A General Auto-Tuning Framework for Software Performance Optimisation

Ben Spencer

Balliol College, University of Oxford

Third Year Project Report

Trinity Term 2011

Thanks to my supervisors Professor Mike Giles and Dr. Alastair Donaldson.

Abstract

This report presents a new general purpose framework for automatically tuning program parameters without resorting to a brute-force approach.

It is difficult, even for the experienced programmer, to choose optimal settings for many program parameters. The rise of GPUs as a widely available parallel programming platform has further highlighted the importance of automatic tuning of program parameters in modern software development. Although the design and development of the system is driven by the problems associated with parallel GPU programming, its general nature makes it applicable not only to other parallel programming architectures, but also to many unrelated software development problems.

Independence between parameters can be exploited to significantly reduce the number of tests which need to be run and therefore the amount of time required for tuning. Testing has confirmed the system's effectiveness—it is faster and more convenient than hand tuning or brute-force auto-tuning, yet still guarantees to find optimal settings for the program's parameters.

The system has already been used in OeRC (*Oxford e-Research Centre*) to provide tuning results which were presented at the *Many-Core and Reconfigurable Supercomputing Conference 2011* [13] and are to be included in a forthcoming paper for the *Journal of Parallel and Distributed Computing* [12]. These results are described in Section 4.3.

Contents

1

Introduction

This chapter presents some of the problems I am aiming to solve and explains why auto-tuning is an important and appropriate solution.

My report focuses on the problems encountered in GPU programming, as that is the context in which the system was developed. The ideas discussed are applicable to many other areas, but for concreteness, the project is presented in terms of parallel programming on GPUs.

1.1 Auto-Tuning

Auto-tuning is the process of automatically choosing settings for a program's parameters, usually with the aim of improving its running time. These parameters may determine anything from compiler optimisations to the control flow of the program.

Importance of Auto-Tuning

Often, an expert programmer will be able to determine a parameter's optimal value. However, most programmers will not typically have in-depth knowledge of the hardware architecture underlying their programs, and knowing how a program will be executed is increasingly difficult in modern systems, especially those which are highly parallel. Auto-tuning helps programmers to get the best possible performance, despite hardware being complex and difficult to predict.

Auto-tuning also has a place in parallel computing frameworks such as OP2 [11], which should provide portability across a variety of different parallel hardware architectures, without sacrificing performance. If such frameworks

make use of auto-tuning, programmers could expect optimal performance from their programs without being experts in parallelising algorithms or having in-depth knowledge of the many target hardware platforms. This makes parallel computing more accessible to scientists without a parallel programming background.

Existing Software Tuning

Individuals wishing to choose good parameter values for their programs typically tune them by hand—manually testing various settings for each variable. Sometimes, simple shell scripts are created (for example Figure 1.1) which perform a brute-force search of all possible valuations. These scripts work well for small parameter ranges and are easy to use, but are typically purpose-built and hard to adapt for other applications [8]. For large parameter ranges, brute-force auto-tuning is very slow.

Some auto-tuning systems exist already, mainly in high-performance and scientific computing libraries. They tend to tune once when installed to determine the best settings and optimisations for the hardware they are running on. This approach is appropriate when the library will be used for a variety of problems and should provide good performance for all of them. These systems are clearly very application specific.

ATLAS [24] is a Linear Algebra library providing BLAS and some LAPACK routines. The system provides portable performance across a range of CPU architectures by using a database of precomputed optimisations for known architectures or by auto-tuning for new architectures. This auto-tuning is used to choose optimisations which are effective for a particular CPU. This install-time tuning uses a set of sample tests of the linear algebra routines. The tests are likely to be very similar to actual use, making this an effective way to choose optimisations——the tuning gives comparable results to carefully hand-tuned, machine-specific libraries. PHiPAC [6] and OSKI [22] use similar techniques.

FFTW [9] and NukadaFFT [17] provide FFT (Fast Fourier Transform) implementations which use auto-tuning to improve performance. FFTW is a CPU implementation which uses auto-tuning to construct good execution plans. NukadaFFT runs on GPUs using CUDA and uses auto-tuning to choose various

script_cuda.cu

```
for i in 64 128 256 512 1024; do
  echo "PARTITION SIZE = $i";
  for j in 64 128 256 512 1024; do
    echo "BLOCK SIZE = $j `./airfoil_cuda OP_BLOCK_SIZE=$j OP_PART_SIZE=$i `";
    echo "++++++++++++++++++++++++++++++++++++++++++++++++++";
  done
  echo "==================================================";
  echo " ";
done
```

Figure 1.1: This is an example of a small shell script used in OeRC. It is used for brute-force tuning of the airfoil simulation discussed in Section 4.3

.

GPU optimisations including data-padding, number of parallel thread blocks and ordering of data. SPIRAL [25] is another auto-tuning based digital signal processing library.

Compiler-based auto-tuning is also the focus of much research, (for example [4], [20] and [26]). These systems typically test various compiler-level optimisations, such as loop-unrolling, to find the best for a particular program.

My project's approach is slightly different, providing a general auto-tuning system applicable to any software development. Focusing on problem-by-problem tuning allows the use of optimisations which are useful in specific cases but not in general. This is advantageous in programs such as simulations, where the type of computation required is consistent within a problem but not necessarily between different problems. The aim is for complete generality—the system should be able to help a programmer on any platform to improve performance on unknown hardware.

FFTW goes some way toward this idea of problem-by-problem optimisation. An execution plan is created based on the hardware being used and the memory layout of the problem being solved. This plan can then be used when solving any problem of the same 'shape' [9, §1].

1.2 GPU Programming

General Purpose GPU programming (using a Graphics Processing Unit for computation other than rendering graphics) has become a very active field in recent years, with many new applications and many new programmers rushing to use this easily available parallel computing resource. Many applications in scientific computing can be naturally mapped onto a parallel architecture, so there is a lot of interest in using GPUs as small and relatively cheap parallel processors.

The increasing popularity of this area makes it necessary to develop tools which can help programmers work with this new and unfamiliar architecture.

GPU Architecture

GPUs provide several parallel processors each containing many individual processing cores. Each core has its own register space and cores on an individual processor share a larger portion of 'shared memory'. This architecture, the numbers of cores, the exact amounts of memory and how it is assigned are all dependent on the exact model of GPU, with significant differences between models. Figure 1.2 gives some examples and Figures 1.3 and 1.4 show how parallel threads are split across the processing cores.

GPU	Multiprocessors	CUDA Cores
GeForce 8800 GTX	16	128
GeForce GTX 460	7	336
GeForce GTX 560 Ti	8	384
Tesla C2070	14	448

Figure 1.2: The number of multiprocessors and processing cores for a variety of NVIDIA GPUs.

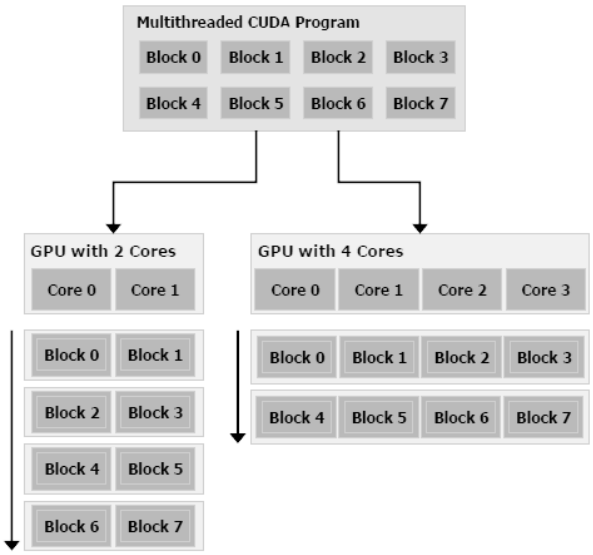


Figure 1.3: From the CUDA Programming Guide. This diagram shows how a parallel program is split into multiple blocks of threads, each of which is assigned to a GPU processing core.

Parallelising a Problem

When developing GPU programs, a large problem must be solved by breaking it into small blocks and solving these in parallel. These solved blocks are then pieced together to give the final result. This piecing together process can often also be performed in parallel, using tree reduction. Each parallel thread has limits on the resources it can use, such as registers and access to shared memory.

Deciding exactly how to break down a problem depends on the underlying hardware architecture and the problem being solved. Different models of GPU provide different features and different problems have varying hardware requirements for each thread or for groups of threads.

Non-Portable Performance

GPU programming frameworks such as OpenCL [1] and CUDA [2] provide a programming environment which guarantees program correctness across a variety of hardware. An OpenCL program can be developed on a system with an AMD GPU which will run on other AMD or NVIDIA GPUs or a CPU.

However, although the program's correctness is guaranteed, performance is generally not portable. Due to the different underlying architectures, different models of GPU—even those designed by the same hardware provider—will perform differently and may require different optimisations for best performance.

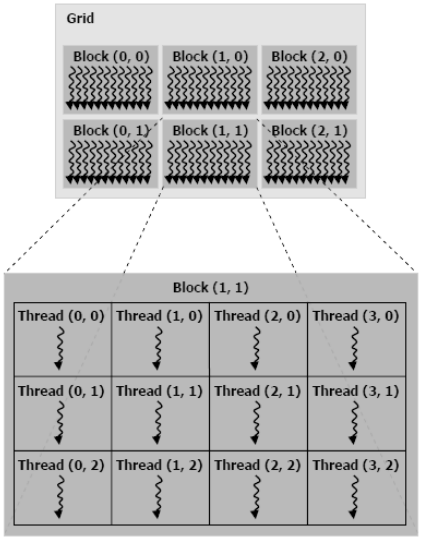


Figure 1.4: From the CUDA Programming Guide. Threads in a CUDA program are grouped into thread blocks and blocks are arranged in a grid. The programmer can choose the block dimensions, which will affect the execution of their program.

Block Size Choice

When executing a GPU program, many individual threads are executed in parallel. As memory access is slow compared to clock frequency, when one thread accesses memory another can be scheduled on the processor while it waits. Threads on a processor share the processor's registers between them so no state needs to be saved and this context switch can be performed in a single clock cycle. This thread scheduling masks the memory latency, but requires a large pool of threads waiting to be executed.

A balance needs to be struck between maximising the parallelism of a problem (providing more threads in the pool to better mask latency) and the limitations on the memory available to each thread [15]. Because the threads on a processor must share registers, only a certain number can be running at once on each processing core. If the problem is split into too many threads, each will still require registers but will not perform enough computation to mask the memory access latency, and the resulting overhead will slow down execution. Also, other factors—such as whether caches in the GPU are used, or how memory is split between local caches and higher level shared memory—will affect performance differently depending on how the problem is broken down.

It is difficult even for an experienced GPU programmer to determine what size blocks the problem should be broken into and how these should be allocated to the GPU's processing cores. In many situations, it is beneficial to try to have as many threads as possible running in parallel. However, this means there are less registers available per thread and may give worse performance than scheduling fewer threads at a time.

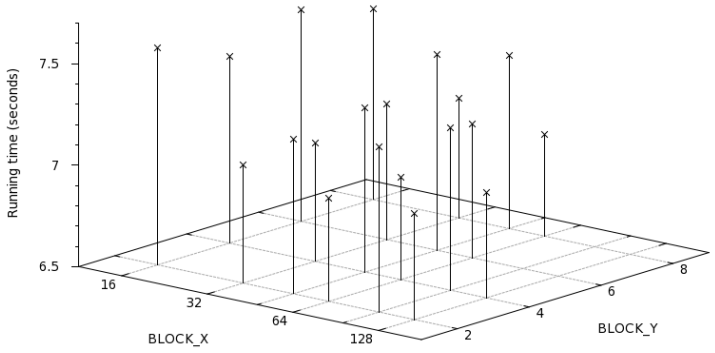


Figure 1.5: This graph shows how the running time of the `adi3d_naive` test from Section 4.2 varies with the thread block dimensions. There is little correlation, making good parameter choice difficult.

1.3 Feasibility of Auto-Tuning

Scaling Optimisations

To perform the tuning, the programmer chooses a small problem, which runs quickly enough to be executed many times during testing. Optimal parameter values from this example are then used in the actual problem, which will be much larger. It is the programmer's responsibility to choose a suitable test which is representative of large problems.

Choosing such a test is easy for many problems, which have obvious small cases running similarly to large cases. In scientific computing, many simulations use time-marching algorithms, where each state is calculated in turn to progress the simulation. As long as the hardware is fully utilised, optimisations for a short simulation of 100 time steps can be translated to the full simulation of 10,000 or 1,000,000 steps. Simulations which can be performed at a lower resolution, such as weather modelling, are another example. Tuning could be performed at a coarse resolution of 10km and the results used in a simulation at 100m resolution which will perform the same kinds of calculations, with similar optimal values.

Running Time

Trying all possible parameter valuations requires an exponential number of tests to be performed. There is no polynomial time algorithm to find an optimal valuation (Theorem 3.4.1) or even to find an approximate solution within a constant factor of optimal (Theorem 3.4.3).

However, the number of parameters and possible values is typically small, even for real-world tuning. The parameters are often hardware or environment dependent and the programmer will have a good idea which are important and what good candidate values are. They will not need to test every possible program variation.

vectorAdd.cu

```
38  // Device code
39  __global__ void VecAdd(const float* A, const float* B, float* C, int N)
40  {
41      int i = blockDim.x * blockIdx.x + threadIdx.x;
42      if (i < N)
43          C[i] = A[i] + B[i];
44  }
```

vectorAdd.cu

```
75      // Invoke kernel
76      int threadsPerBlock = 256;
77      int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
78      VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

Figure 1.6: This code is from a CUDA SDK example program performing the vector addition $C = A + B$. The first extract defines a CUDA kernel, which will be a single thread executing on the GPU. This thread uses its position within the thread block to determine which index of the vector add it is responsible for. The second extract shows how the kernel is invoked, setting the number of blocks in the grid and threads per block.

For example, when choosing a block size for GPU programming, the programmer will know that multiples of 32 are most efficient, as threads are executed 32 at a time on each processing core. Although the optimal block size depends on the problem and the exact GPU, the programmer can easily narrow down the candidate block sizes to, for example, $\{16, 32, 64\}$ in the x-dimension and $\{2, 4, 6, 8\}$ in the y-dimension. The auto-tuning is still beneficial, by optimising for different hardware, but hundreds of inefficient valuations can immediately be discounted.

Many parameters to be optimised do not affect each other's optimal values. This independence can be used to optimise each in turn, significantly reducing the number of tests required. For example, after the vector addition from Figure 1.6, the programmer may launch another kernel to multiply the vectors. The parameters for launching the first kernel would not have any effect on those for the second and could be optimised separately. This type of independence and the improvements which can be gained by exploiting it are discussed in detail in Section 2.3.

Finally, even a small percentage improvement might save days from a month-long simulation. Even if tuning takes many hours, as long as it produces useful results then the potential savings are large enough to make it worthwhile. It is the programmer's responsibility to choose sufficiently few parameters and possible values, and a small enough test case that the tuning runs in an acceptable amount of time.

1.4 Developing an Auto-Tuning Framework

My project is the development of a general-purpose auto-tuning system which is applicable to many problems. In particular, GPU programmers will benefit, as hardware complexity makes getting optimal performance from a program difficult, even for an expert. The system helps to mask subtle but important differences between architectures by tuning individually for each platform.

Objectives

The system should help a programmer choose optimal values for parameters affecting their program's execution. Usually the motivation would be to improve the running time by fully utilising the available hardware, although other properties of the program might be optimised instead. It will allow a single program to be optimised differently for execution on distinct hardware platforms, without knowledge of the underlying architecture.

2

System Requirements

Now the problem has been described, I will discuss in more detail the requirements for the auto-tuning system.

2.1 Usability and Effectiveness

As the auto-tuning system is an aid to program development, it is appropriate for the it to be a command line utility and for the programmer to use shell commands to specify how to compile and run tests.

The system must be easy enough to use that it is a clear improvement over brute-force tuning, either by hand or using simple shell scripts.

A brute-force search of all possible valuations guarantees finding the optimum. As testing is not necessarily exact, in practice a valuation which is close to optimal—within 5% for example—would be fine. My optimisation algorithm *does* in fact give an optimal valuation (Section 3.3) and the optimisation cannot be approximated within any constant factor unless $P = NP$ (Section 3.4).

2.2 Operation

To tune compile-time and run-time parameters, the system must compile and run tests—as specified by the programmer—with parameters set to different values.

Input

Input is provided in a configuration file (an example is included with my source code listings). This is useful because the optimisation can be set up, saved, and re-used and some options are quite complex.

The configuration file defines the following settings:

- Which parameters are to be tuned.
- What their possible values are.
- How to compile a test.
- How to run a test.
- How to determine a score for a particular test. Usually this will be the running time, but other options might be useful. These 'figures of merit' are discussed below.
- Other testing options, such as the number test repetitions, how to combine repeated test scores and whether to minimise or maximise the score.

Generality

Tuning should not be limited to specific programming environments, operating systems, or types of problem. It should be possible to tune any type of parameter: run-time arguments, `#define` constants or compiler flags, for example. The following sections discuss some of these ideas in detail.

Choice of Language

The system is implemented in Python, due to its portability and shell interaction support. Python has interpreters on all major operating systems and is installed by default on many systems used for programming. There are several comprehensive built-in modules for spawning sub-processes and reading their output. Compiling and running arbitrary tests is simple and platform independent. It is also easy to keep the system modular and extensible, making future updates simpler.

Compilation, Execution and Testing

The commands used for compilation and testing are provided by the programmer in the configuration file. As the programmer is able to specify these commands, leaving placeholders for the parameters being tuned, the system is very portable and general-purpose. It does not depend on specific programming environments—using `make` and `gcc`, for example—as the programmer can define exactly what needs to be done to compile and run their program.

During tuning, it is possible that some tests will fail to run due to hardware limitations or poor parameter choice. Failed tests (detected by their return code) should clearly not be considered optimal—even if they finish very quickly—they should simply be ignored. This allows the tuning to automatically choose from only those valuations which will run on that particular hardware configuration.

Figures of Merit

Usually, tuning will be used to minimise the running time of a program. However, in some cases different 'figures of merit' by which each valuation is scored are useful. This allows other properties of the program to be optimised, such as memory usage, data throughput or (in GPU programming) the bandwidth utilisation between system and GPU memory.

Even when measuring the running time, the part of the program being tuned may be obscured by the rest. In GPU programming, data must first be copied onto the GPU, along with the kernel to execute. For large problems this time is negligible, but it may dominate small test cases used for tuning, distorting results. If the programmer adds code to time only the relevant part then the results will be more accurate. The programmer must be able to measure and report any figure of merit they choose from within the test program itself.

Finally, it is useful to maximise some figures of merit (data throughput, for example), so there should be a choice between minimisation and maximisation.

Output

To help programmers improve their programs, more output should be shown than simply the optimal parameter valuation. A log of which tests are run and what they scored is helpful.

Testing Accuracy

Timing

When timing parallel programs, the elapsed time between the beginning and end of execution is the most accurate measurement [23, §2]. Using 'CPU time' (the time a process spends executing on the CPU) is misleading as it does not include memory access times or give accurate results in parallel programs. My system uses a 'wall timer' to evaluate tests (giving the total time from beginning to end, as if measured using a wall clock). This means that other running processes can affect the measurements, decreasing the quality of results. The programmer needs to ensure the system is relatively unloaded to get accurate timings.

Repetition

Testing is not an exact process, so it is useful to repeat a test multiple times. The programmer can decide whether running multiple tests or increasing the size of the test case will give better results. There is a balance between these, as it is sometimes important to measure one-time costs such as setting up a problem or retrieving the results.

When using a wall timer, inaccuracies come from other processes running at the same time, whose running time is partially included in the measurement. If a test is repeated, the most accurate measurement will in fact be the minimum, as other processes only increase the measured time [23, §2.1]. Therefore, the default method to aggregate results from repeated tests is to take the minimum. Clearly this is not always appropriate, so there is a choice between minimum, maximum, mean and median for this aggregation. Each of these is useful for different figures of merit and in different situations.

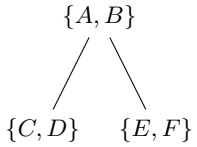
2.3 Variable Independence

In simple examples, the parameters to be optimised in a program can be given as a simple list, for example $\{A, B, C, D, E, F\}$. All variables are treated equally and every possible combination of values is tried in order to find the optimal valuation.

However, in more complex examples—and often in GPU programs—the programmer may know that certain variables are in fact independent from others. For example, A and B may be compiler flags affecting the whole program's execution, whereas C and D control the operation of one parallel loop and E and F control another. Here, C and D could be optimised independently of E and F, even though they all depend on A and B (and conversely their values will all affect the optimal values of A and B). The number of tests required can be reduced by exploiting this independence, making some tests redundant.

The variables can be written in a format describing this independence: $\{A, B, \{C, D\}, \{E, F\}\}$, which says that A and B dominate $\{C, D\}$ and $\{E, F\}$. Specifically, the optimal values of A and B depend on those of all the other variables, but the optimal values for C and D depend only on A and B (and similarly for E and F). This format expresses that C and D are isolated from E and F and that these pairs can be optimised independently—if an optimal valuation is found for C and D at one setting of E and F then it will still be optimal at any other valuation of E and F.

These variable lists can be represented as trees, where each node's variables are independent of its siblings', but depend on the variables of its ancestors and descendants in the tree. The above example would be displayed as follows:



Taking advantage of this type of independence to significantly reduce the number of tests required is one of the most important features of my system. In this example, assuming each variable can take three possible values, the number of tests is reduced from $3^6 = 729$ to only $3^2 \times (3^2 + 3^2 - 1) = 153$.[1] Clearly, this relies on the variables *actually* being independent—the programmer is responsible for making sure their claimed independence holds.

A more realistic CUDA program might have three global options and five parallel loops each governed by three parameters. If each variable can take one of four possible values then there are $4^{18} \approx 6.9 \times 10^{10}$ possible combinations to be tested by a brute-force search. By exploiting independence, there are $3^4 = 81$ possible valuations of the top-level variables. For each of these, there are five groups of $3^4 = 81$ possible tests, giving an overall total of $3^4 \times (5 \times 3^4 - 4) = 32,481$, approximately $2,000,000\times$ less than the number required by brute-force—a huge improvement.

[1] The -1 here is because one of the tests required will already have been run while tuning a previous independent set. Previously run test results are recorded and can be re-used instead of being re-run. This memoization is discussed in Section 3.3.

3

Design

Having described the system's requirements, I now describe its design and development, including overview of the system and a more detailed description of its more complex parts.

3.1 Overview

Problem Breakdown

When designing the system, there were several distinct parts of the problem which could be tackled separately:

- The programmer provides the configuration options for the system which must be read and validated.
- The variable list in the configuration file must be parsed and converted into a variable tree.
- An evaluation function must be defined, which sends commands to the shell in order to compile and run each test, returning its score.
- The optimisation algorithm uses the variable tree, the variables' possible values and the evaluation function to find the optimal valuation.
- The output of the system informs the programmer how the optimisation proceeds and what its results are.

Modules

The system is split into modules, roughly corresponding to the above outline:

`tune.py`
: This is the main program, which sets up and runs the optimisation, showing the programmer how it proceeds. The function evaluate() is defined here, which handles all compilation, test execution and timing.

`tune_conf.py`
: Provides a function which reads the configuration file and performs some simple validation checks and type conversions.

`vartree.py`
: Contains the definition of the *VarTree* data type, which is used throughout the system to represent variable trees. It also defines several utility functions, including the parser vt_parse() and treeprint(), which gives a textual representation of a *VarTree* in tree form.

  This module in particular was designed to be self-contained, making it easy to use throughout the system and to maintain or update. Along with the module `vartree_parser.py` it contains all the functions needed for working with *VarTree* objects.

`vartree_parser.py`
: Provides the *Parser* class, a parser for *VarTree*. This was automatically generated as described in Section 3.2.

`optimisation.py`
: Provides the *Optimisation* class, which contains the optimisation algorithm (described in Section 3.3), parameterised by an evaluation function, a variable tree and a list of possible variable values.

`optimisation_bf.py`
: Provides the *OptimisationBF* class, which is used for testing and mimics *Optimisation* but using brute-force to perform the testing.

`logging.py`
: Allows tuning log files to be created and saved, which can be used to analyse and visualise the testing process.

`testing.py`
: Checks *Optimisation* against *OptimisationBF* for different inputs. If no configuration file is given as input, this test suite is run to demonstrate the operation of the system, as shown in Figure 4.1.

`test_evaluations.py`
: Provides a function *generateEvalFunc*(), which takes a *VarTree* tree as input and generates a sample evaluate() function demonstrating the variable independence given by the tree. This is used for testing and is described in Section 4.1.

3.2 The Variable Tree Mini-Language

To perform optimisation, the programmer creates a configuration file detailing what is to be tested and how the testing is to be performed. In this file, they provide a list of the parameters to be optimised and their possible values. This list is written in a format allowing the programmer to specify any independence between variables. This independence can then be used by the optimiser to cut down the number of valuations which must be checked.

Language Description

The parameters are given in a tree structure, where sibling nodes are independent of each other. This is written in the configuration file using the nested braces notation from Section 2.3, where braces enclose a list of variables and subtrees, with each subtree enclosed by braces.

For example, if A and B are parameters governing the operation of the whole test, such as compiler flags and C, D and E, F control two separate parallel loops in a GPU program, then C, D will be independent of E, F. This independence would be written $\{A, B, \{C, D\}, \{E, F\}\}$, denoting that C, D, E and F are all children of A and B in the parameter tree. This is demonstrated in Figure 3.1.

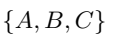
```
>>> vt = vt_parse("{A, B, {C, D}, {E, F}}")
>>> print vt
{A, B, {C, D}, {E, F}}
>>> print treeprint(vt)
      {A, B}
        |
   +----+---+
   |        |
 {C, D}   {E, F}

