Lab 3: Tiling and Optimization for Accelerators

EE 290-2 Hardware for Machine Learning
UC Berkeley, Spring 2020
Instructor: Prof. Sophia Yakun Shao
Teaching Assistants: Alon Amid and Hasan Genc

Due: March 6, 2020

Contents

1 Introduction

This lab will provide you with hands-on experience on the implications of mapping large matrix multiplication operations onto 2D systolic array accelerators, and give you experience using the Amazon Web Services (AWS) EC2 public cloud for FPGA-accelerated simulation.

As most neural network models do not fit in on-chip memory, loop blocking/tiling is an important tool in writing a performant neural network implementation. The additional degree of freedom afforded by the scratchpad in ML accelerators requires further planning of data re-use within the scratchpad. Furthermore, the size of the compute array adds an additional constraint on the tiling hierarchy.

There is a wide body of literature on loop blocking and scheduling for standard scalar processors. However, this body of literature is less extensive with regards to on-chip accelerators with dedicated memories that may be connected to the memory hierarchy in various forms.

Specifically, the DMA of the Gemmini accelerator is currently connected to the shared L2 cache of the scalar host processor. As such, it may see caching affects from the tiling scheme, as well as other parts of the program.

In this lab you will continue using the Chipyard and Gemmini platforms from the previous lab to further improve the performance of DNN execution using ML accelerators through software optimization. You will also be using the FireSim platform (within Chipyard) on the AWS EC2 public cloud.

1.1 FireSim and Amazon Web Services (AWS)

In order to use FireSim on AWS (you will find additional information about FireSim in the next section), you will need to open an AWS account. **AWS is a paid service, so be careful about your usage (more details to follow).** Do not wait with this process until the last minute, because it might have around a 1-day latency due to humans-in-the-loop. The lecture on Wednesday, February 26th, will provide a step-by-step tutorial describing the procedures in this section. We recommend following the instructions in this page[1] of the FireSim documentation.

When you are done opening your AWS account, please fill out this form[2]. After you fill out the form, you will receve an email from us with a $200 AWS credit promo code. This amount should be more than sufficient for the tasks in this assignment, assuming you manage you resources and track your AWS expenses. In addition, you can recieve another promo code of $100 by signing up as a student in AWS Educate[3] with your `berkeley.edu` email address. **Do not** sign up for the AWS Educate Starter Kit, because you will not have access to the FPGA-based F1 instances which are required for this lab. You will need to redeem your AWS promo code credits by following the instructions[4].

Once you have opened an AWS account an applied you promo code for credits, follow the initial FireSim AWS infrastructure setup instructions in this page[5] of the FireSim documentation. **Do not** continue to setting up your manager instance (we will provide you with a prepared manager AMI to shorten some of the build preparation process).

2 Background

2.1 Loop Tiling and Scheduling

We will be working on tiling matrix multiplication, since Gemmini accelerates matrix multiplication operations, and we saw in the previous lab that convolutions can be lowered to matrix multiplication operations. As a reminder, a standard nested-loop matrix multiplication operation looks as follows:

[1] https://docs.fires.im/en/latest/Initial-Setup/First-time-AWS-User-Setup.html
[2] https://forms.gle/YLKirgLGpdzpupk59
[3] https://aws.amazon.com/education/awseducate/
[4] https://aws.amazon.com/awscredits/
[5] https://docs.fires.im/en/latest/Initial-Setup/Configuring-Required-Infrastructure-in-Your-AWS-Account.html

```
for (int j = 0; j < DIM_J; j++) {
  for (int k = 0; k < DIM_K; k++) {
    for (int i = 0; i < DIM_I; i++) {
        Outputs[i,j] += Inputs[i,k]*Weights[k,j]
    }
  }
}
```

An example loop-tiled implementation of this loop will look as follows:

```
for (int i0 = 0; i0 < DIM_I/TILE_I; i0++) {
  for (int j0 = 0; j0 < DIM_J/TILE_J; j0++) {
    for (int k0 = 0; k0 < DIM_K/TILE_K; k0++) {

      for (int j = 0; j < TILE_J; j++) {
        for (int k = 0; k < TILE_K; k++) {
          for (int i = 0; i < TILE_I; i++) {

            Outputs[i0*TILE_I + i, j0*TILE_J + j] +=
                Inputs[i0*TILE_I + i, k0*TILE_K + k] *
                Weights[k0*TILE_K + k, j0*TILE_J + j]
          }
        }
      }
    }
  }
}
```

In this example, the size of the tiles/blocks is defined by `TILE_I`, `TILE_J`, and `TILE_K`. The tiled implementation re-uses the data to the maximal extent within the tile/block, before moving to the next block which requires communication with memory. The size of the block is usually correlated with the size of a level of the memory hierarchy. This example represents one level of blocking. In many cases, there is more that one level of blocking, based on the sizes of register files, buffers, scratchpads and caches.

