FDBB

Fluid dynamics building blocks

Möller, Matthias; Jaeschke, Andrzej

Publication date
2020

Document Version
Accepted author manuscript

Published in
Proceedings of the 6th European Conference on Computational Mechanics

Citation (APA)
Möller, M., & Jaeschke, A. (2020). FDBB: Fluid dynamics building blocks. In R. Owen, R. de Borst, J. Reese, & C. Pearce (Eds.), *Proceedings of the 6th European Conference on Computational Mechanics: Solids, Structures and Coupled Problems, ECCM 2018 and 7th European Conference on Computational Fluid Dynamics, ECFD 2018* (pp. 2293-2304). International Centre for Numerical Methods in Engineering, CIMNE.

FDBB: FLUID DYNAMICS BUILDING BLOCKS

Matthias Möller[1,*], **Andrzej Jaeschke**[2]

[1] Delft University of Technology, Delft Institute of Applied Mathematics,
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands
[2] Łódź University of Technology, Institute of Turbomachinery,
ul. Wólczańska 219/223, 90-924 Łódź, Poland,

Key words: Heterogeneous High-Performance Computing, Expression Templates, Meta-Programming Techniques, Computational Fluid Dynamics

Abstract. High-performance computing platforms are becoming more and more heterogeneous, which makes it very difficult for researchers and scientific software developers to keep up with the rapid changes on the hardware market. In this paper, the open-source project FDBB (Fluid Dynamics Building Blocks) is presented, which eases the development of fluid dynamics applications for heterogeneous systems. It consists of a low-level API that provides a unified interface to many different linear algebra back-ends and a lightweight and extendible high-level expression template library, which provides largely customizable fluid dynamics building blocks, like transformations between primary and secondary variables as well as expressions for Riemann invariants, equations of state, inviscid fluxes and their flux-Jacobians. The performance of the developed approach is assessed both for synthetic micro-benchmarks and within mini-applications.

1 INTRODUCTION

High-performance computing hardware is progressing quite rapidly towards more and more heterogeneous platforms, which makes it very difficult for researchers and developers of scientific software to keep up with the latest developments in chip designs and to explore emerging hardware technologies like, e.g., hybrid CPU-FPGA devices, without spending a large part of their time on reimplementing the same algorithms and core functionalities over and over again for the different compute devices. Software packages that explicitly aim at supporting heterogeneous platforms, e.g., multi-core CPU systems combined with many-core accelerators like GPUs, dedicated vector processors, and/or FPGA expansion cards, suffer even more from the rapid changes of the hardware market, since their developers need to ensure that the implementations of an algorithm for the different hardware platforms are kept at the same maturity and functionality level.

Over the last two decades, the trend in high-performance computing (HPC) for practical large-scale applications goes away from writing hand-optimized application codes towards compiler-based code generation and automated performance tuning [1, 3].

*Corresponding author: m.moller@tudelft.nl

It is a long-living myth that the use of expression template meta-programming techniques *automagically* leads to efficient computer code. However, the use of *specialized* expression template libraries (ETL) like Armadillo [12], ArrayFire [13], Blaze [7], Eigen [6], VexCL [4], ViennaCL [11] that provide hardware-optimized linear algebra routines for vectors as well as dense and sparse matrices allows to write concise source code at an abstract mathematical level that gets compiled into executables that exploit the hardware capabilities to the extent implemented by the authors of the linear algebra back-ends. None of them, of course, supports all target hardware platforms and the provided functionality and foreseen use case scenarios largely differ from one library to the other. Despite all differences, the common aim of expression template libraries is to provide mechanisms to formulate mathematical vector expressions like `y=0.5*sin(x+y)` and evaluate them in a single loop over the vector entries rather than creating temporaries for sub-expression.

