CSE113: Parallel Programming

Homework 1: Instruction Level Parallelism and C++ Threads
Assigned: January 17, 2024
Due: January 27, 2024

Preliminaries

1. This assignment requires the following programs: `make`, `bash`, `python3`, and `g++`. This software is available on the provided docker and it is what the server will be using to run your solution.
2. Sign up for the GitHub Classroom assignment at https://classroom.github.com/a/gAfCaIrw. This will create a GitHub repository for you with the provided assignment files. After this repository is created, clone it locally using Git (command line will do, but feel free to use any other Git GUI tools).

   Do not change the file structure of this repository, or the autograding tests will not work. The autograding failing will impact your score as if you failed all of the autograder tests. Do not modify any file except the ones you are instructed to.
3. This homework contains 3 parts. Each part is worth equal points.
4. If you need help, please visit office or mentoring hours. You can also ask questions on Piazza. You are allowed to ask your classmates questions about frameworks and languages (e.g. Docker, Git, and C++). You are not allowed to discuss homework solution details with them.
5. **Your submission will consist of three parts**: a repository in GitHub Classroom, a pdf design document, and a pdf report. Upload the design and report to the assignment in Canvas when complete. The contents of the report for each part are detailed in each section. The instructions for submitting to GitHub Classroom are included at the end of the document.
6. You should develop locally first. Make sure your solution works as you intend before pushing to the server. This is a big class and we do not want to overwhelm the limited computing resources we have.
7. The results you see from the autograder are not your full grade. We may run additional tests and grade on some manual inspection of the code. Make sure to comment your code clearly and make sure it implements the specification.

Design Document

Before writing code for this homework, you must write up a design document. Your design document must be in PDF (you can easily convert other document formats, including plain text, to PDF).

Your document should describe the design of your code in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the workflow of each part and a description of each function with its purpose, inputs, outputs, and assumptions it makes about inputs or outputs.

Write your design document before you start writing code. It'll make writing code a lot easier. It will help you think about what you need to do for this homework, and it will help you identify possible problems with your planned implementation before you have invested hours in it. You're encouraged to do the design in pieces (e.g., detailed design for each part). We want you to get in the habit of designing components before you build them.

Also, if you want help with your code, the first thing we're going to ask for is your design document. We're happy to help you with the design, but we can't debug code without a design any more than you can.

1 Loop Unrolling Independent Iterations for ILP

Here we will consider unrolling `for` loops that contain independent iterations. However, each iteration will contain a chain of *dependent* instructions. 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 executing each iteration sequentially, and then interleaving the instructions from different iterations. Interleaving instructions should allow the processor to exploit more ILP, which should be seen 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 stubs 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 `part1.py`.
- Read through the entire skeleton code to understand the structure. The python code is writing a C++ file that is then compiled with `g++` 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, the only file that will be graded is `part1.py`. You can change code in `main.py` while testing, but please ensure your code works with the original version when submitting.

1.2 What to Submit

The pdf report for this part will include some experimental timings. Run your program with a dependency chain length of 64, and then with unroll factors of lengths 1, 2, 4, 8, 16, 32, and 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.

Make sure the most updated version of your code is pushed to the "main" branch of your GitHub Classroom repository. You will be able to see the required timing results in the Github Action tab.

In your report, please include your timings both from the server and as you saw on your local machine. Try to include the results on the same graph, but if the results are significantly different, you can use two different graphs.

Write 3 paragraphs about your results and how they related to what we have been learning. Describe if your results are different than the server results; if so, try to hypothesize why.

Grading 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 lecture that we showed that these loops can still be unrolled to exploit ILP. The loop should unrolled so that each iteration has computes several partial reductions. 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 graph 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.

You may be surprised by different timing information on your machine and on the server. Think about different ways to compute the partial reductions and try different approaches. Include graphs for both and document the different strategies you try.

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.

Array size note: In order to get meaningful timings, the code uses a very large array. This may overwhelm the machine you are running on. You can tell if this is happening if your code is taking longer than a few seconds to run.

To help with this, please try closing all programs before running your experiments (especially if you are running chrome). If you still don't have enough memory, you can reduce the size of the array, using the `SIZE` constant in the code. If you do this, please try to do your timing experiments with as large of an array as possible on your machine. Also, make sure to reduce your SIZE by a power of two (e.g. change 512 to 64).

Whether or not you reduce your array, your submission must run correctly, e.g. give the correct result, for the original size.

2.1 What to Submit

This is the same as in Part 1. That is, you will push to your GitHub Classroom repository. The report (which should be submitted on Canvas) should include a line graph with unroll lengths of 1, 2, 4, 8, and 16. You need to include results from both your local machine and the server. Make sure to write 3 paragraphs explaining your results.

3 SPMD Parallel Programming Using C++ Threads

Here you will get some experience writing parallel code using C++ threads. The file you will complete is `part3.cpp`.

The code creates 3 integer arrays of size 1024. They are called a, b, c and d. They are initialized to 0.

Your job is to store the value 1048576 (let's call it K) in each element of each array, but there is a catch. You can only operate on the arrays using the increment operator (i.e. `a[i]++`). Furthermore, each array must be operated on in a different specified function (see the code). Notice that the functions use the `volatile` keyword in the argument list. This is so that all of the memory operations actually perform their loads and stores. You should be able to tell from your timing results if your memory operations are being optimized by the compiler.

1. For array a, you must do this computation sequentially in the function `sequential_increment`.
2. For array b, you must do this computation in parallel by modifying the function `round_robin_increment` as an SPMD parallel function using C++ threads. In this function, threads 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 locations in order, from least to greatest.
3. For array c, you must do this computation in parallel by modifying the function `custom_increment` as an SPMD parallel function using C++ threads. However, you can assign the data to the different 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.