2.1.1 Gemmini Loop Nesting

In Gemmini, the dimensions of the systolic array themselves add another implicit level of tiling to the loop nest. This level is not just *temporal*, but also *spatial*, because it happens in parallel across the PEs of the systolic array, separated by space, rather than being only spread temporally across different time steps. Let's suppose that we build Gemmini with a `DIM`×`DIM` systolic array. Then our tiled loop would become:

```
// Loop-level 3
for (int i0 = 0; i0 < DIM_I/(TILE_I*DIM); i0++) {
 for (int j0 = 0; j0 < DIM_J/(TILE_J*DIM); j0++) {
  for (int k0 = 0; k0 < DIM_K/(TILE_K*DIM); k0++) {

   // Loop-level 2
   for (int j = 0; j < TILE_J; j++) {
    for (int k = 0; k < TILE_K; k++) {
     for (int i = 0; i < TILE_I; i++) {
```

```
// Loop-level 1
for (int spatial_j = 0; spatial_j < DIM; spatial_j++) {
 for (int spatial_k = 0; spatial_k < DIM; spatial_k++) {
  for (int temporal_i = 0; temporal_i < DIM; temporal_i++) {

   Outputs[(i0*TILE_I + i) * DIM + temporal_i, (j0*TILE_J + j) * DIM + spatial_j] +=
     Inputs[(i0*TILE_I + i) * DIM + temporal_i, (k0*TILE_K + k) * DIM + spatial_k] *
     Weights[(k0*TILE_K + k) * DIM + spatial_k, (j0*TILE_J + j) * DIM + spatial_j]

  }}}}}}}}}
```

There isn't anything you can really do about the innermost loop (*Loop-level 1*), because it is determined by the size of the systolic array and the properties of the controller. However, you *can* change the size of `TILE_I`, `TILE_J`, or `TILE_K`. For example, you can calculate these tiling factors based on the size of the scratchpad, so that as many `DIM`×`DIM` submatrices can be reused as many times as possible whenever they are loaded into the scratchpad, and before being written back out to main memory to make space for more input and weight submatrices.

If you wish, you could also add another level of loop tiling above the outermost level shown above, based on the size of the L2 cache of the SoC (512 MB).

This describes how loop tiling works at a high level, but the following subsections will describe these loops are actually implemented in your tests.

2.2 Gemmini Memory Layout

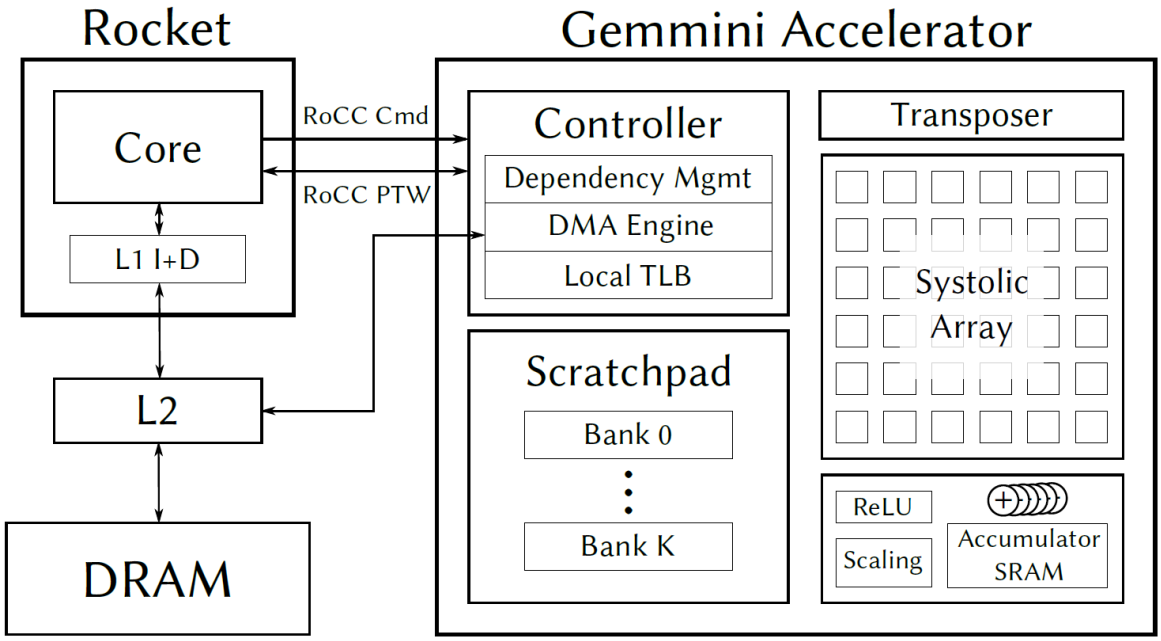

Figure 1: Gemmini Systolic Array Matrix Multiplication Accelerator

As seen in Fig. 1, the scratchpad is made up on K SRAMs. Each row in the SRAMs is `DIM`×`input_bitwidth` bits wide, where the systolic array is made up of `DIM`×`DIM` PEs. Therefore, in this lab assignment, each SRAM row is 32 bytes wide, because we will be using a 32×32 systolic array this time, instead of the 8×8 array we used in Lab 2.

The SRAMs are "row-addressed", which means that if we store one `DIM`×`DIM` matrix in address 0, we can store the next one at address `DIM`.

If each of the K SRAM banks has `BANK_ROWS` rows, then addresses 0 through `BANK_ROWS`−1 would all be on the first bank, while addresses `BANK_ROWS` through `BANK_ROWS`×2 − 1 will be on the second bank, and so on and so forth. Each SRAM bank has one read port and one write port, which means that if we try to perform a matrix multiplication with two matrices that are on the same bank, then our throughput will be halved. It is thus wise to try to put operands on separate banks if possible, which is one possible improvement that you will be able to make to the tiled matrix multiplication software.