The Fluid Dynamic Building Blocks (FDBB) project [8] makes an attempt to develop a unified wrapper interface for the core functionality of *all* of the aforementioned linear algebra back-ends and provides an extendible set of expression templates for developing fluid dynamic applications with the focus placed on compressible flows. These expressions include transformations between conservative, primitive and characteristic variables as well as Riemann invariants, different types of equations of state (EOS), as well as inviscid fluxes and their flux Jacobians. FDBB is a header-only C++11/14 library, which is designed to leave the underlying linear algebra back-ends largely unmodified so that applications automatically benefit from improvements in the ETLs provided that their API does not change between versions. The low-order API of our package is released as standalone software, the Unified Expression Template Library Interface (UETLI) [9].

The rest of the paper is structured as follows. Section 2 describes the implementation and typical usage scenarios in more detail. A performance analysis of the low- and high-level APIs is presented in Section 3 followed by conclusions drawn in Section 4.

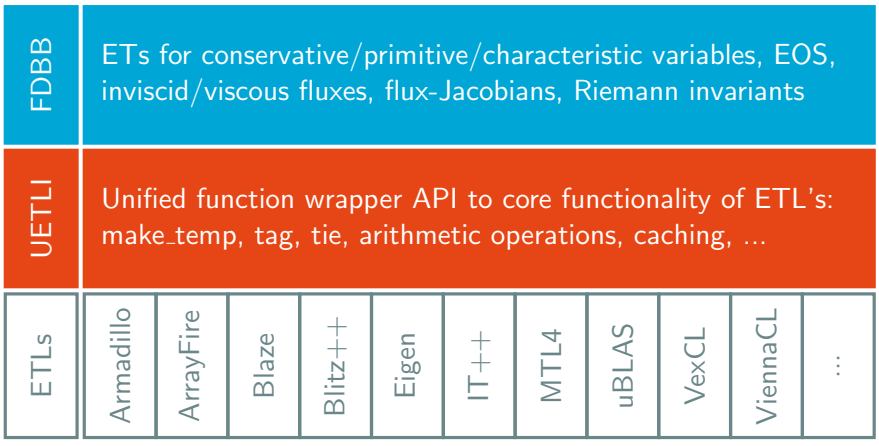

Figure 1: Structure of the low-level Unified Expression Template Library Interface (UETLI) and the high-level Fluid Dynamics Building Blocks (FDBB) open-source header-only C++11 library.

2 IMPLEMENTATION

The overall structure of our software package is depicted in Figure 1.

2.1 Low-level API: UETLI

Core Functionality The different linear algebra back-ends largely vary in functionality, maturity, performance, calling conventions and in the way they evaluate the expressions on the target hardware. Most CPU back-ends employ a delayed evaluation approach based on recursive templates and template meta-programming, to combine several operations into one to reduce (or eliminate) the need for temporaries. In contrast, the multi-device back-ends ArrayFire [13], VexCL [4] and ViennaCL [11] utilize just-in-time (JIT) compilation techniques to convert automatically generated source code into executable code.

```
1 vex::vector<float> x ( ctx, n );
2 vex::vector<float> y ( ctx, n );
3
4 fdbb::tag<1>(y) = CONSTANT( 0.5, y )
5                  * fdbb::elem_sin( fdbb::tag<0>(x)+fdbb::tag<1>(y) );
```

Figure 2: Code snippet for the evaluation of the vector expression $y = 0.5\sin(x + y)$.

Consider the code snippet depicted in Figure 2 that computes the element-wise sine of the vector sum $x + y$, scales the result by the constant 0.5 and assigns it to y. The tag<ID>(expression) function is optional to assign a unique ID tag to the expression, which helps the VexCL [4] back-end to not pass the same expression as multiple arguments to the device kernel. As a general design principle of our software, functionality that is only supported by some ETLs reduce to no-ops in the other cases, which is realized by template specialization. The CONSANT(value, expression) macro ensures that the data type of the constant equals that of the expression result and, moreover, enables further optimization if that is provided by the back-end. The unitary operation elem_sin(expression) is one example of more than 30 element-wise arithmetic operations that can be applied to vectors, dense and sparse matrices, and block expressions, the latter being discussed below.

