PEAK: Generating High-Performance Schedules in MLIR

Amir Mohammad Tavakkoli*, Sameeran Joshi*, Shreya Singh,
Yufan Xu, P. Sadayappan, and Mary Hall

University of Utah, Salt Lake City, UT 84112, USA
{tavak, sameeran, saday, hall}@cs.utah.edu, {shreya.singh, yf.xu}@utah.edu

Abstract. Machine learning frameworks rely on vendor libraries or autotuning frameworks for high-performance implementations of key operators like matrix multiplication and convolution. The transform dialect has recently been developed in the MLIR framework to facilitate the composition of transformations to implement optimized schedules for tensor computations. However, its interface to users is complex and requires an understanding of the underlying compiler transformations. In this paper, we describe a higher level scheduling language PEAK that is built on top of the MLIR transform dialect to ease the process of developing optimized schedules for deep learning operators on GPUs. PEAK expresses a simplified scheduling language in the MLIR/IREE compiler; it exploits domain-specific properties about data reuse in tensor contractions to determine thread mapping strategies and stage data through the GPU memory hierarchy. PEAK integrates an autotuner to explore implementations of high-performance code with schedules based on the MLIR Transform Dialect. PEAK enables a significant reduction of effort to construct high-performance GPU code using MLIR. A comparison with the state-of-the-art TVM/Ansor autotuning compiler framework shows higher performance for matrix-vector products, nearly comparable performance for matrix-matrix multiplication, but lower performance for convolutions. The paper presents insights into the limitations of the MLIR/IREE infrastructure that currently impact the performance achievable with PEAK.

Keywords: Code generator · Scheduling Language · Autotuning · GPU · MLIR/IREE

1 Introduction

The convolution operator is a core component in Convolutional Neural Networks (CNNs), and matrix-matrix multiplication is a fundamental operation in transformer networks used in Large Language Models. Optimizing these two operators for GPUs is a very challenging task. Vendor libraries like cuDNN [10] and cuBLAS [18] are engineered by GPU experts to be well-matched to low-level architectural features and achieve performance close to machine peak for sufficiently large problem sizes. However, for the actual sizes/shapes of convolution operators found in practically used CNN image processing pipelines like ResNet [14] and Yolo [20], the achieved performance is often lower than that achieved by state-of-the-art autotuning frameworks like AutoTVM [9] and Ansor [26]. Further, the manual implementation of high-performance vendor libraries or specific kernel design solutions [16, 24] requires a deep understanding of low-level features of hardware and very significant time investments from expert engineers. Therefore, there is a significant interest in developing frameworks to ease the development of optimized implementations of key machine learning operators for GPUs.

* These authors contributed equally to this work

A scheduling language enables performance experts to express transformation sequences to be applied by the compiler. Schedules separate transformations from computation and enable modular composition of these components. An example of a framework providing a scheduling language is TVM [8], which is a state-of-the-art deep-learning compiler that supports autotuning in AutoTVM [9] and Ansor [26]. Automatic code generators which feature scheduling languages have demonstrated effectiveness in achieving high performance as well as performance portability across multiple hardware platforms [6–8, 11, 12, 15, 19, 22, 25, 27].

The MLIR compiler infrastructure has been developed to facilitate layered implementation of compiler optimization passes [17]. It uses multiple levels of IR (Internal Representation) appropriate for lowering from high-level tensor expressions to multi-level tiled parallel code for different hardware platforms. A recent development is the creation of the `transform` dialect [3] in MLIR to enable effective composition of transformations such as tiling, unrolling, vectorization, etc. However, its interface to users is complex and requires an understanding of the underlying compiler transformations. In this paper, we develop a higher level scheduling language called PEAK implemented on top of the MLIR `transform` dialect, intended to provide greater flexibility and ease of development of optimization strategies in MLIR. We conduct experiments comparing with the state-of-the-art TVM/Ansor autotuning compiler and present observations on some of the current challenges to achieving comparable performance in the MLIR/IREE [1] ecosystem.

The key contributions of this work are:

- We introduce PEAK, a novel high-level domain-specific scheduling language, enabling source-to-source mapping to the `transform` dialect to ease generating schedules and hide low-level MLIR-specific details from developers.
- We integrate and demonstrate the effectiveness of autotuning in the MLIR/IREE ecosystem, facilitating the automatic discovery of high-performance code configurations for GPUs.
- We present an experimental evaluation against the state-of-the-art TVM/Ansor autotuning compiler for four deep learning operators: GEMM, Transposed GEMM, matrix-vector multiplication, 1D, and 2D convolutions.

The rest of the paper is organized as follows. Section 2 provides some background and motivates this work. Section 3 presents the overall design of PEAK and presents implementation details. Section 4 describes the optimizations that enable improved performance over code generated by the current MILR/IREE pipeline. Experimental results and evaluations are presented in Section 5. Section 6 concludes the paper.

2 Background and Motivation

MLIR is a cross-domain compiler framework specifically designed for building reusable and extensible compiler implementations. MLIR provides a declarative system for defining dialects to support high-level abstractions and domain-specific constructs and produce modular libraries. MLIR supports various backend architectures and progressive lowering of the Intermediate Representation (IR), thus supporting a variety of application domains and facilitating performance portability.

