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

A little C implementation (~25 lines)

```c
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, p-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, B+p2, C+m2*fdC+p2, m-m2, 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, n, p, p);
}
```

No Cache-based Performance Drops!

...but absolute performance still sucks

```
<table>
<thead>
<tr>
<th>matrix size m</th>
<th>cache-oblivious multiply</th>
<th>naive matrix multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.0125</td>
<td>0.05</td>
</tr>
<tr>
<td>1000</td>
<td>0.025</td>
<td>0.4</td>
</tr>
<tr>
<td>10000</td>
<td>0.05</td>
<td>4.5</td>
</tr>
</tbody>
</table>
```

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 non 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)