KRIEK: A SYNTHESIS-AIDED TENSOR COMPILER

TECHNICAL REPORT 0.1

Samuel Kaufman **Rastislav Bodik** **Rene Just**

Paul G. Allen School for Computer Science and Engineering
University of Washington
Seattle, WA 98195–2355
{kaufmans,bodik,rjust}@cs.washington.edu

August 29, 2021

ABSTRACT

The problem of compiling ML pipelines onto parallel hardware requires careful mapping of data and computation onto the hardware as well as intelligent compilation of data movements. In recent work, program synthesis produced tensor kernels that performed better than manually implemented kernels.The goal of this work is to extend extend synthesis from the leaves of the implementation tree to the entire implementation tree. In other words, we want to apply synthesis to autoscheduling of ML pipelines.

Our approach to scaling synthesis to large implementations is to decompose synthesis into smaller problems, and to do so hierarchically, so that larger implementations can be composed from smaller implementations which were in turn produced from yet smaller implementations. We describe a hierarchical synthesis decomposition together with a dynamic programming algorithm that constructs optimal implementations without uselessly resynthesizing subproblems.

We also describe a compiler architecture that facilitates this hierarchical synthesis including an specification IR that describes subproblems; an executable IR that describes implementation strategies; and a scheduling language that refines a specification into a partial implementation that in turn invokes subproblems.

Keywords Tensor Compilers · Auto Scheduling · Program Synthesis

1 Overview

This technical report is a snapshot of the current state of Kriek design and implementation. We will update this report as the system matures.

Kriek is a scheduling compiler in that it generates the implementation by applying a schedule on the source program, also called a specification. An example of schduling a single matrix multiplication is shown in Figure 1.

Critical for hierarchical synthesis of schedules is Kriek's ability to represent subproblems with specifications. For example, in Figure 1, the source program has the specification `Matmul` with dimensions (128, 128, 128). This specification is implemented at the top level with a `tile_out` schedule operator. This operator generates the outermost loop of the implementation; this loop is a `tile` loop, which means that the loop iterates over the tiles of the output of the `Matmul`, as well as the corresponding tiles of its input tensors. Now, the `tile_out` schedule operator decides only the topmost level of the implementation; the body of this loop is a subproblem to synthesis. The subproblem introduced by the `tile_out` operator is a `Matmul` with smaller dimensions, given in the figure as specification 2. During synthesis, implementations are computed bottom up, inverting the process by which a schedule generates an implementation.

To the best of our knowledge, Kriek makes the following contributions:

```
# The specification of the source program. Arguments 'level' and 'layout'
# indicate the mapping of the logical (specification) tensors to physical memory.
spec = specs.Matmul(
    specs.TensorSpec((128, 128), level=1, layout=Layout.ROW_MAJOR),
    specs.TensorSpec((128, 128), level=1, layout=Layout.ROW_MAJOR),
    output=specs.TensorSpec((128, 128), level=1, layout=Layout.ROW_MAJOR),
)

# A schedulable specification.
lhs = tensor.Tensor(spec.inputs[0], name="lhs")
rhs = tensor.Tensor(spec.inputs[1], name="rhs")
out = tensor.Tensor(spec.output, name="out")

schedulable_spec = ops.spec_to_hole(spec, (lhs, rhs), out)

# An implementation is produced by applying a schedule on a specification.
impl = (
    schedulable_spec
    .tile_out((32, 32))
    .move_input(0, level=0)
    .split(64)
    .move_input(1, level=0)
    .move_output(level=00)
    .complete()
)

# The implementation generated by applying the schedule on the spec.
#
# Inputs: lhs: Tensor(128×128), rhs: Tensor(128×128)
# Output: out: Tensor(128×128)
#
# 1. Matmul((128×128), (128×128), (128×128), serial)
tile (rhsT: 128×32, lhsT: 32×128, outT: 32×32) <- (rhs, lhs, out)
    # 2. Matmul((32×128), (128×32), (32×32), serial)
    move[lvl=0] lhsTr <- lhsT
    # 3. Matmul((32×128, lvl=0), (128×32), (32×32), serial)
    split (lhsTrT: 32×64, rhsTt: 64×32) <- (lhsTr, rhsT)
        # 4. Matmul((32×64, lvl=0), (64×32), (32×32), serial)
        move[lvl=0] rhsTtR <- rhsTt
        # 5. Matmul((32×64, lvl=0), (64×32, lvl=0), (32×32), serial)
        move*[lvl=0] outTr <- outT
        # 6. Matmul((32×64, lvl=0), (64×32, lvl=0), (32×32, lvl=0), serial)
        tile (lhsTrTt: 1×64, rhsTtRt: 64×1, outTrT: 1×1) <- (lhsTrT, rhsTtR, outTr)
            # 7. Matmul((1×64, lvl=0), (64×1, lvl=0), (1×1, lvl=0), serial)
            split (lhsTrTtT: 1×1, rhsTtRtT: 1×1) <- (lhsTrTt, rhsTtRt)
                # 8. Matmul((1×1, lvl=0), (1×1, lvl=0), (1×1, lvl=0), serial)
                Matmul(lhsTrTtT, rhsTtRtT, outTrT)
        # end 6.
        store outT <- outTr
        # end 5, 4.
    # end 3, 2.
```

