Lab 2: Systolic Arrays and Dataflows

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

Due: February 21, 2020

Contents

1 Introduction

This lab will teach you the components of a basic matrix multiplication hardware accelerator for machine learning, and the use of dataflows within such accelerators.

We will do this using a systolic-array based accelerator called Gemmini, developed here at UC Berkeley. Gemmini is an open-source matrix multiplication accelerator for machine learning which is based on a systolic-array architecture. We will use a fork of Gemmini, which has been stripped of several components for the exercises in this lab. The accelerator will be integrated with a host Rocket processor in an SoC configuration using the Chipyard framework.

1.1 Getting Started With Chipyard

This lab will require using the Chipyard framework. Chipyard is an integrated design, simulation and implementation framework for open source hardware development. Chipyard is open-source online, but we will use a fork that has been modified for the use-case of this lab. In particular, this fork has a custom version of the Gemmini accelerator, which has been stripped of several components for the exercises in this lab. Chipyard is based on the Chisel and FIRRTL hardware description libraries, as well as the Rocket Chip generator SoC generation ecosystem.

Additional information about Chisel can be found in https://www.chisel-lang.org/. While you will not be required to write any Chisel code in this lab, basic familiarity with the language will be helpful in understanding the implementation of generators and interactions between components of the system. An initial introduction to Chisel can be found in the Chisel bootcamp: https://github.com/freechipsproject/chisel-bootcamp. Detailed documentation of Chisel functions can be found in https://www.chisel-lang.org/api/SNAPSHOT/index.html.

This lab assumes you have access to the EECS department instructional machines. In order to create an instructional account, follow the instructions in https://inst.eecs.berkeley.edu/webacct. We will use the eda machines, "eda-1.eecs.berkeley.edu" thru "eda-8".

To install this custom Chipyard version, we will clone the Chipyard code repository from class repository: https://github.berkeley.edu/ee290-2/chipyard. Note that you may also fork the repository if you would like to maintain your own version controlled copy[1].

```
$ git clone https://github.berkeley.edu/ee290-2/chipyard.git
$ cd chipyard
$ ./scripts/init-submodules-no-riscv-tools.sh
```

Again, **make sure you clone from the address that we provide above.** If you try to use the default Chipyard repo, rather than our fork, you will not be able to find tools that we have created specifically for this class[2]. This will take a few minutes, and will clone the course Chipyard repository and initiate the relevant submodules.

Note, that these instructions are slightly different than the instructions found in the main Chipyard documentation due to the use of the course repository and a custom software development toolchain. We provide you a centrally built version of the custom software development toolchain ("esp-tools"). The pointers to this software development toolchain are located in the `/home/ff/ee290-2/chipyard-env.sh` file on the instructional machines. Once the initialization is done, you will need to source `/home/ff/ee290-2/chipyard-env.sh`. You will need to source `/home/ff/ee290-2/chipyard-env.sh` each time you open a new terminal session and want to use Chipyard[3].

[1] The class repository is found on a Berkeley GitHub Enterprise version which is accessible using your CalCentral account - no need for for a public GitHub account

[2] The Berkeley GitHub Enterprise class repository is tracking the `ee290` branch of the public Chipyard repository on public Github at https://github.com/ucb-bar/chipyard/

[3] We provide the centrally built version of the toolchain for efficiency purposes. The same toolchain can be built locally by users by running the `scripts/build-toolchains.sh` script in Chipyard with an `esp-tools` argument. This will result in an `env.sh` file, which is a local equivalent to the `chipyard-env.sh` file we use in this lab

```
$ source /home/ff/ee290-2/chipyard-env.sh
```

Additional documentation for the Chipyard framework can be found in https://chipyard.readthedocs.io

1.2 Gemmini Generator

The Gemmini project [3] is developing a systolic-array based matrix multiplication unit generator for the investigation of software/hardware implications of such integrated SoC accelerators. It is inspired by recent trends in machine learning accelerators for edge and mobile SoCs.

Gemmini is implemented as a Rocket Custom Coprocessor (RoCC) with non-standard RISC-V custom instructions within the Chipyard environment. The Gemmini unit uses the RoCC port of a Rocket or BOOM tile, and by default connects to the memory system through the System Bus (i.e., directly to the L2 cache). The system architecture of a Gemmini accelerator is demonstrated in Figure 1.

You can find the Gemmini accelerator codebase within the `chipyard/generators/gemmini` directory of your chipyard project.

