BTL++: From Performance Assessment to Optimal Libraries

Laurent Plagne and Frank Hülsemann

EDF R&D, 1 avenue du Général de Gaulle, BP 408, F-92141 Clamart, France
{laurent.plagne,frank.hulsemann}@edf.fr

Abstract. This paper presents the *Benchmark Template Library in C++*, in short BTL++, which is a flexible framework to assess the run time of user defined computational kernels. When the same kernel is implemented in several different ways, the collected performance data can be used to automatically construct an interface library that dispatches a function call to the fastest variant available.

The benchmark examples in this article are mostly functions from the dense linear algebra BLAS API. However, BTL++ can be applied to any kernel that can be called by a function from a C++ main program. Within the same framework, we are able to compare different implementations of the operations to be benchmarked, from libraries such as ATLAS, over procedural solutions in Fortran and C to more recent C++ libraries with a higher level of abstraction. Results of single threaded and multi-threaded computations are included.

1 Introduction

Linear algebra is a field of particular relevance in scientific computing in both, academia and industry. Operations on matrices, dense or sparse, or vectors, large or small, feature in many if not most projects. The prominent position of the topic is reflected in a large number of available implementations.

For a project relying on the BLAS interface on a given target architecture, for example x86, one has the choice between netlib's default implementation [1], ATLAS [2], MKL from Intel [3], ACML from AMD [4] and GotoBLAS [5] to name just a few. Furthermore, there exist other solutions such as Blitz++ [6], MTL [7] or uBLAS [8], which offer the same functionality but use their own interfaces. The obvious question for a user who has to choose among the options is then: "Which option works best in a given computing environment?" In order to answer this question, the user has to solve another problem first: "How to assess and how to compare the performances of the different options?"

This paper describes the *Benchmark Template Library in C++* (BTL++) project [9] which is a flexible and user extendible benchmarking framework for computational "actions" (kernels). The extendibility by the user was the dominant design goal of the project. It relies on three main concepts: *implementations, computational actions and performance analysers*. We would like to point out

M. Bubak et al. (Eds.): ICCS 2008, Part III, LNCS 5103, pp. 203–212, 2008.

that by providing realisations of these concepts, a user can interface new libraries, add computational kernels to the benchmarking suite or change the way how or even the type of performance that is measured. Loosely speaking, as long as an implementation of some computational action can be called by a routine from within a C++ program, the chances are high that this implementation can be included in the benchmarking process. The nature of the computational action to be assessed can, of course, be defined by the user. Although BTL++ itself provides different performance analysers that have been sufficient for our purposes, its aim is not to replace dedicated profiling tools like PAPI [10] for instance. On the contrary, the performance counters of PAPI have already been used successfully as performance analysers in the BTL++ framework.

Among the numerous publicly available benchmark suites, BTL++ is related in spirit to the BenchIT [11] project that provides detailed performance evaluations for a fixed set of numerical kernels. While the BenchIT project offers a rich and mature database interface that gathers the performance data for a large variety of architectures, BTL++ is a library that emphasises the extendibility by the user. This generic feature makes BTL++ a very flexible tool with respect to the computational kernels, their implementations and the benchmarking methods.

Originally designed as a flexible benchmark tool, BTL++ now features an optional library generation stage. From the collected measurements, BTL++ can create a new optimal library that routes the user's function calls to the fastest implementation at her disposal. Our performance results underline that the problem size has to be taken into account in the routing process, as the implementation that is fastest for small problems is not necessarily also the fastest on larger ones. Although such a performance assessment/generation sequence is successfully used in projects like ATLAS and FFTW [12], the BTL++ positions itself at a different level since it only generates an **interface** library based on existing implementations. In other words, ATLAS or FFTW are stand alone projects that provide implementations which can be used by BTL++ to generate the optimal interface library on a given machine for a given numerical kernel.

The article is structured as follows. In Sect. 2, the principal building blocks in the implementation of BTL++ are introduced. Section 3 shows benchmark results for both BLAS and non BLAS kernels. In Sect. 4 we use again BLAS examples to demonstrate the BTL++ optimal library generation stage. In Sect. 5 we present our conclusions and discuss some directions for future work.

