CS420: Parallel Programming - Fall 2017
MP3
Due Date: December 18, 5:00PM
Submission Method: SVN

December 4, 2017

In this assignment, you will utilize a multi-core CPU using OpenMP and a GPU using OpenACC to implement a matrix-multiply routine for multiplying a band matrix by a dense matrix. You will also investigate the performance impact of varying the band and matrix size. Lastly, you will gain experience using one of the largest supercomputers in the world.

For this assignment, you will utilize the Blue Waters supercomputer. You will need a separate account from us to log in to Blue Waters. You *must* get this account from us in person (e.g. by coming to class); we *cannot* distribute the login information electronically due to Blue Waters security policies.

You are required to use the stubbed code provided to implement your algorithms. Below we will review band matrices, and provide information on OpenACC and logging in to Blue Waters.

Please read the section on Blue Waters carefully, as the system functions somewhat differently than the Campus Cluster.

WARNING: Do not wait until the last minute to do this assignment. Blue Waters is a busy system (thousands of jobs per day) and it may take time for your experiments to run.

Introduction

Band matrices

Recall from your midterm that a band matrix is a square matrix where all non-zero elements are in a band around the diagonal. A band matrix has *bandwidth* B if the band has width $2B + 1$. Below we illustrate a 7×7 band matrix with bandwidth 2:

$$\begin{pmatrix} 2 & 3 & 1 & 0 & 0 & 0 & 0 \\ 1 & 3 & 5 & 2 & 0 & 0 & 0 \\ 2 & 1 & 4 & 6 & 3 & 0 & 0 \\ 0 & 3 & 7 & 1 & 3 & 2 & 0 \\ 0 & 0 & 4 & 5 & 1 & 2 & 6 \\ 0 & 0 & 0 & 2 & 3 & 7 & 8 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \end{pmatrix}$$

You can exploit this structure to do fewer operations in a matrix multiply involving band matrices. We have provided a serial algorithm to multiply a band matrix with a dense matrix.

Using Blue Waters

Blue Waters is a large, petascale supercomputer (>22,000 nodes, ∼13 PFLOPS) located at UIUC, with both CPUs and GPUs. You should have received an account from us (see above if not), which contains a username and a password. To log in, you should use SSH similarly to how you logged in to the Campus Cluster, but using the username and password you received. The hostname to connect to is `bwbay.ncsa.illinois.edu`. For example, using a terminal to SSH: `ssh username@bwbay.ncsa.illinois.edu`.

The first time you log in, you will need to provide some contact information and agree to their terms of service. For full details on using Blue Waters (and as a first place to look if you have questions not answered here), please see their user guide: `https://bluewaters.ncsa.illinois.edu/user-guide`.

When you log in, the Cray programming environment and compilers will be loaded automatically. We will use these for this assignment. The Cray compiler is similar to the Intel or GCC compilers you used for prior MPs, but it takes slightly different arguments. After logging in, you should run `module load craype-accel-nvidia35 cudatoolkit` to load the Nvidia accelerator module. If you do not do this before compiling, your code will not be offloaded to the GPUs. We have provided a `Makefile` that you should use to build your code.

We have also provided a `run.sub` script to run your code and some basic experiments. You may modify it if you wish to vary the experiments run. To submit the run script, run `qsub run.sub`.

You cannot `scp` files to Blue Waters. Therefore, you can also obtain the MP through Subversion:

```
svn co --username netid
  https://subversion.engr.illinois.edu/svn/fa17-cs420/_shared/mp3 mp3
```

will check out a copy of MP3 into the `mp3` directory.

Some final notes on Blue Waters:

- Do not run big problems on the login nodes; the system will detect and kill your program if it runs too long.
- If you change the `run.sub` script, be sure you still run the application with the `aprun` command. Unlike the Campus Cluster, batch scripts are run on management nodes, not compute nodes.
- If you have any problems or questions not covered in the user guide, please post on Piazza.

OpenACC

This assignment will use OpenACC to offload onto GPUs, instead of the OpenMP directives you learned in class. (Blue Waters has very limited support for OpenMP offloading onto GPUs.) OpenACC is very similar to OpenMP in both functionality and syntax. For full details on OpenACC, including the specification and additional tutorials/guides, you can browse the OpenACC website: `https://www.openacc.org/`. Below we give a brief overview of some features we think are relevant.

