Multi-Target Vectorization With MTPS C++ Generic Library

Wilfried Kirschenmann* [1,3], Laurent Plagne† [1], and Stéphane Vialle‡ [2,3]

[1] *SINETICS Department, EDF R&D, FRANCE*
[2] *IMS Research Group, SUPELEC, FRANCE* §
[3] *AlGorille INRIA Project Team, FRANCE* §

Abstract This article introduces MTPS, a C++ template library dedicated at vectorizing algorithms for different target architectures. Algorithms written with MTPS benefit from optimized memory access patterns and show performances close to hardware limits, both on multicore CPU and on GPU.

Keywords GPU, SSE, Vectorization, c++ Template Metaprogramming, Performances

1 Introduction

In many scientific applications, computation time is a strong constraint. Optimizing these applications for the rapidly changing computer hardware is a very expensive and time consuming task. Emerging hybrid architectures tend to make this process even more complex.

The classical way to ease this optimization process is to build applications on top of High Performance Computing (HPC) libraries that are available on a large variety of hardware architectures. Such scientific applications, whose computing time is mostly consumed within such HPC library subroutines, then automatically exhibit optimal performances for various hardware architectures.

However, most classical HPC libraries implement fixed APIs (e.g., BLAS) and may be too rigid to match the needs of all client applications. In particular, classical APIs are limited to manipulate rather simple data structures like dense linear algebra matrices. As a more complex issue, general sparse matrices cannot be represented with a unified data structure and various formats are proposed by more specialized libraries. In the extreme case, structured sparse matrices cannot be efficiently captured by any of the classical library data structures. Relying on such complex matrices, several neutron transport codes developed at EDF R&D require another kind of library to be used.

Following the model of the C++ Standard Template Library (STL), template based *generic libraries* such as Blitz++ [11] provide more flexible APIs and extend the scope of library-based design for scientific applications. Such generic libraries allow to define Domain Specific Embedded Languages (DSELs) [2].

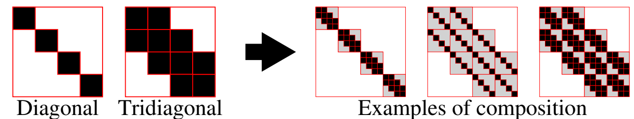

Figure 1: Matrix pattern composition within Legolas++.

Legolas++, a basis for several HPC codes at EDF, is a C++ DSEL dedicated to structured sparse linear algebra. In order to meet EDF's industrial quality standards, a *multi-target* version of Legolas++, currently under development, will provide a unified interface for the different target architectures available at EDF, including clusters of heterogeneous nodes (i.e., with both multi-core CPUs and GPUs). This article presents MTPS (Multi-Target Parallel Skeletons), a C++ generic library dedicated to *multi-target* vectorization that is used to write the *multi-target* version of Legolas++. Only developments concerning a single heterogeneous node are presented here.

The next section presents the principles of Legolas++ and **Section 3** introduces MTPS. Its optimization strategies and the achieved performances are discussed in **Section 4**. Finally, conclusions are drawn in **Section 5**.

2 Towards a Multi-Target Linear Algebra Library

Legolas++ is a C++ DSEL developed at EDF R&D to build structured sparse linear algebra solvers. Legolas++ provides building bricks to describe structured sparse matrix patterns and the associated vectors and algorithms.

Legolas++ is based on the observation that most structured sparse matrix patterns can be described as the composition of simpler patterns. **Figure 1** shows how to compose two simple patterns to create new patterns. **Figure 2** shows an example of matrix pattern issued from a neutron transport code written with Legolas++ (*GLASS* in [8]).

The explicit GPU parallelization of one of our neutron transport code resulted in speed-ups around 30 over the sequential Legolas++ CPU implementation [5]. To generalize this gain of performances to other Legolas++ based applications, a parallel and multi-target version of Legolas++ is being developed. As the parallel CPU and GPU versions exhibit strong similarities, Legolas++ developments

*Email: `wilfried.kirschenmann@edf.fr`
†Email: `laurent.plagne@edf.fr`
‡Email: `stephane.vialle@supelec.fr`
§Authors want to thank Region Lorraine and ANRT for supporting this research.

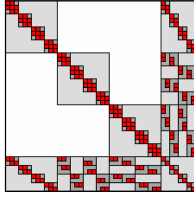

Figure 2: A matrix pattern built with Legolas++.

