How To Optimize Your Software For The Upcoming Intel® Advanced Vector Extensions (Intel® AVX)

Xiaochang Wu  Sr. Software Engineer
Henry Ou  Sr. Software Engineer
Software and Services Group

ARCS003
Agenda

• Setting the Pace for Intel Instruction Set
• Next generation Intel® Core™ processors (codename Sandy Bridge) features – Intel® Advanced Vector Extensions (Intel® AVX)
• Software Development and Tools
• Tuning Tips and Kernels
• Update on FMA
• Summary and Call to Action
Setting the Pace for Intel® Instruction Set

Next:
Leapfrog with wide vectorization, ISA extensions: scalable performance & excellent power efficiency

Now:
Improved upcoming Intel® microarchitectures: ~15% gain/year

- **Nehalem**
  - Intel® SSE4
  - Memory latency, BW
  - Fast Unaligned support

- **Westmere**
  - AES-NI
  - Cryptographic Acceleration

- **Sandy Bridge**
  - Intel® AVX
    - 2X FP Throughput
    - 2X Load Throughput
    - 3-Operand instructions

Future Extensions
- Hardware FMA
- Memory Latency/BW
- Many Other Features


All timeframes, dates and products are subject to change without further notification
Agenda

• Setting the Pace for Intel Instruction Set
• Next generation Intel® Core™ processors (codename Sandy Bridge) features – Intel® Advanced Vector Extensions (Intel® AVX)
  – New Instructions Capabilities
  – A New Instruction Format
  – Examples
• Software Development and Tools
• Tuning Tips and Kernels
• Update on FMA
• Summary and Call to Action
### Key Intel® Advanced Vector Extensions (Intel® AVX) Features

<table>
<thead>
<tr>
<th><strong>KEY FEATURES</strong></th>
<th><strong>BENEFITS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Wider Vectors</td>
<td>• Up to 2x peak FLOPs (floating point operations per second) output with good power efficiency</td>
</tr>
<tr>
<td>- Increased from 128 to 256 bit</td>
<td></td>
</tr>
<tr>
<td>- Two 128-bit load ports</td>
<td></td>
</tr>
<tr>
<td>Enhanced Data Rearrangement</td>
<td>• Organize, access and pull only necessary data more quickly and efficiently</td>
</tr>
<tr>
<td>- Use the new 256 bit primitives to broadcast, mask loads and permute data</td>
<td></td>
</tr>
<tr>
<td>Three and four Operands</td>
<td>• Fewer register copies, better register use for both vector and scalar code</td>
</tr>
<tr>
<td>- Non Destructive Syntax for both AVX 128 and AVX 256</td>
<td></td>
</tr>
<tr>
<td>Flexible unaligned memory access support</td>
<td>• More opportunities to fuse load and compute operations</td>
</tr>
<tr>
<td>Extensible new opcode (VEX)</td>
<td>• Code size reduction</td>
</tr>
</tbody>
</table>

*Intel® AVX is a general purpose architecture, expected to supplant SSE in all applications used today*
Intel® Microarchitecture (Sandy Bridge) Highlights

Instruction Fetch & Decode ➔ Allocate/Rename/Retire ➔ Zeroing Idioms

Scheduler (Port names as used by IACA)

Port 0 ➔ Port 1 ➔ Port 5 ➔ Port 2 ➔ Port 3 ➔ Port 4

Port 0: ALU, VI MUL, SSE MUL, DIV *
Port 1: ALU, VI ADD, SSE ADD
Port 5: ALU, JMP
Port 2: AVX/FP Shuf, AVX/FP Bool
Port 3: Load, Store Address
Port 4: STD

Memory Control

L1 Data Cache

1-per-cycle 256-bit multiply, add, and shuffle
Load double the data with Intel microarchitecture (Sandy Bridge) !!!

* Not fully pipelined
Intel® Advanced Vector Extensions (Intel® AVX) 2X Vector Width
A 256-bit vector extension to SSE

• Intel® AVX extends all 16 XMM registers to 256bits

• Intel AVX works on either
  – The whole 256-bits – for FP instructions
  – The lower 128-bits (like existing SSE instructions)
    • A drop-in replacement for all existing scalar/128-bit SSE instructions
    • The upper part of the register is zeroed out

• Intel AVX targets high-performance
  – 256-bit Multiply, Add and Shuffle engines (2X of today)
  – 2nd load port
Many New Primitives Simplify FP Vectorization

- **Simple in-lane instructions**
  - 2 lanes, 128 bit each

  \[ \text{vAddPS dest, src1, src2} \]

- **New in-lane PS and PD Permuters**
  - Permute controlled via immediate

  \[ \text{vPermilPS dest, src, imm} \]

- **New 128-bit permutes**
  - Useful for lane-crossing operations

  \[ \text{vPerm2F128 dest, src1, src2, imm} \]
Many New Primitives Simplify FP Vectorization (2)

- **128-bit Insertions and extractions**
  - Useful for lane crossing operations
  - `vInsertF128 dest, src, imm`
  - `vExtractF128 dest, src, imm`

- **New Broadcast (SP, DP, 128-bit)**
  - Efficient Vector * Scalar operations
  - `vBroadcastPS dest, mem32`

- **New Conditional SIMD Loads and Stores**
  - Avoid page faults, segment violations, memory transaction if the mask is 0
  - Allow more automatic compiler vectorization
  - `vMaskMovPD dest, mask, mem256`
A Complete Masking Architecture

- In 2007, Intel® SSE4 introduced blending using controls using XMM0 and SIMD tests to quickly branch over infrequent code.
- Intel® Advanced Vector Extensions (Intel® AVX) expands this to enable a fully comprehensive SIMD masking architecture.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBLENDVDB xmm1, xmm2, [rax], xmm11</td>
<td>Any vector registers can be used to select elements from two source operands</td>
</tr>
<tr>
<td>VMASKMOVPS xmm1, xmm12, [rax]</td>
<td>Loads floats or 0 based on xmm12 (mask register). If mask is 0, no load side effects</td>
</tr>
<tr>
<td>VMASKMOVPS [rax], xmm1, xmm14</td>
<td>Store floats (or preserve) based on xmm1 (mask register)</td>
</tr>
<tr>
<td>VTESTPS xmm1, xmm13</td>
<td>AND sign bits and set ZF based on result. ANDN sign bits and set CF based on result</td>
</tr>
</tbody>
</table>
Conditional SIMD Loads and Stores

- Fault-free operation when mask zero
- Masked load returns “0” in masked elements
- Masked store “merges” unmasked elements into memory

```
vmaskmovpd  ymm0, ymm1, mem
vmaskmovpd  mem, ymm2, ymm0
```
A New 3- and 4- Operand Instruction Format

- Intel® Advanced Vector Extensions (Intel® AVX) has a distinct destination argument that results in fewer register copies, better register use, more load/op macro-fusion opportunities, and smaller code size.

<table>
<thead>
<tr>
<th>xmm10 = xmm9 + xmm1</th>
<th>vaddpd xmm10, xmm9, xmm1</th>
</tr>
</thead>
<tbody>
<tr>
<td>movaps xmm10, xmm9</td>
<td></td>
</tr>
<tr>
<td>addpd xmm10, xmm1</td>
<td></td>
</tr>
<tr>
<td>xmm10 = xmm9 + m128</td>
<td>vaddpd xmm10, xmm9, m128</td>
</tr>
<tr>
<td>movups xmm10, m128</td>
<td></td>
</tr>
<tr>
<td>addpd xmm10, xmm9</td>
<td></td>
</tr>
</tbody>
</table>
A New 3- and 4-Operand Instruction Format

- Intel® Advanced Vector Extensions (Intel® AVX) has a distinct destination argument that results in fewer register copies, better register use, more load/op macro-fusion opportunities, and smaller code size.

\[
\begin{align*}
\text{xmm10} &= \text{xmm9} + \text{xmm1} \\
\text{movaps} \text{ xmm10, xmm9} &\quad \text{vaddpd} \text{ xmm10, xmm9, xmm1} \\
\text{addpd} \text{ xmm10, xmm1} &
\end{align*}
\]

\[
\begin{align*}
\text{xmm10} &= \text{xmm9} + \text{m128} \\
\text{movups} \text{ xmm10, m128} &\quad \text{vaddpd} \text{ xmm10, xmm9, xmm1} \\
\text{addpd} \text{ xmm10, xmm9} &
\end{align*}
\]

- New 4-operand Blends example, implicit xmm0 not longer needed.

\[
\begin{align*}
\text{movaps} \text{ xmm0, xmm4} &
\end{align*}
\]

\[
\begin{align*}
\text{movaps} \text{ xmm1, xmm2} &\quad \text{vblendvps} \text{ xmm1, xmm2, m128, xmm4} \\
\text{blendvps} \text{ xmm1, m128} &
\end{align*}
\]
Agenda

• Setting the Pace for Intel Instruction Set
• Next generation Intel® Core™ processors (codename Sandy Bridge) features – Intel® Advanced Vector Extensions (Intel® AVX)

• Software Development and Tools
  – Tools
  – Development Methodology
  – Mixing SSE/AVX code BKMs

• Tuning Tips and Kernels
• Update on FMA
• Summary and Call to Action
## Intel® Advanced Vector Extensions Software Development Tools


