The Intel®Software Autotuning Tool (ISAT)

User Manual (version 1.0.0)

Chi-Keung Luk
Technology Pathfinding and Innovation
DPD/SSG
Intel Corporation

November 30, 2010

1 Introduction

Autotuning is [3, 2, 1] is an approach for producing efficient and portable codes. It works by generating many different variants of the same code and then empirically finding the best performing variant on the target machine. *The Intel®Software Autotuning Tool (ISAT)* is a tool being developed to facilitate autotuning. Version 1.0 supports *program parameter tuning* of C/C++ programs. Future versions would support additional kinds of tuning. This manual describes how to use this tool.

2 System Requirements

ISAT currently runs on Linux (both 32 and 64 bit), and it requires the following packages pre-installed:

1. Python version >= 2.4 but < 3.0
2. Gnuplot (optional, only needed if you want to plot your tuning results)

3 Compiler Compatibility

ISAT is implemented in Python using source-to-source translation. As a result, ISAT can be used on top of any native C/C++ compilers.

4 How to Install ISAT

After your download the ISAT tarball (`isat.tar.gz`), simply untar it into a directory of your choice (`tar -xzvf isat.tar.gz`). Then invoke `config.sh` in the top level directory and follow the directions as prompted.

In the rest of this manual, we assume that ISAT is installed right under your home directory at `~/isat`.

5 Preparing Your Program for ISAT

In order to use ISAT to tune your program, you need to have its source code. You also need to add three scripts to the directory where the source is located: `isat-clean`, `isat-build`, `isat-test` (you can change the names of these scripts using the isat command line; see Section 11 for the switch names). `isat-clean` will be invoked by ISAT to clean up the build environment before the build. `isat-build` will be invoked by ISAT to build the executable from the source. `isat-test` will be invoked by ISAT to run the executable and measure its performance. In many cases, each of these scripts contains simply an one-line command like 'make clean', 'make build', or 'make run'. Examples of these scripts can be found in `~/isat/Examples/BlockedMatrixMultiply/Src/`.

6 How to Invoke ISAT

The simplest way to invoke ISAT is as follows:

```
$~/isat/Source/Python/isat.py -i Src -o Dst
```

where `./Src` is the directory that contains the source files of the program that you want to tune. The tuned version will be generated in `./Dst`.

Please look at the examples in `~/isat/Examples/`. The first example to look at should be `BlockedMatrixMultiply/`. Follow the directions in `BlockedMatrixMultiply/README` file to learn how to apply ISAT to this example.

7 ISAT Pragmas

New pragmas are introduced by ISAT to allow a programmer to specify *where* in the program needs tuning and *how* the tuning should be done. These pragmas are of the form `#pragma isat ...`. There are two types of ISAT pragmas: `marker` and `tuning`. A `marker` pragma marks a location in the program while a `tuning` pragma specifies the tuning details.

As an example, let's look at `~/isat/Examples/BlockedMatrixMultiply/Src/blocked_mm.cpp`. In this example, we use ISAT to find the best block sizes in a blocked matrix-multiply program:

```
static void BlockedMatrixMultiply(FLOAT* A, FLOAT* B, FLOAT* C, const int n)
{
#pragma isat tuning name(tune_mm) scope(M1_begin, M1_end) measure(M2_begin, M2_end) variable(blk_k, rang
, $L1_CACHE_LINESIZE/2)) variable(blk_j, range($L1_CACHE_LINESIZE, 16*$L1_CACHE_LINESIZE, $L1_CACHE_LIN
NESIZE, 16*$L1_CACHE_LINESIZE, $L1_CACHE_LINESIZE/2))

#pragma isat marker M1_begin
    const int blk_i= 64;
    const int blk_j= 64;
    const int blk_k= 64;

    for (int i=0; i<n; i += blk_i)
        for (int j=0; j<n; j += blk_j)
            for (int k=0; k<n; k += blk_k)
                for (int ii=i; ii<MIN(i+blk_i, n); ii++)
                    for (int jj=j; jj<MIN(j+blk_j, n); jj++)
                        for (int kk=k; kk<MIN(k+blk_k, n); kk++)
                            C[ii*n+jj] = C[ii*n + jj] + A[ii*n+kk]*B[kk*n+jj];
#pragma isat marker M1_end
}

main(int argc, char** argv)
{
...
#pragma isat marker M2_begin
    BlockedMatrixMultiply(A, B, C, N);
#pragma isat marker M2_end
...
}
```

