Meta-programming and Multi-stage Programming for GPGPUs

Ian Masliah, Marc Baboulin, Joel Falcou

Meta-programming and Multi-stage Programming for GPGPUs

Ian Masliah
University Paris Sud
F-91405 Orsay, France
Email: ian.masliah@lri.fr

Marc Baboulin
University Paris Sud
F-91405 Orsay, France
marc.baboulin@lri.fr

Joel Falcou
University Paris Sud
F-91405 Orsay, France
joel.falcou@lri.fr

***Abstract*—GPGPUs and other accelerators are becoming a mainstream asset for high-performance computing. Raising the programmability of such hardware is essential to enable users to discover, master and subsequently use accelerators in day-to-day simulations. Furthermore, tools for high-level programming of parallel architectures are becoming a great way to simplify the exploitation of such systems. For this reason, we have extended NT^2 – the Numerical Template Toolbox - a C++ scientific computing library which can generate code for SIMD and multi-threading systems in a transparent way. In this paper, we study how to introduce an accelerator-based programming model into this library to allow developers to reap the benefits of such an architecture from a simple, MATLAB like code. After a brief description of the NT^2 framework, we explain how our accelerator programming model has been designed and integrated in a pure C++ library. We conclude by showing the applicability and performance of this tool on some practical applications.**

I. INTRODUCTION

Developing large applications in a simple, fast and efficient way has always been an issue for software developers. As computing hardware complexity rose with the advent of SIMD, multi-processor, multi-core systems and more recently accelerators like GPUs or Intel Xeon Phi , software design methodologies did not undergo the same amount of changes. This complicates the exploitation of such hardware in mainstream applications.

Designing **Domain Specific Languages** (or *DSL*) has been presented as a solution to these issues: *DSL*s allow solutions to be expressed in their programming idiom with a level of abstraction equivalent to the problem domain leading to improvements in maintainability and quality of code. One of the most popular examples is MATLAB™ which provides a large selection of toolboxes that allow a direct expression of high-level algebraic and numerical constructs in a easy-to-use imperative language. In this scope, **Domain Specific Embedded Languages** (or *DSEL*s) [1], [2] are languages implemented inside a general-purpose host language [3]. This removes the requirement of a dedicated compiler or interpreter as they are designed as a library-like component [4], [5].

NT^2 – The Numerical Template Toolbox – is such a *DSEL* using C++ template meta-programming [6] to provide a MATLAB -inspired API while supporting a large selection of parallel architectures and keeping a high level of expressiveness [7]. NT^2 was designed to support architectural features like SIMD extensions and multi-core programming [8]. However, the support for accelerators like GPGPUs was limited as GPU kernel compilers where unable to process NT^2 C++ 11 based implementation of C++ based *DSEL* . Furthermore, libraries such as Thrust [9] for CUDA can only be used in .cu files and compiled with nvcc which furthers limits the possibilities for code generation.

In this paper, we present a new extension for NT^2 that takes care of such accelerators, especially CUDA based GPUs through multi-stage programming [10] (or *MSP*) for linear algebra and element-wise problems. *MSP* consists in doing multiple compilation phases allowing for type-safe program generation. Our contributions include:

- A programming model supporting both implicit or explicit data transfers to and from GPGPUs with a simple user interface
- An adaptable strategy to generate CUDA kernel directly from a single C++ source file containing NT^2 statements
- The integration of this kernel generator with existing CUDA libraries like cuBLAS or MAGMA.

The purpose of this system is to provide the user some leeway on how to distribute the data between the host and device through a simple mechanism. As having a perfect cost model for load balancing is very complex to put in place and costly, letting the user provide some insight on data locality is beneficial.

After reviewing the concurrent state of the art software libraries (section II), we introduce NT^2 , its programming model (section III) and how we have adapted it to support GPU computation. We then describe the kernel generator process and how it integrates with the existing library (section IV). Finally, we present benchmarks assessing the generated code quality (section V) and conclude on the future work regarding NT^2 and accelerators.

II. RELATED WORKS

Software libraries for GPGPU computing try to simplify the new programming paradigm brought by many-core based systems. The general trend followed in recent years by

C++ libraries is to provide a high-level interface through template meta-programming techniques. This goal is reached by providing device containers with architecture-aware generic parallel algorithms and/or a code generation based process. This approach differs from what can be seen with OpenACC [11] or Halide [12] as it is a *DSEL* based approach and does not rely on language extensions or pragmas. We will give a detailed explanation of both type of libraries that support OpenCL/CUDA or both.

Thrust [9] is a header only library providing a similar interface to the C++ Standard Template Library. Its high-level interface is based on meta-programming and traits to provide efficient parallel skeletons that can run on either CPU or GPU. Container locality is expressed through an explicit container declaration limiting the abstraction but allowing for easier transfers between host and device. Locality for functions is defined by default for device vectors and can be extended with tag dispatching. Overall, it is a well rounded utility library which can be combined with CUDA Toolkit libraries such as CUBLAS, CUFFT and NPP. However, it can only be used in .cu files and compiled with nvcc which limits its usefulness.

