SLERP Experiment Report

In this report, I’m going to compare the speed between different versions of SLERP functions.

SLERP Functions

There are five versions of the function in total:

- `DirectX::XMQuaternionSlerp`
- `SlowMath::Quaternion::Slerp`
- `SlowMath::Quaternion::FastSlerp`
- `FastMath::Quaternion::Slerp`
- `FastMath::Quaternion::FastSlerp`

The first one is from DirectXMath, which is used for reference. The functions under namespace `SlowMath` is the regular version, and the functions under namespace `FastMath` is the intrinsic version with SIMD and SSE instructions.

Both `SlowMath::Quaternion` and `FastMath::Quaternion` has implement the same set of helper functions including overloaded operators, normalizations, dot product and so on. The `FastMath::Quaternion` also has overloaded `new`, `delete`, `new[]` and `delete[]` operators to make sure objects created on heap memory are also memory alligned.

The `Slerp` functions are the general function that call overloaded operators and helper functions to calculate. The `FastSlerp`, however, don't call those functions at all to prevent redundant memory copy. I'll call it the inline version. The `FastMath::Quaternion::Slerp` and `FastMath::Quaternion::FastSlerp` look like this.

```
FastMath::Quaternion FastMath::Quaternion::Slerp(Quaternion v1, Quaternion v2,
float t)
{
        float dot = Dot(v1, v2);

        if (dot < 0.0f)
        {
                v2 = -v2;
                dot = -dot;
        }

        const float omega = acosf(dot);
        const float theta = omega * t;
        const float sinTheta = sinf(theta);
        const float sinOmega = sinf(omega);

        const float s0 = cosf(theta) - dot * sinTheta / sinOmega;
        const float s1 = sinTheta / sinOmega;

        return s0 * v1 + s1 * v2;
}
```

```
FastMath::Quaternion FastMath::Quaternion::FastSlerp(Quaternion v1, Quaternion v2,
float t)
{
	const __m128 sq = _mm_mul_ps(v1.v_, v2.v_);
	__m128 d = _mm_add_ps(sq, _mm_shuffle_ps(sq, sq, _MM_SHUFFLE(0, 3, 2,
1)));
	d = _mm_add_ps(d, _mm_shuffle_ps(sq, sq, _MM_SHUFFLE(1, 0, 3, 2)));
	d = _mm_add_ps(d, _mm_shuffle_ps(sq, sq, _MM_SHUFFLE(2, 1, 0, 3)));
	float dot;
	_mm_store_ss(&dot, d);

	if (dot < 0.0f)
	{
		const __m128 s = _mm_set_ps1(-1);
		v2.v_ = _mm_mul_ps(v2.v_, s);
		dot = -dot;
	}

	const float omega = acosf(dot);
	const float theta = omega * t;
	const float sinTheta = sinf(theta);
	const float sinOmega = sinf(omega);

	const float s0 = cosf(theta) - dot * sinTheta / sinOmega;
	const float s1 = sinTheta / sinOmega;

	const __m128 m0 = _mm_set_ps1(s0);
	const __m128 m1 = _mm_set_ps1(s1);

	Quaternion result;
	result.v_ = _mm_add_ps(_mm_mul_ps(m0, v1.v_), _mm_mul_ps(m1, v2.v_));
	return result;
}
```

Predicted result

The following result is predicted:

- The DirectXMath should be the fastest since it's an industrial level library;
- The intrinsic versions should be 2~4 times faster than the common version;
- The inline versions should be either slightly or much faster than the non-inline versions.

Experiment Platform and Compilers

The related specification of the experiment platform is listed below.

- CPU: Intel i7-8086K (Supports Intel SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2)
- Memory: 32GB
- OS: Windows 10 Pro 18841

The information of the compiler used in the experiment is listed below.

