

CS5220 HW2 Report: Shallow Water Simulation

Team 6: Calvin Wylie (cjw278), Ji Kim (jyk46), Stephen McDowell (sjm324)

1. Introduction

In this report, we explore parallelization strategies for solving the shallow water equation. We start by profiling the naive implementation of the code in order to identify bottlenecks as well as build our intuition for developing a relatively accurate performance model. This information is used to guide the parallelization of the code for both the compute nodes (i.e., Intel Xeon E5-2620) and the accelerator boards (i.e., Intel Xeon Phi 5110P) on the Totient cluster. We further implement and evaluate several optimizations for tuning the parallel code.

2. Profiling the Shallow Water Simulation

Understanding which areas of the code most time is spent, and where opportunities for performance gains are available is key to achieving good speedups in general, and in particular for our parallel implementation.

We will focus on run-time profiling of the code, as well as compile-time reports generated by the Intel compiler to guide our efforts.

As we gain further understanding on the code using these tools, we may expand to more detailed and specific performance analysis tools such as *TAU*, *IACA*, and *MAQAO*.

2.1. Identifying Bottlenecks

As a first pass, we profiled the provided default code using the *amplxe* tool, with truncated results given below.

Function	CPU Time
Central2D<Shallow2D, MinMod<float>>::limited_derivs	1.350 s
Central2D<Shallow2D, MinMod<float>>::compute_step	0.652 s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	0.236 s

Expectedly, the vast majority of the time is spent inside the functions for the limiter, computing the step, and computing the wave speeds. What is surprising is the amount of time spent inside the limiter. Given a cursory glance at the code, one would assume that the *compute_step* function would be much more expensive than *limited_derivs*, yet we are seeing just over 2 times as much time spent inside *limited_derivs*. Identifying the cause and improving this performance bottleneck should be a main priority.

One clue may be that the arithmetic intensity of *limited_derivs* is much lower than *compute_step* and *compute_fg_speeds*. We analyzed the binary with *MAQAO*, and arithmetic intensity of the loops were computed. The results can be seen in the table below.

Function	Arithmetic Intensity (AI)
<i>limited_derivs</i>	0.11
<i>compute_step</i>	0.23-0.27
<i>compute_fg_speeds</i>	0.21

We can also look at more detailed profiling of individual functions. As an example, below is an excerpt from a profiling report for the *compute_step* function.

```
// Predictor (flux values of f and g at half step)
for (int iy = 1; iy < ny_all-1; ++iy)
    for (int ix = 1; ix < nx_all-1; ++ix) {                                0.002s
        vec uh = u(ix, iy);                                              0.013s
        for (int m = 0; m < uh.size(); ++m) {
            uh[m] -= dtcdx2 * fx(ix, iy)[m];
            uh[m] -= dtcdy2 * gy(ix, iy)[m];                                0.024s
        }
        Physics::flux(f(ix, iy), g(ix, iy), uh);
    }

// Corrector (finish the step)
for (int iy = nghost-io; iy < ny+nghost-io; ++iy)
    for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {                    0.010s
        for (int m = 0; m < v(ix, iy).size(); ++m) {
            v(ix, iy)[m] =                                              0.115s
                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] );
            }
        }
    }

// Copy from v storage back to main grid
for (int j = nghost; j < ny+nghost; ++j){
    for (int i = nghost; i < nx+nghost; ++i){
        u(i, j) = v(i-io, j-io);
    }
}
```

Although not very enlightening, it verifies our intuition that the corrector portion should be most heavily targeted for optimization, although a non-negligible amount of time is spent in other areas.

2.1.1. Vectorization

Significant performance gains can be obtained by vectorizing our functions. Vectorization is “the unrolling of a loop combined with [...] SIMD instructions”¹.

¹<https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf>

We decided that rather than attempt to write SSE/AVX instructions by hand, we would rely on the autovectorization capabilities of the Intel compiler. Guiding our efforts is the optimization/vectorization report generated by the intel compiler.

