension in two phases shown in Fig. 9 (c) and (d). Ruby-S replicates groups from Q of size 7, $2\times$ over the M dimension. However, on the final iteration, Ruby-S only requires a group of size 6 (Fig. 9 (d)), necessitating a net 4 accesses to each row along Q. This new mapping maintains the same PE utilization as the handcrafted mapping (85%) with a 16% decrease in EDP and a 10% decrease in energy. Examining the mappings suggests that these energy savings are primarily due to fewer accesses to the GLB.

C. Evaluating Ruby-S on Different Architectures

Figure 10 summarizes the results from mapping ResNet-50 [38] using Ruby-S on our baseline architecture. These results include EDP, energy, and cycle count for each layer-type, normalized to the PFM mapspace. The final column summarizes the results obtained from implementing ResNet-50 in its entirety, with Ruby-S delivering a 14% improvement in EDP owing to a 17% reduction in cycles despite a 2% increase in energy due to the increased utilization. These improvements are primarily due to better mappings generated for pointwise and dense layers, whose dimensions are typically misaligned with the 14×12 array. For these layers, the mappings generated by Ruby-S benefit from a higher degree of parallelism obtained through weight replication.

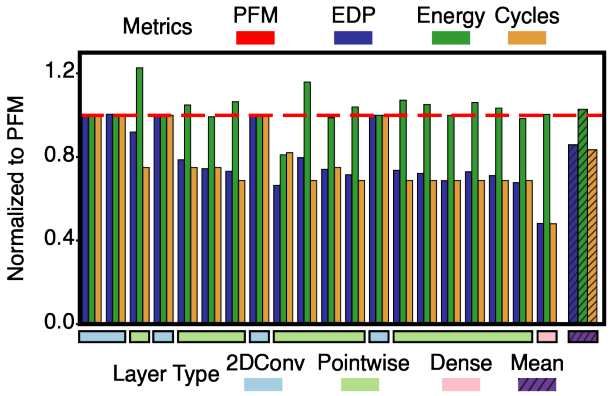


Fig. 10: Comparing Ruby-S to the PFM mapspace over different ResNet-50 layers on an Eyeriss-like architecture [2].

We also demonstrate the general applicability of Ruby-S by comparing it against PFMs for the Renet-50 workload on a Simba-like architecture [3]. This architecture differs from our previous baseline as its PE contains a vector MAC along with a shared local weight, input, and accumulation buffer. The data access pattern supported by this architecture allows for PE-level parallelism across the input channel (C) and output channel (M) dimensions. We choose a configuration with 15 PEs to better understand the performance gains that might be derived from Ruby-S. We retain Simba's PE-level data access patterns with the architecture configured to four, 4-wide vector MACs and dedicated memory banks. Our results in Fig. 12 show Ruby-S delivering a 10% net improvement in EDP over the mappings generated by PFMs for ResNet-50. Several layers see an improvement of up to 25% in EDP, however the complexity of the architecture also results in some suboptimal mappings as seen in layer 1 in this figure. For a configuration of 9 PEs, each with three, 3-wide vector MACs, Ruby-S delivers a 45% improvement in EDP over PFM mappings for ResNet-50.

D. Evaluating Ruby-S for DeepBench

We evaluate Ruby-S's performance using the DeepBench suite [37] to better encapsulate how spatially-constrained imperfect factorization might map different tensors to our baseline accelerator. Deepbench includes a wide variety of common workloads such as vision, speech, and facial recognition tasks using convolutional and dense layers. The diversity of tensor sizes across these tasks ensures that hardware resources cannot be tailored to the tensor sizes. The original 14×12 of PE array of Eyeriss was designed due to the common division of 7 in the activation size of 224×224 in Imagenet workloads [39]. While this architecture can support an arbitrary sized convolutional layer for Imagenet, PFMs cannot optimally find a high quality mapping when the activation size and array size are misaligned.

