Mapspace Optimization for Tensor Computations with Bayesian Learning

J V Iniyaal Kannan

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-91
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-91.html

May 10, 2023

Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

First and foremost I would like to thank Professor James Demmel and Grace Dinh for their invaluable guidance, unwavering support, and encouragement throughout my Master’s program. Their expertise in the field and insightful feedback have been instrumental in shaping my academic growth. I will always be grateful for their mentorship and feel fortunate to have the opportunity to learn from them.
Thanks to Professor Sophia Shao for graciously agreeing to review my thesis and providing thoughtful suggestions.
I would also like to extend my heartfelt appreciation to my family for their unconditional love, support, and motivation. I could not have accomplished this milestone without their trust and belief in me.

Mapspace Optimization for Tensor Computations with Bayesian Learning

Iniyaal Kannan Jegadesan Valsala

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

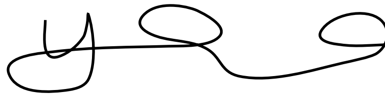
Committee

James Demmel
Research Advisor

9 May 2023

(Date)

* * * * * * *



Sophia Shao
Second Reader

5/10/2023

(Date)

Mapspace Optimization for Tensor Computations with Bayesian Learning

Iniyaal Kannan Jegadesan Valsala

***Abstract*—Tensor computations are becoming increasingly important with the emergence of fields such as AI, data analytics, and robotics. Memory access cost is the bottleneck in performance for these workloads. New architectures with specialized memory layouts and parallelizable elements are being designed for faster computation. To fully exploit such an architecture's capabilities and achieve maximum improvement in performance, an optimal communication avoiding mapping from algorithm to hardware is needed. Manually finding this hardware-specific, energy efficient mapping is time-consuming and requires expertise in multiple domains. Traditional optimization methods like gradient descent are unsuccessful in finding an optimal mapping because the mapping space is non-smooth and non-convex. Other ML based feedback-driven approaches find good solutions, but do not generalise well to new architectures.**

In this paper, we propose using GPTune—an autotuning framework based on Bayesian optimization —to navigate this search space. Our experiments show that GPTune finds efficient mappings in far fewer iterations compared to Timeloop-mapper's random search. GPTune also builds surrogate models that can be used for transfer learning and to potentially reduce the dimensionality of the mapspace. Furthermore, this paper analyses mapspace encodings that work best for tuning.

I. Introduction

The slowing down of Moore's law along with the widespread adoption of ML in recent years has led to an increased focus on developing specialized hardware for ML models like DNNs and Transformers. While these domain specific hardware architectures have sped up computations, they are more complex and offer flexibility that requires expertise to exploit.

Tensors and their computations like matrix multiplication or convolutions are fundamental to DNN training and inference applications. A tensor computation can be executed in very large number of ways depending on the target architecture's properties. A *mapping* specifies how a parameterized tensor *problem* is translated to hardware instructions and executed in the target architecture. The *cost* of a mapping is the runtime, energy or another metric measuring the performance of the problem mapping on the target hardware architecture. The *mapspace* consists of all the possible ways a problem can be mapped onto the target hardware and executed correctly. Exploring this mapspace to find the optimal mapping that minimizes the cost of running the problem on the architecture is critical to fully utilising the hardware architecture. An unguided, poor choice of mapping could hurt performance by multiple orders of magnitude.

The mapspace is very challenging to search because it is highly non-convex and differs with problem and architecture. There are also many algorithmic constraints (e.g. loop carried dependencies in computation have to be respected) and hardware constraints (e.g. data mapped to buffer has to fit into buffer) that need to be satisfied when optimizing over this space. So it is difficult to use straightforward, well-understood optimization methods like linear programming or gradient descent to minimize the cost function. There are broadly 3 approaches utilised to prune the mapping space and find a good mapping.

- *Heuristics* perform one-shot analytic optimizations over an (either explicit or implicit) performance model. This method generalizes across architectures and problems, but it is often limited to optimizing only some of the optimizable parameters (e.g. loop tilings or reorderings only). This leads to sub-optimal outcomes especially with complicated architectures because the interactions between optimization parameters has an impact on the performance. These methods also rely on analytical model evaluations, which only approximate the true performance of the mapping.
- *Random search* methods sample mappings randomly from the search space to find an efficient mapping. The size of the mapspace necessitates a large number of samples to achieve good performance, so these methods use cheap, analytical performance models that are able to evaluate $1000s$ of mappings in minutes. Since the mapspace exploration is not guided, mapping high dimensional problems on complicated architectures would take a long time to converge to a reasonable mapping.
- *Feedback-driven methods* use statistical or ML methods to iteratively explore and learn the mapspace. These approaches encompass black-box optimization techniques such as genetic algorithms, and gradient-based methods, which can build surrogate functions for performance based on input parameters. However, these approaches do not generalize well across architectures.

In this thesis, we focus on exploring the use of Bayesian optimization [17] techniques to search the mapspace using Gaussian processes. GPTune [5] is an autotuning framework based on Bayesian optimization to find optimal performance tuning parameters for high performance computing applications. We conducted experiments and evaluated results for the following questions to determine the feasibility of using GPTune to find high performance mappings.

- **Does GPTune converge faster than random search?** GPTune's mapspace exploration consistently finds performant mappings in far fewer iterations than random

search. For matrix-multiplication and 2D-convolutions, GPTune converges in less than 50 runs (on average) to performance values that require a random mapper over 600 iterations to beat.

- **Does GPTune converge to a better mapping than random search?**
  GPTune converges to better mappings faster than a random mapper. For 2D-convolutions, The random mapper took $\approx$ 350 iterations to achieve an energy usage value attained by GPTune in 50 iterations (on average).
- **Does GPTune's surrogate model enable transfer learning between architectural parameters (e.g. buffer sizes)?**
  GPTune has transfer learning algorithms that enable learning across various parameters of a problem. We focus on learning the mapspace of the same problem size mapped onto architectures with different memory capacities. Based on our experiments for matrix multiplication, GPTune with transfer learning achieved a performance value that took 2X iterations for standard GPTune and 800X iterations for the random mapper to beat.
- **Does GPTune's surrogate model enable us to determine the effect of different mapspace parameters on the cost of the mapping?**
  By using Sobol analysis [18] on the surrogate model, we were able to identify the mapspace parameters that had the most significant impact on performance. For matrix multiplication, we found that the top 4 most sensitive parameters affected the performance almost 2X more than the other parameters.
- **Does GPTune's surrogate model reduce the dimensionality of the problem by finding the most important mapspace parameters?**
  This is our ongoing work. We are experimenting with using only the highest valued parameters from the sensitivity analysis instead of all the parameters for mapspace search.
- **What mapspace encodings lead to better results with GPTune?**
  Since GPTune is a general autotuning framework, it does not provide specific mapspace parameter encodings. So we need to encode mapspace parameters as numerical or categorical values. We have found out that the quality of the encoding has an impact on the mapspace search. Our experiments show that categorical encodings for loop order work better for matrix multiplication (lower-dimensional), and numerical encodings work better for 2D-convolutions (higher-dimensional).

GPTune's mapspace exploration consistently finds performant mappings in far fewer iterations than random search. This enables optimization for hardware architectures where evaluations can be extremely expensive. We have promising results for transfer learning using the surrogate model generated by GPTune. Using transfer learning, performance of algorithms on new, unseen architectures can be evaluated. Using Sobol analysis [18] on the surrogate model, we were also able to find the mapspace parameters that had the most impact on performance. Only optimizing the important parameters reduces the dimensionality of the problem and could potentially lead to better mappings.