<table>
<thead>
<tr>
<th>Tool</th>
<th>Release Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Compiler Suite Professional Edition 11.1</td>
<td>Version 11.1 for Windows and Linux</td>
</tr>
<tr>
<td>- Intel® Integrated Performance Primitives</td>
<td>Version 6.1</td>
</tr>
<tr>
<td>- Intel® Math Kernel Library</td>
<td>Version 10.2</td>
</tr>
<tr>
<td>- Intel® Threading Building Blocks</td>
<td>Version 2.2</td>
</tr>
<tr>
<td>- Intel® Debugger for Linux (IDB)*</td>
<td>IDB Info</td>
</tr>
<tr>
<td>- Intel® Parallel Debugger Extension*</td>
<td>Debugger Info (Available by end of 2009)</td>
</tr>
</tbody>
</table>

*Requires SDE pre-silicon
# Intel® Advanced Vector Extensions Software Development Tools


<table>
<thead>
<tr>
<th>Tool</th>
<th>Release Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Software Development Emulator</td>
<td>SDE</td>
</tr>
<tr>
<td>Intel® Architecture Code Analyzer</td>
<td>Intel® Architecture Code Analyzer</td>
</tr>
<tr>
<td>Intel® Intrinsics Guide</td>
<td>Intrinsics Guide</td>
</tr>
</tbody>
</table>

*Requires SDE pre-silicon*
# Third Party Intel® Advanced Vector Extensions Software Development Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Release Info</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microsoft Visual Studio® 2010 Beta 2</strong></td>
<td><em>Contact Microsoft for Availability</em></td>
</tr>
<tr>
<td>GNU Tools (For Linux*)</td>
<td><em>GNU</em></td>
</tr>
<tr>
<td>− Gcc</td>
<td><em>Version 4.4.1</em></td>
</tr>
<tr>
<td>− gdb (requires SDE pre-silicon)</td>
<td><em>Version 6.8.50.20090915</em></td>
</tr>
<tr>
<td>Binutils</td>
<td><em>2.20.51.0.1</em></td>
</tr>
<tr>
<td>− Disassembler (objdump)</td>
<td><em>Packaged with Binutils</em></td>
</tr>
<tr>
<td>Sun Studio*</td>
<td><em>Contact Sun for Availability</em></td>
</tr>
</tbody>
</table>
Microsoft Visual Studio* 2010

• Integrated Development Environment
  – Visual C/C++ native compiler
    • Supports Intel® Advanced Vector Extensions (Intel® AVX) encodings and optimizations
    • Intrinsics and 32-bit inline assembly
    • New compiler flag: /arch:AVX
  – MASM
    • Supports 32 and 64-bit Intel AVX instructions
  – Disassembler
    • Comprehends Intel AVX instructions
  – Visual Studio Debugger
    • Ymm register state, disassembly view, breakpoints, etc.

Start developing Intel AVX targeted apps in Visual Studio 2010 Beta 2

Other names and brands may be claimed as the property of others
Sun Studio* Software
C/C++/Fortran Tooling for the Multi-core Era

**Parallelism** – feature-rich toolchain (auto-parallelizing compilers, thread analysis / debugging / profiling, OpenMP support, ...) & MPI support via Sun HPC ClusterTools

**Performance** – dozens of industry benchmark records in the past year, including 12 on Intel Nehalem-based systems (int, fp, OpenMP)

**Productivity** – Next-gen IDE, code & memory debuggers, application profiler

**Platforms** – Simplified dev across architectures & OSs (Solaris OS, OpenSolaris OS, Linux)

Other names and brands may be claimed as the property of others
Intel® Advanced Vector Extensions (Intel® AVX) Software Development

Source Code

Intel Compiler

Obj File

Linker

Executable

Intel Architecture Code Analyzer

icl /QxAVX filename

IPP

MKL

Emulator

Or

Hardware

Analyze and Tune

Start developing with Intel AVX tools now!!

Emulator Usage:
sde -- binary name
Intel® C++ Compiler

- Intel® Advanced Vector Extensions (Intel® AVX) processor optimization switch
  - Windows* /QxAVX
  - Linux* -xAVX

- Recompiling legacy C/SSE code without code changes, with Intel AVX opt flag
  - C code – Generates Intel AVX 256-bit code
  - SSE code – Generates Intel AVX 128-bit code
  - Improves autovectorization and generation of Intel AVX instructions from C/C++
    - Compiler takes advantage of 256-bit registers and Intel AVX Instruction Set
    - vmaskmov enables new autovectorization capabilities
  - Performs Intel® microarchitecture (Sandy Bridge) specific optimizations
  - compiler generates VEX prefixed Intel AVX 128-bit code
    - e.g. vaddps xmm0, xmm0, xmm1
    - vs. addps xmm0, xmm0, xmm1
**Example of Intel® AVX Intrinsics**

- **New data types**
  - __m256, __m256d, __m256i
  - 256 bits in size, gets mapped to ymm registers

```
#include <immintrin.h>
void foo(float *a, float *b, float *r) {
    __m256 s1, s2, res;
    s1 = _mm256_loadu_ps(a);
    s2 = _mm256_loadu_ps(b);
    res = _mm256_add_ps(s1, s2);
    _mm256_storeu_ps(r, res);
}
```

```
; -- Begin _foo
; mark_begin;
    ALIGN    16
    PUBLIC _foo
_foo       PROC NEAR
    ; parameter 1: 4 + esp
    ; parameter 2: 8 + esp
    ; parameter 3: 12 + esp
$B2$1:                          ; Preds $B2$0
        mov     eax, DWORD PTR [4+esp]
        mov     edx, DWORD PTR [8+esp]
        mov     ecx, DWORD PTR [12+esp]
        vmovups  ymm0, YMMWORD PTR [eax]
        vaddps   ymm1, ymm0, YMMWORD PTR [edx]
        vmovups  YMMWORD PTR [ecx], ymm1
$B2$2:                          ; Preds $B2$1
        vzeroupper
    ret                                                     ;10.1
$B2$3:                          ; Preds $B2$2
        mov     eax, DWORD PTR [4+esp]
        mov     edx, DWORD PTR [8+esp]
        mov     ecx, DWORD PTR [12+esp]
        vmovups  ymm0, YMMWORD PTR [eax]
        vaddps   ymm1, ymm0, YMMWORD PTR [edx]
        vmovups  YMMWORD PTR [ecx], ymm1
    vzeroupper
        ret

; mark_end;
_foo ENDP
_foo ENDS
```

```
__m256, __m256d, __m256i
-- 256 bits in size, gets mapped to ymm registers
```

- New data types
- New Include File
- 256-bit variables
- Add 256bits
- Load & Store double the data
- Load & Store double the data
- icl /QxAVX <filename>
Example of Intel® AVX GNU Inline asm

```c
#include <immintrin.h>
void foo(const float *a, const float *b, float *r)
{
    asm ( "vaddps %1, %2, %0"
         : "=x" (*((__m256*) r))
         : "x" (*((__m256*) a)), "x" (*((__m256*) b)));
}
```

```
# -- Begin foo
# mark_begin;
    .align 16,0x90
    .globl foo
foo:
# parameter 1: 4 + %esp
# parameter 2: 8 + %esp
# parameter 3: 12 + %esp
..B1.1:                         # Preds ..B1.0
    movl 4(%esp), %eax
    movl 8(%esp), %edx
    movl 12(%esp), %ecx
    vmovaps (%eax), %ymm0
    vmovaps (%edx), %ymm1
    vaddps %ymm0, %ymm1, %ymm2
    vmovaps %ymm2, (%ecx)
    ret
    .align 16,0x90
    # LOE
# mark_end;
    .type foo,@function
    .size foo,-foo
    .data
# -- End foo
    .data
    .section .note.GNU-stack, ""
# End
```

icc -xAVX <filename>
Example of Intel® AVX Microsoft Inline asm

```c
void foo(const float *a, const float *b, float *r)
{
    __asm {
        mov eax, a
        vmovaps ymm0, ymmword ptr [eax]
        mov eax, b
        vmovaps ymm1, ymmword ptr [eax]
        vaddps ymm2, ymm1, ymm0
        mov eax, r
        vmovaps ymmword ptr [eax], ymm2
    }
}
```

icl /QxAVX <filename>
Accessing Lower 128-bit with Cast Intrinsic

- Lower 128-bits of YMM aliased with corresponding XMM
- Convert 256-bit to 128-bit with `_mm256_castps256_ps128`

```c
__m256 ymm0;
__m128 xmm1;
xmm1 = _mm256_castps256_ps128(ymmm0);
_mm_storeu_ps(pDest, xmm1)
```
More Intel® AVX Cast Intrinsics

• Without Extra Moves
  – From 256-bit to 128-bit
    • __m128  _mm256_castps256_ps128(__m256 a);
    • __m128d _mm256_castpd256_pd128(__m256d a);
    • __m128i _mm256_castsi256_si128(__m256i a);

  – From 128-bit to 256-bit
    • __m256  _mm256_castps128_ps256(__m128 a);
    • __m256d _mm256_castpd128_pd256(__m128d a);
    • __m256i _mm256_castsi128_si256(__m128i a);
    • Upper 128-bits are undefined