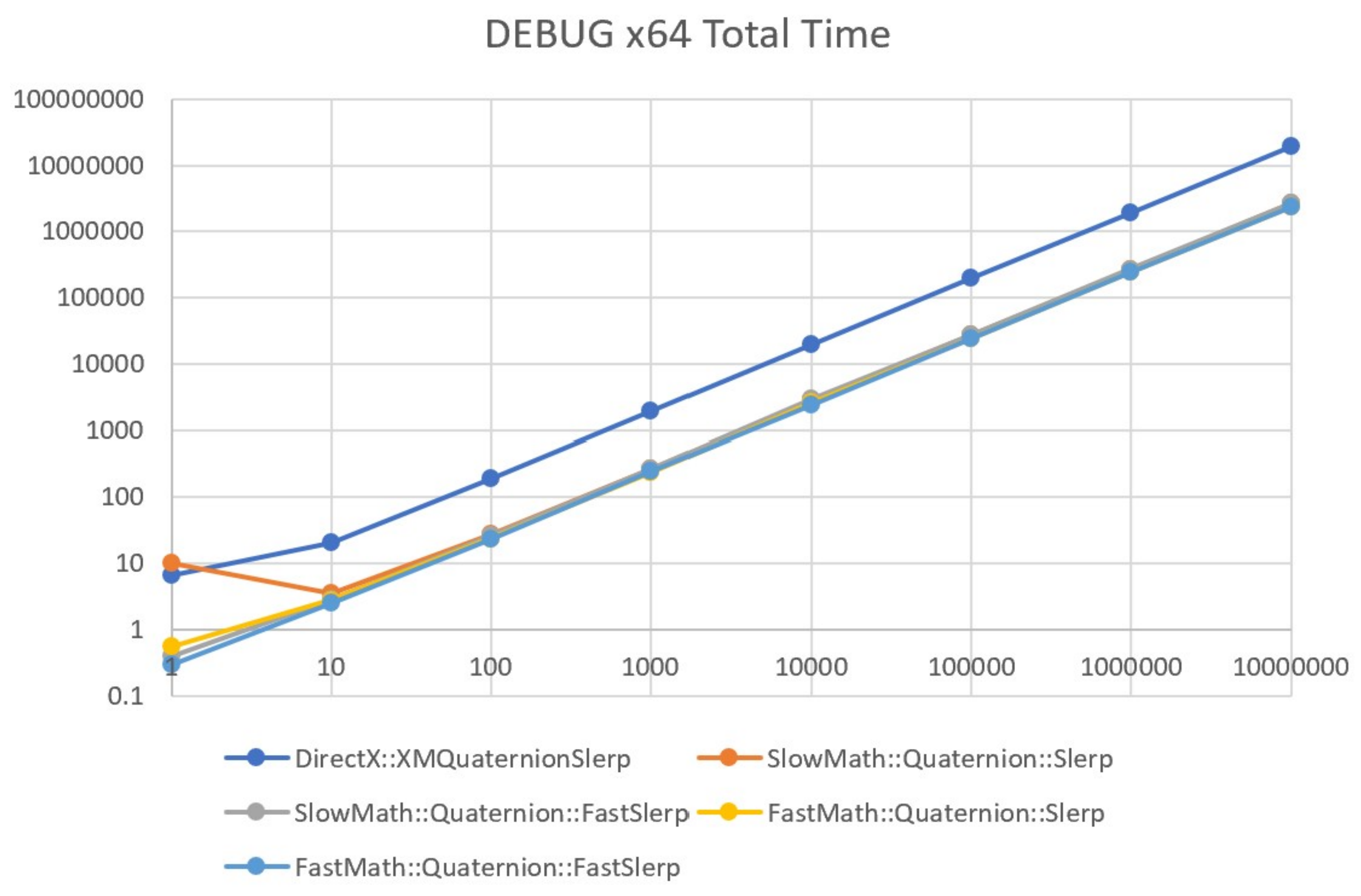
- Compiler: MSVC CL 19.20.27323
- IDE: Visual Studio 2019 Preview 3
- Debug Configuration:
    - Optimization: Disabled (`\Od`)
    - Enable Intrinsic Functions: No
    - Favor Size or Speed: Neither
- Release Configuration:
    - Optimization: Maximum Optimization (Favor Speed) (`/O2`)
    - Enable Intrinsic Functions: Yes (`/Oi`)
    - Favor Size or Speed: Favor fast code (`/Ot`)

