Flamingo Auto-Tuning
Tutorial

Ben Spencer
ben@mistymountain.co.uk

Last updated: 28th September 2011

Introduction

This tutorial will lead you through a step-by-step description of setting up and tuning an example program. I'll explain each step as we go, so working through the tutorial should give you enough information to start tuning your own programs. The program we'll be tuning is a blocked matrix-matrix multiplication test, included with the auto-tuner as an example. If you have any comments or questions about the tuner or this tutorial then please feel free to get in touch.

Contents

What You'll Need 2

What We're Aiming For 3

The Build Chain 6

Modifying The Program 6

Writing a Configuration File 10

Running the Tuner 14

The Results 15

Using a Custom Figure-of-Merit 18

The End 18

What You'll Need

The auto-tuner
: This is the software which performs the tuning, and is found in the main `Autotuning` directory. The tuner can be run with the command `~/Autotuning/autotune` and performs a short self-demonstration if no arguments are provided.

A program to tune
: This tutorial will lead you through the setup and tuning of a small matrix multiplication program, which is included as an example with the tuner. The example code can be found in the `examples/matrix` directory. If you are going to follow along, the `examples/matrix/original` directory contains the unmodified code, which we will use as a starting point. The final version of the program we end up with can be found in `examples/matrix/modified`.

  The source code for this program is shown in Listing 2.

Programming tools
: We'll be editing and compiling the program. So you'll need a text editor and any program compilation tools required for your program. For this tutorial we'll be using `gcc` and `make` under Linux (this is not required by the tuner, which can tune a program using any build tools).

Installing the Tuner

To install the tuner, simply extract the `.tar.gz` file to some convenient location. In this tutorial, I have extracted it into my home directory, so the tuner is run with the command `~/Autotuning/autotune`.

You must also make sure you have Python version 2.5 or later. To check if Python is installed and which version you have, run the command `python --version`. For more information on installing Python see `www.python.org`.

What We're Aiming For

The program calculates the product of two matrices: $C := AB$. I'm representing the matrices simply by a 2D array. The simplest implementation (shown below) processes the array C row-by-row. To calculate an element $C[i][j]$, we need to read in an entire row of A and an entire column of B. The speed of the program can be improved by increasing the re-use of any data in the fast registers or L1 cache.

```
1  /* Perform C = C + A*B */
2  for(i=0; i<C_ROWS; i++)
3      for(j=0; j<C_COLS; j++)
4          for(k=0; k<A_COLS; k++)
5              C[i][j] += A[i][k] * B[k][j];
```

Listing 1: The 'naive' implementation of matrix-matrix multiplication. This is replaced by lines 62–69 of `matrix.c`

For large matrices, row-by-row processing will not allow the re-use of data from calculating $C[i][j]$ when calculating $C[i][j+1]$, even though some of this will be the same, or at least fetched in the same cache lines. The idea of this program is to split the matrix into blocks and process one block at a time. This allows more data re-use and so the program is faster.

The block size for each loop is controlled by the parameters $BLOCK\_I$, $BLOCK\_J$ and $BLOCK\_K$. Choosing good values (that is, giving good performance) for these parameters would require detailed knowledge of how memory is accessed and cached on the machine being used. How large is the L1 cache? How long are cache lines?

We will use the auto-tuner to automatically choose these parameter values and to see how much difference the choice can make to the program's performance.

matrix.c (Original Version)

```
/*
 * Autotuning System
 *
 * matrix.c
 * ORIGINAL VERSION
 *
 * A simple blocked matrix-matrix multiply algorithm,
 * partitioned by the block size. This is used as an example and
 * demonstration of the auto-tuner.
 */

#include <stdio.h>
#include <math.h>

/* Define the size of the matrices to work on. */
#define A_COLS 512
#define A_ROWS 512
#define B_COLS 512
#define B_ROWS A_COLS
#define C_COLS B_COLS
#define C_ROWS A_ROWS

/* The block size for the multiplication */
#define BLOCK_I 16
#define BLOCK_J 16
#define BLOCK_K 16


int main(void)
{
    double A[A_ROWS][A_COLS], B[B_ROWS][B_COLS], C[C_ROWS][C_COLS];
    int i, j, k, i_bl, j_bl, k_bl;

    printf("Blocked Matrix-Matrix Multiplication\n");

    /* Generate some arbitrary sample data. */

    for(i=0; i<A_ROWS; i++)
        for(j=0; j<A_COLS; j++)
            A[i][j] = exp(-fabs(i-j));

    for(i=0; i<B_ROWS; i++)
        for(j=0; j<B_COLS; j++)
            B[i][j] = exp(-fabs(i-j));


    /* Set C[][] = 0 first */
    for(i=0; i<C_ROWS; i++)
        for(j=0; j<C_COLS; j++)
            C[i][j] = 0;


