Software optimization for a RISC-V accelerator: A case study

JULIEN DE CASTELNAU

1 INTRODUCTION

Accelerators have become ubiquitous in the late stages of Moore's Law. To achieve high performance in compute-bound workloads, it is increasingly necessary to exploit the specifics of an application in hardware. It is typical to imagine performant software arising from algorithmic optimizations, or judicious choice of programming language, for instance, C over Python. However, a study[3] found that for a simple matrix multiplication routine written in Python, the performance improvement of switching to C was around 50x, while rewriting the C program to utilize the CPU cache efficiently and use SIMD vector operations to compute yielded a relative speedup of around 1300x. Note that each program employs the same $O(n^3)$ algorithm. With more and more chip area dedicated to highly application-specific compute engines, the need to develop software which properly utilizes them becomes clear. This issue will serve as the focal point for this report. We seek to understand the challenges associated with software development for accelerators and the proposed solutions for automation in the literature. We will use a simple accelerator for dense matrix multiplication, a prominent workload in deep learning, as a case study through which we can discuss this question.

Section 2 provides an overview of the accelerator's architecture and the ISA providing the interface between software and hardware, and later delves into the process of developing an optimized convolution routine by hand to make use of this accelerator. This manually writen routine will serve as a base to compare the results of automated tools to. Section 3 discusses the application of TVM, a compiler which promises productivity improvements through its custom domain specific language (DSL) designed to write high-performance programs. Section 4 introduces Exo, which furthers these ideas through the separation of accelerator support into user-written "libraries", among other advancements. Section 5 discusses OptiTrust, which omits a DSL and instead allows for incremental rewrites directly in C/C++ source.

2 BACKGROUND

The accelerator for this case study is integrated with the RISC-V microcontroller X-HEEP[4]. Similar to the RISC-V Vector extensions (RVV), it is programmed through the use of RISC-V instructions in a custom ISA extension dubbed RVM. We will discuss RVM then turn to our software case study of a 1D convolution kernel.

2.1 Hardware/ISA Overview

At the core of the matrix accelerator lies a 4x4 systolic array capable of matrix multiplications in a range of datatypes, including 8-bit, 16-bit, and 32-bit integers, as well as 32-bit IEEE-754 floats. The accelerator stores matrices in an internal register file, holding 8 "tile" registers of 4x4 32-bit values. Given two matrices in tile registers A and B, the systolic array computes $C \mathrel{+}= AB^T$. Operations on datatypes smaller than 32-bit are handled in a SIMD fashion, that is, the systolic array always loads vectors of 4 32-bit elements at a time, but it can be configured (through the use of different instruction mnemonics) to perform multiplications such that the result is that of multiplying 16 8-bit elements, for instance. The output datatype of the systolic array, however, is always 32-bit (float32 for float inputs and int32 for integer inputs).

Load/store instructions enable transfers from main memory to the accelerator's register file. In addition to an address stored in a general purpose register, these instructions take another register

Author's address: Julien de Castelnau, julien.decastelnau@epfl.ch.

corresponding to the row stride in bytes is provided as well, coresponding to the number of bytes between consecutive rows of a 2D matrix. This primarily exists to allow "windowing" of a larger matrix, that is, loading a sub-matrix. Finally, the accelerator provides a way to clear a tile register with zeros using the `mzero` instruction. Table 1 provides a summary of the ISA instructions.

As it is intended for low-power environments, the core inside the X-HEEP is a simple in-order pipeline with a minimal memory hierarchy composed of a configurable number of SRAM banks. No data cache is present. We note that the accelerator can take advantage of the SRAM bank arrangement; memory accesses to different banks may proceed in parallel, thus matrix load and store instructions will be allowed to proceed 4x faster if the accessed data is striped acrosss multiple banks.

The CPU also does not synchronize with the accelerator. The CPU issues instructions in-order, but it does not block on the completion of accelerator instructions, and the accelerator is allowed to retire received instructions out of program order, so long as it honors true dependencies between its own tile registers (for instance, matrix multiplies wait until dependent loads have finished). There is no load-store queue or reorder buffer. Consequently, the CPU is not memory coherent with the accelerator either; the effects of a matrix store may not be immediately visible to a proceeding scalar load, and the load will not be blocked from proceeding in this case, and vice versa. In addition, the accelerator does not have to honor false dependencies, where the result of some instruction may overwrite that of a previous instruction inflight, which either used the previous value or was itself writing to that register.

arithmetic instructions

format	operand type	accumulator type	description
`fmmacc.s md, ms1, ms2`	`fp32`	`fp32`	md += ms1 · ms2^T
`mmasa.w md, ms1, ms2`	`int32`	`int32`	
`mmada.h md, ms1, ms2`	`int16`	`int32`	
`mmaqa.b md, ms1, ms2`	`int8`	`int32`	

load, store, misc

format	description
`mzero md`	clear register `md`
`mld.w md, (rs1), rs2`	load matrix from `rs1` into register `md`, with byte row stride `rs2`
`mst.w ms1, (rs1), rs2`	store matrix in register `ms1` to `rs1`, with byte row stride `rs2`

Table 1. Instruction listing. `mX` = tile register, `rX` = general purpose register.

2.2 Optimized Convolution Routine

This section discusses the optimization of a kernel computing the convolution operator over a 1-dimensional input source for RVM. This kernel apppears frequently in convolutional neural networks, occupying much of the runtime (need a source), thus it represents an important target. First, we will briefly summarize the convolution operation. Given an input stream $I[I_C][N]$ of length N with I_C input channels, and a array of kernels $K[O_C][I_C][W]$ of width W with O_C output channels, we intend to compute an output $O[O_C][N]$ such that

$$O[i][j] = \sum_{c=0}^{I_C} \sum_{r=0}^{W} \begin{cases} I[c][j+r] \cdot K[i][c][r] & \text{if } j+r < N \\ 0 & \text{otherwise} \end{cases}, 0 \le i \le O_C, 0 \le j \le N$$

```
void conv1d_cpu(int32_t *data, int32_t *kernels, int32_t *out) {
    for (int i = 0; i < OC; i++) {
        for (int j = 0; j < N; j++) {
            out[i][j] = 0;
            for (int c = 0; c < IC; c++) {
                for (int r = 0; r < W; r++) {
                    if ((j + r) < N) {
                        out[i][j] += data[c][j + r]
                            * kernels[i][c][r];
                    }
                }
            }
        }
    }
}
```

Listing 1. conv1d scalar implementation

As a baseline, we implement a scalar-only version of this program in C in Listing 1. We assume the dimensions are ordered in the format described above, with the datatypes being int32. Sizes are defined as constants with the preprocessor. Also, the routines are passed pointer types, but we access them using multidimensional array syntax; we have elided the address computation for readability since the strides have been given above.

Throughout the rest of this report, we will also make several assumptions about the sizes of the inputs. Notably, we assume that the kernel size of the convolution is the same as the tile size. This simplifies the boundary conditions for replacing scalar code with accelerator instructions, but it is possible to compensate for by reshaping the input (both if the kernel size is greater or less than the tile size). That said, if this is the case, reshaping the input is unlikely to be an optimal solution, and more care would be needed to optimize this routine around these boundary conditions. We will also assume whenever a loop needs to be split/tiled into smaller factors (e.g. 4 for the size of the tile registers) that there is no remainder. This is handled by once again either padding the input data or adding epilogue loops to handle the remainders.