II. Mapspace Optimization

A. Mapspace

The mapspace is defined as the set of all valid mappings of a problem onto a target architecture. A mapping is a specification of how a problem is to be laid out and executed in hardware. In this paper, we focus on 2 main mapping parameters - tiling and ordering. The tiling determines how the problem is broken down into smaller blocks and processed at each memory level. The loop ordering determines the computation's dataflow, affecting data reuse. For a mapping to be valid, it must satisfy (1) algorithmic constraints to ensure correctness of computation and (2) the hardware constraints of the architecture.

The size of a mapspace is the *product* of the sizes of the optimization attributes. So, the number of optimization parameters in a mapping leads to a combinatorial explosion of valid mappings. Moreover, the mapspace is non-convex and non-smooth even for basic cost functions like energy and runtime.

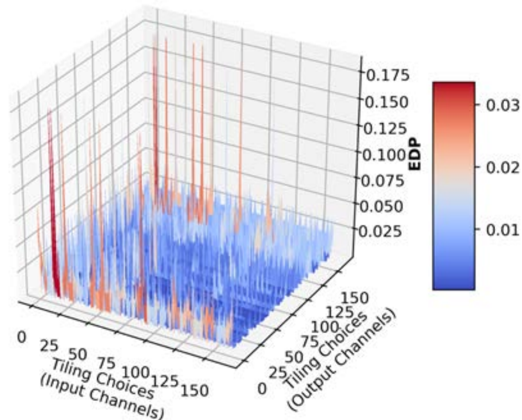


Fig. 1. Mapspace of a CNN layer from Mind Mappings [9] paper

Figure 1 shows the energy cost surface of Convolutional Neural Network(CNN) layers evaluated on an accelerator [9]. The x and y axes represent values for different mapspace parameters (tile sizes for the tensors) and the z axis represents the cost (energy-delay product, or EDP). As we can see, the mapspace is highly irregular with many local minima. Hence, for feedback-driven methods like genetic algorithms, random starting points and increased mapspace exploration can lead to better mapping results. It is due to this mapspace complexity that random search methods are still relevant and can produce high performance mappings (though a significantly higher number of iterations compared to other guided search methods (GAMMA [12], Mind Mappings [9]) are needed).

B. Mapspace Search

The size of the mapspace makes exhaustive search infeasible, and a guided approach to exploring and evaluating mappings requires fewer iterations to converge to a high performance mapping. To navigate the space, (1) cost evaluation of a mapping on the hardware and (2) method to determine the next mapping to explore are needed.

To obtain a mapping's cost on a target architecture, several methods can be utilised.

- *Using cycle-accurate simulators* is expensive. Firesim [13] takes approximately 3 minutes to run and evaluate a mapping on a standard AWS FPGA; For feedback-driven algorithms like GAMMA [12] and Mind Mappings [9], a large number of function evaluations are needed to train the model for a specific problem and hardware. Since these methods do not generalise well to new problems and architectures, additional evaluations are needed to retrain the model and determine an efficient mapping for the new hardware.
- *Fast analytical models* like Timeloop [16] are used by methods that require a large number of iterations to converge. Timeloop, when given a problem, hardware architecture and mapping, computes a cost-estimation for the mapping. Timeloop is cheap and fast to run. Several thousand mapping evaluations can be completed within minutes. However, these analytical models can be inaccurate by up to two orders of magnitude. For feedback-driven methods, these significantly incorrect evaluations could mislead the mapspace exploration and lead to inferior choices of mappings.

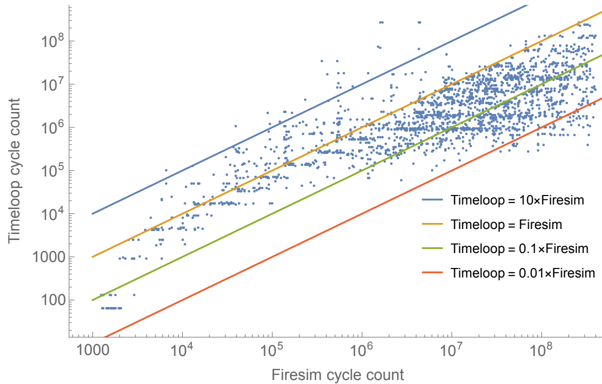


Fig. 2. Scatter plot of the ratios of figures generated by Timeloop [16] and Firesim [13]

Figure 2 shows a log-log plot of the ratios of the cycle counts generated by Timeloop and Firesim for approximately 2000 mappings on the GEMMINI accelerator [8]. The plot shows how analytical model evaluations could deviate by over 2 orders of magnitude from the true value. Optimization methods that converge to high performance mappings in fewer iterations can utilise expensive, cycle-accurate simulators. The increased precision in mapping evaluations enhances mapspace search and enables the discovery of more efficient mappings.

Different techniques are utilised by Mind Mappings [9], GAMMA [12] and Ansor [20] to choose candidate points to evaluate. The selection of these points directly impacts the exploration of the mapspace.

- *Mind Mappings* uses a gradient based approach to find mappings to evaluate. Since the mapspace is non-convex and non-smooth, an approximate cost function is used to obtain a differentiable surrogate function. This surrogate is used to generate gradients that show the direction of steepest descent from a mapping. Following this direction, a new mapping is chosen to evaluate and the process continues till termination.
- *GAMMA* uses evolutionary search methods to determine new mappings. Mapspace optimization parameters (e.g. tile sizes), are mutated to generate new valid mappings. Figure 3 shows GAMMA's logical flow and structure. After randomly initializing a population size of P mappings, different evolutionary methods are used to mutate the population. A *gene* represents the encoded value of one of the mapspace optimization parameters (e.g. tile size). A *genome* comprises a set of genes and represents a complete mapping. Genes and genomes are mutated in the evolution phase. For example, the crossover mutation shown in Figure 3 involves picking two genomes from P and interchanging their genes, and the reorder mutation picks two genes and swaps their positions within the genome, leading to a different ordering of the loops in the mapping. After the evolution phase, each genome is decoded to a mapping and its performance value is computed. GAMMA [12] uses MAESTRO [14], an analytical model to evaluate the mapping. The subset of P that performs better is selected to proceed to the next evolution phase.

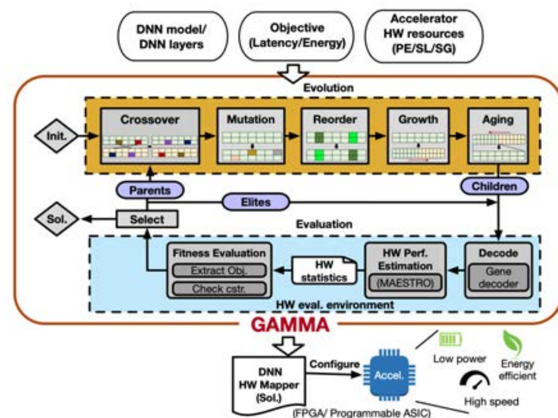


Fig. 3. Genetic Algorithm (GAMMA) [12] flow

- *Ansor* uses a hierarchical representation of the mapspace to explore new combinations of optimization parameters. Ansor's input is a set of DNNs to be optimized. Figure 4 shows the hierarchical approach utilised by Ansor

to construct a large mapspace and search it efficiently. The program sampler in Figure 4 enumerates high-level structures of tensor programs and leaves low-level optimization parameters (e.g. tile size, parallelization) as annotations that need to be tuned. Random sampling is used to select programs from this stage to proceed to the next stage. In the performance tuning module, evolutionary search operations — mutation and crossover — are used to fine tune the annotated optimization parameters. A learned cost model is used to evaluate the mappings generated after fine-tuning. While a learned cost model is significantly faster than utilizing a cycle-accurate simulator, Figure 2 indicates that its performance measurement may deviate considerably, resulting in inferior performance when executed on real hardware.

