
# Internal Vectorization of Scalar Code

Guilherme Amadio
*Intel Parallel Computing Center – UNESP*

April 28, 2015

# Vectorization Study

- Scalar code only uses compiler auto-vectorization
- Need to provide some explicit vectorization of the code
  - Vector and matrix operations in Euclidean space
  - Random Number generation
- Performing tests to see if speedup is worth the effort
  - dot and cross products of vectors and quaternions

# Dot Product: Single Precision

```
float dot(const float * __restrict__ a, const float * __restrict__ b)
{
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2] + a[3] * b[3];
}

float dot_omp(const float * __restrict__ a, const float * __restrict__ b)
{
    float result = 0.0f;
    #pragma omp simd
    for (int i = 0; i < 4; ++i) { result += a[i] * b[i]; }
    return result;
}

float dot_sse1(const float * __restrict__ a, const float * __restrict__ b)
{
    __m128 va = _mm_load_ps(a);
    __m128 vb = _mm_load_ps(b);
    va = _mm_dp_ps(va, vb, 0xFF);
    float * rr = (float*)(&va);
    return (float)(*rr);
}

float dot_sse2(const float * __restrict__ a, const float * __restrict__ b)
{
    __m128 va = _mm_load_ps(a);
    __m128 vb = _mm_load_ps(b);
    return _mm_cvtss_f32(_mm_dp_ps(va, vb, 0xFF));
}
```

# Dot Product: Assembly Code

Intel C/C++ 15.0.2

```
<dot(float const*, float const*)>:
       movss  (%rdi),%xmm0
       movss  0x4(%rdi),%xmm1
       mulss  (%rsi),%xmm0
       mulss  0x4(%rsi),%xmm1
       movss  0x8(%rdi),%xmm2
       addss  %xmm1,%xmm0
       mulss  0x8(%rsi),%xmm2
       movss  0xc(%rdi),%xmm3
       addss  %xmm2,%xmm0
       mulss  0xc(%rsi),%xmm3
       addss  %xmm3,%xmm0
       retq
       nopl   0x0(%rax,%rax,1)
       nopl   0x0(%rax,%rax,1)

<dot_omp(float const*, float const*)>:
       movups (%rdi),%xmm3
       movups (%rsi),%xmm0
       mulps  %xmm0,%xmm3
       movaps %xmm3,%xmm1
       movhlps %xmm3,%xmm1
       addps  %xmm1,%xmm3
       movaps %xmm3,%xmm2
       shufps $0xf5,%xmm3,%xmm2
       addss  %xmm2,%xmm3
       movaps %xmm3,%xmm0
       retq
       nopl   0x0(%rax,%rax,1)

       nopl   0x0(%rax)

<dot_sse(float const*, float const*)>:
       movaps (%rdi),%xmm1
       movaps (%rsi),%xmm0
       dpps   $0xff,%xmm0,%xmm1
       movaps %xmm1,%xmm0
       retq
```

GNU G++ 4.9.2

```
<dot(float const*, float const*)>:
       movss  (%rdi),%xmm0
       movss  (%rsi),%xmm1
       mulss  %xmm0,%xmm1
       movss  0x4(%rdi),%xmm0
       mulss  0x4(%rsi),%xmm0
       addss  %xmm1,%xmm0
       movss  0x8(%rdi),%xmm1
       mulss  0x8(%rsi),%xmm1
       addss  %xmm1,%xmm0
       movss  0xc(%rdi),%xmm1
       mulss  0xc(%rsi),%xmm1
       addss  %xmm1,%xmm0
       retq
       nopw   0x0(%rax,%rax,1)

<dot_omp(float const*, float const*)>:
       pxor   %xmm0,%xmm0
       xor    %eax,%eax
       movss  (%rdi,%rax,1),%xmm1
       mulss  (%rsi,%rax,1),%xmm1
       add    $0x4,%rax
       cmp    $0x10,%rax
       addss  %xmm1,%xmm0
       jne    400ac6 <dot2(float const*, float const*)+0x6>
       repz retq

<dot_sse(float const*, float const*)>:
       movaps (%rdi),%xmm0
       dpps   $0xff,(%rsi),%xmm0
       retq
       nopw   0x0(%rax,%rax,1)
```

# Dot Product: Double Precision

```
double dot(const double * __restrict__ a, const double * __restrict__ b)
{
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2] + a[3] * b[3];
}

double dot_omp(const double * __restrict__ a, const double * __restrict__ b)
{
    double result = static_cast<double>(0);
    #pragma omp simd
    for (int i = 0; i < 4; ++i) { result += a[i] * b[i]; }
    return result;
}

double dot_avx(const double * __restrict__ a, const double * __restrict__ b)
{
    __m256d va = _mm256_load_pd(a);
    __m256d vb = _mm256_load_pd(b);
    va = _mm256_mul_pd(va,vb);
    vb = _mm256_permute2f128_pd(va, va, 0x1);
    va = _mm256_hadd_pd(va, vb);
    vb = _mm256_hadd_pd(va, va);
    double *dp = (double *)(&vb);
    return *dp;
}
```

