

A Scientific Library Case Study: Matrix Multiplication

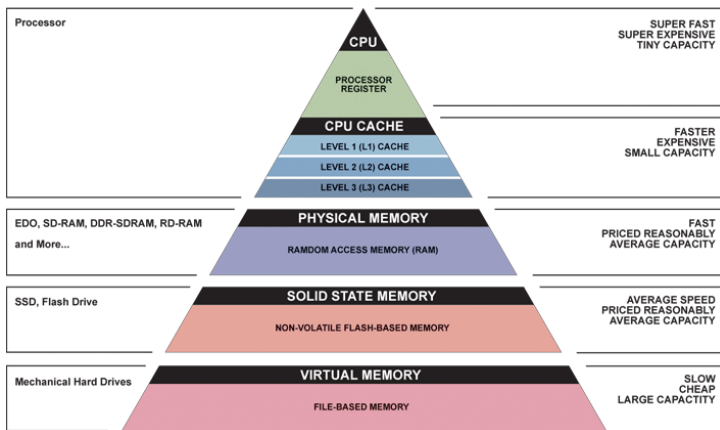
What the library does and what you wouldn't dare to do...

Massimiliano Culpo¹

¹CINECA - SuperComputing Applications and Innovation Department

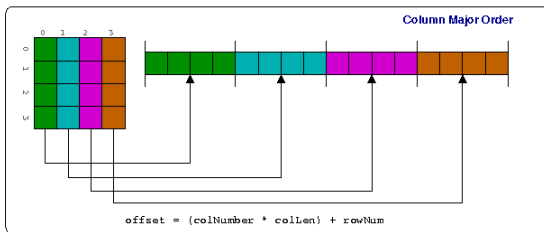
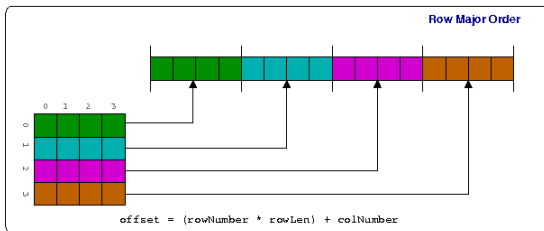
- 1 Outline
- 2 Review of basic concepts
- 3 Matrix Multiplication Algorithm
- 4 Conclusions

Hierarchy of memory levels



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

Memory layout for matrices



Warm-up: poor's man matrix allocation

```
/* Allocate a double matrix with many malloc */  
double** allocate_matrix(int nrows, int ncols) {  
    double ** A;  
    /* Allocate space for row pointers */  
    A = (double** ) malloc( nrows*sizeof(double*) );  
    /* Allocate space for each row */  
    for (int ii = 0; ii < nrows; ++ii) {  
        A[ii] = (double*) malloc( ncols*sizeof(double) );  
    }  
    return A;  
}
```

Warm-up: allocating the raw-memory

```
/* Allocate a double matrix with one malloc */  
double** allocate_matrix(int nrows, int ncols) {  
    double ** A;  
    /* Allocate enough raw space */  
    A = (double** ) malloc(  nrows*ncols*sizeof(double)  
                          + nrows*sizeof(double*) );  
    /* Initialize correctly the pointers */  
    A[0] = (double*) &A[nrows];  
    for (int ii = 1; ii < nrows; ++ii) {  
        A[ii] = (double*) &A[ii-1][ncols];  
    }  
    return A;  
}
```

Matrix Multiplication: a naïve implementation

```
/* Matrix are pointers of pointers to double */  
double** C, A, B  
const size_t m, n, p  
  
/* Each inner cycle computes an element of C */  
for( size_t ii = 0; ii < m; ++ii)  
    for( size_t jj = 0; jj < n; ++jj)  
        for( size_t kk = 0; kk < p; ++kk)  
            C[ii][jj] += A[ii][kk]*B[kk][jj];
```

Better use of spatial locality: loop permutation

```
/* Matrix are pointers of pointers to double */  
double** C, A, B  
const size_t m, n, p  
  
/* Each inner cycle computes partial sums  
with A[ii][kk] fixed. C and B are accessed  
in row major order. */  
for( size_t ii = 0; ii < m; ++ii)  
  for( size_t kk = 0; kk < p; ++kk)  
    for( size_t jj = 0; jj < n; ++jj)  
      C[ii][jj] += A[ii][kk]*B[kk][jj];
```


Better use of spatial locality: transposition

```
/* Matrix are pointers of pointers to double */  
double** C, A, B, T  
const size_t m, n, p  
  
/* Transpose B into T */  
for( size_t ii = 0; ii < n; ++ii)  
    for( size_t jj = 0; jj < p; ++jj)  
        T[ii][jj] = B[jj][ii];  
  
/* Each inner cycle computes an element of C */  
for( size_t ii = 0; ii < m; ++ii)  
    for( size_t jj = 0; jj < n; ++jj)  
        for( size_t kk = 0; kk < p; ++kk)  
            C[ii][jj] += A[ii][kk]*T[jj][kk];
```