2 BTL++ Implementation

The development of a Linear Algebra (LA) based code can follow numerous different strategies. It can involve C++ generic LA libraries (uBLAS, MTL, ...), one of the various BLAS implementations (ATLAS, GotoBLAS,...), or being directly hand coded using raw native languages (C/C++, F77, ...). The BTL++ project aims to compare the performance of these strategies. Even though not all of the different strategies take the form of a library in the linker sense, we refer to all implementations for which a BTL++ library interface exists, as BTL++ *libraries*.

The performance of each such BTL++ library is evaluated against a set of computational actions, in our case, LA numerical kernels such as the dot product, the matrix-matrix product and vector expressions like $W = aX + bY + cZ$. Note that the BTL++ kernels are not limited to the operations included in the BLAS API. Also note that a BTL++ library does not have to implement all computational kernels to be included in the comparison. Hence we can take into account specialised implementations like the MiniSSE library [13] for example, which implements only a subset of the BLAS interface. Last, BTL++ provides a set of different performance evaluation methods. This open-source project relies on cooperative collaboration and its design aims to obtain a **maximal legibility and extendibility**. To be more explicit, a user is able to extend the collection of available BTL++ libraries, kernels and performance evaluation methods, as easily as possible. The implementation of BTL++ with object-oriented and generic programming techniques in C++ results in a good modularity and an efficient factorisation of the source code.

The performance evaluation of the axpy operation ($Y = a * X + Y$) using the Blitz++ library will be used as a running example to illustrate the different parts of the BTL++ design.

2.1 BTL++ Libraries Interfaces: The *Library_Interface* Concept

Before measuring the performance of an implementation, one should be able to check its correctness for all considered problem sizes. In order to ease this calculation check procedure, a pivot (or reference) library is chosen to produce the reference results. Since BTL++ is written in C++, the STL has been a natural choice for being the BTL++ reference library. The two vector operands (X and Y) of the axpy operation are initialised with pseudo-random numbers. This initialisation is first performed on the reference STL operands (X_STL and Y_STL). Secondly, these vectors are used for initialising the corresponding Blitz++ operands (X_Blitz and Y_Blitz) via a vector copy. Then, both the STL and Blitz++ libraries perform the axpy operation. A copy operation from a Blitz++ vector to a STL vector and a comparison of two STL vectors are used to check the result.

Obviously, some of the Blitz++ functions that implement the vector copy operations from and to STL vectors could be reused for the implementation of this init/calculate/check procedure applied to another vectorial BTL++ kernel. Moreover, the same functionality has to be implemented for all libraries in the BTL++ library set. In order to give a standardised form for these functions, the BTL++ framework defines the *Library_Interface* concept that consists in the set of constraints on the types and methods that a *Library_Interface model* class must fulfil. See Table 1a) for details. Note that a given *Library_interface model* class can define an incomplete subset of the BTL++ kernels. Of course, the missing kernels cannot be benchmarked for this particular library.

Following our running example, the blitz_interface class modelling the *Library_interface* concept looks like:

Table 1. Sets of types and functions that a user-defined class must provide to model the BTL++ a) *Library_Interface* concept and b) *Action* concepts

a)

public Types	
RT	Real Type (double or float)
GV	Generic Vector Type
GM	Generic Matrix Type
(static) functions	
std::string name(void)	
void vector_from_stl(GV &, const SV &)	
void vector_to_stl(const GV &,SV &)	
void matrix_from_stl(GM &, const SM &)	
void matrix_to_stl(const GM &, SM &)	
void axpy(RT, const GV &, GV &, int)	
+ dot, copy matrix_vector_product, ...	