# Double Dot Product: Assembly Code

## Intel C/C++ 15.0.2

```
<dot(double const*, double const*)>:
        vmovsd 0x10(%rdi),%xmm0
        vmovsd 0x18(%rdi),%xmm2
        vmulsd 0x10(%rsi),%xmm0,%xmm1
        vmulsd 0x18(%rsi),%xmm2,%xmm3
        vmovsd 0x8(%rdi),%xmm4
        vmovsd (%rdi),%xmm5
        vfmadd132sd 0x8(%rsi),%xmm1,%xmm4
        vfmadd132sd (%rsi),%xmm3,%xmm5
        vaddsd %xmm5,%xmm4,%xmm0
        retq
        nopl   (%rax)

<dot_omp(double const*, double const*)>:
        vmovupd (%rdi),%ymm1
        vxorpd %ymm0,%ymm0,%ymm0
        vfmadd132pd (%rsi),%ymm0,%ymm1
        vextractf128 $0x1,%ymm1,%xmm2
        vaddpd %xmm2,%xmm1,%xmm3
        vunpckhpd %xmm3,%xmm3,%xmm4
        vaddsd %xmm4,%xmm3,%xmm0
        vzeroupper
        retq
        nopl   0x0(%rax,%rax,1)
        nopl   0x0(%rax,%rax,1)


<dot_avx(double const*, double const*)>:
        push   %rbp
        mov    %rsp,%rbp
        and    $0xfffffffffffffe0,%rsp
        vmovupd (%rdi),%ymm0
        vmulpd (%rsi),%ymm0,%ymm1
        vperm2f128 $0x1,%ymm1,%ymm1,%ymm2
        vhaddpd %ymm2,%ymm1,%ymm3
        vhaddpd %ymm3,%ymm3,%ymm4
        vmovupd %ymm4,-0x20(%rsp)
        vmovsd -0x20(%rsp),%xmm0
        vzeroupper
        mov    %rbp,%rsp
        pop    %rbp
        retq
        nopl   0x0(%rax)
        nopl   0x0(%rax)
```

## GNU G++ 4.9.2

```
<dot(double const*, double const*)>:
        vmovsd 0x8(%rdi),%xmm0
        vmulsd 0x8(%rsi),%xmm0,%xmm0
        vmovsd (%rdi),%xmm1
        vfmadd231sd (%rsi),%xmm1,%xmm0
        vmovsd 0x10(%rdi),%xmm2
        vfmadd231sd 0x10(%rsi),%xmm2,%xmm0
        vmovsd 0x18(%rdi),%xmm3
        vfmadd231sd 0x18(%rsi),%xmm3,%xmm0
        retq
        nopw   0x0(%rax,%rax,1)

<dot_omp(double const*, double const*)>:
        xor    %eax,%eax
        vxorpd %xmm0,%xmm0,%xmm0
        vmovsd (%rdi,%rax,1),%xmm1
        vfmadd231sd (%rsi,%rax,1),%xmm1,%xmm0
        add    $0x8,%rax
        cmp    $0x20,%rax
        jne    400a56 <dot2(double const*, double const*)+0x6>
        retq
        nopl   0x0(%rax)

<dot_avx(double const*, double const*)>:
        vmovapd (%rdi),%ymm0
        vmulpd (%rsi),%ymm0,%ymm0
        vperm2f128 $0x1,%ymm0,%ymm0,%ymm1
        vhaddpd %ymm1,%ymm0,%ymm0
        vhaddpd %ymm0,%ymm0,%ymm0
        vzeroupper
        retq
        nopw   0x0(%rax,%rax,1)
```

# Cross Product: Single Precision

```
void cross(const float * __restrict__ a, const float * __restrict__ b, float * result) {
    result[0] = a[1]*b[2] - a[2]*b[1];
    result[1] = a[2]*b[0] - a[0]*b[2];
    result[2] = a[0]*b[1] - a[1]*b[0];
    return;
}

void cross_sse1(const float * __restrict__ a, const float * __restrict__ b, float * result)
{
    __m128 va = _mm_load_ps(a);
    __m128 vb = _mm_load_ps(b);
    __m128 tmp1 = _mm_shuffle_ps(va, va, _MM_SHUFFLE(3,0,2,1));
    __m128 tmp2 = _mm_shuffle_ps(vb, vb, _MM_SHUFFLE(3,1,0,2));
    __m128 tmp3 = _mm_shuffle_ps(va, va, _MM_SHUFFLE(3,1,0,2));
    __m128 tmp4 = _mm_shuffle_ps(vb, vb, _MM_SHUFFLE(3,0,2,1));
    _mm_store_ps(result, _mm_sub_ps(_mm_mul_ps(tmp1,tmp2), _mm_mul_ps(tmp3,tmp4)));
    return;
}

void cross_sse2(const float * __restrict__ a, const float * __restrict__ b, float * result)
{
    __m128 va = _mm_load_ps(a);
    __m128 tmp1 = _mm_shuffle_ps(va, va, _MM_SHUFFLE(3,0,2,1));
    __m128 tmp3 = _mm_shuffle_ps(va, va, _MM_SHUFFLE(3,1,0,2));
    __m128 vb = _mm_load_ps(b);
    __m128 tmp2 = _mm_shuffle_ps(vb, vb, _MM_SHUFFLE(3,1,0,2));
    __m128 tmp4 = _mm_shuffle_ps(vb, vb, _MM_SHUFFLE(3,0,2,1));
    _mm_store_ps(result, _mm_sub_ps(_mm_mul_ps(tmp1,tmp2), _mm_mul_ps(tmp3,tmp4)));
    return;
}
```

