CS 5220 Introduction to Parallel Programming — Fall 2015
Kenneth Lim (kl545), Yalcin Ozhabes (ao294), Joshua Cohen (jbc264) — Project 2

1 Introduction

For compilation, we use a reasonable set of compiler flags:

`-O3, -no-prec-div, -opt-prefetch, -xHost, -ansi-alias, -ipo`

to handle the bare-bones of loop optimization, memory allocation/alignment, and architecture specific instructions. Notably, `-fast` failed to work because the compiler encountered problems inlining statically declared functions.

2 Parallelism

Function	CPU Time
`Central2D<Shallow2D, MinMod<float>>::limited_derivs`	1.434
`Central2D<Shallow2D, MinMod<float>>::compute_step`	0.865
`Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds`	0.549

Table 1: Excerpt from VTune Profiling Output

Intel's VTune Amplifier suggests that most of the CPU time incurred by the code is spent in the `limited_derivs` and `compute_step`, and `compute_fg_speeds` functions, in that order (see Table 1). The rest of the execution time is distributed across program initialization and system overheads in significantly smaller quantities. In the interest of targeting the main bottlenecks, we focus primarily on the two named functions.

```
1 for (int m = 0; m < du.size(); ++m) {
2   du[m] = Limiter::limdiff(um[m], u0[m], up[m]);
3 }
```

Figure 1: Bottleneck code in `limited_derivs`

Within `limited_derivs`, the bulk of the time is incurred by the call to the limiter (`Limiter::limdiff`) from within the loops going over each cell (see Figure 1).

Within `compute_step`, a majority of the time is spent on the correction loops (see Figure 2), followed by the prediction step loops, and finally the copying loops. As a naive

```
for (int iy = nghost-io; iy < ny+nghost-io; ++iy) {
  for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {
    for (int m = 0; m < v(ix,iy).size(); ++m) {
      v(ix,iy)[m] =
        0.2500 * ( u(ix,  iy)[m] + u(ix+1,iy  )[m] +
                   u(ix,iy+1)[m] + u(ix+1,iy+1)[m] ) -
        0.0625 * ( ux(ix+1,iy  )[m] - ux(ix,iy  )[m] +
                   ux(ix+1,iy+1)[m] - ux(ix,iy+1)[m] +
                   uy(ix,  iy+1)[m] - uy(ix,  iy)[m] +
                   uy(ix+1,iy+1)[m] - uy(ix+1,iy)[m] ) -
        dtcdx2 * ( f(ix+1,iy  )[m] - f(ix,iy  )[m] +
                   f(ix+1,iy+1)[m] - f(ix,iy+1)[m] ) -
        dtcdy2 * ( g(ix,  iy+1)[m] - g(ix,  iy)[m] +
                   g(ix+1,iy+1)[m] - g(ix+1,iy)[m] );
    }
  }
}
```

Figure 2: Bottleneck code in `compute_step`

sanity check, these numbers corroborate our expectations—`compute_step` performs most of the CPU-bound work, and within `compute_step` itself, the correction loops are performing the largest number of calculations.

We investigated the use of OpenMP annotations to parallelize the code but were not particularly successful. Profiling multi-threaded code with VTune requires additional diligence, and we generate concurrency reports by running the profiler as such:

```
amplxe-cl -collect concurrency
```

with an additional compiler flag `-parallel-source-info=2`. This yields additional information in the report about CPU utilization and thread idling time. For all parallelism cases described below, we aimed to saturate all available cores by setting the number of OMP threads to the maximum available, i.e.:

```
omp_set_num_threads(omp_get_max_threads())
```

This absolves us of the need to declare `num_threads` on every OMP pragma annotation. In addition, to ensure that all functions receive the full benefit of thread resources, we declare this as early as possible in `driver.cc`.

2.1 Collapsing Loops

Our first approach was a naive "carpet-bomb" collapsing of loops in `compute_step` and `limited_derivs`. Most loop pairs in both functions serve the purpose of iterating through a grid (of ghost cells or otherwise), and can be collapsed by annotation as described in Figure 3.

```
#pragma omp parallel for collapse(2)
for (...) {
  for (...) {
    ...
  }
}
```

Figure 3: Schema of paired for-loops iterating over a grid in parallel

However there are a number of pitfalls with this approach. Firstly, for loop collapsing to be effective, there should ideally be no data dependencies within the innermost loop. In our case, the loops are used to update parent data structures, and it is non-trivial to eliminate the dependencies. Analysis with VTune confirms that while iterations of the loop were indeed being run in parallel, there was very poor CPU utilization and a lot of execution time was lost to communication overhead.

We also explored the use of reduction annotations in the predictor loop in `compute_step`, believing that it might be a viable candidate because of the pair of substraction operations. Unfortunately, we did not obtain any noteworthy results.

2.2 Loop Scheduling

We briefly explored alternative scheduling algorithms in an attempt to increase CPU utilization. Enforcing `schedule(static)` for each parallelized loop improved CPU utilization by a fractional percentage. While this result was consistent, it did not yield significant improvements, and we have omitted it from our current best results. It is possible that we might return to this at a later time.

