Advanced Topics in Numerical Analysis, High Performance Computing: Assignment 2

Mariya Savinov, mariyasavinov@nyu.edu

Repo location: `https://github.com/mariyasavinov/HPC_Spring_22.git`

1 Optimizing matrix-matrix multiplication

1.1 Loop arrangement

We test three options for loop ordering, which give the slowest performance, moderate performance, and the fastest performance. The options are:

	Outmost loop	Middle loop	Innermost loop
Option 1	$j = 1 : n$	$i = 1 : m$	$p = 1 : k$
Option 2	$p = 1 : k$	$i = 1 : m$	$j = 1 : n$
Option 3	$j = 1 : n$	$p = 1 : k$	$i = 1 : m$

For Option 1, we have

```
 Dimension       Time    Gflop/s       GB/s        Error
      1504   6.489317   1.048515   8.393695 0.000000e+00
      1552   7.891938   0.947374   7.583876 0.000000e+00
      1600   9.993178   0.819759   6.562173 0.000000e+00
      1648  11.725248   0.763448   6.111292 0.000000e+00
      1696  14.251409   0.684620   5.480191 0.000000e+00
      1744  16.235304   0.653445   5.230557 0.000000e+00
      1792  20.011666   0.575123   4.603552 0.000000e+00
      1840  21.165341   0.588651   4.711771 0.000000e+00
      1888  22.683473   0.593371   4.749483 0.000000e+00
      1936  26.042871   0.557259   4.460376 0.000000e+00
      1984  28.581064   0.546483   4.374067 0.000000e+00
```

For Option 2, we have

```
 Dimension       Time    Gflop/s       GB/s        Error
      1504  25.810046   0.263624   2.110393 0.000000e+00
      1552  28.633573   0.261114   2.090255 0.000000e+00
      1600  31.235441   0.262266   2.099441 0.000000e+00
      1648  34.974984   0.255943   2.048790 0.000000e+00
      1696  38.173253   0.255593   2.045947 0.000000e+00
      1744  42.253155   0.251079   2.009783 0.000000e+00
      1792  43.364373   0.265406   2.124434 0.000000e+00
      1840  50.433227   0.247040   1.977391 0.000000e+00
      1888  53.705776   0.250620   2.006018 0.000000e+00
      1936  59.156877   0.245324   1.963609 0.000000e+00
      1984  61.772022   0.252850   2.023821 0.000000e+00
```

For Option 3, we have

```
 Dimension       Time    Gflop/s       GB/s        Error
      1504   1.333713   5.101657  40.840390 0.000000e+00
      1552   1.458239   5.127155  41.043666 0.000000e+00
      1600   1.619741   5.057599  40.486077 0.000000e+00
      1648   1.756028   5.097653  40.805968 0.000000e+00
      1696   1.920342   5.080763  40.670069 0.000000e+00
      1744   2.091468   5.072456  40.602912 0.000000e+00
      1792   2.260437   5.091568  40.755278 0.000000e+00
      1840   2.463966   5.056485  40.473867 0.000000e+00
```

1888	2.676593	5.028676	40.250714	0.000000e+00
1936	2.889958	5.021744	40.194701	0.000000e+00
1984	3.100063	5.038305	40.326753	0.000000e+00

There are other options also, but we will now explain why Option 3 is the fastest of ALL possible choices (and thus inherently also explaining why Option 2 is the slowest of the options we tested). Recall that at the inside of the three loops, elements of A, B, and C are accessed by:

```
A_ip = a[i+pm];
B_pj = b[p+jk];
C_ij = c[i+jm];
```

Notice that a loop over i would access elements of A and C which are sequential in storage. A loop over p would access elements of B which are sequential in storage and elements of A which are m storage elements apart. Meanwhile, a loop over j would access elements of B and C which are k or m storage elements apart, respectively. To optimize memory access, it makes sense to try to access elements close together (sequentially ordered) as the matrix is loaded into the cache in chunks which are > 1 storage element in size.

Thus, the innermost loop should be over i to access elements of A and C, then the next inner loop should be over p to access elements of B close to each other. The loop over j should always be on the outermost loop as it provides no benefits with respect to chunk memory access. This explains why Option 3 is the fastest (as it follows our loop recommendation), as well as why Option 2 is the slowest of the tested choices since the loop over j is placed in the innermost loop (the least ideal location). An even slower option would be to swap the first two loops in Option 2.

