Team 05 Assignment 1 Write-up

Nikit Nainwal: matrix multiply algorithms, Eklundh transpose, copy alignment, writeup
Nuochen Lyu: vectorization, register blocking, unrolling
Xiaoyun Zhao: block matrix multiply, writeup

Approaches

1. **Blocked Matrix Multiply with Tuned BLOCK_SIZE**

   The computational intensity (q) for tiled matrix multiple is:

   $q = f/m = 2n^3/(2N+2) * n^2 \approx n/N = b$

   The model of the CPU is Intel(R) Core(™) i7-5930K CPU @ 3.50GHz, with CACHE_SIZE= 15360KB. We tried to tune the BLOCK_SIZE to speed up the cache performance, but we did not get any results significantly different from the default performance with BLOCK_SIZE=41. We also tried using two-layer blocking, but this did not change results significantly either.

2. **Three-loop Transpose**

   We transposed A, so that the naive three-loop multiply would proceed along the cache lines for both A and B. This was the fastest method even without an optimized transpose algorithm. We tried running the blocking algorithm on the transpose matrix, but it was significantly slower. Ultimately we stuck with this method.

```
/* A must already be transposed */
void square_dgemm_transpose (const int lda, double* A, double const*
const B, double* restrict C)
{
  int bpos;
  int apos;
  int cpos;
  double cij;
  /* For each block-column of B */
  for (int j = 0; j < lda; ++j) {
    /* For each block-row of A */
    bpos = j*lda;
    for (int i = 0; i < lda; ++i) {
      cpos = i + j*lda;
      cij = C[cpos];
      apos = i*lda;
```

```
      for (int k = 0; k < lda; ++k) {
        cij += A[k + apos] * B[k + bpos];
      }
      C[cpos] = cij;
    }
  }
}
```

3. **Recursive Matrix Multiplication**

   This is a recursive algorithm which splits the matrix with a larger dimension. It performed about as well as the blocking algorithm (no surprise since they ultimately do the same block multiply). We tried running this algorithm on the transpose matrix, and we also tried writing a stack-based implementation to avoid function call overhead. The recursive transpose variant performed best, but not as well as the three-loop method. The stack-based implementation might perform better for very large matrices, where the function call overhead is more significant.

```
void square_dgemm_recursive (int lda, int n, int m, int p, double* A,
double* B, double* C)
{
  int w = max(n, m, p);
  if (w <= BLOCK_SIZE_R) {
    do_block(lda, n, p, m, A, B, C);
  } else {
    if (w == n) {
      // Split A horizontally
      int n1 = n/2;
      int n2 = n - n/2;
      square_dgemm_recursive(lda, n1, m, p, A, B, C);
      square_dgemm_recursive(lda, n2, m, p, A+n1, B, C+n1);
    } else if (w == p) {
      //Split B vertically
      int p1 = p/2;
      int p2 = p - p/2;
      square_dgemm_recursive(lda, n, m, p1, A, B, C);
      square_dgemm_recursive(lda, n, m, p2, A, B+p1*lda, C+p1*lda);
    } else {
      //Split both
      int m1 = m/2;
      int m2 = m - m/2;
```

```
      square_dgemm_recursive(lda, n, m1, p, A, B, C);
      square_dgemm_recursive(lda, n, m2, p, A+m1*lda, B+m1, C);
    }
  }
}
void square_dgemm_recursive_root(int lda, double* A, double* B,
double* C) {
  square_dgemm_recursive(lda, lda, lda, lda, A, B, C);
}
```

Optimizations

1. **Prefetching**

   We tried using __builtin_prefetch to get elements earlier, but this slowed down the program catastrophically no matter how we tweaked it (slowdowns of up to 80% on Cori; smaller slowdowns on a local computer). This is probably because the compiler is already running prefetch instructions more efficiently.

2. **Vectorization using AVX Intrinsics**

   Use `_mm256_loadu_pd()` to load 4 packed double-precision floating-point elements from memory and `_mm256_storeu_pd()` to store into memory. The theoretical performance is latency(1) and throughput(0.5) on Haswell. Manual vectorization significantly slowed down the program when run on Cori, but increased speed significantly when running on a laptop. This is due to Cori's compiler automatically vectorizing loops. Closer inspection of compiler logs allowed us to rely on the compiler for vectorization. (We also allowed the compiler to unroll loops.)

