Parallel expression template for large vectors

Laurent Plagne, Frank Hülsemann, Denis Barthou, Julien Jaeger

Parallel Expression Template for Large Vectors

Laurent Plagne
EDF R&D
1 av. du Général de Gaulle
F-92141 Clamart France
laurent.plagne@edf.fr

Frank Hülsemann
EDF R&D
1 av. du Général de Gaulle
F-92141 Clamart France
frank.hulsmemann@edf.fr

Denis Barthou
University of Versailles
St Quentin / INRIA, France
denis.barthou@uvsq.fr

Julien Jaeger
University of Versailles
St Quentin, France
julien.jaeger@uvsq.fr

ABSTRACT

This paper describes a short and simple way of improving the performance of vector operations (e.g. $X = aY + bZ + ..$) applied to large vectors. In a previous paper [1] we described how to take advantage of high performance vector copy operation provided by the ATLAS library [2] in the context of C++ Expression Template (ET) mechanism. Here we present a multi-threaded implementation of this approach. The proposed ET implementation that involves a parallel blocking technique, leads to significant performance increase compared to existing implementations (up to $\times 2.7$) on dual socket x86_64 targets.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Parallel programming*

General Terms

C++ template, Expression Template, Parallel Computing, OpenMP, Intel TBB, multi-core processors

1. INTRODUCTION

In this paper we propose a simplified C++ implementation of a linear algebra vector class that allows composing compact and abstract vector expressions such as:

$$\mathtt{X = a * Y + b * (Z + W);} \quad (1)$$

This implementation is based on the Expression Template mechanism (ET) introduced by Veldhuizen [3] and Vandevoorde [4]. ET allows avoiding temporary vectors and the performances of abstract expressions like (1) compete with the ones of the corresponding low-level (loop-based) basic implementations such as:

$$\mathtt{for\ (i = 0;\ i < N;\ i{+}{+})\ X[i] = a * Y[i] + b * (Z[i] + W[i]);} \quad (2)$$

In a previous paper [1] we illustrated the gain to be obtained from mixing our ET based vector class with the ATLAS [2] implementation of the procedural BLAS library. The high performance ATLAS implementation of the vector copy operation, which relies partly on low level assembly language, is used as a kernel for the vector ET evaluation.

In this paper, we present two multi-threaded implementations of our ET based vector class based on Intel's Threading Building Blocks and on OpenMP respectively. The ubiquitous presence of multicore architectures has rekindled the interest in shared memory parallel programming models. While the computational power of a chip scales almost linearly with the number of cores, this is not the case for the memory access bandwidth. Hence, the fraction of the so-called *memory bound* applications, which feature performances that are limited by the memory access bandwidth of target architecture, increases with the mean number of cores included in micro-processors.

Vector expressions like (1) exhibit a small arithmetic intensity and are strongly *memory bound*. Hence it might be surprising that multi-threaded implementations accelerate these tasks significantly on shared memory multi-core machines. Nevertheless, we have observed a $\times 2.5$ acceleration factor on dual socket quadricore processors compared to our previous implementation. The resulting vector class allows composing abstract vector expressions like (1) that perform better than both loop-based implementations such as (2) and off-the-shelf vector libraries such as Blitz++ [5], uBLAS [6] or std::valarray. This performance gain reaches a factor of $\times 2.7$ for large vectors.

The paper is organised as follows: Section 2 presents the considered vector operations. Section 3 presents the large vector operations as typical memory bound problems. Section 4 gives a short description of our C++ vector class implementation based on ET. Performance measurements show that this implementation avoids abstraction penalties. Section 5 presents our enhanced ET vector class relying on the ATLAS dcopy kernel. Performance measurements are car-

Table 1: Description of the four target architectures for performance measurements

Processor	# cores	frequency	RAM	Compiler
Intel Xeon E5570 Nehalem	2×4	2.9 GHz	18 GB	g++ 4.3.3
Intel Xeon X7310 Tigerton	4×4	1.6 GHz	48 GB	g++ 4.3.3
Intel Xeon E5410 Harperton	2×4	2.3 GHz	8 GB	g++ 4.3.3
AMD Opteron 8347 HE	2×4	1.9 Ghz	4 GB	g++ 4.3.0

ried out on four different architectures. Section 6 presents two parallel implementations of our ET vector class and the corresponding results.