The `transform` dialect is a scheduling language inside MLIR that is divided into two parts: the payload IR and the transform IR. The payload IR represents the input computations, while the transform IR encapsulates the transformations to be applied. Through a dialect extension mechanism in the `transform` dialect, researchers and users can introduce new operations targeting specific dialects, such as `gpu`, `vector`, and `linalg`.

However, working with the `transform` dialect directly can be challenging due to its low-level nature, which requires a deep understanding of MLIR internals. Also, researchers and users are required to have specific knowledge of pre-conditions and post-conditions before creating and using transformations to avoid breaking the interfaces over `transform` dialect operations. We demonstrate matrix multiplication (Matmul) as an example to illustrate how users write a schedule in the `transform` dialect and how it generates GPU-targeted code. In the MLIR code shown in Listing 1.1, we express a Matmul operation for 1024x1024 input and output sizes. In this code, we utilize the `linalg.matmul` operator from the `linalg` dialect, specifically designed for matrix multiplication.

```
!A_t = tensor<1024x1024xf32>
!B_t = tensor<1024x1024xf32>
!C_t = tensor<1024x1024xf32>

func.func @linalg_matmul(
    %A : !A_t, %B : !B_t, %C : !C_t) -> !C_t {
  %0 = linalg.matmul ins(%A, %B : !A_t, !B_t)
                     outs(%C : !C_t) -> !C_t
  return %0 : !C_t
}
```

Listing 1.1: Matmul in MLIR

As shown on the left side of Figure 1, the `transform` dialect allows us to express a sequence of transformations for specific optimizations. However, manually writing schedules using the `transform` dialect is not a simple task, and contains massive low-level details. Each colored code block shows a snippet of code that does independent computation. The code block labeled `5)Vectorization` generates vector instructions, requiring information on types, optimizations in the underlying IREE end-to-end compiler, operations in other dialects and other low-level implementation details. The code tries to look for MILR construct `func.func` in the payload and apply predefined patterns on the operator handle; next, `transform.structured.vectorize` performs vectorization and post-processing using predefined patterns. To complete the end-to-end matrix multiplication, it is necessary to carry out these steps consecutively, as depicted in blocks numbered from 1 to 8.

The `transform` dialect is useful for composing commands and is valuable for crafting scheduling languages. However, from the users' and developers' standpoint, writing higher-level schedules with an abstraction layer placed on top of the `transform` dialect is desirable. Although a thorough description of the constructs in the `transform` dialect is beyond the scope of this work, but we will call attention to a common construct `transform.apply_patterns`. It ensures the code matches a pattern before performing downstream transformations. Sometimes there are several optimizations associated with a specific transformation. As demonstrated above, 30 low-level operations – and over 50 lines – in the original `transform` dialect on the left hand side of Figure 1 correspond to just 10 lines of high-level constructs in PEAK on the right hand side.

3 PEAK

PEAK serves two roles in achieving high-performance code using MLIR: it aims to simplify writing schedules, and its high-level interface provides a way to integrate the autotuner into MLIR. Figure 2 demonstrates the overall design of PEAK as a domain-specific language for scheduling transformations for tensor contraction operators with a minimal set of scheduling primitives, which is layered above the `transform` dialect.

Fig. 1: Mapping of `transform` dialect to PEAK for a GEMM schedule

Users express their optimization strategy in PEAK by constructing schedules using the primitives. The details about PEAK's scheduling primitives are presented in subsection 3.1. The autotuner is integrated into PEAK to identify the best optimization parameter values related to tile size at various levels of parallelism and memory hierarchy for the target GPU. A huge combination of choices exists for multi-level tile sizes, determining the size/shape of thread blocks, work distribution among warps in a thread block, and the size/shape of register tiles at each thread. Next, PEAK lowers the scheduling primitives, tile size values are selected and by the autotuner, and these are instantiated to generate the corresponding `transform` dialect. Finally, the original computation IR and the `transform` dialect IR generated by PEAK are passed to the MLIR/IREE compiler to generate a binary for the target architecture.

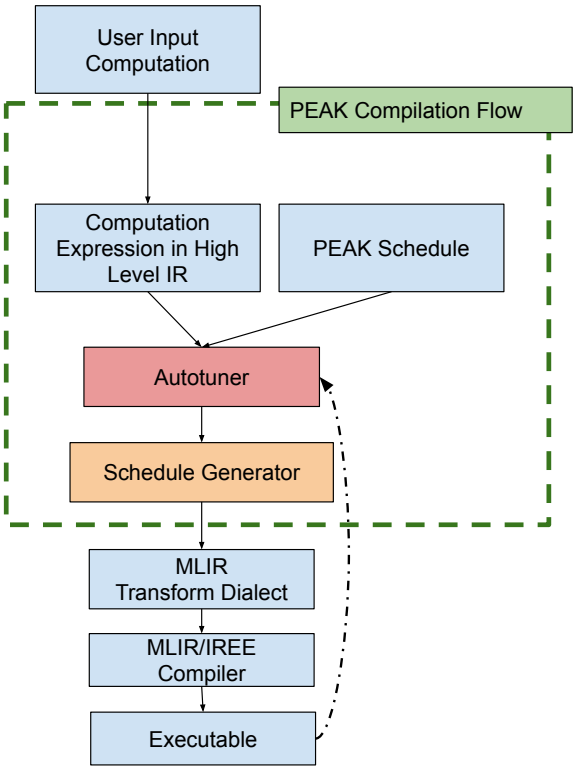


Fig. 2: Overview of PEAK in MLIR

3.1 Scheduling Primitives

PEAK encapsulates complex optimization strategies as scheduling primitives in a minimal set to avoid semantic ambiguity. We have chosen these specific commands because they serve as fundamental building blocks. By combining them, users can create complex scheduling scripts in PEAK.