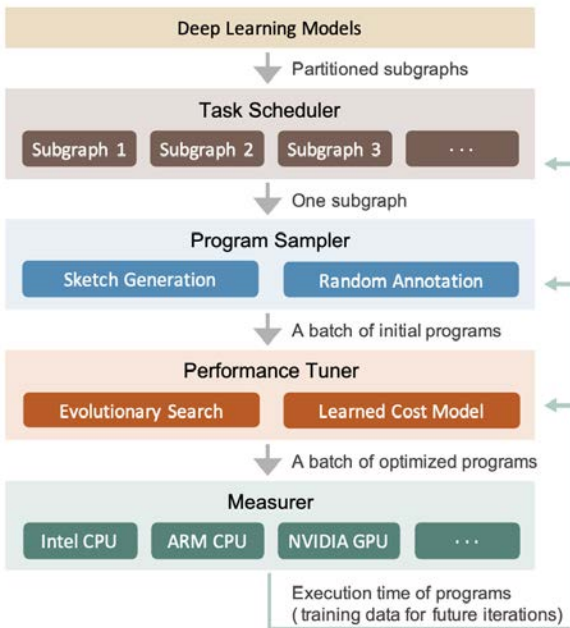


Fig. 4. Ansor [20] Hierarchical flow

C. Our Approach

Bayesian optimization can be applied to the mapspace search problem by encoding mapspace parameters appropriately. Bayesian methods have shown to optimize complex, multidimensional functions with irregular search spaces using limited function evaluations [22]. Bayesian methods are crucial to optimize models in domains where obtaining samples to evaluate are very expensive. In the case of mapspace exploration, obtaining cycle-accurate simulations is expensive.

In this thesis, we utilise Bayesian methods to navigate the mapspace for tensor computations. GPTune is the tool we use to build Gaussian Process surrogate models for Bayesian Learning. Surrogate models can facilitate transfer learning between different hardware architectures or problem sizes. If a surrogate model is generated for a given hardware *X* and problem *P*, the model can be used to guide GPTune's search to faster convergence for hardware *Y*. We also use surrogate models to understand how optimization parameters interact with each other, and to identify which parameters have the biggest impact on performance. Fewer optimization parameters reduces the dimensionality of the problem. We are also exploring how convergence and performance mappings are affected by dimensionality reduction.

III. Background

A. Example Tensor Computation — Matrix Multiplication

A *Tensor* is defined as a generalized, multidimensional array, represented by an array of components that are functions of the coordinates of a space. Tensors are basic building blocks of modern ML. They are used to organize and represent large datasets and their computations are vital to training and testing models. Matrix multiplication and convolutions are the tensor computations we focus on optimizing using GPTune. To access and manipulate elements in a tensor, nested loops are required. So, tensor operations typically have a nested loop structure that can be optimized for the hardware architecture.

Matrix multiplication between two matrices can be defined as

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \quad (1)$$

A and B are matrices of dimensions $m \times n$ and $n \times p$, respectively, and C is the resulting matrix of dimensions $m \times p$.

Equation (1) can be represented with a nested loop structure as follows

```
for i in range(m):
    for j in range(p):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

Listing 1: Python code implementation of matrix multiplication

Memory access cost is the bottleneck in performance for such tensor computations. By changing the nested loop structure, it is possible to take advantage of faster levels of memory and achieve improved performance.

B. Loop Optimizations

A DNN accelerator typically contains multiple hierarchical memory levels, spatial arrays of processing elements (PEs) and networks-on-chip (NoC). In a memory hierarchy, higher levels of memory are limited in size but offer fast data retrieval. Lower levels of memory are bigger in size but it takes multiple orders of magnitude more time to retrieve data. Temporal and spatial locality are typical data access patterns.

- *temporal locality* refers to the tendency of programs to access the same data repeatedly over a period of time.
- *spatial locality* refers to the tendency of programs to access data that is stored near recently accessed data.

Caches exploit these access patterns to improve efficiency. When a piece of data is accessed from a lower level of memory, the entire block of data around the accessed location is brought into the higher level (spatial locality) and the location of the data access is stored for future reference (temporal locality). By optimizing the nested loop structure, memory accesses can be restricted to higher, faster levels of memory leading to significant improvement in performance.

1) Loop Tiling: Nested loops can be tiled to split a loop's computation into smaller chunks that fit into memory levels. Loops have to be tiled ensuring correctness of computation. Nested loops that have data or control dependencies between iterations must be tiled carefully to respect dependencies. Some loop tilings may incur high communication overheads and perform poorly. The optimal tile size for a problem heavily depends on the target architecture and the characteristics of the problem. Hence, the tile size is an important mapspace parameter to determine.

```
for x in range(0, m, t1):
 for y in range(0, p, t2):
  for z in range(0, n, t3):
   for i in range(x, min(x + t1, m)):
    for j in range(y, min(y + t2, p)):
     for k in range(z, min(z + t3, n)):
      C[i][j] += A[i][k] * B[k][j]
```

Listing 2: Python code implementation of tiled matrix multiplication

The python code snippet in Listing 2 describes a tiled matrix multiplication nested loop structure. A loop with range(start, end, stepsize) means the loop runs from start to end(exclusive) in increments of stepsize, with a total of $\lfloor \frac{start-end}{stepsize} \rfloor$ iterations. The implementation of tiled matrix multiplication partitions matrix C into sub-matrices of dimensions $t1 \times t2$, matrix A into sub-matrices of dimensions $t1 \times t3$, and matrix B into sub-matrices of dimensions $t3 \times t2$. The variables x, y, z represent the starting indices of each tile. The loops are tiled for a 2-level memory hierarchy. Loops can be tiled at each level of memory, so for more complicated architectures, there are more tile sizes to choose. Furthermore, tiles need not be split perfectly. They could be split imperfectly and a tail case could be added to complete the remaining computation. Finding the optimal values for each tile size, ensuring correctness and satisfying memory level constraints is the optimization problem.

2) Loop Reordering: Loops in a loop nest can be reordered to change the data access pattern of the computation. Changing the loop order can enable sequential access of data that improves cache usage leading to better performance.

Assume the matrices are stored in row-major order in memory. Listing 3 shows a simple reordering of the matrix multiplication problem in Listing 1. The loop reordering from i, j, k to i, k, j can lead to significant improvements in