The first transformation on this routine is to separate the accessing of `data` with the multiply-accumulate. This transformation is common enough it earns a name, "im2col", because it is often critical in running fast convolutions on hardware which performs fast matrix multiplies. The idea is to re-pack the data so that the access pattern of the compute section matches that of a matrix multiply routine, letting us replace the whole loop nest with a call to matmul. We will make a new array, called `y`, which stores the value of `data[c * IW + j + r]` if `j+r` is in bounds, otherwise, it is 0. Then, we replace the access to data with an access to y. Since out of bounds access are 0, which has no effect when added to `out`, we can safely remove the if statement surrounding the original multiply-accumulate. Listing 2 shows the result of this transformation.

With this optimization, we have reduced the problem of convolution to a problem of data movement followed by matrix multiplication.[1] Indeed, we can handle these two sections separately,

[1]There are 4 loops here instead of the typical 3, but note that c and r make up parts of the same contiguous axis. `y` and `kernels` can be treated as 2D matrices of size $N \times (IC \cdot W)$ and $OC \times (IC \cdot W)$ respectively.

```
void conv1d_im2col(int32_t *data, int32_t *kernels, int32_t *out) {
    int32_t y[N][IC][W];
    // perform im2col
    for (int j = 0; j < N; j++) {
        for (int c = 0; c < IC; c++) {
            for (int r = 0; r < W; r++) {
                if ((j + r) < N) {
                    y[j][c][r] = data[c][j+r];
                } else {
                    y[j][c][r] = 0;
                }
            }
        }
    }
    // matrix multiplication
    for (int i = 0; i < OC; i++) {
        for (int j = 0; j < N; j++) {
            out[i][j] = 0;
            for (int c = 0; c < IC; c++) {
                for (int r = 0; r < W; r++) {
                    out[i][j] += y[j][c][r]
                        * kernels[i][c][r];
                }
            }
        }
    }
}
```

Listing 2. conv1d + im2col

and easily create a fast matrix multiplication routine utilizing our instructions. But this would be missing an opportunity: since our matrix instructions do not block, we could overlap the time spent computing results with time spent running scalar code (im2col). Thus, repacking all the data beforehand not only requires more memory to store intermediate results, but also wastes time. Instead, we will only repack the amount of data which the accelerator can load at a time, which is a 4x4 tile. This means that we are computing as much as possible as soon as the data is ready. Listing 3 shows this revised code. Note some subtle changes in this process: the order of loops has been changed from the original program. Before, for each input channel (c), we accumulated one scalar output result. Now, after having tiled the i and j loops, we are accumulating a 4x4 tile.

Now, the operations in the routine correspond nicely with the instructions supported by our accelerator. Instead of performing the 4x4 matrix multiplication on the CPU, we can directly offload this to the accelerator. We can also hold the intermediate result `out[i][j]` until the end of the loop, when we can store the accumulated register to main memory. Listing 4 displays this code. To properly load the subset of the matrices into tile registers, we have used the stride parameter of the

```c
#define TILE 4
void conv1d_im2col_tile(int32_t *data, int32_t *kernels, int32_t *out) {
    for (int tile_i = 0; tile_i < OC/TILE; tile_i++) {
        for (int tile_j = 0; tile_j < N/TILE; tile_j++) {
            for (int c = 0; c < IC; c++) {
                int32_t y[TILE][TILE];
                // perform im2col
                for (int j = 0; j < TILE; j++) {
                    // assumed that W == TILE!
                    for (int r = 0; r < TILE; r++) {
                        if (((tile_j*TILE + j) + r) < N) {
                            y[j][r] = data[c][(tile_j*TILE + j)+r];
                        } else {
                            y[j][r] = 0;
                        }
                    }
                }
                // matrix multiplication
                for (int i = 0; i < TILE; i++) {
                    for (int j = 0; j < TILE; j++) {
                        out[i][j] = 0;
                        for (int r = 0; r < TILE; r++) {
                            out[i][j] += y[j][r]
                             * kernels[tile_i*TILE + i][c][r];
                        }
                    }
                }
            }
        }
    }
}
```

Listing 3. tiled conv1d + im2col

instruction, for instance, when loading kernels, the width of a row is $IC \cdot W$, so we pass $IC \cdot W \cdot 4$ (4 = `sizeof(int32)`).

At this point, the code in listing 4 performs around 4x faster than the scalar code of listing 1. Still, we can further optimize this routine. Profiling the code reveals that the majority of the time is still spent simply doing im2col, and that the computation practically adds nothing to the total runtime, once again due to the nonblocking nature of the instructions. There is ample time for the matrix load and multiply to compute before the im2col loop provides the next piece of data. However, notice that the im2col result is unecessarily computed for every $tile_i$ iteration: the result is not dependent on $tile_i$ at all. If we could reorder the loops and share the value of y for every iteration of $tile_i$, then in theory we may be able to speed up by a factor of up to $\frac{OC}{TILE}$. In reality, since we are reducing over the c loop, we would need to store the tiles for each $tile_i$ in different

```c
#define TILE 4
void conv1d_im2col_tile(int32_t *data, int32_t *kernels, int32_t *out) {
    for (int tile_i = 0; tile_i < OC/TILE; tile_i++) {
        for (int tile_j = 0; tile_j < IW/TILE; tile_j++) {
            asm volatile ("mzero m2");
            for (int c = 0; c < IC; c++) {
                int32_t y[TILE][TILE];
                for (int j = 0; j < TILE; j++) {
                    for (int r = 0; r < TILE; r++) {
                        if (((tile_j*TILE + j) + r) < N) {
                            y[j][r] = data[c][(tile_j*TILE + j)+r];
                        } else {
                            y[j][r] = 0;
                        }
                    }
                }
                // matrix multiplication
                asm volatile ("mld.w m0, (%0), %1"
                    :: "r"(y), "r"(TILE*4));
                asm volatile ("mld.w m1, (%0), %1"
                    :: "r"(&kernels[tile_i*TILE][c][0]), "r"(IC * W * 4));
                asm volatile ("mmasa.w m2, m0, m1");
            }
            asm volatile ("mst.w m2, (%0), %1"
                :: "r"(&out[tile_i*TILE][tile_j*TILE]), "r"(IW * 4));
        }
    }
}
```

Listing 4. tiled conv1d + im2col using accelerator instructions

registers, which is not feasible as we only have 8. But 8 registers is still enough registers to store 4 different $tile_i$ iterations, so we can unroll by a factor of 4. Listing 5 shows this optimization; as expected, it yields around a 4x speedup from the previous iteration.

