Les Cahiers du GERAD ISSN: 0711–2440

Templating and Automatic Code Generation for Performance with Python

D. Orban

G–2011–30

June 2011

Les textes publiés dans la série des rapports de recherche HEC n'engagent que la responsabilité de leurs auteurs. La publication de ces rapports de recherche bénéficie d'une subvention du Fonds québécois de la recherche sur la nature et les technologies.

Templating and Automatic Code Generation for Performance with Python

Dominique Orban

GERAD and Department of Mathematics and Industrial Engineering
École Polytechnique de Montréal
C.P. 6079, Succ. Centre-ville
Montréal (Québec) Canada
H3C 3A7
dominique.orban@gerad.ca

June 2011

Les Cahiers du GERAD

G–2011–30

Copyright © 2011 GERAD

Abstract

Parameterizing source code for architecture-bound optimization is a common approach to high-performance programming but one that makes the programmer's task arduous and the resulting code difficult to maintain. Certain parameterizations, such as changing loop order, may require elaborate code instrumenting that distract from the main objective. In this paper, we propose a templating and automatic code generation approach based on standard Python modules and the Opal library for algorithm optimization. Advantages of our approach include its programmatic simplicity and the flexibility offered by the templating engine. We provide a complete example for the matrix multiply where optimization with respect to blocking, loop unrolling and compiler flags takes place.

Key Words: Algorithm optimization, nonsmooth optimization, template programming, automatic code generation.

Résumé

La paramétrisation de code source pour l'optimisation vis-à-vis de la plate-forme informatique est une approche courante en calcul de haute performance mais elle complique la tâche du programmeur et le code résultant est difficile à entretenir. Certaines paramétrisations, telles qu'un changement d'ordre de boucles, peuvent demander des annotations complexes du code qui éloignent l'attention du but principal. Dans cet article, nous proposons une approche de particularisation de canevas et sur de génération automatique de code basée sur des modules Python standard et sur la librairie Opal d'optimisation d'algorithmes. Parmi les avantages de notre approche, nous citons la simplicité de programmation et la flexibilité du moteur de particularisation de canevas. Nous illustrons notre approche sur un exemple complet pour la multiplication matricielle où l'optimisation se fait sur les paramètres de blocage, le déroulage de boucles et les options de compilation.

Acknowledgments: Research partially supported by NSERC Discovery Grant 299010-04.

1 Introduction

Given an algorithm, the OPAL library [1, 2] casts the algorithmic parameter tuning problem as the nonsmooth optimization problem

$$\underset{p \in \mathbf{P}}{\text{minimize}} \;\; \phi(p) \quad \text{subject to} \;\; \psi_i(p) \in \mathbf{M}, \quad (i = 1, \ldots, m), \tag{1}$$

where p is the set of parameters under consideration, $\mathbf{P}$ is the domain of p as given by the algorithm specifications, ϕ and each ψ_i are so-called *composite measures* and $\mathbf{M}$ is a user-defined feasible set. In a typical situation, running the algorithm with representative input data produces a collection of *atomic measures* which depend upon the parameter values chosen by the user. Composite measures are arbitrary combinations of atomic measures. In computer architecture-bound tuning, relevant atomic measures may include the elapsed time, the amount of memory consumed, the throughput, or any other relevant performance metric. The objective and constraints of (1) are usually nonsmooth and may be noisy. In OPAL, (1) is solved by way of the direct-search solver NOMAD [3], an implementation of the mesh-adaptive direct-search framework (MADS) [4]. The MADS framework is supported by strong convergence guarantees and does not fit in the class of empirical or heuristic methods.