Finally, as you can see in Fig. 1, Gemmini also includes an "Accumulator SRAM" which is part of its local memory space, just as the Scratchpad SRAMs are. Every Gemmini local memory space address is 32-bits wide, but if the MSB is 1, then it will refer to row in the Accumulator SRAM, rather than in the other scratchpad SRAM banks. When performing matrix multiplications, we will typically write the results of sub-matrix multiplications (*Loop-level 1* shown in Sec. 2.1.1) to the accumulator, where each row has `DIM` elements of 32-bits each, rather than 8-bits like the Scratchpad SRAMs. Once our partial results here have been fully accumulated, we store the results in the accumulator SRAM back into the L2 cache, but only after scaling them down again to 8-bit elements.

If we perform a matrix multiplication and provide it an output address where the MSB is 1, and the next most significant bit is 0, then Gemmini will overwrite whatever was in that address previously with the new output of the systolic array. But if the second most significant bit is 1, instead of 0, then Gemmini will instead accumulate the new output of the systolic array on top of the existing values in that address of the accumulator SRAM.

There is also a special address that we refer to as the "garbage address", where all bits are set to 1. This is a placeholder address which returns either zeros, or random data, depending on the instruction that it is used for.

2.3 Gemmini Low-Level ISA

`gemmini.h` includes numerous low-level macros that are simple wrappers around Gemmini assembly instructions. Our higher-level tiled matrix multiplication functions are essentially wrapper around these lower-level macros. In this section, we briefly describe what they do. We will assume that Gemmini is configured with a `DIM`×`DIM` systolic array.

We'll start with instructions that move data from the L2 cache to Gemmini's scratchpad:

- `gemmini_config_ld`: This instruction sets the *stride* of Gemmini's "move-in" commands. The stride is the difference in bytes between the L2/DRAM addresses of each row that will be moved-in from L2 to the scratchpad.

- `gemmini_mvin`: This instruction moves in `DIM` rows from L2 into an address in the scratchpad or the accumulator. If the address is in the scratchpad, then each row moved in is `DIM`×`input_bitwidth` bits long. If the address is in the accumulator, then each row moved in is as long as an accumulator row. Each row moved in is put in consecutive addresses in Gemmini's local memory.

- `gemmini_block_mvin`: You can use this command to move in several consecutive `DIM`×`DIM` matrices from L2 to the scratchpad, instead of splitting them across different `gemmini_mvin` commands. This is typically faster.

Next, let's consider the commands used to "move-out" data from the scratchpad or accumulator to L2:

- `gemmini_config_st`: This instruction sets the *stride* of Gemmini's "move-out" commands. The stride is the difference in bytes between the L2/DRAM addresses of each row that will be moved-out to L2 from the scratchpad.
- `gemmini_mvout`: This instruction moves in `DIM` rows from the scratchpad or the accumulator into an L2/DRAM address. Each row is moved out from consecutive addresses in Gemmini's local memory.

Finally, let us consider the commands used to perform matrix multiplications with our systolic array. With Gemmini, every matmul essentially computes $A \times B + D = C$. (For this lab, D will always be 0, so we use the "garbage address" to refer to it.) Due to external limitations, we cannot fit four operands into one assembly command, and so every matmul command in Gemmini is split into two consecutive instructions: one to "preload" the systolic array with a B matrix, and one to "compute" the result of multiplying an A matrix with it afterwards.

The exact commands used are as follows:

- `gemmini_config_ex`: Configures the matrix multiplication commands by setting their dataflow, activation function, scaling parameters, etc.
- `gemmini_preload`: Preload a B matrix into the systolic array, and set the output address for the result, C, of the matmul.
- `gemmini_compute_preloaded`: Multiply an A matrix with the B that was just preloaded.
- `gemmini_compute_accumulated`[6]: Multiply an A matrix with the B matrix that was preloaded *before* the preceding "preload" command. This is useful if we want to re-use the same preloaded B value multiple times.
- `gemmini_loop_ws`: This performs a loop of preload-compute commands which is unrolled in hardware.

As an example, let's suppose we want to write code to perform the following low-level operations:

1. Move in A1, A2, and B each of which is a `DIM`×`DIM` matrix in L2/DRAM into the scratchpad.
2. Multiply A1 by B and store the result in the accumulator.
3. Multiply A2 by B and add the result to the result of A1×B which is already in the accumulator.
4. Store the result in L2.

To do so, we would run this code:

```
int8_t A1[DIM][DIM];
int8_t A2[DIM][DIM];
int8_t B[DIM][DIM];
int8_t C[DIM][DIM]; // C = A1 * B + A2 * B

int A1_sp_addr = 0;
int A2_sp_addr = DIM;
int B_sp_addr = DIM*2;
int C_acc_addr = (1 << 31);

gemmini_config_ld(DIM);
```

[6]This name was chosen to make sense for the output-stationary dataflow, which you do not touch in this lab. It doesn't really have anything to do with accumulations for weight-stationary dataflows.

```
gemmini_mvin(A1, A1_sp_addr);
gemmini_mvin(A2, A2_sp_addr);
gemmini_mvin(B, B_sp_addr);

gemmini_config_ex(WEIGHT_STATIONARY, NO_ACTIVATION, 0, 0, 0);

// C = A1 * B
gemmini_preload(B_sp_addr, C_acc_addr);
gemmini_compute_preloaded(A1_sp_addr, GARBAGE_ADDR);

// C += A2 * B
gemmini_preload(GARBAGE_ADDR, C_acc_addr | (1 << 30));
gemmini_compute_accumulated(A2_sp_addr, GARBAGE_ADDR);

gemmini_config_st(DIM);
gemmini_mvout(C, C_acc_addr);
```

2.4 Gemmini High-Level Tiled Matmul Functions

`gemmini.h` provides higher-level functions that wrap around these to perform tiled matrix multiplications. One of these functions operates on hardcoded tiling factors, while the other one computes tiling factors at runtime.