>>> print vt.flatten()
['A', 'B', 'C', 'D', 'E', 'F']
>>> print vt.vars
['A', 'B']
>>> for st in vt.subtrees: print st
...
{C, D}
{E, F}
>>>
```

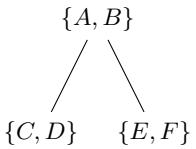
Figure 3.1: The structure of the example *VarTree* $\{A, B, \{C, D\}, \{E, F\}\}$.

An internal *VarTree* node may be empty, signifying that its children are all independent. Finally, it is also valid to simply specify a flat list of variables without braces. Figure 3.2 shows some valid variable lists and the trees they represent. The language's BNF grammar [5] is given in Figure 3.3.

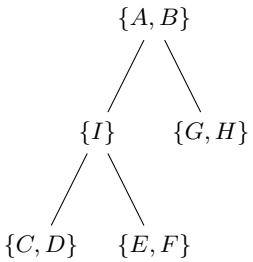
- A, B, C



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



- $\{A, B, \{I, \{C, D\}, \{E, F\}\}, \{G, H\}\}$



- $\{\{A, B\}, \{C, D\}\}$

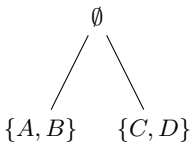


Figure 3.2: Example *VarTree* inputs and their corresponding trees.

```
<vartree>        ::= <vartree_braces> | <flat_list>

<vartree_braces> ::= "{" <list> "}"

<list>           ::= <element> | <element> "," <list>

<element>        ::= <variable> | <vartree_braces>

<flat_list>      ::= <variable> | <variable> "," <flat_list>

<variable>       ::= <char> | <char> <variable>

<char>           ::= <letter_upper> | <letter_lower> | <digit> | <underscore>
```

Figure 3.3: The BNF grammar defining the *VarTree* mini-language.

Parser Generation

I used wisent [18, 21]—a python parser-generator—to create an LR(1) parser [5] based on the mini-language's grammar. Wisent was chosen because it generates self-contained parsers with no dependencies and has features for simplifying the parse tree by removing 'uninteresting' nodes. By carefully constructing the input grammar, it was easy to convert the parse trees to the *VarTree* data type. Because both the grammar and input to be parsed are small, performance was not a concern. Simplicity, in terms of dependencies and parse tree conversion, was more important.

Figure 3.2 shows the grammar file I used, `vartree.wi`, where nonterminal nodes to be omitted from the final parse tree are prefixed by an underscore. The children of these nodes are added directly to the parent node. This makes the conversion to the *VarTree* data type almost trivial, as the parse tree's structure exactly matches the structure of the variable tree described by the input.

The generated parser is included as a dependency by my parsing function, vt_parse(). Once the input has been lexed, it is called to generate the parse tree. Python has a built in module, *re.scanner*, which is used to perform the lexical analysis, converting the input into a string of tokens.

vartree.wi

```
1  VARTREE: LBRACE _LIST RBRACE | _FLATLIST ;
2
3  VARTREE_BR: LBRACE _LIST RBRACE ;
4
5  _LIST: _ELEMENT | _ELEMENT COMMA _LIST ;
6
7  _ELEMENT: VAR | VARTREE_BR ;
8
9  _FLATLIST: VAR | VAR COMMA _FLATLIST ;
```

Figure 3.4: The wisent grammar file used to generate the parser. The first line duplicates the definition of `VARTREE_BR` so there is only one `VARTREE` or `VARTREE_BR` node in the parse tree for each logical *VarTree* subtree.

3.3 The Optimisation Algorithm

I developed a new optimisation algorithm which returns an optimal valuation and exploits variable independence to significantly reduce the search space and therefore the running time compared to brute-force testing.

Algorithm Design

The high-level description of my algorithm is shown here, which recursively optimises all variables in a *VarTree* N:

```
OPTIMISE(N)
 1  if N has Children
 2     then
 3          for each valuation v of the top-level variables of N
 4               do
 5                    Call OPTIMISE recursively on each child in turn.
 6                    The score when all children are optimised is taken as
                        the score for this valuation of N.
 7          return optimal valuation.
 8     else
 9          ▷ N is a leaf node
10          for each valuation v of the top-level variables of N
11               do
12                    Evaluate at this valuation.
13          return optimal valuation.
```

In practice, the recursive call on line 5 requires more thought. To test a valuation requires knowledge of all the variables being tested, including those outside the scope of N or its descendants. The recursive call must include which valuation of the top-level variables is being checked.

The refined algorithm uses a mapping *presets* from variable names to values, which is defined on all variables in the original tree which are *not* in the subtree currently being considered. For the initial call to OPTIMISE(N, *presets*), N is the entire variable tree, N_0, and *presets* is empty. For all recursive calls, it holds that $\text{vars}(N_0) = \text{vars}(N) \cup \text{dom}(\textit{presets})$. This enables the leaves of the recursion to run evaluations of their variables. Some setting is chosen for the variables at the leaf, then the settings of all the other variables are added from *presets* to give a complete valuation, which can be tested. In this way, nodes are optimised, *given values for any variables outside their subtree.*

At an internal node, we wish to find an optimal valuation of all the variables in the subtree, given the valuation of the other variables in *presets*. The child subtrees are independent of each other, meaning each can be optimised while the others are at any arbitrary value. So for a given valuation of the top level variables, the subtrees can simply be optimised separately, one at a time. Then it is simply a case of performing this sub-optimisation for each possible valuation of the top level variables (the sub-optimisations may be different depending on the higher level variables). The valuation of the top level variables (and corresponding optimal valuations of subtrees) which gives the best score when evaluated is chosen and returned.

```
OPTIMISE(N, presets)
 1  ▷ Let 𝕍 be the set of all variables being tuned,
      𝕏 ⊆ 𝕍 be variables disjoint from nodes in N's subtree,
      𝕐 ⊆ 𝕍 be top-level variables in N [𝕐 ≡ vars(N)] and
      ℤ ⊆ 𝕍 be non-top-level variables in N's children.
 2  ▷ Clearly, 𝕏, 𝕐, ℤ are mutually disjoint and 𝕍 = 𝕏 ∪ 𝕐 ∪ ℤ.
 3  ▷ Let 𝕌 be the set of all possible values a variable could take.
 4  ▷ Finally, let 𝕌^𝕍 denote the set of mappings 𝕍 ↦ 𝕌.
 5  ▷ N has type VarTree, and presets ∈ 𝕌^𝕏.
 6  best ← NIL                ▷ best will contain the best valuation found so far.
 7  if children(N) ≠ ∅
 8     then
 9          ▷ N has children, so use recursive optimisation. (ℤ ≠ ∅)
10          for each v ∈ possibleValuations(N)
                                                  ▷ v ∈ 𝕌^𝕐
11               do
12                  ▷ Recursively optimise each child in turn:
13                  ▷ Arbitrary values for child variables, used in recursion.
14                  childvals ← chooseArb(⋃{vars(c) | c ∈ children(N)})
                                                  ▷ childvals ∈ 𝕌^ℤ
15                  for each c ∈ children(N)
16                       do
17                          ▷ Remove this child's variables from childvals.
18                          childvals′ ← childvals↾_{ℤ\vars(c)}
                                                  ▷ childvals′ ∈ 𝕌^{ℤ\vars(c)}
19                          ▷ Get optimal values for this child.
20                          opt ← OPTIMISE(c, presets ⊕ v ⊕ childvals′)
                                                  ▷ opt ∈ 𝕌^𝕍
21                          ▷ Add these optimums back into childvals
22                          childvals ← childvals′ ⊕(opt↾_{vars(c)})
                                                  ▷ childvals ∈ 𝕌^ℤ
23                  ▷ The score for v is tested with all children optimised.
24                  valuation ← presets ⊕ v ⊕ childvals
                                                  ▷ valuation ∈ 𝕌^𝕍
25                  if best = NIL or evaluate(best) ≺ evaluate(valuation)
26                    then
27                         best ← valuation
28     else
29          ▷ N is a leaf node, so test all possibilities. (𝕐 ≠ ∅, ℤ = ∅)
30          for each v ∈ possibleValuations(N)
31               do
32                  valuation ← presets ∪ v
33                  if best = NIL or evaluate(best) ≺ evaluate(valuation)
34                    then
35                         best ← valuation
36  return best
```

Notation

The following functions and operators are used in the algorithm:

vars(N) : returns the set of top-level variables of the *VarTree* N.

children(N) : returns the set of child subtrees of the *VarTree* N.

evaluate(v) : returns the score for a particular valuation, v. The valuation must provide values for *all* variables being tuned (i.e. $v \in \mathbb{U}^{\mathbb{V}}$), not only those in the current subtree.

$\prec$ (comparison) : compares two scores, will be the 'less than' or 'greater than' operator if the optimisation is minimising or maximising, respectively.

possibleValuations(N) : the set of all possible valuations of vars(N). The size of possibleValuations(N) is exponential in the number of N's top-level variables, but it can be generated on the fly.

chooseArb(C) : given a set of variables, chooses a possible valuation of them arbitrarily (chooseArb(C) $\in \mathbb{U}^C$).

$\oplus$ (addition of mappings) : Combines two disjoint mappings. If $a \in \mathbb{U}^A$ and $b \in \mathbb{U}^B$ with $A \cap B = \emptyset$, then $a \oplus b \in \mathbb{U}^{A \cup B}$ where:

$$(a \oplus b)(x) = \begin{cases} a(x) & \text{if } x \in A \\ b(x) & \text{if } x \in B \end{cases}$$

$\restriction$ (restriction) : restricts a mapping to a particular subset of its domain. If $a \in \mathbb{U}^A$, then $a\restriction_B \in \mathbb{U}^{A \setminus B}$ where:

$$(a\restriction_B)(x) = \begin{cases} \text{undefined} & \text{if } x \in B \\ a(x) & \text{if } x \notin B \end{cases}$$

Assumptions

The following assumptions are made about the input to Optimise:

- The variable tree N should be well formed:
  - N is not completely empty
  - None of the leaves of N are empty
  - No variable appears more than once in the tree

  These properties are ensured by the validation checks in `tune_conf.py`.
- The function evaluate() is memoizing, discussed below.
- Empty nodes are handled correctly, discussed below.

Empty Tree Nodes

A problem arises at line 10 if a node in the tree has no top-level variables. If $vars(N) = \emptyset$, then possibleValuations$(N) = \emptyset$ and the loop body will never run.

To address this, the variable tree can be easily transformed beforehand into an equivalent tree where every node has at least one variable. For each empty node N in the tree, a fresh variable x_N (with only a single possible value) is added to that node. Hence, the loop on line 10 is guaranteed to execute at least once, allowing the subtrees to be optimised and without affecting the optimisation.

Continual Optimisation

Initially, the valuations for child subtrees are chosen arbitrarily, but as optimal valuations are found, they are included in *childvals* for further recursive calls (on line 22). This means that optimisations found so far in the search can immediately be used to speed up later tests. This does not affect the results, as the subtrees are all independent. The speed improvement can be seen clearly in Figures 4.9 and 4.10 as the algorithm progresses.

Memoization

To keep the algorithm clear and simple, some calls to evaluate are repeated. For example, on line 25 the current top-level valuation is tested with a call to evaluate(*valuation*), but this valuation will already have been tested by the final recursive call.

Instead of incorporating this into the algorithm, making it considerably more complex, the evaluate() function is memoized. Internally, evaluate() will keep a mapping between valuations and scores which can be used to return quickly on repeated input.

Additionally, the score of a particular test will not change between calls to evaluate—a very desirable property. Memoization guarantees that each valuation will be tested at most once and generate at most one score, which may then be used multiple times.

Hence, the optimisation algorithm assumes the following properties of the evaluation function:

- It is deterministic. Multiple calls to evaluate with the same input will return the same result. This is required for correctness of the algorithm.
- The cost of repeat calls to evaluate is low. Tests are not re-run, improving the performance of the algorithm.

These properties are guaranteed by a wrapper to the evaluate() function provided by the *Optimisation* class. The function passed to the *Optimisation* object may assume that it will only be called once per valuation and need not ensure these properties itself.

As the algorithm can run exponentially many tests, the memoization table may require exponential space. If this were a problem for large instances, the table could be replaced with a cache, only retaining the most recent calls.

An Example Execution

Figure 3.6 shows how the algorithm would run when optimising a small example program. The variable tree is $\{OPT, \{K1\}, \{K2A, K2B\}\}$, where OPT controls some global setting, possibly a compiler optimisation and $K1$ and $K2A/B$ control two independent GPU kernels, possibly the vector addition followed by multiplication from Figure 1.6. $K2A$ and $K2B$ control the same kernel, so they depend on each other, but are independent of $K1$. The scores for each possible valuation are given in Figure 3.5.

OPT	K1	K2A	K2B	Score
-O2	128	128	4	14
-O2	128	128	6	15
-O2	128	256	4	12
-O2	128	256	6	17
-O2	256	128	4	16
-O2	256	128	6	17
-O2	256	256	4	14
-O2	256	256	6	19
-O3	128	128	4	13
-O3	128	128	6	14
-O3	128	256	4	11
-O3	128	256	6	16
-O3	256	128	4	11
-O3	256	128	6	12
-O3	256	256	4	9
-O3	256	256	6	14

Figure 3.5: The possible values and test scores used in the example. Using brute-force testing, each of these 16 possibilities would be tested.

Comment		Tests			
	OPT	**K1**	**K2a**	**K2b**	**Score**
Initial Call:					
OPTIMISE($\{OPT, \{K1\}, \{K2A, K2B\}\}, \emptyset$)					
Optimising $\{OPT\}$, Try '-O2'.					
Chose arbitrary values for children.					
Recursively optimise children:					
OPTIMISE($\{K1\}, \{OPT \mapsto$ '-O2',					
$K2A \mapsto$ '128', $K2B \mapsto$ '4'$\}$)					
	-O2	**128**	128	4	14
	-O2	**256**	128	4	16
When $OPT =$ '-O2', $K1 =$ '128' optimal.					
OPTIMISE($\{K2A, K2B\}, \{OPT \mapsto$ '-O2',					
$K1 \mapsto$ '128'$\}$)					
	-O2	128	**128**	**4**	(14)
	-O2	128	**128**	**6**	15
	-O2	128	**256**	**4**	12
	-O2	128	**256**	**6**	17
When $OPT =$ '-O2', $K2A =$ '256' and					
$K2B =$ '4' optimal.					
The best score for $OPT =$ '-O2' is 12.					
Optimising $\{OPT\}$, Try '-O3'.					
Chose arbitrary values for children.					
Recursively optimise children:					
OPTIMISE($\{K1\}, \{OPT \mapsto$ '-O3',					
$K2A \mapsto$ '128', $K2B \mapsto$ '4'$\}$)					
	-O3	**128**	128	4	13
	-O3	**256**	128	4	11
When $OPT =$ '-O3', $K1 =$ '256' optimal.					
OPTIMISE($\{K2A, K2B\}, \{OPT \mapsto$ '-O2',					
$K1 \mapsto$ '256'$\}$)					
	-O3	256	**128**	**4**	(11)
	-O3	256	**128**	**6**	12
	-O3	256	**256**	**4**	9
	-O3	256	**256**	**6**	14
When $OPT =$ '-O3', $K2A =$ '256' and					
$K2B =$ '4' optimal.					
The best score for $OPT =$ '-O3' is 9.					
This is better than for '-O2'.					
Therefore return the following valuation:					
$\{OPT \mapsto$ '-O3', $K1 \mapsto$ '256',					
$K2A \mapsto$ '256', $K2B \mapsto$ '4'$\}$					
With a minimal score of 9.					

Figure 3.6: An example execution of the optimisation algorithm. Scores marked with brackets would have been memoized and would not be run again. Even in this tiny example, 10 tests are required, compared to 16 for brute-force.

Correctness

Theorem 3.3.1 shows that the new optimisation algorithm is correct, i.e. that it is guaranteed to find an optimal valuation despite ignoring many possible tests.

Theorem 3.3.1 (Correctness of Optimise)**.** *The* Optimise *algorithm returns an optimal parameter valuation, assuming:*

- *The given independence holds*
- *Tests (using* evaluate*()) do not fail and give correct results*
- *The variable tree provided is valid*

Proof. By induction on the structure of a *VarTree* node N.

Inductive Hypothesis (I.H.): Optimise(N) returns a valuation which is optimal for the variables in N's tree, given a valuation of any variables outside the scope of N.

Base Case: N is a leaf node.

> The algorithm tests every possible valuation of the variables of N and returns an optimal one.

Induction Step: N is an internal node.

> For each possible valuation of the top-level variables, the algorithm calculates an optimal setting of all descendant variables for that valuation (by Lemma 3.3.2, which, by the I.H., may assume the correctness of Optimise on the subtrees of N).
>
> The valuation which is best when evaluated along with these 'descendant optimums' must be the optimal valuation of the top-level variables.
>
> Therefore, that valuation and its corresponding 'descendant optimums' are optimal for N's subtree and are returned. □

Lemma 3.3.2. *Given a particular valuation of the top-level variables in N, the optimal setting (for that particular valuation) of all descendant variables is found, under the same assumptions as Theorem 3.3.1 and assuming the correctness of* Optimise *on subtrees of N.*

Proof. For each subtree X, the optimal valuation of X's variables is independent of the setting of any other subtree's variables. Similarly, X's setting does not affect any other subtree's optimal valuation.

Therefore, a call to Optimise(X) (with an arbitrary setting of sibling variables) will return an optimal setting of X's variables, given the valuation of N's top level variables.

Optimise(X) is called on all subtrees X, producing optimal valuations of each subtree (at the current top-level valuation). Taking all of these independent subtree valuations together gives an optimal valuation of all descendant variables, as required. □

Complexity

The new algorithm requires significantly less tests than brute-force tuning, assuming there is enough independence between variables. In some cases, such as when all variables are in a single node, the algorithm will clearly not provide any improvement. However, a more interesting upper bound for the number of tests required is given here.

Assume the variable tree is a complete k-ary tree of height h, with each node having a single variable with p possibilities.

For a brute-force approach, there are

$$n = \frac{k^{h+1} - 1}{k - 1}$$

nodes in the tree [7] and p^n possible combinations.

In my algorithm, a leaf node will require p tests and an internal node, for each of the p possibilities, a recursive call is made to each of the k children. Hence, the number of tests, $T(h)$, (in terms of tree height) is given by:

$$\begin{aligned} T(0) &= p \\ T(h) &= p \cdot k \cdot T(h-1) \end{aligned}$$

expanding this definition gives:

$$\begin{aligned} T(h) &= p \cdot (pk)^h \\ &= p \cdot (pk)^{\lfloor \log_k(n) \rfloor} \end{aligned}$$

So where the brute-force algorithm is exponential in the number of nodes, my algorithm is exponential in the height of the tree.

In reality, it is unlikely that such a tree would be used. For an arbitrary variable tree, take h to be its height, k to be the maximum number of children at any node and p to be the maximum number of distinct valuations of any single node. Multiple variables at a single node simply result in more possibilities. Now, $p(pk)^h$ is an upper bound for the number of tests required. Although p can of course be exponentially large, this shows that the largest improvements will be found when the number of possibilities in a single node is reduced.

3.4 Alternative Algorithm Designs

A Polynomial-Time Algorithm

Finding a polynomial-time optimisation algorithm would allow much more detailed tuning to be feasibly performed. However, unless $P = NP$, no such algorithm exists. As there are no restrictions on the running time of evaluate(), it only makes sense to consider an algorithm which would make a polynomial number of calls to evaluate().

Theorem 3.4.1. *There is no Parameter Optimisation (PO) algorithm making a polynomial number of calls to* evaluate*(), assuming $P \neq NP$.*

Proof. Lemma 3.4.2 shows that $SAT \leq_p PO$. Suppose, for a contradiction, that there is an algorithm solving PO with a polynomial number of calls to evaluate(). In cases where evaluate() runs in polynomial time, this algorithm itself will run in polynomial time. The evaluate() function used in Lemma 3.4.2's reduction requires only linear time, so the reduction's invocation of the PO algorithm will only require polynomial time. Therefore, the reduction demonstrates a polynomial-time algorithm for the boolean satisfiability problem (SAT), an NP-Complete problem, violating the assumption that $P \neq NP$. □

Lemma 3.4.2. *Boolean Satisfiability is polynomial-time reducible to Parameter Optimisation.*

Proof. Given a formula φ, set up the optimisation as follows:

- The parameters to be optimised are exactly the variables in φ.
- The possible parameter values are True and False for each parameter.
- The evaluate() function takes a valuation and returns 1 if and only if that valuation satisfies the SAT instance, returning 0 otherwise.
- The algorithm should maximise the score.

Now, the optimisation will return a valuation maximising the score given by evaluate(). If this valuation satisfies φ then clearly it is satisfiable. Otherwise, the valuation's score when passed to evaluate() must have been 0. The optimiser found the maximal valuation, so there cannot be any valuation returning 1 and therefore cannot be any satisfying assignment.

Hence, φ is satisfiable if and only if the valuation returned by the optimisation is a satisfying assignment.

The reduction is clearly polynomial-time—the problem translation is simple and checking if the returned valuation is a satisfying assignment requires linear time. □

An Approximation Algorithm

Some optimisation problems can be approximated within a constant factor in polynomial time [19]. These ε-approximation algorithms guarantee to return a result which is within a factor of ε of the optimal solution. This would be a good compromise for my system, as finding a perfectly optimal solution is not critical.

Theorem 3.4.3. *If $P \neq NP$ then there is no polynomial time ε-approximation algorithm for PO.*

Proof. Suppose, for a contradiction, that a polynomial-time ε-approximation algorithm for PO does exist. PO is a maximisation problem, so the algorithm would guarantee to find a solution which is not less than $1/\varepsilon$ of the optimal value.

Given a formula φ, construct the optimisation given in Lemma 3.4.2. The result of this (approximated) optimisation will be a valuation v. By construction, evaluate$(v) = 1$ or 0. Let the optimal valuation be v_{opt}.

If the formula is satisfiable then evaluate$(v_{opt}) = 1$. Hence any valuation returned by the ε-approximation algorithm (which must have a score of at least $1/\varepsilon$) must in fact have a score of exactly 1.

If the formula is unsatisfiable then evaluate$(v_{opt}) = 0$ and therefore the ε-approximation algorithm must return a valuation with a score of 0 (as all possible valuations have a score of 0).

Therefore, the SAT instance is satisfiable if and only if the valuation returned by the approximation algorithm is a satisfying assignment. This procedure only requires polynomial time, violating the assumption that $P \neq NP$ (as SAT is NP-Complete). □

As before, this reduction uses a linear-time evaluate() function. This shows that there can be no ε-approximation algorithm using only a polynomial number of tests, because the algorithm as a whole would be polynomial time and Theorem 3.4.3 would still apply.

4

Testing

Testing was divided into four main phases. During development I used synthetic examples to test my implementation of the optimisation algorithm (Section 4.1). Secondly, I tested the system on some small GPU programs which had previously been hand-optimised, to make sure it was working correctly and behaving as expected (Section 4.2). The system was used on a variety of hardware to optimise an example simulation distributed with OP2 [11] (Section 4.3). This program is a 2D airfoil simulation using the OP2 API, so it runs in parallel on GPUs using CUDA and on single- or multi-CPU systems using OpenMP [3]. These results were presented at the *Many-Core and Reconfigurable Supercomputing Conference 2011* [13] and are to be included in a forthcoming paper for the *Journal of Parallel and Distributed Computing* [12]. Finally, I performed a more in-depth auto-tuning of the airfoil simulation, testing the benefit auto-tuning might have in actual program development (Section 4.4).

4.1 Development

During the system's development, I created some testing tools. If the system is run with no configuration file, the tests are used to demonstrate its operation and its improvements over brute-force optimisation (Figure 4.1).

Sample Test Generation

The module `test_evaluations.py` defines generateEvalFunc(), which generates sample tests. A test is represented by a function, evaluate(), which takes a parameter valuation and returns its score. In practice, evaluate() compiles and executes the program being optimised and returns the running time. For testing and debugging, example evaluation functions which did not require this were useful. Although the results returned are meaningless, these sample functions still exhibit variable independence. The optimisation algorithm could be tested in controlled conditions, with none of the uncertainties of running actual tests.

generateEvalFunc() uses a *VarTree* object to define the variable independence of the returned evaluate() function. The function calculates dependent sets of variables from the tree and hashes each set. These hashes are summed and returned. Therefore, an optimal value for a variable will be optimal at any setting of any independent variables, as required.

Variables are dependent if they are in the same *VarTree* node or if they are ancestors or descendants. Variables in sibling subtrees, or siblings of ancestors are independent. The sets of dependent variables can be described by $\{\bigcup\{\text{vars}(a) \mid a \in \text{ancestors}(x)\} \cup \text{vars}(x) \mid x \in \mathit{leaves}(vt)\}$: there is one set for each leaf in the tree, with each set containing all the variables on the path from that leaf to the root.

For example, in the *VarTree* $\{A, B, \{I, \{C, D\}, \{E, F\}\}, \{G, H\}\}$, the interdependent sets are $\{A, B, I, C, D\}$, $\{A, B, I, E, F\}$ and $\{A, B, G, H\}$. Calling generateEvalFunc() for this tree would return a function taking a valuation v and returning the following sum:

$$\begin{aligned} &\textsc{Hash}(v(A), v(B), v(I), v(C), v(D)) \\ +&\textsc{Hash}(v(A), v(B), v(I), v(E), v(F)) \\ +&\textsc{Hash}(v(A), v(B), v(G), v(H)) \end{aligned}$$

Results

In the final system, these sample tests are only used for the demonstration shown in Figure 4.1. However, they were very useful during development for comparing my optimisation algorithm to the brute-force algorithm and for debugging.

