Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search

Kartik Hegde
University of Illinois at Urbana-Champaign
USA
kvhegde2@illinois.edu

Po-An Tsai
NVIDIA
USA
poant@nvidia.com

Sitao Huang
University of Illinois at Urbana-Champaign
USA
shuang91@illinois.edu

Vikas Chandra
Facebook
USA
vchandra@fb.com

Angshuman Parashar
NVIDIA
USA
aparashar@nvidia.com

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
USA
cwfletch@illinois.edu

Abstract
Modern day computing increasingly relies on specialization to satisfy growing performance and efficiency requirements. A core challenge in designing such specialized hardware architectures is how to perform mapping space search, i.e., search for an optimal mapping from algorithm to hardware. Prior work shows that choosing an inefficient mapping can lead to multiplicative-factor efficiency overheads. Additionally, the search space is not only large but also non-convex and non-smooth, precluding advanced search techniques. As a result, previous works are forced to implement mapping space search using expert choices or sub-optimal search heuristics.

This work proposes Mind Mappings, a novel gradient-based search method for algorithm-accelerator mapping space search. The key idea is to derive a smooth, differentiable approximation to the otherwise non-smooth, non-convex search space. With a smooth, differentiable approximation, we can leverage efficient gradient-based search algorithms to find high-quality mappings. We extensively compare Mind Mappings to black-box optimization schemes used in prior work. When tasked to find mappings for two important workloads (CNN and MTTKRP), the proposed search finds mappings that achieve an average 1.40x, 1.76x, and 1.29x (when run for a fixed number of steps) and 3.16x, 4.19x, and 2.90x (when run for a fixed amount of time) better energy-delay product (EDP) relative to Simulated Annealing, Genetic Algorithms and Reinforcement Learning, respectively. Meanwhile, Mind Mappings returns mappings with only 5.32x higher EDP than a possibly unachievable theoretical lower-bound, indicating proximity to the global optima.

CCS Concepts
• Computer systems organization → Special purpose systems; • Software and its engineering → Compilers.

1 INTRODUCTION
The compound effect of the slowing of Moore’s law coupled with a growing demand for efficient compute has ushered in an era of specialized hardware architectures. Due to their inherent performance, energy, and area characteristics, these accelerators are driving innovation in diverse areas such as machine learning [4, 20, 32, 38, 43, 69], medicine [22, 23, 41], cryptography [27, 61], etc. They are seeing a wide variety of deployments ranging from cloud to edge—forcing designers to make complex design decisions to achieve their efficiency objectives.

Although they are specialized, accelerators are often flexible [21, 36, 52, 59, 95], designed to support different parameterizations of a single algorithm to even a range of algorithms within or across domains. This flexibility forces architects to decouple the act of designing the architecture from the act of mapping a specific problem—a parameterized instance of an algorithm—onto the architecture. This is shown in Figure 1. First, pre-fabrication, architects choose architectural parameters to suit the budget and deployment requirements of expected target problems—a process referred to as Architecture Design Space Search. This can include decisions such as the number of processing elements, on-chip buffer sizes, network-on-chip topology, bandwidth to off-chip memory,
Second, post-fabrication and based on the designed-in flexibility of the hardware, architects or users map target algorithms to the hardware—referred to as Mapping Space Search. These decisions can include choosing how much of each buffer to allocate for each data structure, mapping of computations to processing elements, etc., and are analogous to writing/compiling programs for general-purpose processors.

Mapping space search is an important problem, and currently faces severe scalability and performance challenges. To start, prior work has shown that problem efficiency is very sensitive to the choice of mapping [20, 36, 52, 59, 68, 69, 95]. Further, the same studies illustrate how optimal mapping varies significantly depending on problem size and parameters (e.g., DNN model parameters), resource availability, performance and power requirements, etc. This suggests that mapping space search will constitute an increasing recurring cost, as accelerators are re-targeted for new problems.

Making matters worse, simply gaining intuition for how to search through the map space, or how to pose the search to an automated tool, is an ad-hoc and expert-driven process. Accelerators lack consistent hardware-software abstractions, such as instruction set architectures (ISAs) in the general-purpose computing world, and instead rely on bespoke configurable hardware components designed to provide higher degrees of control and efficiency. Further, different accelerators tend to have different degrees of configurability in different hardware components, ranging from programmable networks on chip [52], buffers [36, 73], address generators [100], etc. While this may be suitable for experts with deep knowledge of both the architecture and algorithm, it clearly does not scale to non-experts programming new hardware with new algorithms.

Making matters even worse, while prior work has proposed tools and algorithms (i.e., Mappers) for automatically searching the map space, all existing approaches have serious limitations due to search space complexity [3, 15, 36, 68, 102]. First, the search space is often high dimensional (i.e., each degree of configurability induces a dimension), causing a combinatorial explosion of possible mappings and rendering exhaustive techniques ineffective [68]. Second, the search space is both non-convex (many local minima) and non-smooth (not differentiable), forcing prior work to rely on black-box optimization [30] approaches.

To summarize, while configurable accelerators have demonstrated their potential, the lack of an efficient and high-quality Mapper hinders broader adoption.

### 1.1 This Work

This paper addresses the challenges above by proposing Mind Mappings, a scalable and automated method to quickly and effectively perform mapping space search.

As mentioned previously, the key challenge hindering prior work is that mapping search space is non-smooth, forcing prior work to resort to black-box optimization techniques. This is because the accelerator cost function—which search algorithms use to evaluate the cost of a given candidate mapping—is non-smooth. For example, the cost function might be an architectural simulator or the accelerator itself.

Mind Mappings addresses this challenge by constructing a differentiable approximation of the cost function, called the surrogate [6, 75, 91]. Using the surrogate, Mind Mappings derives gradients for the cost function, with respect to candidate mappings, and uses those gradients to perform a powerful first-order optimization technique, Gradient Descent [53, 54], to quickly find low-cost mappings.

The key insight here is that the differentiable surrogate of the actual non-differentiable cost function can provide us with approximate gradients, which are sufficient to guide the search along the direction of steepest descent, even in the absence of true gradients. This insight simultaneously improves map space search quality and reduces map space search time, as gradients by definition point in the direction of the greatest reduction in cost.

 Crucially, Mind Mappings formulates both predicting the cost of a mapping and finding the optimal mapping as learning problems, thereby doing away with requiring expert knowledge in the target domain.

This paper makes the following contributions:

1. To the best of our knowledge, our work is the first to enable target domain-independent mapping search for programmable accelerators. We require neither expert knowledge in the target application domain(s), nor any domain specific heuristics for handling programmable hardware attributes.

2. To the best of our knowledge, our work is the first to formulate mapping space search as a first-order optimization problem, enabling an efficient gradient-based search that is able to quickly find high-quality mappings.

