CS 267 - HW 1
Team 3: Kaila Cappello, Omar Clinton, Alexander Thomas

Introduction

In this report, we will detail the steps taken to obtain our optimal solution (in terms of performance) for blocked matrix multiplication of the form: `C = C + A * B`. We investigate cache and register blocking, transposing, vectorization, arithmetic intensity and loop unrolling. The optimal code can be found in the Appendix.

Optimization and Results

Blocking (Kaila)

The first optimization technique that was implemented was blocking. We first adjusted the block size to see what gave us optimal performance, and found that a block size of 120 causes the highest percentage of peak. With a block size of 120, the tiles are the optimal size to fit into the cache for quicker access. The graph below shows our optimized code with varying block sizes from 10 to 400 (for the optimal code). With block sizes of 150 and larger, the performance levels off. We also tried larger block sizes up to 1000, but because the matrix is 769x769, block sizes above ~385 (769/2) would have no further improvement in performance. We reasoned that a 120 x 120 block was about the size of our L1 cache. Increasing our box size may had lead to worse performance because then it will take more cycles to fetch from our L2 cache, L3 cache, or physical memory since with a box size greater than our L1 cache will guarantee that we must spin to the next level of our memory hierarchy.

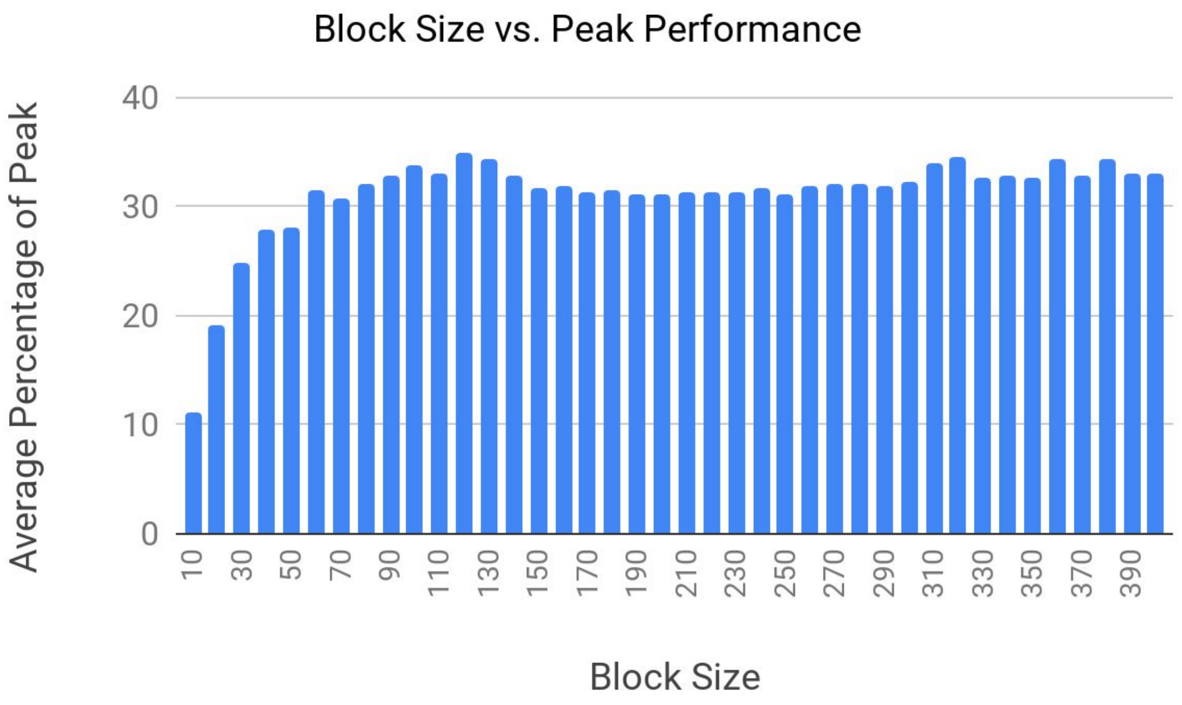


Figure 1 - Block Size vs Performance

Register Blocking (Kaila and Omar)

