Processing Sparse Vectors During Compile Time in C++

Jaakko Järvi

Turku Centre for Computer Science, Finland

Abstract. A C++ template library for a special class of sparse vectors is outlined. The sparseness structure of these vectors can be arbitrary but must be known at compile time. In this case it suffices to store only the nonzero elements of the vectors, and no indexing information about the sparseness pattern is required. This information is contained in the type of the vector as a non-type template parameter. It is shown how common vector operators can be overloaded for these vectors. When compiled the operators yield code which performs only the necessary elementary operations between the nonzero elements with no run-time penalty for indexing. All indexing is performed at compile time, resulting in very fast execution speed. The vector classes are best suited for short vectors up to few dozens of elements.
Automatic differentiation of expressions is given as an example application. It is shown how classes for automatically differentiable numbers can be defined with the library. A comparison against other vector representations gave superior results in execution speed of differentiating a few common expressions, and came very close to the calculation speed of symbolically differentiated expressions.

1 Introduction

Templates are a powerful feature of C++. They are usually used to write generic classes and functions, but we can go beyond that. It is possible to make the compiler perform as an interpreter at compile time. For instance, compile-time bounded loops and branching statements can be written using recursive template definitions and template specialisation. These templates are called *template metaprograms* [1]. The use of non-type template parameters is a key factor behind template metaprograms.

In this article template metaprograms are used in the definition of *compile-time sparse vector* (CTSV) classes. The term *compile-time* here means that the sparseness pattern of the vectors (the positions of zero and non-zero elements) is known at compile time. Due to this restriction CTSVs do not serve as general purpose sparse vectors, but they are very efficient in special applications where the above requirement can be satisfied.

The definitions for CTSV template classes are given. It is shown how common operations can be overloaded for CTSV types. Automatic differentiation [2] is presented as an application of CTSVs. The techniques presented take advantage

Copyright 1998 Springer-Verlag (Lecture Notes in Computer Science, vol. 1343)

of the new template features present in the current draft standard of C++ [3], but are not yet implemented in all compilers. A more detailed discussion can be found in [4].

2 Class Templates for Compile-Time Sparse Vectors

A vector is called sparse if only a few of its elements are nonzero. In sparse storage schemes only the nonzero elements are stored, along with some auxiliary information to determine the logical positions of the elements in the vector. Hence space is saved and computational speed gained since some elementary operations are performed only on nonzero elements. An arbitrary sparse vector can be written as a set of value-position pairs

$$\boldsymbol{x} = (< x_{i_1}, i_1 >, < x_{i_2}, i_2 >, \ldots, < x_{i_n}, i_n >).$$

With this notation we can write the sum of two sparse vectors $(1, 0, 1, 0, 0)$ and $(0, 0, 2, 0, 2)$ as

$$(< 1, 1 >, < 1, 3 >) + (< 2, 3 >, < 2, 5 >) = (< 1, 1 >, < 3, 3 >, < 2, 5 >).$$

As can be seen, instead of five additions between the elements, only one addition and two value copy operations are needed to compute the result. To perform only the necessary elementary operations, some kind of indexing scheme must be used to find the right operands. This bookkeeping causes extra overhead which we want to minimise. In CTSVs the bookkeeping can be avoided totally, since it is done by the compiler with no run-time penalty. The indexing information is contained in the template parameters of CTSV classes. In other words, each vector with a different sparseness pattern is a type of its own.

Template definitions can become lengthy, so the code in this article is given for floating-point vectors instead of generic vectors. It is straightforward to generalise the class definitions by making the element type a template parameter.

A special representation for a vector element is needed:

```
template <unsigned int N> class Elem {
public:
  Elem<N> (float v) {}
  Elem<N> () {}
};
template<> class Elem<1> {
public:
  float value;
  Elem<1>(float v) : value(v) {}
  Elem<1>() {};
};
```

The class Elem has a template parameter N, and the class definition for any N is a class having no data members, just two constructors. For N=1 a specialisation is provided containing a data member for storing a value. So Elem<0> is an empty class, and Elem<1> is a class containing a single floating-point value. Other values than 0 or 1 for N are not meant to be used. The default constructor does nothing. The constructor taking a floating-point parameter initialises the element with a value. To give a uniform interface, the class Elem<0> also has a constructor taking a floating-point value, though it performs no action.

The class representing a CTSV is a collection of Elem<N> objects: Elem<1> objects in nonzero positions and Elem<0> in zero positions. With this kind of element type definitions, we are able to store empty classes as the zero elements. Note however that an object of an empty class may not be totally empty. It is common for a single unused byte to be allocated.

