BLISlab: A Sandbox for Optimizing GEMM

Jianyu Huang Robert A. van de Geijn

Draft
February 8, 2016

Abstract

1 Introduction

Matrix-matrix multiplication (Gemm) is frequently used as a simple example with which to raise awareness of how to optimize modern processors. A reason is that the operation is simple to describe, challenging to fully optimize, and of practical importance. In this paper, we walk the reader through the techniques that underly the currently fastest implementations for CPU architectures.

1.1 A minimal history

Need to mention BLAS3 [1] paper.

For more than a decade after that paper, the intricacies of high-performance optimization of Gemm was considered to be sufficiently complex that it should be left to to the hardware vendors, yielding IBM's ESSL, Intel's MKL, Cray's ???, and AMD's ACML libraries, or auto-generated as advocated in papers on the Portable High Performance ANSI C (PHiPAC) guidelines for writing high-performance matrix-matrix multiplication in C [] and the Automatically Tuned Linear Algebra Software (ATLAS) [].

1.2 The BLIS-like Library Instantiation Software (BLIS)

More recently, the BLAS-like Library Instantiation Software (BLIS) "refactored" the approach pioneered by Goto, exposing additional loops around a *micro-kernel*, as described in

> Field G. Van Zee, Robert A. van de Geijn.
> BLIS: A Framework for Rapidly Instantiating BLAS Functionality.
> ACM Transactions on Mathematical Software (TOMS).
> Volume 41 Issue 3, June 2015, Article No. 14.

One goal of the BLIS paper was to further expose the layering of Goto's approach while simultaneously improving portability by reducing how much code must be written at a low level (e.g., in assembly code).

1.3 You too can optimize like a pro

The purpose of this note is to expose the basic techniques that underlie the best implementations of Gemm so that you too can achieve high-performance for such operations.

2 Step 1: The Basics

2.1 Simple matrix-matrix multiplication

In our discussions, we will consider the computation

$$C := AB + C$$

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

$$A = \begin{pmatrix} \alpha_{0,0} & \cdots & \alpha_{0,k-1} \\ \vdots & & \vdots \\ \alpha_{m-1,0} & \cdots & \alpha_{m-1,k-1} \end{pmatrix}, B = \begin{pmatrix} \beta_{0,0} & \cdots & \beta_{0,n-1} \\ \vdots & & \vdots \\ \beta_{k-1,0} & \cdots & \beta_{k-1,n-1} \end{pmatrix}, \text{ and } C = \begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \vdots & & \vdots \\ \gamma_{m-1,0} & \cdots & \gamma_{m-1,n-1} \end{pmatrix}.$$

$C := AB + C$ 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 as floating point numbers 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.

2.2 Set up

To let you efficiently learn about how to efficiently compute, you start your project with much of the infrastructure in place. We have structured the subdirectory, `step1`, somewhat like a project that implements a real library might. This may be overkill for our purposes, but how to structure a software project is a useful skill to learn.

Consider Figure 3, which illustrates the directory structure for subdirectory `step1`:

`README` Is a file that describes the contents of the directory and how to compile and execute the code.

`sourceme.sh` Is a file that configures the environment variables.

- `BLISLAB_USE_INTEL` determines whether you use Intel compiler or GNU compiler (`true` or `false`).
- `BLISLAB_USE_BLAS` determines whether your reference GEMM adopts `BLAS` implementation (if you have `BLAS` installed on your machine), or the simple triple loops implementation (`true` or `false`).
- `COMPILER_OPT_LEVEL` determines the optimization level for your GNU or Intel compiler (`O0`, `O1`, `O2`, `O3`).
- `OMP_NUM_THREADS` **and** `BLISLAB_IC_NT` determines your thread number for parallel version of your code. For your first step, you can just set them both to 1.

`dgemm` Is the subdirectory routines that implement GEMM can be found. In it

- `bl_dgemm_ref` contains the routine `dgemm_ref` that is a simple implementation of GEMM that you will use to check the correctness of your implementations.
- `my_dgemm` contains the routine `dgemm` that that initially is a simple implementation of GEMM and that you will optimize as part of the first step on your way to mastering how to optimize GEMM.

