# Getting Started

*This handout introduces the environment and tools you will be using to complete your project assignments. You will experiment with a simple program for multiplying rectangular matrices. You should use this assignment to familiarize yourself with the tools you will be using throughout the course. You should complete this assignment by Tuesday, September 14, but you need not hand anything in.*

*NOTE: While you may have tools that you prefer for developing and debugging C programs, we strongly recommend that you build, test, and debug your program on the cloud machines as instructed. Later projects will rely on specific tools that may not work on your personal machines.*

## 1 Get a CSAIL account

If you already have a CSAIL account, then you may skip this step.

You need to sign up for a CSAIL account before you can do anything else. Do so by noon on Friday, September 10, or else you may not be able to do any work over the weekend. The URL to create an account is

```
http://inquir.csail.mit.edu/signup
```

Choose the "Computer Architecture" group with "Saman Amarasinghe" as your sponsor. It will probably take a few hours before you can log in (since Prof. Amarasinghe must approve your account-creation request).

## 2 Course machines

You must connect to the `cloud`*N*`.csail.mit.edu` machines, for $N = 4, 5, \ldots, 15$, to work on and run code. These machines have working compilers, as well as installations of VTune and the PNQ client. In order to connect to these machines, you will need to use ssh. If you're on a Mac OS X or Linux machine, this is as simple as opening a terminal and typing

`ssh` *username*`@cloud`*N*`.csail.mit.edu`

which is exactly the same way you remotely connect to Athena*. If you're using Windows, however, you'll need to install some software. The free, open-source, and widely used PuTTY is a good choice.

## 3 Running the setup script

Your account will need to be configured so that you can use VTune and PNQ. The necessary magic incantations can be performed by a setup script. Use ssh to connect to one of the cloud machines, and run the following command:

```
/afs/csail.mit.edu/proj/courses/6.172/scripts/student-setup
```

You must log out and log back in for the setup to be complete. Everything should then work except for PNQ, which will generate error messages if you try to submit jobs. Give the staff a few hours to register you for PNQ, and you should be off and running.

*Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

# 4 Version control

We shall use the git distributed version control system. The baseline code for this assignment is in the repository

`/afs/csail.mit.edu/proj/courses/6.172/student-repos/project0/`*username*

If you haven't used git before, please be aware that it's a little bit different than subversion (svn), which you may have used in 6.005.

To make a clone — a local copy of the repository for your work — on a machine without AFS, type:

```
git clone ssh://username@login.csail.mit.edu/afs/csail.mit.edu/proj/\
courses/6.172/student-repos/project0/username project0
```

To make a clone on a CSAIL machine or other machine with AFS, type:

```
git clone /afs/csail.mit.edu/proj/\
courses/6.172/student-repos/project0/username project0
```

Edit the code in the project, and when you're done, commit and push your changes back to the repository:

```
git commit -a -m 'Your commit message'
git push
```

For more advanced usage of git, please refer to your favorite search engine.

# 5 Building and running your code

You can build the code by going to the `project0/matrix_multiply` directory and typing `make`. You can then run it by typing `./matrix_multiply`. The program should print out some matrices, and then crash with a segmentation fault.

You may be able to run this code on whatever computer you happen to be working on. We don't promise that it will work on arbitrary platforms, however, and you will inevitably need to see how it performs on the dedicated hardware we'll be evaluating it on.

# 6 Using assertions

The assert package is a useful tool for catching bugs before your program goes off into the weeds. If you look at `matrix_multiply.c`, you should see some assertions in `matrix_multiply_run` routine that check that the matrices have compatible dimensions. Since the extra checks performed by assertions can be expensive, they are disabled for optimized builds, which are the default in our Makefile. If you rebuild the program by typing `make DEBUG=1` and rerun your program, you should now see an assertion failure with a line number for the failing assertion.

You should consider sprinkling assertions throughout your code to check important invariants in your program, since they will make your life easier when debugging. In particular, most nontrivial loops and recursive functions should have an assertion of the loop or recursion invariant.