2. VECTOR DEFINITION AND TEST PLATEFORMS

Let us first define the scope of this paper and what we refer to as *vector operations*. From the linear algebra point of view, vectors can be defined as *indexed* collections of *numerical* elements of the same type. *Indexed* means that the value of every vector elements can be accessed from a given integer *index* to be chosen in a given *range*. While a wide variety of linear algebra vector types (sparse, multidimensional,...) can be considered, we will focus on simple vector types where real type elements (single or double precision) are stored in basic containers that can be defined and exchanged through the following common programming language: F77, C and C++. Within these languages, the location of a contiguous memory region containing a given number of floating point elements, can be manipulated either as a pointer type (C and C++) or as an array type (F77). These arrays are the main Input/Output types for the Basic Linear Algebra Subroutines (BLAS) API [7].

2.1 Performance Measurements and Target Architectures Description

This paper is based on performance measurements that have been carried out on different x86_64 target architectures. Table 1 provides a short description of the main features of theses machines. In the following, the performance curves will be named after these four architectures. Most of the time, the different targets exhibit the same kind of performance behavior. In this case, we will report only the Xeon E5410 curves. The performance measurements are carried out with the tools developed by the BTL++ project [8].

3. LARGE VECTOR OPERATIONS AND SUPERSCALAR ARCHITECTURE

Fig. 1 shows the performance of the vector operation $Y \leftarrow \alpha X + Y$ (`axpy`) on a Pentium Xeon E5410 using respectively single and double precision floating point elements. Performance is maximal for vector in the range of $[10^2, 10^3]$ elements. From sizes around 10^4, performance decreases and reaches its lowest level when vector sizes exceed 10^5 elements. The reason for this behavior, which is common to all vector operations, is that the performance is mainly driven

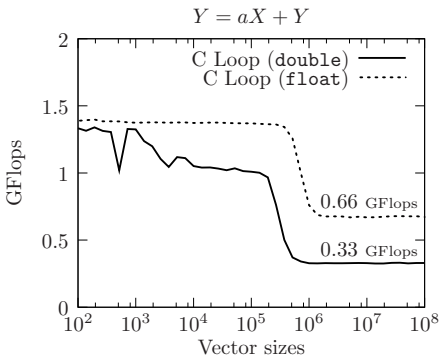


Figure 1: Performance results of axpy operations on the Xeon E5410 (gcc 4.3.3).

by the memory access bandwidth. This is true for all computations involving a ratio r defined by:

$$r = \frac{\text{number of memory accesses (read+write)}}{\text{number of floating point operations}},$$

that is not small compared to 1. In this case, ($r = 3/2$ for the `axpy` operation), performance depends on memory access bandwidth. Most modern architectures exhibit a memory cache hierarchy with high bandwidth for data access inside the cache (data size $\leq$ a few MBytes) and a much lower bandwidth for data access in the main memory (a few MBytes $\leq$ data size $\leq$ a few GBytes). This explains the low level of performance observed for large vectors that do not fit in the cache hierarchy:

$$\text{Large vectors: vector sizes} \in [10^6, 10^8].$$

Since the double precision `axpy` operation requires twice as much memory bandwidth as the single precision one, it runs naturally 2 times slower for large vectors (0.33 Gflops vs 0.66 Gflops).

4. MINIMAL C++ VECTOR CLASS

This section presents a minimal vector C++ class based on Expression Template mechanism.[1]

4.1 Expression Template

ET have been introduced by T. Veldhuizen [3] and D. Vandevoorde [4]. Applied to a vector class, ET allows writing arbitrarily complex vector expressions such as:

```
R=2.0*X+2.0*(Y-Z*2.0);
```

that do not imply temporary vector construction and do not incur any performance penalties compared to the corresponding loop-based implementation:

```
for (int i=0 ; i < N ; i++)
 R[i]=2.0*X[i]+2.0*(Y[i]-Z[i]*2.0);
```

[1]The complete sources of this class can be obtained from the authors.

4.2 The Curiously Recurring Template Pattern (CRTP)

Our proposed ET implementation uses the C++ Curiously Recurring Template Pattern [9, 4] (CRTP) that allows grouping a set of classes in a template-based hierarchy. This hierarchy reflects a common behavior for the class set elements and does not involve any virtual functions. As Vandevoorde and Josuttis write, this pattern "*consists of passing a derived class as a template argument to one of its own base classes*" [4]:

```
class Derived : public Base<Derived>{}
```

This pattern is used to gather a set of classes {`Derived1`, `Derived2`,...} as an ensemble of `Base<>` classes. In our vector case, let us first define the template base class `BaseVec<>`:

```
template <class DERIVED>
class BaseVec{
public:
  typedef const DERIVED & CDR;
  inline CDR getCDR( void ) const {
    return static_cast<CDR>(*this);
  }
};
```

The static cast method `getCDR()` allows extracting the embedded `DERIVED` object from its `BaseVec<>` capsule. The `DERIVED` template class parameter is one of the three following classes:

1. `Vec<>`
2. `VecExpr<>`
3. `VecScalExpr<>`

Fig. 2 presents this template inheritance relationship.