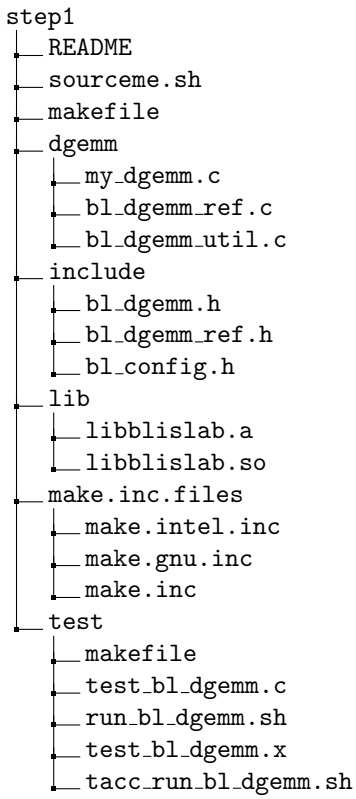


Figure 1: Structure of directory `step1`.

`bl_dgemm_util` contains utility routines that will come in handy later.

`include` This directory contains include files with various macro definitions and other header information.

`lib` This directory will hold libraries generated by your implemented source files (`libblislab.so` and `libblislab.a`). You can also install a reference library (e.g. OpenBLAS) in this directory to compare your performance.

`test` This directory contains "test drivers" and correctness/performance checking scripts for the various implementations.

`test_bl_dgemm.c` contains the "test driver" routine `test_bl_dgemm`.

`test_bl_dgemm.x` is the executable file for `test_bl_dgemm.c`.

`run_bl_dgemm.sh` contains a bash script to collect GFLOPS result for selected problem size.

`tacc_run_bl_dgemm.sh` contains a `SLURM` script for you to submit the job to TACC machines to measure performance.

What we want you to do is to start with the implementation in `my_dgemm.c` and optimize it by applying various standard optimization techniques.

The implementation in `my_dgemm.c` is a straight-forward implementation with three loops:

```
for ( i = 0; i < m; i ++ ) {                    // 2-th loop

  for ( j = 0; j < n; j ++ ) {                  // 1-th loop
```

```
        for ( p = 0; p < k; p ++ ) {             // 0-th loop

           C( i, j ) += A( i, p ) * B( p, j );

        }                                        // End 0-th loop
     }                                           // End 1-th loop
  }                                              // End 2-th loop
```

The first thing to notice is how two-dimensional arrays are mapped to memory in so-called *column-major order*. The reason for this choice is that the original BLAS assumed column-major storage of arrays because the interface was for Fortran users first. Examining
`C( i, j ) += A( i, p ) * B( p, j );`
we notice that, each operand is a `MACRO`. For example,
`#define C( i, j ) C[ (j)*ldc + (i) ]`
The linear array at address C is used to store elements $C_{i,j}$ so that the i, j element is mapped to location `j * ldc + i`. The way to view this is that the columns of C are each stored contiguously. However, think of matrix C as embedded in a large array that has `ldc` rows so that accessing a row means going through array `C` with stride `ldc`. The term *leading dimension* of two-dimensional array `C` is typically used to refer to the row dimension of this larger array, hence the variable `ldc` (leading dimension of `C`).

2.2.1 Execute the default implementation

By default, the exercise compiles and links with Intel's `icc` compiler, which will apply compiler optimizations (`O3` optimization level) to the code. You need to set the environment variable by executing:
`$ source sourceme.sh`
in the terminal, and you will see the output:
```
BLISLAB_USE_INTEL = true
COMPILER_OPT_LEVEL = O3
```

2.2.2 Change to the GNU compiler

Since we want you to explicitly learn about what kind of tricks lead to high performance, and because some of you may not have access to the Intel compiler, you should first change to using the GNU C compiler. You need first edit `sourceme.sh`:
`BLISLAB_USE_INTEL=false`
Then, similar to the default setting, you need to set the environment variable by executing:
`$ source sourceme.sh`
in the terminal, and you will observe:
```
BLISLAB_USE_INTEL = false
COMPILER_OPT_LEVEL = O3
```

