Convolution Operation on FPGA-CPU Heterogeneous System

ELEC5140 Project Report, Spring 2019
CHENG, Wei

Abstract

Artificial Intelligence, computer vision, natural language processing have experienced great success in many practical applications, driven by the development of heterogeneous computing. Recently, multiple heterogeneous systems are applied into these applications, for example, CPU-GPU and CPU-FPGA. In this report, the characteristics of convolution operation is explored, the most basic operation in CNN, based on heterogeneous systems. One end-to-end system based on CPU-FPGA simulation platform PAAS is developed taking a Sobel filtering application as an example. Detailed system working pipeline and implementation procedure is described, which demonstrated it success on CPU-FPGA cooperation on Sobel filtering.

1 Introduction

Heterogeneous computing with hardware accelerators is a promising direction to overcome the power and performance walls in traditional computing systems. CPU-accelerator integrated architectures, such as CPU with ASIC or FPGA based accelerators, are able to provide customized processing according to application requirements and are thus particularly attractive to speed up computation-intensive applications. Therefore, system level simulation showing the interaction among CPUs, hardware accelerators and memory system precisely is important for performing design space exploration leading to architecture and design optimization.

With the development of heterogeneous computing, many techniques artificial intelligence, computer vision, etc, have experienced great leap and shown their power in practical applications. In these techniques, convolution is the most fundamental operation which may applied on linear Euclidean representations, for example, images, or more complicated structures like graphs or manifolds. Investigations on convolution implementation and optimization on hardware accelerators have became a trend among researchers and engineers. Being aware of this impact of convolution operation on heterogeneous systems, a CPU controlled FPGA convolution filter is developed based on the state-of-art CPU-FPGA simulation system, end-to-end filtering is accomplished and successfully demonstrated its effectiveness on CPU and hardware accelerator architecture.

This report is organized in the following structure, first literature on the PAAS system and other integrated system are reviewed in Sec. 2, then system overview and implementation details are investigated in Sec. 3, following with the simulation evaluation in Sec. 4, and finally the conclusion and discussion on this project is describe in Sec. 5.

2 Related Work

2.1 Integrated Simulators

There are some integrated simulators developed for heterogeneous system of CPUs and accelerators, such as gem5- GPU [4], ARAPrototyper [1], gem5-Aladdin [5], and HeteroSim [2]. These heterogeneous system simulators have various concentrations. Aladdin is a pre-RTL accelerator modeling framework which uses dynamic data dependence graphs as a representation of an accelerator without RTL. A through discussion on integrated simulators can be found in [3].

2.2 PAAS

Processor Accelerator Architecture Simulator (PAAS) proposed by Liang *et al.*, is a system level simulator for heterogeneous CPUs-accelerator systems which integrates a modified version of gem5 and the HDL simulator Verilator to enable accurate simulation of dynamic interactions among accelerators, CPUs and memory system. The infrastructure of the gem5 simulator provides flexibility by offering a diverse set of CPU models, system execution modes, and memory system models which could be based on bus or Network-on-Chip (NoC). PAAS integrates gem5 and Verilator and further develops infrastructure to provide system designers with the following features for comprehensive system simulation and exploration.

- Co-simulation of hardware accelerators and CPUs which can be X86 or ARM architecture.
- Shared coherent caches between CPUs and accelerators Capabilities of accelerators for exploiting both fine-grained (beat-based) and coarse-grained (DMA-based) memory access.
- Support of shared virtual address space between accelerators and threads of CPUs.
- Support for accelerator coherency port Runtime control of multiple accelerators.
- Reconfiguration of FPGA
- Parallelized simulation using shared-memory-based inter-process communication

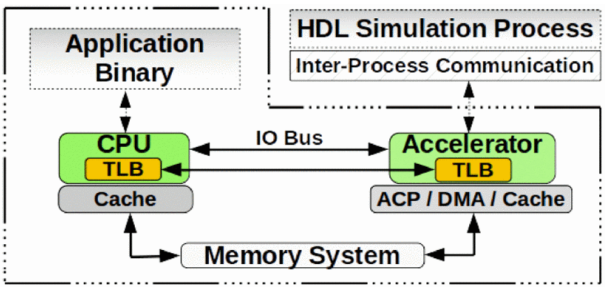

Figure 1: Architecture of PAAS and interactions between various modules during simulation.

