Hardware Implementation: Data-Flow and Design Space

In this assignment, we go into the details of the implementation of DNNs on specific hardware. The final goal of this assignment is to figure out how different dataflows can change the performance of a DNN workload on a given hardware architecture.

1 Preliminaries

1.1 Memory-access simulator

To estimate the performance of a DNN workload on a given architecture for a specific dataflow, we need to simulate the process of transferring data between memory hierarchies. In fact, a dataflow shows how data is staged in the memory hierarchy. Hence, we may obtain the access count of different components in the given architecture. Finally, we may also estimate the energy consumption of the DNN workload on the designated architecture based on the energy models for each component.

A dataflow mapping for the problem in Fig. 3 can be represented as a set of nested loops as in Fig. 1, which shows the tiled version of an output stationary (OS) mapping. The loop bounds (M_1, P, Q, M_2, R, S) are parameters of the mapping (in Fig. 3, we have $M_1 = M$ and $M_2 = 1$). For the mapping to be valid, the parameters must be such that the data structures can fit in the corresponding level of the memory hierarchy, and also such that they correspond to the dimensions of the problem.

In the example of Fig. 1, the buffer can store an input block with size of $1 \times R \times S$, a weight block with size of $M_2 \times R \times S$, and an output block with size of $M_2 \times 1 \times 1$. In each cycle, the required blocks of the input, weight, and outputs that have to be transferred between main memory and buffer are shown. First, we check if they are already available in the buffer or not. If the blocks are not in the buffer, we have to read them from the main memory and store them in the buffer.

1.2 Installing Timeloop/Accelergy

The assignment requires the use of the *Timeloop* and *Accelergy* tools. Timeloop is a tool to determine the number of times a particular memory or processing element is used (similar to Question 1) and to optimize the dataflow mapping, whereas Accelergy is a tool that estimates the energy cost of each operation (data movement or arithmetic operation).

You can install Timeloop-Accelergy on your own computer or on a Google Colab Pro instance. To install on your own computer, the use of Ubuntu is recommended (either directly or by installing a

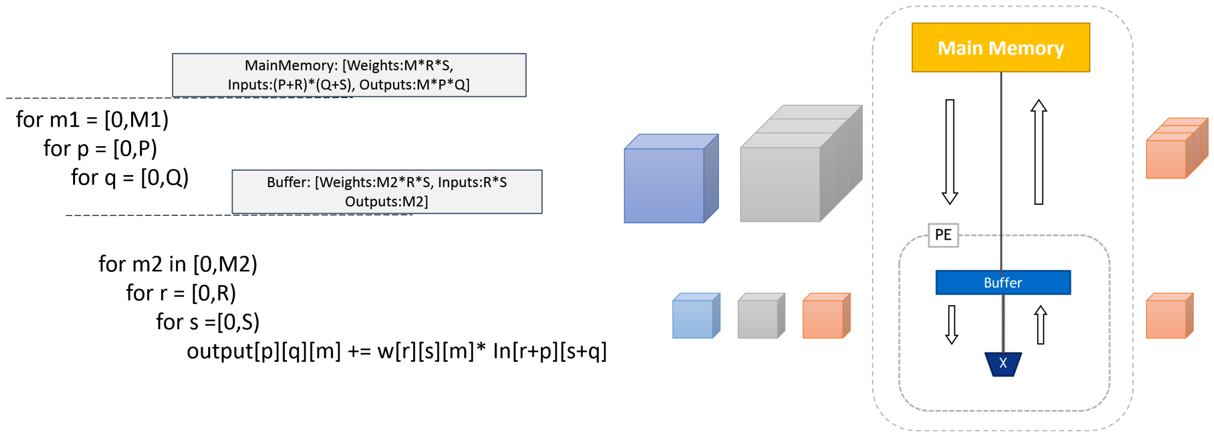
MainMemory: [Weights:M*R*S, Inputs:(P+R)*(Q+S), Outputs:M*P*Q]
for m1 = [0,M1)
for p = [0,P)
for q = [0,Q)
Buffer: [Weights:M2*R*S, Inputs:R*S Outputs:M2]
for m2 in [0,M2)
for r = [0,R)
for s =[0,S)
output[p][q][m] += w[r][s][m]* In[r+p][s+q]
Main Memory
PE
Buffer
X

