MODULE 03 – HIGH PERFORMANCE COMPUTING

MATRIX COMPUTATION

HYUN MIN KANG @ UNIVERSITY OF MICHIGAN

Examining the naïve matrix multiplication

- The time complexity of the naïve matrix multiplication is *$O(nmk)$*
- Can we improve the time complexity?
- Even with the same time complexity, is there a way to reduce the actual running time substantially?

Using Eigen Library

- Download Eigen library at http://eigen.tuxfamily.org/
- Uncompress the library, and move Eigen/ subdirectory into your program
- Use the following lines in your code:

```
#include "Eigen/Dense"
using Eigen::MatrixXd;
```

Matrix multiplication using Eigen

```
int main(int argc, char** argv) {
  int32_t n1, n2, n3;
  int32_t i, j, k;
  n1 = atoi(argv[1]); n2 = atoi(argv[2]); n3 = atoi(argv[3]);
 printTime("Performing a single threaded (%d x %d) by (%d x %d) Eigen matrix multiplication", n1, n2, n2, n3);
  MatrixXd eA(n1, n2);
  for(i=0; i < n1; ++i) {
    for(j=0; j < n2; ++j)
      eA(i,j) = j+1.;
  }
  MatrixXd eB(n2, n3);
  for(i=0; i < n2; ++i) {
    for(j=0; j < n3; ++j)
      eB(i,j) = 1./(i+1);
  }
 MatrixXd eC = eA * eB;
 printTime("Finished performing a single-threaded Eigen multiplication. Sum = %lg", eC.sum());
```

Running time is only 0.66s

```
$ time ./mat_mult 1000 10000 1000
[2016/11/09 06:12:12.911638]    Performing a single
threaded (1000 x 10000) by (10000 x 1000) Eigen
matrix multiplication
[2016/11/09 06:12:13.543937]    Finished performing
a single-threaded Eigen multiplication. Sum =
1e+10
```

How come it is so fast?

It's all about optimization

- **Efficient matrix libraries make use of**
  - Cache-conscious memory access by "blocks"
  - Single-instruction, multiple data (SIMD) vectorization to increase the computational efficiency.
  - Use more efficient algorithms for matrix multiplication.

A recursive algorithm for matrix multiplication

$$A, B, C \in R^n = R^{2^m}$$

$$C = AB$$

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

$$C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Time complexity of the recursive algorithm

$$
\begin{aligned}
T(n) &= 8T\left(\frac{n}{2}\right) + \Theta(n^2) \\
&= 64T\left(\frac{n}{4}\right) + 8\Theta\left(\frac{n^2}{4}\right) + \Theta(n^2) \\
&= 2^9T\left(\frac{n}{8}\right) + 64\Theta\left(\frac{n^2}{16}\right) + 8\Theta\left(\frac{n^2}{4}\right) + \Theta(n^2) \\
&= 2^{3m}T(1) + \sum_{i=1}^{m} 2^{i-1}\Theta(n^2) \\
&= \Theta(n^3) + (2^m - 1)\Theta(n^2) = \Theta(n^3)
\end{aligned}
$$

The 7 matrices for Strassen's algorithm

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$
$$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$$
$$M_4 = A_{2,2}(B_{2,1} - B_{1,1})$$
$$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$$
$$M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$
$$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

Strassen's algorithm

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$
$$C_{1,2} = M_3 + M_5$$
$$C_{2,1} = M_2 + M_4$$
$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

- We now only need 7 matrix multiplications rather than 8

Time complexity of Strassen's algorithm

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$= 49T\left(\frac{n}{4}\right) + 7\Theta\left(\frac{n^2}{4}\right) + \Theta(n^2)$$

$$= 7^m T(1) + \sum_{i=1}^{m} (1.75)^{i-1}\Theta(n^2)$$

$$= \Theta(n^{\log_2 7}) + (n^{\log_2 \frac{7}{4}} - 1)\Theta(n^2) \approx \Theta(n^{2.807})$$

How much difference does this make?