```

matrix.c (Original Version, Continued)

```
53      /* Blocked Multiplication: C = AB */
54      /* Instead of processing an entire row of C at a time,
55       * process in small blocks of dimensions BLOCK_I * BLOCK_J. Elements
56       * required from A and B are also processed in blocks.
57       * This should improve local memory reuse. */
58
59      printf("(BLOCK_I = %d, BLOCK_J = %d, BLOCK_K = %d)\n", BLOCK_I, BLOCK_J,
60              BLOCK_K);
61
62
63      /* Perform C = C + A*B */
64      for(i=0; i<C_ROWS; i+= BLOCK_I)
65          for(j=0; j<C_COLS; j+= BLOCK_J)
66              for(k=0; k<A_COLS; k+= BLOCK_K)
67                  for(i_bl=i; i_bl<(i+BLOCK_I) && i_bl<C_ROWS; i_bl++)
68                      for(j_bl=j; j_bl<(j+BLOCK_J) && j_bl<C_COLS; j_bl++)
69                          for(k_bl=k; k_bl<(k+BLOCK_K) && k_bl<A_COLS; k_bl++)
70                              C[i_bl][j_bl] += A[i_bl][k_bl] * B[k_bl][j_bl];
71
72      /* Use C ... */
73
74      return 0;
75  }
76
```

Makefile (Original Version)

```
1
2   CC = gcc
3   CFLAGS = -lm
4
5   matrix: matrix.c
6           $(CC) $(CFLAGS) -o matrix matrix.c
7
```

Listing 2: The original version of `matrix.c` and the `Makefile` used to compile it, before any modification.

The Build Chain

Before tuning this program, we need to be clear about exactly what happens when we compile and run it. For this example it is fairly simple, but it is worth being sure of, especially when you try to tune a more complex program.

To compile the program, we run `make`. This reads the `Makefile`, which in turn contains a call to `gcc`, which is run by `make`. `gcc` reads in the source file `matrix.c` and compiles it into the executable file `matrix`. This executable can be run with the command `./matrix`.

Modifying The Program

To perform the auto-tuning, the tuner will need to test various different settings of the parameters. So to begin with, we must modify the program a little to allow the parameters to be set at compile-time by the auto-tuner.

We want to tune the three block size parameters: $BLOCK\_I$, $BLOCK\_J$ and $BLOCK\_K$. The first thing to do is wrap the definition of each in an `#ifndef` block, so they can be set by the compiler instead of being constant:

```
...
#ifndef BLOCK_I
    #define BLOCK_I 1
#endif
...
```

After this change, it is possible to set the parameters using compiler options. If you've made these modifications you can try it; for `gcc` the option we need is `-D NAME=VALUE`.

```
$ gcc -lm -o matrix matrix.c
$ ./matrix
Blocked Matrix-Matrix Multiplication
(BLOCK_I = 1, BLOCK_J = 1, BLOCK_K = 1)
$ gcc -lm -o matrix matrix.c -D BLOCK_I=32 -D BLOCK_J=16 -D BLOCK_K=8
$ ./matrix
Blocked Matrix-Matrix Multiplication
(BLOCK_I = 32, BLOCK_J = 16, BLOCK_K = 8)
$
```

We also need to modify the `Makefile` so it will pass the parameters to the compiler. The parameters are supplied as arguments to `make` simply as `NAME=VALUE` pairs and can then be used within the `Makefile` using `$(NAME)`. This allows us to supply the parameters passed to make as the `-D` arguments to `gcc`.

```
gcc -o matrix matrix.c -D BLOCK_I=$(BLOCK_I) ...
```

