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[2] 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[3]. 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

```
#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 final optimization; as expected, it yields around a 4x speedup from the previous iteration.

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

```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

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 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

```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

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 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

```
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).

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 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.

```
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.

4 EXO

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] 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

[3] 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