University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE532, Fall 2019 HW4: SIMD Wednesday, September 18

Due: Friday, September 27, 5:00PM

In this assignment, we will accelerate the streaming application from last homework using the ARM NEON vector processor. Note that there were a few modifications to the application. You can find the sources for this homework the course website along with a data set.

Collaboration

In this assignment, you work with partners that we assigned. You can find the assignment on Canvas in the *Partners* map under the *Files* section. In the event that the partner assignment does not work out, contact the instructor or TA as soon as possible. Partners may share code and results and discuss analysis, but each writeup should be prepared independently. Outside the assigned groups, only sharing of tool knowledge is allowed. See the course policies on the course web page `http://www.seas.upenn.edu/~ese532` for full details of our policies for this course.

ARM NEON

Information about the NEON architecture and datatypes is available in the ARM assembler user guide. Another section in the same guide lists the instructions. Note that not all information may be applicable to the ARMv8 architecture of the Cortex A53 processor that we are using. You are encouraged to locate other sources as needed and to share them.

Homework Submission

1. **Teamwork**

   As the difficulty of homework is ramping up, we encourage you to spend a moment planning on how to tackle the homework as a team.

   (a) Describe which tasks of this homework you will perform, which tasks will be performed by your teammate(s), and which tasks you will perform together (e.g., pair programming, where you both sit together at the same terminal). Motivate your task distribution. (5 lines)

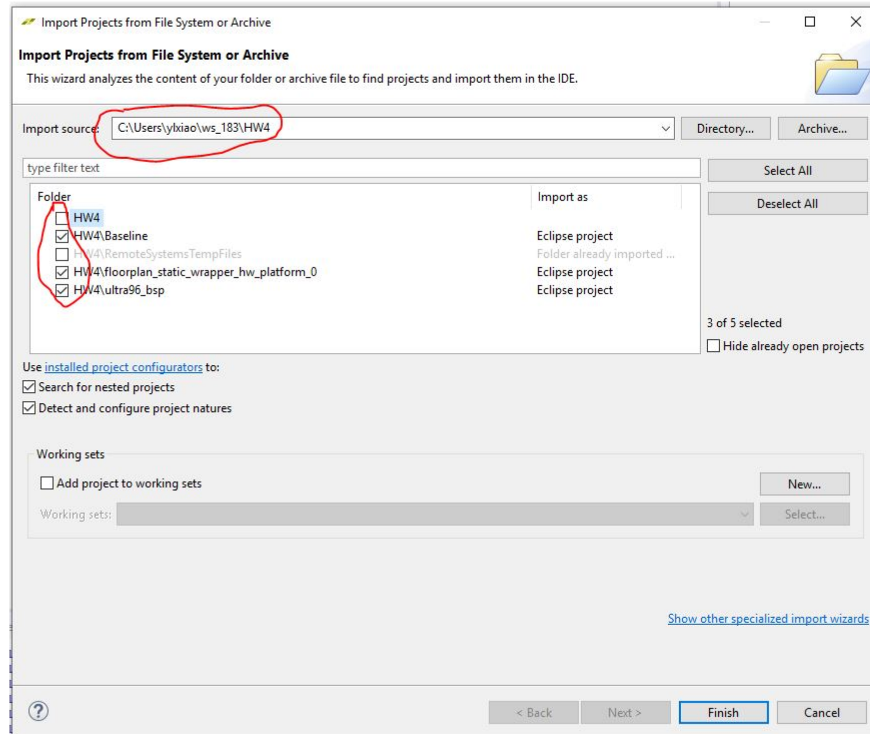

Figure 1: Import projects into SDx

(b) Give an estimate of the duration of each of the tasks. (5 lines)

(c) Record the actual time spent on tasks as you work through the assignment.

(d) Explain how you will make sure that the lessons and knowledge gained from the exercises are shared with everybody in the team. (3 lines)

2. **Compiler Optimizations**

Before we dive into the vector optimizations, we will investigate the effects of different levels of compiler optimizations. Import the project by clicking *File→Open project from file systems*. Choose the right directory for the imported project like Figure 1. In case the build output disappears (because it did to me) you can find your build output by going to: *Project→Properties→C/C++ Build→Logging* and then open the file it points to.

