Seamless GPU Evaluation of Smart Expression Templates

Conference Paper · July 2018

DOI: 10.1109/HPCS.2018.00045

3 authors, including:

Baptiste Wicht
Université de Fribourg
15 PUBLICATIONS **34** CITATIONS
SEE PROFILE

Jean Hennebert
University of Applied Sciences and Arts Western Switzerland
139 PUBLICATIONS **1,427** CITATIONS
SEE PROFILE

Seamless GPU Evaluation of Smart Expression Templates

Baptiste Wicht*, Andreas Fischer†, Jean Hennebert‡
University of Fribourg, Switzerland
HES-SO, University of Applied Science of Western Switzerland

Email: *baptiste.wicht@unifr.ch, †andreas.fischer@unifr.ch, ‡jean.hennebert@unifr.ch

***Abstract*—Expression Templates is a technique allowing to write linear algebra code in C++ the same way it would be written on paper. It is also used extensively as a performance optimization technique, especially as the Smart Expression Templates form which allows for even higher performance. It has proved to be very efficient for computation on a Central Processing Unit (CPU). However, due to its design, it is not easily implemented on a Graphics Processing Unit (GPU). In this paper, we devise a set of techniques to allow the seamless evaluation of Smart Expression Templates on the GPU. The execution is transparent for the user of the library which still uses the matrices and vector as if it was on the CPU and profits from the performance and higher multi-processing capabilities of the GPU. We also show that the GPU version is significantly faster than the CPU version, without any change to the code of the user.**

***Index Terms*—General-Purpose computation on Graphics Processing Units (GPGPU); High Performance Computing; Programming Languages;**

I. Introduction

Smart Expression Templates is a technique allowing to write matrix computation code in a very short and very clear syntax. Moreover, this technique also allows for very efficient evaluation of these complex expressions and is able to perform complex optimization of expressions without changing the user code. There exists many expression templates implementations [1]. However, most of these implementations are mainly focusing on CPU. These days, GPUs are used more and more and are beginning to be the de facto standard for fast computation, especially in machine learning [2]. It has been shown that due to their higher multi-processing capabilities, GPUs are able to significantly outperform CPUs for many tasks [3]–[6]. However, rewriting an existing application to profit from this extra performance is not free [7] and the learning curve may be high [8]. Therefore, it is very important to provide libraries that are able to take advantages of this advanced capabilities without adding the extra complexity to the user, for instance with Smart Expression Templates.

In an ideal GPU Smart Expression Templates implementation, the code written by the user should not change if the code is executed on CPU or GPU. The code should remain exactly the same and the implementation itself should decide where to execute the code, a property we call seamless GPU evaluation. In order to be as convenient as possible, the user code should also be built with the same compiler regardless of the target machine. On the other hand, this simplicity should still allow the compiled code to take maximum advantage of the available capabilities of the target machine. Therefore, the code should perform close to the maximum throughput of either the CPU or the GPU. Finally, the framework should also be designed to be able to take advantage of future hardware implementations.

In this paper, we propose an approach to build a simple, yet powerful, C++ GPU Smart Expression Templates library. Our approach is completely seamless, the code remains exactly the same whether it runs on CPU or GPU. The availability of GPU libraries and the framework options are set through regular compiler options (or macros in an header file). Moreover, it is also possible to compile for both sets of platforms at the same time and choose, at runtime, to disable the GPU for some operations or to force it. The proposed framework was heavily optimized for both CPU and GPU execution. Our approach is made available on the form of a fully-fledged header-only Smart Expression Templates C++ library.

The rest of this paper is organized as follows. Section II provides a detailed explanation of the Expression Templates technique. Next, the related work on GPU Smart Expression Templates is listed in Section III. Section IV defines the architecture of the proposed approach, while its performance is evaluated and discussed in Section V. Finally, conclusions are drawn and possible future work is outlined in Section VI.

II. Expression Templates

In High Performance Computing applications, not only the speed of the program is important but several factors are of equivalent importance [9]. It is necessary to maintain the intent of the code and the extensibility of the code to new technologies. Using a Domain Specific Embedded Language (DSEL) helps improving these factors. In C++, Expression Templates is a very powerful technique that is able to provide a DSEL and that is able to provide a good solution to these factors and execute very efficiently.

Generally, when one wants to write mathematical expressions in C++ close to its mathematical form, one relies on operator overloading. Operators are defined for matrices and vectors and they return a new temporary representing the result of the operation. The first problem with this approach is the creation of a large number of temporaries.