Looking at the vectorization report generated by compiling the default code, we see that the compiler did not vectorize any loop, with many message like this one, which corresponds to the *compute_fg_speeds* function.

```
LOOP BEGIN at central2d.h(268,9)
  remark #15344: loop was not vectorized: vector dependence prevents
                vectorization
  remark #15346: vector dependence: assumed FLOW dependence between
                _M_elems line 74 and _M_elems line 76
  remark #15346: vector dependence: assumed ANTI dependence between
                _M_elems line 76 and _M_elems line 74
LOOP END
```

Full definitions for the different types of vector dependence can be found in the Intel vectorization document linked above, but the basic idea is that the compiler has to assume that arrays may refer to overlapping memory locations. We can invite the compiler to ignore this potential dependency (if we as the programmers know it to be false), with the *#pragma ivdep* directive.

Doing so for the *compute_fg_speeds* results in the compiler vectorizing the loop, and realizing a potential speedup of 3.84.

```
LOOP BEGIN at central2d.h(269,9)
  remark #15300: LOOP WAS VECTORIZED
  remark #15460: masked strided loads: 6
  remark #15462: unmasked indexed (or gather) loads: 6
  remark #15475: — begin vector loop cost summary —
  remark #15476: scalar loop cost: 308
  remark #15477: vector loop cost: 76.870
  remark #15478: estimated potential speedup: 3.840
  remark #15479: lightweight vector operations: 109
  remark #15481: heavy-overhead vector operations: 1
  remark #15487: type converts: 8
  remark #15488: — end vector loop cost summary —
LOOP END
```

Full vectorization of the remaining code is not so simple. The way the data is currently laid out, as an array of structs representing solution values, rather than as a struct of arrays, makes it hard for the compiler for autovectorize. Such a hierarchy is likely why Professor Bindel elected to re-write everything in C – a more in depth discussion of how to remain in C++ will be covered in section 4.2.

3. Parallelizing the Shallow Water Simulation

3.1. Parallelizing for the Compute Node

A relatively simple way to parallelize work in any structured grid computation, including the shallow water equation solver in this assignment, is to divide the grid into blocks and assign each thread a block for which to compute the solution. We will refer to the cells in this block as *live cells*. Depending on the stencil radius of the computation kernel, we also need a ring of padding cells called *ghost cells* that are consumed

in order to calculate the solution for the boundary cells of the block. Here we redefine a *block* as the live cells and ghost cells required by a thread to calculate the solutions for all of the live cells. We say ‘consume’ since after advancing a timestep, an additional outermost r layers of the ghost cell ring become obsolete, where r is the stencil radius. The radius of the ring of ghost cells we need depends on the number of timesteps we want each thread to execute before synchronizing (i.e., updating the ghost cells with new solutions from other threads). Executing more than one timestep before synchronization is referred to as *batching* and will be discussed in Section 4. For this section, we assume each thread only executes a single timestep before synchronization.

In order to apply the blocking parallelization to the shallow water equation solver, we first need to identify which functions can or cannot be parallelized in the computation kernel. `Central2D::run()` calls the following functions:

- `apply_periodic()`
- `compute_fg_speeds()`
- `limited_derivs()`
- `compute_step()`

The `apply_periodic()` function updates the ghost cells of the grid by copying the outermost r layers of the live cells in a periodic fashion, where r is the ghost cell radius. This function forces a synchronization point since the ghost cells in a given block depend on the live cells of other blocks, meaning each thread has to communicate the solutions of the live cells in its block to all of the other threads. As such, this function only needs to be called at the beginning of every super-step (i.e., main computation sub-step + staggered computation sub-step).

