CS 267 Assignment 1 Report

Optimize Matrix Multiplication

12 February 2016

Wei Ni
Cathy Wu
Byung Gon Song

Introduction

The goal of this assignment is understanding computer architecture and improving performance for matrix multiplication by applying several low level parallel programming techniques and taking advantage of the cache structure in the computer system. The targeting hardware, the supercomputer Edison (NERSC), is made by Intel and is capable of four double floating point computations per cycle (through its AVX intrinsics) and has two-state pipelines. **As a result, we are able to achieve an average of 76% of peak performance.** Please see Appendix A for our final benchmark numbers.

This report is split into 2 sections: 1) optimization attempts, and 2) additional optimization techniques. The former describes the story and progression of our implementation, with attempts and some ideas that worked better than others. These ideas led to the backbone of our implementation. Then, the latter section describes additional optimization techniques that more or less "just worked" and that we additionally layered on to the main backbone, to achieve further performance boosts.

Optimization attempts

We tried various optimization techniques before arriving at our final implementation. We begin with the blocked matrix multiplication code and tried the following:

1. Incorporating SSE SIMD intrinsics for fast 2x2 x 2x1 multiplication
2. Transposing the matrix A
3. Two-level blocking
4. 4x4 kernel: AVX SIMD intrinsics for fast 4x4 x 4x4 multiplication.
    a. Zero padding for non-4x4 matrices to make use of the 4x4 kernel

Our final implementation includes 4, 4a, 5, 6, 7 (see the next section for 5-7) and the concept from 3 (with new block sizes selected). Now we describe the attempts in detail.

0. Block matrix multiplication

We wish to perform matrix matrix multiplication on two block matrices A, B. The precise operation is

$$C \mathrel{+}= A \times B.$$

On Edison, the given blocked matrix multiplication performed on average 6.26% of peak performance.

1. SSE/SSE2 Intel Intrinsic for Single Instruction on Multiple Data

```
static void do_block (int lda, int M, int N, int K, double* A, double* B, double* C)
{
  __m128d A1, Bv, cij, cij2; // grouping together
  for (int i = 0; (i+1) < M; i+=2)
  {
    for (int j = 0; j < N; j++)
    {
      cij = _mm_loadu_pd(&C[i+j*lda]);
      for (int k = 0; k < K; k++) // double precision 64 bit and m128 fits with 2 of them
      {
        A1 = _mm_load_pd(&A[i + k*lda]);
        Bv = _mm_load1_pd(&B[k + j*lda]);
        cij =_mm_add_pd(cij, _mm_mul_pd(A1, Bv));
      }
      _mm_storeu_pd(&C[i+j*lda], cij);
    }
  }
}
```

Figure 1: Simple SIMD on a block

Matrix multiplication is an embarrassingly parallel problem in which each data point in the matrix needs exactly the same type of computation, such as multiplication and addition. For this vector computation, we used the Intel Intrinsic for Single Instruction on Multiple Data (SIMD). To warm up, we used the SSE/SSE2 instruction sets, which use 128-bit computation. Two double-precision floating points fit in a `_m128d` type.

This code snippet loads two elements at a time (from two consecutive rows) from the block matrix A, and multiplies it by a single corresponding value from the block matrix B, and stores it into the corresponding consecutive elements of the block matrix C. The three-level loop proceeds much like the original code, except we can process two rows of A at a time (hence the `i+=2`). Recall that the instruction `_mm_load1_pd` loads a single double-precision float but copies it to both elements.

