Report

Hayden LeBaron | CS4230 | 10/23/2020

Log results

NOTE: When I ran this code 2 days ago, problem 1 consistently reported 2.1 GFLOPs, Problem 2 consistently reported 2.7 GFLOPS, and Problem 3 consistently reported 6 GFLOPS. There must be a lot of error in the process because I ran it again on Friday (which is when the summary is from) and I am getting 1.71, 2.32, and 4.92 GFLOPS respectively. I ask that there is some lenience with the grading as I got this assignment finished early and then did not have time to make further optimizations on Friday to account for variation in CPU resources.

```
Host: u1081509@notch031
CPU: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
GCC ver: 9.2.0
Job ID: 1697289
Problem 1 Reference Version: Matrix Size = 4096; 0.24 GFLOPS; Time = 2.741 sec;
Problem 1 Optimized Version: Matrix Size = 4096; 1.71 GFLOPS; Time = 0.392 sec;
Checking correctness for y: Passed Correctness Check
Checking correctness for z: Passed Correctness Check
Problem 2 Reference Version: Matrix Size = 512; 0.34 GFLOPS; Time = 7.960 sec;
Problem 2 Optimized Version: Matrix Size = 512; 2.32 GFLOPS; Time = 1.158 sec;
Correctness Check Passed
Matrix Size = 1000
Problem 3 Reference Version: 0.22 GFLOPs; Time = 9.21
Problem 3 Optimized Version: 4.92 GFLOPs; Time = 0.41
Correctness Check Passed
```

Problem 1

The first optimization I attempted was loop tiling. I tried a variety of ways of tiling but I never broke past 0.7 GFLOPs with loop tiling.

Then I looked closer and noticed that the work done on arrays `y` and `z` were done totally independently. This signaled to me that they could be broken into two separate loops. So I peeled the loops, like so:

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      y[j] = y[j] + m[i][j] * x[i];

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      z[j] = z[j] + m[j][i] * x[i];
```

Now that the loops were peeled, I could permute the order of the loops for the second pair of nested loops so that the 2D array access pattern could match the loop index order. So now I had

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    y[j] = y[j] + m[i][j] * x[i];

for (j = 0; j < n; j++)
  for (i = 0; i < n; i++)
    z[j] = z[j] + m[j][i] * x[i];
```

Now the arrays were being accessed in a way that would leverage spatial locality due to the way caching works. The performance is now at 2.1 GFLOPS.

Problem 2

For problem 2 we started with this code:

```
int i, j, k;
double sum;
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++) {
    sum = 0.0;
    for (j = 0; j < n; j++)
      sum += x[i][j][k] * x[i][j][k];
    y[i][k] = sum;
  }
```

We first can get rid of the sum variable, noticing that we can substitute `y[i][k]` for sum. This doesn't yet improve performance. We now have:

```
int i, j, k;
for (i = 0; i < n; i++) {
  for (k = 0; k < n; k++) {
    y[i][k] = 0.0;
    for (j = 0; j < n; j++) {
      y[i][k] += x[i][j][k] * x[i][j][k];
    }
  }
}
```

We were then able to factor out the initialization of `y[i][k]` to 0.0 (for all i and k) to another loop. We still have not yet improved performance, but we're almost there!

```
int i, j, k;
for (i=0; i < n; i++)
  for (k=0; k < n; k++)
    y[i][k] = 0.0;
for (i = 0; i < n; i++) {
  for (k = 0; k < n; k++) {
    for (j = 0; j < n; j++) {
      y[i][k] += x[i][j][k] * x[i][j][k];
    }
  }
}
```

Now that we have separated the work out into different loops, we can use the same optimization technique used in problem 1 and change the loop order so that the loop order corresponds with the array access patterns. This allows us to leverage spatial locality when caching. Now `y[i][k]` accesses are in i-k and i-j-k loops, and a `x[i][j][k]` access is in an `i-j-k` nested loop. This improves performance to 2.74 GFLOPS

```
int i, j, k;
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++)
    y[i][k] = 0.0;

for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      y[i][k] += x[i][j][k] * x[i][j][k];
    }
  }
}
```

Problem 3

We started with this code:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[j][i] = c[j][i] + a[k][j] * b[k][i];
```

But the array access pattern does not line up with loop index order. We thus are not taking advantage of spatial locality when caching. We have the following access patterns:

```
- c[j][i]
- a[k][j]
- b[k][i]
```

Thus, the optimal solution will have j before i, k before j, and k before i. This implies the pattern k-j-i. Sure enough, when we permute the loops we get performance at 6.06 GFLOPs:

```
for (k = 0; k < n; k++)
  for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
      c[j][i] = c[j][i] + a[k][j] * b[k][i];
```