2.2.3 Turn off optimization

Next, we want you to turn off the optimization performed by the compiler. This has two purposes: first, it means you are going to have to explicitly perform optimizations, which will allow you to learn about how architectures and algorithms interact. Second, it may very well be that the optimizing compiler will try to "undo" what you are explicitly trying to accomplish. Third, the more tricks you build into your code, the harder it gets for the compiler to figure out how to optimize. You need first edit `sourceme.sh`:
`COMPILER_OPT_LEVEL = O0`
Then, similar to the default setting, you need to set the environment variable by executing:
`$ source sourceme.sh`
in the terminal, and you will see the output:

```
BLISLAB_USE_INTEL = false
COMPILER_OPT_LEVEL = O0
```

2.2.4 (Optional) Turn on BLAS as reference implementation

By default, your reference GEMM implementation is a very slow triple-loop implementation. If you have `BLAS` installed on your test machine, you can adopt `BLAS` as your reference implementation by setting:
`BLISLAB_USE_BLAS=true`
in `sourceme.sh`. After executing `$ source sourceme.sh`, you will observe:
`BLISLAB_USE_BLAS = true`

2.3 Basic techniques

2.3.1 Using pointers

Now that optimization is turned off, the computation of the address where an element of a matrix exists is explicitly exposed. (An optimizing compiler would get rid of this overhead.) What you will want to do is to change the implementation in `my_gemm.c` so that it instead uses pointers. Before you do so, you may want to back up the original `my_gemm.c` in case you need to restart from scratch. Indeed, at each step you may want to back up in a separate file the previous implementations.

Here is the basic idea. Let's say we want to set all elements of C to zero. A basic loop, patterned after what you found in `my_gemm.c` might look like

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

Using pointers, we might implement this as

```
double *cp;

for ( j = 0; j < n; j ++ ) {
   cp = &C[ j * ldc ];           // point cp to top of jth column
   for ( i = 0; i < m; i ++ ) {
      *cp++ = 0.0;               // set the element that cp points to to zero and
                                 // advance the pointer.
   }
}
```

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

2.3.2 Loop unrolling

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 updates 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 it 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. If you set `???` to your unrolling factor, then the driver routine that tests your code will automatically only test problems that are nice multiples of the unrolling factor. For the sake of this exercise, you need not worry about this fringe.

2.3.3 Register variables

Notice that computation can only happen if data is stored in registers. A compiler will automatically transform code so that the intermediate steps that place certain data in registers is inserted. One can give a hint to the compiler that it would be good to keep certain data in registers as illustrated in the following somewhat contrived example:

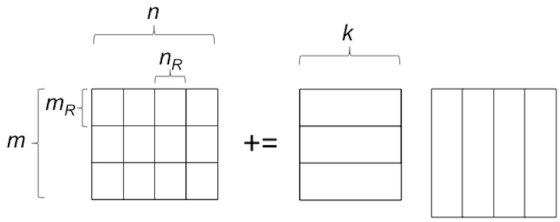
```
double *cp;

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;
   }
}
```

2.3.4 Putting it all together

We now ask you to employ the techniques discussed above to optimize `my_dgemm`. For now, just worry about trying to attain better performance for smallish matrices. In particular, consider the following picture:



What we want you to do is to write your code so that $m_R \times n_R$ blocks of C are kept in registers. You get to choose m_R and n_R, but you will want to update file `??` with those choices.

2.3.5 Vector intrinsics

You will notice that even for smallish matrices, your implementation performs (much) worse than the implementations that are part of MKL and/or BLIS. The reason is that the compiler is not using the fastest instructions for floating point arithmetic. These can be accessed either by using *vector intrinsic functions*, which allows you to explicitly utilize them from C, or by coding in assemply code. For now, let's not yet go there. We will talk more about this in Step 3.

