Lab 2: Using GEM5 for architecture exploration in multi-core processors

Glenn Bergmans

5SIA0 - 2017/2018

Organisation

In this lab, youll get some hands-on experience using the GEM5 simulator (gem5.org). The main goal of this lab is to learn how a simulator can be exploited for exploring the impacts of architectural modifications on performance.

This document contains four main sections. In the first section, we will discuss how to set up the experiment environment. The second section guides you through modifying an existing processor model. The third section will show how to upgrade to a multi-processor platform and how to write multithreaded code. The last section will describe the assignment and deliverables.

1 Setup the environment

In this section we explained the steps that you need to follow to run some examples on gem5.

We have prepared the image file of virtual machine with ubuntu 16.04 and all the tools installed which you can directly use. However, you can install the gem5 with all dependencies on your machine if you like but support from us wont be strong in this regard.

1.1 Virtual Machine

The `gem.ovf` file which is given is the image file for virtual machine. You need to simply download it and import it to your virtualbox (it should also work with vmware although we have not ever checked it). Ubuntu 16.04 is installed on the virtual machine with username and password both set as `eca`.

In home directory, there are folders `gem5` and `eeg` which contain the gem5 source files and the EEG application and data respectively.

The `~/eeg` directory has the following structure:

- The `C` directory has all source files of the application. The main file is `eeg.c` and most feature components have their own files.

- The file `EEG.cvs` has data from a 23-channel measurement. We'll only use the first 256 samples from this dataset.
- The Makefile is used for compiling the EEG application.

1.2 Compiling the EEG application

In order to compile the application for the ARM platform, execute `make` in the `~/eeg` directory. This creates a new file called `eeg.arm` which is the binary executable. You can also compile for your native platform (which can be very useful in testing your code functionally, before running a full simulation) by executing `make native`. This creates a binary called `eeg` and can be executed with `./eeg`.

When using OpenMP, it might be useful to specify how many threads you'd like to create. By default this is 1. For this, use e.g. `NUM_THREADS=4 make`.

1.3 Running the EEG application inside GEM5

The gem5 must be built separately for simulating different microprocessors i.e x86, arm etc. We have built the gem5 for ARM and the binary file for that is accessible at: `/home/eca/gem5/build/ARM/gem5.opt`. We also created an alias for this executable called `gem5.opt`.

In order to simulate an architecture and run an application you need to provide two things: 1-the system configuration script and 2- the binary of application for running on the simulator. Optionally you can also set the number of cores (this defaults to 1). The following example shows how this works:

```
gem5.opt PATH_TO_CONFIG_FILE -n 4 -c PATH_TO_BINARY_FILE
```

For running an application that is compiled for a single core on a single core A9 configuration, use e.g. `gem5.opt ~/gem5/configs/example/armA9.py -n 1 -c eeg.arm`. Note that it is easiest to make your own configurations in `~/gem5/configs/example/` for this allows you to easily link to other example files.

2 Benchmark a single core implementation

We've supplied you with a single thread implementation of the EEG application and with a single ARM A9 core configuration for GEM5. These will be used as a baseline for you to optimize. First run the code natively in order to get the correct output by first executing (in `~/eeg`) `make native` and then `./eeg`. This should finish near instantly and print out 14 feature numbers: these numbers form the array of features that have been extracted and say something about the measured signal. You might want to save these numbers to some file in order to compare the correct features with future implementations in order to verify functional correcness.

Next, run the eeg application on the supplied A9 core. First build the ARM binary (again in `~/eeg`) with `make` and then run this in GEM5 by executing `gem5.opt ~/gem5/configs/example/armA9.py -n 1 -c eeg.arm`. Execution might take up to an hour depending on your hardware and VM configuration. After the compilation finishes, verify the features with the ones you generated with the native executable for functional verification.

2.1 Simulation results

After a succesful simulation, GEM5 will put the simulation results in the `m5out` directory (you can override the location with the command line). You can check this file for relevant numbers about the simulation, such as the number of cycles it took to complete the program and the number of cache misses and hits. There are also detailed numbers on the instruction mix, which might be insightful for optimizing a processor core. Go through the list of statistics and see which information is available to you. Maybe you want to take note of certain keys, so you can easily search for them in the stats file in the future.

2.2 Power estimation

This section is still under construction, and will be updated shortly

3 Memory subsystem

The following two lines in configuration script specifies the l1 instruction cache size and associativity parameter:

```
options.l1i size = "64B"
options.l1i assoc = 1
```

Similarly the following two lines define the l1 data cache size and associativity parameter:

```
options.l1d size = "64B"
options.l1d assoc = 1
```

The L2 cache can be disabled with `options.l2cache = 0` or enabled with `options.l2cache = 1` and the its size and associativity parameter can defined with the following lines:

```
options.l2 size = "256kB"
options.l2 assoc = 4
```

4 Cache assignments

4.1 Assignment 1.1: L1 cache size

Disable the L2 cache and change the L1 instruction cache and data cache sizes and evaluate their impact on the overall performance for ARM A9. Assume that the caches are direct mapped.