• Exactly same bit pattern re-interpreted
  – No real data conversion
  – Use _mm256_cvt intrinsics if need to convert float/double/int
    • __m256  _mm256_castpd_ps(__m256d a);
    • __m256d _mm256_castps_pd(__m256 a);
    • __m256i _mm256_castps_si256(__m256 a);
    • __m256i _mm256_castpd_si256(__m256d a);
    • __m256  _mm256_castsi256_ps(__m256i a);
    • __m256d _mm256_castsi256_pd(__m256i a);
# define N 1000
double pairs[N][2];
double X[N];
void foo(){
    int i;
    for(i=0; i<N; i++){
        // store to every other double
        pairs[i][0] = (double) i;
    }
    for(i=0; i<N; i++){
        // store to every other double
        pairs[i][1] = X[i];
    }
}

icl /QxAVX <filename>
Intel® AVX Short Vector Math Library

- Assembly version of vector variant used by Intel® C++ Compiler Auto-vectorizer since Version 8.0

```assembly
float angles[N];
float results[N];
for(int i = 0; i < N; i++)
{
    results[i] = sin(angles[i]) + cos(angles[i]);
}

B1.5: ; Preds .B1.11 .B1.4
    vmovaps ymm0, YMMWORD PTR [?angles@@3PAMA+esi*4] ;23.16
    call ___svml_sincosf8 ;23.16
    ; LOE ebx esi edi ymm0 ymm1

.B1.11: ; Preds .B1.5
    vaddps ymm2, ymm0, ymm1 ;23.33
    vmovaps YMMWORD PTR [?results@@3PAMA+esi*4], ymm2 ;23.3
    add esi, 8 ;22.2
    cmp esi, 256 ;22.2
    jb .B1.5 ; Prob 99% ;22.2
    ; LOE ebx esi edi
```

icl /QxAVX <filename>
Intel® AVX Short Vector Math Library

- Assembly version of vector variant used by Intel® C++ Compiler Autovectorizer since Version 8.0
- Version 11.1 introduces 128-bit and 256-bit intrinsics

```c
float angles[N];
float results[N];
__m256 vSin;
__m256 vCos;
for(int i = 0; i < N; i+=8){
    vSin = _mm256_sincos_ps(&vCos,
                           _mm256_loadu_ps(&angles[i]));
    _mm256_storeu_ps(&results[i],
                     _mm256_add_ps(vSin, vCos));
}
```

icl /QxAVX <filename>

*Intel® Compiler 11.1 Autovectorizer uses Intel® Advanced Vector Extensions (Intel® AVX) enabled SVML, Introduces new Intel AVX SVML Intrinsics*
Processor-Specific Code Versions

```c
__declspec(cpu_specific(future_cpu_16))
void foo(float *a, float *b, float *r)
{
    __m256 aa, bb, rr;
    aa = _mm256_loadu_ps(a);
    bb = _mm256_loadu_ps(b);
    rr = _mm256_add_ps(aa, bb);
    _mm256_storeu_ps(r, rr);
}
```

**Intel® microarchitecture (Sandy Bridge) specific code**

```c
__declspec(cpu_specific(generic))
void foo(float *a, float *b, float *r)
{
    int i;
    for (i = 0; i < 8; i++) {
        r[i] = a[i] + b[i];
    }
}
```

**Generic code**

```c
__declspec(cpu_dispatch(future_cpu_16, generic))
void foo(float *a, float *b, float *r)
{
    /* empty */
}
```

**Dispatch function**

Other values:
- future_cpu_15: Westmere
- core_i7_sse4_2: Nehalem
- core_2_duo_sse4_1: 45nm Intel® Core™ 2 Duo Processor

Caller: foo(x, y, z);
Intel® AVX ABI Extensions

- __m256, __m256i and __m256d variables will be aligned to a 0 mod 32 byte address
- Function calls
  - Windows32, Linux32
    - ymm0-7 will be caller-save
    - ymm0-2 will be used for parameters/return of type __m128[i,d] and __m256[i,d].
  - Linux64
    - ymm0-15 will be caller-save
    - ymm0-7 being used for parameters/return of type __m128[i,d] and __m256[i,d]
  - Windows64
    - ymm0-5 are caller save
    - lower halves of ymm6-15 (xmm6-15) are callee save
    - upper halves of ymm6-15 are caller save
    - ymm0-3 will be used for parameters/return for variables of type float and double
    - parameters of type __m128[i,d] and __m256[i,d] will continue to be passed by reference.

**New architecture state is caller save – hence the proposed ABI extends current ABIs naturally and is backward compatible to older processors**
Intel® Integrated Performance Primitives (Intel® IPP)

- **Intel® IPP Website**
- Version 6.1 update 1 (current release)
  - More than 100 IPP functions already optimized for Intel® Advanced Vector Extensions (Intel® AVX)
  - FFT, Filtering, Convolution, Correlation, Resizing
- Version 6.1 update 2 (Q4 2009) onwards – additional functions optimized
- Intel® AVX optimization Intel IPP® whitepapers
  - *Intel AVX realization of IIR filter for complex float data*
  - *Intel AVX realization of Lanczos interpolation in Intel IPP 2D resize transform*

**Intel IPP is optimized for Intel AVX**
Intel® Math Kernel Libraries (Intel® MKL)

- Intel® MKL is the flagship for High Performance Computing (Intel MKL Website)
  - Highly optimized, thread-safe math routines
  - Provides automatic parallelization and scaling for multi and many core
  - Includes BLAS, LAPACK, FFTs, Sparse Solvers, Vector Math and Statistical functions and more ...
  - Compiler independent where possible
  - Provides processor specific optimizations
  - Supports C++ and Fortran
  - Windows*, Linux*

- Intel® Advanced Vector Extensions (Intel® AVX) Optimizations released with Intel MKL 10.2 (June 2009) for early application enabling
  - Basic Linear Algebra Subroutines (BLAS) and FFTs
  - Simulation Performance
    - DGEMM – double precision general matrix-matrix multiplication
      - $C = \alpha A*B + \beta C$, where $A$, $B$, and $C$ are matrices, $\alpha$ and $\beta$ are scalars coefficients
    - $C = C + A*B$
      - 1.9x (estimate based on simulation) speedup compared to Intel SSE2

- Intel AVX optimization Intel MKL whitepaper
  - Optimize for Intel® AVX Using Intel® Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine
Intel® AVX Software Dev. Emulator

Running the basic emulator –

\texttt{sde -- foo.exe <foo.options>}

- For ease of use
  - Special command window where every command is run on the emulator
- \% sde \-help

Usage: sde [args] -- application [application-args]

\begin{tabular}{|l|l|}
  \hline
  mix & run mix histogram tool* \\
  \hline
  debugtrace & run mix debugtrace tool* \\
  \hline
  ast & run the AVX/SSE transition checker* \\
  \hline
  no-avx & disable AVX emulation \\
  \hline
\end{tabular}

*prefix with “o” to specify output file
Using Intel® SDE to Count Types of Instructions

```
sde -mix -- mm_256_cmpouunord_ps.opt.vec.exe

## $global-dynamic-counts
#  opcode          count
#
  6 ADD             3092
  19 AND            2694
  38 CALL_NEAR      1739
...
  907 XCHG          1
  908 XGETBV        1
  910 XOR           4981
...
  4052 *isa-ext-AVX 17
  4053 *isa-ext-BASE 147580
  4055 *isa-ext-MODE64 222
  4058 *isa-ext-SSE 21
  4067 *isa-ext-XSAVE 1
  000000 *total     147841
```
Generating Intel® AVX Disassembly

xed* -i _mm256_cmpunord_ps.opt.vec.exe > dis

SYM subb:
XDIS 400a86: PUSH     BASE 55
XDIS 400a87: DATAXFER BASE 4889E5
XDIS 400a8a: LOGICAL BASE 4883E4E0
XDIS 400a8e: DATAXFER BASE B8FFFFFFFF
XDIS 400a93: DATAXFER BASE 89051F381000
XDIS 400a99: DATAXFER BASE 890525381000
XDIS 400a9f: AVX AVX C5FC100511381000 [rip+0x103811]
XDIS 400aa7: DATAXFER BASE 89053F381000
XDIS 400aad: DATAXFER BASE 890541381000
XDIS 400ab3: AVX AVX C5FCC20D1C38100003 [rip+0x10381c], 0x3
XDIS 400abc: AVX AVX C5FC110D34381000 ymm1
XDIS 400ac4: LOGICAL BASE 33C0
XDIS 400ac6: AVX AVX C5FA1080B8425000
XDIS 400ace: LOGICAL BASE 33D2

push rbp
mov rbp, rsp
and rsp, 0xe0
mov eax, 0xffffffff
mov dword ptr [rip+0x10381f], eax
mov dword ptr [rip+0x103825], eax
vmovups ymm0, ymmword ptr
mov dword ptr [rip+0x10383f], eax
mov dword ptr [rip+0x103841], eax
vcmppps ymm1, ymm0, ymmword ptr
vmovups ymmword ptr [rip+0x103834],
xor eax, eax
vmovss xmm0, dword ptr [rax+0x5042b8]
xor edx, edx

*xed is included with Intel® SDE
Note: GNU disassembler also supports Intel® AVX disassembly
Intel® Architecture Code Analyzer

Enables pre-silicon software developers to optimize their code when porting from Intel® SSE to Intel® AVX

Main Features

• Supports Intel® Advanced Vector Extensions (Intel® AVX) and Intel legacy instructions

• Analyzes a consecutive block of Intel® architecture 32-bit or Intel® 64 instructions

• Provides: uop port binding, throughput and latency in cycles, and instructions on longest dependency chain
  – Models 2*128bit load ports, 1*128bit store port, simple front end
  – Assumes: L1 cache hit and no other uArch dynamic penalties

• Command line tool, hosted on Windows* and soon on Linux*

• Accepts as input an executable, a DLL or an object file
  – Insert instruction block start (IACA_START) and end marks (IACA_END) to enable Intel Architecture Code Analyzer to capture the code of interest
  – Can be used in C/C++ and ASM code

• ASCII Output

Download from http://whatif.intel.com
> iaca -f matrix_multiply.exe
Analysis Report
---------------

