Winter 2021

CS 133 Lab 5

FPGA w/ HLS on OpenCL: Convolutional Neural Network (CNN)

Due Date : 3/11/21 11:00PM

Description

Your task is to accelerate the computation of two layers of a convolutional neural network (CNN) using a **high-level synthesis (HLS) tool with OpenCL host-device interface** on an FPGA. Specifically, for an input image with 228×228 pixels of 256 channels, you are going to calculate the tensors after going through a 2D convolutional layer with 256 filters of shape $256 \times 5 \times 5$ using the ReLU activation `relu(x) = max{x, 0}` with a bias value on each output channel. The output tensors are pooled using a max-pooling layer of 2×2. If you are familiar with Karas, they are basically `Conv2D(256, 5, activation='relu', input_shape=(256, 228, 228), data_format='channels_first')` with a given bias and `MaxPooling2D(pool_size=(2, 2))`. You will need to implement this function using HLS:

```
void CnnKernel_YourCode(const float* input, const float* weight,
                        const float* bias,  float* output)
```

where `input` is the input image of size `[256][228][228]`, `weight` is the weights of the convolutional filters of size `[256][5][5]`, `bias` is the offset value added to the output of the convolutional per filter of size `[256]`, and `output` should be written to by you as defined above to store the result of `maxpool(relu(conv2d(input, weight) + bias))`. The output size is `[256][112][112]`.

Please refer to the discussion slides to understand the workflow. You may develop Lab 5 by adapting your own Lab 3 and Lab 4 work. Figure 1 shows the CNN layers in this lab:

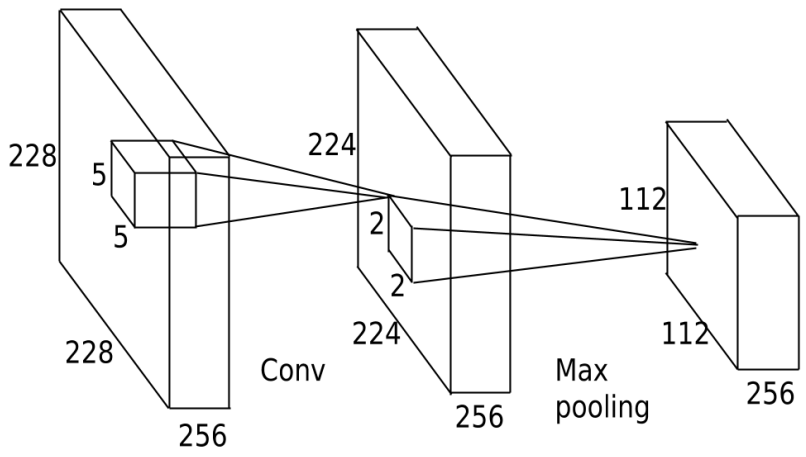

Figure 1: The CNN layers to be accelerated

How-To

FPGA accelerator compilation typically involves three (3) stages: high-level synthesis (HLS), bitstream generation, and onboard execution. The last two stages can take days to complete one design iteration. Therefore, in this lab, we only focus on the first stage: HLS. Your performance will only be assessed using the estimation in the HLS reports, which is usually accurate, instead of the actual end-to-end execution. However, you can try out the last two steps if you are interested.

Create an AWS Instance for Development

Please refer to the discussion section slides and create an **`m5.2xlarge`** instance (or a larger one, e.g. **`c5.4xlarge`**) with **FPGA Developer AMI**. Please use your **personal AWS account** **(otherwise, you will fail to create an instance using the FPGA Developer AMI)** for this lab. You should make sure that you have your **promotional credits** redeemed before you start your lab. Please be careful when using your AWS account as if you use up all your credits, you may end up paying the expenses.

Development Environment Setup

Log in to your AWS instance using username **`centos`** (e.g. `ssh centos@1.2.3.4`) and run:

```
git clone https://github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR
cd $AWS_FPGA_REPO_DIR && source vitis_setup.sh && cd
```

You need to execute the second line every time you log into your instance.