```
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i][j] += A[i][k] * B[k][j]
```

Listing 3: Python code implementation of reordered matrix multiplication

performance because the innermost loop has increased cache utilisation. The i, j, k loop ordering accesses the elements of matrix B in column-major order. When the size of the cache block is smaller than the matrix size, accessing the elements in column-major order can result in thrashing. This leads to cache misses and worse performance. By reordering the loops to i, k, j, the innermost loop now accesses the elements of matrix B in row-major order. Since the entire row is retrieved into the cache, future accesses of matrix B elements are faster. Computation correctness has to be preserved when reordering loops. Not all reorderings are valid because of loop dependencies.

C. GPTune

1) Overview: GPTune is a Bayesian autotuning framework used to find the best values of performance parameters for applications where obtaining function evaluations is expensive and limited. The performance metric used for autotuning can be any measurable quantity. GPTune carefully samples points from the search space to evaluate. Using these points, GPTune builds a Gaussian process (GP) surrogate model. Once the model is built, GPTune predicts a point it believes will perform well. The performance of the predicted optimal point is measured and is used to update the model for future iterations [5]. A GPTune problem can be defined by the following parameters.

- Task Parameter Input Space: Describes the input space for the optimization problem. In the context of matrix multiplication optimization, the input space comprises the dimensions of the matrices involved in the computation as well as the architectural parameters of the hardware (e.g. sizes of different levels of memory hierarchy).
- Tuning Parameter Space: Represents the parameter configurations we are trying to optimize. GPTune samples points from this space to evaluate. Hence, the tuning parameter space should span the entire mapspace of a problem. The point sampled also needs to be encoded to a mapping onto hardware. For matrix multiplication optimization, the tuning parameter space consists of the mapspace optimization parameters like tile sizes and loop orders.
- Output Space: Defines the cost model space. This is the metric used to compare and evaluate different candidate solutions. It can be any measurable quantity like runtime, energy, cycle count.

2) GPTune Database: GPTune also enables crowd-based autotuning [6]. Essentially, GPTune collects function evaluation data used by users to build surrogate models for their problems. These expensive evaluations are retained in the GPTune DB (database). These evaluations are then used to build accurate surrogate models without having to run expensive evaluations. For mapspace exploration, a cheap, fast analytical performance model like Timeloop or MAESTRO can be used to generate 1000s of mapping evaluations. Once these evaluations are loaded into the DB, GPTune can use it to build a surrogate model and converge in fewer iterations.

3) Transfer Learning: Transfer Learning based Autotuning (TLA) leverages the information in a GPTune DB to tune a new, unseen problem. The evaluations in the DB must have a correlation to the new problem to obtain good results. For mapspace exploration, TLA can be used to generalize over (1) different problem dimensions on the same hardware architecture and (2) same problem on different hardware configurations. GPTune offers two types of TLA algorithms. Both these methods can be employed for mapspace TLA optimization, but the high dimensionality of the mapspace could hinder the usage of Type II TLA [5].

- Type I TLA: Builds a surrogate model using data in the GPTune DB. The surrogate model is further improved by obtaining evaluations from the new target task. Pre-collected DB data enables GPTune to learn the mapspace and converge to a high performance point for a new task in fewer iterations. If different tasks are present in the DB, a surrogate model is built for each task and can be combined by summing, stacking or computing weighted averages.
- Type II TLA: In this method, a surrogate model is not built using the function evaluations from all the tasks. Instead surrogate models are developed for the function pairing the task to its optimal tuning parameter configuration. No samples are obtained from the new target architecture to incorporate into the surrogate model. Using the newly generated ”task to optimal mapping” surrogate models, GPTune samples a single configuration as the optimized point for the unseen problem. TLA_II could serve as a cost-effective one-shot optimizer to obtain reasonably efficient mappings in the mapspace.

D. Timeloop

Timeloop is a framework for modeling and evaluating dense and sparse tensor algebra workloads on DNN accelerator architectures. Timeloop is a fast, analytical model for performance evaluations. It provides good approximations of a mapping's performance by calculating throughput and access counts analytically. This is possible because computation and data-movement patterns in DNN operations are mostly deterministic. Timeloop consists of timeloop-model and timeloop-mapper.

1) Timeloop-model: takes as input a problem specification, hardware architecture, a valid mapping and outputs a cost estimation for the mapping. Timeloop-model can compute performance, area and energy projections within seconds. It also provides detailed statistics for each level of the memory hierarchy. Loop optimizations are specified in the mapping section using keys. An error is thrown if the mapping is invalid. Timeloop-model directly invokes Accelergy [1] for its internal computations.

2) Timeloop-mapper: takes as input a problem specification, hardware architecture and mapper configuration to construct and search the problem's mapspace. Timeloop-mapper uses the model cost function to evaluate and compare different mappings. It focuses on 3 primary optimization parameters — loop tiling, loop reordering and spatial execution. Timeloop-mapper can search the mapspace using algorithms like exhaustive search, linear-pruned, random-pruned and hybrid. For mapspace exploration, random-pruned seems to work best. If after sampling a point from the mapspace, the point is invalid for the specified architecture (e.g. tile size doesn't fit in memory), the sample is rejected and the sampling process continues.

IV. Method

A. Overview

Our work leverages GPTune — a Bayesian autotuning framework to build surrogate functions that guide mapspace search. We enable optimization of software taking the target hardware into consideration. This enables us to find high performance mappings that exploit the features specific to the hardware.

We conducted experiments for two important tensor algebra computations — matrix multiplication and 2-D convolution. Timeloop provides the performance evaluations for mappings chosen by GPTune. Bayesian optimization requires a performance metric to minimize. Energy is the performance metric we focus on optimizing, but a different metric could also be optimized with minor changes to the method. We use GPTune's multi-task learning (MLA) framework for the mapspace optimization. MLA has a (1) Modeling Phase and (2) Search Phase. In the Modeling Phase, GPTune randomly samples points from the tuning parameter space and obtains their performance evaluations using Timeloop. These function evaluations are used by GPTune to generate a surrogate model for the Search Phase. The Search Phase uses PyGMO, pymoo or SciPy algorithms to search for the next candidate sample point [5]. The evaluations computed in the Search Phase are used to further improve the surrogate model. The number of pilot samples and total iterations of the Search Phase are user specified parameters.

Our objective is to evaluate whether Bayesian optimization techniques present benefits in contrast to alternative optimization methods. We expand on the questions discussed in Section I and explain the experimental procedure to determine if Bayesian optimization offers the benefits expected.

(1) How effective are Bayesian optimization methods at finding high performance mappings compared to one-shot and random methods? The efficacy of a mapspace exploration method can be evaluated based on factors such

as the time to solution or the rate of convergence. Our focus with GPTune is the rate of convergence. If only a small number of function evaluations are required to find a high performance mapping, expensive cycle-accurate simulators can be leveraged. Other methods like random search utilise analytical models because they are much faster, but their approximations could deviate significantly as shown by Figure 2. GPTune has the added advantage of crowd-tuning. This allows for the storage of costly function evaluations in the GPTune DB, which can then be utilized to construct a more accurate surrogate function.

To compare the rate of convergence to a high performance mapping, we should evaluate GPTune [5], CoSA [11] (one-shot optimizer), and Timeloop-mapper [16] (random search) for identical problem dimensions and target architecture. For a GPTune run, the number of pilot samples and search phase iterations is user specified. Based on our experiments, high tuning space dimensionality and large number of function evaluations (in the pilot or search phase) leads to very slow iterations of the search phase.

(2) Can GPTune enable transfer learning between problem sizes or between target architectures? We would like to understand how well GPTune's surrogate models generalise to generate efficient mappings for new, unseen target architectures. Optimization techniques like GAMMA [12] and Mind Mappings [9] need to be re-run from scratch for new hardware architectures and problem sizes. This is especially a problem when obtaining function evaluations are expensive. If Bayesian optimization techniques are able to learn mapspace structures, the surrogate models can facilitate the discovery of high performance mappings for new architectures in very few iterations.

Transfer learning can be enabled by using MLA, TLA_I, or TLA_II techniques. GPTune incorporates MLA and TLA to share knowledge of obtained performance samples among multiple tasks, improving tuning results. It's possible to do transfer learning of new problem sizes on the same hardware architecture or the same problem sizes on a new target architecture. When both the problem dimensions and the hardware are modified, there is a lack of correlation between the mapspaces for the existing function evaluations and the target. To use TLA_I to find mappings for a problem *P* on a new hardware *Y*, the DB should be pre-populated with function evaluations for problem *P* on hardware *A, B, C...*. TLA_I works better when the DB contains evaluations from different mapspaces. TLA_I requires function evaluations from the target architecture in the search phase. This technique improves the convergence rate by leveraging evaluations from previous mapspaces to learn the new mapspace faster. TLA_II on the other hand, does not require evaluations on the target architecture. It uses previously built surrogate models to perform a one shot optimization to find an efficient mapping on the new architecture.

(3) What encodings for mapspace parameters work well for Bayesian optimization approaches? Since GPTune is a general framework for autotuning applications, it only provides numerical or categorical encodings for tuning parameters. Although tile sizes are numeric variables, it is necessary to impose constraints on them to ensure computational correctness and to satisfy hardware specifications. While GPTune enables the specification of constraints, including all algorithmic and hardware constraints results in a slow sampling process where the majority of sampled points get rejected. So, after GPTune samples a tiling parameter, it has to be encoded into a valid mapping for Timeloop to evaluate its performance. For other mapspace parameters, such as loop orderings and spatial mappings, there is no obvious numeric encoding that would work best. They could also be sampled as categorical variables. Different encodings could work better depending on the problem and hardware architecture, so mapspace encodings are definitely worth exploring.

We have experimented with different encodings for GPTune. For tiling parameters, we sampled them as (1) integers and rounded them to ensure perfect factorization of the loop and (2) real numbers representing the aspect ratios of the tile and modified them to be tile sizes. For loop ordering, we sampled reordering parameters as (1) real numbers and used a ranking based approach to tile the loops and (2) categorical variables with each variable representing a permutation of the loops and the sampled permutation is used as the loop ordering for the memory level.

4) Can we use performance evaluations from analytical models to reduce the cost of optimizing with a cycle-accurate simulator? Surrogate models built by GPTune can be used to model the function evaluations from a cheap, less accurate performance model. These surrogate functions can be used as input parameters for the model of the expensive simulator. The non-convex nature of the mapspace makes exploration of new regions necessary. Assuming that the cheap model provides reasonably accurate approximations, it enables the discovery of localities with high performance mappings. Using the surrogate model built with a large number of cheap evaluations, GPTune gains a better understanding of the search space, thereby reducing the number of evaluations required from the expensive simulator to converge to an optimal mapping.

B. Hardware Architecture

For the experiments we run, we use a hardware architecture based on GEMMINI [8]. It has a 4 level memory hierarchy: a register, accumulator, scratchpad and DRAM. As an example, consider a 2D-Conv with b batches, c input channels, k output channels, and filters of size $r \times s$ and outputs of size $w \times h$.

- *Registers* are typically the smallest level of memory and the fastest. They are single storage units. Multiple registers can be accessed in parallel. For the 2D-Conv problem, when tiling the loops, only the input, weight and output tensor data tiles can be stored in this level.
- *Accumulators* are a type of register used to store intermediate accumulated data in multistep computations. In DNN architectures, they are typically used in multiply-

accumulate (MAC) operations. For the 2D-Conv problem, only the output tensor's data tiles can be kept at this level.
- *Scratchpad* memory is a small, high speed on-chip memory level that is used to temporarily store data during computations. Processing elements in the architecture can access data stored in the scratchpad relatively fast because it is on-chip. When tiling a 2D-Conv problem, only the input and weight tensor data tiles can be stored at this level.
- *DRAM* is typically the main memory for the architecture and stores the the entire input, weight and output tensor data. It is located off-chip so it is much more expensive to access than the other memory levels that are on-chip. It is also much bigger in size in comparison to the on-chip levels.

After an architecture is defined, Timeloop-model requires a mapping of the problem onto the hardware to perform its evaluations. Listing 4 shows an untiled implementation of 2D-Conv. We use Timeloop to evaluate a mapping of this problem onto a 4 memory level GEMMINI architecture. Listing 5 describes how a Timeloop mapping is specified. The *factors* represent the loop sizes, *permutation* specifies the loop ordering for the memory level and the *type* key specifies whether the loop is mapped sequentially or parallelised in hardware. The yaml mapping description in Listing 5 generates a tiled, reordered loop nest as shown in Listing 6.

As an example, the Output's dimensions of the problem are N,P,Q,M. Output data is tiled at the AccumulationBuffer as shown in Listing 6. This means a block of output tensor data of size 6 (register output tile size) * 2 (M) * 4 (N) * 8 (Q) = 384 is retrieved from DRAM and stored in the AccumulationBuffer level for computation. A block of this size is retrieved from the DRAM 3136 times (we multiply the loop sizes for dimensions N,P,Q,M at levels before the AccumulationBuffer to obtain this value) during the entire computation to process all the ouput data. The correctness of this mapping can be verified by ensuring that the tiled implementation traverses all the dimensions of the tensors correctly.