```
$ ./tune.py

                                Autotuning System
                                      v0.11

Usage: Please provide the path to a configuration file as an argument.
When no arguments are provided, some sample tests are run.
Press Enter to run the tests.

Possible Values of Variables:
(we use the same ones for all examples)
A = [3, 2, 1]
B = [3, 5, 7]
C = [2, 3]
D = [10, 5]
E = [4, 2]
F = [1, 2]
G = [9, 4]
H = [11, 2]
I = [6, 4, 2]


                                    Example 1


Syntax:
{A, B, {C, D}, {E, F}}

      {A, B}
        |
   +----+---+
   |        |
 {C, D}   {E, F}


Minimal Valuation (by brute force):
A = 2, B = 3, C = 2, D = 5, E = 2, F = 2
The score is 856, found in 144 evaluations.

Maximal Valuation (by brute force):
A = 1, B = 3, C = 2, D = 10, E = 2, F = 1
The score is 8987, found in 144 evaluations.

Minimal Valuation (by observing independence):
A = 2, B = 3, C = 2, D = 5, E = 2, F = 2
The score is 856, found in 63 evaluations.

Maximal Valuation (by observing independence):
A = 1, B = 3, C = 2, D = 10, E = 2, F = 1
The score is 8987, found in 63 evaluations.

These results are equal.
The new algorithm performed 44% of the number of tests required by brute-force.


                                    Example 2


Syntax:
{A, B, {I, {C, D}, {E, F}}, {G, H}}

          {A, B}
            |
        +---+--------+
        |            |
       {I}         {G, H}
        |
   +----+---+
   |        |
 {C, D}   {E, F}
```

```
Minimal Valuation (by brute force):
A = 2, B = 5, C = 2, D = 5, E = 2, F = 2, G = 9, H = 11, I = 2
The score is 463, found in 1728 evaluations.

Maximal Valuation (by brute force):
A = 3, B = 5, C = 3, D = 10, E = 4, F = 2, G = 9, H = 2, I = 6
The score is 14028, found in 1728 evaluations.

Minimal Valuation (by observing independence):
A = 2, B = 5, C = 2, D = 5, E = 2, F = 2, G = 9, H = 11, I = 2
The score is 463, found in 216 evaluations.

Maximal Valuation (by observing independence):
A = 3, B = 5, C = 3, D = 10, E = 4, F = 2, G = 9, H = 2, I = 6
The score is 14028, found in 216 evaluations.

These results are equal.
The new algorithm performed 44% of the number of tests required by brute-force.
```

Figure 4.1: The automated demonstration of the system's operation.

4.2 Initial Testing

I set up two optimisations of programs with known optimums to test my system and demonstrate it working with some small examples.

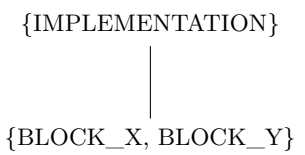
Strategy

The test programs, `laplace3d` and `adi3d`, were written by Prof. Mike Giles and used as examples on his CUDA programming course [10]. They both solve a 3D Laplace equation, by Jacobi iteration and the ADI (Alternating Direction Implicit) method respectively.

Each program is parameterised by constants BLOCK_X and BLOCK_Y, which control the block size. There are also two versions of each program—a naive implementation and a careful implementation with good memory coalescence, making good use of the GPU's memory bandwidth. My tuning used the parameters BLOCK_X, BLOCK_Y and IMPLEMENTATION, which controlled whether the naive or coalescing GPU code was tested. Figure 4.2 shows the possible values for each parameter, these values give block sizes which are multiples of 32 and are mostly below the hardware limit of 512 threads per block of the test GPU.

For variable independence, I chose to put IMPLEMENTATION at the root of the tree, with BLOCK_X and BLOCK_Y as a subtree:

$$\{\text{IMPLEMENTATION}, \{\text{BLOCK\_X}, \text{BLOCK\_Y}\}\}$$



laplace3d.conf

```
37  variables = {IMPLEMENTATION, {BLOCK_X, BLOCK_Y}}
```

laplace3d.conf

```
56  BLOCK_X = 16, 32, 48, 64, 96, 128
57
58  BLOCK_Y = 2, 4, 6, 8
59
60  IMPLEMENTATION = laplace3d.cu, laplace3d_naive.cu
```

Figure 4.2: The configuration file settings for the `laplace3d` test. The `adi3d` test used equivalent settings.

laplace3d.cu

```
15  #ifndef BLOCK_X
16      #define BLOCK_X 32
17  #endif
18  #ifndef BLOCK_Y
19      #define BLOCK_Y 4
20  #endif
```

Figure 4.3: The test code was edited so the parameters BLOCK_X and BLOCK_Y could be set by the compiler, via the `-D` command-line option.

This is clearly equivalent to a flat list (all combinations will still be tested), but it seemed logical as IMPLEMENTATION controls the entire program, whereas BLOCK_X and BLOCK_Y control only a single CUDA kernel.

I edited the source files for each program as shown in Figure 4.3 so that the parameters BLOCK_X and BLOCK_Y could be set by the compiler instead of the original `#define` statement.

The testing was performed on an unloaded machine with a GeForce 8800 GTX graphics card. Each test was repeated five times, taking the minimum as the overall score.

Results

The results of each run are shown in Figures 4.4 and 4.5 and testing process is shown in detail in Figures 4.6 and 4.7. The scores for each test are marked with crosses and the overall score is shown as a bar. Missing bars correspond to tests which did not run due to the hardware limitations of the test GPU. Each parameter's values are plotted below, showing any correlation between parameter values and the resulting score.

The careful implementation performed better in both cases and the performance difference in the `laplace3d` test was particularly distinct. Otherwise, there was no clear correlation between the running time and the parameter values.

```
Minimal valuation:
BLOCK_X = 32, BLOCK_Y = 6, IMPLEMENTATION = laplace3d.cu
Minimal Score:
2.93070292473
The system ran 48 tests.
```

Figure 4.4: The final lines of output from the `laplace3d` testing.