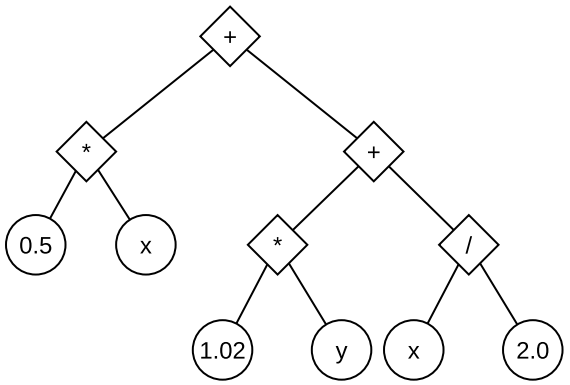


Fig. 1: Expression tree for y = 0.5 ∗ x + 1.02 ∗ y + z / 2.0

```
binary<plus,
       binary<mul,
              scalar,
              matrix&
       >,
       binary<plus,
              binary<mul,
                     scalar,
                     matrix&
              >,
              binary<div,
                     matrix&,
                     scalar>
              >
       >
>
```

Fig. 2: Deducted type for y = 0.5 ∗ x + 1.02 ∗ y + z / 2.0

For instance, the expression

```
y = 0.5 * x + 1.02 * y + z / 2.0
```

would result in the creation of five temporaries. The second problem is that complete execution of the entire expression will need several pass through each matrix or vector. This is highly inefficient on the CPU. To solve these two problems, Expression Templates have been originally introduced for C++ [10], [11]. The user code remains the same as with the naive operator overloading technique, as can be seen in the previous expression example. Instead of relying on temporaries, an expression type is created to represent the complete expression. The expression type is constructed at compilation time following the same concept as the Abstract Syntax Tree of the programming language. Figure 1 shows the expression tree for the code from the previous expression. The type of the expression is outlined in Figure 2. This complex expression can be evaluated in a single loop rather than using several sub computations. It could be evaluated as follows

```
for(size_t i = 0; i < size(y); ++i){
    y[i] = 0.5 * x[i] + 1.02 * y[i] + z[i] / 2.0
}
```

The complete evaluation is done without using a single temporary. With this technique, expressions can be executed as efficiently as the equivalent loop in plain C++ code, yet offer significantly better code. Moreover, advanced optimizations such as vectorization and parallelization can also be performed in order to improve the evaluation time. Finally, they can also be used to implement other features such as automatic differentiation [12] or solving differential equations [13].

Nevertheless, there are issues with Expression Templates. Indeed, although they are highly efficient at the computation of element-wise expressions, they are not nearly as efficient for higher-level operations. For example, they are far from optimal at computing matrix-matrix multiplication operations. Although this operation is easy to write in element-wise form, it is well-known to be very complicated to optimize [14], [15]. While Expression Templates are made to avoid the creation of temporaries, Smart Expression Templates are introducing temporaries when this can improve the efficiency of the expression [16]. For instance, a matrix-matrix multiplication will be computed into a temporary and both of its operand will be evaluated into a temporary if they are expressions. This introduction of temporaries allows the use of advanced kernels for the complex operations such as the matrix-matrix operations. Basic Linear Algebra Subprograms (BLAS) libraries [17] are generally used for this purpose [18]. This was shown to perform significantly faster than standard Expression Templates library [1]. Another advantage is that the expressions can be restructured, to be executed in a more optimized way if there is any. Another problem with the classic Expression Templates way is the poor handling of aliasing [19], [20], that is solved by the analysis of the complete expression types. Complete analysis of the expression type may also fix some of the other problems of the Expression Templates [21]. The code for the user of the library remains the same, but the execution can be significantly faster when using these concepts than when using the classic Expression Templates way.

III. Related Work

Although extensively used for high-performance computing on CPU, execution of Smart Expression Templates on GPUs has only seen few applications.

Wiemann et al. used a technique in which each part of the expression tree is executing a CUDA kernel [22]. However, their proposition is not seamless for the user which must indicates in its code where the data resides. Moreover, CUDA kernels are executed for each sub part of the expression tree, which may not be as efficient as possible.

Chen et al. proposed a technique for offloading the execution of C++ Expression templates to GPU [23]. The CUDA kernels are generated at runtime using Expression Templates and are compiled using Just-In-Time (JIT) compilation technique. Their technique has the advantage of generating full CUDA kernels for each expression. Nevertheless, it needs

access to a JIT compiler which may not be available on each production machine. It also incurs an overhead for compiling the kernels. And it needs access to the hard disk to write the temporary CUDA kernels. A similar approach was taken into the design of the VexCL library [24] and the ViennaCL library [25]. The kernels are also generated at runtime and compiled before being executed to execute the Expression Templates. However, the developer has to choose between CPU and GPU vectors and combination between them is not possible. Finally, this also incurs some overhead for the generation and compilation of the kernels.

Breglia et al. proposed an approach in which the kernel is generated at compile-time using template meta-programming [26]. This technique has the advantage of generating full kernels within the compiler. However, their approach still requires the user to choose between CPU and GPU for each matrix and vector. Moreover, it also requires a CUDA compiler to be available.