First, we describe the `tiled_matmul` function:

```
void tiled_matmul(size_t dim_I, size_t dim_J, size_t dim_K,
        const elem_t A[dim_I][dim_K], const elem_t B[dim_K][dim_J],
        const acc_t * D, elem_t C[dim_I][dim_J],
        int act, int shift, bool repeating_bias,
        size_t tile_I, size_t tile_J, size_t tile_K,
        enum tiled_matmul_type_t tiled_matmul_type);
```

The arguments to this function can be described as follows:

- `dim_*`: The dimensions of the A, B, D, and C matrices.
- `A`, `B`: The multiplicands, used to compute $C = A * B + D$.
- `D`: The bias, used to compute $C = A * B + D$. If `D` is `NULL`, then the bias is 0.
- `C`: The buffer in which to store the result of $C = A * B + D$.
- `act`: The activation function to use. For this lab, this will always be `NO_ACTIVATION` or `RELU`.
- `shift`: The number of bits to right-shift the result in the accumulator by before saturating and casting to 8-bits. This is essentially how we scale 32-bit accumulated results back into 8-bits.
- `repeating_bias`: When doing *im2col*, the bias matrix of a convolution will have the same elements in each row, which increases data duplication significantly. If this parameter is set to `true`, then Gemmini will simply re-read the first row of the bias each time, rather than reading new rows from the L2 cache which would waste bandwidth.
- `tile_*`: The tiling factors, as described in Sec. 2.1.1.
- `tiled_matmul_type`: This will be `OS` (output-stationary), `WS` (weight-stationary), or `CPU` (runs on the CPU instead of on Gemmini).

Then, there is the `tiled_matmul_auto` function, which simply wraps around `tiled_matmul` but calculates the tiling factors at runtime.

In `gemmini_nn.h`, there are also two functions, `tiled_matmul_nn` and `tiled_matmul_nn_auto` which wrap around `tiled_matmul` and `tiled_matmul_auto` in `gemmini.h` respectively. These functions all have the exact same parameters, except that the `*_nn` versions also include a parameter, called `check`, that will run the CPU version of the matmul command at the same time to make sure that the output is exactly the same. This can be useful for debugging purposes.

You can think of `tiled_matmul` as corresponding to *Loop-level 3* in Sec. 2.1.1. It determines which elements in L2/DRAM need to be moved into the scratchpad, computed on, and written back to DRAM as a matmul product. There is an inner function inside of it called `sp_tiled_matmul_ws`, corresponding to *Loop-level 2*, which actually moves those elements from DRAM into the scratchpad, runs the necessary "preload-compute" instructions, and writes the result back. That function is defined as follows:

```
static void sp_tiled_matmul_ws(const elem_t * A, const elem_t * B,
      const acc_t * D, elem_t * C,
      size_t I, size_t J, size_t K, size_t A_row_len,
      size_t B_row_len, size_t D_row_len, size_t C_row_len,
      bool no_bias, bool repeating_bias);
```

As you can see, the arguments here are very similar to the last one. The main differences are as follow:

- `D`: If `D` is `NULL`, then Gemmini will accumulate the results of its matrix multiplications on top of whatever was already in the accumulator. Otherwise, if `D` *is not* `NULL`, then Gemmini will overwrite what was already in the accumulator. If `no_bias` is false, then Gemmini will overwrite it with `D`, otherwise, Gemmini will simply overwrite it with the output of the systolic array, since `D` can be simply considered to be a 0 matrix in this case.
- `C`: If `C` is `NULL`, then the results will remain in the accumulator for this iteration, because they still need to be accumlated to their final value. Otherwise, they will be written out to whichever L2/DRAM address `C` points to.

Inside of `sp_tiled_matmul_ws`, each pair of "preload-compute" instructions corresponds to *Loop-level 1*.

Using these functions, we can rewrite the tiled loop in Sec. 2.1.1 as follows:

```
static void tiled_matmul(size_t dim_I, size_t dim_J, size_t dim_K,
        const elem_t A[dim_I][dim_K], const elem_t B[dim_K][dim_J],
        const acc_t * D, elem_t C[dim_I][dim_J],
        size_t tile_I, size_t tile_J, size_t tile_K,
        int act, int shift, bool repeating_bias) {

  // Loop-level 3
  for (int i0 = 0; i0 < DIM_I/TILE_I; i0++) {
    for (int j0 = 0; j0 < DIM_J/TILE_J; j0++) {
      for (int k0 = 0; k0 < DIM_K/TILE_K; k0++) {

        const acc_t * pre;
        if (k0 != 0) {
          pre = NULL;
        } else {
          size_t bias_row = repeating_bias ? 0 : i0*tile_I*DIM;
          pre = &((acc_t (*)[dim_J])D)[bias_row][j0*tile_J*DIM];
```

```
        }
        elem_t * out = k0 == K0-1 ? &C[i0*tile_I*DIM][j0*tile_J*DIM] : NULL;

        // Loop-level 2
        sp_tiled_matmul_ws(&A[i0*tile_I*DIM][k0*tile_K*DIM],
            &B[k0*tile_K*DIM][j0*tile_J*DIM],
            pre, out,
            tile_I, tile_J, tile_K,
            dim_K, dim_J, dim_J, dim_J,
            no_bias, repeating_bias);

        // Loop-level 1 happens inside of "sp_tiled_matmul_ws", and corresponds
        // to a pair of "preload-compute" instructions.
    }}}
```

The actual code is a little more complicated, to handle cases where `tile_I` is not divisible by `dim_I`, for example, but this is basically what it does.

