Lab 1: Optimizing Matrix Multiplication

**Problem statement**

Your task in this assignment is to write an optimized[matrix multiplication](https://en.wikipedia.org/wiki/Matrix_multiplication) function for TACC Lonestar 5 supercomputer (<https://portal.tacc.utexas.edu/user-guides/lonestar5>).  We will give you a generic matrix multiplication code (also called matmul or dgemm), and it will be your job to tune our code to run efficiently on Lonestar 5’s processors.

Write an optimized *single-threaded* matrix multiply kernel.  This will run on only one core.

We consider a special case of matmul:

*C* := *C* + *A*\**B*

where *A*, *B*, and *C* are *n* x *n* matrices. This can be performed using 2*n^*3 floating point operations (*n^*3 adds, *n^*3 multiplies), as in the following pseudocode:

 for i = 1 to n

   for j = 1 to n

     for k = 1 to n

       C(i,j) = C(i,j) + A(i,k) \* B(k,j)

     end

   end

 end

**Instructions**

**Teams**

Note that you will work in assigned teams of two people for this assignment.

**Starter Code**

The tarball contains starter code for the serial matrix multiply

**dgemm-blocked.c**

A simple blocked implementation of matrix multiply.  It is your job to optimize the square\_dgemm() function in this file.

**dgemm-blas.c**

A wrapper which calls the vendor's optimized BLAS implementation of matrix multiply (here, MKL).

**dgemm-naive.c**

       For illustrative purposes, a naive implementation of matrix multiply using three nested loops.

**benchmark.c**

       A driver program that runs your code.  You will not modify this file, except perhaps to change the MAX\_SPEED constant if you wish to test on another computer (more about this below).

**Makefile**

       A simple makefile to build the executables.

**job-blas, job-blocked, job-naive**

***Example*** job scripts**\***to run the executables on Maverick compute nodes. For example, you can type "sbatch job-blas" to benchmark the BLAS version.

**PLEASE NOTE** : The current dgemm-blocked.c is a C code. But in our experience to optimize dgemm-blocked code using  [C++ vector class library](https://www.agner.org/optimize/#vectorclass) is the most pleasant. This library is available with C++ and not with C language. So, if you plan to use the **vector library** you will have to write **dgemm-blocked** as a CPP file instead of a C file.

**Running our Code**

The starter code should work out of the box.  To get started, we recommend you log in to Lonestar 5 and download the tarball.  This will look something like the following:

Command to login into Lonestar 5 from your terminal

//replace vkariha2 with your username

ssh vkariha2@ls5.tacc.utexas.edu 

Command to transfer hw1.tar.gz to lonestar 5

//replace filepath with the path of hw1.tar.gz on your local machine

//replace vkariha2 with your username//the following command will push hw1.tar.gz in your home directory in lonestar 5

scp filepath/hw1.tar.gz vkariha2@ls5.tacc.utexas.edu:.

login1.ls5(305)$ tar -xf hw1.tar.gz

login1.ls5(307)$ cd cosc6374\_hw1\_2019fall

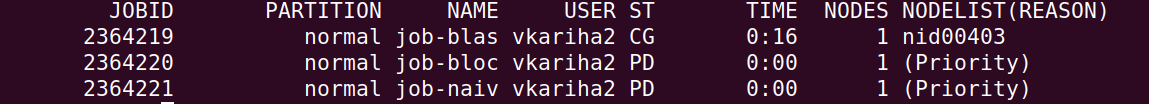
login1.ls5(310)$ make

login1.ls5(311)$ sbatch job-blas

login1.ls5(312)$ sbatch job-blocked

login1.ls5(313)$ sbatch job-naive

login1.ls5(314)$ squeue -u vkariha2



When our job is finished, we'll find new files in our directory containing the output of our program.  For example, we'll find the files job-blas.o9637621. The first file contains the[standard output](https://en.wikipedia.org/wiki/Standard_streams) and[standard error](https://en.wikipedia.org/wiki/Standard_streams).

