Flamingo Auto-Tuning
∽ User's Guide ∽

Ben Spencer
ben@mistymountain.co.uk

Last updated: 21st July 2012

Introduction

This guide describes the features and use of my automatic parameter tuning system. The system can be used to find optimal parameter values in a parameterised program. It has been designed to be very general-purpose and makes few assumptions about how it should be used.

If you have any comments or questions about the tuner or this guide then please feel free to get in touch.

Contents

Background

The auto-tuner is used to find the optimal settings of program parameters. These would often be constants within the program which are used to control some aspect of its operation. If it is unclear exactly which combination of settings to use in a particular situation, the tuner can find them.

If some parameters are independent of each other, this independence can be exploited to speed up the tuning process. Usually, we want to optimise the running time of a program, so this is the default. However, it is possible to provide a custom 'figure of merit' which is used to rank the different tests.

Tutorial

This guide will go through the features of the tuner in detail, and can be used as a reference. A tutorial is available, which should probably be read first. It will lead you through the process of preparing and tuning an example program, explaining each step. This tutorial is `doc/tutorial.pdf`, distributed with the tuner.

The Configuration File

The tuner uses a configuration file to determine what to optimise, and how tests can be run. This file is used to set up and run the optimisation.

The file must contain five sections, each beginning with a `[section_name]` header. Within each section, options are set using the syntax `opt = value`. Lines that begin with `#` or `;` are treated as comments and ignored.

A template configuration file (`examples/template.conf`) is provided, containing some explanation of each option. The example programs also each come with a sample configuration file for tuning them.

All paths and commands used must be relative to the configuration file.

The `[variables]` Section

This section defines the program parameters which are to be optimised, and any independence between them. There is only a single option:

`variables` (Required)
: This option gives a list of the program parameters which should be optimised. There are two possible formats:

  - A simple comma separated list of variable names, for example:
    `variables = FOO, BAR, BAZ`
  - A list describing the independence between variables, as described in the *Variable Independence* section. For example:
    `variables = {{FOO}, {BAR, BAZ}}`

The `[values]` Section

This section gives the possible values that each variable above can take. They are specified as a comma separated list for each variable.

Each variable from the `[variables]` section must have an entry here. Any entries here which are not mentioned in the variable list will be ignored.

All values are interpreted as text strings, which are used without conversion, for example as compiler flags or the program's arguments.

The `[testing]` Section

This section defines how to compile, if needed, and run the tests. Tests can also be removed once their testing is complete, if needed.

All the options in this section specify commands which will be executed by the tuner. The value of any variable from the `[variables]` section can be substituted into the command which is run, using the syntax `%VAR_NAME%`.

For example, if there is a variable *FOO* being optimised, then the `test` command might be: `./myTestProgram -op %FOO%`. If for a particular test the variable *FOO* takes the value 3, then to get a score for that test, the command `./myTestProgram -op 3` is executed and timed.

There is also one special substitution: `%%ID%%`. This provides a unique id for each test (which is a counter increasing from 1).

`compiler` (Optional)
: If specified, this command is executed before testing begins. It can be used to compile a test if a parameter is benig changed at compile time (for example a compiler flag or a `#define` statement being overridden by the compiler).

`test` (Required)
: To perform a test, this command will be executed and timed by the tuning system. The running time of the command is taken as the score for each test. If a custom figure-of-merit is being used (chosen with the `optimal` option below), then the running time is not measured and instead the score is read from the final line of output from this command.

`cleanup` (Optional)
: If specified, this command is executed once the test is complete. It can be used to remove any executables which were compiled.

The `[scoring]` Section

This section specifies how tests are scored against each other to choose which is the best.

`repeat` (Optional, defaults to `1, min`)
: Gives the number of times each test should be repeated. When this option is set, the compilation and cleanup (if present) are still only run once, only the actual test is repeated. The variance, standard deviation and coefficient of variation will be displayed at the end of each test.

When a test is repeated, the repeated test scores must be combined into one overall score for the test. This can be one of the following: `min`, `max`, `med` or `avg`.

