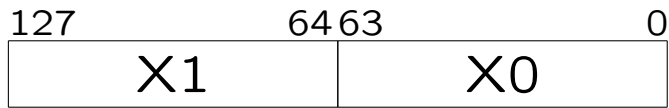


1. SIMD Vectorization

Single Instruction Multiple Data: one instruction operates on a vector of data

- For x86 and PPC, vector length is 128 bits / 16 bytes
- Therefore, can operate on 4 single precision values at a time
 - Must unroll loop by ≥ 4 to apply!
- We will first discuss SSE (SSE1) for single precision
 - Has a few inst that interpret vector as int
- x86 SIMD history:
 1. MMX (64bit int ops aliased to x87 regs)
 2. 3DNow! (AMD floats, non-IEEE)
 3. SSE/SSE1 : SIMD for floats
 4. **SSE2** : SIMD for doubles
 5. SSE3 : SIMD for complex & cleanup

2. Vector Registers (SSE2)



double prec fp words

127	0
xmm7		
xmm6		
xmm5		
xmm4		
xmm3		
xmm2		
xmm1		
xmm0		

Vector Registers

- Register file, not stack
- Each vector reg contains 2 64 bit (double precision) values
- All `xmm` registers *caller*-saved
- `ST(0)` still return value for functions!
→ Must write to mem, `f1d1`
- Can read/write only `X0` directly to scalar
→ Must shuffle vectors around, and reduce to scalar in `X0`
- Same regs used by SSE1/2/3
- Not aliased with `ST(x)`
→ So far, VPU is overlapped with FPU
- Do not mix single & double prec vector operations!

3. Common SSE2 Data Movement Instructions

Mnemonic	Operands	Action
<code>movapd</code>	<code>m16rs, rd</code>	<code>rd[0:1] = m16rs[0:1]</code>
<code>movapd</code>	<code>rs, m16rd</code>	<code>m16rd[0:1] = rs[0:1]</code>
<code>movupd</code>	<code>mrs, rd</code>	<code>rd[0:1] = mrs[0:1]</code>
<code>movupd</code>	<code>rs, mrd</code>	<code>mrd[0:1] = rs[0:1]</code>
<code>movsd</code>	<code>ms, rd</code>	<code>rd[0] = ms[0]; rd[1] = 0.0</code>
<code>movsd</code>	<code>rs, mrd</code>	<code>rmd[0] = rs[0]</code>
<code>movsd</code>	<code>rs, rd</code>	<code>rd[0] = rs[0];</code>
<code>movlpd</code>	<code>rs, md</code>	<code>m[0] = rs[0]</code>
<code>movlpd</code>	<code>ms, rd</code>	<code>rs[0] = m[0]</code>
<code>movhpd</code>	<code>rs, md</code>	<code>m[0] = rs[1]</code>
<code>movhpd</code>	<code>ms, rd</code>	<code>rs[1] = m[0]</code>
<code>movntpd</code>	<code>rs, m16</code>	<code>m16[0:1] = rs[0:1], do not cache m16</code>
<code>movmskpd</code>	<code>xmm, ireg</code>	Copy sign bits of xmm's doubles to low 2 bits of ireg, zero rest of ireg

- **m16**: 16-byte (vector word) aligned memory address
- `movlpd` from mem often cheaper than `movsd` if no need to zero upper

4. SSE2 Permute Operations

Mnemonic	Operands	Action
<code>unpcklpd</code>	<code>mrs, rd</code>	<code>rd[0]=rd[0]; rd[1]=mrs[0];</code>
<code>unpckhps</code>	<code>mrs, rd</code>	<code>rd[0]=rd[1]; rd[1]=mrs[1];</code>
<code>shufpd</code>	<code>imm8, rs, rd</code>	<code>imm8 = 0b00000000ba</code> <code>rd[0]=rd[a]; rd[1]=rs[b];</code>

5. SSE2 Computational Operations

Mnemonic	Operands	Action
andpd	m16rs, rd	$rd[0:1] \&= m16rs[0:1]$
andnpd	m16rs, rd	$rd[0:1] = (\sim rd[0:1]) \& m16rs[0:1]$
orpd	m16rs, rd	$rd[0:1] = m16rs[0:1]$
xorpd	m16rs, rd	$rd[0:1] \hat{=} m16rs[0:1]$ (zero!)
addpd	m16rs, rd	$rd[0:1] += m16rs[0:1]$
addsd	mrs, rd	$rd[0] += mrs[0]$
subpd	m16rs, rd	$rd[0:1] -= m16rs[0:1]$
subsd	mrs, rd	$rd[0] -= mrs[0]$
mulpd	m16rs, rd	$rd[0:1] *= m16rs[0:1]$
mulsd	mrs, rd	$rd[0] *= m16rs[0]$
divpd	m16rs, rd	$rd[0:1] /= m16rs[0:1]$
divsd	mrs, rd	$rd[0] /= m16rs[0]$
maxpd	m16rs, rd	$rd[0:1] = \text{MAX}(rd[0:1], m16rs[0:1])$
maxsd	mrs, rd	$rd[0] += \text{MAX}(rd[0], m16rs[0])$
minpd	m16rs, rd	$rd[0:1] = \text{MIN}(rd[0:1], m16rs[0:1])$
minsd	mrs, rd	$rd[0] += \text{MIN}(rd[0], m16rs[0])$