Directives are prefixed with `#pragma acc` instead of `#pragma omp`. To generate a kernel that runs on an accelerator from a block of code, use the `#pragma acc kernels` directive. This is similar to a parallel loop in an `#pragma omp target` region.

To move data to or from the accelerator, you can use the `#pragma acc data` directive with either `copy`, `copyin`, or `copyout`. If the size of the region to copy is known at compile time, you can write `#pragma acc data copy(var1, var2,...)` to copy `var1` (etc.) to the accelerator. If the size is known only at runtime, you can specify the region to copy with `#pragma acc data copy(var[start:end], ...)`.

The `copy` directive copies the data to the accelerator when the block is encounted, and copies the (potentially modified) data back to the host once the block is finished. The `copyin` and `copyout` directives are similar, except they only copy data to the accelerator or copy it from the accelerator, respectively.

As a simple example, the following code will execute a simple loop on an accelerator. The data in `A` will be copied to the accelerator and then freed after the loop completes. The data in `B` will be copied to the accelerator, and then copied back to the host after the loop completes.

```
int size = 1000;
double* A = new double[size];
double* B = new double[size]
// Initialize A and B somehow...
double alpha = 0.5;
#pragma acc data copyin(A[0:size]) copy(B[0:size])
#pragma acc kernels
for (int i = 0; i < size; ++i) {
  B[i] += alpha * A[i];
}
```

Part A: Familiarize yourself with the code

Log in to Blue Waters and transfer the MP to it (see instructions above). Ensure you have loaded the `craype-accel-nvidia35` and `cudatoolkit` modules. Compile the code.

Examine the source code and binary and answer the following in your writeup. You should have a basic knowledge of the code before proceeding to the next parts.

- What does the application do? What are the arguments to the binary?
- What configurations does the provided `run.sub` script run by default? What kind of node does it run on? (You may need to check the Blue Waters user guide for this.)
- How is a matrix represented in the application?
- How is the memory for a dense and a band matrix laid out?
- Describe the basic algorithm being used to multiply the matrices.

For the subsequent parts, you may want to add the `-h msgs` argument to the compiler in order to get output on the optimizations being performed.

Part B: Multi-threaded implementation

Use OpenMP to implement a multi-threaded algorithm to multiply a banded matrix by a dense matrix. Your implementation should be done in the `omp_mm.cpp` file. Ensure you have a correct implementation. You may test your code by submitting the `run.sub` script and ensure you are getting a good speedup, but note you will do a comprehensive performance analysis in part D.

In your writeup, please provide a brief overview of your implementation choices.

Part C: GPU implementation

Use OpenACC to implement a GPU algorithm to multiply a banded matrix by a dense matrix. Your implementation should be done in the `gpu_mm.cpp` file. Ensure you have a correct implementation. You may test your code by submitting the `run.sub` script and ensure you are getting a good speedup, but note you will do a comprehensive performance analysis in part D.

Hint: You need to think about how the matrices are laid out in host memory to correctly transfer them to the GPU memory. The OpenACC runtime cannot automatically transfer the matrix structure to GPU memory, so one approach to solving this is to treat the matrix data as a 1D array and do the indexing manually.

For this implementation, you should try to avoid unnecessary data movement and consider what a good iteration order is for the GPU. In your writeup, please provide a brief overview of your implementation choices and how you optimized for the GPU.

Part D: Evaluation

In this section, you will evaluate the the performance of your two implementations compared to the serial algorithm and to each other. Once your implementations are correct, submit the `run.sub` script. This runs several application configurations by default; you may wish to add additional ones to better understand the performance. You should, at a minimum, run the configurations provided. If you do not see speedups from your implementations, you should optimize them further.

In your writeup, report the performance of the serial, multi-threaded, and GPU algorithms for the different configurations you ran, and the speedup of your multi-threaded and GPU code relative to the serial code. Then discuss the following (you may need to do additional experiments):

- When is the multi-threaded implementation fastest? When is the GPU implementation fastest?
- What are the overheads involved in your multi-threaded implementation? In your GPU implementation?
- Explain the speedups you observe in your GPU implementation relative to your OpenMP implementation.

Submission instructions

- Everything should be submitted under the `mp3` directory in your Subversion folder.
- Submit your modified `omp_mm.cpp` and `gpu_mm.cpp` files.

- Submit your writeup in a file named `NETID_mp3` (replacing `NETID` with your NetID), in either PDF (.pdf) or plain text (.txt) format at your convenience. Be sure you discuss everything asked above.