General Auto-Tuning Framework
Developer's Guide

Ben Spencer
ben@mistymountain.co.uk

Last updated: 10th August 2011

Introduction

This guide provides an introduction to how the auto-tuner was designed and developed. It should be helpful to anyone who wants to understand how the tuner works, whether to use it more effectively, or modify it to better suit your needs.

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

Contents

Goals

The tuner is implemented in Python, and requires at least version 2.5.

Flexibility has been the overriding goal from the start. The programmer can specify arbitrary shell commands which are used to compile, run and score tests. This guide will give an overview of the high-level design of the tuner: how it is split into modules and what each is designed to do, as well as how they interact and together create the overall system.

I have tried to keep separate parts of the system independent of each other, which has mostly succeeded, but some different parts do rely heavily on the behaviour of others, even when they have been abstracted into separate modules.

Original Project Report

I originally began work on the tuner as a third year project towards my undergraduate degree. My project report[1] (which is very long) contains some very detailed information on the design and development of the system, especially the optimisation algorithm, which I explain in detail and prove correct.

It is worth remembering that the report was based on an older version of the tuner (0.11), so some parts are very different (notably tests are now run by the `Evaluator` class, rather than 'evaluation functions') but the broad architecture of the system and many of the details (for example the optimisation algorithm) remain the same.

High-Level Description

This section details the main classes providing the tuner's core functionality:

`VarTree`

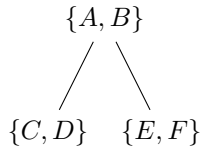
This class represents the variable trees which list the variables to be tuned and which are independent of each other. This structure is described (from a user's perspective) in more detail in the User's Guide (`doc/user.pdf`).

The variables are supplied to the system in a nested braces format:

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

This says that the variables A and B are dependent on each other, and on C, D, E and F, but that the sets $\{C,\ D\}$ and $\{E,\ F\}$ are independent. Being independent means that these sets can be tuned separately: if an optimal valuation of $\{C,\ D\}$ is found at one setting of $\{E,\ F\}$, then this will still be optimal for any other valuation of $\{E,\ F\}$.

This notation represents a tree of variables, where each node contains a set of dependent variables and a set of subtrees which are all mutually independent, but which depend on the 'parent' variables. The above example would be represented:



The `VarTree` class represents these trees. Each instance represents a tree node and they are linked together to represent the entire tree. The objects themselves do not require much extra information or functionality, they only contain a list of top-level variables and a list of subtrees. The only methods provided are to flatten the tree into a list of variables. All other operations on `VarTree` objects are provided as outside functions which manipulate them (the most interesting being `treeprint()`, which returns a string to display the tree structure in a terminal).

[1] `http://people.maths.ox.ac.uk/~gilesm/op2/autotuning-2011-05-30.pdf`

`Optimisation`

This class defines the optimisation algorithm used by the tuner. This algorithm exploits variable independence (given by a `VarTree`) to reduce the number of tests required, while still being comprehensive. It is parameterise by a `VarTree` Instance, listing the variables to tune and their independence; a list of the possible values of each variable; and an `Evaluator` instance, which handles the actual running of the tests.

The algorithm used is recursive, following the structure of the `VarTree`. At leaf nodes, every possible combination of the variables is tested by brute force. At branch nodes, the system recognises that each subtree is independent, so optimises them separately. For each possible valuation of the top-level variables, it recursively optimises the subtrees, one-by-one. The score when each subtree is optimised gves the best possible score for this setting of top-level variables. Once all the top-level valuations have been tried, the one which gave the best score is chosen and returned, along with it's subtree-optimums.

`Evaluator`

This class is used by `Optimisation` to actually test different settings of variables, and to keep a log of the tests, so the results can be looked up as they are needed. The `Evaluator` is parameterised by the commands required to compile and test a particular test, and how to calculate the overall score if repeat tests are being run. This class performs all the direct interaction with the shall and the tests which are run (e.g. reading their output and checking their return code).

When the optimisation algorithm reaches a leaf node of the `VarTree`, it submits a whole group of tests to the `Evaluator` at once. At the moment, these tests are simply run in sequence, compiling one, testing it a number of times and then cleaning it; before moving on to the next. However, because thests are submitted in groups, there is scope for other implementations of `Evaluator` to provide different evaluation strategies, such as running tests interleaved with other tests, to reduce the effects of inconsistent load on the system.

The log of tests performed is kept for two reasons. Firstly, it is used by the optimiser to check the scores of tests which were submitted. Secondly, it is used at the end of testing to create the `.csv` log file listing all tests performed.

A Tuning Run

This section describes a run of the progam from start to finish, showing how the different parts of the system are connected and work together to provide the final result.

The main program is `tune.py`. When the user runs this, they provide a configuration file as an argument. The first step is that `tune_conf.py` is used to read this configuration file (using the Python module `ConfigParser`) and validate the settings.

Next, an instance of `Evaluator` is created, parameterised by the compilation and testing commands, and the number of test repetitions.

An instance of `Optimisation` is created, and provided with the `VarList` and list of possible values read from the configuration file, and the `Evaluator` which it can use to run tests.