Build and Run Baseline with Software Simulation

We have prepared the starter kit for you at GitHub. Please run:

```
git clone https://github.com/UCLA-VAST/cs-133-21w -o upstream  # replace with
    `(cd cs-133-21w && git pull upstream main)` if you are reusing the folder
cd cs-133-21w/lab5
make
```

This command will perform a software simulation of the provided starter FPGA HLS kernel. It should take around two minutes and show “PASS”. **You need to use FPGA Developer AMI in this lab unless you are using a computer with Xilinx Vitis HLS installation.** However, you are still suggested to develop code locally and start the instances to test the program after all the codings are done.

Modify the HLS CNN Kernel

If you have successfully built and run the baseline HLS CNN kernel, you can now optimize the code to design your CNN kernel. **Your task** is to implement a fast, parallel version of the CNN kernel on FPGA. You should start with the provided starter kit. You should edit **`cnn-krnl.cpp`** for this task.

In contrast to Lab 3-4, you should not and cannot adjust the workgroup parameters as Xilinx FPGA only supports global and local work size to be one (1). **Instead, parallelism should be exploited by using FPGA HLS pragmas.** The list of pragmas in Vitis can be found here and in the discussion slides.

We have prepared a CNN kernel with **well-optimized memory access as your starter kit** for your convenience. In this starter kit, we reduced the bit-width of the data for less resource and bandwidth usage, performed tiling on the H and W loops for the data to fit into on-chip memory, and fetch all the data you would need to use from device memory to the chip in a coalesced and efficient manner.

To **modify the kernel,** you can change the code inside the loop annotated as **`main_loop_i`**. Your task is exactly the same as your Lab 3/4, only with a modified bound of **`h`** and **`w`** loops to **`kTileH`** and **`kTileW`**. Please use the given types `input_t`, `weight_t`, `bias_t`, and `output_t` for the corresponding data, and **`compute_t` for your intermediate values**. You can use them as if they are `float` numbers.

For to accelerate the kernel, you would need to apply the taught pragmas including but not limited to `#pragma HLS unroll`, `#pragma HLS pipeline`, and `#pragma HLS array_partition`. Note that to apply an HLS pragma to a loop, you need to **put the pragma inside the loop body instead of before it**. You would also need to perform some loop transformations as in Lab 3 and Lab 4. To change the tiling size of **`h`** and **`w`** loops, you may uncomment the `#define` for `kTileH/W` in `cnn-krnl.cpp`.

Although the skeleton kernel is provided, you are still free to create your own by removing the header file inclusion of "`lib/cnn-krnl.h`" and implement the basic kernel from scratch. However, this would require specific expertise in Xilinx FPGA architecture and is not recommended for this course.

Test Your HLS CNN Kernel with Software Simulation

To perform software emulation of your FPGA implementation of CNN kernel:

```
make
```

If you see something similar to the following message, your implementation is incorrect.

```
Found 2120190 errors
FAIL
```

Since the software simulation step uses the CPU to emulate the hardware behavior, its execution time is meaningless. Your **estimated execution time** should be retrieved using the command below:

```
make estimate
```

This command will print out the estimated latency of your kernel in terms of seconds:

```
| Latency (cycles) |    Latency (absolute)     |     Interval    | Pipeline|
|   min   |   max  |     min     |     max     |   min  |  max   |   Type  |
+---------+--------+-------------+-------------+--------+--------+---------+
|4279.... |4283... | 171.194 sec | 171.325 sec |4279....|4283....|   none  |
```

The **highlighted maximum** is the estimated execution time of your FPGA kernel. Get the performance by "`kNum*kNum*kImSize*kImSize*kKernel*kKernel*2/max`", or `164.4/max` in terms of GOPS.

IMPORTANT: Please make sure that all your loops have fixed loop bounds. **If any of the loop bounds are variable, a performance estimation will not be shown and you will receive no performance grade.**

IMPORTANT: The "`make estimate`" command should finish in 20 minutes, or in two hours with highly-complex optimizations. Our recommendation is to halt your estimation using `Ctrl-C` when the time exceeds 20 minutes, except for your last step (after you reach ~100 GOPS). **More than 12 hours in the estimation will result in zero for the performance score.** As your kernel design becomes more complex, the software simulation and the estimation will start to take a significantly longer time.