4.2 Assignment 1.2: Associativity level

Still keep the L2 cache disabled and set the L1 instruction and data cache size to 2kB. Change the associativity level for both L1 caches from 1 to 8 and evaluate its impact on the overal performance.

4.3 Assignment 1.3: L2 cache size

Pick the L1 instruction and data cache size of 1kB and assume direct mapped. Change the direct mapped L2 cache size from 2kB to 16kB and evaluate its impact on the overall performance.

5 Upgrade the A9 to an A15

GEM5 allows for very detailed models of processors. Using the GEM5 configuration file, you can change the number of processing units in the processor and some details about them. So far you've been working with a model close to the ARM A9 core.

5.1 Assignment 2.1: ARM A15

The given configuration file `armA9.py` that describes the functional units of an ARM A9 processor, you need to change the configuration so that it describes the ARM A15 functional units. The ARM A15 function units specification is as follows:

- An Out-of-Order CPU capable of fetching, decoding, issuing, and dispatching 3 instructions per cycle. 2 integer ALUs
- 1 integer multiplier
- 1 integer divider
- 1 memory read/write port

5.2 Assignment 2.2: L1 cache size

Disable the L2 cache and change the L1 instruction cache and data cache sizes and evaluate their impact on the overall performance for ARM A15. Assume that the caches are direct mapped.

6 Multicore implementation

Next we can extend our configuration to a multicore system. In the `~/eeg/config` directory, an example is given that uses three A9 cores called `arm-multicore-A9-A9-A9.py`. We will use OpenMP to create a simple multi-threaded version of our application: we will not go over the design choices for parallelizing the code, but rather give simple examples that get you going on your own.

6.1 Basic OpenMP

All OpenMP libraries have already been included in the given code. In `C/eeg.c` we can add the required code and pragma's to create a simple multicore version of the EEG algorithm. In the `main` function in this file there is a for loop that loops over all channels and calculates the features for each channel. Since each loop iteration is completely independent of all other iterations, this is very simple to parallelize.

Add the function `omp_set_num_threads(NUM_THREADS);` somewhere at the top of the main function. This function tells OpenMP how many threads to use (unless otherwise specified). We'll set `NUM_THREADS` as a compilation parameter so we can easily change the number of threads we use. Then add the `#pragma omp parallel for` right above the for loop that calculates all features. This is all code required to create a simple parallel version. Save the file.

Next, let's recompile the code. You might want to make a backup of the single core binaries for reference later, but this is up to you. Run `NUM_THREADS=3 make native` to make a native version and test its correctness by executing `./eeg`. Next compile for ARM with `NUM_THREADS=3 make` and run your code using `gem5.opt ~/gem5/configs/example/arm-multicore-A9-A9-A9.py -n 3 -c eeg.arm`. Since GEM5 in SE mode always runs on a single core, simulation time will not improve much, although the simulated time hopefully is shorter.

Evaluate your results. What has changed in the number of cycles? What can you say about the cache usage?

6.2 Advanced OpenMP

In order to get more familiar with OpenMP, this section describes an example of loop parallelization that is a little more complex. If you're already familiar with OpenMP, you may skip this example.

Right under the the loop we just parallelized is a different loop that calculates the mean for each feature. This is a double for loop that loops over all channels and over all features. Note that we can not parallelize this code by splitting the outer for loop (the one that loops over the channels), since we would then create a race-condition on `favg[j]`. However we can create a parallel execution where different features will be averaged in different threads, so first we need to change the loop order, such that the `FEATURE_LENGTH` loop becomes the outer loopo and the `CHANNELS` loop becomes the inner loop.

If we then add the same pragma as before, namely `#pragma omp parallel for`, and run a simulation, you will probably see incorrect results. Why is this? How can you fix this? (Hint: shared vs private variables).

6.3 Assignment 3: Heterogeneous platform

Based on the various configuration examples, create a multicore platform consisting of two A15 cores and two A9 cores. Create a shared L2 cache of 16kiB and a private L1 cache of 1kiB and assume direct mapped. Run a simulation with the algorithm compiled for 4 threads and evaluate the results. Calculate the EDP.

7 Final assignment

For the final assignment, optimize the platform you created in Assignment 3. You can use for example more advanced parallelization strategies using OpenMP, improve the algorithm implementation, optimize the cache sizes and layout and change the configuration of the cores. In your report, state which configurations you've explored and why. Try to design the right configuration based on analysis of the code/algorithm and explain your reasoning. It might be very insightful to simulate only part of the code, in order to determine which parts are most computation heavy and to get more detailed numbers for sections of the code.

8 Deliverables

Hand in a report with your solutions to the three assignments and your optimized version of the code and configuration for the final assignment. Create graphs of your results and include the configuration files and modified sources in your submission.