Figure 1: Scheduling a matrix multiplication in Kriek. Note that the innermost tile/split loop nest was generated by the 'complete' operator which applied the default 'Matmul' schedule on the current spec, which was spec 6.

- An algorithm for synthesis with dynamic programming for tensor programs, based on decomposing a synthesis problem into subproblems. Dynamic programming allows exploring a larger space of implementations than beam search because it memoizes solutions to subproblems and considers all subproblems in constructing implementations for larger subproblems.
- A program representation that allows such hierarchical decomposition for compositions (i.e., pipelines) of convolutions, matrix multiplies, and data moves. In particular, Kriek scheduling operators can decompose (a problem of synthesizing implementations for) such pipelines into (a problem of synthesizing an implementation for) a smaller version of the pipeline obtained by tiling the original pipeline.
- This project evaluates the trade offs adopted in the Kriek design, such as the how much benefit we gain by exploring a larger space of candidate implementations (vis-a-vis a beam search) versus how much cost modeling accuracy we lose by predicting the cost of an implementation compositionally, from the costs of subproblems, as opposed to by feeding to the cost predictor the entire prefix of the implementation, as is common in beam search.

We also evaluate the benefits and limitations of the current design.

- Are there optimizations that we obtain for free by generalizing scheduling operators? For example, pipeline fusion decisions, at multiple levels of computation granularity, are obtained as a side effect of optimally applying just two scheduling operations: tiling a pipeline and buffer introduction.
- Kriek simplifies its design by compiling into a intermediate language with tensor tiling iterators; no iteration space analysis is performed during code generation or during tiling. This decision isolates the code generation effort in the implementation of the iterator but there may be desirable execution strategies that would be difficult to describe with tiling iterators.
- We are exploring how Kriek's explicit mapping of logical tensors into physical arrays can be used to conveniently represent efficient implementations data structures such as sliding window buffers. This is done by making the logical-physical mapping be indexed by the iteration number.

Kriek follows the scheduling approach of Halide. More immediately, Kriek has its intellectual roots in Fireiron, which developed a scheduling language and an IR that treated data moves as explicitly schedulable first-class subproblems. Kriek addresses problems not considered by Fireiron:

- Fireiron handled only input programs that computed a matrix multiplication. Kriek designs an IR that supports also convolutions and pipelines, i.e., compositions of convolutions, matrix multiplications, and data moves.
- Kriek takes advantage of reuse available in convolution computations with sliding windows.
- Fireiron scheduled the computations manually, while Kriek automates schedule generation with synthesis.
- While Fireiron was motivated by advanced implementations on GPUs, Kriek's design has (so far) been motivated by advanced vector implementations.

2 The Kriek Architecture

2.1 The Specification Language Spec

Spec is a high-level language that describes the synthesis sub-problems that arise during the hierarchical synthesis of an implementation for a source program p. The input program p is itself described as a specification from this language. We call expressions from S *specifications* because they fully describe the (sub)computations that need to be implemented.

Figure 2 lists three example specifications. Spec expressions are functions (such as `Matmul`, `Convolution`, `ReduceSum`, and `Move`), and their compositions. Functions are statically typed with rank and dimensions of their operators.