- Suppose that n=10,000

$$\frac{10,000^3}{10,000^{2.807}} \approx 5.9$$

- For n=1,000,000

$$\frac{1,000,000^3}{1,000,000^{2.807}} \approx 14.3$$

Understanding cache effects

```
vector< vector<double> > Bt;
Bt.resize(n3);
for(i=0; i < n3; ++i) {
  Bt[i].resize(n2);
  for(j=0; j < n2; ++j) {
    Bt[i][j] = 1.0/(j+1.);
  }
}
for(i=0; i < n1; ++i) {
  for(j=0; j < n3; ++j) {
    for(k=0; k < n2; ++k) {
      C[i][j] += ( A[i][k] * Bt[j][k] );
    }
    sum += C[i][j];
  }
}
```

Results (single-threaded) : 134s vs. 36s

```
$ time ./mat_mult 1000 10000 1000
[2016/11/09 09:38:57.236854]Initializing 1000 by 10000 matrix
[2016/11/09 09:38:57.368872]Initializing 10000 by 1000 matrix
[2016/11/09 09:38:57.548605]Performing a naive single-threaded (1000 x 10000) by (10000 x 1000) matrix multiplication
[2016/11/09 09:41:11.593900]Finished performing a naive single-threaded matrix multiplication. Sum = 1e+10
[2016/11/09 09:41:11.593958]Performing a transposed single-threaded (1000 x 10000) by (10000 x 1000) matrix multiplication
[2016/11/09 09:41:47.593993]Finished performing a transposed single-threaded matrix multiplication. Sum = 1e+10
```

Much more sophisticated optimizations are implemented in efficient matrix libraries

OpenBLAS : A more efficient matrix library

1. Download OpenBLAS by using
   `git clone https://github.com/xianyi/OpenBLAS.git`

2. Go to the directory and build using
   `make CC=gcc FC=gfortran -j 10`

3. Include `<cblas.h>` in the code

4. When compile, use
   `-I{/path/to/OpenBLAS]` as argument, and add
   `/path/to/OpenBLAS/libopenblas.a` at the end to link

Using OpenBLAS for matrix multiplication

```
double* aA = new double[n1*n2];
for(i=0; i < n1; ++i) {
  for(j=0; j < n2; ++j)
    aA[i + j * n1] = j+1.;
}
double* aB = new double[n2*n3];
for(i=0; i < n2; ++i) {
  for(j=0; j < n3; ++j)
    aB[i + j * n2] = 1./(i+1);
}
double* aC = new double[n1*n3];
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, n1, n3, n2, 1.0, aA, n1, aB,
n2, 0, aC, n1);
sum = 0;
for(i=0; i < n1; ++i) {
  for(j=0; j < n3; ++j)
    sum += aC[i + j * n1];
}
delete[] aA; delete[] aB; delete[] aC;
```

Running time is now only 0.34s

```
$ time ./mat_mult 1000 10000 1000
[2016/11/09 06:12:13.543997]    Performing a single
threaded (1000 x 10000) by (10000 x 1000) OpenBLAS
matrix multiplication
[2016/11/09 06:12:13.889314]    Finished performing
a single-threaded OpenBLAS multiplication. Sum =
1e+10
```

Eigen or OpenBLAS?

