A C++ FRAMEWORK FOR CONDUCTING HIGH-SPEED, LONG-TERM PARTICLE TRACKING SIMULATIONS

A. Kabel*, Stanford Linear Accelerator Center, Stanford, CA 94025

Abstract

For the purpose of conducting parallel, long-term tracking studies of storage rings such as the ones descriped in [3], [4], maximum execution speed is essential. We describe an approach involving metaprogramming techniques in C++ which results in execution speeds rivaling hand-optimized assembler code for a particular tracking lattice while retaining the generality and flexibility of an all-purpose tracking code.

INTRODUCTION

Shifts in computer hardware technology over the last two decades have had a profound impact on the speed optimization of numeric algorithms. The advent of fast data ans instruction cache and the advancement in floating-point hardware have dramatically decreased the impact of floating-point operations on total program performance and increased the impact of cache locality and vectorizability of the code. Improvements in compiler optimization technology have rendered unnecessary common source code optimizations, such as constant folding and common-subexpression elimination, which are now best left to the compiler.

In an ideal world, a high-performance particle tracking code would consist of a source code for a tracking routine specific for a particular magnetic lattice. The compiler could then create an optimized machine code for that lattice, utilizing its knowledge about floating-point hardware, caching, and data dependency. While this is certainly not practical, the C++ language's templating mechanism, which in practice is a code-rewriting and transformation tool, allows one to get very close to this goal. In the sequel, we describe how such metaprogramming techniques allow one to obtain very high processing speeds from a general-purpose tracking code; we also describe some of the properties of the code `dumbbb`.

THE TRACKING CODE `dumbbb`

The test field of these techniques is the tracking code `dumbbb`, which grew from a simple kernel-driven tracking code for the study of limetime behavior of the Tevatron [3] to a general-purpose tracking code, roughly reflecting MAD 8.x's tracking capabilities. In the following, we describe some building blocks of the code.

* akabel@stanford.edu

LIGHTWEIGHT CLASSES

Vectors and Matrixes

The most basic object in our hierarchy is a (phasespace) vector. It is a templatized class, unsing both base type and dimension as a parameter. As the vector size is known at runtime, loop optimizations by the compiler are facilitated. The interface is modeled after the container classes of the ANSI C++ library (formerly known as STL).

Matrices are similarly implemented, using data type, row, and column number as templaet parameters. Flat access to the matrix elements is possible in STL style, allowing for generic copying/clearing/assigning. All usual matrix operations are declared, as is matrix-vector multiplication.

Differential Algebraic Objects

Differential algebraic objects come in two flavors: as generalized symmetric tensor objects, representing homogeneous or inhomeogeneous multivariate polynomials of arbitrary dimension and degree and used for the evaluation of power series provided externally (see below) , and as small order polynomials (up to 3) used to determine Taylor series from generic function definitions at compile time.

Fast Evaluation of Multivariate Polynomials

The evaluation of truncated power series on phase space vectors can be used as an alternative to element-by-element tracking, provided the order is high enough to mask the intrinsic non-symplecticity of the procedure. Metaprogramming allows for an extremely efficient implementation of this method.

Consider the set H_v^n of homogeneous polynomials of degree n over a field $\mathbb{F}$ in v variables $x_1, \ldots, x_v$. Each polynomial $h_v^n \in H_v^n$ can be represented as an ordered pair of polynomials: $h_v^n = (m, f), m \in H_v^{n-1}, f \in H_{v-1}^n$, where f is a polynomial comprising monomials in h_v^n containing x_1, divided by x_1, and where m comprises all other monomials, and $H_1^n = \mathbb{F}, H_v^0 = \mathbb{F}$, i. e., the coefficients live on the leaves of the tree defined by the recursion relation (). The evaluation at a point $(x_1, \ldots, x_v)$ reads

$$\begin{aligned} h_v^n(x_1 \ldots, x_v) &= x_1 m(x_1, \ldots x_v) + f(x_2, \ldots, x_v) \\ h_1^n(x_1) &= x_1^n h_1^n \\ h_v^0(x_1, \ldots, x_v) &= h_v^0 \end{aligned} \tag{1}$$

In the case of inhomogeneous polynomials, as we want to evaluate truncated power series, we just evaluate an element of H_{v+1}^n at $(x_1, \ldots, x_v, 1)$. In this case, in the recursion relation (1), h_1^n will always have an argument of 1 at

Contributed to Particle Accelerator Conference (PAC 05), 16-20 May 2005, Knoxville, TN

Work supported by Department of Energy contract DE-AC02-76SF00515

evaluation, so the exponentation operation in the prescription for $h_1^n(x)$ can be dropped. Similarly, one can also consider a cut-off evaluation, which suppresses terms of order higher than k in $x_1, \ldots, x_\nu$.

