Streaming Load/Store

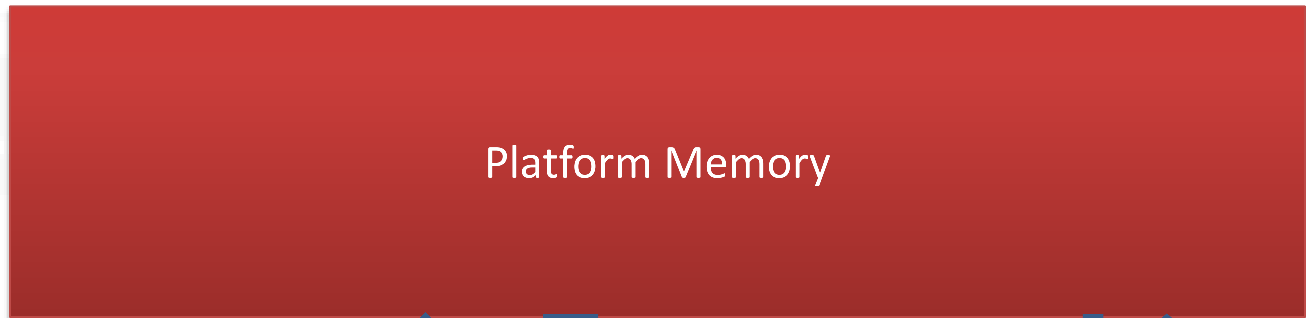
A case study using algorithm for finding solutions of quadratic equations.

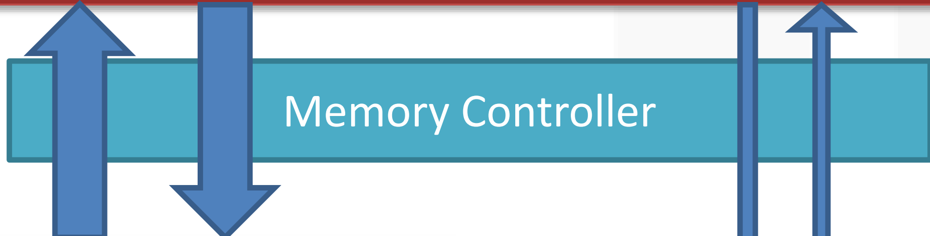
SSE – **Streaming** SIMD extension

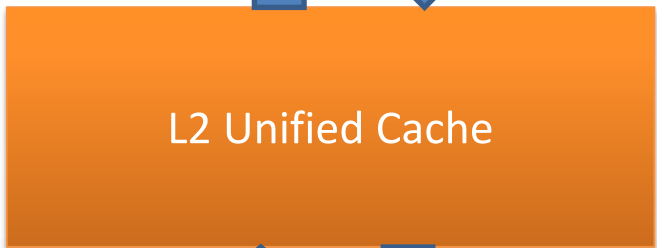
- „The **streaming load** instruction in SSE4.1, MOVNTDQA, provides a non-temporal hint that can cause adjacent 16- byte items within an aligned 64-byte region of WC memory type (a streaming line) to be fetched and held in a small set of temporary buffers (“**streaming load buffers**”). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be satisfied from the streaming load buffer and can improve throughput.”

- „**Streaming stores** (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD)”

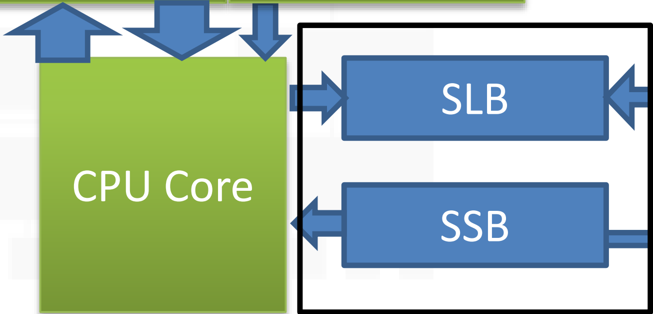
- **Temporal vs. Non-temporal**: „Data referenced by a program can be temporal (data will be used again) or non-temporal (data will be referenced once and not reused in the immediate future).”


Platform Memory


Memory Controller


L2 Unified Cache

L1 Data
L1 Instructions


CPU Core
SLB
SSB

Increases the
memory throughput
when no data-reuse

Quadratic Solver (Intrinsics)

```
UME_FORCE_INLINE void QuadSolveAVX2(
    const float* __restrict__ a,
    const float* __restrict__ b,
    const float* __restrict__ c,
    float* __restrict__ x1,
    float* __restrict__ x2,
    int* __restrict__ roots)
{
    __m256 one = _mm256_set1_ps(1.0f);
    __m256 va = _mm256_load_ps(a);
    __m256 vb = _mm256_load_ps(b);
    __m256 zero = _mm256_set1_ps(0.0f);
    __m256 a_inv = _mm256_div_ps(one, va);
    __m256 b2 = _mm256_mul_ps(vb, vb);
    __m256 eps = _mm256_set1_ps(std::numeric_limits<float>::epsilon());
    __m256 vc = _mm256_load_ps(c);
```

Mind the forced inlining!

