QuickFlow: An Efficient Local Search Method to Map Convolutions on Spatial Architectures

Marco Ronzani **Cristina Silvano**
DEIB, Politecnico di Milano, Italy

2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 979-8-3315-1560-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICCAD66269.2025.11240877

***Abstract*—Efficiently running deep neural networks requires the hardware acceleration of convolutional kernels. Spatial Architectures (SAs) are a natural fit, employing multiple processing elements and a custom memory hierarchy to exploit parallelism and data reuse. In turn, SAs require a mapping to specify data movements and computation order. Thus, specialized mapping tools have been developed to explore the space of possible mappings and retrieve optimal ones, using analytical hardware models for performance feedback. However, for each SA-kernel pair, the mapping space is vast, with significant performance variations arising from subtle interactions between mapping decisions. Therefore, coordinating all problem aspects remains challenging for existing tools, often leading to long execution times. Yet, high-quality mappings must be promptly available to support downstream tasks, like runtime resource allocation and hardware design space exploration. To address this, we propose QuickFlow, a new mapping tool that efficiently finds near-optimal mappings for SAs and AI kernels. In QuickFlow, we build equivalences between mappings based on a detailed analysis of distinct data reuse opportunities. Then, we redesign the optimization paradigm accordingly, comparing tiling and parallelism decisions after quickly selecting their best dataflow up to equivalence, ultimately enabling a single greedy local search to effectively reach near-optimal mappings. QuickFlow also integrates an improved analytical model, supporting arbitrary convolutions through a novel, exact formula for tile sizes. Across our experiments on three SAs and twenty kernels, QuickFlow achieves a 1–2.1× better energy-delay product and up to 182× faster execution time compared to the best results from four state-of-the-art mapping tools.**

***Index Terms*—DNN accelerators, spatial architectures, analytical modeling, data reuse, mapping, map-space exploration**

I. Introduction

Deep learning hardware accelerators are ubiquitous in modern edge and high-performance devices alike, enabling them to run deep neural networks within reasonable power and latency budgets. Their typical workloads consist of convolutions, or general tensor contractions, simple and regular kernels based on Multiply and Accumulate (MAC) operations that deal with very large operands, resulting in high storage, memory bandwidth, and compute requirements. Hence, domain-specific accelerator designs must focus on parallelism and optimized data movements [1, 2].

Spatial Architectures (SAs) are a common and well-proven design in this domain, implementing a multidimensional array of Processing Elements (PEs) to concurrently perform MAC operations and a hierarchy of memories to efficiently supply PEs with data. Examples include Eyeriss [3], Simba [4], and Google TPUs [5, 6]. SAs are flexible, supporting many data storage, movement, and computation patterns, thus a **mapping** is required to specify exactly how a given workload runs on the hardware. For any SA and kernel pair, the **map-space** is the set of all possible mappings [7].

In an era defined by the memory wall, the cost of memory accesses and data transfers has come to far exceed that of computation [1, 6]. That's why SAs employ memories varying in size, bandwidth, and access costs, but, in turn, require a pondered selection of the mapping that best exploits **data reuse** on their hardware. So, how do we run an arbitrary kernel on a SA with minimal energy and latency? Such is the **mapping problem** challenged throughout this work. Its complexity arises from the sheer size of, and performance variation across, map-spaces, leading to the creation of dedicated mapping tools to search them for optimal mappings. Such tools comprise a **mapper** running various optimization routines and operating in a feedback loop with a fast **analytical model** of SAs [8, 9]. Traditionally, the mapping problem is solved once per kernel and at static time, but ongoing research has highlighted two key applications that involve solving it repeatedly or at runtime. For hardware design space exploration [10-12], many different SA configurations can only be compared fairly once optimized mappings are known for them all. In multi-tenant scenarios [13, 14], an accelerator's resources are shared by multiple independent workloads that must be mapped on the fly. Thus, to enable both use cases, mapping tools need to be fast, retrieving near-optimal mappings in seconds.

To this end, mappers must exploit any attainable prior knowledge of data movement and reuse patterns to narrow the search space effectively. Without enough guidance, the problem becomes intractable in reasonable time, forcing reliance on **heuristics** that trade off mapping quality for speed. In this sense, current tools combine a few prior insights with an array of optimization techniques. Timeloop [8] introduces a robust SA analytical model that profoundly inspired this work, it then employs a random search-based mapper with superfluous loop orderings pruning. GAMMA [15] is a genetic algorithm-based mapper paired with the MAESTRO [16] model, it refines mappings by combining the best ones built at each generation. Based on ZigZag's model [10], the LOMA mapper [17] places each prime factor of the kernel dimensions on its own loop, merges some to prune the space, and explores all their orderings, only later allocating loops to memory levels in a single pass, striving to maximize reuse. SALSA [18] improves on LOMA with simulated annealing to optimize the loops order. CoSA [19] solves the mapping problem through a mixed integer linear programming formulation, while LOCAL [20] maximizes parallelism and greedily fills memories inside-out to improve reuse. Forgoing problem details for speed, both tools risk to land far from optimal mappings. Ceiba [21] uses reinforcement learning to train a small transformer network for each map-space until it produces optimal mappings. The technique shows remarkable sample efficiency but is slowed down by the overhead of training the network. Our previous work, FactorFlow [9], quickly yields near-optimal mappings for matrix multiplications, addressing mapping decisions with three nested heuristics, a local, a selective, and an exhaustive search, and demonstrating that greedy techniques can be viable once the map-space complexity gets low enough.

On simpler map-spaces, present strategies yield adequate compromises between efficiency and quality of results. However, they often struggle to scale to more complex SAs and kernels or to be fast enough for hardware exploration and on-the-fly mapping scenarios. In general, this is because such tools assume that a trade-off is inevitable, instead of seeking deeper prior insights to consistently reduce the problem's complexity. Finally, when handling less regular kernels, like batches of strided and dilated convolutions, the speed and flexibility of existing analytical models face limitations as well.

Consequently, our idea is to study how data reuse emerges in mappings and correlate mapping decisions with reuse opportunities, relying on such insights to define equivalences and other relations between similar mappings. Then, we leverage these reuse-based patterns to infer a structure over the map-space and limit the scope of exploration to only mappings with **distinct reuse opportunities**; we also embed such knowledge within our SAs analytical model, improving its speed and versatility on highly dimensional kernels. Ultimately, we show that the mapping problem can be simplified enough to be efficiently tractable with a single **greedy local search**, achieving near-optimal results with a time complexity that scales polynomially in the complexity of SAs and kernels.

These studies culminate in the development of QuickFlow, a new mapping tool that rethinks the optimization paradigm. Where in preceding tools the selection of dataflows is handled first and takes the most effort, in QuickFlow we have our local search managing the allocation of kernel dimensions prime factors as the outermost, sole, exploratory step and dataflows addressed in one-shot inside it.