A specification describes only input-output behavior and is thus free of any implementation decisions. As such, a specification permits multiple implementations, which may differ in how the computation is tiled, fused, and parallelized, and so on. A subset of these implementations are Pareto optimal; they differ in, say, the amount of memory vs. the time needed to perform the computation.

For pragmatic reasons, Kriek specifications include the implementation decision of how the tensor parameters to the specification are mapped to memory. This information includes everything necessary to generate the code of the implementation and to estimate the cost of the implementation. This includes the level of memory (with level=0 being

```
# A matrix multiplication with dimensions (128, 128, 128)
mm = specs.Matmul(
    specs.TensorSpec((128, 128)),
    specs.TensorSpec((128, 128)),
    output=specs.TensorSpec((128, 128)),
)

# A convolution on an 128x128 tensor with a 5x5x32 filter.
conv = specs.Convolution(
    specs.TensorSpec((128, 128),
    specs.TensorSpec((5, 5, 32),
    output=specs.TensorSpec((124, 124, 32),
)

# A pipeline of a convolution, a reduction, and another convolution
img =       specs.TensorSpec((10, 10))
filters_a = specs.TensorSpec((5, 5, 6))
filters_b = specs.TensorSpec((5, 5, 6))
output    = specs.TensorSpec((2, 2, 6))
spec = specs.Compose(
    (specs.Convolution, specs.ReduceSum, specs.Convolution),
    (filters_b, img, filters_a),
    output,
)
```

Figure 2: Example Kriek specifications.

registers) and the layout (such as row-major). Constraining specification with the implementation choices for data produces implementations that compose because their inputs and outputs have compatible layout. The following is the actual specification of the convolution from Figure 2:

```
conv = specs.Convolution(
    specs.TensorSpec((128, 128), level=1, layout=Layout.ROW_MAJOR),
    specs.TensorSpec((5, 5, 32), level=1, layout=Layout.ROW_MAJOR),
    output=specs.TensorSpec((124, 124, 32), level=1, layout=Layout.ROW_MAJOR),
)
```

Since a specification fully describes the computation, we can cache optimal implementations of a large space of specifications. There are at least two reasons to do so:

- to accelerate compilation by reusing the compilations of identical subproblems in the same source program or in other source programs.
- to perform exhaustive synthesis with dynamic programming, by constructing implementations of "larger" specifications from optimal implementations of "smaller" specifications. This will be discussed later in Section 3.

2.2 The Implementation Language Impl

Impl is an intermediate language that expresses the implementation decisions of a schedule, including

- the nested loop structure, including blocked array traversal,
- allocation of temporary arrays,
- data movement, and
- the actual computation.

Kriek does not place many requirements on the design of the Impl language (aside for being able to generate efficient code from it). This is because Kriek does not optimize programs in the Impl languge; once a program is generated by a schedule, it is not transformed to, say, interchange loops. If an implementation with interchanged loops is desired, it

```
spec = specs.Convolution(
    specs.TensorSpec((128, 128), level=1, layout=Layout.ROW_MAJOR),
    specs.TensorSpec((5, 5, 32), level=1, layout=Layout.ROW_MAJOR),
    output=specs.TensorSpec((124, 124, 32), level=1, layout=Layout.ROW_MAJOR),
    serial_only=True,
)

img = tensor.Tensor(spec.inputs[0], name="img")
filters = tensor.Tensor(spec.inputs[1], name="filters")
out = tensor.Tensor(spec.output, name="out")
impl = (
    ops.spec_to_hole(spec, (img, filters), out)
    .sliding_tile_out(sliding_dim=0, output_size=32, level=1)
    .tile_out((32, 32, 1))
    .move_input(0, level=0)
    .move_input(1, level=0)
    .move_output(level=0)
    .complete()
)

# Implementation
#
# Inputs: img: Tensor(128×128), filters: Tensor(5×5×32)
# Output: out: Tensor(124×124×32)

# Conv((128×128), (5×5×32), (124×124×32), serial)
slide[32] (imgS: Tensor(36×128), outT: 32×124×32) <- (img, out)
    # Conv((36×128), (5×5×32), (32×124×32), serial)
    tile (imgSt: conv 36×36, filtersT: 5×5×1, outTt: 32×32×1) <- (imgS, filters, outT)
        # Conv((36×36), (5×5×1), (32×32×1), serial)
        move[lvl=0] imgStR <- imgSt
        # Conv((36×36, lvl=0), (5×5×1), (32×32×1), serial)
        move[lvl=0] filtersTr <- filtersT
        # Conv((36×36, lvl=0), (5×5×1, lvl=0), (32×32×1), serial)
        move*[lvl=0] outTtR <- outTt
        # Conv((36×36, lvl=0), (5×5×1, lvl=0), (32×32×1, lvl=0), serial)
        tile (imgStRt: conv 5×5, outTtRt: 1×1×1) <- (imgStR, outTtR)
            # Conv((5×5, lvl=0), (5×5×1, lvl=0), (1×1×1, lvl=0), serial)
            DirectConv(imgStRt, filtersTr, outTtRt)
        store outTt <- outTtR
```