In this paper, we propose a seamless approach for Smart Expression Templates in which matrices can be held both in GPU memory and CPU memory. This duality is completely transparent for the user. The framework is responsible for transitions between the two memory spaces. Operations on them can be executed either on CPU or on GPU, depending on the operation and the availability of compute kernels. Moreover, our approach does not require any recompilation or runtime compilation of CUDA kernels.

IV. Architecture

The proposed architecture is very similar to other CPU Smart Expression Templates implementations, except that execution can also be done on GPU, when the necessary support is available. All the code from the user is executed on the host and each expression can be evaluated either on the host or on the GPU device, if possible. The form of execution is decided by the framework, either at compilation time or at runtime depending on the configuration of the library. Currently, the decision to choose the CPU or GPU implementation is simply based upon the availability of a GPU. If a GPU is available, all supported operations are evaluated on the GPU. The decision from the framework can be overridden by the user. While the CPU implementation is parallelized, the GPU implementation does not yet take advantage of CUDA threads (or streams).

The proposed library has been designed for CUDA [27], [28]. CUDA is the compute platform for NVIDIA graphics cards. This platform allows developers to use a CUDA GPU to perform general purpose processing, or General-Purpose Computing on GPU (GPGPU). CUDA makes it easy to execute some parts of the code on the GPU in the form of CUDA kernels. The compilation of CUDA kernels require access to a CUDA compiler.

In order to have a seamless experience between CPU and GPU, each matrix or vector from the library has two memory spaces. The first is the regular CPU data to hold the memory of the container. The second is the GPU data, only allocated when a matrix is used on the GPU. Some specific GPU temporaries do not allocate any CPU memory. The algorithms inside the library are responsible for synchronizing the two data spaces when necessary. For this, each data space is tagged with a boolean indicating if it is up-to-date or not. At every point in time, there must be at least one data space that is up-to-date. When it is required, the system performs a full copy in one direction and updates the state of the boolean flags. These copies are synchronous and require a device synchronization (See Section IV-C). When all the operations are executed on GPU, memory copies are only performed before the operations and at the end to gather the result on CPU.

Another, simpler, solution would have been to use the CUDA Unified Memory Access (UMA) feature [29]. This relatively new feature of CUDA allows to use a single virtual memory space for both the CPU and the GPU. The CUDA runtime is responsible for synchronization between the underlying CPU memory and the GPU memory. However, it was shown that this was not as efficient as it could be if all the transfers were done manually correctly [30]. Therefore, the basic technique of using two memory pointers for each matrix and manual synchronization was used for maximum performance.

A. Complex Kernels

Our library supports a wide range of complex operations. For instance, it has support for convolutions, matrix-matrix multiplications and Fast-Fourrier Transforms (FFTs). It also has support for other various machine learning operations. These operations are relatively complex to optimize and they need optimized kernels to be fast enough. For all these kinds of high-level kernels, the Smart Expression Templates technique is followed. The inputs to the operation are evaluated into a temporary if necessary, i.e. if it is an expression. Then, the operation is entirely computed at once into a temporary. It is then treated as a temporary matrix that can be used in expressions like any other matrix. If it is directly assigned to a container (such as $C = A * B$), the temporary is avoided and the result is directly computed into the left-hand-side matrix. This works as if the operation was done on the CPU, except that it is ensuring that the GPU data is up-to-date and then invalidating the CPU data after the operation.

There exists highly-optimized libraries implementing these operations. The proposed approach directly use the NVIDIA libraries that are available with CUDA:

- NVIDIA CUBLAS for matrix-matrix multiplication and other linear algebra problems [31]
- NVIDIA CUDNN for convolutions and machine learning operations [32]
- NVIDIA CUFFT for FFT computations
- NVIDIA CURAND for random number generation

These libraries are already optimized for many types of NVIDIA graphics card and often provide state of the art performance for the necessary operations. Using these libraries also has the advantage of not requiring a CUDA compiler. One thing proved very important when using these libraries. They all need to perform some initialization steps, the result

of which is stored in a so-called handle. It was shown that the cost of the initialization is significant. Therefore, the handle is created only once and shared for each call to a routine of the library and only released at the very end of the application life. If necessary, this optimization can be disabled by the user.

B. Serial expressions

For a CPU Expression Templates library, serial expressions are executed element by element and the entire expression is evaluated at each step, as shown in Section II. Moreover, the expression will probably be vectorized and computed using several threads. The important concept remains that the expression is evaluated at once.