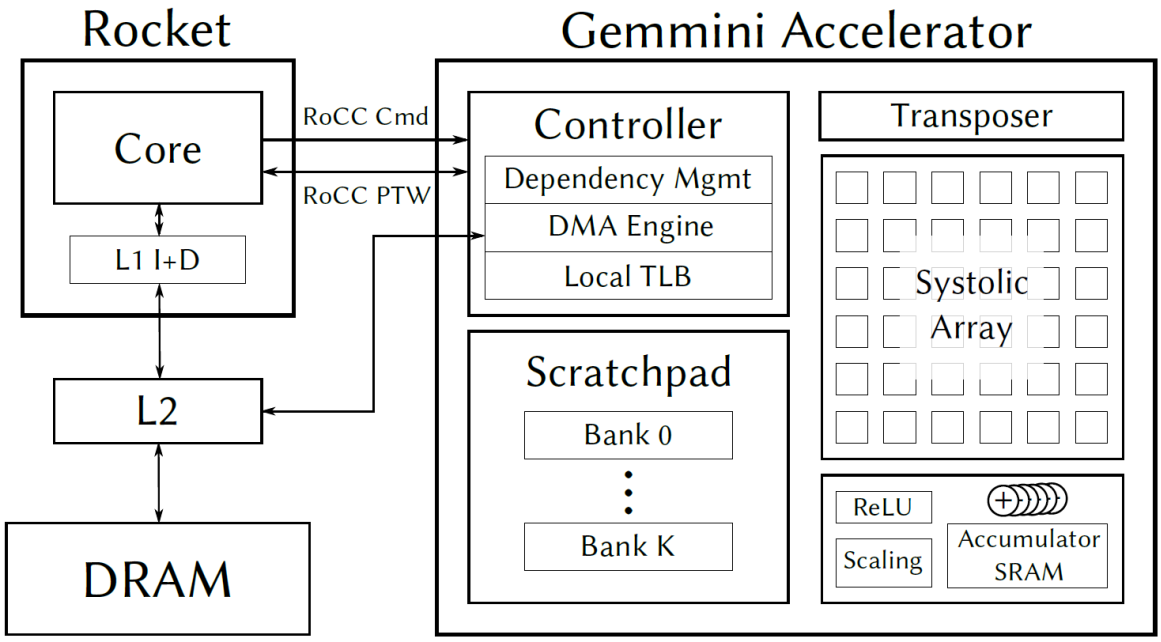

Figure 1: Gemmini Systolic Array Matrix Multiplication Accelerator

In this lab, you will be provided with a version of Gemmini that does not have its systolic array implemented. By implementing the systolic array (also called the "mesh" within the Gemmini codebase), you will learn about the role of various components of a typical ML accelerator, as well as gain hands-on experience with the implementation of a specified dataflow within a systolic array.

2 Background

In this lab we are going to focus on dataflows in matrix multiplication. Dataflows represent the flow of data within a streaming architecture. In particular, in systolic-array architectures, dataflows often embody the ordering of loops within a multi-level nested loop computation.

Convolution operations in convolutional neural networks are often composed of a 7-level nested loops, with the following dimension: input channel dimension (C), output channel dimensions (K), batch-size dimension (N), filter width (R) and height (S) dimensions, feature width (W) and height (H) dimensions.

```
for (int n = 0; n < N; n++) {
  for (int k = 0; k < K; k++) {
    for (int p = 0; p < P; p++) {
      for (int q = 0; q < Q; q++) {
        for (int r = 0; r < R; r++) {
          for (int s = 0; s < S; s++) {
            for (int c = 0; c < C; c++) {
              h = p * stride - pad + r;
              w = q * stride - pad + s;

              Output[n,k,p,q] += Input[n,c,h,w]*Weights[k,c,r,s];
            }
          }
        }
      }
    }
  }
}
```

As mentioned earlier, Gemmini is a matrix multiplication systolic array. Convolution operations in convolutional neural networks are often lowered to a matrix multiplication operation using a procedure called *im2col*. In this procedure, both the data tensor and the filters tensor are lowered to matrices through replication and re-arranging. Further information and examples of *im2col* convolution lowering can be found in [2], [1].

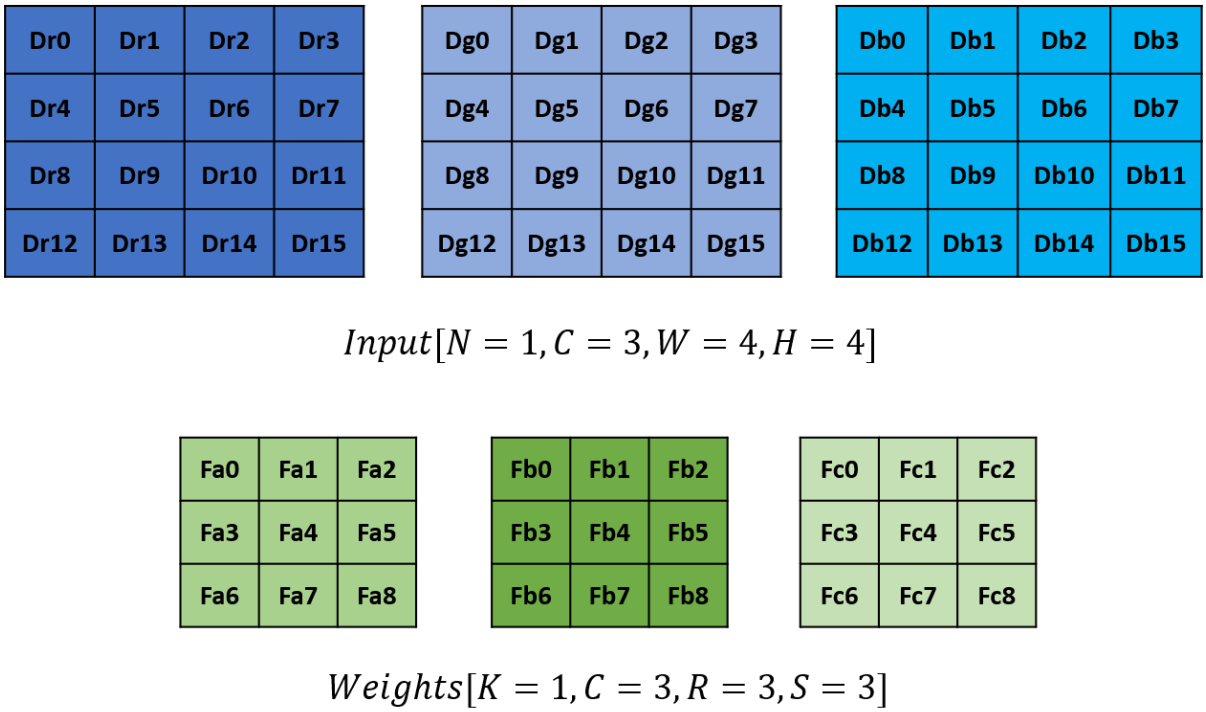

Figure 2: Example convolutional layer with a batch size of 1. The input image of size 4x4 in in the RGB format (3 input channel). The convolutional layer includes one filter with a size of 3x3, applied over 3 channels

