Meta-Tuning in MTL4

Peter Gottschling[*,†] and Cornelius Steinhardt[*,†]

[*]*SimuNova UG, Helmholtzstr. 10, 01069 Dresden, Germany*
[†]*Technische Universität Dresden, 01062 Dresden, Germany*

Abstract. The paper introduces a new paradigm—called meta-tuning—for optimizing high-performance software based on meta-programming. The technique explores the Turing completeness of C++ templates. Meta-tuning allows for generating performance-optimal variations of high-performance implementation within C++ that otherwise would require rewriting or code generation with external tools. Numerical results are presented that show accelerations up to a factor 6.6.

Keywords: Generic programming, performance tuning, meta-programming
PACS: 02.70.-c, 02.70.Wz, 07.05.Bx, 07.05.Tp

INTRODUCTION

High-quality scientific software that is used to realistically simulate real-world problems like the chemical behavior of nano-structures requires enormous computational power. However, this power is not available per se but demands meticulously tuned software implementations. Fundamental packages like BLAS (Basic Linear Algebra Subprograms) are often realized in assembler for every hardware platform. Such implementations are performance-wise almost perfect. However, this contradicts all principles of software engineering regarding: development costs; reusability; readibility; maintainability; adaptibility; extensibility and portability.

In order to overcome the gigantic development efforts of hardware-specific assembler programming, the ATLAS project [WD98, WPD01] (Automatically Tuned Linear Algebra Software) aims for automatizing the generation of high-performance BLAS code. Its developers created a description language and tools for generating C code possibly containing assembler code. This C code is benchmarked with different parametrizations on the considered architecture and the fastest parameter set is selected for the platform-specific version of the library.

In this paper, we show how to achieve similar results without a description language and additional tools nor containing assembler kernels in the sources. The foundation of this methodology—called **Meta-Tuning**—is the Turing completeness of C++ templates [Vel03].

ELEMENT-WISE VECTOR OPERATIONS

In this section we consider vector operations where the n^{th} entry of the result depends on the respective n^{th} entries of the arguments, like in

$$u = 3v + w. \tag{1}$$

Such vector operations appear often in scientific software and their performance is therefore very important.

Implementation:
The easiest implementation of such vector operations is a simple loop as in Listing 1. Performance improvements can be achieved through unrolling as in Listing 2 since this reduces loop overhead and more importantly introduces concurrency.

```
for ( uint i = 0; i < s; i++)
    u[i]= 3.0f * v[i] + w[i];
```

Listing 1: Naïve implementation of (1)

```
for ( uint i = 0; i < s; i += 4) {
    u[i]=   3.0f * v[i]   + w[i];
    u[i+1]= 3.0f * v[i+1] + w[i+1];
    u[i+2]= 3.0f * v[i+2] + w[i+2];
    u[i+3]= 3.0f * v[i+3] + w[i+3];
}
```

Listing 2: HPC implementation of (1)

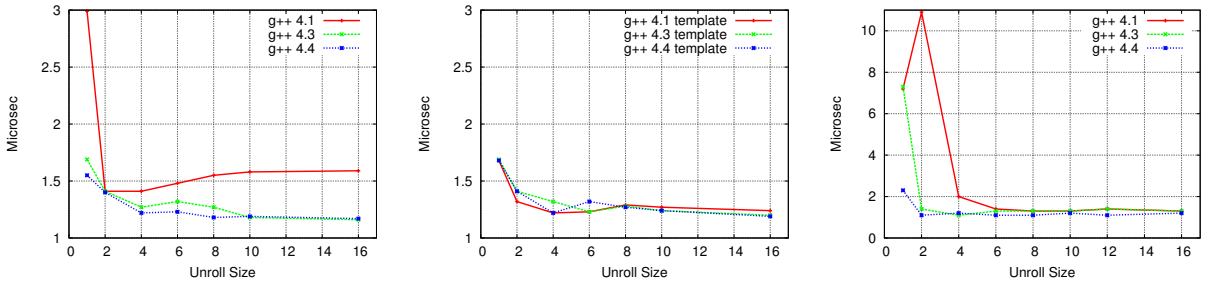

FIGURE 1. Impact of unrolling vector operations: left (1) as function, center (1) as expression template, right L_1 norm.

In Listing 2, we unrolled the loop with block size four. This raises the question whether this is the best choice. Does there exist a single block size that is optimal in all scenarios? Elsewise, does the optimal unroll size depend on the expression? On the types of the arguments? On the computer architecture? The answer is yes. All of them. As a consequence, we must evaluate vector expressions as the one above with context-dependent unrolling. Emulating such unrolling with nested loops and run-time arguments would cause even more loop overhead as the naïve implementation in Listing 1 without enabling concurrency.