When implementing a GPU Expression Templates library, a different approach need to be taken. Indeed, since the NVIDIA CUDA compiler only supports a subset of template programming on which Expression Templates heavily relies, it is not possible to realize this in the same way. Therefore, it is not possible to generate CUDA kernels by metaprogramming. This would also require access to a CUDA compiler which may not be ideal. One solution is to generate the code of the kernel in a character string using template metaprogramming and then compile this code at runtime using the NVIDIA JIT compiler, something already done in some other approaches [23]–[25]. However, this requires access to the compiler and to the hard disk for compilation. Since our approach focus on being as seamless as possible, this is not a satisfying solution. Therefore, the proposed approach is based on the classic way of writing expressions, using temporaries. Each sub expression of the expression tree is able to compute its GPU result in a temporary (or directly inside the final result). This makes each sub expression works as if it was a complex expression. This is the similar to the idea proposed in [22]. Therefore, the expression y = 0.5 ∗ x + 1.02 ∗ y + z / 2.0 will be executed using several sub expressions:

```
t1 = 0.5 * x
t2 = 1.02 * y
t3 = t1 + t2
t4 = z / 2.0
y = t3 + t4
```

This will result in a single kernel call for each sub expression (each line). Since not all of these kernels are available in NVIDIA libraries, they have been implemented in CUDA in a complementary GPU library. While this kind of approach would have poor performance when working on a CPU, this is performing very well on GPU. Indeed, this still executes with a good occupancy on the large number of cores of the GPU and since the memory is significantly faster than on the CPU, it has less impact on the overall performance. Moreover, with the optimizations proposed in Section IV-C and Section IV-D, this approach is able to reach excellent performance. When the library is running in CPU mode, complex serial expression are still executed in one pass.

C. Advanced patterns

Unfortunately, the approach outlined in the previous section is not always optimal in practice. Although it is fairly easy and allows to compute all expressions solely on GPU, it requires the launch of many CUDA kernels. In general, it takes around 10 to 100 microseconds to launch a kernel [33]. For expensive computations such as the matrix-matrix multiplication, this overhead is negligible. However, for computations on small vectors and matrices, the launch of a kernel represents a very large overhead. Therefore, it is important to reduce the number of kernel invocations for serial expressions. This cost can be reduced when kernels are launched asynchronously since one kernel can be launched while the previous one is still executing, amortizing the cost of kernel invocation. Nevertheless, it is not always possible to queue many kernels without synchronization. As soon as some data is necessary on the CPU, synchronization will be necessary, waiting for the complete queue of kernels to finish executing. Moreover, allocations also results in synchronization of the device.

Since it is not possible to create a kernel for each possible expressions, the proposed approach defines a few higher-level CUDA operations for some often used mathematical expressions. For instance, the following expressions are all evaluated with a single GPU kernel call:

- $z = \alpha * x * y$
- $z = \alpha * x + \beta * y$
- $z = (\alpha * x)/(\beta + y)$
- $z = (\alpha + x)/(\beta + y)$
- $z = (x + \alpha)/(y + \beta)$

All the selected advanced patterns are detected at compilation time and their evaluation is directly optimized. These patterns are also detected in sub expressions, thus reducing the overall number of kernel calls by a significant factor. This optimization has no runtime overhead. However, it does increase the compilation time of complex expressions. Nevertheless, this can greatly reduce the number of kernels that are being run in the application and improve the GPU occupancy. Another advantage of this approach is that this will reduce the number of temporaries necessary for the computations of complex expressions (see Section IV-D to see why it is important).

The library can easily be extended by adding more complex patterns. If one expression is detected to be executed in a sub-optimal fashion, writing the new kernel in CUDA is fairly easy. It then needs to be detected using meta-programming. Once this is done, the new pattern will always be executed using the new optimized kernel.

D. Temporaries

Since expressions are sometimes evaluated in several parts, temporaries will be necessary to hold the results. These temporaries need to be allocated from GPU memory. It is very important to reduce the number of temporaries that are being created. The first reason is that allocation and deallocation will take some time. And, more importantly, they will result in a full device synchronization, waiting for all the current kernels in the queue to complete. This means that the next kernel call after the synchronization will have a high overhead since it will be loaded while no kernel is running. Therefore, the number of allocations must be reduced to a minimum.

In our approach, the numbers of temporaries is reduced in several ways. First, temporaries are never created for matrices and vectors which have their own GPU memory. Then, the GPU memory of the result (the left-hand-side of the expression) is propagated through the complete expression to be used whenever possible. Moreover, temporaries are often reused as the result of the next expression during evaluation. Finally, since it is not possible to compute all expression without temporaries, a GPU memory pool is used to avoid allocations and deallocations when possible. The GPU pool has a finite number of entries and a maximum size. While reasonable default values are provided, these two parameters can be configured by the user of the library in order to optimize for specific workload and hardware resources. Due to its finite size, the pool can be implemented in an efficient manner and therefore has a very low overhead. Experiments have shown that in the case of most workloads, a small pool was enough to hold all temporaries necessary for the complete program. Indeed, most workloads are working on matrices and vectors of similar sizes and the same expressions are executed many times, making a simple pool ideal.