Your **estimated resource usage** should be retrieved using the command below:

```
make resource
```

The FPGA resource (BRAM_18K/DSP48E/FF/LUT/URAM) usage will be printed out:

```
|         Name         | BRAM_18K| DSP48E|    FF    |   LUT    | URAM|
+----------------------+---------+-------+----------+----------+-----+
| ...
|Utilization (%)       |       66|   ~0  |    ~0    |         1|    0|
```

IMPORTANT: As you apply more optimizations, your resource usage will also increase. Ideally, you should keep applying optimization until your kernel occupies about 80% of these resources. The remaining 20% should be reserved for the interfaces (DRAM/PCI-e controller) and the downstream flows. Please make sure that resource utilization is **less than 80% for all FPGA resources**. If any of the resources are over this limit, you will receive **no performance grade**.

IMPORTANT: You can check the HLS report by opening **`cnn-krnl.rpt`** with your favorite text editor. This file will be generated with the command **`make estimate`**. **You must submit this file with your final submission. You should not modify this file in your submission and it will be all verified. Any modification to this file in your submission constitutes academic misconduct and will be reported.**

Advanced Tips for HLS

Monitor Progress: If your estimation takes long, you can monitor the progress in another terminal:

```
tail -f _x/*/CnnKernel/vitis_hls.log
```

Kernel Profiling: If you want to "profile" your kernel, you can open **`cnn-krnl.rpt`** with your favorite text editor and scroll down to **`== Performance Estimates > + Latency > + Detail > * Loop`**. You can see the execution time (**`Latency (cycles)`**) and your pipeline initiation interval in the table. For the resource usage, you can go to **`== Utilization Estimates > + Detail`**. No loop level information is available, however. If you want to check the resource usage of a code region, you can warp it with a function then generate the report again.

Annotation for Profiling: If you find the loops in your report unreadable, you can name the loops you are interested in using a goto label. For example, **`this_loop:`** `for (int i = 0; i < n; i++);`

Unrolling and Pipelining: If you mark a loop for unrolling or pipelining, please remember to perform proper array partitioning to support the concurrent access. Otherwise, you may observe no difference in the performance. Note that any loops inside a pipeline will be automatically forced unrolled.

Debugging Pipelining: If you are not sure about why you cannot achieve a specific initiation interval as you expected, you can open the file below and read the logs. HLS usually gives out a reason.
`_x/cnn.hw.xilinx_aws-vu9p-f1_shell-v04261818_201920_2/CnnKernel/vitis_hls.log`

Long Synthesis Time In Pipelining: You will experience long HLS synthesis time (for generating the estimation) if you pipeline a loop with a large loop body. Besides, please note that as all loops inside a pipeline will be unrolled, it may be automatically a large loop body. In this case, you may want to exchange the order of pipelining and unrolling and see if the time can get improved.

Use Functions For Shorter Synthesis Time: If you experience long synthesis time, you may try wrapping some loops into a function and specify `#pragma HLS inline off` inside the function body. However, this may lead to inaccurate dependency analysis or memory port analysis and cause lower performance sometimes. There might be some workarounds, or not. For example, if you have access to `A[k + i][j]` inside the function, passing `A + k` to the function and accessing `A'[i][j]` can allow HLS to understand the array partitioning better than passing `A`. You need to do experiments.

General Tips

- We will use AWS FPGA Developer AMI for grading (the instance type does not matter).
- When you develop on AWS, to resume a session in case you lose your connection, you can run `screen` after login. You can recover your session with `screen -DRR`. You should ***stop*** your AWS instance if you are going to come back and resume your work in a few hours or days. Your data will be preserved but you will be charged for the EBS storage for $0.10 per GB per month (with default settings). You should ***terminate*** your instance if you are not going to come back and resume your work. ***Data on the instance will be lost***.
- You are recommended to use ***private*** repositories provided by GitHub to backup your code. ***Never put your code in a public repo to avoid potential plagiarism.*** To check in your code to a private GitHub repo, create a repo first.

```
git branch -m upstream
git checkout -b main  # skip these two lines if you are reusing the folder in Lab 1
... // your modifications
git add cnn-krnl.cpp
git commit -m "lab5: first version"  # change commit message accordingly
# please replace the URL with your own URL
git remote add origin git@github.com:YourGitHubUserName/your-repo-name.git
git push -u origin main
```