3. We extensively evaluate Mind Mappings across two target algorithms—CNNs and MTTKRP—comparing against multiple baseline search heuristics including simulated annealing (SA), genetic algorithms (GA), and reinforcement learning (RL). For CNNs and MTTKRP, the proposed search method finds mappings with an average 1.40×, 1.76×, and 1.29× (when run for a fixed number of steps) and 3.16×, 4.19×,
and 2.90\times (when run for a fixed amount of time) better energy-delay product over SA, GA, and RL, respectively.

(4) To facilitate further adoption, we provide a reference implementation of the Mind Mappings framework here: https://github.com/kartik-hegde/mindmappings.

2 BACKGROUND

In this section, we formally define the algorithm-accelerator mapping space search problem and elaborate with an example.

2.1 Algorithm-Accelerator Mapping Space

In this paper, we assume that a hardware accelerator \( a \) and a target problem \( p \) is given, where \( a \) is a parameterized instance of an algorithm. For example, if matrix multiplication is the target algorithm, an example target problem is a matrix multiplication between two matrices of fixed shape (dimensions), irrespective of the contents of the matrices. We begin by defining a mapping \( m \) and the mapping space \( M \).

Definition 2.1. Mapping. A mapping \( m \in P_0 \times \cdots \times P_{D-1} \) is a \( D \)-tuple, where \( D \) is the number of programmable attributes of the given accelerator \( a \). Each element in the mapping vector, \( m_d \) for \( d \in \{0, D\} \) belongs to the domain of the \( d \)-th programmable attribute, \( P_d \).

Definition 2.2. Mapping Space. Given an accelerator \( a \) and a target problem \( p \), we define a mapping space as \( M_{a,p} = \{ m \in P_0 \times \cdots \times P_{D-1} \mid a(m,i) = p(i) \ \forall i \in I_p \} \) where \( I_p \) denotes the possible inputs (e.g., all possible matrices with a specific shape) to a problem \( p \), \( a(m,i) \) denotes the accelerator’s output given mapping \( m \) and input \( i \), and \( p(i) \) denotes the golden reference output given input \( i \) for problem \( p \).

In other words, \( M_{a,p} \) is the set of mappings that result in functional correctness for the given problem \( p \) on the accelerator \( a \). We call such mappings valid for \( a \) and \( p \) and write \( M_{a,p} \) as \( M \) for short when the context is clear. Intuitively, the different \( m \in M \) can be thought of as different “programs” representing problem \( p \) on accelerator \( a \). An example of a mapping space is given in Section 3.

With this in mind, the size of the mapping space varies based on the programmability of the underlying accelerator and the problem \( p \). On one extreme, the size of \( M \) (denoted \(|M|\)) can be \( 1 \) for a fixed-function ASIC designed to execute one fixed problem. In general, \(|M| = O(\prod_{d \in [0,D]} |P_d|)\), where the number of attributes \( D \) and the size of each attribute space \(|P_d|\) is large. The Big-Oh captures how some mappings in the Cartesian product of assignments to programmable attributes may be invalid.

2.2 Mapping Space Search

Mapping Space Search is the combinatorial search problem to find the mapping \( m_{\text{opt}} \in M \) that minimizes the cost \( f \), where the cost function \( f \) is the optimization objective set by the designer (discussed further in Section 2.3). That is, \( m_{\text{opt}} = \arg\min_{m \in M_{a,p}} f(a,m) \) (1)

where the user specifies the problem \( p \) and architecture \( a \). We denote \( f(a,m) \) as \( f(m) \) for short. In theory, mapping space search can be performed at compile time or at run time. In practice, programmable accelerators today either perform the search completely offline, e.g., when compiling a new problem to the architecture [36, 51, 80], or partly offline/partly online [66].

In this paper, we assume \( f \) is a function of \( a \) and \( p \)—not the problem input \( i \). This holds for several important workloads, e.g., kernels in dense tensor algebra such as dense matrix multiplication and deep neural network training/inference [68]. However, it does not hold when input-dependent optimizations, e.g., data sparsity, influence cost [37]. We consider efficient mapping space search for input-dependent mappings to be important future work.

2.3 Cost Function

The cost function \( f \) (Equation 1) estimates the cost of running the given mapping \( m \) on the given hardware accelerator \( a \). This serves as the objective for optimization in the mapping space search. It is up to the designer to formulate the cost function based on the design criteria. For example, the cost function can be formulated as a weighted sum or product of several factors, or as a prioritized order of measurable metrics [68]. For example, \( f(a,m) = \sum_{k=0}^{K-1} w_k f_k(a,m) \) where \( K \) is the set of all the factors considered by the designer and \( w_k \) is the importance assigned to the \( k \)-th factor. Factors can include various measures such as power, performance, or meta-statistics such as the number of buffer accesses, etc., and designers may choose to assign appropriate weights for each of the costs based on the requirements/factors. For example, if \( f_k \) represents the number of DRAM accesses, \( w_k \) might represent the energy per DRAM access. Importantly, the function computing each factor \( f_k \) need not be smooth, differentiable, etc.

3 EXAMPLE MAPPING SPACE: 1D-CONV

We now describe the mapping space for a hardware accelerator designed to perform 1D-Convolution (1D-Conv) with energy-delay product as the cost function. This represents a simplified version of the accelerator and algorithm (Convolutional Neural Nets/CNNs) that we evaluate in Section 5, and we will refer to the example in Section 4 to explain ideas.

Algorithmically, for filter \( F \), input \( I \), and output \( O \), 1D-Conv is given as

\[
O[x] = \sum_{r=0}^{R-1} I[x+r] \ast F[r]
\]

\[
0 \leq x < W - R + 1
\]

for input width \( W \) and filter size \( R \). Using the terminology in Section 2.1, the 1D-Conv algorithm forms a family of problems, where each problem \( p \) corresponds to a specific setting of \( W \) and \( R \).

1D-Conv in Equation 2 can be represented as a loop nest:

```
for (x=0; x<W-R+1; x++)
|
|
|
|
0[x] = I[x+r] \ast F[r];
```

Code 1: Untiled 1D-Convolution.

We represent the ordering of the loops in the above loop nest as \( W \rightarrow R \), meaning iteration over \( W \) and \( R \) is the outer and inner loop,
elements, on-chip buffer with $N$ banks, and a NoC.

respectively. Note that due to the commutativity of addition (ignoring floating point errors), we can freely interchange the loops, i.e., as $R \rightarrow W$.

