DGEMM Kernel

(GEneral Matrix Multiplication
in Double precision)

How this presentation is organised

- Objective
- Requirements
- Stepwise progress towards maximum CPU utilization
- Conclusion

Objective

The aim of this task was to write the DGEMM micro-kernel to multiply two matrices of size *M x K* and *K x N* to give a resultant matrix of size *M x N* and to optimize it for maximum CPU utilization (hence faster execution).

$$C_{mn} = \sum_{k=0}^{K-1} A_{mk} \cdot B_{kn} \,, \qquad m = 0, \ldots, M-1, \quad j = 0, \ldots, N-1.$$

Requirements

- Use a Single KNL core
- K = 128
- Results must be the same up to machine precision.
- Allowed to use: auto-vectorisation, #pragma directives, intrinsics, inline assembly.

Step 1

- Original implementation disabling all compiler optimization (Using the -O0 switch)
- Result: 0.05 GFLOPS

Step 2

- Original implementation but allowed compiler optimization.
- Vectorization report reported vectorization in the third level loop.
- Result: 2 GFLOPS

Step 3

- Started optimization process. Unrolled outer loop to compute C four elements at a time. (Matrices in column major order)

```
for ( j = 0; j < n; j += 4 ) { /* Loop over the columns of C, unrolled by 4 */
        for ( i = 0; i < m; i += 1 ) { /* Loop over the rows of C */
                /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
                one routine (four inner products) */
                innerKernel_1x4( k, &A( i, 0 ), lda, &B( 0, j ), ldb, &C( i, j ), ldc );
        }
}
```

- Result: No noticeable Gain

Step 4

- Loaded C being updated into registers and then copied back to matrix.
- Started using pointers to address B.
- Code changes to:

```
bp0_pntr = &B( 0, 0 ); bp1_pntr = &B( 0, 1 );
bp2_pntr = &B( 0, 2 ); bp3_pntr = &B( 0, 3 );
for ( p=0; p<k; p++ ){
  a_0p_reg = A( 0, p );

  c_00_reg += a_0p_reg * *bp0_pntr++;
  c_01_reg += a_0p_reg * *bp1_pntr++;
  c_02_reg += a_0p_reg * *bp2_pntr++;
  c_03_reg += a_0p_reg * *bp3_pntr++;
}
// Copy back to C( i, j ) to C( i, j+4 )
```

Step 4 (2)

- Tried other improvements here like unrolling the innermost loop by 4.
- Also tried using indirect access instead of direct pointer access.
  For example: Instead of <`bp0_pntr++`> used <`bp0_pntr+x`>. Where x is the row we wish to access.
- Not much improvement with all these steps combined.
- Result: ~4 GFLOPS

Step 5

- Start computing blocks of 4x4 instead of 1x4 for C. So the outer loops change to something like:

```
for ( j = 0; j < n; j += 4 ) { /* Loop over the columns of C, unrolled by 4 */
  for ( i = 0; i < m; i += 4 ) { /* Loop over the rows of C */
    /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
    one routine (four inner products) */

    innerKernel_4x4( k, &A( i, 0 ), lda, &B( 0, j ), ldb, &C( i, j ), ldc );
  }
}
```

- We used the optimizations from above and started using registers for B.

Step 5 (2)

- Not much improvement till now but then we switched to using 128 bit vectors.

```
typedef union
{
  // 128 bits to hold two doubles
  __m128d v;
  double d[2];
} v2df_t;
```

- We use these structures to efficiently store 4x4 blocks of C. Still not much improvement. ~5 GFLOPS.
- Switched to `_m256d` registers. Moved up a bit to ~7 GFLOPS
  (KNL only has legacy support for these instructions.)

Step 5 (3)

- Switched to `_m512d` registers. Now we can store 8 doubles in one register. Hence started computing 4x8 blocks of C.

```
for ( j = 0; j < n; j += 8 ) { /* Loop over the columns of C, unrolled by 8 */
  for ( i = 0; i < m; i += 4 ) { /* Loop over the rows of C */
    innerKernel_4x8( k, &A( i, 0 ), lda, &B( 0, j ), ldb, &C( i, j ), ldc );
  }
}
```

- The performance jumped.
- Result: ~13 GFLOPS.

Step 6

- Switched to 8x8 blocks.

```
for (int n = 0; n < N_; n+=8) {
        for (int m = 0; m < M_; m+=8) {
          inner512_8x8(K_, &A(m,0), lda, &B(0,n),ldb ,&C(m,n), ldc);
        }
}
```

- Also started using intrinsics to add (`_mm512_add_pd`) and multiply (`_mm512_mul_pd`) doubles.
- CPU Utilization increased further to about 20 GFLOPS.

Step 6 (2)

- We used the same optimization as before unrolling the innermost loop by 4.

```
void inner512_8x8(int K_, double* A, int lda, double* B, int ldb, double* C, int ldc) {
  /** Declare 512d vector registers **/
  for (int k = 0; k < K_; k+=4) { // Step by 4
      /** Same computation as before now unrolled by 4 **/
  }
}
```

- Result: ~23 GFLOPS

Step 7

- Started to compute blocks of C instead of 8x8. So the outer loop gets broken into two steps

```
for (int k = 0; k < K; k += kc) {
        kb = min(K - k, kc);
        for (int m = 0; m < M; m += mc) {
                mb = min(M - m, mc);
                innerKernel(mb, N, kb, &A(m, k), lda, &B(k, 0), ldb, &C(m, 0), ldc, m == 0);
        }
}
```

  and another loop which computes it with a step size of 8.
- This is done to help performance for larger problem sizes be the same as smaller problem sizes.

Step 8

- We attempted to pack 8xK blocks of matrices A and Kx8 blocks of matrix B.

```
// pack A in contiguous memory
void packA(int K_, double* A, int lda, double* A_dest) {
        //#pragma vector always
        for (int k = 0; k < K_; k++) {
                double
                 *a_ij_ptr = &A(0,k);
                 *A_dest++ = *a_ij_ptr; *A_dest++ = *(a_ij_ptr+1);
                 *A_dest++ = *(a_ij_ptr+2); *A_dest++ = *(a_ij_ptr+3);
                 *A_dest++ = *(a_ij_ptr+4); *A_dest++ = *(a_ij_ptr+5);
                 *A_dest++ = *(a_ij_ptr+6); *A_dest++ = *(a_ij_ptr+7);
        }
}
```

- This surprisingly did not increase the performance for us.
- Result: ~20 GFLOPS (Because of overhead of packing)

Conclusion

- Despite trying different combinations for compiler optimization, forced vectorization and intrinsics we could not go beyond ~23 GFLOPS.

- Matrix packing did not help. Research suggests that this should have helped with a decent boost.