Figure 3 shows the OpenCL source-code that has been auto-generated by the VexCL [4] library from the above vector expression and can be further processed into CPU or GPU code by the OpenCL subsystem. VexCL also provides code generation engines for CUDA and OpenMP as well as experimental support for Maxeler's dataflow computing platform [10], which aims at making field-programmable gate arrays (FPGAs) usable as next-generation accelerator devices. Maxeler Technologies provides a software development kit consisting of a Java-like programming language, MAXJ, as well as compilers and libraries to synthesizes the high-level compute kernels into bitstreams to reconfigure the FPGAs at runtime. In this case, JIT compilation can take up to several hours or days but, still, the fully automated generation of FPGA bitstreams from mathematical expressions is

```
kernel void vexcl_vector_kernel ( ulong n,
                                  global float * prm_tag_1_1,
                                  global float * prm_tag_0_1 )
{
  for(ulong idx = get_global_id(0); idx < n; idx += get_global_size(0))
  {
    prm_tag_1_1[idx] = ( ( 5.0000000000000000e-01f ) *
      sin( ( prm_tag_0_1[idx] + prm_tag_1_1[idx] ) ) );
  }
}
```

Figure 3: Auto-generated OpenCL kernel code for the expression $y = 0.5\sin(x + y)$.

an attractive feature. Switching to another back-end is realized by changing lines 1–2 of Figure 2 leaving the actual mathematical expression in lines 3-4 unmodified.

Caching Mechanism. The library makes extensive use of rvalue references, move semantics, and perfect forwarding. Wrapper functions are implemented based on the design pattern depicted in Figure 4. If type A requires special treatment in back-end BACKEND other than calling sin(std::forward<A>(a)), the get_element_sin_impl<A> trait needs to be specialized to hold EnumETL::BACKEND in its attribute value. The functionality is then provided by the specialized function elem_sin_impl<A,EnumETL::BACKEND>::eval(A&& a).

```
template<typename A>
auto elem_sin(A&& a)
    #if __cplusplus <= 201103L
    -> decltype(...) // C++11 return type deduction
    #endif
{
  return backend::detail::elem_sin_impl<A,
         backend::detail::get_elem_sin_impl<A>::value>
         ::eval(std::forward<A>(a));
}
```

Figure 4: Static back-end dispatching: Example implementation of the element-wise sine expression.

This approach enables minimally-invasive addition of back-ends and fine-grained control over specialized treatment at the level argument types in unary and binary operators.

The elem_sin(a) function automatically returns an expression object, whose type depends on the adopted linear algebra back-end. It can be either assigned (=evaluated) to a vector or matrix object, i.e. y = elem_sin(x), or passed as argument to another function, i.e. y = elem_sin(elem_sin(x)). The latter is used on Section 2.2 to compose expressions for the inviscid fluxes and the flux-Jacobians from smaller modular building blocks.

However, not all back-ends support the construction of expressions from sub-expressions, which is caused by their inability to store temporarily created sub-expressions as *objects* in further expressions but instead just store *references*, which will become invalid once the underlying objects reach the end of their scope. We have implemented a caching mechanism as a remedy, which is itself a lightweight ETL with full UELTI functionality support. Cache expressions encapsulate the temporal expression *objects* that are generated by the back-end and pass references to them transparently to the back-end functions.

```
1 auto x = fdbb::cache::CacheExpr<0, arma::vec>( arma::vec(10) );
2 auto y = fdbb::cache::CacheExpr<1, arma::vec>( arma::vec(10) );
3
4 auto E = CONSTANT( 0.5, y) * fdbb::elem_sin( x+y );
5      y = E;
```

Figure 5: Code snippet illustrating the caching mechanism.