In this paper, we present and demonstrate novel insights and techniques to deal with the mapping problem of arbitrary AI kernels on spatial architectures. In particular, our **contributions** are:

- **(1)** A detailed analysis of how reuse opportunities emerge in the loop nest representation of mappings, leading to the identification of relations that enable the simplification of map-spaces without any loss of unique solutions.
- **(2)** The introduction of a new local search-based heuristic that is capable of single-handedly solving the mapping problem. In it, we handle factors allocations first, through a multi-hop local beam search of adjacent mappings, enabling a one-shot selection of the loop permutations yielding the most reuse for any mapping, which then ensures fair comparisons during the search. Running the process for three rounds achieves a co-optimizing of tiling and parallelism decisions.
- **(3)** **QuickFlow (QF)**, a new mapping tool that implements the above heuristic. To allow QF to model any general tensor contraction kernel while retaining a fast and accurate analytical model, we devise an incremental scheme and a novel exact formula to compute tile sizes that fully support stride and dilation. The resulting model is 3.6× faster than the current academic standard while matching its accuracy.
- **(4)** We evaluate QF across sixty map-spaces and against four state-of-the-art mapping tools, noting that it consistently improves on them, finding 1-2.1× (1.2× avg.) better mappings by EDP with an up to 182× (24× avg.) lower search time, highlighting our local search's effectiveness and efficiency.

This paper is organized as follows: Sec. I provides the context and motivation for our work. Sec. II introduces some basic notions for AI kernels and SAs, while Sec. III goes over the structure of mappings and their link with reuse. Then, Sec. IV presents QuickFlow, with its new analytical model and mapping technique, and Sec. V discusses our experiments and results. Finally, Sec. VI concludes this writing.

II. Preliminaries

A. AI Kernels

Convolutions extend the notion of matrix multiplication to higher-dimensional arrays with linear index arithmetic. They operate by sliding filters (weights tensors) over an input tensor and aggregating local information via weighted sums. Many variants exist, like 1D and 2D, with batches, or transposed ones. A typical 2D convolution without batches, from here on referred to as *simple*, is expressed as

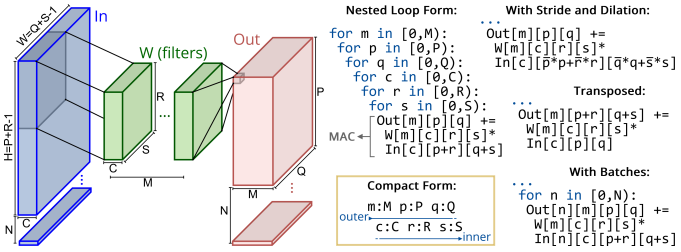

Fig. 1: Loop nest of a simple convolution and its variants.

$$\mathrm{Out}_{p,q,m} = \sum_{r,s,c} \mathrm{In}_{\overline{p}\cdot p+\overline{r}\cdot r,\, \overline{q}\cdot q+\overline{s}\cdot s,\, c} \cdot \mathrm{W}_{r,s,c,m}$$

with tensors $\mathrm{Out} \in \mathbb{T}^{P\times Q\times M}$, $\mathrm{In} \in \mathbb{T}^{H\times W\times C}$, $\mathrm{W} \in \mathbb{T}^{R\times S\times C\times M}$, where $H = \overline{p}\cdot(P-1)+\overline{r}\cdot(R-1)+1$ and $W = \overline{q}\cdot(Q-1)+\overline{s}\cdot(S-1)+1$, while positive integers $\overline{p}, \overline{q}$ are the strides and $\overline{r}, \overline{s}$ the dilation factors.

From a computational perspective, we can expand the summation into a loop nest around a MAC operation, in doing so obtaining the main representation used throughout the hereby work, habitually called polytope model [22]. A simple convolution requires a nest of six loops going over eight dimensions, that we write with an output-centric notation [8], see Fig. 1.

Let us call *affine index* one that arises from an affine transformation, that is the integral linear combination of two or more underlying indices native to their tensor dimensions. These indices implement the sliding filter mechanism characteristic of convolutions. Henceforth, let us use the term *affine dimension* (as opposed to *normal*) to refer to tensor dimensions traversed by affine indices. In our notation, the input's width (W) and height (H) are affine dimensions addressed by affine indices summing together an index of the output (p or q) and one of the weights (r or s), then, the transformation's coefficients are the strides, for indices going over output dimensions, and dilation factors, for weights-related ones. The lack of stride or dilation implies that such coefficients are 1.

Moreover, if an operand depends on a dimension's index, we refer to the two as *coupled*. Vice versa, we say that an operand is *orthogonal* to a particular dimension if it does not carry an explicit index corresponding to that dimension. For instance, in convolutions, the input tensor is orthogonal to the output channels dimension (M). Computationally, this is a powerful property, because it means the same values of an operand are reused across all iterations on any loop that goes over one of its orthogonal dimensions.

B. Spatial Architectures

SAs are a class of hardware accelerators designed to exploit high compute parallelism by using an array of relatively simple processing elements. Each PE is specialized to perform MAC operations and its storage usually consists of a few registers, PEs are then progressively connected with each other and a memory hierarchy through busses or a network-on-chip. The memory hierarchy is explicitly managed and typically consists of multiple on-chip buffers like Register Files (RFs), SRAM scratchpads and FIFOs, backed by a large off-chip DRAM and non-volatile memories. Implementations of SAs, as defined in [3], exist as ASICs, on CGRAs, and FPGAs [1].

Key performance metrics for a SA are its energy, latency, and Energy-Delay Product (EDP) when running a given workload. Being far and with limited bandwidth, the energy and latency required to access larger memories, like DRAM, dominate those of computation, being hundreds of times more than what's needed for storages near

PEs [6]. That's why a SA's memory hierarchy is intended to localize most repeated value accesses on faster and more efficient on-chip memories, exploiting data reuse to minimize costly accesses.

Hereafter, we model SAs as a hierarchy of levels [8, 9, 16]:

- **Memory Level**, denotes a piece of the memory hierarchy, characterized by technology, size, bandwidth, access energy and latency, and which operands it stores or bypasses;
- **Fanout Level**, represents the replication into a given number of instances of all subsequent levels, that proceed to operate concurrently. It also models the interconnects between such inner levels and outer ones, and specifies which kernel dimensions can be parallelized;
- **Compute Level**, standing for a functional unit capable of performing one or more MAC operations within a certain amount of clock cycles and energy usage.

In particular, we consider fanout and compute levels to be *spatial* levels, as both can cause multiple computations to unfold concurrently across multiple instances of themselves or subsequent levels, therefore giving SAs their name.

The mechanisms that enable reuse in hardware are multicast and reduction. Spatial levels can support spatial multicast as one read from an outer memory being used for multiple writes to inner instances, and reduction, where reads from inner levels are accumulated in a single outer write. These can be implemented either with PE-to-PE forwarding, like in systolic arrays, or on the interconnect during memory accesses [1]. Reuse on memory levels is called temporal reuse, where the same value is retained in a memory while being read (multicast) and/or updated in place (reduction) multiple times without it being re-fetched from an outer memory [16].