3 Step 2: Blocking

3.1 Poorman's BLAS

Step 1 of this exercise makes you realize that with the advent of cache-based architectures, high-performance implementation of GEMM necessitated careful attention to the amortization of the cost of data movement between memory layers and computation with that data. To keep this manageable, it helps to realize that only a "kernel" that performs a matrix-matrix multiplication with relatively small matrices needs to be highly optimized, since computation with larger matrices can be blocked to then use such a kernel without an adverse impact on overall performance. This insight was first explicitly advocated in [**?**]

> Bo Køagström , Per Ling , Charles Van Loan.
> GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark.
> ACM Transactions on Mathematical Software (TOMS).
> Volume 24 Issue 3, p.268-302, Sept. 1998.

This at one point was referred to as "poorman's BLAS" in the sense that if one could only afford to optimize matrix-matrix multiplication (with a submatrix), then one could build GEMM, and other important matrix-matrix operations known as the level-3 BLAS, in terms of this. What we will see later is that actually in general this is a good idea, for the sake of modularity as well as for performance.

3.2 Blocked matrix-matrix multiplication

Key to blocking GEMM to take advantage of the hierarchical memory of a processor is understanding how to compute $C := AB + C$ when these matrices have been blocked. Partition

$$A = \begin{pmatrix} A_{0,0} & \cdots & A_{0,K-1} \\ \vdots & & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,K-1} \end{pmatrix}, B = \begin{pmatrix} B_{0,0} & \cdots & B_{0,N-1} \\ \vdots & & \vdots \\ B_{K-1,0} & \cdots & B_{K-1,N-1} \end{pmatrix}, \text{ and } C = \begin{pmatrix} C_{0,0} & \cdots & C_{0,N-1} \\ \vdots & & \vdots \\ C_{M-1,0} & \cdots & C_{M-1,N-1} \end{pmatrix}.$$

where $C_{i,j}$ is $m_i \times n_j$, $A_{i,p}$ is $m_i \times k_p$, and $B_{p,j}$ is $k_p \times n_j$. Then

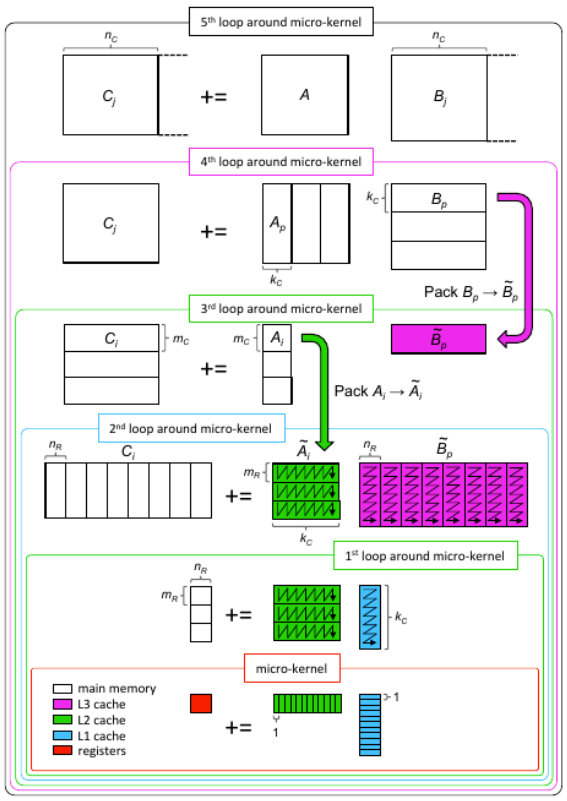
$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p} B_{p,j} + C_{i,j}.$$

4 Step 3: Blocking for Multiple Levels of Cache

4.1 The Goto Approach to Implementing GEMM

Around 2000, Kazushige Goto revolutionized how GEMM is implemented on current CPUs with his techniques that were first published in the paper

> Kazushige Goto, Robert A. van de Geijn.
> Anatomy of high-performance matrix multiplication.
> ACM Transactions on Mathematical Software (TOMS).
> Volume 34 Issue 3, May 2008, Article No. 12.