Figure 3: An implementation of a convolution that first tiles the input image, then the filters, and finally also tiles the resulting convolution.

will be generated by a different schedule. And since Kriek explores all schedules, the implementation will be considered without rewriting of Impl programs.

The language is centered around loops that co-iterate over tiles of tensors that participate in a computation. Figure 1 showed an implementation of a matrix multiply that used `tile` and `split` loops. These two matrices are sufficient to express advanced matrix multiplies, at least on a GPU. The former loop decomposes a matrix multiplication by tiling the output, computing one tile of the output in each iteration. The latter loop decomposes a matrix multiply by tiling the input tensors, computing a contribution to the output using an outer product.

Implementations of convolutions require a variant on the `tile` loop. This is because partitioning the output of a convolution requires walking over the input image using overlapping tiles. Figure 3 shows an implementation of a convolution with such a loop, named `slide[k]`, where k indicates by how much the tile moves in the input tensor.

```
# Inputs: lhs: Tensor(128×128), rhs: Tensor(128×128)
# Output: out: Tensor(128×128)

tile (rhsT: 128×32, lhsT: 32×128, outT: 32×32)
                                  <- (rhs, lhs, out)        450560 = 16 * _     7168     0
    move[lvl=0] lhsTr <- lhsT                                28160 = 7040 + _   7168     0
    split (lhsTrT: 32×64, rhsTt: 64×32)
                                  <- (lhsTr, rhsT)          21120 = 2 * _      3072     0
        move[lvl=0] rhsTtR <- rhsTt                          10560 = 7040 + _   3072     0
        move*[lvl=0] outTr <- outT                            3520 = 3520 + _   1024     0
        tile (lhsTrTt: 1×64, rhsTtRt: 64×1, outTrT: 1×1)
                                  <- (lhsTrT, rhsTtR, outTr)     0 = 1024 * _  0        0
            split (lhsTrTtT: 1×1, rhsTtRtT: 1×1)
                                  <- (lhsTrTt, rhsTtRt)          0 = 64 * _    0        0
                Matmul(lhsTrTtT, rhsTtRtT, outTrT)               0             0        0
        store outT <- outTr
```

Figure 4: An implementation of a matrix multiply annotated with the costs of subproblems computed during dynamic programming.

2.3 The Scheduling Language Sched

The Sched language provides operators that transform a specification into an implementation that includes "smaller" specifications. These specifications are then scheduled with the rest of the schedule.

We extend the scheduling language of Fireiron with operations for scheduling pipelines, where one has the choice of applying an operation (such as `tile_out`) across the entire pipeline or splitting the pipeline by inserting a buffer between some stages and then tiling the two parts of the pipeline. Another decision is how large should be the buffer, which determines at what tile size the two stages are connected.

3 Schedule Synthesis with Dynamic Programming

3.1 Exhaustive Dynamic Programming

The hierarchical synthesis proceeds by applying all possible schedules to the input specification. This happens by applying single scheduling operators at a time. Given a specification s, we apply all single legal schedule operators to s, producing implementations with subproblems. Next, we apply all possible single schedule operators on the generated subproblems. The costs of the best resulting implementations are memoized, so that a specification need not be resynthesized.

Figure 4 shows the example from Figure 1 annotated with the cost of the implementation that was computed during dynamic programming.

3.2 Sparse Dynamic Programming

As the space of specifications increases, it is infeasible to compute optimal solutions for each specification. The Kriek design computes the DP table sparsely and interpolates the cost of the optimal solutions not stored in the table by interpolating among "nearby" specifications. This feature is under development.