# 7 Using a debugger

While debugging your program, if you encounter a segmentation fault, bus error, or assertion failure, or if you just want to set a breakpoint, you can use the debugging tool `gdb`. First, build a "debug" version of the code by typing:

```
make DEBUG=1
```

Once you have created a debug build, you can start a debugging session in gdb:

```
gdb --args ./matrix_multiply
```

This command should give you a `(gdb)` prompt, at which you should type run:

```
(gdb) run
```

Your program will crash, giving you back a prompt, where you can type `backtrace` or `bt` to get a stack trace:

```
...
Running matrix_multiply_run()...
matrix_multiply: matrix_multiply.c:94: matrix_multiply_run:
    Assertion 'A->cols == B->cols' failed.
[New Thread 0x7f1ad8b606e0 (LWP 10495)]

Program received signal SIGABRT, Aborted.
[Switching to Thread 0x7f1ad8b606e0 (LWP 10495)]
0x00007f1ad8429ed5 in raise () from /lib/libc.so.6
(gdb) bt
#0  0x00007f1ad8429ed5 in raise () from /lib/libc.so.6
#1  0x00007f1ad842b3f3 in abort () from /lib/libc.so.6
#2  0x00007f1ad8422dc9 in __assert_fail () from /lib/libc.so.6
#3  0x0000000000400ed4 in matrix_multiply_run (A=0xb53010, B=0xb530e0, out=0xb531b0)
      at matrix_multiply.c:94
#4  0x0000000000400ab1 in main (argc=1, argv=0x7fffa7194b88) at testbed.c:91
```

To get more information, we can walk up the stack and print out some values. Type `up` three times, and then type `p A->cols` and `p B->rows`. You should see the values 5 and 4, which indicates that the problem is that we are multiplying matrices of incompatible dimensions.

Fix `testbed.c`, which creates the matrices, rebuild your program, and verify that it now works.

# 8 Compiler optimizations

To get an idea of the extent to which compiler optimizations can affect the performance of your program, increase the size of the matrices to 1000 by 1000. Rebuild your program in "debug" mode and run it. Rebuild it again with optimizations (just type `make`) and run it. Both versions should print timing information, and you should verify that the optimized version is faster.

## 9 Performance enhancements

Now let's try one of the techniques from the first lecture. Right now, the inner loop produces a sequential access pattern on `A` and skips through memory on `B`.

Let's rearrange the loops to produce a better access pattern. First, you should run the program as is with optimizations to get a performance measurement. Next, swap the `j` and `k` loop, so that the inner loop strides sequentially through the rows of the `C` and `B` matrices. Rerun the program, and verify that you have produced a speedup.

## 10 Using PNQ

You share the `cloud`*N*`.csail.mit.edu` machines on which you are developing with your fellow students. Directly running and timing the execution on these machines may result in measurement errors due to interference with the jobs of others. To get an accurate timing measure on a dedicated machine, you can use the PNQ utility.

*Note:* You will not be able to use PNQ until we have added you and your automatically generated password to the list of authorized users.

Before you can submit your job to PNQ, you must give PNQ's servers read access to your project directory. To do this, issue the command

```
fs sa . username.batch rlidwka
```

To submit a job to PNQ, just place `pnqsub` in front of the command you usually issue. Try submitting your matrix-multiply implementation with:

```
pnqsub ./matrix_multiply
```

If other jobs are in front of you, PNQ will notify you that you are queued. After your job runs and completes, its output will be displayed, for example, as below:

```
$ pnqsub ./matrix_multiply
Command: ./matrix_multiply
Your job has been enqueued as job #2. There are 0 jobs ahead of yours.
Thu Sep 9 13:15:30 EDT 2010     Job #2 - State: Running
Thu Sep 9 13:15:31 EDT 2010     Job #2 - State: Running
Thu Sep 9 13:15:33 EDT 2010     Job #2 - State: Running
Thu Sep 9 13:15:37 EDT 2010     Job #2 - State: Running
Thu Sep 9 13:15:45 EDT 2010     Job #2 - State: Completed

2.stdout:
Elapsed execution time: 8.411977 sec

2.stderr:
Setup
Running matrix_multiply_run()...
```

# 6.172 Performance Engineering of Software Systems

Fall 2010