The results, shown in Fig. 11 demonstrate a dramatic improvement across different benchmark tasks. Since the baseline architecture targeted CNNs, Ruby-S's performance

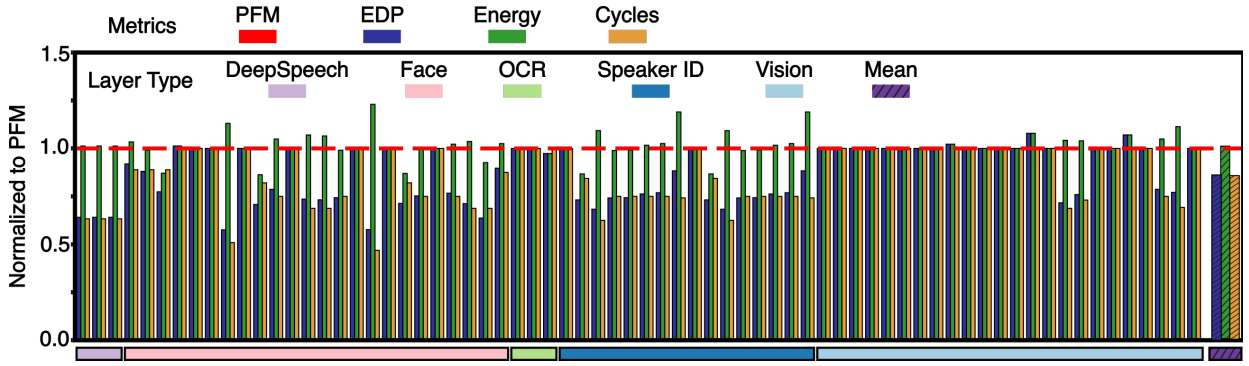


Fig. 11: Comparison between mappings generated by Ruby-S, normalized to PFM over a selection of workloads from DeepBench optimized for EDP [37].

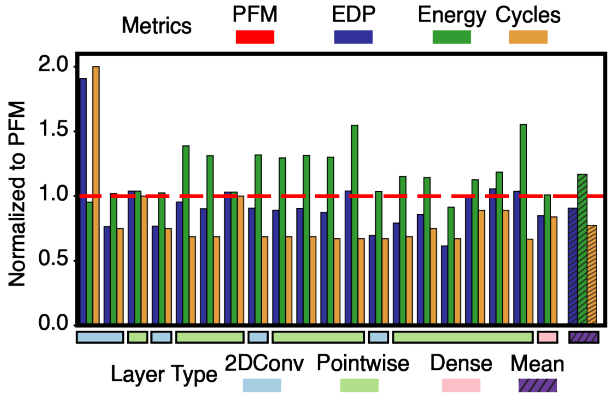
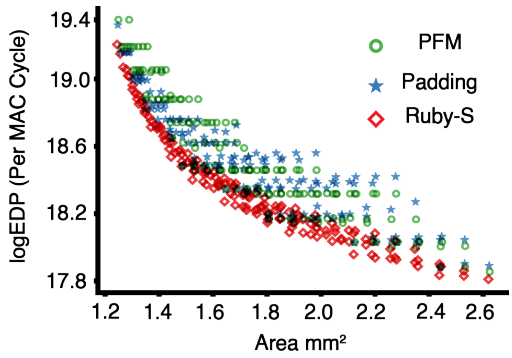


Fig. 12: Comparing Ruby-S to the PFM mapspace over different ResNet-50 Layers on a Simba-like architecture [3]. The channel size of 3 allows the PFM to map this tensor across the 15 PEs. However, Ruby-S's mapspace grows untenably complex due to Simba's architecture, impacting search.