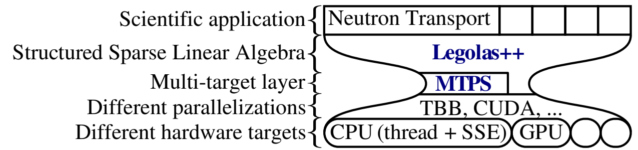

Figure 3: Our hourglass software architecture to achieve a multi-target Legolas++: a minimal MTPS library adapts the code for different hardware architectures.

for the different targets are factorized into an intermediate layer between Legolas++ and the different hardware architectures, namely MTPS (see **Figure 3**).

3 Introduction to MTPS

3.1 Related Work

Many libraries parallelize for different architectures from a single source code. A complete bibliography is beyond the scope of this abstract; only some examples based on C++ meta-programming techniques are discussed.

Some libraries, like TrilinosNode [1], Quaff [3] or Intel TBB [9], require their users to explicitely express the parallelism within the application by using *parallel skeletons*. This expression of available parallelism can be encapsulated into specialized and implicitely parallel STL-like *containers* and *algorithms*, as in Thrust[1] and Honei [10].

Our goal is to provide implicit parallelism within Legolas++ *containers* and *algorithms*. To ease the writing of its *containers* and *algorithms*, Legolas++ relies on MTPS which follows a *parallel skeletons* based approach. Then MTPS optimizes the code for the different architectures.

As this article presents MTPS, only code for MTPS is shown. For Legolas++ users, MTPS details are hidden in its *containers* and *algorithms*.

3.2 *Collections* and *Vectorizable Algorithms*

This section introduces the notions of *collection* and *vectorizable algorithm* on which MTPS relies.

In C++, a *Plain Old Data* (POD) is a type whose memory representation pattern can be changed without altering its value [4]. POD members can be either integral types or PODs. In the following code snippet, `MyPOD` is a POD with three `float` data members:

```
1 struct MyPOD{ float a,b,c; };
```

Let a *collection* be a data structure containing different instances of the same POD and `f` be a *pure* function (i.e., `f` has no side effects). An algorithm applying `f` to all elements of a *collection* is said to be *vectorizable*. To parallelize such algorithms, MTPS provides two parallel skeletons optimized for different target architectures: *map* and *fold* which correspond to a *parallel for loop* and to a *parallel reduction* respectively.

An algorithm is *vectorizable* in reference to a given *collection*. For instance, an algorithm operating on a matrix can be either *vectorizable* or not depending on whether the matrix is considered as a *collection* of columns or as a *collection* of rows. Two algorithms *vectorizable* in reference to the same *collection* are said to be in the same *vectorial context*. On the contrary, if two consecutive algorithms are not *vectorizable* in reference to the same *collection*, a *context switch* is required. In a distributed memory system, *context switches* correspond to communications.

3.3 Linear Algebra *Hello World* of MTPS: `saxpy`

This section presents the MTPS implementation for the `saxpy` operation whose implementation in C is:

```
1 float *X, *Y, a;
2 for(int i=0; i<N; ++i) Y[i]+=a*X[i];
```

First, the iteration-dependent data are gathered in a POD `XYData` whose members correspond to `X[i]` and `Y[i]`. The types of the two members (`float`) are passed as template arguments to `MTPS::POD` and their names (`x` and `y`) are given in the `Fields` enum:

```
1 struct XYData
2   :public MTPS::POD<float,float>{
3   enum Fields{x, y};
4 };
```

Second, a *collection* of `XYData` elements, `xyCol`, can be built using MTPS containers. A class type for optimized container is provided as member of the class corresponding to the target architecture. Two levels of parallelism are available on CPUs: thread parallelism and SIMD parallelism. The choice for each level is made by passing two arguments to the `CPU` template class. `Thread` can be one of `MTPS::Sequential`, `MTPS::OMP` (openMP) or `MTPS::TBB` (Intel TBB). `SIMD` can be one of `MTPS::Scalar` or `MTPS::SSE`. On CUDA-enabled GPUs, only the SIMD parallelism is used.

```
1 //typedef MTPS::GPU::CUDA Target;
2 typedef MTPS::CPU<Thread, SIMD> Target;
3
4 Target::collection<XYData> xyCol(N);
```

Third, the function that is to be applied to all elements of the collection must be written as a functor class `AxpyOp` which contains the value `a` internally:

```
1 struct AxpyOp{
2   float a_;
3   template <template <class> class View>
4     INLINE void operator()(View<XYData> xy)
5   const {
```