**Interactive Session**

Use command idev -m [minutes] to get an interactive session with a compute node.

Interactive session is good for quick test runs and development.

**Our Harness**

The benchmark.c file generates matrices of a number of different sizes and benchmarks the performance.  It outputs the performance in[FLOPS](https://en.wikipedia.org/wiki/FLOPS) and in a percentage of theoretical peak attained.  Your job is to get your matrix multiply's performance as close to the theoretical peak as possible.

**Lonestar 5's Processors**

**Theoretical Peak**

Our benchmark harness reports numbers as a percentage of theoretical peak.  Here, we show you how we calculate the theoretical peak of Lonestar 5’s Haswell processors.  If you'd like to run the assignment on your own processor, you should follow this process to arrive at the theoretical peak of your own machine, and then replace the MAX\_SPEED constant in benchmark.c with the theoretical peak of your machine.  Be sure to change it back if you run your code on Lonestar 5 again.

**One Core**

One core has a clock rate of 2.6 GHz and can turbo to 3.5GHz, so it can issue 3.5 billion instructions per second.  Lonestar 5's processors also have a 256-bit *vector width (AVX2 vector extension)*, meaning each instruction can operate on 4 double data elements(64 bits) at a time.  Furthermore, the Lonestar 5 microarchitecture includes two *fused multiply-add* (FMA) instruction, which means 2 floating point operations can be performed in a single instruction.  So, the theoretical peak of Lonestar 5’s nodes is:

* 2.6 GHz \* 4-element vector \* 2 ops in an FMA \* 2 (FMA's) = 41.6 GFlops/s

**Optimizing Part One**

Now, it's time to optimize!  A few optimizations you might consider adding:

1. Perform blocking.  The dgemm-blocked.c already gets you started with this, although you'll need to tune block sizes.
2. Write a register-blocked kernel, either by writing an inner-level fixed-size matrix multiply and hoping (and maybe checking) that the compiler inlines it, writing AVX2 intrinsics, or even writing inline assembly instructions.
3. Add manual prefetching.

You may, of course, proceed however you wish.  We recommend you look through the lecture notes as reference material to guide your optimization process, as well as the references at the bottom of this write-up.

**Available Compilers**

You will probably want to try your code with different compilers to see if you can get a performance boost for free.  The default compiler is GCC 6.3.0  You can also use Intel compilers but you have to figure out how to compile and link with MKL and modify the Makefile accordingly.

**Grading**

We will grade your assignment by reviewing your assignment write-up, looking at the optimization methods you attempted, and benchmarking your code's performance.

**Benchmarking**

To benchmark your code, we will compile it with the Makefile provided by you, run the binaries, and take the performance result.

**Submission Details :**

For a group there will be a single submission. And you will be able to upload it via Blackboard.

Your submission should be a gzipped tar archive, formatted (for Team 4) like: team04\_hw1.tgz. It should contain:

* dgemm-blocked.c, a C-language source file containing your implementation of the routine:   void square\_dgemm(int, double\*, double\*, double\*);
* described in pseudocode above. We provide an example dgemm-blocked.c, below.
* Makefile, only if you modified it. If you modified it, make sure it still correctly builds the provided benchmark.c, which we will use to grade your submission.
* (e.g. for Team 4) team04\_hw1.pdf, your write-up.
* **Please do use these formats and naming conventions.**  Not following these instructions leads to more busy work for the TA's, which makes the TA's sad...
* [This link](https://kb.iu.edu/d/acfi) tells you how to use tar to make a .tgz file.
* Submit your .tgz to Blackboard.
* Your write-up should contain:
* the names of the people in your group (and each member's contribution),
* the optimizations used or attempted,
* the results of those optimizations,
* the reason for any odd behavior (e.g., dips) in performance, and
* how the performance changed when running your optimized code on a *different* machine.
* For the last requirement, you may run your implementation onyour laptop, cellphone, toaster, etc.
* Please carefully read the notes for implementation details. Stay tuned toPiazza for updates and clarifications, as well as discussion.
* If you are new to optimizing numerical codes, we recommend reading the papers in the references section.