We use `marker` pragmas to define two regions in the source code: `(M1_begin, M1_end)` and `(M2_begin, M2_end)`. These regions are referenced in the `tuning` pragma, which contains a `name` clause, `scope` clause, a `measure` clause, and three `variable` clauses. A `name` clause specifies the name of region being tuned, which shall be referenced in the tuning report. A `scope` clause specifies the lexical scope that contains the program parameters being tuned. A `measure` clause specifies where we should time a code variant.

A `scope` clause usually has two arguments to specify the starting and ending points of the region being tuned. The two points must be in the same source file. Nevertheless, one can use `scope(*)` to specify that the lexical scope includes every source file.

A `measure` clause must have two arguments to specify the starting and ending points of the region being measured.

A `variable` clause has two arguments. The first argument is the variable that requires tuning. The second argument is the list of values that should be tried for the variable. A list can be in the *enumerated* form or in the *formula* form.

The enumerated form explicitly lists the values of a variable that should be tried such as `[2, 4, 6]` and `[red, blue, yellow, green]`. Note that an identifier like `red` or `blue` can be used in an enumerated list only if it is valid to substitute for the variable in the original program.

A list in the formula form is constructed by the `range()` method defined in ISAT, which accepts two to four arguments:

`range`(`beginExpr`, `endExpr`, `strideExpr`, `fun`)

The first argument `beginExpr` is an expression that calculates the beginning value of the list. The second argument `endExpr` is an expression that calculates the ending value of the list. Following C notation, the beginning value is inclusive while the ending value is exclusive. The third argument `strideExpr` is optional, which is an expression that calculates the step value. If it is not given, a step value of one will be used. The fourth argument `fun` is also optional, which is the name of the function that will be applied to each list element to get the actual value to be substituted for the variable. Currently, `fun` must be `identical`, `pow2`, or `pow10` (it is default to `identical` if not specified). Followings are some examples:

```
range(1, 6)            // generates [1, 2, 3, 4, 5]
range(1, 6, 2)         // generates [1, 3, 5]
range(1, 6, 1, pow2)   // generates [2, 4, 8, 16, 32]
range(1, 6, 1, pow10)  // generates [10, 100, 1000, 10000, 100000]
range($L1_CACHE_LINESIZE, 4*$L1_CACHE_LINESIZE+1, $L1_CACHE_LINESIZE/2)
   // Assuming that $L1_CACHE_LINESIZE is 64, the above range expression
   // generates [64, 96, 128, 160, 192, 224, 256]
```

As the last example illustrates, ISAT-predefined values (those started with `$`) can be used in range expressions to represent system-dependent parameters. Table 1 lists all predefined values with their meanings:

Name	Meaning
`$NUM_CPU_THREADS`	Total number of hardware threads on the CPU
`$L1_CACHE_LINESIZE`	L1 cache line size in bytes

Table 1: ISAT pre-defined values

Finally, note that a program with ISAT pragmas can be built by any native compiler—the ISAT pragmas will be simply ignored by the compiler.