```
Minimal valuation:
BLOCK_X = 64, BLOCK_Y = 2, IMPLEMENTATION = adi3d.cu
Minimal Score:
6.27469491959
The system ran 48 tests.
```

Figure 4.5: The final lines of output from the `adi3d` testing.

The system returned optimal values which were slightly better than the values previously chosen by hand. The `laplace3d` optimisation found the best valuation to be BLOCK_X = 32 and BLOCK_Y = 6, with a minimum running time of $2.931s$. Previously, BLOCK_X = 32, BLOCK_Y = 4 was used, which had a running time of $3.061s$. The `adi3d` test was similar, with the same original values giving a score of $6.300s$ and the auto-tuned optimum of BLOCK_X = 64 and BLOCK_Y = 2 giving an improved score of $6.275s$.

Analysis

Although these improvements are small, they are exactly the type of saving the system is designed to find—when scaled up to a much larger problem the reduction in running time could vastly outweigh the 20 minutes spent on testing. This scaling is investigated in Section 4.4. Also, only slight improvement can be expected from these examples as they have already been tuned by hand. The auto-tuning was not performed on the same hardware as the hand-tuning so possibly 32×4 *was* in fact optimal, but it is not quite as good on the test PC.

These tests also demonstrated the saving in development time. The manual tuning took over an hour for each program and was not as thorough as this auto-tuning.

The GeForce 8800 GTX GPU used for testing has a limit of 512 threads per block. This meant that some tests (with block sizes over 512 threads) failed to run. This was not a problem—discarding valuations which cannot be executed on a particular system is part of the tuning process.

Conclusion

The results of this testing were as expected. The performance improvements were very small compared to the hand-optimisations and there was no clear correlation between BLOCK_X, BLOCK_Y and the corresponding score, although there *was* significant variation in score.

In the `laplace3d` test, the variation in scores was much greater for the careful implementation than for the naive implementation. Block size choice is clearly more important with this version. The same effect was not seen in the `adi3d` code—the score variation was similar between the two versions.

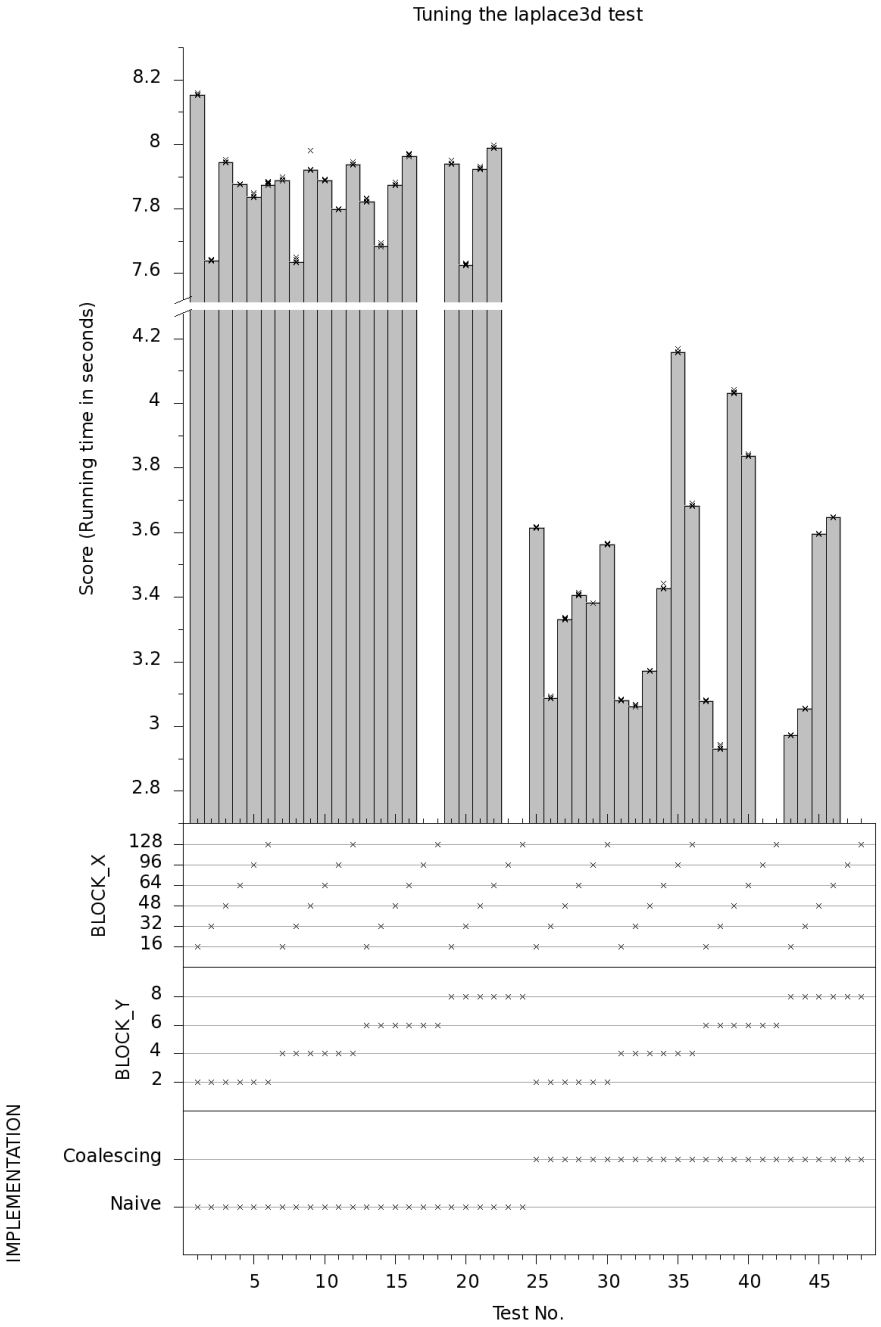


Figure 4.6: The `laplace3d` test results. The y-axis has been split to more clearly show the two separate groups of results.

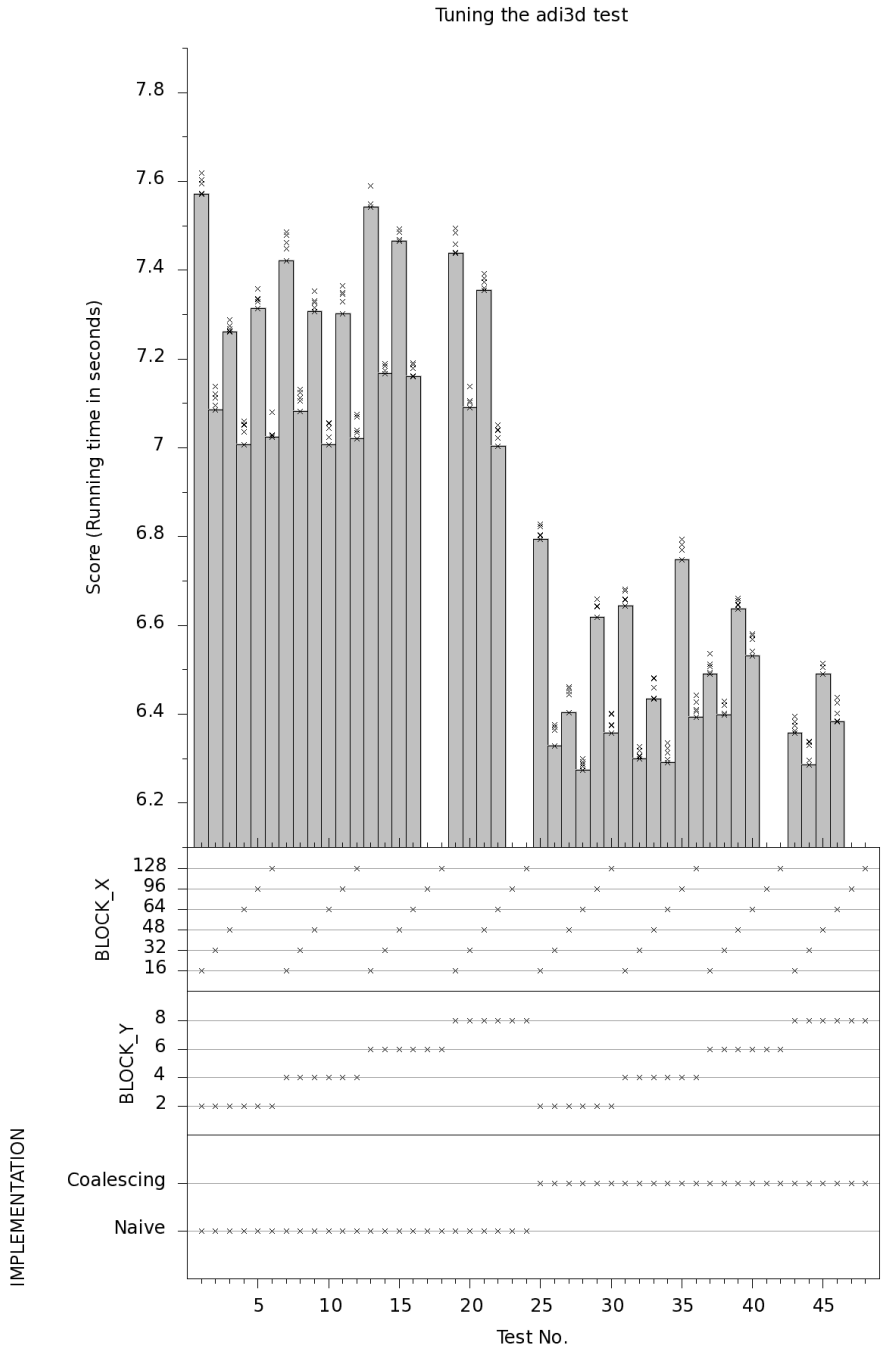


Figure 4.7: The `adi3d` test results.

4.3 Hardware Tuning

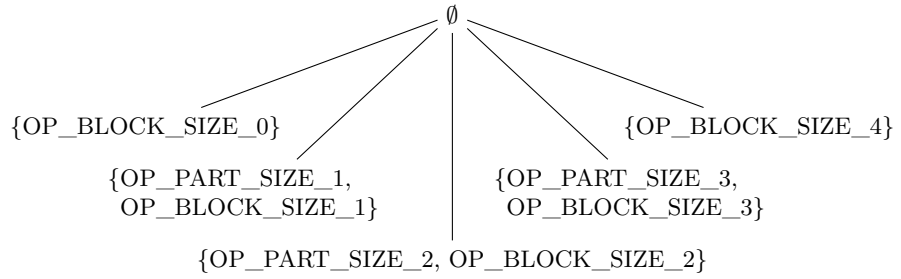
I collaborated with Dr. Gihan Mudalige, a researcher at OeRC, to test the parameter settings required for best performance across different hardware architectures. These results were presented at the *Many-Core and Reconfigurable Supercomputing Conference 2011* [13] and are being prepared for inclusion in a paper for the *Journal of Parallel and Distributed Computing* [12]. Here, I describe how the tuning was performed and explain the results obtained.

Strategy

The test program was a simple CFD simulation of an airfoil, distributed with OP2 to demonstrate its capabilities. The OP2 API can translate this simulation into CUDA GPU code or CPU code using OpenMP.

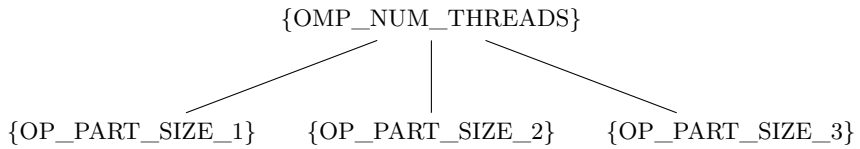
The airfoil simulation is parameterised by three parameters controlling the partition size of the simulation and and five controlling the GPU block size of different parallel loop calls. The first loop, `save_soln`, is parameterised by OP_BLOCK_SIZE_0; the second, `adt_calc`, by OP_PART_SIZE_1 and OP_BLOCK_SIZE_1; the third, `res_calc`, by OP_PART_SIZE_2 and OP_BLOCK_SIZE_2; the fourth, `bres_calc`, by OP_PART_SIZE_3 and OP_BLOCK_SIZE_3; and the final loop, `update`, by the parameter OP_BLOCK_SIZE_4. Because these parameters control different parallel loops, they are independent and can be optimised independently. There were no parameters being tuned which affected the whole program, so the variable independence was described as follows:

$$\begin{aligned}&\{\{OP\_BLOCK\_SIZE\_0\},\\ &\ \{OP\_PART\_SIZE\_1, OP\_BLOCK\_SIZE\_1\},\\ &\ \{OP\_PART\_SIZE\_2, OP\_BLOCK\_SIZE\_2\},\\ &\ \{OP\_PART\_SIZE\_3, OP\_BLOCK\_SIZE\_3\},\\ &\ \{OP\_BLOCK\_SIZE\_4\}\}\end{aligned}$$



For the CPU tests, a new global parameter, OMP_NUM_THREADS, controlling the number of OpenMP threads was introduced:

{OMP_NUM_THREADS, {OP_PART_SIZE_1},
{OP_PART_SIZE_2},
{OP_PART_SIZE_3}}



The following possible parameter values were used:

OP_BLOCK_SIZE_0 :	64, 128, 256, 512, 1024	(all GPUs)
OP_BLOCK_SIZE_1 :	64, 128, 256, 512, 1024	(all GPUs)
OP_BLOCK_SIZE_2 :	64, 128, 256, 512, 1024	(all GPUs)
OP_BLOCK_SIZE_3 :	64, 128, 256, 512, 1024	(all GPUs)
OP_BLOCK_SIZE_4 :	64, 128, 256, 512, 1024	(all GPUs)
OP_PART_SIZE_1 :	64, 128, 256, 512, 1024	
OP_PART_SIZE_2 :	64, 128, 256, 512, 1024	
OP_PART_SIZE_3 :	64, 128, 256, 512, 1024	
OMP_NUM_THREADS :	4, 8, 16	(Nehalem CPU)
OMP_NUM_THREADS :	6, 8, 12, 16, 24	(Westmere CPU)
OMP_NUM_THREADS :	2, 4	(Sandy-Bridge CPU)

The following hardware was used for tuning, with each platform being tuned once using single precision arithmetic and again using double precision.

NVIDIA Tesla C2070, a high-performance GPGPU card.

NVIDIA GTX460, a consumer GPU.

NVIDIA GTX560 Ti, a consumer GPU.

Intel Xeon CPU X5650, 12 cores @ 2.67GHz, a Westmere CPU.

Intel Xeon CPU E5540, 16 cores @ 2.53GHz, a Nehalem CPU.

Intel Core i5 2500K, 4 cores @ 3.3GHz, a Sandy-Bridge CPU.

The figure of merit used for this testing was the execution time of the simulation part of the program, excluding the problem setup and copying data to the GPU.

Results

Graphs of the testing results are shown at the end of this section. As before, the main part shows how the running time varied as the parameter values in the lower part changed. This allows large speed changes to be correlated with parameter changes.

The data presented in this section was collected by Dr. Gihan Mudalige using my auto-tuning system. The analysis and graphs are my own.

Analysis

The testing results were very interesting, showing some variations between hardware and some consistent patterns.

The clearest demonstration of the optimisation process was the Tesla C2070 tuning (Figures 4.9 and 4.10). These runs have very similar performance characteristics, although slightly different optimal values. Both show a large oscillation during tests 30–53, while OP_BLOCK_SIZE_2 and OP_PART_SIZE_2 were varied. There is no noticeable change as OP_BLOCK_SIZE_2 changes, so almost all the speed-up came from choosing a good OP_PART_SIZE_2 value (512 in this case). OP_PART_SIZE_1 caused a similar, smaller oscillation, showing that this parameter also has a noticeable, predictable effect.

The other GPU tests (Figures 4.11, 4.12, 4.13 and 4.14) responded similarly, with a lot of variation while OP_PART_SIZE_2 varied and more modest variation from varying OP_PART_SIZE_1. For the consumer GPUs, the initial scores were fairly good, so the oscillations are shown as large *increases* in running time when bad choices are made, rather than the large speed-up as good values were chosen in the Tesla C2070 testing.

The CPU tests (Figures 4.15, 4.16, 4.17, 4.18, 4.19 and 4.20) showed very different performance characteristics. The most influential parameter during these tests was OMP_NUM_THREADS—most of the graphs show noticeable 'steps' as the value of OMP_NUM_THREADS was changed, with similar performance variation within each step as the other variables are tuned. For example, in the Sandy-Bridge tuning (Figures 4.19 and 4.20), there is some speed-up from OP_PART_SIZE_2, but the main speed-up comes from the choice of OMP_NUM_THREADS (4 in this case). Within each choice of OMP_NUM_THREADS, tuning OP_PART_SIZE_2 provides a consistent improvement and the other parameters don't cause much noticeable effect.

The parallel loops `adt_calc` (parameterised by OP_PART_SIZE_1 and OP_BLOCK_SIZE_1) and `res_calc` (OP_PART_SIZE_2 and OP_BLOCK_SIZE_2) comprise the majority of the simulation's running time (around 29% and 52% respectively), so it is not surprising that the most speed-up can be gained in these loops. Even considering this, these loops seem to vary more than the others.