# Vector Cross Product Assembly Code

## Intel C/C++ 15.0.2

```
<cross(float const*, float const*, float*)>:
        movss  0x4(%rdi),%xmm6
        movss  0x8(%rdi),%xmm3
        movaps %xmm6,%xmm1
        movss  (%rdi),%xmm4
        movaps %xmm3,%xmm0
        movss  0x8(%rsi),%xmm2
        movss  0x4(%rsi),%xmm7
        movss  (%rsi),%xmm5
        mulss  %xmm2,%xmm1
        mulss  %xmm7,%xmm0
        mulss  %xmm5,%xmm3
        mulss  %xmm4,%xmm2
        mulss  %xmm4,%xmm7
        mulss  %xmm5,%xmm6
        subss  %xmm0,%xmm1
        subss  %xmm2,%xmm3
        subss  %xmm6,%xmm7
        movss  %xmm1,(%rdx)
        movss  %xmm3,0x4(%rdx)
        movss  %xmm7,0x8(%rdx)
        retq

<cross_sse(float const*, float const*, float*)>:
        movaps (%rdi),%xmm2
        movaps (%rsi),%xmm1
        movaps %xmm2,%xmm3
        movaps %xmm1,%xmm0
        shufps $0xc9,%xmm2,%xmm3
        shufps $0xd2,%xmm1,%xmm0
        shufps $0xd2,%xmm2,%xmm2
        shufps $0xc9,%xmm1,%xmm1
        mulps  %xmm0,%xmm3
        mulps  %xmm1,%xmm2
        subps  %xmm2,%xmm3
        movaps %xmm3,(%rdx)
        retq
```

## GNU G++ 4.9.2

```
<cross(float const*, float const*, float*)>:
        movss  0x4(%rdi),%xmm0
        movss  0x8(%rdi),%xmm1
        mulss  0x8(%rsi),%xmm0
        mulss  0x4(%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  %xmm0,(%rdx)
        movss  0x8(%rdi),%xmm0
        movss  (%rdi),%xmm1
        mulss  (%rsi),%xmm0
        mulss  0x8(%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  %xmm0,0x4(%rdx)
        movss  (%rdi),%xmm0
        movss  0x4(%rdi),%xmm1
        mulss  0x4(%rsi),%xmm0
        mulss  (%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  %xmm0,0x8(%rdx)
        retq

<cross_sse(float const*, float const*, float*)>:
        movaps (%rdi),%xmm1
        movaps (%rsi),%xmm0
        movaps %xmm1,%xmm3
        movaps %xmm0,%xmm2
        shufps $0xc9,%xmm1,%xmm3
        shufps $0xd2,%xmm0,%xmm2
        shufps $0xd2,%xmm1,%xmm1
        shufps $0xc9,%xmm0,%xmm0
        mulps  %xmm2,%xmm3
        mulps  %xmm1,%xmm0
        movaps %xmm3,%xmm1
        subps  %xmm0,%xmm1
        movaps %xmm1,(%rdx)
        retq
```

# Quaternion Product: Single Precision

```
void cross4(const float * __restrict__ q1, const float * __restrict__ q2, float *result) {
    result[0] = q1[3] * q2[0] + q1[0] * q2[3] + q1[1] * q2[2] - q1[2] * q2[1];
    result[1] = q1[3] * q2[1] + q1[1] * q2[3] + q1[2] * q2[0] - q1[0] * q2[2];
    result[2] = q1[3] * q2[2] + q1[2] * q2[3] + q1[0] * q2[1] - q1[1] * q2[0];
    result[3] = q1[3] * q2[3] - q1[0] * q2[0] - q1[1] * q2[1] - q1[2] * q2[2];
}

void cross4_sse(const float * __restrict__ q1, const float * __restrict__ q2, float *result)
{
    __m128 xyzw = _mm_load_ps(q1), abcd = _mm_load_ps(q2);
    __m128 baba = _mm_shuffle_ps(abcd, abcd, _MM_SHUFFLE(0,1,0,1));
    __m128 dcdc = _mm_shuffle_ps(abcd, abcd, _MM_SHUFFLE(2,3,2,3));
    __m128 mul1 = _mm_mul_ps(xyzw, baba), mul3 = _mm_mul_ps(xyzw, dcdc);
    __m128 wzyx = _mm_shuffle_ps(xyzw, xyzw, _MM_SHUFFLE(0,1,2,3));
    __m128 mul2 = _mm_mul_ps(wzyx, dcdc), mul4 = _mm_mul_ps(wzyx, baba);
    /* variable names below are for components of result (X,Y,Z,W), nX for -X */
    /* znxwy  = (xb - ya, zb - wa, wd - zc, yd - xc) */
    __m128 ZnXWY = _mm_hsub_ps(mul1, mul2);
    /* xzynw  = (xd + yc, zd + wc, wb + za, yb + xa) */
    __m128 XZYnW = _mm_hadd_ps(mul3, mul4);
    /* _mm_shuffle_ps(XZYnW, ZnXWY, _MM_SHUFFLE(3,2,1,0)) = (xd+yc, zd+wc, wd-zc, yd-xc) */
    /* _mm_shuffle_ps(ZnXWY, XZYnW, _MM_SHUFFLE(2,3,0,1)) = (zb-wa, xb-ya, yb+xa, wb+za) */
    /* _mm_addsub_ps adds elements 1 and 3 and subtracts elements 0 and 2, so we get:    */
    /* _mm_addsub_ps(*, *) = (xd+yc-zb+wa, xb-ya+zd+wc, wd-zc+yb+xa, yd-xc+wb+za)        */

    __m128 XZWY = _mm_addsub_ps(_mm_shuffle_ps(XZYnW, ZnXWY, _MM_SHUFFLE(3,2,1,0)),
                                _mm_shuffle_ps(ZnXWY, XZYnW, _MM_SHUFFLE(2,3,0,1)));

    /* now we shuffle components in place and save the result */
    _mm_store_ps(result, _mm_shuffle_ps(XZWY, XZWY, _MM_SHUFFLE(2,1,3,0)));
}
```

