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

ESE532, Fall 2018 HW3: Thread Parallel Wednesday, September 12

Due: Friday, September 21, 5:00PM

In this assignment, we will map the application from homework 2 on the two ARM cores of the Zynq platform. We will encounter different parallel implementations and analyze their impact on performance.

Collaboration

You can find the partner 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.

Communication

We will divide the work into threads that run on different processors. To distribute and coordinate the work, the processors must communicate. We will let the processors communicate via the SDRAM. This is possible because the SDRAM is mapped into the address spaces of both processors. In order to share data, the processors have to agree on the location and organization of the data. In the provided code, we have mapped a small structure (`com_area`) at a fixed address in memory, which we use for communicating pointers to shared memory areas and synchronization. We also have to make sure that both processors respect each other's private memory areas. We have already done that in the provided code by mapping private code and data of both processors at different locations. Sharing SDRAM is complicated by the fact that SDRAM is cached in L1 and L2 caches. Data that one processor attempts to write to SDRAM may not have been written to SDRAM yet, but instead remain in the private L1 cache of the processor. When another processor reads the same memory location, it may observe an old value. Fortunately, our Zynq has a Snoop Control Unit, which bypasses data directly between processors as needed to maintain a consistent view of the SDRAM. Therefore, this is no concern.

Another problem that we face when we communicate via shared memory is that the reading processor should not start reading the memory until the writing processor has completed

writing the data. In other words, we need a form of synchronization between the cores. Design of synchronization functions is a rather complex subject, which is dealt with in other courses such as CIS 501, so we will just provide a few functions for this purpose without discussing their implementation:

- `Initialize` prepares a processor core for communication.
- `Wait_for_start` blocks processor core 1 until core 0 calls `Start_core_1`.
- `Start_core_1` stops a blocking `Wait_for_start` function on core 1.
- `Wait_for_core_1` blocks processor core 0 until core 1 calls `Return_to_core_0`.
- `Return_to_core_0` stops a blocking `Wait_for_core_1` function on core 1.

These functions have known shortcomings, but they should be sufficient for this assignment.

Note that you may have to adapt `Initialize` and `com_area` if you want to communicate more pointers between the processors. Other than that, you should not have to change these functions.

Obtaining and Running the Code

In the previous homework, we dealt with a streaming application that compressed only one picture. For this homework, we will use the same application, except that it will take a video stream instead of a single picture. The input stream is available here. We provide an archive with projects as starting point, which can be obtained from here.

The parallel implementations, which start with `Coarse` and `Pipeline`, are split over two projects. Each project has the suffix `core_0` or `core_1`, which indicate on which ARM core they will run. We have provided debug configurations as well. It is essential that the code for core 1 is started before the code of core 0. Otherwise, the initialization code of core 1 overwrites certain data structures of core 0. The provided debug configurations (*Debug coarse* and *Debug pipeline*) take care of this by putting a breakpoint at the start of core 0's entry point. You only have to resume execution of core 0 to run the code. Synchronization in the initialization function guarantees that core 1 does not start processing data before core 0 has finished initializing.

You may notice that there are two more projects, `zed_hw_platform` and `core_1_bsp`. The reason is that we created the application for core 1 as application project because an SDSoC project does not allow us to adapt the memory layout of the application. We have to change the memory layout to avoid that code and data for core 1 overlaps with core 0. Every application needs a Board Support Package (BSP), which provides low-level routines to access the hardware. An SDSoC project automatically includes a BSP. For an application project, we have to create one manually. In addition, we need a platform description, which is in the `zed_hw_platform` project.

Homework Submission

Your writeup should follow http://www.seas.upenn.edu/~ese532/writeup_guidelines.pdf. Your writeup should include your answers to the following questions:

1. **Baseline**

   (a) Determime the throughput of `Baseline` in pictures per second. This is your baseline. Ignore overhead such as loading and storing pictures for this and the following questions. (1 line)

2. **Coarse-grain parallelism** We will parallelize the application by processing half of each picture on core 0 and the other half on core 1, a form of coarse-grain, data-level parallelism. There are two projects, `Coarse_core_0` and `Coarse_core_1` in the provided workspace, which form the starting point of your implementation. We have parallelized `Scale` already for you.

   (a) Can we parallelize all streaming functions in our application, i.e. `Filter_horizontal`, `Filter_vertical`, `Differentiate`, and `Compress` in the same way as `Scale`? Motivate your answer. Assume that we synchronize our cores between each producer-consumer pair. (3 lines)

   (b) What speedup do you expect from parallelizing the functions that you considered parallelizable in the previous question? [Include an equation for the expected parallel runtime and show the equation you use for computing the speedup as well as your final, numeric result.] (3 lines)

   (c) Complete the implementation by parallelizing the functions that you considered parallelizable in the previous question. Provide the relevant sections of code in your report.

   (d) Measure the throughput of your parallel implementation. (1 line)

   (e) Compare your measurement with your expected speedup. How much overhead does your implementation have? (1 line)

   (f) To what can we attribute most of the overhead? (1 line)

   (g) Why is it necessary to synchronize between every producing and consuming function? (7 lines)

   (h) Create a performance model for execution that accounts for synchronization.

      i. Expand your equation for parallel runtime from Part 2b by adding terms for the number of synchronizations and the time for each synchronization event.

      ii. Identify the number of synchronization events for the current implementation.

      iii. Based on the equation from (i), your count from (ii), and your expected runtime that did not include synchronization (Part 2b), estimate the time for each synchronization event.

(i) Reflecting on the model above, how could we reduce the synchronization overhead by reducing the number synchronization points while leaving our paradigm of processing each half of the picture on a core largely intact? (3 lines)

(j) Based on your model from Part 2h, estimate the throughput of your change from Part 2i.

Note: In Parts 2h through 2j, we've guided you through modeling an effect previously omitted and estimating a potential optimization. When you actually perform the optimization (which we are not asking you to do here), you will further want to validate and potentially refine the model and model paramerters. This is a technique you will want to generalize and adapt for other effects that you may encounter later in the term.

3. **Pipelining** As an alternative to coarse-grain, data-level parallelism, we will investigate a pipelined implementation in this question. The initial implementation, which can be found in the projects `Pipeline_core_0` and `Pipeline_core_1`, maps `Scale` and `Filter` on core 1 and `Differentiate` and `Compress` on core 0. The provided stream has only 10 frames, but assume in your performance computations that you are dealing with a stream of infinite length.

   (a) Report the throughput of the pipelined implementation in pictures per second. (1 lines)

   (b) What is the best performance that one could theoretically achieve with a pipelined mapping of the streaming application?

   (c) Describe the mapping that achieves the best performance.

   (d) Adapt the implemention such that it utilizes your new mapping. Include the sections of the code that you modified in your report.

   (e) Report the throughput of your new application in pictures per second.

   (f) Let's investigate the performance if we incorporate the optimized pipeline in a video broadcast server. The input data is read from the USB interface of the ZedBoard. 75% of traffic is video traffic that is compressed using our pipeline. The remaining 25% is other traffic that we protect with an error correction code (ECC) that adds 10% overhead in size. The ECC unit processes 20 MB/s. The output of the ECC unit and compression pipeline are output to a Gigabit Ethernet port of the ZedBoard.

      i. Draw a streaming dataflow diagram for the network server. Indicate throughput and data transfer ratios where applicable.

      ii. What is the maximum throughput that the server can achieve? (10 lines)

      iii. Where is the bottleneck? (1 line)

      iv. How much smaller do we have to make the kernel (FILTER_LENGTH) of `Filter` to move the bottleneck? (7 lines)