

Vectorization with SIMD

Simon Taylor / JLab

- SSE instruction sets
 - Some examples
 - Benchmarks

Vectorization

- Modern CPUs have special registers that support **Single Instruction, Multiple Data** operations
 - Operations on 4 ints, 4 floats, or 2 doubles at a time can be done in parallel (“vectorization”)
 - 128-bit registers XMM0, XMM1, ...
 - AMD-specific: 3DNow! (more or less obsolete as of last year...)
 - Intel-specific: **SSE**=“Streaming SIMD Extensions”
- Modern compilers can take advantage of SIMD instructions without effort on the programmer's part = “auto-vectorization”
 - The compiler determines whether operations can be vectorized (unrolling loops, etc.) and generates the appropriate SIMD-ized code
 - **gcc**: on by default at optimization level *-O3*
 - ... however, many applications require writing one's own code...

SSE instruction sets

- There have been four iterations of SSE
 - **SSE**, circa 1999 → 8 new 128-bit registers XMM0,...,XMM7
 - New integer operations
 - Single-precision (32-bit) floating point operations
 - **SSE2**, circa 2000
 - Double precision (64-bit) floating point operations
 - **SSE3/SSSE3**, circa 2004
 - “Horizontal” arithmetic operations
 - Byte permutations
 - **SSE4**, circa 2007, added many more operations
 - Dot product instructions
 - More integer instructions
 - ... however, most of the new instructions do not appear to be available for AMD processors...

x86-64 extensions added 8 additional 128-bit registers XMM8,...,XMM15 (only for 64 bit systems)

SSE instructions

• Floating point instructions

• Memory-to-Register / Register-to-Memory / Register-to-Register data movement

- Scalar – MOVSS
- Packed – MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS

• Arithmetic

- Scalar – ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
- Packed – ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS

• Comparisons

- Scalar – CMPSS, COMISS, UCOMISS
- Packed – CMPPS

• Data shuffle and unpacking

- Packed – SHUFPS, UNPCKHPS, UNPCKLPS

• Data-type conversion

- Scalar – CVTSS2SI, CVTSS2SI, CVTTSS2SI
- Packed – CVTPI2PS, CVTPI2PS, CVTTPS2PI

• Bitwise logical operations

- Packed – ANDPS, ORPS, XORPS, ANDNPS

• Integer instructions

• Arithmetic

- PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAWSW, PMINSW

• Data movement

- PEXTRW, PINSRW

• Other

- PMOVMSKB, PSHUFW

• Other instructions

• MXCSR management

- LDMXCSR, STMXCSR

• Cache and Memory management

- MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE

(from Ref. 1)

SSE2 instructions

• Floating point instructions

• Memory-to-Register / Register-to-Memory / Register-to-Register data movement

- Scalar – MOVSD
- Packed – MOVAPD, MOVHPD, MOVLPD, MOVMSKPD, MOVUPD

• Arithmetic

- Scalar – ADDSD, DIVSD, MAXSD, MINSR, MULSD, SQRTSD, SUBSD
- Packed – ADDPD, DIVPD, MAXPD, MINPD, MULPD, SQRTPD, SUBPD

• Comparisons

- Scalar – CMPSD, COMISD, UCOMISD
- Packed – CMPPD

• Data shuffle and unpacking

- Packed – SHUFFPD, UNPCKHPD, UNPCKLPD

• Data-type conversion

- CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSS2SD, CVTTPD2DQ, CVTTPD2PI, CVTTPS2DQ, CVTTSD2SI

• Bitwise logical operations

- Packed – ANDNPD, ANDPD, ORPD, XORPD

• Integer instructions

• Memory-to-Register / Register-to-Memory / Register-to-Register data movement

- MOVDQ2Q, MOVDQA, MOVDQU, MOVQ2DQ

• Arithmetic

- PADDQ, PSUBQ, PMULUDQ

• Data shuffle and unpacking

- PSHUFW, PSHUFLW, PSHUFD, PUNPCKHQDQ, PUNPCKLQDQ

• Shift operations

- PSLLDQ, PSRLDQ

(from Ref. 2)

SSE3/SSSE3 instructions

- **SSE3 instructions**

- **Memory-to-Register / Register-to-Memory / Register-to-Register data movement**

- MOVDDUP, MOVSHDUP, MOVSLDUP

- **Arithmetic:**

- ADDSUBPD, ADDSUBPS

- **“Horizontal” arithmetic**

- HADDPD, HADDPS, HSUBPD, HSUBPS

- **Instructions for multi-threaded applications**

- MONITOR, MWAIT

- **Other instructions**

- FISTTP, LDDQU

- **SSSE3 instructions**

- **Arithmetic:**

- PABSB, PABSD, PABSW, PMULHRWS, PMADDUBSW, PSIGNW, PSIGND, PSIGNB

- **“Horizontal” arithmetic**

- PHSUBW, PHSUBSW, PHSUBD, PHADDW, PHADDSW, PHADDD

- **Other**

- PALIGNR, PSHUFB

(from Refs. 3&4)