In this paper, we are interested in the application of OPAL to the automated tuning of high-performance numerical libraries. The vast diversity of computer architectures makes the implementation of such libraries notoriously challenging. Though some appropriate optimizations may be performed by modern compilers, they are often entirely heuristic and insufficient. In the recent past, researchers have turned to code parameterization to facilitate automated tuning. The prime successful example of this approach is the ATLAS [5] implementation of the BLAS [6] dense linear algebra kernels. Various approaches are possible and most consist in instrumenting the source code by way of customized annotations (such as C `pragmas`) or of an augmenting scripting language. POET [7] is an XML-type augmenting scripting language used to wrap the source code and to describe parameters and how they can be used to perform source transformations. POET scripts can be rather lengthy and contain numerous cryptic symbols that make the code difficult to read and maintain. PLW [8] is an augmenting scripting language defined as a restricted subset of Python[1]. PLW annotations can provide a Python script with static typing and assist in porting to a parallel platform. Some of PLW's goals are now covered by the increasingly popular Cython[2] extension of Python which provides static typing and seamless interfacing with external C, C++ and Fortran libraries. The X language [9] is an annotation-based system aiming to support multiple versions of a program with empirical tuning in mind. A disadvantage of the X language is the restricted set of rules that can be used for code transformation. Finally, the ROSE [10] compiler allows to define rules for source-to-source transformation by code fragment substitution.

We propose a simpler approach that does not necessitate the development of a new augmenting language or of a customized compiler designed specifically for the parameter optimization task. Instead, our Python-based approach relies on established standard Python tools each focusing on a narrow specific goal. Code is written as a set of *templates* which support some functional programming constructs such as tests and loops. Instances of templates may be *rendered* by specifying concrete parameter values. In our examples we use C as the base low-level language but other languages, such as Fortran or Python, could be used. The resulting instance may be compiled on-the-fly as a shared library and dynamically imported into the current Python session. When wrapped into an OPAL description script, the result is a Python-based toolchain for the automatic generation of optimized code based on templates.

Code parameterization is by no means new or even recent. The use of optimization for stability analysis of computational methods may be traced back to the mid-70's. The authors of [11, 12, 13, 14] devise languages in which numerical algorithms are to be implemented. Upon compilation, a descent method exercises the algorithm by varying its input so as to maximize an error measure with the intent of assessing the numerical stability of the method as implemented. The programming languages impose a number of stringent rules on the implementation which, for instance, may not make use of loops.

[1] `www.python.org`
[2] `www.cython.org`

The rest of this paper is organized as follows. Section 2 gives an overview of the algorithm to be optimized and the OPAL modeling and solving environment, and §3 describes code parameterization via templates and how they are used to generate code automatically. Numerical experience is reported in §4 and further research is discussed in §5.

2 The OPAL Framework and the Target Algorithm

OPAL is a versatile Python framework providing tools to formulate a parameter optimization problem based on a description of the target algorithm, the name and type of its parameters, and the atomic measures available after running the algorithm on sample data. We describe the modeling process below after a brief discussion of our target algorithm.

The dense matrix-multiply is a crucial linear algebra kernel and much attention has been, and still is, given to tuning it to the host architecture. A number of search heuristics are compared in [15] but in the present work, we propose a search based on a black-box solver with strong convergence guarantees. The Level-3 BLAS matrix multiply performs several types of update, among which $C = AB + \beta C$, where $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times p}$, $B \in \mathbb{R}^{p\times n}$ and $\beta \in \mathbb{R}$. For simplicity, we only consider real matrices. In order to avoid unnecessary memory reads or scalar multiplies, different kernels are usually written depending on whether $\beta = 0$, $\beta = 1$, or $\beta \notin \{0, 1\}$. Several parameters can be considered. Three of them are related to *blocking*, which consists in performing the multiply by updating entire submatrices of C instead of a single element at a time. Blocking leads to a more efficient implementation than the straightforward three-loop implementation because it increases the probability of cache hits and decreases that of code branch misprediction. Suppose C is decomposed into blocks $C_{[ij]}$, each of size m_b-by-n_b except perhaps those in the last block row and column, and that A and B are similarly decomposed into blocks of size m_b-by-k_b and k_b-by-n_b, respectively. Updating $C_{[ij]}$ can now be performed via $C_{[ij]} = (A_{[i1]}B_{[1j]} + \beta C_{[ij]}) + \sum_{k\geq 2} A_{[ik]}B_{[kj]}$. In other words, we need a single call to a kernel using the user-requested value of β and operating on blocks of size $m_b \times n_b$, $m_b \times k_b$ and $k_b \times n_b$, respectively, followed by several calls to a similar kernel using $\beta = 1$. If smaller blocks are encountered in the last block row or column, a specific kernel, referred to as a *cleanup kernel* in [16], is called. In our implementation, we do not use blocking in the cleanup kernel, though it would be possible. In ATLAS, blocking parameters are bounded between 16 and $\min(\sqrt{L_1}, 80)$, where L_1 is the size of the L_1 cache. Our implementation relies on the optimization engine and lets blocking parameters vary between 1 and the row or column size. Three additional parameters concern *loop unrolling* and they are the extent to which each loop i, j or k, is unrolled. This determines the extent to which a code fragment will be vectorized. Compilers typically perform some amount of loop unrolling based on heuristics but the optimal amount is architecture dependent. Unrolling amounts vary between zero and the number of elements in the loop. A last set of parameters is that of compiler flags. We describe the flags used in our tests in §4. Other parameters, such as *loop order*, are explored by heuristic search in [15].