The sparseness pattern of a vector $\boldsymbol{x}$ can be characterised with a bit sequence $\boldsymbol{b_x}$, where 1 corresponds to a nonzero element and 0 to a zero element. Since integral types can be manipulated as bit patterns in C++, an unsigned integer template parameter can be used to represent the bit sequence in the generic CTSV class definition. The bit pattern of this template parameter determines the nonzero positions of the CTSV. The class definition is:

```
template <unsigned int N> class CTSV {
public:
  Elem< N&1 > head;
  CTSV< N>>1 > tail;
  CTSV<N>(float v) : head(v), tail(v) {}
  CTSV<N>() {};
  CTSV< N>>1 >& GetTail() { return tail; }
};
template<> class CTSV<0> {
public:
  Elem<0> head;
  CTSV<0>(float v) : head() {};
  CTSV<0>(){};
  CTSV<0>& GetTail() { return *this; }
};
```

The definition is recursive. The head member holds the value of an element. Whether it will be of type Elem<0> (zero element) or Elem<1> (nonzero element) is determined by the least significant bit of the template parameter N. This can be examined by taking the bitwise AND operation with 1. The tail member holds the remaining part of the CTSV. The value of the template parameter for the next step of the recursion is given by right-shifting N one bit. This will eventually result in the template parameter being zero, and the specialisation for $\mathsf{N} = 0$ ends the recursion.

The default constructor is defined to do nothing. In addition, a constructor taking a single float argument is provided for initialising a vector with a given value. It will pass the same argument to the head and tail members. In the case of Elem<1> type head, the value is stored, otherwise nothing is done, since the respective Elem<0> constructor is empty. The recursion is ended with the empty constructor of CTSV<0>. Since CTSV<0> objects have no tail member, GetTail functions are provided. They are needed in the operator definitions to allow uniform access to the vector tail. We may need to get the tail of a CTSV<0> object, and a tail of a tail of a CTSV<0> object, and so on.

The above class definition generates many function calls. Thus it is crucial that all functions be inlined, so empty functions are discarded from the compiled code.

2.1 CTSV Operations

The most common mathematical operations defined for vectors (addition, subtraction, unary negation, multiplication by a scalar, and dot product) can be implemented easily. The definition of addition for CTSVs is given below. Consider two sparse vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ with bit sequences $\boldsymbol{b_x}$ and $\boldsymbol{b_y}$. Vector $\boldsymbol{x} + \boldsymbol{y}$ has then characteristic bit sequence $\boldsymbol{b_x}$ OR $\boldsymbol{b_y}$. In C++ this is:

```
template <unsigned int N, unsigned int M>
inline CTSV<N|M> operator+(const CTSV<N>& a, const CTSV<M>& b) {
  CTSV<N|M> c; plus<N,M>::add(a,b,c); return c;
};
```

This template can be instantiated with two CTSVs having arbitrary bit sequences. The resulting type is a CTSV having a characteristic bit sequence formed by bitwise OR. The operator+ serves as an interface to the actual addition operation implemented as a static member function add of a generic class plus. The template parameters of the plus class are the characteristic bit sequences of the operands of the addition. The code for the plus class is:

```
template<unsigned int N, unsigned int M> class plus {
public:
  static inline void
  add(const CTSV<N>& a, const CTSV<M>& b, CTSV<N|M>& c) {
    add(a.head, b.head, c.head);
    plus< N>>1, M>>1 >::add(a.GetTail(), b.GetTail(), c.GetTail() );
  }
};
template<> class plus<0,0> {
public:
  static inline void add(const CTSV<0>& a, const CTSV<0>& b,
  CTSV<0>& c){}
};
```

In the body of the operator+, an object of the resulting CTSV type is created and passed to the function plus<N,M>::add along with the vectors to be added. This function adds the heads of the vectors with the add function of the Elem<N> classes (defined below) and calls the plus<N>>1 ,M>>1 >::add function recursively with the tails of the operands. The template parameters are shifted right during the recursion, leading eventually to a call to plus<0,0>::add, which ends the recursion. The add functions for the Elem classes are:

```
inline void add(const Elem<0>& a, const Elem<0>& b, Elem<0>& c){}
inline void add(const Elem<0>& a, const Elem<1>& b, Elem<1>& c){c.v=b.v;}
inline void add(const Elem<1>& a, const Elem<0>& b, Elem<1>& c){c.v=a.v;}
inline void add(const Elem<1>& a, const Elem<1>& b, Elem<1>& c)
        {c.v=a.v+b.v;}
```

The resulting type is “promoted” from Elem<0> to Elem<1> if an Elem<1> type object is involved in the operation. In this way the types are handled correctly. It is easy to see that the compilation of these definitions yields optimal code: for addition of two Elem<0> objects, no code is produced, the addition of Elem<0> and Elem<1> results in a single move, and the addition of two Elem<1> objects generates a single addition.

