

# BLISlab: A Sandbox for Optimizing GEMM

Jianyu Huang  
Chenhan Yu  
Tyler Smith  
Robert van de Geijn

SHPC UT Austin  
<http://shpc.ices.utexas.edu/>

February 16, 2016

## 1 Introduction

- Basic Linear Algebra Subprograms (BLAS)
- Matrix-matrix multiplication (GEMM)
- High-performance implementation
- Other similar exercises
- We need you! HPC's Got Talent!

## 2 Step 1: The Basics

- Simple matrix-matrix multiplication
- Setup
- Basic techniques
  - Using pointers
  - Loop unrolling
  - Register variables
  - Vector intrinsics

- Standard building blocks for performing vector and matrix operations.
- Under the hood of high performance scientific applications.

BLAS category	operations	computation complexity	memory complexity
BLAS1	vector-vector	$O(n)$	$O(n)$
BLAS2	matrix-vector	$O(n^2)$	$O(n^2)$
BLAS3	matrix-matrix	$O(n^3)$	$O(n^2)$

- GEMM is supported by BLAS with the call

```
   dgemm( transa, transb, m, n, k,  
          alpha, A, lda, B, ldb,  
          beta, C, ldc )
```

- By appropriately choosing transa and transb, dgemm computes

$$C := \alpha AB + \beta C; \quad C := \alpha A^T B + \beta C;$$

$$C := \alpha AB^T + \beta C; \quad C := \alpha A^T B^T + \beta C.$$

Here  $C$  is  $m \times n$  and  $k$  is the “third dimension”.

- We consider the simplified version of GEMM,

$$C := AB + C,$$

where  $C$  is  $m \times n$ ,  $A$  is  $m \times k$ , and  $B$  is  $k \times n$

Vendors	BLAS library name
Intel	MKL
AMD	ACML
Cray	LibSci
IBM	ESSL
Nvidia	cublas
AMD	clblas

Open Source:

ATLAS

**GotoBLAS**: from UT

OpenBLAS

**BLIS**: from UT

- 🖱 GEMM: From Pure C to SSE Optimized Micro Kernels by Michael Lehn
- 🖱 Optimizing Gemm by Robert van de Geijn

- The purpose of this tutorial is to guide you towards high-performance implementation of GEMM.
- Our ulterior motive is that our BLIS framework for implementing BLAS requires a so-called micro-kernel to be highly optimized for various CPUs.
- In teaching you the basic techniques, we are hoping to identify “The One” who will contribute the best micro-kernel.
- Our BLIS library supports architectures that include x86 processors by AMD and Intel, IBM's Power processors, ARM processors, and DSP processors.

## 1 Introduction

- Basic Linear Algebra Subprograms (BLAS)
- Matrix-matrix multiplication (GEMM)
- High-performance implementation
- Other similar exercises
- We need you! HPC's Got Talent!

## 2 Step 1: The Basics

- Simple matrix-matrix multiplication
- Setup
- Basic techniques
  - Using pointers
  - Loop unrolling
  - Register variables
  - Vector intrinsics



$$C := AB + C$$

where  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ ,  $m \times n$  matrices, respectively. Letting

$$A = \begin{pmatrix} \alpha_{0,0} & \dots & \alpha_{0,k-1} \\ \vdots & & \vdots \\ \alpha_{m-1,0} & \dots & \alpha_{m-1,k-1} \end{pmatrix}, B = \begin{pmatrix} \beta_{0,0} & \dots & \beta_{0,n-1} \\ \vdots & & \vdots \\ \beta_{k-1,0} & \dots & \beta_{k-1,n-1} \end{pmatrix},$$

$$C = \begin{pmatrix} \gamma_{0,0} & \dots & \gamma_{0,n-1} \\ \vdots & & \vdots \\ \gamma_{m-1,0} & \dots & \gamma_{m-1,n-1} \end{pmatrix}$$

$C := AB + C$  computes

$$\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p} \beta_{p,j} + \gamma_{i,j}.$$

