|  |
| --- |
| **Objective** |
| The objective of this homework is to measure the performance improvements gained by resolving some of the data hazards in a straightforward program. |

In this exercise we will request time on a compute node (1 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 PBSpro 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.

|  |
| --- |
| MCj04247960000[1]**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 store2load.c***

Download and save the supplied store2load.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 data hazards that arise due to Store-to-load dependencies can enhance performance of a program. The supplied store2load.c program uses a standard sequence of calculations involving 3 vectors (or arrays of values). Such transformations are commonly performed in graphical applications and many numerical simulations involving Ordinary Differential Equations (ODEs).

Open the store2load.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 a sequence of vector operations. The exact nature of the operations being performed is not as important. What is important is to note that in the two for-loops in this function, there is a data dependency or **a data hazard** (specifically a Store-to-Load dependency) between x[k] and x[k-1] (yes, it does take a discerning eye to note something like that). Consequently, until x[k-1] is computed and written, x[k] cannot be computed. This introduces a stall in the pipeline operation due to a Store-to-Load dependency between x[k] and x[k-1].

A store-to-load dependency exists when data is stored to memory, only to be read back shortly thereafter. Processors from Intel and AMD contain hardware to accelerate such store-to-load dependencies, allowing the load to obtain the store data before it has been written to memory through forwarding. However, it is still faster to avoid such dependencies altogether and keep the data in an internal register.

Avoiding store-to-load dependencies is especially important if they are part of a long dependency chain, as may occur in a recurrence computation. If the dependency occurs while operating on arrays, many compilers are unable to optimize the code in a way that avoids the store-to-load dependency. In some instances the language definition may prohibit the compiler from using code transformations that would remove the store-to-load dependency. Therefore, it is recommended that the programmer remove the dependency manually, for example, by introducing a temporary variable that can be kept

in a register.

The approach2() function in store2load.c applies the above strategy to the body of approach1() to minimize the Load-to-Store dependency in the body of the for-loops by introducing a temporary variable t. Review approach2() function.

***Task 4: Measure timings for approach1() in store2load.c***

1. Ensure that the main() function in store2load.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:
   1. 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.
   2. Although a core is reserved for your job, still your task may get preempted for performing other OS related actives.
   3. Large number of repetitions ensures that caching issues do not skew the timings.
2. Compile store2load.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:

|  |
| --- |
| **$** g++ **–O2** –std=c++11 –g –Wall store2load.c -o store2load  **$** time ./store2load  This is the time we care about. The program ran for zero minutes 19.788 seconds.  real 0m19.788s  user 0m19.688s  sys 0m0.004s |

1. Repeat the above command (that is time ./store2load) 5 times and note the time taken to run the same program five times in the following table:

|  |  |
| --- | --- |
| ***Run #*** | ***Time (sec)*** |
| 1 | 9.455s |
| 2 | 9.499s |
| 3 | 10.015s |
| 4 | 9.541s |
| 5 | 9.461s |

1. 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 t1, t2, t3, t4, and t5. First calculate the mean (μ) and standard deviation (σ) using the formulas:

Mean = 9.5932; standard deviation: 0.0451

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:

= .0406

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>.

1. Compute your timing in the form μ±CI seconds and record it below (don’t forget the unit):

|  |  |
| --- | --- |
| Time for approach1(): | 9.5932 ± .0406 seconds |

***Task 5: Measure timings for approach2() in store2load.c***

1. Now, edit the main() function in store2load.c and ensure it is now calling approach2() function for testing.
2. Compile (using command line shown earlier with **–O2**), and repeatedly run the program (as described earlier) to obtain 5 runtimes and note the time taken in the following table:

|  |  |
| --- | --- |
| ***Run #*** | ***Time (sec)*** |
| 1 | 4.678s |
| 2 | 4.756s |
| 3 | 4.756s |
| 4 | 4.697s |
| 5 | 4.775s |

Mean = 4.7324 ; Standard Deviation = 0.0013 2.015 0.0026

1. Compute the average and 95% CI for running the program using approach2(). Record the timings below:

|  |  |
| --- | --- |
| Time for approach2(): | 4.7324 ± 0.0011 seconds |

***Task 6: Compare timings for approach1() and approach2() in store2load.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 different timing values and record them in the table further below. Note that you must compute percentage gain using the following formulas and information provided below:

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: | 10.055 – 4.6769 = 5. 3781s |  | 53.48% |
| Average performance gain observed: | 4.8608 s |  | 50.66% |
| Least performance gain observed: | 9.4144 – 4.7761 = 4.6383s |  | 49.26% |

***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 #*** | ***Time (sec)*** |
| 1 | 5.407s |
| 2 | 5.455s |
| 3 | 5.932s |
| 4 | 5.484s |
| 5 | 5.276 |

Mean: 5.5108 standard deviation = 0.2485

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

|  |  |
| --- | --- |
| Time for approach1 & -O3: | 5.5108 ± 0.2245 |

1. Edit the main() function in store2load.c and ensure it is now calling approach2() 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 #*** | ***Time (sec)*** |
| 1 | 6.018s |
| 2 | 5.372s |
| 3 | 5.412s |
| 4 | 6.119s |
| 5 | 5.380s |

Mean: 5.6602s Standard deviation: 0.37473

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

|  |  |
| --- | --- |
| Time for approach2 & -O3: | 5.6602 ± 0.3386 |

1. 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.

|  |
| --- |
| It can be seen that approach 1 takes longer time to execute than approach 2 with respect to -02 flag. It is because in approach 1, there is a data dependency between x[k] and x[k-1]. So x[k] cannot be computed till x[k-1] is computed. This includes a stall in the pipeline. However, there is no data dependency between x[k] and x[k-1] in approach 2 due to the introduction of a temporary variable.  It can be seen that approach 2 takes longer time to compile and execute than approach 1 with respect to -03 flag. But there is not much of a difference. Both the approaches have almost the same execution time. This is because the –O3 automatically optimizes the code. The compiler would look at the entire code and optimize approach 1 so that it works as good as approach 2 due to which both the approaches have almost the same execution time. It is important to note that, -O3 optimization may not work favorable at certain times. So it is good to try both the levels of optimization while executing a program to see which works better.  Also, the –O3 optimization performs better than –O2 in this case because there are no space-speed tradeoffs in optimization level 2 unlike optimization level 3. Optimization level 3 gives importance to speed of execution than size as it uses rename registers and inline functions due to which the approaches have almost the same execution time. |