The "Smooth" Challenge

Ben Graham (bh110, c3),
Paul Gribelyuk (pg1312, a5)

March 7, 2013

1 Introduction

The "Smooth" program is a mesh pre-processing algorithm responsible for improving a specific, measurable characteristic, known as **quality**. A higher **quality** mesh aids local convergence properties in finite element analysis, where non-uniform domains require varying degrees of granularity. The "Smooth" algorithm concerns a a triangular discretization in 2D. The **quality** of the individual triangle faces based on a nonlinear combination of the components of a metric tensor at each of the nodes of that triangle. In our simulations, a suboptimally constructed mesh typically sees an average improvement of 6% with the minimum **quality** of any triangular face increasing 30-35%. We investigated various performance enhancements on the **Intel Xeon CPU E5606 2.13GHz** CPU with 4 cores. Our approach was twofold. We first conisdered serial optimisations by using various profiling tools to minimize instruction count and by exploiting locality of some of the data. Next, we exploited the inherent parallelism using the OpenCL programming on the **NVIDIA GE Force 570 (GF110 architecture)**, available through the computer lab on the "Graphics" machines. OpenCL version 1.1 was used for the GPU code, while the GCC 4.6.3 compiler compiled C++ code. All recorded benchmark results were obtained with the -O3 optimisation flag.

2 Profiling the Code

We first tackled the problem of optimising the serial version of the code. Optimisations at this stage will also help in the parallel implementation, especially when those performance gains are seen in the parallel region of the code (generally speaking). An Apple MacBook Air (with Intel i7 2.0GHz) with XCode, GCC 4.2 and Instruments (for profiling) was initially used. This process identifield that the code spending 60% of the time in `element_quality`, which is used to evaluate the quantitative effect of local changes to node coordinates within a mesh. For a node neighboring n triangles, $2n$ calls are made to this function. Furthermore, if N nodes exist in the mesh and we use I iterations to converge to a more optimal mesh, the total number of these calls is bounded by:

$$2N \cdot n\Delta(M)$$

where $\Delta(M)$ is the maximum degree of the mesh. Within `element_quality`, the standard power function $pow(x, y) = x^y$ took an inordinate amount of time, as well as computations for `element_area` and accesses to elements of `vector` fields in the `Mesh` class. Some further time was spent solving a 2-by-2 system of linear equations and in helper functions measuring properties of the mesh. The profiler allowed us to further determine that these resource uses were far from optimal since processor usage varied greatly throughout the program's execution. As expected the program does not tax the memory system, given that

the only updates are to two floating point values in the form of coorodinate updates.

3 Optimizing in Serially

The `Mesh` class uses `std::vector` objects to encapsulate relationships between Nodes and mesh Elements. However, adjacent Nodes have coordinates and metric values in spacially disparate places. However, we will tackle this challenge on the GPU by passing vectorized arrays to the device. We gained an initial speedup by re-writing the `pow()` call explicitly as 3 multiplications, which showed decrease in time from 7.05 seconds to 6.72 (4.7% improvement). Next, the branching code for the variable `f = min(l/3.0, 3.0/l)` to eliminate the division, thus saving clock cycles and reducing the mis-prediction penalty at that stage. Although earlier in the `element_quality` function, the call to `element_area` seems to retrieve the same data elements already retrieved in the calling method, combining the functions showed no effect, since, we believe, the compiler is already making this optimisation for us.

4 Parallelising using OpenCL

As first, naiive, parallelisation attempt, we used OMP to distribute work among the cores of the CPU. Although the loop controlling convergence of the algorithm could not be parallelised (since prior iterations had changed the mesh to an incrementally higher quality), the next loop, over the Nodes assigned nodes available cores. Although this saw a factor 2 speedup and produced almost identical results to the serial version, it opens the door for the possibility of adjacent Nodes being altered (or worse, the same Node) by parallel threads of execution, leaving a lower quality grid than previously found. When we inserted critical regions around all nodes neighboring each worker's Node, but saw a significant decrease in performance as threads stalled waiting for other threads to release their data locks.

5 Results

6 Conclusion