This level of abstraction aligns well with our goals and ensures that users only need to know the essential parameters, keeping the underlying complexity hidden. Our inspiration for PEAK comes from the challenges encountered when working with the `transform` dialect. We listed the essential scheduling primitives in the PEAK DSL with a brief explanation in Table 1.

Table 1: List of PEAK Scheduling Commands and Descriptions

No.	Command	Description
1	findOp	Find a specific operation and return Op handle
2	tile	Divide a computation into smaller blocks
3	fuse	Combine multiple operations into a single entity
4	tileForGrids	Divide computations for specific grid structures
5	parallelizeForBlocks	Parallelism at the thread block level in GPU
6	parallelizeForThreads	Parallelism at the thread level in GPU
7	bufferize	Materialize tensor data types into memref
8	promote	Move data in memory hierarchy
9	unroll	Unroll loops for given unrolling factor
10	vectorize	Generate vector instructions for computation

Case study of Scheduling Primitives We use Matmul as a case study to explain the succinctness of scheduling primitives in PEAK. The right-hand side in Figure 1 shows a matrix multiplication scheduling example written in PEAK according to the default implementation in the original MLIR `transform` dialect.

The full PEAK code for Matmul is shown on the right-hand side in Figure 1. In the Matmul operator, I and J are parallel dimensions, and K is the reduction dimension.

$$C[i,j] = \sum_{k} A[i,k] \times B[k,j] \tag{1}$$

In the block labeled `1) Get a handle to matmul` (hereafter referred to as Block 1), the file name of the generated `transform` dialect IR is specified, and the Operator handle of the Matmul operation is obtained. In Block 2, the computation is distributed among thread blocks at the grid level of the GPU. The computation's parallel dimension I, J has been tiled with grid level tile size BX and BY and mapped over thread blocks. Block 3 specifies the tiling on the reduction dimension K with tile size TK and marks the first and second operands for *memory promotion*. For the GPU architecture, memory promotion means moving data from the slower GPU global memory to the GPU shared memory with faster access time to reuse data across threads at the thread block level. Block 4 specifies the mapping of the remaining computation over threads with sizes TX and TY in the view of one single thread block. Vectorization is enabled in Block 5, where the `vector` dialect instruction is emitted to boost the performance of the generated code. Vectorization may apply a number of optimizations, such as unrolling static-shape vector operations, warp-synchronized operations, and efficient memory latency hiding when lowering to other dialects.

Until this point, the input data of the operator is represented as a `tensor` data type in MLIR whose lower-level memory placement has not been determined. The `tensor` data type provides a convenient high-level data buffer view for the user and developer and avoids complicated manual

memory management. The `bufferize` in Block 6 handles memory allocation and deallocation of operands and output of the operator. The memory promotion in Block 3 is managed explicitly in this step, and the data is placed into the GPU-shared memory. Block 7 shows the lowering steps necessary for generating parallel constructs in MLIR. The last block in the code finalizes the code generation at the PEAK level.

Comparing Scheduling Primitives with TVM/Ansor We compare PEAK's Matmul schedule with that of TVM/Ansor [2]. Ansor automatically generates code templates and forms a search space for the computation based on a set of predefined rules. Once the template is constructed, exploring and exploiting the search space in Ansor relies on ML techniques to find the best schedule in the large search space.

There are several differences in the schedules of PEAK and TVM/Ansor. PEAK differs from TVM/Ansor in its approach to tiling and loop reordering. An example of a shortened generated schedule by Ansor is shown in Listing 1.2. First, all parallel dimensions of Matmul are tiled using the `split` primitive at line 5, and then the loops are reordered with `reorder` primitive at line 8. That is, TVM/Ansor stripmines each dimension for blocks, threads, and shared memory and then permuting the tile controlling loops to the outer loops to distribute them among blocks and threads. Instead, we have taken a stepwise approach by first tiling for blocks, then shared memory, and subsequently for threads. In both cases, the resulting loop order is the same. As shown in Fig. 1, in PEAK at code blocks 2, 3, and 4, this is achieved by using a primitive for the tiling of blocks and threads (`tileForGrids`) and another primitive for tiling the reduction dimension (`tile`). The tiling value zero in block 3 indicates that the I and J loops in the computation are not tiled by setting the tile value to zero, and only the K dimension is tiled with the given *TK* value.

```
1  # Matrix dimensions i, j, k
2  matmul_i, matmul_j, matmul_k = tuple(matmul.op.axis) + tuple(matmul.op.
       reduce_axis)
3  out_i, out_j = tuple(out.op.axis) + tuple(out.op.reduce_axis)
4  # Tiling i dimension by factor of 2
5  matmul_i_o_i, matmul_i_i = s[matmul].split(matmul_i, factor=2)
6  ...
7  # Reorder the loops
8  s[out].reorder(out_i_o_o_o, out_j_o_o_o, out_i_o_o_i, out_j_o_o_i,
       out_i_o_i, out_j_o_i, out_i_i, out_j_i)
9  # Move data pointed by B to shared memory.
10 B_shared = s.cache_read(B, "shared", [matmul])
11 ...
12 # Fuse loops into single loop
13 out_i_o_o_o_j_o_o_o_fused = s[out].fuse(out_i_o_o_o, out_j_o_o_o)
14 # Bind fused loop onto threads
15 s[out].bind(out_i_o_o_o_j_o_o_o_fused, te.thread_axis("blockIdx.x"))
16 ...
17 # Unroll loop by 1024
18 s[matmul].pragma(matmul_i_o_o_o_o, "auto_unroll_max_step", 1024)
19 s[matmul].pragma(matmul_i_o_o_o_o, "unroll_explicit", True)
```