We took advantage of the speed of registers versus memory access by reusing values loaded from memory into registers in our matrix kernel. To calculate the entry `C[i, j]` of our resultant matrix, we need the product of `A[i, k]` and `B[k, j]`. To calculate `C[i, j+1]`, we need `A[i, k]` and `B[k, j+1]`. Therefore, we can reuse the value of `A[i, k]`. That is, we can note from this pattern that unrolling the ith and jth loop exposes register use. So this stemmed the idea of reusing values loaded from memory, rather than reloading it into memory in the next iteration. We implemented register blocking by iterating through the ith (row of `A`) and jth loop (column of `B`) in steps of 2, 3 and 4. We found the best performance by using a factor of 3 which is the optimal size based on the size of the register memory. With this factor, 9 entries of `C` are calculated in a single loop. With a factor of 4, the performance had a large dip, signifying that the registers can only fit a block size of up to 3x3. We implemented this inside our matrix multiplication loops to further block our cache tiles. The performance dip, we reasoned, was due to the limited number of registers in our machine. Too many variables to reference may require the ISA to spill registers into memory.

The following is an example of reusing register values in the innermost loop. Notice that the ith `t` variable is being used several times to calculate 9 entries of `C`. This means less reading from memory.

```
for (k = 0; k < K; k += 1) {
    //1st Row
    t0 = A[k + i * lda];
    //1st Col
    t1 = B[k + j * lda];
    //2nd Row
    t2 = A[k + (i + 1) * lda];
    //2nd Col
    t3 = B[k + (j + 1) * lda];
    //3rd Row
    t4 = A[k + (i + 2) * lda];
    //3rd Col
    t5 = B[k + (j + 2) * lda];


    cij += t0 * t1;
    ci1j += t2 * t1;
    cij1 += t0 * t3;
    ci1j1 += t2 * t3;

    ci2j += t4 * t1;
    ci2j1 += t4 * t3;
    ci2j2 += t5 * t4;
    ci1j2 += t2 * t5;
    cij2 += t0 * t5;

}
```

Transposing Matrix A (Omar)

Spatial locality was improved by transposing matrix A, which consequently improved cache performance. Consider matrix multiplication when A is not transposed:

```
for (int i = 0; i < M; ++i) {
    //For each column j of B
    for (int j = 0; j < N; ++j) {
        // Compute C(i,j)
        double cij = C[i + j * lda];
        for (int k = 0; k < K; ++k) {
            cij += A[i + k * lda] * B[k + j * lda];
        }
        C[i + j * lda] = cij;
    }
}
```

In the inner loop (`k`), `A` is indexed with strides of `lda` and `B` with strides of 1. By transposing `A`, the following code is obtained:

```
for (int i = 0; i < M; ++i) {
    //For each column j of B
    for (int j = 0; j < N; ++j) {
        // Compute C(i,j)
        double cij = C[i + j * lda];
        for (int k = 0; k < K; ++k) {
            cij += A[k + i * lda] * B[k + j * lda];
        }
        C[i + j * lda] = cij;
    }
}
```

Since `A` is now indexed using strides of 1 in the inner loop, spatial locality is improved.

Vectorization (Omar)

Our optimal code was benchmarked using the `icc` compiler. Auto-vectorization is enabled by `icc` when the source code is compiled with a flag of `O2` or higher (see [1]). There is a significant performance difference between the vectorized code and non-vectorized code. The vectorized code is approximately two times faster. The results are presented below for the optimal code:

Table 1 - Performance of vectorized and non-vectorized code.

Block Size	Performance (%)	
	Non-Vectorized	Vectorized
120	18.0	34.8

The `-qopt-report=1` and `-qopt-report-phase=vec` flags were added to determine which parts of the code was vectorized.

The inner loop (`j`) of the transposing operation was successfully vectorized:

```
//Transpose A -> row-major format
for (int i = 0; i < lda; i++) {
    for (int j = 0; j < lda; j++) {
        A_t[j + lda * i] = A[i + lda * j];
    }
}
```

Furthermore, the inner loop (`k`) of the function performing the product of the block matrices (see `do_block_unroll_transform` in appendix) was vectorized:

```
for (k = 0; k < K; k += 1) {
    //1st Row
    t0 = A[k + i * lda];
    //1st Col
    t1 = B[k + j * lda];
    //2nd Row
    t2 = A[k + (i + 1) * lda];
    //2nd Col
    t3 = B[k + (j + 1) * lda];
    //3rd Row
    t4 = A[k + (i + 2) * lda];
    //3rd Col
    t5 = B[k + (j + 2) * lda];


    cij += t0 * t1;
    ci1j += t2 * t1;
    cij1 += t0 * t3;
    ci1j1 += t2 * t3;

    ci2j += t4 * t1;
    ci2j1 += t4 * t3;
    ci2j2 += t5 * t4;
    ci1j2 += t2 * t5;
    cij2 += t0 * t5;

}
```