`VecExpr<>` and `VecScalExpr<>` instances are constructed by arithmetic operators applied to `BaseVec<>` objects. These two classes store references to the operands. In addition, the `VecExpr<>` class statically defines the type of operation (`+` or `-`) as a template parameter:

Operators `+/-`:

`BaseVec<L>+BaseVec<R>` → `VecExpr<L,Add,R>`

`BaseVec<L>-BaseVec<R>` → `VecExpr<L,Minus,R>`

Operator `*`:

`scalar*BaseVec<V>` → `VecScalExpr<V>`

`BaseVec<V>*scalar` → `VecScalExpr<V>`

Operator `=`

`V<T>=BaseVec<T>` → `BaseVec<T>[i]` evaluation

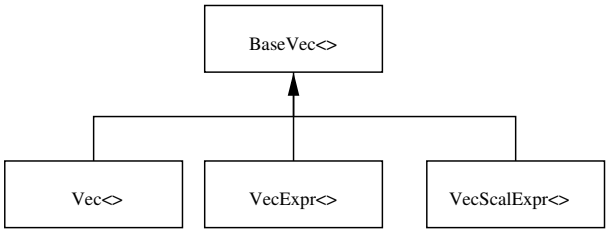


Figure 2: `BaseVec` class hierarchy.

Both expression classes define an operator `[]` that performs the actual evaluation of the expression. Note that this evaluation is not performed at the expression classes construction stage. This is a lazy evaluation process that allows avoiding temporary vectors involved in standard implementations of operators. A more detailed presentation of these classes is given in [1].

4.3 `Vec<T>`

The class `Vec<T>` is the main vector class. Its implementation is classical except for the assign operator `=` specialized for `BaseVec<DERIVED>` right hand side:

```
template <class ELEMENT_TYPE>
class Vec : public BaseVec< Vec<ELEMENT_TYPE> >
{
public:
  ...
  template <class DERIVED>
  Vec & operator =(const BaseVec<DERIVED> & v){
    const DERIVED & r=v.getCDR();
    for (int i=0 ;i<size_ ;i++) data_[i]=r[i];
    return (*this);
  }
  ...
private:
  ELEMENT_TYPE * data_;
  int            size_;
};
```

The template parameter `ELEMENT_TYPE` is either `float` or `double` and the vector elements are stored in a C array of this type (`data_`). The operator `=` evaluates the right operand value that can be the result of an arbitrarily complex expression.

4.4 Expression Template Performance

A large variety of vector expressions is handled by this ET vector class implementation. In this paper we present performance measurements carried out for a limited set of vector expressions that should give a fair picture of the general performance level to be expected from the `Vec` implementation.

4.4.1 Considered Set of Vector Operations:

The vector operations that we will study in this paper are linear combinations of vectors:

$$T = \sum_{i=0}^{N_c} a_i S_i + \alpha T, \text{ with } \alpha \in \{0, 1, -1\}$$

where T is a given target vector, $\{S_i\}$ is a set of source vectors of the same size and $\{a_i\}$ the corresponding set of scalar factors. These combinations can be characterized by the number N_c of involved source vectors:

- $N_C = 1$: unary combinations (part of L1 BLAS API).
- $N_C = 2$: binary combinations.
- $N_C = 3$: ternary combinations.
- ...

5. EXPRESSION TEMPLATE AND ATLAS BASED BLOCKED EVALUATION

In this section, we combine the previous Expression Template mechanism with a blocked copy technique. The principle is to take advantage of the high performance copy operation provided by the ATLAS library.

