Final Exam

Issued: September 02, 2021, 12:30
Hand in: September 02, 2021, 15:30

Last Name:	
First Name:	
Legi ID:	

With your signature you confirm that you:

- Have read and followed the exam directives
- Have solved the exam without any unauthorized help

Signature: _____

Grading table

Question	Maximum score	TA 1	TA 2	Score
Question 1	30			
Question 2	30			
Question 3	30			
Question 4	30			
Total:	120			

Question 1: Roofline Model (15 points)

Given the computers A and B:

		A	B
Peak Performance	[GFLOP/s]	1000	50
Peak Bandwidth	[GB/s]	50	250

a) Compute the ridge point of the two computers.

b) Which computer is going to reach a higher performance on a kernel with an operational intensity of 0.5 FLOP/B and why?

c) For which values of operational intensity does computer A reach a higher performance than computer B?

d) Consider two routines `First()` and `Second()` computing the trajectory of a particle driven by a force `F` and collecting some statistics. Both produce the same result but the second one replaces one memory access with an additional evaluation of the function `F`.

```
1  const int n = 1000;
2  float F(float);
3  float x[n + 2], u[n + 2], p;
4
5  void First() {
6    for (int i = 1; i <= n; ++i) {
7      float xi = x[i];
8      float ui = u[i];
9      float xip = xi + ui;
10     float f = F(xip);
11     x[i + 1] = xip;
12     u[i + 1] = ui + f;
13     p += u[i - 1] * f;
14   }
15 }
16
17 void Second() {
18   for (int i = 1; i <= n; ++i) {
19     float xi = x[i];
20     float ui = u[i];
21     float xip = xi + ui;
22     float f = F(xip);
23     x[i + 1] = xip;
24     u[i + 1] = ui + f;
25     p += (ui - F(xi)) * f;
26   }
27 }
```

Let ϕ be the number of floating point operations performed on one call of function `F`. By computing the operational intensity, show that `Second()` always has higher (or equal) performance than `First()` in terms of FLOP/s. Is it possible that `Second()` nevertheless has longer execution time for some values of ϕ? If yes, find the minimum such ϕ. Assume no cache, the code is executed on computer A, only floating point operations take time, memory access is needed only for arrays `x` and `u` and perfectly overlaps with floating points operations.

Question 2: Intel Intrinsics bug hunting (30 points)

a) The following code tries to vectorize the operation

$$b_i = a_{2i} + a_{2i+1} \quad , \quad i = 0, \ldots, N-1$$

for a given integer N and two arrays a and b of size $2N$ and N respectively. Identify the bugs in the given implementation. You should assume that SSE intrinsics with 128 bit registers are used and that header files (*.h) are correctly included. Also, you can assume that N is a multiple of 4 and that a and b are aligned to 16-byte addresses.

```
1  float a[2*N], b[N];
2  //...
3  //Assume that array a is initialized with proper values here
4  //...
5  __m128 *av = (__m128*)a;
6  __m128 *bv = (__m128*)b;
7  for (i = 0; i < N/2; i++)
8  bv[i] = _mm_hadd_ps(av[2*i], av[2*i+1]);
```

Note that if x and y are two vector registers that each hold 4 single-precision numbers then

```
__m128 result = _mm_hadd_ps(x, y);
```

is equivalent to

```
__m128 result;
result[0] = x[0] + x[1];
result[1] = x[2] + x[3];
result[2] = y[0] + y[1];
result[3] = y[2] + y[3];
```

b) The following code is a serial version of a 2×2 matrix-matrix multiplication

```
1  double A[4],B[4],C[4];
2  //...
3  //Assume A and B are initialized here
4  //
5  //C = AB
6  C[0] = A[0]*B[0] + A[1]*B[2];
7  C[1] = A[0]*B[1] + A[1]*B[3];
8  C[2] = A[2]*B[0] + A[3]*B[2];
9  C[3] = A[2]*B[1] + A[3]*B[3];
```

