고속 크로스 플랫폼 SIMD 벡터 라이브러리 설계

(Designing Fast Cross-Platform SIMD Vector Libraries)

구스타브 올리베이라(Gustavo Oliveira)

가마수트라 등록일(2010. 1. 20.)

http://www.gamasutra.com/view/feature/4248/designing_fast_crossplatform_simd_.php

도입

대부분의 3D 애플리케이션 내부에는 벡터 산술, 논리 회로, 비교, 교차 프로덕트 등 과 같은 루틴 계산을 수행하기 위한 벡터 라이브러리가 있다. 이런 형태의 라이브러리를 설계하는 데에는 수없이 많은 방법이 있지만, 개발자들은 종종 벡터 라이브러리가 가능한 가장 빠른 방법으로 이러한 계산을 수행할 수 있는 핵심 요소들을 간과한다.

2004 년 말경, “벡터 수학”을 의미하는 VMath 로 명명된 벡터 라이브러리 코드 개발직에 임명되었다. VMath 의 주요 목적은 가장 빠르고 다른 플랫폼을 쉽게 가로지르는 휴대용이었다.

2009 년 놀랍게도, 컴파일러 기술이 크게 변하지는 않았다. 실제로, 이 시기에 나의 연구 결인 본 기사에서 제시한 결과는 약간의 예외는 있지만, 내가 5 년 전에 Vmath 를 작업했던 때와 거의 동일하다.

“가장 빠른” 라이브러리

이 기사는 대부분 C++로 씌어져 주로 실행에 초점이 맞춰져 있기 때문에, 때로 가장 빠른 라이브러리를 규정하는 오류를 범할 수 있다.

따라서, 여기에서 설명하고 있는 가장 빠른 라이브러리는 동일한 설정을 이용하여 동일한 코드를 컴파일할 때 다른 라이브러리와 비교할 경우 가장 작은 어셈블리 코드를 생성시키는 것이다 (과정의 해제 모드 가정). 이것은 가장 빠른 라이브러리가 동일하게 정확한 계산을 실행하기 위한

더 적은 도구를 만들기 때문이다. 즉, 다르게 표현하자면, 가장 빠른 라이브러리는 최소한으로 코드를 팽창시키는 것이다.

SIMD 명령

현대의 프로세서 주위의 단일 명령 다중 데이터 처리 (SIMD)의 광범위한 전개로 벡터 라이브러리 개발의 업무는 더욱 쉬워지게 되었다. SIMD 연산은 정확하게 FPU 등록기의 FPU 연산처럼 SIMD 레지스터를 작동시킨다. 하지만, 장점은 SIMD 레지스터는 통상 쿼드 워드, 즉, 각각 32 비트짜리 4 개의 “floats”나 “ints”는 128 비트 너비이다. 이는 개발자들이 단일 명령으로 4D 벡터 계산의 실행을 가능하게 한다. 이것 때문에, 벡터 라이브러리가 가질 수 있는 최상의 특징은 그 안에서 SIMD 명령을 이용하는 것이다.
그럼에도 불구하고, SIMD 명령을 가지고 수행할 때, 여러분은 라이브러리가 코드를 팽창시킬 수 있는 일반적인 오류를 경계해야 한다. 사실, SIMD 벡터 라이브러리의 코드 팽창은 단순하게 FPU 명령을 사용하는 것이 더 나을 수 있다는 점에서 철저할 수 있다.

벡터 라이브러리 인터페이스

벡터 라이브러리의 고수준 인터페이스를 설계할 때 SIMD 명령에 대하여 가장 좋은 방법으로 말할 수 있는 것은 내인자의 활용에 의한 것이다. 이들은 SIMD 명령을 가지고 프로세서를 목표로하는 대부분의 컴파일러로부터 이용 가능하다. 또한, 각 내인자는 단일 SIMD 명령으로 번역한다. 하지만 직접적으로 어셈블리를 기록하는 대신 내인자를 이용하는 장점은 컴파일러가 일정계획과 표식 최적화의 실행을 가능하게 하는 것이다. 이는 코드 팽창을 상당하게 최소화시킬 수 있다.
아래는 내인자의 예이다:

```
Intel & AMD:

vr = _mm_add_ps(va, vb);

Cell Processor (SPU):

vr = spu_add(va, vb);

Altivec:

vr = vec_add(va, vb);
```

1. 값에 의한 결과 복귀

내인자 인터페이스의 관찰을 통하여, 벡터 라이브러리는 실행을 극대화할 수 있는 인터페이스를 모방해야 한다. 따라서, 여러분은 다음과 같이 참조가 아니라 값에 의하여 결과를 복귀시켜야 한다:

```
//correct
inline Vec4 VAdd(Vec4 va, Vec4 vb)
{
   return(_mm_add_ps(va, vb));
};
```

한편으로 참조에 의하여 데이터를 복귀시킨다면, 인터페이스는 코드 팽창을 생성시킬 것이다. 아래는 부정확한 버전이다:

```
//incorrect (code bloat!)
inline void VAddSlow(Vec4& vr, Vec4 va, Vec4 vb)
{
   vr = _mm_add_ps(va, vb);
};
```

여러분이 값에 의하여 데이터를 복귀시켜야 하는 이유는 쿼드 워드 (128 비트)는 하나의 SIMD 레지스터 내부에 잘 맞기 때문이다. 그리고, 벡터 라이브러리의 주요 요소 중 하나는 가능한 많이 이들 레지스터내부에 데이터를 유지하는 것이다. 그렇게 함으로써, SIMD 레지스터로부터 메모리나 FPU 레지스터까지 불필요한 하중과 저장 연산을 피할 수 있다. 다중 벡터 연산을 결합시킬 때, “값에 의한 반환” 인터페이스는 FPU 나 메모리 이송에 대하여 SIMD 를 최소화시킴으로써 이 적재와 기억장치의 최적화를 가능하게 한다.

2. “순수” 선언 데이터

여기에서, “순수 데이터”는 단순한 “형정의”나 “정의”에 의하여 “등급”이나 “구조” 바깥에서 선언한 데이터로 정의한다. 내가 VMath 를 코딩하기 전에 여러 가지 벡터 라이브러리를 연구할 시기에, 내가 보았던 모든 라이브러리 중에서 한가지 공통 패턴을 보게 되었다. 모든 경우에, 개발자들은 아래와 같이 이것을 순수라고 선언하는 대신, “등급”이나 “구조” 내의 기본적인 쿼드 워드 형태를 완료하였다:

```
class Vec4
{
   ...
   private:
   __m128 xyzw;
};
```

이러한 데이터 요약 유형은 소프트웨어의 구성을 강건하게 만드는 C++ 개발자들 사이의 일반적은 관습이다. 자료는 단지 등급 인터페이스 기능에 의하여서만 방지하고 접근할 수 있다. 그러함에도 불구하고, 이러한 설계는 특히 몇몇 종류의 GCC 포트를 사용하고 있을 경우, 다른 플랫폼에서 많은 다양한 컴파일러에 의하여 코드 팽창을 불러 일으킨다.

컴파일러에게 아주 친숙한 접근법은 아래와 같이 벡터 데이터를 “순수”라고 선언하는 것이다:

```
typedef __m128 Vec4;
```

확실히 그 방법으로 설계한 벡터 라이브러리는 괜찮은 근본 데이터의 요약과 방지를 잃어버리게 될 것이다. 하지만, 결말은 확실히 알 수 있다. 이 문제를 명확히 하기 위하여 예를 들어 보기로 하자.