Listing 1.2: TVM scheduling snippet for matmul (Shortened)

Vectorization in PEAK can have the same effect as loop unrolling in Ansor. In lines 18 and 19 the Ansor schedule in Listing 1.2, `auto_unroll_max_step` and `unroll_explicit` pragmas indicate the loop unrolling. Instead, PEAK uses the `vectorize` primitive, as shown in Block 5. Third, Ansor stages data in shared memory by using the `cache_read` primitive shown at line 10 in Listing 1.2. However, in PEAK, this is done using a primitive (`promote`) to mark the operands to the shared memory, and PEAK generates vectorized memory transfers automatically if possible. Fourth, in Ansor loops are coalesced/collapsed into one single loop using the `fuse` primitive (line 13); the parallel loops are mapped to either thread block or grid level on the GPU using the `bind` primitive shown at line 15. In the `transform` dialect, currently, there is no support for loop coalescing. Therefore, PEAK does not support loop coalescing and is limited to computations with parallel dimensions less than three dimensions (GPU compute dimensions).

3.2 Autotuner Integration

The existing cost model in the MLIR compiler has a simple heuristic, and sometimes its generated code does not perform well. Autotuners search for the best-performing implementation in a large space of possible code variants. We employ autotuning search in MLIR, guided by using a dynamically constructed predictive performance model trained with timing data from the execution of many code variants on the target GPU platform. The autotuner searches for values of parameters, such as tile size and unrolling depth, in a code configuration defined by the schedule. Many existing known works [4–6, 13, 21, 26, 27] show the advantage of combining autotuners with scheduling languages.

We choose the ytopt autotuner [23], which uses Bayesian optimization. It implements various search algorithms like Random Forest(RF), Extra Trees(ET), Gradient Boosting Regression Trees(GBRT), and Gaussian process(GP). We use Matmul as an example with two different search algorithms, GBRT and RF.

Figure 3 shows how we integrate ytopt into the existing MLIR and IREE compilation flow. The generated PEAK schedule, according to computation and defined search space, are two inputs of the autotuner. In the PEAK schedule, we annotate the code with parameter placeholders, like threadblock shape (TX, TY) and grid shape (BX, BY), which are associated with multiple-level tile sizes and other performance factors. Next, the PEAK schedule is bound to parameter values chosen from the search space, and a small group of code variants are lowered down to the transform dialect and later compiled by the MLIR/IREE compiler. Each code variant is executed on the target hardware, and the execution time is collected. In the autotuner, we define the objective function to minimize the execution time.

4 Optimizations to MLIR/IREE Backend

We use the MLIR/IREE compiler as the GPU code generator in this work. The computation is expressed in MLIR IR. An MLIR `transform` dialect file is provided in MLIR/IREE to generate the final executable on the GPU. We noticed some limitations of the GPU code generation flow in the default MLIR/IREE `transform` dialect. We have implemented several optimizations in the `transform` dialect pipeline to enable MLIR/IREE to generate high-performance code for GPUs.

Thread Distribution of Shared Memory Copy Memory access efficiency is the key factor that determines performance. In the original MLIR/IREE compilation pipeline, shared memory copies were not distributed across threads. Threads in a thread block do not fetch data from/to