The `compute_fg_speeds()` function is actually comprised of two separate logical functions: (1) calculating the maximum wave speeds in the x/y directions, and (2) updating the flux vectors. (1) is only used to calculate the dt which does not change within a super-step and uses expensive square root operations, so we separate this into a `compute_wave_speeds()` function that is called once before every super-step. This function is similar to `apply_periodic()` in that it requires synchronization across threads, in this case to calculate the maximum wave speeds across all blocks in the grid, thus making it difficult to parallelize naively. On the other hand, (2) still needs to be called at every sub-step, but can independently calculate the f and g vectors for a block without synchronization.

The `limited_derivs()` function calculates the u_x , u_y , f_x , and g_y vectors from the u , f , and g vectors to implement the MinMod limiter. The `compute_step()` function then uses these output vectors to calculate the solution vectors, u , for the current timestep. These functions can also independently calculate the output vectors without synchronization.

For now, we apply the blocking parallelization to the `compute_flux()`, `limited_derivs()`, and `compute_step()` functions and leave parallelization of the other functions for future optimizations. Because the ghost cells of each block become contaminated with incorrect vectors and ghost cells of one block overlap with live cells of other blocks, it is important that we allocate per-thread memory for storing various vectors of the block separate from the *global* state. As such, we encapsulate this per-thread information in a class called `LocalState` that holds the vectors for the *local* state: u_- , f_- , g_- , u_x_- , u_y_- , f_x_- , g_y_- , and v_- . In fact, all of these vectors with the exception of u_- are only used in the local state computation; we only need to keep a separate global state for u_- . Figure 1 shows a visual representation of how the global grid is divided into local blocks. A vector of pointers to per-thread `LocalState` objects is kept as a member variable of `Central2D` called `locals_` that can be indexed by the thread ID.

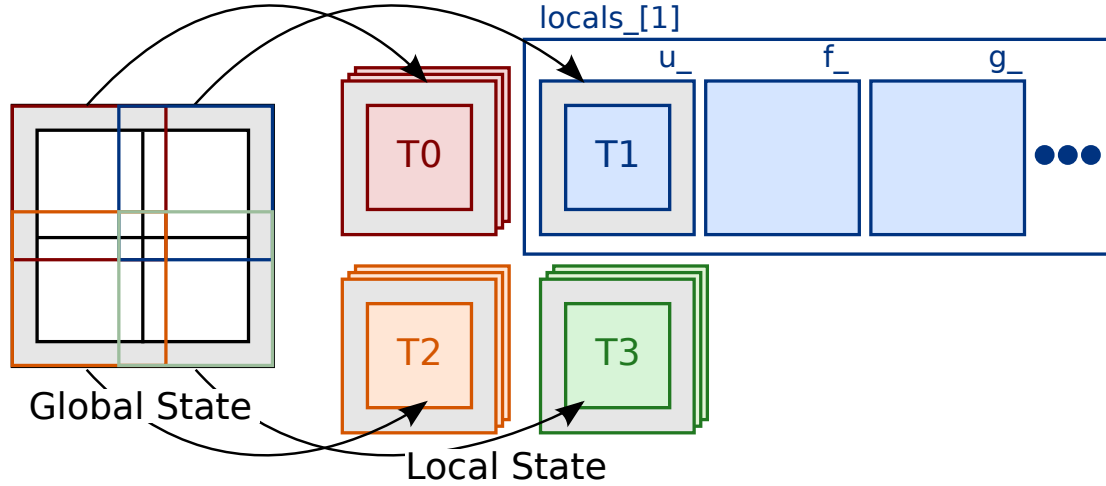


Figure 1: Parallelizing Computation Using Per-Thread Local State – In this example, the global grid is divided into four local blocks, each of which is copied to per-thread memory allocated separately from the global grid. The LocalState class encapsulates all vectors in a local block including u _, f _, g _, ux _, uy _, fx _, gy _, and v _. Each thread is only responsible for computing the solution of the live cells (i.e., non-gray area) in its local block. In order to do so, the vectors in the ghost cells (i.e., gray area) must be used. Note that the ghost cells of one local block overlap with the live cells of other local blocks grid, causing redundant computation.