우리는 아래와 같이 머클로린 급수를 사용하여 사인 함수의 근사값을 구할 수 있다:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \text{ for all } x$$

(*) 방지 코드에서 사인 함수의 근사값을 구할 수 있는 더 낫고 빠른 방법들이 있다. 머클로린 급수는 여기에서 설명을 위한 목적으로 사용하였다.

만약 개발자가 위의 공식을 이용한 사인 함수의 벡터 버전을 코드화한다면, 코드는 다소간 아래와 같이 될 것이다:

```
Vec4 VSin(const Vec4& x)
{
   Vec4 c1 = VReplicate(-1.f/6.f);
   Vec4 c2 = VReplicate(1.f/120.f);
   Vec4 c3 = VReplicate(-1.f/5040.f);
   Vec4 c4 = VReplicate(1.f/362880);
   Vec4 c5 = VReplicate(-1.f/39916800);
   Vec4 c6 = VReplicate(1.f/6227020800);
   Vec4 c7 = VReplicate(-1.f/1307674368000);

   Vec4 res = x +
      c1*x*x*x +
      c2*x*x*x*x*x +
      c3*x*x*x*x*x*x*x +
      c4*x*x*x*x*x*x*x*x*x +
      c5*x*x*x*x*x*x*x*x*x*x*x +
      c6*x*x*x*x*x*x*x*x*x*x*x*x*x +
      c7*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x;

   return (res);
}
```

이제 좌측란에는 “순수”로 우측란에는 부류내에서 선언한 Vec4 로 컴파일 된 동일한 함수의 어셈블러를 보기로 하자. (표 문서를 다운 받으시려면 여기를 클릭하라. 표 1 을 참조하라.)
똑같이 정확한 코드는 단순히 기본적인 데이터를 선언한 방법을 변경하여 약 15% 정도의 요소만큼 축소된다. 만약 이 기능이 약간의 내부 루프 계산을 실행하고 있었다면, 코드 크기를 절약하고 확실히 더 빠르게 실행되었을 것이다.

4. 과부하 연산자 vs. 절차상 인터페이스

명확성으로 가장 현대적인 애플리케이션에서 인기가 있었던 또 다른 괜찮은 C++ 특징은 과부하 연산자의 사용이다. 하지만, 일반적인 규칙과 같이 과부하 연산자는 코드를 팽창시킨다. 빠른 SIMD 벡터 라이브러리는 또한 절차적 C 모양의 인터페이스를 제공해야 한다.
간단한 수학식을 처리할 때, 과부하 연산자로부터 발생한 코드 팽창을 일어나지 않는다. 아마도 이것 때문에 벡터 라이브러리를 처음 설계할 때 대부분의 개발자들이 이 문제를 무시하는 것 같다. 하지만, 식이 복잡해 질수록, 최적화를 위한 프로그램은 정확한 결과를 보증하기 위한 여분의 작업을 해야 한다. 이것은 SIMD 레지스터와 기억장치 사이에서 보통 더 많은 로드/기억 저장 연산으로 번역하는 불필요한 일시적인 기억 변수들을 발생시키는 것과 관련 있다.
예로써, 과부하 연산자와 절차 인터페이스를 이용한 3-밴드 이퀄라이저를 컴파일 해 보기로 하자. 그 다음 우리는 생성된 어셈블리 코드를 본다. 과부하 연산자로 쓴 소스 코드는 아래와 같다:

```
Vec4 do_3band(EQSTATE* es, Vec4& sample)
{
    Vec4 l,m,h;

    es->f1p0 += (es->lf * (sample - es->f1p0)) + vsa;
    es->f1p1 += (es->lf * (es->f1p0 - es->f1p1));
    es->f1p2 += (es->lf * (es->f1p1 - es->f1p2));
    es->f1p3 += (es->lf * (es->f1p2 - es->f1p3));
    l = es->f1p3;

    es->f2p0 += (es->hf * (sample - es->f2p0)) + vsa;
    es->f2p1 += (es->hf * (es->f2p0 - es->f2p1));
    es->f2p2 += (es->hf * (es->f2p1 - es->f2p2));
    es->f2p3 += (es->hf * (es->f2p2 - es->f2p3));
    h = es->sdm3 - es->f2p3;
    m = es->sdm3 - (h + l);

    l *= es->lg;
    m *= es->mg;
    h *= es->hg;

    es->sdm3 = es->sdm2;
    es->sdm2 = es->sdm1;
    es->sdm1 = sample;
    return(l + m + h);
}
```

다음은 절차 인터페이스를 이용하여 다시 쓴 동일한 코드이다:

```
Vec4 do_3band(EQSTATE* es, Vec4& sample)
{
    Vec4 l,m,h;

    es->f1p0 = VAdd(es->f1p0, VAdd(VMul(es->lf, VSub(sample, es->f1p0)), vsa));
    es->f1p1 = VAdd(es->f1p1, VMul(es->lf, VSub(es->f1p0, es->f1p1)));
    es->f1p2 = VAdd(es->f1p2, VMul(es->lf, VSub(es->f1p1, es->f1p2)));
    es->f1p3 = VAdd(es->f1p3, VMul(es->lf, VSub(es->f1p2, es->f1p3)));
    l = es->f1p3;

    es->f2p0 = VAdd(es->f2p0, VAdd(VMul(es->hf, VSub(sample, es->f2p0)), vsa));
    es->f2p1 = VAdd(es->f2p1, VMul(es->hf, VSub(es->f2p0, es->f2p1)));
    es->f2p2 = VAdd(es->f2p2, VMul(es->hf, VSub(es->f2p1, es->f2p2)));
    es->f2p3 = VAdd(es->f2p3, VMul(es->hf, VSub(es->f2p2, es->f2p3)));
    h = VSub(es->sdm3, es->f2p3);
    m = VSub(es->sdm3, VAdd(h, l));

    l = VMul(l, es->lg);
    m = VMul(m, es->mg);
    h = VMul(h, es->hg);

    es->sdm3 = es->sdm2;
    es->sdm2 = es->sdm1;
    es->sdm1 = sample;

    return(VAdd(l, VAdd(m, h)));
}
```

마지막으로 양쪽의 어셈블리 코드를 보기로 한다 (표 문서를 다운받으려면 여기를 클릭하라; 표 2 를 참조하라.)

절차상 호출을 사용했던 코드는 대략 21% 가량 더 작았다. 또한 이 코드는 동시에 4 가지의 오디오의 흐름을 균등하게 하고 있음을 알아라. 따라서, FPU 상에서 동일한 계산을 하는 것과 비교한다면, 속도의 상승은 엄청날 것이다.

나는 벡터 라이브러리에서 과부하 연산자를 위한 지원에 훼방 놓지 않기를 강조한다. 실제로, 나는 개발자들이 식이 코드 팽창을 발생시킬 만큼 충분히 복잡한가를 쓸 수 있도록 둘 다 제공할 것을 조장하였다.

실제로 내가 VMath 를 시험하고 있을 때, 나는 종종 과부하 연산자를 이용한 나의 첫번째 수학식을 기록하였다. 그 다음 나는 어떠한 차이가 나타났는가를 알아보기 위하여 절차 호출을 이용한 동일한 코드를 시험하였다. 코드 팽창은 앞서 언급한 것처럼 식의 복잡성에 의존하였다.