The number of repetitions and the aggregation function are specified as a comma separated pair. If only the number is given, `min` is used as the default aggregate.

`optimal` (Optional, defaults to `min_time`)
: Specifies whether to take the minimum or maximum score as being the best. The settings `min_time` and `max_time` use the running time of the `test` command as the score for a test. The settings `min` and `max` will use a custom figure-of-merit, which is read from the final line of output. This is described in the *Figures of Merit* section.

The `[output]` Section

Finally, a log can be saved, detailing the testing process:

`log` (Optional)
: Specifies the name of a `.csv` file which will be generated. The file will contain details of all the tests performed, which parameter values were used for each and what the individual and overall scores were. This can be used for more detailed analysis after tuning.

Warning: If this file already exists, it will be overwritten.

`script` (Optional)
: Specifies the name of a file which will be used to log a 'script' of the tuner's work. This includes which tests are being run and any output from the compilation and testing.

When this option is used, only a summary is shown on screen.

Warning: If this file already exists, it will be overwritten.

The Example Programs

There are some example programs to tune in the `examples` directory. These show how a few different parameterised programs can be tuned, the changes required to the programs and makefiles, and the configuration files required.

`examples/hello/`
: A trivial test case, which demonstrates how different parts of the system are connected. It compiles a 'hello world' program, written in C.

  Two parameters, named FOO and BAR, are completely ignored and one, $OPTLEVEL$, controls the compiler optimisation level flag.

  The running time of the program is used as a figure of merit, aiming to find the minimum (fastest). There are also example settings in the configuration file to optimise the file size of the generated executable.

`examples/laplace3d/`
: Uses compile-time parameters to tune the block size used by a CUDA GPU program.

`examples/looping/`
: This is another fairly trivial C program, which runs a loop as determined by the parameters $XLOOP$ and $YLOOP$.

  There are settings in the configuration file to measure the running time as measured by the tuner, or by using a custom script, `loop_test.sh`, which uses the `time` utility.

`examples/maths/`
: A very simple test case to demonstrate the use of a custom figure of merit. The parameters X, Y and Z are simply summed using the `expr` utility, with no compilation being performed.

`examples/matlab/`
: Demonstrates the use of run-time parameters to find determine the optimum level of 'strip-mining' vectorisation in a MATLAB program.

`examples/matrix/`
: Blocked matrix multiplication test case. This is the example used in the tutorial to demonstrate the changes required to the program. The original (not tunable) version is given in `original/`, the modified version after completing the tutorial is given in `modified/` and finally, there is a version which checks the blocked version against the naive version in `comparison/`.

The Utilities

The tuner comes with several utilities which can be used to analyse or visualise the results of the tuning process. They use `.csv` log files generated by the tuner. The utilities are all found in the `utilities` directory.

`utilities/output_gnuplot.py`
: This script converts a CSV log file into a gnuplot PLT file. This PLT file can be used with the gnuplot plotting program to produce a detailed graph of the testing process. If required, the PLT file can be modified by hand. The following options can be specified:

  `-h` or `--help`
  : Outputs some usage information and exits.

  `-r SCORE` or `--reference=SCORE`
  : Plots a reference score for comparison with the tuner's results.

  If `mylog.csv` is the tuning log and `myplot.plt` is the plot file to generate (it will be overwritten if it exists):

```
./output_gnuplot.py [-h] [-r SCORE] mylog.csv myplot.plt
```

`utilities/output_screen.py`
: This script reads a CSV log file and produces a graph displayed on the screen. This can then be saved if needed. The 'matplotlib' python library is required, which may not be installed by default. The following options can be specified:

  `-h` or `--help`
  : Outputs some usage information and exits.

  `-r SCORE` or `--reference=SCORE`
  : Plots a reference score for comparison with the tuner's results.

  `-s` or `--stddev`
  : Plots the standard deviation of multiple test repetitions.

  If `mylog.csv` is the tuning log:

```
./output_screen.py [-h] [-r SCORE] [-s] mylog.csv
```

`utilities/csv_plot.m`
: This is a MATLAB program which can be used to display a graph of the testing process.

