Uncovering Magic with Magic: Schedule Reconstruction from High-Performance Kernel Libraries

Hongzheng Chen*
Cornell University
Ithaca, New York, USA
hzchen@cs.cornell.edu

1 Introduction

To fully harness the power of hardware accelerators, high-performance kernels are essential for various application domains in deep learning and scientific computing. Many of these kernels are carefully crafted and optimized by hand using languages like C/C++, demanding significant engineering efforts. Recent developments in kernel libraries have embraced schedule languages [1–4] that decouple program optimization (i.e., schedule) from algorithm specification. Users only need to write a few lines of schedule code to transform a basic program into an optimized one. However, this shift merely transfers the coding burden from writing optimized kernels to writing efficient schedules. While autoschedulers help generate optimized programs automatically, users have little control over the optimizations and their granularity remains opaque.

To tackle this challenge, we introduce a novel research problem: *schedule reconstruction*, that is, reconstructing the schedule primitives from an optimized kernel implementation. Unlike existing approaches that focus on generating optimized kernels, our goal is to *reveal the optimizations within these kernels by generating step-by-step schedule primitives*. By providing an algorithm specification and an optimized kernel implementation, we propose a two-stage program synthesis technique to automatically generate the necessary schedule for this transformation. While this paper represents an initial exploration of schedule synthesis, our work demonstrates the potential of this idea to: (1) Aiding programmers in comprehending the intricacies of high-performance program optimizations, making it easier to debug and allowing them to experiment with different tradeoff combinations. (2) Potentially providing templates for automating compilers to generate even more high-performance code than manually optimized ones.

2 Background and Related Works

Several domain-specific languages (DSLs) have emerged to decouple program optimizations from algorithm definition, offering users a set of schedule primitives for efficiently conducting complex program transformations. Halide [1] pioneered the concept of algorithm and optimization decoupling in the domain of image processing. TVM [2] subsequently extended this approach to support deep learning applications, enabling the description of CPU and GPU optimizations using a schedule language. HeteroCL [3] and Exo [4] further developed and applied this idea to hardware accelerators. In Fig. 1(a), we provide an example of a general matrix-matrix multiplication (GEMM) kernel written in a schedule language. This programming style is common in those scheduling languages[1–3], denoted as `dsl`. Users start by providing an algorithm specification

```
1  # Algorithm specification
2  import dsl
3  M, N, K = 1024, 1024, 1024
4  k = dsl.reduce_axis((0, K), "k")
5  A = dsl.placeholder((M, K), name="A")
6  B = dsl.placeholder((K, N), name="B")
7  C = dsl.compute((M, N), lambda i, j:
8    dsl.sum(A[i, k] * B[k, j], axis=k),
9    name="C") # S1
10
11 # Schedule construction
12 s = dsl.create_schedule(C)
13 s.<primitive>(<arguments>)
```

```
1  void gemm(float A[M][K], float B[K][N],
2            float C[M][N]) {
3   #pragma HLS partition variable=A dim=1
4   #pragma HLS partition variable=B dim=0
5   for (int v0 = 0; v0 < 16; v0 += 1) {
6    for (int v1 = 0; v1 < 64; v1 += 1) {
7     for (...) {
8      #pragma HLS pipeline II=1
9      float v18 = A[v0 * 64 + v1][v2];
10     float v19 = ...
11     // ...
12     // more computation
13 }}}}
```

Figure 1: (a) Left: An example GEMM kernel described in a schedule language. (b) Right: An optimized GEMM implementation in high-level synthesis (HLS) targeting accelerators.

Table 1: A partial list of the schedule primitives supported by commonly used schedule languages like TVM and HeteroCL.

Primitive	Description
`s.split(i,v)`	Split loop `i` into a two-level nested loop with `v` as the bound of the inner loop.
`s.reorder(*l)`	Switch the order of sub-loops `l`.
`s.unroll(i,v)`	Unroll loop `i` by factor `v`.
`s.parallel(i)`	Schedule loop `i` in parallel.
`s.pipeline(i,v)`	Schedule loop `i` in a pipeline manner with a target initiation interval `v`.
`s.partition(A,d,v)`	Cyclic/Block partition dimension `d` of array `A` with a factor `v`.