Figure 1: Dataflow

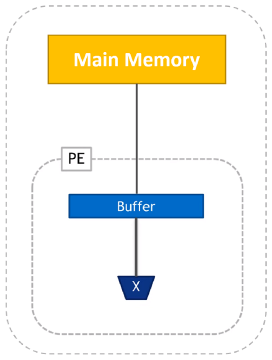

Figure 2: Architecture

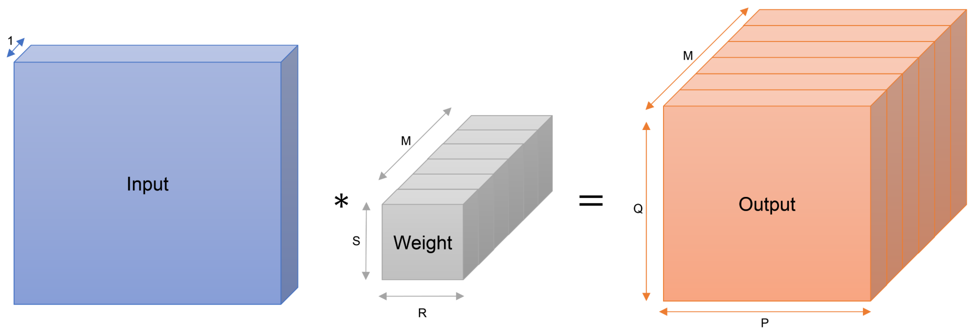

Figure 3: Problem

virtual machine). A local install is the best way to become familiar with Timeloop-Accelergy. However, note that Question 3 will ask you to optimize a DNN model based on the energy consumption predicted by Timeloop. This will require having access to GPU acceleration and Timeloop-Accelergy at the same time. Using Colab Pro is a simple way to achieve this. A script is provided in the homework git repository to install Timeloop-Accelergy on Ubuntu or in a Colab Pro instance (which runs Ubuntu).

Also, make sure to have a look at the Timeloop-Accelergy Tutorial.

2 Questions

Q1. Memory-access simulation (45 points)

In this question, we want to discover the effect of different dataflows for 1D multi-channel convolution (shown in Fig. 4) on the architecture shown in Figure 2. The main memory consists of an SRAM module with a logical depth of 32768 and logical width of 8 bits. The buffer consists of a register file (`regfile`) module with a depth of 64 and width of 8 bits. Also, we use an 8-bit integer MAC (`intmac`) in the PE. For this question, we use the simple energy model given in Table 1.

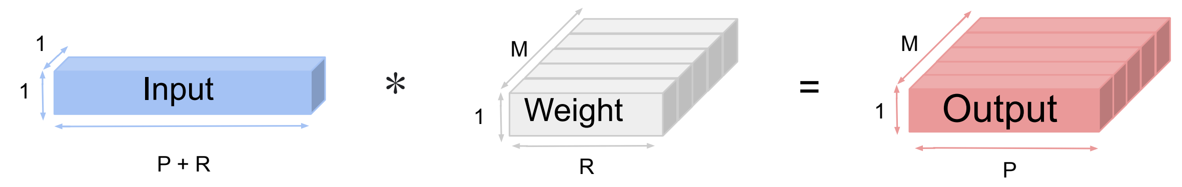

Figure 4: 1-D multi-channel convolution

Your task is to implement your own simulator and test it for different dataflows: weight stationary, untiled output stationary, and tiled output stationary as presented in Table 2. Configuration files are provided in YAML format that describe the architecture (`Q1_arch.yaml`), the problem (`Q1_prob.yaml`), and the different dataflows/mappings. The problem can be equivalently described by the following pseudo-code:

```
for(m=0; m<M; m++)
  for(p=0; p<P; p++)
     for(r=0; r<R; r++)
        o[m,p]+= i[p+r] * f[m,r];
```

Several dataflows and mappings are provided as configuration files. Using your simulator, generate the statistics indicated in Table 2 for each case:

(a) Weight stationary: `Q1_ws.map.yaml`

(b) Output stationary, untiled.
Mapping 1: `Q1_os-untiled.map.yaml`,
Mapping 2: `Q1_os-untiled.map2.yaml`.

(c) Output stationary, tiled.
Mapping 1: `Q1_os-tiled.map.yaml`,
Mapping 2: `Q1_os-tiled.map2.yaml`

Validate your results with *Timeloop-Accelergy* and discuss the results.

