Sections 3.6 – 3.8

Subword Parallelism, MMX, SSE, AVX in x86, Going Faster

Subword Parallellism

- History of support for small integers
  - Bytes, half words
    - Only load and store instructions
- Popularity of multimedia applications
  - 8-bit color, 16-bit audio
  - Arithmetic instructions to support narrower operations
    - Operate in parallel
      - Wide registers and multiple functional units

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction Multiple Data (SIMD)

The Intel x86 ISA (참고자료)

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

The Intel x86 ISA (참고자료)

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added 57 MMX (Multi-Media eXtension) instructions (1997)
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added 70 SSE (Streaming SIMD Extensions) instructions and associated registers
    - Cache prefetch instructions, streaming store instructions
  - Pentium 4 (2001)
    - New microarchitecture
    - Added 144 SSE2 instructions

The Intel x86 ISA (참고자료)

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added 13 SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - AVX: Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

MMX and SSE

- MMX
  - Use (64 bits of) existing FP registers
  - Small integer data types (8bit color, 16bit audio)
    - Packed byte (8×8), packed word (4×16), (2×32), (1×64)
- SSE
  - Add new 8 × 128-bit FP registers (XMM0-XMM7)
  - Packed single precision FP (4×32)

SSE2, …, AVX

- SSE2
  - Adds 8 × 128-bit registers
    - Generalize SSE
  - Can be used for multiple FP/INT operands
    - 2 × 64-bit FP (addpd %xmm0, %xmm4)
    - 4 × 32-bit FP (addps %xmm0, %xmm4)
    - 2 × 64-bit, 4 × 32-bit, 8 × 16-bit,18 × 8-bit INT
- AVX (advanced vector extension)
  - Double the width of registers
    - e.g., 4 × 64-bit double precision FP

SSE/SSE2 (skip)

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm,mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm,mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm,mem/xmm	
	DIV{SS/PS/SD/PD} xmm,mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

Going Faster

- Demonstrate the performance benefit of adapting software to the underlying hardware

Going Faster

- With and without AVX, Intel Core i7
- DGEMM (Double Precision General Matrix Multiply)
  - AVX version 3.85 times fast
    - 4 double precision FP operations in parallel
- With Turbo mode turned off
  - Turbo mode
    - Temporarily run at higher clock until chip gets too hot
    - Particularly useful when using only single core of multicore chip

Matrix Multiply, DGEMM (참고자료)

- Unoptimized code:

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {
6.     double cij = C[i+j*n]; /* cij = C[i][j] */
7.     for(int k = 0; k < n; k++ )
8.      cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.     C[i+j*n] = cij; /* C[i][j] = cij */
10.   }
11. }
```

Matrix Multiply, DGEMM (참고자료)

- x86 assembly code:

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx         # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
   element of A
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

Matrix Multiply, DGEMM (참고자료)

- Optimized C code:

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
6.    __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j]
   */
7.    for( int k = 0; k < n; k++ )
8.     c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.              _mm256_broadcast_sd(B+k+j*n)));
11.   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.  }
13. }
```

Matrix Multiply, DGEMM (참고자료)

- Optimized x86 assembly code:

```
1. vmovapd (%r11),%ymm0          # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx                 # register %rcx = %rbx
3. xor %eax,%eax                 # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax                 # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx                  # register %rcx = %rcx + %r9
8. cmp %r10,%rax                 # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0   # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>          # jump if not %r10 != %rax
11. add $0x1,%esi                # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)         # Store %ymm0 into 4 C elements
```

C Intrinsics

- Intrinsic function
  - Function available for use in a given programming language whose implementation is specially handled by compiler
  - Often used to explicitly implement vectorization and parallelization in languages which do not address such constructs
    - E.g., MMX, SSE, OpenMP

x86 FP Architecture (skip)

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions (skip)

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

Related to Chapter 6 (especially Section 6.3)