III. Analysis of Mappings and Reuse

For a pair of kernel and SA, a **mapping** determines the allocation of operands throughout the memory hierarchy and plans their movements down to the binding of each MAC operation to a PE. The **map-space** is the set of all possible mappings, but not all are valid; a usable mapping must respect all constraints derived from limited hardware resources or defined by the user. Therefore, the **mapping problem** is a constrained optimization problem, and solving it requires devising a procedure to search the map-space for a valid and optimal mapping w.r.t. energy and/or latency [8, 9].

A. Mapping Representation

Starting from the nested loop representation for a kernel, mappings are represented by replicating each loop once for every level of the SA. Then, for each original loop, its number of iterations, that being the size of its target dimension, is broken down in its **prime factors** (hereafter just called "factors" for brevity), and those are distributed among all the copies of that loop. Fig. 2 reports an example.

Loops on a memory level unfold over time, and at any point during execution, each memory in the hierarchy must store all the values used in upcoming iterations on its loops or inner ones. Instead, loops on spatial levels play out all at once, in parallel across multiple hardware instances, each seeing a different portion of the workload. Notation-wise, let "λ" denote loops on spatial levels.

In this form, tile sizes, that is, the extent of the dimensions of the operand tiles that each memory must retain at any given time, are implicit. However, they can be easily calculated on any normal dimension as the product of all iterations on such dimension at the present and inner levels. While for affine dimensions, that lack a dedicated index, the matter is more involved, as their tile size is the count of distinct values that their affine index can assume during

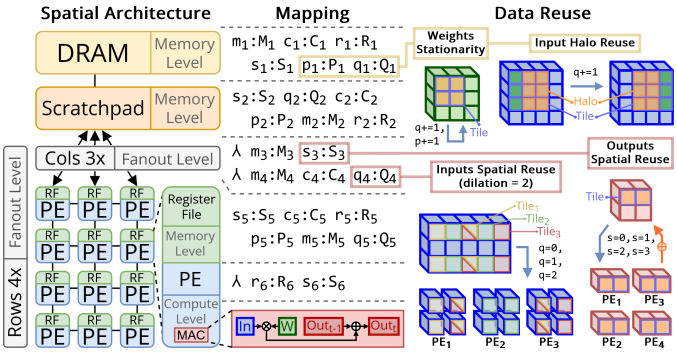

Fig. 2: SA with mapping representation and reuse opportunity examples.

iterations at the present or inner levels. Sec. IV-A discusses how to compute such count in the general case, but for now, when there's no stride nor dilation, that is equivalent to the sum of tile sizes of all dimensions involved in the affine dimension's index minus a unit for each after the first one. Then, the volume of an operand's tile stored on a memory level is the product of tile sizes across its dimensions. From this, we can observe a side-effect of fanout levels: they "eat up" some iterations, leading to smaller tile sizes on inner memories for all parallelized dimensions. For operands coupled to such dimensions, the resulting smaller tiles will differ in content between instances, while tiles of operands orthogonal to them will be mirrored in each instance, a fact that affects reuse, as we'll see.

B. Mapping Decisions

A mapping can be broken down into three decisions, all critical to its performance, but that involve different considerations [15]:

- **Tiling**, to fit on the limited size of inner memories, operands need to be broken down into progressively smaller tiles. This is controlled by allocating prime factors on inner memory levels, progressively reducing their dimensions' tile sizes.
- **Parallelism Strategy**, the dual of tiling, allocating iterations on spatial levels. This determines by how much and along which dimensions is the kernel parallelized.
- **Loop Ordering**, each per-level copy of the kernel's loop nest can have its loops arbitrarily reordered. Any permutation may bring a different order of computation and data accesses.

Tiling and the parallelism strategy determine which tiles of each operand need to be available when and where, while loop ordering schedules computations and data movements between levels.

C. Reuse Opportunities

A SA may offer the reuse mechanisms described in Sec. II-B, however, it is the mapping that selects the ones to exploit. Tiling predisposes data in memories, determining the potential of temporal reuse. The choice of parallelism strategy dictates spatial reuse. Lastly, the order of loops determines the mapping's dataflow, that is, when and which operands get to exploit reuse and the actual data transfers that are still required afterward.

We identify three main classes of reuse, that comprise all dataflows in present literature [3, 8, 16] (examples in Fig. 2):

Stationarity: Each operand can be temporally reused (remains stationary) on a memory level across all back-to-back innermost iterations that occur on dimensions it is orthogonal to. In other words, so long as a level's innermost loops don't move an operand's indices, its accessed tile remains constant throughout them.

This is the most common type of reuse, as well as the one sparing the most memory accesses. In the case of a simple convolution,

where each dimension is orthogonal to exactly one operand, there can only be one stationary operand per memory level, and ideally, all the dimensions it is orthogonal to should be the innermost iterated ones, maximizing reuse.

Halo Reuse: On memory levels, consider now for each operand the innermost loop on a dimension coupled to it, if that is an affine dimension, the operand's accessed tiles between subsequent iterations will overlap, and the resulting common elements can be reused. Put differently, from one outer iteration to the next, an inner memory needs to only be provided with the non-overlapping part of its operands' tiles, while the rest remains stationary. This occurs because when just one index within an affine index is incremented, the offset may not be enough to clear the tile size of the affine dimension, hence, some values of the affine index may repeat during inner iterations even after one of its components changed. The amount of overlap depends on the coefficients of the affine index (stride and dilation) and tile sizes on the present level.

Spatial Reuse: The basic spatial reuse cases occur when a non-affine dimension is parallelized, there, each instance operates on the same tile of operands orthogonal to the spatially-mapped dimension, while instead storing a strictly distinct tile of operands coupled to it. When an affine dimension appears on spatial levels, however, just like in halo reuse, its coupled operand's tiles seen by each instance will overlap, and common elements can be spatially multicasted or reduced accordingly. Though, in this spatial case, multicasts and reductions happen all at once, e.g. with memories outside the spatial level supplying all inner tiles in one big sweep or outputs being accumulated while traveling through PEs. If the index coefficients for the spatially mapped affine dimension are all one, this results in the same amount of memory accesses as halo reuse would. Otherwise, the total amount of reuse may be more than individual overlaps, because gaps in the overlap of strictly subsequent tiles may get reused one or more tiles down the line, hence, all tiles considered together can retain the maximum amount of overlap by exploiting non-contiguous tiles, as demonstrated in Fig. 2.

In light of this, loop ordering on spatial levels does not matter for mappings, only the parallelism strategy does.