You are asked to write a vectorized version that uses $SSE2$ intrinsics. You can assume that all addresses are properly aligned in memory and that the correct header files are used. Your solution should operate on `__m128d` registers and only needs to use the following instructions:

- `_mm_load1_pd(double const * A)` : loads double-precision element from memory into both elements of the result. Equivalent to:

```
__m128d result;
result[0] = A[0];
result[1] = A[0];
```

- `_mm_loadu_pd(double const * A)` : loads two double-precision elements from memory into the result. Equivalent to:

```
__m128d result;
result[0] = A[0];
result[1] = A[1];
```

- `_mm_add_pd(__m128d A, __m128d B)`: Adds double-precision elements in A and B. Equivalent to:

```
__m128d result;
result[0] = A[0]+B[0];
result[1] = A[1]+B[1];
```

- `_mm_mul_pd(__m128d A, __m128d B)`: multiplies double-precision elements in A and B. Equivalent to:

```
__m128d result;
result[0] = A[0]*B[0];
result[1] = A[1]*B[1];
```

c) Vectorize the following code (arrays are 16-byte aligned):

```
1 float a[4*N], b[4*N], x;
2 //Assume that the arrays a and b are initialized here somehow
3 for (i = 0; i < 4*N; i++)
4   a[i] = b[i] + x;
```

Your solution should operate on `__m128` registers and only needs to use the following instructions:

- `_mm_set1_ps(float a)` : fills a vector register with a single-precision number. Equivalent to:

```
__m128 result;
result[0] = a;
result[1] = a;
result[2] = a;
result[3] = a;
```

- `_mm_add_ps(__m128 a, __m128 b)` : adds the elements of two single-precision vector registers. Equivalent to:

```
__m128 result;
result[0] = a[0]+b[0];
result[1] = a[1]+b[1];
result[2] = a[2]+b[2];
result[3] = a[3]+b[3];
```

Question 3: Julia Set (30 points)

The Julia set is the set of complex numbers $z = x + iy$ for which the series

$$z_{n+1} = z_n^2 + c, \tag{1}$$

does not diverge for $n \to \infty$. c is a complex constant, one gets a different Julia set for each c. The initial value z_0 is formed by the coordinates in the complex plane.

For this application we select $c = -0.624 + 0.435i$. In the special case where $|c| < 2$ it can be shown that the divergence occurs if $|z_n|$ becomes larger than 2 for some n.

In Figure 1, each pixel represents one value of z in the image plane. The pixel color is determined by the minimum n for which $|z_n| > 2$. For some pixels, such n either does not exist or is too large, so we stop iterating after $n = 1000$.

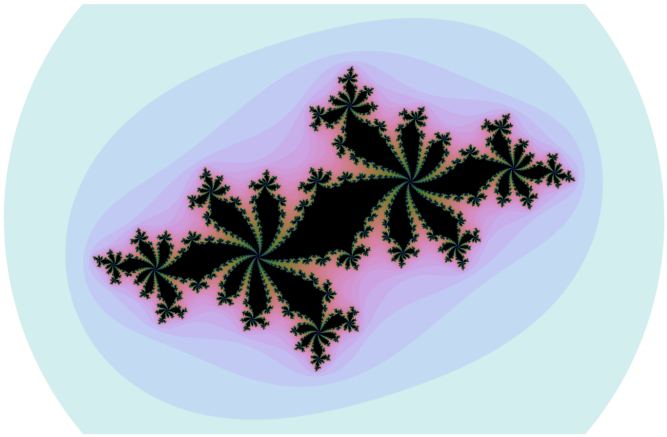

Figure 1: Visualization of the Julia set for $c = -0.624 + 0.435i$.

The given skeleton code computes the iteration count for each pixel. The Julia set can be visualized with the given plotting script. The commands `make && make run && make plot` compile the code, run it and plot the set.

a) Parallelize the function `julia_set` using OpenMP. Report the execution time for 1, 2 and 4 threads (use `make run`).