b)

public Types	
Interface	*Library_Interface* model
methods	
Ctor(int size)	Ctor with problem size argument
void initialize(void)	
void calculate(void)	
void check_result(void)	
double nb_op_base(void)	
(static) functions	
std::string name(void)	

```
template<class real> struct blitz_interface{
  typedef real                 RT;
  typedef blitz::Vector<RT>    GV;
  static std::string name( void ){return "Blitz";}
  static void vector_from_stl(GV & B, const std::vector<RT> & B_stl){
    B.resize(B_stl.size()); // Note the () operator for Blitz vectors
    for (int i=0; i<B_stl.size() ; i++) B(i)=B_stl[i];
  }
  static void vector_to_stl(const GV & B, std::vector<RT> & B_stl){
    for (int i=0; i<B_stl.size() ; i++) B_stl[i]=B(i);
  }
  static void axpy(const RT coef, const GV & X, GV & Y, int N){
    Y+=coef*X; // Blitz++ Expression Template !
  }
  ...follows dot, matrix_vector_product,..
};
```

For each library in the BTL++ library set, the definition of the *Library_Interface* concept requires the creation of the corresponding *model* classes, such as ATLAS_interface or F77_interface, for example. To extend the BTL++ library interface collection to another library, one has to define the *Library_Interface* concept *model* class that implements all or a part of the BTL++ kernels. Because of the repetitive nature of the different interfaces, this work is greatly simplified via the inheritance mechanism in C++. We believe that the extendibility of the BTL++ library collection has been confirmed by the successful addition of various sequential [14] and parallel [15] libraries by different users. Now we can make use of this unified interface to implement each element of the BTL++ kernel set.

2.2 *Action* Concept

The BTL++ timing procedure begins with the problem size definition before initialising the reference (STL) and test (e.g. Blitz++) operands. Next, the

calculation duration is evaluated with a chosen (user definable) method. Once the calculations have been checked the performance results are stored. From this timing description on can see that all kernels across all libraries can be benchmarked in a similar way. The BTL++ *Action* concept allows the standardisation of these benchmarks by providing a uniform interface to deal with the different kernels. Table 1b) describes this concept. A class modelling the *Action* concept is implemented for each BTL++ kernel. The previously introduced *Library_Interface* concept allows us to implement only one *Action* template *model* class per BTL++ kernel. For example, the axpy operation is handled by the following action_axpy<> template class:

```
template<class LIB_INTERFACE> class action_axpy {
public :
  ...
  static  std::string name( void ){return "axpy";}
  double nb_op_base( void ){return 2.0*_size;}
  void calculate( void ) { LIB_INTERFACE::axpy(_coef,X,Y,_size);}
private :
  typename STL_INTERFACE::gene_vector  X_stl,Y_stl;
  typename LIB_INTERFACE::gene_vector  X,Y;
};
```

All the other BTL++ kernels are handled by their corresponding template *Action* class (action_dot, action_copy, ...). For example, one could execute the following code:

```
action_axpy< blitz_interface<double> > a(1000);
a.calculate();    // Compute axpy
a.check_result(); // Compare with STL reference result
```

3 The Benchmark Results

In this chapter we present some performance comparisons obtained with BTL++ and discuss various ways of using the collected information. Due to the extendible design of BTL++, the effort needed to specify which computational actions to compare is rather small, so that, once a new library has become available, a user can easily decide which kernels to test. The results of a benchmark run are currently stored in a file the name of which identifies the library, the action and the floating point type used in the computations. This rather rudimentary storage solution for the benchmarking results is most likely to be replaced by a lightweight data base in the future. In any case, once the performance data has been collected, it is straightforward to create graphical representations or tables of comparison as the examples in this article show.

The observed performance results for a matrix-matrix multiplication in Table 2 illustrate clearly the performance advantage of optimised BLAS libraries, such as ATLAS or GotoBLAS, over straightforward, but un-tuned implementations such as the FORTRAN77 example. The results for Blitz++ and uBLAS reiterate the point that alternative interfaces to the same computational action

can be accommodated. The gap between the tuned and the un-tuned implementations is independent of the programming language and the programming paradigm used. All un-tuned, or shall we say straightforward, implementations, be it in C, FORTRAN77 or C++ (procedural or object-oriented), suffer from the same performance problem as they do not take the memory hierarchy of the hardware into account.