SSE4 instructions

- **SSE4.1**, added with Core 2 manufactured in 45nm
 - **Arithmetic**
 - MPSADBW (offset sums of absolute differences)
 - PMULLD, PMULDQ, PMINSB, PMAxSB, PMINUW, PMAxUW, PMINUD, PMAxUD, PMINSD, PMAxSD
 - **Rounding**
 - ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD
 - **Dot products**
 - DPPS, DPPD
 - **Memory-to-Register / Register-to-Memory / Register-to-Register data movement**
 - PHMINPOSUW
 - BLENDPS, BLENDPD, BLENDVPS, BLENDVPD, PBLENDVB, PBLENDW
 - INSERTPS, PINSRB, PINSRD/PINSRQ, EXTRACTPS, PEXTRB, PEXTRW, PEXTRD/PEXTRQ
 - **Extension to wider types**
 - PMOVSBW, PMOVZBW, PMOVXBD, PMOVZBD, PMOVXBQ, PMOVZBQ, PMOVXWD, PMOVZWD, PMOVXWQ, PMOVZXWQ, PMOVXDQ, PMOVZXDQ
 - **Comparison:** PCMPEQQ
 - **Other**
 - MOVNTDQA, PACKUSDW, PTEST
- **SSE4a**, for AMD's Barcelona architecture
 - **Bit manipulation**
 - LZCNT, POPCNT (POPulation CouNT, also available for Intel chips)
 - EXTRQ/INSERTQ
 - **Scalar streaming store instructions**
 - MOVNTSD/MOVNTSS
- **SSE4.2**, added with Nehalem processors
 - **Checksum operation:** CRC32
 - **Comparisons**
 - PCMPSTRI, PCMPSTRM, PCMPISTRI, PCMPISTRM, PCMPGTQ

(from Ref. 5)

Some useful operations in more detail

- SSE2 instructions

- **ADDPD** $\boxed{A1} \boxed{A0} + \boxed{B1} \boxed{B0} \rightarrow \boxed{A1+B1} \boxed{A0+B0}$

- **SUBPD** $\boxed{A1} \boxed{A0} - \boxed{B1} \boxed{B0} \rightarrow \boxed{A1-B1} \boxed{A0-B0}$

- **MULPD** $\boxed{A1} \boxed{A0} * \boxed{B1} \boxed{B0} \rightarrow \boxed{A1*B1} \boxed{A0*B0}$

- SSE3 instructions

- **HADDPD** = horizontal add $\boxed{A1} \boxed{A0} \oplus \boxed{B1} \boxed{B0} \rightarrow \boxed{B0+B1} \boxed{A0+A1}$

- Implementation in C:

- Basic unit: `__m128d v`;

- Double precision operations have the form `_mm_xxx_pd()`;

- For example: `_mm_add_pd(__m128d A, __m128d B)`;

- Applications: **2-** and **3-vector** classes, **matrix** classes

Potential GOTCHA! – most SSE2 instructions require 16-byte (128 bit) data alignment...

A simple example

- Addition of a 2-component vector

- gcc flags: `-msse2 -mfpmath=sse`

- Header file: `#include <emmintrin.h>`

- Basic unit:

```
union dvec{
    __m128d v;
    double d[2];
}vec;
```

- Vector access:

```
__m128d GetV() const {return vec.v;}
```

- Code fragment:

```
// Vector addition
DVector2 operator+(const DVector2 &v1) const{
    return DVector2(_mm_add_pd(GetV(),v1.GetV()));
}
```

A more complex example

```
#include <emmintrin.h> // SSE2 instructions
```

```
// Matrix multiplication: (3x3) x (3x2)
```

```
DMatrix3x2 operator*(const DMatrix3x2 &m2){  
    __m128d m11=_mm_set1_pd(m2(0,0));  
    __m128d m12=_mm_set1_pd(m2(0,1));  
    __m128d m21=_mm_set1_pd(m2(1,0));  
    __m128d m22=_mm_set1_pd(m2(1,1));  
    __m128d m31=_mm_set1_pd(m2(2,0));  
    __m128d m32=_mm_set1_pd(m2(2,1));  
    return  
        DMatrix3x2(_mm_add_pd(_mm_mul_pd(GetV(0,0),m11),  
                               _mm_add_pd(_mm_mul_pd(GetV(0,1),m21),  
                                           _mm_mul_pd(GetV(0,2),m31))),  
                  _mm_add_pd(_mm_mul_pd(GetV(0,0),m12),  
                               _mm_add_pd(_mm_mul_pd(GetV(0,1),m22),  
                                           _mm_mul_pd(GetV(0,2),m32))),  
                  _mm_add_pd(_mm_mul_pd(GetV(1,0),m11),  
                               _mm_add_pd(_mm_mul_pd(GetV(1,1),m21),  
                                           _mm_mul_pd(GetV(1,2),m31))),  
                  _mm_add_pd(_mm_mul_pd(GetV(1,0),m12),  
                               _mm_add_pd(_mm_mul_pd(GetV(1,1),m22),  
                                           _mm_mul_pd(GetV(1,2),m32))));  
}
```

Three-vector operation benchmarks

- Compare Root-based 3-vectors (TVector3) to custom SIMD-ized matrices (DVector3, implemented in double precision)
 - 10,000,000 “events” on 2.8 GHz Nehalem

Operation	DVector3 time (s)	TVector3 time (s)	T3/D3
$V_2 = -V_1$	0.054	0.171	3.17
$V_3 = V_1 + V_2$	0.092	0.191	2.08
$V_3 += V_1$	0.056	0.048	0.86
$V_3 = V_1 - V_2$	0.092	0.194	2.11
$V_2 = k V_1$	0.054	0.174	3.24
$V_2 *= k$	0.030	0.018	0.60
$V_3 = V_1 \times V_2$	0.107	0.193	1.80

Many 3-vector operations can be done 2-3 times faster with SIMD instructions...

