Report of Experiment Optimization of Polynomial Evaluation

Name: Zekihan AZMAN
Number: 250201007

Index

Index **2**

Experiment **3**

Result Table **3**

Methods **4**

Direct Polynomial Evaluation 4

Horner’s Polynomial Evaluation 4

Mixed Polynomial Evaluation 5

Compiler Optimizations **6**

O1 6

O2 6

O3 and Ofast 6

Appendix A: Modified Code **7**

Horner’s method optimized 7

Poly optimized 8

Mixed 9

Mixed optimized 10

Appendix B: Data **11**

Experiment

I used 4-core 15 gb memory standart google cloud compute engine to run and test times of execution. Main method calls 1000000 times of every method with same array of constant values.

```
(double)clock() - time / CLOCKS_PER_SEC;
```

This method has been used to calculate time of execution of methods.

For bonus part as it is to lower get any significant difference in values i run it 1000 instead of 10 times to get precise results. As i didn't want to change printing method in c files.

Source code, output excel files for every flag and overall results have been included.

Result Table

index	mixed	mixed optimized	poly	poly optimized	horner	horner optimized
noflag	2.0413268100	1.5965996300	3.7652934400	2.3622142800	4.8621455700	3.4472851400
o1	0.0006087700	0.4298143100	0.0003805600	0.9776759900	0.0003787100	0.0003795200
o2	0.0000016213	0.0000009935	0.0000007853	0.0000007659	0.0000007317	0.0000008349
o3	0.0000016708	0.0000009492	0.0000007991	0.0000007865	0.0000007418	0.0000008233
ofast	0.0000016255	0.0000009625	0.0000007983	0.0000007679	0.0000007452	0.0000008081

Methods

Direct Polynomial Evaluation

In direct polynomial evaluation, the for loops result is not dependent of result variable. It means we can use concurrency or unrolling, pipelining to increase performance. At first i just duplicated the loops content with changed indexes. It made almost no significant performance boost. After that i separated odd and even degrees with two result and xpwr variables. This made significant performance boost.

Reason for that first method cannot take advantage of pipelining as it has to wait result of previous line but in the second one it uses pipelining as it has two result and xpwr variables. Then i realised i keep multiplying xpwr with x^2 so i added a second variable just holds x^2 . This added a little bit of performance boost.

In result original code has 3.76529344 second of execution time whereas optimized code has 2.36221428 second. It has %59 performance boost.

Horner's Polynomial Evaluation

In Horner's polynomial evaluation, the for loops result is dependent of previous loop. It means we can use unrolling but not pipelining or concurrency to increase performance. Thus i started unrolling first significant time difference was in 4 step of unrolling. But then i realised it is not stable as i can get better or worse. In average it made no boost to performance. Then i used a temp value to store results in the loop but that made it worse. So i extracted declaration of temp variable outside of the loop but it didnt get better than original once again. This time i removed temp variable, and fitted all 4 steps in one big line. This made significant performance boost to program.

Reason for that is i reduce the amount of assignment of variable result from 4 to 1.

In result original code has 4.86214557 second of execution time whereas optimized code has 3.44728514 second. It has %41 performance boost.

Mixed Polynomial Evaluation

Merging direct polynomial evealuatioınb and horner's method. First implemented this formula $a_0 + x * (a_1 + a_2 * x) + x^3 * (a_3 + a_4 * x) \ldots$

It was better than poly or horner's methods. In fact it is %84 better than poly and %138 better than horner's method.

As for it's optimization at first i tried what i have done to poly as its structure is similar, but it yielded little to none performance boost. I unrolled it 4 steps. Then reduced steps to one line similar to horner's optimization. This time it was a little bit better but not enough to show it. Then i changed a little bit of mathematics and used parentheses to extract same multiplications. Again it made it faster a little bit more. But i realised there is awful lots of x^2 multiplication so i extracted that to outside of the loop and that was the breaking point after that there has been significant time difference.

In result original code has 2.04132681 second of execution time whereas optimized code has 1.59659963 second. It has %28 performance boost.