# Quaternion Cross Product: Assembly Code

## Intel C/C++ 15.0.2

```
<cross4(float const*,
        float const*, float*)>:
        movss  0xc(%rdi),%xmm0
        movss  (%rdi),%xmm2
        movaps %xmm0,%xmm11
        movss  (%rsi),%xmm3
        movaps %xmm2,%xmm8
        movss  0xc(%rsi),%xmm1
        movaps %xmm0,%xmm15
        mulss  %xmm3,%xmm11
        movaps %xmm1,%xmm12
        mulss  %xmm1,%xmm8
        movss  0x4(%rdi),%xmm5
        movaps %xmm3,%xmm13
        movss  0x8(%rsi),%xmm7
        movaps %xmm5,%xmm9
        mulss  %xmm7,%xmm9
        addss  %xmm8,%xmm11
        mulss  %xmm5,%xmm12
        addss  %xmm9,%xmm11
        movss  0x8(%rdi),%xmm6
        movaps %xmm1,%xmm8
        movss  0x4(%rsi),%xmm4
        movaps %xmm6,%xmm10
        mulss  %xmm4,%xmm10
        movaps %xmm2,%xmm9
        mulss  %xmm4,%xmm15
        mulss  %xmm6,%xmm8
        mulss  %xmm6,%xmm13
        mulss  %xmm4,%xmm9
        subss  %xmm10,%xmm11
        addss  %xmm12,%xmm15
        movss  %xmm11,(%rdx)
        movaps %xmm0,%xmm11
        mulss  %xmm7,%xmm11
        movaps %xmm3,%xmm10
        mulss  %xmm1,%xmm0
        addss  %xmm13,%xmm15
        mulss  %xmm2,%xmm3
        mulss  %xmm5,%xmm10
        addss  %xmm8,%xmm11
        mulss  %xmm4,%xmm5
        subss  %xmm3,%xmm0
        addss  %xmm9,%xmm11
        subss  %xmm5,%xmm0
        subss  %xmm10,%xmm11
        movaps %xmm2,%xmm14
        mulss  %xmm7,%xmm14
        mulss  %xmm6,%xmm7
        subss  %xmm14,%xmm15
        subss  %xmm7,%xmm0
        movss  %xmm15,0x4(%rdx)
        movss  %xmm11,0x8(%rdx)
        movss  %xmm0,0xc(%rdx)
        retq
        nopl   0x0(%rax)
        nopl   0x0(%rax)
```