2. Transpose and re-structure of data with AVX

```
static void do_block_even (int lda, int M, int N, int K, double* A, double* B, double* C)
{
  union {
      __m128d temp;
      double u[2];
  } U1;
  union {
      __m128d temp2;
      double u[2];
  } U2;
  __m128d A1,  A2, Bv, temp, temp2; // grouping together
  /* For each row i of A */
  double temp3[2];
  for (int j = 0; j < N; j++)
  {
    int i = 0;
    for (; (i+1) < M; i += 2)
    {
      int k;
      temp = _mm_setzero_pd();
      temp2 = _mm_setzero_pd();
      for (k = 0; k < K; k += 2)
      {
        A1 = _mm_load_pd(&A[k + i*lda]);
        A2 = _mm_load_pd(&A[k + (i+1)*lda]);
        Bv = _mm_load_pd(&B[k + j*lda]);
        A1 = _mm_mul_pd(A1, Bv);
        A2 = _mm_mul_pd(A2, Bv);
        temp = _mm_add_pd(temp, A1);
        temp2 = _mm_add_pd(temp2, A2);
      }
      //Copy the lower double-precision (64-bit) floating-point element of a to dst.
      U1.temp = temp;
      U2.temp2 = temp2;
      temp3[0] = U1.u[0] + U1.u[1];
      temp3[1] = U2.u[0] + U2.u[1];

      // Reductino on temp and temp
      temp = _mm_loadu_pd(temp3);
      // Reduce time for loading and storing time in C
      temp = _mm_add_pd(_mm_loadu_pd(&C[i+j*lda]), temp);
      _mm_storeu_pd(&C[i+j*lda], temp);
    }
  }
}
```

Figure 2: Applying transpose

In a separate attempt (not part of our final implementation), we created a transposed matrix of matrix access matrix A and matrix B in same direction when performing our dot products. We also tried to minimize access to matrix C, so we used the `_mm_setzero_pd` function to get a size-two zero vector as temporary storage for the dot product, which we add to matrix C when completed.

However, this does not take advantage of the cache. We observed at this point that if we worked with matrices that all fit inside the L1 cache, then we would not need to consider our memory access pattern (row-wise or column-wise) of the matrices. In our final design, we operate on such a size of matrices such that all three matrices can fit into the L1 cache (See #3).

3. Two level blocking

Blocking a big matrix into smaller matrices that can fit into cache can help reduce cache misses and improve spatial locality as well as temporal locality. Initially, we did only one level blocking (targeting the L1 cache). By trying different block size, we found the optimal value to be 32x32. This optimal block size theoretically can fit in the L1 cache (32kb for data) on the ivy bridge processor of Edison. This improves the performance by roughly 10% (from #3). The performance on L2 cache can also be a bottleneck, so two level blocking is essentially necessary. After several trials, we set the size of upper level block as 160, which can fit into the L2 cache (256kb on the ivy bridge processor).

As we will describe in #4a, a further optimization triggered a new search over block sizes. The final block sizes of our implementation are 64x64 and 180x180 for the L1 and L2 levels, respectively.

4. 4x4 kernel: optimize 4x4 matrix multiplication via Intel AVX

We make use of the Intel AVX intrinsics to do four multiplications and additions per cycle. In order to access the matrix faster (using `_mm256_load_pd` and `_mm256_store_pd` instead of `_mm256_loadu_pd` and `_mm256_storeu_pd`), we copy and align the data with a 32-byte memory boundary (described in more detail in #5).

```
inline void do_fast_4x4_block (int lda, double *A, double *B, double *C){

  register __m256d colc1 = _mm256_load_pd(C);
  register __m256d colc2 = _mm256_load_pd(C+lda);
  register __m256d colc3 = _mm256_load_pd(C+2*lda);
  register __m256d colc4 = _mm256_load_pd(C+3*lda);
  register __m256d cola;
  __m256d brod1, brod2, brod3, brod4;

  for(int i = 0; i < 4; ++i){
    cola = _mm256_load_pd(A+i*lda);
    brod1 = _mm256_set1_pd(B[i]);
    brod2 = _mm256_set1_pd(B[i+lda]);
    brod3 = _mm256_set1_pd(B[i+2*lda]);
    brod4 = _mm256_set1_pd(B[i+3*lda]);
    colc1 = _mm256_add_pd(colc1, _mm256_mul_pd(brod1, cola));
    colc2 = _mm256_add_pd(colc2, _mm256_mul_pd(brod2, cola));
    colc3 = _mm256_add_pd(colc3, _mm256_mul_pd(brod3, cola));
    colc4 = _mm256_add_pd(colc4, _mm256_mul_pd(brod4, cola));
  }

  _mm256_store_pd(C, colc1);
  _mm256_store_pd(C+lda, colc2);
  _mm256_store_pd(C+2*lda, colc3);
  _mm256_store_pd(C+3*lda, colc4);
}
```