```
DRAM [ Weights:31104 (31104) Inputs:7750656 (7750656) Outputs:1204224 (1204224) ]
---------------------------------------------------------------------------------
| for M in [0:24)
|   for Q in [0:56)
|     for R in [0:3)
|       for P in [0:56)
|         for N in [0:16)
|           for S in [0:3)
|             for C in [0:144)
|                    Output[N][P][Q][M] +=
|                                    Weight[C][R][S][M] *
|                                    Input[C][N][S+Q][P+R]
```

Listing 4: Original, untiled 2D-Conv problem

C. Mapspace Encoders

GPTune can sample numerical mapspace optimization parameters (e.g. tile sizes) as integers or real numbers. Categorical mapspace parameters (e.g. loop ordering) can be sampled as categorical variables or as numerical variables and then appropriately encoded.

```
mapping:
- factors: N=1 M=4 S=1 P=28 Q=1 R=1 C=4
  permutation: CQRSPMN
  target: DRAM
  type: temporal
- factors: N=4 M=1 S=1 P=1 Q=7 R=1 C=9
  permutation: QCMPNSR
  target: Scratchpad
  type: temporal
- factors: N=4 M=2 S=1 P=1 Q=8 R=1 C=1
  permutation: CMPNRQS
  target: AccumulationBuffer
  type: temporal
- factors: N=1 M=3 S=3 P=2 Q=1 R=3 C=4
  permutation: NCPMSRQ
  target: Registers
  type: temporal
```

Listing 5: A valid 2D-Conv mapping description for Timeloop-model. Note that the loop permutation is specified from innermost to outermost loop

```
DRAM [ Weights:31104 (31104) Inputs:7750656 (7750656) Outputs:1204224 (1204224) ]
---------------------------------------------------------------------------------
| for M in [0:4)
|   for P in [0:28)
|     for C in [0:4)

Scratchpad [ Weights:1944 (1944) Inputs:133632 (133632) ]
---------------------------------------------------------
|       for N in [0:4)
|         for C in [0:9)
|           for Q in [0:7)

AccumulationBuffer [ Outputs:384 (384) ]
----------------------------------------
|             for Q in [0:8)
|               for N in [0:4)
|                 for M in [0:2)

