4.12 Going faster: Instruction-level parallelism and matrix multiply

This section has been set as optional by your instructor.

Returning to the DGEMM example from COD Chapter 3 (Arithmetic for Computers), we can see the impact of instruction-level parallelism by unrolling the loop so that the multiple-issue, out-of-order execution processor has more instructions to work with. The figure below shows the unrolled version of COD Figure 3.24 (Optimized C version of DGEMM using C intrinsics ...), which contains the C intrinsics to produce the AVX instructions.

Figure 4.12.1: Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86 and loop unrolling to create more opportunities for instruction-level parallelism (COD Figure 4.78).

COD Figure 4.79 (The x86 assembly language for ...) shows the assembly language produced by the compiler for the inner loop, which unrolls the three for-loop bodies to expose instruction-level parallelis

```
1  #include <x86intrin.h>
2  #define UNROLL (4)
3
4  void dgemm (int n, double* A, double* B, double* C)
5  {
6     for ( int i = 0; i < n; i += UNROLL * 4 )
7        for ( int j = 0; j < n; j++ ) {
8           __m256d c[4];
9           for ( int x = 0; x < UNROLL; x++ )
10             c[x] = _mm256_load_pd(C + i + x * 4 + j * n);
11
12          for( int k = 0; k < n; k++ )
13          {
14             __m256d b = _mm256_broadcast_sd(B + k + j * n);
15             for (int x = 0; x < UNROLL; x++)
16                c[x] = _mm256_add_pd(c[x],
17                   _mm256_mul_pd(_mm256_load_pd(A + n * k + x * 4 + i),
18          }
19
20          for ( int x = 0; x < UNROLL; x++ )
21             _mm256_store_pd(C + i + x * 4 + j * n, c[x]);
22       }
23 }
```

Like the unrolling example in COD Figure 4.69 (The unrolled and scheduled code ...) above, we are going to unroll the loop four times. Rather than manually unrolling the loop in C by making four copies of each of the intrinsics in COD Figure 3.24 (Optimized C version of DGEMM using C intrinsics ...), we can rely on the gcc compiler to do the unrolling at —O3 optimization. (We use the constant `UNROLL` in the C code to control the amount of unrolling in case we want to try other values.) We surround each intrinsic with a simple *for* loop with four iterations (lines 9, 15, and 20) and replace the scalar `C0` in COD Figure 3.24 (Optimized C version of DGEMM using C intrinsics ...) with a four-element array `c[]` (lines 8, 10, 16, and 21).

The figure below shows the assembly language output of the unrolled code. As expected, in the figure below there are four versions of each of the AVX instructions in COD Figure 3.25 (The x86 assembly language for the body of the nested loops ...), with one exception. We only need one copy of the `vbroadcastsd` instruction, since we can use the four copies of the B element in register `%ymm0` repeatedly throughout the loop. Thus, the five AVX instructions in COD Figure 3.25 (The x86 assembly language for the body of the nested loops ...) become 17 in the figure below, and the seven integer instructions appear in both, although the constants and addressing changes to account for the unrolling. Hence, despite unrolling four times, the number of instructions in the body of the loop only doubles: from 12 to 24.