Once this is done, we can check that the parameter values are being correctly passed through the build chain. The `-B` option forces `make` to compile, even if there has been no change to the source code. `make` usually only recompiles a file if the source code has been modified, but in this case only the block size parameters passed to `gcc` have been changed, so we need to force a recompilation.

```
$ make -B BLOCK_I=3 BLOCK_J=4 BLOCK_K=5
gcc -lm -o matrix matrix.c \
                                        -D BLOCK_I=3 \
                                        -D BLOCK_J=4 \
                                        -D BLOCK_K=5
$ ./matrix
Blocked Matrix-Matrix Multiplication
(BLOCK_I = 3, BLOCK_J = 4, BLOCK_K = 5)
$
```

The final versions of `matrix.c` and the `Makefile` are given in Listing 3.

Aside: Make sure the test takes long enough

We'll add a loop to the program which performs the multiplication multiple times. This means the running time of the program is dominated by the time taken by the multiplication, so the program's 'overheads' will not add too much noise to the results. Any significant difference in the running time will be due to changes in the parameter values, not to random fluctuations in the time required to allocate memory for the arrays, and so on.

```
...
/* For timing the test, repeat the multiplication a number of times */
#define TEST_REP 5
...
    for(rep=0; rep<TEST_REP; rep++){
        ...
    }
...
```

matrix.c (Modified Version)

```
/*
 * Autotuning System
 *
 * matrix.c
 * AUTO-TUNING VERSION
 *
 * A simple blocked matrix-matrix multiply algorithm,
 * partitioned by the block size. This is used as an example and
 * demonstration of the auto-tuner.
 */

#include <stdio.h>
#include <math.h>

/* Define the size of the matrices to work on. */
#define A_COLS 512
#define A_ROWS 512
#define B_COLS 512
#define B_ROWS A_COLS
#define C_COLS B_COLS
#define C_ROWS A_ROWS

/* The block size for the multiplication */
#ifndef BLOCK_I
    #define BLOCK_I 1
#endif
#ifndef BLOCK_J
    #define BLOCK_J 1
#endif
#ifndef BLOCK_K
    #define BLOCK_K 1
#endif

/* For timing the test, repeat the multiplication a number of times */
#define TEST_REP 5


int main(void)
{
    double A[A_ROWS][A_COLS], B[B_ROWS][B_COLS], C[C_ROWS][C_COLS];
    int i, j, k, i_bl, j_bl, k_bl, rep;

    printf("Blocked Matrix-Matrix Multiplication\n");

    /* Generate some arbitrary sample data. */

    for(i=0; i<A_ROWS; i++)
        for(j=0; j<A_COLS; j++)
            A[i][j] = exp(-fabs(i-j));

    for(i=0; i<B_ROWS; i++)
        for(j=0; j<B_COLS; j++)
            B[i][j] = exp(-fabs(i-j));


```

Makefile (Modified Version, Continued)

```
56      /* Blocked Multiplication: C = AB */
57      /* Instead of processing an entire row of C at a time,
58       * process in small blocks of dimensions BLOCK_I * BLOCK_J. Elements
59       * required from A and B are also processed in blocks.
60       * This should improve local memory reuse. */
61
62      printf("(BLOCK_I = %d, BLOCK_J = %d, BLOCK_K = %d)\n", BLOCK_I, BLOCK_J,
63              BLOCK_K);
64      for(rep=0; rep<TEST_REP; rep++){
65
66          /* Set C[][] = 0 first */
67          for(i=0; i<C_ROWS; i++)
68              for(j=0; j<C_COLS; j++)
69                  C[i][j] = 0;
70
71
72          /* Perform C = C + A*B */
73          for(i=0; i<C_ROWS; i+= BLOCK_I)
74              for(j=0; j<C_COLS; j+= BLOCK_J)
75                  for(k=0; k<A_COLS; k+= BLOCK_K)
76                      for(i_bl=i; i_bl<(i+BLOCK_I) && i_bl<C_ROWS; i_bl++)
77                          for(j_bl=j; j_bl<(j+BLOCK_J) && j_bl<C_COLS; j_bl++)
78                              for(k_bl=k; k_bl<(k+BLOCK_K) && k_bl<A_COLS; k_bl++)
79                                  C[i_bl][j_bl] += A[i_bl][k_bl] * B[k_bl][j_bl];
80
81      }
82
83      /* Use C ... */
84
85      return 0;
86  }
87
```

