BLISlab: A Sandbox for Optimizing GEMM

Jianyu Huang Robert A. van de Geijn Others?

Draft
February 13, 2016

Abstract

The BLAS-like Library Instantiation Software (BLIS) is a framework for instantiation the Basic Linear Algebra Subprograms (BLAS) and similar functionality. The matrix-matrix operations known as the level-3 BLAS is an important subset that can achieve high performance by amortizing the cost of moving data between memory layers. The BLIS implementation of the level-3 BLAS require a micro-kernel, a matrix-matrix multiplication with small matrices, to be highly optimized.

This note describes a set of exercises that culminate in the implementation of the micro-kernel. On the one hand it is meant to equip the reader with the understanding that facilitates high-performance implementation of matrix-matrix multiplication. On the other hand it is meant to facilitate the "crowd sourcing" of the optimization of BLIS.

1 Introduction

Matrix-matrix multiplication (GEMM) is frequently used as a simple example with which to raise awareness of how to optimize code on modern processors. The 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 Basic Linear Algebra Subprograms (BLAS)

The Basic Linear Algebra Subprograms (BLAS) [9, 4, 3, 12] are an interface to a set of linear algebra operations upon which higher level linear algebra libraries. Of these, the level-3 BLAS constitute a set of matrix-matrix operations that, if all matrix operands are $n \times n$ in size, $O(n^3)$ computation is performed over $O(n^2)$ data so that the cost of moving data between memory layers (main memory, the caches, and the registers) can be amortized over many computations. As a result, high performance can often be achieved if these operations are carefully implemented.

1.2 Matrix-matrix multiplication (GEMM)

In particular, general matrix-matrix multiplication (GEMM) with double precision floating point numbers is supported by the BLAS with the call

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

which, by appropriately choosing `transa` and `transb`, computes

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

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 $k \times n$ in this paper.

1.3 High-performance implementation

The intricacies of high-performance implementations are such that implementation of the BLAS in general and GEMM in particular was often relegated to unsung experts who develop numerical libraries for the hardware vendors, for example as part of IBM's ESSL, Intel's MKL, Cray's LibSci, and AMD's ACML libraries. In the past, these libraries were often written at least partially in assembly.

The key paper [1] showed how an "algorithms and architectures" approach to hand-in-hand designing architectures, compilers, and algorithms allowed BLAS to be written in a high level language (Fortan) for the IBM Power architectures. The Portable High Performance ANSI C (PHiPAC) [2] project subsequently provided guidelines for writing high-performance code in C and suggested how to autogenerate and tune GEMM written this way. The Automatically Tuned Linear Algebra Software (ATLAS) [15, 16] built upon these insights and made autotuning and autogeneration of BLAS libraries mainstream.

As part of the current paper we discuss more recent papers on the Goto approach to implementing GEMM [5] and the BLIS refactoring of that approach [14], as well as other papers that are of more direct relevance.

1.4 Other similar exercises

There are others who have put together exercises based on GEMM. Recent efforts relevant to this paper are ☛ GEMM: From Pure C to SSE Optimized Micro Kernels by Michael Lehn at Ulm University and a wiki on ☛ Optimizing Gemm that we ourselves put together.

1.5 We need you!

The purpose of this paper 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. Think of it of our version of "HPC's Got Talent". Although we focus in our description on optimization for the Intel Haswell architecture, the setup can be easily modified to instead help you (and us) optimize for other CPUs. Indeed, BLIS itself supports architectures that include x86 processors by AMD and Intel, IBM's Power processors, ARM processors, and DSP processors [13, 10, 7].

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

```
step1
├── README
├── sourceme.sh
├── makefile
├── dgemm
│   ├── my_dgemm.c
│   ├── bl_dgemm_ref.c
│   └── bl_dgemm_util.c
├── include
│   ├── bl_dgemm.h
│   ├── bl_dgemm_ref.h
│   └── bl_config.h
├── lib
│   ├── libblislab.a
│   └── libblislab.so
├── make.inc.files
│   ├── make.intel.inc
│   ├── make.gnu.inc
│   └── make.inc
└── test
    ├── makefile
    ├── test_bl_dgemm.c
    ├── run_bl_dgemm.sh
    ├── test_bl_dgemm.x
    └── tacc_run_bl_dgemm.sh
```

Figure 1: Structure of directory `step1`.

```
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 4, 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` sets whether you use the Intel compiler (`true`) or the GNU compiler (`false`).
- `BLISLAB_USE_BLAS` indicates whether your reference GEMM employs `BLAS` implementation (`true` if you have `BLAS` installed on your machine), or the simple triple loops implementation (`false`).
- `COMPILER_OPT_LEVEL` sets the optimization level for your GNU or Intel compiler (`O0`, `O1`, `O2`, `O3`).