Table 2. Performances of Matrix-Matrix product ($A \times B$) and $X = \alpha Y + \beta Z$ operations in MFlop/s. Small matrices are understood to have up to 333 rows, while large matrices have more than 333 rows. For vectors, the small/large size threshold is set to 10^5 elements. These definitions were found to be adequate on a 1.6 GHz PentiumM processor with 1MB L2 cache. The performance values are the algebraic mean over all measurement points in the respective category. The GNU Compiler Collection version 3.3.5 provided the C, C++ and Fortran compilers, the optimisation level was "-O3".

$A \times B$	small matrices	large matrices	$X = \alpha Y + \beta Z$	small vectors	large vectors
ATLAS 3-7-24	**1228**	**1409**	C	**797**	134
Goto baniasp-r1.15	1171	1333	STL	795	134
MKL 9.0	1111	1192	Blitz++ 0.9	691	135
C	758	208	blocked ET	673	**200**
Blitz++ 0.9	685	203	uBLAS (Boost 1.32)	635	135
STL	670	184			
f77	646	270			
uBLAS (Boost 1.32)	617	180			

For more information we refer to the web page of the project [9]. In particular, the web site provides detailed information on the interfaces to the different libraries and how the computational actions were implemented when the standard BLAS call was not available.

In addition to identifying which implementation works best in a given computing environment, `BTL++`, like any benchmark, can be used to compare different environments, such as different compilers or different machines. However, this aspect of building up a knowledge base over time is not integrated into the `BTL++` installation. As indicated earlier, we plan to change the data output and storage part of our benchmarking framework, which will make the building up of and the information retrieval from the performance data base much easier.

To illustrate the point that the benchmark kernels are not limited to BLAS operations, and to show that libraries do not always offer the best performance, we present results for the vector operation $X = \alpha Y + \beta Z$. The results in Table 2 and Fig.1a) show that for large vectors, the authors' blocked expression template performs better than all the other options by on average 48% [16].

The results in Fig. 1b) demonstrate that `BTL++` is not limited to mono-threaded computational kernels. The multithreaded computations were carried

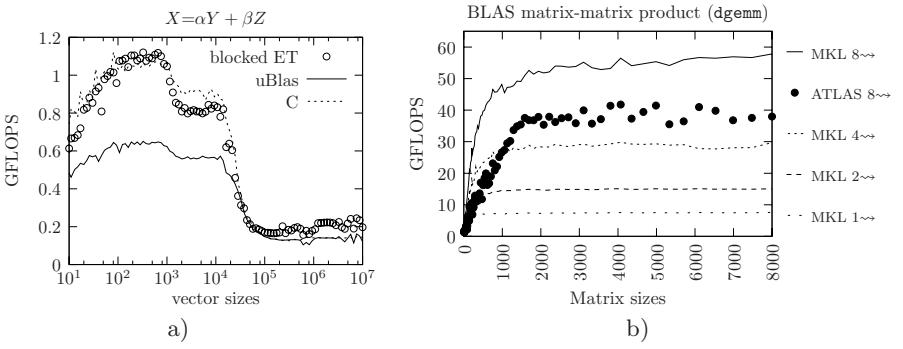

Fig. 1. a) Performance data in GFlop/s for the $X = \alpha Y + \beta Z$ operation. b) Multi-threaded performance results in GFlop/s for the dgemm routine on eight processor cores (2 sockets, 4 cores per socket).

out on a dual processor/quad core Intel Xeon with 2.83GHz with Intel MKL version 10 and ATLAS version 3.8.0.

4 Generation of the Optimal BLAS Library

In this chapter, we show how the so-called BTL++ hybrid library, which is the *optimal* BLAS interface library on a given machine, is constructed from previously collected performance data.