Compiler Optimizations

O1

O1 makes 3000-12000 times faster the original code. But for mixed optimized and poly optimized it can only make them 2-3 times faster.

O2

O2 makes 1000000-7000000 times faster than no flag results for all methods. In my educated guess -frerun-cse-after-loop flag in O2 makes poly and mixed optimization that has been not optimized enough in O1. Re-run common subexpression elimination after loop optimizations are performed.

O3 and Ofast

At this point optimizations has no meaning as execution is already is in under 1/1000 of ms. They cannot make it faster than O2.

Appendix A: Modified Code

Horner’s method optimized

```
// Horner's polynomial evaluation with optimizations
double horner_poly_optimized(double a[], double x, int degree)
{
    long int i;
    double result = a[degree];
    for (i = degree - 1; i >= 4; i -= 4)
    {
        result = a[i - 3] +
                 (x * (a[i - 2] +
                   (x * (a[i - 1] +
                     (x * (a[i] +
                       (x * result)))))));
    }
    for (; i >= 0; i--)
    {
        result = a[i] + (x * result);
    }
    return result;
}
```

Poly optimized

```
// Direct polynomial evaluation with optimizations
double poly_optimized(double a[], double x, int degree)
{
    int i;
    int limit = degree - 1;
    double result1 = a[0];
    double result2 = 0;
    double xs = x * x;
    double xpwr1 = x;
    double xpwr2 = xs;
    for (i = 1; i <= limit; i += 2)
    {
        result1 += a[i] * xpwr1;
        xpwr1 *= xs;

        result2 += a[i + 1] * xpwr2;
        xpwr2 *= xs;
    }
    for (; i <= degree; i++)
    {
        result1 += a[i] * xpwr1;
        xpwr1 *= xs;
    }
    return result1 + result2;
}
```

Mixed

```
// Mixed polynomial evaluation
double mixed(double a[], double x, int degree)
{
    long int i;
    double result = a[0];
    double xpwr = x;
    for (i = 1; i <= degree - 2; i += 2)
    {
        result += xpwr * (a[i] + (a[i + 1] * x));
        xpwr *= x * x;
    }
    for (; i <= degree; i++)
    {
        result += xpwr * (a[i]);
        xpwr *= x;
    }
    return result;
}
```

Mixed optimized

```
// Mixed polynomial evaluation with optimizations
double mixed_optimized(double a[], double x, int degree)
{
    int i;
    int limit = degree - 1;
    double result = a[0];
    double xpwr = x;
    double xs = x * x;
    for (i = 1; i <= degree - 8; i += 8)
    {
        result += xpwr * (((a[i] + (a[i + 1] * x))) +
                    xs * (((a[i + 2] + (a[i + 3] * x))) +
                      xs * (((a[i + 4] + (a[i + 5] * x))) +
                        (xs * (a[i + 6] + (a[i + 7] * x))))));
        xpwr *= xs * xs * xs * xs;
    }
    for (; i <= degree; i++)
    {
        result += xpwr * (a[i]);
        xpwr *= x;
    }
    return result;
}
```

Appendix B: Data

index	mixed	mixed optimized	poly	poly optimized	horner	horner optimized
noflag	2,0413268100	1,5965996300	3,7652934400	2,3622142800	4,8621455700	3,4472851400
o1	0,0006087700	0,4298143100	0,0003805600	0,9776759900	0,0003787100	0,0003795200
o2	0,0000016213	0,0000009935	0,0000007853	0,0000007659	0,0000007317	0,0000008349
o3	0,0000016708	0,0000009492	0,0000007991	0,0000007865	0,0000007418	0,0000008233
ofast	0,0000016255	0,0000009625	0,0000007983	0,0000007679	0,0000007452	0,0000008081

index	mixed boost	poly boost	horner boost
noflag	28%	59%	41%
o1	-100%	-100%	0%
o2	63%	3%	-12%
o3	76%	2%	-10%
ofast	69%	4%	-8%