```
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
```

Figure 2: Simple implementation of GEMM.

`OMP_NUM_THREADS` **and** `BLISLAB_IC_NT` sets the number of threads used for parallel version of your code. For Step 1, you can just set them both to `1`.

`dgemm` Is the subdirectory where the routines that implement GEMM exist. In it

- `bl_dgemm_ref.c` 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.c` 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.
- `bl_dgemm_util.c` 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" for testing routine `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 performance results.
- `tacc_run_bl_dgemm.sh` contains a `SLURM` script for you to (optionally) submit the job to TACC machines to measure performance.

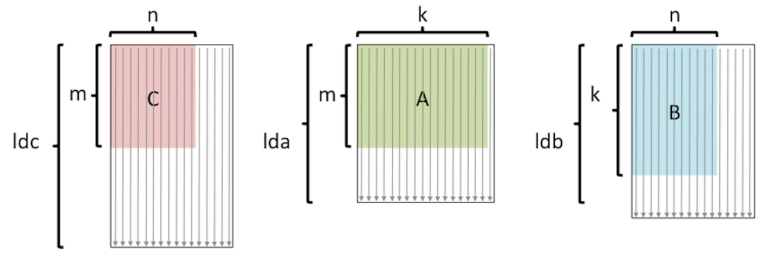
What we want you to do is to start with the implementation in `my_dgemm.c` and optimize it by applying various standard optimization techniques. The initial implementation in that file is the straight-forward implementation with the three loops given in Figure 2. The first thing to notice is how two-dimensional 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, earlier in the file

```
#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`). This is illustrated for all three matrices in the following figure:



in which the arrows are meant to indicate that columns are stored contiguously.

2.2.1 Configure the default implementation

By default, the exercise compiles and links with Intel's `icc` compiler, which will apply compiler optimizations (level `O3`) 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 Compile, execute and collect results

You can compile, execute your code and collect the performance result by executing

```
make
cd test
./run_bl_dgemm.sh
```

in `step1` subdirectory. You will see the performance result output:

```
run_step1_st=[
%m     %k    %n    %MY_GFLOPS %REF_GFLOPS
16     16    16    0.82  2.15
32     32    32    0.74  5.50
48     48    48    0.85  5.66
......
];
```

You can change the sampling block size in `run_bl_dgemm.sh`. Notice that if you have errors in your code, these will be reported as, for example,

```
C[ 0 ][ 0 ] != C_ref, 1.253000E+00, 2.253000E+00
```

2.2.3 Draw the performance graph

Finally, you can use `MATLAB` to draw your performance graph with our scripts. In `src/results` directory, after executing

```
./collect_result_step1
```

you will get a MATLAB file "step1_result.m", with the performance results. You can then execute

```
bl_dgemm_plot.m
```

in MATLAB, which will then generate the performance graph.

2.2.4 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 next change to using the GNU C compiler. For this, you must 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
```

2.2.5 Turn off optimization

Next, we want you to turn off the optimization performed by the compiler. This serves 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
```

2.2.6 (Optional) Use optimized BLAS library as reference implementation

By default, your reference GEMM implementation is a very slow triple-loop implementation. If you have a BLAS library installed on your test machine, you can adopt the dgemm from that library as your reference implementation by setting:

```
BLISLAB_USE_BLAS=true
```

in `sourceme.sh`. If you use Intel compiler, you don't need to explicitly specify the path of MKL. However, if you use GNU compiler, you need to specify the path of your BLAS library, for example,

```
BLAS_DIR=/home/lib/blis
```

After executing `$ source sourceme.sh`, you will observe:

```
BLISLAB_USE_BLAS = true
BLAS_DIR = /home/lib/blis
```

and now performance and accuracy comparisons of your implementation will be against this optimized library routine.

2.3 Basic techniques

In this subsection we describe some basic tricks of the trade.

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( i, j ) = 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. For the sake of this exercise, you need not worry about this fringe, as reiterated in Section 2.4.