**Notes:**

Your grade will mostly depend on two factors:

* performance sustained by your codes on the Lonestar5 supercomputer,
* explanations of the performance features you observed (including what didn't work)
* There are other formulations of matmul (e.g.,[Strassen](https://en.wikipedia.org/wiki/Strassen_algorithm)) that are mathematically equivalent, but perform asymptotically fewer computations - we will not grade submissions that do fewer computations than the two *n*cubed algorithm. This is actually an optional part of HW1.
* Your code must use[double-precision](https://en.wikipedia.org/wiki/Double-precision_floating-point_format) to represent real numbers. A common reference for double-precision matrix multiplication is the[dgemm](http://www.netlib.org/blas/dgemm.f) (double-precision general matrix-matrix multiply) routine in the level-3[BLAS](http://www.netlib.org/blas/). We will compare your implementation with the tuned dgemm implementation in the vendor-provided BLAS library (Intel MKL)
* Besides compiler intrinsic functions and built-ins, your code (dgemm-blocked.c) must only call into the C standard library, and the VCL template library.
* You may not use compiler flags that automatically detect dgemm kernels and replace them with BLAS calls, i.e. Intel's[-matmul](https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-matmul) flag.
* You should try to use your compiler's automatic vectorizer before manually vectorizing.
* GNU C provides[many](https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html) extensions, which include intrinsics for vector (SIMD) instructions and data alignment. (Other compilers may have different interfaces.)
* Ideally your compiler injects the appropriate intrinsics into your code automatically (eg, automatic vectorization and/or automatic data alignment).[gcc's auto-vectorizer](https://gcc.gnu.org/projects/tree-ssa/vectorization.html) reports diagnostics that may help you identify if manual vectorization is required.
* To manually vectorize, you must add compiler intrinsics to your code.
* Consult your compiler's documentation.
* You may assume that A and B do not alias C; however, A and B may alias each other. It is semantically correct to qualify C (the last argument to square\_dgemm) with the C99 restrict keyword. There is a lot online about restrict and pointer-aliasing -[this](https://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html) is a good article to start with, along with the[Wikipedia article](https://en.wikipedia.org/wiki/Restrict) on the restrict keyword.
* The matrices are all stored in [column-major order](https://en.wikipedia.org/wiki/Row-_and_column-major_order), i.e. *Ci,j* == C(i,j) == C[(i-1)+(j-1)\*n], for i=1:n, where n is the number of rows in C. Note that we use 1-based indexing when using mathematical symbols and MATLAB index notation (parentheses), and 0-based indexing when using C index notation (square brackets).
* We will check correctness by the following componentwise error bound:  |square\_dgemm(n,A,B,0) - A\*B| < eps\*n\*|A|\*|B|.
* where eps := 2-52 = 2.2 \* 10-16 is the[machine epsilon](https://en.wikipedia.org/wiki/Machine_epsilon).

**Optional Parts**

These parts are not graded. You should be satisfied with your square\_dgemm results and write-up before beginning an optional part.

* Implement Strassen matmul. Consider switching over to the three-nested-loops algorithm when the recursive subproblems are small enough.
* Support the dgemm interface (ie, rectangular matrices, transposing, scalar multiples).
* Try float (single-precision).
* Try complex numbers (single- and double-precision) - note that complex numbers are part of C99 and[supported in gcc](https://gcc.gnu.org/onlinedocs/gcc/Complex.html).[This forum thread](https://stackoverflow.com/questions/3211346/complex-mul-and-div-using-sse-instructions) gives advice on vectorizing complex multiplication with the conventional approach - but note that there are[other algorithms](https://en.wikipedia.org/wiki/Multiplication_algorithm#Gauss.27s_complex_multiplication_algorithm) for this operation.
* Optimize your matmul for the case when the inputs are symmetric. Consider[conventional](http://www.netlib.org/lapack/lug/node122.html) and[packed](http://www.netlib.org/lapack/lug/node123.html) symmetric storage.