We can still do slightly better with some final optimizations. The bottleneck is still in the scalar section at this point. We've attempted to do less of it, now we will try to make it go faster. At the innermost loop for im2col (r), we are currently doing a branch for a bounds check. However, as we are only branching on the value to store at `y[j][r]`, we can instead implement this conditional through bitwise logic. This adds slightly more arithmetic instructions, but saves on branches, whose misprediction latency tends to be larger. Next, when accessing arrays, we are always writing out the expression to compute the offset from the indices. This would of course be quite slow if lowered directly by the compiler, but Clang and GCC are capable of generating efficient code for these sorts of circumstances. Still, it turns out that writing out the accumulators lets the compiler save a couple instructions. In all, these tiny differences blow up to saving about 5% of the entire runtime, being at the inside of this loop.

```c
#define TILE 4
void conv1d_im2col_tile(int32_t *data, int32_t *kernels, int32_t *out) {
    for (int tile_i = 0; tile_i < OC/(TILE*4); tile_i++) {
        for (int tile_j = 0; tile_j < IW/TILE; tile_j++) {
            asm volatile("mzero m1");
            asm volatile("mzero m2");
            asm volatile("mzero m3");
            asm volatile("mzero m4");
            for (int c = 0; c < IC; c++) {
                int32_t y[TILE][TILE];
                for (int j = 0; j < TILE; j++) {
                    for (int r = 0; r < TILE; r++) {
                        if (((tile_j*TILE + j) + r) < N) {
                            y[j][r] = data[c][(tile_j*TILE + j)+r];
                        } else {
                            y[j][r] = 0;
                        }
                    }
                }
                // matrix multiplication
                asm volatile("mld.w m0, (%0), %1"
                    :: "r"(y), "r"(TILE * 4));
                asm volatile("mld.w m5, (%0), %1"
                    :: "r"(kernel_base), "r"(IC * KW * 4));
                asm volatile("mmasa.w m1, m0, m5");
                asm volatile("mld.w m6, (%0), %1"
                    :: "r"(kernel_base+TILE * IC * KW), "r"(IC * KW * 4));
                asm volatile("mmasa.w m2, m0, m6");
                asm volatile("mld.w m7, (%0), %1"
                    :: "r"(kernel_base+TILE * IC * KW*2), "r"(IC * KW * 4));
                asm volatile("mmasa.w m3, m0, m7");
                asm volatile("mld.w m5, (%0), %1"
                    :: "r"(kernel_base+TILE * IC * KW*3), "r"(IC * KW * 4));
                asm volatile("mmasa.w m4, m0, m5");
            }
            asm volatile("mst.w m1, (%0), %1"
                :: "r"(&out[tile_i*TILE][tile_j*TILE]), "r"(IW * 4));
            asm volatile("mst.w m2, (%0), %1"
                :: "r"(&out[tile_i*TILE+TILE*1][tile_j*TILE]), "r"(IW * 4));
            asm volatile("mst.w m3, (%0), %1"
                :: "r"(&out[tile_i*TILE+TILE*2][tile_j*TILE]), "r"(IW * 4));
            asm volatile("mst.w m4, (%0), %1"
                :: "r"(&out[tile_i*TILE+TILE*3][tile_j*TILE]), "r"(IW * 4));
        }
    }
}
```

Listing 5. tiled conv1d + im2col using accelerator instructions, with reordering

In this section we have seen firsthand the impacts that simply reorganizing data movement and computation can have. The routines we started and ended with employ arguably the same algorithm, are written in the same programming language, but have vastly different performance characteristics. We can also see the blowup in complexity incurred by these optimizations. What was once 5 lines of for loops has become a mess of tiled loops and inline assembly; at minimum, it is no longer immediately evident what the program is supposed to do. This forms the basis for a key concept in the tools discussed further in this report: the idea that these so-called "scheduling" details can be separated from the algorithm, or the core functionality, of a program. This idea was popularized by the tool Halide, which offered the ability to incrementally rewrite the schedule of some simple algorithmic description, achieving the same performance results as hand-tuned programs while greatly reducing the effort required to develop and debug. TVM, which we will discuss next, furthered Halide's automation capabilities and optimized the workflow towards writing tensor programs for machine learning.

3 TVM

TVM[1] is an end-to-end compiler stack for deep learning workloads. Its primary contributions include a high-level optimizer and compiler for neural networks expressed as dataflow graphs, a collection of DSLs for scheduling individual layers/operators within neural networks, and a runtime for model inference execution. These components are each linked to each other to enable end-to-end compilation, with some notable areas of automation: for example, operators detected in the graph representation of a neural network may be automatically optimized for the target architecture, depending on the target backend in question.

This section will primarily cover TVM from the perspective of the scheduling DSLs within TVM. We will attempt to optimize a generic matrix multiplication routine to use our instructions, as a gentler introduction to TVM's language than looking at convolution. We start with the generic definition of matirx multiplication using 3 loops as expressed in TVM's DSL, called TensorIR, which is a language embedded in Python. Listing 6 shows the code. Most of it is self-explanatory, but we point out several things:

- `T` is the module within which the AST nodes of the DSL are defined. Thus, constructs such as block, grid, Buffer, etc. are accessed using this module.
- This code uses TensorIR; TVM also has an older IR called Tensor Expressions. TensorIR is designed with more features. todo
- `T.Buffer` objects represent the base types for tensors within TensorIR. The function's signature contains the dimension and datatypes of the tensors accessed.
- `T.grid()` corresponds to a tiled loop nest in the order of the arguments. So, `for i,j,k in T.grid(N, N, N):` corresponds to `for i in range(N) { for j in range(N) { ... }}`.
- `T.axis.spatial` and `T.axis.reduce` indicate to TVM the access pattern for some dimension in an array. The first argument is the length of the total access and the second argument is an affine expression of the indices. Reduce is used for axes over which a value is reduced (i.e. summed), while spatial is used when it is not.

The function `matmul` serves as the definition for our computation; from here, we will use TVM's scheduling directives to perform optimizations on the code. At the end, TVM can be used to lower the code to either C or directly to LLVM IR, depending on the backend selected.

To start with the optimizations, we can try to tile the loops, similar to what we did manually for `conv1d`. We want the inner loops to correspond to the size of our accelerator's tiles, 4, so we can use the scheduling directive `sched.split` with 4 as the factor. We perform this for each loop

```python
from tvm.script import ir as I, relax as R, tir as T

N = 128
@T.prim_func
def matmul(
    A: T.Buffer((N, N), "float32"),
    B: T.Buffer((N, N), "float32"),
    C: T.Buffer((N, N), "float32"),
) -> None:
    for i,j,k in T.grid(N, N, N):
        with T.block("update"):
            vi = T.axis.spatial(N, i)
            vj = T.axis.spatial(N, j)
            vk = T.axis.reduce(N, k)
            C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vj, vk]
```

Listing 6. Matmul description in TVM

in the nest. TVM automatically adjusts the indexing expressions in `vi,vj,vk`. Then, we want to reorder the loops so that the inner 3 loops are the ones we just created, so we can replace the loop nest with a call to some set of accelerator instructions. By default, they are in the order we created them (inner and outer loops interleaved). We use `sched.reorder` for this. Listing 7 shows these transformations and the resulting TensorIR printed by TVM.

The routine now has an inner loop of length 4 which corresponds exactly to doing 4x4 matrix multplication on some submatrices of A,B, and C. This scalar code is equivalent to performing matrix loads, a matmul, and then a store on our accelerator. TVM lets us express this equivalence, and gives us the ability to replace the segment, through a directive called `tensorize`. We provide a specification and implementation of our procedure; the former expressed using scalar for loops exactly like our source, and the latter invoking our specialized instructions. If TVM can match the segment of the code we would like to replace with the specification, modulo strides and offsets, then that segment is replaced with the implementation. More precisely, it attempts to unify the provided code block with the body of the specification. Listing 8 shows the invocation of `tensorize`. The implementation routine, `mma_intrin`, instructs TVM to pass the pointers corresponding to the argument buffers to the C function, which we provide for the code for as a string in `matmul_4x4_update`. TVM's compiler simply cuts and pastes this section into the generated C file.

Using `tensorize`, we have successfully transformed the code to make use of our accelerator, and each of the transformation steps we used to get there are explicitly written out in a simple Python-like language. As far as maintainability, this is a definite win over the process we had incrementally rewriting the conv1d C routine. But could we have written a better routine here by hand?