(Lines 3-9) to describe *what* is the computation. Then, they can create a schedule and apply primitives (Lines 12-13) to specify *how* the computation should be executed. Table 1 lists some commonly used primitives supported by TVM and HeteroCL.

In this project, we consider schedule reconstruction. Given the algorithm specification of the kernel (Lines 1-9 in Fig. 1(a)) and an optimized kernel implementation (Fig. 1(b)), our goal is to reconstruct the schedule (Lines 12-13) that can transform the vanilla program defined in the specification to the optimized one. The optimized code here is for demonstration purposes, showing that achieving high performance in a kernel may require significant code restructuring and pragma insertion.

The closest related work of this paper is Dexter [5], which leverages program synthesis to automatically translate image processing libraries to Halide. However, it only recovers the algorithm definition but not the schedule, making it again a black-box optimization approach.

3 Methodology

Fig. 2 provides an overview of the synthesis process. We first parse both the DSL and the target C++ program into an intermediate

*This paper was presented at the PLDI'24 Student Research Competition (SRC) and earned third place in the final round.

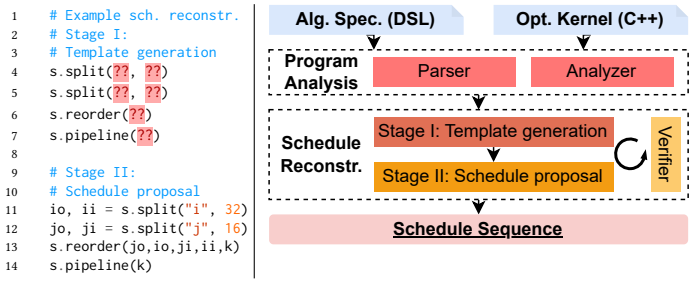

Figure 2: Overview of the schedule reconstruction process.

representation (IR) and perform static analysis to extract essential information, such as variable names and loop bounds. With this information, we employ a two-stage synthesis technique to reconstruct the schedule sequence. First, we generate a schedule template containing the primitives that will be invoked but without the arguments. Then, we search through the loop variables and factors to complete the template. Once a schedule is generated, we utilize a verifier [6] to ensure the schedule can indeed produce the target optimized program. If verification fails, the synthesizer is notified and proposes new schedules iteratively.

(1) Program Analysis. We first analyze the source program to extract necessary information for schedule synthesis. As these DSLs typically describe nested loops, we leverage Presburger representation [7] in polyhedral analysis [8] to represent loop-based programs. Specifically, we represent the *memory access pattern* as:

$$S(\mathbf{i}) \rightarrow \text{Array}(f_0(\mathbf{i}), \ldots, f_N(\mathbf{i})), \mathbf{i} = (i_0, \ldots, i_N), \quad (1)$$

where $\mathbf{i}$ is the vector of loop variables of N nested loops, and $f_{(\cdot)}$ is the function mapping from loop variables to memory access indices. For example, for the GEMM program in Fig. 1(a), we can explicitly represent the memory access of arrays A and C as follows, with each mapping the loop indices to the actual memory access indices:

$$R_{S1,A}(i,j) = \{S1(i,j,k) \rightarrow A(i,k) : 0 \le i,j,k < 1024\}$$
$$W_{S1,C}(i,j) = \{S1(i,j,k) \rightarrow C(i,j) : 0 \le i,j,k < 1024\}$$

where R and W denote different read and write accesses. The subscript indicates the compute statement (e.g., S1 – Line 7 in Fig. 1(a)). $S1(i,j,k) \rightarrow A(i,k)$ basically means the loop indices i,j,k are mapped to access the location of (i,k) of array A.

(2) Schedule Reconstruction. To make the reconstruction process scalable, we decompose the synthesis problem into two stages.

Stage I: Template generation. We first identify which primitives will be used in the schedule. We classify the primitives into two categories based on whether they require changing the program structure, as shown in Table 2. The first three primitives need to modify the program structure, requiring an explicit search process to determine their arguments. By carefully examining these transformations, we observe patterns that can aid our synthesis. For instance, the `.reorder()` primitive maintains the same number of iteration variables in the transformed program, while the `.split()` primitive increases the number of iteration variables by one. Therefore, we can determine the number of these primitives in the schedule. The latter three primitives only require annotations and can thus be handled at the end of the synthesis process. Based on these observations, we create a template schedule program that consists of a sequence of primitives with holes to be filled, as shown in the left of Fig. 2. By leveraging domain knowledge and generating a template, we can greatly reduce the search space and simplify the synthesis process.