D. The Map-Space

Unique for each kernel and SA pair, the map-space is the finite, but highly-dimensional and extremely large set of all possible mappings that can be built through the aforementioned decisions. The contributions to the map-space's size can be broken down between different *factors allocations*, resulting from tiling and the parallelism strategy, and per-level *permutations* of loops, enstating different dataflows. Then, the cardinality of a map-space when considering both valid and invalid mappings can be expressed as:

$$|\mathrm{MS}| = \underbrace{(|\mathcal{D}|!)^{L_m}}_{\text{loop permutations}} \cdot \underbrace{\prod_{d\in\mathcal{D}}\prod_{p\,|\,d}\binom{v_p(d)+L-1}{L-1}}_{\text{factors allocations}}$$

where L is the SA's number of levels, of which L_m are memory ones, $\mathcal{D}$ is the set of normal kernel dimensions, $p|d$ stands for p running over all distinct prime factors of d, and $v_p(d)$ yields the highest power of p dividing d (the p-adic valuation of d). The first term is the number of possible combinations of permutations for the kernel's loops across levels, while the count of factors allocations is the product of distinct partitions of the multiset of prime factors of each dimension over as many possibly empty labeled multisets as there are levels. Even with a conservative choice of $L = 5$, $L_m = 4$, and six dimensions with five factors each, $|\mathrm{MS}|$ easily exceeds 10^{21}. For a simple convolution, with $|D| = 6$, permutations amount to 720^{L_m}, a large number that grows exponentially with the complexity of SAs, immediately eclipsing what an exhaustive search can manage. This suggests that, in the interest of scaling L, permutations should be handled as efficiently as possible while searching the map-space of convolutions or higher-dimensional kernels. Meanwhile, factors allocations tend to grow polynomially in L, but with the total of p-adic valuations of dimensions as the exponent, which in turn depends on $|D|$, thereby posing a serious challenge too.

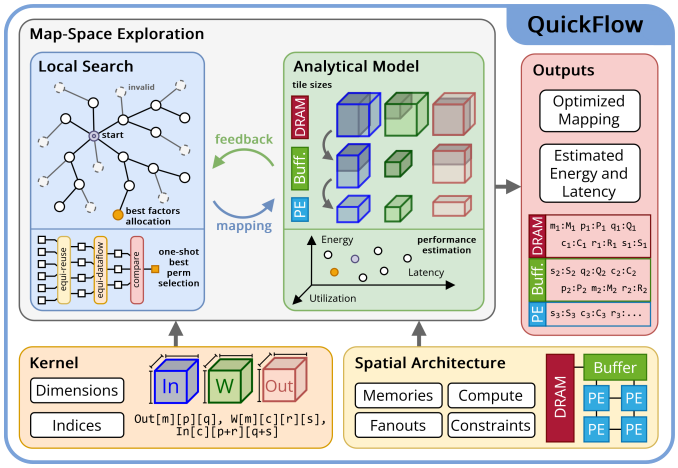

Fig. 3: Overview of the proposed mapping tool.

Furthermore, across most map-spaces, the EDP can vary by up to $10^4\times$, while near-optimal mappings are scarce [15, 20, 23].

IV. Proposed Mapping Tool

QuickFlow is a new mapping tool that shifts the paradigms of map-space exploration from handling permutations first to factors allocations first and from filtering mappings heuristically to skipping redundant ones, consistently simplifying any map-space and solving it with a specialized local search. It follows current state-of-the-art practices and comprises an **analytical model** to estimate a mapping's resulting energy and latency and a **mapper** that searches the map-space using the model for feedback, as outlined in Fig. 3.

QF is built on top of our existing mapping tool FactorFlow [9], its model was first extended to support generalized tensor contractions, then its mapper was fully replaced by the hereby presented routines. QF has been implemented in Python 3.13 (with free-threading) [24].

A. Analytical Model

The objective of the analytical model is to estimate the performance metrics for a given mapping on a SA. QF's model is fully flexible, its inputs are the levels describing any SA and a mapping formulated as in Sec. III-A, while its outputs include per-level and per-operand memory accesses, active instances, required bandwidth, and wasted cycles, together with the overall energy and latency.

Moreover, QF can work with any generalized tensor contraction kernel, these are an extension of convolutions to anything expressible in Einstein summation notation plus affine transformations of native indices [25]. To achieve efficient model support for all such kernels, QF implements data structures explicitly representing all involved tensor dimensions, couplings, and affine indices with optional coefficients. These are in turn exposed to the mapper, which will take advantage of the kernel's properties.

With the insights from Sec. III-C, a single traversal of a mapping's loops can infer each SA level's dataflow and reuse patterns. This constitutes the first pass of the model and derives memory accesses from dataflows, tile sizes, and tile volumes. Follow two other passes over the SA's levels, working on latency and then energy, including static and dynamic power. QF uses energy consumption estimates of all modeled hardware components provided by Accelergy [26].

Key to the model's function, the derivation of tile sizes is performed separately for affine and normal dimensions. The tile sizes along each level's normal dimensions are computed incrementally, by keeping track of the product of factors that are allocated on the present level or inner ones while constructing a mapping, as to minimize recomputation when factors are moved. As a nice side-effect, all invalid mappings can be identified early, with no need for a full model evaluation. For each affine dimension instead, we start from the indices that are linearly combined in its affine index and their coefficients. By definition, the tile size is the count of distinct values that the affine index can assume, coinciding with the unit slices along the affine dimension of the associated operand that a memory must store. In 2D convolutions, affine indices are in the form $ma+nb$, where $a \in \{0, \dots, A-1\}$ and $b \in \{0, \dots, B-1\}$ are normal indices, with A and B being their tile sizes, while the coefficients m and n are positive co-prime (if they are not, take their primitives) integers. Hence, let $S \coloneqq \{ma + nb \colon (a, b) \in \{0, \dots, A-1\} \times \{0, \dots, B-1\}\}$ be the set of distinct values of which we need the cardinality. Then, we could resort to enumerating S, but that would be very inefficient, thus we present a new exact formulation to solve our problem based on results from the closely related Frobenius problem [27]. That is:

$$|S| = \begin{cases} m(A-1) + n(B-1) - (m-1)(n-1) + 1 & \text{if } A > n,\ B > m \\ AB & \text{otherwise.} \end{cases}$$

For space constraints, the proof is reported in [28].

Another key aspect of the model where the count of distinct index values and this formula come into play is the calculation of the overlap in case of halo and spatial reuse. Such problem can be reformulated as computing the count of distinct indices over two (e.g. $A \longrightarrow 2A$) or more subsequent tiles, from which the overlap can be trivially inferred by knowing a single tile's size.

Introducing this exact formula saves around 42% of model execution time on strided and dilated convolutions w.r.t. enumeration. This result is trivial if no stride and dilation are present ($m = n = 1$), but is worth considering due to the increasing diversity of kernels in modern AI [29, 30]. Finally, the case of three or more coefficients is far more challenging, but beyond rare, justifying the use of dynamic programming to count distinct values via enumeration.

As shown in Sec. V, QF's model runs on average 3.6× and 3.8× faster than Timeloop's and ZigZag's respectively on the SAs and kernels we examined. Even so, QF's model is functionally equivalent to Timeloop, thus reaching the same mean accuracy of 95% [8].

B. Relations Over the Map-Space

Let us here introduce the structure needed over the map-space to handle permutations in one-shot and navigate it with a local search. For this purpose, the correlation between mappings representation and reuse from Sec. III-C will be vital. Related examples are in Fig. 4.

Relations Between Permutations: Considering that permutations control reuse opportunities on a per-memory-level basis and depending on the innermost iterated dimensions being coupled or orthogonal to each of the three operands, we can intuit that many are redundant and would result in mappings with identical behavior.