5. 모든 인라인

인라인 키워드는 컴파일러가 값비싼 함수 요청의 제거를 가능하게 하여 코드를 더 큰 코드 팽창의 가격에서 최적화 시킨다. 함수 요청을 수행하지 못하도록 모든 벡터 함수를 인라인하는 것은 일반적인 관습이다.

윈도우 컴파일러들은 (마이크로소프트 & 인텔) 함수들을 언제 인라인 할 지 안 할지에 관하여 매우 인상적이다. 컴파일러에게 이 작업을 맡기는 것이 최상이므로, 여러분의 모든 벡터 함수에 인라인을 추가함으로써 잃을 것은 없다.
하지만, 인라인을 가지고 있는 한 가지 문제점은 명령 캐시가 (I-Cache)를 놓치는 것이다. 만약 컴파일러가 목표 플랫폼에 대하여 "너무 큰" 때를 알기에 충분하지 않다면, 여러분은 큰 문제에 빠질 수 있다. 코드는 팽창하며 또한 수 많은 I-Cache 의 손실이 있을 것이다. 다음에 벡터 함수들은 더 느려질 수 있다.
나는 I-Cache 가 영향을 받았던 점까지 코드 팽창을 야기시켰던 "인라인 된" 벡터 함수를 가지고 있던 윈도우 개발에서 하나의 경우를 따라잡을 수 없다. 사실, 나는 함수들을 인라인하지 않을 때라도, 윈도우 컴파일러들은 나를 위하여 그것들을 인라인할 만큼 훌륭하다. 하지만, 내가 PSP 와 PS3 개발을 했을 때, GCC 포트는 그렇게 좋지 않았다. 사실 인라인하지 않는 것이 더 나은 경우도 있었다.
하지만, 만약 여러분이 이런 특수한 경우를 만나게 된다면, 다음과 같이 "인라인"되지 않은 플랫폼 특이적인 요청에 대하여 여러분의 인라인 기능을 감싸는 것은 항상 사소하다:

```
Vec4 DotPlatformSpecific(Vec4 va, Vec4 vb)
{
   return (Dot(va, vb));
}
```

한편, 만약 벡터 라이브러리가 인라인없이 설계된다면, 여러분은 쉽게 그들을 인라인할 수는 없다. 따라서 여러분은 자신을 위하여 함수들을 인라인할 만큼 훌륭한 컴파일러에 의존한다. 이것은 여러분이 다른 플랫폼에서 예상한대로 작동하지 않을 수 있다.

6. SIMD 레지스터로의 복제 결과 및 데이터 액세서 제공.

SIMD 명령으로 작업할 때, 하나의 계산과 네 개의 계산을 더하는 데에는 똑 같은 시간이 걸린다. 하지만 만약 여러분이 4 개의 레지스터의 결과가 필요하지 않다면 어떻게 할 것인가? 만약 실행이 동일하다면, SIMD 레지스터로 결과를 복제하는 것은 여전히 여러분에게 이익이 된다. SIMD 쿼드 워드로 결과를 복제함으로써, 이것은 컴파일러를 벡터 식으로 최적화시키는데 도움을 줄 수 있다.
똑 같은 이유로, GetX, GetY, GetZ 와 GetZ 와 같은 데이터 액세서들을 제공하는 것은 중요하다. 이러한 유형의 인터페이스를 제공하여 개발자가 그것을 사용한다는 것을 가정함으로써, 벡터 라이브러리는 SIMD 레지스터와 FPU 레지스터 사이의 비싼 캐스팅 연산을 최소화할 수 있다.
곧 나는 이 문제를 설명하고 있는 고전적 예에 관하여 논의할 것이다.

SIMD 벡터 라이브러리를 위한 좋은 프로그래밍 실행

잘 설계한 벡터 라이브러리를 이용할 수 있더라도, 특히 SIMD 명령으로 작업할 때, 여러분은 그것의 사용 방법에 관하여 주의를 기울여야 한다. 최상의 코드를 생성하는 컴파일러에 전적으로 의존하는 것은 좋지 않은 생각이다.

1. 컴파일러에 친숙하게 식 쓰기

여러분이 그것을 코드하는 것을 기록하는 방법에 의존하는 것은 최종적으로 결집된 코드에 아주 영향을 미칠 수 있다. 강건한 컴파일러를 사용하더라도, 차이는 알 수 없을 것이다. 사인의 예로 돌아가서, 만약 그 함수를 더 작은 식을 사용하여 기록하였다면, 최종 코드는 더욱 더 효과적일 것이다. 다음과 같이 새로운 버전의 똑 같은 코드를 살펴보자:

```
Vec4 VSin2(const Vec4& x)
{
    Vec4 c1 = VReplicate(-1.f/6.f);
    Vec4 c2 = VReplicate(1.f/120.f);
    Vec4 c3 = VReplicate(-1.f/5040.f);
    Vec4 c4 = VReplicate(1.f/362880);
    Vec4 c5 = VReplicate(-1.f/39916800);
    Vec4 c6 = VReplicate(1.f/6227020800);
    Vec4 c7 = VReplicate(-1.f/1307674368000);

    Vec4 tmp0 = x;
    Vec4 x3 = x*x*x;
    Vec4 tmp1 = c1*x3;
    Vec4 res = tmp0 + tmp1;

    Vec4 x5 = x3*x*x;
    tmp0 = c2*x5;
    res = res + tmp0;

    Vec4 x7 = x5*x*x;
    tmp0 = c3*x7;
    res = res + tmp0;

    Vec4 x9 = x7*x*x;
    tmp0 = c4*x9;
    res = res + tmp0;

    Vec4 x11 = x9*x*x;
    tmp0 = c5*x11;
    res = res + tmp0;

    Vec4 x13 = x11*x*x;
    tmp0 = c6*x13;
    res = res + tmp0;

    Vec4 x15 = x13*x*x;
    tmp0 = c7*x15;
    res = res + tmp0;

    return (res);
}
```

이제 결과를 비교해 보자. (표 문서를 다운 받으시려면 여기를 클릭하라. 표 3 을 참조하라.) 코드는 컴파일러에게 좀 더 친숙한 방법에서 단순히 다시 쓰기를 함으로써 거의 40% 정도가 축소된다.

2. SIMD 레지스터로의 결과 유지

이전에 언급한 것처럼, SIMD 레지스터와 FPU 레지스터 사이의 배치 연산은 비쌀 수 있다. 이것이 발생했을 때에 관한 고전적 예는 다음과 같이 두 개의 벡터로부터 하나의 스칼라를 가져오는 "스칼라 곱"이다.

```
Dot (Va, Vb) = (Va.x * Vb.x) + (Va.y * Vb.y) + (Va.z * Vb.z) + (Va.w * Vb.w);
```

현재 스칼라 곱을 사용하는 싹둑 잘려지는 것을 보기로 하자:

```
Vec4& x2 = m_x[i2];

Vec4 delta = x2-x1;

float deltalength = Sqrt(Dot(delta,delta));

float diff = (deltalength-restlength)/deltalength;

x1 += delta*half*diff;

x2 -= delta*half*diff;
```

위의 코드를 조사함으로써, "델타 길이"는 벡터 "x1"과 벡터 "x2" 사이의 거리이다. 따라서 "점" 함수의 결과는 스칼라이다. 다음은 이 스칼라는 벡터 "x1"과 "x2"의 크기를 조정하기 위하여 코드의 나머지를 통하여 사용하고 변경하였다. 명백히 벡터에서 스칼라까지 혹은 그 반대로 진행하는 수 많은 배치 연산들이 있다. 컴파일러가 데이터를 SIMD 와 FPU 레지스터에서 SIMD 와 FPU 레지스터까지 이동할 코드를 생성할 필요가 있기 때문에 이것은 비싸다.
하지만, 만약 우리가 위의 "점" 함수가 결과를 SIMD 4-쿼드 워드와 "w" 구성요소 소거로 복사한다면, 실제로 아래와 같이 코드를 다시 쓰기에 아무런 차이가 없다:

```
Vec4& x2 = m_x[i2];

Vec4 delta = x2-x1;

Vec4 deltalength = Sqrt(Dot(delta,delta));

Vec4 diff = (deltalength-restlength)/deltalength;

x1 += delta*half*diff;

x2 -= delta*half*diff;
```

“델타 길이”는 현재 쿼드 워드로 복사된 동일한 결과를 가지고 있기 때문에, 비싼 배치 연산 등은 더 이상 필요하지 않다.