We can also add additional levels to the loop to model tiling or blocking. For example, if $F$ cannot fit in a buffer, we can tile it as shown here:

```c
1. for (rc=0; rc< Rc; rc++) {
  2.   // Rc=ceil(R/Rt);
  3.   for (rt=0; rt <Rt; rt ++) {
  4.     roff = rc*Rt + rt;
  5.     O[x] += I[x+roff] * F[roff];
  }
}
```

**Code 2: Tiled 1D-Convolution.**

That is, we complete the 1D-Conv for an $R_t$ chunk of $F$ at a time, adding an outer loop over the number of tiles $R_c$. This loop nest is therefore written as $R_c \rightarrow W \rightarrow R_t$. Beyond tiling, we can describe parallelism in the computation by conceptually pre-pending parallelism as in Code 2, indicating that iterations of that loop can run in parallel.

An example target accelerator for the 1D-Conv algorithm is depicted in Figure 2. At a high-level, it has $M$ processing elements, an on-chip buffer with $N$ banks allocatable to different tensors at bank granularity, a flexible NoC that can be optimized for different communication patterns such as broadcast, unicast, etc.

Let us assume that the accelerator’s programmable attributes are:

1. $P_0 = \mathbb{R}^3$: a 3-tuple indicating the percentage of banks allocated to each of I, O, and F.
2. $P_1 = \mathbb{Z}^3$: a 3-tuple representing the tile shape of I, O, and F to be fetched from DRAM.
3. $P_2 = \{W \rightarrow R, R \rightarrow W\}$: loop order for the untiled 1D-Conv algorithm represented in Code 1. To support more loops levels due to tiling (as in Code 2), we add additional tiled loop orders to $P_2$ (e.g., $R_c \rightarrow W \rightarrow R_t$).
4. $P_3 = \mathbb{Z}^4$: the loop bound for each loop, e.g., $R_c, W + R - 1, R_t$ in Code 2. Note, we write this attribute explicitly for clarity. In this example, loop bound can be inferred from the current assignment to $P_1$ and $P_2$.
5. $P_4 = \{\text{unicast, multicast, broadcast}\}^3$: NoC communicating patterns for each of the 3 tensors.
6. $P_5 = \mathbb{Z}^4$: Amount of parallelism per PE for each loop level.

**Mapping Space.** Given the above, one can construct a mapping space $M_{a,p}$ for the accelerator and specific 1D-Conv problem (i.e., the specific setting of $W$ and $R$). The accelerator has $D = 6$ programmable attributes and the mapping space size is bounded by the size of the Cartesian product of assignments to each programmable attribute. As discussed in Section 2.1, the mapping space size will be smaller than this upper bound, as some complete assignments to programmable attributes are invalid, e.g., if $R_t < R$ must hold since $R_t$ is a tile of $R$.

**Cost Function.** In the above example, the cost of a mapping $f(m)$ is defined as energy + delay. This can be obtained by simulators or analytical models that represent the actual hardware or using the actual hardware itself. For example, Timeloop [68] (which we use in our evaluation) is a tool that can calculate mapping cost for algorithms representable as affine loop nests (e.g., convolutions).

### 3.1 Challenges in Mapping Space Search

A major challenge in mapping space search is the nature of cost function $f$, which is non-convex and non-smooth. For example, consider the 1D-Conv example from the previous sub-section. The cost of a mapping $f(m)$ is influenced by the various programmable attributes $P_i$ in subtle ways. For example, consider $P_0$, the attribute that represents buffer allocation for each operand/result tensor I, O, and F. If the F tensor is 1 KB in size and allocation falls short of 1 KB, the mapping cost will see a significant energy bump as valid mappings will be forced to tile F, requiring some operand(s) to be re-fetched multiple times to fully compute O. In other words, seemingly minor changes to mapping $m$ can result in non-smooth changes to the overall cost $f(m)$.
We start with a target algorithm. The goal is to find low-cost mappings for a potentially unbounded number of target problems given that algorithm. To accomplish this: First (Phase 1, Section 4.1), we train a differentiable surrogate model to approximate the accelerator’s cost function for all problems making up the target algorithm. This approach fails because the resulting surrogate will be specialized for a given accelerator and the target algorithm, but should be able to return accurate costs for the different problems making up the algorithm. Notice, $p$ is specified as an input to $f^*$. This is a notational convenience that will be important in Phase 2.

While there are multiple choices of differentiable surrogate functions, we use Multi-layer Perceptron (MLP)-based Deep Neural Networks (DNNs) in this paper as they are known to be capable of modeling high-dimensional functions and feature a mature training infrastructure due to the recent advances in Deep Learning. We leave the question of whether simpler, differentiable models are sufficient as future work.

Figure 4, Phase 1, gives an overview of how to construct/train the surrogate. At a high level: We construct a training dataset of input mappings $m$ and their associated costs $c = f(m)$, where $f$ is the accelerator’s reference cost model. Each element in the training set is stored as a 3-tuple: mapping $m$, problem identifier $p_{id}$ (from which $m$ was sampled) and reference model cost $c$.

The distance between the predicted cost, $c^*$, and the actual cost $c$ is used to generate a loss, which is used to train the surrogate using back-propagation [53]. The training procedure can be carried out until satisfactory loss is reached, and well-studied techniques in deep learning can be used to improve the speed of convergence [25, 86].

Superficially, the above process is a “standard” supervised training with Stochastic Gradient Descent (SGD), widely used in modern deep learning approaches. Getting it to work properly for mapping space search, however, entails addressing multiple issues, as described below.

4.1 Phase 1: Approximating the Map Search Space

As discussed in Section 2.2, the cost function $f$ that maps input mappings to a designer-defined cost is non-smooth and non-differentiable. To make $f$ differentiable, which will allow us to generate gradients, we use function approximation (FA) with differentiable surrogates. FAs are often used to reduce a function’s dimensionality, which has been shown to simplify optimization problems in different fields such as reinforcement learning [10, 90]. It has also been used in prior works [15, 42, 55, 62] to improve the speed of evaluation of $f$ to enable rapid searches.

With FA, we generate a smooth, differentiable approximation of the cost function, denoted $f^*$, called the surrogate. Given a mapping $m$ and problem $p$, the surrogate predicts the cost $c^* = f^*(m, p)$, where $c^*$ approximates the actual cost, $c = f(m)$. Therefore, each surrogate is specialized for a given accelerator and target algorithm, but should be able to return accurate costs for the different problems making up the algorithm. Notice, $p$ is specified as an input to $f^*$. This is a notational convenience that will be important in Phase 2.