Total Throughput: 4 Cycles  Throughput Bottleneck: Port 5
Total Latency:    12 Cycles Total number of Uops: 13

Port Binding in cycles:

<table>
<thead>
<tr>
<th>Port</th>
<th>0 - DV</th>
<th>1</th>
<th>2 - D</th>
<th>3 - D</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

N - port number or number of cycles port was bound, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
X - other ports that can be used by this instruction
F - Macro Fusion with the next instruction occurred
* - instruction micro-ops not bound to a port
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

<table>
<thead>
<tr>
<th>Num of</th>
<th>Ports pressure in cycles</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Uops</td>
<td>0 - DV</td>
<td>1</td>
</tr>
</tbody>
</table>
------------------------------------------------------------
| 1       |    |    |    |    |    |    |    |    |    |    | CP | vmovsd xmm0, qword ptr [rax+rbx*1] |
| 2       |    | X |    |    |    |    |    |    |    |    | CP | vunpcklqd xmm0, xmm0, xmmword ptr [rax+rbx*1+0x20] |
| 1       |    | 1 | X |    |    |    |    |    |    |    | CP | vmovsd xmm1, qword ptr [rax+rbx*1+0x40] |
| 2       |    |    |    |    |    |    |    |    |    |    | CP | vunpcklqd xmm1, xmm1, xmmword ptr [rax+rbx*1+0x60] |
| 1*      |    |    |    |    |    |    |    |    |    |    | CP | vinsertf128 ymm0, ymm0, ymm1, 0x1 |
| 1       |    |    |    |    |    |    |    |    |    |    | CP | vxorps ymm1, ymm1, ymm1 |
| 1       |    | 1 |    |    |    |    |    |    |    |    | CP | vmaxpd ymm1, ymm1, ymm0 |
| 2       |    | 1 |    |    |    |    |    |    |    |    | CP | vmovaps ymmword ptr [rcx+rbx*4], ymm1 |
| 1       |    |    |    |    |    |    |    |    |    |    | CP | add rbx, 0x8 |
| 0       |    |    |    |    |    |    |    |    |    |    | CP | cmp rbx, 0x20 |
| 1       |    |    |    |    |    |    |    |    |    |    | CP | jnz 0xffffffcc |

port 5 is the bottleneck
Actual port binding
Alternative port binding
Identifies instructions in critical path
not bound to a port
CMP & JNZ are macro-fused
Intel® AVX Debugger Support

- Intel® C++ Application Debugger For Linux* OS with Bi-Endian Technology
  - Packaged in Intel Compiler Suite (Future Release)
- Integrated Microsoft Visual Studio* 2010 Debugger
- Intel® Parallel Debugger Extension
  - For Windows*
  - Packaged in Intel Compiler Suite and Intel® Parallel Studio (Future Release)
- GDB
  - For Linux*

Support standard debugging features including
  - View Intel® Advanced Vector Extensions Disassembly
  - Examine registers including YMMs

All unreleased products, computer systems, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
GDB Debugger with Intel® SDE for Linux*

Refer to [SDE](#) how to run gdb with sde

YMM Register
Visual Studio* 2010 IDE (Pre-Release)
Debugging Visual C++* 2010 Built Intel® AVX Application

Leverage familiar Visual Studio* 2010 development/debug experience!
Intel® AVX/SSE Transitions

- Mixing Intel® SSE/AVX code may incur penalty
- Executing AVX 256 dirties upper 128 bits
  - Executing SSE incurs penalty (hw saves upper 128 bits)
  - Executing AVX incurs penalty (hw restores upper 128 bits)

1. movaps xmm0, [208 + esp]
2. vinsetf128 ymm1, ymm1 xmm0, 0
3. call sub_with_sse_code
   ...
4. movaps xmm1, [160 + esp]
5. movaps xmm2, [172 + esp]
   ...
6. ret
7. vinsetf128 ymm3, ymm3, xmm2, 1

*Returning to an Intel AVX friendly state via Intel AVX 128/256 including VZeroxxx instruction incurs penalty!
*State is saved by hardware internally
Mixing Legacy Intel® SSE and Intel® AVX Code

• Performance penalty for each transition to/from Intel® AVX to/from Intel® SSE

• Two strategies the compiler can use to avoid these penalties
  – No penalty in either direction if the upper 128 bits of all YMM’s are ‘zeroed’ before transitioning to Intel SSE
    – Before calling into any legacy Intel SSE routine from a routine using Intel AVX instructions
    – Before returning into any legacy Intel SSE routine from a routine using Intel AVX instructions
    – vZeroUpper zeros the upper 128 bits of all ymm vector registers
    – vZeroAll resets all the ymm vector registers to zero
  – There is no penalty if you use Intel AVX-128 bit instructions
    – Convert legacy Intel SSE code to Intel AVX-128 bit instructions.
    – Programs with Intel SSE intrinsics compiled under /QxAVX will automatically use Intel AVX-128 bit instructions.
    – Programs with inline Intel SSE asm compiled under /QxAVX will automatically use Intel AVX-128 bit instructions

• Legacy Intel SSE instructions preserve the value of the upper 128 bits
• 128-bit Intel AVX instructions will zero the upper 128 bits
Mixing Intel® AVX/SSE correctly

1. movaps xmm0, [208 + esp]
2. vinsertf128 ymm1, ymm1 xmm0, 0
3. VZEROUPPER
4. call sub_with_sse_code
   ... 
5. movaps xmm1, [160 + esp]
6. movaps xmm2, [172 + esp]
   ... 
7. ret
8. vinsertf128 ymm3, ymm3, xmm2, 1

Correct way to mix Intel SSE code inside Intel AVX 256 apps is to clean upper register state using VZEROxxxx

* State is saved by hardware internally
Mixing Intel® AVX/SSE correctly

1. movaps xmm0, [208 + esp]
2. vinsertf128 ymm1, ymm1 xmm0, 0
3. VZEROUPPER
4. call sub_with_sse_code

... Executing VZEROUPPER puts hw in clean state

5. movaps xmm1, [160 + esp]
6. movaps xmm2, [172 + esp]

... Executing SSE

7. ret
8. vinsertf128 ymm3, ymm3, xmm2, 1

Correct way to mix Intel SSE code inside Intel AVX 256 apps is to clean upper register state using VZEROxxxx
* State is saved by hardware internally

- Avoid Intel® AVX/SSE Transitions
- Re-Compile all code with /QxAVX flag
## Intel® AVX SSE Transition Checker

**sde –ast -- foo.exe**

<table>
<thead>
<tr>
<th>BlockPC</th>
<th>Dynamic AVX to SSE Transition</th>
<th>Dynamic SSE to AVX Transition</th>
<th>Static Icount</th>
<th>Executions</th>
<th>Dynamic Icount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>===</td>
<td>===</td>
<td>==</td>
<td>==</td>
<td>==</td>
</tr>
<tr>
<td># TID 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400993</td>
<td>1</td>
<td>0</td>
<td>16</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>4009f2</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>4009da</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>28</td>
</tr>
</tbody>
</table>

# SUMMARY

- AVX_to_SSE_transition_instances: 14
- SSE_to_AVX_transition_instances: 13
- Dynamic_insts: 147841
- AVX_to_SSE_instances/instruction: 0.0001
- SSE_to_AVX_instances/instruction: 0.0001
- AVX_to_SSE_instances/100instructions: 0.0095
- SSE_to_AVX_instances/100instructions: 0.0088
## Intel® AVX SSE Transition Checker

### Corresponding Disassembly code

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>ID Signature</th>
<th>Op Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4009b3</td>
<td>AVX</td>
<td>C5FC1005FD381000</td>
<td>vmovups ymm0, ymmword ptr [rip+0x1038fd]</td>
<td></td>
</tr>
<tr>
<td>4009bb</td>
<td>DATAFER</td>
<td>89052B391000</td>
<td>mov dword ptr [rip+0x10392b], eax</td>
<td></td>
</tr>
<tr>
<td>4009c1</td>
<td>DATAFER</td>
<td>89052D391000</td>
<td>mov dword ptr [rip+0x10392d], eax</td>
<td></td>
</tr>
<tr>
<td>4009c7</td>
<td>AVX</td>
<td>C5FCC20D0839100003</td>
<td>vcmpps ymm1, ymm0, ymmword ptr [rip+0x103908], 0x3</td>
<td></td>
</tr>
<tr>
<td>4009d0</td>
<td>AVX</td>
<td>C5FC110D20391000</td>
<td>vmovups ymmword ptr [rip+0x103920], ymm1</td>
<td></td>
</tr>
<tr>
<td>4009d8</td>
<td>LOGICAL</td>
<td>33C0</td>
<td>xor eax, eax</td>
<td></td>
</tr>
<tr>
<td>4009da</td>
<td>AVX</td>
<td>C5FA1080B8425000</td>
<td>vmovss xmm0, dword ptr [rax+0x5042b8]</td>
<td></td>
</tr>
<tr>
<td>4009e2</td>
<td>LOGICAL</td>
<td>33D2</td>
<td>xor edx, edx</td>
<td></td>
</tr>
<tr>
<td>4009e4</td>
<td>SSE</td>
<td>0F2EC0</td>
<td>ucomiss xmm0, xmm0</td>
<td></td>
</tr>
<tr>
<td>4009e7</td>
<td>COND_BR</td>
<td>7B05</td>
<td>jnp 0x4009ee</td>
<td></td>
</tr>
<tr>
<td>4009e9</td>
<td>DATAFER</td>
<td>BA01000000</td>
<td>mov edx, 0x1</td>
<td></td>
</tr>
<tr>
<td>4009ee</td>
<td>LOGICAL</td>
<td>85D2</td>
<td>test edx, edx</td>
<td></td>
</tr>
<tr>
<td>4009f0</td>
<td>COND_BR</td>
<td>7518</td>
<td>jnz 0x400a0a</td>
<td></td>
</tr>
<tr>
<td>4009f2</td>
<td>AVX</td>
<td>C5FA1080D8425000</td>
<td>vmovss xmm0, dword ptr [rax+0x5042d8]</td>
<td></td>
</tr>
<tr>
<td>4009fa</td>
<td>LOGICAL</td>
<td>33D2</td>
<td>xor edx, edx</td>
<td></td>
</tr>
<tr>
<td>4009fc</td>
<td>SSE</td>
<td>0F2EC0</td>
<td>ucomiss xmm0, xmm0</td>
<td></td>
</tr>
<tr>
<td>4009ff</td>
<td>COND_BR</td>
<td>7B05</td>
<td>jnp 0x400a06</td>
<td></td>
</tr>
<tr>
<td>400a01</td>
<td>DATAFER</td>
<td>BA01000000</td>
<td>mov edx, 0x1</td>
<td></td>
</tr>
</tbody>
</table>