Remove aliasing and simplify dereferentiation

```
/* Matrix are now pointers to double */  
double* restrict C, A, B, T  
const size_t m, n, p  
  
/* Transpose B into T */  
for( size_t ii = 0; ii < n; ++ii)  
    for( size_t jj = 0; jj < p; ++jj)  
        T[ii*p + jj] = B[jj*n + ii];  
  
/* Each inner cycle computes an element of C */  
for( size_t ii = 0; ii < m; ++ii)  
    for( size_t jj = 0; jj < n; ++jj)  
        for( size_t kk = 0; kk < p; ++kk)  
            C[ii*n + jj] += A[ii*p + kk]*T[jj*p + kk];
```

Intermezzo: aliasing in a few words

```
/* Aliasing between two pointers */  
void f (float *fltArrOut , float *fltArrIn , int n) {  
    int i = n;  
    fltArrOut[ i ]    = fltArrIn[ i ];  
    fltArrOut[ i+1 ] = fltArrIn[ i ];  
}  
/* Assembly: icc -std=c99 -g -O2 -fstrict-aliasing */  
# int i = n;  
movslq %edx,%rdx  
# load fltArrIn[ i ]  
mov (%rsi,%rdx,4),%eax  
# fltArrOut[ i ] = fltArrIn[ i ];  
mov %eax,(%rdi,%rdx,4)  
# load fltArrIn[ i ]  
mov (%rsi,%rdx,4),%ecx  
# fltArrOut[ i+1 ] = fltArrIn[ i ];  
mov %ecx,0x4(%rdi,%rdx,4)
```

Intermezzo: aliasing in a few words

```
/* Aliasing between two pointers */  
void f ( float * restrict fltArrOut ,  
         float * restrict fltArrIn , int n ) {  
    int i = n;  
    fltArrOut[ i ] = fltArrIn[ i ];  
    fltArrOut[ i+1 ] = fltArrIn[ i ];  
}  
  
/* Assembly: icc -std=c99 -g -O2 -restrict */  
# int i = n;  
movslq %edx,%rdx  
# load fltArrIn[ i ]  
mov (%rsi,%rdx,4),%eax  
# fltArrOut[ i ] = fltArrIn[ i ];  
mov %eax,(%rdi,%rdx,4)  
  
# fltArrOut[ i+1 ] = fltArrIn[ i ];  
mov %eax,0x4(%rdi,%rdx,4)
```

Better use of temporal locality: cache-oblivious algorithms

```
/* Matrix are now pointers to double */  
double* restrict C, A, B, T  
const size_t m, n, p, threshold  
  
/* Let r represent the maximum dimension */  
if ( r < threshold ) {  
/* Perform one of the previous algorithms  
   on a small block */  
} else {  
/* Divide dimension and recurse */  
}
```

Better use of temporal locality: cache-aware algorithms

```
/* Matrix are now pointers to double */  
double* restrict C, A, B, T  
const size_t m, n, p,  
  
/* Transpose B into T */  
/* Matrix multiplication */  
for( size_t ii = 0; ii < m; ii += TILE_II) {  
  for( size_t jj = 0; jj < n; jj += TILE_JJ) {  
    for( size_t kk = 0; kk < p; kk += TILE_KK){  
      for( size_t iix = ii; iix < min(ii+TILE_II,m); ++iix )  
        for( size_t jjx = jj; jjx < min(jj+TILE_JJ,n); ++jjx )  
          for( size_t kkx = kk; kkx < min(kk+TILE_KK,p); ++kkx )  
            C[iix*n + jjx] += A[iix*p + kkx]*T[jjx*p + kkx];
```

Other techniques in the bag of tricks

```
/* Bithacks on integers to avoid branch */
```

```
int max(const int x, const int y) {  
    return ( x ^ ((x ^ y) & -(x < y)) );  
}
```

```
int min(const int x, const int y) {  
    return ( y ^ ((x ^ y) & -(x < y)) );  
}
```

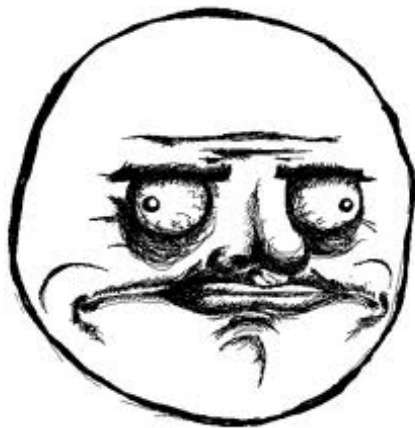
```
/* Local variables to remove false dependencies  
and use multiple registers */
```

```
double d1, d2;  
d1 = a[ii]; d2 = a[ii + 1];  
b[ii] = d1 + c; b[ii + 1] = d2*f;
```