V. Performance

The efficiency of the proposed approach has been evaluated by comparing the GPU performance against the CPU performance. The library itself was heavily optimized for CPU as well as GPU. The code is exactly the same for both versions, only the compilation options are different in order to enable the GPU capabilities. All the tests are performed on containers in single-precision floating point elements. The tests are performed as follows. Each expression is evaluated many times, to account for a total of two seconds of runtime. A device synchronization is performed just before measuring the GPU total time to ensure that all kernels have finished their work. The average evaluation time is reported as final metric.

All the results presented in this section have been computed on a Gentoo Linux machine, with 12 GB of RAM, on an Intel® Core™ i7-2600, running at 3.4 GHz (CPU frequency scaling has been disabled for the purpose of these tests). Both SSE and AVX vectorization extensions were enabled on the machine. The BLAS operations are executed with the Intel® Math Kernel Library (MKL), in parallel mode. The GPU used for the benchmarks is a NVIDIA Geforce® GTX 960 card. CUDA 9 and CUDNN 5.0.5 are used. The benchmark has been compiled with GCC 7.1.

A. Simple expression

```
yy = 3.3f * x + y;
```

The first expression that is evaluated is the well-known axpy operation. This operation is used repeatedly in many fields.

Figure 3 shows the performance of this operation for both the CPU and the GPU. For 10'000 elements and less, the CPU —●— version is significantly faster than the GPU —■— version. Interestingly, for small number of elements, the GPU time remains the same. This shows that the cost of launching the kernel is higher than the execution time of the kernel. Starting from 100'000 elements, the GPU version is significantly faster, up to one order of magnitude faster for 100 millions elements.

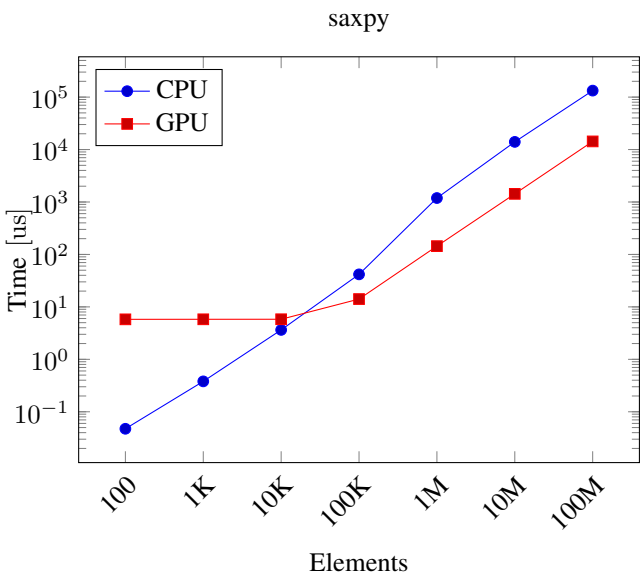


Fig. 3: Performance of saxpy in CPU-mode and GPU-mode for different vector size.

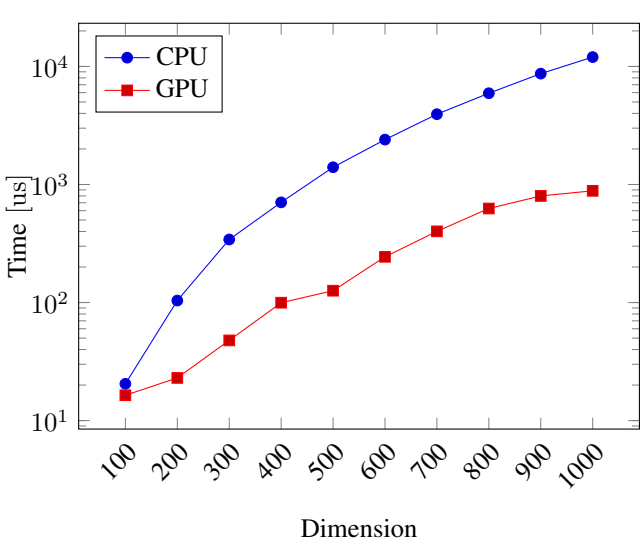


Fig. 4: Performance of sgemm in CPU-mode and GPU-mode for different vector size.

B. Matrix multiplication

```
YY = X * Y;
```

The next expression that is evaluated is the matrix-matrix multiplication operation. This operation is one of the most important operation for linear algebra and machine learning.

