CSE211: Compiler Design

Homework 3: Parallelism
Assigned: Nov. 3, 2021
Due: Nov. 17, 2021

Preliminaries

1. This assignment requires the following programs: `make`, `bash`, `python3`, and `clang++`. This software should all be available on the provided docker. You will additionally need the `z3-solver` pip package for Python.
2. Find the assignment packet either online at: `https://sorensenucsc.github.io/CSE211-fa2021/assignment_data/homework3_code.zip`.

   or it is stored in the docker in `assignment_data/homework3_code` if you pull.
3. This homework contains 3 parts. Each part is worth equal points.
4. For each part, read the *what to submit* section. Add any additional content to the file structure. To submit, you will zip up your modified packet and submit it to canvas.

1 Loop Unrolling Independent Iterations for ILP

Here we will consider `for` loops with independent iterations. However, each iteration will contain a chain of *dependent* instruction. This chain is intended to impede the processor's ability to utilize ILP, either through pipelining or superscalar components. The length of dependent instruction chains is a parameter of the program. Your assignment is to unroll the loops, first doing each iteration sequentially, and then interleaving the instructions from different iterations. Interleaving instructions should allow the processor to exploit more ILP, which should appear in timing experiments. You will measure the execution time of various dependent chain lengths and unrolling factors.

This assignment is based on a C++ loop structure, parameterized by a dependency chain length N, that looks like this:

```
void loop(float *a, int size) {
  for (int i = 0; i < size; i++) {
     float tmp = a[i];
     tmp += 1.0f;
     tmp += 2.0f;
     tmp += 3.0f;
     // and many more
```

```
    tmp += N;
    a[i] = tmp;
  }
}
```

A few things to note about this loop: the dependency chain cannot be re-ordered or statically pre-computed in the compiler because floating point operations must be done in order (i.e. they are non-associative). So the compiler must must produce an ISA instruction for each of the addition operations. The code will generate as many addition operations as specified by the chain length N.

I will provide a skeleton python code that performs the following:

1. it generates the reference loop
2. it generates unfinished function outlines for the functions you need to implement
3. it puts these functions in a C++ wrapper that will print timing information about the execution of the functions.
4. it compiles the code
5. it runs the code

The python script takes two command line args, the length of the dependency chain and the unroll factor. Run with `-h` to view the argument specification. You can view the generated C++ file in `homework.cpp` (it will change each time you run the python script). You can specify various chain lengths to see how the reference loop changes. The unroll factor currently does nothing because that is your job to implement. Initially, your C++ code will compile and run, but it will not be correct, as the two loops that you are supposed to implement contain empty bodies. Once you implement the functions, the C++ code will report the speedups that unrolled loops provide.

1.1 Technical notes

- The coding aspect of this assignment is constrained entirely to `skeleton.py`. In fact, everything except optional testing is contained to two python functions in `skeleton.py`: The first one starts at line 43. The second one starts at line 75. There are long comments for each function with additional instructions.
- Read through the entire skeleton code to understand the structure. The python code is writing a C++ file that is then compiled with `clang++` and executed. The C++ file will time your implementation loop against a reference.
- You can assume that the size is always a power of 2, and so is the unroll factor. That is, you do not need to implement "clean" up iterations.
- Remember that floating point constants need a `f` character, otherwise they will be considered double type, which will mess up your timings! For example, the floating point of `2` is `2.0f`
- For your submission, you are not allowed to change the `clang++` compile line, the reference loop, or the main string.
- Feel free to play around with the compiler flags on your end if you are so motivated. You will quickly find at higher optimization levels, the compiler will do this unrolling and interleaving for you.

1.2 What to Submit

You will submit a completed skeleton file. I suggest you run it with a variety of arguments and incorporate some tests into the generated C++ code to build confidence in your solution. Please keep the name of your skeleton file the same as the name in the original download.