Modeling a parameter optimization problem in OPAL consists in writing three Python files which we now briefly describe. The *declaration* file contains a description of the parameters of the target algorithms along with their type and bounds, if any. In our case, those are the blocking factors n_b, m_b, k_b, the loop unrolling amounts n_u, m_u, k_u, and the compiler flags. All are integer and are bounded as specified above. In the present case, the declaration file also contains the test data—randomly generated matrices that are preserved throughout the search—and an atomic measure representing elapsed time. The *run file* contains the commands necessary to run the target algorithm with the parameter values suggested by the black-box solver. Those values are exchanged via files for maximum portability. The test matrices are read from file each time the run file is run. After running the run file, the atomic measure is known for the current parameter values. Finally, the *model file* describes the parameter optimization problem and may combine atomic measures into arbitrary composite measures to define the objective function and constraints of (1). This file may also specify options to the black-box solver.

3 Code Generation and Compilation by Templating

Our kernels, main code and cleanup code are generated from *templates*. In our case, templates are strings containing C code and special markup that will allow us to specialize those strings and generate customized code from them. We use the JINJA2 template engine[3] for its ease of use and flexibility. It appears that templating engines were initially created with web development in mind but it turns out that they can be extremely useful in other areas where automatic code generation is crucial. In a template, the double curly brace notation `{{NB}}` is used to mark a place where the value of the parameter `NB` may be substituted. Consider for example the simplified template `t = Template("for (i=0; i<{{NB}}; i++)")` describing the opening statement of a `for` loop in the C language. The command `t.render(NB=16)` produces the expected output; the string `"for (i=0; i<16; i++)"`. JINJA2 also supports constructs for tests and loops in its templates which makes it convenient for automatic code generation. A blocking kernel template illustrating the curly brace notation and conditionals, strongly inspired from [16], is shown in Listing 1. The standard `{% for ... %}` / `{% endfor %}` JINJA2 construct may be used for loop unrolling. Statements involving the loop variable can be included in the loop and sets of variables can be declared inside the loop. A loop unrolling fragment with loop constructs is shown in Listing 2.

As indicated in §2, a kernel using a generic value of β and a kernel using $\beta = 1$ will be needed. Similarly, cleanup code with both β and $\beta = 1$ will be needed. In our numerical experiments, cleanup code is never blocked and its loops are never unrolled. Clearly, this choice may limit the attainable performance. At the very least, cleanup code parameters should be different from kernel parameters.

Once the necessary templates have been *rendered*, they may be concatenated and it is necessary to compile them as a Python extension module. Several possibilities exist and our choice is to use INSTANT[4]—a package for inlining C code within Python code by creating and importing an extension module on the fly—for its small footprint, ease of use and ability to work seamlessly with Numpy[5] arrays—the *de facto* standard data structure for arrays in Python. Rendering templates and compiling them as an extension module that is immediately dynamically imported is illustrated in Listing 3.

In the present example, the templates are relatively short strings. On 64bit platforms, the maximum length of Python strings is essentially limited by the memory available. Note however that template-based code generation only requires those segments of the code that depend on the parameters to be templated. Other components of the code may be precompiled in a library that is linked in by INSTANT. INSTANT accepts user-selected compiler flags via the `cppargs` keyword argument to `inline_module_with_numpy()`.