2.5 Gemmini Generated Tuning Parameters

During elaboration, the Gemmini generator automatically generates a C header file named `gemmini_params.h` with tuning parameter definitions based on the hardware configuration that was selected. For example:

```
#define DIM 32
#define BANK_NUM 4
#define BANK_ROWS 2048
#define ACC_ROWS 512

typedef int8_t elem_t;
typedef int32_t acc_t;
```

These parameters include the size of the systolic array, the size of the scratchpad, the size of the accumulator SRAM, the input types (`elem_t`), and the bitwidth of the accumulated partial sums (`acc_t`). These constants can be used when implementing loop tiling schemes for performance optimization.

From these parameters, we can see that the total number of addressable rows in the scratchpad is `BANK_NUM`×`BANK_ROWS` which equals 8192, which corresponds to 8192×`DIM`×`sizeof(elem_t)`/1024 = 256 kilobytes. There is only ever one accumulator bank, so the total number of addressable rows in the accumulator is simply `ACC_ROWS`, which equals 512 rows, or 512×`DIM`×`sizeof(acc_t)`/1024 = 64 kilobytes.

In this lab, we are going to work with a 32×32 systolic array, which is more representative of accelerators in edge applications (as oppposed to the 8×8 array that was used to test Lab 2).

3 Simulation Infrastructure

3.1 Spike ISA Simulator

In Lab 2, your assignment was to write hardware components, and you were provided with software tests. In contrast, in this lab, you are provided with the hardware simulation, and you are writing software components. Naturally, your initial implementation might have some bugs in it. While it is possible to debug the software using the hardware simulation, that will usually result in a long debugging cycle (since simulation is slow). An alternative approach is to use a higher level of abstraction, by using a *functional simulator* (instead of a detailed performance-accurate simulation). Gemmini has a functional

model implemented in the non-standard version of the Spike ISA simluator. The Spike ISA simulator was originally used as a "golden model" for the RISC-V ISA. As a functional simulator, in Spike every instruction takes only 1 cycle to execute (no matter how complicated the instruction is, or whether it should have memory latency).

A binary of the Spike simulator is located in the software development tools that you get when you source `/home/ff/ee290-2/chipyard-env.sh` in Chipyard. Spike should be on your path after you source this file, so in order to run a software binary in Spike, you should just need to run the following command:

```
spike --extension=gemmini <path/to/software/binary>
```

Spike has various visibility features that may help you while debugging your software implementations. You can read about most of these features in https://github.com/ucb-bar/esp-isa-sim/#interactive-debug-mode

You are also provided with a special version of Spike, which includes additional details about the functional simulation of the Gemmini functional model. In order to enable this version of spike, you will need to source the `/home/ff/ee290-2/chipyard-debug-env.sh` file instead of `/home/ff/ee290-2/chipyard-env.sh`. This version generates many print statements, so we recommend using it only when neccessary. This version is identical to the regular Spike version (command line options, flags, etc.), with the addition of these print statements.

```
source /home/ff/ee290-2/chipyard-debug-env.sh
spike --extension=gemmini <path/to/software/binary>
```

3.2 FireSim FPGA-Accelerated Simulation

FireSim is an FPGA-accelerated cycle-exact simulation platform which uses FPGAs on the AWS EC2 public cloud. In contrast to software RTL simulation, FPGA-accelerated simulation enables us to run long workloads (billions-trillions of cycles) within reasonable wall-clock time. Running these workloads in software RTL simulation would take many hours/days/weeks. In contrast to standard FPGA prototyping, FireSim's simulation maintains cycle accurate timing behavior of the entire system (including memory and peripherals). For example, the simulations you are going to run in this lab are going to use a timing-accurate DDR3 memory model. Additional information about FireSim can be found on the FireSim website (https://fires.im/) and in the FireSim documentation at https://docs.fires.im/en/latest/.

FireSim is included as part of the Chipyard framework, which we used in Lab 2. FireSim is located in the `sims/firesim` directory of Chipyard. You will use FireSim in this lab in order to evaluate the performance of real DNN models on the Gemmini accelerator RTL using simulations which take billions of cycles to run.

The remainder of this section will guide you through setting up a FireSim manager instance for this assignment, which you will use for the second half of the assignment. We recommend going through this part of the setup of the manager instance only after you completed the first part of the assignment (on the local `eda` machines), since this will help conserve AWS resources while your instance is not in active use.

FireSim uses a central *manager* instance in order to manage its operations on AWS. In order to set up your FireSim manager instance, head to the EC2 Management Console. In the top right corner, ensure that the correct region is selected.

To launch a manager instance, follow these steps:

1. From the main page of the EC2 Management Console, click `Launch Instance`. We use an on-demand instance here, so that your data is preserved when you stop/start the instance, and your data is not lost when pricing spikes on the spot market.

2. When prompted to select an AMI, search for the following AMI name: `Berkeley EE290-2 FireSim Manager AMI - Spring 2020`. If you have completed the form in the the introduction section, and recieved an AWS promocode from us, it should appear in the "My AMIs" section (we are sharing this AMI manually with you, since there are limitations on public AMIs in AWS. If more than a day has passed since you completed the form in the introduction section and the AMI doesn't appear in your , please email us). ****DO NOT USE ANY OTHER VERSION.****[7].