Let's look at the C code generated by TVM (Listing 9). We have a tiled loop nest where the body is a call to our `matmul_4x4_update` function using the instructions. For each invocation of this function, we load A, B, and C from the passed pointers, then multiply-accumulate them, and store the result back to C. It is not necessary to be exchanging C from main memory though; we can hold C in a register during the entirety of the k loop. Ideally, we'd want to load `C` at the start of the

```
from tvm import tir

sched = tir.Schedule(matmul)
i, j, k = sched.get_loops(sched.get_block("update"))
i0, i1 = sched.split(i, factors=[None, 4])
j0, j1 = sched.split(j, factors=[None, 4])
k0, k1 = sched.split(k, factors=[None, 4])
sched.reorder(i0, j0, k0, i1, j1, k1)

print(sched.mod["main"].script())

# Result:
@T.prim_func
def matmul(A: T.Buffer((128, 128), "float32"),
    B: T.Buffer((128, 128), "float32"),
    C: T.Buffer((128, 128), "float32")):
    # with T.block("root"):
    for i_0, j_0, k_0, i_1, j_1, k_1 in T.grid(32, 32, 32, 4, 4, 4):
        with T.block("update"):
            vi = T.axis.spatial(128, i_0 * 4 + i_1)
            vj = T.axis.spatial(128, j_0 * 4 + j_1)
            vk = T.axis.reduce(128, k_0 * 4 + k_1)
            T.reads(C[vi, vj], A[vi, vk], B[vj, vk])
            T.writes(C[vi, vj])
            C[vi, vj] = C[vi, vj] + A[vi, vk] * B[vj, vk]
```

Listing 7. Scheduling directives to tile the loops and reorder the resulting inner loops. Text after `# Result:` is the output of the program (note matmul is defined above).

loop and then store after the result is computed. But the signatures of TVM's functions take *buffers*, which are expected to be in main memory. In fact, TVM doesn't allow the programmer to model the contents of the accelerator memory at all within its IR. These effects are invisible in the code and entirely up to black-box C code like we have used here, or the details of the downstream compiler. We *could* consider creating a "dot product" routine which models the entirety of the k-loop, and tensorize using that instead, but what happens when we want to add code into this loop, like in the case of conv1d, where the loop is with im2col? We would be required to make a variant of dot product with each bit of scalar processing which needs to go in the main loop. And these variants cannot take advantage of TVM's scheduling: by definition, they must be written by hand.

Because TVM's language is not itself concerned with the aspects of the hardware, to regain the flexibility we need to achieve high performance, we are forced to resort to bespoke solutions such as hand-coded microkernels or LLVM compiler passes, all living outside of TVM's purview. Indeed, while TVM is a well-established tool, its codebase is burdened with the maintenance of more than X number of targets. TVM's target system is highly complex and intricate. As a motivating example, most devices are implemented under the umbrella of the LLVM target, which itself has many variants with custom codegen passes for different architectures, including x86, ARM, etc. More

```
def matmul_4x4_update():
    return """
    extern "C" int matmul_4x4_update(float *cc,float *aa,float *bb,int32_t stride) {
    asm volatile("mld.w m0, (%0), %3\\n\\r"
        "mld.w m1, (%1), %3\\n\\r"
        "mld.w m2, (%2), %3\\n\\r"
        "fmmacc.s m2, m1, m0\\n\\r"
        "mst.w m2, (%2), %3\\n\\r"
        :: "r"(aa), "r"(bb), "r"(cc), "r"(stride << 2));
    return 0; } """

@T.prim_func
def mma_spec(a: T.handle, b: T.handle, c: T.handle) -> None:
    A = T.match_buffer(a, (4, 4), align=64, offset_factor=1)
    B = T.match_buffer(b, (4, 4), align=64, offset_factor=1)
    C = T.match_buffer(c, (4, 4), align=64, offset_factor=1)

    with T.block("root"):
        T.reads(C[0 : 4, 0 : 4], A[0 : 4, 0 : 4], B[0 : 4, 0 : 4])
        T.writes(C[0 : 4, 0 : 4])
        for i, j, k in T.grid(4, 4, 4):
            with T.block("update"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                C[vi, vj] =  C[vi, vj] + A[vi, vk] * B[vj, vk]

@T.prim_func
def mma_intrin(a: T.handle, b: T.handle, c: T.handle) -> None:
    s0 = T.int64()
    s1 = T.int64()
    A = T.match_buffer(a, (4, 4), align=64, offset_factor=1, strides=[s0,s1])
    B = T.match_buffer(b, (4, 4), align=64, offset_factor=1)
    C = T.match_buffer(c, (4, 4), align=64, offset_factor=1)

    with T.block("root"):
        T.reads(C[0 : 4, 0 : 4], A[0 : 4, 0 : 4], B[0 : 4, 0 : 4])
        T.writes(C[0 : 4, 0 : 4])
        T.call_extern("int", "matmul_4x4_update",
            C.access_ptr("w"), A.access_ptr("r"), B.access_ptr("r"), s0)

TensorIntrin.register("test_mma_intrin", mma_desc, mma_intrin)
# 'i1' refers to for i1 in .. loop
sched.annotate(sched.get_block("root"), "pragma_import_c", matmul_4x4_update())
sched.tensorize(i1, "test_mma_intrin")
```

Listing 8. Matrix multiply spec and implementation prim_func definitions with `tensorize()` invocation. Result has been omitted for brevity. Note that `T.axis.remap` is syntactic sugar for `T.axis.spatial`, `T.axis.reduce`, etc.

```
TVM_DLL int32_t matmul(void* args, int32_t* arg_type_ids, int32_t num_args,
 void* out_ret_value, int32_t* out_ret_tcode, void* resource_handle) {
  /* code to unpack tensor pointers from argument structs ommitted
     A,B,C are simply float pointers */
  for (int32_t i_0 = 0; i_0 < 32; ++i_0) {
    for (int32_t j_0 = 0; j_0 < 32; ++j_0) {
      for (int32_t k_0 = 0; k_0 < 32; ++k_0) {
        int32_t cse_var_2 = (k_0 * 4);
        int32_t cse_var_1 = (i_0 * 512);
        matmul_4x4_update((&(((float*)C)[(cse_var_1 + (j_0 * 4))])), (&(((float*)A)[(
        cse_var_1 + cse_var_2)])), (&(((float*)B)[((j_0 * 512) + cse_var_2)])), 128);
      }
    }
  }
  return 0;
}
```

Listing 9. Output C code from running matmul example on TVM.

obscure targets such as a Vulkan backend for GPUs use their own backend. Yet, ARM still has its own target in TVM to utilize CMSIS-NN, a set of hand-optimized C kernels which tend to perform faster on ARM hardware than the schedules generated by TVM + LLVM.

In a sense, this lack of expressibility defeats the advantage a user-scheduled language can offer: the schedule of the program becomes an incomplete picture of its performance, diminishing its utility. The custom compiler passes or hand-tuned microkernel routines instead become the focal point to ensure optimal performance on hardware. The next two tools we look at, while disparate in their approaches and goals, both avoid this expressibility problem by offering the programmer near complete control over the generated code, while maintaining the highly structured aspects of a user-scheduled language. Our next language in particular, Exo, offers explicit constructs for managing the state of hardware accelerators.

4 EXO

Exo [2] is a user-scheduled language implemented as a DSL embedded in Python, similar to TVM. Exo expands upon languages like Halide and TVM with the concept of *exocompilation*: instead of deferring to the compiler for critical decisions on instruction selection, accelerator memory management, etc., the programmer is given control. Unlike TVM, Exo has no backends or targets. All Exo provides is a language with a standard set of scheduling directives, and constructs for the programmer to express their own hardware in a library which can be used by the schedule. In this section, we will see how Exo can be applied in the practical setting of our convolution routine.

To introduce Exo, we will start by expressing the “direct translation" convolution routine from Listing 1. This will serve as our algorithm, which we will continually rewrite to improve performance. Listing 10 shows the code.