Let us take a look at the example in Figure 2. This example demonstrates a small convolutional layer executed on a 4x4 input RGB image, with one 3x3x3 convolution filter. In your lab report, answer the following questions:

1. Perform a lowering of this convolution operation into a matrix multiplication operation using the *im2col* technique. Write down the lowered data matrix and lowered filters matrix using the notations in the example.

2. This example uses a batch size of 1 (which is common for image inference on edge devices). How would the lowered matrices change with a larger batch size? Would a larger batch size be useful for a matrix multiplication systolic array architecture?

Now that we have lowered the convolution operation into a GEMM operation, let us look at a common 3-level matrix multiplication loop for C = A*B:

```
for (int k = 0; k < DIM_K; k++) {
  for (int j = 0; j < DIM_J; j++) {
    for (int i = 0; i < DIM_I; i++) {
      C[i,j] += A[i,k] * B[k,j];
    }
  }
}
```

This particular dataflow is called a *weight-stationary* dataflow, since the weight (filters) elements remain constant, while the output data and feature input data flow during it's computation.

In your lab report, answer the following question:

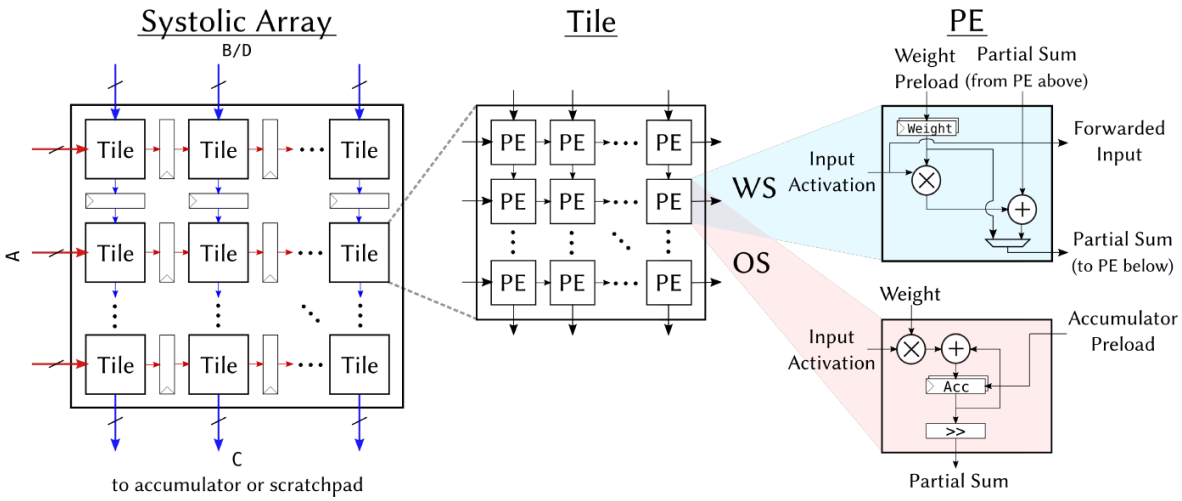

Figure 3: Mesh hierarchy within the original Gemmini implementation. In this lab, you will replace this entire component.

1. Write down the loop ordering for an *output-stationary* dataflow, where the output matrix is the stationary matrix.

3 Your Assignment

The goals of this lab are to familiarize you with the concepts of dataflows in systolic array architectures, as well as the Chipyard and Gemmini tools. Therefore, in this lab, you will replace the existing Chisel implementation of the systolic array mesh in Gemmini with your own Verilog implementation. This will enable you to get hands-on experience with dataflow routing and processing elements implementations, as well as the components of the Gemmini accelerator and the Chipyard framework.

We have provided a wrapper to a Chisel Verilog BlackBox (`generators/gemmini/src/main/scala/gemmini/Mes.scala`), which implements the same interfaces as the Mesh module of the Gemmini accelerator. Your assignment is to write a Verilog implementation of `MeshBlackBox.v` (`generators/gemmini/src/main/resources/MeshBlackBox.v`) which implements a weight-stationary dataflow. You will notice that your mesh implementation is constrained by the properties of the Gemmini controller. You will need to learn the properties of this controller (under the default configuration) in order to understand when to pipeline your mesh, and how to process to control signals within the mesh.

While the original Gemmini mesh implementation implements both a weight-stationary (WS) and an output-stationary (OS) dataflow, in this lab you are required to implement only a weight-stationary dataflow. As a sidenote, this is the dataflow which is implemented by the TPU [4].

3.1 Implementation Notes

The original Gemmini implementation supports both WS and OS dataflows, as well as additional parameterization such as the level of pipelining with the systolic array. These options are implemented using a hierarchy of *Processing Elements* (PEs) which are capable of multiple dataflows, as well as *Tiles*, which are non-pipelined arrays of PEs. A pipelined *Mesh* is composed of multiple *Tiles* which is composed of multiple *PEs* as demonstrated in Figure 3.

