Programming Assignment 1: REPORT

Optimization of Double Precision Floating Point Matrix Multiply Accumulator

- David Garcia, Rahul Godbole

The goal of this project was for us to optimize the matrix multiply accumulate operation C = C + A*B. All matrices are 64-bit double precision floating type. The computation is performed on a machine with 32kB L1 cache and 4MB L2 cache.

Following optimizations were tried out:

1. **Blocking (+)**: Chunks of sub matrices computed with varying block sizes.
Specialized matrix sizes used multiple blocking levels. Optimum Block Size for final implementation found to be blocks of 256 for the outermost loop, 16 for the middle loop. This was mostly a trial and error approach. The goal was for us to exploit spatial locality in the L1 cache based on the size of the blocks.
2. **Prefetching** (-): Did not seem to help us much. Compiler was already implementing some prefetching. Also, the HW prefetcher was doing its job.
3. **Transpose** (+): Allows us to access memory sequentially i.e. data in the same cache line.
4. **SSE2 intrinsics** (+): Maximum benefit seen with SIMD extension intrinsics. We used the _mm_load_pd, _mm_add_pd, _mm_mul_pd and _mm_set1_pd intrinsic functions.
5. **Compiler flags** (+): Several of these were used in the project.
(a) -funroll-loops: Looking at the Intel compiler reference guide, this option is OFF by default. In conjunction with -march=corei7, there was a slight degradation in performance - this is also acknowledged in the Intel compiler reference guide.
(b) -march=corei7: This option enables SSE, SSE2, SSE3 intrinsics. Hence, is required for the SSE2 optimizations to take effect.
(c) -std=gnu99: no effect, was hoping it would better enforce __restrict__
(d) -opt-args-in-regs=all: Not much of an impact. Causes function parameters to be passed in registers (instead of on the stack), where the function may or may not be in the compilation unit scope.
(e) -fprefetch-loop-arrays: increased the N=128 case to 4500 MFLOPS, decreased the rest
(f) --param l1-cache-line-size, l1-cache-size, l2-cache-size: no effect

6. Strassen Algorithm: This algorithm reduces the number of multiplications in exchange for additions-- for block sizes 1024, 2048, where improvement would have been seen, it resulted in slowdown. (See code in appendix A). It was necessary to use our algorithm from before changing order of operations (see #7) as the new algorithm was less efficient for multiplying matrices without summing to the result matrix.

7. Changing Order of operations: Obtained speedup by iterating through C and B multiple times while keeping A’s values stored every iteration of the middle loop. This rendered transposition of B unnecessary.

8. __restrict__ keyword: Added this to pointer declarations to allow compiler optimizations that rely on multiple pointers referencing overlapping data, but it had no observable effect.

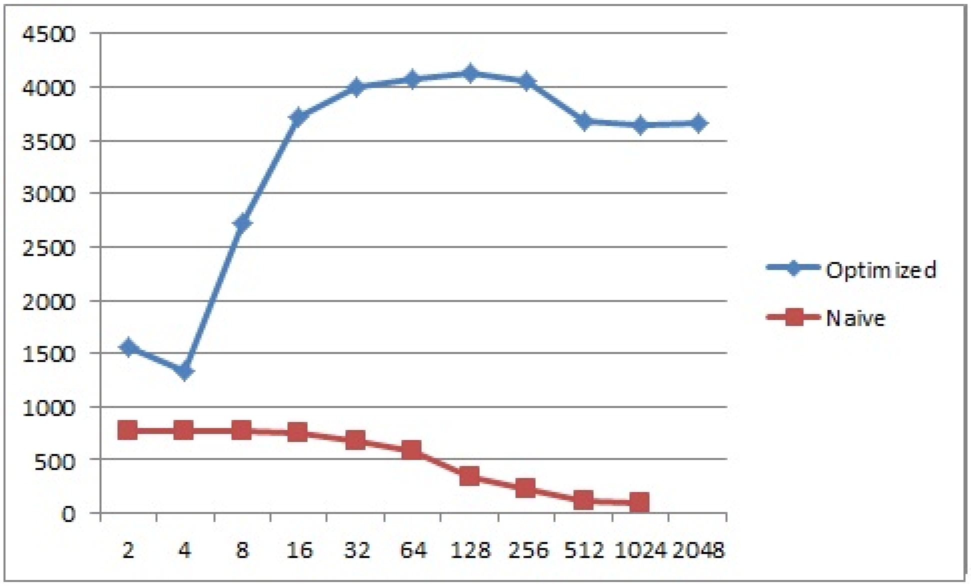
9. Moving operations as far out of loops as possible: This was done in case the compiler or prefetcher couldn’t optimize for strides of i*N--instead the array pointers for eg A[i*n+k] were moved to double* a = A + i*n, and then referenced as a[k]. This gave no observable speedup, so the compiler optimizations were probably working.