5.1 Vector Class Modification

The only change in the `Vec` class is a new definition of the operator `=`:

```
template <class ELEMENT_TYPE>
class Vec : public BaseVec< Vec<ELEMENT_TYPE> >
{
  typedef BlockAssign<ElementType> BA;
public:
  ...
  template <class DERIVED>
  Vec & operator = (const BaseVec<DERIVED> & v){

    BA::apply(size_,v.getCDR(),data_);
    return (*this);
  }
  ...
};
```

where the `BlockAssign` template class implementation is specialized for `double` elements as:

```
template <>
struct BlockAssign<double>{
  typedef double * Data;
  static const int largeSize=50000;
  static const int blockSize=1024;

  template <class DERIVED>
  static void apply(int N,
                    const DERIVED & source,
                    Data & target) {
    if (N<largeSize){
      for (int i=0 ; i < N ; i++)
        target[i]=source[i];
    }
    else{
      double * tempo = new double[blockSize];
      const int nblocks=N/blockSize;
      int offset=0;
      for (int i=0 ; i<nblocks ; i++){

        for (int j=0 ; j < blockSize ; j++)
          tempo[j]=source[j+offset];
        ATL_dcopy(blockSize,tempo,1,
                  target+offset,1);
        offset+=blockSize;
      }
      for (int i=offset ; i < N ; i++)
        target[i]=source[i];
      delete[] tempo;
    }
  }
};
```

5.2 Sequential Blocked Results

Fig. 3 shows the performance results of binary and ternary linear combinations implemented via our ET `Vec` class, with and without blocking, and via direct loop-based C implementations. One can see that the abstract expression of the combinations does not lead to any performance penalties. Moreover, the blocked ET implementation accounts for a performance improvement on both binary (+25%) and ternary (+16%) vector combinations.

6. PARALLEL EXPRESSION TEMPLATE

In this section, we present a parallel implementation of the previous Blocked Expression Template mechanism. The possibility of such an approach is mentioned in the reference [10].

6.1 Vector Class Modification

The only change in the `Vec` class is a replacement of the `BlockAssign` template class by the `OpenMPBlockAssign` template class:

```
template <class REAL>
struct OpenMPBlockAssign{
  typedef REAL       RealType;
  typedef RealType * Data;

  static const int largeSize=100000;
  static const int blockSize=1024;

  template <class DERIVED>
  static ivoid apply(int N,
                     const DERIVED & source,
                     Data & target) {
    if (N<largeSize){
      for (int i=0 ;i<N ;i++)
        target[i]=source[i];
    }
    else{
      const int nblocks=N/blockSize;
#pragma omp parallel for schedule(static)
      for (int i=0 ; i<nblocks ; i++){
        REAL * tempo = new REAL[blockSize];
        const int offset=i*blockSize;
        for (int j=0 ; j < blockSize ; j++)
          tempo[j]=source[j+offset];
        ATL_dcopy(blockSize,tempo,
                  1,target+offset,1);

        delete[] tempo;
      }
      for (int i=blockSize*nblocks ;i<N ; i++)
        target[i]=source[i];
    }
  }
};
```

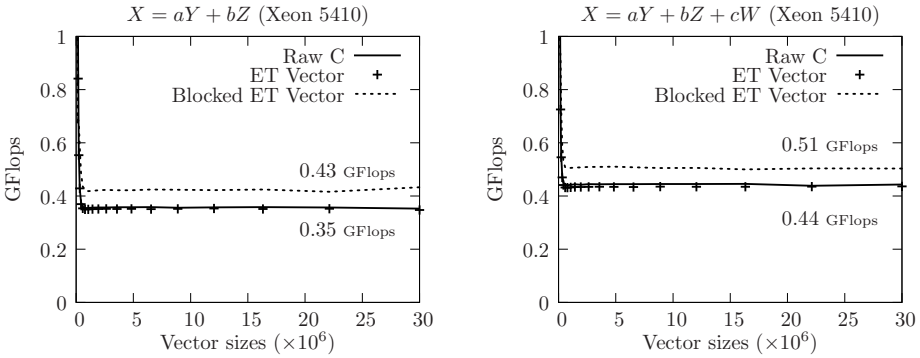


Figure 3: Performance results of binary (left) and ternary (right) combinations using the Blocked ET `Vec` class and loop-based implementations (`double` precision vectors ; Xeon 5410 with gcc 4.3.3).

In Fig. 4 we compare this implementation to another parallel implementation of the `BlockAssign` based on the Intel Threading Building Blocks [11].

6.2 Parallel Results

Fig. 4 shows the performance improvement that this parallel blocked ET implementation entails for both binary ($\times 2.7$) and ternary ($\times 2.6$) vector combinations.