The structure closely resembles that of a real Python program, but like TVM, each construct in the code is compiled down to a representation within the Exo language. Now, we will apply scheduling directives to optimize it. We’ll follow a slightly different order from the transformations

```
# TODO: FIX THESE INDICES!
@proc
def generic_conv1d(
    data: i32[C, W],
    kernels: i32[K, C, R],
    out: i32[K, W],
):
    # do the convolution
    for k in seq(0, K):
        for i in seq(0, W):
            # zero out the result memory
            out[k, i] = 0.0
            for c in seq(0, C):
                for r in seq(0, R):
                    y: i32
                    if i + r < W:
                        y = data[c, i + r]
                    else:
                        y = 0
                    out[k, i] += kernels[k, c, r] * y
```

Listing 10. Base algorithm definition in Exo.

applied by hand, as a certain order makes it easiest to express. Namely, we'd like to do all of our tiling beforehand, so we start with those directives:

```
# Before scheduling, grab cursors to the object code.
k_loop = p.find("for k in _:_")
i_loop = p.find("for i in _:_")
c_loop = p.find("for c in _:_")
y_alloc = p.find("y : _")
y_assign = p.find("y = data[_]")

# Tile outer loops to TILE size for RVM
p, _ = tile_loops(p, [(k_loop, TILE), (i_loop, TILE)], perfect=True)
p, _ = tile_loops(p, [(k_loop, 4)], perfect=True)
k_loop_reg = p.find("for koi in _:_")
p = reorder_loops(p, k_loop_reg)
```

In this snippet and those that come, `p` refers to the `generic_conv1d` function defined above, passed as an object to this function which manipulates the schedule.

Exo's scheduling directives need "cursors" that point to places in the program to manipulate. You can provide these as strings which are pattern matched against parts of the AST at the point the directive is applied, or grab a cursor to that node as an object, which is carried throughout transformations. Here we grab these references to the parts of the code we want to transform

for convenience. Next, we apply a directive called `tile_loops`, similar to `split` from TVM, to tile these loop nests by the factor of the tile size. We tile once again to handle the 4x compute optimization. Finally, we reorder the loop corresponding to the 4 registers which we will unroll later.

If we print `p` at this point using Exo, we get the following output:

```
def exo_conv1d_tile_lt_kw(data: i32[4, 16] @ DRAM,
                          kernels: i32[16, 4, 4] @ DRAM,
                          out: i32[16, 16] @ DRAM):
    for koo in seq(0, 1):
        for io in seq(0, 4):
            for koi in seq(0, 4):
                for ki in seq(0, 4):
                    for ii in seq(0, 4):
                        out[ki + 4 * (4 * koo + koi), ii + 4 * io] = 0.0
                        for c in seq(0, 4):
                            for r in seq(0, 4):
                                y: i32 @ DRAM
                                if ii + r + 4 * io < 16:
                                    y = data[c, ii + r + 4 * io]
                                else:
                                    y = 0
                                out[ki + 4 * (4 * koo + koi),
                                    ii + 4 * io] += kernels[ki + 4 *
                                                            (4 * koo + koi), c,
                                                            r] * y
```

Now we would like to reorder the `c` output to surround the inner tile dimensions, since inside each iteration of `c` we'd like to do a single tile-size im2col and matrix multiplication. Exo won't let us reorder the loop with the initialization of `out[]` in the way though, so we will separate them out first. But we also want to stage `out[]` to a new buffer, since the compute loop is only operating on the accelerator's memory, not directly to `out[]` in the function signature. We will see later how to express that this buffer corresponds to the register held in our accelerator.

```
# Stage output to out_tile
p, (_, out_tile, body, _) = auto_stage_mem(
    p, p.find_loop("c").expand(1, 0), "out", "out_tile", rc=True
)
p = autolift_alloc(p, out_tile, max_size=4 * 4 * 4, dep_set=["koi","ki","ii"])

# Block the zero initialization and store blocks
p = fission_as_much_as_possible(p, body)
p = fission_as_much_as_possible(p, body[0])

# Reorder c loop to the top
p = lift_scope_n(p, c_loop, 3)
```

Exo provides a function `auto_stage_mem` to handle the staging of out into a new memory. Here, we are describing the name of the new variable and in which block of code we want to replace the accesses to the original memory. Exo auto-inserts code to copy over the contents of the staged memory back to the original once the loop terminates. Next, we'd like to lift the newly generated allocation for `out_tile` so that it spans 4 tile registers, and then we can finally reorder the loops so that the tile axes are on the inside. Exo provides a directive in its standard library called `lift_alloc`, which moves the allocation out of a loop (this transformation is always correct), and another called `expand_dim` which adds another dimension to a buffer dependent on some index variable. We can easily combine these for each index to achieve the result we want, but Exo lends itself well to composing these rules for concise code. In this case, we have defined a custom rewrite for automatically applying this `lift_alloc-expand_dim` process, until we have traversed enough parent loops to reach some threshold size of the new buffer:

```
def autolift_alloc(p, alloc_c, dep_set=None, max_size=0, lift=True):
    """
    for i in seq(0, 10):
        for j in seq(0, 20):
            a : R          <- alloc_c, dep_set = {'i'}
            a[i] = ...
    ---->
    a : R[10]              <- if size is less than max_size
    for i in seq(0, n):
        for j in seq(0, m):
            a[i] = ...
    """
    alloc_c = p.forward(alloc_c)
    loop_c = get_enclosing_loop(p, alloc_c)
    accum_size = 1
    while True:
        try:
            if not isinstance(loop_c, pc.ForCursor):
                break
            if dep_set == None or loop_c.name() in dep_set:
                if (
                    isinstance(loop_c.hi(), LiteralCursor)
                    and accum_size * loop_c.hi().value() <= max_size
                ):
                    p = expand_dim(p, alloc_c, loop_c.hi().value(), loop_c.name())
                    accum_size = accum_size * loop_c.hi().value()
                    if lift:
                        p = lift_alloc(p, alloc_c)
            loop_c = loop_c.parent()
        except:
            break
    return p
```

After applying `autolift_alloc`, we can fission (i.e. separate the statements into their own loops) the new blocks for zeroing and storing `out_tile`, as these do not depend on the `c` loop. Finally, we reorder the `c` loop so that is on the outside as we intended. Note that these functions (`fission_as_much_as_possible`, `lift_scope_n`) are also custom transformations composed from functions given by Exo. Printing `p` now, we get

```
def exo_conv1d_tile_lt_kw(data: i32[4, 16] @ DRAM,
                          kernels: i32[16, 4, 4] @ DRAM,
                          out: i32[16, 16] @ DRAM):
    for koo in seq(0, 1):
        for io in seq(0, 4):
            out_tile: i32[4, 4, 4] @ DRAM
            for koi in seq(0, 4):
                for ki in seq(0, 4):
                    for ii in seq(0, 4):
                        out_tile[koi, ki, ii] = 0.0
            for c in seq(0, 4):
                for koi in seq(0, 4):
                    for ki in seq(0, 4):
                        for ii in seq(0, 4):
                            for r in seq(0, 4):
                                y: i32 @ DRAM
                                if ii + r + 4 * io < 16:
                                    y = data[c, ii + r + 4 * io]
                                else:
                                    y = 0
                                out_tile[koi, ki,
                                         ii] += kernels[ki + 4 *
                                                        (4 * koo + koi), c,
                                                        r] * y
            for koi in seq(0, 4):
                for ki in seq(0, 4):
                    for ii in seq(0, 4):
                        out[ki + 4 * (4 * koo + koi),
                            ii + 4 * io] = out_tile[koi, ki, ii]
```

We still have to apply the im2col transformation, where we separate out the setting of `y` into its own loop nest, making `y` a large buffer in the process. We can see that this is a simple matter of applying fission between setting `y` and doing the multiply-accumulate, then lifting the `y` allocation up the loop nest. Afterwards, we stage the kernel and data matrices into new buffers just like with the output. We have:

```
# Stage y
p = autolift_alloc(p, y_alloc, max_size=4 * 4, dep_set=["r","ii"])
p = lift_alloc(p, y_alloc, n_lifts=2)

# Fission the initialization loop and remove redundant loops
p = fission_as_much_as_possible(p, y_assign.parent())
p = remove_redundant_loops(p, y_assign.parent(), num=2)

# Stage kernels to kernel_tile and y to data_tile
ki_loop = p.forward(c_loop).body()[2].body()[0]
p, (kernel_alloc, _, _, _) = auto_stage_mem(
    p, ki_loop, "kernels", "kernel_tile", rc=True
)
p = simplify(expand_dim(p, kernel_alloc, 4, ki_loop.parent().name()))
p = lift_alloc(p, kernel_alloc)
p, (data_alloc, _, _, _) = auto_stage_mem(
    p, ki_loop.parent(), "y", "data_tile", rc=True
)
```

We ommitted the result of the code, since we have reused many of the same constructs as previously.