VexCL [13] is an expression template library for OpenCL/CUDA. It provides a high-level generic interface that is suitable for both back-ends with static parameters defined within a *DSEL* for linear algebra. The expression template mechanism allows for code generation by lazy evaluation of vectors and elementary operations within the AST. Similarly to Thrust, it provides STL-like functions on containers that have a defined locality. It is also possible for the user to define custom functions on device that will be dynamically generated for CUDA. However, the transform used for the generation process requires a unique data locality limiting hybrid algorithms.

ViennaCL [14] is also an expression template library for OpenCL/CUDA. This library strictly follows the uBLAS programming interface and STL like algorithms for easier integration with other softwares. It has implementations for BLAS kernels and high-level solvers for sparse and dense computation that provide good performance. Through the mechanism of expression templates it can evaluate basic linear algebra operations with operator overloading. ViennaCL focuses more on OpenCL due to the necessity for separate compilation with CUDA limiting its support through the OpenCL language. It is however possible to generate CUDA code with ViennaCL at runtime.

Boost.Compute [15] is a header only C++ library based on the OpenCL standard. Similar to other libraries, it manages device memory through a designated container. It provides an interesting concept of future for asynchronous copy on the device allowing for more versatility. Boost.Compute also supports closures and adaptable structures for device. Similarly to thrust, it is a well rounded library based on OpenCL to simplify the coding process on accelerators. It however lacks support for numerical analysis and cannot generate CUDA code.

Eigen [16] is a popular library to solve linear systems using expression templates. It should be able to support accelerator code with CUDA by writing Eigen code in a .cu file. It however does not provide any high-level interface or special containers for GPU computing. Eigen does not support OpenCL.

Feature	Thrust	VexCL	ViennaCL	Boost.C	NT^2
MATLAB API	–	–	–	–	✓
AST optimization	–	✓	✓	✓	✓
Device Arrays	✓	✓	✓	✓	✓
Cuda code gen	✓	✓	–	–	✓
OpenCL code gen	–	✓	✓	✓	✓
parallel skeletons	✓	✓	–	✓	✓
CUBLAS support	✓	–	✓	–	✓
Static code gen	–	–	–	–	✓
dense LA solvers	–	–	✓	–	✓
sparse LA solvers	–	–	✓	–	–

Fig. 1. Feature set comparison between NT^2 and similar libraries

In Figure 1, we compare features between NT^2 and the previously described libraries. We did not include Eigen as it does not have any real support for code generation or a device API.

The purpose of the previously described libraries is usually to provide a wrapper over the C++ language for GPGPU computing. For this reason, the *DSEL*s based on expression templates or `Boost.Proto` are usually lightweight and consist mostly of overloading elementary operations for containers. The code generation phase then ends up doing a dynamic compilation of the CUDA code/OpenCL kernel which adds a significant overhead on small sized or low arithmetic intensity problems.

Furthermore, as mentioned in Section I, it is not possible to compile an NT^2 source code with nvcc even with the latest version (7.5). To address these issues, we have added a two step compilation in NT^2 using a serialization mechanism based on Boost.Serialization. Before describing this process, we will first detail the execution model for GPGPU computing in NT^2.

III. NT^2 Execution Model

NT^2 [7] is a numerical computing C++ library implementing a subset of the MATLAB language as a *DSEL* . NT^2 simplifies the development of data-parallel applications on a large selection of architectures currently including multi-core systems[8] with SIMD extensions[17]. Simply put, a MATLAB program can be converted to NT^2 by copying the original code into a C++ file and performing minor cosmetic changes (defining variables, calling functions in place of certain operators). NT^2 also takes great care to provide numerical precision as close to MATLAB as possible, ensuring that results between a MATLAB and an NT^2 code

are sensibly equal.

Internally, NT^2 is designed to leverage the well known *Expression Templates* C++ idiom to build at compile time a flexible representation of the abstract syntax tree of any C++ expression containing at least one NT^2 component. This compile-time tree is then transformed in actual code to be executed on a parallel system. Contrary to other libraries based on the same technique, NT^2 relies on `Boost.Proto` , an external *Expression Templates* system to handle the creation and transformation of ASTs [18]. `Boost.Proto` allows us to replace the direct walk-through of the compile-time AST done in most C++ *DSEL*s by the execution of a mixed compile-time/runtime algorithm over the predefined AST structure generated by `Boost.Proto` (fig. 2).