When vector sizes exceed the total size of the cache hierarchy, the performance improvement remains small compared to the number of cores involved in the computation. The main reason is that performance is then determined by the memory bandwidth and other hardware mechanisms such as the size of the in-flight cache requests for each cache.

In Fig. 5, we use the STREAM benchmarks [12, 13] to evaluate memory performance of our implementation on different architectures. These benchmarks correspond to OpenMP parallel C code for some few particular vector expressions (copy, triad, daxpy for instance) and are used here as reference code. The benchmarks are compiled using the same `gcc` versions, and Intel `icc` version 10.1.

Fig. 5 shows different important results:

- Memory bandwidth usage increases as the number of threads increases. The increase is not linear (from 2 to 4 threads for the Xeon machine for instance), performance only slightly increases. This is due to the fact that cores do not have a uniform memory access. On each chip, four cores compete for the access to memory through the same Front Side Bus. Further more, two cores share the same L2 cache, that can sustain a limited number of in-flight memory requests (cache misses). This could explain for the performance stall in the Xeon machine between 2 and 4 threads.
- Performance of expression template code is comparable to C code. There is no loss of performance, while there is a gain in the abstraction of the formulation.
- There appears a difference between performance obtained through gcc and performance obtained with the Intel compiler. While this provides an upper and sustainable bound on memory bandwidth, investigation on the causes of this difference is left for future work.

The apparent memory bandwidth obtained does not correspond to the peak memory bandwidth of the machine, and changing for instance the compiler (considering icc instead of gcc on the Intel machine) would probably lead to some further performance improvements. Depending on how the assembly instructions are scheduled, cycles taken in the decoding phase of the program execution and usage of the functional units will differ (removing possible stalls for instance).

Finally, Fig. 6 shows the variation on memory bandwidth performance according to the number of operands in vector expressions. While on the 16-core Xeon, it appears that performance improves when the number of operand increases, this is the opposite for the Opteron machine. In the Xeon case, this shows that the limit of memory requests that can be in-flight at some point of the computation has not been reached, even using ternary expressions. Memory requests are not serialized and there is some amount of overlap during their resolution, accounting for the performance increase. In the Opteron case, on the contrary, some requests are serialized or introducing some additional stalls. This may occur in the cache hierarchy and more detailed explanation requires further investigation.

6.3 Comparison with other libraries

We have carried out performance comparisons with other available expression template linear algebra libraries: armadillo [14], Blitz++ [5], DealII [15], Eigen [16], FLENS [17], genial [18], gmm++ [19], MTL4 [20], Seldon [21], uBlas [6] and std::valarray.

Fig. 7 shows that the proposed implementation outperforms all the available libraries for the $X = aY + bZ$ and $X = aY + bZ + cW$ operations. In the case of the $X += Y$, $X += aY + bZ$ and $X += aY + bZ + cW$ operations, the libraries Seldon, FLENS and gmm++ remain competitive compared to our parallel implementation. These libraries replace these operations by successive `daxpy` calls. For example $X += aY + bZ$ can be written as :

```
daxpy(N,coefY,Y,1,X,1);
```