We are now ready to replace the loops in this code with offloads to our accelerator. For this purpose, Exo offers a code replacement feature, `replace()`, like TVM's tensorize. Like tensorize, strides and offsets are automatically handled when unifying against the function body. But Exo's implementation differs in 2 key ways:

(1) The functions which provide the "specification" may take an argument which is not in main memory. Whereas TVM's functions had to take types of `T.Buffer`, an arbitrary annotation for the location (e.g. `@ DRAM`) is permitted. Combined with Exo's ability to express custom memories, this allows for fine-grain control over which parts of the code are replaced, rather than having to replace all accelerator code at once.
(2) The replacement implementation is not limited to invoking a C function or pasting LLVM IR. A piece of C code may be inserted directly. This works once again in tandem with Exo's custom memory support to allow, for example, implementations consisting of inline assembly using parameterized registers.

`replace()` takes as arguments the procedure to replace, the segment of the code within that procedure to match, and the function whose body will be unified with the code selected. Let's see how the functions corresponding to the instruction specifications are written:

```
@instr(
    'asm volatile("mld.w "{dst_int}", (%1), %0"\
    :: "r"(4*({src}.strides[0])), "r"(&{src_data}));'
)
def rvm_mld(dst: [i32][4, 4] @ RVM_TILE, src: [i32][4, 4] @ DRAM):
    assert stride(src, 1) == 1
    assert stride(dst, 1) == 1

    for i in seq(0, 4):
        for j in seq(0, 4):
            dst[i, j] = src[i, j]
```

The arguments for load are of type RVM_TILE for the destination, corresponding to our register file, and the source is an address in main memory. The function body is nothing more than standard Exo code, whose semantics can be understood without knowing the accelerator's details. Here, we are just copying from one array to the next. But what is RVM_TILE and how did we define it?

```
class RVM_TILE(StaticMemory):
    NUM_RVM_TILES = 8
    StaticMemory.init_state(NUM_RVM_TILES)
    tile_dict = {}

    ...

    @classmethod
    def alloc(cls, new_name, prim_type, shape, srcinfo):
        if not (shape[0].isdecimal() and int(shape[0]) == 4):
            raise MemGenError("Number of tile rows must be 4.")
        if not (shape[1].isdecimal() and int(shape[1]) == 4):
            raise MemGenError("Number of tile columns must be 4.")

        tile_num = cls.find_free_chunk()
        cls.mark(tile_num)
        cls.tile_dict[new_name] = tile_num
        return f'#define {new_name} "m{7-tile_num}"'

    @classmethod
    def free(cls, new_name, prim_type, shape, srcinfo):
        tile_num = cls.tile_dict[new_name]
        del cls.tile_dict[new_name]
        cls.unmark(tile_num)
        return f"#undef {new_name}"
```

Exo allows us to define our custom memories as custom classes which we can invoke as seen above. Semantically, these classes are all viewed as residing in global, dynamically allocated memory.

Thus, a directive `set_memory` exists to freely change the selected memory for a specific allocation in a program. The difference comes down to the code generation phase, when the Exo program is lowered to C: the implementations of the methods in this class dictate how to allocate a buffer, how to free it, etc. In our case, "allocation" is simply a matter of selecting the right register. For this we just need to define a string literal which we can use in our inline assembly statements. In our implementation, the `alloc()` function tracks the allocated registers for each named buffer in a dictionary and maintains a bitmap of the free registers. Even though we have called it a memory allocator, what we have essentially implemented here is a trivial register allocator, something only a compiler would typically have control of! But this class is simply defined in our Python module, and does not reside in any sort of compiler backend for Exo.

Going back to the instructions, we define others in a similar fashion. For example, here is the definition for our matrix multiplication:

```
@instr('asm volatile("mmasa.w "{md_int}", "{ms1_int}", "{ms2_int});')
def rvm_mmasa(
    md: [i32][4, 4] @ RVM_TILE, ms1: [i32][4, 4] @ RVM_TILE,
     ms2: [i32][4, 4] @ RVM_TILE
):
    assert stride(md, 1) == 1
    assert stride(ms1, 1) == 1
    assert stride(ms2, 1) == 1
    for i in seq(0, 4):
        for j in seq(0, 4):
            for k in seq(0, 4):
                md[i, j] += ms2[i, k] * ms1[j, k]
```

You can see once again that the specifications here are nothing more than scalar code, like how we'd write unoptimized code for the CPU. The `instr` decorator provides the implementation where we invoke inline assembly. The parameters `md_int`, `ms1_int`, etc. are filled in by Exo based on the name of the variable passed to the function.

To use these instructions, we apply `set_memory` and `replace` as follows (`replace_all` is a wrapper around replace that applies it wherever applicable):

```
# Set adequate memories
p = set_memory(p, y_alloc, DRAM_STATIC)
p = set_memory(p, out_tile, RVM_TILE)
p = set_memory(p, kernel_alloc, RVM_TILE)
p = set_memory(p, data_alloc, RVM_TILE)

# Replace inner loops to calls to RVM instructions
p = replace_all(p, [rvm_mzero, rvm_mst, rvm_mld, rvm_mmasa])
```

The resulting Exo code has all offloadable sections replaced by calls to the functions we wrote earlier. Once again, from the Exo perspective, these are just calls to other Exo functions, but they carry a special meaning during code generation which allows them to represent our assembly instructions.

```
def exo_conv1d_tile_lt_kw(data: i32[4, 16] @ DRAM,
                          kernels: i32[16, 4, 4] @ DRAM,
                          out: i32[16, 16] @ DRAM):
    for koo in seq(0, 1):
        for io in seq(0, 4):
            out_tile: i32[4, 4, 4] @ RVM_TILE
            for koi in seq(0, 4):
                rvm_mzero(out_tile[koi + 0, 0:4, 0:4])
            for c in seq(0, 4):
                y: i32[4, 4] @ DRAM_STATIC
                for ii in seq(0, 4):
                    for r in seq(0, 4):
                        if ii + r + 4 * io < 16:
                            y[ii, r] = data[c, ii + r + 4 * io]
                        else:
                            y[ii, r] = 0
                kernel_tile: i32[4, 4, 4] @ RVM_TILE
                data_tile: i32[4 - 0, 4 - 0] @ RVM_TILE
                rvm_mld(data_tile[0:4, 0:4], y[0:4, 0:4])
                for koi in seq(0, 4):
                    rvm_mld(
                        kernel_tile[koi + 0, 0:4, 0:4],
                        kernels[4 * koi + 16 * koo + 0:4 * koi + 16 * koo + 4,
                                c + 0, 0:4])
                    rvm_mmasa(out_tile[koi + 0, 0:4, 0:4], data_tile[0:4, 0:4],
                              kernel_tile[koi + 0, 0:4, 0:4])
            for koi in seq(0, 4):
                rvm_mst(
                    out_tile[koi + 0, 0:4, 0:4],
                    out[4 * koi + 16 * koo + 0:4 * koi + 16 * koo + 4,
                        4 * io + 0:4 * io + 4])
```