Figure 3: Optimized 4X4 Matrix Multiplication

4a. Optimizing matrix multiplication for non-32x32 matrices

The above optimizations mostly benefit blocks of exactly size 32x32, which, incorporating the optimizations 5-7, gave us a 32% peak performance on Edison. For blocks of different sizes from 32x32 (the "leftover" columns and rows of the original matrices A, B, C), we were performing naive matrix multiplication (via an unoptimized microkernel). After running several experiments, we note that our poorest performing matrices are of size 97, 129, 229, 191, 257. We note that the sizes 97, 129, and 257 have exactly 1 "leftover" column/row after the 32x32 blocking. The 229 size is a few dimensions beyond the nice 224 size, and the 127 and 191 sizes are just 1 dimension shy of the nice 192. (Here, we use *nice* to mean that the dimension size is a multiple of 32.)

We optimized our implementation for non-32x32 blocks by making use of our fast 4x4 matrix multiplication kernel. For small matrices with dimensions not divisible by 4, we pad them to their next multiple of 4, to automatically make use of the 4x4 kernel. Initially, a naive computation of the new matrix dimensions (using integer division) gave us a small performance boost (~5%). However, when we computed the new matrix dimensions (using `mod` instead of integer division), this gave us a further performance boost (~6%). Finally, further optimizing over block sizes (See #3; now 64x64 and 180x180 for the two levels respectively), we achieved a performance boost of 32%, **thereby achieving 76% peak performance on Edison**.

At this point, we are not sure why the optimized use of the 4x4 kernel triggered such a performance boost with **different** intermediate block sizes. (We had experimentally verified that 32x32 blocks were much better than 64x64 before the `mod` optimization.) Reasonably, the `mod` operation reduces the number of cycles required for every operation on a block that is not perfectly the selected block size. It is possible that this enabled the compiler to further (and massively!) pipeline instructions, and subsequently triggering changes in memory access patterns.

Additional optimization techniques

In this section, we discuss additional optimization that we were able to layer on to the above backbone to further optimize our implementation of matrix matrix multiplication.

In short the optimization techniques we used are:

5. Copy optimization for L1 tiling.
6. Change the looping order, using j-k-i.
7. Intel compiler opt flags

We now discuss them further.

5. Copy optimization

Reading from memory into cache is much faster if the data is in continuous memory. If we do the computation directly on the given matrix, memory addressing would be a considerable cost (much more expensive than the computation cost itself). Thus, in our *blocked* matrix multiplication code, each time we would like to operate on three 32x32 matrices, we first copy the three corresponding blocks into three pieces of pre-allocated continuous memory.

```
static double copy_a[BLOCK_SIZE*BLOCK_SIZE] __attribute__((aligned(32)));
static double copy_b[BLOCK_SIZE*BLOCK_SIZE] __attribute__((aligned(32)));
static double copy_c[BLOCK_SIZE*BLOCK_SIZE] __attribute__((aligned(32)));

inline void copy_to_arr(int lda, int m, int n, double *org, double *des){
  for(int j = 0; j < n; ++j){
    memcpy(des+m*j, org+lda*j, m*8);
  }
}

inline void copy_to_blk(int lda, int m, int n, double *org, double *des){
  for(int j = 0; j < n; ++j)
    memcpy(des+lda*j, org+m*j, m*8);
}
```

Figure 5: Copying matrix

6. Loop ordering: Change to the “j-k-i” order

```
void square_dgemm (int lda, double* A, double* B, double* C)
{
  for (int j = 0; j < lda; j += BIG_BLOCK_SIZE)
    for (int k = 0; k < lda; k += BIG_BLOCK_SIZE)
      for (int i = 0; i < lda; i += BIG_BLOCK_SIZE)
        {
          int M = min (BIG_BLOCK_SIZE, lda-i);
          int N = min (BIG_BLOCK_SIZE, lda-j);
          int K = min (BIG_BLOCK_SIZE, lda-k);
          do_big_block(lda, M, N, K, A + i + k*lda, B + k + j*lda, C + i + j*lda);
        }
}
```

Figure 4: Changing loop order

As the assignment hints suggest, we tried to switch the loop order of the indices “i-j-k”. We tried all the possible combinations and found the order “j-k-i” to give the best performance. Following the “j-k-i” order, some data values are pre-fetched (such as from the C matrix), which improves spatial locality. Note that the least re-usable memory accesses are for the matrix A (since we access it row-wise, but it is stored column-wise), which is correspondingly traversed in the innermost loop.

