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

divide and conquer:
divide C into 4 blocks
compute block multiply recursively

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

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+n2, B+p2+n2*fdB, C, m2, n-n2, p-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+n2, B+p2+n2*fdB, C+m2*fdC, m-m2, n-n2, p-p2, fdA, fdB, fdC);
    add_matmul_rec(A+m2*fdA+n2, B+p2+n2*fdB, C+m2*fdC, m-m2, n-n2, p-p2, fdA, fdB, fdC);
    add_matmul_rec(A+m2*fdA+n2, B+p2+n2*fdB, C+m2*fdC, m-m2, n-n2, p-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, p, p);
}
No Cache-based Performance Drops!
...but absolute performance still sucks

(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 .EQ. Cache

- The registers (~100) 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)