Looking for 4x speedups?
SSE to the rescue!

Mostafa Hagog
Microprocessor Technology Labs
Agenda

- SSE overview.
- SSE to accelerate heavy computational applications.
- SSE to accelerate Video encoding and Image processing.
- SSE to provide Graphics building blocks.
- Streaming auxiliary instructions.
- Using the Compiler to generate SSE
SSE overview

• SSE = Streaming SIMD Extension
• SIMD – Single Instruction Multiple Data.
• One instruction to do the same operation on 4 packed elements simultaneously.

```c
void foo (float *a, float *b, float *c, int n){
  for (i = 0 ; i < n; i++){
    a[i] = b[i]*c[i];
  }
}
```

Scalar loop:
```
L1:
  movss   xmm0, [rdx+r13*4]
  mulss   xmm0, [r8+r13*4]
  movss   [rcx+r13*4], xmm0
  add     r13, 1
  cmp     r13, r9
  jl      L1
```

Vector loop:
```
L1:
  movups  xmm1, [rdx+r9*4]
  movups  xmm0, [r8+r9*4]
  mulps   xmm1, xmm0
  movaps  [rcx+r9*4], xmm1
  add     r9, 4
  cmp     r9, rax
  jl      L1
```

6 instructions
4 elements
7 instructions
4 elements
1.75 instructions
4 elements
SSE Data types

- 4x floats
- 2x doubles
- 16x bytes
- 8x 16-bit shorts
- 4x 32-bit integers
- 2x 64-bit integers
SSE instructions overview

• Arithmetic:
  - Multiply, Add, Subtract, Divide, Square root and more.

• Logic:
  - and, and-not, or, xor

• Other:
  - Min/Max, shuffle, packed compares, Blending, type conversion (e.g. int to float and float to double).

• Dedicate functionality:
  - MPSADBW (Fast Block Difference), DPPS (Dot Product).
C Compiler support for SSE – A glance to Intrinsics

• Vector Data Types:
  - __m128 for single precision.
  - __m128i for integers.
  - __m128d for double precision

• Each instructions has its equivalent intrinsic, example intrinsics:
  - extern __m128 _mm_add_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_mul_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_and_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_cmpeq_ps(__m128 _A, __m128 _B);

• More details will follow.
Evolution of SSE

Intel® SSE
1999

70 instructions
• Single-Precision Vectors,
• Streaming operations

Intel® SSE2
2000

144 instructions
• Double-precision Vectors
• 128-bit vector integer

Intel® SSE3
2004

13 instructions
• Complex Arithmetic

Intel® SSSE3
2006

32 instructions
• Decode

Intel® SSE4
2007

47 instructions
• Video Accelerators
• Graphics building blocks
• Coprocessor Accelerators

Copyright © Intel Corporation, 2007
Agenda

- SSE overview.
- SSE to accelerate heavy computational applications.
- SSE to accelerate Video encoding and Image processing.
- SSE to provide Graphics building blocks.
- Streaming auxiliary instructions.
- Using the Compiler to generate SSE
Complex Multiply – using ADDSUB

\[(C.\text{real} + i*C.\text{img}) = (A.\text{real} + i*A.\text{img} ) * (B.\text{real} + i*B.\text{img} )\]

\[
\begin{align*}
C.\text{real} &= A.\text{real}*B.\text{real} - A.\text{img}*B.\text{img} \\
C.\text{img} &= A.\text{real}*B.\text{img} + A.\text{img}*B.\text{real}
\end{align*}
\]

2.5 instructions
Instead of 6
per one complex multiply
FIR Filter using SSE (1/3)

- Used in Filtering of speech signals in modern voice coders and many other signal processing areas.
- An \( M \), length filter \( h[0, ..., M - 1] \), applied to an input sequence \( x[0, ..., N-1] \) generates output sequence \( y[0, ..., N-1] \), as described in the following equation:

\[
 y(n) = \sum_{i=0}^{M-1} h(i) x(n - i)
\]
FIR Filter using SSE (2/3)

Input signal X

8 Tap FIR

Output signal Y
FIR Filter using SSE (3/3)