2.2 Generated Code

To clarify how the above works, we examine the previous example more closely. Vectors $\boldsymbol{x} = (<1,1>, <1,3>)$ and $\boldsymbol{y} = (<2,3>, <2,5>)$ can be constructed and added with the code: `CTSV<5> x(1); CTSV<20> y(2); x+y;` Note that $5 = 00101_2$ and $20 = 10100_2$. Compilation[1] with Borland C++ 5.01 for Intel 80486 using no optimisation resulted the assembly language code:

```
mov dword ptr [ebp-12],1 // create x
mov dword ptr [ebp-7],1
mov dword ptr [ebp-174],2 // create y
mov dword ptr [ebp-169],2
mov eax,dword ptr [ebp-12] // x+y: 1st
mov dword ptr [ebp-192],eax
mov edx,dword ptr [ebp-7] // 3rd
add edx,dword ptr [ebp-174]
mov dword ptr [ebp-187],edx
mov ecx,dword ptr [ebp-169] // 5th
mov dword ptr [ebp-182],ecx
mov eax,dword ptr [ebp-192] // ret
mov dword ptr [ebp-208],eax
mov edx,dword ptr [ebp-187]
mov dword ptr [ebp-203],edx
mov ecx,dword ptr [ebp-182]
mov dword ptr [ebp-198],ecx
```

The initialisations are the two inevitable move commands for x an y, both having two nonzero elements. Two moves (1st and 5th position) and one addition (3rd) are needed to carry out x + y, which can be seen from the resulting code. The last six lines of the assembly language code originate from the return statement and the implicit invocation of the copy constructor. A clever compiler may avoid this by creating the object directly on the caller's stack. Even without a compiler supporting this *named return-value optimisation* technique, the extra copy can be avoided if we content ourselves with a bit more awkward syntax and use the add function of the plus class directly.

[1] Template operator+ was instantiated manually due to limited template support.

3 An Application: Automatic Differentiation

As an alternative to symbolic calculation of derivatives or using approximate difference values, *automatic differentiation* can be used to obtain derivatives. The derivatives are computed using the well-known chain rule. The function value and the derivatives are evaluated simultaneously with the same expression, but instead of scalars we compute using *automatically differentiable numbers* (ADN). These are objects consisting of a function value and a vector of partial derivatives at the same point. To implement the method, we code the differentiation rules for elementary mathematical operations. In C++ this is done by overloading these operations for ADN objects. There are several texts describing automatic differentiation [2, 5] and also software packages available [6, 7].

The derivative vectors of ADNs are usually sparse. Typically there is only one nonzero derivative at the leaves of an expression tree. When approaching the root, the derivative vectors become more dense. CTSVs are ideally suited for ADN derivative vectors.

A class definition for ADNs using CTSVs as the derivative vectors can be easily constructed. Two data members are needed: a floating-point number for the value of the ADN, and a CTSV for the derivatives. So an ADN class is also generic and shares the template parameter of the derivative CTSV. The overloading of elementary mathematical functions and operators for ADN objects is also simple, with vector addition and scalar multiplication of CTSVs defined. To use ADNs in expressions, a certain element position i is chosen for each variable. Then the characteristic bit sequences having only the ith bit set corresponds to the ith variable. The expression is written using ADN objects with these bit sequences: ADN<1>, ADN<2>, ADN<4> and so forth.

Since CTSVs have no run-time penalty for indexing, ADN expression calculations can be further improved. At the leaf level of the expression tree, the derivatives are either 0 or 1, leading to many multiplications by 1. These can be avoided by keeping track of the positions of 1's. Instead of having zero and nonzero elements, we then have zero, unity and arbitrary elements. There is of course no point in tracking the unity elements at run time just to replace a few multiplications with value moves. But since with CTSVs the tracking is done at compile time, it is perfectly feasible and will in some cases produce significant savings in computation time. Since we now distinguish between three types of elements, there must be three specialisations of the `Elem` class, so two bits of the characteristic bit sequence of the CTSV are needed for each element. See [4] for C++ code and a more detailed discussion about CTSVs in automatic differentiation.