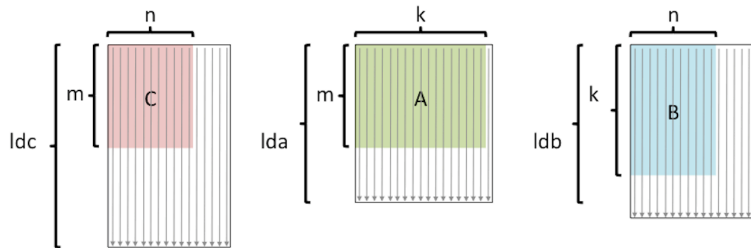
If  $A$ ,  $B$ , and  $C$  are stored in two-dimensional arrays  $A$ ,  $B$ , and  $C$ , the following pseudocode computes  $C := AB + C$ :

```
for i=0:m-1
    for j=0:n-1
        for p=0:k-1
            C( i,j ) := A( i,p ) * B( p,j ) + C( i,j )
        endfor
    endfor
endfor
```

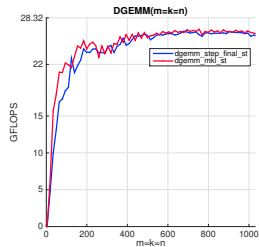
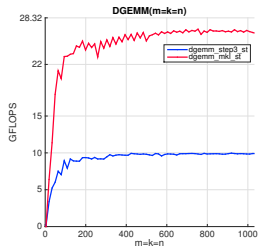
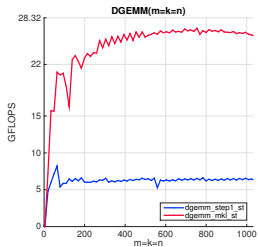
Counting a multiply and an add separately, the computation requires  $2mnk$  floating point operations (flops).

- Structure of directory step1
- Configure on `sourceme.sh`
- Compile, execute and collect result
- Draw the performance graph

```
#define C( i, j ) C[ (j)*ldc + (i) ]
```



```
bl_dgemm( m, n, k, A, lda, B, ldb, C, ldc )
```



```
for ( i = 0; i < m; i ++ ) {  
    for ( j = 0; j < n; j ++ ) {  
        C( i, j ) = 0.0;  
    }  
}
```

```
double *cp;  
for ( j = 0; j < n; j ++ ) {  
    cp = &C[ j * ldc ];  
    for ( i = 0; i < m; i ++ ) {  
        *cp++ = 0.0;  
    }  
}
```

Notice that we purposely exchanged the order of the loops so that advancing the pointer takes us down the columns of  $C$ .

Updating loop index *i* and the pointer *cp* every time through the inner loop creates considerable overhead. For this reason, a compiler will perform *loop unrolling*. Using an unrolling factor of four, our simple loop for setting *C* to zero becomes

```
double *cp;

for ( j = 0; j < n; j ++ ) {
    cp = &C[ j * ldc ];
    for ( i = 0; i < m; i += 4 ) {
        *(cp+0) = 0.0;
        *(cp+1) = 0.0;
        *(cp+2) = 0.0;
        *(cp+3) = 0.0;
        cp += 4;
    }
}
```

Importantly:



- `i` and `cp` are now only updated once every four iterations.
- `*(cp+0)` uses a machine instruction known as *indirect addressing* that is much more efficient than if one computed with `*(cp+k)` where `k` is a variable.
- When data is brought in for memory into cache, it is brought in a cache line of 64 bytes at a time. This means that accessing contiguous data in chunks of 64 bytes reduces the cost of memory movement between the memory layers.

Notice that when you unroll, you may have to deal with a “fringe” if, in this case, `m` is not a multiple of four.



```
double *cp;

for ( j = 0; j < n; j ++ ) {
    cp = &C[ j * ldc ];
    for ( i = 0; i < m; i += 4 ) {
        register double c0=0.0, c1=0.0, c2=0.0, c3=0.0;
        *(cp+0) = c0;
        *(cp+1) = c1;
        *(cp+2) = c2;
        *(cp+3) = c3;
        cp += 4;
    }
}
```

-  Intel Intrinsics Guide
-  Intel ISA Extensions

```
bl_dgemm( m, n, k, A, lda, B, ldb, C, ldc )
```