Other techniques in the bag of tricks

```
/* Explicit unroll of loops to expose  
optimization opportunities */  
for (int ii = 0; ii < imax; ii += 4) {  
  a1 = a[ii];      a2 = a[ii + 1];  
  a3 = a[ii + 2];  a4 = a[ii + 3];  
  b1 = b[ii];      b2 = a[ii + 1];  
  b3 = b[ii + 2];  b4 = a[ii + 3];  
  c[ii]           = a1*b1; c[ii + 1] = a2*b2;  
  c[ii + 2] = a3*b3; c[ii + 3] = a4*b4;  
}
```

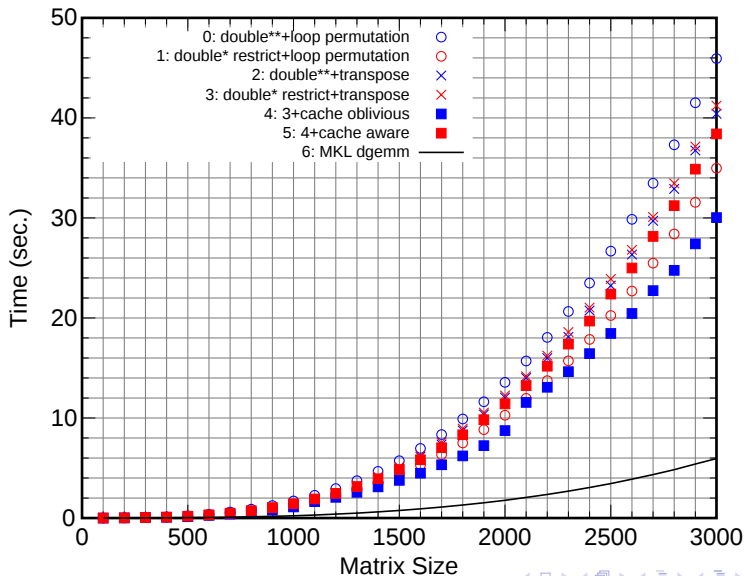

Final resource: go for intrinsics or assembly!



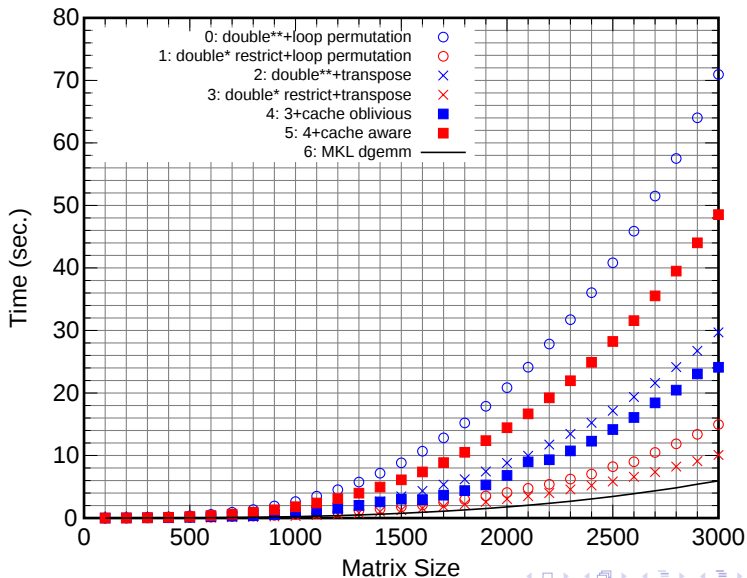
Let other people do the dirty job!

```
/* Matrix are now pointers to double */  
double* restrict C, A, B, T  
const size_t m, n, p,  
  
/* Painless and Fast! */  
#include "mkl_cblas.h"  
cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasNoTrans,  
             m, n, p, 1.0, A, p, B, n, 1.0, C, n );
```

GNU compiler



Intel compiler



Conclusions

- The actual performance of a *simple* program can be a complicated function of the architecture
- Architectural optimization may occur at different levels:
 - 1 registers
 - 2 L1-dcache
 - 3 L1-icache
 - 4 ...
- Low-level optimizations are *hard* to code by hand ...
- ...and they tend to leave an *unreadable* code!

IF VENDOR OPTIMIZED LIBRARIES ARE AVAILABLE USE THEM!

Now I want you to have fun and improve your code!