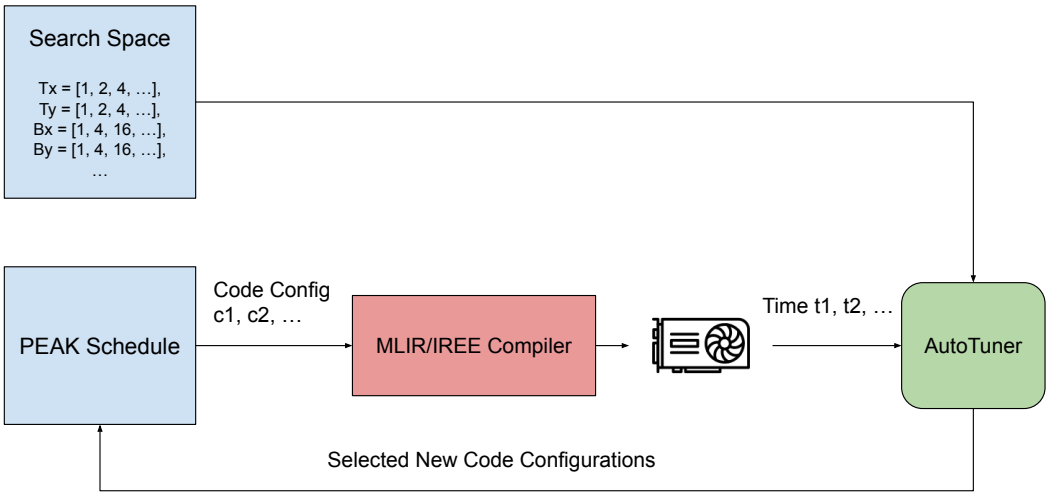


Fig. 3: Autotuner Integration in MLIR

global memory to/from shared memory in a collaborative way, instead copying the entire data chunk individually. The redundant shared memory copying results in significantly high data movement, which slows down performance. We first locate the `linalg.copy` instructions, which manages shared memory copying. An additional control flag as an attribute is added to the operation and carries the information to the MLIR/IREE compilation passes. Next, the attribute is utilized in the `GPUDistributeSharedMemoryCopy` pass in MLIR/IREE and allows data copying to be distributed among threads in a warp in the cyclic distribution way. Additionally, to generate efficient memory transfer GPU code, we enable the `GPUVectorizationPass`, which emits vectorized memory transfer instructions. Specifically, it generates `vector.transfer_read` and `vector.transfer_write` operations for data sizes with inner dimensions divisible by 128 (optimal size for data transfer in an NVIDIA GPU). The vectorized memory load and store instructions reduce the total amount of memory requests, which results in the performance gain of the final GPU code.

Memory Request Optimization In GPUs, the memory requests from a warp are grouped to form transactions. If contiguous threads in a warp request contiguous memory locations in global memory (coalesced accesses), then the total number of transactions is minimized. The MLIR/IREE compiler can emit vectorized memory transfer instructions when the inner dimension of the data is a multiple of 128. For cases where the inner dimension data type size is not divisible by 128, the default `GPUVectorizationPass` implementation falls back into a block-cyclic distribution among threads. This leads to non-coalesced data accesses and causes a performance penalty. To resolve this inefficiency and enable vectorization for arbitrary problem sizes, we modify and enhance the existing `GPUVectorizationPass`. We emit vectorization instructions with an inner dimension size of 1, represented as $vector\langle n \times 1 \rangle$, which supports any data shape, and the total number of memory instructions is reduced by the vector length. The optimization effectively hides memory latency and improves the utilization of the GPU memory pipeline unit.

5 Evaluation

In this section, we compare performance of the code generated by PEAK with the state-of-the-art autotuning framework Ansor [26] on five benchmarks. We study the performance tradeoffs and effectiveness of key optimizations in MLIR/IREE and explain the main bottleneck in MLIR/IREE compilation flow, which impacts the performance of the generated code.

Experiment Setup The experiments were carried out on NVIDIA Ampere A100 80GB GPU hosted on an AMD EPYC 7513 with a 32-core CPU running CentOS. For collecting TVM/Ansor's performance, we allowed Ansor to explore 1000 trials and used the best version found for each benchmark. We use the same number of trials in our work to present the performance results. We employ the ytopt autotuner with two distinct ML algorithms: Random Forest (RF) and Gradient Boosted Regression Trees (GBRT). Across all benchmarks, we maintained uniformity by utilizing the same schedule template, as demonstrated in Section 3.1.

Benchmarks We evaluated the following benchmarks: GEMM (General Matrix Multiplication), MatVec (Matrix Vector Multiplication), Transposed GEMM, Convolution 1D, and 2D on input sizes commonly found in AI and deep learning applications. Optimizing these kernels would potentially speed up end-to-end deep learning models. We list the benchmarks with detailed problem sizes used in our evaluation in Table 3, Table 2 for convolutions; Problem sizes of MatVec and GEMM are shown in the x-axis in figures.

Table 2: Problem sizes of 2D convolution operators in VGG (Left), ResNet (Middle) and Yolo (Right); **CI**: Input channels; **CO**: Output channels; **H**, **W**: Input image height and width; **r/s**: kernel height and width; **stride**: 1/2 (2 if marked with * after kernel name, 1 otherwise); **p**: padding value.

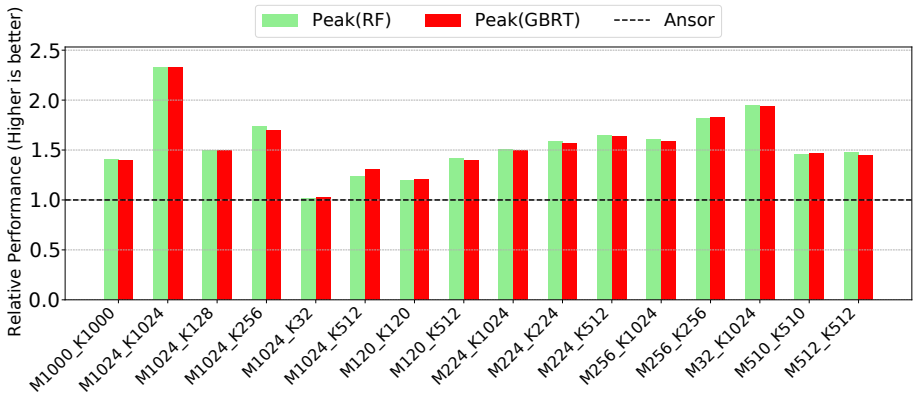
Layer	H,W	CO	CI	r/s	p
V1	224	64	3	3	1
V3	112	128	128	3	1
V5	56	256	256	3	1
V7	28	512	512	3	1
V9	14	512	512	3	1