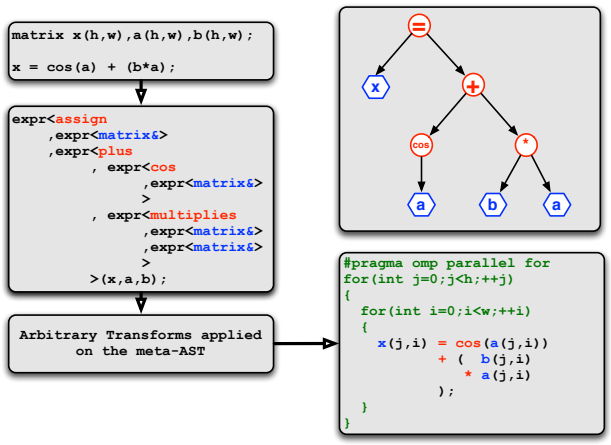

Fig. 2. *Expression Templates* in NT^2

Finally, the other main difference between NT^2 and similar tools is the fact that architectural specificities of the generated code are handled by an execution model based on **Algorithmic Skeletons**[19]. Those skeletons simplify the extension of NT^2 for various hardware by separating the concerns of optimizing the AST code for different type of architecture.

A. Support for CPU/GPU execution

If the single system computation model of NT^2 is rather classic, we needed a way to handle accelerators in a generic and extensible way. The first step is to provide a simple way to **locate** tables on either the host system or on a device. This is simply done by using a couple of settings in the definition of NT^2 table instances. Listing 1 show cases the `nt2::host_` and `device_,pinned_` settings that specifies if a `table` contains data stored on the host memory or device memory. Note that a special function `on_device` can be used to specify on which device the memory must be allocated in the case where multiple devices are available.

Semantic of operations between host and device tables is quite straightforward as they will be carried on the proper memory segment of each table.

```
// Generates a host table by default
table<double> A( of_size(1e3,1e3) );

// Generates a host table with cuda host pinned memory
table<double,pinned_> A2( of_size(1e3,1e3) );

// Generates a device table
table<double,device_> D( of_size(1e3,1e3) );

// Generates a device table on device #2
table<double,device_> D2( of_size(1e3,1e3), on_device(2) );
```

Listing 1. NT^2 host and device specifications

When mixing tables of different location, memory transfers are implicitly performed. This means that assigning a host/pinned table to a device table is equivalent to performing a CUDA memory transfer. This can be used for example to simplify interaction with existing GPU kernels as shown in listing 2. As streams are no assigned to tables, this transfer will be synchronous.

A copy function is also available to perform asynchronous memory transfers when a non-default stream is given.

```
// X is a 1e3 x 1e3 matrix full of 1.
table<double> X = ones(1e3,1e3);

// Transfer to device
table<double,device_> Y = X;

// cuBLAS direct call
cublasDscal( Y.size(), 5. , Y.data(), 1.);

// Transfer back to host
X = Y;
```

Listing 2. NT^2 interaction with cuBLAS

This semantic of transfer by assignment is a classical way of performing such operation transparently. It as been used by tools like Thrust or VexCL and have been proved to be easy enough for the user while allowing for fine grain performance tuning.

B. MAGMA Integration

Our extension to NT^2 also provides a direct high-level integration of MAGMA [20] for handling most linear algebra tasks as shown on listing 3 for the `linsolve`.

```
table<float> A, B, X;
table<float,device_> dA, dB, dX;

X  = linsolve(A,B);
dX = linsolve(A,dB);
```

Listing 3. NT^2 interface to MAGMA kernels

Calls to `linsolve` can mix device and host tables. The implementation of such kernels will take care of transferring the strictly required data over the device and to perform transfer back to the host only if the result is assigned to a host table. Most MAGMA kernels are mapped onto their MATLAB equivalent API and handle the same sets of optional behaviors.

C. Kernel optimizations

As complex programs often feature multiple statements involving NT^2 tables, it may become hard in some cases to maintain a high level of performance as the cost of transfers between statements may become significant. To locally solve this issue, NT^2 provides a function called `tie` that allows for library-based loop fusion as depicted in Listing 4.

```
table<T> bs( table<T> const& B, table<T> const& C,
           , table<T> const& Y, table<T> const& Z
           )
{
  table<T> A( B.extent() ), X( Y.extent());

  tie(A,X) = tie( B + C * sin(B)
                , Y / ( Z + A )
                );

  return X;
}
```

Listing 4. NT^2 merged kernels

In this scenario, the single assignment statement between call to `tie` will generate exactly one call to a single CUDA kernel composed of the fusion of the two loop nests over the variable `A` and `X`. Every table will be transferred in a single sequence of streaming operations and a single kernel will be executed on the device.

IV. Device Code Generation

As described in section II, code generation for accelerators in C++ is based on Meta-programming techniques. This process is however limited by the fact that C++ does not natively support language extensions for runtime code generation and program execution [21]. Creating a multi-stage paradigm is therefore necessary to enable the compilation of CUDA code within a C++ framework.

