332 Advanced Computer Architecture

Exercise (ASSESSED): The "Helmholtz Assembly" Challenge

This is the second of two equally-weighted assessed coursework exercises. **You may work in groups of two or three if you wish, but your report must include an explicit statement of who did what.** Submit your work in a pdf file electronically via CATE.[1] The CATE system will also indicate the deadline for this exercise.

Background This exercise concerns a scientific simulation based on the Helmholtz partial differential equation. The Helmholtz equation is used in the analysis of standing waves, with applications in acoustics (eg noise in cars, or musical instruments), antenna design and plasmas. Computational kernels in the application, similar to those used for the first assessed coursework, have been generated by Firedrake, an automated system for the portable solution of partial differential equations using the finite element method (FEM). For more information about Firedrake, visit http://www.firedrakeproject.org.

To solve the partial differential equation, the application uses a semi-unstructured mesh to discretise the spacial domain. Unstructured meshes (usually triangles in 2D, tetrahedra in 3D) are often preferred over simpler (grid-like) structured meshes, as they allow more flexibility in capturing complex geometry and focusing computational effort where it is needed. The semi-structured mesh we use belongs to the class of extruded meshes. Extruded meshes are the exponents of problems involving high aspect ratio domains present in ocean and atmosphere simulations. Their main feature is the presence of a short, vertical direction that allows for column-wise data alignment. This basically means that they allow iteration along "columns of cells" without the need for indirect memory accesses (e.g. `A[B[i]]`), which are often the main contributors to performance degradation. The utils.{c,h} files contain functions that are used to read a mesh directly from a file.

From the computational point of view, the Finite Element Method consists of two steps: assembly and resolution of a linear system of equations. The provided code implements the *assembly phase* for a Helmholtz-based problem. During this step, the behaviour of the equation is approximated "locally" in each cell of the unstructured mesh. This is essentially done by evaluating integrals corresponding to both left and right hand side of the equation (respectively *lhs* and *rhs*). Iteration over all mesh cells is implemented in wrapper_kernels.c, by functions whose name is prefixed with *wrap*. In the same file, functions called kernel_lhs and kernel_rhs implement the actual evaluation of integrals. These functions represent the bulk of the computation.

WARNING: Do not modify the addto_vector() function in wrappers_kernels.c. The function is meant to simulate the behaviour of incrementing the values of a global matrix. No matrix is provided in this case as this is not the purpose of the exercise and it would make results harder to check.

Running the benchmark

Getting the benchmark code

Copy the benchmark code to your own directory, e.g.

[1] `https://cate.doc.ic.ac.uk/`


```
prompt> mkdir /homes/yourid/ACA14 (you will probably have already done this)
prompt> cd /homes/yourid/ACA14
prompt> cp -r /homes/phjk/ToyPrograms/ACA14/HelmholtzChallenge ./
```

(The `./` above is the destination of the copy – your current working directory).

List the contents of the benchmark directory:

```
prompt> cd HelmholtzChallenge
prompt> ls
helmholtz.c gettimemicroseconds.c gettimemicroseconds.h Makefile script
utils.c  utils.h  wrappers_kernels.c  wrappers_kernels.h
```

To compile and run the code:

Compile it:

```
prompt> make
```

Run the program:

```
./ACA2-2014 /homes/phjk/ToyPrograms/ACA14/HelmholtzMeshes/spacefilling4
```

On a fairly recent Sandy Bridge machine, this took 45-135 seconds. You may wish to try with the larger meshes as well. For debugging and basic experiments, you should use the test mesh which contains only two triangular cells.

You can (and *should*) check that the results are correct. You can do this by comparing with the output of the unmodified program. For example, we have provided reference output from the "small" mesh:

```
./compare_dat rhs_out ../HelmholtzMeshes/rhs_out-small-reference.txt
./compare_dat lhs_out ../HelmholtzMeshes/lhs_out-small-reference.txt
```

The meshes can be found in:

```
/homes/phjk/ToyPrograms/ACA14/HelmholtzMeshes
```

They are quite big so don't copy them on the CSG systems, but you will need to copy them to run on your own machine.

All-out performance

Basically, your job is to figure out how to run this program as fast as you possibly can, and to write a brief report explaining how you did it.