In this lab, you are going to replace the Gemmini Chisel implementation of the mesh with your own Verilog implementation. Nevertheless, in this lab you are required to support only a WS dataflow, and

a fully pipelined systolic array. As a result, you can assume that your tile size parameters (`TILEROWS`, `TILECOLUMNS`) will always have a value of 1, and that the dataflow selection signal will always select a WS dataflow (with a value of 1).

In order to not diverge too much from the original Gemmini implementation, you will still see the implications of these additional options within your codebase, but you can assume they will accept only the specified value.

Throughout your implementation, you will find that the mesh is highly constrained by the controller. We do not expect you to modify the controller, but you will likely need to understand the interaction between the controller and the mesh in order to obtain a functional result. Therefore, the next section will describe the input patterns that the controller will feed into your systolic mesh, and the output pattern that you are expected to implement.

3.2 Input and Output Patterns

In this lab, you will implement a weight-stationary systolic array which computes $A \times B + D = C$, where A, B, and D are all matrices. In this lab, we will fix D to 0, so you can instead assume that you're solving $A \times B = C$.

Now, let us suppose that x_{ij} is the element at row i and column j of matrix X. Then, we could compute $A \times B = C$ for 2-by-2 matrices as shown in Figure 4. Note that the square matrices have been shifted into something more akin to a parallelogram so that each element reaches the right PE at the right cycle. Google also created a helpful GIF[4] which shows this pattern as well.

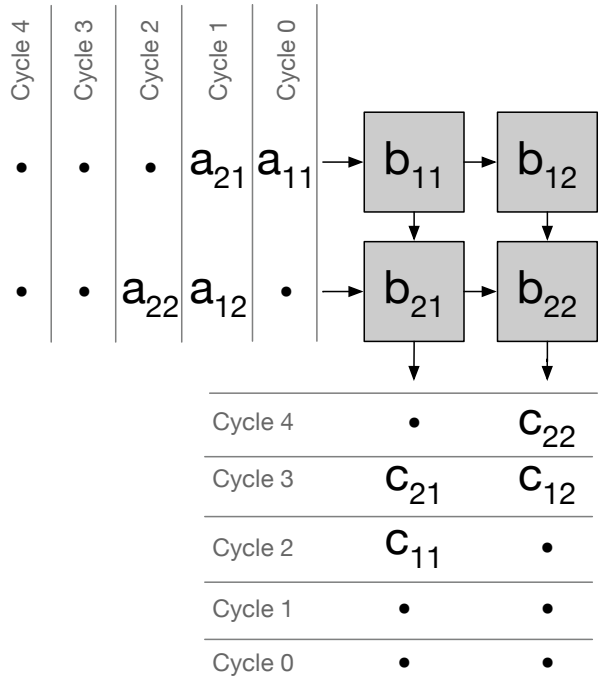

Figure 4: $A \times B = C$, where B is kept stationary in the systolic array.

However, Figure 4 still doesn't tell us how B is fed into the systolic array in the first place. To do that, we first preload B into the systolic array and *then* feed in A, as shown in figure Figure 5.

[4] https://storage.googleapis.com/gweb-cloudblog-publish/original_images/Systolic_Array_for_Neural_Network_2g8b7.GIF