A. Multi-stage programming in C++

MSP is usually applied by adding specific language extensions to trigger a new compilation phase. As an example, MetaOcaml [22] is a multi-stage language based on Ocaml with three basic constructs for runtime code optimization.

More recent adoptions of *MSP* for parallel computing include techniques like Lightweight modular staging (LMS) [23] in Scala, Language Virtualization [24] (base on LMS) or Terra [25] using language extensions for HPC in Lua . LMS is similar to Meta-programming techniques in C++ applied to the *Domain Engineering Method for Reusable Algorithmic Libraries* [26] methodology on which NT^2 is build. Language Virtualization is an interesting concept but it is limited by the need to make the *DSEL* identical to the stand-alone language. This greatly increases the complexity of the generation process and makes it hard to extend.

Our approach to *MSP* in C++ is based on doing a compilation phase with an incomplete link to generate an object file. This object file can then be demangled (with a tool like `cppfilt` or `nm`) to decode the C++ ABI names. Each demangled symbol will correspond to the internal C++ representation with a complete prototype of the incomplete function. By parsing this representation we can generate the CUDA/OpenCL kernel code with a corresponding host code to complete the link phase. Figure 3 describes this multi-stage process.

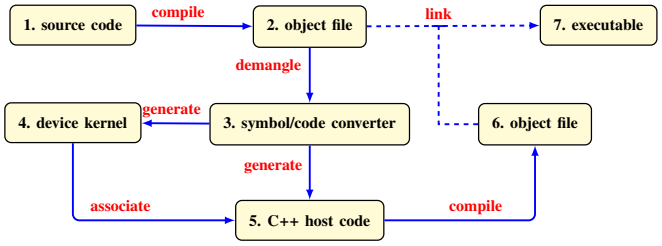

Fig. 3. Two phase compilation for device code generation

A specific tool called *symbol/code converter* (Figure 3, Part 3) coupled with a serialization process was developed as an add-on to NT^2 to solve the issues described previously. The serialization process used to solve the readability of the abstract representation is based on `Boost.Serialization` . This representation is similar to the *expression-template* AST based on `Boost.Proto` that is parsed by the C++ compiler. This enables us to get the semantic information of the container described in section III like its data locality, data type or matrix shape. Having run-time informations like the container size is not possible as it is dynamic.

The role of *symbol/code converter* is two-fold : to parse the demangled symbols obtained and to generate the device and host code. To demangle the symbols, we use `Boost.Spirit` [27], a C++ library to parse expressions and generate outputs based on them. These outputs correspond to the semantic information of the containers and operators. We can then generate the device and host code (Figure 3, Part 4-5) by de-serializing the abstract representation obtained with `Boost.Spirit` .

B. Integration in NT^2

We use the Triad kernel to describe the generation process. This consists in doing a fused multiply-add (or fma : $a = b + c * d$). The resulting code in NT^2 is :

```
// Define host table
table<float> A,B,C,D;

// Triad kernel
 A = B + C*D;
```

Listing 5. NT^2 Triad kernel

The code in Listing 5 corresponds to Part 1 of Figure 3. During the compilation phase, the operation in Listing 5 is replaced with a call to the CUDA transform skeleton. This skeleton will then call the device kernel that is not implemented yet resulting in the incomplete link sequence (Part 2, Figure 3). The equivalent code for transform is detailed in Listing 6. This transform will analyze the informations on

the input matrices and the architecture to decide if it should proceed with the generation process.

```
table<float> A,B,C,D;

// Triad kernel replace with transform call
 transform(A, B + C*D);
```

Listing 6. NT^2 Triad transform

From there, once we compile our code and go through phase 2 of Figure 3 we obtain the mangled code for our transform. We can then demangle the symbols resulting in the simplified code described in Listing 7. This code corresponds to the `Boost.Proto` AST in NT^2 that we parse to generate the host and device code.

```
void nt2::external_kernel<nt2::tag::transform_, nt2::tag::
    cuda_<boost::simd::tag::avx_> >::call<nt2::container::
    table<float, nt2::settings ()>, nt2::container::
    expression<boost::proto::exprns_::basic_expr<boost::
    simd::tag::fma_, boost::proto::argsns_::list3<nt2::
    container::view<nt2::tag::table_, float const, nt2::
    settings ()>, nt2::container::view<nt2::tag::table_,
    float const, nt2::settings ()>, nt2::container::view<
    nt2::tag::table_, float const, nt2::settings ()> >, 31
    >, nt2::memory::container<nt2::tag::table_, float, nt2
    ::settings (nt2::of_size_<-1l, -1l, -1l, -1l>)> > const
    >(nt2::container::table<float, nt2::settings ()>&)
```

Listing 7. NT^2 CUDA Triad demangled

The architectural tag of the machine is depicted in purple in Listing 7 and the fma computation node in red. The function tag in gray corresponds to each element-wise operations in NT^2 . Any function can be constructed using external kernel if necessary. As an example, algorithmic skeletons (scan, reduce, zip ...) will call external kernel when the CUDA back-end is activated. Then, we can generate a **thrust** code in a .cu file that will then be compiled with nvcc. This enables us to overcome a weakness of the **thrust** library while benefiting from its optimized routines.

