Applications of Parallel Computers

Problem Description

CS267 Assignment 1: Optimize Matrix Multiplication

Assignment 1: Optimize Matrix Multiplication

Due Date: To be determined by your instructor.

Problem statement

Your task is to optimize matrix multiplication (matmul) code to run fast on a single processor core of XSEDE's Bridges cluster.

We consider a special case of matmul:

C := *C* + *A***B*

where *A*, *B*, and *C* are $n \times n$ matrices. This can be performed using $2n^3$ 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
```

Notes

- Please carefully read the this section for implementation details. Stay tuned to Moodle 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.
- There are other formulations of matmul (eg, Strassen) that are mathematically equivalent, but perform asymptotically fewer computations - we will not grade submissions that do fewer computations than the $2n^3$ algorithm. Once you have finished and are happy with your square_dgemm implementation you should consider this and other optional improvements for further coding practice but they will not be graded for HW1.
- Your code must use double-precision to represent real numbers. A common reference implementation for double-precision matrix multiplication is the `dgemm` (double-precision general matrix-matrix multiply) routine in the level-3 BLAS. We will compare your implementation with the tuned `dgemm` implementation available - on Bridges , we will compare with the Intel MKL implementation of `dgemm`. Note that `dgemm` has a more general interface than `square_dgemm` - an optional part of HW1 encourages you to explore this richer tuning space.
- You may use any compiler available. We recommend starting with the GNU C compiler (`gcc`). If you use a compiler other than `gcc`, you will have to change the provided `Makefile`, which uses `gcc`-specific flags. Note that the default compilers, every time you open a new terminal, are PGI - you will have to type "`module unload pgi`" or "`module purge`" and then "`module load gnu`"to switch to, eg, GNU compilers. You can type "`module list`" to see which compiler wrapper you have loaded.
- You may use C99 features when available. The provided `benchmark.c` uses C99 features, so you *must* compile with C99 support - for `gcc`, this means using the flag `-std=gnu99` (see the `Makefile`). Here is the status of C99 functionality in `gcc` - note that C90 (ANSI C) is fully implemented.
- Besides compiler intrinsic functions and built-ins, your code (`dgemm-`

`blocked.c`) must only call into the C standard library.

- You may not use compiler flags that automatically detect `dgemm` kernels and replace them with BLAS calls, i.e. Intel's `-matmul` flag.
- You should try to use your compiler's automatic vectorizer before manually vectorizing.
    - GNU C provides many 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 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 is a good article to start with.
- The matrices are all stored in column-major order, i.e. $C_{i,j}$ == `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 $C_{i,j}$ and MATLAB index notation `C(i,j)` , and 0-based indexing when using C index notation `C[(i-1)+(j-1)*n]`.
- 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.
- One possible optimization to consider for the multiple tuning parameters in an optimized Matrix Multiplication code is autotuning in order to find the optimal/best available value. Libraries like OSKI and ATLAS have shown that achieving the best performance sometimes can be done most efficiently by automatic searching over the parameter space. Some papers on this topic can be found on the **Be**rkeley **B**enchmarking and **Op**timization (BeBOP) page
- The target processor on the Bridges compute nodes is a Xeon Intel 14-Core 64-bit E5-processor running at 2.3GHz and supporting 8 floating-point operations per clock period with a peak performance of 21.6 GFLOPS/core.

Instructions

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*);
- 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.

- This link tells you how to use tar to make a .tgz file.
- Your write-up should contain:

1. the names of the people in your group (and each member's contribution),
2. the optimizations used or attempted,
3. the results of those optimizations,
4. the reason for any odd behavior (e.g., dips) in performance, and
5. how the performance changed when running your optimized code on a different machine.

Grading

- Your grade will depend on the performance sustained by your codes on Bridges as a percentage of peak:
  - If your sustained performance is between 0% and 50% you will receive a score between 0 and 75 proportional to your sustained performance (Ex:25% gives a score of 37.5)
  - If your sustained performance is between 50% and 80% you will receive a score between 75 and 100 proportional to your sustained performance (Ex:56% gives a score of 80)
  - If your sustained performance is above 80% you will receive 100
- Your submission must be a single C file that starts with the name dgemm
- Any compiler information, flags and options must be copied to the starting comments of the dgemm submitted file; The default options available in the Makefile will already be present in dgemm_blocked.c
- Your submission must pass the error bound test and cannot call BLAS for dgemm; any submissions that fail these tests this will receive a grade of 0

Optional:

These parts are not graded. You should be satisfied with your `square_dgemm` results 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). This means you can use 8-way SIMD parallelism on Bridges.
- Try complex numbers (single- and double-precision) - note that complex numbers are part of C99 and supported in `gcc`. This forum thread gives advice on vectorizing complex multiplication with the conventional approach - but note that there are other algorithms for this operation.
- Optimize your matmul for the case when the inputs are symmetric. Consider conventional and packed symmetric storage.
- Run the optimized code on one of the other supercomputers available and check relative performance and what optimizations need to change or become more relevant

Source files

The starting files with their descriptions can be found in the Starting Code folder

To download all the files directly to Bridges you can use the following command:

```
wget --no-check-certificate https://people.eecs.berkeley.edu/~aditya/cs267_2018/hw1/xsede_hw1_2018.tar
```

Login, Compilation and Job submission

The easiest way to access the machines is to login directly with your own ssh client to **login.xsede.org** and from there **gsissh** into the correct machine. More information on is available here on the single login system. For this assignment we will be using the Bridges supercomputer.

Another easy way to login to XSEDE resources is via the Accounts tab in XSEDE User Portal. To reach this page login to XSEDE User Portal, navigate to MY XSEDE tab and from there select the Accounts page. All machines you have access to will have a **login** option next to them that will launch OpenSSH via a Java Applet.

Please be aware that most XSEDE resources are not available via direct ssh and all requests must go through the XSEDE single login system (While some machines, Bridges included, do allow direct login they require further separate account creation)

To unarchive the files from the tar archive use the following command

```
tar -xf cs267_hw1_2018.tgz .
```

and then you can simply type **make** to compile the code.

To submit jobs on the Bridges system you will use the SLURM interface for example:

```
sbatch job-blocked
```

To check the status of running jobs you can use the following squeue command where again $USER should be replaced with your username

```
squeue -u $USER
```

```
For more details on sbatch commands please see Bridges' documentation page
```

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, *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.
- (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; 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.)
- The OpenMP Cheat Sheet
- Lawrence Livermore's OpenMP Tutorial
- Hints for HW1 and SIMD example pptx and pdf

Documentation:

- Bridges's computing environment documentation
- AMD architecture documentation.

You are also welcome to learn from the source code of state-of-art BLAS implementations such as GotoBLAS and ATLAS. However, you should not reuse those codes in your submission.

Last modified: Friday, 8 February 2019, 12:24 PM

◀ Quiz 3 Jump to... Starting code ▶