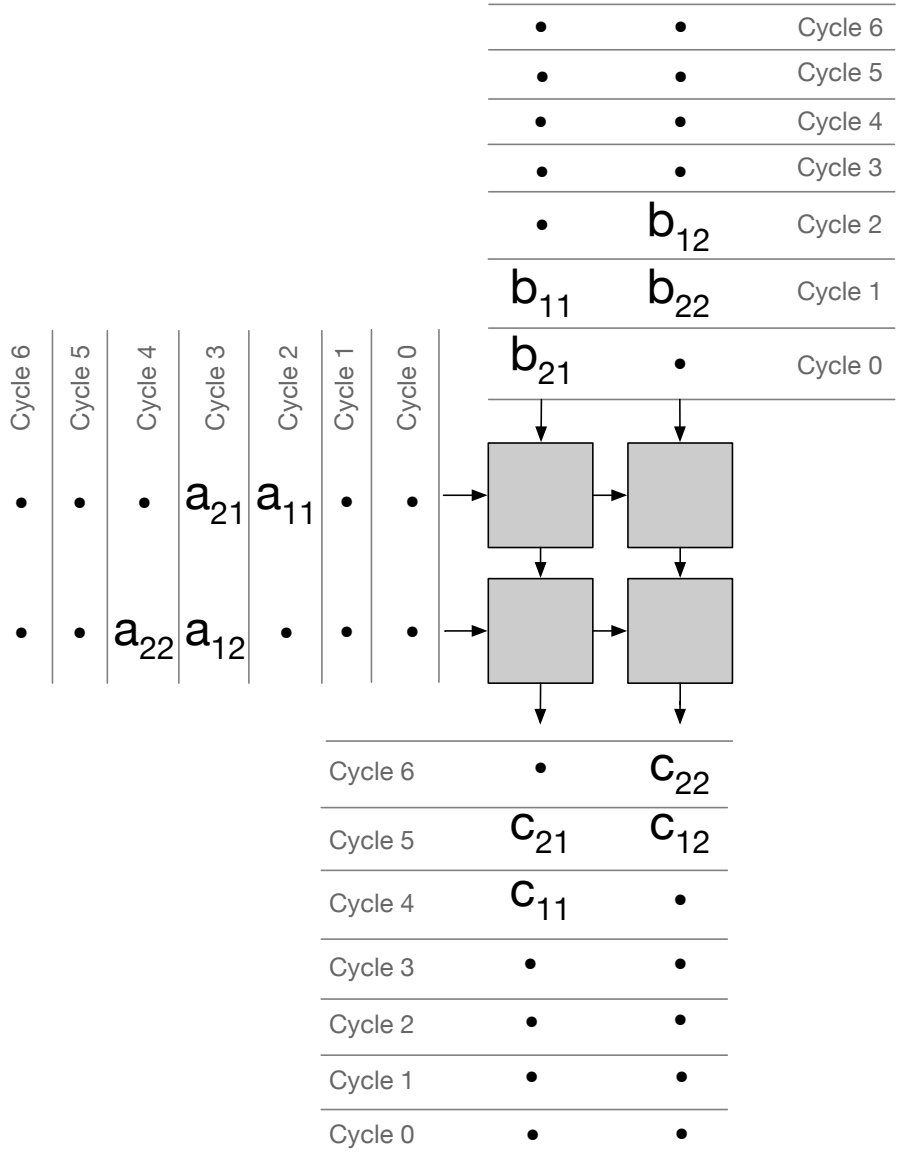

Figure 5: $A \times B = C$, where B is first preloaded into the systolic array.

This still has an obvious issue though: while preloading B, we aren't able to do any useful work. If we want to preload a different B for every matrix multiplication, then our throughput will actually be halved. Therefore, our systolic array must actually be *double-buffered*, which means that every PE must have **at least two registers** for the elements on B that it is responsible for storing. During any particular cycle, one register will be used to perform multiply-accumulate operations, while the other will be propagating elements of B downwards as part of the preloading process. Figure 6 illustrates how our input pattern can be changed to accommodate this. In this example, we first compute $\alpha \times \beta = \gamma$, and then compute $A \times B = C$, overlapping the preloading stage of the second matrix multiplication with the computation of the first one.

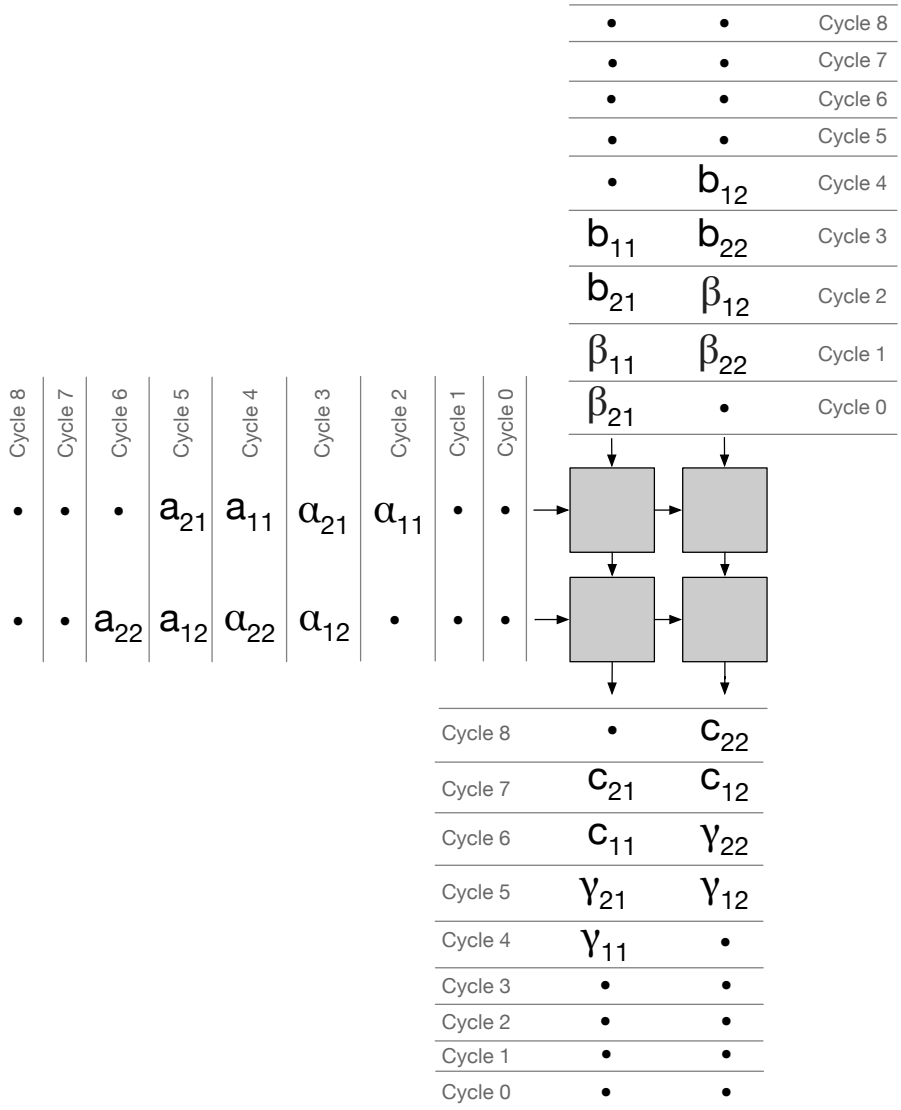

Figure 6: $\alpha \times \beta = \gamma$, followed by $A \times B = C$.