The principle of the BTL++ BLAS hybrid implementation is very simple. For each considered subroutine f of the BLAS API and for each library L that implements f, the BTL++ tool suite provides a set of performance measurements $\{\text{perf}(\texttt{f}, L, s)\}$ at different problem sizes s. From these results the BTL++ project selects two libraries L_1 and L_2 as well as a threshold size t such that the performance sum $S(L_1, L_2, t)$ is maximised:

$$S(L_1, L_2, t) = \sum_{s \leq t} \text{perf}(\texttt{f}, L_1, s) + \sum_{s > t} \text{perf}(\texttt{f}, L_2, s) \ . \tag{1}$$

The result of the optimisation step is an automatically generated C file named Hybrid.c which contains the switches for all BLAS subroutine implementations. For the PentiumM target, the generated call for the daxpy operation in Hybrid.c reads as follows:

```
void cblas_daxpy(const int N, const double alpha,
                 const double *X, const int incX,
                 double *Y, const int incY){
  if (N<18738){
    MiniSSE1_cblas_daxpy(N,alpha,X,incX,Y,incY);
  }
  else{
```

```
    ATLAS_cblas_daxpy(N,alpha,X,incX,Y,incY);
  }
}
```

Again for the PentiumM at 1.6GHz, the result of the optimisation process, i.e. the choice of libraries and the threshold values, for different BLAS operations is given in Table 3.

Table 3. `BTL++` interface automatically generated for the PentiumM (1.6GHz) target

BLAS routine	Best Library for small problem sizes	Small/large threshold	Best Library for large problem sizes
`daxpy`	MiniSSE	18738	ATLAS
`dcopy`	Netlib	86974	ATLAS
`ddot`	MiniSSE	18738	ATLAS
`dgemm`	ATLAS		ATLAS
`dgemv`	MKL		MKL

Note that a switch between two libraries is only generated when needed. In other words, when one particular implementation offers the best performance across the board, no switch statement is generated and hence, no run time overhead occurs.

4.1 BLAS Implementation Wrappers

The main difficulty in building the hybrid BLAS is to cope with the differences in the BLAS installations. For example, MKL is distributed as a dynamic library (`libmkl.so`) implementing the `C` BLAS interface (`c_blas_xxx`), while ATLAS automatically builds a static library (`libatlas.a`) implementing the same interface. Last example, the `Goto` library is built dynamically (`libgoto.so`) for numerous computer targets but only implements the BLAS F77 interface that has to be used through the static netlib CBLAS wrapper (`libcblas.a`).

The main idea in the `BTL++` approach is to wrap each library, whether static or not, that contributes to the optimal BLAS implementation into a dynamic library and to use the `dlopen()/dlsym()` functions to load these dynamic wrappers when needed.[1]

To illustrate that the generated library interface does indeed offer the best performance possible, we show the results for the daxpy and dcopy operations in Fig. 2.

[1] In order to avoid infinite recursions at link time, the proposed solution relies on the `-Bsymbolic` option of the gnu `ld` linker, which binds the static library symbols to the intermediate dynamic wrappers. The portability of this solution is an open issue.

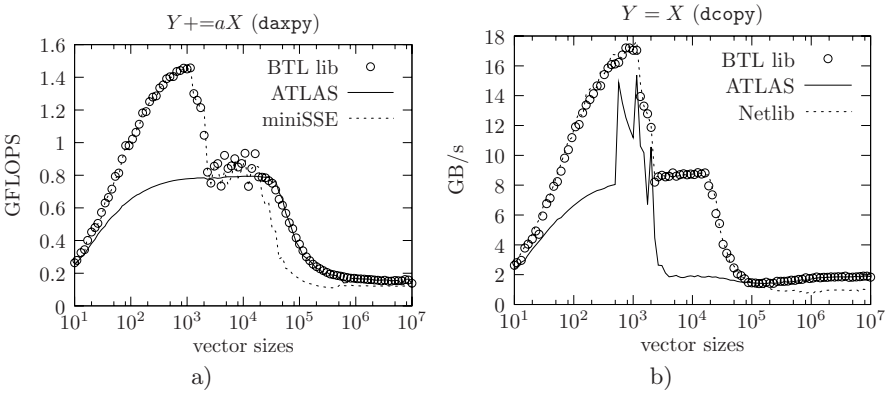

Fig. 2. Performance data for the `daxpy` (a) and `dcopy` (b) operations using the generated, optimal BLAS interface (BTL lib). The threshold values and the best implementations available are determined automatically.