[1]Thrust: `http://code.google.com/p/thrust/`

```
6   typedef View<XYData> XYV;
7   int x = XYV::x;
8   xy(XYV::y)+=a_*xy(x);
9  }
10 };
```

As `XYData` elements may not be stored identically on different target architecture, `AxpyOp::operator()` does not take an `XYData` as argument. A `View` is provided instead. `XYData` members can be accessed with the `operator()` of the `View` which takes an `int` as argument. Elements of the `Fields` enum can be used either to initialize an `int` (line 7) or directly (line 8). The declaration of `AxpyOp::operator()` must be preceeded by the `INLINE` macro which defines target-dependent keywords (e.g. `__device__` for CUDA).

Finally, the functor can be passed to the `map` and `fold` parallel skeletons provided by the *collection* container:

```
1 AxpyOp axpyOp; axpyOp.a_=...;
2 xyCol.map(axpyOp);
3 ...
4 DotOp dotOp;
5 float dot = xyCol.fold(dotOp);
```

4 Optimization of performances

For each architecture, the specific optimizations required to enable good performances will be presented. The implementation of an example will then be discussed.

4.1 Multi-Target Performance Optimizations

Parallelizing a *vectorizable algorithm* is straightforward. However, achieving good performances is not: modifications of the *collection* storage pattern may be required. Indeed, achieving efficient usage of memory bandwidth on a given hardware architecture requires specific access patterns [6]. **Figure 4** shows how a matrix of 8 TriDiagonal Symmetric (TDS) blocks of size 4 is stored on three different architectures to optimize the memory access pattern.

Performances achieved thanks to this optimization will be shown in **Section 4.3**. As this optimization is made in MTPS *collection* container, MTPS user must define both the size per POD-element of each field (4 for the diagonal field on **Figure 4**) and the number of POD-element in order to construct a *collection*. Using these two information, MTPS optimizes the storage for each target architecture.

A *context* switch imply a data reordering. For instance, switching a *collection* of matrix rows to a *collection* of matrix columns modifies the effective storage (i.e. the matrix is transposed). This part of MTPS is under development.

4.2 Implementation of a Linear System Resolution

The example presented in this section corresponds to a basic operation that represents the major part of the execution time of a neutron transport code [5]. Let A be a block-diagonal matrix with TDS blocks. smaller problems. The $AX = B$ linear system can be seen as a *collection* of smaller

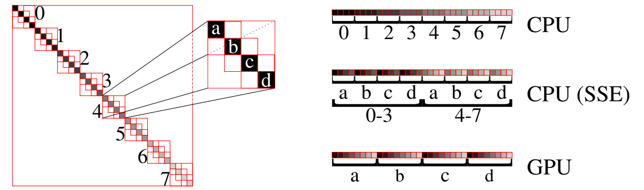

Figure 4: The storage of the diagonal is adapted by MTPSfor the target architecture.

block systems $ax = b$ that can be solved independently. To solve one $ax = b$ system, the matrix a is factorized in-place with a LDL^T factorization and a forward and backward substitution is then applied on x. Only the code for the factorization is shown here.

Let us introduce `TData` which represents a TDS block. `TData` elements are stored in two vectors corresponding to the diagonal and the lower diagonal:

```
1 struct TData
2   :public MTPS::POD<float,float>{
3  enum Fields{diag, low};
4  typedef MTPS::POD<float,float> Base;
5  typedef typename Base::Shape Shape;
6
7  static Shape createShape(int size){
8   Shape out;
9   out[diag] = size;
10  out[low] = size-1;
11 }
12 };
```

The `Shape` type of line 5 contains the effective sizes of the two fields. All elements of a *collection* have the same shape. As both the number of `TData` elements and their shape is known, `tCol` storage pattern can be optimaly stored according to the target architecture (see **Figure 4**):

```
1 TData::Shape s=TData::createShape(size);
2 Target::collection<TData> tCol(N, s);
```

The following code snippet corresponds to `TLDLtOp` which factorize the matrix a in-place using a LDL^T decomposition:

```
1 struct TLDLtOp{
2  template <template <class> class View>
3   INLINE void operator()(View<TData> a)
4  const{
5   typedef View<TData> TV;
6   int low = TV::low, diag = TV::diag;
7   typename TV::template Type<low>::Type l;
8   int size = a.shape()[diag];
9   for (int i = 1 ; i < size ; i++){
10   l=a(low, i-1)/a(diag, i-1);
11   a(low, i-1)=l;
12   a(diag, i)-=a(diag, i-1)*l*l;
13  }
14 }
15 };
16 TLDLtOp op;
17 tCol.map(op);
```

