Shallow Water Simulation

CS 5220: Homework 2

Group 1

Ze Jin (zj58) Jason Setter (jls548) Guo Yu (gy63)

1 Profiling

We refer to the use profiling tools on the note "Optimization and analysis tools".

1.1 VTune Amplifier

We can use the command line interface `amplxe-cl` to run the VTune GUI from our head node. The collection phase will run on the compute nodes in the cluster. Once the data has been collected, reports can be generated on the front-end node.

From the report below, we find out that we should give our priority to improving functions `limited_derivs`, `compute_step` and `compute_fg_speeds` in `Central2D`, because they take the majority of CPU time.

```
Function                                                  Module        CPU Time
--------------------------------------------------------  ------------  --------
Central2D<Shallow2D, MinMod<float>>::limited_derivs       shallow         1.356s
Central2D<Shallow2D, MinMod<float>>::compute_step         shallow         0.642s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds    shallow         0.229s
_IO_fwrite                                                libc-2.12.so    0.023s
```

```
[Outside any known module]                                        [Unknown]       0.019s
_IO_file_xsputn                                                   libc-2.12.so    0.014s
Central2D<Shallow2D, MinMod<float>>::solution_check               shallow         0.007s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame          shallow         0.005s
Central2D<Shallow2D, MinMod<float>>::run                          shallow         0.003s
Central2D<Shallow2D, MinMod<float>>::offset                       shallow         0.002s
```

1.2 Modular Assembly Quality Analyzer and Optimizer (MAQAO)

MAQAO is a relatively new tool that provides detailed analysis of the shortcomings of our code based on a static analysis of the generated assembly language.

From the report below, we know that we should give priority to improving function `compute_step` in `Central2D`, since it is the most problematic and promising one.

```
Section 1: Function: Central2D<Shallow2D, MinMod<float> >::compute_step(int, float)
================================================================================
Section 1.1: Source loop ending at line 338
========================================
Pathological cases
------------------
Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could
benefit from full vectorization.
Since your execution units are vector units, only a fully vectorized loop can use
their full power.
By fully vectorizing your loop, you can lower the cost of an iteration from 13.25
to 3.18 cycles (4.17x speedup).
Two propositions:
 - Try another compiler or update/tune your current one:
   * Intel: use the vec-report option to understand why your loop was not vectorized.
```

```
  If "existence of vector dependences", try the IVDEP directive. If, using IVDEP,
  "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
 - Remove inter-iterations dependences from your loop and make it unit-stride.
Detected EXPENSIVE INSTRUCTIONS, generating more than one micro-operation.
Only one of these instructions can be decoded during a cycle and the extra
micro-operations increase pressure on execution units.
VCVTSD2SS: 2 occurrences
VCVTSS2SD: 3 occurrences
 - Pass to your compiler a micro-architecture specialization option:
  * Intel: use axHost or xHost.
Fix as many pathological cases as you can before reading the following sections.
Bottlenecks
-----------
The ROB-read stage is a bottleneck.
By removing all these bottlenecks, you can lower the cost of an iteration from
13.25 to 12.00 cycles (1.10x speedup).
Section 1.2: Source loop ending at line 357
=========================================
Pathological cases
------------------
Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could
benefit from full vectorization.
Since your execution units are vector units, only a fully vectorized loop can use
their full power.
By fully vectorizing your loop, you can lower the cost of an iteration from 54.00
to 7.50 cycles (7.20x speedup).
Two propositions:
 - Try another compiler or update/tune your current one:
```

```
  * Intel: use the vec-report option to understand why your loop was not vectorized.
  If "existence of vector dependences", try the IVDEP directive. If, using IVDEP,
  "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
 - Remove inter-iterations dependences from your loop and make it unit-stride.
Detected EXPENSIVE INSTRUCTIONS, generating more than one micro-operation.
Only one of these instructions can be decoded during a cycle and the extra
micro-operations increase pressure on execution units.
VCVTSD2SS: 3 occurrences
VCVTSS2SD: 12 occurrences
 - Pass to your compiler a micro-architecture specialization option:
  * Intel: use axHost or xHost.
Fix as many pathological cases as you can before reading the following sections.
Bottlenecks
-----------
The FP add unit is a bottleneck.
Try to reduce the number of FP add instructions.
By removing all these bottlenecks, you can lower the cost of an iteration from 54.00
to 40.50 cycles (1.33x speedup).
Section 1.3: Source loop ending at line 364
========================================
Pathological cases
------------------
Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could
benefit from full vectorization.
Since your execution units are vector units, only a fully vectorized loop can use
their full power.
By fully vectorizing your loop, you can lower the cost of an iteration from 3.00 to
0.75 cycles (4.00x speedup).
```

```
Two propositions:
 - Try another compiler or update/tune your current one:
  * Intel: use the vec-report option to understand why your loop was not vectorized.
  If "existence of vector dependences", try the IVDEP directive. If, using IVDEP,
  "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
 - Remove inter-iterations dependences from your loop and make it unit-stride.
Fix as many pathological cases as you can before reading the following sections.
Bottlenecks
-----------
The store unit is a bottleneck.
Try to reduce the number of stores.
For example, provide more information to your compiler:
 - hardcode the bounds of the corresponding 'for' loop,
By removing all these bottlenecks, you can lower the cost of an iteration from 3.00
to 2.25 cycles (1.33x speedup).
```