The representation of the AST in Figure 4 corresponds to the operation obtained from the demangled symbol that we parsed.

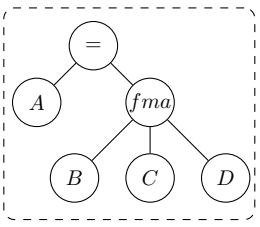

Fig. 4. Triad kernel transform AST

The generated code for the .cu file is described in Listing 8 for Kepler cards. It corresponds to the AST representation with additional semantic information specific to the CUDA language. A function wrapper that calls the CUDA kernel (*fma4_wrapper*) is used to separate the compilation of the .cu file with nvcc from the corresponding C++ host code. The triad kernel directly calls the fma function from CUDA as the NT^2 AST recognizes every occurrence of an fma and replaces it by its function (Listing 7, see boost::simd::tag::fma_).

```
__global__ void fma4(float* t0, const float*  t1, const
    float* t2, const float* t3)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  t0[idx] = fmaf(t1[idx],t2[idx],t3[idx]);
}

void fma4_wrapper(float* t0, const float* t1 , const float*
     t2, const float* t3, dim3 Grid , dim3 Block,
    cudaStream_t & Str, int Shr)
{
  triad<<<Grid,Block,Shr,Str>>>(t0,t1,t2,t3);
}
```

Listing 8. NT^2 CUDA Triad kernel

This optimization in itself is not essential as it can be done by nvcc but demonstrate the potential of code generation. As NT^2 already optimizes the AST by replacing patterns with corresponding functions, we can benefit from this analysis. We can then call the corresponding CUDA function if available or our own implementation for each pattern.

The host code is partially described in Listing 9 and 10. Listing 9 corresponds to the initialization of parameters and data before doing the actual computation on the device. The blockSize and stream number are determined during the generation process depending on the number of parameters and architecture through prior tests. The block-size is generated by measuring the bandwidth of data transfers from host to device in a range and choosing the most optimal one. To benefit best from our Kepler GPU bandwidth for the triad kernel we use a block size of 40000 for single precision values. Double precision computations or a high number of parameters to transfer will lead to a lower block size generated. To decide the block dimension for CUDA, we use is the high value available to the card as element-wise operations are highly parallel and have no dependencies.

```
using boost::proto::child_c;
using boost::proto::value;

size_t size = numel(boost::proto::child_c<0>(a1));
size_t blockSize = std::min(std::size_t(40000),size);
size_t nStreams = std::min(std::size_t(2),size/blockSize);
size_t n = size / blockSize;
size_t leftover = size % blockSize ;
dim3 blockDim = std::min(std::size_t(1024),size);
dim3 dimGrid  = blockSize/size_t(1024);
cudaStream_t stream[nStreams];

// Allocating memory on the device
value(a0).specifics().allocate(blockSize,nStreams,size,true
    );
value(child_c<0>(a1)).specifics().allocate(blockSize,
    nStreams,size);
value(child_c<1>(a1)).specifics().allocate(blockSize,
    nStreams,size);
value(child_c<2>(a1)).specifics().allocate(blockSize,
    nStreams,size);

// checks redundancy between inputs and outputs
std::unordered_set<const float*> addr;
addr.insert(child_c<0>(a0).data());

for(std::size_t i=0; i < nStreams ; ++i)
{
  cudaStreamCreate(&stream[i]);
}
```

Listing 9. NT^2 CUDA Triad Host Code 1

The allocation process includes pinned memory and device memory allocation optimizations. Containers located on the GPU (defined with $nt2 :: device\_$) do not appear in the allocation phase. To limit redundancy in data transfers between outputs and inputs (as we have no information on the pointer) we use an unordered set.

Listing 10 describes the computation phase. It relies on block streaming with transfers to the GPU only if NT^2 tables are on the host. This streaming process is based on the overlap data transfers concept described by NVIDIA. It consists in creating multiple streams (the number depends on the architecture and problem intensity/size) and launching for each stream a transfer host to device, the CUDA kernel and the transfers device to host for a block. As the host memory has already been allocated, we must first transfer the data to pinned memory with cudaHostAlloc to benefit from GPU optimizations. Since the difference in bandwidth between pinned and page-able memory only increases with newer architectures, this optimization can give a speedup even with a mono-stream program. If the host table is declared with $nt2 :: pinned\_$, it can greatly diminish the time needed to transfer the data.

```
for(std::size_t i = 0; i < n ; ++i)
{
  std::size_t j = i % nStreams;
  value(a0).specifics().transfer_htd(a0, i, stream[j], j
      );
  value(child_c<0>(a1)).specifics().transfer_htd(child_c
      <0>(a1), i, stream[j], j ,addr);
  value(child_c<1>(a1)).specifics().transfer_htd(child_c
      <1>(a1), i, stream[j], j ,addr);
  value(child_c<2>(a1)).specifics().transfer_htd(child_c
      <2>(a1), i, stream[j], j ,addr);

  fma4_wrapper( value(a0).specifics().data(j), value(
      child_c<0>(a1)).specifics().data(j), value(
  child_c<1>(a1)).specifics().data(j), value(child_c<2>(
      a1)).specifics().data(j),dimGrid,blockDim,stream[j
      ]);

  boost::proto::value(a0).specifics().transfer_dth(a0, i,
       stream[j], j );
}

if(leftover !=0)
{
  ...
}
```