To facilitate copying of vectors between the global and local states, we implement the `copy_to_local()` and `copy_from_local()` functions. Note that these functions can easily be parallelized since copying to the local state only has conflicting accesses on reads, and copying from the local state only writes the live cells in the global grid which do not overlap between local blocks. The ghost cells of the global grid are updated by the `apply_periodic()` function.

The pseudocode for our current parallelization is as follows:

```
while (t < tfinal) {
    apply_periodic();
    real dt = calculate_dt(compute_wave_speed());
    #pragma omp parallel num_threads(nthreads)
    {
        int tid = omp_get_thread_num();
        copy_to_local(tid);
        for (int io : sub_steps) {
            compute_flux(tid);
            limit_derivs(tid);
            compute_step(tid);
        }
        copy_from_local(tid);
    }
    t = update_time(dt);
}
```

The preliminary results comparing the performance of the serial and parallel implementations of the shallow water equation solver running the compute nodes is shown in Figure 2.

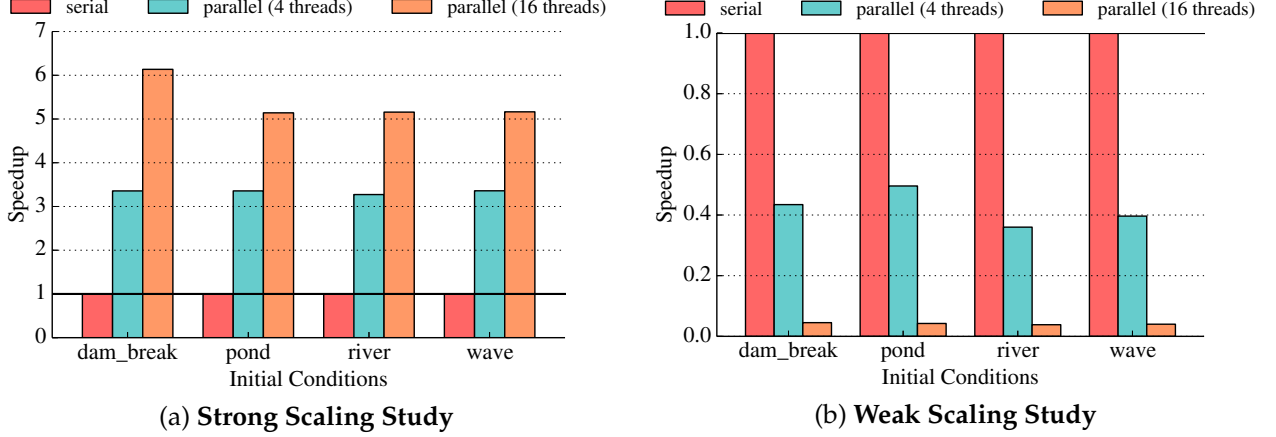


Figure 2: Performance Results of Parallel Implementation of Shallow Water Equation Solver Running on Compute Nodes – Performance of the parallel implementation of the shallow water equation solver running on the Totient compute nodes compared against the provided serial implementation for all initial conditions. All speedups are the execution time normalized to the serial implementation. For the strong scaling experiment, we use the default 200x200 grid. For the weak scaling study, we increase the problem size at the same factor as the increasing number of threads (e.g., 200x200 grid for 1 thread, 400x400 grid for 4 threads, etc.).

Currently we only support square block sizes and block sizes that evenly divide the grid dimensions, so we only experiment with running 4 and 16 threads for the parallel implementation for all initial conditions. Overall, in a strong scaling study, we achieve speedups of 3X using 4 threads and 5–6X using 16 threads.

3.2. Parallelizing for the Accelerator Board

There are several ways we can leverage the Intel Xeon Phi accelerator boards to further increase the speedups we see with the parallelized implementation for the compute nodes. One approach is to execute the code natively on the accelerator itself with no fine-grain offloading. Although this approach makes it easier to tailor the code for the accelerator and reduces overheads of copying data between the host and the device memories, we are forced to run all the computation from start to end on the accelerator. Another approach is to only offload certain sections of the computation to the accelerator. With this approach, there is a greater burden on the programmer to identify and parallelize the most compute-intensive section of the code, but allows him or her more flexibility in choosing which sections to accelerate.

