High Performance Computing

Homework #4 Part B

Due: Tuesday March 3 2015 by 11:59 PM (Midnight)

Email-based help Cutoff: 5:00 PM on Mon, Mar 2 2015

Points (for Part B): 10

Submission Instructions

This homework assignment must be turned-in electronically via Niihka. Ensure that you save this MS-Word document with the naming convention `MUid_Homework4_PartB.docx` prior to proceeding with this homework. Once you have completed filling-in the necessary information in this document, **save the file as a PDF and upload the PDF document** back onto Niihka.

Objective

The objective of this homework is to measure the performance improvements gained by resolving some of the control hazards in a straightforward program.

Grading Rubric:

Scoring for this assignment will be determined as follows:

- 5 points: The programs were properly compiled, run, and the data was collated correctly as per the instructions. The data was properly tabulated in the report and any graphs were plotted correct as directed. The homework has a good summary discussing various performance characteristics.
- 5 points: Graduate & undergraduate students: Develop a different C++ example (it cannot be a matrix-based operation) that demonstrates the effectiveness of loop unrolling. Similar to the supplied `unroll.cpp` your example must also have `approach1` and `approach2` methods (they may call other methods from them). Ensure your file is named with the convention `MUid_hw4_partb.cpp`.

Task 1: Log onto the RedHawk cluster

1. First run X-Ming our X-Server. If needed, refer to `LinuxEnvironemnt.doc` available off Niihka (under `Course Documents` → `Handouts` folder) for details.
2. Use a properly configured `PuTTY` session to log onto the head node of the cluster, namely: `redhawk.hpc.muohio.edu`.
3. Verify that you have completed the above two steps correctly by launching an `emacs` at the shell prompt and ensuring you see a graphical `emacs` window on your local machine.

Task 2: Request time on a compute node via PBSpro

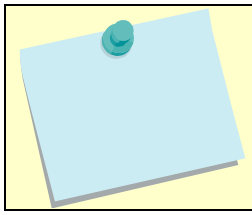
The computational infrastructure of RedHawk cluster is managed using a cluster management software system called `PBSpro`. `PBSpro` provides a suite of commands to run, manage, and remove jobs on the cluster such that each job is guaranteed to receive the requested computational infrastructure (in terms of number of processors, number of nodes, amount of memory, and run time). Refer to `LinuxEnvironemnt.doc` available off Niihka (under `Resources` → `Handouts` folder) for some additional details on `PBSpro`.

In this exercise we will request time on a compute node (1 processor/core) for performing interactive operations. This task is performed so that when time sensitive operations are performed, it is guaranteed that one core on a compute node is reserved solely for the jobs we are running. Reservation of a CPU for jobs is critical to ensure that we obtain reasonably reliable timings for comparisons and measurements.

Accordingly, request reservation of a single compute node via `PBSpro` job management system by issuing the following command at the shell prompt:

```
$ qsub -I -X
```

In the above command `-I` option requests PBSpro to start an interactive session and the `-X` option instructs PBS to continue to honor forwarding of graphical windows started using the X-windows system. The `-X` option essentially ensures that `emacs` and other graphical tools will start and run properly.



In future whenever you are performing timing related experiments for performance comparison ensure you use the above command (`qsub -I -X`) to reserve a compute node for your use. Note that the process of reserving multiple compute nodes is different and will be covered later on in this course.

Task 3: Download and review unroll.c

Download and save the supplied `unroll.c` program in your work folder. This program was adapted from the software optimization guide provided by AMD. This program is used to demonstrate how resolving control hazards that arise in small loops causing stalls in the pipeline.

The supplied `unroll.c` program uses a standard sequence of calculations reflect a typical 3-D transformation that is performed for rendering graphics on the screen. The program performs a standard 3-D transformation (specified in 4x4 matrix `m`) to a given point (`v` on the x, y, and z planes) resulting in a new point (`r`) via the following matrix multiplication operation: `r = v * m`; where

- `m` is a 4x4 matrix that is setup to do a graphics operation (such as: translation, rotation, scale, etc.)
- `v` is a 1x4 array containing the x, y, and z value for a given point.
- `r` is A 1x4 array that will contain the result of applying the transformation (`m`) to the specified point (`v`) through matrix multiplication.