However, this still leaves us with one particular question. What if we want to *preserve* a previously preloaded matrix, instead of overwriting it with a new matrix? How do we communicate that to the PEs of the systolic array? Gemmini's controller does this by passing a one-bit signal, called `propagate`, through the systolic array as well. The signal describes which of the two registers used to store B should be propagated downwards in that cycle, and which one should be used for multiply-accumulates. If we want to re-use a previously preloaded B matrix, then we can simply decide not to change the value of the `propagate` signal, and the old B values will continue to be used.

To hopefully make this more clear, Figure 7 shows us performing two consecutive matrix multiplications, $\alpha \times \beta = \gamma$ and $A \times B = C$, but with the `propagate` signals added.

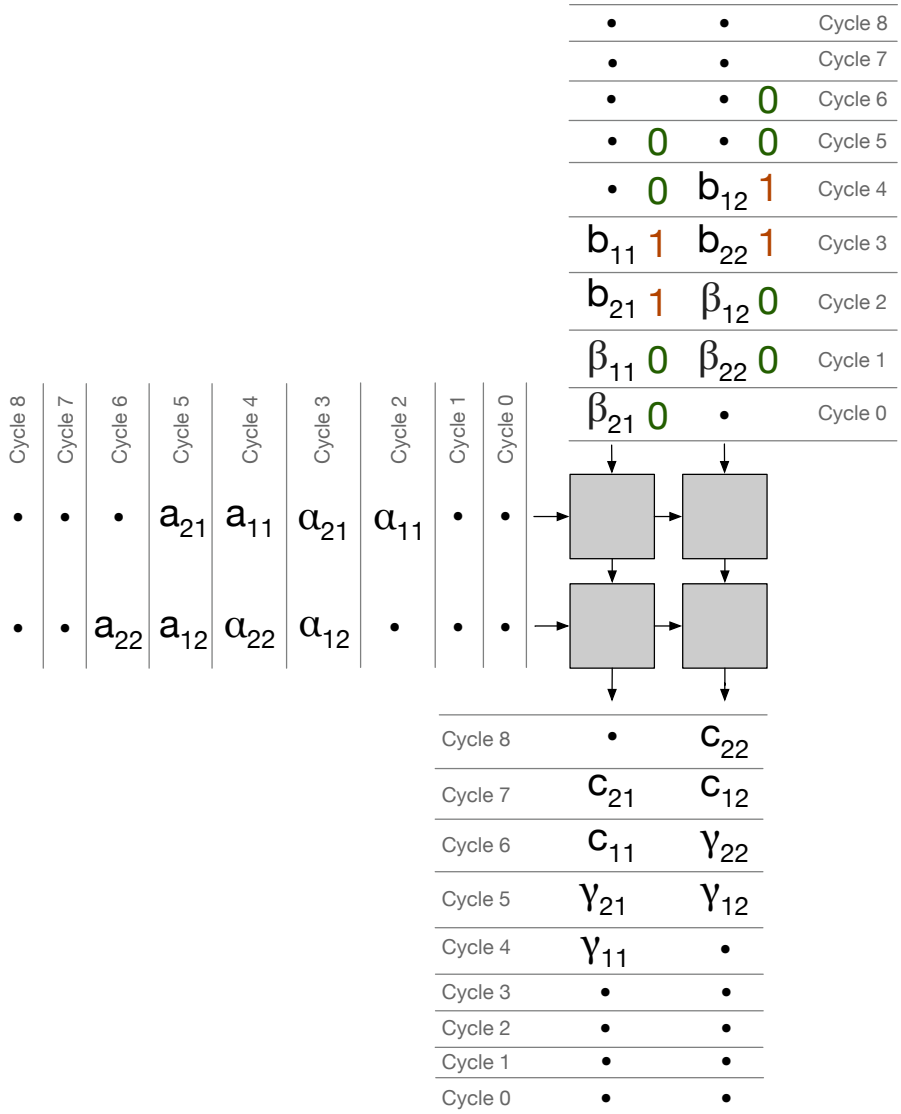

Figure 7: $\alpha \times \beta = \gamma$, followed by $A \times B = C$. This time, we add the `propagate` signals on top. If we wanted to reuse β for the second matrix multiplication instead of multiplying with a new, preloaded B matrix, we would have kept the `propagate` signals constant at 1 after preloading β.

3.3 Interface

Observe the Chisel interface of the Mesh:

```
val io = IO(new Bundle {
   val in_a    = Input(Vec(meshRows, Vec(tileRows, inputType)))
   val in_b    = Input(Vec(meshColumns, Vec(tileColumns, inputType)))
   val in_d    = Input(Vec(meshColumns, Vec(tileColumns, inputType)))
   val in_control    = Input(Vec(meshColumns, Vec(tileColumns, new PEControl(accType))))
   val out_b   = Output(Vec(meshColumns, Vec(tileColumns, outputType)))
   val out_c   = Output(Vec(meshColumns, Vec(tileColumns, outputType)))
```

```
   val out_control  = Output(Vec(meshColumns, Vec(tileColumns, new PEControl(accType))))
   val in_valid = Input(Vec(meshColumns, Vec(tileColumns, Bool())))
   val out_valid = Output(Vec(meshColumns, Vec(tileColumns, Bool())))
})
```

Where the PEControl bundle is defined as:

```
class PEControl[T <: Data : Arithmetic](accType: T) extends Bundle {
  val dataflow = UInt(1.W)
  val propagate = UInt(1.W)
}
```

