1 Profiling

In this section we discuss only the results of the profiling tools that we have used, and any insights we have obtained. A discussion of the changes we have made to the codebase follows in the next section.

For each of the profiling jobs below, we run `./shallow` with $n = 1000$ and $F = 100$ to collect data over a larger calculation window. We use a standard offering of compiler flags (`-O3, -no-prec-div, -opt-prefetch, -xHost, -ansi-alias, -ipo`).

1.1 VTune

VTune profiling suggests that most of the CPU time incurred by the code is spent in the `limited_derivs` and `compute_step` functions, in that order. The rest of the execution time is distributed across program initialization and system overheads.

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. Within `compute_step`, a majority of the time is spent on the correction loops. Beyond that, the next set of bottlenecks in `compute_step` occur in the prediction step loops and copying loops.

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

Figure 1: Bottleneck code in `limited_derivs`

1.2 Maqao

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`. In addition, its recommendation for eliminating expensive instructions is to compile for the host architecture, which we have already incorporated prior to profiling.

A 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.

```
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`

1.3 IPO

Intel's Vectorization Report was generated using the compiler flags `-qopt-report=5 -qopt-report-phase=`

2 Optimization

2.1 Minimizing Precision Conversions

2.2 Parallelism

2.3 Vectorization

3 Ongoing Work