Figure 5 illustrates the general use of the caching mechanism for the Armadillo [12] back-end. The two column vectors are encapsulated by the CacheExpr<Tag,ExprType> objects, which require unique ID tags next to the expression types. All unary and binary operations return themselves cache-type objects that hold the sub-expression objects internally. The expression object can be obtained using the get() function, which makes it even possible to combine the caching mechanism with native expression template code

```
x.get() = y.get() + arma::randu(10);
```

It should be noted that line 5 in the above code snippet does not trigger evaluation of the cached expression but only assigns the unevaluated and encapsulated expression to the object E. Evaluation happens during the assignment to y in line 6. In practice, lines 5-6 are typically fused unless E serves as sub-expression in multiple further expressions.

The caching mechanisms is moreover a helpful debugging tool. E.pretty_print(os) yields

```
(E(0.5)*sin((E(N4arma3ColIdEE)+E(N4arma3ColIdEE))))
```

An extensive dump of the entire expression tree is produced by E.print_debug(os).

Block Expressions. Lastly, UETLI provides a framework for working with block expressions, that is, expressions that are composed from block matrices and vectors of fixed block size. As an example, consider the following block matrix-vector multiplication

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \sin(x_0 + x_1) \\ \cos(x_0 - x_1) \end{bmatrix}, \tag{1}$$

where $x_0, x_1 \in \mathbb{R}^n$, $y_0, y_1 \in \mathbb{R}^m$, and $A, B, C, D \in \mathbb{R}^{m \times n}$. With the aid of block expressions, this can be implemented as shown in the code snippet depicted in Figure 6. Here, M and

```
1  vex::vector<double> x0 ( ctx, n );
2  vex::vector<double> x1 ( ctx, n );
3  // Initialize row, col, and value vectors for matrices beforehand...
4  vex::sparse::matrix<float> A ( ctx, m, n, rowA, colA, valA );
5  vex::sparse::matrix<float> B ( ctx, m, n, rowB, colB, valB );
6  vex::sparse::matrix<float> C ( ctx, m, n, rowC, colC, valC );
7  vex::sparse::matrix<float> D ( ctx, m, n, rowD, colD, valD );
8
9  fdbb::BlockMatrixView<vex::sparse::matrix<float>,2,2> M ( A, B, C, D );
10 fdbb::BlockColVector<vex::vector<double>,2>            Y (
11                                                          vex::vector<double> ( ctx, m ),
12                                                          vex::vector<double> ( ctx, m ) );
13 auto E  = fdbb::makeBlockExpr<2,1> ( fdbb::elem_sin( x0+x1 ),
14                                       fdbb::elem_cos( x0-x1 ) );
15 Y = M * E;
```

Figure 6: Code snippet for the block matrix-vector multiplication in Eq. (1).

Y (lines 9-12) are a block matrix *view* and block column vector, respectively, and E (lines 13-14) is a block expression consisting of 2 rows and 1 column. Block matrices and vectors can only store objects of the same type, whereas block expressions accept an arbitrary combination of types as it is required to handle the different expression objects returned from the element-wise sine and cosine function. View objects in contrast to the non-view counterparts only store references to the arguments passed, which implies that the sparse scalar matrices A, B, C, and D are not duplicated and copied into M (line 9), which would have been the case if M was of BlockMatrix type. In contrast, the move constructor of the block column vector is used in lines 10-12 so that, again, no data duplication takes place.

makeBlockExpr(exprs...) creates an object of type BlockExpr<nrow,ncol,Exprs...>, which is the most flexible block container since it supports the mixing of back-ends by design. However, most linear-algebra back-ends do not support mixed expressions, i.e., matrix-vector multiplication between a sparse Eigen matrix and a Blaze vector (yet). The aforementioned block types support all unitary and binary operations and element-wise functions that can be applied to a scalar matrix or vector object if they make mathematical sense.

The idx-th sub-item of a block object is accessible via fdbb::utils::get<idx>(obj), whereby all block types adopt row-major storage ordering by default. The latter can be adjusted by passing StorageOrder::ColMajor as template parameter to the block objects.

2.2 High-level API: FDBB

On top of UETLI, we created the expression template library FDBB, which provides the main building blocks for developing fluid dynamics applications.