4.1.1 Generating the Surrogate Model Training Set. The first step is to build a training set to train the surrogate model. We face the following questions:

(1) Which map spaces should be used to populate the training set? The naive approach is to populate the training set with mappings associated with a single problem for the target algorithm. This approach fails because the resulting surrogate will not generalize to unseen problems, requiring us to re-train the surrogate for each new problem we encounter. For example, a new surrogate will be needed every time the $W, R$ settings change for 1D-Conv—clearly undesirable. Instead, we generate training points by uniformly sampling from multiple map spaces, thereby generalizing the surrogate across the family of problems associated with the target algorithm. We empirically observe that the model is able to interpolate and predict correctly for problem instances it hasn’t seen before.
Figure 4: Mind Mappings search procedure. Phase 1: Training the surrogate model $c^* = f^* (m, p_{id})$ based on (mapping, problem id, cost) tuples $(m, p_{id}, c)$. DNN (surrogate) weights $w$ are trained with back-propagation. Phase 2: Given a target problem $p_{\text{target}}$, use the trained surrogate model to iteratively guide a random initial mapping $m@0$ ("mapping at search iteration 0") towards an optimal mapping $m_{\text{opt}}$. In each iteration, $m@t$ is updated using back-propagation with a gradient $\nabla$ of $f^*$ based on $m@t$ with a learning rate $\alpha$. The trained model weights $w$ and the target problem $p_{\text{target}}$ are held constant in this phase.

(2) Based on the choice of map spaces, which mappings should we sample to populate the training set? There are three issues here. First, we must be able to check if a mapping is valid, i.e., belongs to a specific map space. For this, we assume there exists a membership testing function $\text{isMember}(m, p)$ for every accelerator $a$ which returns true if $m \in M_{a,p}$ and false otherwise.

Second, we must decide whether to populate the training set with only valid members of each map space—i.e., mappings that are functionally correct for their given problem—or also consider invalid mappings. In this work, we only populate the training set with valid mappings. Considering invalid mappings may enable the surrogate to better avoid functionally incorrect mappings in Phase 2, e.g., by assigning them infinite cost. However, this may face implementation challenges such as exploding gradients and slow convergence [70].

Third, we need to decide on a sampling strategy to sample from each map space. One option, which we use in this work, is to sample uniformly at random. Specifically, given a problem $p$, we sample from the associated map space $M_{a,p}$ uniformly to generate a mapping and re-sample if the mapping is not valid (see above). This ensures a reasonable representation of the map space, but might suffer from under-sampling from regions that are more important from a training perspective. Other advanced sampling methods can be used, such as sampling from a probability distribution trained to maximize the amount of learning per sample [60]. As seen from our evaluation (Section 5), uniform random sampling facilitates training well and therefore we leave improved sampling methods to future work.

(3) How to uniquely associate each mapping $m$ with its map space $M_{a,p}$? It is possible to have the same mapping $m$ present in multiple map spaces, where each mapping instance has a different cost $c$. Therefore, to ensure correct generalization of the surrogate across different problems for the given algorithm, we must uniquely identify each mapping in the training set with its map space. For this, we need to tag each mapping $m$, added to the training set, with a problem identifier $p_{id}$ unique to the map space associated with its problem $p$. In this paper, we encode each $p_{id}$ as the specific parameterization of the problem, e.g., a tuple indicating the $W, R$ values associated with the problem for the 1D-Conv algorithm (Section 3).

(4) How to calculate cost per mapping? To train the surrogate, we require a reference cost $c = f(m)$ that can be used to calculate the loss w.r.t. surrogate’s predicted cost $c^*$. This can be estimated via running the problem with mapping $m$ on the target hardware or using a cost function estimator such as those described in Section 2.3. As this is a one-time, offline procedure that generalizes over different problems for the target algorithm, its cost is amortized over multiple mapping space searches performed using this surrogate.

We now describe input mapping vector and output cost representation. We describe ideas and challenges here. The concrete representations we use for our evaluation are detailed in Section 5.5.
We achieve this by leveraging the differentiability of the surrogate with respect to the corresponding values in other cost vectors in
whereas the lower bound for energy is given by
\( m \) vector is normalized to have mean 0 and standard deviation of 1
model \( f \) mapping \( o \) only meant to give us a good normalization metric. note that while this lower bound is possibly not achievable, it is
evaluate an architecture with an inclusive buffer hierarchy, meaning
\( \sqrt{\left(\frac{W + W - R + 1 + R}{(W - R + 1 + R) \times \max \_f l o p s}\right) (1 \times \text{cycle/FLOP})} \), whereas the lower bound for energy is given by \( (W + W - R + 1 + R) \times \text{energy} \) times the energy needed to access each word of data once. We
\( c^* = f^*(m@t, p_{target}) \) by forward propagation through the MLP.

4.1.3 Output Cost Representation. A crucial decision to make is to choose a representation for the cost vectors \( c \) and \( c^* \) that facilitates a high quality surrogate. A straightforward approach, often used by prior works, is to represent costs as a combined statistic or the final target, such as EDP, performance/watt, etc.

Instead, we encode costs as a vector of meta-statistics as we observed that this results in higher quality surrogates. For example, for the surrogate we evaluate in Section 5, we represent cost as a vector containing the energy spent accessing each level of the memory hierarchy by each data type (e.g., input/output tensor in 1D-Conv), compute utilization, total cycles, and total energy, although the final metric of interest is EDP. We empirically found that this rich output representation enabled the surrogate model to achieve a high quality surrogate. A straightforward approach, often used
\( \nabla f^* \_\text{p}_{target}(m, p_{target}) \) and combine it with \( m@t \) to form the next candidate mapping \( m@t' \). This process is shown in more detail in Figure 4.

Projected Gradient Descent. Applying Gradient Descent to mapping space search presents two issues. First, after applying gradients, each value \( m_d \) in the mapping vector falls potentially outside the domain of programmable attribute \( P_d \). To address this, we round each \( m_d \) to the nearest value in \( P_d \). Second, after rounding each \( m_d \), the overall \( m \) may be invalid with respect to \( p_{target} \). To ensure the final mapping is valid, we check mapping validity of \( m@t \) at each step \( t \). If validity fails, we calculate nearest neighbor valid mappings based on euclidean distance to \( m@t \) and switch \( m@t \) to the nearest valid mapping before continuing the search. This is a standard approach, often referred to as Projected Gradient Descent [65], used in applications where gradient may steer the parameters out of the valid region [9, 19].

Avoiding Local Minimas. Gradient Descent is infamous for getting stuck in local minima for non-convex optimization problems [46] and our scheme runs the same risk. To handle non-convexity, we introduce randomness at regular intervals throughout the search. Specifically, we add an outer loop to the Gradient Descent algorithm described above, where after every \( N \) iterations, a new random mapping vector is introduced. The decision to replace the current mapping \( m@t \) with the newly sampled valid mapping is based on a probability function accept that helps us balance the trade-off between exploration and exploitation. The choice of \( N \) and the probability function is up to the implementation. For example, the designer may choose to gradually decay the probability of accepting a mapping with higher cost than the already-seen mappings over time, similar to Simulated Annealing. We follow this strategy in Section 5.