   As a clarification: you must make sure that every increment operation goes to memory. You cannot load the value into a local variable and increment the variable. You must do all of the additions to a single location before moving to the next one.

The code contains timing code. Please print out the timing information and pay attention to the following timings:

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

3.1 What to Submit

You will submit a completed `part3.cpp` file. Make sure the most updated version of this file is pushed to the "main" branch of your GitHub Classroom repository.

Similar to the previous two problems, part of your submission will be some timing experiments. Please run your code locally with 1,2,4,8 threads (this can be changed with a variable in `main.cpp`.

Provide a line graph illustrating your results. The x-axis is the number of threads and the y-axis is the speedup relative to the computation of a (you will have two of these lines). The server will provide results for 8 threads; please make a note of these results as well. Please provide one paragraph describing your strategy for computing array c and then two paragraphs describing your results.

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?

Submitting to GitHub Classroom

As mentioned in the introduction, signing up for the GitHub Classroom assignment will automatically create a repository for you under the `ucsc-cse113` organization. This repository is for you to store the changes to your assignment, and will be where you submit your solution. Do not rename the homework files, as this will cause the autograding scripts to fail. Do not modify anything inside the `.github/` folder. The only files that should need editing for this assignment are `part1/part1.py`, `part2/part2.py`, and `part3/part3.cpp`. Any others may be overwritten by the autograding scripts.

For a refresher on Git/GitHub, we are providing an (ungraded) assignment with a quick read-through of some of the main concepts. This can be accessed using this invite link: https://classroom.github.com/a/GGrp9xbI.

The autograding scripts are currently set up to run on any pushes to the "main" branch of the repository. Make sure to not commit every superficial change, as that this will run a series of autograder actions every time you push to the repository.

The results of the autograding passes can be seen in GitHub in your repository's "Actions" tab. Opening the most recent workflow run from this tab will allow you to see the total points acquired in the summary. Next, go to "Autograding" under the "Jobs" category on the left side of the page. You can view the results as they are processed by opening the `Run education/autograding@v1` task. This will give you a detailed list of the autograding tests run and their results. The autograder will currently check whether the values you compute match what is expected, and will give information on timing and speedups. We reserve the right to run more tests once the assignment is complete.

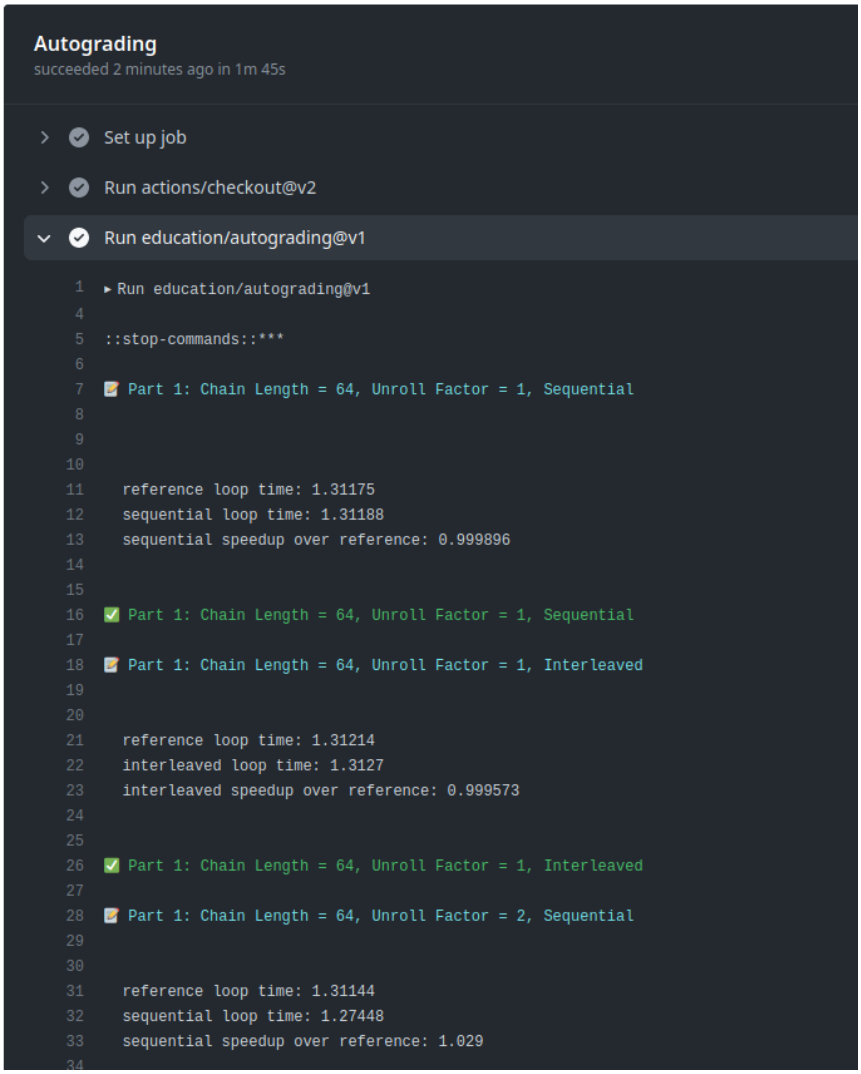

Figure 1: GitHub Classroom Workflow Run

As a reminder, it is advisable not to wait until the last minute to submit to GitHub Classroom, as all of the autograding is run in a queue, and it may take some time to get results for your submission.

In addition to submitting your solution to GitHub Classroom, please also submit a pdf design and a pdf report, including the graphs and results detailed in each section, to the Canvas assignment.