Listing 10. NT^2 CUDA Triad Host Code 2

As stated in section III, computations on the device only occur if some conditions are met. As of now, these conditions are limited to the problem size and data locality but can be extended as the call to transform is automatic when NT^2 has defined that CUDA is available. Due to the hierarchical tag dispatching in NT^2 , a system with an Intel processor coupled with an NVIDIA card will have a tag similar to the following : $cuda\_ < openmp\_ < simd\_extension >>$. Therefore, if conditions for dispatch on the GPU are not met we will call the next level of transform (*i.e.* openmp). This enables us to use both the CPU and GPU in parallel depending on the problem which is a functionality rarely implemented in libraries. We further this process with our MAGMA back-end which does hybrid computations on most of its solvers.

The code generation is hidden from the user as the generation process is done during the standard compilation phase. The compilation overhead is negligible as the analysis of the AST is linear and the code generation is usually not very long as seen above. The user interface only contains the added semantic information while all the complex allocations are hidden from the user.

V. Experiments

In this section, we will show that our generation process produces satisfactory performances in most situations. The benchmarks are realized with the following components :

- CPU : 2 x 6 cores Intel Xeon E5-2620 15MB L3, AVX
- GPU : Tesla K40m
  - Pageable host to device (HTD) : 3 GB/s
  - Pinned host to device : 9.8 GB/s
- Memory : 65 GB with a memcpy bandwidth of 5GB/s
- GCC 4.9, CUDA 7.5

A. Black & Scholes kernel

The Black & Scholes algorithm represents a mathematical model that gives a theoretical estimate of the price of European call and put options on a non-divideend-paying stock. It is a bandwidth bound algorithm for GPU if we take into account the memory transfers.

The code given in Listing 11 uses the loop-fused technique described previously with the operator tie. The $nt2 :: device\_$ and $nt2 :: pinned\_$ tags are specific to accelerator enabled architectures. If the tag is used while no accelerator is available, NT^2 will fall back to the default architecture which is $nt2 :: host\_$.

```
table<T> blackscholes(table<T> const& S ,table<T> const& X
                      ,table<T> const& Ta , T const r
                      , T const v
                          )
{
  auto s = extent(Ta);
  table<T , device_ > d(s) , d1(s), d2(s);
  table<T> r;

  tie(d,d1,d2,r) = tie( sqrt(Ta)
                      , log(S/X)+(fma(sqr(v),0.5f,r)*Ta)/(v
                          *d)
                      , fma(-v,d,d1)
                      , S*normcdf(d1)-X*exp(-r*Ta)*normcdf(
                          d2)
                      );
  return r;
}
```

Listing 11. NT^2 black and scholes

This additional semantic information on memory locality can help to avoid useless memory transfers while staying simple enough for the user. In our experiment, we consider the worst case scenario where all entry tables are on cpu and not allocated as pinned.

This results in the .cu file in Listing 12 generated for floating point values. Since the AST does not contain the name of the parameters, the kernel generator has to give a different name to each one. This does not lead to a change in performance for the kernel as we just pass multiple times the

same pointer. Memory allocation on device and data transfers between host and device do pointer checking in the host code (see triad example) to insure no redundant work is done incurring also negligible overhead. The fnms function comes from an NT^2 AST transformation and corresponds to the fused negated multiply-subtract of three values.

```
__global__ void bs ( float* t0 , float* t1 , float* t2 ,
      float* t3, const float* t4, const float* t5, const
      float* t6, const float t7, const float* t8, const
      float t9, const float* t10, const float t11, const
      float* t12, const float* t13, const float* t14,
      const float t15, const float* t16, const float* t17,
      const float* t18, const float* t19)
{
 int i = blockIdx.x*blockDim.x+threadIdx.x;
 t0[i] = sqrtf(t4[i]);
 t1[i] = plus(logf(divides(t5[i],t6[i])),divides(multiplies
      (t7,t8[i]),multiplies(t9,t10[i])));
 t2[i] = fnms(t11,t12[i],t13[i]);
 t3[i] = fnms(multiplies(t14[i],expf(multiplies(t15,t16[i])
      )),fastnormcdf(t17[i]),multiplies(t18[i],fastnormcdf(
      t19[i])));
}
```

Listing 12. NT^2 black and scholes fused cuda kernel

The Black & Scholes algorithm involves high latency and high register count operations. This results in sub-optimal performance on SIMD for the CPU due to spilled registers while a Kepler GPU has no such problem.