Layer	H,W	CO	CI	r/s	p
RN1*	224	64	3	7	3
RN2(1)	56	64	64	1	0
RN2(2)	56	64	64	3	1
RN2(3)	56	256	64	1	0
RN3(1)*	56	128	256	1	0
RN3(2)	28	128	128	3	1
RN3(3)	28	512	128	1	0
RN4(1)*	28	256	512	1	0
RN4(2)	14	256	256	3	1
RN4(3)	14	1024	256	1	0
RN5(1)*	14	512	1024	1	0
RN5(2)	7	512	512	3	1
RN5(3)	7	2048	512	1	0

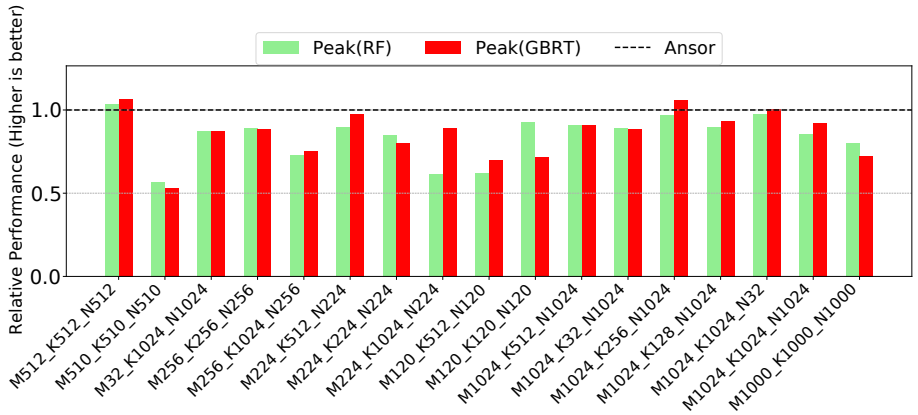
Layer	H,W	CO	CI	r/s	p
Y0	544	32	3	3	1
Y2	272	64	32	3	1
Y4	136	128	64	3	1
Y8	68	256	128	3	1
Y9	68	128	256	1	0
Y12	34	512	256	3	1
Y13	34	256	512	1	0
Y14	68	512	256	3	1
Y19	17	512	1024	1	0
Y20	17	1024	512	3	1

Performance Results Figures 4(a)-(c) show the relative performance of the code generated by PEAK compared to the best code generated by Ansor over 1000 trials, for MatVec, GEMM, and Transposed GEMM, respectively. Similarly, Figures 5(a)-(b) show the relative performance of the code generated by PEAK as compared to Ansor for Conv1D and Conv2D.

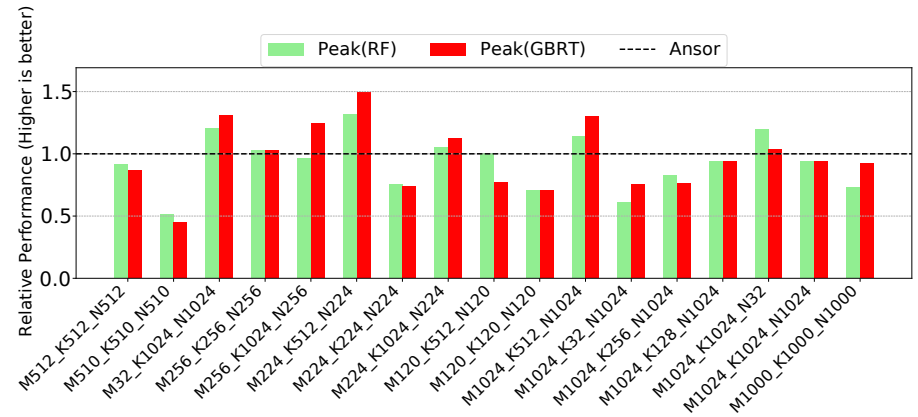
Our work, PEAK, outperforms the best code generated by Ansor in all problem sizes for MatVec (Figure 4a). PEAK achieves up to a 2.3× performance gain, with a geomean of 1.52×. The best



(a) MatVec



(b) GEMM



(c) Transposed GEMM

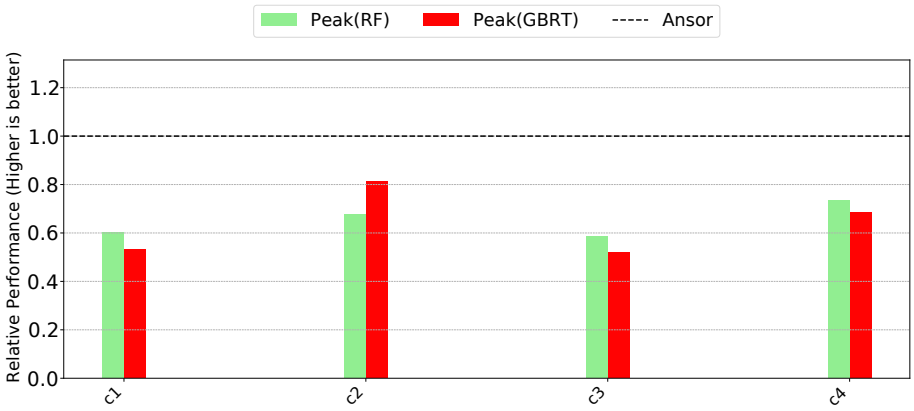
Fig. 4: Relative performance of PEAK and Ansor for (a) MatVec, (b) GEMM, and (c) Transposed GEMM.