Registers [ Weights:108 (108) Inputs:48 (48) Outputs:6 (6) ]
------------------------------------------------------------
|                   for R in [0:3)
|                     for S in [0:3)
|                       for M in [0:3)
|                         for P in [0:2)
|                           for C in [0:4)
```

Listing 6: The mapping described in Listing 5

To obtain valid mappings, the tilings and orderings obtained after the encoding must satisfy hardware constraints. To ensure tile sizes are less than the loop dimensions, we sample tiling parameters from the $(0, 1]$ space, then scale and round the tile sizes to factors of the loop dimension. Rounding is done to ensure perfect factorization. To meet the hardware memory constraints for higher dimensional problems, more complicated inequalities have to be satisfied by the tiles.

As an example, let's take a 2D-Conv defined by b batches, c input channels, k output channels, and filters of size $r \times s$ and outputs of size $w \times h$. To tile the problem, the tile sizes $t_b, t_c, ...$ must satisfy constraints that ensure the dimensions fit

in the level of memory. (2) is the constraint to be satisfied for a scratchpad level (only inputs and weights can be stored) of size S. (3) is the constraint to be satisfied for the accumulator level (only outputs can be stored here) of size A.

$$t_c t_k t_r t_s + t_b t_c (t_w + t_r)(t_h + t_s) \leq S \tag{2}$$

$$t_n t_m t_p t_q \leq A \tag{3}$$

Rejection sampling could be used in this scenario instead of constraining the space. In our initial rejection sampling approach, we set the objective value to 1e+30 when an invalid mapping was found. This approach did not work well because the large value could overwhelm the surrogate model computations. We also noticed that as the problem dimension increased, rejections also increased, driving up the number of GPTune iterations required.

So, encoding every sampled point to be a valid mapping in the architecture improves GPTune's performance. In our new encoding scheme, we optimize over the aspect ratio of the tiles. Instead of sampling tiling parameters from the $(0, 1]$ space, we sample the tile aspect ratios $a_b, a_c, ...$ from $(0, 1]$. We believe this approach improves the generalizability of the model over problem sizes, as communication-optimal tiles tend to retain their aspect ratios over large problem sizes. So, instead of directly optimizing over $t_b, t_c, ...$, we optimize over the scaled aspect ratios. (2) and (3) now become

$$t_{b,c,...} \approx \alpha a_{b,c,...} \tag{4}$$

$$\alpha^4 \left[a_c a_k a_r a_s + a_b a_c (a_w + a_r)(a_h + a_s) \right] \leq S \tag{5}$$

$$\alpha^4 \left[a_n a_p a_m a_q \right] \leq A \tag{6}$$

The resulting tile sizes are rounded down to the nearest valid factor of the loop dimension to ensure perfect factorization of the loop.

Loop ordering parameters can be sampled as categoricals or numericals. The approach we use to sample loop orders as categoricals is to enumerate the possible permutations of a nested loop and let GPTune sample a point from this categorical space. This method seems to work well for low dimensional problems like matrix multiplication, but its performance suffers for the high dimensional 2D-Conv problem.

When sampling numerical values for loop ordering, a score-based approach is used to determine the ordering in the mapping. We sample ordering parameters from the $[0, 1)$ space. The sampled parameters are ordered according to the NAAS [15] "importance-based" encoding method. The importance value of each loop is sampled by GPTune as the ordering parameter. The sampled values are sorted in decreasing value and the ordering is determined by associating the loop to its sampled value in the sorted list.

V. Evaluation

Despite our approach being aimed at costly performance models, we use Timeloop to evaluate the mappings selected by GPTune because of its random search algorithm, Timeloop-mapper. Using a cycle-accurate simulator to run random search would be infeasible because it requires 1000s of evaluations to converge.

We optimize two tensor computations — matrix multiplication (matmul) and 2D-convolution (2D-Conv) on a hardware architecture based on GEMMINI with 4 memory levels. We selected these 2 algorithms because 2D-Conv has high dimensionality and matmul is relatively low dimensional. Loop tiling and ordering is done for each memory level.

To perform Bayesian optimization, GPTune's MLA and TLA_I algorithms are used. We define GPTune's problem parameters as follows:

- Task Parameter Input Space : Sizes of each of the problem dimensions as well as the storage capacities of the memory levels. In general, the input space should include problem dimensions that affect the mapspace of a problem. The aim of our transfer learning approach is to enable generalization over the sizes of the memory levels and problem dimensions for the GEMMINI architecture. This would enable GPTune to learn the mapspaces for unseen problem and memory sizes with fewer iterations.
- Tuning Parameter Space : The tile size and loop order parameters. As an example, matmul has 3 nested for loops. Each loop has to be tiled at every memory level and there are 4 memory levels in total. Computing the tile size at the last level is trivial because we need a perfect factorization of the loop, so the $lastTile = \frac{LoopSize}{productOfLoopTilesAtOtherLevels}$ so in total there are $3 \times 3 = 9$ tiling parameters for the 4 memory levels and 3 loops at each memory level. There are also $3 \times 4 = 12$ loop ordering parameters for the 4 memory levels and 3 loops at each memory level. In total there are 21 optimization parameters to tune. The number of optimization parameters varies according to the number of loops and memory levels in the problem.
