CS 5220 Assignment 2: Shallow Water Simulation

Unmukt Gupta, Eric Lee, Scott Wu

1 Introduction

The purpose of this project is to improve the performance of a shallow water simulation. The shallow water simulation is a grid based physics simulation where every cell is dependent on the the neighboring cells. We begin with a simple framework of the simulator which implements the shallow water physics. Then we analyze the code and optimize bottlenecks and data structures. Finally, we will parallelize the simulation by splitting the problem into smaller blocks.

2 Profiling and Analysis

2.1 Profiling

We first used Intel VTune (`amplxe-cl`) to get a sense of the performance bottlenecks. After generating the report, we find three function calls that take the majority of our time.

```
Function                                               Module          CPU Time
-----------------------------------------------------  ------------    --------
Central2D<Shallow2D, MinMod<float>>::limited_derivs     shallow         1.456s
Central2D<Shallow2D, MinMod<float>>::compute_step      shallow         0.700s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds  shallow         0.369s
_IO_fwrite                                              libc-2.12.so    0.021s
...
```

First, the `limited_derivs` function computes derivatives in all 4 directions for every cell. The actual derivative is done in the MinMod class.

Second, the `compute_step` function computes a predictor and corrector, both which require an iteration over the entire grid.

Third, the `compute_fg_speeds` function initializes the values at each cell using the Physics module.

2.2 Vectorization Report

The vectorization report (written to ipo_out.optrpt) demonstrates very little vectorization in those three functions of interest.

In `limited_derivs`, the compiler claims dependence within the inner loop. Upon inspection, there does not appear to be any read/write conflicts (at most multiple reads from the same vector).

In `compute_step`, none of the nested loops were vectorized. In the predictor, corrector and the copy loops, there is no dependency, and perhaps the compiler is assuming that the arrays may be overlapping. This may be resolved by using the `restrict` keyword.

In `compute_fg_speeds`, vectorization seems to be blocked by the fact that we are computing the upper bound of the wave speed (write after read).

3 Implementation

3.1 Offloading to the Phis

Documentation on offloading to the Xeon Phis is sparse at best. We experimented with a toy problem (namely, a reduction operation on an array) to figure out the proper way to offload. First, we removed a number of flags in the serial version that caused problems when compiling instructions for the phis. Next, we include our own flags, namely "**-axCORE-AVX512,CORE-AVX2 -offload-attribute-target=mic**". The compiler will try to automatically offload work to the phis, but this works rather poorly. In practice, we have to actually write out offloading instructions. We refer to the toy problem of a reduction on an array:

```
double reduction(double *A){
	//sum up elements with a for loop
}
```

To do the same thing on the phis:

```
#pragma offload target(mic) in(A: length(foo)), out(sum: length(1))
{
	#pragma omp parallel for
	//sum up elements with a for loop and return sum
}
```

If we want to call reduction inside the offload region but declare it elsewhere:

```
#pragma offload target(mic) in(A: length(foo)), out(sum: length(1))
{
	reduction(sum, A);
}

__declspec( target(mic) ) void reduce(double *sum, double *A){
	#pragma omp parallel for
	//sum up elements with a for loop and return sum
}
```

There are a number of problems with this limited API. We note that the behavior of nested xeon phi functions declared with "__declspec" is erratic at best. Furthermore, experimentation revealed offloading to only support arrays of primitives.

The max number of threads on the phis is 236, corresponding to four openmp threads per core with the last core acting as a supervisor. You can also set the number of phi threads as an environment variable.

This necessitated a rewrite of the code base we were given (and something we are still currently working on).

3.2 OpenMP Testing

To test the OpenMP directives, we wrote a simple program to sum a collection of numbers.

```
double fn(double x) {
    for (int j = 0; j < fn_iters; j++) {
        x = cos(x + j);
    }
    return x;
}

double sum = 0.0;
for (int i = 0; i < iters; i++) {
    sum += fn(i);
}
```

We modified and added OpenMP directives to produce three different versions: serial, parallel and parallel with offloading. Additionally, we can adjust the `iters` and `fn_iters` parameters to achieve different timing results.

```
// serial.cpp
double sum = 0.0;
for (int i = 0; i < iters; i++) {
    sum += fn(i);
}

// parallel.cpp
double totalsum = 0.0;
#pragma omp parallel shared(totalsum)
{
    double sum = 0.0;
    #pragma omp for
    for (int i = 0; i < iters; i++) {
        sum += fn(i);
    }

    #pragma omp atomic
    totalsum += sum;
}

// parallel_offload.cpp
double totalsum = 0.0;
#pragma offload target(mic)
#pragma omp parallel
{
    double sum = 0.0;
    #pragma omp for
    for (int i = 0; i < iters; i++) {
        sum += fn(i);
    }

    #pragma omp atomic
    totalsum += sum;
}
```

As we vary the two number of iterations, we begin to see the advantages and restrictions on offloading to the phis.

`iters` = 100 million	`fn_iters` = 1	
Serial	**1.194437 s**	
Parallel		**0.203028 s**
Offload	**3.497940 s**	

`iters` = 1 million	`fn_iters` = 1000	
Serial	**6.975630 s**	
Parallel		**0.418235 s**
Offload	**0.810991 s**	

`iters` = 1000	`fn_iters` = 1 million	
Serial	**10.775726 s**	
Parallel		**0.567881 s**
Offload	**0.836295 s**	

3.3 Changes to Codebase

The current design and types used in the simulator make it a bit difficult to auto-vectorize and offload the computation.

First, we hardcoded the grid size and the number of ghost cells. The number of ghost cells is at least 3, and we assume the grid is a square. By replacing the loop bounds with a defined constant, the compiler knows exactly what the iteration bounds are, and can better vectorize the loop.

Second, we changed the vector array of 3-element vectors to three respective native arrays. We can do this now because our grid sizes are defined. Additionally, we can add the `restrict` keyword to each arrays to resolve the vectorization issue. Consequently, we had to manually unroll a few of the loops for each vector element.

Third, we added some AVX instructions to our `compute_step` method. Since the phis support 32 512-bit registers, there is a ton of potential in utilizing these vector operations.

Finally, we added OpenMP parallel and offload directives to the outer loops.

4 Future Work

Our next steps in this project are to profile and analyze our parallel and offloaded implementations, and also to test domain decomposition. Currently the offloaded version is slower than the serial, and we need to run our offloaded program on a larger problem to see any performance improvements. In domain decomposition, we will split the grid into blocks for each processor. Since we have shared memory, and overall our computations are uniform and branchless, it may not be necessary to keep extra cells and synchronize every few steps.