Results

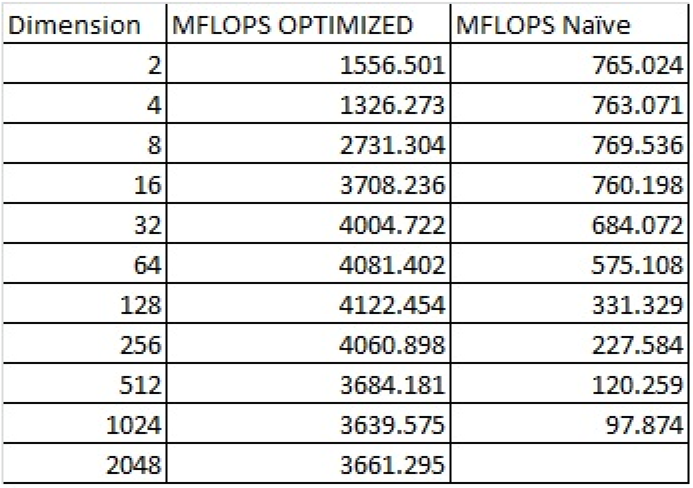
Shown below are the performance numbers we observed on running our optimized program.

```
Dim.   MFLOPS     Runtime       0x80000032   0x8000003b   0x80000000   0x80000007
   2 1556.507      0.0000               49           24            0            0
   4 1326.273      0.0001              505          224            0            0
   8 2731.304      0.0004             2109          871            0            0
  16 3708.236      0.0022            12645         5130            0            0
  32 4004.722      0.0164            80368        38010           27            0
  64 4081.402      0.1285           562659       299508         2962            0
 128 4122.454      1.0174          4183379      2374191        21592            0
 256 4060.898      8.2628         32199215     19120767      1310396          293
 512 3684.181     72.8616        257592982    169475282     17715403       137782
1024 3639.575    590.0370       2060742586   1367032028    141584878      1364390
2048 3661.295   4692.2930      16485936910  10892166149   1133779672     11282069
```

1. Graph



2. Table



Dimension	MFLOPS OPTIMIZED	MFLOPS Naïve
2	1556.501	765.024
4	1326.273	763.071
8	2731.304	769.536
16	3708.236	760.198
32	4004.722	684.072
64	4081.402	575.108
128	4122.454	331.329
256	4060.898	227.584
512	3684.181	120.259
1024	3639.575	97.874
2048	3661.295	

The numbers shown in the above table were observed on average after multiple runs, hence they are representative of the mean.