2 Parallelization

We use OpenMP to parallelize our code, and start with a naive parallelization (e.g. parallelizing the for loops in the various subroutines).

2.1 `Central2D<Shallow2D, MinMod<float>>::limited_derivs`

2.1.1 `for`

We apply the `parallel` region and `for` directive.

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::limited_derivs()
{
```

```
    int iy, ix;
    #pragma omp parallel private(iy, ix)
    {
        #pragma omp for
        for (iy = 1; iy < ny_all-1; ++iy)
            for (ix = 1; ix < nx_all-1; ++ix) {

                // x derivs
                limdiff( ux(ix,iy), u(ix-1,iy), u(ix,iy), u(ix+1,iy) );
                limdiff( fx(ix,iy), f(ix-1,iy), f(ix,iy), f(ix+1,iy) );

                // y derivs
                limdiff( uy(ix,iy), u(ix,iy-1), u(ix,iy), u(ix,iy+1) );
                limdiff( gy(ix,iy), g(ix,iy-1), g(ix,iy), g(ix,iy+1) );
            }
    }
}
```

2.1.2 for

We rewrite the two-for loop as one-for loop, then apply the `parallel` region and `for` directive.

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::limited_derivs()
{
    int ixy, iy, ix;
    #pragma omp parallel private(iy, ix)
    {
        #pragma omp for
```

```
        for (ixy = 0; ixy < (nx_all-2)*(ny_all-2); ++ixy) {
            ix = ixy % (nx_all-2) + 1;
            iy = ixy / (nx_all-2) + 1;

            // x derivs
            limdiff( ux(ix,iy), u(ix-1,iy), u(ix,iy), u(ix+1,iy) );
            limdiff( fx(ix,iy), f(ix-1,iy), f(ix,iy), f(ix+1,iy) );

            // y derivs
            limdiff( uy(ix,iy), u(ix,iy-1), u(ix,iy), u(ix,iy+1) );
            limdiff( gy(ix,iy), g(ix,iy-1), g(ix,iy), g(ix,iy+1) );
        }
    }
}
```

2.1.3 sections

We apply the `parallel` region and `sections` directive.