3. **Register Blocking + Vectorization**

   We tried writing a 4x4 vectorized inner loop for the blocking function. Ideally, this would provide register blocking as well as some vectorization-based speedup. This provided a minor speedup to about average 12% for the blocking variants but the three-loop transpose still performed much better overall.

   The unrolled version of register blocking boost the speed to 17% on average and 21% in peak after we unrolled the four loop iteration. So the expanse of loading and storing is greatly reduced.

4. **Alignment**

We implemented partial copy alignment (only for large arrays with even dimensions); it seems to provide a minor speedup for those cases, likely because it realigns most blocks on 16-byte boundaries during Eklundh transposition. In conjunction with this, we tried to use the __assume_aligned feature from the Intel compiler. This provided a minor speedup for very large arrays when used in the copy alignment function; everywhere else we used it, it caused catastrophic slowdowns. (This is doubly suspicious since __assume_aligned only gives the compiler information and does not execute anything.)

5. **Eklundh Transposition**

Eklundh transposition is a recursive in-place matrix transposition algorithm for square matrices of dimension 2^i; we had to modify it slightly to work for all square matrices. It is highly efficient since its recursive case copies vectors along cache lines. Using this algorithm over the naive transposition algorithm almost doubled the speed of three-loop transpose.

```
/* Recursively transpose a block (width w) of a matrix A. */
void self_transpose_eklundh_(const int lda, const int w, double* A,
double* temp) {
  int p1;
  int p2;
  if (w < BLOCK_SIZE_TR) {
    for (int i = 0; i < w; ++i) {
      p1 = i * lda;
      for (int j = 0; j < i; ++j) {
        temp[j] = A[p1 + j];
      }
      for (int j = 0; j < i; ++j) {
        A[p1 + j] = A[i + j*lda];
      }
      for (int j = 0; j < i; ++j) {
        A[i + j*lda] = temp[j];
      }
    }
  } else {
    int d1 = w - w/2;
    int d2 = w/2;
    //Swap matrices: A[:d2, d1:] A[d1:, :d2]
    for (int j = 0; j < d2; ++j) {
      p1 = (d1+j)*lda;
```

```
      p2 = d1 + j*lda;
      for (int i = 0; i < d2; ++i) {
        temp[i] = A[p1+i];
      }
      for (int i = 0; i < d2; ++i) {
        A[p1+i] = A[p2+i];
      }
      for (int i = 0; i < d2; ++i) {
        A[p2+i] = temp[i];
      }
    }
    //Recurse on all 4 submatrices
    self_transpose_eklundh_(lda, d1, A, temp);
    self_transpose_eklundh_(lda, d2, A+d1*lda, temp);
    self_transpose_eklundh_(lda, d2, A+d1, temp);
    self_transpose_eklundh_(lda, d1, A+d2+d2*lda, temp);
  }
}
/* Transpose a matrix in-place using Eklundh. */
void self_transpose_eklundh(const int lda, double* A) {
  double* temp = _mm_malloc(sizeof(double) * (lda + 2), 64);
  self_transpose_eklundh_(lda, lda, A, temp);
  _mm_free(temp);
}
```

Performance

1. **Patterns**

   We observed no performance dips, but there were two clear patterns of performance. First, the smaller arrays do not perform that well, probably because transposition (an extra cost not counted in the performance metric) dominates their runtime. Inversely, our solution performs better for larger arrays, which is the more important factor. (Scaling up, our solution performs at 37-38% for a 768x768 array, and 43-44% for a 2000x2000 array.) Second, the even-length arrays perform significantly better than the odd-length arrays. This is probably because Eklundh transposition on an even-length array involves copying vectors which mostly start at even (aligned to 16 bytes) positions, and the Intel compiler seems to optimize this with a special memcpy call. On a laptop, smaller arrays performed better and even-length arrays performed the same as odd-length arrays

(probably because multiplication time dominated transposition time and the compiler did not optimize vector copying).

2. **On Another Computer**

   In addition to the previous remarks, running on a laptop, the blocked and recursive implementations ran fairly faster than the three-loop transpose implementation (by 20-50%). This contradicted the Cori results, where the three-loop transpose implementation ran faster, even without Eklundh transposition. This is likely due to the mechanisms of cache retention; Cori seems to perform extremely well when moving along cache lines. When we optimized the transposition algorithm (using Eklundh over a naive n^2 transpose), there was no speedup on a laptop, but Cori ran the program about twice as fast. This is presumably because multiplication time dominated transposition time on the laptop.