Overall, Phase 2 performs the following steps until termination, given a target problem \( p_{target} \):

1. Choose a random valid mapping vector \( m@t \) where \( t = 0 \).
2. Compute \( c^* = f^*(m@t, p_{target}) \) by forward propagation through the MLP.

4.2 Phase 2: Gradient Search to Find High-Quality Mappings

Phase 2, the online part of the search procedure, finds a low-cost mapping \( m_{opt} \) for the target problem \( p_{target} \), as depicted in Figure 4. We achieve this by leveraging the differentiability of the surrogate model \( f^* \) (Phase 1) to obtain gradients, where gradients represent
the change in \( m \) that maximally reduces the cost \( f \) (m). With access to gradients, unlike black-box optimization approaches, we can perform powerful first-order optimization methods which, in effect, incrementally guide any random valid mapping \( m \) towards \( m_{opt} \) using Gradient Descent. The key insight here is that, while the cost function representing the accelerator itself is non-differentiable, its differentiable approximation \( f^* \) should give us access to approximate gradients that can guide the search along the direction of steepest descent.

Gradient Descent with MLP-based Differentiable Surrogates. Section 4.1 described the procedure to obtain a differentiable surrogate \( f^* \) by training an MLP to approximate the cost function. We now use the surrogate to generate gradients that indicate the direction of steepest descent with respect to a candidate mapping \( m \), i.e., \( \nabla f^* \_\text{p}_{target}(m) = [\frac{df^*}{dm_0}, ... , \frac{df^*}{dm_D-1}] \). \( \nabla f^* \_\text{p}_{target} \) is computed using the chain rule across layers of the MLP (surrogate), assuming the MLP parameters and \( p_{target} \) are fixed.

We use the generated gradients to perform a standard Gradient Descent starting from a randomly-chosen initial mapping \( m@0 \) ("m at iteration 0"), setting \( p_{target} = p_{target} \). That is, we iteratively compute \( \nabla f^* \_\text{p}_{target}(m@t) \) and combine it with \( m@t \) to form the next candidate mapping \( m@t' \). This process is shown in more detail in Figure 4.

Avoiding Local Minimas. Gradient Descent is infamous for getting stuck in local minima for non-convex optimization problems [46] and our scheme runs the same risk. To handle non-convexity, we introduce randomness at regular intervals throughout the search. Specifically, we add an outer loop to the Gradient Descent algorithm described above, where after every \( N \) iterations, a new random mapping vector is introduced. The decision to replace the current mapping \( m@t \) with the newly sampled valid mapping is based on a probability function accept that helps us balance the trade-off between exploration and exploitation. The choice of \( N \) and the probability function is up to the implementation. For example, the designer may choose to gradually decay the probability of accepting a mapping with higher cost than the already-seen mappings over time, similar to Simulated Annealing. We follow this strategy in Section 5.

Overall, Phase 2 performs the following steps until termination, given a target problem \( p_{target} \):

1. Choose a random valid mapping vector \( m@t \) where \( t = 0 \).
2. Compute \( c^* = f^*(m@t, p_{target}) \) by forward propagation through the MLP.
(3) Derive gradients using back-propagation via the surrogate with respect to \( m @ t \), \( \nabla = \partial f^*/\partial m @ t \).

(4) Update the mapping vector as \( m @ t + 1 = m @ t - \alpha \nabla \), where \( \alpha \) is a user-specified learning rate.

(5) Project \( m @ t + 1 \) to the valid target map space.

(6) If \( t \% N = 0 \), sample a random valid mapping \( m_{\text{rand}} \).

If accept\( (m_{\text{rand}}, m @ t, T) \) returns true, update \( m @ t + 1 = m_{\text{rand}} \), where \( T \) is a temperature term that is decayed over time.

(7) Increment \( t \) and go to Step 2 until completion.

5 EVALUATION

We now evaluate Mind Mappings. We design representative flexible hardware accelerators using Timeloop [68] and search through their map spaces in the context of two target algorithms, while comparing against several other search methods.

5.1 Experimental Setup

5.1.1 Algorithms. We evaluate Mind Mappings by evaluating two target algorithms, Convolutional Neural Networks (CNN) and Matrixized tensor times Khatri-Rao product (MTTKRP). We evaluate two target algorithms to demonstrate generality, and selected these two in particular given the ongoing effort in the architecture community to build efficient hardware accelerators for CNNs [4, 18, 20, 26, 38, 69, 101] and MTTKRP [37, 87]. CNN layers feature similar, but higher-dimensional, computations as our CNNs and MTTKRP.

5.1.2 Hardware Accelerators. We model the programmable hardware accelerator using Timeloop [68], which uses an analytical cost model to provide a high-fidelity cost estimation for hardware accelerators that implement affine loop nest.

The hardware accelerators we evaluate for both algorithms have the same memory hierarchy, namely a two-level hierarchy with 512 KB of shared buffer and 64 KB of private buffer for each of 256 processing elements (PEs). Buffers are banked and can be flexibly allocated to store any algorithm operand/partial result (tensors in the case of our target algorithms). Each level of the memory hierarchy is coupled with control and address generation logic to support any loop order and tile size (similar to [36]). The Network-on-Chip (NoC) provides parallelism across the PEs along any combination of problem dimensions.

5.1.3 Map Spaces. Given the accelerator architecture \( a \) and target problem \( p \), each mapping \( m \in M_a p \) (Section 4.1.2) is defined by the following programmable attributes for CNN-Layer/MTTKRP, typical in recent accelerators [20, 36, 38, 52, 59, 69].

(1) Tiling: The tile sizes for each dimension (7/4) for each of the 3 levels in the memory hierarchy (DRAM, L2, and L1). (21/12 attributes for CNN-Layer and MTTKRP, respectively.)

(2) Parallelism: The degree of parallelism for each dimension across the PEs. (7/4 attributes.)

(3) Loop Orders: The ordering for each dimensions for each of the 3 memory hierarchies. (3/3 attributes.)

(4) Buffer Allocation: The allocation of banks for each tensor (3/4) for 2 levels of on-chip memory hierarchy. (6/8 attributes.)

These attributes induce a map space that is too large to exhaustively search. For example, the map space size for the ResNet Conv_4 layer (CNN-Layer) is \( \approx 10^{25} \) valid mappings.

To characterize the search space, we sampled 1 M samples from each of the map spaces implied by Table 1 and computed the energy of each sample, which resulted in a \( \text{mean, std} \) of (44.2, 231.4), (48.0, 51.2) for CNN-Layer/MTTKRP respectively, when energy
was normalized to a theoretical lower-bound energy for the given problem.

