PHY905, section 4
Project Report

Håkon V. Treider
MSU id: treiderh

May 6, 2016

Introduction

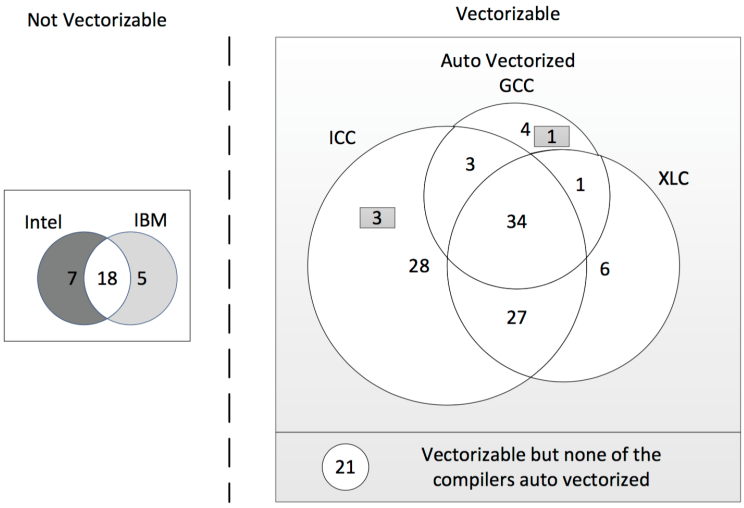
The code has been run on Mac OS X (10.11.4) with a 2.5 GHz Intel Core i7.

As the reader may become aware of, this project has drifted in another direction than was initially described in the project proposal. I will give some short reasoning behind this and talk briefly about my motivation for doing this specific project.

This autumn I will start working on my Master Thesis back home at the University of Oslo. The described topic will be molecular dynamics simulations (MD) of neutron star crust. A big part of the thesis is actually writing the MD code myself. This is encouraged by the professors, as coding and numerical simulations quickly is becoming one of the most important tools in physics, both for theorist and experimentalists. However, in order to get a reasonably sized domain (in space and time), the use of clever optimization techniques like parallelization and vectorization are not just useful, but compulsory. This level of optimization is quite difficult, and I came to realize that I needed to focus on a smaller, more specific target for this project.

That was when I got word of the ISPC compiler: `https://ispc.github.io`. From their webpage: *ISPC compiles a C-based SPMD programming language to run on the SIMD units of CPUs and the Intel Xeon Phi^{TM} architecture; it frequently provides a 3x or more speedup on CPUs with 4-wide vector SSE units and 5x-6x on CPUs with 8-wide AVX vector units, without any of the difficulty of writing intrinsics code. (...)*

In this course we have seen times, and times again that compilers try to autovectorize, but fails to do so in a number of cases - and different compilers tackles different problems. In lecture 13 we were presented with figure 1, which illustrates this problem. After some googling, I found the proposed topic for exploration on the GROMACS website: *Explore usablity of ispc for SIMD kernels*,



S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. *An Evaluation of Vectorizing Compilers*. PACT 2011.

Figure 1: Auto-vectorization differences across different compilers

(`http:www.gromacs.orgProject_ideas#Explore_usablity_of.c2.a0ispc_for_SIMD_kernels`). So in short, I want to try this out on my own MD code in the future, but since ISPC requires some "getting used to", I thought it would be better to start off in simpler fashion. With my background from physics, I decided to simulate the 2D wave equation, and see how ISPC did in respect to this problem.