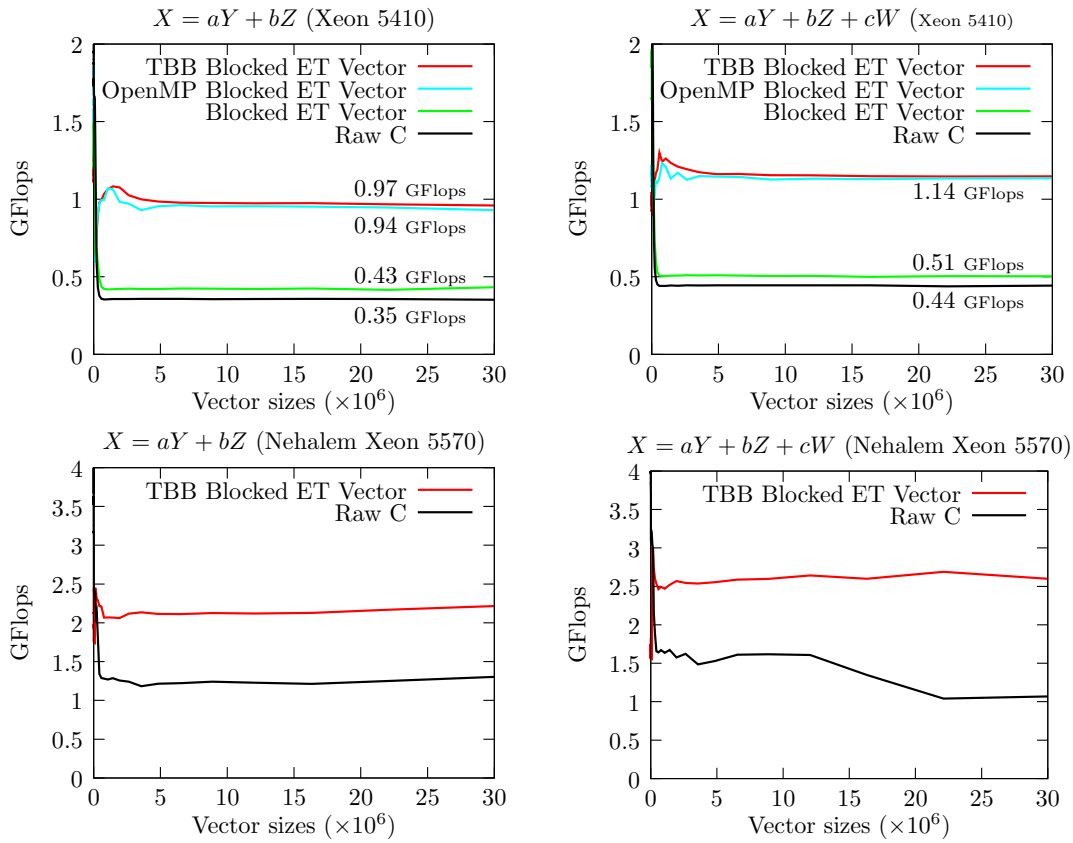


Figure 4: Performance results of binary (left) and ternary (right) combinations using the Parallel Blocked ET `Vec` class and loop-based implementations (`double` precision vectors gcc 4.3.3).

```
daxpy(N,coefZ,Z,1,X,1);
```

The `daxpy` operations are performed by the Intel Math Kernel Library (v10.1) [22] which provides a parallel implementation.

7. CONCLUSION AND OUTLOOKS

This paper presents some new formulation for vector expressions, using Expression Templates. This formulation is a short and simple way to describe vector expressions and improve performance. Parallelism expressed as OpenMP code is used in the vector class definition, but does not appear at all the vector expression level and the same expression may be implemented using different strategies.

Large vector operations correspond to codes accessing data out of cache. Compared to a large panel of different dedicated libraries (such as BLAS) and template libraries, the Expression Templates proposed in the paper outperform them by at least a factor of 2, on different Xeon and Opteron parallel architectures. Compared to OpenMP C code, our templates exhibit the same level of performance.

Finally, when vectors are too large to fit in caches, performance is driven by memory accesses. Experiments with Intel icc compiler and gcc compiler show that there is still a gap between the best performance achieved with the Expression Templates and a maximal sustainable performance. This requires further investigation left for future work, in particular by looking for stalls due to cache usage (saturation of the number of memory requests) and not due to Front Side Bus or memory bank limited bandwidth.

Note that all the reference urls have been checked and were found to be valid on June 12 2009.

8. REFERENCES

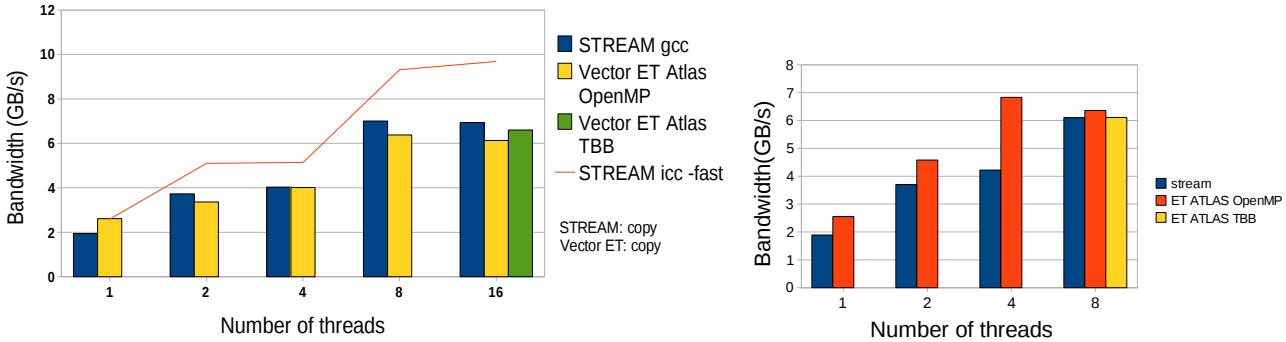
[1] Plagne, L., Hülsemann, F.: Improving Large Vector Operations with C++ Expression Template and ATLAS (2007) MPOOL07 web page: `http://homepages.fh-regensburg.de/~mpool/mpool07/programme.html`.