(a) Measure the latency and size of the `Baseline` project at the different optimization levels. As we create the project in SDK, you need to run and debug the *Baseline* project in a different way. Click *Run → Run Configurations*. Choose *Debugger Baseline (Default)*. Specify the elf file in *Application* tab as Figure 2 and configure *Target Setup* as Figure 3. Click *Run*. Put your measurements in a table like Table 1. You can change the optimization level as follows: Right-click on the project in the *Project Explorer*, and select *C/C++ Build Settings* from the popup menu. In the *Settings* tab, go to *ARM v8 gcc compiler → Optimization*, and select

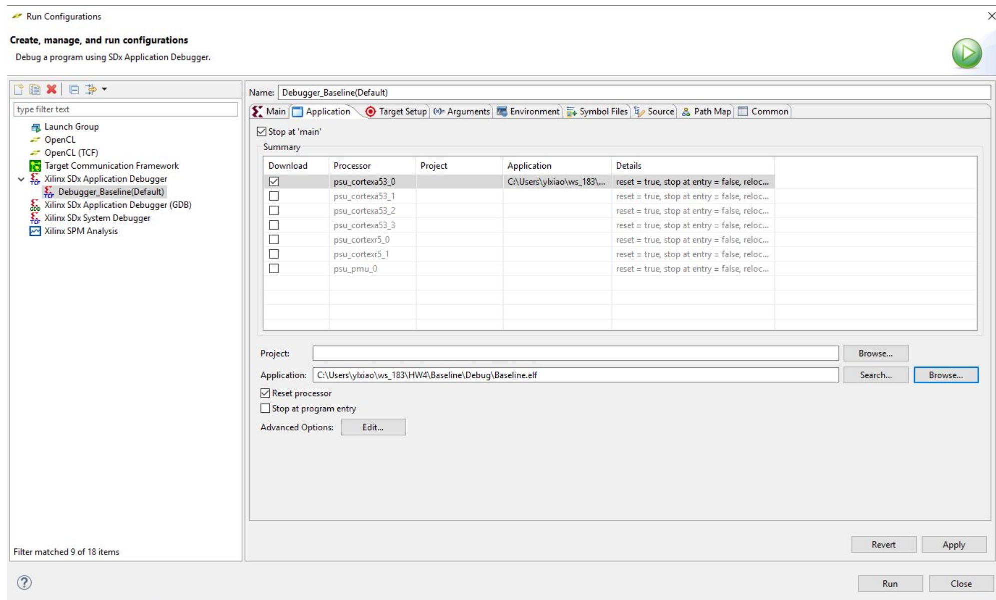

Figure 2: Debugger Application Configurations for Baseline

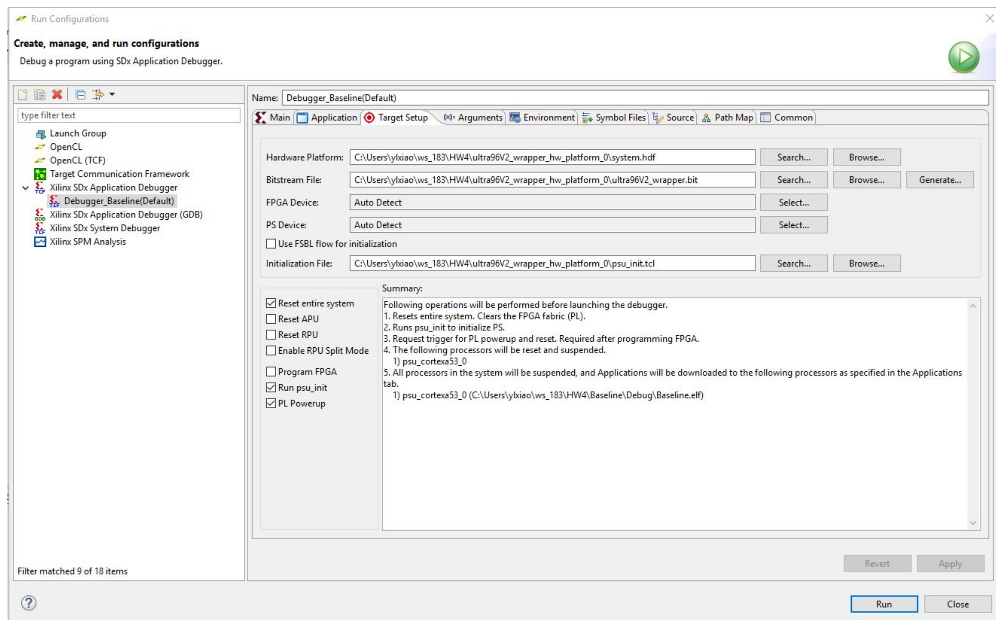

Figure 3: Debugger Target Setup Configurations for Baseline

Optimization level	Latency (ms)	Code size (bytes)
`-O0`		
`-O1`		
`-O2`		
`-O3`		
`-Os`		

Table 1: Latency and Code Size per Optimization Level

one of the optimization levels under *Optimization Level*. You can see the code size by opening the *CDT Global Build Console*. The code size is in the column *text*.

(b) Include the assembly code of inner loop of `Filter_horizontal` at optimization level `-O0` in your report. Click *Run→ Debug Configurations*. Choose *Debugger Baseline (Default)*. Specify the elf file in *Application* tab and configure *Target Setup* as Figure 3. Click *Debug* to enter the debug mode.

(c) Include the assembly code of inner loop of `Filter_horizontal` at optimization level `-O2` in your report.