3. SIMD 연산에 친숙한 데이터의 재배치

가능할 때 마다, 여러분의 데이터를 재배치함으로써, 여러분은 벡터 라이브러리를 이용할 수 있다. 예를 들면, 오디오를 가지고 작업할 때, 여러분은 친숙한 SIMD 가 되도록 여러분의 데이터 흐름을 저장시킬 수 있다.

여러분이 다음과 같이 네 개의 다른 배열에 저장되어 있는 오디오 샘플의 4 개의 흐름을 가지고 있다고 하자:

베이스 오디오 샘플 배열:

B0	B1	B2	B3	B4	B5	B6	B7	Etc…

드럼 오디오 샘플 배열:

D0	D1	D2	D3	D4	D5	D6	D7	Etc…

기타 오디오 샘플 배열:

G0	G1	G2	G3	G4	G5	G6	G7	Etc…

트럼펫 오디오 샘플 배열:

T0	T1	T2	T3	T4	T5	T6	T7	Etc…

하지만, 여러분은 또한 이러한 오디오 채널을 인터리브 할 수 있으며, 다음과 같이 동일한 데이터를 저장할 수 있다:

풀 밴드 오디오 샘플 배열:

B0	D0	G0	T0	B1	D1	G1	T1	Etc…

장점은 현재 여러분이 여러분의 배열을 직접 SIMD 레지스터로 로드하여 동시에 모든 샘플 상에서 계산을 실행할 수 있다는 것이다. 단점은 만약 여러분이 그 데이터를 4 개의 유선형 배열을 필요로 하는 다른 시스템으로 보낼 필요가 있다면, 여러분은 그 자료를 원래 형태로 재편성할 것이다. 만약 벡터 라이브러리가 데이터에서 어려운 계산을 수행하지 않는다면 이것은 비싸고 강한 기억 장치가 될 수 있다.

XNA 수학 라이브러리 – 과연 이것이 더 빠른가?

마이크로소프트는 윈도우와 Xbox 360 에서 운영되는 XNA 수학을 가지고 있는 DirectX 를 이동시킨다. XNA 수학은 FPU 벡터 인터페이스(뒤쪽 호환성용)와 SIMD 인터페이스를 가지고 있는 벡터 라이브러리이다.

2004 년 말 경으로 되돌아가서 내가 XNA 수학 라이브러리에 접근하지 못하였다면 (만약 이것이 그 때 존재했다면), 나는 5 년 전에 XNA 수학이 VMath 에서 사용하였던 아주 똑같은 인터페이스를 현재 사용하고 있음을 알게 되어 기뻤다. 단지 차이는 목표 플랫폼이었으며, 이것은 나의 경우에 윈도우와 PS3, PSP, PS2 등 이었다.

XNA 수학은 여기에 묘사된 SIMD 벡터 라이브러리의 주요 특징을 이용하여 설계하였다. 이것은 값으로 결과를 복귀시키며, 벡터 데이터는 순수하게 선언되었으며, 이것은 과부하 연산자와 절차적 요청을 위한 지원을 가지고 있으며, 모든 벡터 함수들을 인라인 시키고 데이터 액세서 등을 제공한다.

두 명의 개발자들이 똑 같은 결과를 가져오면, 여러분은 “가장 신속한”에 아주 접근할 가능성이 있다. 그래서 만약 여러분이 벡터 라이브러리를 코딩하는 업무와 마주치게 된다면, 크로스-플랫폼 SIMD 벡터 라이브러리를 위한 여러분의 최상의 방법은 XNA 수학이나 VMath 의 단계를 따르는 것이다.

그것을 염두에 두고서 여러분은 XNA 수학이 더 빠를 수 있을까 이렇게 물을 수 있다. SIMD 명령과 접속하는 한, 나는 그렇게 믿지 않는다. 하지만, 가장 빠른 인터페이스를 가지는 장점은 위임되는 것은 순수하게 함수들의 실행이라는 것이다.

만약 누군가가 동일한 인터페이스를 가지고 좀 더 빠른 버전의 함수를 제안한다면, 이것은 단지 새로운 코드에 결합하는 문제이며, 이미 끝난 일이다. 하지만, 만약 여러분이 좋은 인터페이스를 SIMD 명령에 제공하지 않는 벡터 라이브러리에서 작업하고 있다면, 더 나은 실행을 제안하는 것은 여전히 여러분 뒤에 남겨져 있을 것이다.

XNA 수학 vs Vmath, 헤비급 싸움

여러분이 고수준의 코드에서 작업한다고 가정한다면, XNA 수학보다 더 빠르기란 아주 어렵다. 남겨진 선택은 다음과 같다: 만약 존재한다면, 잘 개정된 라이브러리에서 그 함수를 최적화하거나 결함을 발견하라. 하지만, 또한 앞서 언급한 것처럼, 여러분이 XNA 수학을 가지고 작업하지만 그 최대 잠재성까지 사용 방법을 모를지라도, 여러분은 코드를 팽창시켜 뒤에 남겨지게 할 수 있다.

그러한 도전을 하기 위하여, VMath 나 XNA 수학으로 컴파일 할 경우 나는 차이를 알아차릴 수 있을 알고리즘을 발견하였다. 나는 단지 사소한 계산과 관련한 알고리즘이 동일한 속도로 돌아갈

것이라는 것을 알고 있었다. 따라서, 알고리즘은 몇 가지 관련 수학적 복잡성을 가지고 있어야 했다.

그것을 염두에 두고서, 나의 최종 선택은 오디오 데이터와 옷감 시뮬레이터 상의 3-대역 이퀄라이저를 실행하는 DSP 코드를 이식하는 것이었다. 원래의 샘플 코드에 대한 URL 은 그들 각자의 작가들을 따라 참조에서 볼 수 있다.

이것은 포트와 다른 실행과 같이 느낄 수 있도록 원래 작가의 코드가 손상되지 않도록 최선을 다하여 시도하였다. 3-대역 이퀄라이저 코드는 직선이었지만, 나는 특별히 옷감 시뮬레이션을 위하여 여분이 거의 없는 선을 첨가해야 했다.

실행하는 동안, 나는 또한 VMath 나 XNA 수학의 주요 특징을 따르지 않는 샘플 코드인 일반 벡터 등급에 포함시켰다. 데이터가 하나의 등급 내에서 요약하고 절차 인터페이스를 제공하지 않는 표준 벡터 등급이다. 나는 또한 단지 실행 비교용으로 Vclass 에 코드를 이식하였다.