5.2 Search Methods and Comparison Metrics

We compare Mind Mappings with following popular search methods used in prior work.

1. **Algorithmic Minimum**: Refers to the theoretical lower-bound, possibly unachievable.
2. **Simulated Annealing (SA)**: A popular black-box optimization method [45].
3. **Genetic Algorithms (GA)**: Another popular black-box optimizer that uses evolutionary learning [89].
4. **Reinforcement Learning (RL)**: A popular unsupervised learning approach.
5. **Mind Mappings (MM)**: This paper.

We elaborate on how we implement each search method in Appendix A.

Two key metrics to judge a search method’s effectiveness are the number of steps and amount of time needed to obtain an optimal solution. Accordingly, we compare Mind Mappings against the above methods on two key metrics:

1. **Iso-iteration Search Quality**: All approaches are run for fixed number of cost function evaluations. In case of Mind Mappings, the cost function is the trained surrogate (Phase 1), whereas the other approaches query the actual cost function (timeloop).
2. **Iso-time Search Quality**: All approaches are run until a fixed wall-clock time.

5.3 Surrogate Model

As discussed in Section 4.1, we implement the surrogate as a Multi-Layer Perceptron DNN. We run Phase 1 once for each target algorithm. That is, one surrogate is trained for all CNN-Layer results and a second is trained and used for all MTTKRP results. This shows how the surrogate generalizes across problems for a given algorithm. We elaborate on the details of the surrogate model and the training procedure in Section 5.5.

5.4 Comparing Mind Mappings to Black-Box Optimization Approaches

We plot iso-iteration and iso-time comparisons between Mind Mappings (MM) and the other search techniques (Section 5.2) for different problems (Table 1) in Figures 5 and 6, respectively. In both figures, the y-axis represents normalized EDP with respect to the algorithmic minimum for that problem (Section 5.2). The x-axis represents iterations and time, respectively. To isolate the effects of randomness, each method was run 100 times and the averaged results are plotted, i.e., for each iteration across the 100 runs, we average the EDP across the runs.

5.4.1 **Iso-iteration Comparison**. Figure 5 shows iso-iteration comparisons for all the search methods for every target problem represented in Table 1. Overall, MM outperforms SA, GA, and RL with mappings that have $1.40\times$, $1.76\times$, and $1.29\times$ lower EDP respectively, on average. Further, solutions proposed by MM are only $5.3\times$ away from the (possibly unachievable) algorithmic minimum. MM has a distinct advantage in that it performs a guided search using the approximate gradients derived from the surrogate model, where gradients by definition point at the steepest descent.

For CNN-layer based problems, MM converges to slightly better solutions compared to other approaches and does so within 1000 iterations. The speed of convergence is the key characteristic that demonstrates the effectiveness of gradients and the guided nature of the search. More importantly, MM performs well on every target problem (layer shape for CNN-layer); indicating that the surrogate indeed generalizes and generates useful gradients across the family of problems associated with the target algorithm.

We note that for MTTKRP-based problems, MM converges to much better solutions compared to other approaches. MTTKRP-based problems have much smaller map space sizes ($\approx 10^{19}$ for MTTKRP_0 vs. $\approx 10^{25}$ for ResNet Conv_4) and much lower variance in their EDP (standard deviation of 51.2 vs 231.4 across a
Other methods have known weaknesses, and sometimes map space search-specific weaknesses, that may be contributing to their lower search quality. Specifically: SA suffers from an inefficient traversal in the search space due to the unguided nature of search. Not only is the performance of GA sensitive to the choice of initial population, but GA also heavily relies on an assumption that combining two strong attributes in two mappings (e.g., \( m_i^d \) and \( m_j^d \)) together will make an individual \( m' \) stronger. However, this is not necessarily true in mapping space search. For example, a good tiling for one loop order may not do well when combined with another loop order. RL performs well compared to other black-box approaches, thanks to its superior heuristics using the actor-critic model based on deep deterministic policy gradient (DDPG) [56].

5.4.2 Iso-time Comparison. Figure 6 performs an iso-time study by plotting time as the x-axis (note the log scale) when all search methods are run on an Intel Xeon E5-2637 v4 CPU. Overall, MM outperforms SA, GA, and RL by \( 3.16 \times \), \( 4.19 \times \), and \( 2.90 \times \) respectively on iso-time metric when run MM is run until convergence (62.5 seconds). This is possible since MM does not need to query the expensive cost function (timeloop) every step, unlike other approaches. Instead, MM uses the surrogate model to predict meta-statistics at every step (Section 4.1.3), which in turn generates gradients to guide the next step.

Accordingly, we find that MM is \( 153.7 \times , \ 286.8 \times , \) and \( 425.5 \times \) faster per step than SA, GA, and RL respectively. With recent hardware/software advances in Deep Learning, the gap between MM and other approaches can further widen by using state-of-the-art infrastructure instead of a CPU, as in this evaluation. While RL outperforms SA/GA in iso-iteration search quality, the per step cost of RL is significantly higher than MM. To summarize: MM not only generates higher-quality mappings compared to other approaches, but generates those mappings faster.

Using Surrogate Models for Black-box Approaches. We note that it is possible to improve traditional black-box methods in terms of time-per-step by using a surrogate, as explored in a several prior works [3, 15, 62] (refer to Section 6.1). While such surrogates are not beneficial in finding better mappings (i.e., will not improve iso-iteration search quality), they enable more cost function queries per unit time, which improves iso-time search quality.

5.4.3 Summary. Overall, we note four key takeaways:

1. Generality: Mind Mappings generalizes over different algorithms, architectures, and target problems, as demonstrated by the search performance.
2. Quality of Solution: Mind Mappings finds better/as good mappings compared to other popular methods.
3. Optimality: The Mind Mappings returns mappings that are within \( 5.3 \times \) of the possibly unachievable lower bound, suggesting they are close to the achievable global optimum.
4. Time per Step: Mind Mappings uses a surrogate model instead of the (expensive) accelerator cost function at every step, allowing it to perform more steps per unit time relative to other methods.

5.5 Surrogate Model

We now provide details for the MLP used as the surrogate, and show sensitivity studies used to determine the MLP’s training procedure and architecture.

Input and Output Vectors. The input mapping vector is 62/40 values in length for CNN-Layer/MMTKRP, respectively, which includes the representation of the problem id, tile sizes, parallelism, loop ordering, and buffer allocations. We elaborate on the input vector representation for CNN-layer below.