As shown in Fig. 1, PAAS is built upon gem5 and Verilator. gem5 is designed tosimulate the system of CPUs and memory system which could be interconnected via bus or NoC. Verilator can be used to compile synthesizable Verilog and generate an executable file which implements the function of

FPGA.Such integration is based on IPC (Inter-Process Communication) to realize parallel simulation and reduce the simulation time. When the process of gem5 is running, the process for implementing the function of FPGA based accelerator will be also launched simultaneously. When a particular FPGA based accelerator is involved in the simulation, its corresponding executable file will be launched and the gem5 process will wake up an interface for the FPGA process. This interface collects data or commands for FPGA and then sends them to the process of FPGA based accelerator through IPC of Linux which is the host operating system for running PAAS.

3 System Implemetation

In this section, the proposed Sobel filtering system using CPU-FPGA based on PAAS platform is introduced. First a global view of the proposed system is decribed, then the implementations details are discussed which is more focusing on my modification veruses PAAS.

3.1 System Overview

The proposed the system works in a prototype of PAAS system, which is shown in Fig. 2. One CPU module and FPGA module are utilized, the CPU module is works as the brain in this system and controls the FPGA module under the defined work load program. The FPGA module act as a hardware Sobel filter who receives the image from CPU and process the convolution filtering, then transfer back to the CPU under the instructions of CPU. The sharing memory of both modules are with the ruby memory system.

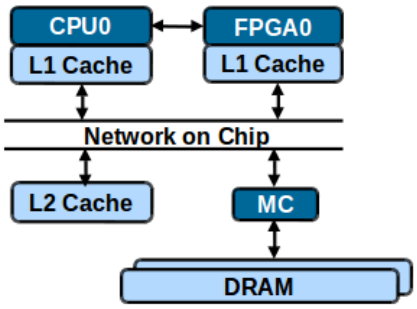

Figure 2: Architecture of proposed Sobel filtering system which is a variation of PAAS.

3.2 FPGA Module

I followed the structure of FPGA module in PAAS system, which works in a Finite State Machine (FSM) mode, which contains six states, which are IDLE, WAIT_READ, DEAL_READ, HANDLING, READY_WRITE, WAIT_WRITE and DEAL_WRITE. An implementation draft code of this FSM is shown in Fig. 3.

```
if (state == IDLE)
begin…
end
else if (state == WAIT_READ)
begin…
end
else if (state == DEAL_READ)
begin…
end
else if (state == HANDLING)
begin…
end
else if (state == READY_WRITE)
begin…
end
else if (state == WAIT_WRITE)
begin…
end
else if (state == DEAL_WRITE)
begin…
end
```

Figure 3: Finite State Machine used in PAAS and proposed system.

The data interface between the FPGA module and CPU includes an enable signal, address bus, and ready, finish status signal and the data bus. The memory transition of this interface should obey the memory read and write timing policy which is shown in Fig. 4.

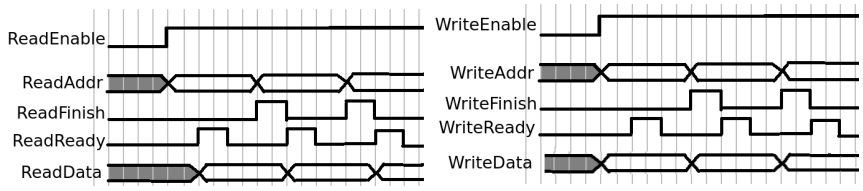

Figure 4: Memory read (left) and write (right) timing policy in PAAS and proposed system.