2.3 Master Execution Thread

Our final approach with OpenMP was at a coarser level. Rather than attempting to aggressively parallelize loops within the kernel, we adopt Prof. Bindel's advice and define a "master" execution frame in `run`. This creates a thread pool which we use to handle subregions of the grid. We can thus rewrite `run` as in Figure 4. This sets us up for better design as a whole, as we discuss in the next section.

```
#pragma omp master
{
  bool done = false;
  real t = 0;

  while (!done) {
    real dt;
    for (int io = 0; io < 2; ++io) {
      #pragma omp parallel for schedule(auto)
      {
        ...

        #pragma omp critical
        if (io == 0) {
          dt = cfl / std::max(cx/dx, cy/dy);
          ...
        }

        ...

        #pragma omp atomic
        t += dt;
      }
    }
  }
}
```

Figure 4: Modified `run` function incorporating a master thread of execution.

2.4 Coprocessor Offloading

We explored offloading portions of work to the Intel Xeon Phi Coprocessors, but faced difficulty in doing so due to heavy use of C++ templates in the codebase. All functions and variables that are not inlined have to be targeted for compilation on the coprocessor's architecture, and a good implementation has to account for thread affinity (viz. `KMP_AFFINITY`) combinations on both the host and coprocessors.

However, the prospect of using the coprocessors is tempting for the following reason: in the previous section we described the parallelization of the entire kernel pipeline as a whole, but each instance of the kernel pipeline can be offloaded to a coprocessor. This is ideal because re-targeting the entire `Central2D` class is more straightforward than cherry picking functions from within. We expect to spend a significant amount of time in this area leading up to our final submission.

3 Maqao and Single/Double Precision Conversion

Maqao's evaluation was not particularly useful. In accordance with the profiling data from VTune, we focus on Maqao's recommendations for `compute_step` and `limited_derivs`. For both cases, Maqao observes that there is no loop vectorization and recommends annotation with `#pragma ivdep`. This is potentially dangerous without further verification that there are no loop dependencies. In addition, its recommendation for eliminating expensive instructions is to compile for the host architecture, which we have already incorporated prior to profiling.

A single useful recommendation was obtained for `compute_step`, where Maqao recommended that single-double precision conversions be avoided by suffixing constants with `f` where double precision is not needed. This is applied to the constants 0.2500 and 0.0625 in the correction step.

4 Vectorization

Intel's Vectorization Report was generated using the compiler flags

```
-qopt-report=5 -qopt-report-phase=vec
```

and provided more substantial recommendations compared to Maqao. The speedup values we cite subsequently are obtained from this report unless otherwise stated.

4.1 Resolving Dependencies

A large part of the work done to aid compiler auto-vectorization involved either modifying the loop structure to eliminate flow/anti/output dependencies, or marking suspicious loops with `#pragma ivdep` to encourage the compiler to consider vectorization.

For the predictor loop in `compute_step`, we cache `uh`, `fx`, and `gy` outside the innermost loop. This provide a small measure of memory locality, but more importantly eliminates flow dependencies when updating `uh[m]`. Next, we squash the two subtraction operations into a single line so the compiler can detect the common value update pattern and vectorize it accordingly. Were the loops not squashed, the compiler would consider the pair of operations as an output dependency. The updated loop is shown in Figure 5.

```
1  for (int iy = 1; iy < ny_all-1; ++iy) {
2    for (int ix = 1; ix < nx_all-1; ++ix) {
3      vec uh = u(ix, iy);
4      vec thisFx = fx(ix, iy);
5      vec thisGx = gy(ix, iy);
6
7      for (int m = 0; m < uh.size(); ++m) {
8          uh[m] = uh[m] - dtcdx2 * thisFx[m] - dtcdy2 * thisGx
               [m];
9      }
10
11     Physics::flux(thisFx, thisGx, uh);
12   }
13 }
```

Figure 5: Vectorization-friendly code for the prediction step.

This results in a 1.56x speedup. Future work here might involve refactoring of the vector extraction functions to minimize single-double precision conversions.

Similarly, for the correction loop, we extract `v(ix, iy)` to eliminate the output dependency. The updated loop is shown in Figure 6.

This results in a very modest speedup ($\approx 2\%$) because the stride length of the array is still large. Further work here might involve refactoring to decrease the stride length. Prof. Bindel's attempt is instructive because appears to split the single large computation into two loops, which would add an additional opportunity for vectorization.

Not all loops were successfully vectorized. The copy step at the end of `compute_step` can be made a candidate for vectorization by annotating with `#pragma ivdep`. However there are no benefits to be gained because the overhead from vectorizing the loop outweighs the speed gains. Similar conclusions were made for the main loops in `apply_periodic` and `compute_fg_speeds`.

```
for (int iy = nghost-io; iy < ny+nghost-io; ++iy) {
  for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {
    vec context = v(ix, iy);

    for (int m = 0; m < context.size(); ++m) {
      context[m] = ...
    }
  }
}
```

Figure 6: Vectorization-friendly code for the correction step.

4.2 Elemental Functions

We attempted to create elemental functions for segments of the code with high utilization that could not be vectorized. The true source of the bottleneck in `limited_derivs` is in fact the `limdiff` function, which is challenging to vectorize as-is because of a flow/anti dependency incurred when updating `du[m]` in place. Instead, we annotate the entire function with `#pragma omp declare simd` and instruct the main loop in `limited_derivs` to execute the function via SIMD. Since there are four inputs to `limdiff`, the compiler successfully optimizes the loop, resulting in a 3.14x speedup.

5 Ongoing Work

A large part of our efforts is dedicated towards refactoring the kernel to work on appropriately sized subgrids. The calculations for each subgrid will be offloaded to the coprocessors as a bundled kernel evaluation function, and executed in parallel there with the support of vectorized instructions. Due to cluster instability leading up to the deadline, we were unable to generate quantitative timing data for our scaling studies, but expect to give the topic a more thorough treatment in our final submission.