We provide an Verilog adapter between the Chisel interface and the Verilog blackbox interface. The equivalent Verilog interface of the BlackBox Mesh module is:

```
module MeshBlackBox
  #(parameter MESHROWS, TILEROWS, MESHCOLUMNS, TILECOLUMNS, INPUT_BITWIDTH, OUTPUT_BITWIDTH)
   (
     input                                clock,
     input                                reset,
     input  signed [INPUT_BITWIDTH-1:0]   in_a[MESHROWS-1:0][TILEROWS-1:0],
     input  signed [INPUT_BITWIDTH-1:0]   in_d[MESHCOLUMNS-1:0][TILECOLUMNS-1:0],
     input  signed [INPUT_BITWIDTH-1:0]   in_b[MESHCOLUMNS-1:0][TILECOLUMNS-1:0],
     input                                in_control_dataflow[MESHCOLUMNS-1:0][TILECOLUMNS-1:0],
     input                                in_control_propagate[MESHCOLUMNS-1:0][TILECOLUMNS-1:0],
     input                                in_valid[MESHCOLUMNS-1:0][TILECOLUMNS-1:0],
     output signed [OUTPUT_BITWIDTH-1:0]  out_c[MESHCOLUMNS-1:0][TILECOLUMNS-1:0],
     output signed [OUTPUT_BITWIDTH-1:0]  out_b[MESHCOLUMNS-1:0][TILECOLUMNS-1:0],
     output                               out_valid[MESHCOLUMNS-1:0][TILECOLUMNS-1:0]
    );
```

The mesh ingests three controls signals, which are all input from the North side of the systolic array:

- `in_control_dataflow` - this signal controls the dataflow of the mesh. In this lab, it will be fixed to a weight-stationary dataflow, with a signal value of `1'b1`.
- `in_control_propagate` - this signal controls which of the double buffers within each PE is currently propagating. Notice, this means you will need to implement double-buffering PEs.

The mesh ingests three input data signals. You are expected to compute $A * B + D = C$, but for this lab, D will always be a 0 matrix.

- `in_a` - enters from the West side of the systolic array.
- `in_b` - enters from the North side of the systolic array.
- `in_d` - enters from the North side of the systolic array.

The mesh outputs two data signals, both exiting from the South side of the systolic array:

- `out_b` - relevant only when implementing both the OS and WS dataflows together. Irrelevant for this lab. You can output anything you want for this.
- `out_c` - the result of your matrix multiplication.

Finally, there are input and output valid signals as well:

- `in_valid` - enters from the North side of the systolic array. Describes whether the signals are actually meaningful, or whether they should be ignored.
- `out_valid` - exits from the South side of the systolic array. Describes whether the outputs are valid, or whether they are garbage data (probably as a result of cycles where there were invalid inputs).

3.4 Building and Simulating Gemmini

We have provided a template Verilog file with the implemented interfaces with the `generators/gemmini/src/main/resources/vsrc/` directory of your Chipyard environment. **You are required to complete the implementation of the `MeshBlackBox` module.**

In order to elaborate your design and construct a software RTL simulation using VCS, run the following steps within the Chipyard directory:

```
$ cd sims/vcs
$ make CONFIG=GemminiEE290Lab2RocketConfig
```

These will use the Synopsys VCS simulator on the eda machines. If you choose so, you may also use the Verilator open-source simulator by running the same command in the `sims/verilator` directory. However, in our experience, it is much slower to compile. The first time you run these commands may take a while (10-15 minutes) since many scala packages will be downloaded. The generated source files (including Verilog) for the design will now appear in the `sims/vcs/generated-src/example.TestHarness.GemminiEE290Lab2RocketConfig/` directory.

If you want to generate a vpd file for debugging using waveforms, you will need to generate a debug simluator using the `make debug CONFIG=GemminiEE290Lab2RocketConfig` command.

More information about building a Verilator or VCS simulation which can execute tests can be found here[5].

3.5 Checking Correctness

You will test your implementation using the bare-metal software tests found in the Gemmini repository. In order to build these tests, run:

```
$ cd generators/gemmini/software/gemmini-rocc-tests/
$ ./build.sh
```

This will build a set of bare-metal software tests that can be run using a VCS or Verilator simulation.

```
$ cd sims/vcs
$ # To run the "identity-baremetal" test:
$ ./simv-example-GemminiEE290Lab2RocketConfig
    ../../generators/gemmini/software/gemmini-rocc-tests/build/ee290/identity-baremetal
$ # To generate a .vpd waveform:
$ ./simv-example-GemminiEE290Lab2RocketConfig-debug
    ../../generators/gemmini/software/gemmini-rocc-tests/build/ee290/identity-baremetal
```

For the lab report, you are required to run the following tests:

- `identity-baremetal` - A matrix, whose elements increase consecutively by 1 starting from 10, multiplied by the identity matrix. This test should pass even if double-buffering has not yet been properly implemented.