3. When prompted to choose an instance type, select the instance type of your choosing. A good choice is a `r5.large`.

4. On the "Configure Instance Details" page, first make sure that the `firesim` VPC is selected in the drop-down box next to "Network". Any subnet within the `firesim` VPC is fine. Additionally, check the box for "Protect against accidental termination." This adds a layer of protection to prevent your manager instance from being terminated by accident. You will need to disable this setting before being able to terminate the instance using usual methods.

5. You can skip the "Add Tags" page.

6. On the "Configure Security Group" page, select the "firesim" security group that was automatically created for you earlier.

7. On the review page, click the button to launch your instance. Make sure you select the `firesim` key pair that we setup earlier.

Note: once the instances was launched, your AWS account is charged for its use. You should "stop" your manager instance when you are not using it for more than a few hours—especially at night. This will make sure your allocated AWS credits will suffice for this lab.

FireSim recommends using mosh instead of `ssh`, or using `ssh` with a screen/tmux session running on your manager instance to ensure that long-running jobs are not killed by a bad network connection to your manager instance. In either case, `ssh` (or `mosh`) into your instance (e.g. `ssh -i firesim.pem centos@YOUR_INSTANCE_IP`). Now that the manager instance is started, copy the private key that you downloaded from AWS earlier when you set up the intrastructure (`firesim.pem`) to `~/firesim.pem` on your manager instance. This step is required to give the manager access to the instances it launches for you.

Go into the firesim directory in `chipyard/sims/firesim` on the manager instance, and source the `sourceme-f1-manager.sh` file.

```
$ source sourceme-f1-manager.sh
```

Finally, run the `firesim managerinit` command.

```
$ firesim managerinit
```

This will first prompt you to setup AWS credentials on the instance, which allows the manager to automatically manage build/simulation nodes (these are the same credentials you entered in the infrastructure setup section with the `aws configure` command (choose the `us-east-1` region, and `json` default output format).

Next, it will create initial configuration files we will use. Finally, it will prompt you for an email address, which is used to send email notifications upon FPGA build completion and optionally for workload completion. You can leave this blank if you do not wish to receive any notifications.

[7] This is a version of the Amazon FPGA Developers AMI version 1.6 in which we have pre-installed Chipyard, FireSim, the software toolchain, and other dependancies and configurations found on the `ee290` branch of Chipyard, in order to save you time

You are now done with setting up your manager instance. You can continue on the executing simulations, or *stop* the instance in order to reduce the charges it inccurs. Note the difference between *terminating* an instance, and *stopping* an instance.

4 Your Assignment

In this lab, we will continue working with the Gemmini matrix multiplication accelerator. However, this time we will focus of the software aspects of it rather than the hardware aspects.

In Lab 2, you tested the correctness of your implementation using pre-written benchmark tests. The goal of this lab will be to optimize these simple tests, as well as more complex neural networks to achieve higher performance.

Additionally, in this lab we are going to work with a 32×32 systolic array, which is more representative of accelerators in edge devices (as opposed to the 8×8 array that was used to test Lab 2).

This lab will consist of two parts:

- Code optimization and evaluation on small benchmarks using Spike and software RTL simulation.
- Large scale performance testing on full DNNs using the FireSim FPGA-accelerated simulation plarform.

In order to decouple this lab from the success or quality of your implementation in Lab 2, we have updated the Chipyard repository (and its submodules) with a Chisel Mesh implementation and several additional updates. To use the correct, reference implementation, uncomment everything in `generators/gemmini/src/main/scala/gemmini/C-Mesh.scala`, and comment everything in `generators/gemmini/src/main/scala/gemmini/Mesh.scala`.

For your local work on the `eda` machines for the first part of this lab, while you can pull and recursively update the submodules, we would recommend to just re-clone and initialize the repository on the `eda` machines in that same way as instructed in the introduction of Lab 2. For the second part of this lab you will use EC2 instances with prepared images in which the repository has already been clone and prepared.

4.1 Code Optimization

Your assignment is to optimize the matrix multiplication implementation used by Gemmini by improving its tiling and scheduling.

You will try to optimize the functions described in Sec. 2.4. The implementation of these functions can be found in the `generators/gemmini/software/gemmini-rocc-tests/include/gemmini.h` file.

To keep things simple, we recommend focusing primarily on the `tiled_matmul_auto` function (although you are allowed to modify any of the functions in the file if you choose to). This function can be optimized and improved by calculating better tiling factors, as it currently uses 1-by-1-by-1 tiling factors for everything (as you might have noticed in Lab 2 when a larger scratchpad did not improve inference time for your CNN).

As mentioned previously, you will use a 32×32 systolic array, instead of the 8×8 systolic array you implemented in Lab 2. With a 32×32 systolic array, the peak theoretical performance should be 1024 MACs/cycle (or 2048 OPs/cycle). We do not expect you to reach the peak theoretical performance, but it can be used as an estimate of how much you have improved from the baseline implementation, and how far you are from the theoretical peak.

The hardware configuration you will use in this lab are called `GemminiEE290Lab3RocketConfig` (instead of the `GemminiEE290Lab2RocketConfig` you worked with in Lab 2). You should build the VCS simulator for this configuration in a similar fashion to Lab 2. In this part of the assignment, you will test the performance of your implementation using the same benchmarks you used in Lab 2. In

order to re-build the software benchmarks using your new matrix multiplication implementation, go into `generators/gemmini/software/gemmini-rocc-tests/` and re-run `build.sh`.