   Due to the symmetry, the code computes only the bottom half of the image.

b) The number of iterations may differ from pixel to pixel, which causes load imbalance. Improve your parallelization to fix the load imbalance issue, without adding a large overhead. Describe your approach. Report the execution time. Do you get the desired speed-up?

c) To avoid false sharing, different threads should access cache lines. Propose a simple way to incorporate that into your code.

d) Implement the function `compute_histogram` that will tell you how much of each n, i.e. each pixel color, appears in the Julia set. Minimize the usage of critical regions, locks and atomics.

Question 4: Particles MPI (30 points)

In this task you will implement an N-body solver describing a system of positively charged inertialess particles with MPI parallelization. The provided skeleton code

- seeds N particles on a unit circle

$$\boldsymbol{x}_i = (x_i, y_i) = \big(\cos\frac{2\pi i}{N},\ \sin\frac{2\pi i}{N}\big), \qquad i = 0, \ldots, N-1;$$

  distributing them over P ranks such that each rank takes N/P particles (assume N is divisible by P);
- performs time steps calling functions `Step()` and `PrintStat()`;
- writes the particle positions to files `init.dat` and `final.dat`.

a) Implement `Step()` which computes the forces and advances the particles

$$\boldsymbol{f}_i = \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \frac{\boldsymbol{x}_i - \boldsymbol{x}_j}{|\boldsymbol{x}_i - \boldsymbol{x}_j|^3},$$
$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \Delta t\, \boldsymbol{f}_i^t,$$

where $|\boldsymbol{x}| = \sqrt{x^2 + y^2}$ and Δt is a given time step.
Each rank can make $\mathcal{O}(P)$ MPI calls[1] and use $\mathcal{O}(N/P)$ bytes of memory. Your solution will be given more points for overlapping communication and computation. You may use `make` to build the executable, `make run` to run it on 2 processors and `make plot` to create `image.png` with particle positions as shown in Figure 2.

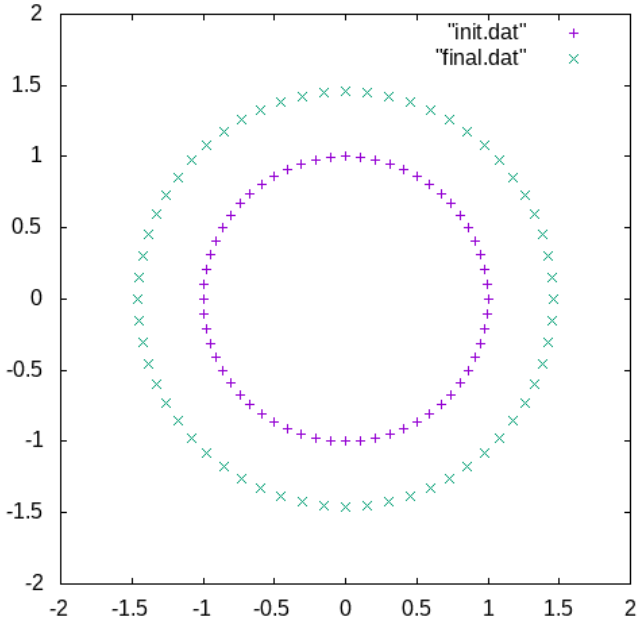

Figure 2: Expected output `image.png` produced by `make plot`.

[1] Notation $p = \mathcal{O}(q)$ means that $p < Mq$ for a large enough constant M.

b) Implement `PrintStat()` which computes and prints the mean and variance of the radial distance $r_i = |\boldsymbol{x}_i|$

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} r_i,$$

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} r_i^2 - \mu^2.$$

Expected output after 10 time steps:

```
mean=1            var=0
mean=1.0672       var=6.6613e-16
mean=1.1261       var=6.6613e-16
mean=1.1791       var=0
mean=1.2274       var=-8.8818e-16
mean=1.272        var=8.8818e-16
mean=1.3135       var=-2.2204e-16
mean=1.3524       var=4.4409e-16
mean=1.3891       var=4.4409e-16
mean=1.4239       var=1.3323e-15
mean=1.4571       var=0
```

Rounding errors may lead to differences up to 10^{-3} for the mean and 10^{-14} for the variance.