COMP2006 Computer Organization
Lab 11: Cache and Multicore Optimization

Optimization DGEMM on Desktop Computer

```
void dgemm_avx(int n, double *A, double *B, double *C)
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j += 4)
        {
            __m256d c0 = _mm256_load_pd(C + i * n + j);
            for (k = 0; k < n; k++)
            {
                __m256d vA = _mm256_broadcast_sd(A + k + i * n);
                __m256d vB = _mm256_load_pd(B + j + k * n);
                __m256d vC = _mm256_mul_pd(vA, vB);
                c0 = _mm256_add_pd(c0, vC);
            }

            _mm256_store_pd(C + j + i * n, c0);
        }
}
```

In the last lab, we learned the *AVX Intrinsics* that makes the **dgemm** function run faster on a desktop computer. We now are going to learn how to apply different optimization techniques on the AVX optimized version of the **dgemm** function (the code above).

AVX with Unrolling

In the unrolled version, we replicate loop body to expose more parallelism. It can reduce unnecessary loop overhead instructions, e.g. pointer arithmetic, end of loop tests on each iteration, etc. For each iteration, the value of i is increased by **4*UNROLL** instead of **4**. Also, the loop body is repeated four times because the value of **UNROLL** is equal to **4**.

```
void dgemm_avx_unroll(int n, double* A, double* B, double* C) {
    int i, j, k;
    for(i = 0; i < n; i+=4*UNROLL)
        for(j = 0; j < n; j++) {
            __m256d c[4] ={ _mm256_load_pd(C+i+j*n), _mm256_load_pd(C+i+4+j*n),
                            _mm256_load_pd(C+i+8+j*n), _mm256_load_pd(C+i+12+j*n) };
            for(k = 0; k < n; k++) {
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                c[0] = _mm256_add_pd(c[0],
                    _mm256_mul_pd(_mm256_load_pd(A+i+k*n), b));
                c[1] = _mm256_add_pd(c[1],
                    _mm256_mul_pd(_mm256_load_pd(A+i+k*n+4), b));
                c[2] = _mm256_add_pd(c[2],
                    _mm256_mul_pd(_mm256_load_pd(A+i+k*n+8), b));
                c[3] = _mm256_add_pd(c[3],
                    _mm256_mul_pd(_mm256_load_pd(A+i+k*n+12), b));
            }
            _mm256_store_pd(C+i+j*n, c[0]);
            _mm256_store_pd(C+i+j*n+4, c[1]);
            _mm256_store_pd(C+i+j*n+8, c[2]);
            _mm256_store_pd(C+i+j*n+12, c[3]);
        }
}
```

AVX with Unrolling and Cache Optimization

The cache version is developed upon the AVX with unrolling version.

```
void do_block(int n, int si, int sj, int sk, double* A, double* B, double* C) {
   int i, j, k;
   for(i = si; i < si+BLOCKSIZE; i+=UNROLL*4)
      for(int j = sj; j < sj+BLOCKSIZE; j++) {
         __m256d c[4] ={ _mm256_load_pd(C+i+j*n), _mm256_load_pd(C+i+4+j*n),
                         _mm256_load_pd(C+i+8+j*n), _mm256_load_pd(C+i+12+j*n) };

         for(k = sk; k < sk+BLOCKSIZE; k++) {
            __m256d b = _mm256_broadcast_sd(B+k+j*n);
            c[0] = _mm256_add_pd(c[0],
               _mm256_mul_pd(_mm256_load_pd(A+i+k*n), b));
            c[1] = _mm256_add_pd(c[1],
               _mm256_mul_pd(_mm256_load_pd(A+i+k*n+4), b));
            c[2] = _mm256_add_pd(c[2],
               _mm256_mul_pd(_mm256_load_pd(A+i+k*n+8), b));
            c[3] = _mm256_add_pd(c[3],
               _mm256_mul_pd(_mm256_load_pd(A+i+k*n+12), b));
         }
         _mm256_store_pd(C+i+j*n, c[0]);
         _mm256_store_pd(C+i+j*n+4, c[1]);
         _mm256_store_pd(C+i+j*n+8, c[2]);
         _mm256_store_pd(C+i+j*n+12, c[3]);
      }
}

void dgemm_avx_unroll_block(int n, double* A, double* B, double* C) {
   int si, sj, sk;
   for(sj = 0; sj < n; sj += BLOCKSIZE)
      for(si = 0; si < n; si += BLOCKSIZE)
         for(sk = 0; sk < n; sk += BLOCKSIZE)
            do_block(n, si, sj, sk, A, B, C);
}
```

AVX with Unrolling, Cache Optimization, and Multithreading

The multithreading version is developed upon the AVX with unrolling and cache optimization.

```
void dgemm_avx_unroll_block_multicore(int n, double* A, double* B, double* C) {
   int si, sj, sk;
   #pragma omp parallel for
   for(sj = 0; sj < n; sj += BLOCKSIZE)
      for(si = 0; si < n; si += BLOCKSIZE)
         for(sk = 0; sk < n; sk += BLOCKSIZE)
            do_block(n, si, sj, sk, A, B, C);
}
```

Checking Performances

Let’s download dgemm_dt.cpp from our course webpage. Then, compile and run it. Because it is a C++ program, we use **g++** to compile the program.

For Windows:

*Compile**: g++ -mavx -fopenmp dgemm_dt.cpp -o dgemm_dt***

*Run: **dgemm_dt***

For Macintosh, you need to install Xcode with libomp and llvm:

Installation:

1. In App Store, find Xcode and click “Get” to install it.
2. After the installation, run the following commands to install the libraries:
   - brew install libomp
   - brew install llvm

*Compile: **g++ -mavx -Xpreprocessor -fopenmp dgemm_dt.cpp -o dgemm_dt -lomp***

*Run: ./**dgemm_dt***

You will get the following result:

```
Enter the matrix width: 1024
DGEMM
Calculation time for 1024 x 1024 matrix: 15.207597 seconds
DGEMM AVX
Calculation time for 1024 x 1024 matrix: 6.213696 seconds
DGEMM AVX Unroll
calculation time for 1024 x 1024 matrix: 2.473862 seconds
DGEMM AVX Unrolling and Cache Optimization
Calculation time for 1024 x 1024 matrix: 1.147007 seconds
DGEMM AVX with Unrolling, Cache Optimization and Multithreading
Calculation time for 1024 x 1024 matrix: 0.280153 seconds
```

You may get different result. The calculation time is depended on the performance of your computer and the current loading of the processor.

Exercise

Run the **dgemm_dt** program repeatedly with the matrices of different sizes and write down the results in the worksheet.