__m128 X, X1, X2, Y, H, H0, H1, H2, H3;

...  
H0 = _mm_shuffle_ps (H, H, _MM_SHUFFLE(0,0,0,0));
X = _mm_mul_ps (X1, H0);  
Y = _mm_add_ps (Y, X);
H1 = _mm_shuffle_ps (H, H, _MM_SHUFFLE(1,1,1,1));
X = _mm_alignr_epi8 (X2,X1, 4);
X = _mm_mul_ps (X, H1);  
Y = _mm_add_ps (Y, X);
H2 = _mm_shuffle_ps (H, H, _MM_SHUFFLE(2,2,2,2));
X = _mm_alignr_epi8 (X2,X1, 8);
X = _mm_mul_ps (X, H2);  
Y = _mm_add_ps (Y, X);
H3 = _mm_shuffle_ps (H, H, _MM_SHUFFLE(3,3,3,3));
X = _mm_alignr_epi8 (X2,X1, 12);
X = _mm_mul_ps (X, H3);  
Y = _mm_add_ps (Y, X);
...
_mm_store_ps(&y[i], Y);
Agenda

• SSE overview.
• SSE to accelerate heavy computational applications.
• SSE to accelerate Video encoding and Image processing.
• SSE to provide Graphics building blocks.
• Streaming auxiliary instructions.
• Using the Compiler to generate SSE
Accelerate Video encoding using Fast Block Differences instruction

- **Sum of Absolute Differences** \( (SAD) \) is a widely used, extremely simple video quality metric used for block-matching in motion estimation for video compression.

\[
SAD(x, y) = \sum_{i=1}^{N} \sum_{j=1}^{N} |f(i, j, t) - f(i + x, j + y, t - 1)|
\]

- \(-w < (x, y) < w, w\) is the search area of motion vector and \(N\) is the block size.

- \(f(i, j, t)\) represents a pixel with coordinate \((i, j)\) on frame \(t\).

- Search exhaustively within a search window to find the motion vector:

\[
(mvx, mvy) = \min_{x,y} SAD(x, y)
\]
SSE4 Fast Block Difference instruction

extern __m128i __cdecl _mm_mpsadbw_epu8(__m128i s1, __m128i s2, const int msk);

msk[0:1]  control
Bit 0,1  select 4 bytes in src
Bit 2  select 1 group (11 bytes) in dst

xm3 = _mm_mpsadbw_epu8(xm1, xm0, 0);
xm4 = _mm_mpsadbw_epu8(xm1, xm0, 5);
xm4 = _mm_add_epi16(xm4, xm3);
xm5 = _mm_alignr_epi8 (xm2, xm1, 8);
xm6 = _mm_mpsadbw_epu8(xm5, xm0, 2);
xm7 = _mm_mpsadbw_epu8(xm5, xm0, 7);
xm7 = _mm_add_epi16(xm7, xm6);
_mm_store_si128((__m128i *)(aptr+offset), xm4);
_mm_store_si128((__m128i *)(aptr+offset+8), xm7);

Calculate 16 4-byte SAD,
Add the results to get 8 8-byte SAD,
increase offset by 8 bytes,
Calculate the next 8 8-byte SAD
Only 9 instructions!
Agenda

• SSE overview.
• SSE to accelerate heavy computational applications.
• SSE to accelerate Video encoding and Image processing.
• SSE to provide Graphics building blocks.
• Streaming auxiliary instructions.
• Using the Compiler to generate SSE
Blends: To Boost Conditionals SIMD flows

/*Integer blend instructions */
_mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);
_mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);
/*Float single precision blend instructions */
_mm_blend_ps (__m128  v1, __m128  v2, const int mask);
_mm_blendv_ps(__m128  v1, __m128  v2, __m128  v3);
/*Float double precision blend instructions */
_mm_blend_pd (__m128d v1, __m128d v2, const int mask);
_mm_blendv_pd (__m128d v1, __m128d v2, __m128d v3);

• **Used to code conditional SIMD flows**
  for (i=0; i<N; i++)
    if (a[i]<b[i]) c[i]=a[i]*b[i];
    else c[i]=a[i];

  **Vector code assuming:**
  for (i=0; i<N; i+=4) {
    A = _mm_loadu_ps(&a[i]);
    B = _mm_loadu_ps(&b[i]);
    C = _mm_mul_ps (A, B);
    mask = _mm_cmplt_ps (A, B);
    C = _mm_blend_ps (C, A, mask);
    _mm_storeu_ps (&c[i], C);
  }
Dot Product