- You are recommended to `git add` and `git commit` often so that you can keep track of the history and revert whenever necessary.
- ***Make sure your code produces correct results!***

Submission

You need to report the estimated performance results of your FPGA-based OpenCL implementation on a Xilinx Ultrascale+ VU9P FPGA (the FPGA we are using, specified in the makefile). Please express your performance in GOPS and the speedup compared with the CPU sequential version (reuse the result from Lab 3 or Lab 4) and **summarize your results in a table.** Your report should summarize:

- Please explain the **parallelization and optimization strategies** you have applied for each loop in the CNN program (convolution, max pooling, etc) in this lab. Include the pragmas (if any) or code segments you have added to achieve your strategy.
- Please incrementally **evaluate the performance of each parallelization/optimization** that you have applied and explain why it improves the performance.
- Please explain **how your strategy differs** from your Lab 3 and Lab 4, and **why**.
- Please report the **FPGA resources (LUT/FF/DSP/BRAM) usages**, in terms of resource count and percentage of the total. Which resource has been used most, in terms of percentage?
- *Optional*: The challenges you faced, and how you overcame them.
- ***(Bonus +5pts)***: Analyze your code and check if the DSP/BRAM resource usage matches your expectation. Only the adders, multipliers, and size of arrays need to be considered. Please attach related code segments to your report, and show how you computed the expected number. Provide a discussion on possible reasons if they differ significantly.

You also need to submit your optimized kernel code. Do not modify code in the `lib` directory. Please submit on Gradescope. Your final submission should contain and only contain these files individually:

```
├ cnn-krnl.cpp
├ cnn-krnl.rpt
└ lab5-report.pdf
```

File `lab5-report.pdf` must be in PDF format exported by any software.

Grading Policy

Your submission will only be graded if it complies with the formatting requirements. Missing reports/code or compilation errors will result in 0 for the corresponding category(ies).

Correctness (50%)

Please check the correctness using the command "**`make`**".

Performance (25%)

Your performance will be evaluated based on the estimation report generated using the command "**`make estimate`**". The performance point will be added only if you have the correct result, so please prioritize the correctness over performance. Your performance will be evaluated based on the ranges of throughput (GOPS). Ranges A+ and A++ will be defined after all the submissions are graded:

- Range A++, better than Range A+ performance: 25 points + 20 points (bonus)
- Range A+, better than Range A performance: 25 points + 10 points (bonus)

- Range A GOPS [80, 160]: 25 points
- Range B GOPS [40, 80): 20 points
- Range C GOPS [20, 40): 15 points
- Range D GOPS [10, 20): 10 points
- Range E GOPS [1, 10): 5 points
- Lower than range E [0, 1): 0 points

Report (25%)

Points may be deducted if your report misses any of the sections described above.

Academic Integrity

All work is to be done individually, and any sources of help are to be explicitly cited. You must not modify the HLS report `cnn-krnl.rpt` in your submission. Any instance of academic dishonesty will be promptly reported to the Office of the Dean of Students. Academic dishonesty, includes, but is not limited to, cheating, fabrication, plagiarism, copying code from other students or from the internet, modifying the software-generated report, or facilitating academic misconduct. We'll use automated software to identify similar sections between **different student programming assignments**, against **previous students' code**, or against **Internet sources**. We'll run **Vitis HLS on all submissions and compare the reproduced HLS report with the submitted report**. Students are not allowed to post the lab solutions on public websites (including GitHub). Please note that any version of your submission must be your own work and will be compared with sources for plagiarism detection.

Late policy: Late submission will be accepted for **17 hours** with a 10% penalty. No late submission will be accepted after that (you lost all points after the late submission time).