SSE and SSE2

Jeremy Johnson
Timothy A. Chagnon

All images from Intel® 64 and IA-32 Architectures Software Developer's Manuals
Overview

- **SSE**: Streaming SIMD (Single Instruction Multiple Data) Extensions
- Successor to 64-bit MMX integer and AMD's 3DNow! extensions

### SSE

- Introduced in 1999 with the Pentium III
- 128-bit packed single precision floating point
- 8 128-bit registers $\text{xmm0-xmm7}$, hold 4 floats each
- Additional 8 on x86-64 ($\text{xmm8-xmm15}$)

### SSE2

- Introduced with Pentium 4
- Added double precision FP and several integer types

### SSE 3 & 4

- More ops: DSP, permutations, fixed point, dot product...
Data Types

- Instructions end in D, S, I, Q etc. to designate type
- Conversion instructions CVTxx2xx

**SSE**

Contains 4 Single-Precision Floating-Point Values

**SSE2**

128-Bit Packed Double-Precision Floating-Point

128-Bit Packed Byte Integers

128-Bit Packed Word Integers

128-Bit Packed Doubleword Integers

128-Bit Packed Quadword Integers

**Figure 10-4. 128-Bit Packed Single-Precision Floating-Point Data Type**

**Figure 11-2. Data Types Introduced with the SSE2 Extensions**
Instruction Basics

- **Packed vs. Scalar Ops**
  - Most operate on 2 args
    - OP xmm-dest, xmm/m128-src
    - In-place operation on dest.

- **Instruction Varieties**
  - MOV loads, stores
  - Arithmetic & Logical
  - Shuffle & Unpack
  - CVT conversion
  - Cache & Ordering Control

*Figure 10-5. Packed Single-Precision Floating-Point Operation*

*Figure 10-6. Scalar Single-Precision Floating-Point Operation*
Arithmetic & Logical Instructions

- Can take a memory reference as 2\textsuperscript{nd} argument
- All put results into 1\textsuperscript{st} argument
  - CMP can be made to use EFLAGS
- Full instruction names are OP+TYPE...
  - ADDPD add packed double
  - ANDSS and scalar single
  - etc.

\begin{align*}
\text{MOVAPD} & \quad \text{xmm0, [eax]} \\
\text{MOVAPD} & \quad \text{xmm1, [eax+16]} \\
\text{MOVAPD} & \quad \text{xmm2, xmm0} \\
\text{ADDPD} & \quad \text{xmm2, xmm1} \\
\text{SUBPDP} & \quad \text{xmm0, xmm1} \\
\text{MOVAPD} & \quad \text{[eax], xmm2} \\
\text{MOVAPD} & \quad \text{[eax+16], xmm0}
\end{align*}
Moving Data to/from Memory

- MOVAPD move aligned
- MOVUPD move unaligned
  - reg-reg, reg-mem, mem-reg
- Memory should be 16-byte aligned if possible
  - Starts at a 128-bit boundary in Virtual?/Physical? addressing
  - Special types and attributes are used in C/C++
- MOVUPD takes much longer than MOVAPD
  - Intel's optimization manual says it's actually faster to do

MOVSD xmm0, [mem]
MOVSD xmm1, [mem+8]
UNPCKLPD xmm0, xmm1
Shuffle & Unpack

- SHUFPD xmm0, xmm1, pattern
- UNPCKHPD xmm0, xmm1
- UNPCKLPD xmm0, xmm1
- Combine parts of 2 vectors
- In-Place
- Value(s) from 1\textsuperscript{st} argument alway go into low ½ of dest.
- Note that UNPCKs are just a shortcut of SHUF
  - But is it faster or slower?
C/C++ Intrinsics

- Both Intel's ICC and GCC have “intrinsic” functions to access the SSE instructions
  - More or less a 1:1 mapping of instructions
- Instead of in-place, they look like 2-in, 1-out functions
  - Does this indicate that penalties are incurred for unnecessary register copies/loads/stores?
  - Or are these optimized away? ... It appears that way.
- MOV is split up into load, store, and set

```c
__m128d _mm_add_pd(__m128d a, __m128d b)

Adds the two DP FP values of a and b.
r0 := a0 + b0
r1 := a1 + b1
```

```c
__m128d _mm_load_pd(double const*dp)
(uses MOVAPD) Loads two DP FP values.
The address p must be 16-byte aligned.
r0 := p[0]
r1 := p[1]
```


