CSE113: Parallel Programming

Homework 1: Instruction Level Parallelism and C++ Threads
Assigned: April 8, 2021
Due: April 22, 2021

Preliminaries

1. This assignment requires the following programs: `make`, `bash`, `python3`, and `clang++`. This software should all be available on the provided docker.
2. Review the April 8 lecture: it should contain a most of the required background. Material from earlier lectures will also be useful.
3. Find the assignment packet at `/home/tsorensen/public/homework3/part1`. Download the packet and unzip it. But do not change the file structure.
4. This homework contains 3 parts. Each part is worth equal points.
5. 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. If you have questions about the structure of your zip file, please ask!
6. It is okay to discuss docker questions and file structure questions with your classmates, or in canvas discussions.

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. I would advise you to add extra code to test your implementations.
- 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. We will remove points for renamed files.

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 SPMD Parallel Programming Using C++ Threads

Here you will get some experience writing parallel code using C++ threads. Unlike the previous two parts, you will create this code from scratch. However, it must compile with the Makefile I provided.

Your file will be called `homework1_part3.cpp`.

Your job is to create a 3 integer arrays of size 1024. Call them a, b, c and d. You should initialize every element of each array to 0.

Your job is to store the value 1048576 (lets call it K) in each element of each array, but there is a catch. You can only use the increment operator (i.e. `++`). Furthermore, each loop must be operated on differently. I suggest you write a function for each task. Use the `volatile` keyword in the function argument list so that all all of your memory operations actually access a memory location. You should be able to tell from your timing results if your operations are being optimized by the compiler.

1. For array a, you must do this computation sequentially.
2. For array b, you must do this computation in parallel by writing an SMPD style function. You must access elements in a round-robin style. That is, a thread with thread id of tid must compute the elements at index i where

   $$i\%NUM\_THREADS = tid$$

   a thread must complete all increment operations on a location before moving on to the next location. It must operate on its location in order.
3. For array c, you must do this computation in parallel by writing an SMPD style loop, however you can partition data to the threads in any way you'd like. Think about how you can do this in a more efficient way than what is done for array b.

You should time each function. You can use the chrono library similar to the C++ timing code in part 1. Print out the speedups of:

- the computation of a related to b
- the computation of a related to c
- the computation of b related to c

4 What to Submit

Similar to the previous two problems, part of your submission will be some timing experiments. Please run your code with 1,2,4,8 threads. Provide a line graph illustrating your results. The X axis is threads and the Y axis is the speedup relative to the computation of a. Please provide one

paragraph describing your strategy for computing array c and then two paragraphs describing your results. If you do not have 8 cores, you may not see speedups up to eight threads. That is fine; just say so in your description. Also, you may need to increase the size of K to get meaningful performance results. You should try to select a size on your machine where the computation of a takes around 1 second. Please include this number in your report.

You can modify the Makefile, but it should be able to compile your C++ file on your machine. There should be 3 files in this directory:

- The C++ file
- The Makefile
- A PDF of your report.

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 implement the SPMD programming model? is array b computed under the right constraints.
- Performance: do your performance results match roughly what they should? Could you identify ways to improve the performance of array c compared to b?
- Explanation: do you explain your results accurately based on our lectures.