Figure 4.12.2: The x86 assembly language for the body of the nested loops generated by compiling the unrolled C code in the figure above (COD Figure 4.79).

```
 1 vmovapd (%r11), %ymm4                 // Load 4 elements of C into %ymm4
 2 mov %rbx, %rax                        // register %rax = %rbx
 3 xor %ecx, %ecx                        // register %ecx = 0
 4 vmovapd 0x20(%r11), %ymm3             // Load 4 elements of C into %ymm3
 5 vmovapd 0x40(%r11), %ymm2             // Load 4 elements of C into %ymm2
 6 vmovapd 0x60(%r11), %ymm1             // Load 4 elements of C into %ymm1
 7 vbroadcastsd (%rcx, %r9, 1), %ymm0    // Make 4 copies of B element
 8 add $0x8, %rcx                        // register %rcx = %rcx + 8
 9 vmulpd (%rax), %ymm0, %ymm5           // Parallel mul %ymm1, 4 A element
10 vaddpd %ymm5, %ymm4, %ymm4           // Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax), %ymm0, %ymm5      // Parallel mul %ymm1, 4 A elements
12 vaddpd %ymm5, %ymm3, %ymm3           // Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax), %ymm0, %ymm5      // Parallel mul %ymm1, 4 A elements
14 vmulpd 0x40(%rax), %ymm0, %ymm5      // Parallel mul %ymm1, 4 A elements
15 add %r8, %rax                        // register %rax = %rax + %r8
16 cmp %r8, %rax                        // compare %r8 to %rax
17 vaddpd %ymm5, %ymm2, %ymm2           // Parallel add %ymm5, %ymm2
18 vaddpd %ymm0, %ymm1, %ymm1           // Parallel add %ymm0, %ymm1
19 jne 68 <dgemm + 0x68>                // jump if not %r8 != %rax
20 add $0x1, %esi                       // register % esi = % esi + 1
```

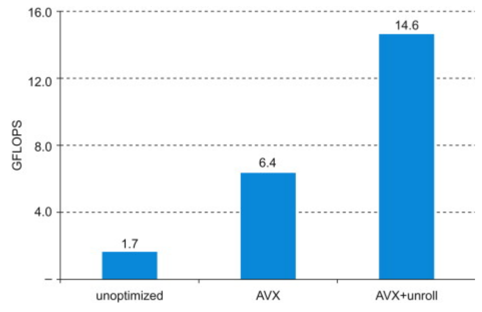
```
vmovapd %ymm4,(%r11)              // Store %ymm4 into 4 C elements
vmovapd %ymm3, 0x20(%r11)         // Store %ymm3 into 4 C elements
vmovapd %ymm2, 0x40(%r11)         // Store %ymm2 into 4 C elements
vmovapd %ymm1, 0x60(%r11)         // Store %ymm1 into 4 C elements
```

The figure below shows the performance increase DGEMM for 32x32 matrices in going from unoptimized to AVX and then to AVX with unrolling. Unrolling more than doubles performance, going from 6.4 GFLOPS to 14.6 GFLOPS. Optimizations for **subword parallelism** and **instruction-level parallelism** result in an overall speedup of 8.59 versus the unoptimized DGEMM in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply ...).

PARALLELISM

Figure 4.12.3: Performance of three versions of DGEMM for 32x32 matrices (COD Figure 4.80).

Subword parallelism and instruction-level parallelism have led to speedup of almost a factor of 9 over the unoptimized code in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply ...).



Elaboration

As mentioned in the Elaboration in COD Section 3.9 (Going faster: Subword parallelism and matrix multiply), these results are with Turbo mode turned off. If we turn it on, like in COD Chapter 3 (Arithmetic for Computers), we improve all the results by the temporary increase in the clock rate of 3.3/2.6 = 1.27 to 2.1 GFLOPS for unoptimized DGEMM, 8.1 GFLOPS with AVX, and 18.6 GFLOPS with unrolling and AVX. As mentioned in COD Section 3.9 (Going faster: Subword parallelism and matrix multiply), Turbo mode works particularly well in this case because it is using only a single core of an eight-core chip.

Elaboration

There are no pipeline stalls despite the reuse of register %ymm5 in lines 9 to 17 of COD Figure 4.79 (The x86 assembly language for the body of the nested loops ...) because the Intel Core i7 pipeline renames the registers.

PARTICIPATION ACTIVITY 4.12.1: Check yourself: Intel Core i7 characteristics.

1) The Intel Core i7 uses a multiple-issue pipeline to directly execute x86 instructions.
   - ○ True
   - ○ False

2) Both the A53 and the Core i7 use dynamic multiple issue.
   - ○ True
   - ○ False

3) The Core i7 microarchitecture has many more registers than x86 requires.
   - ○ True
   - ○ False

4) The Intel Core i7 uses less than half the pipeline stages of the earlier Intel Pentium 4 Prescott (see COD Figure 4.73 (Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power)).
   - ○ True
   - ○ False

Provide feedback on this section