Interestingly, the partition size parameters provided all the performance benefit; the block size did not make a significant difference.

Test	Tests	BF	Score (s)	Prev (s)
Tesla C2070 (DP)	81	390,625	17.69	16.72
Tesla C2070 (SP)	81	390,625	7.93	7.55
GeForce GTX 460 (DP)	81	390,625	27.01	
GeForce GTX 460 (SP)	81	390,625	10.41	
GeForce GTX 560 Ti (DP)	81	390,625	19.65	22.44
GeForce GTX 560 Ti (SP)	81	390,625	7.82	8.15
Nehalem Xeon E5540 (DP)	39	375	54.00	
Nehalem Xeon E5540 (DP)	39	375	39.08	
Westmere Xeon X5650 (DP)	65	625	45.29	
Westmere Xeon X5650 (SP)	65	625	33.71	
Sandy-Bridge Core i5 (DP)	26	250	63.60	
Sandy-Bridge Core i5 (DP)	26	250	48.63	

Figure 4.8: An overview of the tuning results. The number of tests which would have been required by brute-force for the same optimisation is given for comparison (BF). Also included (where known) is the simulation's running time using the previous parameter settings of OP_PART_SIZE = OP_BLOCK_SIZE = 256 (Prev). In fact, this is a very good choice, giving almost optimal values. Running time cannot be measured exactly, so there is some variation in results, especially those measured at different times. The 'default' results which appear lower than the auto-tuned results are due to this variation.

Conclusion

Auto-tuning is important for frameworks such as OP2—parallel programming libraries aiming to run a large variety of problems on a variety of hardware. Good performance is hardware and implementation dependent for this type of problem, so being able to tune the library parameters depending on the hardware being used and the problem being solved is useful.

These results also demonstrate the potential for discovering performance-critical parts of a program. The tuning logs show which parameters affect the score and which have little impact. This guides the programmer to focus only on important optimisations or to add extra parameterisation to important areas for finer-level tuning.

Clearly, a programmer without an in-depth knowledge of the underlying hardware cannot be expected to accurately predict a good block size. When running the airfoil simulation on a GPU, choosing OP_PART_SIZE_2 is critical. In the Tesla C2070 test for example, only assuming that multiples of 32 are good values is not enough—64 gives a score around 55 seconds but 512 gives a much better score of just under 18 seconds (with all other parameters being constant). Different multiples can give large variations in performance, so even this knowledge is not sufficient to making a good choice.

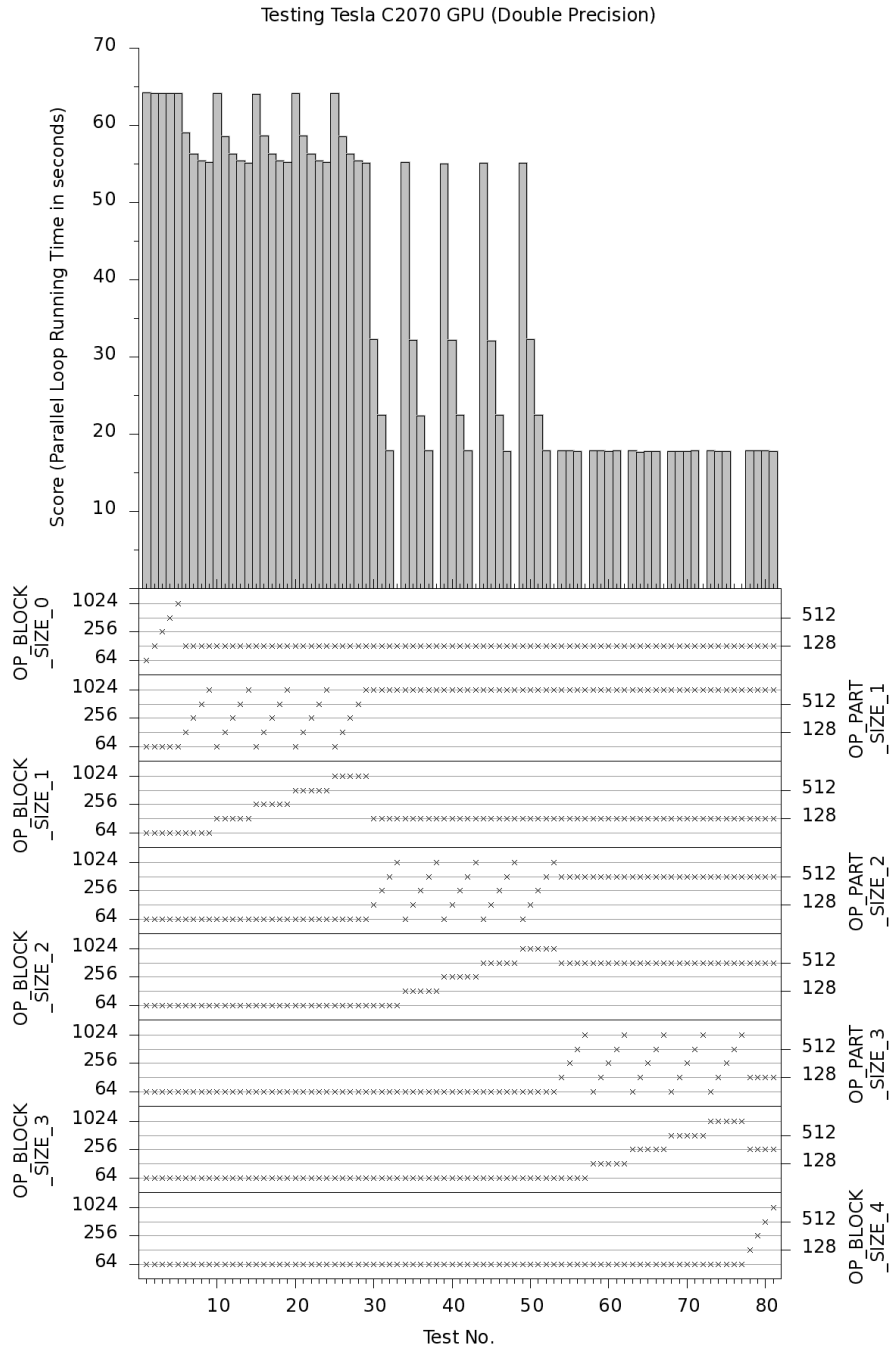


Figure 4.9: Airfoil tuning on a Tesla C2070, Double Precision.

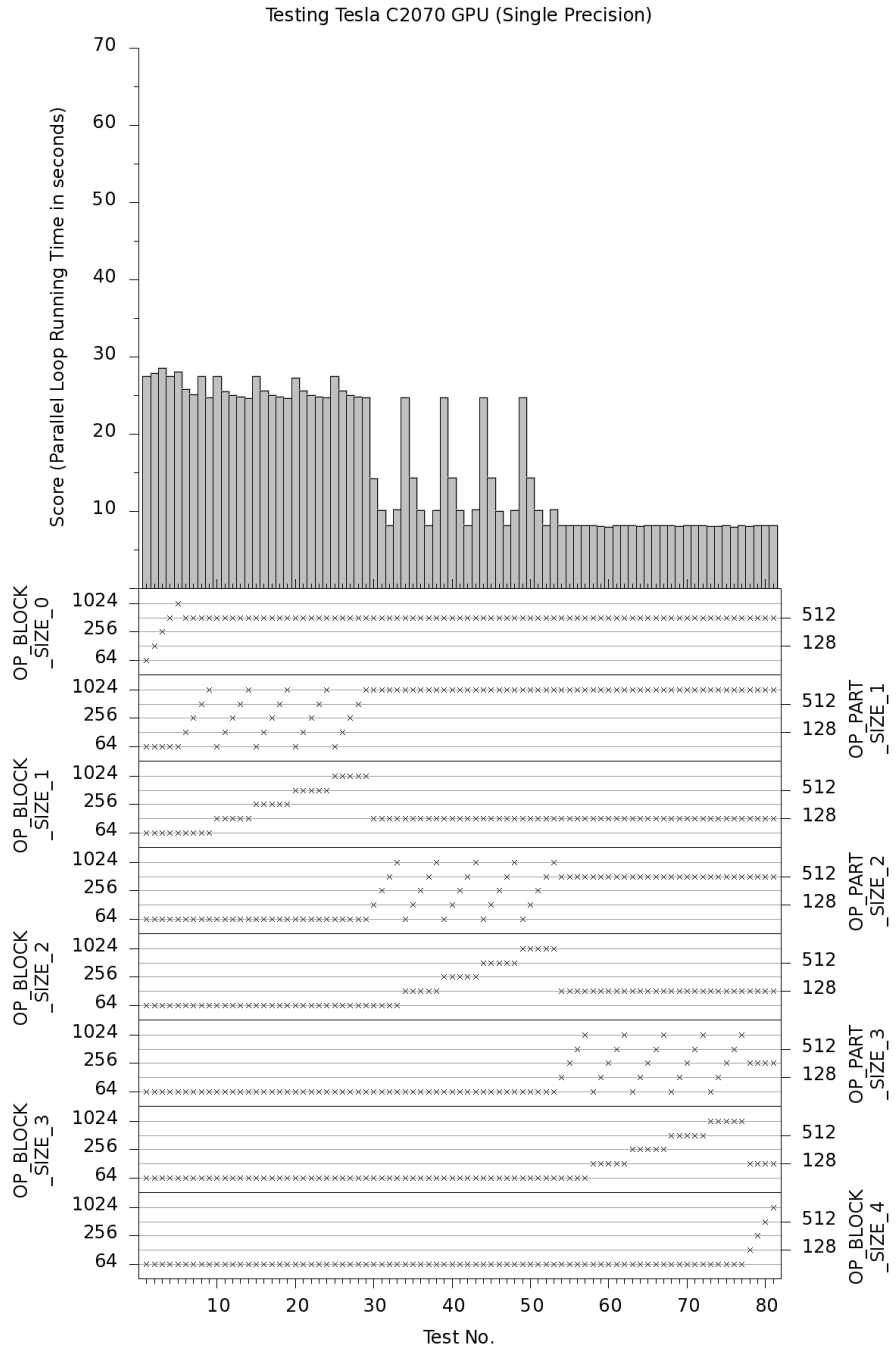


Figure 4.10: Airfoil tuning on a Tesla C2070, Single Precision.

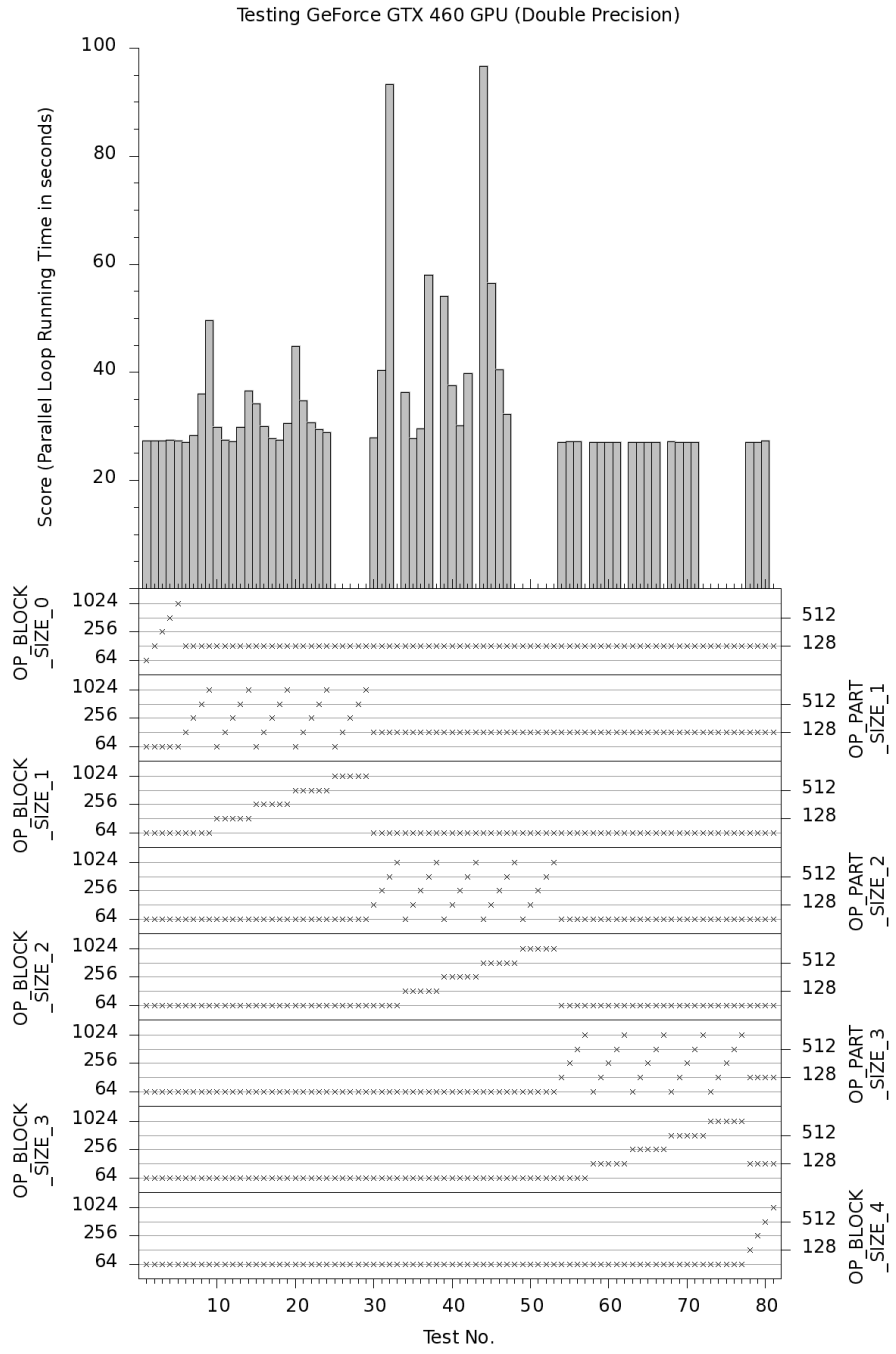


Figure 4.11: Airfoil tuning on a GeForce GTX 460, Double Precision.

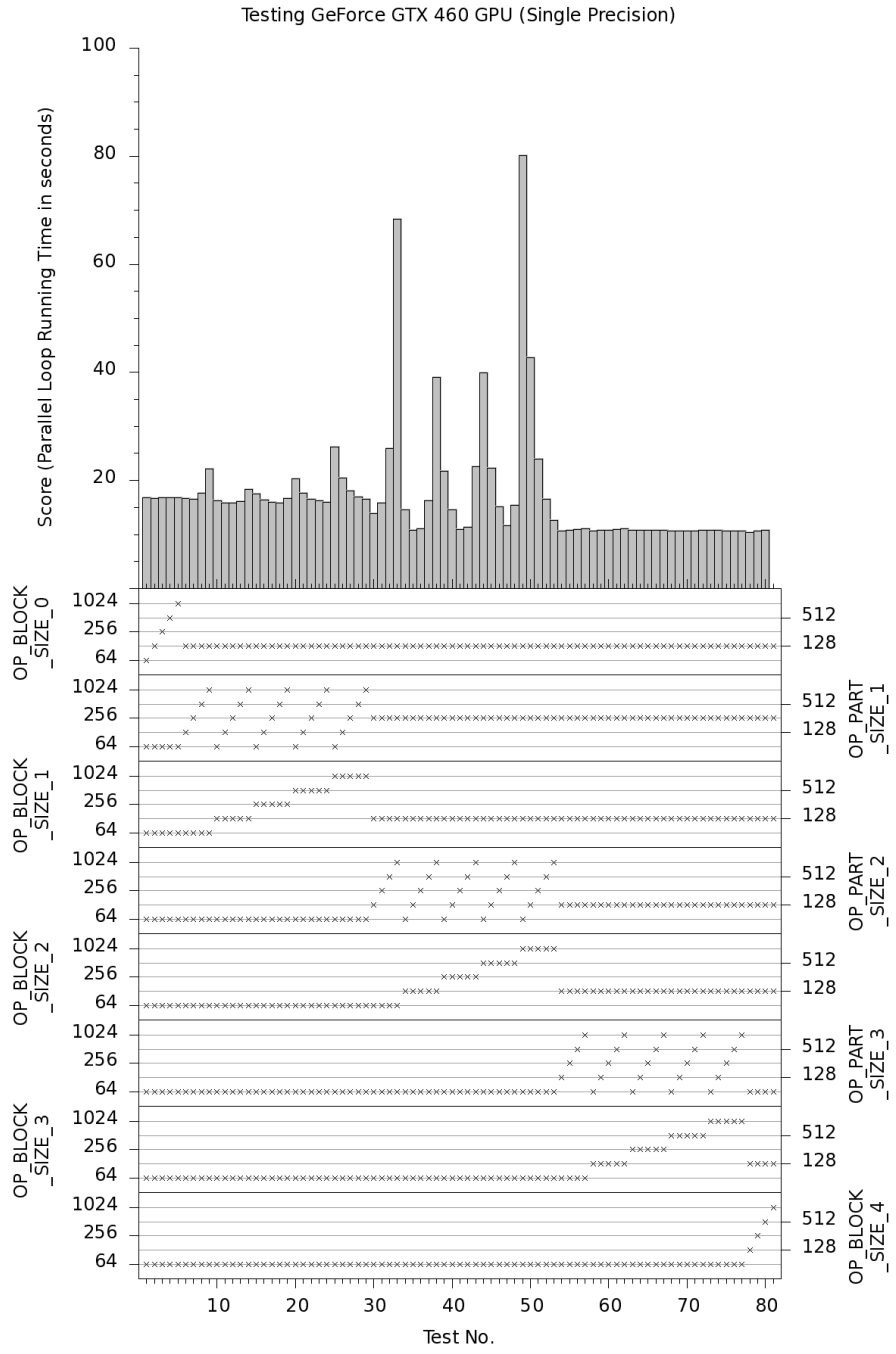


Figure 4.12: Airfoil tuning on a GeForce GTX 460, Single Precision.

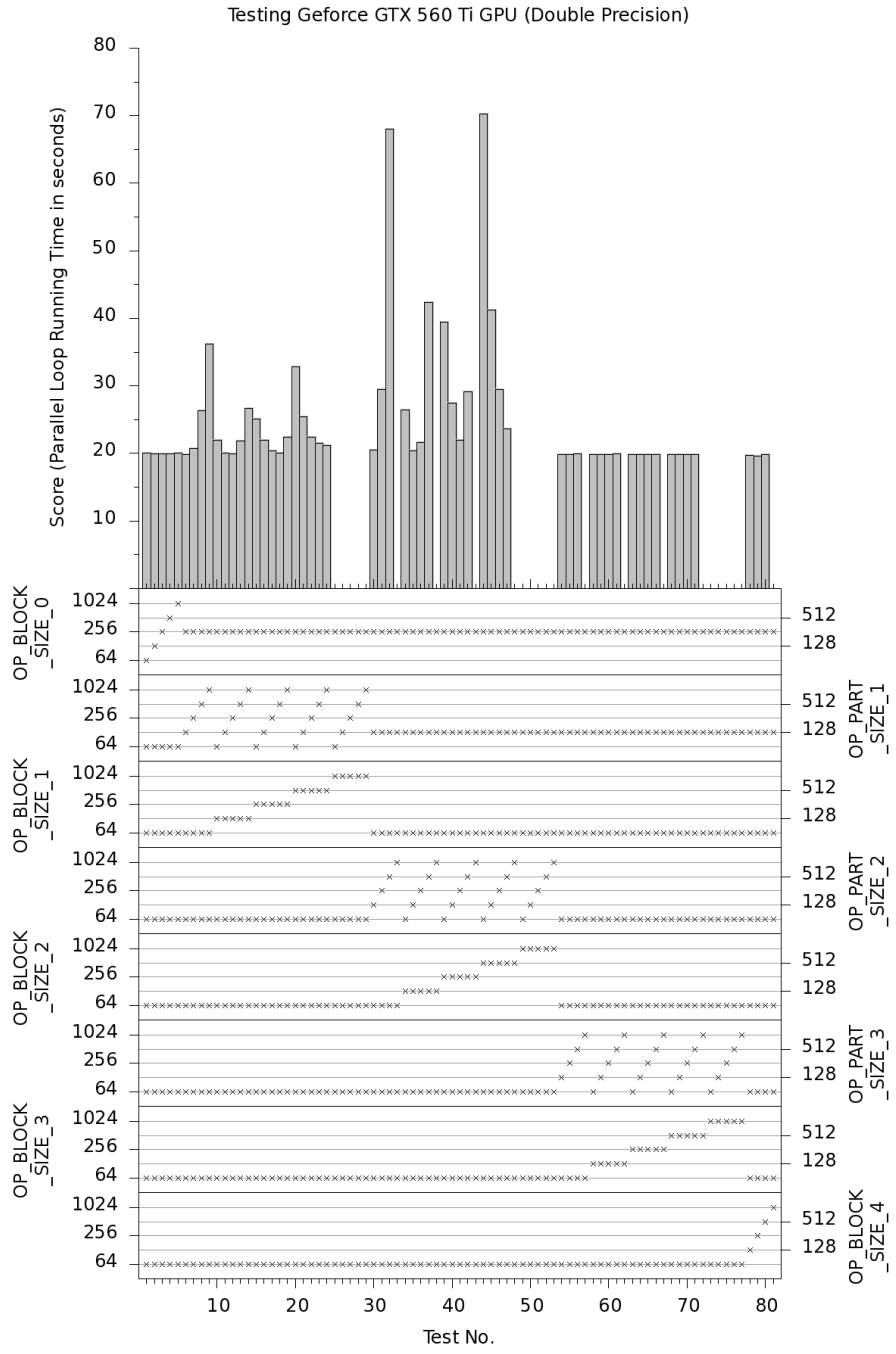


Figure 4.13: Airfoil tuning on a GeForce GTX 560 Ti, Double Precision.

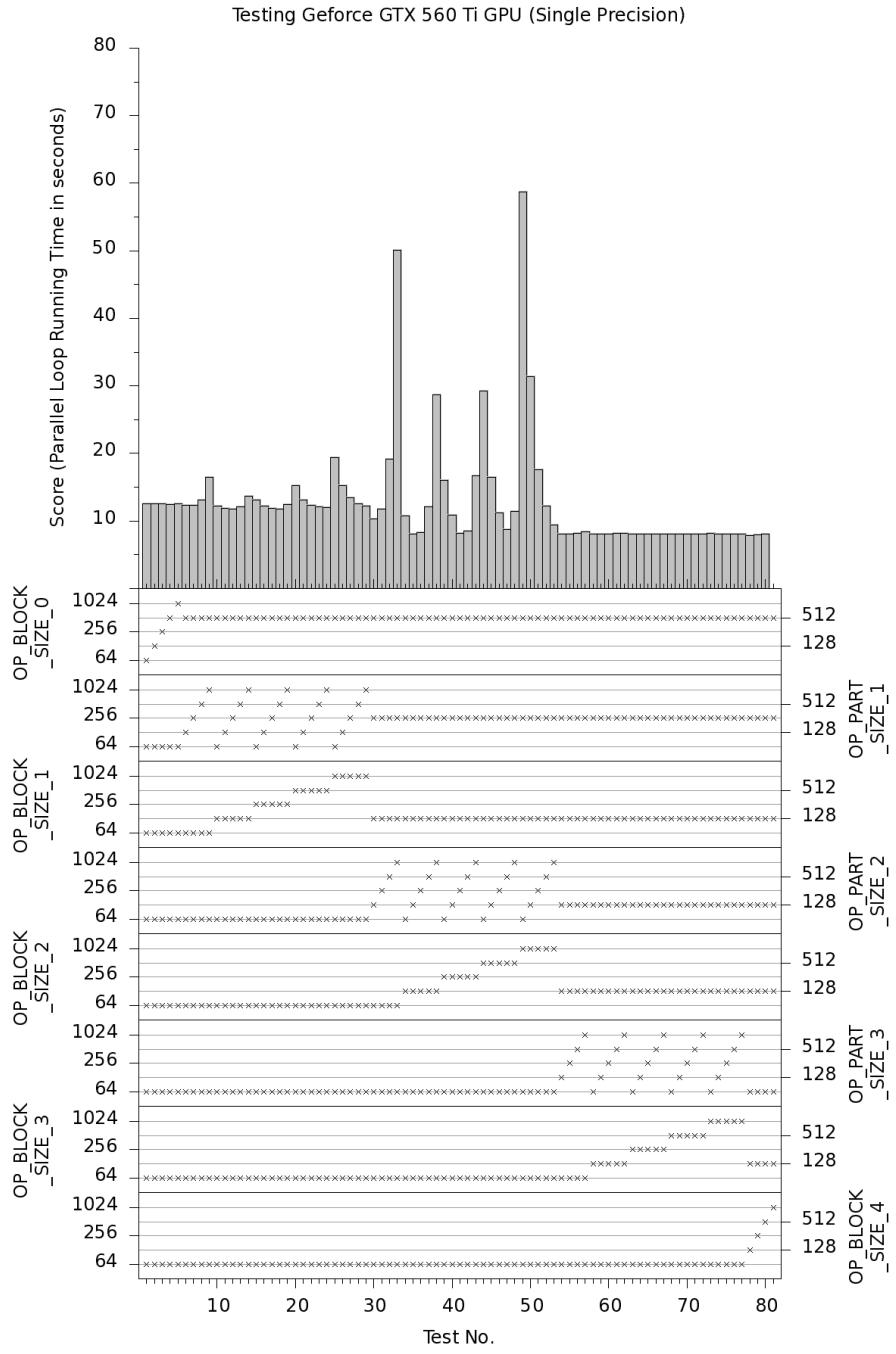


Figure 4.14: Airfoil tuning on a GeForce GTX 560 Ti, Single Precision.

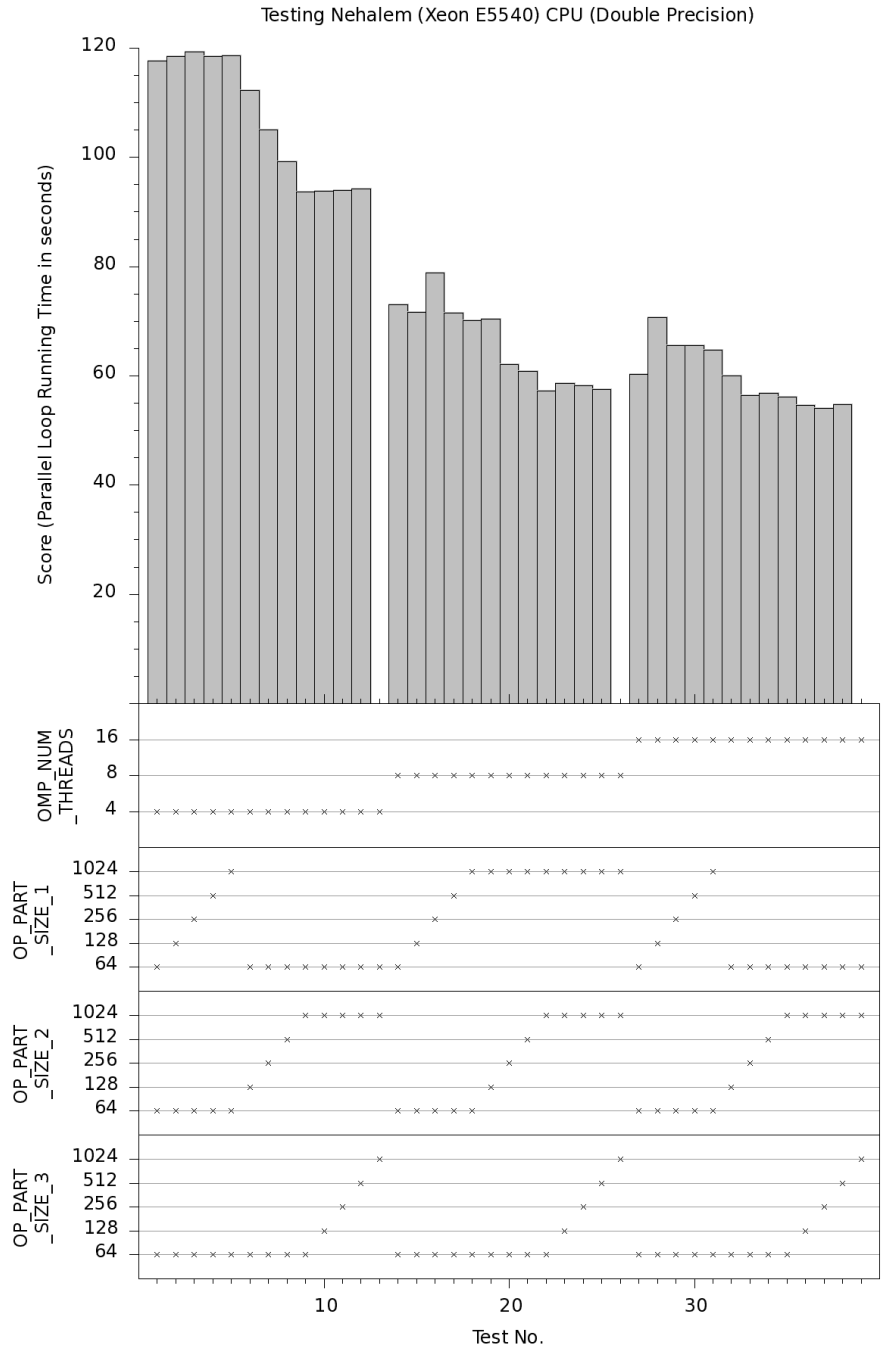


Figure 4.15: Airfoil tuning on a Nehalem Xeon E5540, Double Precision.

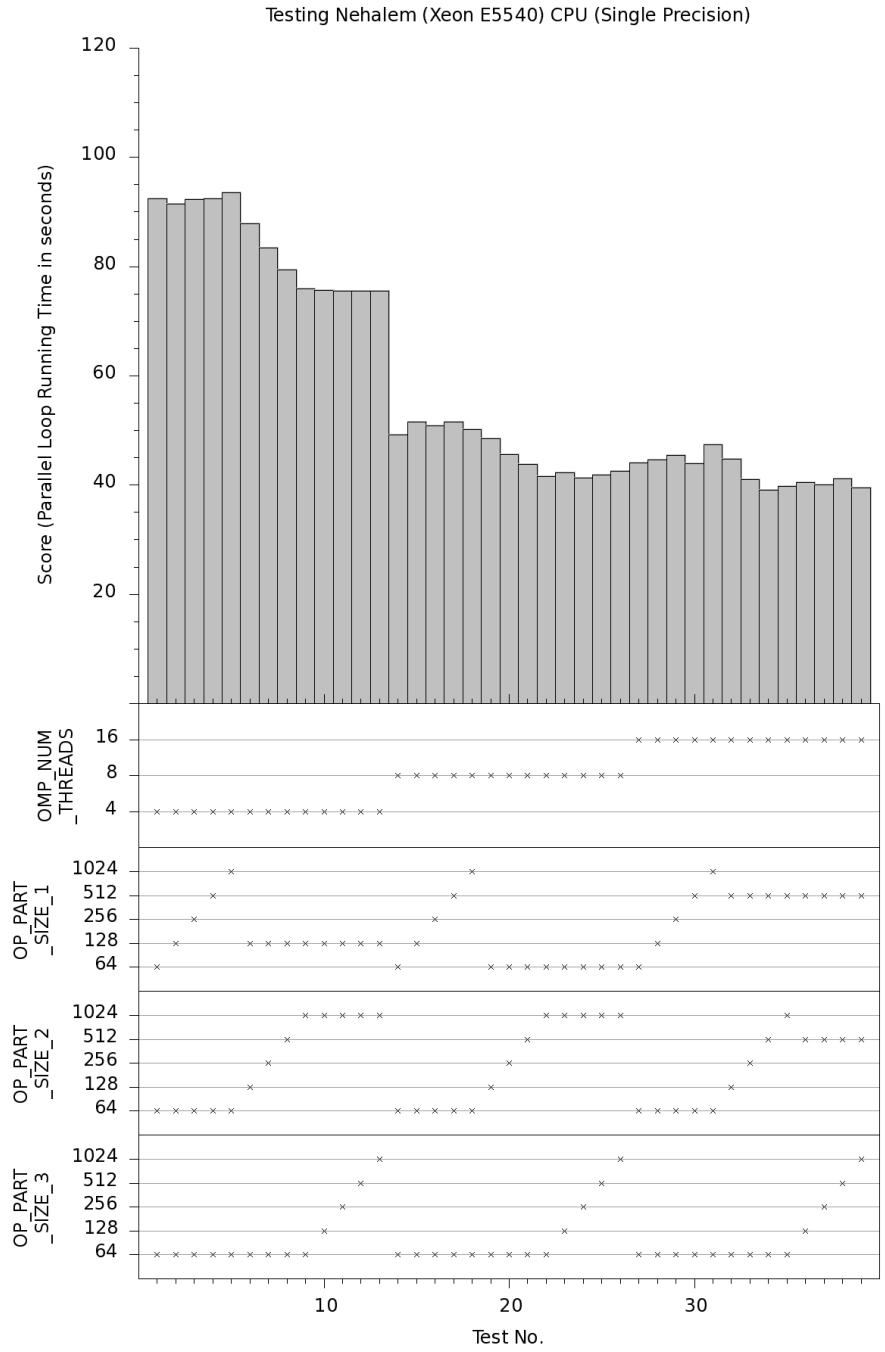


Figure 4.16: Airfoil tuning on a Nehalem Xeon E5540, Single Precision.

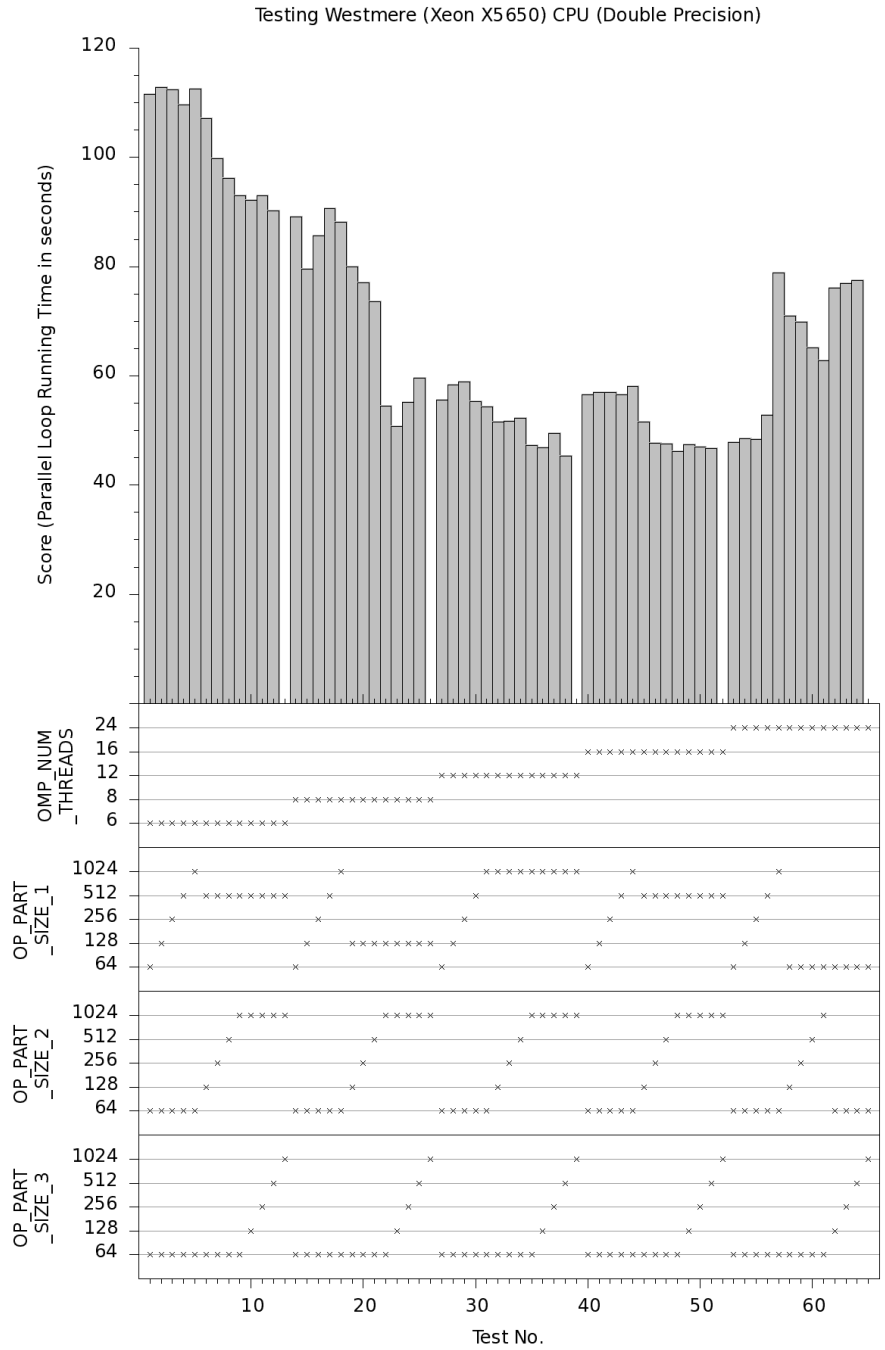


Figure 4.17: Airfoil tuning on a Westmere Xeon X5650, Double Precision.

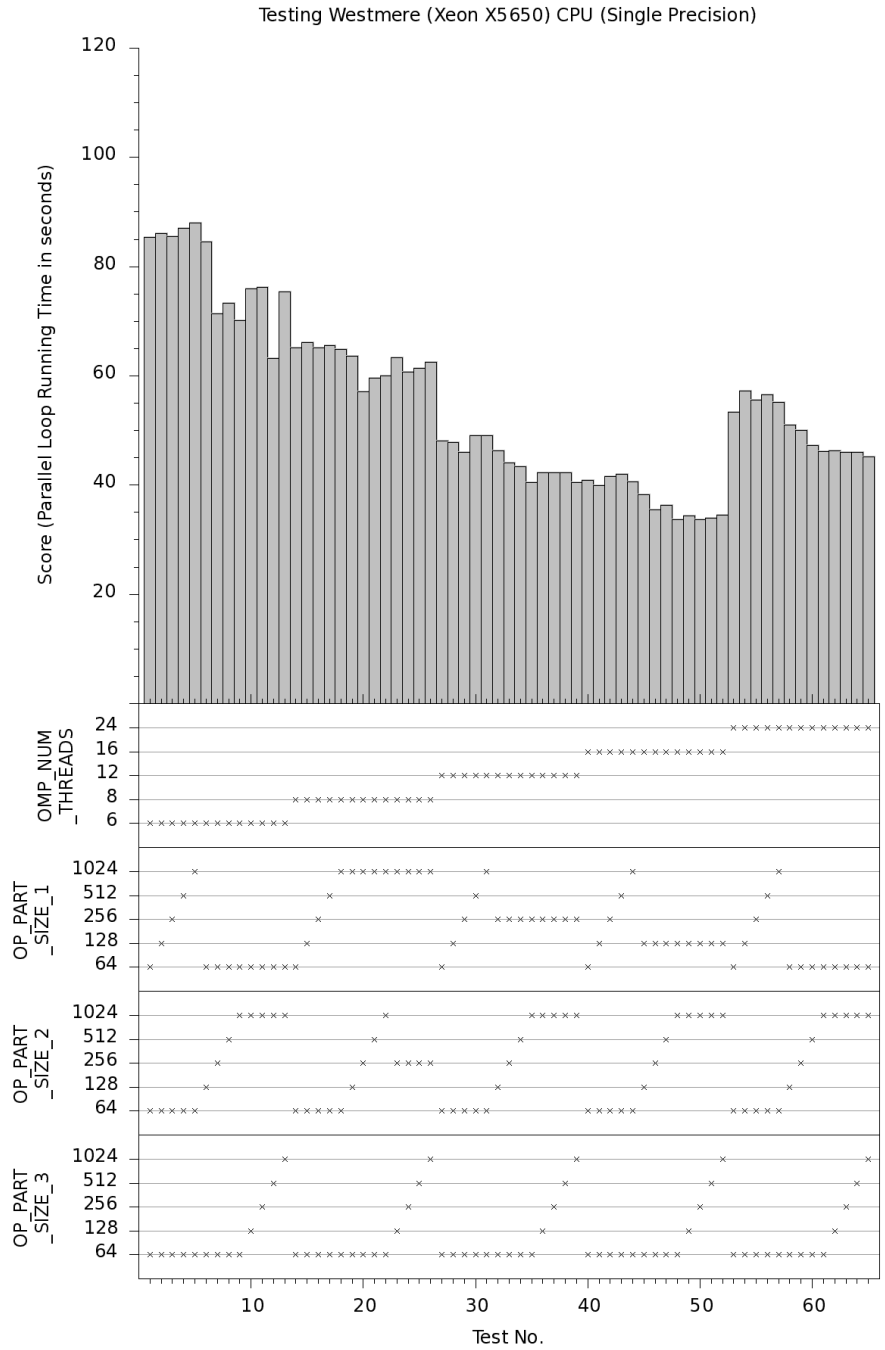


Figure 4.18: Airfoil tuning on a Westmere Xeon X5650, Single Precision.

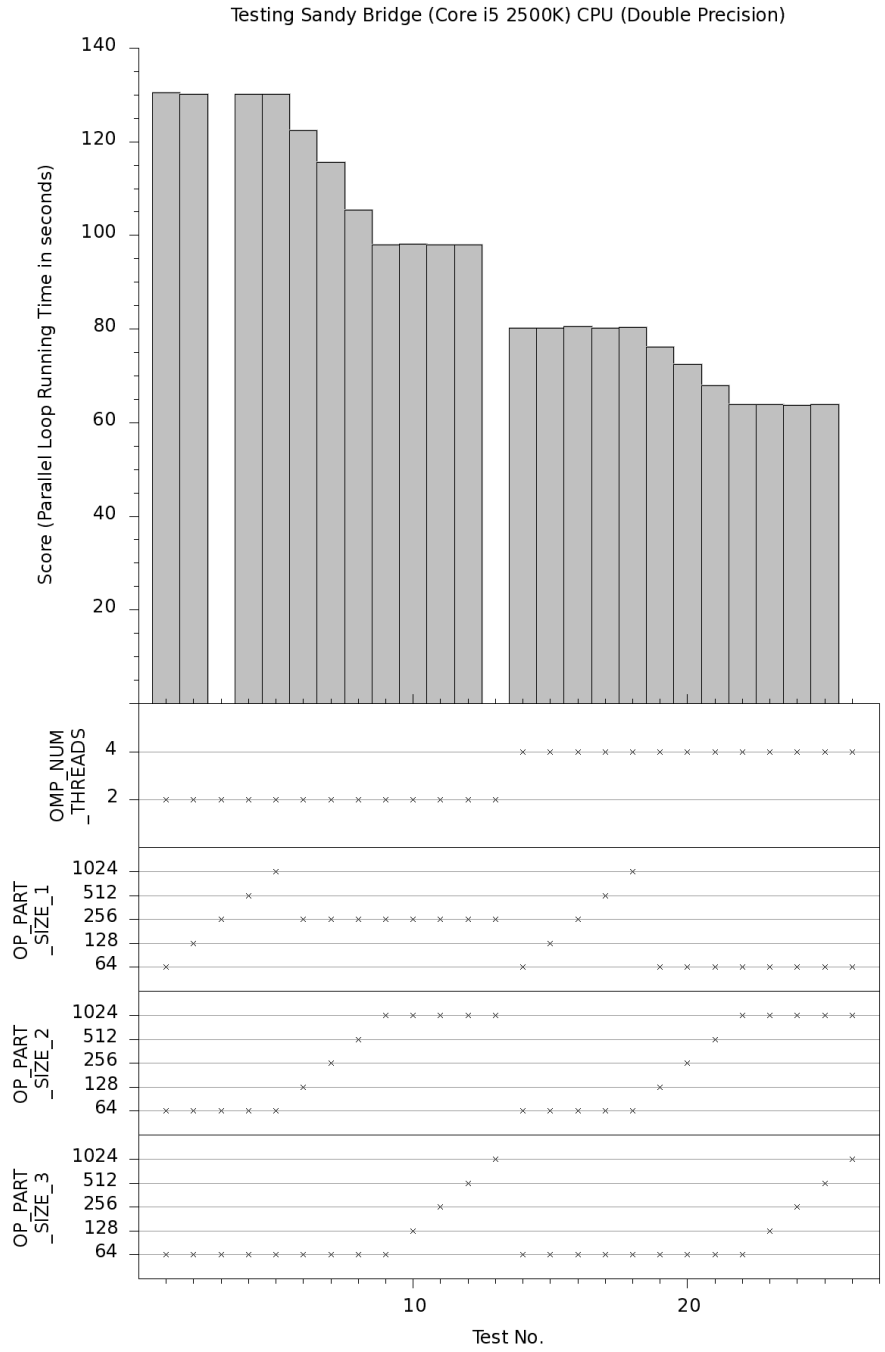


Figure 4.19: Airfoil tuning on a Sandy-Bridge Core i5, Double Precision.

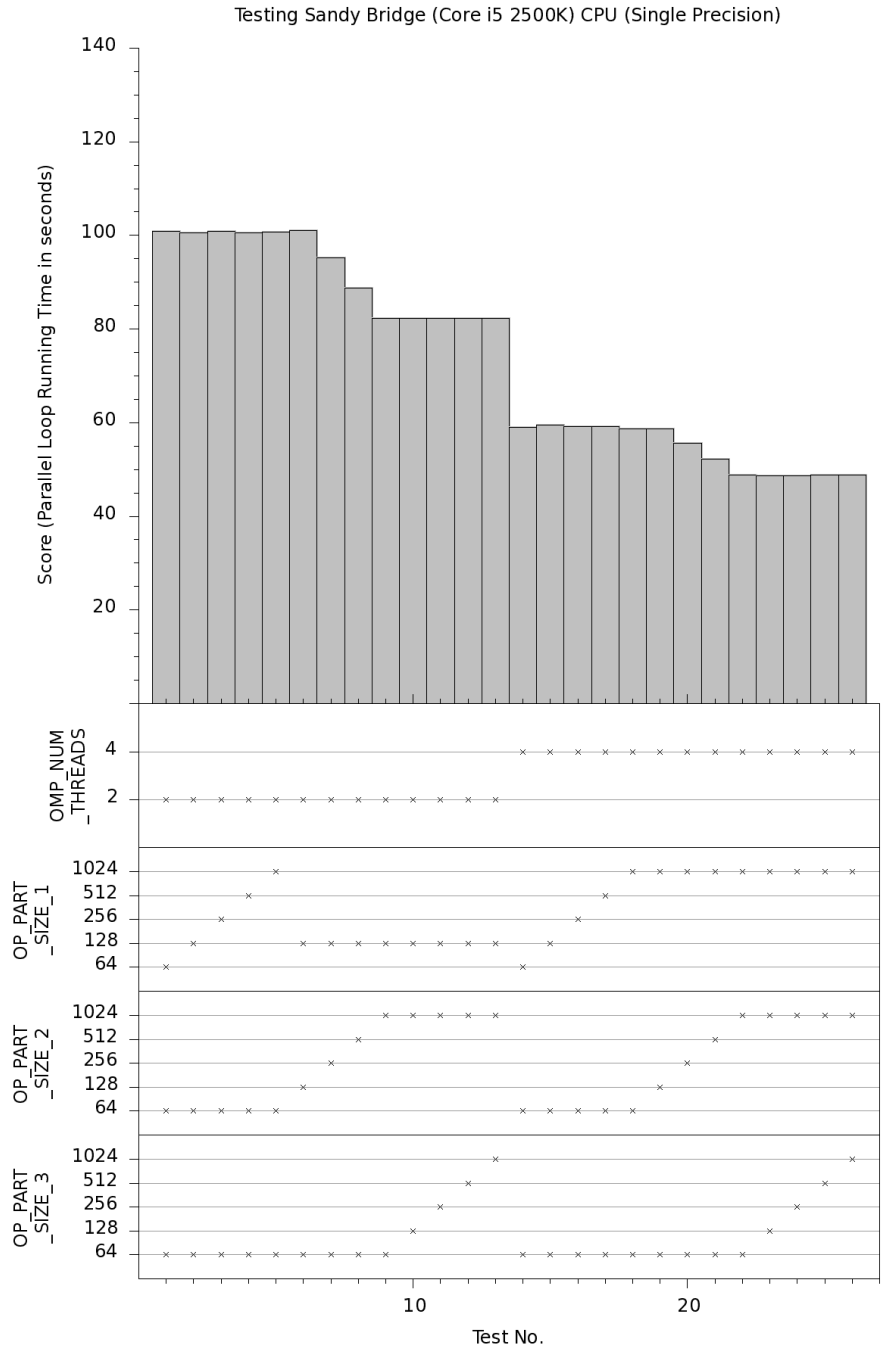


Figure 4.20: Airfoil tuning on a Sandy-Bridge Core i5, Single Precision.

4.4 In-Depth Tuning

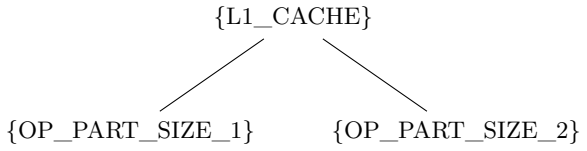
To demonstrate how a programmer might be able to further improve their code using auto-tuning a second time, I looked into the airfoil simulation in more detail. The simulation uses a time-marching algorithm, where each state is calculated from the previous one. I ran the tuning once using 500 iterations and once using 10,000. This showed that a small test case could be fairly quickly tuned to provide information which was relevant to a much longer run.

Strategy

The two parallel loops most affecting performance are `adt_calc` and `res_calc` so this tuning focused on those loops. I reduced the set of parameters and expanded the sets of possible variables to study these performance-critical loops in detail. For OP_PART_SIZE_1, I expanded the range of values to both lower and higher than before. For OP_PART_SIZE_2, which had the greatest effect, I tested every multiple of 64 up to 1024, as hardware limitations stop the test from running at higher values.

The Tesla C2070 I used for testing can cache memory accesses by the processing cores. Data is cached in 128B lines, whereas a direct memory access may be as small as 32B. So if a thread's memory access pattern is sparse with a large 'stride' it can be beneficial to disable this cache, using the CUDA compiler flags `-Xptxas -dlcm=cg` (disable) or `-Xptxas -dlcm=ca` (enable). My tuning tested the effect of this caching on the simulation.

OP_PART_SIZE_1 :	$32, 128, 512, 1024, 1536$
OP_PART_SIZE_2 :	$64, 128, 192, 256, 320, 384, 448, 512,$ $576, 640, 704, 768, 832, 896, 960, 1024$
L1_CACHE :	`cg`, `ca`



Results

The results of my tuning are shown in Figures 4.21 and 4.22 for the 500 and 10,000-iteration runs, respectively.

Analysis

The caching made surprisingly little difference, with very similar scores and performance characteristics with or without it. It had little effect on the speed or the other parameters' optimums. Otherwise, the tuning was consistent with previous results: OP_PART_SIZE_1 provided moderate improvement and OP_PART_SIZE_2 had a significant effect. Both parameters were generally better set as high as possible. There is a small 'dip' in running time around OP_PART_SIZE_2 = 512 which gave the overall optimum (L1_CACHE = `ca`, OP_PART_SIZE_1 = 1536, OP_PART_SIZE_2 = 640), although these differences were all very small.

The 500 and 10,000-iteration tuning have almost identical performance characteristics. An optimal valuation from a small test case can be used on larger examples with almost optimal performance. Choosing a representative test case is the responsibility of the programmer, but is easy for these time-marching simulations.

Conclusion

This testing successfully showed that detailed information on important parts of the program could be gained by two relatively short tuning runs, instead of one very large, detailed run. I also showed that optimisations for small test cases are applicable to large simulation runs, as long as a suitable small-scale test is chosen.

When I ran the 10,000-iteration code using the optimal valuation found in the 500-iteration testing, the score was 190.5s—within 8.6% of the optimal score from the full 10,000-iteration tuning (175.4s, at valuation L1_CACHE = `ca`, OP_PART_SIZE_1 = 1024, OP_PART_SIZE_2 = 896). My system's improvement over the worst test run (taking 789.3s) was 97.5% of the optimal improvement.

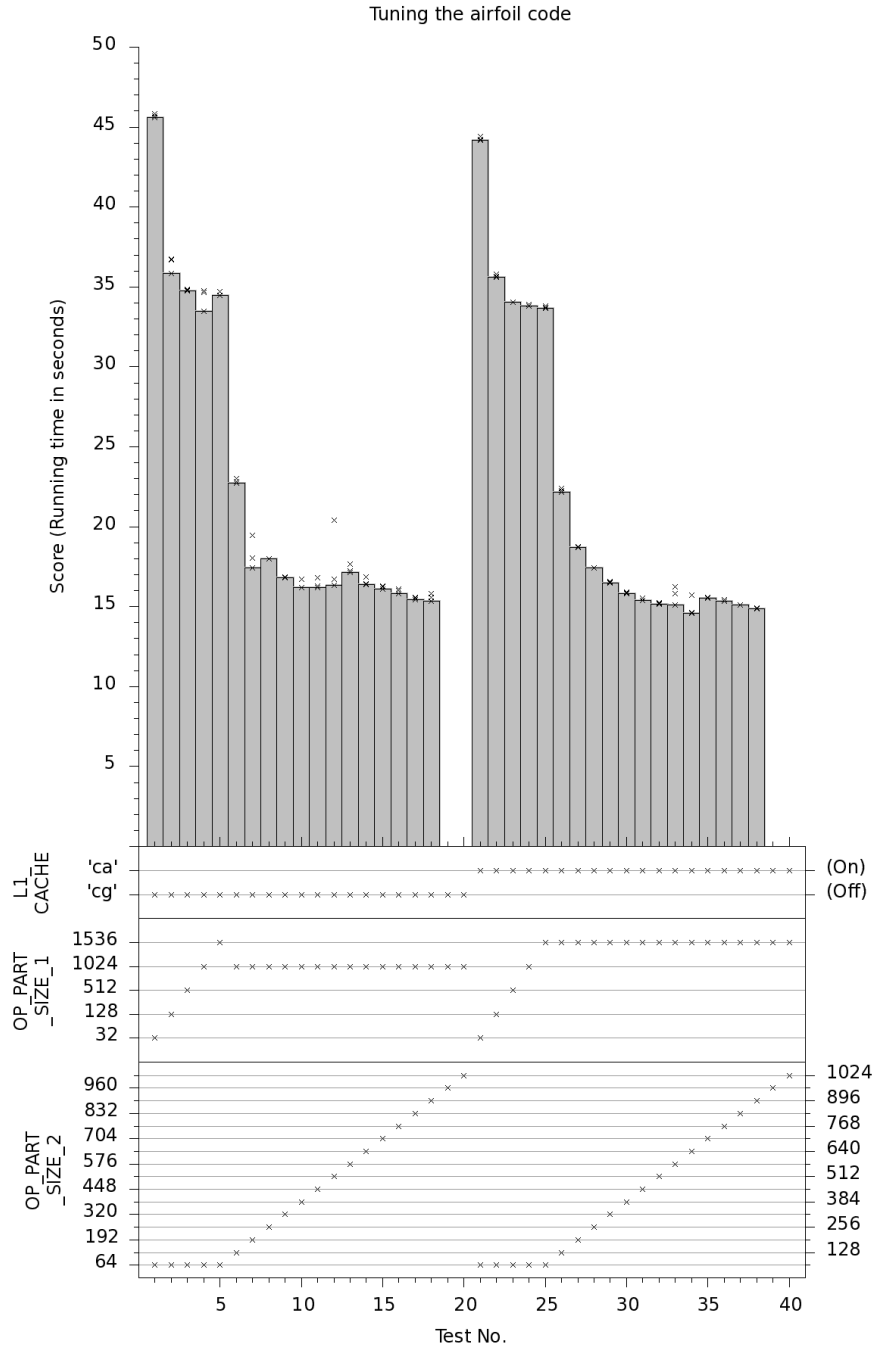


Figure 4.21: Airfoil code with 500 iterations tuning results on Tesla C2070, Double Precision. Each test was run 3 times (marked with crosses).

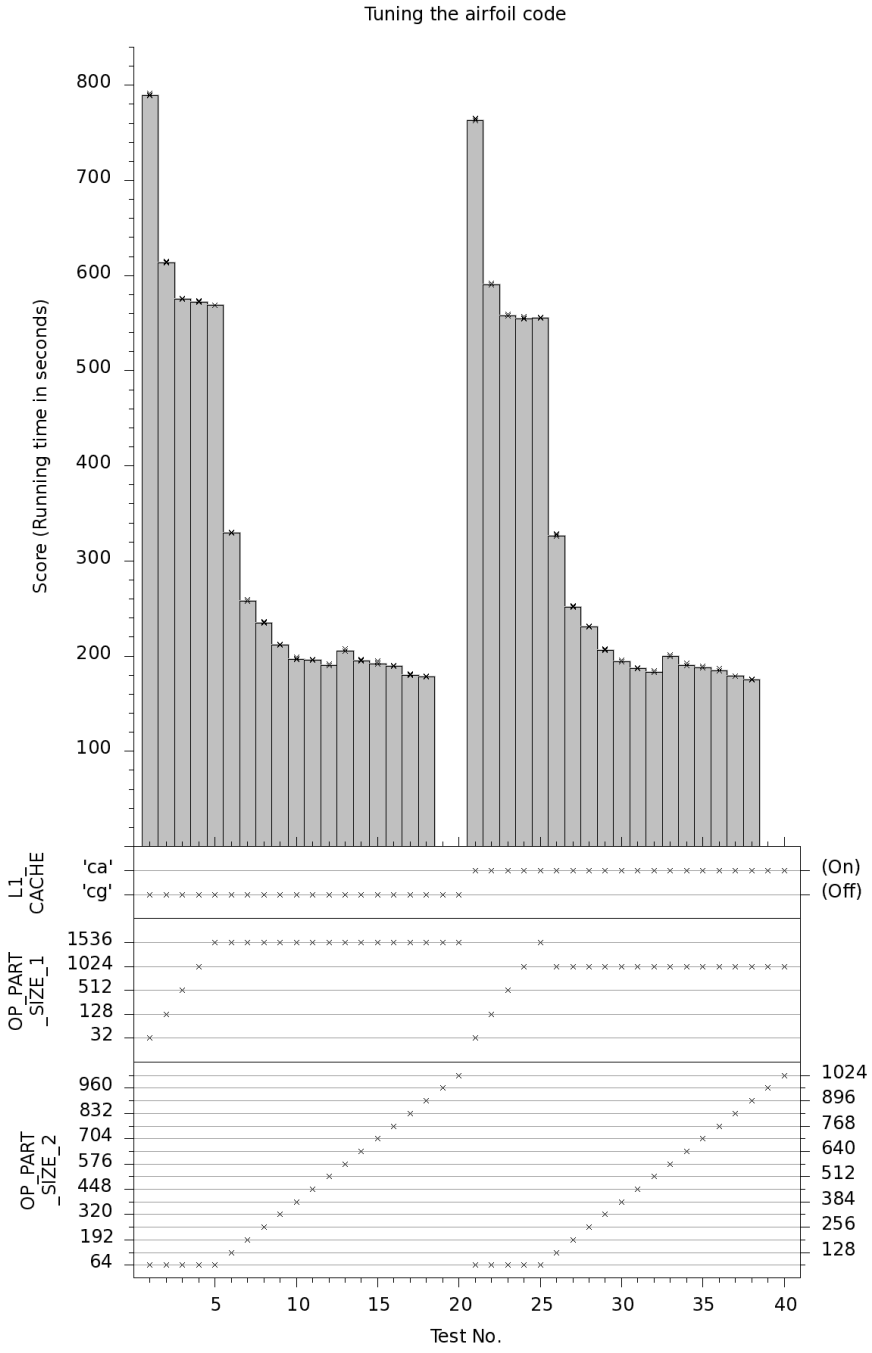


Figure 4.22: Airfoil code with 10,000 iterations tuning results on Tesla C2070, Double Precision.

5

Conclusions

I have created a general-purpose auto-tuning framework to help programmers determine optimal parameter values for their programs.

The system's design and development was driven by real-world needs, so it is genuinely useful for developing real software. The system has appropriate and useful features without being overly complex and is, I believe, far superior to previous brute-force and hand-tuning approaches.