Makefile (Modified Version)

```
1
2   CC = gcc
3   CFLAGS = -lm
4
5   matrix: matrix.c
6           $(CC) $(CFLAGS) -o matrix matrix.c \
7                                           -D BLOCK_I=$(BLOCK_I) \
8                                           -D BLOCK_J=$(BLOCK_J) \
9                                           -D BLOCK_K=$(BLOCK_K)
10
```

Listing 3: The modified versions of `matrix.c` and the `Makefile`, which are now ready for tuning.

Writing a Configuration File

Now that the program and build chain is ready for auto-tuning, we must create a configuration file. This file will tell the tuner which parameters need to be tuned, what the possible values for them are and how to compile and run each test. The configuration file is simply a normal text file, similar to Linux configuration files, or Windows `.ini` files.

The file is split into five sections: *variables*, *values*, *testing*, *scoring* and *output*, which begin with a line `[section_name]`. Options are set using the syntax `name = value` and lines beginning with `#` are comments.

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

The final configuration file is shown in Listing 4, and should be saved in the same directory as the program. A detailed description of all the configuration file options and their operation can be found in the User's Guide (`doc/user.pdf`).

The `[variables]` Section

This section contains a single option, `variables`, which says which program parameters should be optimised. For this example we simply provide the list. It is possible to describe independences between the parameters here, which is often useful, but not for this program. Look up 'Variable Independence' in the main User's Guide for more information on this.

```
[variables]

variables = BLOCK_I, BLOCK_J, BLOCK_K
```

The `[values]` Section

This section lists the possible values for each variable. I've chosen quite a wide range so I can get an idea of how blocking is affecting the running time. However, don't put too many possibilities in, or the tuner will have a huge number of tests to run. Remember, as we haven't specified any independence between the variables, every possible combination of parameter values will be tested.

```
[values]

BLOCK_I = 4, 8, 16, 32, 64
BLOCK_J = 4, 8, 16, 32, 64
BLOCK_K = 4, 8, 16, 32, 64
```

The `[testing]` Section

This section contains the settings which define how each test is run.

The `compile` option sets the command used to compile each test. For our example, this will be a call to `make`, passing the parameters. Placeholders such as `%BLOCK_I%` will be substituted with the value of $BLOCK\_I$ for each particular test. The substitution `%%ID%%` expands to a unique test ID, which is useful when more than one test might exist at once, but is not needed here. For more information, see the User's Guide.

```
[testing]

compile = make -B BLOCK_I=%BLOCK_I% BLOCK_J=%BLOCK_J% BLOCK_K=%BLOCK_K%
```

The `test` option gives the command required to run each test. This command will be timed to give a score for the test.

```
test = ./matrix
```

The `clean` option can be used to run a command after a particular test is over (for example to remove any generated executables or object files), but we don't need it for our example.

The `[scoring]` Section

This section defines how tests are scored to determine which is best. Each test is assigned a score, which would typically be its running time, but any other property can be tuned.

The tuner can either minimise or maximise the score, which is chosen with the `optimal` option. The default is to minimise, which is what we want here. The possible settings are: `min_time` (the default), `max_time`, `min` and `max`. When the `min_time` or `max_time` options are used, the tuner will time the execution of the `test` command above.

If the options `min` or `max` are used, then the `test` command must calculate its own score and output it as the final line of output. For this test, we want to measure the running time, but custom scoring is discussed in the section *Using a Custom Figure-of-Merit*.

Each test can be repeated a number of times, using the `repeat` option. This helps to smooth over any variation in running time, so we'll set this to 3.