The system prints out a description of the information read from the configuration file and the tuning it is about to perform, then begins the tuning.

The optimiser works recursively over the structure of the `VarTree` and at each leaf node it submits a group of tests to the `Evaluator`. The `Evaluator` runs each test in sequence, logging the results. If each test is to be repeated, it runs them and calculates the overall score.

If the tests are to be timed by the system, the `Evaluator` does this timing to determine a test's score. If the tests use a custom figure-of-merit, then the `Evaluator` captures their output and checks the final line for the score, which is interpreted as a float.

The scores for each test are checked by the optimiser to find the best setting of the variables at that leaf node, which it selects and continues optimising the rest of the tree. As higher level variables are changed, it will be necessary to return to the leaves and fin new optimal settings for the changes in higher-level variables.

Once the optimisation is complete, the optimiser returns the best valuation found, as well as the score for that valuation, and the number of tests performed.

The log of testing is passed from the `Evaluator` to `logging.py` and written out as a `.csv` file, which contains all the details of the testing.

The `.csv` log file can then be processed (by the user, not automatically) with any of the log analysis utilities. These can be used to generate graphs of the testing process, to show (for example) which variables had the greatest effect.

File Listing

`autotune`
: The tuner (A link to `tuner/tune.py`).

`doc/`
: Documentation for the tuner.

  `dev.pdf`
  : Developer's documentation (this document).

  `tutorial.pdf`
  : A Beginner's Tutorial. Leads them through the setup and tuning of the matrix-multiplication test case.

  `user.pdf`
  : User's guide. A more comprehensive reference detailing all the features and abilities of the tuner.

`examples/`
: Example programs to demonstrate the system's use. Each comes with a sample configuration file which can be used to tune it, and most have some sample results from this tuning.

  `hello`
  : A simple test case which compiles a 'hello world' program with different levels of compiler optimisation.

  `laplace3d`
  : A CUDA test case. Compiles and tests a version of the `laplace3d` CUDA example from Mike Giles' CUDA programming course[2].

  `looping`
  : A simple test case where parameters control the number of loop iterations performed in the program.

  `maths`
  : A simple test case where parameters are summed using the `expr` command. Demonstrates the use of a custom figure-of-merit.

  `matlab`
  : A MATLAB program being tuned to determine the optimum level of 'strip-mining' vectorisation.

  `matrix`
  : A blocked matrix-matrix multiplication test case, which is tuned to find the optimal block size.

`README_Dev`
: Breif intro for developers, contains a file listing and change log.

`README_User`
: Breif intro for users, mostly points to the proper documentation.

[2] `http://people.maths.ox.ac.uk/gilesm/cuda/`

`tuner/`

The Python source code for the tuner itself.

`evaluator.py`
: Defines the `Evaluator` class, which controls how tests are evaluated, handling all compilation, execution and timing of tests. It also keeps a log of all tests performed, for use by the optimiser.

`helpers.py`
: Provides several small helper functions, which are used elsewhere.

`logging.py`
: Provides a function to output the testing log as a `.csv` file.

`optimisation_bf.py`
: A brute force optimiser, `OptimisationBF`, implementing the same methods as `Optimisation`. THis is used for testing, and in the system's self-demonstration.

`optimisation.py`
: The optimisation algorithm. Defines the `Optimisation` class. This class sets up and runs the recursive optimisation algorithm, which exploits variable independence to reduce the number of tests that must be performed.

`output.py`
: Controls where the tuner's output is sent. This can be printed to the screen or saved to a file. Sets up the three output possiblities `all`, `short` and `full` for the tuner to use.

`test_evaluations.py`
: Provides a 'dummy' `Evaluator` class, `FuncEvaluator`. This implements the same interface as `Evaluator`, but uses an 'artifical' evaluation function to score each test, rather than running any shell commands. This is used for the system's self-demonstration, as well as for testing. The evaluation function respects the variable independence of the problem being solved.

`testing.py`
: Checks `Optimisation` against `OptimisationBF` for a couple of different inputs. This is used as a demonstration of the system.

`tune_conf.py`
: Reads settings from the config file and performs some validation.

`tune.py`
: The main script. Reads the config file from the command line, sets up and runs the optimisation and reports the final reusults to the user.

`vartree_parser.py`
: Defines a `VarTree` parser, which was generated by `wisent`, a Python parser-generator[3]. This is used by `vartree.py` to convert token strings into parse trees. The language grammar was constructed so that these parse trees have the same structure as the corresponding `VarTree`, making the conversion simple.

[3] `http://seehuhn.de/pages/wisent`

`vartree.py`
: Defines the `VarTree` class and several related functions. These include a parser for creating `VarTree` objects from strings and a function to print a `VarTree` in a tree representation.

`utilities/`
: Contains utilities to analyse nd visualise the tuning results.

`common.py`
: Some helper functions for reading in the CSV file.

`csv_plot.m`
: A matlab script to draw plots.

`output_gnuplot.py`
: Outputs a `gnuplot` script for plotting graphs.

`output_screen.py`
: Displays a graph on screen with `matplotlib` (a Python graph plotting library).