Hairan Zhu (cs61c-eu)
Benjamin Han (cs61c-mm)
March 18, 2012

1 SSE and SIMD

The innermost loop takes advantage of xmm 128-bit wide registers to operate single instructions on 4 floating point numbers simultaneously using the _mm_mul_ps and _mm_add_ps commands. The innermost loops unrolls 4 sets of multiply and add SSE instructions on xmm registers.
Our loop is ordered kji and we used a stride of 16 for loop tiling (an outer kji loop advances by the stride length and an inner kj loop advances by 1). The innermost i loop is unrolled using SSE instructions.

Within the inner k loop, we load a subcolumn of A into 4 XMM registers. XMM1 gets ($A_{i,k}$, $A_{i+1,k}$, $A_{i+2,k}$, $A_{i+3,k}$), XMM2 gets the next 4 elements in column k, and so on. Since we won't reload A in the innermost loop, this is an example of register-blocking.

Within the innermost j loop, we load $B_{j,k}$ into all 4 elments of a XMM register. For each of the 4 A registers, we compute $< A_{i,k} >$ * $B_{k,j}$ using SIMD mulitply. Basically, this gives $< A_{i,k}$ * $B_{k,j}$, ..., $A_{i+15,k}$ * $B_{k,j} >$.

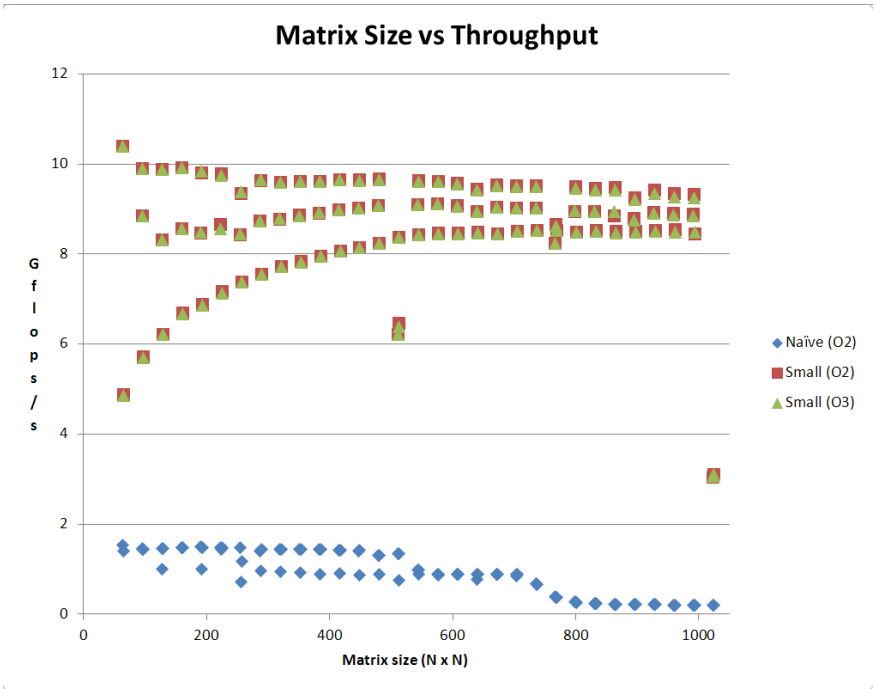
Then we load a subcolumn of C in 4 XMM registers, representing $< C_{i,j}$, ..., $C_{i+15,j} >$. We add the products to it, and then store back – all done using SIMD. The end result is that we've now computed $< A_{i,k}$ * $B_{k,j} + C_{i,j}$, ..., $A_{i+15,k}$ * $B_{k,j}$, $C_{i+15,j} >$. In other words, the partial sum of one of C's subcolumns.

2 Matrix Padding

We zero-pad the input matrices to a size divisible by stride length, which allowed us to use SSE operations. Afterwards, we unpad the result matrix C and copy it back into the original address.

Hairan Zhu (cs61c-eu)
Benjamin Han (cs61c-mm)
March 18, 2012

3 Figures



The plot demonstrates that the naive implementation was nearly linear for n < 700 before rapidly plummeting for larger values. For our serial (small) implementation, we found three relationships between matrix size and gflops: (n % stride) == 0 (linear decreasing), (n % stride) == stride - 1 (linear), and (n % stride) == 1 (logarithmic increasing). Notable values were 511, 512, and 1023 and 1024. The input matrix addresses values may have lined up the stride and/or register associativity, resulting in a large number of conflict misses and significantly slower throughput. Values near $768 = 512 + 256$ demonstrated slightly lower throughput as well.

4 Assembly Code

4.1 XMM

Our code makes use of 5 xmm registers.

4.2 Spilled values

The function does not spill values to the stack in the innermost loop.

4.3 Scalar floating point instruction

We use two movss calls in matrix multiply function: one for each case (n % stride == 16 and else). These calls are necessary to transfer the final value of the C element as a floating point back in to memory.