Intel AVX/SSE Transitions

Intel® Advanced Vector Extensions; Intel® Streaming SIMD Extensions
Agenda

• Setting the Pace for Intel Instruction Set
• Next generation Intel® Core™ processors (codename Sandy Bridge) features – Intel® Advanced Vector Extensions (Intel® AVX)
• Software Development and Tools
  • Tuning Tips and Kernels
    – Matrix Addition
    – AoS to SoA
    – Matrix Transpose
    – Complex Multiply
• Update on FMA
• Summary and Call to Action
Intel® AVX Tuning Tips

- Memory
  - Loads/Stores limit scaling
  - Code dominated with compute intensive operations compared to memory operations produce higher Intel® AVX speedups
  - Minimize cache line splits (cross cache line access)
    - Align on 32 Byte boundary when possible
- Penalty to/from Intel® SSE
  - Zero out upper 128-bits via VZEROUPPER to avoid penalty
  - Use Intrinsics to increase productivity and opportunities to run in 64-bit mode
  - If still using assembly, port Intel SSE assembly to Intel AVX
  - Compile with –xAVX flag (auto generates 128-bit AVX), Autovectorize
- General purpose shuffler can be an overhead
  - Use other ways i.e. blend, extract, insert
  - Reduce port pressure
- Masked stores
  - Hoist loads high; determine mask early
Memory Recommendations Tuning Tips

• Loads and stores are the main cause of scaling limiters from Intel® Advanced Vector Extensions (Intel® AVX)

• Compared to Intel® Streaming SIMD Extensions (Intel® SSE) baselines, Intel AVX offers increased throughput to first-level cache only for situations with more loads than stores (up to 1.5x for 2:1 ratio of loads to stores with 256-bit loads and stores)
  – Don’t expect gains for code with a high ratio of only loads, or only stores, or an even ratio (like copy loops)
  – Combine simple copy loops with arithmetic to exploit the higher execute capabilities of Intel AVX
**Intel® AVX Data Alignment Tuning Tips**

- **Align Data to Vector Length**
- **Intel® SSE** - Align data to 16 Bytes, which is Intel® SSE vector length
- **Intel AVX (Intel® microarchitecture (Sandy Bridge))** - Align Data to 32 Bytes
  - Cache line length is 64 bytes
  - Intel AVX register length is 32 bytes
  - Unaligned data will cause every second load on consecutive memory accesses to be a cache line split.

---

**SNB 32B load/store**

- **Cache line 1**
  - 32B Load sequence when address aligned to 32B:
  - 32B Load sequence when address is not aligned to 32B:
  - **Cache line 2**
Intel® AVX Data Alignment Tuning Tips

- Consider 16 Byte loads when data is not aligned
- No Penalty for unaligned loads on aligned memory
- Aligning Stores is more important

Converting 32 Byte memory accesses

\[
\text{movups ymm, mem} \rightarrow \text{movups xmm, mem} \\
\text{vinsertf128 ymm, ymm, mem+16, 1}
\]

Converting 32 Byte memory accesses + op

\[
\text{vaddps ymm, mem} \rightarrow \text{vmovups xmm, mem} \\
\text{vinsertf128 ymm, ymm, mem+16, 1} \\
\text{vaddps ymm, ymm, ymm}
\]
**Intel® AVX Shuffler Tuning Tips**

- Shufflers can only execute on port 5
  - This is often a bottleneck in SIMD code
- Not all data manipulation instructions use port 5
- Consider using alternative to shuffles!
  - Immediate Blends can use either port 0 or 5
  - InsertF128 from memory turns into a blend
  - ExtractF128 to memory executes on the store port
  - Mov*dup and BroadcastSS (128-bit forms) execute on the load port
- Design algorithms with fewer shufflers
**Intel® AVX Masked Moves Tuning Tips**

**New Conditional SIMD Loads and Stores**
- Avoid page faults + segment violations + memory transaction if the mask is 0

**Masked Stores**
- Masked Stores have a dependence between the mask and the STA.
  - Non-masked stores don’t have this kind of dependence.
  - It means that disambiguation can be greatly delayed

- Hoist loads above masked stores
- Determine mask as early as possible
- It may be beneficial to not use masked stores for very small loops (< 30 iterations)
Optimizing Matrix Operations with Intel® AVX

Matrix Addition

For all Kernels: All speedup results are based on comparisons of SSE vs Intel® AVX on Sandy Bridge silicon
Matrix Addition: Intel® Streaming SIMD Extensions (Intel® SSE) Implementation

- 16-byte aligned 4x4 SPFP Matrix
- Add Completely Unrolled - 4 times

```c
xmm0 = _mm_load_ps(pIn1); // [m03,m02,m01,m00]
xmm1 = _mm_load_ps(pIn2); // [n03,n02,n01,n00]
xmm0 = _mm_add_ps(xmm0, xmm1); // [m03+n03,m02+n02,m01+n01,m00+n00]

xmm2 = _mm_load_ps(pIn1 + 4); // [m13,m12, mm11,m10]
xmm3 = _mm_load_ps(pIn2 + 4); // [n13, n12, n11,n10]
xmm2 = _mm_add_ps(xmm2, xmm3); // [m13,m12, mm11,m10]

xmm4 = _mm_load_ps(pIn1 + 8);
xmm5 = _mm_load_ps(pIn2 + 8);
xmm4 = _mm_add_ps(xmm4, xmm5);

xmm6 = _mm_load_ps(pIn1 + 12);
xmm7 = _mm_load_ps(pIn2 + 12);
xmm6 = _mm_add_ps(xmm6, xmm7);

_mm_store_ps(pOut, xmm0);
_mm_store_ps(pOut + 4, xmm2);
_mm_store_ps(pOut + 8, xmm4);
_mm_store_ps(pOut + 12, xmm6);
```

- Unrolled
- Loads hoisted above stores
4x4 Matrix Addition – Intel® AVX Implementation

- **Load four pairs of single precision floats**
  - 2 loads from 2 arrays of floats

  - `_mm256_load_ps(ymm0, inPtr1)`
  - `_mm256_load_ps(ymm1, inPtr2)`

  Repeat for rows 3 and 4

- **Add the elements**

  - `ymm1 = _mm256_add_ps(ymm1, ymm0)`

  Repeat for Rows 3 and 4
Matrix Addition: Intel® AVX Implementation

- 32-byte aligned 4x4 SPFP Matrix
- Add Completely Unrolled – 2 times

```
__m256 Ymm_A1 = _mm256_load_ps(pImage1);
__m256 Ymm_B1 = _mm256_load_ps(pImage2);
__m256 Ymm_C1 = _mm256_add_ps (Ymm_A1, Ymm_B1);

__m256 Ymm_A2 = _mm256_load_ps(pImage1 + 8);
__m256 Ymm_B2 = _mm256_load_ps(pImage2 + 8);
__m256 Ymm_C2 = _mm256_add_ps (Ymm_A2, Ymm_B2);

_mm256_store_ps(pOutImage + 8, Ymm_C1);
_mm256_store_ps(pOutImage + 8, Ymm_C2);

pImage1 += 16;
pImage2 += 16;
pOutImage += 16;
```

Speedup – 1.42x

With Intel® AVX Load and Add Twice the Data
Strategies for in-lane coding with Intel® AVX

• Application of Strided Load Technique to Convert Array of Structures to Structure of Arrays AoS

```c
struct complex {
    float a;
    float b;
};
struct complex Com_numbers[100];
```

SoA

```c
struct complex {
    float a[100];
    float b[100];
};
struct complex Com_numbers;
```
AoS to SoA

- Order the operands to optimize parallel operations
xor rbx, rbx
xor rdx, rdx
mov rcx, len
mov rdi, inPtr
mov rsi, outPtr1
mov rax, outPtr2

loop1:
    movups xmm0, [rdi+rbx]       // [i1 r1 i0 r0]
    movups xmm1, [rdi+rbx+16]   // [i3 r3 i2 r2]
    movaps xmm2, xmm0
    shufps xmm0, xmm1, 0xdd     // [i3 i2 i1 i0]
    shufps xmm2, xmm1, 0x88     // [r3 r2 r1 r0]
    movups [rax+rdx], xmm0
    movups [rsi+rdx], xmm2
    add rdx, 16
    add rbx, 32
    cmp rcx, rbx
    jnz loop1
AoS to SoA – Strided Load Method