What we need is passing the block size as **compile-time argument**. We must generate implementations equivalent to Listing 2 at compile time with meta-programming [AG04]. The Turing completeness of templates enables this. Listing 3 implements the considered operation with customizable unroll size. The latter is passed at compile time as template argument. The inner block of the first loop is realized by Listing 4. The evaluation of the recursive function calls happens at compile time whereas by C++ convention member functions defined inside a class are inlined. The increment Offset is a template argument and can be handled as constant during run time. Thus, the call my_axpy<4>(u, v, w) performs exactly the same operations as in Listing 2.

```
template <uint BSize, typename U,
          typename V, typename W>
void my_axpy(U& u, const V& v, const W& w)
{
    uint  s= u.size(),   sb= s /  BSize * BSize;

    for ( uint  i = 0;  i  <  sb;  i += BSize)
        my_axpy_ftor<0, BSize>()(u, v, w, i );
    for ( uint  i = sb;  i  <  s;  i ++)
        u[i ]= 3 *  v[i ] +  w[i];
}
```

Listing 3: Customizable unrolling for (1)

```
template <unsigned Offset, unsigned Max>
struct my_axpy_ftor {
    template <typename U, typename V, typename W>
    void operator()(U& u, const V& v, const W& w, unsigned i) {
        u[i +Offset]= 3 *  v[i +Offset] +  w[i +Offset];
        my_axpy_ftor<Offset+1, Max>()(u, v, w, i);      }
};
template <unsigned Max>
struct my_axpy_ftor<Max, Max> {
    template <typename U, typename V, typename W>
    void operator()(U& u, const V& v, const W& w, unsigned i) {}
};
```

Listing 4: Inner block of Listing 3

In libraries based on expression templates [Vel95] like MTL4 [GWA07], it is sufficient to modify one single function—e.g., the vector assignment—to provide customizable evaluation of **all vector expressions**. To customize the unrolling of computing eq. (1) as expression, the user writes conveniently 'unroll<4>(u)= 3 * v + w;'.

Benchmarks:
Figure 1 shows the influence of unrolling size on computing Equation (1) for different compiler versions. On the left side, the implementation from Listing 3 is used and in the center, the operation is performed with an (unrolled) expression template. Our test system is an AMD Phenom™ 9150e Quad-Core Prozessor with 4×1.8 GHz and the vectors have 1000 entries.

ACCELERATED REDUCTIONS

Reduction operations on vectors like the L_1 norm can be accelerated similarly. The significant difference is that the temporary variable where the partial results are accumulated makes every operation dependent on the previous one

and impedes concurrency. The challenge in reduction operations is therefore to introduce a customizable number of temporary variables.

The introduction of additional temporaries can be realized differently: by nested classes, by nested functors or by arrays. For the sake of simplicity, we show here only the latter Listing 5. In Listing 6, we like to draw the reader's attention to the fact that in each operation of the block a different entry of sum is incremented.

```
template <uint BSize, typename Vector>
typename Vector::value_type
inline  one_norm(const Vector& v) {
    typename Vector::value_type sum[BSize]= {0};

    uint  s= size(v),  sb= s /  BSize * BSize;
    for ( uint  i = 0;  i  <  sb; i += BSize)
        one_norm_ftor<0, BSize>()(sum, v, i);

    for ( uint  i = 1;  i  <  BSize; i ++) sum[0]+= sum[i];
    for ( uint  i = sb;  i  <  s;  i ++) sum[0]+= abs(v[i]);
    return sum[0];
}
```

Listing 5: Block-wise L_1 function

```
template <uint Offset,  uint  Max>
struct one_norm_ftor {
    template <typename S, typename V>
    void operator()(S* sum, const V& v, uint i) {
        sum[Offset]+= abs(v[i+Offset]);
        one_norm_ftor<Offset+1, Max>()(sum, v, i); }
};
template <uint Max>
struct one_norm_ftor<Max, Max> {
    template <typename S, typename V>
    void operator()(S* sum, const V& v, uint i) {}
};
```

Listing 6: Inner block L_1 functor

The impact of introducing concurrency in reduction operations is shown in the right plot of Figure 1 for different compilers. One can see that light unrolling (block size 2–4) already yields a significant performance boost (except for g++ 4.1) while further unrolling might result in additional acceleration or in slight slow-down.

MATRIX OPERATIONS

Dense matrix multiplication is without any doubt the single-most often used benchmarked operation in high-performance computing. We have shown in earlier publications [GWA07] that the combination of meta-tuning and recursion allows for $\approx$70 % peak performance that scales with matrix size.

The matrix product is realized with the same principles as the vector operations. However, this operation enables a two-dimensional tuning as shown by two parameters in function mult of Listing 7 (abbreviated, full version in [GL]).

```
template <uint Size0, uint Size1, typename Matrix>
void inline mult(const Matrix& A, const Matrix& B, Matrix& C) { // ...
    mult_block<0, Size0−1, 0, Size1−1> block;
    for (uint i= 0; i < num_rows(A); i+= Size0)
        for (uint k= 0; k < num_cols(B); k+= Size1) {
            multi_tmp<Size0 * Size1, typename Matrix::value_type> tmp(value_type(0));
            for (uint j= 0; j < num_cols(A); j++)
                block(tmp, A, B, i, j, k);
            block.update(tmp, C, i, k); }
}
template <uint Index0, uint Max0, uint Index1, uint Max1>
struct mult_block { // ...
    template <typename Tmp, typename Matrix>
    void operator()(Tmp& tmp, const Matrix& A, const Matrix& B, uint i, uint j, uint k) {
        tmp.value+= A(i + Index0, j) * B(j, k + Index1); next()(tmp.sub, A, B, i, j, k);
    }
};
```