Matrix operation benchmarks

- Compare Root-based matrices (TMatrixD) to custom SIMD-ized matrices (DMatrix, implemented in double precision)

- 10,000,000 “events” on 2.8 GHz Nehalem

Operation	TMatrixD(s)	DMatrix(s)	T/D
$A_{2 \times 2}^{-1}$	1.493	0.090	16.59
$A_{5 \times 5}^{-1}$	6.274	2.274	2.76
$A_{5 \times 5} + B_{5 \times 5}$	1.470	0.609	2.41
$A_{5 \times 5} B_{5 \times 5}$	3.821	0.974	3.82
$A_{5 \times 5}^T$	1.178	0.393	3.00
$A_{5 \times 5}^T B_{5 \times 5} A_{5 \times 5}$	7.892	1.857	4.25
$C_{1 \times 5} A_{5 \times 5} D_{5 \times 1}$	2.466	0.520	4.74
$E_{2 \times 5} A_{5 \times 5} F_{5 \times 2}$	3.630	0.651	5.58

- This is not a fair comparison due to the rank-checking in TMatrix → check with non-SIMD version of DMatrix:

Operation	Ratio of calculation rates = SIMD / non-SIMD
$A_{5 \times 5} + B_{5 \times 5}$	2.44
$A_{5 \times 5} B_{5 \times 5}$	1.66
$A_{5 \times 5}^T B_{5 \times 5} A_{5 \times 5}$	1.81

Application: Track fitting with Kalman Filter

- Track described by **5-parameter state vector** at each point along its path
- State vector **S** and covariance matrix **C** propagated step-by-step toward target

$$\begin{aligned} \mathbf{S}_k^{k-1} &= \mathbf{S}_k^{k-1,0} + \mathbf{F}_k (\mathbf{S}_{k-1}^{k-1} - \mathbf{S}_{k-1}^{k-1,0}) \\ \mathbf{C}_k^{k-1} &= \mathbf{F}_{k-1} \mathbf{C}_{k-1}^{k-1} \mathbf{F}_{k-1}^T + \mathbf{Q}_{k-1} \end{aligned}$$

- Account for **multiple scattering (Q)** and **energy loss** at each step
- Jacobian matrix (F) elements computed analytically
- Use measurements to update state vector

$$\begin{aligned} \mathbf{K}_k &= \mathbf{C}_k^{k-1} \mathbf{H}_k^T [\mathbf{V}_k + \mathbf{H}_k \mathbf{C}_k^{k-1} \mathbf{H}_k^T]^{-1} \\ \mathbf{S}_k^k &= \mathbf{S}_k^{k-1} + \mathbf{K}_k (m_k - h(\mathbf{S}_k^{k-1})) \\ \mathbf{C}_k^k &= \mathbf{C}_k^{k-1} - \mathbf{K}_k \mathbf{H}_k \mathbf{C}_k^{k-1} \end{aligned}$$

The many matrix operations can be SIMD-ized...

Reconstruction rates

- Generated 50000 events, pass through GlueX analysis chain
 - Use simulation of GlueX detector and GlueX reconstruction code
 - (more about these in the GlueX Tutorial session on Friday...)
- 4 threads on 2.8 GHz Nehalem

Topology	Kalman(Hz)	KalmanSIMD(Hz)
π^+ (particle gun)	72.7	101.4
π^- (particle gun)	138.3	188.4
ρ (particle gun)	81.2	112.5
$\rho\rho$	24.9	32.8
$n\pi^+\pi^+\pi^-$	20.8	26.5
$\rho b_1^+\pi^-$	12.4	15.8

Future prospects

- Bit depth for SIMD registers will be increased from 128 bits to 256 bits (4 doubles) or more (512,1024, ...?)
 - Intel/AMD machines in 2011?
- Relaxation of data alignment requirements?
- More arithmetic operations will be added
 - Fused Multiple-and-Add: $d = a + b \times c$

References

- There are several useful articles on Wikipedia:

1) http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

2) <http://en.wikipedia.org/wiki/SSE2>

3) <http://en.wikipedia.org/wiki/SSE3>

4) <http://en.wikipedia.org/wiki/SSSE3>

5) <http://en.wikipedia.org/wiki/SSE4>

- The most comprehensive compendium of the C intrinsics for SSE/SSE2 I've found is on the Microsoft web site:

• <http://msdn.microsoft.com/en-us/library/y0dh78ez%28v=VS.90%29.aspx>