Greg Roberts (cs61c-il)
Owen Lu (cs61c-ck)

Using SSE Registers in the Innermost Loop

In order to leverage the use of SSE registers in our multiplication, we first transposed the first matrix onto a 16 byte-aligned boundary. This was helpful because the rows of the first matrix were now set up as columns and thus could be accessed in blocks of four floats and stored into SSE registers. Our innermost loop filled four registers with four vectors from successive columns in the transposed matrix. It then set two registers with two vectors from adjacent columns in the second matrix. The values from the matrices were multiplied and added in order to accumulate eight dot products. Getting large amounts of information from both matrices at a time, especially the first matrix, and calculating multiple dot products allowed us to minimize the amount of loading operations from memory into SSE registers. Also, since we accessed four columns at a time from the transposed matrix, we were able to store four floats at a time into the result matrix. This greatly reduced the number of individual stores we had to perform.

Handling the Fringes

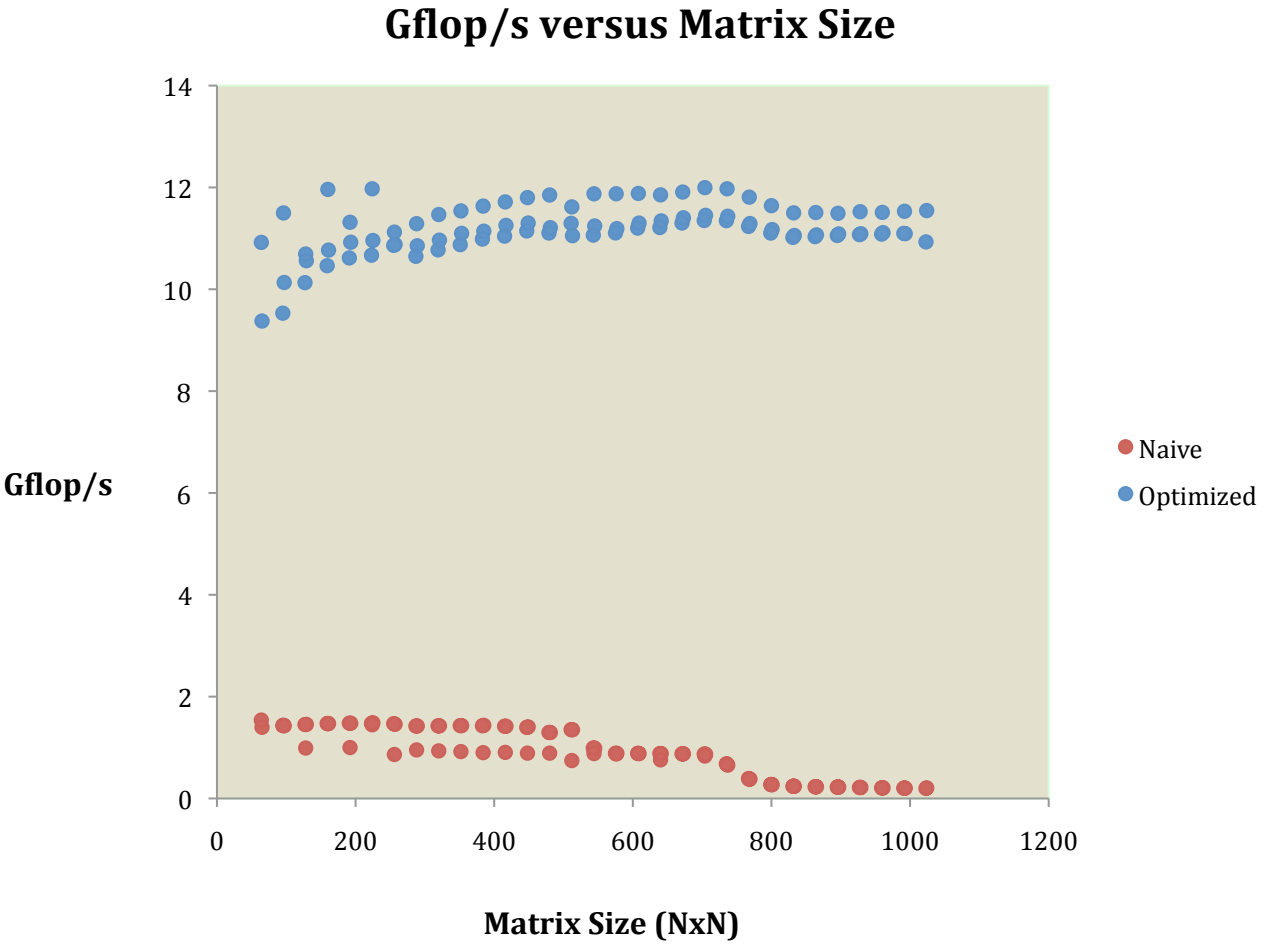
Define:
i : indexes rows of A/columns of A transpose
j: indexes columns of B
k : dot product iterator

We needed a fringe cleanup at each nested level due to register blocking. In every case, the loop index had to be incremented one at a time. When k reached the edge of the matrix, scalar float operations were used to finish the dot product computation. All register blocking was preserved at this stage.

When j reached the edge of B, the dot product was computed normally using simd instructions, but only a single _m128 from B was used at a time. A similar k-cleanup was then repeated.

When i reached the edge of A, j began incrementing by 8, since we then had more unused xmm registers. This stage had its own j and k cleanups similar to the code before, but it could no longer load/store 128 bits at a time from/into C.

Greg Roberts (cs61c-il)
Owen Lu (cs61c-ck)



Assembly Code

Our code used 16 total xmm registers. We did not have to spill any to the stack during our innermost loop.

Our compiled assembly code had the following breakdown of scalar floating point instructions:

- addss: 33
- mulss: 21
- movss: 41
- Total: 95

These occurred because of the transpose where we stored values that were not contiguous in memory. Also, in dealing with the fringe in the k-loop (dot-product iteration), we had to perform floating point operations on values that were not in vector form.