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::limited_derivs()
{
    int iy, ix;
    #pragma omp parallel private(iy, ix)
    {
        for (iy = 1; iy < ny_all-1; ++iy)
            for (ix = 1; ix < nx_all-1; ++ix) {

                #pragma omp sections
```

```
                {
                    // x derivs
                    #pragma omp section
                    limdiff( ux(ix,iy), u(ix-1,iy), u(ix,iy), u(ix+1,iy) );
                    #pragma omp section
                    limdiff( fx(ix,iy), f(ix-1,iy), f(ix,iy), f(ix+1,iy) );

                    // y derivs
                    #pragma omp section
                    limdiff( uy(ix,iy), u(ix,iy-1), u(ix,iy), u(ix,iy+1) );
                    #pragma omp section
                    limdiff( gy(ix,iy), g(ix,iy-1), g(ix,iy), g(ix,iy+1) );
                }
            }
    }
}
```

2.2 Central2D<Shallow2D, MinMod<float>>::compute_step

2.2.1 for

We apply the `parallel` region and `for` directive for two blocks.

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::compute_step(int io, real dt)
{
    real dtcdx2 = 0.5 * dt / dx;
    real dtcdy2 = 0.5 * dt / dy;

    int iy, ix, m;
```

```
vec uh;


#pragma omp parallel private(iy, ix, uh, m)
{
    #pragma omp for
    // Predictor (flux values of f and g at half step)
    for (iy = 1; iy < ny_all-1; ++iy)
        for (ix = 1; ix < nx_all-1; ++ix) {
            uh = u(ix,iy);
            for (m = 0; m < uh.size(); ++m) {
                uh[m] -= dtcdx2 * fx(ix,iy)[m];
                uh[m] -= dtcdy2 * gy(ix,iy)[m];
            }
            Physics::flux(f(ix,iy), g(ix,iy), uh);
        }


    #pragma omp for
    // Corrector (finish the step)
    for (iy = nghost-io; iy < ny+nghost-io; ++iy)
        for (ix = nghost-io; ix < nx+nghost-io; ++ix) {
            for (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] );
                }
            }
    }

    // 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);
        }
    }
}
```

3 Method

We assemble modules to obtain different methods as follows.

3.1 Version 1

We combine 2.1.1 and 2.2.1 to get the method of version 1.

3.2 Version 2

We combine 2.1.2 and 2.2.1 to get the method of version 2.

4 Performance

We set up both strong and weak scaling studies, varying the number of threads we employ.

4.1 Strong Scaling Study

We fix the number of cells per side $nx = 200$ and vary the number of threads $p = 1, \cdots, 8$.
From the Figure 1 below, we can see that the speedup is roughly linear in the number of threads, which reaches its maximum 3.2 at 8 threads.

4.2 Weak Scaling Study

We vary both the number of cells per side $nx = 200 * p$ and the number of threads $p = 1, \cdots, 8$.
From the Figure 2 below, we can see that the speedup is roughly linear in the number of threads at the beginning and then flattens out, which reaches its maximum 2.6 at 5 threads.

Reference

(1) David Bindel, Shallow water simulation, Applications of Parallel Computers (CS 5220), Fall 2015.
(2) David Bindel, Optimization and analysis tools, Applications of Parallel Computers (CS 5220), Fall 2015.
(3) Blaise Barney, OpenMP, https://computing.llnl.gov/tutorials/openMP.