In sum, the best performance is for the order presented in Option 3.

1.2 -O3 versus -O2 in case of Loop arrangement problem

The above results are with an -O3 optimization level flag. We now will also test with a -O2 optimization level flag specifically for the case of the fastest loop ordering (Option 3, j outermost, p middle loop, i innermost). For the same dimension sizes, we find that we get:

Dimension	Time	Gflop/s	GB/s	Error
1504	2.635640	2.581591	20.666462	0.000000e+00
1552	2.903790	2.574779	20.611505	0.000000e+00
1600	3.186875	2.570543	20.577196	0.000000e+00
1648	3.497229	2.559632	20.489483	0.000000e+00
1696	3.815919	2.556869	20.467011	0.000000e+00
1744	4.164501	2.547455	20.391323	0.000000e+00
1792	4.522243	2.545013	20.371469	0.000000e+00
1840	4.897135	2.544142	20.364200	0.000000e+00
1888	5.300973	2.539103	20.323585	0.000000e+00
1936	5.723151	2.535776	20.296686	0.000000e+00
1984	6.159435	2.535795	20.296583	0.000000e+00

This is approximately two times slower than using the -O3 optimization level flag. After implementing blocking and the parallel section, we find that this trend for optimization level flags still holds, so all further numbers are reported using the -O3 case.

1.3 Block size optimization

My particular machine Linux machine with 64-bit kernel architecture and Intel i7-8700 processor with 6 CPUs, 12 total threads, CPU speed 3.20 GHz. The command `lscpu` reports that the L1d and L1i caches are 32kB abd the L2 cache is 256kB – we suspect these sizes will determine the limit on timings for blocks of different sizes. In the blocking version of `MMult1`, a matrix-matrix multiplication of the form $\tilde{C} = \tilde{C} + \tilde{A}\tilde{B}$ is computed in the manner of `MMult0`. The idea is that these block portions of full matrices A, B, and C should be loaded into the cache all at once, and should not exceed the size of the L2 cache so they are the most quickly accessible. If 3 matrices of sizes $b \times b$ are stored in the L2 cache, that is a total of $3b^2 \times 8$ bytes. If the cap is 256kB, this suggests that at most we should use a block size $b \leq 103$.

So, we try $b = 20$, 40, 60, 80, 100, 104 block sizes and compare their timings as well as Gflop/s.

For $b = 20$, the first few dimensions and last few dimensions yield timings and Glop/s of

Dimension	Time	Gflop/s
20	0.282320	7.084212
60	0.285106	7.015496
100	0.283738	7.055814
...		
260	0.284007	7.054970
300	0.294731	6.962292
...		
1700	1.436145	6.841928
1740	1.533879	6.868889
...		
1900	2.007400	6.833716
1940	2.144940	6.808008
1980	2.281362	6.805051

For $b = 40$, the Glop/s increase:

Dimension	Time	Gflop/s
40	0.242918	8.233760
80	0.243210	8.227041
...		
200	0.244202	8.255465
240	0.246442	8.189766
280	0.245296	8.233242
...		
1720	1.300703	7.824148
1760	1.395504	7.813346
1800	1.496530	7.794029
1840	1.618860	7.696163
1880	1.691526	7.856422
1920	1.767604	8.008453
1960	1.926979	7.814859

and further for $b = 60$:

Dimension	Time	Gflop/s
60	0.229838	8.702470
120	0.232589	8.603264
180	0.232828	8.616711
240	0.234660	8.600958
300	0.238282	8.611650
...		
1740	1.253615	8.404533
1800	1.384800	8.422878
1860	1.541226	8.350307
1920	1.683825	8.406919
1980	1.849099	8.395865

There is an increase still for $b = 80$:

```
Dimension       Time    Gflop/s
       80   0.222690   8.985124
      160   0.226050   8.878726
      240   0.230100   8.771413
      320   0.235119   8.640808
      ...
     1760   1.262699   8.635116
     1840   1.446757   8.611682
     1920   1.645600   8.602199
```