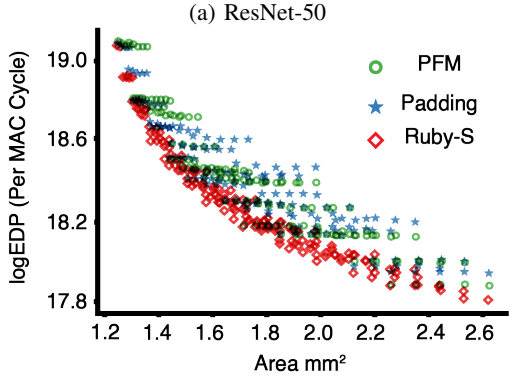
almost always matches that of the PFM on most of the vision layers. However, on workloads where the filter and feature map sizes do not map well onto the 14×12 PE array, Ruby-S is able to deliver mappings that are up to 33% lower in EDP. Ruby-S performance is comparable to the PFM on vision workloads due to the prominence of benchmarks using Imagenet. PFM is able to factorize feature maps based on the factor 7 occurring in both the feature map dimensions and the PE array size (14×12). However, across other domains such as face recognition, speaker identification, and speech-to-text, the variety in tensor sizes (e.g., DeepSpeech layer 1 IFM is $341 \times 79 \times 32$ and a filter is $5 \times 10 \times 32$.) yields greater opportunities for Ruby-S to exploit. Across these workloads, Ruby-S prioritizes mappings that improve the utilization. Mappings generated by Ruby-S decrease the EDP by an average of 10% across the entire benchmark suite. When targeting latency instead of EDP, Ruby-S generates mappings that reduce the latency 14% compared to PFMs.

E. Architectural Design Space Exploration

Next, we examine the interaction between different architectural configurations and the mapspace, when optimizing for EDP. Accurately evaluating this interaction can provide insight



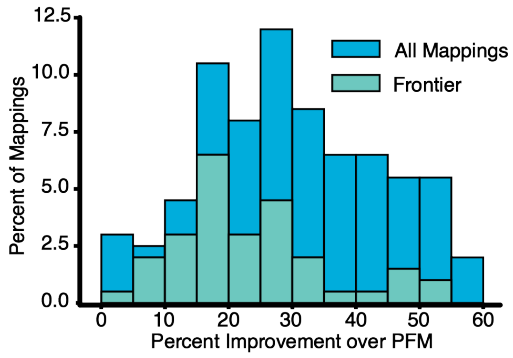
(a) ResNet-50



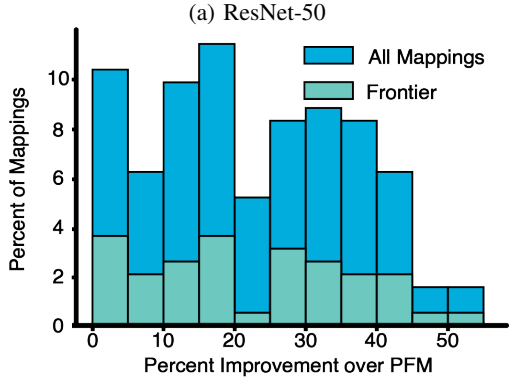
(b) Subselection of DeepBench Workloads

Fig. 13: Sweeping different configurations of an Eyeriss-like from an array size of 2×7 to 16×16 over ResNet-50 (a) and DeepBench (b). Ruby-S forms a Pareto curve improving the EDP of previous designs.

into architectural design choices and enable co-design. We evaluate Ruby-S against Timeloop's PFM with and without padding over various PE array sizes ranging from 2×7 to 16×16 and evaluate the resulting EDP for various workloads. As seen from Figs. 13a and 13b, mappings generated by Ruby-S form the Pareto frontier when examining the trade-off between accelerator area and EDP for ResNet-50 and the DeepBench suite respectively. Ruby-S consistently improves upon the alternative mapping strategies. For ResNet-50 some architectural configurations see performance improvements of 60%, with those on the Pareto frontier seeing improve-



(a) ResNet-50



(b) Subselection of DeepBench workloads

Fig. 14: Evaluating different PE configurations for an Eyeriss-like architecture (2×7 to 16×16) for ResNet-50 (a) and DeepBench (b). Ruby-S improves performance while forming a new Pareto frontier.

ments of 50%–55% (see Fig. 14a). Across all architectural configuration, we observe an average improvement of 24%. Investigating similar trade-offs for a subset of the DeepBench suite, shows that Ruby-S improves upon the PFM results by up to 55%, with those on the Pareto frontier improving by an average of 20% as shown in Fig. 14b.