As mentioned in the background section of this lab, if your benchmark tests are not passing in software RTL simulation, it might be useful to run your software implementation on the Spike functional model to debug that software implementation's functionality.

Some hints to your optimization method:

- Start by using the `BANK_NUM`, `BANK_ROWS`, and `ACC_ROWS` parameters to calculate tiling factors for the loops based on the scratchpad size in `tiled_matmul_auto`. Think about the data-movement and maximum re-use within the scratchpad.
- In `sp_tiled_matmul_ws`, the B submatrices are stored directly after the A submatrices (as you can see by looking at the `B_sp_addr_start` variable). This increases the chances that they will be on the same bank, reducing throughput. Can you calculate a different starting address for the B submatrices which minimizes the chances of A and B sharing scratchpad banks?
- Consider adding an additional level of loop tiling based on the L2 cache size (there is no automatically generated parameter for this size, so you will need to manually estimate it, or use some auto-tuning method).
- In the large-matrix-multiplication, MLP, and CNN tests, we always use functions which automatically calculate the tiling factors are runtime. If you wish, you may replace these with function calls where you use hardcode tiling factors that you have precomputed.
- If you have any other ideas, then feel free to change any part of the code, such as by changing the ordering of the loops, or anything else. Feel free to ask about new ideas you might have on Piazza or in office hours.

In this part of the assignment, your implementation will be evaluated on the following benchmarks:

- `large_matmul_without_cpu`: A 64×64 tiled matrix multiplication, as in Lab 2, but without CPU code running to check the result.
- `very_large_matmul`: A 256×544 matrix multiplied by a 544×256 matrix. By default, this will not run any CPU code to check the result, but if you wish, you may change the `CHECK_RESULT` macro in `very_large_matmul.c` to 1 if you want to check the result.
- `cifar_quant`: Run inference on a batch of 4 CIFAR-10 images, using LeNet.

In your lab report, answer the following questions

1. Write down the speedup you achieved on each benchmark compared to the baseline implementation. How close is it to the theoretical peak? For the `cifar_quant` test, use the total number of cycles spent, rather than just the matrix multiplication cycles, when calculating how close you are to the theoretical peak.
2. Which optimization techniques did you use to achieve this speedup? How much did each technique contribute to the speedup?
3. Note that the first two benchmarks evaluate your optimization of matrix multiplication, while the last benchmark (LeNet on CIFAR) evaluates a CNN. Do you notice a difference in speedup between the different benchmarks? Which benchmark demonstrated the least improvement? What is the reason for that?

4.2 DNN Inference Performance

In this part of the assignment, you will test the performance of your implemenetation on larger DNN models using the FireSim FPGA-accelerated simulation platform. We could use the standard software RTL simulation we used in the previous part of this lab for these benchmarks as well, but it would take many hours to simulate every benchmark.

In this part of the assignment, your implementation will be evaluated on the following benchmarks:

- `mobilenet`: Run inference of a batch of 4 ImageNet images, using MobileNetV2.
- `resnet50`: Run inference of a batch of 4 ImageNet images, using ResNet50.
- `mlp1`: Run a multi-layer perceptron detecting handwritten digits[8]. This test uses random weights and doesn't actually check the final result, unlike the CNNs.
- `mlp2`: Run a multi-layer perceptron also detecting digits[9]. This test uses random weights and doesn't actually check the final result, unlike the CNNs.

Log in to your FireSim manager instance. FireSim has a particular environment file that needs to be sourced every time you open a new terminal session:

```
cd sims/firesim/
source sourceme-f1-manager.sh
```

Copy your implementation of `gemmini.h` (and any additional code you wrote) into the equivalent directory on your FireSim manager instance[10].

Now, we can prepare a Linux image with your optimized software implementation. We will do that using the FireMarshal[11] workload management system.

While a Linux image is not strictly necessary for these workloads, it makes interaction with FireSim somewhat simpler. While it is possible to run bare-metal tests (i.e., not in a Linux environment) on a FireSim simulation, it is much more conveinent to batch all the tests together into a Linux images with some scripts. Before doing this, make sure that your `gemmini_params.h` file makes sense for the build you are working on. We have wrapped the relevant FireMarshal commands in a convenience script that will build and install your simulated Linux image (this should take 2-6 minutes):

```
cd ../../generators/gemmini/software
./build-ee290-firesim-workload.sh
```

We have already prepared an FPGA image for you, with a 32×32 systolic array, with a scratchpad of 256 KiB, and 64 KiB accumulators. The detail of these FPGA images can be found in several FireSim configuration files we have put in the `/generators/gemmini/software/firesim-configs` directory. These configuraiton files will provide information for FireSim in the following steps.

To run a simulation, we will start by launching a FireSim simulation `runfarm` (an AWS `f1.2xlarge` instance with an FPGA):

```
firesim launchrunfarm --runtimeconfigfile /home/centos/chipyard/generators/gemmini/
    software/firesim-configs/config_runtime_ee290.ini --hwdbconfigfile /home/centos/
    chipyard/generators/gemmini/software/firesim-configs/config_hwdb.ini
```

Next, we will prepare the simulation by copying the simulation infrastructure to the runfarm, and flash the FPGAs:

[8] D. Claudiu Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big simple neural nets excel on handwritten digit recognition," *arXiv preprint arXiv:1003.0358*, 2010.

[9] U. Meier, D. C. Ciresan, L. M. Gambardella, and J. Schmidhuber, "Better digit recognition with a committee of simple neural nets," in *2011 International Conference on Document Analysis and Recognition*, pp. 1250–1254, IEEE, 2011.

