332 Advanced Computer Architecture

Assessed exercise 1: instruction-level-parallelism in the Fast Fourier Transform

This is the first of two equally-weighted assessed coursework exercises. Working individually, do the exercise and write up a short report presenting and explaining your results. Submit your work in a pdf file electronically via CATE.[1] The CATE system will also indicate the deadline for this exercise.

Background

Fourier analysis converts a sample of a periodic signal (such as a sound) into the frequency domain, yielding a spectrum giving the energy present at each distinct frequency. The phase of the constituent frequencies is captured as the spectrum is presented using complex numbers. Thus the Fourier transform is invertible - we can get back to the original input signal.

Computing the Fourier transform has an immense range of applications, including image compression, convolutional neural networks, solving partial differential equations, signal processing, audio watermarking, and many more.

The Fast Fourier Transform is, arguably, one of the fundamental big ideas in algorithm design. The simplest version, seen in the code we use here, works on a power-of-two-sized input array, and works in a divide-and-conquer fashion, recursively splitting the input into odd- and even-halves. This leads to an $O(n \log n)$ complexity.

This exercise involves running an implementation of FFT under a processor microarchitecture simulator, called SimpleScalar (`http://www.simplescalar.com/`).

SimpleScalar's toolchain does not include maths libraries, so I had to find an implementation of $sin(x)$ and $cos(x)$. There are various ways to do this; I chose an iterative algorithm, though I think you can do a bit better using piecewise approximation and a lookup table of polynomials for each piece. Of course some CPUs have sine and cosine instructions, and hardware implementations (this is common with GPUs, though such instructions may not achieve full accuracy).

Your task is to find a configuration for a processor core that runs this benchmark using the minimum *total amount of energy.*

Log into a Department of Computing Linux teaching lab desktop machine

Although it is possible to do this exercise using your own machine, it's actually recommended to use a DoC lab machine - not least so that you can leave it running for hours while you use your own machine for something else!

You will need to connect via ssh remotely. First connect to one of the login servers `shell1`, `shell2`, `shell3`, `shell4` — for example as follows:

[1] `https://cate.doc.ic.ac.uk/`

```
ssh shell2.doc.ic.ac.uk
```

Then, from there, connect to one of the DoC lab desktop machines. You are recommended to try `texel1`, `texel2`, ... `texel44`. For example:

```
ssh texel19
```

Not all the `texel` machines are available — keep trying til you find one (ideally an idle one) [2]

Running the program natively

Copy the exercise's directory tree to your own directory:

```
cd
cp -r /homes/phjk/ToyPrograms/ACA21-22/fft ./
cd fft
```

Build the program:

```
make
```

This creates several runnable binaries, both for x86 and for SimpleScalar. Now (in this directory) you can run the program:

```
./fft.x86-test
```

This "test" version prints out what it's doing. You should get a printout showing some sample input, FFT, and inverse-FFT results. To run a larger example, use the "benchmark" version of the program:

```
./fft.x86-benchmark
```

The problem size is controlled by a macro "SCALE" which you can change in the Makefile if you wish.

Running the test program under the Simplescalar-Wattch simulator

You can now run the program using the simulator using a script that is provided to you:

```
./run-wattch
```

Here we use a much-reduced-size problem size as the simulator is rather slow - this takes about 20 seconds. There is a danger that using such a small size might lead to misleading

[2] a list of all the DoC lab machines is here: `https://www.doc.ic.ac.uk/csg/facilities/lab/workstations`.

results, so some care is needed. You can modify the Makefile to use a larger (or even smaller) problem if you wish, but please report results for the default problem size.

The output from running `run-wattch` consists of four parts:

1. A record of the configuration being simulated (this also documents how to use command-line arguments to change the configuration).
2. The output from the simulated program's execution.
3. Simulation statistics from the `sim-outorder` simulator, given the number of simulated clock cycles the program took to run, and many statistics on utilisation of hardware resources — including cache miss rates, branch misprediction rates, etc.
4. Statistics from the `wattch` power estimator. Wattch uses the `sim-outorder` configuration to estimate the energy consumed by the execution. To do this, it uses the statistics to determine how much each unit in the configuration is actually being used.

For your convenience the `scripts` subdirectory contains some example scripts for running the benchmark — see `varyarch`, `varyarch-energy`, `varyarch-energy-2params` and `varyarch-energy-3params`, explained below.

