Programming Assignment #2

Due Feb 24, 2020 by 11:59pm **Points** 4
Submitting a text entry box, a website url, a media recording, or a file upload
Available until Feb 25, 2020 at 11:59pm

This assignment was locked Feb 25, 2020 at 11:59pm.

Your second programming assignment will focus on the two following skills:

- Ensuring that you have a certain level of comfort with timing individual functional components (i.e. *kernels*) of a performance-sensitive algorithm, and
- Giving you some exposure to designing and optimizing kernels that fit into a larger algorithm, in our particular case by **merging** multiple kernels into one, for purposes of performance optimization.

The code you will use as a basis of your implementation will be taken from ***LaplaceSolver/LaplaceSolver_0_2,*** and you will likely be following the example illustrated in ***LaplaceSolver/LaplaceSolver_0_3*** for using Timers to benchmark kernel calls.

Your tasks are as follows:

- The code directories above provide a sample implementation for the Conjugate Gradients algorithm we described in class (use our slides from Feb 6th as reference **[Link] (http://pages.cs.wisc.edu/~sifakis/courses/cs639-s20/media/CS639_6_Feb_2020.pdf)** , for example looking at page 24 of those notes).
  ***[Note:** As we discussed in class, the statement of this algorithm in the notes is a bit more generic than the narrow case in the implementation. In particular, (a) the algorithmic steps in red color are not necessary for the type of problem we used in our example, and may be omitted, and (b) operations marked by the "dagger" symbol (†) in lines 4 and 13 are stand-ins for "preconditioners" that we will discuss later. You are free to consider the matrix **M** in those expressions as being simply the identity, which reduces them simply to assignment of **r** into **p** in line 4, and **r** into **z** in line 13].*
  In the supplied implementation, we have partitioned all the nontrivial computation into "kernels", such as **ComputeLaplacian**, **InnerProduct**, **Norm**, and **Saxpy**. For one of these kernels (the Laplacian stencil computation) we have previously timed the cost of individual executions, but done so in isolation; the kernel was executed repeatedly by itself, and not in the context of a larger algorithm, as Conjugate Gradients here.

  Your task will be to instrument the code so that the cumulative time dedicated to each kernel call is computed, for the *entire duration* of the Conjugate Gradients algorithm (as set up in the example, about 250-260 iterations are executed, at the preset resolution). That is, we want to know how much time was spent on each kernel call across all iterations of the algorithm, which will give us the proper

idea of how much this kernel contributed to the execution time of the entire Conjugate Gradients algorithm.
[Optional/Recommended, but not mandatory : if a kernel is invoked more than once per iteration, you should report on it separately, for example:
InnerProduct() on line 6 : Time = xxx ms.
InnerProduct() on line 13 : Time = yyy ms.
1st Saxpy() on line 16 : Time = zzz ms.
2nd Saxpy() on line 16 : Time = www ms.]

Report the timing breakdown for two runs : (a) A serial run, with no parallelism, using only a single thread, and (b) A parallel run, using all threads available on your computer. For each test, also include timing for the **entire** Conjugate Gradients algorithm; if you have done this exercise properly, that time should be almost equal to the sum of the individual kernels you have broken down execution into.

It is strongly recommended that you disable the "output" to an image file while doing this performance measurement; you do not (and should not) include the cost of writing out the result of each iteration to an image file.

- In our February 13th class, we talked about the possibility of combining a number of kernels into a single (different) kernel, which could allow us to minimize memory usage, by streaming through certain arrays just once per kernel execution, instead of several times. Pages 43 through 48 of the aforementioned class notes **[Link]** **(http://pages.cs.wisc.edu/~sifakis/courses/cs639-s20/media/CS639_6_Feb_2020.pdf)** illustrate groups of kernels for which such merging may be possible; in class, we discussed specifically the example of the two Saxpy calls on line 16 (page 48 of the notes), and how they could be combined together.
  Your task will be to combine **at least one group** of kernels from those illustrated in pages 43 through 48, into a new kernel. Doing several (or all) groups will put you in consideration for extra credit.

  You are free to specialize the kernel as much as you need, assuming that it is correct for the task that is needed! For example, general-purpose Saxpy calls may allow the destination array to be different than one of the input arrays; when designing your own "replacement/merged" kernel, you are free to specialize such instance of the kernel, for example to re-use some of the input arrays as the output arrays by design (rather than by-coincidence).

  After you implement and test your new kernels (make sure you still get the correct/same behavior as before!) time the "old" group of kernels and compare with the new "merged" kernel. Report the results for single-thread, and multi-threaded (on all available cores) execution.

For your deliverable, include both code and a brief report, i.e. for task #1, show what you needed to change to instrument the code to collect timing information, and for task #2 show your replacement

kernel (it will presumably be a separate pair of h/cpp files), and how you linked it in your existing project. Also include a brief report (in PDF or text format) with the timing information requested, and (optionally) other observations or comments from your implementation experience.