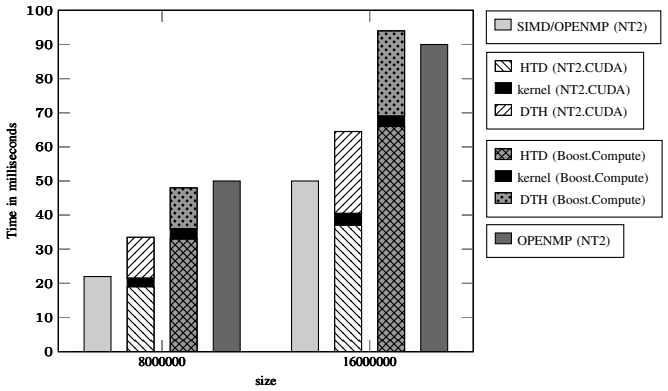

Fig. 5. Black and Scholes Performance Comparison (in ms)

As seen in Figure 5, the execution time of the kernel on the GPU is negligible (3 ms) compared to the overall time of the optimized version with SIMD and OPENMP in NT^2 . Most of the time is spent transferring the data between host and device memory which can be avoided with the right semantic information available ($nt2 :: device\_$) . In the scenario presented (Listing 11) , which is the worse case, we still have better performance than the `Boost.Compute` version on which most C++ libraries are based as we use block streaming with pinned memory reaching a near-optimal throughput (average of 9.7 GB/s) for device transfers. As the bandwidth of a memcpy on the CPU (5 GB/s) is faster than page-able transfers (3GB/s) even without any overlap between transfers and computation we still increase the performance. This optimization can be disabled depending on the CUDA architecture.

As computations on the GPU are often done in great number, if the user allocates the data on the GPU he will pay no transfer cost for the rest of the computations and in this case the CUDA kernel is up to twelve times faster than the hand-optimized version for CPU. Also, using the flag $nt2 :: pinned\_$ if tables must stay on cpu allows us to gain up to a x3 in transfer time. The optimized Black & Scholes version we implemented in CUDA reached the same performance in kernel and transfers time when using the same type of memory allocation for data.

B. Linsolve kernel

Linsolve is a MATLAB routine for solving linear systems. As NT^2 has its own implementation of linsolve with a LAPACK or MAGMA back-end, we can combine it with the code generation process. It also supports mixed-precision algorithms in both CPU and GPU versions through LAPACK/MAGMA or their own implementation. In this benchmark, we consider the solution of a dense linear system using the LU factorization and apply one step of iterative refinement [28] :

1) Compute $r = b - A\hat{x}$.
2) Solve $Ad = r$.
3) Update $y = \hat{x} + d$.

The equivalent NT^2 code is the following :

```
table<T, device_> A,b,x,e;
table<T, settings(device_, upper_triangular_)> u ;
table<T, settings(device_, lower_triangular_)> l ;

tie(l,u) = lu(A)
x = linsolve(l,b);        // lower triangular solve
x = linsolve(u,x);        // upper triangular solve

// One-step refinement
e = b - nt2::mtimes(A,x);
e = nt2::linsolve(l,e);
e = nt2::linsolve(u,e);

x = x + e;
```

Listing 13. NT^2 LU linear solve with iterative refinement

If executed on the CPU, the code in Listing 13 will call the LAPACK routines. The semantic information upper_triangular_ allows `linsolve` to call the triangular solver instead of doing the classic linear solve. If executed on the GPU, the same optimizations will be applied and the iterative refinement process will trigger calls to transform for both element-wise operations.

The performance results in Figure 6 attest that the performance obtained with our model is relevant. The GPU version with *MSP* calls magma kernels using the CUBLAS dgemm routine without doing any transfer and reaches near peak performance of a K40m GPU which corresponds to 1.40 Tflop/s. The version that does not use *MSP* is slower as transfers are done during the iterative refinement step. The CPU version quickly reaches the peak performance of both CPU which is 210 Gflop/s. As we can see, there is no performance loss while call the LAPACK/MAGMA back-ends and if device pointers are passed to our code generator, there will be no memory transfers. Similar performance would also be reached using the other factorizations available in NT^2 .

