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

ESE532, Spring 2017 HW3: SIMD Wednesday, January 25

Due: Friday, February 3, 5:00PM

In this assignment, we will accelerate the MPEG-2 encoder implementation using the ARM NEON vector processor. You can find the encoder source for this homework again on the course website. With respect to homework 2, the motion estimation code has been further simplified. We replaced the algorithm with an exhaustive search that locates the 16×16 block that is most similar to the current macroblock. The `dist1` function that we encountered before has been renamed to `dist` and simplified. It computes the sum of absolute differences (SAD) of the luma values of the two blocks.

Collaboration

Work with assigned partners.[1] Writeups should be individual. Partners may share code and profiling results and discuss analysis, but each writeup should be prepared independently.

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 ARMv7 architecture of the Cortex A9 processor that we are using. You are encouraged to locate other sources as needed and to share them.

Building and Running

To enable generation of code for the ARM NEON, we must follow these steps:

1. Launch SDSoC, and open the workspace that you used before. You can also create a new workspace.
2. Create a new BSP and Hardware Platform Specification similar to the one in homework 2. Enable `xilffs` and set `_USE_STRFUNC` again to 1 in the BSP settings dialog.

[1] To be posted under Files in canvas. Note assignments are different from previous homeworks.

3. Select `ps7_cortexa9_0` in the pane on the left side of the BSP settings dialog. Append `-mfloat-abi=hard -mfpu=neon` to the current value of `extra_compiler_flags`. Adding the value to the end ensures that any earlier occurrences of the respective settings are overridden.

4. Build the BSP project.

5. Create an empty *Application Project* in SDSoC. The kind of project that we used before (described as *SDSoC Project* in SDSoC) does not support NEON at the moment. The reason is that SDSoC Projects always add certain compiler flags that take precedence over our flags and disable compilation for NEON. Select your newly created BSP and hardware platform specification in the *New Project* dialog. Press *Finish* to dismiss the dialog. SDSoC will ask whether you want to open the *C/C++* perspective. The differences between *SDSoC* and *C/C++* perspective are probably irrelevant for the remainder of the instructions, but to avoid problems, stay in the *SDSoC* perspective.

6. Import the code into SDSoC into the `src` subdirectory of your project.

7. Increase the heap size in the linker script `lscript.ld` to `0x3800000`. The script specifies how code and data sections are arranged in memory. The given value guarantees that there is enough memory space allocated for all dynamic memory allocations (such as `malloc`).

8. Add the `src` directory of your project as include path to the directory. You can do this by opening the *Properties* dialog for you project. Select *C/C++ Build* → *Settings* in the pane on the left. After selecting the *Tool Settings* tab, choose *ARM v7 gcc compiler* → *Directories*, and add the path to the list. Keep the dialog open for the next step.

9. Add the math library (`libm.a`) to your project. Choose *ARM v7 gcc linker* → *Libraries*, and simply add `m` to the list on top. The library is in the default library search path, so no need to add a library path. Keep the dialog again open.

10. Enable NEON code generation. Select *C/C++ Build* → *Settings* in the pane on the left. After selecting the *Tool Settings* tab, choose *ARM v7 gcc compiler* → *Miscellaneous*. Append `-mfloat-abi=hard -mfpu=neon` to the string shown in the *Other flags* textbox. Switch to *ARM v7 gcc linker* → *Miscellaneous*, and add the same string to the *Linker Flags*. Do not close the dialog.

11. Enable compiler optimizations. Switch to the *-O2* optimization level under *ARM v7 gcc compiler* → *Optimization*. Now, you can close the dialog.

12. We will use the same video and configuration files as in homework 2. Make sure they are on the SD-card, and place the card in the ZedBoard.

13. Build the project.

14. You can run the application as before, except that you should select the *System Debugger* instead of the *SDSoC Debugger*.

Homework Submission

1. **Learning and navigating the tools**

   Document all the problems you encountered in working the assignment and how you solved them. For each, include:

   (a) Step (if applicable)

   (b) OS you were running

   (c) Error message or unexpected behavior you encounted

   (d) How the problem was resolved

   (e) How you found the solution (e.g., how did you experiment or reason through an answer, which document had the answer, where on the web did you find an answer, which student or TA was able to point out the answer)

2. **Self Learning**

   Document any additional resources you used to understand NEON, NEON intrinsics, compiler optimizations, and automatic vectorization.

3. **Teamwork**

   (a) Document the work sessions with your partner (where, when, duration).

   (b) How did you collaborate on the work? (pair programming is acceptable, explain)

   (c) What tricks or techniques did you learn from your partner?

   (d) Provide highlights for cases where working together allowed you to understand deeper, combine insights, avoid pitfalls, and/or build confidence in your directions and solutions.

4. **NEON Intrinsics**

   We will accelerate the `dist()` function using intrinsics. An intrinsic behaves syntactically like a function, but the compiler translates it to a specific instruction that is inlined in the code. The Neon intrinsics are listed in the ARM documentation.

   (a) What is the range of

      i. the difference between two pixels

      ii. absolute value of the difference

      iii. sum of all absolute differences

   (b) What is the maximum number of sums that we can store in a NEON register during an SAD computation?

   (c) Using NEON intrinsics, accelerate the `dist()` function. Include the accelerated function in your report.

(d) Report the time for the `dist()` for both the original version and NEON-intrinsics version and the speedup obtained. Original version is original code compiled with `-O2` optimization, but no NEON intrinsics.

(e) Explain how your NEON-intrinsic-optimized code is faster and why it achieves the speedup that it does.

(f) How good is your optimization and how do you know? Identify the resource bound and latency bound and relate the performance of your solution to those bounds.

(g) Report the total execution time of the accelerated encoder.

(h) What speedup does your accelerate coder achieve compared with the original code, compiled with `-O2` optimization, but no NEON intrinsics?

(i) Explain the relation between the full encoder speedup and the `dist()` speedup in terms of Amdhal's Law.

5. **Compiler Optimizations**

(a) Report the latency of your vectorized function without optimizations (`-O0`).

(b) Show the machine code of your vector instructions without optimizations. The easiest way to do this is by executing the code until it reaches the intrinsics. Now, open the *Disassembly* view.

(c) Report the latency of your vectorized function with optimizations (`-O2`).

(d) Show the machine code of your vector instructions with optimizations.

(e) Explain what the compiler does differently between the `-O0` and `-O2` versions. Why is it correct for the compiler to perform these transformations? Why is it beneficial for the compiler to perform these transformations?

(f) Why is it important to enable optimizations?

6. **Automatic Vectorization**

Writing vector instructions using intrinsics or assembly can be tedious. Fortunately, the GCC compiler supports automatic generation of NEON instructions from loops. It does not remove the burden from the developer entirely, however, because the number of loop structures that a compiler can recognize is limited. 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) Add a compiler flag for automatic vectorization. We recommend adding `-fopt-info-vec-optimized` as well such that you can easily see which loops are vectorized.

(b) Still working with the original version of `dist()`, what benefit, if any, is the compiler able to achieve on the original, unmodified code.

(c) Create a new `dist()` function that does not use intrinsics, but can be vectorized by the compiler. Attempt to maximize the performance obtainable using the compiler; this may require experimentation. Include the code in your report.

(d) Report the execution time after automatic vectorization for both `dist()` and the total encoder.

(e) Explain the key changes you made to enable automatic vectorization from the compiler and why each was important.