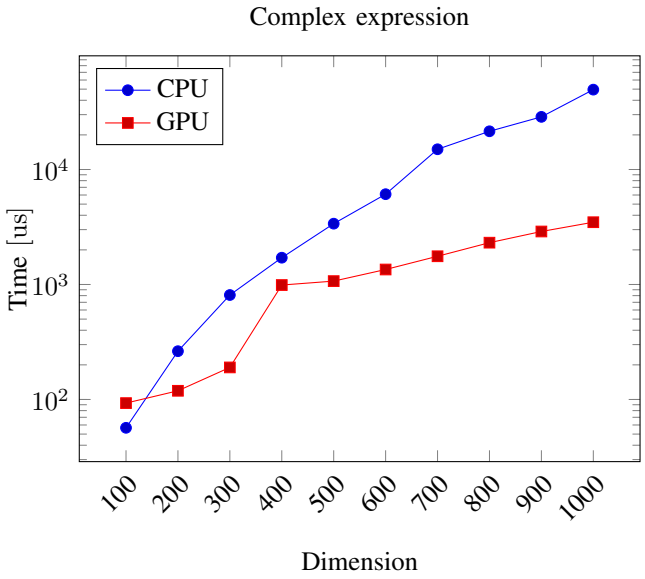


Fig. 5: Performance of a complex expression in CPU-mode and GPU-mode for different vector size.

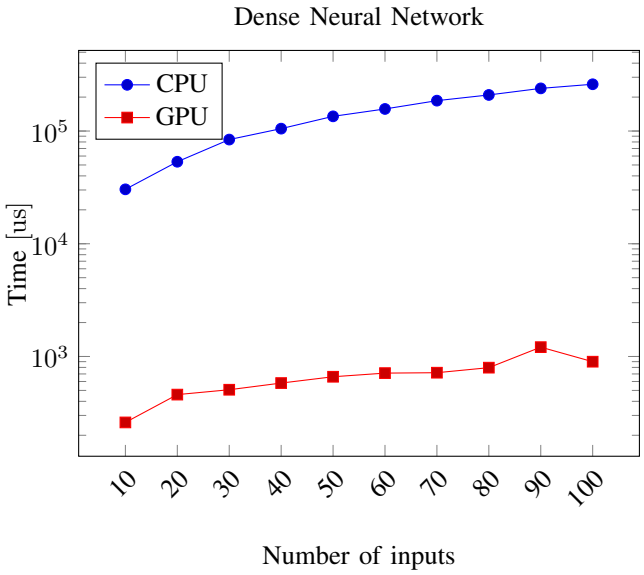


Fig. 6: Performance of the forward pass of a dense neural network in CPU-mode and GPU-mode for different number of inputs.

Figure 4 presents the result of this test for square matrix-matrix multiplication with increasing dimensions. Even for a multiplication of matrices of 100x100 dimensions, the GPU version is already slightly faster and becomes significantly faster than the CPU version as the size increases. For matrices of size 1000x1000, the GPU version is around one order of magnitude faster than the CPU version and the scaling is better, indicating that the speedup should be even better for larger matrices.

C. Complex expression

```
YY = X * (X * 1.2F + Y) * (-1.2F * Y - X);
```

The third expression is a complex expression with several matrix-matrix multiplications and several serial expressions.

Figure 5 presents the result of this test for square matrices with increasing dimensions. Except for matrices of size 100x100, the GPU version is always faster than the CPU version. For the bigger matrices, the speedup is more than one order of magnitude. Nevertheless, the gains are first smaller than they were with a single matrix multiplication for smaller matrices. Once the matrix are becoming bigger, the speedup is becoming bigger as well, amortizing the costs of the memory transfers and the kernel calls.

D. Dense Neural Network

```
O1 = sigmoid(bias_add_2d(I1 * W1, B1));
O2 = sigmoid(bias_add_2d(O1 * W2, B2));
O3 = sigmoid(bias_add_2d(O2 * W3, B3));
```

The next benchmark is evaluating the performance of the forward pass of a fully-connected three-layer neural network. The inputs are 784-dimensional, the two first layers have outputs of size 1000 and the last layer has an output of size 10. This consists of three matrix-matrix multiplications, three additions of biases and three element-wise logistic sigmoid computations.

Figure 6 presents the result of this test for varying number of inputs. In that test, even for 10 inputs at a time, the GPU version is significantly faster and the scaling is much better than the CPU version. For the maximum number of inputs at a time, the GPU version is more than two orders of magnitude faster than the CPU version. In that case, some of the speedup is coming from the matrix-matrix multiplication as before, but more speedup is coming from the cache-inefficient bias computation and the very expensive logistic sigmoid computation.

E. Convolutional Neural Network

```
O1 = relu(bias_add_4d(
        convolution_forward<1,1,1,1>(I1, W1), B1));
O2 = max_pool_forward<2, 2>(O1);

O3 = relu(bias_add_4d(
        convolution_forward<1,1,1,1>(O2, W2), B2));
O4 = max_pool_forward<2, 2>(O3);

O5 = relu(bias_add_2d(
        reshape(O4, N, 32 * 8 * 8) * W3, B3));
O6 = sigmoid(bias_add_2d(O5 * W4, B4));
```

The last benchmark is evaluating the performance of the forward pass of a Convolutional Neural Network (CNN). The inputs of the network are color images of 32x32 size. The first convolutional layer has 16 filters of 3x3 size with padding and the second convolutional layer has 32 filters of the same configuration. Both convolutional layers are followed by a max pooling layer with 2x2 kernal. Finally, two dense layers are used, the first with 1000 hidden units and the last one with 10