Listing 7: Block-wise matrix multiplication

In Figure 2, we compared 32 parameter sets for multiplying 128×128 **double** matrices on three compilers. The plots show that there is no obvious scheme for how the performance depends on the parameters. This underlines the need for experimentation to determine a platform's optimal configuration. The run-time behavior of executables generated by g++ 4.4 and 4.3 was very similar and superior to those of 4.1. The compile time of g++ 4.1 was enormous and we omitted more expensive parameter sets.

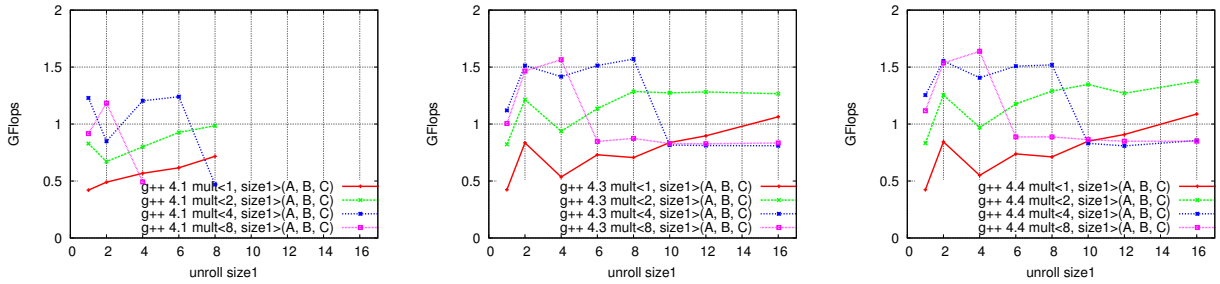

FIGURE 2. Impact of nested unrolling in matrix multiplication.

CONCLUSIONS AND OUTLOOK

The numerical results have shown that specialized implementations can achieve significant accelerations even in the presence of advanced compiler technology. The element-wise vector operations could be doubled in speed while reduction operations enabled speed-ups up to a factor 6.6 and dense matrix multiplication up to 3.9.

As mentioned before, the parameter tuning of HPC implementations can be achieved without external tools by just exploring the Turing completeness of standard-compliant C++ compilers with advanced meta-programming techniques. Unless the user desires to influence parameters explicitly, the meta-tuning is usually applied user-transparently in the internals of library implementations like in MTL4 [GL]. As soon as enough empirical knowledge about optimal parametrization in different scenarios is known, the parameters can be computed with meta-programming depending on platform, types and computed expressions (all these information are known at compile time). The paper describes how the performance-relevant parameters are explored in code generation. This raises the question where the parameters come from. We see currently three options: (i) the user determines it explicily in the function call, (ii) the parameters are determined for specified expressions and types by profiling and written into header files (we develop portable python scripts for this task) and (iii) the parameters will be computed by meta-programming.

C++ programs will still not be exactly as fast as leading assembler implementations of dense linear algebra but this is not necessary since BLAS and LAPACK routines can be used in C++ applications user-transparently (when a corresponding routine exist for the considered expression and argument types).

The meta-tuning methodology is not limited to loop unrolling but can be accordingly used in other programming models, like GPU programming. The presented optimizations are readily applicable to all intrinsic floating-point types. The application to user-defined types can yield incorrect results when semantic requirements are not fulfilled, i.e. the modified evaluation order of operations and the usage of extra temporaries requires that the vector and matrix elements are commutative monoids. If this is not the case for a given user type, the optimized calculation will be wrong. To guarantee properties of non-intrinsic types, meta-tuning must be combined with semantic concepts [GL08] which are expected to become part of future C++ standards.

REFERENCES

AG04. D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2004.

GL. P. Gottschling and A. Lumsdaine. The Matrix Template Library 4. `www.osl.iu.edu/research/mtl/mtl4/`.

GL08. P. Gottschling and A. Lumsdaine. Integrating semantics and compilation. In *Proc. 7th GPCE*, pages 67–76. ACM Press, New York, NY, USA, 2008.

GWA07. P. Gottschling, D.S. Wise, and M.D. Adams. Representation-transparent matrix algorithms with scalable performance. In *Proc. 21st ICS*, pages 116–125. ACM Press, New York, NY, USA, 2007.

Vel95. T.L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

Vel03. T.L. Veldhuizen. C++ templates are Turing complete. `citeseer.ist.psu.edu/581150.html`, 2003.

WD98. R.C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SC'98: Proc. 1998 IEEE/ACM Conf. Supercomputing*, pages 1–27. IEEE Computer Society Press, Los Alamitos, CA, USA, November 1998.

WPD01. R.C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.