2.3.3 Register variables

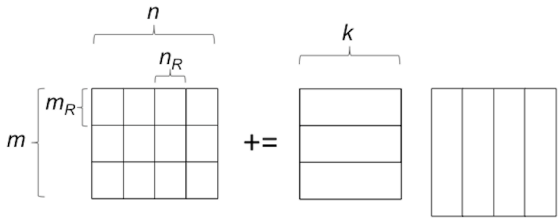
Notice that computation can only happen if data is stored in registers. A compiler will automatically transform code so that the intermediate steps that place certain data in registers is inserted. One can give a hint to the compiler that it would be good to keep certain data in registers as illustrated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

2.4 Your mission, if you choose to accept it

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 `include/bl_config.h` with those choices. This ensures that the test driver only tries problem sizes that are multiples of these block sizes, so you don't have to worry about "fringe".

2.4.1 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 [8]

> 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 submatrices), 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.

In the last section you already saw an example of blocking.

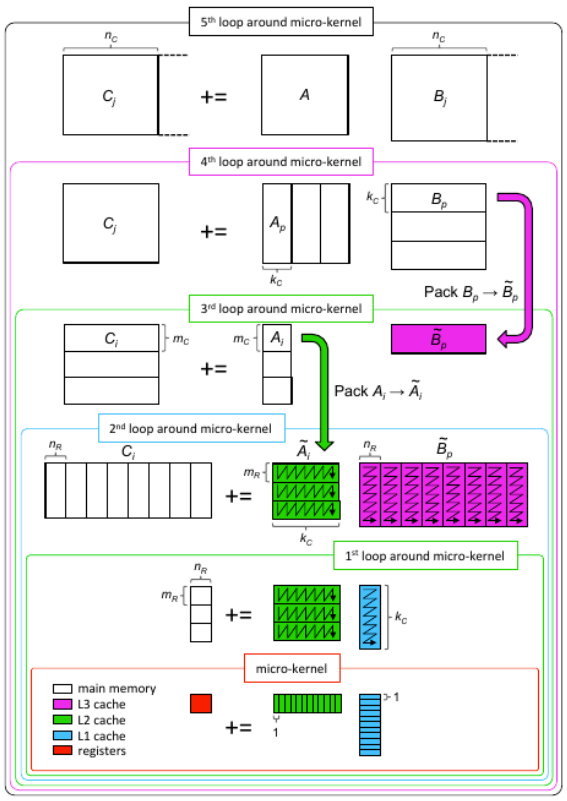
3.2 Blocked matrix-matrix multiplication

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

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



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

Figure 3: Left: The Goto algorithm for matrix-matrix multiplication as refactored in BLIS. Right: the same algorithm, but expressed as loops.

3.3 Your mission, if you choose to accept it

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 [5]

> 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.
> ☛ http://shpc.ices.utexas.edu/publications.html

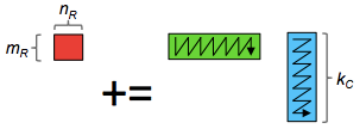
A further "refactoring" of this approach was more recently described in [14]

> 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.
> ☛ http://shpc.ices.utexas.edu/publications.html

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*. In this section, we briefly describe the highlights

of the approach. However, we strongly suggest the reader become familiar with the above two papers themselves.

Figure 3 (left) illustrates the way the Goto approach structures the blocking for three layers of cache (L1, L2, and L3). In the BLIS framework, the implementation is structured exactly this way so that only the micro-kernel at the bottom needs to be highly optimized and customized for a given architecture. In the original GotoBLAS implementation, now maintained as OpenBLAS [], the operation starting with the second loop loop around the micro-kernel is instead customized. In order to get the best performance, it helps is all data is accessed contiguously, which is why at some point prior to reaching the micro-kernel, data is packed in the order indicated by the arrows:



Now, notice that each column of the block of A in the above picture is multiplied by each element in the corresponding row of the block of B. (We call these blocks of A and B *micro-panels*.) This means that the *latency* to the L2 cache (the time required to bring in an element of the micro-panel of A from that cache) can be amortized over $2n_R$ flops. For this reason, we can organize the computation so that the micro-panel of A can reside in the L2 cache. Actually, we can do better: while a rank-1 update is happening with a column of the micro-panels of A and B, the next column of the micro-panel of A can be brought into registers so that computation masks the cost of that data movement. The fact that we want to keep the micro-panel of B in the L1 cache (because it will be reused for many micro-panels of A) limits the blocking parameter k_C.

With the insights, the rest of the picture hopefully becomes clear. The first loop around the microkernel works with a block of A, $\widetilde{A}_i$, that has been packed and resides in the L2 cache (by virtue of how the computation is ordered). This limits the blocking parameter m_C. That block of A multiplies a block of B, $\widetilde{B}_p$, that has been packed to reside in the L3 cache (if the processor has an L3 cache). Notice that the packing into $\widehat{A}_i$ is amortized over all computation with $\widetilde{B}_p$ and the packing into $\widehat{B}_p$ is amortized over computations with many blocks A_i. The outermost loop partitions B so that the block $\widetilde{B}_p$ fits in the L3 cache or, if a processor does not have an L3 cache, limits the amount of workspace for packing $\widehat{B}_p$ that is needed. This limits the blocking parameter n_C.