- Output Space : Real value space from $-inf$ to $+inf$ representing the energy or runtime evaluations of the sample point chosen by GPTune. For our experiments, we use the energy metric (measured in $\mathrm{pJ/compute}$) evaluated for mappings by Timeloop.

For all experiments, we average over three independent runs to reduce statistical variance.

A. Convergence

Moving forward, we will use the term *iteration* to describe the evaluation of one point chosen by GPTune. Figures 5 and 6 show the energy value of the best mapping found so far at each iteration, comparing Timeloop and GPTune (we run 100 iterations for GPTune).

For both matmul and 2D-Conv, GPTune converges to a good mapping much faster than for random search. As the

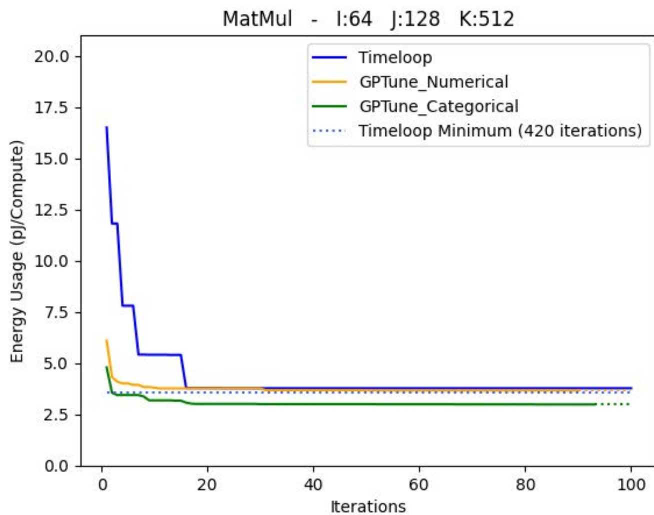


Fig. 5. Energy (lower is better) attained by Timeloop's random-pruned search mapper and GPTune for matrix multiplication. 100 iterations of GPTune are run. The blue dotted line indicates the energy value for the best mapping found by Timeloop. Timeloop attains this value at 420 iterations.

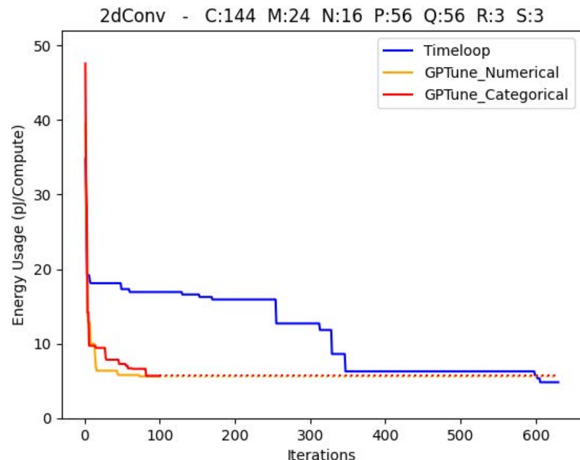


Fig. 6. Energy (lower is better) attained by Timeloop's random-pruned search mapper and GPTune for 2D-convolution. 100 iterations of GPTune are run.

dimensionality of matmul is relatively small, random search performs reasonably well, but for 2D-Conv, random search performs significantly worse than GPTune. For GPTune, the plots show optimization results for both the score based numerical and categorical encodings for loop ordering. For matmul, categorical encoding seems to work better for tuning, whereas for 2D-Conv the numerical encoding works better. This is likely because of the number of possible loop orderings for the problems. Matmul (3 nested loops) has $(3!)^4 = 1296$ choices for loop orderings over the four levels of the memory hierarchy, while the 2D-Conv (7 nested loops) results in $(7!)^4 \approx 6e14$ choices.

For matmul in Figure 5, GPTune_Categorical converges in less than 30 runs to 2.98 pJ/compute. Timeloop is run a total of 4000 iterations. The best value found by Timeloop is 3.56 pJ/compute, a value 16% worse than GPTune's result. After ≈ 420 iterations, the minimum performance value found by Timeloop remains the same. No better mapping is found by Timeloop after 420 iterations.

For 2D-Convs in Figure 6, GPTune converges in (on average) 50 iterations to a minimum of 5.26 pJ/compute, a value that it took an average of 627 iterations for Timeloop-mapper to beat. Timeloop is run a total of 4000 iterations. The best performance value obtained by Timeloop was 4.32 pJ/compute, only 17% better than GPTune's. The minimum performance value found by Timeloop-mapper remains the same from ≈ 627 to 4000 iterations (no better mapping is found in this interval).

Future work for convergence experiments include (1) determining if convergence could be improved by using crowd-tuning methods using a GPTune DB, (2) varying the size of the DB to determine how many prior evaluations would lead to improvements in optimization, (3) analysing the mapspace exploration done by GPTune to determine how and why it performs well, (4) reducing GPTune's runtime using parallelism, (5) exploring new encoding schemes that lead to better performance, (6) benchmarking GPTune's optimization against other algorithms like GAMMA and Mind Mappings, (7) using multi-objective tuning (e.g optimizing both runtime and energy), and (8) incorporating other loop optimizations like spatial mapping, loop unrolling etc.

B. *Transfer Learning*

In hardware-software codesign settings, generalizable learning methods are preferred because they enable improved optimization of unseen problem configurations. Random search methods and feedback-driven algorithms like GAMMA or Mind Mappings are not able to transfer learn effectively. For every new problem to be optimized, they must be rerun and modeled from scratch. Differentiable surrogate functions are constructed by Mind Mappings. However, when these models are applied for transfer learning on unseen hardware architectures, they result in a significant decrease in performance, by one or two orders of magnitude compared to Timeloop-mapper and GAMMA [9].

Surrogate models built by GPTune using evaluations from different problem and hardware specifications can be used for transfer learning. We fixed the dimensions of a matmul problem and changed the target architecture specifications to obtain a collection of different function evaluations for the GPTune DB. Figure 7 shows transfer learning for a matrix multiplication problem. The surrogate model used was trained using 100 total evaluations from 4 different hardware architectures for the same matmul problem dimensions. This model was further fine-tuned for the target architecture with 20 evaluations (in the search phase) from the target architecture (this architecture was not present in the DB).

TLA_I in Figure 7 converges to a mapping of energy 3.81pJ/compute (on par with an uninitialized GPTune) in 10 iterations (roughly half that of uninitialized GPTune). This

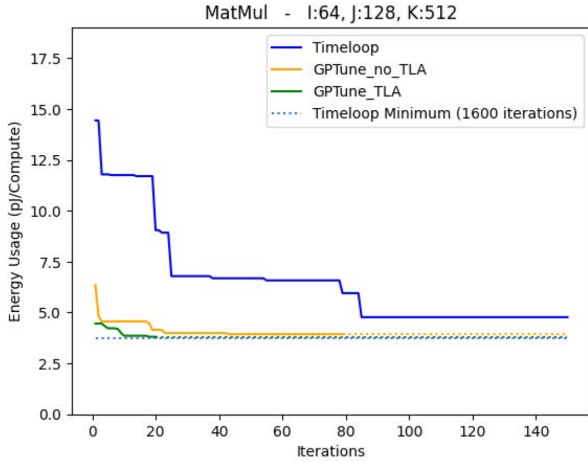
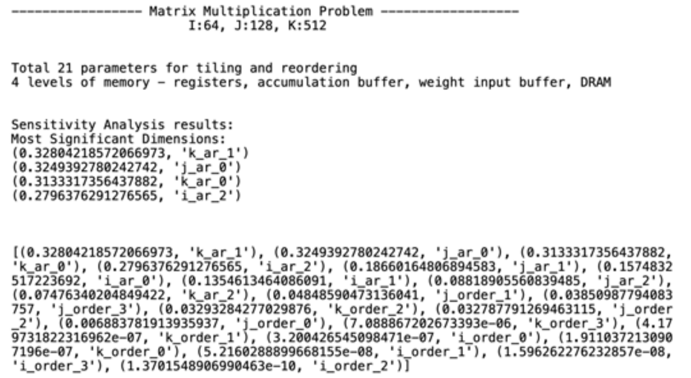


Fig. 7. Transfer learning to a new (not in training set) hardware configuration for matrix multiplication, compared to GPTune with no prior knowledge and Timeloop-mapper. The blue dotted line indicates the energy value for the best mapping found by Timeloop. Timeloop attains this value at 1600 iterations.