1 Description of the problem

The wave equation says that the second order derivative in time, is equal to the second order derivative in space, multiplied by the wave velocity squared. For two dimensions, you thus need the derivative in x- and y-direction:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \quad u(x, y, t) \tag{1}$$

The standard finite (central) difference approximation for second order derivatives is used:

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_{i,j}^{t+\Delta t} - 2u_{i,j}^{t} + u_{i,j}^{t-\Delta t}}{\Delta t^2} \tag{2}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j}^{t} - 2u_{i,j}^{t} + u_{i-1,j}^{t}}{\Delta x^2} \tag{3}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1}^{t} - 2u_{i,j}^{t} + u_{i,j-1}^{t}}{\Delta y^2} \tag{4}$$

When we combine all this terms into a single equation, we see that we can solve for the solution at the next time step, and thus creating a recursive formula. In this task I have assumed the spatial mesh grid to be similar in x- and y-direction, meaning we can factor out Δx^2, since its the same for y:

$$u_{i,j}^{t+\Delta t} = 2u_{i,j}^{t} - u_{i,j}^{t+\Delta t} + \tag{5}$$

$$\frac{c^2 \Delta t^2}{\Delta x^2}\left(u_{i+1,j}^{t} + u_{i-1,j}^{t} + u_{i,j+1}^{t} + u_{i,j-1}^{t} - 4u_{i,j}^{t}\right) \tag{6}$$

Assuming the wave velocity to be constant over the domain, a (deeper) analysis of stability shows that the equation is numerically stable when the following condition is met: (In my code this value is set to 0.5, which basically means whatever waves we have propagate slower)

$$\frac{c^2 \Delta t^2}{\Delta x^2} \leq 1 \tag{7}$$

Some notes on the setup

I have used Dirichlet boundary conditions, which means the boundary of the domain is always equal to zero. Thus we only need to worry about the points on the inside, and no special care needs to be taken with edge points and corners. With Neumann boundary conditions this would be similar to using ghost points, but these would not be part of the actual solution.

The serial, OpenMP and ISPC versions of the 2D wave solver can be seen in section 1. The "interior" of the double loop over the domain is the same for all of the three different solvers. However, ISPC uses a different syntax for things like how the loop is written etc. When compiling only the ISPC solver with ISPC, it creates in addition to an object, a header file which we import into the main program.

The problem is pleasantly parallelizable, as both arrays that we read values from stay constant during each time step, and the array that are written to, is only written to once per matrix element and never read. This means the OpenMP code can be implemented very simply. After each time step is finished, we swap the pointers of `u` and `u_previous` - and then - `u` and `u_next` with `std::swap`.

Does it work?

A proper way to test this is to construct a test case where we know the exact solution, and test the solvers against this. We can then increase the time resolution bit-by-bit and see if the logarithm of the rate of convergence (of the solution), r, converges to 2. Since the error in our finite difference approximation is inversely proportional to the time step squared, r should converge to 2. This is quite tedious and typically requires us to add a counteracting source term $f(x, y, t)$ to the wave equation, so I have skipped this part. However, visually confirming that the solution looks "as it should", has been done by writing the solution to file and then plotting it in 3D with `matplotlib.pyplot` in python. This can be seen in figure 2. Given an initial distribution of a thin gaussian droplet, the output of all three solvers agree to machine precision, so I think it's pretty safe to say it works as it should!

Results

I have run the code with increasingly larger grid size for both 10 and 100 time steps. The results can be seen in table 1 and 2. They contain both the speed of the solver, and the speedup factors from serial code to OpenMP, serial code to ISPC and OpenMP to ISPC. The speed is defined to be how many million grid points can be computed each second.

Discussion

If we start out by ignoring the edge cases where the domain is small enough to fit entirely in cache, and the number of time steps is too low, we see that we get a consistent speedup of 6X with ISPC over the serial code, and 3X with OpenMP. I have four cores, so I set OpenMP to use 4 threads, which seemed to give the best results in this limit. For the ISPC compiler, they give on average a 5-6X increase in speed when used on processors with 8-wide AVX vector units (according to themselves). Since the wave equation is a simple algorithm, it seems to gain very good performance out of it! Keep in mind ISPC was only running on a single core, and was consistently 2X faster than OpenMP running on 4. This is really great motivating for the task at hand - making my own molecular dynamics code faster!