3-대역 이퀄라이저 사례 연구

이퀄라이저 알고리즘에 대하여 나는 명백히 규정을 어겼다. XNA Math 와 비교하자면, VMath 의 실행에 대하여 내가 최선을 다하지는 못했다. 정확하게 내가 한 것은 XNA Math 와 과부하 연산자를 이용하여 이퀄라이저 코드를 이식하는 것이었다. 그 다음 VMath 에 대하여, 나는 과부하 연산자 등이 코드를 팽창시킬 것을 알았기 때문에, 절차적 요청의 사용은 제외하고 똑같은 것을 하였다.

하지만, 나의 치트는 목적이 있으며, 그것은 여러분이 가장 빠른 벡터 라이브러리에서 작업하고 있더라도, 컴파일러가 코드를 생성하는 방법을 모른다면 여러분은 뒤쳐질 수 있음을 다시 보여주는 것이었다.

또한 나는 SIMD 에 편하도록 데이터를 준비하였다. 그래서 로드와 기억장치 등을 최소화하기 위하여 오디오 채널을 인터리브 하였다. 하지만, 채널을 작동시키기 위하여, 나는 유선형의 PCM 오디오 데이터의 4 개의 배열로써 데이터를 XAudio2 로 보내도록 재구성해야 했다. 그것은 많은 부하와 저장장치를 야기시켰다. 그 이유에 대하여, 나는 또한 오디오 샘플의 분명한 배열로 저장된 데이터의 다른 사본에 대하여 직접적으로 계산을 실행하는 FPU 버전을 포함시켰다. FPU 버전의 유일한 장점은 데이터 배열의 재조합이 아니었다.

VMath 와 XNA Math 에 의하여 생성되는 어셈블리 코드에 대하여 알아보려면 여기를 클릭하라 – 표 4 를 참조하라.

결과는 “과부하 연산자 vs. 절차적 인터페이스”의 논쟁 기간 동안 내가 보여 주었던 것과 매우 유사하다 (표 2). 아래의 수치는 실행 시간 통계를 가지고 있는 데모의 스크린샷을 보여준다.

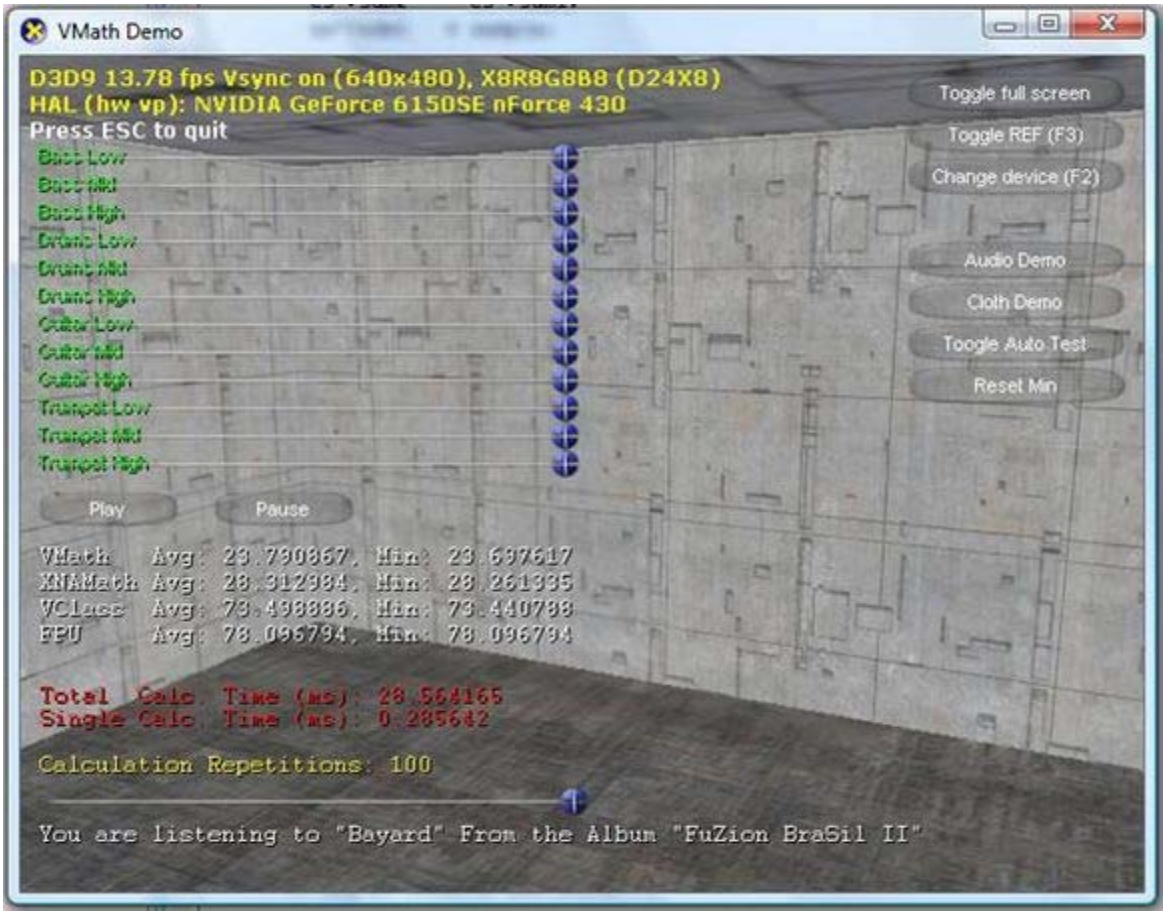

그림. 1 – 3 대역 이퀄라이저

최종 통계는 또한 명확하게 하기 위하여 아래의 표에 다시 쓰기를 하였다.

라이브러리	평균 시간
Vmath (절차상 요청)	23.79
XNA Math (과부하 연산자)	28.31
VClass	73.49
FPU (*)	78.09

참조로 VMath 를 사용하면, 단순히 다른 인터페이스 요청을 사용함으로써, XNA Math 보다 평균 약 16% 더 빠르게 진행할 수 있었다. XAudio2 에 전달된 오디오 데이터를 재구성하기 위한 대량 로드와 저장장치를 가지고, FPU 버전보다 3 배 이상 빠르게 진행할 수 있었다. 이것은 실제로 아주 인상적이다.

VClass 는 덮개 아래에서 생성되는 거대한 양의 코드 팽창 때문에 FPU 버전보다 단순히 5% 정도만 빨라졌다는 것에 주목하라.