5 Conclusion and Outlook

In this article, we have described the benchmarking project `BTL++`, which helps a user to assess and compare the performance of user definable, computational actions in her given computing environment of hard- and software. The paramount design goal of the `BTL++` project was flexibility, which allows a user to adapt or extend the three parts (libraries, computational kernels and performance analysers) of the framework to her needs with little effort. The only requirement the approach currently imposes is that the user defined parts can be called from a C++ main program.

Several extensions, ranging from interfacing new libraries over adding new computational kernels to adapting the approach to parallel computing, have been carried out successfully by different users. The fact that these extension efforts were possible without interaction with the `BTL++` developers indicates that the design goal of user extendibility has been achieved.

We have shown how the results of the benchmarking exercise of the BLAS operations can be used to create a new library automatically that for each function and in each problem size range calls the best implementation available. Numerical results confirm that the generated interface library does indeed obtain the same performance as the best implementation identified in the benchmarking comparison.

Concerning the evolution of the project, we have identified several areas of possible improvement. In the area of parallel computing, we intend to build on the work described in [15] to enlarge the scope of our benchmarking tool to distributed and shared memory programs.

The current implementation of the generation of the optimal interface library has been developed for Linux/GNU x86-platforms. The question concerning the portability of this implementation remains open.

Acknowledgements. The authors are grateful to their colleagues Christian Caremoli, Ivan Dutka-Malen, Eric Fayolle and Antoine Yessayan for their valuable help and expertise.

References

1. Netlib: BLAS web page, `http://www.netlib.org/blas`
2. Whaley, R.C., Petitet, A.: Minimizing development and maintenance costs in supporting persistently optimized BLAS. Software: Practice and Experience 35(2), 101–121 (2005)
3. Intel: MKL web page can be found from, `http://www.intel.com`
4. AMD: ACML web page, `http://developer.amd.com/acml.jsp`
5. Goto, K., van de Geijn, R.A.: Anatomy of a High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software 34(3) (September 2007)
6. Veldhuizen, T.L.: Arrays in blitz++. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) ISCOPE 1998. LNCS, vol. 1505, pp. 223–230. Springer, Heidelberg (1998)
7. Siek, J.G., Lumsdaine, A.: The matrix template library: Generic components for high-performance scientific computing. Computing in Science and Engineering 1(6), 70–78 (1999)
8. Walter, J., Koch, M.: uBLAS web page, `http://www.boost.org/libs/numeric/ublas`
9. Plagne, L.: BTL web page, `http://projects.opencascade.org/btl`
10. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. The International Journal of High Performance Computing Applications 14(3), 189–204 (2000)
11. Schöne, R., Juckeland, G., Nagel, W.E., Pflüger, S., Wloch, R.: Performance comparison and optimization: Case studies using BenchIT. In: Joubert, G.R., Nagel, W.E., Peters, F.J., Plata, O.G., Tirado, P., Zapata, E.L. (eds.) Parallel Computing: Current & Future Issues of High-End Computing. Proceedings of the International Conference ParCo 2005, vol. 33, pp. 877–884. Central Institute for Applied Mathematics, Jülich, Germany (2006)
12. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005)
13. Berghen, F.V.: miniSSEL1BLAS web page, `http://www.applied-mathematics.net`
14. Petzold, O.: tvmet web page, `http://tvmet.sourceforge.net`
15. Mello, U., Khabibrakhmanov, I.: On the reusability and numeric efficiency of C++ packages in scientific computing (2003), `http://citeseer.ist.psu.edu/634047.html`
16. Plagne, L., Hülsemann, F.: Improving large vector operations with C++ expression template and ATLAS. In: 6th intl. workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2007) (July 2007)