[5] https://chipyard.readthedocs.io/en/latest/Simulation/Software-RTL-Simulation.html#verilator-open-source

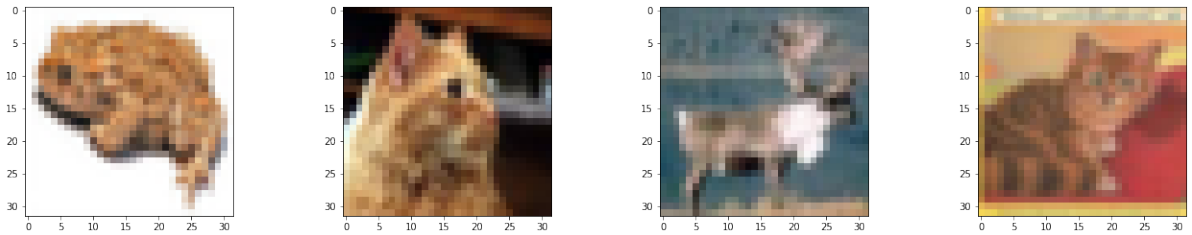

Figure 8: Images that you will classify from CIFAR10 in the `cifar_quant-baremetal` test. The correct classifications are "frog", "cat", "deer", and "cat", but our network classifies these as "frog", "dog", "car" , and "cat" respectively.

- `identity_negative-baremetal` - Identical to the previous test, but with a negative identity matrix instead.
- `double-baremetal` - Two consecutive matrix multiplications, in order to test double-buffering.
- `double_keep_weights-baremetal` - Two consecutive matrix multiplications, in which the second matrix multiplication uses the preloaded weights from the first one.
- `double_keep_weights_then_change-baremetal` - Three consecutive matrix multiplications, in which the second matrix multiplication uses the preloaded weights from the first one, but the third matrix multiplication uses new weights.
- `random_matmuls-baremetal` - Eight consecutive matrix multiplications with random values.
- `large_matmul-baremetal` - A large (64-by-64) matrix multiplication with random values. Because your systolic array will not be 64-by-64 itself, this operation is tiled across multiple smaller matrix multiplications. This test will run a matrix multiplication on the simulated Rocket CPU, as well as the same matrix multiplication on the Gemmini accelerator. This simulation may take around 20 minutes to run.
- `cifar_quant-baremetal` - Classify 4 images (shown in Figure 8) from the CIFAR-10 dataset using the CNN you quantized in Lab 1. The network should correctly classify two of them, and incorrectly classify the other two. This simulation may take around 50-60 minutes to run. The output of this benchmark will also give you a breakdown of cycles between different components of the CNN.

If any of these tests fail, they will print `FAIL`. If you're struggling to debug any of these, then try reducing the size of your systolic array by changing the `meshColumns` and `meshRows` parameters to smaller numbers like 2, or 4, or 8. You can change these parameters in `generators/gemmini/src/main/scala/gemmini/ConfigsEE290.scala`. This may make the waveforms easier for you to understand. However, eventually, make sure that your Verilog code works for a 8-by-8 array, as this is what it will be tested with.

If you do change these parameters, make sure to rebuild the software tests by re-running `./build.sh` as described above after rebuilding your simulator, because a file called `gemmini_params.h` in your tests will have been updated with your systolic array's new parameters.

In your lab report, answer the following questions:

1. How many cycles does the `large_matmul-baremetal` test report for the CPU and the WS systolic array? What is the speedup that your accelerator achieves?
2. How many cycles does a CIFAR inference using your quantized CNN take? In which code segment does it report spending most of the cycles?

3.6 On-Chip Memory Implications

We would like to examine what is the impact of the on-chip scratchpad size on the performance of the accelerator. The default Gemmini scratchpad size is 256 KiB (across 4 banks). We have also provided an additional configuration (`GemminiEE290Lab2BigSPRocketConfig`) in which the Gemmini scratchpad size has been set to 2 MiB. Build the new configuration with your mesh implementation (using the make argument `CONFIG=GemminiEE290Lab2BigSPRocketConfig`). **Make sure to re-build the software tests, since a new header file was generated with the parameters of the configurations. Make sure to run your simulation software test with the correct config name.** Note that the software currently does not change the loop-nesting based on the hardware configuration.

In your lab report, answer the following questions:

1. Did the performance change compared to the 256 KiB scratchpad configurations?
2. What are possible reasons for this?

4 Lab Report Structure

Submit a PDF writeup of your responses to the questions in Sections 2 and 3 on Gradescope. Make sure to include your name and student ID number in the writeup.

Finally, please copy the `MeshBlackBox.v` file (or any other implementation of the Mesh that your wrote), including the template code we wrote, and put it in an Appendix. We value code documentation. The lab report will be considered incomplete without properly commented code.

5 Parting Thoughts

In this lab, we explored dataflow implementation in a systolic-array accelerator using the Gemmini generators. Systolic-array accelerators have a much broader design space. As you can observe in the `generators/gemmini/src/main/scala/Configs.scala` file, the Gemmini generator has several parameters that can affect the design space, and impact the system-level performance of the accelerator.

If you're up for an extra challenge, you could try changing some of the parameters in the config file, and see the resulting impact on cycle-level performance. Even with simple tests like the bare-metal tests provided in this lab, you should still see some interesting phenomenon and performance cliffs.

If you're up for a different challenge for extra credit, you can try exploring the energy implications of the accelerator design. You can use the open-source ASAP7 PDK and the Hammer flow within the Chipyard framework to try to run power estimation for your systolic array implementation. For this exercise, a simple synthesis-based estimation will be sufficient (no need for a full placed and routed design). Note, you will need access to a synthesis tool to explore this option.

References

[1] Why gemm is at the heart of deep learning. https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/. Accessed: 2020-01-24.

[2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[3] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolić, I. Stoica, and K. Asanović. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv:1911.09925 [cs.DC]*, 2019.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.