**Documentation**

* TACC and Maverick documentation
* [GCC](https://gcc.gnu.org/onlinedocs/) documentation - we are using GCC 6.3.0.
* [Intel Intrinsic Guide](https://software.intel.com/en-us/isa-extensions) (scroll down). This Java program has a complete reference of all SIMD intrinsics on Intel architectures. Note thatHaswell support AVX2, which includes fused multiply-add instructions.
* Crunching Numbers with AVX and AVX2 [https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX](https://www.google.com/url?q=https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX&sa=D&ust=1535481742219000)

You are also welcome to learn from the source code of state-of-art BLAS implementations such as[GotoBLAS](https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2) and[ATLAS](http://math-atlas.sourceforge.net/). However, you should not reuse those codes in your submission.

**For the brave**

A resources for achieving >50% peak:

[Combining Assembly Code with C Program](https://elearning.uh.edu/bbcswebdav/pid-6137521-dt-content-rid-43941535_1/xid-43941535_1)

[An example High Performance (>90% peak) implementation of a micro-kernel 8x6 dgemm](https://github.com/flame/blis/blob/master/kernels/haswell/3/bli_gemm_haswell_asm_d8x6.c)

**References**

* Goto, K., and van de Geijn, R. A. 2008. Anatomy of High-Performance Matrix Multiplication, *ACM Transactions on Mathematical Software 34*, 3, Article 12.
* (Note: explains the design decisions for the GotoBLAS dgemm implementation, which also apply to your code.)
* Chellappa, S., Franchetti, F., and PÃƒÂ¼schel, M. 2008.[How To Write Fast Numerical Code: A Small Introduction](http://spiral.ece.cmu.edu:8080/pub-spiral/abstract.jsp?id=100), *Lecture Notes in Computer Science 5235*, 196-259.
* (Note: how to write C code for modern compilers and memory hierarchies, so that it runs fast. Recommended reading, especially for newcomers to code optimization.)
* Bilmes, *et al.*[The PHiPAC (Portable High Performance ANSI C) Page for BLAS3 Compatible Fast Matrix Matrix Multiply](http://www1.icsi.berkeley.edu/~bilmes/phipac/).
* (Note: PHiPAC is a code-generating autotuner for matmul that started as a submission for this HW in a previous semester of CS267. Also see[ATLAS](http://math-atlas.sourceforge.net/); both are good examples if you are considering code generation strategies.)
* Lam, M. S., Rothberg, E. E, and Wolf, M. E. 1991. The Cache Performance and Optimization of Blocked Algorithms, *ASPLOS'91*, 63-74.
* (Note: clearly explains cache blocking, supported by with performance models.)

**Update**

If you plan to use C++ and the Vector library, I have uploaded**hw1\_C++.tar.gz**  which contains slight modification. Following are the modification.

1. The default version of gcc on login node is 6.3.0. The Vector class library doesnot compile with gcc 6.3.0.  So, when you are on the login node where you land by default when you login to lonestar 5, run the following command on the command prompt.

module load gcc

module load mkl

The above command will load gcc 7.3.0 which is required for successfully compiling the Vector library. You need to run above commands on login nodes, before you run your makefile.

2. In the job files, I have added instruction for loading module gcc 7.3.0 on the compute node. So as to be able to run the code build using Vector library successfully on the compute node.

3. It contains the folder version2-master which contains the Vector class library file required. If you plan to use Vector library in your code, the Makefile already contains instruction to look for the Vector header file in version2-master