We see the maximum Glop/s (and least runtime) for $b = 100$ block size:

```
 Dimension       Time    Gflop/s
       100   0.220060   9.097522
       200   0.227445   8.863682
       300   0.234741   8.741544
       ...
      1700   1.156547   8.495983
      1800   1.359572   8.579170
      1900   1.592443   8.614438
```

As soon as the block size is increased past the limit determined by the L2 cache size, i.e. increasing b to $b = 104$ (just slightly larger than the limit), we see an instant drop in the Glop/s from the numbers seen in the $b = 100$ case:

```
 Dimension       Time    Gflop/s
       104   0.232813   8.590607
       208   0.238537   8.450496
       312   0.239662   8.363906
       ...
      1768   1.350214   8.186045
      1872   1.600589   8.197240
      1976   1.894955   8.143139
```

Therefore, we conclude that the optimal value for `BLOCK_SIZE` is 100, for this machine as determined by the size of the L2 cache and observed in the output of MMul1.cpp.

1.4 Parallelization of blocking with OpenMP

After implementing a parallel version of MMult1, we find that the differences between the timings and Glop/s of the blocked and blocked+parallel code is minimal at the optimal block size $b = 100$ (In fact, for small blocks the OpenMP version is *worse* because the overhead is high). However, for larger block sizes $b = 104$ and $b = 120$, we see that the parallel version yields a speedup (quite dramatically for $b = 120$):

Block size	Dimension	timing	Gflops/s	parallel timing	parallel Gflops/s
100	1900	1.592443	8.614438	1.601912	8.563516
104	1976	1.894955	8.143139	1.850938	8.336790
120	1920	2.430863	5.823353	1.706006	8.297612

Note that the parallel section in these cases utilizes 6 threads at a time.

The theoretical peak Gflop/s for a machine with a single CPU can be found by finding the product of: (CPU speed)×(number of cores)×(CPU instructions per cycle). In our case this is

$$3.20\text{GHz} \times 6 \times 4 = 76.8 Gflops/s$$

So we find that at best, the **parallel blocking code achieves approximately** 11.15% **of the peak flop rate.**

2 OpenMP version of 2D Jacobi/Gauss-Seidel smoothing

2.1 Reporting timings

The Linux kernel architecture is 64-bit and the processor used for these timings is the Intel i7-8700 with 6 CPUs, 12 total threads, CPU speed 3.20 GHz.

We expect that Gauss-Seidel converges faster, so Gauss-Seidel should have overall shorter run-times over the Jacobi method. In both cases, using OpenMP should greatly decrease the runtime–in particular, as one goes from 1 to 6 threads, the runtime decreases with each additional thread. We test both methods for different choices of N (where the 2D grid is $(N+2) \times (N+2)$ in size) and for 1, 3, and 6 threads, reporting a few cases where the maximum number of iterations does not exceed 50,000 (the set maximum) for the Jacobi method.

	Jacobi, iterations	residual	runtime (s)	**Gauss Seidel**, iterations	residual	runtime (s)
N=50, 1 threads	7176	$9.99e-7$	5.59	3679	$9.99e-7$	2.71
N=50, 3 threads	7176	$9.99e-7$	2.05	3679	$9.99e-7$	0.98
N=50, 6 threads	7176	$9.99e-7$	1.18	3679	$9.99e-7$	0.55
N=80, 1 threads	18100	$1.00e-6$	51.63	9280	$1e-6$	24.62
N=80, 3 threads	18100	$1.00e-6$	18.96	9280	$1e-6$	8.75
N=80, 6 threads	18100	$1.00e-6$	10.55	9280	$1e-6$	4.66
N=120, 1 threads	40385	$1.00e-6$	365.64	20707	$9.99e-7$	170.93
N=120, 3 threads	40385	$1.00e-6$	133.81	20707	$9.99e-7$	60.28
N=120, 6 threads	40385	$1.00e-6$	70.02	20707	$9.99e-7$	31.36

As expected, Gauss-Seidel converges in less iterations and thus faster than Jacobi. For both methods with various N, as the number of available threads for the parallel sections increases, the runtime decreases drastically (for large N, the decrease appears to be by a factor close to $\frac{\text{1 thread}}{\text{n threads}}$, though not exact and confirming something like this would require more testing).