Arithmetic Intensity (Alex)

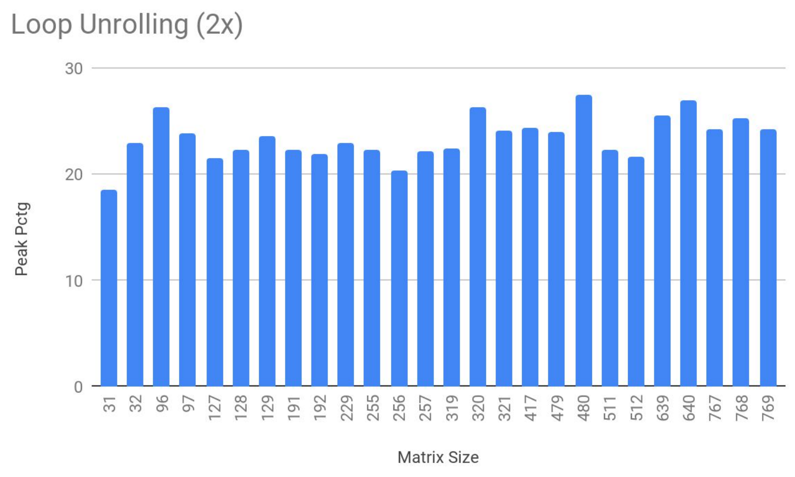
Another optimization we noticed was calculating the indexes of matrix `A` and `B`. In the naive implementation (where matrix `A` is transposed to be row-major), we indexed matrix `A` using `A[k + i*lda]` and matrix `B` using `B[k + j*lda]`. We noticed that per iteration, we would do the computation `i*lda` and `j*lda`. Multiplication is an intensive ALU operation compared to adding and subtracting.
We stored the result of `i*lda` and `j*lda` at the top of their respective loops. This meant we didn't have to recalculate this multiplication operation each time. The effect was negligible. We reasoned that memory operations took most of the cycles of our CPU.

```
int lda_i, lda_j;
```

```
for (int i = 0; i < M; ++i) {
    lda_i = i *lda;
    //For each column j of B
    for (int j = 0; j < N; ++j) {
        lda_j = j*lda;
        // Compute C(i,j)
        double cij = C[i + lda_j];
        for (int k = 0; k < K; ++k) {
            cij += A[k + i_lda] * B[k + j_lda];
        }
        C[i +j_lda] = cij;
    }
}
```

Loop Unrolling (Alex)

Another optimization that we tried was loop unrolling. With a deep pipeline, a branch misprediction could be costly. We tried to minimize the number of branch instructions in our assembly code. In order to do this, we loop unrolled. Unfortunately, we did not get a favorable speed up from loop unrolling by a factor of 2 or a factor of 4 (where the i^{th} and j^{th} index had a stride of 2 or 4). We received 23.43 peak performance percentage for 2 factor loop unrolling and 19.20 peak performance percentage for 4 factor unrolling.



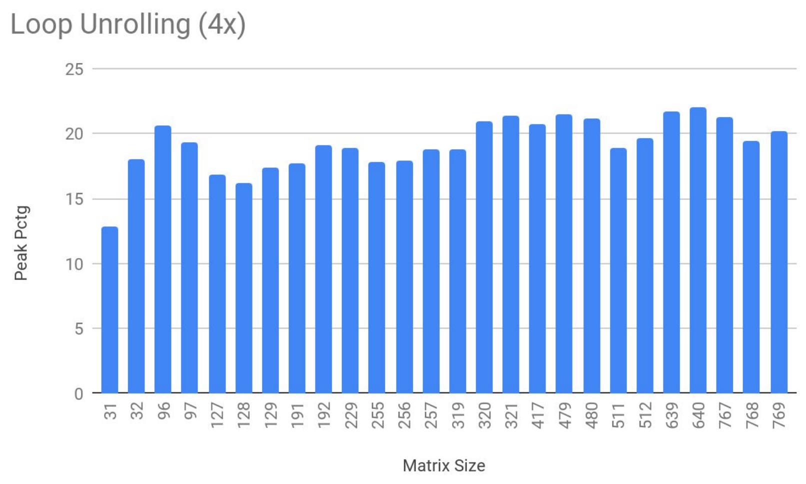


Figure 2 - Performance of loop unrolling with factor of 2 and 4

Copy Optimization (Alex)