헝겊 시뮬레이션 사례 연구

시뮬레이션 알고리즘은 대부분의 계산이 발생하는 2 개의 주요 루프를 가지고 있다. 하나는 아래에 나타나 있는 “Verlet” 적분 회로망이다:

```
void Cloth::Verlet()
{
    XMVECTOR d1 = XMVectorReplicate(0.99903f);
    XMVECTOR d2 = XMVectorReplicate(0.99899f);

    for(int i=0; i
    {
        XMVECTOR& x = m_x[i];
        XMVECTOR temp = x;
        XMVECTOR& oldx = m_oldx[i];
        XMVECTOR& a = m_a[i];
        x += (d1*x)-(d2*oldx)+a*fTimeStep*fTimeStep;
        oldx = temp;
    }
}
```

두 번째는 아래에 나와 있는 모든 속박에 대한 해결책을 찾기 위하여 가우스-자이델 반복 루프를 사용한 속박 해결사이다:

```
void Cloth::SatisfyConstraints()
{
    XMVECTOR half = XMVectorReplicate(0.5f);

    for(int j=0; j
    {
        m_x[0] = hook[0];
        m_x[cClothWidth-1] = hook[1];

        for(int i=0; i
        {
            XMVECTOR& x1 = m_x[i];

            for(int cc=0; cc
            {
                int i2 = cnstr[i].cIndex[cc];

                XMVECTOR& x2 = m_x[i2];
                XMVECTOR delta = x2-x1;
                XMVECTOR deltalength = XMVectorSqrt(XMVector4Dot(delta,delta));
                XMVECTOR diff = (deltalength-restlength)/deltalength;
                x1 += delta*half*diff;
                x2 -= delta*half*diff;
            }
        }
    }
}
```

코드를 검사하여, 절차적 요청에 대한 과부하 연산자를 변경하는 것은 충분히 이행되지 못할 것이다. 수학식은 최적화를 위한 프로그램이 똑같은 코드를 생성시키기에 충분할 만큼 간단하다. 나는 또한 “결과를 SIMD 레지스터내에서 유지”라고 설명되어 있는 원래의 코드로부터 유동 배치를 제거하였다. 이제 루프는 아주 단단하며 친 SIMD 이므로 남은 것은 무엇인가? 유일하게 남은 것은 수행할 수 있는 다른 것이 있는가 알아보기 위하여 XNA Math 내부를 보는 것이다. 그것이 있었던 것으로 판명되었다. 처음에 XMVector4Dot 를 자세히 살펴 보아서, 아래와 같이 수행된다:

```
XMFINLINE XMVECTOR XMVector4Dot(FXMVECTOR V1, FXMVECTOR V2)
{
    XMVECTOR vTemp2 = V2;
    XMVECTOR vTemp = _mm_mul_ps(V1,vTemp2);
    vTemp2 = _mm_shuffle_ps(vTemp2,vTemp,_MM_SHUFFLE(1,0,0,0));
    vTemp2 = _mm_add_ps(vTemp2,vTemp);
    vTemp = _mm_shuffle_ps(vTemp,vTemp2,_MM_SHUFFLE(0,3,0,0));
    vTemp = _mm_add_ps(vTemp,vTemp2);
    return _mm_shuffle_ps(vTemp,vTemp,_MM_SHUFFLE(2,2,2,2));
}
```

실행은 1 개의 승법, 3 개의 셔플과 2 개의 가산으로 구성되어 있다.
따라서, 나는 착수하여 동일한 결과지만 아래와 같이 하나의 적은 셔플 명령을 가지고 생산하는 다른 SSE2 4D 점을 기록하였다:

```
inline Vec4 Dot(Vec4 va, Vec4 vb)
{
   Vec4 t0 = _mm_mul_ps(va, vb);
   Vec4 t1 = _mm_shuffle_ps(t0, t0, _MM_SHUFFLE(1,0,3,2));
   Vec4 t2 = _mm_add_ps(t0, t1);
   Vec4 t3 = _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(2,3,0,1));
   Vec4 dot = _mm_add_ps(t3, t2);
   return (dot);
}
```

불행하게도, 놀랍게, 이것은 명령 사이에 더 많은 상호 의존을 가지고 있기 때문에, 나의 새로운 4D Dot 는 많은 차이를 만들지 못하였다.
나는 더 많이 보았기 때문에, XNA Math 에서 하나의 헷갈리는 사항을 발견하였다. 이것은 분할을 실행시키기 위하여 상호 함수를 요청한다. 사실 아래와 같이 과부하 연산자로부터 다중 요청에 의하여 연결된 함수는 :

```
XMFINLINE XMVECTOR XMVectorReciprocal(FXMVECTOR V)
{
   return _mm_div_ps(g_XMOne,V);
}

XMFINLINE XMVECTOR operator/ (FXMVECTOR V1,FXMVECTOR V2)
{
   XMVECTOR InvV = XMVectorReciprocal(V2);
   return XMVectorMultiply(V1, InvV);
}
```

유일한 문제는 이것이 “g_XMOne”를 로드한 다음 분할을 실행하기 위하여 내인성을 요청한다. 나는 마이크로소프트에서 왜 이 방법을 실행하는지 명확하게 알지 못하지만, 단순히 직접적으로 분할을 요청하는 것이 더 나을 수 있다. 따라서 VMath 에 대하여 나는 아래와 같이 여분의 로드가 없는 것을 의미하는 이러한 방법을 실행하였다:

```
inline Vec4 VDiv(Vec4 va, Vec4 vb)
{
   return(_mm_div_ps(va, vb));
};
```

이제 VMath 에서 실행한 작은 최적화를 가지고 두 개의 라이브러리를 이용한 “Verlet”와 “제한 해결사”의 어셈블러를 살펴보기로 한다.

표 문서를 다운 받으려면 여기를 클릭하라 – 표 5 와 표 6 을 참조하라.

내가 VMath 에 대해서는 절차적 요청을 XNA Math 에 대해서는 과부하 연산자를 이용하여 “Verlet” 함수를 기록하였지만, 아무런 차이는 없었다. 나는 수학식이 코드 팽창을 야기시키기에 충분할 만큼 복잡하지 않았을 것으로 기대한다. 하지만, “제한 해결사”는 더 빠른 분할로 인하여 정확하게 하나의 명령을 더 작게 하였다. 나는 여분의 movaps 명령의 원인이 되는 표에서 여분의 “g_XMOne”를 강조하였다. 아래의 스크린샷은 결과를 나타낸다.