8 Searching Scheme

When there are multiple parameters being tuned in a `tuning` pragma, the programmer can select the scheme used for searching the parameter space. Currently, ISAT supports two schemes: `independent` search and `dependent` search. The search scheme can be specified in a `tuning` by adding a `search` clause. For instance, you can choose to use independent search in the following pragma:

```
#pragma isat tuning scope(M1_begin, M1_end) measure(M2_begin, M2_end)
variable(x, [10,20,30]) variable(y, [70, 80, 90, 100]) search(independent)
```

For the above pragma, ISAT will first search for `x` from `[10, 20, 30]` that results in the minimum time measured between `M2_begin` and `M2_end`. Then it will search for `y` from `[70, 80, 90, 100]`. And ISAT will use the best value of `x` it just found while searching `y`. So, the total number of code variants tried is the sum of the code variants of each variable, which is 3+4 in this example.

If you change the search scheme from independent to dependent in the above pragma,

```
#pragma isat tuning scope(M1_begin, M1_end) measure(M2_begin, M2_end)
variable(x, [10,20,30]) variable(y, [70, 80, 90, 100]) search(dependent)
```

Then ISAT will search for the best performing `(x,y)` from all possible combinations: (10, 70), (10, 80), (10, 90), (10, 100), (20, 70), (20, 80), (20, 90), (20, 100), (30, 70), (30, 80), (30, 90), (30, 100).

If no searching scheme is specified in a `tuning` pragma, independent search will be used.

9 Threading-library Specific Tuning

ISAT has built-in support to tune certain parameters in the OpenMP and TBB APIs, as described below:

9.1 OpenMP

ISAT supports the tuning of the scheduling parameters in OpenMP parallel loops (`parallel do` and `parallel for`). The parameters are the scheduling type and the chunk size, which can be tuned independently or dependently. These parameters can be explicitly or implicitly defined in `parallel` statements. When they are implicitly defined, you can refer them as `@omp_schedule_type` and `@omp_schedule_chunk` in a ISAT tuning pragma. See `~/isat/Examples/Omp/Src` for an example.

9.2 TBB

ISAT supports the tuning of the grain size in TBB `parallel_for`, `parallel_reduce`, and `parallel_scan`. The grain size is the last parameter in a `blocked_range`. It can be explicitly or implicitly defined. When it is implicitly defined, you can refer it as `@tbb_grain_size` in a ISAT tuning pragma. See `~/isat/Examples/Tbb/Src` for an example.

10 Using the Output of ISAT

ISAT puts the tuning results into the output directory that you specified via the "-o" option when invoking ISAT. The output directory is essentially a copy of the input directory (the one you specified via the "-i" option) except that source files in the output directory use the optimal values of all parameters tuned by ISAT. So, if you simply do a build in the output directory, the resulting executable will use the tuned values.

ISAT also generates a tuning report with the default name `isat-report.txt` in the top level of the output directory. From this report, you can find out the performance of all trial runs.

10.1 Plotting the Tuning Results

You can plot the results in `isat-report.txt` using the `xgraph` tool in the ISAT package as follows:

```
$~/isat/Source/Python/xgraph.py -i isat-report.txt -o graphs
```

The graphs generated are put in the directory `./graphs`. A 2D graph like Figure 1(a) will be generated for an independently tuned variable. A 3D temperature graph like Figure 1(b) will be generated for a pair of dependent variables (if there are more than two dependent variables, a 3D graph will be generated for each distinct pair of dependent variables).