Experiment Result

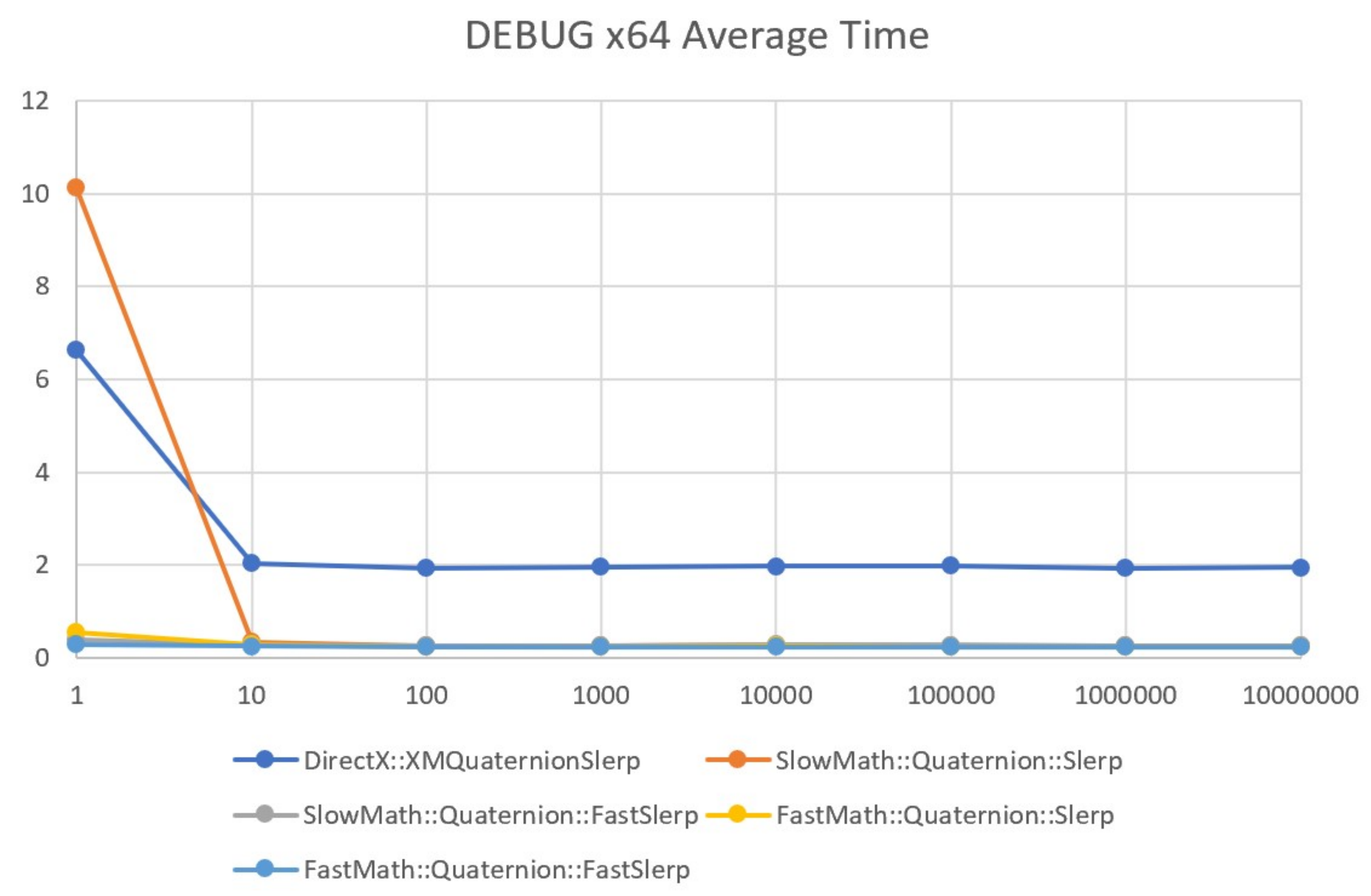
Five experiments run under each configuration. In each experiment, the time for 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000 times of SLERP calculation is recorded and the average of five experiments is used for final analyzation.

Debug Configuration (With no optimization)

The following chart shows the total time used in the debug experiment.