Variables. Secondary variables can be computed from the primary ones with only a few lines of code. Let $U = [\rho, \rho\mathbf{v}, \rho E]^\top$ denote the state vector of conservative variables in 3D and assume a perfect gas with adiabatic index $\gamma = c_p/c_v = 1.4$. Then the absolute pressure p is computed in a single line as shown in the code snippet depicted in Figure 7. By making dimension N_D ('3') and variable type ('EnumForm::conservative') template parameters, it is even possible to write dimension- and formulation-independent application code. This approach is most effective in combination with factory-based object creation [5].

```
1 using  eos = fdbb::EOSidealGas<double,
2                                std::ratio<7, 2> /* Cp */,
3                                std::ratio<5, 2> /* Cv */>;
4 using varU = fdbb::Variables<eos,3,fdbb::EnumForm::conservative>;
5 // Create and fill vectors rho, mx, my, mz, and rhoE beforehand...
6 auto     p = varU::p( rho, mx, my, mz, rhoE );
```

Figure 7: Code snippet for computing the pressure p from the conservative variables $U = [\rho, \rho\mathbf{v}, \rho E]^\top$.

Instead of passing the scalar variables one by one, it is handy to collect them in a block expression that can be passed as single parameter (see paragraph 'Passing of Variables' below for details). The scalar variables can be accessed via fdbb::utils::get<idx>(U).

```
1 auto U = varU::conservative ( rho, mx, my, mz, rhoE );
2 auto p = varU::p( U );
```

Figure 8: Collection of state variables into block expressions

A further advantage of using block expression is the easy conversion between state vectors. Let the vector of conservative values U be defined as in line 1 of Figure 8. Then the conversion from conservative to primitive variables and vice versa can be realized elegantly as illustrated in the following code snippet. Assuming that well-designed linear algebra back-ends will not perform copy operations if source and destination vectors are the same, no memory bandwidth is lost on unnecessary transfers of the density variable.

```
1 using varV = fdbb::Variables<eos,3,fdbb::EnumForm::primitive>;
2 auto     V = varV::conservative ( rho, vx, vy, vz, p );
3
4          V = varU::primitive    ( U );
5          U = varV::conservative( V );
```

Figure 9: Conversion from conservative to primitive variables and vice versa.

Equations of state. User-defined equations of state of the form $f(p, V, T) = 0$ with absolute pressure p, volume V, and absolute temperature T can be specified by implementing a derived class that implements the prototype EOF_pVT depicted in Figure 10. In a forthcoming release of FDBB, experimental support for the open-source thermophysical property library CoolProp [2] will be enabled, which provides a large collection of equations of state for (pseudo-)pure fluids. Since CoolProp does not accept vector expressions as arguments, its use is currently limited due to the generation of temporary objects. This shortcoming can be overcome by extending CoolProp to accept generic vector arguments and perform computations based on expression templates rather than data directly.

```
1  struct userDefinedEOS : public EOS_pVT
2  {
3    template<typename Trho, typename Te>
4    static FDBB_INLINE auto constexpr p_rhoe(Trho&& rho, Te&& e);
5
6    template<typename Trho, typename Te>
7    static FDBB_INLINE auto constexpr T_rhoe(Trho&& rho, Te&& e);
8    ...
9    static std::ostream& print(std::ostream& os);
10 };
```

Figure 10: Prototype of a user-defined equation of state of the form $f(p, V, T) = 0$.

Inviscid Fluxes. The $N_U \times N_D$ dimensional tensor of inviscid fluxes

$$\mathbf{F}(U) = \begin{bmatrix} \rho\mathbf{v} \\ \rho\mathbf{v} \otimes \mathbf{v} + \mathcal{I}p \\ \mathbf{v}(\rho E + p) \end{bmatrix}$$

is implemented as a ready-to-use block expression, cf. Figure 11, whereby it is assumed that eos and varU are defined as in lines 1–4 of Figure 7 and U is the state vector of conservative variables defined in line 1 of Figure 8. It should be noted that F only holds the expressions, while their evaluation takes place upon assignment to block matrix f.

```
1  using fluxU = fdbb::Fluxes<varU>;
2  auto      F = fluxU::inviscid ( U );
3  fdbb::BlockMatrix<vec_t,5,3> f( F );
```

Figure 11: Code snippet for computing the inviscid fluxes for conservative state variables.