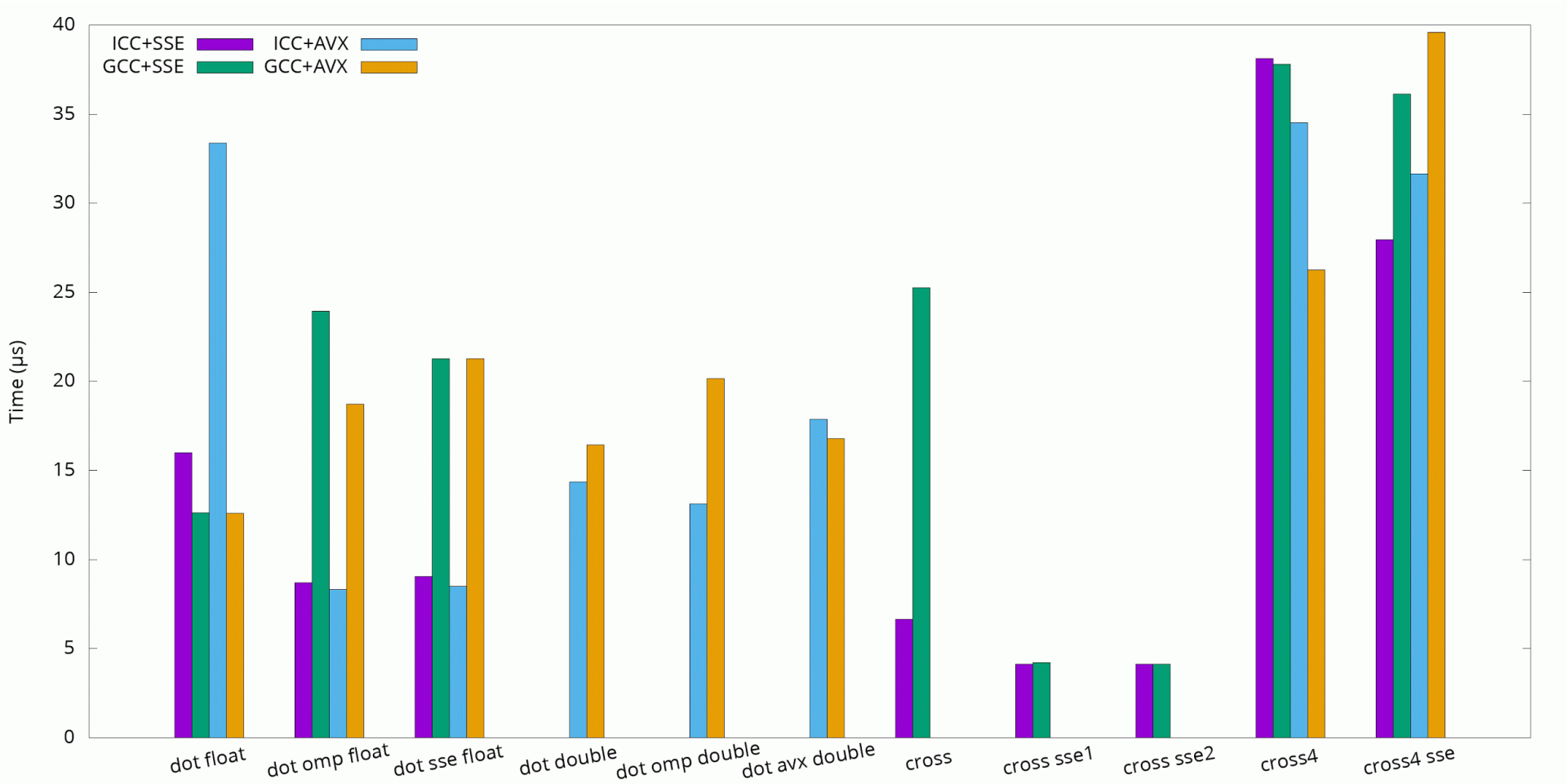
```
<cross4_sse(float const*,
            float const*, float*)>:
        movaps (%rsi),%xmm1
        movaps (%rdi),%xmm0
        movaps %xmm1,%xmm2
        shufps $0x11,%xmm1,%xmm2
        movaps %xmm0,%xmm4
        shufps $0xbb,%xmm1,%xmm1
        movaps %xmm0,%xmm3
        shufps $0x1b,%xmm0,%xmm0
        mulps  %xmm2,%xmm4
        mulps  %xmm1,%xmm3
        mulps  %xmm0,%xmm2
        mulps  %xmm0,%xmm1
        haddps %xmm2,%xmm3
        hsubps %xmm1,%xmm4
        movaps %xmm3,%xmm5
        shufps $0xe4,%xmm4,%xmm5
        shufps $0xb1,%xmm3,%xmm4
        addsubps %xmm4,%xmm5
        shufps $0x9c,%xmm5,%xmm5
        movaps %xmm5,(%rdx)
        retq
        nopl   (%rax)
        nopl   0x0(%rax)
```