[2] Whaley, R.C., Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS. Software: Practice and Experience **35**(2) (February 2005) 101–121 "ATLAS web page: `http://math-atlas.sourceforge.net`".

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

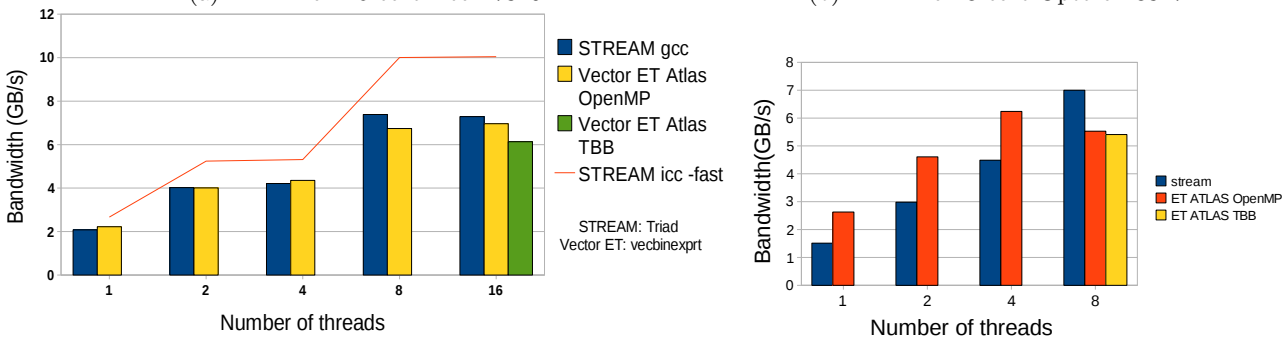
[4] Vandevoorde, D., Josuttis, N.M.: C++ Templates. Addison-Wesley, BostonMA, MA USA (2002)

[5] Veldhuizen, T.L.: Arrays in blitz++. In: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98).



(a) $X = Y$ on 16-core Xeon 7310

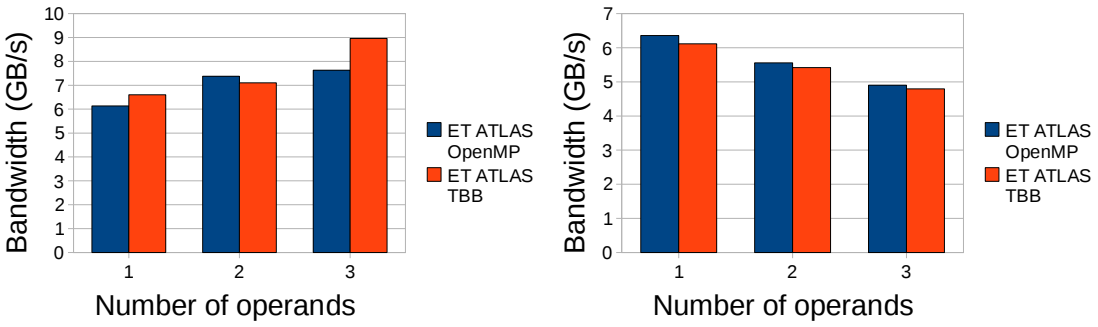
(b) $X = Y$ on 8-core Opteron 8347 HE



(a) $X = aY + bZ$ on 16-core Xeon 7310

(b) $X = aY + bZ$ on 8-core Opteron 8347 HE

Figure 5: Sustained memory bandwidth for out-of-cache vector expressions using the OpenMP C code (Stream), Parallel Blocked ET `Vec` class using OpenMP and TBB, depending on the number of threads used.



(a) 16-core Xeon 7310

(b) 8-core Opteron 8347 HE

Figure 6: Sustained memory bandwidth for out-of-cache vector expressions using Parallel Blocked ET `Vec` class with OpenMP and TBB, according to the arity of the vector expression.

Lecture Notes in Computer Science, Springer-Verlag (1998)

[6] Walter, J., Koch., M.: uBLAS web page: `http://www.boost.org/libs/numeric/ublas`.