Some load operations at the beginning...

⋮

... some CPU bound computing ...

```
    r2 = _mm256_blendv_ps(r3, r2, mask2);
    _mm256_store_si256((__m256i*)roots, _mm256_cvtps_epi32(nr));
    _mm256_store_ps(x1, r1);
    _mm256_store_ps(x2, r2);
}
```

... and some store operations at the end.

NO STREAMING OPERATIONS USED DIRECTLY!

Quadratic Solver (UME)

```
template <typename SCALAR_FLOAT_T, typename FLOAT_VEC_T, typename INT_VEC_T>
UME_FORCE_INLINE void QuadSolveSIMD(
    const SCALAR_FLOAT_T* __restrict a,
    const SCALAR_FLOAT_T* __restrict b,
    const SCALAR_FLOAT_T* __restrict c,
    SCALAR_FLOAT_T* __restrict x1,
    SCALAR_FLOAT_T* __restrict x2,
    int* __restrict roots
    )
{
    typedef typename UME::SIMD::SIMDTraits<FLOAT_VEC_T>::MASK_T MASK_T;
    typedef typename UME::SIMD::SIMDTraits<FLOAT_VEC_T>::INT_VEC_T INT_VEC_T;

    FLOAT_VEC_T one(1.0f);
    FLOAT_VEC_T va(&a[0]);
    FLOAT_VEC_T vb(&b[0]);
    FLOAT_VEC_T zero(0.0f);
    FLOAT_VEC_T a_inv = one / va;
    FLOAT_VEC_T b2 = vb * vb;
    FLOAT_VEC_T eps(std::numeric_limits<SCALAR_FLOAT_T>::epsilon());
    FLOAT_VEC_T vc(&c[0]);
```

Mind the forced inlining!

Some load operations at the beginning...

•
•
•

... some CPU bound computing ...

```
    INT_VEC_T int_roots(nr);
    UME::SIMD::SIMDVec<int, INT_VEC_T::length()> int_roots2(int_roots);
    int_roots2.store(roots);
    r1.store(x1);
    r2.store(x2);
}
```

... and some store operations at the end.

NO STREAMING OPERATIONS USED DIRECTLY!

How do we measure

```
    FLOAT_T *a = (FLOAT_T *)UME::DynamicMemory::AlignedMalloc(ARRAY_SIZE * sizeof(FLOAT_T), 64);
    FLOAT_T *b = (FLOAT_T *)UME::DynamicMemory::AlignedMalloc(ARRAY_SIZE * sizeof(FLOAT_T), 64);
    FLOAT_T *c = (FLOAT_T *)UME::DynamicMemory::AlignedMalloc(ARRAY_SIZE * sizeof(FLOAT_T), 64);

    int *roots = (int *)UME::DynamicMemory::AlignedMalloc(ARRAY_SIZE * sizeof(int), 64);
    FLOAT_T *x1 = (FLOAT_T *)UME::DynamicMemory::AlignedMalloc(ARRAY_SIZE * sizeof(FLOAT_T), 64);
    FLOAT_T *x2 = (FLOAT_T *)UME::DynamicMemory::AlignedMalloc(ARRAY_SIZE * sizeof(FLOAT_T), 64);
```

```
    start = __rdtsc();
    for (int i = 0; i < ARRAY_SIZE; i += LENGTH) {
        QuadSolveSIMD<FLOAT_T, FLOAT_VEC_T, INT_VEC_T>(&a[i], &b[i], &c[i], &x1[i], &x2[i], &roots[i]);
    }
    end = __rdtsc();
```

Initial performance

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2

Scalar naive (float): 248987648, dev: 115082 (speedup: 1.0x)
AVX2 intrinsic code (float, 8): 50073976, dev: 76471 (speedup: 4.9724x)
UME::SIMD (float, 8): 74357808, dev: 134799 (speedup: 3.34851x)
```

Quick look at assembly

```
    r2 = _mm256_blendv_ps(r3, r2, mask2);
    _mm256_store_si256((__m256i*)roots, _mm256_cvtps_epi32(nr));
    _mm256_store_ps(x1, r1);
    _mm256_store_ps(x2, r2);
}
```

```
        vroundps  $11, %ymm4, %ymm1                                 #96.9
        vcvtps2dq %ymm1, %ymm2                                      #96.9
        vmovntdq  %ymm2, (%rbx,%rax,4)                              #96.9
        vmovntps  %ymm0, (%r14,%rax,4)                              #96.9
        vmovntps  %ymm3, (%rdx,%rax,4)                              #96.9
        addq      $8, %rax                                          #95.37
        cmpq      $8388608, %rax                                    #95.25
        jl        ..B41.93      # Prob 99%                          #95.25
                                # LOE rax rdx rbx r12 r13 r14 r15 esi ymm
..B41.94:                       # Preds ..B41.93
        rdtsc                                                       #98.11
```

Streaming store

Regular store