Rules

1. You can choose any hardware platform you wish. You are encouraged to find interesting and diverse machines to experiment with. The goal is high performance on your chosen platform, so it is OK to choose an interesting machine even if it's not the fastest available. On Linux, type `cat /proc/cpuinfo`.

   You are encouraged to look at graphics processors, mobile devices, etc. If in doubt please ask. If you are interested in using research facilities please ask.

2. Make sure the output of the program you submit is **numerically correct**. The program we provided you with creates a file containing the output of the computation. You can run the original program once to get a file containing the output you must match (we do not provide it because depending on 1) the platform you run on and 2) the compiler you used, there may be minor discrepancies). You can run the `compare_dat` (see above) to verify whether the output of the modified program is correct, within a certain confidence.

   WARNING: Results will be checked by selecting submissions at random. Unless given a plausible explanation, submissions with wrong results will be given ZERO marks.

3. Make sure the machine is quiescent before doing timing experiments. Always repeat experiments for statistical significance.

4. Always script your experiments, so that it is easy to check how each data point was generated, and it's easy to rerun your experiments when you change something.

5. Choose a mesh size which suits the performance of the machine you choose - the runtime must be large enough for any improvements to be evident.

6. Take care to report precisely and fully the details of the hardware (CPU part number, clock rate) and software (compiler version numbers, OS etc).

7. If you use a laptop, make sure it's plugged into mains power, as some mobile platforms can adapt and mess up your experiments.

8. You can achieve full marks even if you do not achieve the maximum performance.

9. Marks are awarded for

   - Systematic analysis of the application's behaviour
   - Systematic evaluation of performance improvement hypotheses
   - Drawing conclusions from your experience
   - A professional, *well-presented* report detailing the results of your work.

10. You should produce a report in the style of an academic paper for presentation at an international conference such as Supercomputing.[2] The report should be not more than seven pages in length.

Changing the rules

If you want to bend any of these rules just ask.

Performance analysis tools: You may find it useful to find out about:

- cachegrind and cg_annotate
- kcachegrind - graphical interface to cachegrind: `http://kcachegrind.sourceforge.net`

[2] `http://sc13.supercomputing.org/`

- oprofile: `http://oprofile.sourceforge.net` - performance profiling by sampling hardware performance counters. Should install with apt-get on Ubuntu; works nicely with kcachegrind.
- Likwid-perfctr (http://code.google.com/p/likwid/wiki/LikwidPerfCtr) - handy performance counter tools that you can easily install on your own (non-virtual) Linux system.
- Intel's VTune - tool (Windows, Linux, MacOS) for understanding CPU performance issues and mapping them back to source code (free trial). `http://www.intel.com/software/products/vtune`.
- AMD's CodeAnalyst (http://developer.amd.com/tools/codeanalyst/pages/default.aspx)
- Apple's XCode "Instruments" performance tools
- OpenSpeedshop for Linux: `http://www.openspeedshop.org/wp/`

Compilers You are *strongly* advised to investigate the potential benefits of using more sophisticated compilers:

- Intel's compilers: these are not installed on CSG linux systems but you should be able to download student and evaluation copies for your own machines from `http://www.intel.com/software/products/compilers`
- AMD's APP SDK for OpenCL on x86 CPUs and AMD GPUs: `http://developer.amd.com/gpu/AMDAPPSDK`
- NVIDIA's CUDA and OpenCL compilers: `http://developer.nvidia.com/object/gpucomputing.html`

Source code modifications You are strongly invited to modify the source code to investigate performance optimisation opportunities.

How to finish The main criterion for assessment is this: you should have a reasonably sensible hypothesis for how to improve performance, and you should evaluate your hypothesis in a systematic way, using experiments together, if possible, with analysis.

What to hand in Hand in a concise report which

- Explains what hardware and software you used,
- What hypothesis (or hypotheses) you investigated,
- How you evaluated what the potential advantage could be,
- How you explored the effectiveness of the approach experimentally
- What conclusions can you draw from your work
- Specify how you ensured the correct results were obtained or justify why that is not the case
- If you worked in a group, indicate who was responsible for what.

Please do not write more than seven pages.

Dora Bercea, Fabio Luporini, Paul H.J. Kelly, George Rokos, Imperial College London, 2014