The system is already being used in OeRC, which backs this up. Results from the airfoil tuning were included in a presentation at the MRSC 2011 conference, which shows that this type of optimisation is certainly useful.

5.1 Testing Results

When programming GPUs, there is a huge performance difference between good and bad block size choices, but it is difficult to predict an appropriate size even for an experienced programmer. In my testing the performance gap was commonly a factor of three or even more, even between sensible-seeming values.

By exploiting variable independence the auto-tuning can feasibly test many more possible parameter values than brute-force testing. The small-case tuning of the airfoil code performed 40 tests in under 45 minutes; an exhaustive search would require 160. Previously found optimums are used by my algorithm, so brute-force testing would take significantly more than four times as long.

For many scientific applications, choosing a small test case which accurately represents much larger problem instances is easy. It is critical that short tuning runs can be performed relatively quickly and the results are still applicable to large problems.

5.2 Limitations

Running Time

Tuning may require an exponential number of tests to be performed, limiting the number of parameter values which can feasibly be tested. In practice, the number of parameters and values is likely to be relatively small—the programmer can select a few suitable candidate values. Also, one high-level tuning run can be used to guide a second, focused run, allowing detailed analysis of important parts of a program without requiring a huge parameter space. Finally, variable independence can hugely reduce the number of tests required.

Scope

The system can only perform tests specified by the programmer—there is no analysis or suggestion of optimisations, which might be useful to inexperienced programmers. The tool is designed to help choose between well defined potential solutions, not to suggest them. New optimisations can be discovered by performing multiple tuning runs, focusing on the most important areas with detailed tuning.

Programmer Education

The system's ability to educate the programmer could be expanded—as well as giving an optimal valuation, the ability to explore the results in more detail may be useful. Log files are currently generated which can be used to show which variables were most influential, but the system does not perform any such analysis automatically. Graphical output and analysis of the tuning could help indicate important areas of the program which should be inspected more closely.

5.3 Future Work

My goal of a general parameter tuning framework has been fulfilled. However, if this type of tuning were used in a more restricted setting, there would be many potential extra features, at the expense of generality.

Variable Independence

Currently, the variable independence is provided by the programmer, but could theoretically be determined by a static analysis of the program. This would be difficult for a general system, as it would need to parse and understand program source code. If auto-tuning were included in an IDE or a compiler, this parsing would already be in place, so the variable tree could be built automatically, using similar techniques to those used for automatic parallelisation [14].

Testing in Parallel

It may be useful to perform tests in parallel, particularly to scientific programmers with lots of parallel hardware, such as a GPU cluster. If each test requires

a single GPU, tuning could be accelerated almost linearly by using multiple GPUs to run multiple tests at once. Clearly, it must be guaranteed that tests do not interfere with each other or run differently depending on which hardware they are assigned, this would be an interesting idea for a much more specialised system.

Optimisation Methods

Machine Learning techniques have already been applied to compiler-based auto-tuning (for example [16] and [4]) to reduce the search space of possible solutions. Genetic algorithms are another method to quickly find good solutions to optimisation problems. Both methods could be very interesting to explore in a system such as mine, especially if combined with knowledge of variable independence to further guide the optimisation.

Run-Time Tuning

For libraries such as OP2, which have some control over how a problem is executed, it would be interesting to investigate run-time optimisation. At each iteration of a simulation, new parameter values could be tried, so the simulation self-tunes as it runs, rather than requiring an initial tuning run.

5.4 Assessment

My system has already proved itself useful in tuning programs across a variety of hardware platforms, highlighting their similarities and differences. The presentation of tuning results at MRSC 2011 and inclusion in a forthcoming JPDC paper support this. My goals of generality and ease of use have been achieved—the system provides massive benefit over previous solutions, as well as providing a being block for more tightly-focused systems using similar ideas.

References

[1] Khronos Group: OpenCL. `http://www.khronos.org/opencl/`.

[2] NVIDIA: CUDA. `http://www.nvidia.com/object/cuda_home.html`.

[3] OpenMP API. `http://openmp.org`.

[4] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:295–305, 2006.

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2nd edition, 2006.

[6] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT press, 2nd edition, 2001.

[8] James Demmel, Jack Dongarra, Armando Fox, Sam Williams, Vasily Volkov, and Katherine Yelick. Accelerating Time-to-Solution for Computational Science and Engineering. *Scientific Discovery Through Advanced Computing*, (15):46–57, 2009.

[9] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Platform Adaptation"*, 93(2):216–231, 2005.

[10] Mike Giles. Course on CUDA programming on NVIDIA GPUs. Course materials available at: `http://people.maths.ox.ac.uk/gilesm/cuda/`.

[11] Mike Giles. A framework for parallel unstructured grid applications on GPUs. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 2010.

[12] Mike Giles and Gihan Mudalige. Optimising the OP2 Framework for GPU Architectures. In *Journal of Parallel and Distributed Computing, Special edition on "Novel Architectures in HPC"*, 2011 (In preparation).

[13] Mike Giles, Gihan Mudalige, and Ben Spencer. Optimising the OP2 Framework for GPU Architectures. In *Many-core and Reconfigurable Supercomputing Conference*, April 2011. Presentation slides available at: `http://www.mrsc2011.eu/sites/default/files/mrsc11_giles.pdf`.

[14] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[15] Lee Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H.J. Kelly. Towards Metaprogramming for Parallel Systems on a Chip. In *Proceedings of the 3rd EuroPar Workshop on Highly Parallel Processing on a Chip*, volume 6043 of *Lecture Notes in Computer Science*. Springer, 2009.

[16] Shun Long and Michael O'Boyle. Adaptive java optimisation using instance-based learning. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 237–246. ACM, 2004.

[17] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the ACM/IEEE conference on Supercomputing*, page 10. ACM Press, Portland, November 2009.

[18] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7:249–268, 1977.

[19] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, International 2nd edition, 2006.

[20] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. *International Parallel and Distributed Processing Symposium*, 2009.

[21] Jochen Voß. Wisent: a python parser generator. `http://seehuhn.de/pages/wisent`.

[22] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[23] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software: Practice & Experience*, 38(15):1621–1642, April 2008.

[24] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[25] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.

[26] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358–386, February 2005.

Source Code Listing

tune.py