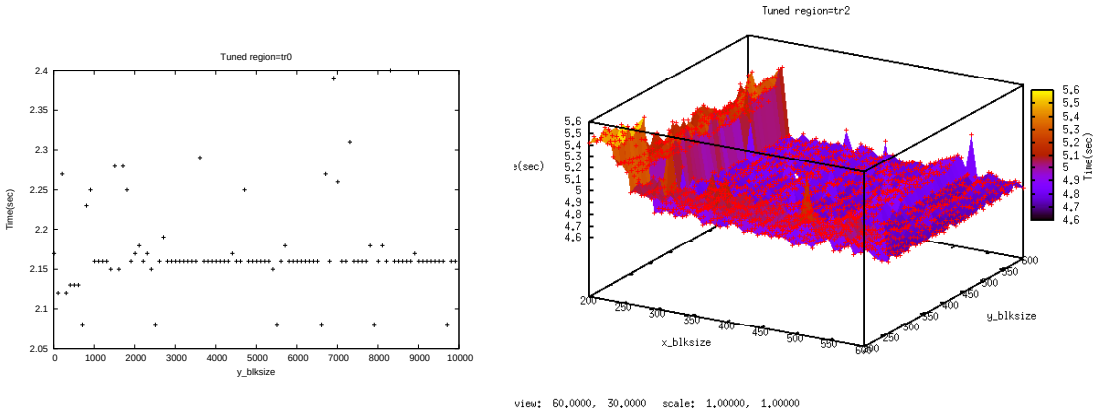
11 ISAT Command-line Switches

You can do "`isat -h`" to see all the command-line switches of ISAT. Table 2 lists these switches and their meanings.

Switch	Description
`-i`	Input directory that contains the untuned version
`-o`	Output directory that contains the tuned version
`--nt`	Number of test runs per code variant (default = 1)
`--rp`	Tuning report name (default = "isat-report.txt")
`--mainfile`	Name of the file that contains `main()`
`--clean_command`	The command that ISAT calls to clean old executables (default="isat-clean")
`--build_command`	The command that ISAT calls to build the executable (default="isat-build")
`--test_command`	The command that ISAT calls to run and time the executable (default="isat-test")
`--ignore_buggy_testrun`	If a particular code variant results into errors, ignore it and continue trying other code variants
`--no_interactive`	Proceed the tuning without waiting for user's response

Table 2: ISAT command-line switches

Followings are more explanations on when you may need to use some of these switches:



(a) 2D tuning graph (b) 3D tuning graph

Figure 1: Examples of plotting ISAT results

`--mainfile` ISAT needs to know which source file contains the main function. It will try to figure this out itself by looking for the unique file that contains `main()` in the given source directory. However, if there are multiple files that contains `main()` (only one of them is actually linked in), ISAT needs you to specify the main file.

`--ignore_buggy_testrun` It is possible that some values tried for a particular variable being tuned may be invalid for that variable, resulting into errors like segmentation faults. By default, ISAT will stop tuning when such errors occur. However, you can use the `ignore_buggy_testrun` option to force ISAT to continue the tuning after encountering an error.

12 Debugging ISAT yourself

To debug ISAT, the first thing to do is to look at logging/warning/error messages generated by ISAT. These messages are with the prefix "`@@@ ISAT-`" and are output to the console.

The ultimate way to debug ISAT is to inspect the code variants that it generates. When you run ISAT, it will print the name of the working directory on the console, such as:

```
@@@ ISAT-log: Generate top working directory: "/tmp_proj/cluk1/ISAT/Src-f7ZOdy"
```

Now go to that working directory. There should be a list of subdirectories `v0/`, `v1/`, ..., `v99/` etc under the top working directory. Each of these subdirectories contains a code variant that ISAT generated. Go to one of them (or the one where ISAT's error happened as reported by the ISAT warning/error message). That subdirectory is a copy of the original source directory (which you specified with the `-i` switch) with a parameter being tuned set to a particular value.

Now, type "isat-clean" (or the command you specified with `--clean_command`) and then type "isat-build" (or the command you specified with `--build_command`). You should be able to build the executable. If there is any compilation or linking error, look at the error message to understand what ISAT did wrongly in generating the code variant that caused the build error. If the build is successful, then type "isat-test" (or the command you specified with `--test_command`). Then check if the execution is successful and generates the expected result. If everything is correct, there should be a file named `isat-feedback.out` generated after running this code variant which contains the timing of this run.

References

[1] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[2] M. Puschel, J. Moura, J. Johnson, D. Pauda, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaption*, 93(2):232–275, 2005.

[3] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.