  To use, modify the file as needed and use MATLAB to generate a graph.

Variable Independence

We say two variables are independent when they can be optimised separately. This is, FOO and BAR are independent if when we optimise FOO with BAR held at some fixed value, then that optimal value of FOO will be optimal for *any* setting of BAR (and vice-versa).

Independent variables are written as separate 'sets' of variables. The example we just saw would be written as:

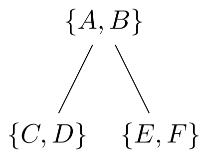
$$\{\{FOO\},\ \{BAR\}\}$$

More complex independences can also be written. As an example, suppose A and B control the operation of the entire program (maybe they are compiler flags), C and D control one aspect of the program and E and F another aspect. The aspect controlled by C and D is not related to the aspect controlled by E and F. This independence would be written like this:

$$\{A,\ B,\ \{C,\ D\},\ \{E,\ F\}\}$$

This shows the two sub-lists $\{C,\ D\}$ and $\{E,\ F\}$ are independent of each other and that the variables A and B 'dominate' those sub-lists. However, the sub-list $\{C,\ D\}$ shows that C and D are dependent on each other and must be tuned together. Similarly, A and B are dependent on each other.

This notation essentially describes a 'tree' of variables, where higher nodes dominate their subtrees and sibling nodes are independent:



Variable lists can be nested in this way as much as needed to describe the independences in the parameters being optimised.

Figures of Merit

By default (when using `optimal = *_time`), each test is run and its execution time is measured. This timing is used to rank the tests and choose the best. Sometimes, it is more useful to be able to choose a custom figure of merit. This can be because the part of the program being optimised is not the longest running part of the program, or because you wish to measure some other property, such as memory or network usage.

When the `optimal` option is set to `min` or `max`, the auto-tuner reads the output from the program and interprets the last line of output as the figure of merit. You are free to make any measurements necessary within the test itself (given by the `test` option). The score (a floating point or integer number with no units or other text) for the test must be output as the final line of the command's output.

Data Movement

Because arbitrary commands can be used for compilation, execution and cleaning, it is possible to use the parameters in any way you could from a command prompt. However, it is not always obvious how the parameter values can be ‘passed’ through the toolchain to where they are needed. The following list of tips may help. Although they are fairly linux/`make`/`gcc` specific, the ideas should still be applicable to any build tools.

To perform a ‘dry-run’ with only one test to check these settings, simply set a single possible value for each parameter.

Tuner $\Longrightarrow$ Shell Command

To pass the value of a parameter to a shell command, simply use the % substitution. If you are tuning a parameter named *FOO* (in the variable list) then `%FOO%` will substitute the current value of *FOO* being tested into the command to execute.

There is also a special substitution, `%%ID%%`, which is replaced by a

This can be used to set compiler flags, program arguments, and so on. For example, `hello.conf` contains the following lines:

examples/hello/hello.conf

```
compile = gcc %OPTLEVEL% -o bin/test_%%ID%% hello.c
...
test = ./bin/test_%%ID%% %FOO% %BAR%
...
clean = rm ./bin/test_%%ID%%
```

This will cause the following commands to be executed by the tuner:

```
gcc -O0 -o bin/test_1 hello.c
./bin/test_1 1 1                                          (Timed)
./bin/test_1 1 1                                          (Timed)
./bin/test_1 1 1                                          (Timed)
rm ./bin/test_1
gcc -O0 -o bin/test_2 hello.c
./bin/test_2 2 1                                          (Timed)
./bin/test_2 2 1                                          (Timed)
./bin/test_2 2 1                                          (Timed)
rm ./bin/test_2
...
```

The commands you enter are executed directly, as if typed into a command prompt. You are not limited to only running a compiler, or just your test program. It can be useful to use a makefile for compilation or a shell script for measuring and returning a custom figure of merit.

Tuner $\Longrightarrow$ Makefile

Parameters can be passed to the `make` command simply by appending `NAME=VALE` to the call to `make`, which will allow them to be used within a

`Makefile`. However, changing the parameters does not trigger a recompilation of the affected files (because there has been no change to the source code), so this must be forced with the `-B` option to `make`.