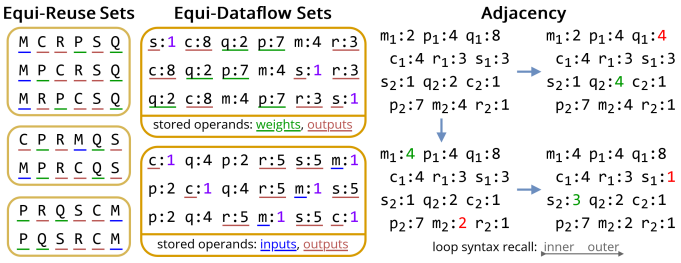

Fig. 4: Examples of relations over permutations/mappings. Underlines mark orthogonality to inputs (blue), weights (green) and outputs (red).

Looking at a single level's loops, we define two permutations as *equi-reuse* if, for each operand, they have: (1) the innermost coupled dimension in the same position, let it be the k-th loop; (2) the same dimension on loop k iff it is an affine one; (3) all loops inside k on the same dimensions, even if in a different order. It follows that two equi-reuse permutations expose the same reuse opportunities, with (3) locking stationarity, while (1, 2) ensures the same halo reuse.

Assuming now to know a factors allocation, and in particular for each level which dimensions are iterated and which are not, in the sense of having a single iteration and no allocated factors, we can find additional redundancy among permutations. We say that, on a level, two permutations are in *equi-dataflow* if they have the same order of loops with more than one iteration, since any non-iterated dimension is irrelevant, regardless of it being in a position to affect reused or not. This definition is already used in [8, 9]. However, we can generalize the concept further, to two permutations having for each there-stored operand: (1) the same set (order does not matter) of innermost iterated dimensions orthogonal to it, up to, and excluding, the first coupled dimension; (2) the same innermost iterated coupled dimension iff that is an affine dimension in either permutation. This relation further shrinks the set of permutations yielding distinct dataflows for the given factors allocation, while all equi-dataflow permutations exhibit identical performance.

Additionally, when a SA does not support one or more of the three presented reuse classes on a certain level or operand, the conditions for these relations can be relaxed accordingly.

Relation Between Factors Allocations: To enact a local search we need a relation that we can follow to traverse the map-space, factors allocations in particular, with full reachability. Thus, we say that two mappings are *adjacent* if one can be constructed from the other by moving exactly one prime factor between two loops on the same dimension. Occasionally, we'll exploit a generalization of adjacency, that allows for moves involving any multiplicity of the same factor.

Notably, transitions between adjacent mappings can be implemented very efficiently (see Sec. IV-A): only one normal dimension's tile sizes need to be updated, with that done incrementally, and all constraints checks can be performed while moving factors, instantaneously preventing the construction of invalid mappings.

C. Map-Space Exploration

QuickFlow's mapper searches the map-space by addressing mapping decisions in order, but departs from the ways of contemporary approaches, by handling the factors allocation first. This allows QF to solve permutations in one-shot because it already knows the amount of available reuse in each dataflow by fixing first the tiling and parallelism strategy. And what's more, by picking permutations after building some factors allocations, but before evaluating them, QF can compare them fairly, once their best dataflows are known, and with a single model call for each! This is a paradigm shift that

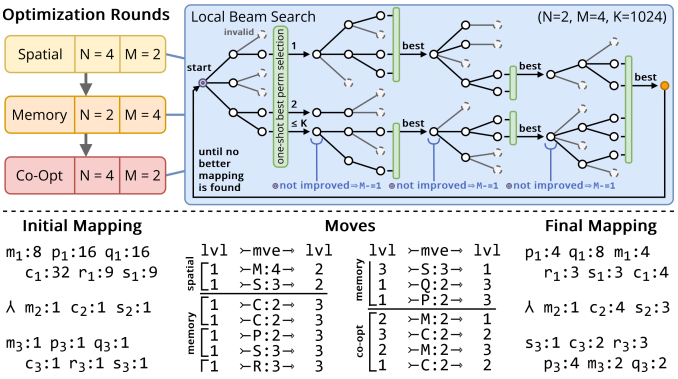

Fig. 5: Multi-hop local beam search rounds and moves examples.

enables QF to use a greedy local search to maximize its sample efficiency, minimizing model invocations and quickly converging to near-optimal mappings. To ensure that a greedy approach can balance exploiting a SA's memory sizes, reuse, and parallelism effectively, QF minimizes the following compound metric [9], where the denominator is the fraction of active spatial instances:

$$EDPoU := \frac{Energy \cdot Latency}{Utilization}.$$

Solving Permutations in One-Shot: Given a mapping with a fixed factors allocation, QF picks its best permutations in one-shot and while going level-by-level, never backtracking on past decisions. That's because by fixing factors, QF doesn't need to go through the $(|\mathcal{D}|!)^{L_m}$ combinations that influence different tiling choices, but can instead focus on maximizing reuse on each level. However, even for a simple convolution, $6! = 720$ permutations per level are still a lot to consider, so let's take advantage of the above-defined relations.

Even prior to starting the map-space exploration, just from the kernel and SA's specifications, we keep a single permutation from each equi-reuse class. On simple convolutions, this results in no more than 26 useful permutations down from 720 per level.

Now on to finding the best permutation for each level while looking at a specific factors allocation. First, there are a few trivial cases to solve. If at most one dimension is iterated, a level's permutation is irrelevant. While, if two dimensions are iterated, the best permutation can be determined by comparing the total reuse between the two permutations differing solely for their order. Then, in the general case, we preserve only one permutation from each set of equi-dataflow ones. Throughout our experiments, these steps resulted in an average of 2.17 surviving permutations. Finally, we calculate the reuse yielded by each of the remaining permutations and pick the best one, where the metric used to rank reuse is the number of spared memory accesses on the present level.

Remarkably, this technique simplifies the map-space without excluding any reuse opportunity that could lead to a distinct dataflow, while making the process of selecting permutations very efficient.

Multi-hop Local Beam Search: Let us define an N-hops local search as follows: from a given mapping, we move once to all adjacent ones, and for each of them to all its adjacent ones again, up to N times. Moving to a mapping constructs its factors allocation, then its best permutations are selected as previously explained, and finally, the model is invoked to get its $EDPoU$. The search runs depth-first to reduce the tile sizes calculation overhead. With this, in the end, all reached mappings are fairly compared, ranked, and returned.

QF's main optimization routine iteratively improves an initial mapping by running an N-hops local search around it, using its output as candidate mappings and each time selecting, greedily, the best candidate to replace the initial mapping, repeating until no improving move is returned. When this occurs, QF transitions to a beam search by re-running the local search starting once from where each of the top candidate mappings was left off before, up to a maximum beam width of K, further exploring them for up to N more moves. Then, only the best candidate mapping from each re-run is kept and, if none improves on the initial one, the process repeats, for at most M times. If M is exceeded, the whole routine ends, while if a better mapping is found, it replaces the initial one, and the whole routine repeats. The final mapping will be by construction a local optima w.r.t. the considered adjacency relation.