1. Problem ID \( p_{id} \): \( p_0 = \mathbb{Z}^7 \): a 7-tuple indicating the problem shape \((N, K, C, H, W, R, S); \) see Table 1).
2. Tile Sizes: \( p_1 = \mathbb{R}^{21} \): a 21-tuple representing what factor larger each tile dimension is relative to the corresponding
Choosing a loss function.

Sensitivity to training set size.

Figure 7: Experiments to determine the DNN topology and the loss function.

The output cost vector has 12/15 neurons for CNN-Layer and MTTKRP, respectively. Each neuron represents the energy spent in accessing a specific level of the memory hierarchy (e.g., \( R_e \) in the 1D-Conv tiled example in Code 2). There are 21 factors, for the 7 dimensions in the 3-level memory hierarchy.

(3) **Parallelism:** \( P_3 = Z^{7} \): a 7-tuple to indicate the degree of spatial parallelism for each dimension. Spatial parallelism is represented in terms of a factor for each dimension, similar to how tile sizes are represented (above).

(4) **Loop Order:** \( P_3 = Z^{21} \): a 21-tuple indicating the loop order, represented as a permutation of the 21 (7 dimensions times 3 levels of memory) loops. For example, in 1D-Conv \( W \rightarrow R \) is represented as \([0, 1]\), and \( R \rightarrow W \) is \(1, 0\).

(5) **Buffer Allocation:** \( P_4 = R^6 \): a 6-tuple indicating the percentage of banks in each of the 2 levels of on-chip memory allocated to each of the 3 tensors \((I, O, F)\) in Equation 3).

The output cost vector has 12/15 neurons for CNN-Layer and MTTKRP, respectively. Each neuron represents the energy spent in accessing a specific level of the memory hierarchy (3) for each input/output tensor (3/4), overall energy, compute utilization, and overall cycles for execution. Inputs and outputs are normalized over the dataset to have a mean of 0 and standard deviation of 1, as discussed in Section 4.1.

**DNN Topology and Training.** We choose a 9-layer deep MLP with \([64, 256, 1024, 2048, 2048, 1024, 256, 64, 12/15]\) neurons in each layer for CNN-Layer/MTTKRP, respectively, as the surrogate model based on a grid search. We train the MLP for 100 epochs with a learning rate of \(10^{-2}\), which is decayed by a factor of 0.1 every 25 epochs, and a batch size of 128. We use the Stochastic Gradient Descent (SGD) optimizer with a momentum value of 0.9. Loss over the training duration is depicted in Figure 7a. The MLP converges at around 60 epochs, and the test loss closely follows the train loss, indicating that we do not overfit.

**Dataset.** We train the model with 10 M samples drawn with uniform random probability from the space of representative problems associated with the target algorithm (Section 4). “Representative problems” means we sample from a range of typical values for each parameter making up the problem (e.g., the \(N, K, C, H, W, R, S\) dimensions for CNN-layer). For example, we randomly sample the value of \(K\) for CNN-layer from the range \([32, 512]\), which should cover the typical range of \(K\) in practice \([35, 50, 82]\). That Mind Mappings performs well given this methodology suggests that the surrogate is able to interpolate and predict costs for unseen combinations of problem parameters. Figure 7c compares the search performance on surrogate models trained with 1 M, 2 M, 5 M, and 10 M samples. While the datasets with more than 5 M samples lead to a well-trained model for this problem, surrogates used for the evaluation were trained with 10 M samples. We note that, even when the dataset size is smaller than 5 M, search quality is not significantly hampered.

**Loss Function Choice.** We use the Huber loss \([40]\) function as the loss criterion for training with the SGD optimizer. Figure 7b compares the performance of several popular loss functions used in regression such as the Mean Squared Error (MSE) and Mean Absolute Error (MAE). Interestingly, MSE loss, a widely popular loss function used in regression problems, performs poorly in our setting. We attribute this to the fact that MSE greatly punishes outliers, leading to a large variance in loss and instability in training. By contrast, mean absolute error punishes small variations, leading to sub-par performance. Huber loss is similar to MSE when variations are small and is similar to MAE when the variations are larger, thus creating a good balance between the two loss functions.

**Model size.** When weights are represented as 32 bit floats, the surrogate model takes 35 MB of storage. With recent advances in pruning and quantization \([32, 33]\), the model can be likely be compressed significantly. Therefore, we believe that model size should not be a constraint in adopting Mind Mappings for real world applications.

6 RELATED WORK

We now describe prior work studying mapping space search as well as related search techniques applied to other problems. We summarize the former in Table 2.

6.1 Mapping Space Search

To overcome the challenges in mapping search space, prior works use two main approaches: (i) reduce the time to evaluate the cost function, and (ii) avoid exhaustive search by adopting better heuristics.

6.1.1 Faster Cost Estimation. For unguided mapping space search techniques that rely on exhaustive search or black-box methods,
evaluating more mappings is the key to find higher-quality mappings. However, a key challenge is that the cost to evaluate a mapping using the actual hardware or a representative simulator is non-trivial. To get around this, prior works use several techniques, described below.

dMazeRunner [24], Time-loop [68], Gamma [44] use analytical models built by domain experts that are faster to evaluate than the actual hardware, and are sufficiently representative and flexible to support different mappings. However, building such analytical models is difficult and requires strong domain expertise. Some other works instead do away with domain expertise requirements by leveraging machine learning to build an approximate cost function. For example, AutoTVM [15] and RELEASE [3] use gradient-boosted trees [14]. On the other hand, Adams et al. [2] use Multi-layer Perceptrons (MLPs). We note that while we also use MLPs to build the surrogate, we utilize the differentiability of the surrogate to perform a guided search.

6.1.2 Mapping Space Search with Heuristics. Orthogonal to techniques mentioned in Section 6.1.1 that speed up the cost evaluation, prior works also develop custom/learned heuristics to improve the search itself, so as to avoid brute-force search. dMazeRunner [24], Marvel [12] and Time-loop [68] prune the search space to reduce the number of mappings that need to be evaluated using domain expert knowledge. The key idea is that points in the search space can be eliminated without evaluation, e.g., tile sizes that do not fit in the on-chip buffer. Again, this solution is difficult to scale since it requires extensive domain expertise to create rules to prune the search space.

Several prior works leverage black-box optimization methods to perform the search. For example, AutoTVM uses parallel simulated annealing [45] (SA) to search through the map space. OpenTuner [5] is a program auto-tuner that uses the AUC Bandit Meta technique to combine several methods such as differential evolution, RELEASE [3] and FlexTensor [102] both use Reinforcement Learning (RL) as the cost heuristic to guide the search. Tiramisu [7] and Adams et al. [2] both employ beam search. Finally, TensorComprehensions [97] and Gamma [44] use Genetic Algorithms [39], which are a popular approach used in combinatorial optimization [29, 64, 88].