For example, `looping.conf` contains the following:[1]

examples/looping/looping.conf

```
compile = make -f MakeLoop -B EXEC_NAME=./bin/loop_%%ID%% XLOOP=%XLOOP% \
                            YLOOP=%YLOOP% OPTLEVEL=%OPTLEVEL%
```

These variables can now be used within the `Makefile` as if they were environment variables, with the syntax `$(NAME)`.

Makefile $\Longrightarrow$ Compiler

Once the parameters have been passed into the `Makefile` as described above, they can be used in the commands to be executed. For example, the `Makefile` for the 'looping' example contains the following, which passes the parameters to the compiler:

examples/looping/MakeLoop

```
gcc $(OPTLEVEL) -o $(EXEC_NAME) loop.c -D XLOOP=$(XLOOP) -D YLOOP=$(YLOOP)
```

Compiler $\Longrightarrow$ Program Code

As shown above, the 'looping' `Makefile` uses the `-D` option to `gcc` to set the parameter values for $XLOOP$ and $YLOOP$. This flag sets them as if they had been set with a `#define` statement in the program itself, so they can be used as constants within the program source code.

Tuner $\Longrightarrow$ Environment Variable

Some programs may be controlled by a certain environment variable, which you want to tune. For example, programs using OpenMP for parallelism can have the number of parallel threads controlled with the environment variable `OMP_NUM_THREADS`. To tune this variable, the `env` command can be used to run the program to be tested in an environment where `OMP_NUM_THREADS` has the value we want:

```
test = env OMP_NUM_THREADS=%OMP_NUM_THREADS% ./program
```

Command Sequencing

The commands given in the `compile`, `test` and `clean` options are passed directly to the shell, so features of the shell, such as using ; to run one command after another, can be used. For example, another possible solution to the above problem of setting an environment variable would be to export it, then run the command which depends on it:

```
test = export OMP_NUM_THREADS=%OMP_NUM_THREADS%; ./program
```

[1]Note that $EXEC\_NAME$ is a new variable for the `Makefile` only, it is not being tuned.

Return Codes

The tuner will check the return code of all the commands which are executed (`compile`, `test` and `clean`) to see if they failed. Any non-zero return code is considered a failure. This check is used to discount tests which either do not compile or do not run with one particular setting of the parameters.

This is part of the tuning process, as it discounts tests which will not run in your environment which may still run elsewhere. For example, if the number of parallel threads is being tuned, valid possible values on one machine may fail to compile on machines where there are more threads than processors.

If this causes problems, for example if your program routinely returns non-zero error codes, then the `test` command could be run from a 'wrapper' shell script which always succeeds, or the shell's sequencing operator (`;`) could be used to set the return code afterwards, the following will always return with code 0 (success), whatever the behaviour of `./program`:

```
test = ./program; true
```

If you do this, the system will not be able to detect failed tests, so they will still be timed (and will often fail quickly, leading to very low running times), so be careful of erroneous results.

Running the Tuner

The main script which runs the tuner is `tuner/tune.py`. In the top-level directory, `autotune` is a link to this and should be used to run the tuner.

While the tuner is running tests it can be interrupted with the `Ctrl+C` command. If the `log` option was used in the configuration file, then a partial log of the tests completed so far will be saved.

Dependencies

The tuner is written in **Python**, so you will need to have the Python interpreter installed. At least **version 2.5** is required, and Python 3000 is not supported.

So far, the tuner has mostly been tested under linux, but is designed to work on any platform (the most compelling reason for choosing Python was this portability and flexibility). On linux, the tuner will be directly executable with `./autotune` or `./tuner/tune.py`, but on windows it will need to be given as an argument to the python interpreter: `python tuner\tune.py`.

The utility `output_screen.py` (which plots graphs of the testing process on screen) requires the `matplotlib` Python module, which may not be installed with a standard python distribution. If it is not installed, you will not be able to use this utility, but the rest of the system will still work as normal.