As a starting point, we choose the second approach of offloading specific sections of the computation to the accelerator and build off of the parallel implementation for the compute nodes discussed in Section 3.1. In this naive first-pass, we offload the parallel section of the code and pass in the dimensions of the global grid and the blocks, as well as a pointer to the global grid itself. Inside the offloaded kernel, we still spawn off the specified number of threads normally, except that instead of using pre-allocated member LocalState objects, each thread creates its own LocalState object on the stack. The `copy_to_local()` and `copy_from_local()` functions were modified to copy the flattened elements of the global grid passed in by the host to the local grid vectors. Once the copy is complete, the other functions in the offloaded kernel can take a pointer to the per-thread LocalState object as an argument and access the vectors in this object similar to before.

Although this naive implementation requires us to offload computation before every timestep, there is an unnecessarily high overhead of transferring data between the host and device memories, but we believe that using efficient batching will help us alleviate this issue. Currently, the parallel implementation for the

accelerator board runs roughly an order of magnitude slower than the implementation for the compute nodes with the same number of threads.

4. Tuning the Shallow Water Simulation

The following optimizations are explored in this section:

- Batching time-steps with ghost cells
- Vectorizing with AVX extensions
- Eliminating redundant computation
- Offloading vs. natively executing the parallel kernel
- Overlapping computation with communication
- Implementing alternative physics modeling algorithms

4.1. Batching Time-Steps with Ghost Cells

4.2. Vectorizing with AVX Extensions

Vectorizing with AVX Extensions can fall under two general categories:

1. Having the compiler auto-vectorize your code for you, and
2. Writing your own vectorized kernels for use as subroutines.

Both approaches have benefits and detriments. For example, when using the compiler to vectorize your code you save yourself the pain of having to reason about which register has what data, what SSE/AVX function callbacks wrap the appropriate instruction level code, etc. This provides great convenience, but does come at a cost. The compiler is only as smart as you let it be, and if you arrange your code improperly, you may not only slow down your program but you may very well yield the wrong result by accident. Writing your own vectorized code, on the other hand, can be quite difficult to get right. Often times it may be best to take a mixed approach, where you examine the instructions the compiler generated for a given code segment and determine whether or not you can refine this approach further.

4.2.1. Auto-vectorization Using ICC

As discussed previously, you can use the `amplxe` tool to help with profiling, as well as use the intel compiler to generate an `optprt` file describing what was / was not vectorized, how effective it was if vectorization occurred, and why vectorization did not occur if that were the case. On the note of when vectorization does not occur, we would like to mention a couple of things:

1. Not all loops are created equal

That is not every loop can be vectorized, either because its length cannot be known at compile-time, or because the data elements being accessed cannot be made to execute in parallel.

2. Understanding the codes in the .optrpt file can help you rearrange your code so that it is vectorized (recalling from 1 that not all of this is possible).

We found the following presentation extremely useful in rearranging loop statements:

<https://engineering.purdue.edu/milind/ece573/2011spring/lecture-14.pdf>

3. Introducing compile-time constants / constexpr members of a class can go a long way in assisting the compiler understand what can / cannot be vectorized.
4. Taking great care to enforce memory alignment as well as declare said alignment to the compiler will also enable it to vectorize even more.