```
#! /usr/bin/env python

# Autotuning System

# v0.11


# Command line arguments
import sys
# Configuration file handling
from tune_conf import get_settings
# The optimiser
from optimisation import Optimisation
# Running commands
import os
from subprocess import Popen, PIPE, STDOUT
# Timing commands
import time
# Maths
import math
# Test Logging and output.
from logging import TestLog


# The main script ##################################################################

def main():
    # The script has been executed directly

    print
    print "Autotuning System".center(80)
    print "v0.11".center(80)
    print

    # Command Line Arguments
    # Read name of config file from command line

    if len(sys.argv) == 2:
        configFile = sys.argv[1]
    else:
        print "Usage: Please provide the path to a configuration file as an
             argument."
        print "When no arguments are provided, some sample tests are run."
        print "Press Enter to run the tests."

        try:
            raw_input()
        except KeyboardInterrupt: # Ctrl-C
            print
            exit()

        from testing import run_testing

        run_testing()

        exit()




    # Get the settings
    settings = get_settings(configFile)


    # Set the working directory to that of the config file
    # This means that commands in the config file can be written
    # RELATIVE TO THE CONFIG FILE
    # and this program will respect this. The config writer need not assume
    # where the program will be run from, only where the config is stored.
    path = os.path.dirname(os.path.realpath(configFile))
    os.chdir(path)
```

```
71
72
73      if(True):
74          print "Retrieved settings from config file:"
75          print
76          print "vartree:\n" + settings['vartree']
77          print
78
79          from vartree import treeprint_str
80          print "displayed as a tree:\n"
81          print treeprint_str(settings['vartree'])
82
83          print "possible values:\n" + strVarVals(settings['possValues'])
84          print
85
86          for opt in ['compiler', 'test', 'evaluation', 'cleanup']:
87              if opt in settings:
88                  print opt + ": \n" + str(settings[opt]) + "\n"
89
90          print
91
92
93
94      # Check if we are keeping logs and if so initialise the logging.
95      logging = settings['log'] is not None
96      if logging:
97          testLog = TestLog(True)
98
99
100
101     # Build the evaluation function
102
103     def evaluate(valuation):
104         # valuation is a dictionary mapping variable names to values.
105         # returns the figure of merit (usually time taken) of runnning the test
                with these values.
106
107         PRINT_PROGRESS = True
108         PRINT_TESTING = True
109
110         # The optimiser will only call this function once per valuation.
111         # So there is no need to check for multiple runs.
112         # We store a static variable holding the number of tests performed, so
                each can be given a unique id.
113         # This variable is incremented here, but defined after evaluate, so it
                will be static.
114         evaluate.testNum += 1
115
116
117         # Add this test to the log
118         if logging:
119             testLog.createTest(evaluate.testNum, valuation)
120
121
122         if PRINT_PROGRESS:
123             print "Test " + str(evaluate.testNum) + ":"
124             print strVarVals(valuation, ", ")
125
126
127         # First, compile the test, if needed.
128         if 'compiler_mkStr' in settings:
129             if PRINT_PROGRESS: print "Compiling test " +  str(evaluate.testNum)
130
131             cmdStr = settings['compiler_mkStr'](evaluate.testNum, valuation)
132             # Start the compilation
133             if PRINT_TESTING:
134                 p = Popen(cmdStr, shell=True)
135             else:
136                 p = Popen(cmdStr, shell=True, stdout=PIPE, stderr=STDOUT) #
                        Collect the output, without printing.
137             # Wait for the compilation to finish, this sets the return code.
138             p.wait()
139             # Check the retun code.
140             if(p.returncode != 0):
```

```
141                 evaluate.failures.append(("COMPILATION OF TEST " + str(evaluate.
                        testNum) + " FAILED.", valuation))
142                 return None
143 
144 
145         # Repeat the tests the number of times specified
146         result = 0.0;
147         result_sq = 0.0;
148         min_result = None;
149         max_result = None;
150         median_result = []
151 
152         for i in xrange(1, settings['repeat']+1):
153 
154             # Run the test
155             # One of these branches should always be true.
156             if 'evaluation_mkStr' in settings:
157 
158                 if PRINT_PROGRESS:
159                     nthRun = ""
160                     if settings['repeat'] > 1:
161                         nthRun = " ("+ordinal(i)+" run)"
162                     print "Running test " +  str(evaluate.testNum) + nthRun
163 
164                 # Execute the evaluation, the result will be output on the last
                        line.
165 
166                 cmdStr = settings['evaluation_mkStr'](evaluate.testNum, valuation
                        )
167                 # Start the evaluation
168                 p = Popen(cmdStr, shell=True, stdout=PIPE, stderr=STDOUT)
169                 # Wait for the evaluation to finish, this sets the return code.
170                 p.wait()
171                 # Check the retun code.
172                 if(p.returncode != 0):
173                     evaluate.failures.append(("EVALUATION OF TEST " + str(
                            evaluate.testNum) + " FAILED.", valuation))
174                     return None
175                 # Get the program output.
176                 output = p.stdout.readlines()
177 
178                 if PRINT_TESTING: print ''.join(output)
179 
180                 if len(output) == 0:
181                     print "Test did not return any output!"
182                     exit()
183 
184                 # Take the last line of output to be the FOM.
185                 result += float(output[-1])
186                 result_sq += float(output[-1]) ** 2
187                 min_result = min(min_result, float(output[-1])) if min_result is
                        not None else float(output[-1])
188                 max_result = max(max_result, float(output[-1])) if max_result is
                        not None else float(output[-1])
189                 median_result.append(float(output[-1]))
190 
191                 if PRINT_PROGRESS and settings['repeat'] > 1:
192                     print "Result of test " + str(evaluate.testNum) + ", " +
                            ordinal(i) + " run: " + str(float(output[-1]))
193 
194 
195                 #Add this score to the log:
196                 if logging:
197                     testLog.logTest(evaluate.testNum, float(output[-1]))
198 
199 
200             elif 'test_mkStr' in settings:
201 
202                 if PRINT_PROGRESS:
203                     nthRun = ""
204                     if settings['repeat'] > 1:
205                         nthRun = " ("+ordinal(i)+" run)"
206                     print "Running test " +  str(evaluate.testNum) + nthRun
207 
```

```
208                 # Execute test, the result will be the time taken.
209
210                 cmdStr = settings['test_mkStr'](evaluate.testNum, valuation)
211
212
213                 start = time.time()
214
215                 # Start the test
216                 if PRINT_TESTING:
217                     p = Popen(cmdStr, shell=True)
218                 else:
219                     p = Popen(cmdStr, shell=True, stdout=PIPE, stderr=STDOUT) #
                         Collect the output, without printing.
220                 # Wait for the test to finish, this sets the return code.
221                 p.wait()
222
223
224                 stop = time.time()
225
226
227                 # Check the retun code.
228                 if(p.returncode != 0):
229                     evaluate.failures.append(("RUNNING OF TEST " + str(evaluate.
                             testNum) + " FAILED.", valuation))
230                     return None
231
232
233                 result += stop - start
234                 result_sq += (stop - start) ** 2
235                 min_result = min(min_result, stop - start) if min_result is not
                         None else stop - start
236                 max_result = max(max_result, stop - start) if max_result is not
                         None else stop - start
237                 median_result.append(stop - start)
238
239                 if PRINT_PROGRESS and settings['repeat'] > 1:
240                     print "Result of test " + str(evaluate.testNum) + ", " +
                             ordinal(i) + " run: " + str(stop - start)
241
242
243                 #Add this score to the log:
244                 if logging:
245                     testLog.logTest(evaluate.testNum, stop - start)
246
247
248             else:
249                 # Should never be reached.
250                 evaluate.failures.append(("COULD NOT RUN TEST " + str(evaluate.
                         testNum) + ".", valuation))
251                 return None
252
253
254         # End for loop running the test multiple times
255         # Divide result by settings['repeat'] to get an average.
256         # This might as well be sum, but an avg seems more intuitive.
257         # (and also allows us to calculate other things, such as std deviation)
258         result = result / settings['repeat']
259
260         median_result.sort()
261         median_result = (median_result[len(median_result)/2]) if bool(len(
                 median_result)%2) else (median_result[len(median_result)/2] +
                 median_result[len(median_result)/2 -1])/2.0
262
263         if PRINT_PROGRESS:
264             if settings['repeat'] > 1:
265                 print "Average result of test " + str(evaluate.testNum) + ": " +
                         str(result)
266                 print "Minimum Result: " + str(min_result)
267                 print "Maximum Result: " + str(max_result)
268                 print "Median Result: " + str(median_result)
269                 variance = (result_sq / settings['repeat']) - (result ** 2)
270                 print "Variance: " + str(variance)
271                 std_dev = math.sqrt(variance)
272                 print "Standard Deviaton: " + str(std_dev)
```

```
273                 c_o_v = std_dev / abs(result) if result != 0 else "Undefined (avg
                        is 0)"
274                 print "Coefficient of Variation: " + str(c_o_v)
275             else:
276                 print "Result of test " + str(evaluate.testNum) + ": " + str(
                        result)
277
278
279         # Run the cleanup, if needed
280         if 'cleanup_mkStr' in settings:
281             if PRINT_PROGRESS: print "Cleaning test " +  str(evaluate.testNum)
282
283             cmdStr = settings['cleanup_mkStr'](evaluate.testNum, valuation)
284             # Start the cleanup
285             if PRINT_TESTING:
286                 p = Popen(cmdStr, shell=True)
287             else:
288                 p = Popen(cmdStr, shell=True, stdout=PIPE, stderr=STDOUT) #
                        Collect the output, without printing.
289             # Wait for the cleanup to finish, this sets the return code.
290             p.wait()
291             # Check the retun code.
292             if(p.returncode != 0):
293                 evaluate.failures.append(("CLEANUP OF TEST " + str(evaluate.
                        testNum) + " FAILED.\n(test was still used)", valuation))
294                 # Need not return None, as we still got a result.
295
296
297
298         if PRINT_PROGRESS: print
299
300
301         # Set the overall result to min/max/med/avg:
302         if settings['overall'] == 'avg':
303             overall = result
304         elif settings['overall'] == 'max':
305             overall = max_result
306         elif settings['overall'] == 'med':
307             overall = median_result
308         else:
309             overall = min_result
310
311         # Add this overall score to the log
312         if logging:
313             testLog.logOverall(evaluate.testNum, overall)
314
315         # Return the FOM.
316         return overall
317
318
319     # Define evaluate.testNum, which will be static inside evaluate.
320     evaluate.testNum = 0
321     # Define a list of evaluations which failed in some way.
322     evaluate.failures = []
323
324
325
326
327     # Set up the optimiser
328     test = Optimisation(settings['vartree'], settings['possValues'], evaluate)
329
330
331     if(settings['optimal'] == 'min'):
332         test.minimiseScore()
333     if(settings['optimal'] == 'max'):
334         test.maximiseScore()
335
336
337     test.calculateOptimum()
338
339     if not test.successful():
340
341         print
342         print "Not enough evaluations could be performed."
```

```
343         print "There were too many failures."
344 
345     else:
346 
347 
348         print
349         print settings['optimal'].capitalize() + "imal valuation:" # Minimal or
                Maximal
350         print strVarVals(test.optimalValuation(), ", ")
351         print settings['optimal'].capitalize() + "imal Score:" # Minimal or
                Maximal
352         print test.optimalScore()
353         print "The system ran " + str(test.numTests()) + " tests."
354 
355 
356 
357     # Check for any failures during the evaluations
358     if(len(evaluate.failures) > 0):
359         print
360         print "FAILURES:"
361         for f in evaluate.failures:
362             print "    " + f[0]
363             print "    " + strVarVals(f[1], ", ")
364             print
365 
366 
367 
368     # Write the CSV log file, if needed
369     if logging:
370         testLog.writeCSV(settings['log'])
371 
372 
373 
374 
375 
376 
377 
378 
379 
380 
381 # A little helper
382 
383 
384 def strVarVals(d, sep="\n"):
385     return sep.join([str(var) + " = " + str(val) for var,val in sorted(d.items())
            ])
386 
387 
388 def ordinal(n):
389     if 10 <= n % 100 < 20:
390         return str(n) + 'th'
391     else:
392         return  str(n) + {1 : 'st', 2 : 'nd', 3 : 'rd'}.get(n % 10, "th")
393 
394 
395 
396 # Actually run the script ###########################################################
397 
398 if __name__ == '__main__':
399     main()
```

optimisation.py

```
"""
Autotuning System

optimisation.py

Defines the Optimisation class.
This represents the optimisation algorithm.
"""


# defines the VarTree class and a parser converting strings to VarTrees.
from vartree import VarTree, vt_parse




# The optimisation class ##############################################################

class Optimisation:

    # vartree - an instance of VarTree giving the variable tree to work on.
    # possValues - a dictionary mapping variable names to lists of possible values.
    #              N.B. this must include mappings for all variables in vartree
    # evaluationFunc - a function to evaluate a particular test.
    #                  takes a dictionary mapping variable names to values,
    #                  returns some figure of merit.

    # Initialisation
    def __init__(self, vartree, possValues, evaluationFunc):

        vt = vt_parse(vartree)

        self.__vartree = vt
        self.__possValues = possValues
        self.__evaluationFunc = evaluationFunc

        self.minimiseScore() # by default we take smaller scores as better.

        self.__resetStoredVals()


    # Resets memoized info and saved results.
    # Can be called if anything is changed which would invalidate this.
    def __resetStoredVals(self):
        self.__evaluationMemory = {}
        self.__optValuation = None
        self.__optScore = None
        self.__numTests = None
        self.__successful = None


    # updates possValues
    def setPossValues(self, possValues):
        self.__possValues = possValues
        self.__resetStoredValues()


    # updates evaluationFunc
    def setEvaluationFunc(self, evaluationFunc):
        self.__evaluationFunc = evaluationFunc
        self.__resetStoredValues()


    # use 'min' to calculate optimum valuation.
    def minimiseScore(self):
        self.__best = min
        self.__resetStoredVals()

    # use 'max' to calculate optimum valuation.
    def maximiseScore(self):
```

```
71          self.__best = max
72          self.__resetStoredVals()
73
74
75      # The evaluate function called by my code
76      # This provides a memoizing wrapper to evaluationFunc
77      # and also updates __numTests
78      def __evaluate(self, test):
79
80          # converts dictionaries to a form which can be used as a key
81          def makeKey(d):
82              return tuple(sorted(test.items()))
83
84          k = makeKey(test)
85
86          if k not in self.__evaluationMemory:
87              self.__evaluationMemory[k] = self.__evaluationFunc(test)
88              if self.__numTests is None:
89                  self.__numTests = 1
90              else:
91                  self.__numTests += 1
92
93          return self.__evaluationMemory[k]
94
95
96      # Returns an optimal valuation
97      def optimalValuation(self):
98          if self.__optValuation is None:
99              self.calculateOptimum()
100
101         return self.__optValuation
102
103     # Returns the score of an optimal valuation
104     def optimalScore(self):
105         if self.__optScore is None:
106             self.calculateOptimum()
107
108         return self.__optScore
109
110     # Returns the number of tests performed during optimisation.
111     def numTests(self):
112         if self.__numTests is None:
113             self.calculateOptimum()
114
115         return self.__numTests
116
117
118     # Performs optimisation routine
119     # then sets __optValuation and __optScore
120     # (__numTests is set implicitly by __evaluate)
121     def calculateOptimum(self):
122
123         opt = self.__optimise(self.__vartree, {})
124
125         if opt is None:
126             self.__successful = False
127             self.__resetStoredVals()
128         else:
129             self.__optValuation, self.__optScore = opt
130             self.__successful = True
131
132         return None
133
134
135     # Checks if the optimisation was a success.
136     # This will only be false if no evaluations could be successfully performed.
137     def successful(self):
138         return self.__successful
139
140
141     # The actual optimisation function
142     def __optimise(self, vt, presets):
143
144         # First, calculate all the possible valuations at this node.
```

```
145             # (this is used in both the branch and leaf node cases)
146
147             # The valuations at this level are the cross product of the possible
                    values of the variables at this level.
148
149             # List of lists of (variable name,possible value) pairs (each sublist
                    deals with a single variable)
150             topLevelVarVals = [[(var, val) for val in self.__possValues[var]] for var
                     in vt.vars]
151
152             # List of dictionaries of possible tests (each dict contains a single
                    value for each var at this level)
153             topLevelTests = map(dict, crossproduct(topLevelVarVals))
154
155             # These dictionaries only contain mappings for variables at this level.
156             # So we merge the existing presets into topLevelTests
157             [t.update(presets) for t in topLevelTests] # (update topLevelTests in
                    place)
158
159
160
161             # Now split the branch and leaf cases.
162
163             if vt.subtrees != []: # Then vt.subtrees is nonempty and so vt is a
                    branch node.
164
165                 # For each valuation, we must optimise the subtrees,
166                 # then we can find the optimal valuation.
167
168                 # possValuations will store valuations for ALL variables, which have
                        optimised subtrees.
169                 possValuations = []
170
171                 for valuation in topLevelTests:
172
173                     # To optimise the subtrees, we must choose arbitrary values
174                     # for the variables in the other subtrees.
175                     # These are arbitrary because different subtrees are independent.
176
177                     valuation.update(self.__restrictArb(vt.flattenchildren(), self.
                            __possValues))
178
179                     # Now valuation contains mappings for ALL variables.
180                     # before testing each subtree, the variables in that subtree
                            should be removed from the valuation.
181
182                     for st in vt.subtrees:
183
184                         localValuation = valuation.copy()
185
186                         # Remove the variables in this subtree from the valuation
187                         for v in st.flatten():
188                             del localValuation[v]
189
190
191                         # Recursively optimise the subtree
192                         # the local optValuation returned here will be the same as
                                valuation
193                         # for all variables opther than those within st
194                         # so we can overwrite it here.
195                         # This also means we 'accumulate' an optimum valuation
                                overall.
196                         recurse = self.__optimise(st, localValuation)
197
198                         if recurse is None:
199                             return None # There are no valuations of the subtrees
                                    which can be successfuly evaluated.
200
201                         valuation, localOptScore = recurse
202
203                     # Because we overwrote valuation, it is now set so that
204                     # all subtree varaibles are set to their optimums
205                     # (for this valuation of vt)
206
```