To perform the desired functionality in FPGA module, the core modification we should apply is to design a convolution filter inside the HANDLING state in the FSM. To leverage the parallel computing power of FPGA hardware, we should design a filtering scheme which convolve on all data elements on the same edge. So I draft the following Python code to generate the parallel version of convolution operation in Verilog, the code is display in Fig. 5.

```python
import numpy as np

with open("conv_v.txt",'w') as f:
    for i in range(0,100):
        for j in range(0,100):
            f.write("ans[%d]<=" % (100*i+j))
            for a in range(-1,2):
                for b in range(-1,2):
                    if (i + a >= 0) and (i + a <= 99) and (j + b >= 0) and (j + b <= 99):
                        if ((a == 1) and (b == 1 or j+b==99)) or (b == 1 and (a==1 or i+a==99))
                            or (j+b==99 and i+a==99):
                            f.write("tmp[%d]*kernel[%d];\n" % (100*(i+a)+j+b, 3*(a+1)+b+1))
                        else:
                            f.write("tmp[%d]*kernel[%d]+" % (100*(i+a)+j+b, 3*(a+1)+b+1))
```

Figure 5: Python code to brutal-forcely generate the Verilog code.

In this Python code a zero-padding 2D convolution operation on a 100 × 100 serialized input using a 3 ×3 serialized kernel is implemented in a brutal force way. After that we should install our generate 2D convolution code to the HANDLING state inside the FSM in FPGA module which is demonstrated in Fig. 6.

```
    end
    else if (state == DEAL_READ)
    begin…
    end
    else if (state == HANDLING)
    begin
        ans[0]<=tmp[0]*kernel[4]+tmp[1]*kernel[5]+tmp[100]*kernel[7]+tmp[101]*kernel[8]
        ans[1]<=tmp[0]*kernel[3]+tmp[1]*kernel[4]+tmp[2]*kernel[5]+tmp[100]*kernel[6]+tm
        ans[2]<=tmp[1]*kernel[3]+tmp[2]*kernel[4]+tmp[3]*kernel[5]+tmp[101]*kernel[6]+tm
        ans[3]<=tmp[2]*kernel[3]+tmp[3]*kernel[4]+tmp[4]*kernel[5]+tmp[102]*kernel[6]+tm
        ans[4]<=tmp[3]*kernel[3]+tmp[4]*kernel[4]+tmp[5]*kernel[5]+tmp[103]*kernel[6]+tm
        ans[5]<=tmp[4]*kernel[3]+tmp[5]*kernel[4]+tmp[6]*kernel[5]+tmp[104]*kernel[6]+tm
        ans[6]<=tmp[5]*kernel[3]+tmp[6]*kernel[4]+tmp[7]*kernel[5]+tmp[105]*kernel[6]+tm
        ans[7]<=tmp[6]*kernel[3]+tmp[7]*kernel[4]+tmp[8]*kernel[5]+tmp[106]*kernel[6]+tm
        ans[8]<=tmp[7]*kernel[3]+tmp[8]*kernel[4]+tmp[9]*kernel[5]+tmp[107]*kernel[6]+tm
        ans[9]<=tmp[8]*kernel[3]+tmp[9]*kernel[4]+tmp[10]*kernel[5]+tmp[108]*kernel[6]+
        ans[10]<=tmp[9]*kernel[3]+tmp[10]*kernel[4]+tmp[11]*kernel[5]+tmp[109]*kernel[6
        ans[11]<=tmp[10]*kernel[3]+tmp[11]*kernel[4]+tmp[12]*kernel[5]+tmp[110]*kernel[
        ans[12]<=tmp[11]*kernel[3]+tmp[12]*kernel[4]+tmp[13]*kernel[5]+tmp[111]*kernel[
```

Figure 6: The generated Verilog code and the installation into the FSM in FPGA module.

Finally, the generated verilog file of FPGA module should be converted into a executable binary file by the verilog simulator *Verilator*, in order to be applied in the simulation system. The following command should be executed.

verilator -Wall –cc our.v –exe sim_main.cpp && make -j -C obj_dir -f Vour.mk Vour

```
int indata[100*100];

int main(int argc, char** argv)
{
    p0 =(unsigned long long *) new((unsigned long long *)0xc0000000) unsigned long long[10];
    //printf("111111");
    p0[1] = (unsigned long long)indata;//ReadBase
    p0[2] = (unsigned long long)indata;//WriteBase
    p0[0] = 81;
    p0[8] = 1;
    p0[3] = 9;//CurrentThreadID
    p0[4] = 100*100+2;//Memory Range
    p0[5] = 4;//MemorySize
    p0[7] = 0;//Terminat
    /* Initialize array(s). */
    p0[6]=0;p0[6]=0;
    init_array (indata);
    //Strating occupy FPGA
    p0[6]=1;
    //Wait for release
    while(p0[6]);
    //Save for data
    print_array(indata);

    return 0;
}
```

Figure 7: Implementation of the work load in CPU working process.