6. Vector Comparisons

Mnem	Operands	Action
cmppd	imm8, m16rs, rd	rd ← all 1s if comparison is true, else all zeros
cmpXXpd	m16rs, rd	rd ← all 1s if comparison is true, else all zeros
movmskps	xmm, ireg	Copy sign bits of xmm's floats to low 2 bits of ireg, zero rest of ireg
comisd	mrs, rd	set ICC as shown in table on right

COMP	XX	imm8	NaN
rd = m16rs	eq	0	0
rd < m16rs	lt	1	0
rd ≤ m16rs	le	2	0
rd or m16rs is NaN	unord	3	1
rd ≠ m16rs	neq	4	1
rd ⋈ m16rs	nlt	5	1
rd ⋈ m16rs	nle	6	1

⇒ Use movmskpd to get to ireg

⇒ Then use bt and test to branch

rd[0] > mrs[0]	ZF=0, PF=0, CF=0
rd[0] < mrs[0]	ZF=0, PF=0, CF=1
rd[0] = mrs[0]	ZF=1, PF=0, CF=0
NaN	ZF=1, PF=1, CF=1

comisd result

7. SSE2 Integer Operations

- All mmx instructions ported to xmm instructions
 - doubles the number of operands
- Has additional dq (double quadword) instructions (64-bit words)
- Has pshufd which may be mixed with fp code, which can move any combo of 32 bit words from src to dest
- Has pinsrw, our use is moving low 16 bits of constructed ival from ireg to xmm reg (can do other things too)

Mnem	Operands	Action
pshufd	imm8, m16rs, rd	imm8 = ddcbbbaa (binary) rd[0] = m16rs[aa]; rd[1] = m16rs[bb]; rd[2] = m16rs[cc]; rd[3] = m16rs[dd]
pinsrw	imm8, r32, xmmd	xmmd[aaa*4:aaa*4+15] = r32[0:15] (in bits) imm8=00000aaa

8. Simple SSE2 DAXPY

```
#define ralp    %xmm0
#define rX     %xmm1
#define rY     %xmm2
#define N      %eax
#define X      %edx
#define Y      %ecx
#define II     %edi
#define FSIZE  8
#define Y_b    %cl
#define II_w   %di
/*           4           8           16
void ATL_UAXPY(int N, double alpha, TYPE *X,
               int incX, TYPE *Y, int incY)
               20          24          28 */

.text
.global ATL_UAXPY
ATL_UAXPY:
    subl    $FSIZE, %esp
    movl    %edi, (%esp)
    movl    FSIZE+4(%esp), N
    movsd   FSIZE+8(%esp), ralp    # ralp = {0.0,alp}
    unpcklpd    ralp, ralp    # ralp = {alp,alp}
    movl    FSIZE+16(%esp), X
    movl    FSIZE+24(%esp), Y
    test   $0x0F, Y_b
    jnz    FORCE_Y_ALIGN
```

```
YALIGNED:
#   for (i=N2=(N/2)*2; i; i--)
#       Y[N2-i] = alpha * X[N2-i]
    cmpl    $2, N
    jl     CLEANST
    movl    N, II
    andw   0xFFFFE, II_w    # II = (N/2)*2
    lea    (X,II,8), X      # X += N2;
    lea    (Y,II,8), Y
    sub    II, N
    neg    II
LOOP2:
    movupd (X,II,8), rX
    movapd (Y,II,8), rY
    mulpd  ralp, rX
    addpd  rX, rY
    movapd rY, (Y,II,8)
    addl   $2, II
    jnz    LOOP2
CLEANST:
    cmpl    $0, N
    jnz    CLEANUP
DONE:
    movl    (%esp), %edi
    addl    $FSIZE, %esp
    ret
```

9. Simple SSE2 DAXPY (continued)

CLEANUP:

```
    lea    (X, N, 8), X
    lea    (Y, N, 8), Y
    movl   N, II
    neg    II
```

CULOOOP:

```
    movsd  (X,II,8), rX
    mulsd  ralp, rX
    addsd  (Y,II,8), rX
    movsd  rX, (Y,II,8)
    inc    II
    jnz    CULOOOP
    jmp    DONE
```

FORCE_Y_ALIGN:

```
    cmp    $0, N
    je     DONE
    movsd  (X), rX
    mulsd  ralp, rX
    addsd  (Y), rX
    movsd  rX, (Y)
    addl   $8, X
    addl   $8, Y
    dec    N
    jnz    YALIGNED
    jmp    DONE
```