Open the `unroll.c` program in `emacs` and review function `approach1()`. This function provides the default/reference implementation which is typically used by almost all programmers to implement simple matrix multiplication. The exact nature of the operations being performed is not as important. What is important is to note that in the nested `for`-loops in this function constantly require branching back to the top of the loops. This introduces a control hazard in the pipeline causing the pipeline to stall.

On the other hand, the `approach2()` function in `unroll.c` takes an aggressive approach by completely unrolling the loop. Loop unrolling is a standard programming technique that is used to minimize the number of control hazards (that arise due to branching in loops) in looping constructs by simply repeating the body of the loop several times as shown in the example below:

Original Loop	***Unrolled Loop***

`for(int i = 0; (i < 1000); i++) {`
`  // Body doing some operation with i`
`}`
	`for(int i = 0; (i < 1000); i += 5) {`
	`  // Body doing some operation with i`
	`  // Body doing some operation with i+1`
	`  // Body doing some operation with i+2`
	`  // Body doing some operation with i+3`
	`  // Body doing some operation with i+4`
	`}`

Many compilers don't aggressively unroll loops. Manually unrolling loops can benefit performance, especially if the loop body is small, which makes the loop overhead significant. Review `approach2()` function that aggressively unrolls the nest `for`-loops.

Task 4: Measure timings for approach1() in unroll.c

1. Ensure that the `main()` function in `unroll.c` is calling `approach1()` function for testing. Note that the `main()` function calls the function-under-test millions of times. The function-under-test is repeatedly invoked a large number of times so that the overall runtime is at an acceptable (in 10s of seconds) value so that the timing measurement is more reliable and less noisy. Noise in timings are introduced because of:
    a. The resolution of hardware clock and its interaction with the OS is usually in microseconds if not milliseconds. Consequently, if a program runs for a fraction of a second the timings will have considerable errors in them.
    b. Although a core is reserved for your job, still your task may get preempted for performing other OS related actives.
    c. Large number of repetitions ensures that caching issues do not skew the timings.

2. Compile `unroll.c` (calling `approach1()`), run, and note the program execution time (execution time is measured using Linux's `time` command; simply put `time` before any program you run to measure the time the program takes to run) using the commands indicated below:

```
$ g++ -O2 -std=c++11 -g -Wall unroll.c -o unroll
$ /usr/bin/time ./unroll
```

3. Repeat the above command (that is `time ./unroll`) 5 times and note the time taken to run the same program five times in the following table:

Run #	*Elapsed Time (sec)*
1	11.64
2	11.57
3	11.55
4	11.65
5	11.67

4. Observe that the runtimes for exactly the same program vary with each run. This is a typical and expected behavior. However, the key question is what the correct runtime is and what value to report? The answer is that all of the runtimes are correct and each user may get a different runtime. However, to ensure that numbers are comparable you must report the mean (average) value along with 95% confidence interval (CI). The 95% CI value essentially defines a range around the average that essentially indicates that you are confident that 95% of the time, the observed execution time would lie within the range you report. In other words there is a 5% chance that the numbers you report are wrong. In practice researchers often perform 10 to 20 runs and report statistics to obtain better CI values. The confidence interval for the data must be computed using the following formulas:

   Assume the five timings you have recorded are t_1, t_2, t_3, t_4, and t_5. First calculate the mean (μ) and standard deviation (σ) using the formulas:

$$\mu = \sum_{1}^{n} t_i/n \quad \textbf{and} \quad \sigma = \sqrt{\left(\sum_{1}^{n} (t_i - \mu)^2\right)/n}$$

   Where n =5 (in this specific exercise since we are using only 5 timing values). Now, from the student t-distribution tables, the 95% CI is computed using the formula:

$$95\%\ \text{CI} = (2.132 * \sigma)/\sqrt{n}$$

   Note that the number `2.132` (aka the z-value) varies depending on the value of n. Refer to the following URL for a nice table containing various zed values: http://en.wikipedia.org/wiki/Student's_t-distribution.