4 Experimental Framework and Numerical Results

Our tests are run under OSX 10.6.7 on a dual-core 2.66 GHz Intel Core i7-620M *Arrandale* processor with 256 KB of L2 cache per core and 4 MB of L3 cache. Throughout, the compiler used is gcc 4.2.1 (Apple Inc. build 5664). We performed two sets of tests. The first helped us determine appropriate parameters for NOMAD itself by optimizing over a three-dimensional space only, corresponding to the three blocking factors m_b, n_b and k_b. NOMAD is a direct-search method in which the search space is implicitly discretized uniformly using a given mesh size. A typical iteration consists in an optional search step and a mandatory poll step. In the search step, a set of candidates is proposed via a user-supplied procedure. Should one of those candidates improve upon the current iterate, it will be accepted as next iterate and the poll step is bypassed. The role of the poll step is to offer convergence guarantees. The objective function is evaluated at some mesh neighbors of the current iterate along a set of iteration-dependent directions. One of those improving upon the current iterate becomes the next iterate. Should none realize an improvement, the mesh size is decreased. When good improvement is obtained the mesh size may increase. Convergence is guaranteed via a procedure that selects the set of directions defining the neighbors of the current iterate. This procedure should ensure the directions form a positive basis and that in the limit, as the mesh size converges to zero, the union of all

[3] `jinja.pocoo.org`
[4] `launchpad.net/instant`
[5] `numpy.scipy.org`

```
kernel_code = """
        void opal_dgemm_kernel_{{- suffix -}}(
            int nrowA, int ncolA, const double *A,
            int nrowB, int ncolB, const double *B,
            int nrowC, int ncolC, double *C, double beta) {

        int i, j, k;
        register double cij;

        for (i = 0; i < {{ NB }}; i++) {
            for (j = 0; j < {{ MB }}; j++) {

                {% if beta0 %}
                cij = 0.0;
                {% elif beta1 %}
                cij = C[i*ncolC + j];         /* = C[i,j] */
                {% else %}
                cij = C[i*ncolC + j] * beta;  /* = C[i,j] * beta */
                {% endif %}

                for (k = 0; k < {{ KB }}; k++)
                    cij += A[i*ncolA + k] * B[k*ncolB + j];

                C[i*ncolC + j] = cij;
    }}}"""
```

Listing 1: Blocking kernel template with conditionals.

```
{% for ku in range(KU) -%}
  {% for ju in range(JU) -%}
    {% for iu in range(IU) -%}
      cij_{{- iu -}}_{{- ju }} += A[(i + {{iu}})*ncolA + k + {{ku}}] *
                                  B[(k + {{ku}})*ncolB + j + {{ju}}];
    {% endfor -%}
  {% endfor %}
{% endfor %}
```

Listing 2: Loop unrolling fragment with loop constructs.

```
from instant import inline_module_with_numpy
from dgemm_template import *    # Templates are defined here.
from numpy.random import random
kernel1 = kernel_template.render(beta1=True, suffix='1', NB=8, MB=16, KB=32)
kernelb = kernel_template.render(suffix='b', NB=8, MB=16, KB=32)
dgemm = dgemm_template.render(NB=8, MB=16, KB=32)
cleanup1 = cleanup_template.render(beta1=True, suffix='1')
cleanupb = cleanup_template.render(suffix='b')
code = kernel1 + kernelb + cleanup1 + cleanupb + dgemm
dgemm_module = inline_module_with_numpy(code.encode('ascii'),
                                        arrays=[['nrowA','ncolA','A'],
                                                ['nrowB','ncolB','B'],
                                                ['nrowC','ncolC','C']],
                                        modulename='dgemm')
dgemm_module.opal_dgemm(random((50,50)),random((50,50)),random((50,50)),2.0)
```

Listing 3: Rendering the kernel, main and cleanup templates of DGEMM, compiling as an extension module and running on 50×50 random matrices.

normalized directions forms a dense set in the unit sphere. For details, we refer the reader to [4]. Our tests use square matrices of size 1024 and each elapsed time reported is the median of five runs. The first variant of NOMAD uses a set of $n+1$ directions, where n is the dimension of the search space. The second variant uses $2n$ directions. The direction-setting procedure used by the two variants is the one referred to as the *Lower Triangular* procedure in [4]. A third variant uses $2n$ orthogonal directions and is detailed in [17]. They are denoted Lt-$n+1$, Lt-$2n$ and Ortho-$2n$ in what follows. The advantage of the third variant over the first two is that it generates reproducible results. For each variant, two additional options were considered. The first is the *opportunistic* strategy in which the first neighbor yielding an improvement in the poll step is accepted as next iterate. This is in contrast with the non-opportunistic strategy in which the neighbor yielding the best improvement is selected. The second option concerns the use of a quadratic model of the objective in the search step to propose a new incumbent. Should the search step fail when using the opportunistic strategy, the quadratic model is evaluated at all neighbors which are ordered by increasing model value before the poll step.