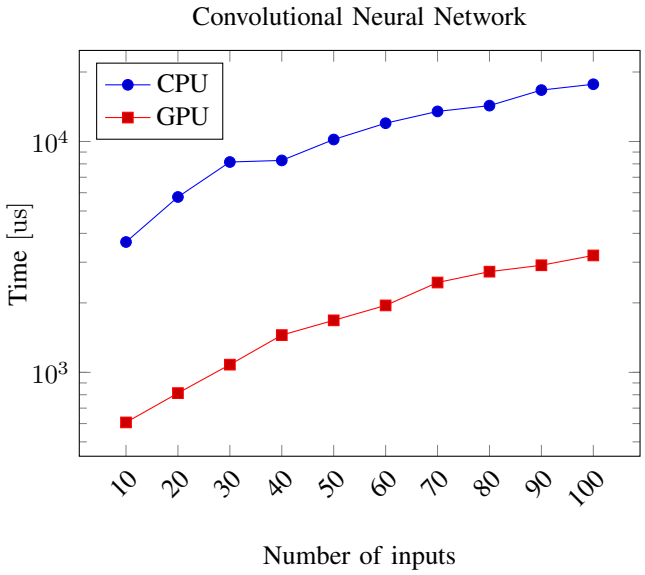


Fig. 7: Performance of the forward pass of a Convolutional Neural Network in CPU-mode and GPU-mode for different number of inputs.

hidden units. All layers use the Rectified Linear Unit (ReLU) function, except for the last layer, using the softmax function.

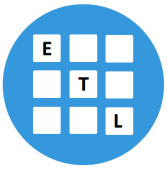
The results are presented in Figure 7 for various number of inputs. In that test, the GPU —■— version is significantly faster than the CPU —●— version. For the maximum number of inputs, the GPU is more than five times faster. For a small network and small inputs and considering that the CPU implementation was also heavily optimized, this is an expected result. In practice, it can be seen that the speedup for larger networks is larger.

VI. Conclusion

In this paper, we presented a novel approach for implementing a GPU expression templates library in a completely seamless way. This approach can be used either on CPU or GPU while always using the full capabilities of the platform, without changing a single line of code. It was shown that the execution of the code on GPU was significantly faster than on CPU, at the cost of a simple recompilation of the code. We believe that GPU Smart Expression Templates offer a very effective way to give the user the full capabilities of the GPU while not adding in complexity to the code.

In the proposed approach, there is still work to be done. The first important task would be to compare the performance of our approach with the performance of other GPU Expression Templates frameworks. Then, some expressions are still only available on CPU. Although this is completely transparent for the user of the library, it means that the data will go back and forth between CPU and GPU, implying a large overhead. Moreover, some operations are still using some device synchronization, either implicitly or because some operations need some extra memory whose allocation forces a synchronization. Reducing as much as possible these points of synchronization could also improve the overall performance. Using CUDA streams could also speed up the library. Indeed, using multiple streams would allow memory copies and kernels to overlap execution. Moreover, when many small kernels are run, making them run in different streams can greatly improve the GPU utilization. However, this would require an advanced dependency analysis of the expressions. Advanced selection of GPU and CPU could also profit the performance of the library. For instance, a small computation could run on the CPU while the GPU is already busy. The existing architecture could also be leveraged to provide new features such as automatic differentiation [34]. Finally, the library does not have any support for multi-GPU. Although, it is able to run on a multi-GPU system, it will only use one GPU. Simply allowing the user to choose the GPU for each operation would be a step in the correct direction. In the case of some very large matrices, it could also be interesting to use multiple GPUs for a single operation, such as matrix-matrix multiplication. This could be achieved using a multi-GPU BLAS implementation [35].

Appendix A
Implementation



Our C++ approach to GPU Smart Expression Templates is implemented on the form of a header-only library, Expression Templates Library (ETL). This library supports vectors and matrices with any number of dimensions. It supports element wise operations such as adding two matrices together or the Hadamard product. It also supports matrix and vector multiplications. Unary operations such as logarithms, exponentials, trigonometric functions or square root are also available and can be applied to matrices or expressions. Not only does the library support single-precision and double-precision floating point values, it also supports integers, booleans and complex numbers. It also has support for various random number generators and distributions. Since the library was built to support machine learning experiments, it has extensive support for operations such as various forms of convolutions, multi-dimensional pooling, embeddings, batch operations or Fast Fourrier Transform (FFTs).