## GNU G++ 4.9.2

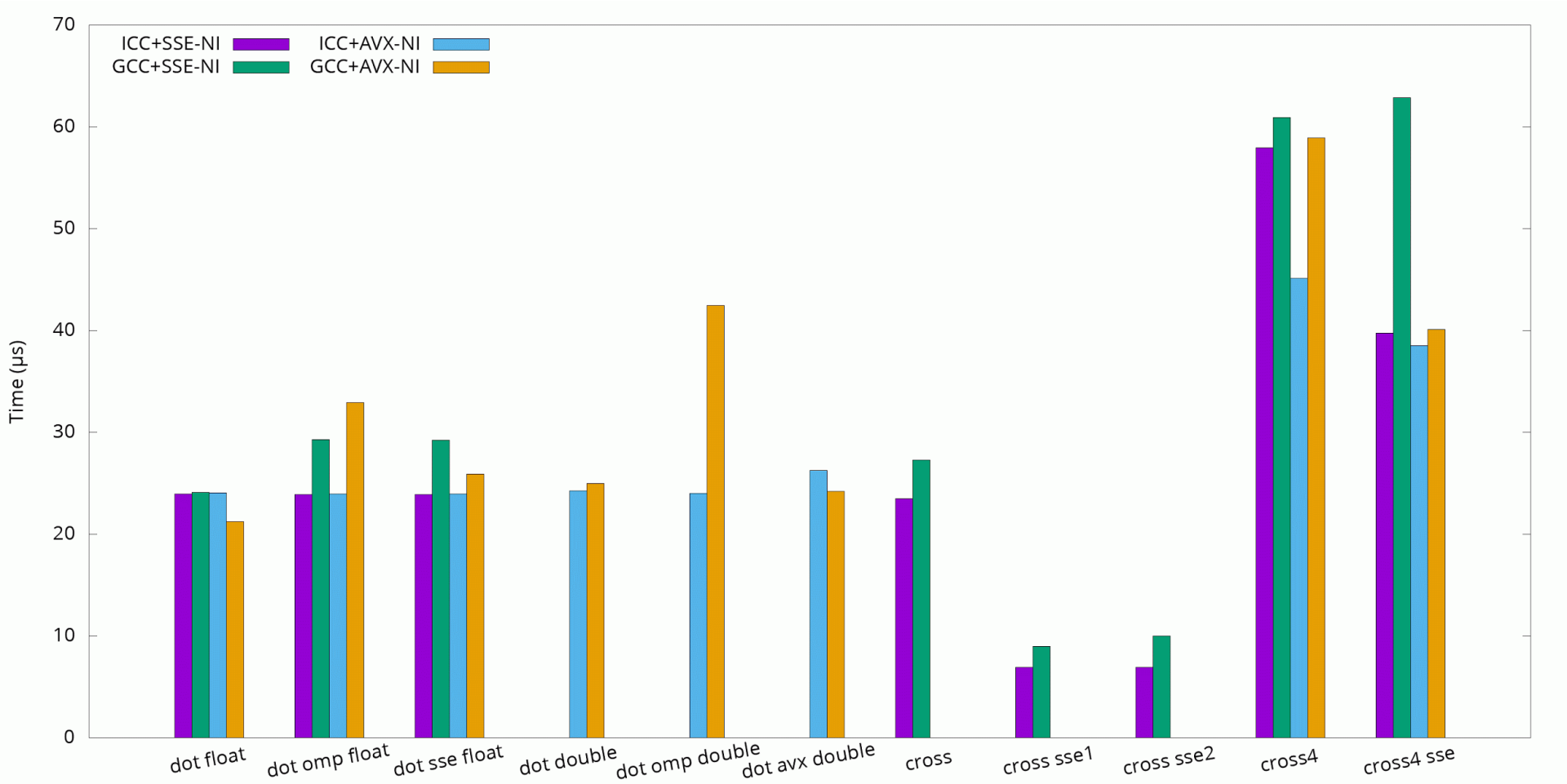
```
<cross4(float const*,
        float const*, float*)>:
        movss  0xc(%rdi),%xmm0
        movss  (%rsi),%xmm1
        mulss  %xmm0,%xmm1
        movss  (%rdi),%xmm0
        mulss  0xc(%rsi),%xmm0
        addss  %xmm1,%xmm0
        movss  0x4(%rdi),%xmm1
        mulss  0x8(%rsi),%xmm1
        addss  %xmm1,%xmm0
        movss  0x8(%rdi),%xmm1
        mulss  0x4(%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  %xmm0,(%rdx)
        movss  0xc(%rdi),%xmm0
        movss  0x4(%rsi),%xmm1
        mulss  %xmm0,%xmm1
        movss  0x4(%rdi),%xmm0
        mulss  0xc(%rsi),%xmm0
        addss  %xmm1,%xmm0
        movss  0x8(%rdi),%xmm1
        mulss  (%rsi),%xmm1
        addss  %xmm1,%xmm0
        movss  (%rdi),%xmm1
        mulss  0x8(%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  %xmm0,0x4(%rdx)
        movss  0xc(%rdi),%xmm0
        mulss  0x8(%rsi),%xmm0
        movaps %xmm0,%xmm1
        movss  0x8(%rdi),%xmm0
        mulss  0xc(%rsi),%xmm0
        addss  %xmm1,%xmm0
        movss  (%rdi),%xmm1
        mulss  0x4(%rsi),%xmm1
        addss  %xmm1,%xmm0
        movss  0x4(%rdi),%xmm1
        mulss  (%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  %xmm0,0x8(%rdx)
        movss  0xc(%rdi),%xmm0
        movss  (%rdi),%xmm1
        mulss  0xc(%rsi),%xmm0
        mulss  (%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  0x4(%rdi),%xmm1
        mulss  0x4(%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  0x8(%rdi),%xmm1
        mulss  0x8(%rsi),%xmm1
        subss  %xmm1,%xmm0
        movss  %xmm0,0xc(%rdx)
        retq
        nopw   0x0(%rax,%rax,1)
```

```
<cross4_sse(float const*,
            float const*, float*)>:
        movaps (%rsi),%xmm1
        movaps (%rdi),%xmm3
        movaps %xmm1,%xmm4
        movaps %xmm3,%xmm2
        movaps %xmm3,%xmm0
        shufps $0x11,%xmm1,%xmm4
        shufps $0x1b,%xmm3,%xmm3
        shufps $0xbb,%xmm1,%xmm1
        mulps  %xmm4,%xmm2
        mulps  %xmm1,%xmm0
        mulps  %xmm3,%xmm1
        mulps  %xmm4,%xmm3
        hsubps %xmm1,%xmm2
        haddps %xmm3,%xmm0
        movaps %xmm2,%xmm1
        shufps $0xb1,%xmm0,%xmm1
        shufps $0xe4,%xmm2,%xmm0
        addsubps %xmm1,%xmm0
        shufps $0x9c,%xmm0,%xmm0
        movaps %xmm0,(%rdx)
        retq
        nopw   %cs:0x0(%rax,%rax,1)
```