For all of the above: by definition of being black box, heuristics can only guide the search based on previously visited samples, and therefore require a large number of samples to perform well. As demonstrated in Section 5, Mind Mappings outperforms SA, GA, and RL by utilizing powerful gradient-based optimization with the differentiable surrogate.

6.2 Related Works in other Areas

Beyond mapping space search, the combinatorial search represented in Equation 1 is widely found in other areas such as neural architecture search [103], device placement [72], etc., and insights from related works in these areas can apply to the mapping space search problem.

Surrogate Modeling. Using surrogates for solving black-box optimization problems has been well explored [49]. To predict the performance of a program on a CPU, Ithermal [62] and DiffTune [78] use Recurrent Neural Networks, Ipek et al. [42] use Artificial Neural Nets, and Lee et al. [55] use regression modeling. Deep generative models are proposed as a surrogate in [81], which are differentiable approximations. Function approximation or surrogate modeling has been at the core of modern Reinforcement Learning methods to approximate the large state-space present in real-life problems. Similarly, Mind Mappings uses a differentiable surrogate, while carefully tuning the supervised training methods to adapt to the mapping space search problem.

Search Heuristics. Black-box approaches such as Simulated Annealing, Genetic Algorithms [29, 64, 88], Bayesian Optimization [72, 77, 79], etc., have been widely used in different applications. Recent advances in Reinforcement Learning (RL) have influenced several works [98, 103] to adopt the same, with promising results. Several works have explored gradient-based methods [57, 78, 81, 99], in spite of the cost function being non-differentiable. For example, FBNet [99] uses Gumbel-Softmax [60] to make the discrete choices in their problem differentiable, thereby obtaining gradients.

While gradient-based optimization applied to combinatorial optimization problems with black-box cost functions is not new [17, 31, 57, 58, 81, 99, 96, 99], adapting this to the mapping space search problem—the focus of this paper—is new and faces non-trivial challenges (Section 4).
7 CONCLUSION
This paper proposed Mind Mappings, an efficient method for performing algorithm-accelerator mapping space search. The key idea is to approximate the non-differentiable accelerator cost function with a differentiable surrogate, and to use that surrogate to perform a powerful Gradient Descent-based search.

While Mind Mappings significantly closes the gap to finding optimal mappings quickly, there is still gap left to close. In particular, Section 4 details several areas where the method can be further optimized, ranging from improved sampling methods for training the surrogate to more efficient encodings of accelerator programmable attributes. Long term and with these refinements, we hope that the methods in this paper advance mapping space search to a level closer to its more mature cousin, compilation for general-purpose devices.

ACKNOWLEDGMENTS
We thanks the anonymous reviewers and our shepherd Daniel Jimenez for their valuable feedback. This work was funded in part by NSF under grant 1942888 and by DARPA SDH contract DARPA SDH 4HR011-18-3-0007. Kartik Hegde was funded in part by a Facebook Ph.D. fellowship.

Appendices
A EVALUATION: POINTS OF COMPARISON
We now provide implementation details for each search method used in Section 5.

Algorithmic Minimum. Our baseline represents the theoretical lower-bound EDP for the given accelerator, algorithm and problem. We construct this oracle EDP by taking the product of the minimum energy and minimum execution cycles. The minimum energy is achieved when each input data is read only once and each output data is written only once at each level in the memory hierarchy. The minimum execution cycles are achieved when PEs maintain 100% utilization, i.e., when cycles equals \( \text{required}_{\text{flops}} / (\text{flops}_\text{per pe} \times \text{num}._\text{pes}) \).

Note that in practice, one usually trades-off energy for cycles and cannot achieve the best of both worlds. Thus, the above algorithmic minimum is likely unachievable. We do not calculate the achievable lower-bound EDP, as this requires an intractable exhaustive search.

Simulated Annealing (SA). We implement SA in Python using a popular library simanneal [74]. For each problem evaluation, we let the library perform auto-tuning to get the best hyper-parameters for SA such as the temperature and annealing factor. We interface the library with the Mind Mappings tuner to perform mapping space search.

Genetic Algorithm (GA). We implement GA in Python using DEAP [28], a popular GA library. Based on the extensive literature on parameter tuning for GA [13, 16, 34, 67, 84] and a grid search, we set an initial population size of 100 and crossover/mutation probabilities of 0.75/0.05, respectively. Each individual is a mapping ranked based on fitness, which represents the optimization objective, EDP. Every iteration, we perform a cross-over and mutation over the population. A cross-over results in swapping attributes of one individual with the other while a mutation is implemented as a 0.05 probability of a random update for each of the mapping’s attributes. At the end of each generation, individuals are chosen based on their fitness for the next generation.

Reinforcement Learning (RL). We implement RL in PyTorch [71], based on the Deep Deterministic Policy Gradient (DDPG) [56] implementation from HAQ [98]. In the RL setting, the mapping problem is modeled as a Markov Decision Process (MDP) [5], where each mapping is a state in the MDP, an action results in a move to a target state and the cost of the mapping is the reward. In each episode, the RL agent starts from a random initial state, takes an action to move to a target state and updates its policy based on the reward. In this process, the agent learns the optimal action to take given a state in the space. The learning process uses the actor-critic method [48], which is a widely-used policy gradient algorithm. The actor and critic functions are approximated with two fully-connected DNNs with 300 neurons respectively.

Mind Mappings (MM). We implement Mind Mappings (Section 4) using a trained surrogate model (elaborated in Section 4.2) as described in Section 4.1. We inject randomness at an interval of every 10 iterations to avoid local minimas, as described in Section 4.2. We use simulated annealing with a temperature of 50 initially to decide the acceptance of random injections, which is annealed every 50 injections by a factor of 0.75. We use a learning rate of 1, and we do not decay the learning rate throughout the procedure. We choose the learning rates and injection interval via a grid search.

B MIND MAPPINGS API
The Mind Mappings API exposes an optimization framework for mapping space search that can be used in compilers and frameworks targeting a specialized hardware accelerator, such as TVM [15], PyTorch [71], TensorFlow [1], etc. A surrogate model is trained offline for the target algorithm-accelerator pair to approximate mapping cost, using techniques described in Section 4.1. Then, during the compilation, the Mind Mappings API takes the trained surrogate model and the target problem \( p \) as input and returns a low-cost (ideally optimal) mapping \( m_{opt} \) that minimizes the problem’s execution cost on the given accelerator.

The Mind Mappings API requires the following routines: (1) getMapping: gives a random valid mapping, (2) isMember: checks if a mapping is valid, and (3) getProjection: returns a projection from an invalid mapping to the nearest valid mapping. We have open sourced the Mind Mappings framework here: https://github.com/kartik-hegde/mindMappings.

REFERENCES