[10] In a standard FireSim developement flow, you would maintain a Chipyard fork that can be synced between your local development repository and the repository on the FireSim manager instance

[11] FireMarshal details and documentaiton can be found in https://firemarshal.readthedocs.io/

```
firesim infrasetup --runtimeconfigfile /home/centos/chipyard/generators/gemmini/
    software/firesim-configs/config_runtime_ee290.ini --hwdbconfigfile /home/centos/
    chipyard/generators/gemmini/software/firesim-configs/config_hwdb.ini
```

This step should take 1-2 minutes (unless, for some reason, you have editted some scala files, in which case FireSim will attempt to elaborate a design for scratch. However, this design might not match the FPGA image we have prepared for you, so this is probably not great).

Finally, we will start the simulation:

```
firesim runworkload --runtimeconfigfile /home/centos/chipyard/generators/gemmini/
    software/firesim-configs/config_runtime_ee290.ini --hwdbconfigfile /home/centos/
    chipyard/generators/gemmini/software/firesim-configs/config_hwdb.ini
```

You will now get a monitor screen which shows you details about your simulation under execution. If everything is ok, your simulation should finish executing within 10-15 minutes. If you want to see the progress of the execution of the simulation, you can ssh into the simulation host from a new terminal session:

```
ssh <IP address in that appears in the monitor>
screen -r fsim0
```

This will attach to the simulated console so you can view the progress of your simulated program. To exit `screen`, you can use `ctrl-a ctrl-d` key sequence. The majority of time will likely be spent in an inital stage of "zeroing FPGA DRAM". Don't worry about this.

The final output of your simulation will appear in the `sims/firesim/deploy/results-workload` directory. Each time you run a simulation, a new results directory will be created here with a unique timestamp identifier. Within this results directory, you will find an output file called `uartlog` which will have the uart output of your simulation. The output of the simulation will print the results you need in order to analyze the performance of the DNN models execution of your optimized software implementation.

Have a look at the execution performance of inference using these complete DNN models using your optimized software, and answer the following questions in your report:

1. For each DNN model, how much did the optimization of matrix multiplication impact the overall execution time of the full DNN? (Don't forget to run measurements on FireSim of the baseline non-optimized software).

2. Are there certain DNN types (CNN, MLP, etc.) that exhibited better speedups as a result of the optimization? If so, what are likely reasons?

3. Amdahl's law is a useful tool in estimating the potential speedup of a workload as a result of the acceleration of a specific component. In our case, the component we accelerated is matrix multiplication. Based on the speedup of a single matrix multiplication that you calculated in previous parts of the assignment, and the overall speedup of the DNN models you observed in this part of the assignment, can you estimate what is the percentage of the overall DNN model which is made of matrix multiplication?

We will now evaluate whether your tiling scheme is flexible across different scratchpad sizes. We have provided an additional gemmini parameter header file called `gemmini_params_ee290_smallsp.h`. Replace `gemmini_params.h` with this new parameter configuration, and rebuild the Linux image:

```
cd ../../generators/gemmini/software
cp gemmini-rocc-tests/include/gemmini_params_ee290_smallsp.h gemmini-rocc-tests/
    include/gemmini_params.h
./build-ee290-firesim-workload.sh
```

Now, repeat the procedure for running the FireSim simulation, but instead of using the `config_runtime_ee290.ini` FireSim configuration file, use the `config_runtime_ee290_smallsp.ini` configuration file. This will use an FPGA image in which Gemmini has only a 128 KiB scratchpad and 32 KiB accumulators (in contrast to the 256 KiB scratchpad and 64 KiB accumulator we have worked with until now).

Answer the following questions based on the results of your simulation with a smaller scratchpad:

1. What performance improvement (compared to the baseline software implementation) do you see on the various DNNs with the accelerator with the smaller memory (128 KiB scratchpad, 32 KiB accumulators)? How does this compare to the performance improvement you observed with the larger memory (256 KiB scratchpad, 64 KiB accumulators)

2. What can you learn about your software tiling implementation or about the accelerator for the performance of the accelerator with a larger memory and a smaller memory? (hint: think about memory and compute boundness, roofline models, the portability of your tiling scheme, etc.)

When you are done with the lab, remember to stop your manager instance. We also recommend verifying in the AWS management console that you do not have any other instances running. **At the end of the course, we recommend terminating your manager instance in order to not incur any additional AWS charges**.

5 Lab Report Structure

Submit a PDF writeup of your responses to the questions in Sections 2 and 4 on Gradescope. Make sure to include your name and student ID number in the writeup. Copy the output results of your simulations (with the cycle count breakdown). Finally, please copy the `gemmini.h` file (or any other code that you wrote), including the template code we wrote, and put it in an Appendix. Please highlight the code segments that you edited/wrote. We value code documentation. The lab report will be considered incomplete without properly commented code.

6 Parting Thoughts

In this lab, we explored mapping a matrix multiplication operation (and higher level DNN) onto an accelerator with limited arrays size and scratchpad size.

If you're up for an extra challenge, think about the implications of this type of mapping in a multi-core (and multi-accelerator) SoC. Many of the recent ML accelerators presented by commercial companies are composed of multiple symmetric tiles, each of which has a scalar processor, a vector procesor, and a 2D processor with some form of local memory (in some way - similar to a Rocket+Gemmini configuration). Scheduling tasks across these tiles adds an additional level of complexity to the mapping problem.

This can be done "automatically" using common shared-memory parallel processing libraries such as OpenMP, or more manually using thread pinning an static memory management.