We leave items 1 and 2 as an exercise for the reader ;) For item 3, we must first acknowledge that the purpose of the data type `std::array` is largely just for compilation hints, and we can wield this to our advantage. In the original implementation of `Shallow2d.h`, we had that

```
// Type parameters for solver
typedef float real;
typedef std::array<real,3> vec;
```

were the primary solver types used throughout the program. The issue, though, is that regardless of us declaring with `<real,3>`, with high probability (given the architectures we are compiling on) this will get padded to 16 bytes regardless. The issue with this padding though, is that it is not guaranteed to be a `float` and treating it as such can potentially give problems. Recognizing this, we can modify this definition to be:

```
// Type parameters for solver
#define VEC_DIM 4 // change this and we all die...
typedef float real;
#ifdef __INTEL_COMPILER
    typedef __declspec(aligned(16)) std::array<real, VEC_DIM> vec;
#else // GCC
    typedef __attribute__((aligned(16))) std::array<real, VEC_DIM> vec;
#endif

// allow loop unrolling over 'vec'
static constexpr int vec_size = VEC_DIM;
static constexpr int VEC_ALIGN = 16;
```

Woah. Ugly. But necessary. What this does is ensure that we will have 4 floats per type `vec`, and depending on the compiler you are using also declares the alignment of this type. This stage may not be necessary. We also define two `static constexpr` members to allow us to use say `Physics::vec_size` as a compile time constant to allow us to unroll loops / `__assume_aligned` on memory. For example, the original computation of the corrector step in `Central2d.h` was:

```
// Corrector (finish the step)
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] );
        }
    }
}
```

Though extremely verbose, we can now use these new additions to write a new loop:

```
// Corrector (finish the step)
for (int iy = nghost-io; iy < ny+nghost-io; ++iy) {
    for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {
        /* Nomenclature:
         *   u_x0_y0 <- u(ix, iy)
         *   u_x1_y0 <- u(ix+1, iy)
         *   u_x0_y1 <- u(ix, iy+1)
         *   u_x1_y1 <- u(ix+1, iy+1)
         */
    }
```



```
LOOP END
LOOP END
```

The last stage in this phase, which we are uncertain if it is working correctly, is to create a custom allocator for `std::vector`. Unfortunately there is no byte aligned allocator in the standard yet, so we snagged somebody else's that seems to be popular on the interweb:

<https://gist.github.com/donny-dont/1471329>

We believe by virtue of the fact that it gives the right simulation that we at least have a valid allocator, but honestly at this point we are very frustrated with how difficult it has been to use `std::vector`. After declaring

```
#define BYTE_ALIGN 64
#ifdef __INTEL_COMPILER
    typedef __declspec(align(BYTE_ALIGN)) std::vector<vec, aligned_allocator<vec, BYTE_ALIGN>> aligned_vector;
#else // GCC
    typedef __attribute__((aligned(BYTE_ALIGN))) std::vector<vec, aligned_allocator<vec, BYTE_ALIGN>> aligned_vector;
#endif
aligned_vector u;           // Solution values
aligned_vector f_x;         // Fluxes in x
aligned_vector g_y;         // Fluxes in y
aligned_vector ux;          // x differences of u
aligned_vector uy;          // y differences of u
aligned_vector fx;          // x differences of f
aligned_vector gy;          // y differences of g
aligned_vector v;           // Solution values at next step
```

it turns out that memory aligned allocators (in general) greatly conflict with other locations in the code that were previously vectorized. So although we were able to get some interesting vectorization results from the above, we may switch over to C-style arrays in the near future. The chaos of the above code, in particular the newly vectorized loop, do not seem worth the trade-off of how ugly / difficult to follow it is.

We may also explore other allocators e.g. using Eigen's aligned allocator, but generally do not see the benefit as we have probably spent more time trying to utilize these vectors through arcane trickery than we would have to just rewrite it based off of say `float *u` and use a memory aligned malloc. Time will tell whether we decide to keep these or not.

4.2.2. Manual Vectorization

When it comes to writing custom kernels, we actually are very excited to do this. After much deliberation we have finally been able to get `#pragma offload target(mic)` to cooperate with things like `std::vector` and `std::array`.

Getting them working on the Phi's was as far as we got at this point, as discussed in section 3.2, so we hope to be able to play with this more and compare with the intel compiler's vectorization and *maybe* even beat it! If anything, we can just pull the same trickery we did in the corrector step described in the previous section.