Table 3: Problem sizes of 1D convolution operators; **CI**: Input channels; **CO**: Output channels; **W**: Input image width; **kw**: kernel width; **stride**: 1/2 (2 if marked with * after kernel name, 1 otherwise); **p**: padding value.

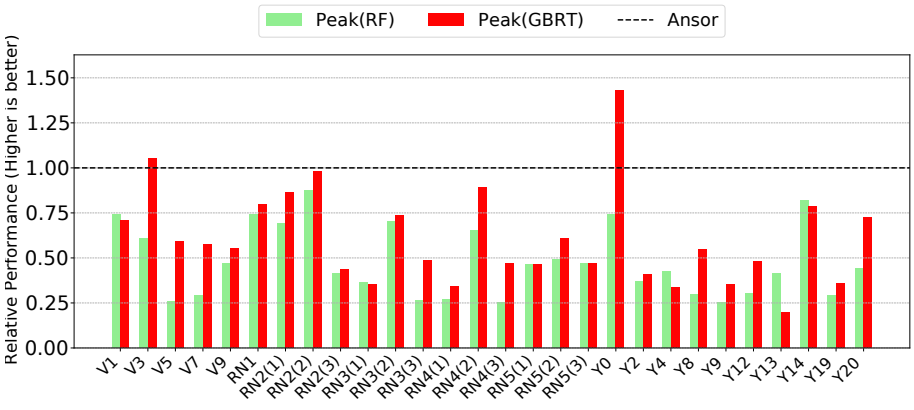
Layer	W	CO	CI	kw	p
C1*	128	256	128	1	0
C2*	256	128	64	3	1
C3	32	512	512	3	1
C4	64	256	256	5	2

code generated by PEAK minimizes bank conflicts, which results in higher memory throughput. In addition, we notice that it is beneficial in some cases to avoid using the fused multiply-add (FFMA) instructions and instead use two instructions (FADD and FMUL), and the increased ILP potentially leads to performance gain. We present the performance data of GEMM and Transposed GEMM benchmarks in Figure 4b and Figure 4c, respectively. The absolute performance results exhibit a noticeable performance gap compared to Ansor, where the geomean of speedup is 0.84× and 0.92× over problem sizes in GEMM and Transposed GEMM, respectively. The presence of increased bank conflicts in shared memory storage and reduced loop unrolling of iterations results in lower performance. This issue becomes more pronounced when dealing with input data sizes that are not powers of two. We see the performance differences in Convolution 1D and 2D shown in Figure 5a and Figure 5b. These benchmarks are particularly sensitive to optimizing memory accesses and effectively utilizing shared memory. As a result, the code's performance shows a decline in these cases.

Performance Bottleneck Analysis We notice the performance bottleneck of our work is the data copies in the underlying MLIR/IREE compiler. We investigate one 2D convolution case, which has a large performance difference. We start from the default MLIR/IREE implementation to code variants that have applied multiple optimizations in MLIR/IREE, as discussed in Section 4. As shown in Table 4, the default MLIR/IREE implementation performs poorly against TVM/Ansor, and the absolute performance is roughly 23× slower. One of the main reasons is the distribution of shared memory copy is not enabled in the IREE pipeline, which leads to redundant memory transfers and data movement. After applying thread distribution in shared memory optimization, the total number of shared memory store requests has dramatically reduced, and the new code version issues 24× less shared memory requests. Compared to these two code versions, performance improves by 6×. We further apply memory request optimization to resolve non-coalesced memory access patterns by allowing vectorized data movement. The optimization leads to a 4× improvement in shared memory store and 2× reduction in global load transactions. The overall performance has a 2.32× speedup. However, the performance difference against TVM/Ansor is still noticeable after applying the above optimizations in PEAK. Compared to TVM/Ansor, the amount of wavefront in the shared memory store is 8× more and the performance is slower by 1.69×. We conducted further investigation, and we noticed two main points in TVM/Ansor: (1) TVM employs techniques such as software pipelining, unrolling, and virtual threads to fully utilize registers, to maintain good instruction-level parallelism (ILP), and to distribute data copying workload evenly among threads in the thread block; and, (2) TVM/Ansor has the linearized data buffer view instead of an N-dimensional buffer in MLIR/IREE which allows contiguous memory data copy and avoids non-coalesced access on GPU.



(a) Conv1D



(b) Conv2D

Fig. 5: Performance comparison in convolution operation

Table 4: Comparison over various optimizations; Performance is measured in the unit of TFLOPS on A100 GPU; ST means store and LD means load.