```
---------------- Matrix Multiplication Problem -----------------
                    I:64, J:128, K:512

Total 21 parameters for tiling and reordering
4 levels of memory - registers, accumulation buffer, weight input buffer, DRAM

Sensitivity Analysis results:
Most Significant Dimensions:
(0.32804218572066973, 'k_ar_1')
(0.3249392780242742, 'j_ar_0')
(0.3133317356437882, 'k_ar_0')
(0.2796376291276565, 'i_ar_2')


[(0.32804218572066973, 'k_ar_1'), (0.3249392780242742, 'j_ar_0'), (0.3133317356437882,
'k_ar_0'), (0.2796376291276565, 'i_ar_2'), (0.18660164806894583, 'j_ar_1'), (0.1574832
517223692, 'i_ar_0'), (0.1354613464086091, 'i_ar_1'), (0.08818905560839485, 'j_ar_2'),
(0.07476340204849422, 'k_ar_2'), (0.04848590473136041, 'j_order_1'), (0.03850987794083
757, 'j_order_3'), (0.03293284277029876, 'k_order_2'), (0.032787791269463115, 'j_order
_2'), (0.006883781913935937, 'j_order_0'), (7.088867202673393e-06, 'k_order_3'), (4.17
9731822316962e-07, 'k_order_1'), (3.200426545098471e-07, 'i_order_0'), (1.911037213090
7196e-07, 'k_order_0'), (5.2160288899668155e-08, 'i_order_1'), (1.596262276232857e-08,
'i_order_3'), (1.3701548906990463e-10, 'i_order_2')]
```

Fig. 8. Matrix multiplication sensitivity analysis with 1000 function evaluations

performance value requires Timeloop an average of 1600 iterations to beat.

Future work for transfer learning experiments include (1) problem dimension transfer learning, (2) finding mapspace encodings that work best for TLA algorithms, (3) multi-fidelity TLA, i.e. using cheap analytical model results as a guide to enhance tuning using expensive simulators, and (4) determining if TLA_II can be used as a one-shot optimization method to obtain a reasonable mapping for an unseen architecture.

C. Sensitivity Analysis

The surrogate models generated by GPTune can be used to determine the most important optimization parameters for the mapspace. GPTune offers a sensitivity analysis framework that analyzes how different tuning parameters affect the performance output results. GPTune currently applies Sobol [18] Analysis to measure the contribution of each of the inputs to the variance of the output. Sobol is a global sensitivity analysis method that assesses parameter sensitivity. GPTune also internally invokes SALib [10] to compute Sobol indices from the trained surrogate model.

We use a large number of cheap, function evaluations from Timeloop to build the surrogate model. Then a variance based mathematical analysis is conducted to compute the sensitivity values. We are primarily interested in the total effect index ST_i which measures the *total* contribution (including interactions between other variables) of a parameter to the total variance.

In Figure 8, we see the results after running sensitivity analysis on 1000 evaluations of a $64 \times 128 \times 512$ size matrix multiplication problem. There are 4 memory levels and the memory levels are 0 indexed starting from the smallest level of memory (register) to the biggest level of memory (DRAM) at index 3. So the indices are (0) registers, (1) accumulator, (2) scratchpad, (3) DRAM. Each tiling parameter is formatted as *loopName*_ar_*memoryLevel*. For example, *k_ar_1* is the tiling parameter for the k loop at the 1st memory level (accumulator). The numerical loop ordering encoding is used and each ordering parameter is formatted as *loopName*_order_*memoryLevel*. For example, *n_order_2* represents the ordering parameter for loop n at the 2nd memory level (scratchpad).

We see that the most important axes are the tilings of k at the register and accumulator levels and the tiling of j at the register level. The surrogate model is several orders more sensitive to tiling parameters as compared to ordering parameters. This is an interesting result because it aligns with previous work showing that loop tilings are the most important mapspace parameter for optimization.

These results are significant because it shows that most parameters in the space have a marginal impact on the performance. This reduces the dimensionality of the problem and could lead to better results. GPTune's sensitivity analysis tool enables us to extract the most imporant mapspace parameters and fine-tune them further to potentially discover more efficient mappings.

Future work for sensitivity analyisis include (1) running sensitivity analysis for other tensor computations like 2D-Conv to determine if low dimensionality is a general property for mapspaces, (2) setting up an automated dimension reduction framework and further fine tuning the important mapspace parameters, (3) running sensitivity analysis with a cycle-accurate simulator for more precise sensitivity results, and (4) multi-objective sensitivity analysis to determine which mapspace optimization parameters have an impact on different objective functions.

VI. Related Work

There has been a significant amount of work done in the mapspace exploration domain. CoSA [11] is a one-shot optimizer that takes into account hardware and problem specifications to formulate the mapspace into a mixed-integer program (MIP) and solves it to find an efficient mapping. Similarly, polyhedral compilation methods like Tensor Comprehensions [19] and Pluto [7] model the mapspace as integer linear programs (ILP) and minimize the communication cost. Mind Mappings [9] uses gradient-based methods. Most other

mapspace exploration techniques use machine learning to navigate the mapping search space. GAMMA [12] uses analytical models for mapping evaluation and evolutionary search methods for space exploration. FlexTensor [21] uses a heuristic method based on simulated annealing to find mappings and reinforcement learning to guide the search from the chosen mappings. AutoTVM [4] uses a statistical cost model based on Gradient Boosted Trees (GBTs) and an ML based exploration model with a rank loss function. Adams et. al [3] presents a method to optimize Halide [2] using tree search for mapping space generation and a variant of beam search to navigate the mapspace.

VII. Conclusion

In this thesis, we demonstrate the feasibility of using Bayesian optimization techniques for mapspace optimization using very few (less than 100) samples. Our work will prove beneficial to optimize mapspaces where the cost of obtaining performance data is expensive. Our method facilitates transfer learning. Transfer learning enables faster convergence to high performance mappings and uses knowledge from prior evaluations to optimize searching an unseen mapspace. It also enables efficient tailoring of architectures to execute specific problems efficiently. The results of our sensitivity analysis experiments are optimistic; they signify that the dimensionality of the mapspace can be reduced, further improving Bayesian optimization techniques. We list future work and potential research paths in the sub-sections under Section V. Although there are more enhancements to be made, we are excited to present these promising Bayesian optimization results. With further refinement of our method, we anticipate achieving even better outcomes and significantly accelerating tensor computations.

Acknowledgements

First and foremost I would like to thank Professor James Demmel and Grace Dinh for their invaluable guidance, unwavering support, and encouragement throughout my Master's program. Their expertise in the field and insightful feedback have been instrumental in shaping my academic growth. I will always be grateful for their mentorship and feel fortunate to have the opportunity to learn from them.

Thanks to Professor Sophia Shao for graciously agreeing to review my thesis and providing thoughtful suggestions.

I would also like to extend my heartfelt appreciation to my family for their unconditional love, support, and motivation. Their sacrifices and encouragement have been the driving force behind my achievements, and I could not have accomplished this milestone without their trust and belief in me.

Lastly, I would like to thank the University of California, Berkeley and acknowledge all the individuals who have supported me during my academic journey, including fellow Beboppers, colleagues, and friends. They have made my journey a happy and memorable one.

References

[1] "An architecture-level energy estimation methodology for accelerator designs." [Online]. Available: https://accelergy.mit.edu/

[2] "Halide." [Online]. Available: https://www.csail.mit.edu/research/halide

[3] "learning to optimize halide with tree search and random programs." [Online]. Available: https://halide-lang.org/papers/autoscheduler2019.html

[4] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," Jan 2019. [Online]. Available: https://arxiv.org/abs/1805.08166

[5] Y. Cho, J. W. Demmel, G. Dinh, X. S. Li, Y. Liu, H. Luo, O. Marques, and W. M. Sid-Lakhdar, "GPTune user guide," 2022. [Online]. Available: https://github.com/gptune/GPTune/tree/master/Doc

[6] Y. Cho, J. W. Demmel, J. King, X. S. Li, Y. Liu, and H. Luo, "Harnessing the Crowd for Autotuning High-Performance Computing Applications," in *The 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS23)*. IEEE, 2022, pp. 1–12.

[7] J. D. Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori, and et al., "Pluto: Enabling massively parallel computation in dram via lookup tables," Oct 2022. [Online]. Available: https://arxiv.org/abs/2104.07699

[8] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[9] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, apr 2021. [Online]. Available: https://doi.org/10.1145%2F3445814.3446762

[10] J. Herman and W. Usher, "SALib: An open-source Python library for sensitivity analysis," *The Journal of Open Source Software*, vol. 2, no. 9, jan 2017. [Online]. Available: https://doi.org/10.21105/joss.00097

[11] 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," May 2021. [Online]. Available: https://arxiv-export3.library.cornell.edu/abs/2105.01898?context=cs.LG

[12] 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.

[13] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[14] 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, pp. 754–768. [Online]. Available: https://doi.org/10.1145/3352460.3358252

[15] Y. Lin, M. Yang, and S. Han, "Naas: Neural accelerator architecture search," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE Press, 2021, pp. 1051–1056. [Online]. Available: https://doi.org/10.1109/DAC18074.2021.9586250

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

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

[18] I. M. Sobol, “Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates,” *Mathematics and computers in simulation*, pp. 271–280, 2001.

[19] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions,” Jun 2018. [Online]. Available: https://arxiv.org/abs/1802.04730

[20] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, “Ansor: Generating High-Performance Tensor Programs for Deep Learning,” arXiv, Tech. Rep., arXiv:2006.06762 [cs, stat] type: article. [Online]. Available: http://arxiv.org/abs/2006.06762

[21] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, “Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system,” in *ASPLOS ’20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 859–873. [Online]. Available: https://doi.org/10.1145/3373376.3378508

[22] X. Zhu, Y. Liu, P. Ghysels, D. Bindel, and X. S. Li, *GPTuneBand: Multi-task and Multi-fidelity Autotuning for Large-scale High Performance Computing Applications*, pp. 1–13. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611977141.1