Our results are summarized in Fig. 1, which plots the objective function value against time. The best found objective value is indicated on the vertical axis of each plot. It is not surprising to note that using quadratic models alone does not yield the best performance as the true objective function is likely highly non-quadratic and even non-smooth. It seems however that the opportunistic strategy pays off, even in conjunction with the quadratic models. Note in all cases the typical L-shaped descent curves exhibiting a sharp decrease in the early iterations and tailing off for most of the search process. Within 1000 seconds or less, most of the descent is achieved for all variants which indicates that this is a reasonable time budget in the absence of a problem-specific search step procedure. In all cases, the final solutions differ sufficiently to indicate that many parameter sets produce similar timings. This was already noted in [15]. Note that all total run times reported include the time required to read the test matrices from file at each black-box evaluation and the time required to compile and load the generated code as a Python extention module. All elapsed times reported only represent the time required to perform the matrix update.

The second set of tests comprises more parameters and is run using the dominant variants of NOMAD. We first selected the Lt-$n+1$ procedure with the opportunistic strategy alone, and the Ortho-$2n$ procedure with the opportunistic strategy and quadratic models. The search space comprises this time the three blocking factors, three unrolling amounts and two compiler flags. The first compiler flag is the optimization level and can take the values `-O0`, `-O1`, `-O2` or `-O3`. The second flag, `-march=core2`, can be either turned on or off and determines whether the compiler should produce generic or machine-specific code. When turned on, this flag activates MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support. In all cases, all initial parameters are set to 1, optimization is set to `-O2` and the `-march` flag is deactivated. The initial elapsed time is about 22s. The final parameter values identified by the Lt-$n+1$ variant are $n_b = 33$, $m_b = 12$, $k_b = 25$, $n_u = 3$, $m_u = 3$, $k_u = 1$ with compiler flags `-O2` and `-march=core2` for a final elapsed time of 0.95s, a total run time of 4799s and 635 evaluations. The Ortho-$2n$ variant stopped with $n_b = 33$, $m_b = 40$, $k_b = 20$, $n_u = 3$, $m_u = 14$, $k_u = 1$ and compiler flags `-O1` and `-march=core2` for a final elapsed time of 0.91s, a total run time of 7428s and 1249 evaluations. As in the first set of tests, the Lt-$n+1$ variant appears to yield good results faster than other variants and appears desirable despite it not being entirely reproducible. The value $k_u = 1$ is quite surprising as this parameter is known to be influential but in other runs, larger values of k_u were obtained with a similar final elapsed time. Numerous improvements on these preliminary tests are possible, including re-starting from the final values with a large mesh size to escape from a local solution, and using problem-specific search step procedures. We do not explore such possibilities here in order to keep the focus on the templating framework.

5 Discussion

Only the parts of a code that require optimization need be templated. The rest may be precompiled and linked in at compile time. This makes templating a viable option for general-purpose optimization of scientific computing codes.