The recursion relations for the coefficient tree, its evaluation at a point, and its cut-off evaluation can be implemented directly in terms of templated classes in C++, using F, n, v as parameters. The evaluation function is implemented as an 'inlined' function, i. e. the compiler is requested to substitute an in-place compilation of the called function at the place of a function call. Consequently, a function call to the evaluation function, which is recursively defined in terms of calls to other evaluation functions, should be completely unwrapped as a single arithmetic expression. Thus, the performance of hand-optimized machine code should be attainable. This behavior, however, is dependent on the inlining strategy and optimization capabilities of the compiler. Inspecting the generated code for a typical compiler (we used GNU g++) shows that indeed the evaluation function can be inlined completely, i. e. no administrative overhead, such as function calls, index calculations, is generated; the generated machine code for the evaluation function is a sequence of load/store, addition, and multiplication operations.

To benchmark performance, we have implemented the evaluation of the polynomials in terms of a table-driven stack machine: it consists of a stack of intermediate results, a table of coefficients c, and a v-tuple of variables x, and a string of operations implementing the evaluation algorithm. (In is interesting to note that the required stack depth is dependent on the order of operations in the recursion relation (1, this fact can also be observed in the compiler-generated code for the templated version.)

Timing a test program which tracks 10^6 test particles with 6 coordinates each through a densely populated Taylor map, we obtained the following values using the GNU C++ compiler, v3.3.2, and the Intel C++ Compiler, v7.1, on an 1.8GHz Xeon processor and maximum optimization settings for both stack machine ("SM") and metaprogramming ("MP") versions; Under the assumption that the theoretically predicted number of additions and multiplications is performed, and that the Xeon processor performs at a peak floating-point-rate equal to its clock frequency, we get the following efficiencies in table 1.

Order	7	8	9
Operations:	1716	3002	5004
gcc, MP	1.0	.96	n/a
icc, MP	.38	.34	.35
gcc, SM	.32	.32	n/a
icc, SM	.30	.31	.30

Table 1: Tracking efficiency of different algorithms and code generated by different compilers

We can observe a speed gain of about 3 (which is of the same order as the gain of 4...8 observed when comparing to to other, traditional codes[1]) for the GNU compiler. This is in agreement with our observation of complete expression unrolling for this compiler. The numbers are normalized to a sustained floating point performance of $P_{max} = 1.08$GFlops observed in the best case.

Determining Multivariate Power Series

Power series can be determined at compile time by augmenting objects of the type $\boldsymbol{x}+\alpha_1 \mathrm{d}\boldsymbol{x}+\alpha_2 \mathrm{d}\boldsymbol{x}\vee\mathrm{d}\boldsymbol{x}+\ldots$ with overloaded operators for the conventional arithmetic operators, providing truncated power series arithmetic. Again, these class (called `Interval` because it can be used to establish numeric bounds on calculated quantities) is templated by base type, order, and dimension. By providing a set of traits classes templated by the transcendental functions from `<cmath>`, the expressions for n-th order derivatives of arbitrary functions can be determined at compile time, allowing the compiler to optimize out common subexpressions and trivial constant values. This allows us fast tracking of particles with attached infinitesimal neighborhoods, which can be used for the determination of Lyapunov exponents.

Parsing and Lattice Manipulation

`dumbbb` will read a MAD 8.x conformant input file. As the MAD syntax seems not to be formally defined, the parser is restricted to the well-defined subset of files generated by MAD's `SAVE` command. The parser is realized in terms of a `bison` grammar and a `flex` DFA scanner. Global and local symbol lookup is realized in terms of a stack of hashtables. All MAD definitions are stored as complete parsetrees, expressed in terms of reference-counted pointers to an abstract `Evaluable` class. Specialization of these classes are MAD elements with `Evaluable` member variables representing MAD element attributes. Attributes can be altered or evaluated; elements can be copied with a virtual `Clone` member function.

Beamline definitions can be expanded into a flat list of MAD elements. Flat lists, in turn, can be subject to insertions at arbitrary longitudinal positions, including within elements, which are split in two sections. This is used in `dumbbb`for inserting beam-beam elements at positions according to the currently examined bunch and cogging stage.