Appendix A:

```
double *R = malloc(N*N*sizeof(double));
double *M1 = malloc(n*n*sizeof(double));
double *M2 = malloc(n*n*sizeof(double));
double *M3 = malloc(n*n*sizeof(double));
double *M4 = malloc(n*n*sizeof(double));
double *M5 = malloc(n*n*sizeof(double));
double *M6 = malloc(n*n*sizeof(double));
double *M7 = malloc(n*n*sizeof(double));
double *tmp1 = malloc(n*n*sizeof(double));
double *tmp2 = malloc(n*n*sizeof(double));

const double *A12 = A + n, *A21 = A+N*n, *A22 = A + (N+1)*n;
const double *B12 = Bp + n, *B21 = Bp+N*n, *B22 = Bp + (N+1)*n;
double *R12 = R + n, *R21 = R+N*n, *R22 = R + (N+1)*n;

addM(n, A, A22, tmp1, N, N, n);
addM(n, Bp, B22, tmp2, N, N, n);
mulM(n, tmp1, tmp2, M1, n, n, n);
addM(n, A21, A22, tmp1, N, N, n);
mulM(n, tmp1, Bp, M2, n, N, n);
subM(n, B12, B22, tmp1, N, N, n);
mulM(n, A, tmp1, M3, N, n, n);
subM(n, B21, Bp, tmp1, N, N, n);
mulM(n, A22, tmp1, M4, N, n, n);
```

```
    addM(n, A, A12, tmp1, N, N, n);
    mulM(n, tmp1, B22, M5, n, N, n);
    subM(n, A21, A, tmp1, N, N, n);
    addM(n, Bp, B12, tmp2, N, N, n);
    mulM(n, tmp1, tmp2, M6, n, n, n);
    subM(n, A12, A22, tmp1, N, N, n);
    addM(n, B21, B22, tmp2, N, N, n);
    mulM(n, tmp1, tmp2, M7, n, n, n);

    addM(n, M1, M4, tmp1, n, n, n);
    subM(n, tmp1, M5, tmp2, n, n, n);
    addM(n, tmp2, M7, R, n, n, N);
    addM(n, M3, M5, R12, n, n, N);
    addM(n, M2, M4, R21, n, n, N);
    subM(n, M1, M2, tmp1, n, n, n);
    addM(n, M3, M6, tmp2, n, n, n);
    addM(n, tmp1, tmp2, R22, n, n, N);
    addMTo(N, R, C, N, N);
}
}

void addM(int N, const double*__restrict__ A, const
double*__restrict__ B, double *__restrict__ C, int incA, int incB,
int incC) {
    int i, j;
    int n = N/2;
    int inA = incA/2;
    int inB = incB/2;
    int inC = incC/2;
    __m128d* a = (__m128d*) A;
    __m128d* b = (__m128d*) B;
    __m128d* c = (__m128d*) C;
    for (i = 0; i < N; i++)
        for (j = 0; j < n; j++)
            c[i*inC + j] = _mm_add_pd(a[i*inA + j], b[i*inB + j]);
}

void addMTo(int N, const double*__restrict__ A, double *__restrict__
C, int incA, int incC) {
    int i, j;
    int n = N/2;
    int inA = incA/2;
    int inC = incC/2;
    __m128d* a = (__m128d*) A;
    __m128d* c = (__m128d*) C;
    for (i = 0; i < N; i++)
        for (j = 0; j < n; j++)
            c[i*inC + j] += a[i*inA + j];
}

void subM(int N, const double*__restrict__ A, const
double*__restrict__ B, double *__restrict__ C, int incA, int incB,
int incC) {
    int i, j;
    int n = N/2;
    int inA = incA/2;
    int inB = incB/2;
    int inC = incC/2;
    __m128d* a = (__m128d*) A;
    __m128d* b = (__m128d*) B;
    __m128d* c = (__m128d*) C;
    for (i = 0; i < N; i++)
        for (j = 0; j < n; j++)
            c[i*inC + j] = _mm_sub_pd(a[i*inA + j], b[i*inB + j]);
}

void mulM (int N, const double*__restrict__ A, const
double*__restrict__ Bp, double*__restrict__ C, int incA, int incB,
int incC) {
  int i, j, k, ii, jj, kk, iii, jjj, kkk;
  int n = N/2;
    int inA = incA/2;
    int inC = incC/2;
  __m128d* mA = (__m128d*) A;
  __m128d* mC = (__m128d*) C;

  union pd_d {
    __m128d pd;
    double d[2];
  } r1;
  double *B = malloc(N*N*sizeof(double));
  __m128d* mB = (__m128d*) B;
  // transpose array to line up memory access
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      B[i*N+j] = Bp[j*incB+i];
```

```
 int inB = N/2;

//  for (jj = 0; jj < N; jj += BSJ)
 for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j) {
//    for (j = jj; j < min(jj+BS,N); ++j) {
   // sse2 register to accumulate 2 additions at a time,
   // for every 2*kth and 2*k+1th array element (= kth element
   // of __m128d* array)
   r1.pd = _mm_set_pd(0.0, 0.0);
   for (k = 0; k < n; ++k) {
      // multiply elements from mA and mB and add to cumulative
total
                  r1.pd   =   _mm_add_pd(   r1.pd,
_mm_mul_pd(mA[i*inA+k],mB[j*inB+k]));
   }
   C[i*incC + j] = r1.d[0] + r1.d[1];
  }

}
```

```
Output:
Dim.   MFLOPS     Runtime      0x80000032   0x8000003b   0x80000000   0x80000007
   2 1612.141      0.0000              50           23            0            0
   4 1155.054      0.0001             507          259            0            0
   8 2102.033      0.0005            2111         1136            0            0
  16 3701.522      0.0022           12647         5140            0            0
  32 4000.786      0.0164           80368        38053           22            0
  64 3701.244      0.1417          562651       328972         2975            0
 128 4097.581      1.0236         4183367      2379841        21540            0
 256 4077.076      8.2300        32199187     19180943      1307174          131
 512 3698.681     72.5760       257592731    169259916     17675235       131711
1024 2925.544    734.0460      1669939669   1714349160    121877518      6042034
2048  987.968  17389.0860     13020613551  40373901775    960540039    818772448

qsub: job 2022296.cyclades-master.stanford.edu completed
```