QF's routine runs for three rounds. The initial mapping is built with all factors on the outermost level's loops, constituting a robust starting point (more details later). In round (1), QF optimizes the SA's utilization, thus factors are only allowed to move between spatial levels plus the outermost level. For round (2), only the allocation of factors on memories is changed, with no moves involving spatial levels. Lastly, in round (3), memory and spatial levels are co-optimized and all factors are allowed to move. This way, every round's complexity is reduced, each can have ad-hoc parameters, and the costly co-optimization starts from an already good mapping.

Throughout the entire process, a mapping is never revisited unless it is reached in fewer moves, while invalid ones are immediately discarded. As such, when considering the adjacency relation as a graph, only the paths towards the nearest occurrence of each mapping are being traversed, resulting in a tree rooted in the starting mapping. Fig. 5 shows a diagram of the whole process based on this interpretation, alongside an example of the resulting moves.

Ultimately, QF's optimization process strives for a high search speed by wasting minimal effort in finding moves that improve the initial mapping, without giving up on the quality of the final result.

Optimization Routine Complexity: The parameters N and M control the depth and the spread of the technique. An upper bound on the number of mappings visited by the multi-hop local beam search is:

$$(\#\,\text{main routine steps} + \#\,\text{beam steps} \cdot K) \cdot E^N$$

where $E := (L-1)\sum_{d\in\mathcal{D}} \omega(d)$ is the maximum degree of adjacent mappings, L is the number of SA levels, $\mathcal{D}$ contains all normal kernel dimensions and $\omega(d)$ is the count of all prime factors of d. Crucially, for fixed N, M, and K, the process's complexity grows polynomially with L and $|\mathcal{D}|$, enabling QF to efficiently scale to large map-spaces. While, in general, the number of main routine and beam steps should grow linearly with the number of prime factors in the kernel, varying between 6-68 and 0-16 respectively during the hereby experiments. Increasing N allows QF to accept up to $N-1$ adjacency hops that worsen the $EDPoU$ in pursuit of a last hop improving it. The toll on the mapper's execution time is exponential, but, in turn, many poor local optima can be escaped. We found that $N = 4$ works best in rounds (1) and (3), while round (2) is fine with $N = 2$. On the other hand, higher Ms explore multiple paths in parallel and for longer, reaching further away mappings without excessive overhead, by repeatedly preserving only the best move found along each path. Thus, raising M is far cheaper than N, but has a comparable effect in reducing the chances of getting stuck with a poor mapping. Our experiments never found more than eight moves between one local optimum and any closest other one, so the values of M have been chosen to allow chains of up to 8 moves to be explorable. Finally, K defaults to 1024, but in practice a local search rarely retrieves more than 300 valid mappings with the above values for N, inducing smaller actual beam widths.

Dimensions \ Kernel		Simple Convolutions										Pointwise		Stride & Dilation			Transposed		Batched		
		I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII	XIV	XV	XVI	XVII	XVIII	XIX	XX
Input Channels	C	128	512	3	64	128	256	256	576	3	72	64	24	16	128	256	128	576	256	72	256
Output Channels	M	256	512	64	64	128	256	512	576	96	72	256	88	16	128	256	256	576	256	72	256
Output Height/Width	P, Q	56	28	112	56	28	14	7	7	176	28	56	28	224	112	56	32	7	14	28	56
Weights Height/Width	R, S	3	3	7	3	3	3	3	5	3	3	1	1	3	9	3	4	5	3	3	5
Batch Size	N	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	64	128	32
Stride Height/Width	$\overline{p}$, $\overline{q}$	1	1	2	1	1	1	2	1	2	2	1	1	3	4	2	1	1	1	2	2
Dilation Height/Width	$\overline{r}$, $\overline{s}$	1	1	1	1	1	1	1	1	1	1	1	1	4	3	3	1	1	1	1	3

TABLE I: Dimensions of the convolutions used in the experiments.

In round (1) a simple greedy approach would immediately move the largest prime factors to occupy spatial instances and decrease the *EDPoU* early, but if such factors are not shared primes with those of the number of instances, they will never achieve full utilization. Therefore, in round (1), a higher N and the generalized adjacency relation are used, ensuring maximum utilization with a minimal impact on the algorithm's complexity due to the usually limited size and number of spatial levels. During round (3), instead, it often happens that multiple primes on a spatial level could benefit from being taken from a different dimension than the one round (1) chose, see an example in Fig. 5. This is the main purpose of co-optimization, to update the parallelism strategy in response to the latest tiling decisions, and the reason it uses a higher N.

Further Improvements: The choice of the starting mapping for the local search is crucial, it must be simultaneously far from poor local optima and be within a few adjacency hops of as many globally optimal mappings as possible. Allocating all iterations on the outermost level naturally fits both requirements, since that is the point furthest away from the boundaries of hardware constraints. Therefore, such a starting point sits in the most densely connected area of the adjacency relation's graph, typically having the highest count of valid adjacent mappings, also resulting in the fewest nearby local optima. Still, it can be improved to speed up the mapper: if all fanout levels support a single dimension, they can be allocated a priori the maximum number of iterations that can fit on their instances, then, the first round of local beam search can be skipped.

QF's implementation leverages multithreading with a producer-consumer approach: each first possible move in a local search becomes a task, with a thread exploring its $N-1$ subsequent moves.

V. Experimental Results

To demonstrate the effectiveness of QuickFlow, we compare it against four other state-of-the-art mapping tools across three spatial architectures and twenty convolutions taken from well-established neural networks. As such, we go over sixty different map-spaces.

A. Experimental Setup

As target architectures (see Table II) we selected Eyeriss [3], due to its significance in the field, Simba (single chiplet) [4], for its variety of dataflows, and the TPUv1 (8-bit mode) [5], as an example of systolic array. The latter two's complex memory hierarchies are meant to stress the scalability of compared tools. For convolutions (see Table I), we took simple and pointwise ones from popular convolutional neural networks [31-33], while others have been inspired by works relying heavily on strided, dilated, and transposed convolutions [29, 30]. Some variations with batches have also been included. Compared tools were chosen to cover a wide range of techniques: random search with Timeloop [8], loop ordering and merging-based with LOMA-7 [17], simulated annealing with SALSA [18], and a permutations-before-factors-allocations scheme with an updated FactorFlow (FF) [9]. To make it support convolutions, in this version of FF we use the presented analytical model (Sec. IV-A) and change the exhaustive search of permutations to go over only one per equi-reuse class (Sec. IV-B), all other heuristics are unchanged.

Name	On-chip Mem	# Mems	Fanout Instances	Fanout Dims	Target
Eyeriss [3]	196 KiB	5	14, 12	Q/M, S/C/M	Edge
Simba [4]	752 KiB	6	16, 4, 4	M/C, M, C	Both
TPUv1 [5]	30 MiB	6	256, 256	M, C	HPC

TABLE II: Features of the targeted spatial architectures.