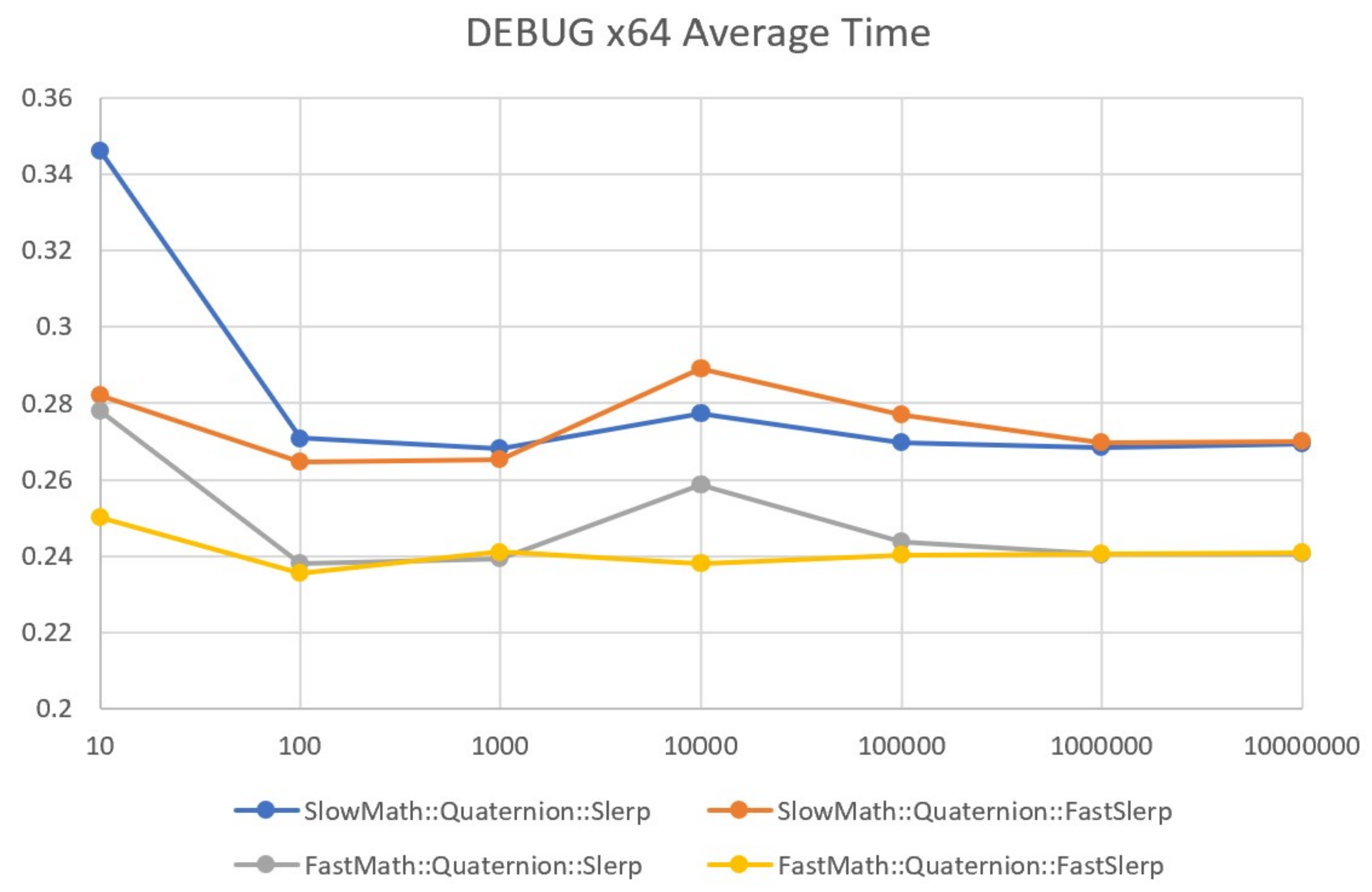
The following chart shows the average time used in the debug experiment.



The unit of y-axis is us.

We can see that, unlike the prediction, the DirectXMath is always the slowest, and it is slower for a constant time, about 2us per calculation. This might caused by some safeguard or other things. According to the Microsoft Article, this might also because the Floating Point Model (`/fp`) is not set to FAST (`/fp:fast`). However, when I changed the option, nothing significant happens.