Table 1: Energy per action

	Energy (pJ)
SRAM read access	7.95
SRAM write access	5.45
regfile read access	0.42
regfile write access	0.42
MAC use	0.56

Q2. Energy model with Accelergy (15 points)

An interesting feature of Accelergy [1] is its flexibility in using different plug-ins to estimate the energy of different components. For instance, Aladdin [2] is used for FIFO, register files, multipliers, bit-wise operators, and so on, while Cacti [3] is mainly used for general memory components (SRAM, DRAM, etc.). In this question, you will generate figures that illustrate the energy models of DRAM, SRAM, register files, and multiply-accumulate (MAC)

Table 2: Simulation results

		Main Memory		Global Buffer		MAC	**Total MAC Energy**
		read access	write access	read access	write access	uses	
Weight Stationary	$\bigvee_{m=1}^{M} \bigvee_{r=1}^{R} \bigvee_{p=1}^{P}$	??	??	??	??	??	**??**
Untiled Output Stationary	$\bigvee_{m=1}^{M} \bigvee_{p=1}^{P} \bigvee_{r=1}^{R}$	??	??	??	??	??	**??**
	$\bigvee_{p=1}^{P} \bigvee_{m=1}^{M} \bigvee_{r=1}^{R}$	??	??	??	??	??	**??**
Tiled Output Stationary	$\bigvee_{m_1=1}^{M_1} \bigvee_{p=1}^{P} \bigvee_{m_2=1}^{M_2} \bigvee_{r=1}^{R}$	??	??	??	??	??	**??**
	$\bigvee_{p_1=1}^{P_1} \bigvee_{m=1}^{M} \bigvee_{p_2=1}^{P_2} \bigvee_{r=1}^{R}$	??	??	??	??	??	**??**

operations. In the case of memories, plot the energy for read and write access. You can use `timeloop-metrics` for this purpose, which provides a simplified view of an architecture's energy consumption. Determine experimentally how the energy scales as a function of each parameter, and justify the observed scaling based on the corresponding CMOS circuit models.

(a) First, fix the SRAM depth to 8192 and change the width from 16 to 2048. Then, fix the width to 128 and change the depth from 512 to 8192.

(b) First, fix the regfile depth to 8192 and change the width from 16 to 2048. Then, fix the width to 128 and change the depth from 512 to 8192.

(c) Change the DRAM (logical) width from 16 to 2048. (Note: The depth is not a parameter in this case, the model assumes a large-capacity DRAM chip.)

(d) Change the `intmac` width from 2 to 64.

Q3. **Pruning and design space exploration (40 points)** In this question, we consider a joint optimization of the DNN model and of the HW accelerator. For the base HW architecture, we consider the *Eyeriss* [4] chip. Configuration files for this architecture are provided with Timeloop.

For the base DNN model, we consider the ResNet-32 model that was used in Assignment 1. In this question, we will consider optimizing the sparsity of the weights separately for each layer of the model, and the objective will be to minimize energy consumption, as estimated by Timeloop-Accelergy, subject to a minimum task accuracy constraint of 85% correct classifications. We will use structured pruning so that the reduction in the number of non-zero parameters can be directly translated into energy gains even on a hardware architecture like *Eyeriss* that is not specialized for sparse models.

First, complete the `model_to_spars` and `generate_resnet_layers` functions in `solution.py`. Here we focus on structured pruning applied to the output channel dimension of the convolutional layers. You can use PyTorch pruning to prune the network. Follow the instructions in the `main.ipynb` notebook and prune the network. After fine-tuning (by any training recipe), save the model and generate the YAML "problem" files for each layer of the pruned network using `generate_resnet_layers`. Then you can use `run_Accelergy` to optimize the mapping and estimate the energy consumption of the pruned network.

Any reasonable attempt at exploring the design space will be awarded full marks, and more ingenuous approaches will be given bonus points.

3 Deliverables

The answers to all the questions should be presented in a single report. Please also submit the code you developed, either as an archive or using a github link (make sure the repository is public)[1].

For Question 1, make sure to briefly explain the structure of your software simulator. For instance, for an object-oriented implementation, list the classes as well as the methods and attributes of each class. For each element, give a brief explanation of its purpose. Alternatively, you can include documentation that is generated from code comments as an appendix to your report.

Report and code are due on Sunday March 10, 23h59.

[1]Any commit pushed after the due date will not be considered.

References

[1] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *IEEE/ACM International Conference on Computer-Aided Design*, 2019.

[2] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM/IEEE International Symposium on Computer Architecture*, 2014.

[3] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *IEEE/ACM International Conference on Computer-Aided Design*, 2011.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 2016.