Studying microarchitecture effects

Study the effect of various architectural features on the performance of the benchmark. We are using the SimpleScalar simulator; further details on the processor model can be found at `http://www.simplescalar.com/docs/users_guide_v2.pdf`.

Parameters interact - RUU and LSQ size for example

Vary the RUU size between 2 and 256 (only powers of two are valid). (Use the provided script `varyarch` - we are looking at the "sim_IPC" metric, the number of instructions completed per clock cycle, as modelled by the simulator).

Plot a graph showing your results. Explain what you see. Now do the same but look at the total energy consumed — use the script `varyarch-energy`.

Now try varying the "lsq" size (you might use the `varyarch-energy-2params` script). You will see that the lsq size and the ruu size parameters interact.

Bottleneck: What microarchitectural structure is limiting the simulated execution speed when running this application in the default simulated architecture? Under what circumstances is it the RUU size? Under what circumstanes is it the LSQ size?

(You may find it helpful to refer to the SimpleScalar documentation to see what these two parameters actually mean - see for example slides 42 and 43 in `http://www.simplescalar.com/docs/simple_tutorial_v2.pdf`).

Race-to-finish doesn't always win: What combination of RUU size and LSQ size leads to the lowest total energy comsumption to complete the computation? The simulator reports

the total energy required as “total_power_cycle_cc1” (again you can use the `varyarch-energy-2params` script).

You will observe that minimising execution time may or may not minimise energy consumption.

Minimising total energy:

Can you find the “sweet spot” architecture that runs this program with optimum energy efficiency? To be precise: find the simple-scalar configuration which finishes the computation in the minimum total energy. For this step you are invited to change the cache configuration as well as the microarchitecture configuration (but beware that some parameters correspond to unrealistic choices — see below). See the section below on how SimpleScalar’s Wattch extension estimates the energy utilisation.

Write your results up in a short report (**not more than four pages** including graphs and discussion). The best solutions will be the ones which report a systematic strategy to find the optimum implementation, and which offer some insight and analysis of the results that you observe.

What to hand in: Submit the pdf of your report *and* a bash script that runs the simulation of the best configuration you were able to find. The CATE system insists that your filenames are exactly `FFT-report.pdf` and `FFT-bashscript.sh`.

Tools and tips

The first output to look at from the simulator is “sim_cycle” - the total number of cycles to complete the run. It’s often also useful to look at “sim_IPC”, the instructions per cycle - provided you always execute the same number of instructions. The time taken to perform the simulation “sim_elapsed_time” simply tells you how long the simulator took.

Other outputs from the simulator can be helpful in guiding your search - eg “ruu_full”, the proportion of cycles when the RUU is full.

How wattch reports energy utilisation

Wattch reports an estimate of the total energy (roughly in nanojoules) required for the computation, tagged with "total_power_cycle_cc1" (the comment on this line is misleading - it *is* energy, not power).

Wattch is documented in this article,

`http://www.eecs.harvard.edu/~dbrooks/isca2000.pdf`

Figure 5 in the article shows the impact of Wattch’s three different clock gating models - "cc1", "cc2" and "cc3". For this exercise you are invited to use "cc1" (this is what the script "varyarch-energy" reports). "cc2" and "cc3" are approximate models reflecting more optimistic/ambitious circuit-level implementations of power optimisation. "cc1" assumes that each unit is is fully on if any of its ports are accessed in that cycle. "cc2" assumes power scales linearly with port usage. "cc3" assumes ideal clock gating where the power scales linearly with port usage as in "cc2", but disabled units are entirely shut off.

Parameters that should not be changed

You are invited to change both the microarchitecture and the cache configuration. Note that the “perfect" branch predictor is not a real thing — it is a simulation trick to help you understand the scope for performance improvement through better branch prediction. Similarly, the “-issue:wrongpath" option must be true — setting it false is not realistic. Note that some simplescalar parameters make arbitrary changes to assumptions about the latency of operations — they are not architectural choices — eg “-fetch:speed" (speed of front-end of machine relative to execution), and “-fetch:mplat" (extra branch mis-prediction latency). Please do not change “mem:width" and “res:memport". The “spec_update" parameter changes at what stage the predicator can be updated (where ID is at the decode stage and WB is at the writeback stage) — but its power estimation appears to be buggy, so don’t use it. If you do find more anomalies, let us know!

Paul Kelly, Imperial College London, 2021