The implementation of flux-Jacobian matrices for the inviscid fluxes is not yet finished and will be enabled in a forthcoming release of the FDBB library.

Passing of Variables. FDBB has been designed with utmost flexibility in mind. Except for a few exceptions, all functions exhibit the same generic interface shown in Figure 12, which allows the user to pass any combination of arguments and leave the mapping to variables to an extra trait that is passed to the variable type, say varU, as additional template parameter. The default behavior is a perfectly forwarding 1-to-1 map from the parameter pack vars ... to the variables, whereby the mapping from the variable, dimension, and formulation triple to the argument index is realized via an extensive specialization of the MapVar2Arg<Var, dim, Form> trait. The default behavior can be changed by providing a user-defined mapping that follows the structure depicted in Figure 13. Additional traits the support the passing of state variables as block objects as illustrated in Figures 8 and 9 are provided and can be further adjusted to the needs of the user.

3 PERFORMANCE ANALYSIS

An extensive performance analysis of all possible combinations of linear algebra back-ends and FDBB features on all supported hardware platforms is beyond the scope of this paper. We restrict ourselves to a synthetic micro-benchmark to measure the computational overhead introduced by the extra FDBB layer and one mini-application.

Micro-benchmark. The kinetic energy or a multiple thereof occurs quite frequently as sub-expression in fluid dynamics applications. We therefore chose the calculation of $\|\mathbf{v}\|^2$ from the conservative state vector in 3D, i.e. varU::v_mag2(U), as micro-benchmark.

All tests were run under CentOS Linux 6.7 with thread pinning (likwid −pin −c N:0−15) on a dual-socket workstation (Intel E5-2670 @ 2.6 GHz, 20MB cache) with 64GB main memory. The ArrayFire and VexCL back-ends were tested in CUDA-mode on an NVIDIA Tesla K20Xm GPU with 6GB memory and ECC turned off. The exact compiler versions were gcc 5.3.0 and nvcc 7.5.17 with CUDA driver version 352.93.

Figure 14 shows the compute performance (left) and the memory bandwidth (right) measured for a wide range of problem sizes for the element-wise expression

$$y \leftarrow (m_x. * m_x + m_y. * m_y + m_z. * m_z)./(\rho. * \rho). \tag{2}$$

Remarkably, no measurable performance loss is observed between the back-end specific implementations (straight lines) and the FDBB-enabled generic ones (symbols).

Mini-application. To estimate the performance of FDBB in real-life scenarios we implemented a mini-app, in which the conservative variables are initialized once by physical values and used to evaluate the inviscid fluxes multiple times. The computing times are

```
1 template<typename... Vars>
2 static auto constexpr conservative(Vars&&... vars)->decltype(...){...}
```

Figure 12: Code snippet for computing the inviscid fluxes for conservative state variables.

```
template<std::size_t dim, fdbb::EnumForm Form>
struct TraitsPerfectForwarding
{
  template<fdbb::EnumVar var, typename... Vars>
  static auto constexpr getVariable(Vars&&... vars) noexcept
    -> const
    typename std::tuple_element<fdbb::MapVar2Arg<var, dim, Form>::index,
                                std::tuple<Vars...>>::type
  {
    return std::get<fdbb::MapVar2Arg<var, dim, Form>::index>(
      std::tuple<Vars...>(vars...));
  }
};
```

Figure 13: Code snippet for a perfectly forwarding 1-to-1 map from arguments to variables.