When a test is repeated, the overall score is some combination of the individual test scores. As we're measuring running time, which is only increased by any inaccuracies (other programs running in the system), we want to choose the minimum of the individual scores as the overall score for each test. This is also set with the `repeat` option, and `min` is the default (the possible settings are: `min`, `max`, `med` or `avg`).

```
repeat = 3, min

#optimal = min_time
```

The `[output]` Section

The `log` option allows a log of the testing process to be created. If set, it gives the file name of a `.csv` file which will list all tests performed. This log file can be used to generate graphs of the testing process which will help us to explore the effect that block size has on the program. We'll see how these logs can be analysed later.

```
log = results/matrix_log.csv
```

There is one final option, `script`, which allows the output of the tuner to be dumped into a 'script' file. This reduces the amount of output shown on screen, so we won't use this for now.

matrix_tune.conf

```
# Autotuning System
#
# matrix_tune.conf
#
# Blocked Matrix Multiplication Tuning

[variables]

variables = BLOCK_I, BLOCK_J, BLOCK_K


[values]

BLOCK_I = 4, 8, 16, 32, 64
BLOCK_J = 4, 8, 16, 32, 64
BLOCK_K = 4, 8, 16, 32, 64


[testing]

compile = make -B BLOCK_I=%BLOCK_I% BLOCK_J=%BLOCK_J% BLOCK_K=%BLOCK_K%

test = ./matrix

#clean =


[scoring]

repeat = 3, min

#optimal = min_time


[output]

log = results/matrix_log.csv
```

Listing 4: The configuration file used for testing.

Running the Tuner

Now that we have prepared the program and written a configuration file, we're ready to begin tuning.

Before you start, it is worth making a quick calculation of how long the testing is likely to take. On my machine, with $TEST\_REP$ set to 5(the number of repetitions of the multiplication part of the program), running the program takes around 11s. There are 3 variables with 5 possible values each, so $5^3 = 125$ tests will be required. Each test runs the program 3 times, so the tuning will take around 70 minutes in total, which is fine for our tuning. If this is too long, you can reduce the number of possible values for the parameters, reduce the number of test repetitions or reduce the $TEST\_REP$ variable within the program.

To begin testing, simply run the tuner, `~/Autotuning/autotune`, with the name of your configuration file as an argument. This will first output what testing will be performed (the variables, their vales, and so on) and then begin running tests. This will usually take some time, but you will be able to see the tuner's progress on screen as it runs tests.

Once all the tests have been run, the log files will be saved and the tuner will tell you which setting of the parameters it found to be the best.

```
$ ~/Autotuning/autotune matrix_tune.conf

                                Autotuning System
                                      v0.16

Retrieved settings from config file:

Variables:
BLOCK_I, BLOCK_J, BLOCK_K

Displayed as a tree:

 {BLOCK_I, BLOCK_J, BLOCK_K}

Possible values:
BLOCK_I = ['4', '8', '16', '32', '64']
BLOCK_J = ['4', '8', '16', '32', '64']
BLOCK_K = ['4', '8', '16', '32', '64']

compile:
make -B BLOCK_I=%BLOCK_I% BLOCK_J=%BLOCK_J% BLOCK_K=%BLOCK_K%

test:
./matrix

Number of tests to be run: 125
(with 3 repetitions each)


Test 1:
...
```

The Results

Now the testing is done, it's time to look at the results. The last few lines of output from the tuner will tell you which parameter values resulted in the shortest running time. In this case, the optimal valuation was to set $BLOCK\_I$ and $BLOCK\_J$ to 64, and $BLOCK\_K$ to 4.

```
...
Minimal valuation:
BLOCK_I = 64, BLOCK_J = 64, BLOCK_K = 4
Minimal Score:
9.95934987068
The system ran 125 tests, taking 75m47.83s.
A testing log was saved to 'matrix_log.csv'
$
```

This may be useful on its own, but we can get a better understanding of how the parameters affect the program by looking at the log files. An extract from the `.csv` log from my tuning is shown below.

TestNo	BLOCK_I	BLOCK_K	BLOCK_J	Score_1	Score_2	Score_3	Score_Overall
1	4	4	4	12.620	12.647	13.141	12.620
2	8	4	4	11.237	11.242	11.585	11.237
3	16	4	4	11.355	11.362	11.533	11.355
4	32	4	4	11.107	11.293	11.341	11.107
5	64	4	4	10.805	11.086	11.207	10.805
6	4	8	4	12.140	12.188	12.830	12.140
7	8	8	4	11.541	11.556	11.581	11.541
⋮		⋮			⋮		⋮
124	32	64	64	12.921	12.936	12.961	12.921
125	64	64	64	12.873	12.879	12.924	12.873