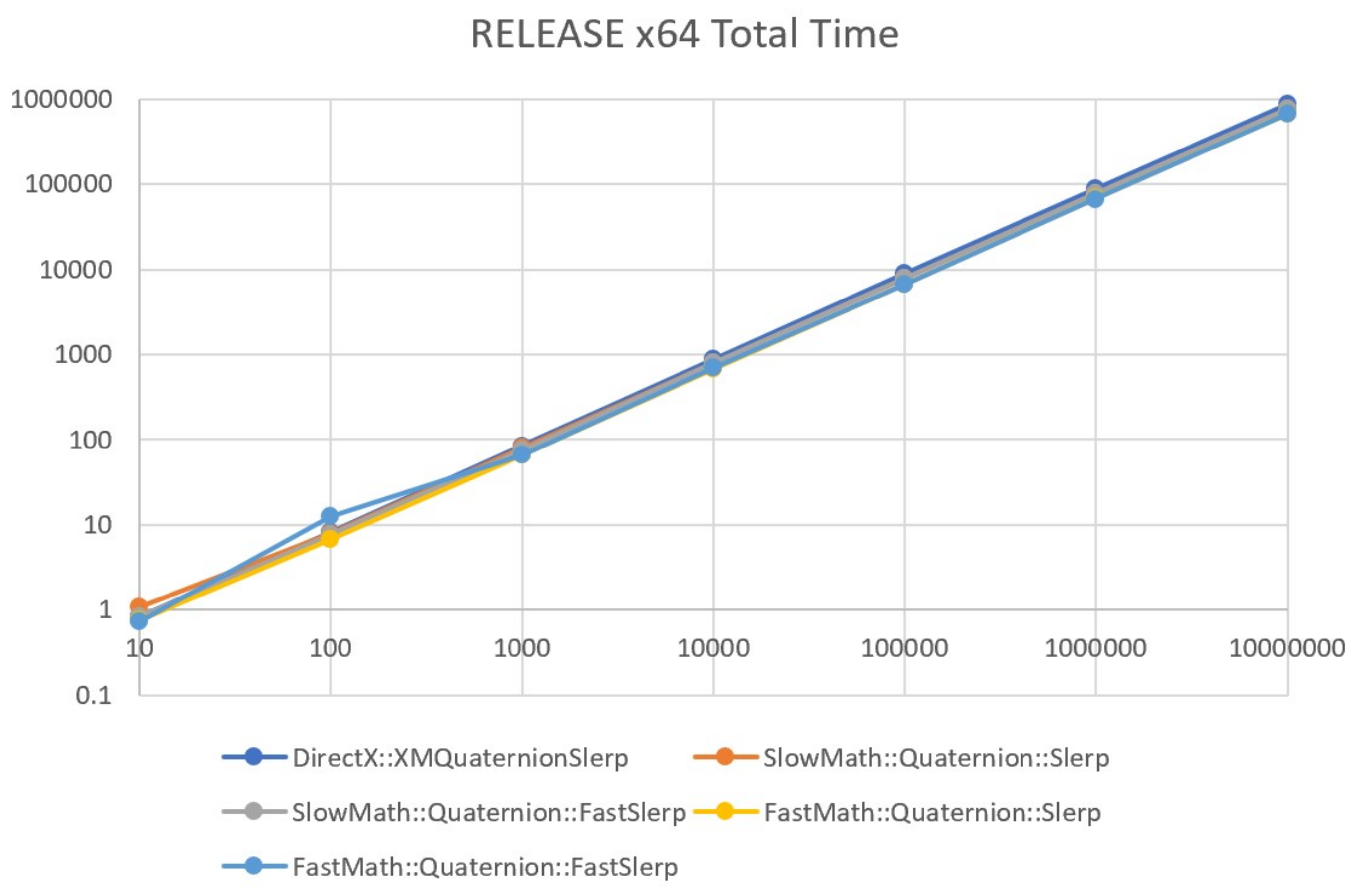
To see the performance of the other four functions, I removed the first data point which is apperantly influenced by some other factors and the whole DirectXMath series. The chart is shown below.