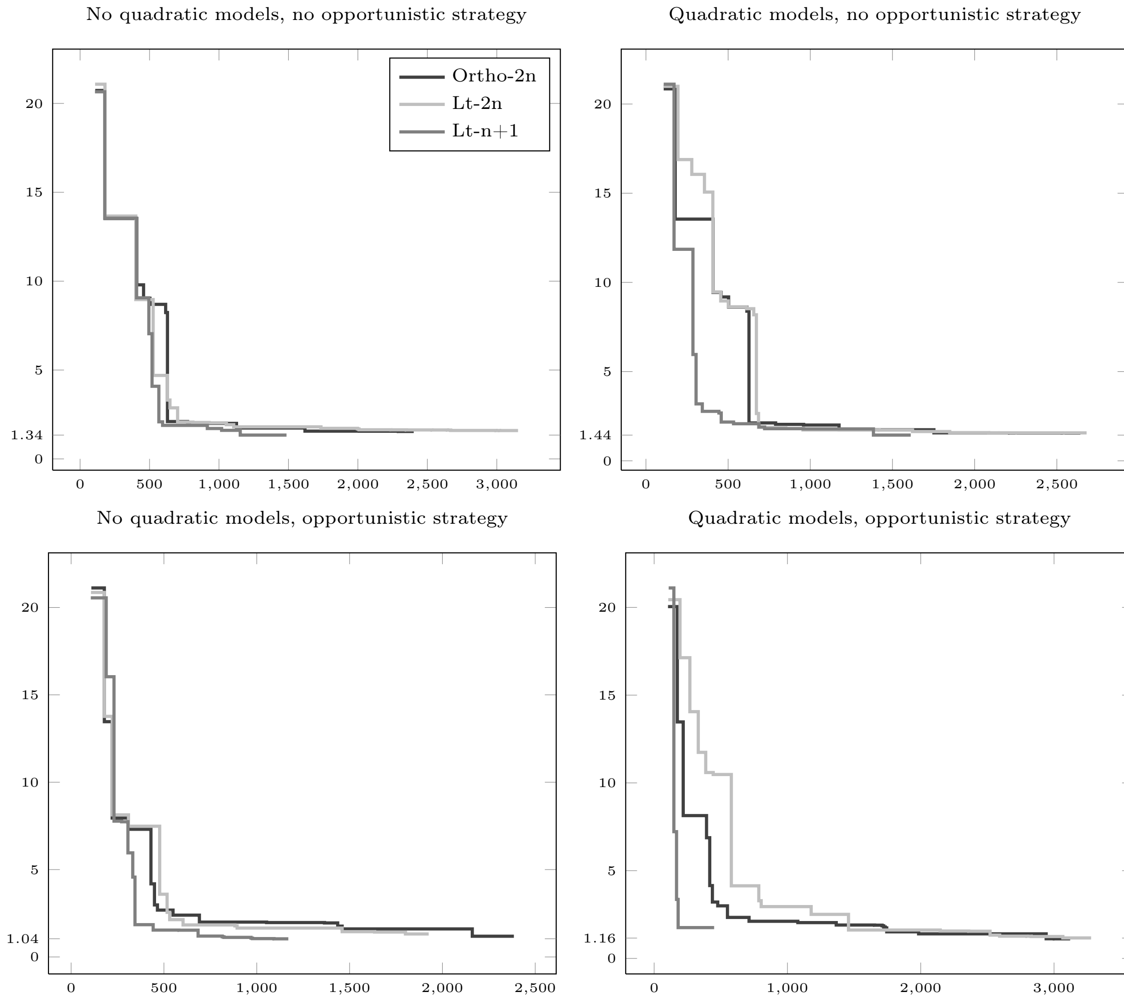

Figure 1: Evolution of the objective (s) versus time (s) for various NOMAD settings.

Using a direct-search method with convergence guarantees has several advantages over heuristic searches such as those used by [15]. Firstly, some variants are deterministic and ensure reproducibility of the results when run in similar conditions, although measures such as elapsed time may be relatively noisy. The stopping conditions offer a certain optimality certificate—we refer the interested reader to [4] for full details. At each iteration, the search step allows for a surrogate model to be used to steer the iterates towards promising regions. Using as a surrogate the search procedure currently used in, say, ATLAS, guarantees that the direct-search approach will identify parameters that are at least as good as those suggested by ATLAS. Any other search procedure, whether heuristic or not, could be used as a surrogate model. In our opinion, this is the feature that differentiates the OPAL approach the most from previous proposals.

