Experiments with Cache-Oblivious Matrix Multiplication for 18.335

Steven G. Johnson
MIT Applied Math

platform: 2.66GHz Intel Core 2 Duo, GNU/Linux + gcc 4.1.2 (-O3) (64-bit), double precision
(optimal) Cache-Oblivious Matrix Multiply

\[ C_{m \times p} \times A_{m \times n} \times B_{n \times p} = \]

\textit{divide and conquer:}
\begin{itemize}
  \item divide \( C \) into 4 blocks
  \item compute block multiply recursively
\end{itemize}

achieves optimal \( \Theta(n^3/ \sqrt{Z}) \) cache complexity
A little C implementation (~25 lines)

/* \( C = C + AB \), where \( A \) is \( m \times n \), \( B \) is \( n \times p \), and \( C \) is \( m \times p \), in row-major order. Actually, the physical size of \( A \), \( B \), and \( C \) are \( m \times fdA \), \( n \times fdB \), and \( m \times fdC \), but only the first \( n/p/p \) columns are used, respectively. */

void add_matmul_rec(const double *A, const double *B, double *C, int m, int n, int p, int fdA, int fdB, int fdC)
{
    if (m+n+p <= 48) { /* <= 16x16 matrices "on average" */
        int i, j, k;
        for (i = 0; i < m; ++i)
            for (k = 0; k < p; ++k) {
                double sum = 0;
                for (j = 0; j < n; ++j)
                    sum += A[i*fdA + j] * B[j*fdB + k];
                C[i*fdC + k] += sum;
            }
    } else { /* divide and conquer */
        int m2 = m/2, n2 = n/2, p2 = p/2;

        add_matmul_rec(A, B, C, m2, n2, p2, fdA, fdB, fdC);
        add_matmul_rec(A+n2, B+n2*fdB, C, m2, n-n2, p2, fdA, fdB, fdC);

        add_matmul_rec(A, B+p2, C+p2, m2, n2, n-p2, fdA, fdB, fdC);
        add_matmul_rec(A+n2, B+p2+n2*fdB, C+p2, m2, n-n2, n-p2, fdA, fdB, fdC);

        add_matmul_rec(A+m2*fdA, B, C+m2*fdC, m-m2, n2, p2, fdA, fdB, fdC);
        add_matmul_rec(A+m2*fdA+n2, B+n2*fdB, C+m2*fdC, m-m2, n-n2, p2, fdA, fdB, fdC);

        add_matmul_rec(A+m2*fdA, B+p2, C+m2*fdC+p2, m-m2, n2, n-p2, fdA, fdB, fdC);
        add_matmul_rec(A+m2*fdA+n2, B+p2+n2*fdB, C+m2*fdC+p2, m-m2, n-n2, n-p2, fdA, fdB, fdC);
    }
}

void matmul_rec(const double *A, const double *B, double *C, int m, int n, int p)
{
    memset(C, 0, sizeof(double) * m*p);
    add_matmul_rec(A, B, C, m, n, p, n, p, p);
}

note: base case is \(~16\times16\)

recursing down to \(1\times1\)

would kill performance
(1 function call per element, no register re-use)

dividing \(C\) into 4
— note that, instead, for very non-square matrices, we might want to divide \(C\) in 2 along longest axis
No Cache-based Performance Drops!
...but absolute performance still sucks

(optimized BLAS dgemm from ATLAS)

(of course, there are lots of little optimizations, but there must be something big…?)

÷ “unfair” factor of 2 from using SSE2 instructions

if this difference is not L1/L2 cache, what is it?
Registers \texttt{.EQ.} Cache

- The registers ($\sim100$) form a very small, almost ideal cache
  - Three nested loops is not the right way to use this “cache” for the same reason as with other caches

- Need long blocks of unrolled code: load blocks of matrix into local variables (= registers), do matrix multiply, write results
  - Loop-free blocks = many optimized hard-coded base cases of recursion for different-sized blocks … often automatically generated (ATLAS)
  - Unrolled $n \times n$ multiply has $(n^3)!$ possible code orderings — compiler cannot find optimal schedule (NP hard) — cache-oblivious scheduling can help (c.f. FFTW), but ultimately requires some experimentation (automated in ATLAS)