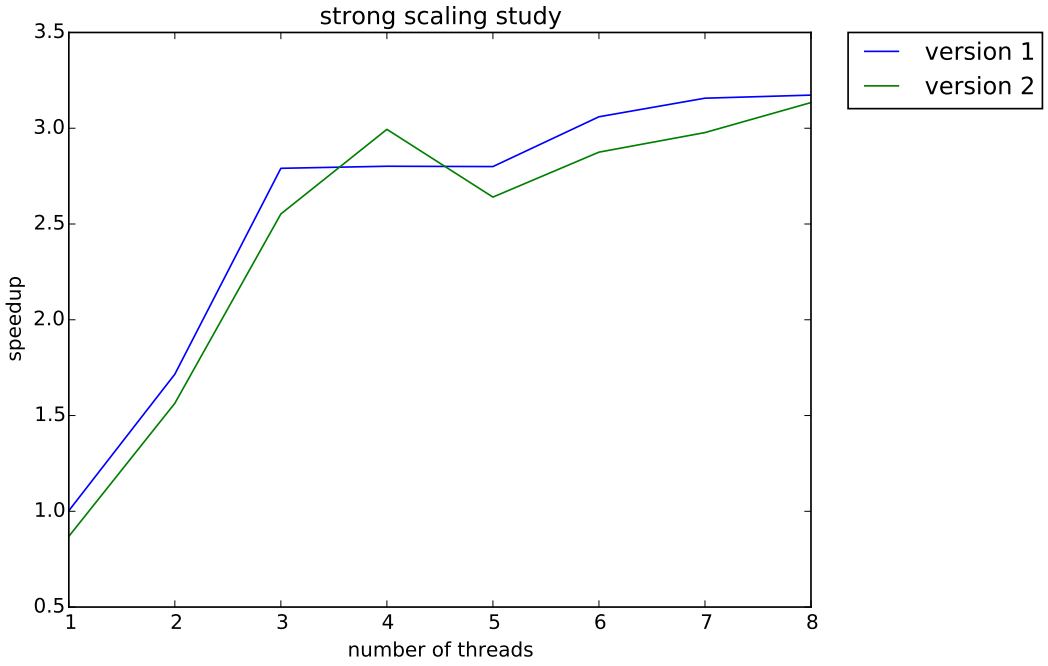

Figure 1: strong scaling

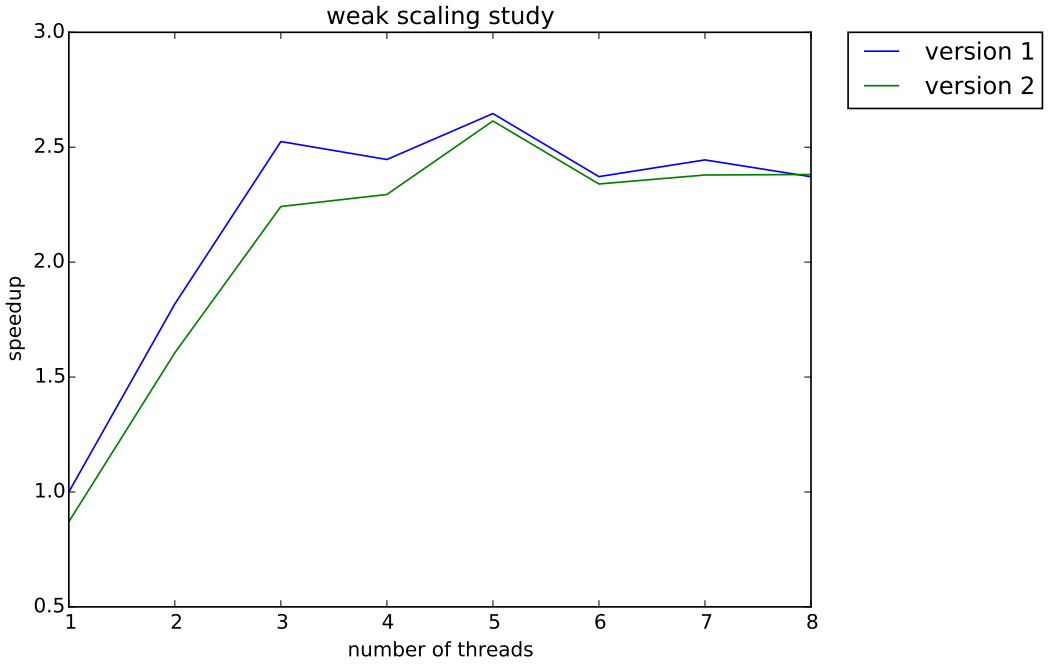

Figure 2: weak scaling