```
    INT_VEC_T int_roots(nr);
    UME::SIMD::SIMDVec<int, INT_VEC_T::length()> int_roots2(int_roots);
    int_roots2.store(roots);
    r1.store(x1);
    r2.store(x2);
}
```

```
        vroundps  $11, %ymm4, %ymm1                                 #133.9
        vmovups   %ymm0, (%r15,%rax,4)                              #133.9
        vmovups   %ymm3, (%rdx,%rax,4)                              #133.9
        vcvtps2dq %ymm1, %ymm2                                      #133.9
        vmovdqu   %ymm2, (%rbx,%rax,4)                              #133.9
        addq      $8, %rax                                          #132.37
        cmpq      $8388608, %rax                                    #132.25
        jl        ..B27.93      # Prob 82%                          #132.25
                                # LOE rax rdx rbx r12 r13 r14 r15 esi ymm5
..B27.94:                       # Preds ..B27.93
        rdtsc                                                       #145.11
```

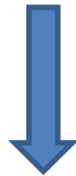
Disabling auto-generation of stream operations.

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2

Scalar naive (float): 248987648, dev: 115082 (speedup: 1.0x)
AVX2 intrinsic code (float, 8): 50073976, dev: 76471 (speedup: 4.9724x)
UME::SIMD (float, 8): 74357808, dev: 134799 (speedup: 3.34851x)
```

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2 -qopt-streaming-stores=never

Scalar naive (float):249520416, dev: 84336 (speedup: 1.0x)
AVX2 intrinsic code (float, 8):71763880, dev: 92393 (speedup: 3.47696x)
UME::SIMD (float, 8): 73422344, dev: 96860 (speedup: 3.39843x)
```

Enhancing UME implementation

```
    INT_VEC_T int_roots(nr);
    UME::SIMD::SIMDVec<int, INT_VEC_T::length()> int_roots2(int_roots);
    int_roots2.store(roots);
    r1.store(x1);
    r2.store(x2);
}
```



```
    INT_VEC_T int_roots(nr);
    UME::SIMD::SIMDVec<int, INT_VEC_T::length()> int_roots2(int_roots);
    int_roots2.sstore(roots);
    r1.sstore(x1);
    r2.sstore(x2);
}
```

Enhancing UME implementation

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2

Scalar naive (float): 248987648, dev: 115082 (speedup: 1.0x)
AVX2 intrinsic code (float, 8): 50073976, dev: 76471 (speedup: 4.9724x)
UME::SIMD (float, 8): 74357808, dev: 134799 (speedup: 3.34851x)
```

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2 -qopt-streaming-stores=never

Scalar naive (float):249520416, dev: 84336 (speedup: 1.0x)
AVX2 intrinsic code (float, 8):71763880, dev: 92393 (speedup: 3.47696x)
UME::SIMD (float, 8): 73422344, dev: 96860 (speedup: 3.39843x)
```

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2

Scalar naive (float):249546208, dev: 121980 (speedup: 1.0x)
AVX2 intrinsic code (float, 8):50443712, dev: 83484 (speedup: 4.94702x)
UME::SIMD (float, 8): 54759896, dev: 194244 (speedup: 4.5571x)
```

Reducing workload size

- Original workload: 192 MB (96MB in/96 MB out)

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2

Scalar naive (float): 248987648, dev: 115082 (speedup: 1.0x)
AVX2 intrinsic code (float, 8): 50073976, dev: 76471 (speedup: 4.9724x)
UME::SIMD (float, 8): 74357808, dev: 134799 (speedup: 3.34851x)
```

- Small workload (1/50 of original): 3.84MB (2MB in/2MB out)

```
$ icc -std=c++11 -O3 *.cpp -xCORE-AVX2

Scalar naive (float):4411481, dev: 13534 (speedup: 1.0x)
AVX2 intrinsic code (float, 8):944364, dev: 2417 (speedup: 4.67138x)
UME::SIMD (float, 8): 940389, dev: 288 (speedup: 4.69112x)
```

‚Correct’ solution

```
UME_NEVER_INLINE void QuadSolveAVX2(
    const float* __restrict__ a,
    const float* __restrict__ b,
    const float* __restrict__ c,
    float* __restrict__ x1,
    float* __restrict__ x2,
    int* __restrict__ roots)
```

- In this scenario data will be passed through vector registers, and used again.
- Inlining will use this peep-hole optimization in some cases, and not use it for others
- Forbidding inlining on all benchmarking functions introduces some error.

Conclusions

1. Streaming load/store operations might hide latency.
2. Effect might be only visible when memory usage is ‚big enough'.
3. Some effects cannot be detected by static analysis (e.g. Assembly inspection)
4. Adding explicit streaming stores might be useful in generic backend system!!!

Know what you are benchmarking
and
make sure compiler does what you expect.