6.12 Going faster: Multiple processors and matrix multiply

(Original section[1])

This section is the final and largest step in our incremental performance journey of adapting DGEMM to the underlying hardware of the Intel Core i7 (Sandy Bridge). Each Core i7 has eight cores, and the computer we have been using has two Core i7s. Thus, we have 16 cores on which to run DGEMM.

The figure below shows the OpenMP version of DGEMM that utilizes those cores. Note that line 30 is the *single* line added to COD Figure 5.47 (Optimized C version of DGEMM using cache blocking) to make this code run on multiple processors: an OpenMP pragma that tells the compiler to use multiple threads in the outermost loop. It tells the computer to spread the work of the outermost loop across all the threads.

Figure 6.12.1: OpenMP version of DGEMM (COD Figure 6.25).

OpenMP version of DGEMM from COD Figure 5.47 (Optimized C version of DGEMM using cache block

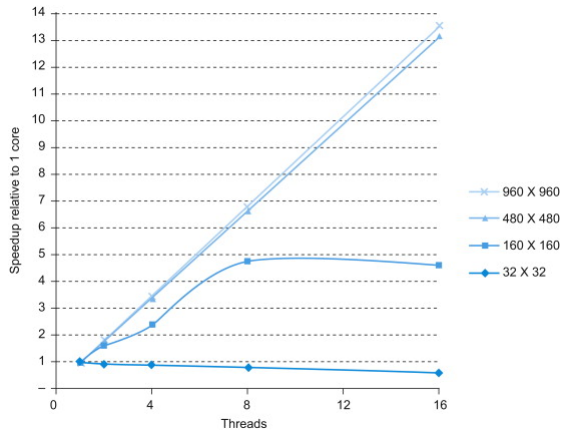
Line 30 is the only OpenMP code, making the outermost for loop operate in parallel. This line is the onl from COD Figure 5.47 (Optimized C version of DGEMM using cache blocking).

```
1  #include <x86intrin.h>
2  #define UNROLL (4)
3  #define BLOCKSIZE 32
4  void do_block (int n, int si, int sj, int sk,
5                 double *A, double *B, double *C)
6  {
7     for ( int i = si; i < si + BLOCKSIZE; i += UNROLL * 4 )
8        for ( int j = sj; j < sj + BLOCKSIZE; j++ ) {
9           __m256d c[4];
10          for ( int x = 0; x < UNROLL; x++ )
11             c[x] = _mm256_load_pd(C + i + x * 4 + j * n);
12          /* c[x] = C[i][j] */
13          for( int k = sk; k < sk + BLOCKSIZE; k++ )
14          {
15             __m256d b = _mm256_broadcast_sd(B + k + j * n);
16          /* b = B[k][j] */
17             for (int x = 0; x < UNROLL; x++)
18                c[x] = _mm256_add_pd(c[x],               /* c[x] += A[i]
19                       _mm256_mul_pd(_mm256_load_pd(A + n * k + x * 4 +
20          }
21
22          for ( int x = 0; x < UNROLL; x++ )
23             _mm256_store_pd(C + i + x * 4 + j * n, c[x]);
24             /* C[i][j] = c[x] */
25       }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30 #pragma omp parallel for
31    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
32       for ( int si = 0; si < n; si += BLOCKSIZE )
33          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
34             do_block(n, si, sj, sk, A, B, C);
35 }
```

The figure below plots a classic multiprocessor speed-up graph, showing the performance improvement versus a single thread as the number of threads increase. This graph makes it easy to see the challenges of strong scaling versus weak scaling. When everything fits in the first-level data cache, as is the case for 32 × 32 matrices, adding threads actually hurts performance. The 16-threaded version of DGEMM is almost half as fast as the single-threaded version in this case. In contrast, the two largest matrices get a 14 × speedup from 16 threads, and hence the classic two "up and to the right" lines in the figure below.

Figure 6.12.2: Performance improvements relative to a single thread as the number of threads increase (COD Figure 6.26).

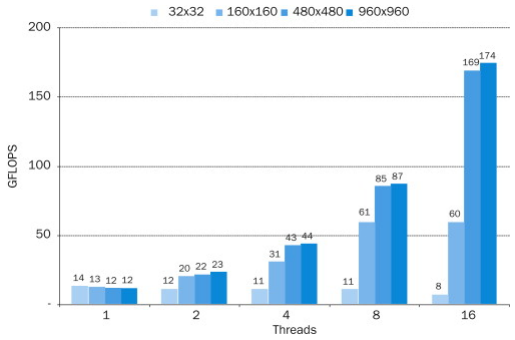
The most honest way to present such graphs is to make performance relative to the best version of a single processor program, which we did. This plot is relative to the performance of the code in COD Figure 5.47 (Optimized C version of DGEMM using cache blocking) *without* including OpenMP pragmas.



The figure below shows the absolute performance increase as we increase the number of threads from one to 16. DGEMM now operates at 174 GLOPS for 960 × 960 matrices. As our unoptimized C version of DGEMM in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …) ran this code at just 0.8 GFLOPS, the optimizations in COD Chapter 3 (Arithmetic for Computers) to 6 (Parallel Processor from Client to Cloud) that tailor the code to the underlying hardware result in a speed-up of over 200 times!

Figure 6.12.3: DGEMM performance versus the number of threads for four matrix sizes (COD Figure 6.27).

The performance improvement compared unoptimized code in COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …) for the 960 x 960 matrix with 16 threads is an astounding 212 times faster!



Next up is our warnings of the fallacies and pitfalls of multiprocessing. The computer architecture graveyard is filled with parallel processing projects that have ignored them.

Elaboration

These results are with Turbo mode turned off. We are using a dual chip system in this system, so not surprisingly, we can get the full Turbo speed-up (3.3/2.6 = 1.27) with either one thread (only one core on one of the chips) or two threads (one core per chip). As we increase the number of threads and hence the number of active cores, the benefit of Turbo mode decreases, as there is less of the power budget to spend on the active cores. For four threads the average Turbo speed-up is 1.23, for eight it is 1.13, and for 16 it is 1.11.

Elaboration

Although the Sandy Bridge supports two hardware threads per core, we do not get more performance from 32 threads. The reason is that a single AVX hardware is shared between the two threads multiplexed onto one core, so assigning two threads per core actually hurts performance due to the multiplexing overhead.

(*1) This section is in original form.

Provide feedback on this section