Eigen

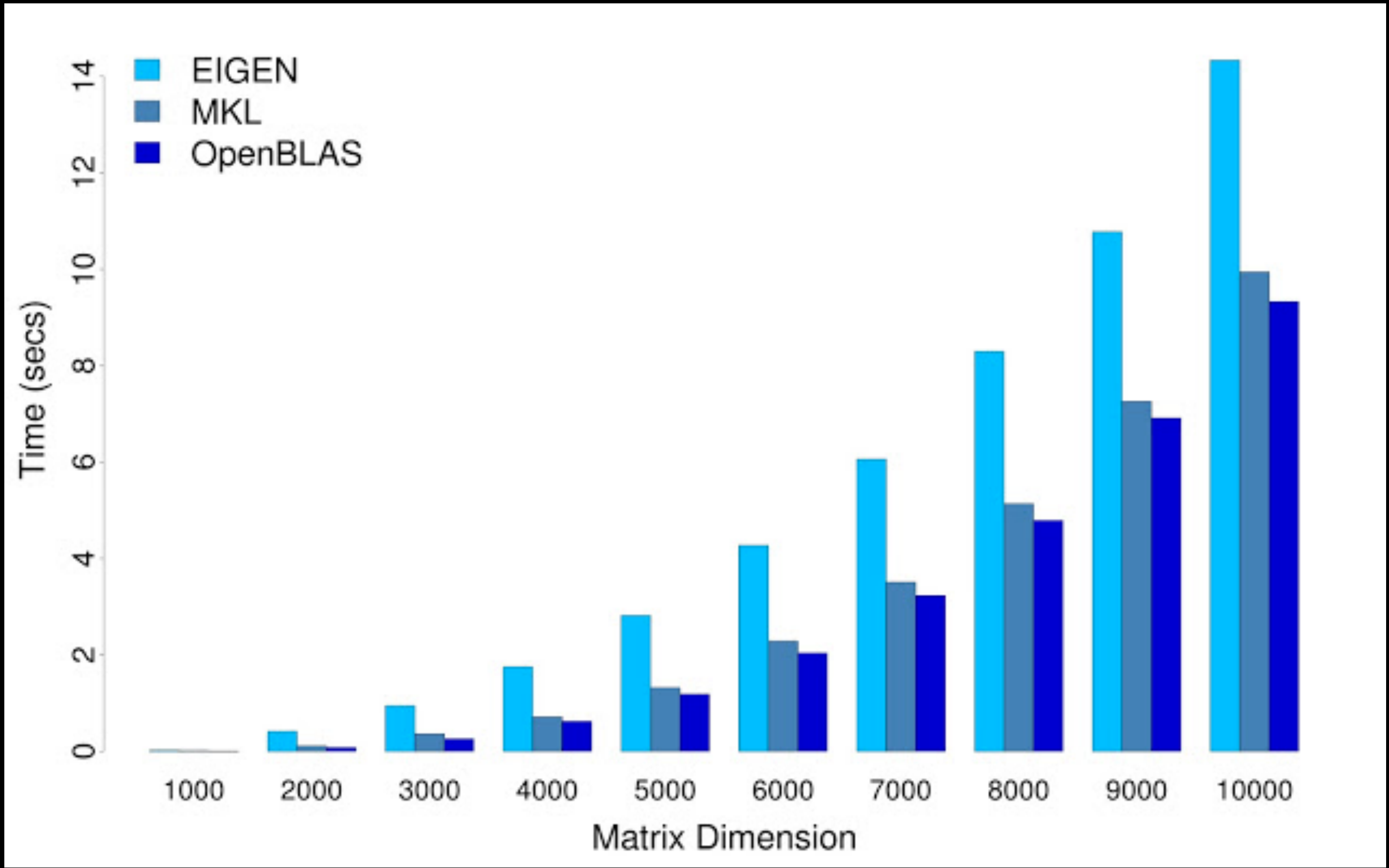
- Easy to use
- No library required
- C++ friendly
- Not as optimal as BLAS packages or R matrix library

OpenBLAS

- Very well optimized
- Need to build the library
- Cumbersome to use
- BLAS/LAPACK has long history with wide usage

Using NumericMatrix in Rcpp would be also quite efficient

Performance benchmark examples



http://gcdart.blogspot.co.uk/2013/06/fast-matrix-multiply-and-ml.html

Time complexity of common matrix algebra

- **Matrix inversion**
  - Gaussian-Jordan elimination $O(n^3)$
  - Strassen algorithm $O(n^{2.807})$
- **Singular value decompoision (SVD)** $O(mn^2)\ (m \leq n)$
- **LU decomposition** $O(n^3)$
- **QR decomposition** $O(n^3)$

Summary

- **The performance of matrix operations can be vary by a lot by the algorithms and implementations.**
  - Even with the same time complexity, sometimes you can make a substantial difference in the algorithm by careful optimizations.

- **Eigen and OpenBLAS are useful software libraries to use in software implementation involving large matrices.**