# Vectorization Benchmarks: Inline Functions



# Vectorization Benchmarks: No Inlining



# VTune Assembly Code Annotation

Cross4 Vector Product Example

- Assembly code much shorter for hand-coded version
- More cache friendly
- Twice as fast in VTune, although not twice as fast in small test
- What would happen when it's inside a large code?

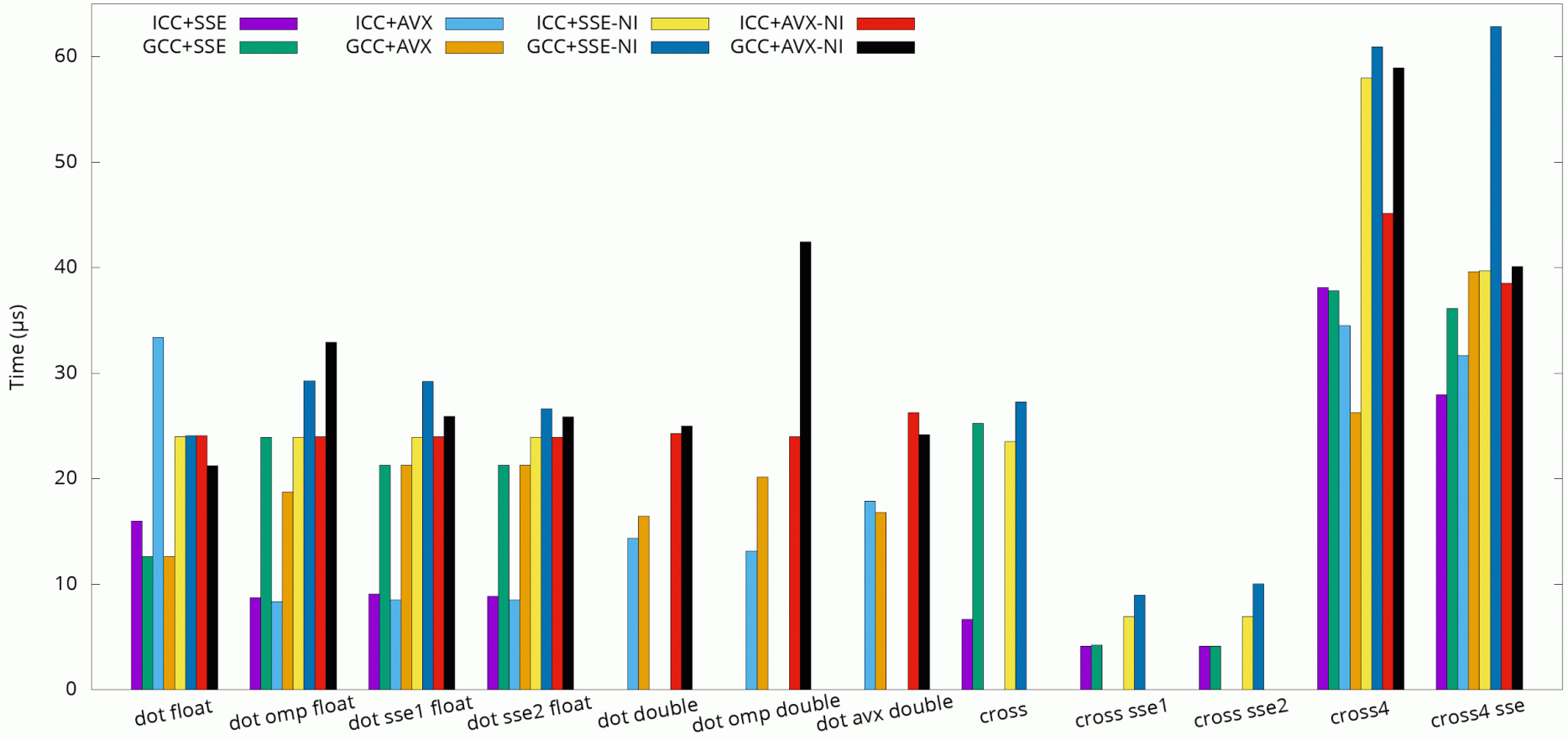
Portable Version

| Assembly | CPU Time: Total |
|---|---|
| **Block 1:** | |
| movssl 0xc(%rdi), %xmm0 | 153.683ms |
| movssl (%rdi), %xmm2 | 23.070ms |
| movaps %xmm0, %xmm11 | 485.211ms |
| movssl (%rsi), %xmm3 | 1.003ms |
| movaps %xmm2, %xmm8 | 0.072ms |
| movssl 0xc(%rsi), %xmm1 | 6.018ms |
| movaps %xmm0, %xmm15 | 493.703ms |
| mulss %xmm3, %xmm11 | |
| movaps %xmm1, %xmm12 | 1.003ms |
| mulss %xmm1, %xmm8 | 7.384ms |
| movssl 0x4(%rdi), %xmm5 | 469.474ms |
| movaps %xmm3, %xmm13 | 1.003ms |
| movssl 0x8(%rsi), %xmm7 | |
| movaps %xmm5, %xmm9 | 3.010ms |
| mulss %xmm7, %xmm9 | 512.035ms |
| addss %xmm8, %xmm11 | 1.003ms |
| mulss %xmm5, %xmm12 | 5.214ms |
| addss %xmm9, %xmm11 | 9.029ms |
| movssl 0x8(%rdi), %xmm6 | 517.630ms |
| movaps %xmm1, %xmm8 | |
| movssl 0x4(%rsi), %xmm4 | |
| movaps %xmm6, %xmm10 | 9.028ms |
| mulss %xmm4, %xmm10 | 497.442ms |
| movaps %xmm2, %xmm9 | |
| mulss %xmm4, %xmm15 | |
| mulss %xmm6, %xmm8 | 8.026ms |
| mulss %xmm6, %xmm13 | 496.159ms |
| mulss %xmm4, %xmm9 | 1.003ms |
| subss %xmm10, %xmm11 | |
| addss %xmm12, %xmm15 | 11.964ms |
| movssl %xmm11, (%rdx) | 539.392ms |
| movaps %xmm0, %xmm11 | |
| mulss %xmm7, %xmm11 | |
| movaps %xmm3, %xmm10 | 8.025ms |
| mulss %xmm1, %xmm0 | 512.767ms |
| addss %xmm13, %xmm15 | 2.006ms |
| mulss %xmm2, %xmm3 | 1.003ms |
| mulss %xmm5, %xmm10 | 6.021ms |
| addss %xmm8, %xmm11 | 513.832ms |
| mulss %xmm4, %xmm5 | |
| subss %xmm3, %xmm0 | |
| addss %xmm9, %xmm11 | 7.022ms |
| subss %xmm5, %xmm0 | 505.845ms |
| subss %xmm10, %xmm11 | |
| movaps %xmm2, %xmm14 | 0.934ms |
| mulss %xmm7, %xmm14 | 8.025ms |
| mulss %xmm6, %xmm7 | 484.301ms |