5. Record your timing in the form μ±CI seconds below (don't forget the unit):

Time for `approach1()`: 11.563 --- 11.665

Task 5: Measure timings for approach2() in unroll.c

1. Now, edit the `main()` function in `unroll.c` and ensure it is now calling `approach2()` function for testing.

2. Compile (using command line shown earlier using `-O2` flag), and repeatedly run the program (as described earlier) to obtain 5 runtimes and note the time taken in the following table:

Run #	*Elapsed Time (sec)*
1	7.10
2	7.06
3	7.08
4	7.04
5	7.11

3. Compute the average and 95% CI for running the program using `approach2()`. Report the timings to your instructor and record it below:

   Time for `approach2()`: 7.052 --- 7.106

Task 6: Compare timings for approach1() and approach2() in unroll.c

Now compare the timings for `approach1()` (from Task 4) and `approach2()` (timings from Task 5). Based on the averages (and CI) draw inferences on the performance gained by using the two approaches by computing the values in the following table. In this exercise, the timing value for `approach1()` is the reference timing. The most performance is when `approach1()` is slowest (add CI value to `approach1()` timing) and `approach2()` is the fastest (subtract CI value from `approach2()` timing). For least performance gain use the converse of the above rationale. The average value is computed by simply using the average values.

Type of performance	Timing Difference	Percentage gain
Most performance gain observed:	4.60	39.6%
Average performance gain observed:	4.53	39.0%
Least performance gain observed:	4.45	38.3%

Note that you must compute percentage gain using the following formulas and information provided below:

$$\%\text{Gain} = \frac{\text{Timing Difference}}{\text{Reference Timing}} * 100$$

Task 7: Measuring timings with more compiler optimizations

In the previous tasks the performance of approach1 and approach2 was quantitatively measured and compared using compiler optimization level of `-O2`. This compiler optimization level is typically used for most programs as it provides a good balance between various competing factors and provides good performance. The g++ compiler is capable of performing more aggressive optimizations using the `-O3` flag. However, the `-O3` optimization may or may not improve performance (in some cases it has known to degrade performance) when compared to `-O2` optimization level.

It is time to determine if the `-O3` optimization performs better than `-O2` for the problem at hand in the following manner:

1. Edit the `main()` function in `store2load.c` and ensure it is now calling `approach1()` function for testing.

2. Compile (using command line shown earlier with `-O3` instead of ~~`-O2`~~), and repeatedly run the program (as described earlier) to obtain 5 runtimes and note the time taken in the following table:

Run #	*Elapsed Time (sec)*
1	4.14
2	4.14
3	4.13
4	4.14
5	4.15

3. Compute the average and 95% CI for running the program and record the timing below:

   Time for `approach1 & -O3`: 4.142

4. Edit the `main()` function in `store2load.c` and ensure it is now calling `approach2()` function for testing.

5. Compile (using command line shown earlier with **-O3** instead of ~~-O2~~), and repeatedly run the program (as described earlier) to obtain 5 runtimes and note the time taken in the following table:

Run #	*Elapsed Time (sec)*
1	4.14
2	4.14
3	4.15
4	4.15
5	4.14

6. Compute the average and 95% CI for running the program and record the timing below:

   Time for `approach2 & -O3`: 4.146

7. Discuss in reasonable detail (that means at least 6-9 sentences) your observations about the performance of `approach1` and `approach2` with **-O2** and **-O3** flags. In addition, provide a plausible explanation about the potential optimizations that the `g++` compiler may be performing (or failing to perform) on the code.

When using –O2 and –O3 compiler flags to compile program, since approach1 didn't do any optimization like unroll loop, there is a big time difference between approach1 and approach2, when using approach2 compilered with –O3 and –O2 flags, the compiler notice there is unrolled loop, and the data involved in operations are independent, so it optimized the code by using SIMD feature of hardware or even pre-compute the value to speed up the execution, that why there is such a big time difference between approach1 and approach2 when using –O2 and –O3 flags.

Task 8: Develop different example to demonstrate loop unrolling

Develop a sufficiently different C++ example (graduate students – it cannot be a matrix-based operation) that demonstrates the effectiveness of loop unrolling at the `-O2` optimization level. Similar to the supplied `unroll.cpp` your example must also have `approach1` and `approach2` methods (they may call other methods from them). Ensure your program is named with the convention `MUid_hw5_partb.cpp`.

1. Record your timing from your custom example (namely: `MUid_hw5_partb.cpp`) in the form μ±CI seconds below (don't forget the unit):

   Time for your `approach1`: 4.02

   Time for your `approach2`: 2.236

Task 9: Upload this document back to Niihka

Once you have successfully completed this exercise, computed and filled in all the necessary values into the document, **save the document as a PDF and upload the PDF file** to Niihka using the appropriate submission links. Ensure you upload `MUid_hw4_partb.cpp`. Ensure you click the `Submit` button after you have uploaded all the necessary files for various parts of this homework.