(d) Based on the machine code of questions 2b and 2c, explain the most important difference between the `-O0` and `-O2` versions. (2 lines)
Hints (leading questions):

- for each case (-O0, -O2), how many times does the loop read the variable i?
- for each case (-O0, -O2), how many times does the loop read and write the variable Sum?
- why is the -O2 loop able to avoid recalculating `Y*INPUT_WIDTH+X` inside the loop body?
- what else is the -O2 loop able to avoid reading from memory? recaculating?
- how is the -O2 loop able to perform fewer operations?

(e) Why would you want to use optimization level `-O0`?
Hint: Compile the code with `-O3` and track the values of the variables `X`, `Y`, and `i` as you step through `Filter_horizontal`. (3 lines)

(f) Include the assembly code of inner loop of `Filter_horizontal` at optimization level `-O3` in your report.

(g) Based on the machine code of questions 2c and 2f, explain the most important difference between the `-O2` and `-O3` versions. (1 line)

(h) What are two drawbacks of using a higher optimization level? (5 lines)

3. **Automatic Vectorization**

   The easiest way to take advantage of vector instructions is by using the automatic vectorization feature of the GCC compiler, which automatically generates NEON instructions from loops. We will tell you how to change the compilation flag to enable the vectorization in this part. Automatic vectorization in GCC is sparsely documented in the GCC documentation. Although we are not using the ARM compiler, the ARM compiler user guide may give some more insight.

   (a) Report the latency of each stage of the baseline application at `-O3`. (Start a table that includes each stage and an overall application latency; we will continue to expand this table throughout this problem.)

   (b) Based on your understanding of the C code, which loops in the streaming stages of the application have sufficient data parallelism for vectorization? Motivate your answer. (Add a column to the table you started in Q 3a for marking suitability; add explanation in 2–5 lines after table.)

   (c) Identify the critical path lower bound for `Filter_vertical` in terms of compute operations. Focus on the data path. Ignore control flow and offset computations. (5 lines)
   Hint: Consider only the dependencies in the computation. What happens if you unroll the loops completely?

   (d) Report the resource capacity lower bound for `Filter_vertical`. Focus on the computation; you may ignore control flow and addressing computations. There are many resources that may limit the performance. (5 lines)
   Hint: As with any resource capacity lower bound analysis, you may have multiple resources and may need to consider them each to identify the one that is most constraining.
   Hint: you will need to review the NEON architecture (which we discuss in class) and reason about what resources is has available to be used on each cycle.

   (e) What speedup do you expect your application can achieve under ideal circumstances? (5 lines)
   Hint: remember Amdahl's Law; think about critical path lower bounds and resource capacity lower bounds.

   (Add another column to the table you started in Q 3a showing expected performance after ideal vectorization; separately show Amdahl's Law calculation for overall speedup.)

   (f) We will enable the vectorization in gcc. Right click the project, choose *Properties→C/C++Build→Settings→ARM v8 gcc compiler→Miscellaneous*, in "Other flags" change "nosimd" to "simd", and build the project again. Report the program size.

   (g) Report the speedup of the vectorized code with respect to the baseline. (Add two more columns to the table you started in Q 3a showing per stage and overall latency (first column) and speedup relative to non-vectorized baseline (second column)).

(h) Explain the discrepancy between your measured and ideal performance based on the optimization of `Filter_horizontal`. (3 lines)
Hint: look at the size of the multiplications in the disassembly. if the disassembly window cannot show the compelet assembly code (like .word 0x2e20c2a1), you can open the elf file. In *Project Explorer*, double click *Baseline*→ *Binaries*→*Baseline.elf*. You can use line number as clues.

Hint: to read this code, you probably need to understand the relation between Q and V registers. Perhaps useful:

- `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/ch01s03s02.html`
- `https://developer.arm.com/docs/den0024/latest/armv8-registers/neon-and-floa scalar-register-sizes`
- `https://developer.arm.com/docs/den0024/latest/armv8-registers/neon-and-floa vector-register-sizes`

(i) Show how you can resolve the issue that you identified in the previous problem. (1 line)

(j) Report the speedup with respect to the baseline after resolving the issue in both `Filter_horizontal` and `Filter_vertical`. (Add two more columns to the table you started in Q 3a showing per stage and overall speedup after resolving.)

4. **Reflection**

   Reflect on the cooperation in your team.

   - Compare your actual time on tasks with your original estimates. (table with 1-2 line explanation of major disrepancies)
   - Reflect on your task decomposition (Q 1a). Were you able to complete the task as you originally planned? What aspects of your original task distribution worked well and why? Did you refine the plan during the assignment? How and why? In hindsight, how should you have distributed the tasks? (paragraph)
   - What was the most useful thing you learned from or working with your teammate? (2–4 lines)
   - What do you believe was the most useful thing that you were able to contribute to your team? (1–3 lines)