To produce a graph of the testing results, we use one of the tuner's output utilities to convert the `.csv` log into a `gnuplot .plt` script. This `.plt` file can be edited by hand to alter the appearance of the graph, for example to change the labels or colours. Then we can use the `gnuplot` plotting program to generate the graph. Some instructions are given at the top of the generated `.plt` file, and here is what I did. The graph I generated is shown on Page 16. For more information on `gnuplot`, see `www.gnuplot.info`.

```
$ ~/Autotuning/utilities/output_gnuplot.py matrix_log.csv matrix_log.plt
Reading 'matrix_log.csv'
Generating gnuplot script
Writing 'matrix_plot.plt'
Done
There are some instructions for generating a png at the top of the file.
$ gnuplot
...
gnuplot> set terminal png large size 1500, 1800
Terminal type set to 'png'
Options are 'nocrop font /usr/share/fonts/truetype/ttf-liberation/LiberationSans
-Regular.ttf 14 size 1500,1800 '
gnuplot> set output './matrix_plot.png'
gnuplot> load './matrix_plot.plt'
gnuplot> exit
$
```

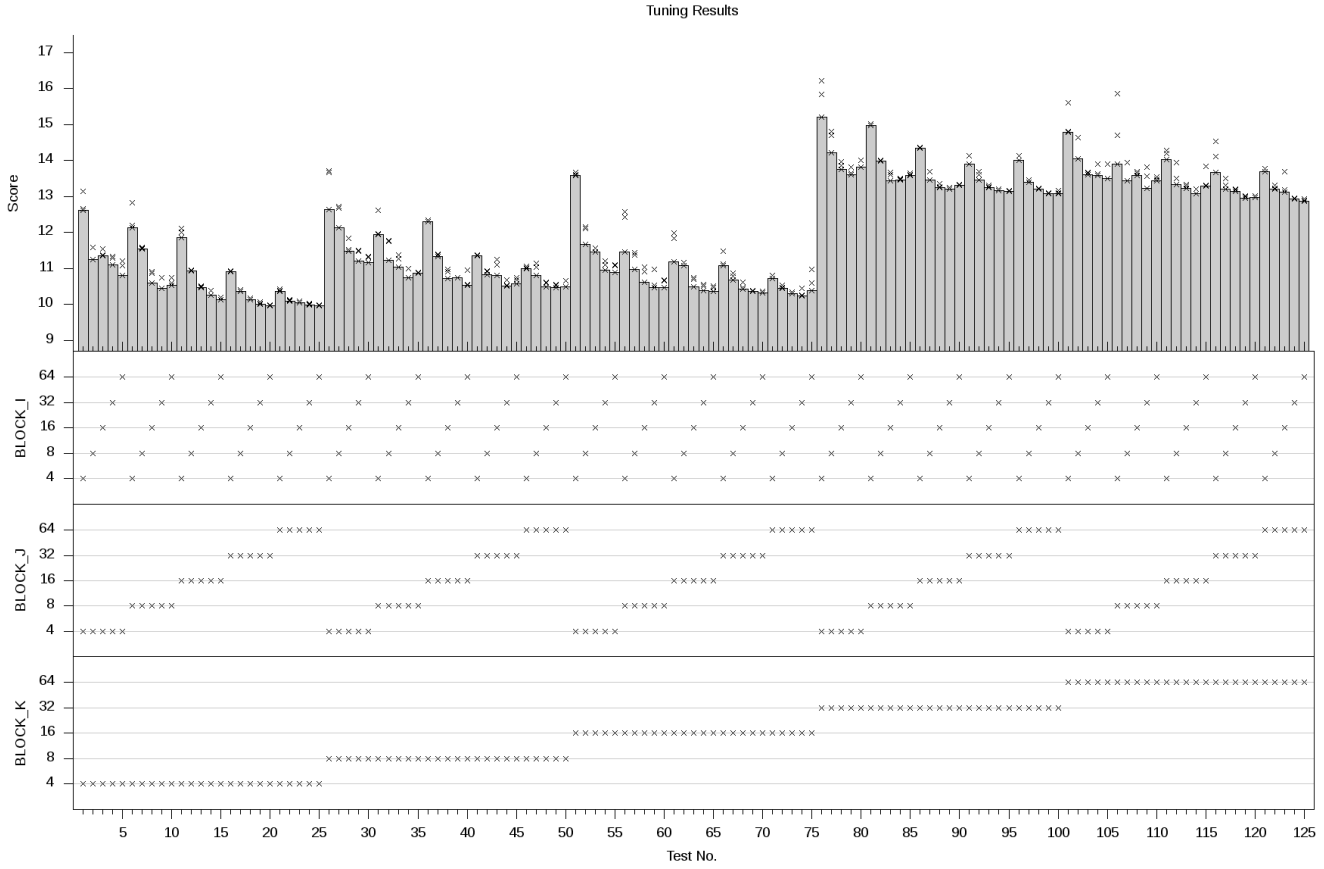

Figure 1: The graph of testing results, generated by `gnuplot` using `matrix_plot.plt`

Analysis

The bottom half of the graph shows which tests were run. The test number is shown along the x-axis, and the setting of each variable is shown on the sub-graphs. The top half of the graph shows the score for each test. The grey bars show the overall score which is used to compare the tests, and the small crosses show the individual scores from each repetition of the test.

The patterns in the graph allow us to clearly see how each variable affects performance. Firstly, the graph has five distinct 'sections' or 'steps', which correspond to the changing values of $BLOCK\_K$. It seems that 4, 8 and 16 were all quite good values, while 32 and 64 were significantly worse. Next, we can see a 'saw-tooth' pattern within each section. These correspond to the settings of $BLOCK\_J$ and seem to decrease as $BLOCK\_J$ increases, indicating that high values are better. Finally, each peak and dip of the saw-tooth pattern corresponds to varying $BLOCK\_I$. Higher values of $BLOCK\_I$ also seem to be best.

Further Tuning

The optimal results of both $BLOCK\_I$ and $BLOCK\_J$ were at the upper extreme of the possible values we supplied. Also, there is a definite trend towards higher values of these two variables being better. This could motivate the idea to try another tuning run, but with even higher values of $BLOCK\_I$ and $BLOCK\_J$. For $BLOCK\_K$, we might choose to tune a lower range of values, or we might choose a wider, sparser range, to cover more possibilities both higher and lower.