Part of your submission will be some experimental timings. Run your program with dependency chain length of 64, and then with unroll factors of length 1, 2, 4, 8, 16, 64. Present your results as a line graph where the unroll factor is the x axis and the speedup of your two loops (relative to the reference) is the y axis. Write 2 paragraphs about your results and how they related to what we have been learning. Place a PDF containing your graph and write-up in the same directory as the skeleton code.

Your grade will be based on 4 criteria:

- Correctness: do your functions compute the right result. If not, we cannot grade the rest of the code.
- Conceptual: do your functions actually unroll and interleave instructions. Please comment your code.
- Performance: do your performance results match roughly what they should.
- Explanation: do you explain your results accurately based on our lectures.

2 Unrolling Reduction Loops for ILP

Part 2 is identical to part 1, except we are targeting a different type of `for` loop. Here we are targeting reduction loops, where each iteration depends on the previous one. Recall in class, we showed that these loops can be unrolled to exploit ILP. You should apply a "chunking" unroll style, i.e. how we described in lecture. That is, the input array should be divided into N equal sized chunks (where N is the unrolling factor). Each loop iteration can then execute N reduction commands. At the end of the function, there needs to be a loop adding up the totals for each N.

There is only one parameter in this part, the unroll factor. Your timing results can be displayed as a line graphs where the x axis is the unroll factor (or partitions in the code), and the y axis is the speedup relative to the reference. You only have to go up to an unrolling factor of size 16 in this part (you will see why).

Again you can assume the size and unroll factor is always a power of 2.

All other aspects of this part can are the same as part 1.

2.1 What to Submit

This is the same as in Part 1.

3 Detecting SPMD Parallel Loops

In this part, you will be given some nested `for` loops, and two index calculations for a memory access (a read and a write). Your job is to determine if it is safe to make the outer-most loop parallel. That is, you will need to determine if the index calculations could conflict with two threads executing the outer-most loop. You will build up constraints that model a *reader thread* and a *writer thread*. You will use the Z3 constraint solver to check if the two threads can conflict.

3.1 Getting started

Please go through the Python Z3 tutorial at:
https://ericpony.github.io/z3py-tutorial/guide-examples.htm

You only need to go up to the "Functions" section (immediately after the "Machine Arithmetic" section).

Your assignment is constrained to `skeleton.py`. Read through this code to understand the structure. I have done the work to parse the python AST for you. You will need to implement the constraint solving in `check_parallel_safety` around line 172. Please read the specification and comments (especially the description at the top of the file) carefully. You do not need to fully understand the AST parsing (I have labeled those functions).

3.2 Technical Work

Your job is to implement constraints to determine if two threads would contain a read-write conflict if the outer-most loop is made parallel. The programs have an extremely limited form. They are an arbitrary nest of `for` loops followed by a read index calculation and a write index calculation. I have parsed the AST and provided you with a list of `ForLoops` and read/write index strings.

Use Z3 to create two variables per `for` loop: one for the reader thread, and one for the writer thread. Add the constraints to the solver such that these variables respect their loop bounds. The outer-most loop will need an additional constraint as the reader and writer thread cannot have the same value for the loop variable.

The read/write index strings will be expressions consisting only of numbers, loop variables, and operators (+ or *). I suggest using string replace functions to substitute the Z3 loop variables into these strings. You can then set a constraint where these two strings are equal to each other. This string can be evaluated using `eval`. This is not the cleanest solution, but I did not want to subject you to the actual Python AST.

At this point, ask Z3 to solve the equation. If the equation is satisfiable, it means there is some iteration from the writer thread that conflicts with some iteration of the reader thread, and thus it is not safe to make the outer-most loop parallel.

3.3 Evaluation

Much like the previous assignments, I have provided a `tester.py` that you can run. There are 8 test cases in the `test_cases` directory. You can run your the skeleton script given one of these files.

3.4 Submission

Zip up the entire directory (including `tester.py`) and upload to homework 3 in Canvas.

Your grade will be based on 4 criteria:

- Correctness: are you able to determine if the programs in `test_cases` have reader-writer conflicts?

- Conceptual: do you generate the right constraints for Z3 in your implementation?
- Comments: do you describe your code using comments?