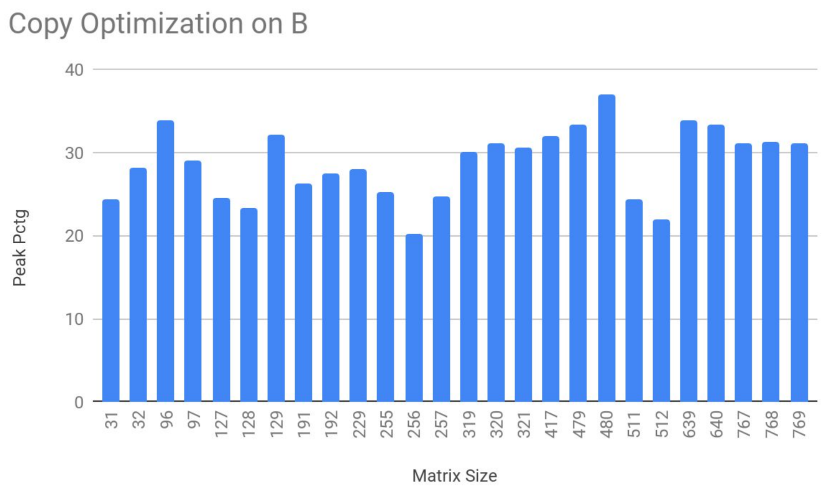


Figure 3 - Performance of copy optimization on B

Another optimization that we attempted to do was copy optimization on matrix B. This was because the input of our matrix may not be cache-aligned. We did this by allocating a buffer and copying the contents from B to the buffer. Doing so did not give us a favorable speed up. We reasoned that transferring the contents from B to our buffer. We got a peak percentage of 28.760.

Results from Running on Local CPU (MBP 2018 Model) (Alex)

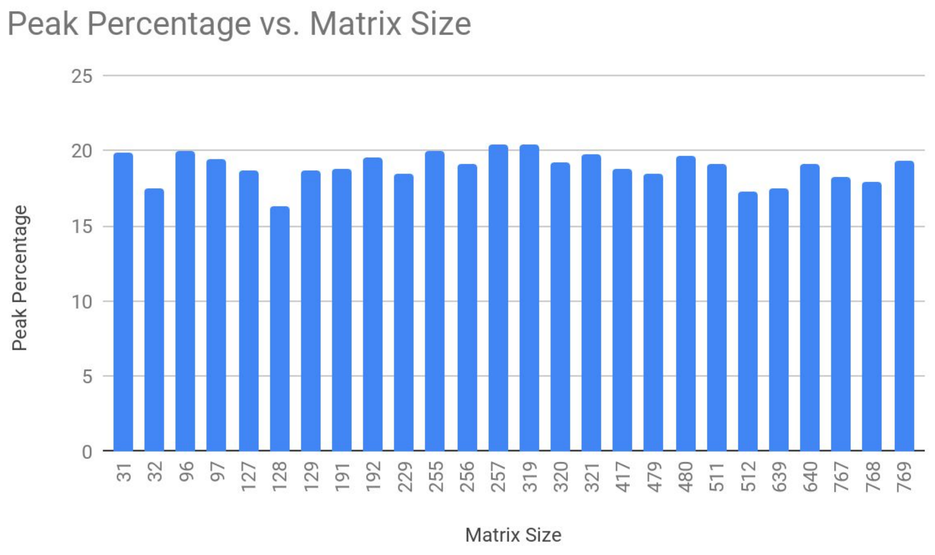


Figure 4 - Performance on local computer

Running on a 2018 “Macbook Pro” model with a 2.7 GHz processor and 4 physical cores, we get 18.90% of the peak performance. It has a 64 KB L1 cache, which is the same as the Cori Phase 1 nodes. We suspected the difference in performance is due to higher throughput (multiple FP units) on the Cori Phase 1 nodes.

Conclusion:

We optimized the performance of our matrix multiplication code by carefully picking the block size and level of loop unrolling that maximizes the usage of the resources provided by Cori's architecture. We also transposed the matrix `A` to improve spatial locality. Furthermore, a significant portion of our performance increase can be attributed to auto-vectorization. Which we found offered a considerable speed up compared to AVX intrinsics.