For our final transformations, we'd like to unroll each of the `koi` loops, and allocate 4 different `out_tiles`, rather than having only one with an extra dimension. Exo provides us with a routine to unroll loops, but also to "unroll the buffer": replace an allocation for a constant size n on a given dimension with n buffers without that dimension. We utilize these directives below.

```
# Clean up
p = unroll_loop(p, "koi")
p = unroll_loop(p, "koi")
p = unroll_loop(p, "koi")
p = simplify(p)
p = unroll_buffer(p, kernel_alloc, 0)
p = reuse_buffer(p, "kernel_tile_0: _", "kernel_tile_3: _")
```

At this point, using Exo, we're at about 95% of the performance (in terms of 1/execution time) of our manual routine. How can we express the remaining 5%? When writing originally, we added some optimizations for the indexing expressions in the im2col loops, since it was generating better code with Clang. How can we control this from Exo? Indexing expressions in Exo itself are represented as abstract multidimensional array lookups, and are lowered to the real expressions when the C code is generated. While Exo gave us great flexibility in instruction selection, it does not offer us control over this aspect of compilation. We've ended up in a situation similar to TVM: we are in a position of having to trust a compiler to generate good code in a later phase, since although we know of an optimization, we cannot schedule the language where we could apply it.

While the example of indexing expressions is particularly salient in our case, other limitations may manifest from Exo's design. Exo thrives with workloads dominated by compute-heavy for loops with simple control, usually dictated by affine expressions. But the benefits of a scheduling language need not be limited to these workloads. We write many programs where a simple expression of the algorithm exists, but we ship a vastly more complicated variant to maximize performance. Such programs could benefit from a user-scheduled languages, but as they may be written by hand to make use of features such as structs which Exo currently has no answer to. Transformations on these features exist too, such as turning an array of structs into a struct of arrays, typically used to increase locality.

These limitations bring us to our final tool: OptiTrust. OptiTrust foregoes a DSL and allows the user to perform source-to-source transformations on C and C++ code. We will see how OptiTrust can be applied in our case study, specifically to make the indexing optimization that we were not able to with Exo.

5 OPTITRUST

OptiTrust is a tool allowing for source-to-source transformations of C/C++ code. These transformations, similar to the scheduling rules we have seen previously, are driven by an OCaml script which manipulates parts an AST corresponding to the program. OptiTrust's primary goal is not just to produce fast software, but to produce *correct* fast software. OptiTrust thus places a large focus on performing optimizations that preserve the invariants of the source.

In this section we will demonstrate OptiTrust on the 1D convolution routine, attempting to apply the same process as we did with Exo, and expanding on it as well. We will start with the C code for direct convolution, which is that of Listing 1. In our OCaml script for OptiTrust, we write a function of type `unit -> unit` (so, it has side effects) which applies the various scheduling functions. OptiTrust provides a number of these and they work similar those we have seen in other tools. OptiTrust's code selection system works on a constraint system; all directives take in a *target* to represent the location(s) in the code desired to be transformed, which are lists of constraints matching against parts of the AST. For example, the target `[cFor "i"; occLast]` matches the AST node for the last occurence of a for loop statement with iteration index "i". Transformation functions take in targets and any other paremeters and return `unit`, manipulating the AST as a side effect. The code in Listing 11 shows the segment of the script used to tile the loops at the start, like we did in Exo.

Many of the familiar constructs such as tiling, loop reordering, loop fission, etc. exist in OptiTrust, as we see in Listing 3. Like Exo, we can compose the provided rewrite rules to make our own, like the convenience function `prefix_indices` which allows for bulk renaming of loop indices. OptiTrust lets us go a step further here, however. The user can write transformations which manipulate the AST itself. This gives quite extensive control. We will now see an instance where this control becomes useful.

```
(* tile according to size supported by accelerator *)
!! Loop.tile (trm_int 4) ~index:"tile_i"
 ~iter:TileIterGlobal ~bound:TileDivides [cFor "i"];
!! Loop.tile (trm_int 4) ~index:"tile_j"
 ~iter:TileIterGlobal ~bound:TileDivides [cFor "j"];
!! Loop.reorder ~order:["tile_j"; "i"] [cFor "i"];
(* tile again to have 4x compute *)
!! Loop.tile (trm_int 4) ~index:"tile_i_hi"
 ~iter:TileIterGlobal ~bound:TileDivides [cFor "tile_i"];
!! Loop.reorder ~order:["tile_j"; "tile_i"] [cFor "tile_i"];

(* sum a tile at a time *)
!! Loop.hoist_alloc_loop_list [1; 1; 1;] [cVarDef "sum"];
(* zeroing *)
!! Loop.fission ~nest_of:3 [tBefore; cFor "k"; occFirst];
(* storing *)
!! Loop.fission ~nest_of:3 [tAfter; cFor "k"; occFirst];

let prefix_indices (idxs: (string * target) list) (pfx: string): unit =
let _ = (List.map (fun idx ->
    Loop.rename_index (pfx ^ "_" ^ fst idx) (snd idx)) idxs) in () in
!! prefix_indices [("tile_i", [cFor "tile_i"; occFirst]);
 ("i", [cFor "i"; occFirst]); ("j", [cFor "j"; occFirst])] "zero";
!! prefix_indices [("tile_i", [cFor "tile_i"; occLast]);
 ("i", [cFor "i"; occLast]); ("j", [cFor "j"; occLast])] "st";
```

Listing 11. OCaml script to tile conv1d in OptiTrust. Each line beginning with `!!` is a scheduling directive whose effects can be viewed in a trace.

Listing 12 shows the code for offloading loops to the accelerators. We also include the staging of memory which we accomplish using a combination of `Variable.bind`, `hoist_alloc`, and `fission`. For the replacement parts, in Exo, we did this using `replace`, which offered unification between code blocks and function bodies. We use a similar strategy by employing `uninline` from OptiTrust. We define a function that gives the specification of the accelerator instruction (i.e. the code block we'd like to match against) and use `uninline` to replace some code with a call to that function. Then, we can use another OptiTrust directive to replace this function call with another that invokes inline assembly, which serves as the implementation. We will deal with the implementation side shortly. For now, let's see how we perform replacement over our instruction specification.

The first roadblock we would run into trying to use uninline is that of *windowing*. That is, the code we are replacing is indexing into some offset of a large matrix (possibly with more dimensions), while the code we are replacing with assumes the passed pointer is the start of a 4x4 matrix. We have to account for the stride and offset, which TVM and Exo do behind the scenes as this problem is extremely common in their domains. OptiTrust, on the other hand, does not, and so unification would fail. However, this is where we can employ OptiTrust's flexibility: We can define our own rules to handle this windowing use case. When we encounter an indexing expression for an n

```
(* Data Load *)
!! Variable.bind "data_tile" [cArrayRead "y"];
Loop.hoist_alloc_loop_list [0; 0; 1; 1;] [cVarDef "data_tile"];
Loop.fission ~nest_of:4 [tAfter; cArrayWrite "data_tile"];

!! delete_loop [cFor "tile_i"; occFirst] [cFor "i"; occFirst];
delete_loop [cFor "i"; occFirst] [cFor "j"; occIndex 1];

!! Function.uninline ~fct:[cFunDef "rvm_mld"] [cFor "j"; occIndex 1];

(* Kernel Load *)
!! Variable.bind "kernel_tile" [cArrayRead "W"];
Loop.hoist_alloc_loop_list [1; 0; 1;] [cVarDef "kernel_tile"];
Loop.fission ~nest_of:3 [tAfter; cArrayWrite "kernel_tile"];
!! delete_loop [cFor "j"; occIndex 1] [cFor "r"; occIndex 1];

!! reduce_mindex 0 [cFun "MINDEX3"; occIndex 1];
!! factor_window ["i"; "r"] [cMindexAcc "W";];
!! Function.uninline ~fct:[cFunDef "rvm_mld"] [cFor "i"; occIndex 0];
```

Listing 12. Staging and offloading of data and kernel loads.

dimensional array, represented in OptiTrust by a special `MINDEX` function, we can rewrite this as an index into an $n-1$ dimensional array where two of the axes have been "folded" into one. For example, if the lowest axis of a 3x3x3 matrix is folded, then we have a 3x9 matrix. We call this rule `reduce_mindex`. From here we can rewrite the indexing only in terms of the indices which change in the loop. Any terms added to these indices are extracted out and rewritten as a pointer with the offset added. This is `factor_window`. When we apply these, the index expression matches exactly that of the function assuming a 4x4 matrix, thus we are able to uninline.