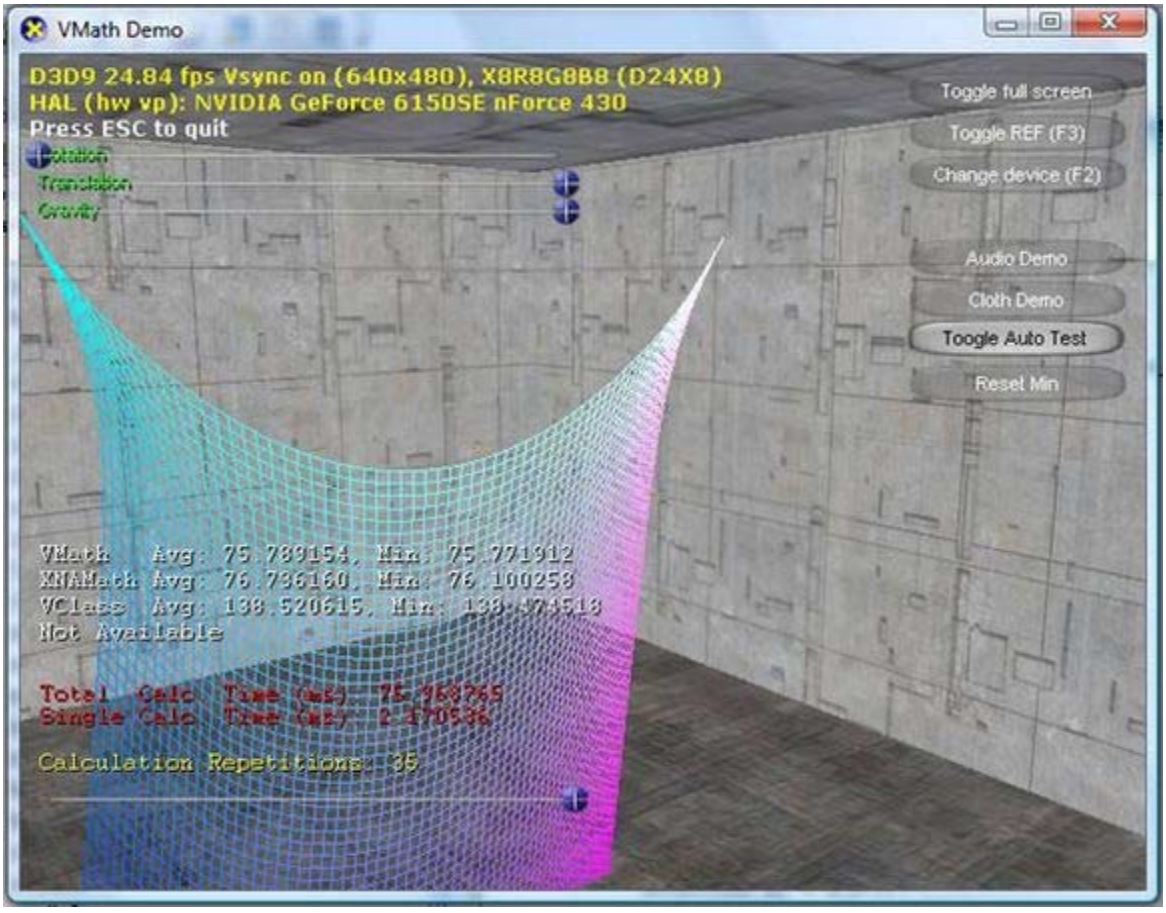

그림. 2 – 헝겊 시뮬레이션

라이브러리	평균 시간
VMath (절차적 요청)	75.78
XNA Math (과부하 연산자)	76.73
VClass	138.52
FPU	실행 안 함

기술적으로 이것은 한 명령 때문에 VMath 에 의한 녹아웃이었지만, 통계에서는 단지 약 1% 더 빠른 유의한 속도 상승을 보여주기에는 충분하지 못하였다.
사실 결과는 너무 가까워 그들이 몇 개의 배경 업무를 수행하는 OS 와 같은 기타 요소들에 의존하여 떠돌아 다녔다. 이러한 종류의 결과로 인하여, 결론은 VMath 와 XNA Math 는 실제로 똑같았다는 것이다.
그럼에도 불구하고, 두 가지 라이브러리는 VClass 보다 약 50% 더 빠르게 진행할 수 있었다. 이것은 주요한 차이가 라이브러리가 동일한 내인자와 인터페이스로 접속하는 방법에 관한 사실을 계산하면서 여전히 인상적이다.

인텔 컴파일러, "마법"

비주얼 스튜디오를 수송하는 마이크로소프트 컴파일러는 지금까지 모든 테스트를 수행하였다. 이제 기어를 바꾸고 인텔 컴파일러를 가지고 동일한 코드를 컴파일 할 시간이다.
나는 사전에 인텔 컴파일러와 가장 빠르다는 그의 명성에 관하여 얘기 들은 적이 있었다. 나는 인텔로 샘플 코드를 만들었을 때, 정말 감명받았다. 실제 인텔은 더 빠른 결과와 일반적으로 말하자면, 예기치 못한 결과를 나에게 가져다 준 인지할 수 있는 컴파일러를 만들었다.
마이크로소프트 컴파일러를 이용하여 컴파일한 코드와 비교할 경우 3-대역 이퀄라이저의 어셈블러 덤프를 검사하여 시작하기로 한다.
표 문서를 다운 받으려면 여기를 클릭하라 – 표 7 을 참조하라.
인텔 컴파일러는 약 20% 가량 코드를 축소할 수 있었다. 그리고, "Verlet Integrator"와 "제한 해결사"에 대한 결과는 각각 약 7%와 6% 가량 더 작았다.
하지만 실제 "마법"이었던 것은 코드를 만들었던 일반 VClass 가 더욱 작아졌다는 것이다. 내가 그 코드를 검사해 본 결과, 인텔 컴파일러는 절차적 요청보다 과부하 연산자에서 더 나았다. 또 다른 흥미로운 결과는 등급 내의 데이터 요약은 코드를 거의 팽창시키지 않으며, 사실상 헝겊에서는 전혀 팽창되지 않는다. 이것은 모든 데모 통계를 다소간 평평하게 만들었다.
다음은 인텔 컴파일러를 이용한 헝겊 데모의 스크린샷이다.