“Strided load” method does not require “cross lane” shuffles
AoS to SoA – Intel® AVX Implementation (1)

• Load four pairs of complex numbers
  – 2 loads
  
  \[
  \text{vmovups xmm0, [mem]} \\
  \text{vmovups xmm1, [mem+16]}
  \]

• Load four more pairs
  – Aliasing benefit
  
  \[
  \text{vinsertf128 ymm0, ymm0, [mem+32], 1} \\
  \text{vinsertf128 ymm1, ymm1, [mem+48], 1}
  \]
• Separate the components

\[ \text{vshufps ymm2, ymm0, ymm1, 0xdd} \]

\[ \text{vshufps ymm3, ymm0, ymm1, 0x88} \]
AoS to SoA – Intel® AVX Implementation

Speedup – 1.84x

rcx - Buffer Length; rdi – Input Buffer; rsi,rax – Output Buffers

loop1:

vmovups xmm0, [rdi+rbx]
vmovups xmm1, [rdi+rbx+16]

vinsertf128 ymm0, ymm0, [rdi+rbx+32] , 1
vinsertf128 ymm1, ymm1, [rdi+rbx+48] , 1

vshufps ymm2, ymm0, ymm1, 0xdd
vshufps ymm3, ymm0, ymm1, 0x88
vmovups [rax+rdx], ymm2
vmovups [rsi+rdx], ymm3
add rdx, 32
add rbx, 64
cmp rcx, rbx
jnz loop1
Matrix Transpose with Intel® AVX

Matrix Transpose

<table>
<thead>
<tr>
<th>A00</th>
<th>A01</th>
<th>A02</th>
<th>A03</th>
<th>A04</th>
<th>A05</th>
<th>A06</th>
<th>A07</th>
</tr>
</thead>
<tbody>
<tr>
<td>A10</td>
<td>A11</td>
<td>A12</td>
<td>A13</td>
<td>A14</td>
<td>A15</td>
<td>A16</td>
<td>A17</td>
</tr>
<tr>
<td>A20</td>
<td>A21</td>
<td>A22</td>
<td>A23</td>
<td>A24</td>
<td>A25</td>
<td>A26</td>
<td>A27</td>
</tr>
<tr>
<td>A30</td>
<td>A31</td>
<td>A32</td>
<td>A33</td>
<td>A34</td>
<td>A35</td>
<td>A36</td>
<td>A37</td>
</tr>
<tr>
<td>A40</td>
<td>A41</td>
<td>A42</td>
<td>A43</td>
<td>A44</td>
<td>A45</td>
<td>A46</td>
<td>A47</td>
</tr>
<tr>
<td>A50</td>
<td>A51</td>
<td>A52</td>
<td>A53</td>
<td>A54</td>
<td>A55</td>
<td>A56</td>
<td>A57</td>
</tr>
<tr>
<td>A60</td>
<td>A61</td>
<td>A62</td>
<td>A63</td>
<td>A64</td>
<td>A65</td>
<td>A66</td>
<td>A67</td>
</tr>
<tr>
<td>A70</td>
<td>A71</td>
<td>A72</td>
<td>A73</td>
<td>A74</td>
<td>A75</td>
<td>A76</td>
<td>A77</td>
</tr>
</tbody>
</table>

Matrix A ➔ Transpose Matrix A
Matrix Transpose – Intel® SSE Implementation (1)

movaps xmm0, [rbx] // [a03,a02,a01,a00]
movaps xmm10, xmm0
movaps xmm1, [rbx+1*rax] // [a13,a12,a11,a10]
movaps xmm2, [rbx+2*rax] // [a23,a22,a21,a20]
movaps xmm12, xmm2
movaps xmm3, [rbx+1*rsi] // [a33,a32,a31,a30]
unpcklps xmm0, xmm1 // [a11,a01,a10,a00]
unpckhps xmm10, xmm1 // [a13,a03,a12,a02]
movaps xmm1, xmm10
unpcklps xmm2, xmm3 // [a31,a21,a30,a20]
unpckhps xmm12, xmm3
movaps xmm3, xmm2
movhlp xmm2, xmm0 // [a31,a21,a11,a01]
movhlps xmm0, xmm3 // [a30,a20,a10,a00]
movlhps xmm1, xmm12
movhlps xmm12, xmm10
movaps [rdx], xmm0 // store[a30,a20,a10,a00]
movaps [rdx+1*rax], xmm2
movaps [rdx+2*rax], xmm1
movaps [rdx+1*rsi], xmm12

Load four floats from Row 0 Col 0-3
Load four floats from Rows 1, 2, and 3
Combine Cols 0 & 1 of Rows 0 & 1
Combine Cols 2 & 3 of Rows 0 & 1
Repeat for Rows 2 & 3
Combine all Cols 1
Repeat for Cols 0, 2, and 3

A 4x4 block has been transposed, similar operations for other 4x4 blocks

Intel® Streaming SIMD Extensions
## Matrix Transpose – Intel® AVX Implementation

```assembly
vmovaps ymm8, [rcx]  // [a07,a06,a05,a04, a03,a02,a01,a00]
vmovaps ymm9, [rcx+r8*4]  // [a47,a46,a45,a44, a43,a42,a41,a40]
vperm2f128 ymm0, ymm8, ymm9, 0x20  // 0x20 = 00 10 00 00
    // ymm0 = [a43,a42,a41,a40, a03,a02,a01,a00]
vperm2f128 ymm4, ymm8, ymm9, 0x31  // 0x31 = 00 11 00 01
    // ymm4 = [a47,a46,a45,a44, a07,a06,a05,a04]
vmovaps ymm8, [rcx+r8*1]  // [a17,a16,a15,a14,a13,a12,a11,a10]
vmovaps ymm9, [rcx+rdi*1]  // [a57,a56,a55,a54,a53,a52,a51,a50]
vperm2f128 ymm1, ymm8, ymm9, 0x20
    // ymm1 = [a53,a52,a51,a50, a13,a12,a11,a10]
vperm2f128 ymm5, ymm8, ymm9, 0x31
    // ymm5 = [a57,a56,a55,a54, a17,a16,a15,a14]
vunpcklqd ymm2, ymm0, ymm1
    // ymm2 = [a51,a50,a41,a40, a11,a10,a01,a00]
vunpckhqd ymm10, ymm0, ymm1
vunpcklqd ymm3, ymm4, ymm5
vunpckhqd ymm11, ymm4, ymm5
    // ymm11 = [a47,a46,a57,a57, a17,a16,a07,a07]
vmovaps ymm8, [rcx+r8*2]
vmovaps ymm9, [rcx+r11*1]
vperm2f128 ymm0, ymm8, ymm9, 0x20
vperm2f128 ymm4, ymm8, ymm9, 0x31
vmovaps ymm8, [rcx+rsi*1]
vmovaps ymm9, [rcx+rax*1]
vperm2f128 ymm1, ymm8, ymm9, 0x20
vperm2f128 ymm5, ymm8, ymm9, 0x31
```

- **Load eight floats from Row 0 & 4 Col 0-7**
- **Combine Cols 0-3 of Rows 0 & 4**
- **Combine Cols 4-7 of Rows 0 & 4**
- **Repeat for Rows 1 & 5**
- **Combine Cols 0-1 of Rows 0, 1, 4, 5**
- **Combine Cols 2-3, 4-5, and 6-7**
- **Repeat for Rows 2, 3, 6, 7**
Matrix Transpose – Intel® AVX Implementation

```
vunpcklqd ymm6, ymm0, ymm1
vunpckhqd ymm12, ymm0, ymm1
vunpcklqd ymm7, ymm4, ymm5
vunpckhqd ymm13, ymm4, ymm5

vshufps ymm4, ymm2, ymm6, 0x88
  // ymm4 = [a70,a60,a50,a40,a30,a20,a10,a00]
vmovaps [rbx], ymm4
vshufps ymm4, ymm2, ymm6, 0xdd
vmovaps [rbx+r8*1], ymm4

vshufps ymm4, ymm10, ymm12, 0x88
vmovaps [rbx+r8*2], ymm4
vshufps ymm4, ymm10, ymm12, 0xdd
vmovaps [rbx+rsi*1], ymm4

vshufps ymm4, ymm3, ymm7, 0x88
vmovaps [rbx+r8*4], ymm4
vshufps ymm4, ymm3, ymm7, 0xdd
vmovaps [rbx+rdi*1], ymm4

vshufps ymm4, ymm11, ymm13, 0x88
vmovaps [rbx+r11*1], ymm4
vshufps ymm4, ymm11, ymm13, 0xdd
vmovaps [rbx+rax*1], ymm4
```

Combine Cols 0-1, 2-3, 4-5 and 6-7 of Rows 2, 3, 6, & 7

Shuffle to complete transpose of Column 0

Store to Output Row 0

Shuffle to complete transpose of Col 1. Store to Output Row 1.