_mm_dp_ps(__m128 val1, __m128 val2, const int mask);
_mm_dp_pd(__m128d val1, __m128d val2, const int mask);

result = _mm_dp_ps(v1, v2, 0xF3)
### Non-unit Stride Operations

```c
_mm_insert_ps(__m128 dst, __m128 src, const int ndx);
_mm_insert_epi8_epi32_epi64 (__m128i dst, int src, const int ndx);
_mm_extract_ps(__m128 src, const int ndx);
_mm_extract_epi8_epi32_epi64 (__m128i src, const int ndx);
```

#### Strided Load

```
xm1 = _mm_load_ss (a);
xm1 = _mm_insert_ps (xm1, a+stride, 0x10);
xm1 = _mm_insert_ps (xm1, a+2*stride, 0x20);
xm1 = _mm_insert_ps (xm1, a+3*stride, 0x30);
```

#### Strided Store

```
*a = _mm_extract_ps (x1, 0);
*(a +stride) = _mm_extract_ps (x1, 1);
*(a +2*stride) = _mm_extract_ps (x1, 2);
*(a +3*stride) = _mm_extract_ps (x1, 3);
```

#### Gather

```
i = _mm_extract_epi32 (xm1, 0);
xm2 = _mm_load_ss (&arr[i]);
i = _mm_extract_epi32 (xm1, 1);
xm2 = _mm_load_ss (&arr[i]);
i = _mm_extract_epi32 (xm1, 2);
xm2 = _mm_load_ss (&arr[i]);
i = _mm_extract_epi32 (xm1, 3);
xm2 = _mm_load_ss (&arr[i]);
xm2 = _mm_insert_ps (xm2, &arr[i], 0x30);
```

#### Scatter

```
i = _mm_extract_epi32 (xm1, 0);
_mm_store_ss (&arr[i], xm2);
i = _mm_extract_epi32 (xm1, 1);
arr[i] = _mm_extract_ps (xm2, 1);
i = _mm_extract_epi32 (xm1, 2);
arr[i] = _mm_extract_ps (xm2, 2);
i = _mm_extract_epi32 (xm1, 3);
arr[i] = _mm_extract_ps (xm2, 3);
```
Vector Early Out

_mm_test_all_zeros(mask, val)
_mm_test_all_ones(val)
_mm_test_mix_ones_zeros(mask, val)

Example

for (i=0; i<N; i++) {
    if (!a[i]) continue;
    BODY;
}

for (i=0; i<N; i++) {
    if (_mm_test_all_zeros (mask, arr))
        continue;
    VECTORIZED BODY;
}
Agenda

- SSE overview.
- SSE to accelerate heavy computational applications.
- SSE to accelerate Video encoding and Image processing.
- SSE to provide Graphics building blocks.
- Streaming auxiliary instructions.
- Using the Compiler to generate SSE
Streaming Load - Communicate with Coprocessors an Order of Magnitude Faster

`__m128i_mm_stream_load_si128(__m128i *p);`

**Existing Platforms**

- Core™
- GFX
- Graphics Mem
- High BW Writes
- But ... Low BW Reads
- DDR2
- DDR2

**With Streaming Loads**

- 45nm Next Generation Core™
- GFX
- DDR3
- DDR3
- High BW MMIO Writes
- But .. Low BW MMIO Reads
- 7000 MB/s

**Fast Streaming Reads!**

- Streaming Load is 9X faster reading from WC Memory

- Core™
- FPGA
- 800 MB/s
- 7000 MB/s
- Fast Streaming Reads!
Agenda

• SSE overview.
• SSE to accelerate heavy computational applications.
• SSE to accelerate Video encoding and Image processing.
• SSE to provide Graphics building blocks.
• Streaming auxiliary instructions.
• Using the Compiler to generate SSE
Using compiler intrinsics to generate SSE code (1/3)

• Use Intel® c++ compiler 10.0 for SSE4 support.
• #include <smmintrin.h>
• Data Types:
  - __m128
    Vector type of 4 single precision floating point elements.
  - __m128i
    Vector type of 4 integer elements.
  - __m128d
    Vector type of 2 double precision floating point elements.
• Load/Store:
  - __mm_load_pd/ps/si128, __mm_loadu_pd/ps/si128
  - __mm_store_pd/ps/si128, __mm_storeu_pd/ps/si128
Using compiler intrinsics to generate SSE code (2/3)