Contributions:

Alex: Implementing vectorization, arithmetic intensity, and copy-optimization.
Kaila: cache and register blocking
Omar: Transposing, vectorizing, register blocking

References:

[1] “Using Automatic Vectorization.” Intel® Software, Intel, 7 Nov. 2018, software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-using-automatic-vectorization.

Appendix:

```
/*
 * This auxiliary subroutine performs a smaller dgemm operation
 *  C := C + A * B
 * where C is M-by-N, A is M-by-K, and B is K-by-N.
 */
static void do_block_unroll_transpose(int lda, int M, int N, int K,
double *A, double *B, double *C) {

    double cij;
    double ci1j;
    double cij1;
    double ci1j1;

    double ci2j;
    double ci2j1;
    double ci2j2;
    double ci1j2;
    double cij2;



    double t0, t1, t2, t3, t4, t5, t6, t7;
    int i, j, k;
    // For each row i of A
    for (i = 0; i < M / 3 * 3; i += 3) {
        //For each column j of B
        for (j = 0; j < N / 3 * 3; j += 3) {
            // Compute C(i,j)
            cij = C[i + j * lda];
            // Compute C(i+1,j)
            ci1j = C[i + 1 + j * lda];
            // Compute C(i,j+1)
            cij1 = C[i + (j + 1) * lda];
            // Compute C(i+1,j+1)
            ci1j1 = C[(i + 1) + (j + 1) * lda];


            //Compute C(i+2, j)
            ci2j = C[(i + 2) + (j) * lda];
            //Compute C(i+2, j+1)
            ci2j1 = C[(i + 2) + (j + 1) * lda];
            //Compute C(i+2, j+2)
            ci2j2 = C[(i + 2) + (j + 2) * lda];
            //Compute C(i+1, j+2)
            ci1j2 = C[(i + 1) + (j + 2) * lda];
            //Compute C(i, j+2)
            cij2 = C[(i) + (j + 2) * lda];

            for (k = 0; k < K; k += 1) {
                //1st Row
                t0 = A[k + i * lda];
                //1st Col
                t1 = B[k + j * lda];
                //2nd Row
                t2 = A[k + (i + 1) * lda];
                //2nd Col
                t3 = B[k + (j + 1) * lda];
                //3rd Row
                t4 = A[k + (i + 2) * lda];
                //3rd Col
```

```
                t5 = B[k + (j + 2) * lda];


                cij += t0 * t1;
                ci1j += t2 * t1;
                cij1 += t0 * t3;
                ci1j1 += t2 * t3;

                ci2j += t4 * t1;
                ci2j1 += t4 * t3;
                ci2j2 += t5 * t4;
                ci1j2 += t2 * t5;
                cij2 += t0 * t5;

            }



            C[i + j * lda] = cij;
            C[i + 1 + j * lda] = ci1j;
            C[i + (j + 1) * lda] = cij1;
            C[(i + 1) + (j + 1) * lda] = ci1j1;

            C[(i + 2) + (j) * lda] = ci2j;
            C[(i + 2) + (j + 1) * lda] = ci2j1;
            C[(i + 2) + (j + 2) * lda] = ci2j2;
            C[(i + 1) + (j + 2) * lda] = ci1j2;
            C[(i) + (j + 2) * lda] = cij2;

        }
//         The odd col of matrix B, this should only have ONE
iteration!
        for (j = N / 3 * 3; j < N; ++j) {
            cij = C[i + j * lda];
            ci1j = C[i + 1 + j * lda];
            ci2j = C[i + 2 + j * lda];
            for (k = 0; k < K; ++k) {
                cij += A[k + i * lda] * B[k + j * lda];
                ci1j += A[k + (i + 1) * lda] * B[k + j * lda];
                ci2j += A[k + (i + 2) * lda] * B[k + j * lda];
            }
            C[i + j * lda] = cij;
            C[i + 1 + j * lda] = ci1j;
            C[i + 2 + j * lda] = ci2j;
        }

    }

    //The odd row of matrix A, this should only have ONE iteration!
    for (i = M / 3 * 3; i < M; ++i) {
        //For each column j of B
        for (j = 0; j < N; ++j) {
            cij = C[i + j * lda];
            for (k = 0; k < K; ++k) {
                cij += A[k + i * lda] * B[k + j * lda];
            }
            C[i + j * lda] = cij;
        }
    }

}
```