#include <xmmintrin.h>

int main(void){
    int i;
    __m128d a, b, c;
    double x0[2] __attribute__((aligned(16))) = {1.2, 3.5};
    double x1[2] __attribute__((aligned(16))) = {-0.7, 2.6};

    a = _mm_load_pd(x0);
    b = _mm_load_pd(x1);
    c = _mm_add_pd(a, b);
    _mm_store_pd(x0, c);

    for(i = 0; i < 2; i++)
        printf("%.2f\n", x0[i]);

    return 0;
}
Efficient Use

- Use 16-byte aligned memory
- Compiler driven automated vectorization is limited
- Use struct-of-arrays (SoA) instead of AoS (3d vectors)
  - i.e. block across multiple entities and do normal operations
To Investigate

- Denormals and underflow can cause penalties (1500? cycles)
- Are intrinsics efficient? Do they use direct memory refs?
- SHUF vs UNPCK?
- Cache utilization, ordering instructions ...
- x87 FPU and SSE are disjoint and can be mixed ...
- MMX registers can be used for shuffle or copy
- Intel Core architecture has more efficient SIMD than NetBurst
- Direct memory references
- Trace cache and reorder buffer effects on loop unrolling
- Non-temporal stores
- Prefetch
SHUF vs. UNPCK

- Intel Optimization Manual shows, sometimes UNPCK is faster
  - Probably because there's no decoding of immediate field
- gcc-4.x will convert a call to the shuffle intrinsic to an unpck instruction if it can
- Table data represents usage of register arguments; delays from direct memory references depend heavily on cache state.

### NetBurst Arch. (Pentium 4, Xeon)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency(^1)</th>
<th>Throughput</th>
<th>Execution Unit(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPD xmm, xmm</td>
<td>0F_03H</td>
<td>0F_02H</td>
<td>FP_ADD</td>
</tr>
<tr>
<td>SHUFPD(^3) xmm, xmm, imm8</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>UNPCKHPD xmm, xmm</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>UNPCKLPD(^3) xmm, xmm</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

### Core Arch. (Core Duo, Pentium M)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency(^1)</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisplayFamily_DisplayModel</td>
<td>06_0FH</td>
<td>06_0EH</td>
</tr>
<tr>
<td>ADDPD xmm, xmm</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>SHUFPD xmm, xmm, imm8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>UNPCKHPD xmm, xmm</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>UNPCKLPD xmm, xmm</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

DisplayFamily_DisplayModel represent variation across different processors. Family 0F is NetBurst Architecture, Family 06 is Intel Core Architecture. Model for NetBurst in range 00H to 06H, data for 03H applies to 04H and 06H. Model for Pentium M are 09H and 0DH, Core Solo/Duo is 0EH, Core is 0FH.
References

- Wikipedia: Streaming SIMD Extensions

- Intel® 64 and IA-32 Architectures Software Developer's Manuals

- Intel® C++ Compiler Documentation

- Apple Developer Connection – SSE Performance Programming
Simple Example (loop)

void loop(float c[], float a[], float b[], int n)
{
    int i;

    for (i=0;i<n;i++)
        c[i] = a[i] + b[i];
}
void uloop(double c[], double a[], double b[], int n)
{
    int i, nb;
    nb = n - (n % 4);
    for (i=0; i<n; i+=4) {
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }
    for (i=nb; i<n; i++)
        c[i] = a[i] + b[i];
}
Vectorized Loop

#include <xmmintrin.h>

void vloop(float c[], float a[], float b[], int n)
{
    int i,nb;
    __m128 v1, v2;

    nb = n - (n % 4);
    for (i=0;i<n;i+=4) {
        v1 = _mm_load_ps(&a[i]);
        v2 = _mm_load_ps(&b[i]);
        v2 = _mm_add_ps(v1,v2);
        _mm_store_ps(&c[i],v2);
    }
    for (i=nb;i<n;i++)
        c[i] = a[i] + b[i];
}
```c
#include <xmmintrin.h>
void vloop(double c[], double a[], double b[], int n)
{
    int i,nb;
    _m128d v1, v2;

    nb = n - (n % 2);
    for (i=0;i<n;i+=2) {
        v1 = _mm_load_pd(&a[i]);
        v2 = _mm_load_pd(&b[i]);
        v2 = _mm_add_pd(v1,v2);
        _mm_store_pd(&c[i],v2);
    }
    for (i=nb;i<n;i++)
        c[i] = a[i] + b[i];
}
```
Assembly Code

... 