An advantage of the framework proposed in this paper is its modularity. Other black-box optimization solvers can be interfaced with OPAL and used in a code templating application. There exist numerous templating engines for Python—see `wiki.python.org/moin/Templating`—each with different feature sets and those could be interchanged to better suit the needs of the current application. Finally, several Python modules for automatic compilation of C code are available and INSTANT is just one of them, which stands out for its simplicity, its ability to handle Numpy arrays and its small footprint. Alternatives include WEAVE, which is part of the SciPy project[6], F2Py, which is part of NumPy, and Cython, which will become a superset of the Python language providing static typing and tight integration with NumPy arrays. Another

[6] `www.scipy.org`

advantage is that all the tools required are already part of the Python tool chain. As a consequence, though our approach is currently not able to annotate code automatically as in, e.g., the Rose LoopProcessor [10], annotation via templating is easy and uses classical programmatic constructs. For a simple matrix multiply, rather than a matrix update, other parameters could have been considered. Loop order is the most natural and could be implemented in Opal as a periodic integer variable, i.e., each loop order has two "neighbors". In particular, the neighbors of the first in the list are the second and the last. In future versions of Opal, they will be modeled as categorical parameters, i.e., discrete parameters on which no specific order is prescribed. Modeling categorical parameters requires the user to describe the set of neighbors of each parameter value. One possibility would be for each loop order to have every other possible loop order as neighbors.

Templating in other languages than C presents no additional difficulty save for the fact that there may not exist a way to easily compile and import the rendered code as a Python extension module. However Opal does not impose that the resulting code be run as an extension module—it could simply be run by issuing system commands, which can be done from Python as well.

We intend to generalize the framework proposed in the present paper to a more complete template-based set of linear algebra kernels. We expect that the modularity and flexibility of our approach will ease improvement of the search engine, the templating engine, or any other part of the optimization process.

References

[1] Audet, C., Orban, D.: Finding optimal algorithmic parameters using derivative-free optimization. SIAM Journal on Optimization **17**(3) (2006) 642–664

[2] Audet, C., Dang, C.K., Orban, D.: Algorithmic parameter optimization of the DFO method with the OPAL framework. In K. Naono, K. Teranishi, J.C., Suda, R., eds.: Software Automatic Tuning: From Concepts to State-of-the-Art Results. Springer, New-York, NY (2010) 255–274

[3] Le Digabel, S.: Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. ACM Transactions on Mathematical Software **37**(4) (2011) 44:1–44:15

[4] Audet, C., Dennis, Jr., J.E.: Mesh adaptive direct search algorithms for constrained optimization. SIAM Journal on Optimization **17**(1) (2006) 188–217

[5] Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1–2) (2001) 3–35

[6] Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of basic linear algebra subprograms (BLAS). ACM Transactions on Mathematical Software **28** (2002) 135–151

[7] Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: POET: Parameterized optimizations for empirical tuning. International Parallel and Distributed Processing Symposium (2007) 447

[8] Luszczek, P., Dongarra, J.: High performance development for high end computing with python language wrapper (PLW). International Journal of High Performance Computing Applications **21**(3) (2007) 360–369

[9] Donadio, S., Brodman, J., Roeder, T., Yotov, K., Barthou, D., Cohen, A., Garzarán, M., Padua, D., Pingali, K.: A language for the compact representation of multiple program versions. In Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P., eds.: Languages and Compilers for Parallel Computing. Volume 4339 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 136–151

[10] Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In Böszörményi, L., Schojer, P., eds.: Modular Programming Languages. Volume 2789 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2003) 214–223

[11] Miller, W.: Software for roundoff analysis. ACM Transactions on Mathematical Software **1** (1975) 108–128

[12] Miller, W., Spooner, D.: Sofware for roundof analysis II. ACM Transactions on Mathematical Software **4** (1978) 369–387

[13] Larson, J.L., Sameh, A.H.: Algorithms for roundoff error analysis—a relative error approach. Computing **24** (1980) 275–297

[14] Larson, J.L., Pasternak, M.E., Wisniewski, J.A.: Algorithm 594: Software for relative error analysis. ACM Transactions on Mathematical Software **9** (1983) 125–130

[15] Seymour, K., You, H., Dongarra, J.J.: A comparison of search heuristics for empirical code optimization. In: Proceedings of the 2008 IEEE International Conference on Cluster Computing. Third international Workshop on Automatic Performance Tuning (iWAPT 2008), Tsukuba International Congress Center, EPOCHAL TSUKUBA, Japan (2008) 421–429

[16] Whaley, R.C.: A guide to user contribution to ATLAS (April 2011)

[17] Abramson, M.A., Audet, C., Dennis, Jr., J.E., Le Digabel, S.: OrthoMADS: A deterministic MADS instance with orthogonal directions. SIAM Journal on Optimization **20**(2) (2009) 948–966