Model Metric	I-X	XI-XII	XIII-XV	XVI-XVII	XVIII-XX
QF speed [evaluations/s]	3255	7941	5280	1359	1314
QF speed w.r.t. Timeloop [8]	2.65×	2.81×	5.70×	2.85×	4.21×
QF speed w.r.t. ZigZag [10]	3.05×	2.73×	6.48×	2.54×	4.46×
QF accuracy w.r.t. Timeloop	1.0	1.0	1.0	1.0	1.0

TABLE III: Model speed and accuracy, by kernel type, averaged on SAs.

Reported EDP comparisons are based on QF's analytical model. All obtained mappings have also been evaluated in Timeloop, that gave matching results, and ZigZag, where the relative ranking of their quality metrics remained unchanged. Each compared tool, however, could only use its internal model during map-space exploration, hence all SAs were carefully modeled with equivalent specifications in all of them. Every mapper was used with its default settings; for Timeloop that is the "D" configuration, while "L" indicates that it was allowed to try 15× more mappings before terminating. Missing values denote a failure of a tool to return a valid mapping. Execution time was measured on an AMD Ryzen 3900xt @ 3.8 GHz and averaged across 10 runs without any significative variance. Parallel execution with 8 threads was allowed for all mapping tools.

B. Comparison Results

Across all SAs and kernels, as reported in Fig. 6, QF consistently attains the highest-quality mappings in 3.8 s on average. Looking at per-SA trends, on Eyeriss, the simplest of the three, QF improves on the EDP of the best mapping found by other tools by 1.0-1.6× (1.2× avg.) and reaches it in 1.4-18× (4.6× avg.) less time. Whereas, crucial for our goal, on the more complex Simba and TPUv1 SAs, we see that QF experiences only a minor increase in execution time compared to Eyeriss, that is a 1.2-182× (34× avg.) cut relative to other tools, while achieving a 1.0-2.1× (1.2× avg.) EDP reduction.

As shown, QF always bests Timeloop by 1.0-18k× in EDP and 1.2-2.7k× in execution time. This can be generally attributed to the inefficiency of the random search approach. Instead, LOMA and SALSA focus on loop ordering and handle factors allocations in a single pass that maximizes reuse on one memory at a time, this simplifies the exploration, but unfortunately, a single pass can't account for all trade-offs in tiling and/or parallelism decisions, leading to suboptimal mappings. Hence, QF, with its iterative co-optimization approach, finds 1.0-27× better mappings by EDP in 1.4-182× less time. Notably, in terms of EDP, QF is always tied with or better than FF, validating our reasoning that handling factors allocations first and fairly comparing mappings creates the best scenario for the local search, which can thusly run deeper and find niche moves that improve EDP by up to 2.4×, all while running on average 77× faster despite them sharing the same model.

Looking at Table III, QF's model is on average 3.7× faster than the rest. On simple, pointwise, and transposed convolutions the speedup is around 2.8×, while on strided and dilated or batched ones we see

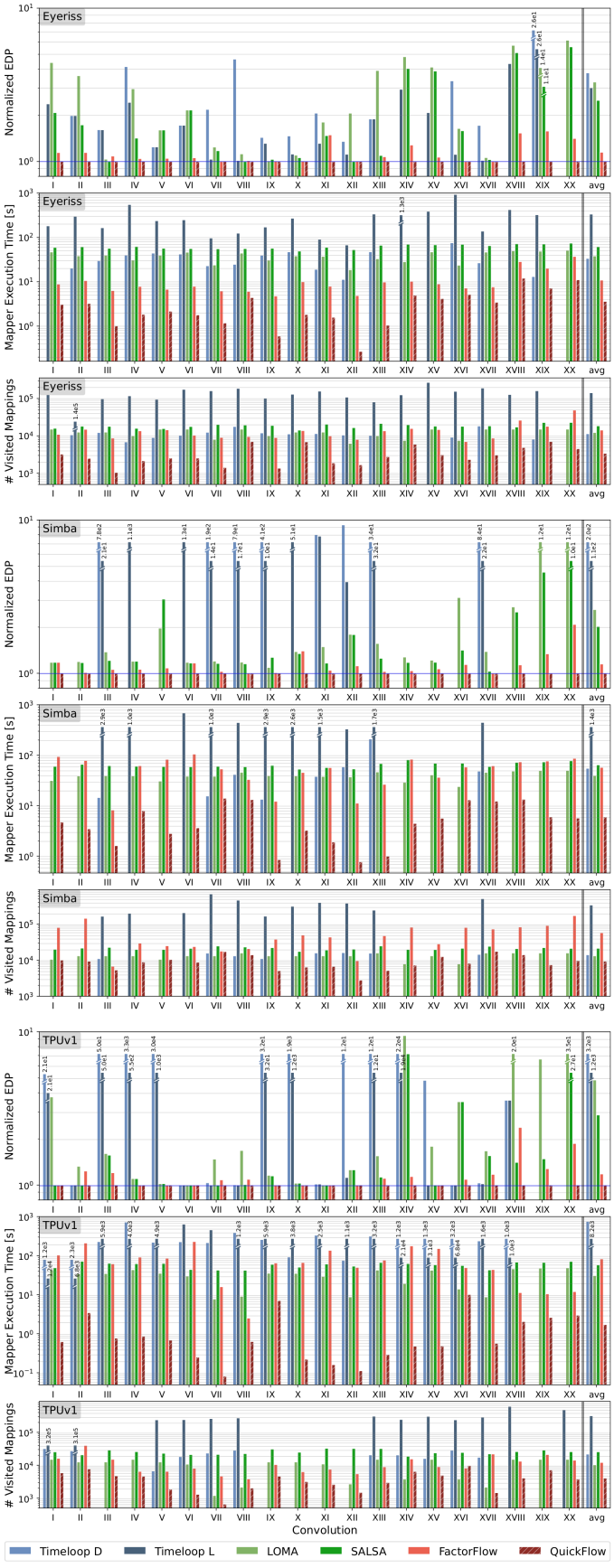

Fig. 6: EDP, tool execution time, and number of visited mappings for QuickFlow compared to four other state-of-the-art mapping tools.

a 5.2× speedup, with QF overall retaining similar execution times despite the workloads diversity. In particular, batched convolutions were the most challenging we tested ($|D| = 7$), and there, the scalability of QF is highlighted by its mere 6% increase in execution time over its average on other kernels, while still managing to improve EDP by 1.6× (avg.) against FF, with other tools trailing at 10×.

To assess the raw efficiency of mapping techniques, in Fig. 6 we also report the number of mappings visited by each tool. We consider "visited" a mapping that has been fully built and evaluated by the model. We observe that Timeloop tries the most mappings, but still, a random search can't reliably find good ones. On the other hand, the execution time of LOMA and SALSA is almost constant because they are set to cap the number of explored mappings, through pruning in the former and a maximum number of annealing steps in the latter, as such, they find increasingly worse mappings on larger map-spaces. In contrast, QF perseveres until it converges to a local optimum among adjacent factors allocations, nonetheless our beam search and careful permutation selection waste minimal effort on suboptimal and invalid mappings, typically traversing only a few thousand, that is a mean of 40× less than Timeloop, 4.8× less than LOMA and SALSA, and 5.1× less than FF. Any discrepancy between execution time and visited mappings is due to imperfect load balancing in the algorithms' multithreaded implementations.

