Ideas for auto-tuning

Mike Giles

September 11, 2010

Abstract

This document outlines my ideas on developing an auto-tuning framework. The initial motivation is the OP2 implementation on GPUs, where it is very hard, even for an expert programmer, to predict the best way to implement algorithms and the best values to use for various parameters (such as number of thread blocks and the number of threads per block). I suspect that auto-tuning could deliver performance gains of factor 2 over the initial implementation of an expert programmer.

Longer-term, I think that CPU programming is also becoming more complex, and it will become harder to achieve good execution efficiency. Auto-tuning may be an important way to reduce this performance gap.

1 A bottom-up view

1.1 Parametrized codes

Programs can be parametrized in various ways.

In my current OP2 implementation on GPUs, I am planning to use the following parameters:

- Size of block partition, and hence the number of threads blocks
- Number of threads per block
- Whether to use thread coloring or atomic increments
- Whether to use registers or shared-memory pointers

and there will be a set of these for each parallel loop in the program.

The key observations here are

- As an expert programmer I know what the critical parameters are, but I don't know their optimal value
- The parameters are either boolean or integers with a very limited set of values (I think I could do well with no more that 4 possible values)

At the practical level, these parameters are embedded within the C/C++ code as variables which can be set through a `#define` statement or through a `-Dparam=value` compiler flag.

1.2 Parametrized compilation and execution

In addition to parameters within the code, there are also parameters at the compilation level (e.g. choice of compiler flags, and even choice of compilers).

Within a large-scale enterprise there might also be parameters at the execution level, such as

- whether the application is run on an Intel system or a AMD system
- whether it is better to run two 4-thread jobs at the same time, or two 8-thread jobs, one after the other

1.3 Optimization

Finally, we have a heavily parametrized application, and the goal with auto-tuning is to select the best parameter values. If there are relatively few parameters and possible parameter values then exhaustively checking all combinations may be feasible. In other cases, this would be prohibitively expensive and so some form of optimization technique will be required.

The important thing for me is to develop an approach, and software, which is very generic and able to address parameter optimization at the various different levels discussed above. I think this is the key distinguishing feature of my objectives compared to what has been done previously.

2 A top-down abstract view

I think a clean software design can often come from looking at a problem and constructing a clear abstraction which contains the essence of the problem to be addressed.

In this case, I think an appropriate abstraction has 4 elements:

- a figure of merit

  This is the quantity to be minimized (or maximized) by the optimization. In my current GPU work, it would be the execution time, but if one was choosing which machine to execute the application on then the figure of merit may be the overall cost of the computation.

- a set of parameters and possible parameter values

  The user clearly needs to specify the set of parameters for the optimization, and the set of possible values which each one can take. Because high-dimensional optimization can be extremely expensive, I think it is important that the user should also specify which parameters can be optimized independently. For example, in my GPU work, the number of threads used in one GPU kernel is independent of the number of threads in another kernel and so the value of each can be optimized independently.

  A possible syntax for specifying this would be:

  ```
  { param1, param2, {param3, param4}, {param5, param6} }
  ```

  signifying that the optimal values for `param3, param4` depend on each other and `param1, param2` but not on `param5, param6`.

- an evaluation mechanism

  The auto-tuning framework needs a mechanism to obtain the figure of merit for a particular set of parameter values. The optimization will need to cope with the fact that this evaluation mechanism is unlikely to be precisely repeatable (e.g. the execution time will inevitably vary a bit when the code is re-run). Also, the framework will need to be told whether it can run several evaluations at the same time.

- an optimization methodology

  Finally, we need a method to carry out the optimization. As stated previously, in simple cases this might be brute-force optimization, trying every combination of parameter values, but in more complex cases it might require something like genetic algorithm optimization.

Figure 1 gives a concrete example of a possible configuration file for defining an auto-tuning optimization.

```
#
# declare parameters
#

PARAMS = { flags, {thread1, atomics1}, {thread2, atomic2} }

#
# declare possible parameter values
#

flags   = {-O2, -O3}    # compiler flags
thread1 = {32, 64, 96}  # number of threads per block in loop 1
atomic1 = {0, 1}        # boolean for atomic updates in loop 1
thread2 = {32, 64, 96}  # number of threads per block in loop 2
atomic2 = {0, 1}        # boolean for atomic updates in loop 2

#
# declare files to be used
#

PARAMETERS_FILE      = { params.dat }
FIGURE_OF_MERIT_FILE = { f_o_m.dat }

#
# declare evaluation mechanism
#

EVALUATION = { make; ./executable }

#
# declare optimization strategy and how many
# evaluations can be done in parallel
#

STRATEGY = { brute_force parallel:4 }
```

Figure 1: An example of a possible configuration file