One may ask if the above described scheme is optimal. In [6] a theory is given that shows that under an idealized model the above is locally optimal (in the sense that assuming data is in a certain memory layer in the hierarchy, the proposed blocking at that level optimally amortizes the cost of data movement with the next memory layer). A theory that guides the choice of the various blocking parameters is given in [11].

4.2 Setup

Figure 4 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.

- `bl_dgemm_int_8x4.c` Is an `AVX` intrinsics micro-kernel implementation for Sandy Bridge/Ivy Bridge micro-architecture.
- `bl_dgemm_asm_8x4.c` Is an `AVX` assembly micro-kernel implementation for Sandy Bridge/Ivy Bridge micro-architecture.
- `bl_dgemm_asm_8x6.c` Is an `AVX2` assembly micro-kernel implementation for Haswell micro-architecture.
- `bl_dgemm_asm_12x4.c` Is an alternative `AVX2` assembly micro-kernel implementation for Haswell micro-architecture.

4.3 Advanced techniques

Describe vector instructions and how to access them through vector intrinsics.

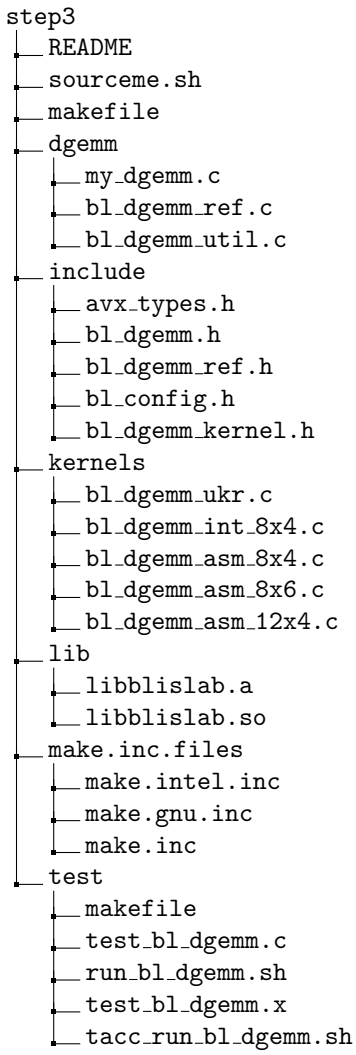


Figure 4: Structure of directory `step3`.

4.4 Your mission, if you choose to accept it

5 Step 4: Parallelizing with OpenMP

6 Conclusion

Conclusion.

Additional information

For additional information on FLAME visit
`http://www.cs.utexas.edu/users/flame/`.

Acknowledgments

We thank the other members of the FLAME team for their support. This research was partially sponsored by NSF grant ACI-1148125/1340293 and Intel through its funding of the Science of High-Performance Computing (SHPC) group as an Intel Parallel Computing Center.

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] R.C. Agarwal, F.G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5), Sept. 1994.

[2] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[3] 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.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[5] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12, May 2008. Article 12, 25 pages, Available from ☛ http://shpc.ices.utexas.edu/publications.html.

[6] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[7] Francisco D. Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, Timothy Wentz, , and Robert van de Geijn. Unleashing the high-performance and low-power of multi-core dsps for general-purpose hpc. In *SC12*, 2012.

[8] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[9] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[10] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, , and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *International Parallel and Distributed Processing Symposium 2014*, 2014.

[11] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. Analytical modeling is enough for high performance blis. *ACM Transactions on Mathematical Software*. in review. Available from ☛ http://shpc.ices.utexas.edu/publications.html.

[12] Robert van de Geijn and Kazushige Goto. *Encyclopedia of Parallel Computing, Part 2*, chapter Robert van de Geijn and Kazushige Goto, pages 157–164. Springer, 2011.

[13] Field G. Van Zee, Tyler Smith, Bryan Marker, Tze Meng Low, Robert A. van de Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, and Lee Killough. The blis framework: Experiments in portability. *ACM Transactions on Mathematical Software*. to appear.

[14] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating blas functionality (replicated computational results certified). *ACM Trans. Math. Soft.*, 41(3):14:1–14:33, June 2015. Available from ☛ http://shpc.ices.utexas.edu/publications.html.

[15] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[16] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.