4 Test Runs

The speed of CTSV operations was compared with ordinary dense vectors (regular C arrays) and sparse vectors with indexing performed at run time (vectors of value/index pairs). The speeds of the addition operations were measured with

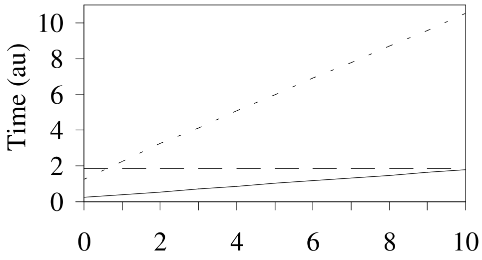

Fig. 1. CPU Time of Vector Addition. *Solid line:* CTSVs; *Dashed line:* dense vectors; *Dotted line:* run-time sparse vectors. Number of nonzero elements is on the x-axis.

10-element vectors, and the number of nonzero elements ranged from 0 to 10. The code was compiled with Borland C++ 5.01 for Intel Pentium processor and optimised for speed. Since the optimiser could not perform return-value optimisation, we used the more awkward syntax to avoid the extra copy constructor invocations. The sparse vectors were allocated statically. Consequently the run-time penalty originates only from indexing, not from memory allocation. The addition functions were called in a loop, and the operand parameters were passed by reference. The CTSV addition for 0 nonzero elements yielded no code. Hence the cost of the function call and looping should approximately equal the cost of the `CTSV<0>` case. The results are shown in Fig. 1.

The speed of ADN expressions was compared to manually optimised symbolically differentiated code. The ADN derivative vectors were implemented as CTSVs (using the refinement described above), as ordinary dense vectors and as ordinary sparse vectors. Results of the test runs are shown in Fig. 2.

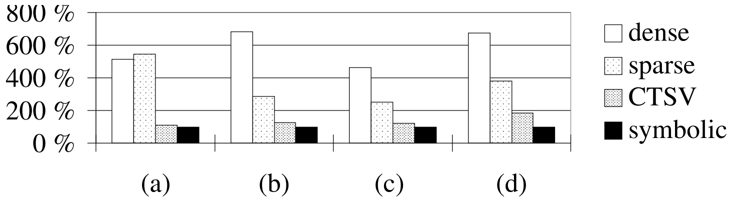

Fig. 2. Relative CPU Time (symbolic = 100%) for Computing Expressions and their Derivatives. (a,b): a multiplication of five variables differentiated with respect to all variables. (c,d): $\sum_{i=1}^{3} a_i e^{b_i t}$ differentiated with respect to a_i and b_i. (a,c): `double` variables. (b,d): `complex` variables.

5 Discussion

A collection of classes for representing sparse vectors in C++ was described. The sparseness pattern of the vector must be known at compile time. For special applications where this restrictive precondition is met, very efficient code can be generated. It was shown how common vector operators can be overloaded for the presented CTSV classes to generate this minimal code from abstract vector expressions.

The classes rely heavily on C++ templates, especially on recursive definitions with non-type template parameters. The sparseness pattern of each vector is represented as a template parameter, i.e. as part of the type information. The sparseness pattern changes in vector operations. Each operation may potentially produce a new vector type. These new template instances are automatically generated from the vector templates by the compiler when encountered. Some of the template features used are quite new and not yet available at all compilers. The features are however part of the standard proposal for C++ [3].

Execution of selected vector operations was compared with dense vectors and ordinary sparse vectors. The speed of CTSVs outperforms both alternatives. The execution speed depends to some extent on the compilers optimisation capabilities; for best results the compiler should be capable of return-value optimisation.

Automatic differentiation was presented as an application of CTSVs. It was shown how to define template classes for automatically differentiable numbers using CTSVs as the derivative vectors. Speed of evaluation for some common expressions was compared with ADNs having alternative vector representations. With our examples, ADNs with CTSV derivatives required CPU time ranging from 20% to 50% of the time used by ADNs with alternative vector representations. The execution speed was only slightly slower than the speed of the symbolically differentiated code.

References

1. Veldhuizen, T.: Using C++ template metaprograms. C++ Report **7** (1995) 36-43.
2. Rall, L. B.: Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science **120** (1981) Springer-Verlag, Berlin.
3. 1997 C++ Public Review Document: Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++. ANSI X3J16/96-0225 ISO WG21/N1043.
4. Järvi J.: Compile Time Sparse Vectors in C++. Turku Centre for Computer Science Technical Report No. 107 (1997) (http://www.tucs.abo.fi/publications).
5. Barton, J. J., Nackman L.R.: Scientific and Engineering C++. Ch. 19, Addison-Wesley, Reading Massaschusetts, 1994.
6. Griewank, A., Juedes, D., Utke, J.: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. ACM Transactions on Mathematical Software **22** (1996) 131–167.
7. Bischof, C. H., Carle, A., Corliss, G. F., Griewank, A., Hovland, P.: ADIFOR: Generating derivative codes from Fortran programs. Sci. Prog. **1** (1992) 1-29.