.p2align 4,,7
.L4:
  movaps (%rax,%rsi), %xmm0
  addl$4, %edx
  addps (%rax,%r8), %xmm0
  movaps %xmm0, (%rax,%rdi)
  addq $16, %rax
  cmpl %edx, %ecx
  jg .L4  
  ...
Shuffle Instruction

__m128 a, b, c;

// _mm_load_ps and _mm_store_ps: The address must be 16-byte aligned

float inp1_sse[4] __attribute__((aligned(16))) = { 0.0, 1.0, 2.0, 3.0 };  
float inp2_sse[4] __attribute__((aligned(16))) = { 4.0, 5.0, 6.0, 7.0 };  
float out_sse[4] __attribute__((aligned(16)))

int main(void){
    int i;

    printf("...__mm_shuffle_ps ... 2 inputs\n");

    a = _mm_load_ps(inp1_sse);
    b = _mm_load_ps(inp2_sse);
    c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(1,0,3,2));
    _mm_store_ps(out_sse,c);
# Inner Product (version 1)

```c
#include <xmmintrin.h>
#include <stdio.h>

float vinner(float a[], float b[], int n)
{
    int i,nb;
    float s;
    float temp[4] __attribute__((aligned(16)));
    __m128 v1, v2, acc;

    nb = n - (n % 4);
    acc = _mm_xor_ps(acc,acc);
    for (i=0;i<nb;i+=4) {
        v1 = _mm_load_ps(&a[i]);
        v2 = _mm_load_ps(&b[i]);
        v2 = _mm_mul_ps(v1,v2);
        acc = _mm_add_ps(acc,v2);
    }
    _mm_store_ps(temp,acc);
    printf("%g %g %g %g\n",temp[0],temp[1],temp[2],temp[3]);
    s = 0.0;
    for (i=nb;i<n;i++)
        s += a[i] * b[i];
    printf("s = %g\n",s);
    return s;
}
```
Inner Product (version 2)

#include <xmmintrin.h>
#include <stdio.h>
float vinner(float a[], float b[], int n)
{
    int i,nb;
    float s;
    float temp[4] __attribute__((aligned(16)));
    __m128 v1, v2, acc;

    nb = n - (n % 4);
    acc = _mm_xor_ps(acc,acc);
    for (i=0;i<nb;i+=4) {
        v1 = _mm_load_ps(&a[i]);
        v2 = _mm_load_ps(&b[i]);
        v2 = _mm_mul_ps(v1,v2);
        acc = _mm_add_ps(acc,v2);
    }
    v1 = _mm_shuffle_ps(acc,acc,_MM_SHUFFLE(1,0,3,2));
    v1 = _mm_add_ps(v1,acc);
    v2 = _mm_shuffle_ps(v1,v1,_MM_SHUFFLE(2,3,0,1));
    v2 = _mm_add_ps(v2,v1);
    _mm_store_ss(&s,v2);
    for (i=nb;i<n;i++)
        s += a[i] * b[i];
    return s;
}
Vectorized WHT_4
Recursive $\text{WHT}_N$
Vectorized $\text{WHT}_2 \otimes I_N$

Code Transformation:
  Loop code $\rightarrow$ Unrolled $\rightarrow$ Interleaved $\rightarrow$ Vectorized
Vectorized $WHT_2 \otimes I_N$

Code Transformation:

Loop code $\rightarrow$ Unrolled $\rightarrow$ Interleaved $\rightarrow$ Vectorized
Vectorized $\text{WHT}_2 \otimes I_N$

Code Transformation:
Loop code $\rightarrow$ Unrolled $\rightarrow$ Interleaved $\rightarrow$ Vectorized
Vectorized $WHT_2 \otimes I_N$

Code Transformation:
  Loop code $\rightarrow$ Unrolled $\rightarrow$ Interleaved $\rightarrow$ Vectorized
Vectorized $\text{WHT}_8$