[7] 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**(2) (June 2002) 135–151

[8] Plagne, L., Hülsemann, F.: BTL++: From Performance Assessment to Optimal Libraries. In Bubak, M., van Albada, G.D., Dongarra, P.J.J., Sloot, M., eds.: Proceedings of the 8th International Conference on Computational Science (ICCS'08), Part III. Volume 5103 of LNCS., Kraków, Poland, Springer-Verlag (June 2008) 203–212

[9] Barton, J.J., Nackman, L.R.: Scientific and Engineering C++. Addison-Wesley, Reading, MA (1994)

[10] Czarnecki, K., Odonnell, J.T., Striegnitz, J., Walid, Taha: DSL Implementation in MetaOCaml, Template Haskell, and C++. LNCS: Domain-Specific Program Generation **3016**(2) (2004) 51–72

[11] Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc. (2007)

[12] D.McCalpin, J.: Memory bandwidth and machine balance in current high performance computers. IEEE

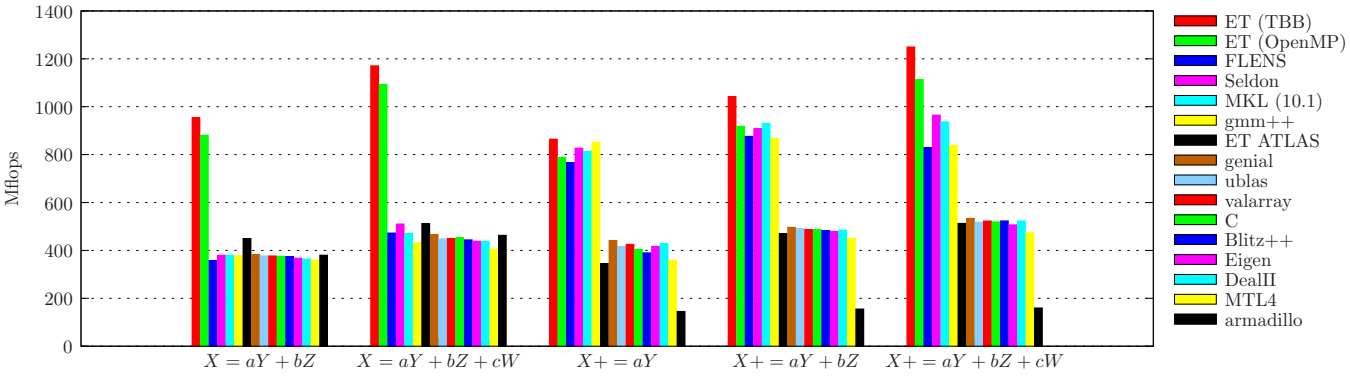


Figure 7: Out of Cache performance of Vector Expressions with various Libraries (Xeon 5410). Theses figures correspond to the mean performance measured for large vectors with 10^6 to 10^7 entries. Note that the order of the libraries in the color legend corresponds to the order of the bar charts from left to right within each operation bar clusters.

Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (December 1995) `http://www.cs.virginia.edu/stream/`.

[13] McCalpin, J.D.: Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia (1991-2007) A continually updated technical report. http://www.cs.virginia.edu/stream/.

[14] Sanderson, C.: Armadillo web page: `http://arma.sourceforge.net`.

[15] Bangerth, W., Hartmann, R., Kanschat, G.: deal.II — a general-purpose object-oriented finite element library. ACM Trans. Math. Softw. **33**(4)

[16] Guennebaud, G., Jacob, B.: Eigen web page: `http://eigen.tuxfamily.org`.

[17] Lehn, M., Stippler, A., Urban, K.: FLENS-a flexible library for efficient numerical solutions. In: Proceedings of Equadiff. Volume 11. (2005) 467–473

[18] Laurent, P.: genial web page: `http://www.ient.rwth-aachen.de/~laurent/genial/genial.html`.

[19] Renard, Y.: gmm++ web page: `http://home.gna.org/getfem/gmm_intro`.

[20] Gottschling, P., Witkowski, T., Voigt, A.: Integrating object-oriented and generic programming paradigms in real-world software environments: Experiences with AMDiS and MTL4. In: POOSC 2008 workshop at ECOOP08, Paphros, Cyprus. (2008)

[21] Durufle, M., Mallet, V.: Seldon web page: `http://seldon.sourceforge.net`.

[22] Intel: MKL web page can be found from: `http://www.intel.com`.