V. Conclusion

We develop a new mapspace (Ruby) that uses imperfect factorization to generate high quality mappings on user-defined architectures. We analyze the trade-off between mapspace expansion and the quality of mappings generated and develop a subset of Ruby, Ruby-S. Ruby-S expands the mapspace using spatial constraints to balance mapping quality with mapspace expansion. On an Eyeriss-like architecture Ruby-S reduces EDP by up to 50% for ResNet-50 and up to 40% on an NVDLA-like architecture. Furthermore, Ruby-S yields improvements of up to 45% with an average improvement of 10% on DeepBench. Ruby-S also is robust to accelerator configurations and improves EDP by 20% on average, with a maximum improvement of 55% when implementing ResNet-50 on different designs. Ruby-S expands the Pareto-frontier of these achieved by these designs by an average of 30% and 20% for ResNet-50 and DeepBench respectively.

We plan to release the modifications to Timeloop publicly. We are currently working to migrate our code to be fully integrated in future releases of *Timeloop* [2].

References

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

[2] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[3] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–27. [Online]. Available: https://doi.org/10.1145/3352460.3358302

[4] J. N. Kutz, "Deep learning in fluid dynamics," *Journal of Fluid Mechanics*, vol. 814, pp. 1–4, 2017.

[5] J. Behler, "Perspective: Machine learning potentials for atomistic simulations," *The Journal of chemical physics*, vol. 145, no. 17, p. 170901, 2016.

[6] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[7] V. Sitzmann, J. Thies, F. Heide, M. Nießner, G. Wetzstein, and M. Zollhofer, "Deepvoxels: Learning persistent 3d feature embeddings," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2437–2446.

[2] Currently available as a test branch at https://github.com/NVlabs/timeloop

[8] O. Kuchaiev, J. Li, H. Nguyen, O. Hrinchuk, R. Leary, B. Ginsburg, S. Kriman, S. Beliaev, V. Lavrukhin, J. Cook, P. Castonguay, M. Popova, J. Huang, and J. M. Cohen, "Nemo: a toolkit for building ai applications using neural modules," 2019.

[9] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018. [Online]. Available: http://arxiv.org/abs/1801.04381

[10] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 137–151. [Online]. Available: https://doi.org/10.1145/3297858.3304025

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[12] Y.-H. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017.

[13] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," 2018.

[14] M. Tan and Q. V. Le, "Efficientnetv2: Smaller models and faster training," 2021.

[15] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, "Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021.

[16] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[17] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 807–820. [Online]. Available: https://doi.org/10.1145/3297858.3304014

[18] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," *CoRR*, vol. abs/2105.01898, 2021. [Online]. Available: https://arxiv.org/abs/2105.01898

[19] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 754–768. [Online]. Available: https://doi.org/10.1145/3352460.3358252

[20] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 369–383. [Online]. Available: https://doi.org/10.1145/3373376.3378514

[21] R. Frostig, M. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," 2018. [Online]. Available: https://mlsys.org/Conferences/doc/2018/146.pdf

[22] Y. Chen, J. S. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *CoRR*, vol. abs/1807.07928, 2018. [Online]. Available: http://arxiv.org/abs/1807.07928

[23] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, p. 751–764, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093337.3037702

[24] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–284. [Online]. Available: https://doi.org/10.1145/2541940.2541967

[25] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, T. Krishna, and V. Sarkar, "MARVEL: A decoupled model-driven approach for efficiently mapping convolutions on spatial DNN accelerators," *CoRR*, vol. abs/2002.07752, 2020. [Online]. Available: https://arxiv.org/abs/2002.07752

[26] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "Dmazerunner: Executing perfectly nested loops on

dataflow accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358198

[27] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, and Y. Liang, "Tenet: A framework for modeling tensor dataflow based on relation-centric notation," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 720–733. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00062

[28] S.-C. Kao and T. Krishna, "Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[29] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: Enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 943–958. [Online]. Available: https://doi.org/10.1145/3445814.3446762

[30] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991, pp. 30–44.

[31] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE transactions on computers*, vol. 35, no. 01, pp. 1–12, 1986.

[32] J. Wang and J. Cong, "Search for optimal systolic arrays: A comprehensive automated exploration framework and lessons learned," 2021.

[33] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2017, pp. 1–6.

[34] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2019.

[35] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '11. IEEE Press, 2011, p. 694–701.

[36] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. IEEE Press, 2014, p. 97–108.

[37] DeepBench, "http://www.github.com/baidu-research/deepbench."

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.