Complete Transpose
## Intel® Architecture Code Analyzer - Transpose

### Total Throughput: 25 Cycles;
Total Latency: 12 Cycles;
Total number of uops: 51

<table>
<thead>
<tr>
<th>Port Binding in Cycles:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
</tr>
<tr>
<td>Cycles</td>
</tr>
</tbody>
</table>

### Speedup – 2x

- Vmovaps ymm8, ymmword ptr [rcx]
- Vmovaps ymm9, ymmword ptr [rcx+8*4]
- Vpermf23128 ymm0, ymm8, ymm9, 0x20
- Vpermf23128 ymm4, ymm8, ymm9, 0x31
- Vmovaps ymm8, ymmword ptr [rcx+8*2]
- Vmovaps ymm9, ymmword ptr [rcx+11*4]
- Vpermf23128 ymm0, ymm8, ymm9, 0x20
- Vpermf23128 ymm4, ymm8, ymm9, 0x31
- Vmovaps ymm8, ymmword ptr [rcx+8*4]
- Vmovaps ymm9, ymmword ptr [rcx+rxax*4]
- Vpermf23128 ymm1, ymm8, ymm9, 0x20
- Vpermf23128 ymm5, ymm8, ymm9, 0x31
- Vunpckhpd ymm2, ymm0, ymm1
- Vunpckhpd ymm10, ymm0, ymm1
- Vunpckhpd ymm3, ymm4, ymm5
- Vmovaps ymm8, ymmword ptr [rcx+8*2]
- Vmovaps ymm9, ymmword ptr [rcx+11*4]
- Vpermf23128 ymm0, ymm8, ymm9, 0x20
- Vpermf23128 ymm4, ymm8, ymm9, 0x31
- Vmovaps ymm8, ymmword ptr [rcx+8*4]
- Vmovaps ymm9, ymmword ptr [rcx+rxax*4]
- Vpermf23128 ymm1, ymm8, ymm9, 0x20
- Vpermf23128 ymm5, ymm8, ymm9, 0x31
- Vunpckhpd ymm6, ymm0, ymm1
- Vunpckhpd ymm12, ymm0, ymm1
- Vunpckhpd ymm7, ymm4, ymm5
- Vmovaps ymm8, ymmword ptr [rcx+8*2]
- Vmovaps ymm9, ymmword ptr [rcx+11*4]
- Vpermf23128 ymm0, ymm8, ymm9, 0x20
- Vpermf23128 ymm4, ymm8, ymm9, 0x31
- Vmovaps ymm8, ymmword ptr [rcx+8*4]
- Vmovaps ymm9, ymmword ptr [rcx+rxax*4]
- Vpermf23128 ymm1, ymm8, ymm9, 0x20
- Vpermf23128 ymm5, ymm8, ymm9, 0x31
- Vunpckhpd ymm13, ymm4, ymm5
- Vshufps ymm4, ymm2, ymm6, 0x88
- Vmovaps ymmword ptr [rbx], ymm4
- Vshufps ymm4, ymm2, ymm6, 0xdd
- Vmovaps ymmword ptr [rbx+8*4], ymm4
- Vshufps ymm4, ymm10, ymm12, 0x88
- Vmovaps ymmword ptr [rbx+8*2], ymm4
- Vshufps ymm4, ymm10, ymm12, 0xdd
- Vmovaps ymmword ptr [rbx+rsi*1], ymm4
- Vshufps ymm4, ymm3, ymm7, 0x88
- Vmovaps ymmword ptr [rbx+8*4], ymm4
- Vshufps ymm4, ymm3, ymm7, 0xdd
- Vmovaps ymmword ptr [rbx+rxax*1], ymm4
- Vshufps ymm4, ymm11, ymm13, 0x88
- Vshufps ymm4, ymm11, ymm13, 0xdd
- Vshufps ymm4, ymm11, ymm13, 0xdd
- Vmovaps ymmword ptr [rbx+rxax*1], ymm4
- Add rcx, 0x20
- Lea rbx, ptr [rbx+8*8]
- Dec rl8
- Jnz 0xffffffffffffffff15

### Port 5 Utilization is high
Transpose – Intel® AVX Stride Load Method

- Load eight floats
  - Four from Row 0
  - Four from Row 4 (the stride)

`vmovaps xmm0, [mem]`

`vinsertf128 ymm0, ymm0, [mem+4*rowSize], 1`
Improved Matrix Transpose – Intel® AVX Implementation

vmovaps  xmm0, [rcx]    // [A03,A02,A01,A00]
vinstrtf128 ymm0, ymm0, [rcx + 4*r8], 1 // [A43,A42,A41,A40, A03,A02,A01,A00]
vmovaps  xmm1, [rcx + r8]  // [A13,A12,A11,A10]
vinstrtf128 ymm1, ymm1, [rcx + rdi], 1 // [A53,A52,A51,A50, A13,A12,A11,A10]
vunpcklpd ymm8, ymm0, ymm1  // [A51,A50,A41,A40, A11,A10,A01,A00]
vunpckhpd ymm9, ymm0, ymm1  // [A53,A52,A43,A42, A13,A12,A03,A02]
vmovaps  xmm0, [rcx+2*r8]  // [A23,A22,A21,A20]
vinstrtf128 ymm0, ymm0, [rcx + r11], 1 // [A63,662,A61,A60, A23,A22,A21,A20]
vmovaps  xmm1, [rcx+rxi] // [A33,A32,A31,A30]
vinstrtf128 ymm1, ymm1, [rcx + rax], 1 // [A73,A72,A71,A70, A33,A32,A21,A30]
vunpcklpd ymm10, ymm0, ymm1 // [A71,A70,A61,A60, A31,A30,A21,A20]
vunpckhpd ymm11, ymm0, ymm1 // [A73,A72,A63,A62, A33,A32,A23,A22]
vshufps  ymm4, ymm8, ymm10, 0x88 // [A70,A60,A50,A40,A30,A20,A10,A00]
vunmovaps [rbx], ymm4  // store first row of transposed matrix

Similar operations to do transpose for other 7 rows
## Intel® Architecture Code Analyzer

### Analysis of Transpose – After

**Total Throughput:** 17 Cycles;  
**Total Latency:** 10 Cycles;  
**Throughput Bottleneck:** Port 5  
**Total number of Uops:** 59

### Port Binding in cycles:

<table>
<thead>
<tr>
<th>Port</th>
<th>0 - DV</th>
<th>1</th>
<th>2 - D</th>
<th>3 - D</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>0</td>
<td>12</td>
<td>8</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

### Speedup – 2.53x

**Port 5 pressure reduction provides speedup!**

<table>
<thead>
<tr>
<th>Num of Uops</th>
<th>0 - DV</th>
<th>1</th>
<th>2 - D</th>
<th>3 - D</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Complex Multiply with Intel® AVX

Complex Multiply
Complex Multiply - AoS

- AoS scenario in which SIMD can be applied without an AoS to SoA transformation

```
struct complex {
    float a;
    float b;
};

struct complex ab[100];
struct complex AB[100];
```

\[(a_0 + b_0i)(A_0 + B_0i) = a_0A_0 - b_0B_0 + (a_0B_0 + b_0A_0)i\]
**Complex Multiply – Intel® SSE Implementation (Unroll once)**

```c
__m256* inPtr1 = A[N*2];
__m256* inPtr2 = B[N*2];
__m256* outPtr = C[N*2];

for(int i=0 ; i<N; i+=4) {
    xmm0 = _mm_loadu_ps(inPtr1);         // [b1,a1,b0,a0]
    xmm1 = _mm_loadu_ps(inPtr2);         // [B1,A1,B0,A0]
    xmm2 = _mm_movehdup_ps(xmm1); // [B1,B1,B0,B0]
    xmm1 = _mm_moveldup_ps(xmm1);  // [A1,A1,A0,A0]
    xmm1 = _mm_mul_ps(xmm1, xmm0);  // [A1*b1,A1*a1,A0*b0,A0*a0]
    xmm0 = _mm_shuffle_ps(xmm0, xmm0, 0xb1);  // [a1,b1,a0,b0]
    xmm2 = _mm_mul_ps(xmm2, xmm0);  // [B1*a1,B1*b1, B0*a0,B0*b0]
    xmm0 = _mm_addsub_ps(xmm1, xmm2); // [A1*b1+B1*a1,A1*a1-B1*b1, A0*b0+B0*a0, A0*a0-B0*B0]
    _mm_storeu_ps(outPtr, xmm0);
    ... Unroll once (instructions are not shown)
    inPtr1+=8;
    inPtr2+=8;
    outPtr+=8;
}
```
**Complex Multiply – Intel® AVX Implementation (1)**

- **Load four pairs of complex numbers**
  - 2 loads from 2 arrays of complex numbers

```
_mm256_loadu_ps(ymm0, inPtr1);
_mm256_loadu_ps(ymm1, inPtr2);
```

- **Duplicate Imaginary & Real parts of 2nd array**

```
_mm256_movehdup_ps(ymm1);
_mm256_moveldup_ps(ymm1);
```
Complex Multiply – Intel® AVX Implementation (2)

- **Multiply Reals of 2\textsuperscript{nd} Array with 1\textsuperscript{st} Array**
  - Dot Products

  \[
  \text{YMM1} = \text{ymm256\_mul\_ps(ymm1, ymm0)}
  \]

- **Multiply Imaginaries of 2\textsuperscript{nd} Array with Reverse of 1\textsuperscript{st} Array**
  - Dot Products

  \[
  \text{YMM0} = \text{ymm256\_shuffle\_ps(ymm0, ymm0, 0xb1)}
  \]

  \[
  \text{YMM2} = \text{ymm256\_mul\_ps(ymm2, ymm0)}
  \]
Complex Multiply – Intel® AVX Implementation (3)

- Calculate Add/Subtract Multiply Products

\[ \text{ymm0} = \_\text{mm256\_addsub\_ps(ymm1, ymm2)} \]
Complex Multiply – Intel® AVX Implementation (Unroll Once)

for(int i=0 ; i<Size ; i+=8) {
    ymm0 = _mm256_loadu_ps(inPtr1);
    ymm1 = _mm256_loadu_ps(inPtr2);
    ymm2 = _mm256_movehdup_ps(ymm1);
    ymm1 = _mm256_moveldup_ps(ymm1);
    ymm1 = _mm256_mul_ps(ymm1, ymm0);
    ymm0 = _mm256_shuffle_ps(ymm0,ymm0,0xb1);
    ymm2 = _mm256_mul_ps(ymm2, ymm0);
    ymm0 = _mm256_addsub_ps(ymm1, ymm2);
    _mm256_storeu_ps(outPtr, ymm0);