The library was designed to take as much advantage as possible of the available hardware. It has extensive support for vectorization (with SSE and AVX) and parallelization. When available, it uses the efficient BLAS implementation for many routines. Nevertheless, optimized implementations of operations on small matrices have been written when BLAS was not as fast as possible. When efficient implementations are not available, such as for the convolution operations, hand-crafted vectorized implementations have been done. When

possible, optimized kernels are written for the most used configuration such as some convolution or pooling kernel size. Finally, as explained in this paper, it is able to take full advantage of the GPU when it is available. All operations of the library are always available in all compilation and hardware configurations, optimized kernels are only used when possible and when set by the user. Therefore, this library is highly portable as long as a C++ compiler exists for the target platform. Moreover, the library architecture makes it easy to add new implementations of existing operations in order to take advantage of better hardware or better software, without ever changing a line of user code.

This library is available online[1], free of charge, under the terms of the MIT license. The complementary GPU library, etl-gpu-blas, is also available online[2], under the same terms.

REFERENCES

[1] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, "High performance smart expression template math libraries," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 2012, pp. 367–373.

[2] S. R. Upadhyaya, "Parallel approaches to machine learning: A comprehensive survey," *Journal of Parallel and Distributed Computing*, vol. 73, no. 3, pp. 284–292, 2013.

[3] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[4] V. W. Lee, C. Kim, J. Chhugani *et al.*, "Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 451–460.

[5] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, vol. 30, no. 2, 2010.

[6] D. B. Gajić, R. S. Stanković, and M. Radmanović, "A performance analysis of computing the lu and the qr matrix decompositions on the cpu and the gpu," *International Journal of Reasoning-based Intelligent Systems*, vol. 9, no. 2, pp. 114–121, 2017.

[7] A. Heinecke, "Accelerators in scientific computing is it worth the effort?" in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. IEEE, 2013, pp. 504–504.

[8] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Understanding the impact of CUDA tuning techniques for Fermi," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 2011, pp. 631–639.

[9] J. Falcou, "Designing HPC libraries in the modern C++ world," in *High Performance Computing & Simulation (HPCS), 2015 International Conference on*. IEEE, 2015, pp. 458–459.

[10] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, 1995.

[11] D. Vandevoorde and N. M. Josuttis, *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] P. Aubert, N. Di Césaré, and O. Pironneau, "Automatic differentiation in C++ using expression templates and. application to a flow control problem," *Computing and Visualization in Science*, vol. 3, no. 4, pp. 197–208, 2001.

[13] C. Pflaum, "Expression templates for partial differential equations," *Computing and Visualization in Science*, vol. 4, no. 1, pp. 1–8, 2001.

[14] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[15] F. G. Van Zee and R. A. Van De Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, p. 14, 2015.

[16] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, "Expression templates revisited: a performance analysis of current methodologies," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C42–C69, 2012.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[18] K. Iglberger and U. Rüde, "The math library of the pe physics engine–combining smart expression templates and BLAS efficiency," Technical report, Institut für Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, Tech. Rep., 2009.

[19] J. Härdtlein, A. Linke, and C. Pflaum, "Fast expression templates," *Computational Science–ICCS 2005*, pp. 153–179, 2005.

[20] J. Härdtlein, C. Pflaum, A. Linke, and C. H. Wolters, "Advanced expression templates programming," *Computing and Visualization in Science*, vol. 13, no. 2, pp. 59–68, 2010.

[21] F. Bassetti, K. Davis, and D. Quinlan, "C++ expression template performance issues in scientific computing," in *Proceedings of the First Symposium on Parallel and Distributed Processing 1998*. IEEE, 1998, pp. 635–639.

[22] P. Wiemann, S. Wenger, and M. Magnor, "CUDA expression templates," 2011.

[23] J. Chen, B. Joo, W. Watson III, and R. Edwards, "Automatic offloading C++ expression templates to CUDA enabled GPUs," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 2359–2368.

[24] D. Demidov, "VexCL: Vector expression template library for OpenCL," 2012.

[25] P. Tillet, K. Rupp, and S. Selberherr, "An automatic OpenCL compute kernel generator for basic linear algebra operations," in *Proceedings of the 2012 Symposium on HPC*. SCSI, 2012, p. 4.

[26] A. Breglia, A. Capozzoli, C. Curcio, and A. Liseno, "CUDA expression templates for electromagnetic applications on GPUs [em programmer's notebook]," *IEEE Antennas and Propagation Magazine*, vol. 55, no. 5, pp. 156–166, 2013.

[27] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[28] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[29] M. Harris, "Unified memory in CUDA 6," *GTC On-Demand, NVIDIA*, 2013.

[30] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[31] C. Nvidia, "CUBLAS library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2017.

[32] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[33] W. Nicholas, "The CUDA handbook: A comprehensive guide to GPU programming," 2013.

[34] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in c++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 4, p. 26, 2014.

[35] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, "Blasx: A high performance level-3 BLAS library for heterogeneous multi-GPU computing," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 20.

[1] https://github.com/wichtounet/etl

[2] https://github.com/wichtounet/etl-gpu-blas