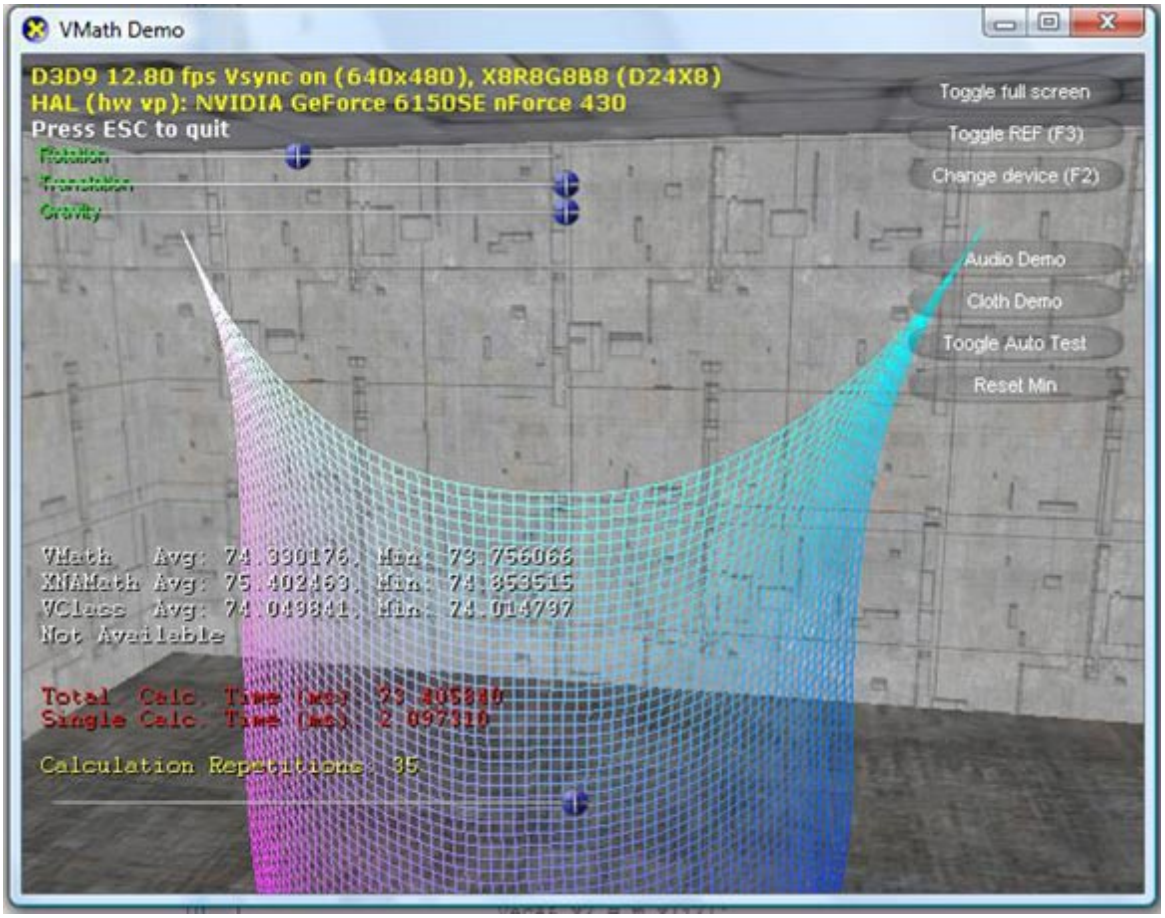

그림 3 – 인텔 컴파일러를 이용한 헝겊 데모

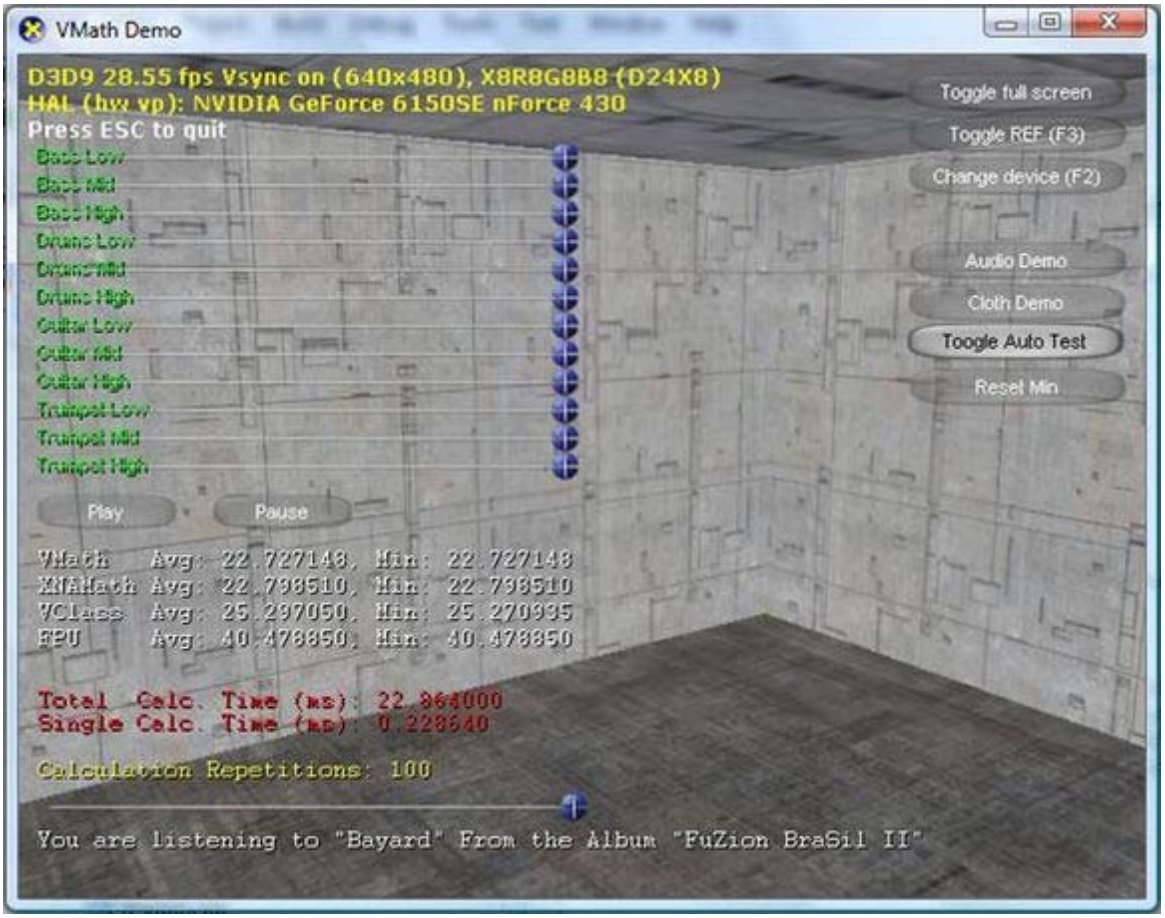

그림 4 – 인텔 컴파일러를 이용한 3 대역 EQ

최종 통계

이제 비교를 하기 위하여 모든 통계를 모아 보았다:

3-대역 이퀄라이저 알고리즘

마이크로소프트 컴파일러		인텔 컴파일러	
라이브러리	평균 시간	라이브러리	평균 시간
VMath (절차적 요청)	23.79	XNA Math (과부하 연산자)	22.72
XNA Math (과부하 연산자)	28.31	XNA Math (과부하 연산자)	22.79
XNA Math (과부하 연산자)	73.49	XNA Math (과부하 연산자)	25.29
FPU	78.09	FPU	40.47

헝겊 시뮬레이션 알고리즘

마이크로소프트 컴파일러		인텔 컴파일러	
라이브러리	평균 시간	라이브러리	평균 시간

VMath (절차적 요청)	75.78	XNA Math (과부하 연산자)	74.33
XNA Math (과부하 연산자)	76.73	XNA Math (과부하 연산자)	75.40
XNA Math (과부하 연산자)	138.52	XNA Math (과부하 연산자)	74.04

3-Band Equalizer Algorithm

Microsoft Compiler		Intel Compiler	
Library	Average Time	Library	Average Time
VMath (procedural calls)	23.79	VMath (overloaded operators)	22.72
XNA Math (overloaded operators)	28.31	XNA Math (overloaded operators)	22.79
VClass (overloaded operators)	73.49	VClass (overloaded operators)	25.29
FPU	78.09	FPU	40.47

Cloth Simulation Algorithm

Microsoft Compiler		Intel Compiler	
Library	Average Time	Library	Average Time
VMath (procedural calls)	75.78	VMath (overloaded operators)	74.33
XNA Math (overloaded operators)	76.73	XNA Math (overloaded operators)	75.40
VClass (overloaded operators)	138.52	VClass (overloaded operators)	74.04

결론

인텔 컴파일러가 등급 내의 데이터와 과부하 연산자에 의하여 생성된 코드 팽창의 문제를 해결하였지만, 나는 여전히 교차 플랫폼 SIMD 라이브러리에 대한 이러한 접근법을 배서하지 않았다. 이는 내가 PS2, PSP 및 PS3 용의 6 개의 다른 컴파일러 GCC 와 SN 시스템을 가지고 동일한 시험한 결과, 그들 모두는 마이크로소프트 컴파일러보다 더 나쁜 결과를 생산하는데, 코드를 더욱 팽창시키는 것을 의미하는 것이기 때문이다. 따라서, 만약 여러분이 윈도우 상에서만 운영되는 라이브러리를 특별히 기록하지 않는다면, 여러분의 최상의 선택은 여전히 여러분의 SIMD 벡터 라이브러리용으로 본 기사의 주요 요점을 따르는 것이다.

샘플 코드

모든 비교표에 따른 본 기사의 샘플 코드는 아래의 URL 에서 다운 받을 수 있다:

http://www.guitarv.com/ComputerScience.aspx?page=articles

참조:

[1] Microsoft, XNA Math Library Reference

[2] Intel, SHUFPS -- Shuffle Single-Precision Floating-Point Values

[3] Jakobsen, Thomas, Advanced Character Physics

[4] C., Neil, 3 Band Equaliser

[5] Wikipedia, Taylor Series