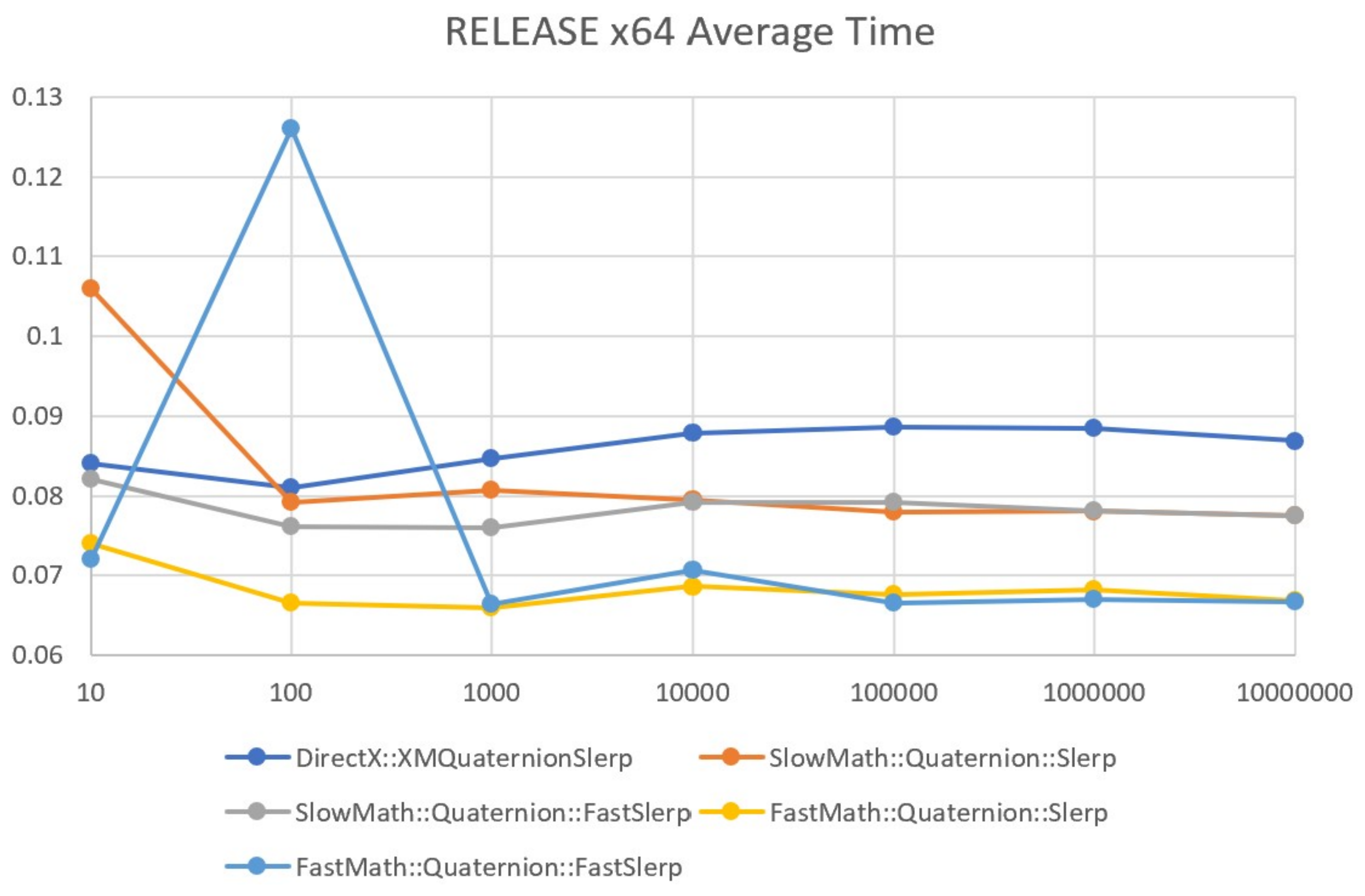


From cthe chart, we can see that, the intrinsic versions are indeed faster than the regular version, but the speed gain is not obvious. The predicted time would be 2~4 times faster, but in fact is only around 0.03us faster. And the inline versions are not necessary faster, sometimes they are even slower.

Release Configuration (With fastest optimization)

There are two charts for release configuration, since the DirectXMath is quite fast this time. I also removed the first data point since it's still not very correct.





We can see that, compared with debug configuration, the compiler optimized a lot of things this time. The DirectXMath is a lot faster than it was, and the speed is very close to other functions, but it is still generally the slowest. The inline versions still don't give us a speed gain this time, and the result is even closer since I guess the redundant memory copy is optimized by the compiler. The intrinsic version is still faster, but not as predicted.

Conclusion

From the result, we can indeed see the speed difference between the intrinsic version and the non-intrinsic version, but it's not as fast as we thought. The assumption was that the memory copy took a long time, but the inline versions are not faster either. I could only assume that there are other factors that slow down the intrinsic calculation.