3.3 CPU Module

In the proposed system the work load of CPU is designed to load image, transfer the data, control FPGA model and finally read back the data and save the processed image.

The following simple but effective code is used to achieved this task shown in Fig. 7, where first a control register is defined to control the read write data base address and determine the memory range. After loading the image data, the CPU module will send a signal to activate the FPGA module to work and then CPU module will wait util the FPGA module releases its occupancy, then finally read back and save the processed results.

Aiming to simplified the working load in this system, the library used in the CPU work load program is restricted to Standard C Library. To process with images, a little trick is used in the proposed system. For any input image, this image will be preprocessed and saved as a text file in a uppermost program, thus the work load can use standard library to read the image data without using other libraries like *OpenCV*. The load and save data processing in CPU module is implemented using the code in Fig. 8. The implementation of this trick in uppermost program will later be discussed in Sec. 3.5.

```
/* Array initialization. */
static
void init_array(int *array)
{
    FILE *fp;
    int i,j;
    fp=fopen("image.txt","r");
    for (i=0;i<100;i++){
        for (j=0;j<99;j++){
            fscanf(fp, "%d,", &(array[j+100*i]));
        }
        j = 99;
        fscanf(fp, "%d,\n", &(array[j+100*i]));
    }
}


/* DCE code. Must scan the entire live-out data.
   Can be used also to check the correctness of the output. */
static
void print_array(int *array)
{
  FILE *fp;
  int i, j;
    fp=fopen("image_out.txt","w");
    for (i=0;i<100;i++){
        for (j=0;j<99;j++){
            fprintf(fp, "%d,", array[j+100*i]);
        }
        j = 99;
        fprintf(fp, "%d\n", array[j+100*i]);
    }
}
```

Figure 8: Read and load processing in CPU module with a trick which transfer the image into a text file.

Finally, the CPU work load should be compiled into executable binary file in order to be loaded into the CPU module during simulation. The following command should be executed

g++ sobel-fpga.c -o sobel-fpga –static

3.4 SysCall-Emulation Simulation

In PAAS platform the top level simulation is a Python script called by the modified GEM5 simulator which define the simulation environment and system components. In the proposed system, I adopted the the single-GPU-single-FPGA prototype, for detailed setup please check the paper and user manual of PAAS [3].

Here, serveral modification should be applied in order to assign our CPU work load and call the complied FPGA module I complied. First to assign the CPU work process, we should assign the *process1.cmd*

pointing to our complied binary file as in Fig. 9.

```
process1 = LiveProcess()
process1.pid = 1100
# process1.cmd = ['tests/test-progs/polybench-c-4.2/2mm-fpga']
process1.cmd = ['tests/test-progs/my/sobel-fpga']

(CPUClass, test_mem_mode, FutureClass) = Simulation.setCPUClass(options)
CPUClass.numThreads = numThreads

np = options.num_cpus
system = System(cpu = [DerivO3CPU() for i in xrange(np)],
#system = System(cpu = [TimingSimpleCPU() for i in xrange(np)],
                mem_mode = 'timing',
                mem_ranges = [AddrRange('512MB')],
                cache_line_size = 64)
```

Figure 9: Modification in top level Python file to assign the CPU work load.

Correspondingly, the FPGA module in the original prototype should also be adjusted, which shown in the following Fig. 10.

```
system.fpga[0].clk_domain = SrcClockDomain(clock = options.fpga_clock)
system.fpga[0].clk_domain.voltage_domain = VoltageDomain()
system.fpga[0].fpga_bus_addr = 1073741824*2
system.fpga[0].size_control_fpga = 29*8
system.fpga[0].ModuleName = 'my/obj_dir/Vour'

system.cpu[0].workload = process1
system.cpu[0].createThreads()
system.piobus = IOXBar()
```

Figure 10: Modification in top level Python file to assign FGPA module.

Finally, we should call the PAAS platform to perform the syscall-emulation simulation.