Loop 5 **for** $j_c = 0 : n-1$ **steps of** n_c
  $\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4 **for** $p_c = 0 : k-1$ **steps of** k_c
    $\mathcal{P}_c = p_c : p_c + k_c - 1$
    $B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow B_c$ // Pack into B_c
Loop 3 **for** $i_c = 0 : m-1$ **steps of** m_c
      $\mathcal{I}_c = i_c : i_c + m_c - 1$
      $A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow A_c$ // Pack into A_c
      // Macro-kernel
Loop 2 **for** $j_r = 0 : n_c - 1$ **steps of** n_r
        $\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1 **for** $i_r = 0 : m_c - 1$ **steps of** m_r
          $\mathcal{I}_r = i_r : i_r + m_r - 1$
          // Micro-kernel
Loop 0 **for** $k_r = 0 : k_c - 1$
            $C_c(\mathcal{I}_r, \mathcal{J}_r)$
              $+= A_c(\mathcal{I}_r, k_r)$ $B_c(k_r, \mathcal{J}_r)$
          endfor
        endfor
      endfor
    endfor
  endfor
endfor

Figure 2: Left: The Goto algorithm for matrix-matrix multiplication as refactored in BLIS. Right: the same algorithm, but expressed as loops.

A further "refactoring" of this approach was more recently described in

> Field G. Van Zee, Robert A. van de Geijn.
> BLIS: A Framework for Rapidly Instantiating BLAS Functionality.
> ACM Transactions on Mathematical Software (TOMS).
> Volume 41 Issue 3, June 2015, Article No. 14.

The advantage of the BLIS framework is that it reduces the kernel that must be highly optimized, possibly with vector intrinsics or in assemply code, to a *micro-kernel*.

The

4.2 Set up

Figure 3 illustrates the directory structure for subdirectory `step3`. Comparing to `step2`, we have modified/added the following directories/files:

`kernels` This directory contains the micro-kernel implementations for various architecture.

- `bli_dgemm_int_d8x4.c` Is a `AVX` intrinsics micro-kernel implementation for Sandy Bridge/Ivy Bridge micro-architecture.
- `bli_dgemm_asm_d8x4.c` Is a `AVX` assembly micro-kernel implementation for Sandy Bridge/Ivy Bridge micro-architecture.
- `bli_dgemm_asm_d8x6.c` Is a `AVX2` assembly micro-kernel implementation for Haswell micro-architecture.
- `bli_dgemm_asm_d12x4.c` Is an alternative `AVX2` assembly micro-kernel implementation for Haswell micro-architecture.

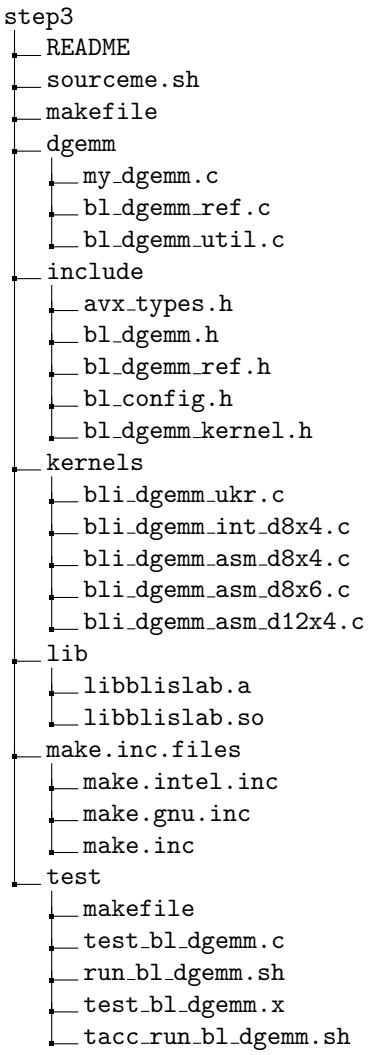


Figure 3: Structure of directory `step3`.

5 Conclusion

Conclusion.

Additional information

For additional information on FLAME visit

`http://www.cs.utexas.edu/users/flame/`.

Acknowledgements

We thank the other members of the FLAME team for their support. This research was partially sponsored by NSF grant CCF–***.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.