    ...Unroll once (instructions are not shown)

    inPtr1+=16;
    inPtr2+=16;
    outPtr+=16;
}

Speedup – 1.79x
## Intel® SSE to Intel® AVX Kernel Speedup Summary

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Speedup*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Transpose</td>
<td>2.53x</td>
</tr>
<tr>
<td>Matrix Addition</td>
<td>1.42x</td>
</tr>
<tr>
<td>AoS to SoA</td>
<td>1.84x</td>
</tr>
<tr>
<td>Complex Multiply</td>
<td>1.79x</td>
</tr>
</tbody>
</table>

Achieve great speedup with Intel® AVX - Wider vectors, Non Destructive Destination, and Enhanced Data Arrangement Primitives

Intel SSE and Intel AVX kernels run on Intel® microarchitecture (Sandy Bridge) based hardware. See backup slide for details on system configuration used for the measurement.
Agenda

• Setting the Pace for Intel Instruction Set
• Next generation Intel® Core™ processors (codename Sandy Bridge) features – Intel® Advanced Vector Extensions (Intel® AVX)
• Software Development and Tools
• Tuning Tips and Kernels
• Update on FMA
• Summary and Call to Action
Fused Multiply Add ("FMA")
- Not supported on Intel® Microarchitecture (Sandy Bridge)

- \( \pm A \times B \pm C \); IEEE-754-2008 compliant (only round is at the end)
- 3 formats allow selection which of A, B & C comes from memory and which one is being overwritten
  - vFMAAdd<nnn> srcdst1,src2,src3/mem, where <nnn> can be:
    - 132: srcdst1 = srcdst1*src3/mem+src2
    - 213: srcdst1 = src2*srcdst1+src3/mem
    - 231: srcdst1 = src2*src3/mem+srcdst1

- Key benefits:
  1) Increased FP compute density
  2) Improved numeric accuracy
  3) Benefits vector and scalar workloads

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMADD231(PS/PD/SS/SD)</td>
<td>( C = A \times B + C )</td>
</tr>
<tr>
<td>VFMSUB231(PS/PD/SS/SD)</td>
<td>( C = A \times B - C )</td>
</tr>
<tr>
<td>VFNMADD231(PS/PD/SS/SD)</td>
<td>( C = - A \times B + C )</td>
</tr>
<tr>
<td>VFNMSUB231(PS/PD/SS/SD)</td>
<td>( C = - A \times B - C )</td>
</tr>
</tbody>
</table>
| VFMADDSUB231(PS/PD)          | \( C_{odd} = A_{odd} \times B_{odd} + C_{odd} \)
                                | \( C_{even} = A_{even} \times B_{even} - C_{even} \) |
| VFMSUBADD231(PS/PD)          | \( C_{odd} = A_{odd} \times B_{odd} - C_{odd} \)
                                | \( C_{even} = A_{even} \times B_{even} + C_{even} \) |

Extensible Architecture – More features in upcoming uArch!
Agenda

• Setting the Pace for Intel Instruction Set
• Intel® Microarchitecture (Sandy Bridge) features – Intel® Advanced Vector Extensions (Intel® AVX)
• Software Development and Tools
• Tuning Tips and Kernels
• Update on FMA

• Summary and Call to Action
# Key Intel® Advanced Vector Extensions (Intel® AVX) Features

<table>
<thead>
<tr>
<th><strong>KEY FEATURES</strong></th>
<th><strong>BENEFITS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Wider Vectors</td>
<td>• Up to 2x peak FLOPs (floating point operations per second) output with good power efficiency</td>
</tr>
<tr>
<td>– Increased from 128 to 256 bit</td>
<td></td>
</tr>
<tr>
<td>– Two 128-bit load ports</td>
<td></td>
</tr>
<tr>
<td>• Enhanced Data Rearrangement</td>
<td>• Organize, access and pull only necessary data more quickly and efficiently</td>
</tr>
<tr>
<td>– Use the new 256 bit primitives to broadcast, mask loads and permute data</td>
<td></td>
</tr>
<tr>
<td>• Three and four Operands</td>
<td>• Fewer register copies, better register use for both vector and scalar code</td>
</tr>
<tr>
<td>– Non Destructive Syntax for both Intel AVX 128 and Intel AVX 256</td>
<td></td>
</tr>
<tr>
<td>• Flexible unaligned memory access support</td>
<td>• More opportunities to fuse load and compute operations</td>
</tr>
<tr>
<td>• Extensible new opcode (VEX)</td>
<td>• Code size reduction</td>
</tr>
</tbody>
</table>

___

*Intel® AVX is a general purpose architecture, expected to supplant Intel® SSE in all applications used today*
Call to Action

- Get your software ready for Intel® Advanced Vector Extensions (Intel® AVX) and Intel® microarchitecture (Sandy Bridge)
- Utilize the benefits of Intel AVX
  - Wider vectors: 256-bit registers
  - New 256-bit primitives
    - Masked load & stores
    - Broadcast
  - Non-destructive destination
  - Two 128-bit load ports on Intel microarchitecture (Sandy Bridge)
  - Natural extension of existing programming model
Hosting a Community of Resources and Developers to get you started on Intel® Advanced Vector Extensions (Intel® AVX)

Go to [http://www.intel.com/software/avx](http://www.intel.com/software/avx) to get more whitepapers and information on Intel® AVX
Additional Sources of Information

- More Intel® AVX information at the Intel Software Network website:
  - http://www.intel.com/software/avx
Legal Disclaimer

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.

- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Sandy Bridge and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user.
- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.
- Intel, and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- *Other names and brands may be claimed as the property of others.
- Copyright © 2010 Intel Corporation.
Risk Factors

The above statements and any others in this document that refer to plans and expectations for the first quarter, the year and the future are forward-looking statements that involve a number of risks and uncertainties. Many factors could affect Intel’s actual results, and variances from Intel’s current expectations regarding such factors could cause actual results to differ materially from those expressed in these forward-looking statements. Intel presently considers the following to be the important factors that could cause actual results to differ materially from the corporation’s expectations. Demand could be different from Intel’s expectations due to factors including changes in business and economic conditions; customer acceptance of Intel’s and competitors’ products; changes in customer order patterns including order cancellations; and changes in the level of inventory at customers. Intel operates in intensely competitive industries that are characterized by a high percentage of costs that are fixed or difficult to reduce in the short term and product demand that is highly variable and difficult to forecast. Additionally, Intel is in the process of transitioning to its next generation of products on 32nm process technology, and there could be execution issues associated with these changes, including product defects and errata along with lower than anticipated manufacturing yields. Revenue and the gross margin percentage are affected by the timing of new Intel product introductions and the demand for and market acceptance of Intel's products; actions taken by Intel's competitors, including product offerings and introductions, marketing programs and pricing pressures and Intel’s response to such actions; defects or disruptions in the supply of materials or resources; and Intel's ability to respond quickly to technological developments and to incorporate new features into its products. The gross margin percentage could vary significantly from expectations based on changes in revenue levels; product mix and pricing; start-up costs, including costs associated with the new 32nm process technology; variations in inventory valuation, including variations related to the timing of qualifying products for sale; excess or obsolete inventory; manufacturing yields; changes in unit costs; impairments of long-lived assets, including manufacturing, assembly/test and intangible assets; the timing and execution of the manufacturing ramp and associated costs; and capacity utilization; Expenses, particularly certain marketing and compensation expenses, as well as restructuring and asset impairment charges, vary depending on the level of demand for Intel's products and the level of revenue and profits. The majority of our non-marketable equity investment portfolio balance is concentrated in companies in the flash memory market segment, and declines in this market segment or changes in management’s plans with respect to our investments in this market segment could result in significant impairment charges, impacting restructuring charges as well as gains/losses on equity investments and interest and other. Intel's results could be impacted by adverse economic, social, political and physical/infrastructure conditions in countries where Intel, its customers or its suppliers operate, including military conflict and other security risks, natural disasters, infrastructure disruptions, health concerns and fluctuations in currency exchange rates. Intel’s results could be affected by the timing of closing of acquisitions and divestitures. Intel's results could be affected by adverse effects associated with product defects and errata (deviations from published specifications), and by litigation or regulatory matters involving intellectual property, stockholder, consumer, antitrust and other issues, such as the litigation and regulatory matters described in Intel's SEC reports. An unfavorable ruling could include monetary damages or an injunction prohibiting us from manufacturing or selling one or more products, precluding particular business practices, impacting our ability to design our products, or requiring other remedies such as compulsory licensing of intellectual property. A detailed discussion of these and other risk factors that could affect Intel’s results is included in Intel’s SEC filings, including the report on Form 10-Q.
Back Up
System Configurations used for Intel® SSE to Intel® AVX Kernel Speedup Measurement (slide 86)

- Intel Sandy Bridge 3.0 GHz Processor
- 8 GB DDR3 memory
- Microsoft 64bit Win7 OS

Disclaimer:

Intel® SSE to Intel® AVX Kernel speedup were measured from kernels written by Intel software Engineers. Testing were conducted on system with above configurations. Any difference in software design or system hardware configurations may affect actual performance.

Other names and brands may be claimed as the property of others