Elements

The flat list of elements acts as a 'class factory' for tracking elements proper. Each MAD element will generate a finite number of tracking elements, acting on a variety of types (see below). The currently implemented elements and their implementation methods are:

Drift Spaces. Either linear or first-order chromatic.

Quadrupoles. Either linear or first-order chromatic. First-order chromaticity is implemented either by the exact solution of the 3rd-order Hamiltonian or by a sequence of thick-lens matrices and chromatic thin-lens kicks.

Sector Bends. Either linear or first-order chromatic. Chromaticity is implemented as a sequence of thick-lens matrices and kicks according to the 3rd-order Hamiltonian.

Multipoles, RF Cavities, Electrostatic Separators Implemented by a sequence of, possibly chromatic, drift spaces and thin-lens kicks.

Beam-Beam Elements, Tune Prints, Apertures, Counters. See below.

Kernels

In implementing the actual tracking elements, i. e. the classes acting on particles during tracking, one is faced with a dilemma: On the one hand, the action of a tracking element is generic and can, in most cases, be written down as a simple formula which reads the same for `float`,`double`,... or differential-algebraic objects; thus, a templated transformation member function is called for. On the other hand, the action has to be virtualized, as there are different kinds of transformation, depending on the element type. Both requirements are obviously incompatible, as there are no virtual templated member functions in C++.

The solution lies in the creation of a 'fat interface' class:

```
struct AbstractElement {
  virtual void Xform(
    ParticleBunch<double> &) const=0;
  virtual void Xform(
    ParticleBunch<long double> &) const=0;
  ...
  virtual void Xform(
    ParticleBunch<DiffAlg<1> >&) const=0;
  ...
  virtual void Xform(
    ParticleBunch<DiffAlg<3> >&) const=0;

};
```

This abstract base class can be created from an Andriescu-style typelist and an automatic recursive instantiator.

Loopers

Having created the abstract interface, one now needs to create concrete classes. To do this, one creates a second recursive instantiator, which, instead of creating the interface, inherits from `AbtractElement` and, in each recursion step, implements each `Xform(ParticleBunch<T> &)` in terms of a generic-type `Kernel<T>`, which is injected as a template parameter.

Usually, the action of a transformation element can be serialized in terms of individual particles. Thus, a kernel acting on a `ParticleBunch` can be written as a loop, acting with a `kernel<T>` on phasespace vectors.

At the application level, all one needs to implement a new element type is a class definition with a templated member function `xform(Phasespace<T> &)` , e.g.

```
class QuadKernel {
public:
 template<typename T>
 xform(PhaseSpace<T> &v) { v = M * v; }
private:
 const Matrix M;
 ....
};
```

A tracking element, with compiler-optimized member functions looping over particles in a bunch and separately rewritten for each data type known (by typelists) to the framework will then be generated by simply writing

```
class Quad:
public TransformElem<Looper<QuadKernel> > {};
```

The advanteage of injecting `loopers` at instantiation time lies in the fact that the compiler can optimize out common subexpressions and loop invariants; this would not be possible by virtualizing on the particle level.

FlipLoopers and Blocking

Variations of this method are possible for different circumstances: in some cases, it is advantageous to write the single-particle kernel in terms of in-out variables: `xform(const Phasespace<T> &in, Phasespace<T> &out)`, e.g., for matrix multiplications, thus avoiding load-restore operations, which tend to be expensive on some platforms.

Results

The resulting code has been benchmarked and validated against MAD 8, we observe a speed gain of up to an order of magitude for element-by-element tracking. A speed measurement for the Tevatron case is given in [4]. I is worth stressing that direct lifetime calculation for the Tevatron, using element-by-elemnt tracking, now seem feasible.

ACKNOWLEDGMENTS

Work supported in part by the U. S. Department of Energy under contract number DE-AC02-76SF00515.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U. S. Department of Energy under Contract No. DE-AC03-76SF00098.

I wish to thank Y. Cai and T. Sen for helpful discussions.

REFERENCES

[1] Y. Cai (SLAC), private communication.

[2] American National Standard Institute, International Standard ISO/IEC 14882:1998(E) (1998)

[3] A. Kabel, Y. Cai, B. Erdelyi, T. Sen, M. Xiao, Proceedings of the 2003 IEEE Particle Accelerator Conference (2003)

[4] A. Kabel et al., this conference

[5] K. Hirata, H, Moshammer, F Ruggiero, Part. Acc. **40**,205 (1993)

[6] M. Bassetti, G Erskine, CERN ISR TH80-06 (1980)

[7] F. Matta and A. Reichel, Math. Comp. **25**, 339 (1971)