Table 2: Schedule primitives in Presburger representation.

Primitives Requiring Program Structure Modification	
`.split(t,c)`	$S(i_0,\ldots,i_{t_i},i_{t_o}\ldots i_N) \rightarrow A(f_0(\mathbf{i}'),\ldots,f_N(\mathbf{i}'))$ $\mathbf{i}' = (i_0,\ldots,i_{t_i}+i_{t_o}\cdot c,\ldots,i_N)$
`.reorder(σ)`	$S(i_{\sigma(0)},\ldots,i_{\sigma(N)}) \rightarrow A(f_{\sigma(0)}(\mathbf{i}),\ldots,f_{\sigma(N)}(\mathbf{i}))$ σ is the permutation function
`.unroll(t,c)`	$S(i_0,\ldots,i_{t'}\ldots i_N) \rightarrow A(f_0(\mathbf{i}'),\ldots,f_N(\mathbf{i}'))$ $\mathbf{i}' = (i_0,\ldots,i_{t'} \mod c,\ldots,i_N)$
Primitives Only Annotating Operators	
`.parallel(i)`	Annotation
`.pipeline(i)`	Annotation
`.partition(A,d,v)`	Annotation

Stage II: Schedule proposal. The next step is to fill in the holes in the template. We establish an equivalent relation between all the memory access patterns generated by the schedule and the target memory access patterns, leveraging the z3 solver [9] to propose a new solution that satisfies these constraints. Since we only ensure the access patterns match and cannot guarantee that the computation is identical, we employ the verifier [6], which formally verifies the equivalence of two C++ programs, to further verify the schedule is valid. The process is iterative, where each iteration proposes new schedules, applies transformations, and verifies program equivalence until a valid schedule is found.

4 Evaluation

To assess the effectiveness of our proposed synthesizer across diverse programs, we conduct experiments using PolyBench [10], a widely-used benchmark containing multiple kernels from scientific applications. We leverage kernels generated by ScaleHLS [11] as the inputs, which are highly optimized kernels for FPGA accelerators. Our approach utilizes HeteroCL [3] as the frontend DSL to reconstruct the schedules, but the methodology is applicable to other schedule languages such as TVM [2] and Exo [4].

Experimental results are shown in Fig. 3. Across all the 13 benchmarks, our synthesizer consistently and successfully produced schedule sequences, demonstrating its effectiveness in handling various application programs. Furthermore, the synthesis process for all benchmarks was completed in seconds, highlighting the practicality and efficiency of our approach.

Table 3: Synthesis results on PolyBench.

Benchmark	# of Synthesized Primitives	Benchmark	# of Synthesized Primitives
2mm	9	jacobi-2d	8
3mm	15	mvt	6
atax	6	symm	8
bicg	7	syr2k	10
correlation	5	syrk	9
gemm	3	trmm	10
gesummv	2		

5 Conclusion and Limitations

In this project, we introduce a synthesizer designed to reconstruct the schedule primitives for an optimized kernel, thereby unveiling the black box of program optimizations and enabling programmers to understand the optimization process.

However, several limitations exist. (1) Our current approach only considers compute and annotation-only primitives. To achieve high performance on CPUs and GPUs, programmers often need to create multi-level caches in the program, requiring support for memory customizations (e.g., bufferization). (2) Our synthesizer currently only supports finely-structured program pairs. Expanding its capabilities to handle more general program structures, such as hierarchical programs with function calls, would significantly enhance its utility and generality.

References

[1] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, 2013.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 2018.

[3] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

[4] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022.

[5] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Trans. Graph.*, 38(6), nov 2019.

[6] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. Formal verification of source-to-source transformations for hls. In *The 2024 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2024.

[7] Sven Verdoolaege. Presburger formulas and polyhedral compilation. *Polly Labs and KU Leuven*, 2016.

[8] Polyhedral Compilation. Polyhedral compilation. http://polyhedral.info/, 2023.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Louis-Noël Pouchet. Polybench/c: The polyhedral benchmark suite. https://web.cs.ucla.edu/~pouchet/software/polybench/, 2023.

[11] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. Scalehls: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 1355–1358, New York, NY, USA, 2022. Association for Computing Machinery.