• Casting examples – used for type safety no real instruction is executed.
  - __m128 _mm_castpd_ps(__m128d in);
  - __m128i _mm_castpd_si128(__m128d in);
  - __m128d _mm_castps_pd(__m128 in);
  - __m128i _mm_castps_si128(__m128 in);
  - __m128 _mm_castsi128_ps(__m128i in);
  - __m128d _mm_castsi128_pd(__m128i in);

• Type conversion – Instructions are generated to convert from one representation to the other
  - For example: {1.1, 1.0, 1.0, 1.0} ➔ {1, 1, 1, 1}
  - __m128d _mm_cvtepi32_pd(__m128i a);
  - __m128i _mm_cvtpd_epi32(__m128d a);
  - __m128 _mm_cvtepi32_ps(__m128i a);
  - __m128i _mm_cvtps_epi32(__m128 a);
  - __m128 _mm_cvtpd_ps(__m128d a);
  - __m128d _mm_cvtps_pd(__m128 a);
Using compiler intrinsics to generate SSE code (3/3)

- All the instructions have their intrinsic equivalent, example intrinsics:
  - Arithmetic:
    - extern __m128 _mm_add_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_sub_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_mul_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_div_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_sqrt_ps(__m128 _A);
    - extern __m128 _mm_min_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_max_ps(__m128 _A, __m128 _B);
  - Logical
    - extern __m128 _mm_and_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_andnot_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_or_ps(__m128 _A, __m128 _B);
    - extern __m128 _mm_xor_ps(__m128 _A, __m128 _B);
  - Comparison
    - extern __m128 _mm_cmpeq_ps(__m128 _A, __m128 _B);
Intel® compiler Automatically generate SSE instructions for C code.

- Using Intel compiler on the following C code generates SIMD code automatically

```c
void foo (float* restrict a, float* restrict b, float* restrict c, int n){
    for (i = 0 ; i < n; i++){
        a[i] = b[i]*c[i];
    }
}
```

```
$B4$21:
    movups  xmm1, XMMWORD PTR [rdx+r10*4]  
movups  xmm0, XMMWORD PTR [r8+r10*4]  
mulps   xmm1, xmm0  
movaps  XMMWORD PTR [rcx+r10*4], xmm1  
add     r10, 4  
cmp    r10, rbp  
jl      $B4$21
```
Compiler switches for SSE4

• To Auto-generate SSE4 instructions Intel® compiler use:

/Qxs or /Qaxs
(-xs or –axs on Linux)
Using Pragmas to hint vectorization (1/3)

• `#pragma vector always`
  - instructs the compiler to override any efficiency heuristic during the decision to vectorize or not.
  - Will vectorize non-unit strides or very unaligned memory accesses.
  - **No correctness issues.**

• **Example:**
  ```c
  void vec_always(int *a, int *b, int m){
    #pragma vector always
    for(int i = 0; i <= m; i++) a[32*i] = b[99*i];
  }
  ```
Using Pragmas to hint vectorization (2/3)

• #pragma ivdep
  - Instructs the compiler to ignore assumed vector dependencies.
  - Only use this when you know that the assumed loop dependencies are safe to ignore.

• Example:
  
  ```c
  void ignore_vec_dep(int *a, int k, int c, int m){
    #pragma ivdep
    for (int i = 0; i < m; i++) a[i] = a[i + k] * c;
  }
  ```
  
  - The pragma binds only the for loop contained in current function. This includes a for loop contained in a subfunction called by the current function.
  - The loop in this example will not vectorize without the ivdep pragma, since the value of k is not known; vectorization would be illegal if k<0.
Using Pragmas to hint vectorization (3/3)

• `#pragma vector {aligned | unaligned}`
  - The pragma indicates that the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability.

• **Caution:**
  - The compiler may generate incorrect code if the information is wrong.

• **Example**
  ```c
  void vec_aligned(float *a, int m, int c)
  {
    int i;
    #pragma vector aligned
    for (i = 0; i < m; i++)
      a[i] = b[i] * c;
    // Alignment unknown but compiler can still align.
    for (i = 0; i < 100; i++)
      a[i] = a[i] + 1.0f;
  }
  ```
Thank You!