7. Compiler flag optimization

In this assignment, we use Intel c compiler. Our compiling flags include:

```
-O3 -xAVX -funroll-loops.
```

The `-O3` flag forces the compiler to do maximal optimization. The `-xAVX` enables the SIMD vectorization via AVX intrinsics. The `-funroll-loops` tells the compiler to automatically unroll some `for` loops. The Intel compiler (`icc`) gives better performance than `gcc`. From the original `-O1` to current setting, the average performance improves by nearly 10%.

Conclusion

Our implementation achieves 76% peak performance on Edison. We implemented a two-level blocking scheme designed for our submatrices to fit into the L1 and L2 caches, and we make use of the AVX SIMD instructions. We also introduced copy optimization, compiler flag optimization, and loop ordering optimization. We noticed for certain size of matrix, the peak speed percentage is over 100%, which might come from Intel turbo boost technology. We have further considered the alternative strategy of recursive matrix multiply, but concluded that the submatrices would not be as well aligned with the cache structure as the blocked implementation, and that the overhead of memory rearrangement would be too great.

Contributions

This project was truly a team effort, and the team met several times (and early!) despite paper deadlines and other commitments. Everyone participated in discussions. We collaborated over Github, helped one another over email, and wrote the report together using Google Docs. To give a few highlights: Wei implemented many of the hints suggested for the assignment, performed many experiments, and provided a backbone for the report. Cathy provided the final "`mod`" optimization that triggered a giant performance boost, and she contributed heavily to the report. Byung pioneered the idea of transposing the A matrix; Cathy pioneered the attempt at recursive matrix multiply.

Appendix A: Our final benchmark numbers

```
Size: 31        Mflop/s:  10598.6       Percentage: 55.20
Size: 32        Mflop/s:  11639.1       Percentage: 60.62
Size: 96        Mflop/s:  17911.1       Percentage: 93.29
Size: 97        Mflop/s:    14183       Percentage: 73.87
Size: 127       Mflop/s:  24921.7       Percentage:129.80
Size: 128       Mflop/s:  7416.05       Percentage: 38.63
Size: 129       Mflop/s:  6862.71       Percentage: 35.74
Size: 191       Mflop/s:  12808.6       Percentage: 66.71
Size: 192       Mflop/s:  13515.4       Percentage: 70.39
Size: 229       Mflop/s:  19259.3       Percentage:100.31
Size: 255       Mflop/s:  13769.8       Percentage: 71.72
Size: 256       Mflop/s:  13376.1       Percentage: 69.67
Size: 257       Mflop/s:  12727.1       Percentage: 66.29
Size: 319       Mflop/s:  11814.5       Percentage: 61.53
Size: 320       Mflop/s:  12399.5       Percentage: 64.58
Size: 321       Mflop/s:  12262.1       Percentage: 63.86
Size: 417       Mflop/s:    17924       Percentage: 93.35
Size: 479       Mflop/s:  15868.2       Percentage: 82.65
Size: 480       Mflop/s:  16476.6       Percentage: 85.82
Size: 511       Mflop/s:  12912.6       Percentage: 67.25
Size: 512       Mflop/s:  14391.1       Percentage: 74.95
Size: 639       Mflop/s:  16564.1       Percentage: 86.27
Size: 640       Mflop/s:    18512       Percentage: 96.42
Size: 767       Mflop/s:  20028.4       Percentage:104.31
Size: 768       Mflop/s:  16178.5       Percentage: 84.26
Size: 769       Mflop/s:  15959.7       Percentage: 83.12
Average percentage of Peak =  76.1779
```

Oddly, there are a few values above 100%, but there is no conclusion as to why this is the case (See Piazza Post 27). Also interestingly, the 129 matrix still performs poorly.