./build/X86/gem5.opt configs/my/sobel-fpga.py –ruby –num-cpus=1 –num-fpgas=1

3.5 Run the Whole System in a Topmost Script

Finally, we should put every pieces of our design together and run the whole system in a single script. Therefore, I drafted the topmost script which read the target image and call the simulation and output the resulting image, which is shown in Fig. 11.

The code the structure as:

- Give the input directory, the script reads the color image, and transfers it to the 100 × 100 low resolution grayscale image, then save the image into a text file.
- Use the operation system command to call the PAAS simulation platform to execute simulation.
- This code will wait until the output text file generated by the simulator is ready, then it will read this result file and transfer it back to a image and save it.

```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='PASS')
    parser.add_argument('--input',type=str, nargs='+',
                        help='input image')
    parser.add_argument('--output', type=str, nargs='+',
                        help='input image')
    args = parser.parse_args()
    print(args)

    # read input data and transfer it to plain txt
    im = cv2.imread(args.input[0])
    im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    im = cv2.resize(im, (100,100))
    cv2.imwrite('resize.png', im)
    write_text(im)

    # system command call PAAS to run sobel-fpga.py
    os.system("./build/X86/gem5.opt configs/my/sobel-fpga.py --ruby --num-cpus=1 --num-fpgas=1")

    # wait until data ready
    while(1):
        try:
            with open('image_out.txt', 'r+') as f:
                sobely = read_text(f)
                sobely = cv2.resize(sobely, (100,100))
                cv2.imwrite(args.output[0], sobely)
        except EnvironmentError:
            # File not ready, wait
            sleep(0.01)
```

Figure 11: The Python code of topmost script. In this script, given the input and output target, the script read and process the image, call the simulation and then output the result in target directory.

4 Simulation Evaluation

I evaluated the whole system with the following command

python run_sobel_fpga.py –input lena_color.jpg –output lena_out.jpg

The input image, the preprocessed image sent to the simulator and the output image is display in Fig. 12. We can conclude that the system is run in a correct way from the result, denoted that the Sobel operator used in FPGA module is

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & 1 \end{bmatrix}$$

Figure 12: Left: Origin image; Middle: Preprocessed image sent to simulator. Right: Output image.

5 Conclusion

To conclude, in this project, a end-to-end executable Sobel filtering system on CPU-FPGA heterogeneous simulator is proposed, which amplifies the strength of both architecture and achieved correct result. Work flow of proposed sytem and detailed the implementation on code modification is thoroughly explained to highlight improvement and contribution.

Through this project, got familiar with GEM5 simulation and work flow of its cooperation between hardware accelerators. Although my project seems easy and contains no detailed analysis on memory settings, etc, the progress of put everything together and make the whole system run is not trivial, which is exact meaning of engineering.

I update the code, presentations slides and this report to github, if any other people who interest in this project they can get access to these by visiting the following link

$$https://github.com/wchengad/course\_2019\_spring/tree/master/ELEC5140$$

6 Acknowlegement

At the end of the project report, I want ot thank Prof. ZHANG for her effort and dedication on this course, which greatly helped me quickly grasp the knowledge on advanced computer architecture both theoretically and systematically. One of the spotlights of this course is the design of lab exercises which gives us a better comprehension on the course contents, I really appreciate all the course design and arrangements.

I also want to thank Mr. LIANG, Tingyuan who is the first author of PAAS [3], who created a good simulation platform and provided abundant example codes and detailed user manual. During my implementation on this project, Mr. LIANG also gave me a lot of advice and hits which all eventually helped me to complete this project although that I was total new to this course before this semester.

References

[1] Yu-Ting Chen, Jason Cong, Zhenman Fang, Bingjun Xiao, and Peipei Zhou. Araprototyper: Enabling rapid prototyping and evaluation for accelerator-rich architectures. *arXiv preprint arXiv:1610.09761*, 2016.

[2] Liang Feng, Hao Liang, Sharad Sinha, and Wei Zhang. Heterosim: A heterogeneous cpu-fpga simulator. *IEEE Computer Architecture Letters*, 16(1):38–41, 2016.

[3] Tingyuan Liang, Liang Feng, Sharad Sinha, and Wei Zhang. Paas: A system level simulator for heterogeneous computing architectures. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2017.

[4] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.

[5] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.