3.8 Going faster: Subword parallelism and matrix multiply

This section has been set as optional by your instructor.

To demonstrate the performance impact of subword parallelism, we'll run the same code on the Intel Core i7 first without AVX and then with it. The figure below shows an unoptimized version of a matrix-matrix multiply written in C. As we saw in COD Section 3.5 (Floating point), this program is commonly called *DGEMM*, which stands for Double precision GEneral Matrix Multiply. Starting with this edition, we have added a new section entitled "Going Faster" to demonstrate the performance benefit of adapting software to the underlying hardware, in this case the Sandy Bridge version of the Intel Core i7 microprocessor. This new section in COD Chapters 3 (Arithmetic for Computers), 4 (The Processor), 5 (Large and Fast: Exploiting Memory Hierarchy), and 6 (Parallel Processor from Client to Cloud) will incrementally improve DGEMM performance using the ideas that each chapter introduces.

Figure 3.8.1: Unoptimized C version of a double precision matrix multiply, widely know as DGEMM for Double-precision GEneral Matrix Multiply (COD Figure 3.22).

Because we are passing the matrix dimension as the parameter `n`, this version of DGEMM uses single-dimensional versions of matrices `C`, `A`, and `B` and address arithmetic to get better performance instead of using the more intuitive two-dimensional arrays that we saw in COD Section 3.5 (Floating point). The comments remind us of this more intuitive notation.

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

The figure below shows the x86 assembly language output for the inner loop of COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply ...). The five floating-point instructions start with a `v` like the AVX instructions, but note that they use the XMM registers instead of YMM, and they include `sd` in the name, which stands for scalar double precision. We'll define the subword parallel instructions shortly.

Figure 3.8.2: The x86 assembly language for the body of the nested loops generated by compiling the unoptimized C code in the figure above (COD Figure 3.23).

Although it is dealing with just 64-bits of data, the compiler uses the AVX version of the instructions instead of SSE2 presumably so that it can use three address per instruction instead of two (see the *Elaboration* in COD Section 3.7 (Real stuff: Streaming SIMD extensions ...)).

```
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,%r11                 // register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)              // Store %xmm0 into C element
```

PARTICIPATION ACTIVITY 3.8.1: Unoptimized DGEMM example.

1) The `dgemm()` function operates on single dimensional arrays A[n], B[n], and C[n].
   - ○ True
   - ○ False

2) The resulting x86 assembly language output contains _____ floating-point instructions.

- 5
- 12

3) Each iteration of the resulting x86 assembly language calculates C = C + A * B for one element of C.
   - True
   - False

While compiler writers may eventually be able to produce high-quality code routinely that uses the AVX instructions of the x86, for now we must "cheat" by using C intrinsics that more or less tell the compiler exactly how to produce good code. The figure below shows the enhanced version of COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …) for which the Gnu C compiler produces AVX code.

Figure 3.8.3: Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86 (COD Figure 3.24).

The next figure shows the assembly language produced by the compiler for the inner loop.

```
1.  //include <x86intrin.h>
2.  void dgemm (size_t n, double* A, double* B, double* C)
3.  {
4.    for ( size_t i = 0; i < n; i+=4 )
5.      for ( size_t j = 0; j < n; j++ ) {
6.        __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.        for( size_t 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. }
```

The figure below shows annotated x86 code that is the output of compiling using gcc with the -O3 level of optimization.

Figure 3.8.4: The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in the above figure (COD Figure 3.25).

Note the similarities to COD Figure 3.23 (The x86 assembly language for the body of the nested loops …), with the primary difference being that the five floating-point operations are now using YMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision.

```
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
```

The declaration on line 6 of COD Figure 3.24 (Optimized C version of DGEMM using C intrinsics …) uses the `__m256d` data type, which tells the compiler the variable will hold four double-precision floating-point values. The intrinsic `_mm256_load_pd()` also on line 6 uses AVX instructions to load four double-precision floating-point numbers in parallel (`_pd`) from the matrix `C` into `c0`. The address calculation `C + i + j * n` on line 6 represents element `C[i + j * n]`. Symmetrically, the final step on line 11 uses the intrinsic `_mm256_store_pd()` to store four double-precision floating-point numbers from `c0` into the matrix `C`. As we're going through four elements each iteration, the outer *for* loop on line 4 increments `i` by 4 instead of by 1 as on line 3 of COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …).

Inside the loops, on line 9 we first load four elements of `A` again using `_mm256_ load_pd()`. To multiply these elements by one element of `B`, on line 10 we first use the intrinsic `_mm256_broadcast_sd()`, which makes four identical copies of the scalar double precision number—in this case an element of `B`—in one of the YMM registers. We then use `_mm256_mul_pd()` on line 9 to multiply the four double-precision results in parallel. Finally, `_mm256_add_pd()` on line 8 adds the four products to the four sums in `c0`.

COD Figure 3.25 (The x86 assembly language for the body of the nested loops …) shows resulting x86 code for the body of the inner loops produced by the compiler. You can see the five AVX instructions—they all start with `v` and four of the five use `pd` for parallel double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in COD Figure 3.23 (The x86 assembly language for the body of the nested loops …) above: both use 12 instructions, the integer instructions are nearly identical (but different registers), and the floating-point instruction differences are generally just going from *scalar double* (`sd`) using XMM registers to *parallel double* (`pd`) with YMM registers. The one exception is line 4 of COD Figure 3.25 (The x86 assembly language for the body of the nested loops …). Every element of `A` must be multiplied by one element of `B`. One solution is to place four identical copies of the 64-bit `B` element side-by-side into the 256-bit YMM register, which is just what the instruction `vbroadcastsd` does.

For matrices of dimensions of 32 by 32, the unoptimized DGEMM in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …) runs at 1.7 GigaFLOPS (FLoating point Operations Per Second) on one core of a 2.6 GHz Intel Core i7 (Sandy Bridge). The optimized code in COD Figure 3.24 (Optimized C version of DGEMM using C intrinsics …) performs at 6.4 GigaFLOPS. The AVX version is 3.85 times as fast, which is very close to the factor of 4.0 increase that you might hope for from performing four times as many operations at a time by using **subword parallelism**.

Elaboration

As mentioned in the Elaboration in COD Section 1.6 (Performance), Intel offers Turbo mode that temporarily runs at a higher clock rate until the chip gets too hot. This Intel Core i7 (Sandy Bridge) can increase from 2.6 GHz to 3.3 GHz in Turbo mode. The results above are with Turbo mode turned off. If we turn it on, we improve all the results by the increase in the clock rate of 3.3/2.6 = 1.27 to 2.1 GFLOPS for unoptimized DGEMM and 8.1 GFLOPS with AVX. Turbo mode works particularly well when using only a single core of an eight-core chip, as in this case, as it lets that single core use much more than its fair share of power since the other cores are idle.

PARTICIPATION ACTIVITY 3.8.2: Optimized DGEMM example.

1) ____ is a C intrinsic.
   - `_mm256_load_pd()`
   - `dgemm()`
   - `vbroadcastsd`

2) Complete the outer loop of the optimized C version of DGEMM:

```
for (int i = 0; i < n; _____
)
```

   - ++i
   - i+=4

3) Compiling the optimized C code replaces most of the x86 floating-point instructions from a `sd` variation to a ____ variation of the instruction.
   - `pd`
   - `ymm`
   - `__m256d`

Provide feedback on this section