This type of additional run allows us to perform very detailed tuning, but only on a very small part of the space of possible values. If we had tried to perform the detailed tuning in the first place, there would have been far too many tests. A coarse tuning gives enough information to intelligently choose a more refined search.

Using a Custom Figure-of-Merit

The testing we performed timed the entire running of the program and used this as the score for a test. In some cases, you will want to use some other measurement as the score for a test. In our example, if we decided that the program overheads were large compared to the part we wanted to time, we could use a timer of only the multiplication part to provide the score.

To set this up, we would add timing code to our program, which timed only the multiplication. In the configuration file, instead of using the option `optimal = min_time`, we would use `optimal = min`. This tells the tuner that you are going to provide the score yourself, so it will not time the entire execution. Instead, it will expect the score to be given as the last line of output of the test. So for our example, we would print out the time taken by the multiplication as the last piece of output from the program. The tuner would then read this and use it as a score.

When using a custom figure-of-merit like this, you are not restricted to using running time at all. If you wanted to optimise the memory bandwidth, or some other property, all you need to do is have your program measure it, then output the score (as a float or integer) as the last line of output.

The User's Guide contains more information about using a custom figure-of-merit to tune other properties of a program than the overall running time.

The End

Hopefully this tutorial has shown you enough to let you begin tuning your own programs. The main takeaway is to see how the tuning process works: first modifying the program and build chain, then setting up a configuration file, and finally running the tuner. It is important to be clear about how the variables from the tuner will be passed from the tuner to the `Makefile`, then to `gcc` and finally to the program.

I have covered all but one of the program's main features. I have not discussed Variable Independence at all, which allows you to reduce the number of tests which need to be run. For more complex programs, the savings in tuning time can be substantial.

For more information about the tuner and a more detailed description of each option, you can look at the User's Guide (`doc/user.pdf`). Alternatively, try some of the other example programs included with the tuner (found in `examples/`).

If you have any comments or questions on the tuner or this tutorial, please feel free to get in touch.