```
207                 # So valuation now holds optimal settings for this choice of
                        variables st this level.
208                 # And localOptScore is set to the score of the optimum for this
                        choice at this level.
209
210                 possValuations.append(valuation)
211
212
213             # Once the loop is complete, possValuations contains one entry
214             # for each possible valuation at this level, but with the subtree
                    variables
215             # set to their optimums for that particular valuation at this level.
216
217             # First filter out any tests which failed (evaluate returns None).
218             possValuations = filter((lambda v: (v is not None) and (self.
                    __evaluate(v) is not None)), possValuations)
219             if possValuations == []:
220                 return None # There are no tests which evaluated correctly.
221
222             # Now choose the best to be returned.
223             optValuation = self.__best(possValuations, key=self.__evaluate)
224             optScore = self.__evaluate(optValuation)
225
226
227         else: # Then vt.subtrees is empty and so vt is a leaf node.
228
229             # Each valuation has all the variables set, we simply evaluate them
230             # The one with the minimum (or max) score is consisdered the best.
231
232             # First filter out any tests which failed (evaluate returns None).
233             topLevelTests = filter(lambda v: self.__evaluate(v) is not None,
                    topLevelTests)
234             if topLevelTests == []:
235                 return None # There are no tests which evaluated correctly.
236
237             # Now choose the best to be returned.
238             optValuation = self.__best(topLevelTests, key=self.__evaluate)
239             optScore = self.__evaluate(optValuation)
240
241
242         # In either case, we have found optimums for this level.
243
244         return (optValuation, optScore)
245
246
247
248     # Return a dictionary mapping each variable to one of its possible values
249     # in this case we choose the first one which was listed
250     def __restrictArb(self, vs, vals):
251         return dict([(var,vals[var][0]) for var in vs])
252
253
254
255
256 ################################################################################
257
258
259
260 # Return the cross product of a list of lists
261 def crossproduct(xss):
262     cp = [[]]
263     for xs in xss:
264         cp = [xs2 + [x] for x in xs for xs2 in cp]
265     return cp
266
267
268
269
270
271 if __name__ == "__main__":
272     print __doc__
```

vartree.py

```
"""
Autotuning System

vartree.py

Defines the VarTree class.
Provides a parser, vt_parse, for converting strings to instances of VarTree.
"""


# VarTree parser generated with wisent
from vartree_parser import Parser
# Built in regex based lexer
from re import Scanner




# The VarTree data type ############################################################

class VarTree:
    # vars is a list of strings of variable names at the current node
    # subtrees is a list of VarTree which are the children of the node

    def __init__(self, vars, subtrees):
        self.vars = vars
        self.subtrees = subtrees

    def __str__(self): # Convert the VarTree to a string representastion
        allStrs = self.vars + [st.__str__() for st in self.subtrees]
        return "{" + ", ".join(allStrs) + "}"

    def flatten(self): # Return a list of all the variables in the tree
        return self.vars + self.flattenchildren()

    def flattenchildren(self): # Return a list of all variables in child subtrees
        return sum([st.flatten() for st in self.subtrees], [])


################################################################################




# Some example VarTrees, used for testing ######################################

# {A, B, C, D}
sample0 = VarTree(["A", "B", "C", "D"], [])

# {A, B, {C, D}, {E, F}}
sample1 = VarTree(['A','B'], [VarTree(['C','D'],[]), VarTree(['E','F'],[])])

# {A, B, {I, {C, D}, {E, F}}, {G, H}}
sample2 = VarTree(["A","B"], [VarTree(["I"],[VarTree(["C","D"],[]), VarTree(["E",
    "F"],[])]), VarTree(["G","H"],[])])


################################################################################




# A helper function used to return a list of the variable names used in a (string
    ) VarTree
def get_variables(s):
    return vt_parse(s).flatten()




# The Lexer/Parser #############################################################
```

```
70
71
72  def vt_parse(str):
73
74      # We'll memoise this function so several calls on the same input don't
75      # require re-parsing.
76
77      if(str in vt_parse.memory):
78          return vt_parse.memory[str]
79
80
81      # Use the built in re.Scanner to tokenise the input string.
82
83      def s_lbrace(scanner, token):  return ("LBRACE", token)
84      def s_rbrace(scanner, token):  return ("RBRACE", token)
85      def s_comma(scanner, token):   return ("COMMA",  token)
86      def s_varname(scanner, token): return ("VAR",    token)
87
88      scanner = Scanner([
89          (r'{', s_lbrace),
90          (r'}', s_rbrace),
91          (r',', s_comma),
92          (r'[a-zA-Z_]\w*', s_varname),
93          (r'\s+', None)
94      ])
95
96      tokens = scanner.scan(str)
97
98      # tokens is a pair of the tokenised string and any "uneaten" part.
99      # check the entire string was eaten.
100
101     if(tokens[1] != ''):
102         print "Could not read the variable tree given:"
103         print str
104         #print "could not lex: " + tokens[1].__str__()
105         exit()
106
107
108     tokens = tokens[0] # Just the list of tokens.
109
110     p = Parser()
111     try:
112         tree = p.parse(tokens)
113     except p.ParseErrors, e:
114         print "Could not read the variable tree given:"
115         print str
116         exit()
117
118
119
120     # A function converting the parse tree to a VarTree.
121     def pt_to_vt(tree):
122
123         # If the current node is a VARTREE,
124         # then create a new VarTree object and fill it with the children of this
                node.
125         #
126         # If the current node is not a VARTREE, then something has gone wrong.
127
128         def is_var(t):
129             return t[0] == "VAR"
130
131         def is_vt(t):
132             return t[0] == "VARTREE" or t[0] == "VARTREE_BR"
133
134         if is_vt(tree):
135
136             vars = filter(is_var, tree[1:])
137             vars = map(lambda t: t[1], vars)
138
139             children = filter(is_vt, tree[1:])
140             children = map(pt_to_vt, children)
141
142             return VarTree(vars,children)
```

```
143
144         else:
145             return None # Should not be reached
146
147
148
149
150     # Put the result in the memoisation table.
151     vt_parse.memory[str] = pt_to_vt(tree)
152
153
154     # Check nothing went wrong
155     if vt_parse.memory[str] is None:
156         print "Could not read the variable tree given:"
157         print str
158         #print "error in conversion from parse tree to vartree"
159         exit()
160
161
162     # Finally, check there is no repettition of variables in the VarTree.
163     # Each variable can appear at most once.
164     def hasDups(xs):
165         return len(set(xs)) != len(xs)
166
167     if hasDups(vt_parse.memory[str].flatten()):
168         print "A variable was repeated in the variable tree."
169         print "Variables can only appear once."
170         exit()
171
172
173     return vt_parse.memory[str]
174
175
176 # Define the memoisation table, which will be satic inside vt_parse
177 vt_parse.memory = {}
178
179
180
181
182
183
184 ################################################################################
185
186
187
188
189
190
191 # A tree printer for VarTree ###################################################
192
193
194
195 # {A, B, {I, {C, D}, {E, F}}, {G, H}}
196 #
197 #           {A, B}
198 #             |
199 #         +---+---------+
200 #         |             |
201 #        {I}          {G, H}
202 #         |
203 #    +----+---+
204 #    |        |
205 # {C, D}   {E, F}
206
207
208
209
210 # N.B. this could easily be wider than the terminal.
211 def treeprint(vt):
212
213     return "\n".join(print_vt(vt)) + "\n"
214
215
216 def treeprint_str(s):
```

```
217     return treeprint(vt_parse(s))
218
219
220
221 # Trees are represented as a list of lines
222 def print_vt(vt):
223
224
225     if vt.subtrees: # Recursive case
226
227         # Recursively get subtrees
228         subtrees = map(print_vt, vt.subtrees)
229
230         # find the max height of a subtree
231         subtreeheight = len(max(subtrees, key=len))
232
233         # pads a subtree to be subtreeheight layers tall and then to be square
234         # (no ragged edge)
235         def padout(st):
236
237             # Add empty lines to the subtree until it is subtreeheight layers
                    tall
238             stheight = len(st)
239
240             newlines = [""] * (subtreeheight - stheight)
241
242             st2 = st + newlines
243
244             # Pad each line to be the same width
245             linewidth = len(max(st2, key=len))
246
247             st3 = [line + (" " * (linewidth - len(line))) for line in st2]
248
249             return st3
250
251         subtrees = map(padout, subtrees)
252
253
254         # Add connecting bars to the top of each subtree
255         subtrees = [["|".center(len(st[0]))] + st for st in subtrees]
256
257
258         # Stick the subtrees together
259         tree = subtrees[0]
260
261         for st in subtrees[1:]:
262             # Add st to tree, with a bit of padding.
263
264             tree = [tline + " " + app for (tline, app) in zip(tree, st)]
265
266
267
268         # Add the connecting branches above the subtrees
269
270         # The width of the whole tree
271         fullwidth = len(tree[0])
272
273
274         # Generate the wide connecting branch
275         # By looking at the top line of tree (which is now the upwards pointing
276         # connecting branches) and basically copying it.
277         connectingbranch = ""
278         numencountered = 0
279
280         for c in tree[0]:
281             if c == "|":
282                 connectingbranch += "+"
283                 numencountered += 1
284             else:
285                 if numencountered <= 0 or numencountered >= len(subtrees):
286                     connectingbranch += " "
287                 else:
288                     connectingbranch += "-"
289
```

```
290 
291 
292 
293         # Add this node on the very top
294 
295         topline = VarTree(vt.vars, []).__str__().center(fullwidth)
296         topconnector = "|".center(fullwidth)
297 
298 
299 
300         # Add one final "+" at the point in connectingbranch which will connect
                upwards.
301         # Again, we will cheat a little by copying the position of the | in
                topconnector.
302         midpoint = topconnector.find("|")
303 
304         connectingbranch = connectingbranch[:midpoint] + "+" + connectingbranch[
                midpoint+1:]
305 
306 
307 
308 
309 
310         # Put it all together to finish
311 
312         tree = [topline, topconnector, connectingbranch] + tree
313 
314 
315         return tree
316 
317 
318     else: # Base case
319 
320         return [" " + vt.__str__() + " "]
321 
322 
323 
324 
325 
326 ##############################################################################
327 
328 
329 
330 
331 
332 if __name__ == "__main__":
333     print __doc__
```

tune_conf.py

```
"""
Autotuning System

tune_conf.py

Sets up the configuration for the optimisation. The settings (variable names,
testing methods, etc.) are read from a configuration file provided.
"""


from ConfigParser import RawConfigParser
from vartree import get_variables


# The settings function takes the name of a configuration file, and returns the
# settings to be used for the optimisation.
def get_settings(configFile):


    # The settings dictionary will contain the configuration data.
    settings = {}



    # Read Configuration File ###########################################################

    config = RawConfigParser()

    config.read(configFile)


    # Must have sections [variables], [values] and [testing]
    if not(config.has_section('variables') and config.has_section('values') and
         config.has_section('testing')):
        print "Config file does not contain all the sections [variables], [values
            ] and [testing]."
        exit()

    # Must have option 'variables' in [variables]
    if not(config.has_option("variables", "variables")):
        print "Config file does not contain the option 'variables' in section [
            variables]."
        exit()


    # Get variable tree
    varTree = config.get("variables", "variables")

    settings['vartree'] = varTree

    variables = get_variables(varTree)


    # Must have the correct variables defined in [values]
    if not(all([config.has_option("values", v) for v in variables])):
        print "Config file does not contain possible values (in [values]) for all
             the variables defined in [variables]."
        exit()


    # possValues is a dictionary keyed by variable names [strings] with values
         which are lists of possible values [also string]
    possValues = {}

    for thisVar in variables:
        possValues[thisVar] = [x.strip() for x in config.get("values", thisVar).
            split(",")]

    settings['possValues'] = possValues


```

```
66
67      # Must have one of options 'test' or 'evaluation' in [testing]
68      if not(config.has_option("testing", "test") or config.has_option("testing", "
            evaluation")):
69          print "Config file does not contain either 'test' or 'evaluation' in
                section [testing]."
70          exit()
71
72      # Must have only one of options 'test' or 'evaluation' in [testing]
73      if (config.has_option("testing", "test") and config.has_option("testing", "
            evaluation")):
74          print "Config file contains both 'test' and 'evaluation' in section [
                testing], only one may be set."
75          exit()
76
77
78      # Set whichever of 'compiler', 'test', 'evaluation', 'cleanup' are present.
79      # Also build functions to create the actual compile (etc.) commands to run.
80      # Given a test ID n and a mapping of variables to values, returns an
81      # executable (by the shell) string.
82      if(config.has_option("testing", "compiler")):
83          settings['compiler'] = config.get("testing", "compiler")
84
85          def compiler_mkStr(n, varDict):
86              s = settings['compiler'].replace("%%ID%%", str(n))
87              for varName, varVal in varDict.iteritems():
88                  s = s.replace("%" + varName + "%", str(varVal))
89              return s
90
91          settings['compiler_mkStr'] = compiler_mkStr
92
93      if(config.has_option("testing", "test")):
94          settings['test'] = config.get("testing", "test")
95
96          def test_mkStr(n, varDict):
97              s = settings['test'].replace("%%ID%%", str(n))
98              for varName, varVal in varDict.iteritems():
99                  s = s.replace("%" + varName + "%", str(varVal))
100             return s
101
102         settings['test_mkStr'] = test_mkStr
103
104     if(config.has_option("testing", "evaluation")):
105         settings['evaluation'] = config.get("testing", "evaluation")
106
107         def evaluation_mkStr(n, varDict):
108             s = settings['evaluation'].replace("%%ID%%", str(n))
109             for varName, varVal in varDict.iteritems():
110                 s = s.replace("%" + varName + "%", str(varVal))
111             return s
112
113         settings['evaluation_mkStr'] = evaluation_mkStr
114
115     if(config.has_option("testing", "cleanup")):
116         settings['cleanup'] = config.get("testing", "cleanup")
117
118         def cleanup_mkStr(n, varDict):
119             s = settings['cleanup'].replace("%%ID%%", str(n))
120             for varName, varVal in varDict.iteritems():
121                 s = s.replace("%" + varName + "%", str(varVal))
122             return s
123
124         settings['cleanup_mkStr'] = cleanup_mkStr
125
126
127
128
129     # Check if they have chosen to maximise or minimise the FOM.
130     if(config.has_option("testing", "optimal")):
131         if(config.get("testing","optimal").lower() in ['max', 'min']):
132             settings['optimal'] = config.get("testing","optimal").lower()
133         else:
134             print "Config file contains an invalid setting for 'optimal' in
                    section [testing]."
```

```
135             exit()
136     else:
137         # Default to min
138         settings['optimal'] = 'min'
139 
140 
141 
142     # Check if they have set a number of tests to be run.
143     if(config.has_option("testing", "repeat")):
144         settings['repeat'] = int(config.get("testing","repeat"))
145 
146         # Check for invalid input
147         if settings['repeat'] < 1:
148             settings['repeat'] = 1
149     else:
150         # Default to 1
151         settings['repeat'] = 1
152 
153 
154 
155 
156     # Check how they want to aggregate multiple runs of tests.
157     if(config.has_option("testing", "overall")):
158         if(config.get("testing","overall").lower() in ['max', 'min', 'med', 'avg'
                ]):
159             settings['overall'] = config.get("testing","overall").lower()
160         else:
161             print "Config file contains an invalid setting for 'overall' in
                    section [testing]."
162             exit()
163     else:
164         # Default to min
165         settings['optimal'] = 'min'
166 
167 
168 
169 
170     # Check if they want to log tests.
171     if(config.has_option("testing", "log")):
172         settings['log'] = config.get("testing","log")
173 
174     else:
175         # Default to None
176         settings['log'] = None
177 
178 
179 
180 
181 
182 
183 
184     return settings
185 
186 
187 
188 
189 
190 
191 
192 
193 
194 
195 
196 
197 
198 
199 if __name__ == "__main__":
200     print __doc__
```

logging.py

```
"""
Autotuning System

logging.py

Used to keep a log of tests being run, and to output these logs.

Does not directly interface with the optimisation, but tests can be logged as
they are run from the evaluate() function.
"""


# usage:
#
# Create a test (by passing a new unique test id)
# Add one or more test results with logTest
# Add the overall result with logFinal
#
# The tests can then be dumped out with writeCSV.


class TestLog:

    def __init__(self, pr=False):
        self.__tests = {}
        self.__vars = set()
        self.__printResult = pr


    def createTest(self, testId, valuation):
        self.__tests[testId] = SingleTest(testId, valuation)
        self.__vars = self.__vars.union(set(valuation.keys()))


    def logTest(self, testId, score):
        if testId in self.__tests:
            self.__tests[testId].results.append(score)


    def logOverall(self, testId, score):
        if testId in self.__tests:
            self.__tests[testId].overall = score


    def writeCSV(self, filename):

        # Open the file
        try:
            f = open(filename, 'w')
        except IOError:
            print "Could not open log file for writing."
            return

        # List of variables, because we need ordering here
        vars = list(self.__vars)

        # Number of results per test
        nResults = 0
        for t in self.__tests.values():
            nResults = max(nResults, len(t.results))


        # Create title line
        f.write(", ".join(["TestNo"] + vars + ["Score_"+str(n) for n in range(1,
             nResults+1)] + ["Score_Overall"]) + "\n")

        # Create a string for each line
        for k, t in sorted(self.__tests.iteritems()):

            # Add test no.
            l = [str(t.testId)]
```

```
71
72              # Add variable values
73              for v in vars:
74                  if v in t.valuation:
75                      l.append(str(t.valuation[v]))
76                  else:
77                      l.append('')
78
79              # Add scores
80              l += [str(x) for x in t.results]
81              l += [""] * (nResults - len(t.results))
82
83              # Add overall score
84              if t.overall is None:
85                  l.append('')
86              else:
87                  l.append(str(t.overall))
88
89              # Finished line
90              f.write(", ".join(l) + "\n")
91
92
93          # Done
94          f.close()
95
96          if self.__printResult:
97              print "A testing log was saved to " + filename
98
99
100
101
102
103 class SingleTest:
104
105     def __init__(self, testId, valuation):
106         self.testId = testId
107         self.valuation = valuation
108         self.results = []
109         self.overall = None
```

airfoil_autotuning.conf

```
# Autotuning System
# Configuration file for tuning airfoil code.

[variables]

# This section must contain a definition of variables, with a list of variable names
# These variable names will be varied over to find an optimal valuation.
# The possible values for each variable are specified in the next section.

variables = {L1_CACHE, {OP_PART_SIZE_1}, {OP_PART_SIZE_2}}


[values]

# This section gives the possible values that each variable above can take.
# These are specified as a list for each variable.

OP_PART_SIZE_1 = 32, 128, 512, 1024, 1536
OP_PART_SIZE_2 = 64, 128, 192, 256, 320, 384, 448, 512, 576, 640, 704, 768, 832, 896, 960, 1024

L1_CACHE       = cg, ca


[testing]

# This section defines how to compile (if needed) and run the tests.
#
# 1. How to compile each test (this would typically be a makefile or similar)
# 2. How to check each test. This might be a shell script or similar.
#       This should take the name of the executable as an argument and return
#       some figure-of-merit for that test (typically execution time).
# 3. How to clean up each test (typically deleting the test file).


# compiler
#
# This sets the command to issue to compile the program
# e.g. a makefile or similar
#
# Use %%ID%% for the test number, to generate unique files.
# You may also use your variables defined above e.g. %FOO%, %BAR% and so on
#
# This may be omitted, in which case no compilation will be performed.

compiler = make -f Makefile_Autotuning -B airfoil_cuda OP_PART_SIZE_1=%OP_PART_SIZE_1% OP_PART_SIZE_2=%OP_PART_SIZE_2% L1_CACHE=%L1_CACHE%


# testing
#
# Exactly one of the options 'test' or 'evaluate' should be set.

# test
# The command used to run a test.
# When this is used, the Autotuning system will time the execution of the test
# and use this timing to optimise the variables.
# e.g. the executable produced by the compilation step above.
#
# Use %%ID%% for the test number, so each test is unique.
# You may also use your variables defined above e.g. %FOO%, %BAR% and so on

test = ./airfoil_cuda


# evaluate
# This allows you to define your own figure of merit for the tests.
# This command is run and the final line of output is assumed to be a number
# which will be used to optimise the variables.
# e.g. a command returning the size of the executable generated.
```

```
69  #
70  # Use %%ID%% for the test number, so each test is unique.
71  # You may also use your variables defined above e.g. %FOO%, %BAR% and so on
72
73  #evaluation =
74
75
76  # cleanup
77  #
78  # This sets the command to be run to clean up any tests, if required.
79  # e.g. removing the executables created by the compilation step.
80  #
81  # Use %%ID%% for the test number, so each test is unique.
82  # You may also use your variables defined above e.g. %FOO%, %BAR% and so on
83  #
84  # This may be omitted, in which case no cleanup will be performed.
85
86  #cleanup =
87
88
89  # optimal
90  #
91  # This setting determines whether large or small results are considered better.
92  # It can be set to 'min' (the default), or 'max'.
93  # 'min' will cause the optimiser to find the minimum possible score.
94  # 'max' will cause the optimiser to find the maximum possible score.
95  #
96  # For timing, it would be normal to use 'min' to find the shortest time.
97  # However, it may be useful to use 'max' if you have some custom figure of merit
98  # which should be maximised.
99
100 optimal = Min
101
102
103 # repeat
104 #
105 # This setting gives the number of times each test should be run.
106 # It is optional and defaults to 1.
107 # The average score from all the tests will be taken and used as the score for
108 # the test overall.
109
110 repeat = 3
111
112
113 # overall
114 #
115 # If multiple runs are being used for each test (using 'repeat'), then this
116 # option defines how the multiple scores should be combined into the overall
117 # score for that test.
118 #
119 # The possible values are 'min' (default), 'max', 'med' and 'avg'
120 # These will take the minimum, maximum, median and average, repsectively of all
121 # the repeated test scores and use it as the overall score by which each test
122 # is judged.
123
124 overall = min
125
126
127 # log
128 #
129 # If defined, this is the name of a CSV file which a log of the tests
130 # performed will be written to.
131 # If not defined, no log will be saved.
132 #
133 # This file will be overwritten!
134
135 log = ../airfoil_autotuning.csv
```