To empirically validate our one-shot technique for permutations, we repeated all sixty experiments without it, forcing QF to evaluate through the model all permutations for each visited factors allocation. As expected, this resulted in hours-long execution times but no performance difference was found neither in the final mappings nor in any visited one, with the one-shot technique always finding the best dataflow by EDP. Similarly, to test our selection of parameters for the mapper, we ran QF with $N = 8$, $M = 1$, and generalized adjacency on all rounds: execution time again exceeded an hour while we measured better mappings only in 6/60 cases, with a maximum EDP reduction of 0.7%. Such a result also strongly suggests that QF finds a global optimum in the majority of cases, since it is otherwise unlikely for no improvement to exist within eight moves. Regrettably, fully searching any map-space to confirm this isn't feasible, so no definitive claim on global optimality can be made.

Ultimately, we can affirm that QF's local search is a sound and efficient technique to map convolutions on spatial architectures, achieving near-optimal performance with little mapping effort.

VI. Conclusions

Throughout this work we analyzed mappings and their ties with reuse opportunities on spatial architectures, revealing equivalence classes between them. Then, with QuickFlow, we introduced a new local search-based mapping technique that by exploring factors allocations first manages to very efficiently select their best dataflow before comparing them. This allowed QF to emerge as nearly one order of magnitude faster than previous state-of-the-art tools across a variety of SAs and kernels, while always finding equivalent or better mappings. For applications relying on efficient mapping tools, like hardware design space exploration and multi-tenant environments, this possibly means exploring ten times more designs and ten times less runtime overhead respectively. These use cases will be the subject of our future studies. At last, as spatial architectures become more and more pervasive, the mapping problem will cement as a recurring challenge that must be effectively addressed to exploit their full potential on any new workload. In this landscape, we trust that the present study advanced the field of mapping techniques.

Alongside this paper, QF has been released as open-source: [34].

Acknowledgements

This work has been partially supported by the Spoke 1 on *Future HPC* of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Mission 4 - Next Generation EU.

References

[1] C. Silvano, D. Ielmini, F. Ferrandi, L. Fiorin, S. Curzel, L. Benini, F. Conti, A. Garofalo, C. Zambelli, E. Calore, S. Schifano, M. Palesi, G. Ascia, D. Patti, N. Petra, D. De Caro, L. Lavagno, T. Urso, V. Cardellini, G. C. Cardarilli, R. Birke, and S. Perri, "A survey on deep learning hardware accelerators for heterogeneous hpc platforms," *ACM Comput. Surv.*, vol. 57, no. 11, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729215

[2] F. Ferrandi, S. Curzel, L. Fiorin, D. Ielmini, C. Silvano, F. Conti, A. Burrello, F. Barchi, L. Benini, L. Lavagno, T. Urso, E. Calore, S. F. Schifano, C. Zambelli, M. Palesi, G. Ascia, E. Russo, N. Petra, D. D. Caro, G. D. Meo, V. Cardellini, S. Filippone, F. L. Presti, F. Silvestri, P. Palazzari, and S. Perri, "A survey on design methodologies for accelerating deep learning on heterogeneous architectures," 2023.

[3] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[4] Y. S. Shao, J. Clemons *et al.*, "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

[5] N. P. Jouppi, C. Young *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[6] N. P. Jouppi, D. H. Yoon *et al.*, "Ten lessons from three generations shaped google's tpuv4i," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 1–14. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00010

[7] S.-C. Kao, H. Kwon, M. Pellauer, A. Parashar, and T. Krishna, "A formalism of dnn accelerator flexibility," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, Jun. 2022. [Online]. Available: https://doi.org/10.1145/3530907

[8] 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.

[9] M. Ronzani and C. Silvano, "Factorflow: Mapping gemms on spatial architectures through adaptive programming and greedy optimization," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 706–712. [Online]. Available: https://doi.org/10.1145/3658617.3697670

[10] 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.

[11] C. Hong, Q. Huang, G. Dinh, M. Subedar, and Y. S. Shao, "Dosa: Differentiable model-based one-loop search for dnn accelerators," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 209–224. [Online]. Available: https://doi.org/10.1145/3613424.3623797

[12] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, "Magnet: A modular accelerator generator for neural networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[13] S. Kim, H. Genc, V. V. Nikiforov, K. Asanović, B. Nikolić, and Y. S. Shao, "Moca: Memory-centric, adaptive execution for multi-tenant deep neural networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 828–841.

[14] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 681–697.

[15] 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.

[16] 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

[17] A. Symons, L. Mei, and M. Verhelst, "Loma: Fast auto-scheduling on dnn accelerators through loop-order-based memory allocation," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2021, pp. 1–4.

[18] V. J. Jung, A. Symons, L. Mei, M. Verhelst, and L. Benini, "Salsa: Simulated annealing based loop-ordering scheduler for dnn accelerators," in *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2023, pp. 1–5.

[19] 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," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 554–566.

[20] M. Reshadi and D. Gregg, "Local: Low-complex mapping algorithm for spatial dnn accelerators," in *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2021, pp. 1–7.

[21] F. Wang, M. Shen, Y. Lu, and N. Xiao, "Ceiba: An efficient and scalable dnn scheduler for spatial accelerators," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 2, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3715123

[22] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: a polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 193–205.

[23] S. Kim, C. Hooper, T. Wattanawong, M. Kang, R. Yan, H. Genc, G. Dinh, Q. Huang, K. Keutzer, M. W. Mahoney, Y. S. Shao, and A. Gholami, "Full stack optimization of transformer inference: a survey," 2023.

[24] Python Software Foundation, "Python (Version 3.13.0, free-threading variant) [Computer software]," 2024, accessed: 2025-04-14. For details on the free-threading build, see the official documentation: https://docs.python.org/3/howto/free-threading-python.html. [Online]. Available: https://www.python.org/downloads/release/python-3130/

[25] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov, "High-performance tensor contractions for gpus," *Procedia Computer Science*, vol. 80, pp. 108–118, 2016, international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA. [Online]. Available: https://doi.org/10.1016/j.procs.2016.05.302

[26] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[27] J. L. R. Alfonsín, "The diophantine frobenius problem." Oxford University Press, 12 2005.

[28] L. D. Gaspari and M. Ronzani, "Short proof: Exact solution to the finite frobenius coin problem," 2025. [Online]. Available: https://arxiv.org/abs/2508.08464

[29] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, 2017.

[30] N. Takahashi and Y. Mitsufuji, "Densely connected multidilated convolutional networks for dense prediction tasks," 2021. [Online]. Available: https://arxiv.org/abs/2011.11844

[31] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1409.1556

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

[33] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for mobilenetv3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1314–1324.

[34] M. Ronzani, "FactorFlow," https://github.com/EMJzero/FactorFlow, 2025, accessed: 2025-08-12.