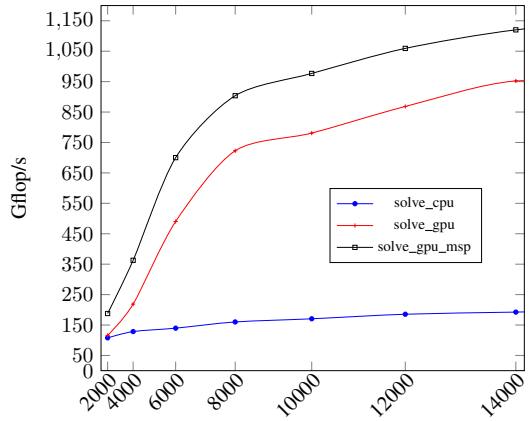

Fig. 6. Performance comparison of NT^2 linear solve (in Gflop/s)

VI. Conclusion

The development of tools for simplifying accelerator programming has been an active topic since accelerators have become a mainstream element of high-performance computing systems. In this paper, we proposed an extension of a high-level, data-parallel scientific computing library to specifically handle GPU accelerators. Our main objectives were to keep a similar level of expressiveness in the client code with a MATLAB -like interface while supporting different use cases of accelerator programming.

To reach this goal, we have implemented a **multi-stage** system in which the initial C++ code is used to automatically generate the equivalent CUDA kernel by reusing our internal representation of this code. This representation is based on C++ *Expression Templates* and the **Algorithmic Skeleton** to identify and classify expressions based on the kind of loop nest that is required. Finally, we showed on a selection of examples that the performance obtained is close to the hardware capability and exhibits benefits compared to other solutions.

Work is still on-going on this system, including the final integration into the main NT^2 release and support for more specific functions on the latest GPUs. Implementing a more thorough cost model to ensure better scheduling of computation between CPU and GPU is also being studied.

References

[1] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, Dec. 1996.

[2] L. Tratt, "Model transformations and tool integration," *Journal of Software and Systems Modelling*, vol. 4, no. 2, pp. 112–122, May 2005.

[3] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha, "DSL Implementation in MetaOCaml, Template Haskell, and C++," in *Domain-Specific Program Generation*, 2003, pp. 51–72.

[4] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen, "Generative Programming and Active Libraries," in *International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, 1998, pp. 25–39.

[5] T. L. Veldhuizen and D. Gannon, "Active Libraries: Rethinking the roles of compilers and libraries," in *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO98*. SIAM Press, 1998.

[6] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[7] P. Esterie, J. Falcou, M. Gaunard, J.-T. Lapresté, and L. Lacassagne, "The numerical template toolbox: A modern c++ design for scientific computing," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3240–3253, 2014.

[8] A. Tran Tan, J. Falcou, D. Etiemble, and H. Kaiser, "Automatic task-based code generation for high performance domain specific embedded language," *7th International Symposium on High-Level Parallel Programming and Applications (HLPP 2014)*, 2014.

[9] J. Hoberock and N. Bell, "Thrust: A parallel template library," *Online at http://thrust. googlecode. com*, vol. 42, p. 43, 2010.

[10] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, and W. Taha, "Implicitly heterogeneous multi-stage programming," *New Gen. Comput.*, vol. 25, no. 3, pp. 305–336, Jan. 2007.

[11] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[12] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[13] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling, "Programming CUDA and OpenCL: a case study using modern C++ libraries," *SIAM Journal on Scientific Computing*, vol. 35, no. 5, pp. C453–C472, 2013.

[14] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs," *Proc. GPUScA*, pp. 51–56, 2010.

[15] K. Lutz, "Boost.Compute," http://github.com/kylelutz/compute, 2015.

[16] G. Guennebaud, B. Jacob *et al.*, "Eigen: A C++ Linear Algebra Library," http://eigen.tuxfamily.org/, 2014.

[17] P. Estérie, M. Gaunard, J. Falcou, J.-T. Lapresté, and B. Rozoy, "Boost.SIMD: generic programming for portable SIMDization," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 431–432.

[18] E. Niebler, "Proto : A compiler construction toolkit for DSELs," in *Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design*, 2007.

[19] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel computing*, vol. 30, no. 3, pp. 389–406, 2004.

[20] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5&6, pp. 232–240, 2010.

[21] W. Taha, "A gentle introduction to multi-stage programming," in *Domain-Specific Program Generation*. Springer, 2004, pp. 30–50.

[22] W. Taha, C. Calcagno, X. Leroy, E. Pizzi, E. Pasalic, J. Eckhardt, R. Kaiabachev, O. Kiselyov *et al.*, "Metaocaml-a compiled, type-safe, multi-stage programming language, 2006," *See: http://www. metaocaml. org*.

[23] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Acm Sigplan Notices*, vol. 46, no. 2. ACM, 2010, pp. 127–136.

[24] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, "Language virtualization for heterogeneous parallel computing," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 835–847.

[25] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: a multi-stage language for high-performance computing," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 105–116.

[26] K. Czarnecki and U. W. Eisenecker, *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.

[27] J. de Guzman, "The boost spirit parser generator framework, 2008," *URL http://spirit. sourceforge. net*.

[28] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.