Thread	SIMD	Time (ms)	Speed up	GFlops GB/s	% peak
sequential	scalar	184.1	1.0	0.5 1.0	0.8 4.2
	SSE	48.8	3.8	1.7 3.8	3.1 15.9
intel TBB	scalar	23.3	7.9	3.6 8.0	6.5 33.3
	SSE	8.5	21.6	9.9 21.9	17.9 **91.3**
openMP	scalar	23.1	8.0	3.6 8.0	6.6 33.6
	SSE	8.1	22.7	10.3 23.0	18.8 **95.8**
CUDA C		4.2	43.9	20.0 71.0	5.7 **95.9**

Table 1: Performances of MTPS for the TDS example. Computation are carried out in single precision floating point. The smallest time over 1000 executions is given.

The elements of a field can be accessed by passing their index as the second argument of the view `operator()`. If this index is not provided, its default value is 0. Line 7 shows how the type of a field elements can be retrieved.

4.3 Performances

Table 1 shows the performances obtained to solve the $AX = B$ system with A having 10^5 blocks of size 100. Speed-ups are given compared to the sequential scalar CPU version. CPU tests are run on a machine with two 2 GHz Intel E5504 quad-core processors. GPU tests are run on a Nvidia Quadro FX5800 card. Both architectures were launched at the end of 2008. Computation performances are given in GFlops. Data throughput is given in GB/s and takes into account the data transfers to and from the memory: on CPU, an element remaining in cache between two loads is considered to have been loaded only once. The achieved performances are compared to the measured peak performances. Peak computational power is measured with large BLAS matrix-matrix multiplications (`sgemm`): 55 GFlops on CPU and 348 GFlops on GPU. Peak memory throughputs are measured with the stream benchmark [7] on CPU (24 GB/s) and with the CUDA SDK bandwidth benchmark on GPU (74 GB/s).

On both CPU and GPU, parallel performances are limited by the memory bandwidth and our approach reaches more than 95% of the peak memory bandwidth utilization.

5 Conclusions and perspectives

We have presented MTPS, a C++ generic library simplifying the parallelization and the optimization of *vectorizable algorithms* for different architectures. In particular, one can program an algorithm once, compile it for execution on the SSE units of multicore CPUs or on CUDA-enabled GPUs and obtain performances close to hardware limits: 95% of peak performances were achieved.

For further developments of MTPS, two main directions are considered. On the one hand, having an efficient implementation of *context switches* between different *vectorizable algorithms* is essential to write complex applications. On the other hand, targeting other architectures, including distributed memory architectures, will add more opportunities to increase the performances. These two directions of research will be investigated in a near future.

References

[1] C. G. Baker, H. Carter Edwards, M. A. Heroux, and A. B. Williams. A light-weight api for portable multicore programming. In *PDP 2010: Proceedings of The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Washington, DC, USA, 2010. IEEE Computer Society.

[2] K. Czarnecki, J. T. Odonnell, J. Striegnitz, Walid, and Taha. Dsl implementation in metaocaml, template haskell, and c++. *LNCS: Domain-Specific Program Generation*, 3016(2):51–72, 2004.

[3] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[4] ISO. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003. (§3.9).

[5] W. Kirschenmann, L. Plagne, S. Ploix, A. Ponçot, and S. Vialle. Massively parallel solving of 3D simplified P_N equations on graphic processing units. In *Proceedings of Mathematics, Computational Methods & Reactor Physics*, May 2009.

[6] W. Kirschenmann, L. Plagne, and S. Vialle. Multi-target c++ implementation of parallel skeletons. In *POOSC '09: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 1–10, New York, NY, USA, 2009. ACM.

[7] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, dec 1995.

[8] L. Plagne and A. Ponçot. Generic programming for deterministic neutron transport codes. In *Proceedings of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications*, Palais des Papes, Avignon, France, September 2005.

[9] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[10] D. van Dyk, M. Geveler, S. Mallach, D. Ribbrock, D. Göddeke, and C. Gutwenger. Honei: A collection of libraries for numerical computations targeting multiple processor architectures. *Computer Physics Communications*, 180(12):2534 – 2543, 2009.

[11] T. L. Veldhuizen. Arrays in blitz++. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 223–230, London, UK, 1998. Springer-Verlag.