Cases	Performance	SM LD Wavefront	SM ST Wavefront	Global LD Wavefront
Default IREE	0.11	7,741,440	96,337,920	77,236,464
Thread Distribution(SM)	0.64	6,881,280	4,042,752	3,999,744
Memory Request Optimization	1.49	8,945,664	1,634,304	1,724,860
TVM/Ansor	2.52	4,344,041	272,384	532,233

6 Conclusion and Future Work

In our work, we proposed PEAK, a high-level scheduling language in MLIR, making it useful for users to generate high-performance code without knowing the low-level MLIR details. We enable

the autotuner to replace the existing naive cost model in MLIR. In comparing PEAK with state-of-the-art Ansor, our analysis provides valuable insights into performance variations across different compiler stacks and paves the way for future improvements in MLIR. In future work, we plan to conduct more experiments with end-to-end model evaluation and include more types of computation operators. We plan to generalize the scheduling language design beyond GPUs and extend this work to more target architectures.

7 Acknowledgement

Thanks to Nicolas Vasilache, Alex Zinenko, Mahesh Ravishankar, Thomas Raoux, Albert Cohen, Mahesh Lakshminarasimhan, and Akash Kulkarni for their help and guidance. Thanks to the anonymous reviewers for their thorough and insightful feedback. The support and resources from the Center for High-Performance Computing at the University of Utah are gratefully acknowledged. Research reported in this publication was supported in part by the National Science Foundation through awards CCF-2107556 and OAC-2217154. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Science Foundation.

References

1. Iree: Intermediate representation execution environment. `https://openxla.github.io/iree/`, last accessed 4 September 2023
2. Schedule primitives in tvm. `https://tvm.apache.org/docs/how_to/work_with_schedules/schedule_primitives.html`, last accessed 2 September 2023
3. Transform dialect in mlir. `https://mlir.llvm.org/docs/Dialects/Transform/`, last accessed 2 September 2023
4. Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., et al.: Learning to optimize halide with tree search and random programs. ACM Transactions on Graphics (TOG) **38**(4), 1–12 (2019)
5. Anderson, L., Adams, A., Ma, K., Li, T.M., Jin, T., Ragan-Kelley, J.: Efficient automatic scheduling of imaging and vision pipelines for the gpu. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 1–28 (2021)
6. Bansal, M., Hsu, O., Olukotun, K., Kjolstad, F.: Mosaic: An interoperable compiler for tensor algebra. Proceedings of the ACM on Programming Languages **7**(PLDI), 394–419 (2023)
7. Chen, C., Chame, J., Hall, M.: Chill: A framework for composing high-level loop transformations. Tech. rep., Citeseer (2008)
8. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: Tvm: An automated end-to-end optimizing compiler for deep learning (2018)
9. Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., Krishnamurthy, A.: Learning to optimize tensor programs. Advances in Neural Information Processing Systems **31** (2018)
10. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
11. Hagedorn, B., Elliott, A.S., Barthels, H., Bodik, R., Grover, V.: Fireiron: a data-movement-aware scheduling language for gpus. In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques. pp. 71–82 (2020)
12. Hagedorn, B., Lenfers, J., Koehler, T., Gorlatch, S., Steuwer, M.: A language for describing optimization strategies. arXiv preprint arXiv:2002.02268 (2020)

13. Haj-Ali, A., Genc, H., Huang, Q., Moses, W., Wawrzynek, J., Asanović, K., Stoica, I.: Protuner: tuning programs with monte carlo tree search. arXiv preprint arXiv:2005.13685 (2020)
14. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
15. Ikarashi, Y., Bernstein, G.L., Reinking, A., Genc, H., Ragan-Kelley, J.: Exocompilation for productive programming of hardware accelerators. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 703–718 (2022)
16. Kerr, A., Merrill, D., Demouth, J., Tran, J.: Cutlass: Fast linear algebra in cuda c++. `.https://devblogs.nvidia.com/cutlass-linear-algebra-cuda`
17. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: Mlir: Scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308
18. Nvida: Basic linear algebra on nvidia gpus. `https://docs.nvidia.com/cuda/cublas/index.html/`
19. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices **48**(6), 519–530 (2013)
20. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 779–788 (2016)
21. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A scalable auto-tuning framework for compiler optimization. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–12. IEEE (2009)
22. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. arXiv preprint arXiv:1802.04730 (2018)
23. Wu, X., Balaprakash, P., Kruse, M., Koo, J., Videau, B., Hovland, P., Taylor, V., Geltz, B., Jana, S., Hall, M.: ytopt: Autotuning scientific applications for energy efficiency at large scales. arXiv preprint arXiv:2303.16245 (2023)
24. Xu, Y., Yuan, Q., Barton, E.C., Li, R., Sadayappan, P., Sukumaran-Rajam, A.: Effective performance modeling and domain-specific compiler optimization of cnns for gpus. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. pp. 252–264 (2022)
25. Zhang, Y., Yang, M., Baghdadi, R., Kamil, S., Shun, J., Amarasinghe, S.: Graphit: A high-performance dsl for graph analytics. arXiv preprint arXiv:1805.00923 (2018)
26. Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C.H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., et al.: Ansor: Generating {High-Performance} tensor programs for deep learning. In: 14th USENIX symposium on operating systems design and implementation (OSDI 20). pp. 863–879 (2020)
27. Zheng, S., Liang, Y., Wang, S., Chen, R., Sheng, K.: Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 859–873 (2020)