| subss %xmm14, %xmm15 | 1.003ms |
| subss %xmm7, %xmm0 | |
| movssl %xmm15, 0x4(%rdx) | 11.034ms |
| movssl %xmm11, 0x8(%rdx) | 480.891ms |
| movssl %xmm0, 0xc(%rdx) | 1.003ms |
| retq | |
| **Block 2:** | |
| nopl %eax, (%rax) | |
| nopl %eax, (%rax) | |

Hand-coded in SSE

| Assembly | CPU Time: Total |
|---|---|
| **Block 1:** | |
| movapsx (%rsi), %xmm1 | 215.082ms |
| movapsx (%rdi), %xmm0 | 24.188ms |
| movaps %xmm1, %xmm2 | 380.657ms |
| shufps $0x11, %xmm1, %xmm2 | 3.009ms |
| movaps %xmm0, %xmm4 | 62.185ms |
| shufps $0xbb, %xmm1, %xmm1 | 9.027ms |
| movaps %xmm0, %xmm3 | 411.533ms |
| shufps $0x1b, %xmm0, %xmm0 | 2.006ms |
| mulps %xmm2, %xmm4 | 63.188ms |
| mulps %xmm1, %xmm3 | 13.039ms |
| mulps %xmm0, %xmm2 | 416.924ms |
| mulps %xmm0, %xmm1 | 5.015ms |
| haddps %xmm2, %xmm3 | 75.227ms |
| hsubps %xmm1, %xmm4 | 423.350ms |
| movaps %xmm3, %xmm5 | 480.945ms |
| shufps $0xe4, %xmm4, %xmm5 | 4.012ms |
| shufps $0xb1, %xmm3, %xmm4 | 71.071ms |
| addsubps %xmm4, %xmm5 | 19.621ms |
| shufps $0x9c, %xmm5, %xmm5 | 436.723ms |
| movapsx %xmm5, (%rdx) | 11.158ms |
| retq | 83.507ms |
| **Block 2:** | |
| nopl %eax, (%rax) | |
| nopl %eax, (%rax) | |

# All Vectorization Benchmarks



# Package Management Using Gentoo Prefix

- The Gentoo Prefix project develops and maintains a way of installing Gentoo systems in a non-standard location
- Supported systems: Linux, Mac OS, Solaris. Installable into other systems, such as Windows and FreeBSD, but support is bad (lack of manpower)
- Gentoo is a source distribution, Portage package manager compiles everything from source
- Installation via a shell script that bootstraps Portage and the toolchain
- Automatic dependency management, high configurability via USE flags
- yum install $package → emerge $package
- New packages can be easily added to the Portage database
- Portage can also cross compile software for the Xeon Phi with appropriate settings, good way of managing dependencies of cross compiled packages
- More information: https://wiki.gentoo.org/wiki/Project:Prefix