ISPC can of course also be parallelized to run on all cores. I am currently working on this, but it seems to be a lot more time consuming to get implemented correctly, than initially thought. (Optimistic til the bitter end!)

Performance model

Let's have a closer look at the algorithm itself. Per time step, the number of FLOPS is $9n_x n_y = 9n^2$. The number of loads and writes is $8n^2$. With N being

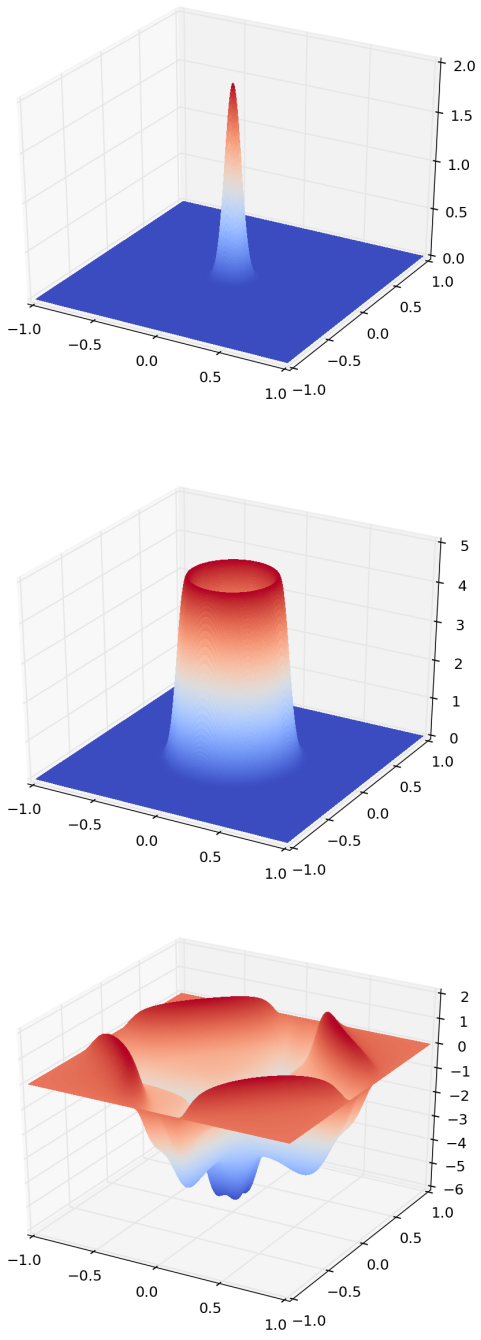


Figure 2: Solution of $u(x, y)$ at t=0,100 and 600.

Table 1: Computational speed for 2D wave equation (million grid points per second). Timesteps: 10

	Speed			Speedup factor		
Grid size	Serial	ISPC	OpenMP	ISPC / Serial	OpenMP / Serial	ISPC / OpenMP
128	168	1365	365	8.11	2.17	3.74
256	48	310	157	6.49	3.28	1.98
384	61	425	230	6.91	3.74	1.85
512	71	511	311	7.24	4.41	1.64
640	84	625	320	7.47	3.82	1.96
768	97	661	325	6.84	3.37	2.03
896	106	717	335	6.74	3.15	2.14
1024	117	744	328	6.34	2.79	2.27
1152	125	826	348	6.60	2.78	2.37
1280	137	893	378	6.54	2.77	2.36
1408	147	937	402	6.38	2.74	2.33
1536	152	949	420	6.25	2.77	2.26
1664	159	974	438	6.13	2.76	2.22
1792	159	878	462	5.52	2.90	1.90
1920	169	913	481	5.41	2.85	1.90
2048	170	840	486	4.95	2.86	1.73

the total number of time steps we get the following formula for the performance and the speed:

$$f(c,r) = n^2(9c+8r)N \tag{8}$$

$$s(c,r) = (10^6(9c+8r))^{-1} \tag{9}$$

$$\approx 170 \text{ mill. grid points/sec} \tag{10}$$

...using standard values, $c = 0.2 \cdot 10^{-9}$ and $r = 0.5 \cdot 10^{-9}$. What do we get from our test runs? Well, a convergence towards 170 million grid points per second. That is of course a high dose of pure luck, but still, it means we are getting close to the performance ceiling with the serial code. In turn, this means that ISPC and OpenMP are very close to the limits in terms of performance, since they give a speedup factor close to the theoretical value.

Table 2: Computational speed for 2D wave equation (million grid points per second). Timesteps: 100

	Speed			Speedup factor		
Grid size	Serial	ISPC	OpenMP	ISPC / Serial	OpenMP / Serial	ISPC / OpenMP
128	228	2298	614	10.07	2.69	3.74
256	99	695	256	6.99	2.57	2.72
384	68	455	213	6.71	3.14	2.14
512	72	440	256	6.08	3.55	1.71
640	85	535	324	6.28	3.81	1.65
768	98	663	362	6.76	3.69	1.83
896	108	737	374	6.79	3.45	1.97
1024	116	772	348	6.67	3.01	2.22
1152	124	821	357	6.63	2.88	2.30
1280	128	845	358	6.59	2.80	2.36
1408	136	866	385	6.37	2.84	2.25
1536	141	934	396	6.62	2.81	2.36
1664	146	949	425	6.48	2.90	2.23
1792	152	972	450	6.39	2.96	2.16
1920	157	956	462	6.08	2.94	2.07
2048	157	959	487	6.11	3.10	1.97

Code

Serial code

```
static inline int index(int i, int j, int size) {
    return j * (size+2) + i;
}

void waveSERIAL(int size, float c,
                        float u[], float u_p[], float u_n[]){
    for (int i = 1; i < size+1; i++) {
        for(int j = 1; j < size+1; j++) {
            float uxx = u[index(j+1, i, size)] +   u[index(j-1, i, size)];
            float uyy = u[index(j, i+1, size)] +   u[index(j, i-1, size)];
            float utt = u[index(j, i, size)]*2 - u_p[index(j, i, size)];
            u_n[index(j,i,size)] = c*(uxx + uyy - 4*u[index(j, i, size)]) + uut;
        }
    }
}
```

OpenMP code

```
#include <libiomp/omp.h>

static inline int index(int i, int j, int size) {
    return j * (size+2) + i;
}

void waveOMP(int size, float c,
                        float u[], float u_p[], float u_n[], int omp_threads){
    omp_set_num_threads(omp_threads);
#pragma omp parallel for
    for (int i = 1; i < size+1; i++) {
        for(int j = 1; j < size+1; j++) {
            float uxx = u[index(j+1, i, size)] +   u[index(j-1, i, size)];
            float uyy = u[index(j, i+1, size)] +   u[index(j, i-1, size)];
            float utt = u[index(j, i, size)]*2 - u_p[index(j, i, size)];
            u_n[index(j,i,size)] = c*(uxx + uyy - 4*u[index(j, i, size)]) + uut;
        }
    }
}
```

ISPC code

```
static inline int index(int i, int j, int size) {
    return j * (size+2) + i;
}

export void waveISPC(uniform int size, uniform float c, uniform const float u[],
                            uniform const float u_p[], uniform float u_n[]){
    foreach(i = 1 ... size+1, j = 1 ... size+1) {
        float uxx = u[index(j+1, i, size)] +   u[index(j-1, i, size)];
        float uyy = u[index(j, i+1, size)] +   u[index(j, i-1, size)];
        float utt = u[index(j, i, size)]*2 - u_p[index(j, i, size)];
        u_n[index(j,i,size)] = c*(uxx + uyy - 4*u[index(j, i, size)]) + uut;
    }
}
```