We continue replacing segments of the code with offloads to the accelerator using uninline, until we have theoretically landed at the same point we left the Exo program. We can turn our attention to the original question prompting writing directly on C - can we apply our indexing optimization? Since OptiTrust is extensible, we can once again introduce new scheduling functions:

```
!! Variable.bind_multi ~dest:[cIf (); tBefore] "ofs" [sExpr "tile_j * 4 + j + r"];
!! Matrix_basic.elim_mindex [cMindex ~args:[[cVar "IC"]; [cVar "IW"];
 [cVar "k"]; [cVar "ofs"]] ()];
!! Matrix_basic.elim_mindex [nbMulti; cMindex ~args:[[cInt 4]; [cInt 4];
[cVar "j"]; [cVar "r"]] ()];

!! loopize_decl [cFor "r"; occFirst] [cVarDef "ofs"];
!! Arith_basic.simplify [sExpr "0 + k * IW + ofs"];
!! Variable.bind "drow_ofs" [sExpr "k * IW + ofs"];
!! hoist_if [cIf ()] [cVarDef "drow_ofs"];
!! loopize_decl ~loop_ind_in:(find_var_in_current_ast "ofs")
 [cFor "r"; occFirst] [cVarDef "drow_ofs"];
!! loopize_expr [cFor "tile_j"] "data_base" [sExpr "tile_j * 4"];
!! loopize_expr [cFor "k"] "data_row" [sExpr "k * IW"]
```

This section accomplishes the transformation by binding the indexing expression to a new variable, `ofs`, expanding the calls to `MINDEX` to mathematical expressions, and then applying a custom transformation dubbed `loopize` to remove the dependence on the index so that the new variable can be accumulated itself. Continuing this way we are able to rewrite the expression exactly in the manner carried out by hand. Although these rules may be quite tailored to this use case, and may fail to apply in other circumstances where our assumptions are violated, what we still gain over doing it manually is that the *intent* of the optimizations can be made explicit. Through judicious composition of simple rules, we can create rules that succinctly capture the original intent in code itself, without extensive comments or documentation. The important aspect of capturing intent is its relevance in building auditable software, which is key to building correct software.

So far, OptiTrust has provided a compelling answer to the inherent shortcomings of DSL based systems like TVM and Exo. Whenever we have been unable to express a transformation with OptiTrust's standard library, we can make one of our own. We can expect to run into few language-level limitations this way, having nearly the full C language at our disposal. But we have not yet addressed how our accelerator code generation will look. We set aside this problem, representing accelerator instructions as C functions, which only take buffers in memory. Yet we know from TVM that this interface works poorly to represent instructions: we are actually interfacing with buffers that live in the accelerator.

It turns out there is a bigger problem, however. It is not possible to write inline assembly at all in OptiTrust. There are a number of C features which OptiTrust doesn't currently support, like function pointers, but inline assembly is slightly different. There is no C specification for the behavior of inline assembly. How does a tool like OptiTrust perform analysis a priori on some code utilizing inline assembly? This suggests that the features needed for high-performance programming in use cases like ours need more specification than merely the that of the language we write in (C). That is, this discipline of writing code for custom hardware employs behavior which is only specified in the ISA (which may not even be familiar to the compiler, like is the case here), and thus to do any of the meaningful static analysis we'd like with these constructs, we need some kind of corresponding specification which must be given by the user. As OptiTrust lacks this currently, we are unable to use its scheduling capabilities to use our hardware.

6 CONCLUSION

We have seen through the development of our convolution routine some important challenges in the discipline of high performance software development. Firstly, we saw the blowup in complexity caused by optimizing software: the representation we had which clearly correlated with the mathematical formula was lost when we made it run efficiently on our hardware. With nearly 5x the code length due to this process, we can see how writing high performance code can be time consuming to develop, debug, and maintain, especially as the source algorithm scales in complexity. Secondly, we know there is a serious penalty to overhead in these circumstances: as we build complex software (such as neural networks with billions of parameters) with these routines at the base, small differences in performance characteristics are amplified at the high level. Thirdly, we saw how necessary it was to have control over the low-level details of this software, sometimes down to the assembly level, to be able to program efficiently for the hardware. When this control cannot be exercised it may introduce an overhead which quickly becomes unacceptable.

We saw how user-scheduled tools like TVM, Exo, and OptiTrust make huge strides in improving the development and maintainability of these sorts of programs. By separating concerns between algorithms and optimizations, we do not have to prove the correctness of an optimized routine, the complexity for which we saw, but only that we are applying correct-in-context transformations on a program with a well-understood correctness criterion. As explicit *processes* to manipulate the algorithm itself, the optimizations we apply can also be made to be easily auditable.

Still, user-scheduled languages must contend with the remaining challenges posed in this space, namely the need for low-overhead control over hardware details. We saw with TVM how stopping short of explicit control over the accelerator instructions greatly reduced the benefit of the tool, as some of the most important details to the program had to be relegated to a program written by hand anyway. In response to these problems Exo provided a system for the programmer to better control instruction selection from the schedule itself, but we saw it eventually succomb to the same lack of expressibility under a niche, but relevant optimization case. We saw OptiTrust, a system which foregoes a domain specific language and opts for C-based source-to-source rewriting, handle these missed cases gracefully by allowing the programmer to express their own rewritings of the program's AST. Despite having nearly the full C language at our disposal, we were still limited in our ability to express our hardware-specific optimization as a verifiable rewrite in OptiTrust, as the semantics we relied on exist in the hardware, not in the language itself. To replace handwritten code, user-schedulable languages must deal with the common need to work with poorly-specified hardware constructs, and a wide range of generic language features, but need to grapple with performing meaningful static analysis.

REFERENCES

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[2] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[3] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744. https://doi.org/10.1126/science.aam9744 arXiv:https://www.science.org/doi/pdf/10.1126/science.aam9744

[4] Pasquale Davide Schiavone, Simone Machetti, Miguel Peon Quiros, José Angel Miranda Calero, Benoît Walter Denkinger, Christoph Thomas Müller, Rubén Rodríguez Álvarez, Saverio Nasturzio, and David Atienza Alonso. 2023. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller. *Proceedings Of The 20Th Acm International Conference On Computing Frontiers 2023, Cf 2023*, 379–380. https://doi.org/10.1145/3587135.3591431

GLOSSARY

scalar or "scalar section", is used in this document to refer to code which computes using scalar (single value) operations, like those found in basic CPU integer instructions, as opposed to SIMD, vector, or matrix machines

stride defined on a given dimension of a multidimensional array, the space between consecutive indices