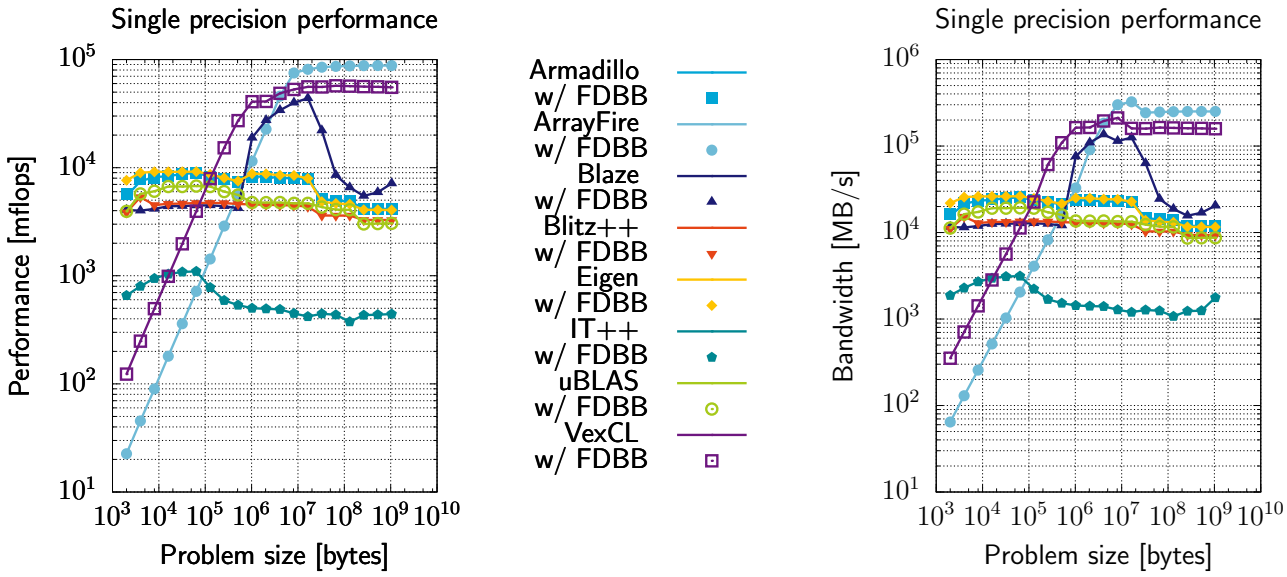

Figure 14: Compute performance (left) and memory bandwidth (right) for the expression given in Eq. (2) computed in single precision with the different linear algebra back-ends on CPUs and GPUs.

given in Table 1. Columns 2–4 show the wall clock-times (in μs) measured for the three efficient linear algebra back-ends Blaze, Eigen, and VexCL. Though all back-ends run in CPU-mode and employ OpenMP parallelization, the VexCL back-end is 1,5x faster then the slowest one for the largest problem size, which consumes 1,25 GB of main memory.

The same mini-app has been run on an IBM Power S822LC server, which consists of 128 cores running at 4.02 GHz and features Nvidia's NVLink interconnect to communicate with four Nvidia P100 Pascal GPUs. The results are given in columns 5–7 of Table 1.

The performance of the two CPU back-ends surprisingly differs by 5x. The savings in terms of computing times resulting from using a single GPU is up to 30x.

Table 1: Wall-clock times (in μs) measured for the mini-app on different hardware platforms.

	2x Intel E5-2670 @ 2.6 GHz			POWER8NVL @ 4.02 GHz + GP100GL		
Problem size	Blaze	Eigen	VexCL	Blaze	Eigen	CUDA-VexCL
1.024	428	420	575	367	149	4.487
2.048	1.264	1.282	1.385	743	322	1.076
4.096	2.667	2.810	2.383	1.476	640	1.074
8.192	5.100	5.330	4.122	2.941	1.277	1.158
16.384	9.610	9.286	7.134	5.932	2.606	5.292
32.768	17.390	15.981	12.135	11.857	5.095	2.896
65.536	29.907	25.778	19.024	353.000	10.495	4.644
131.072	52.377	49.042	34.706	332.000	22.113	10.111
262.144	110.000	110.000	71.622	126.000	46.626	16.983
524.288	250.000	197.000	126.000	48.985	93.918	32.395
1.048.576	338.000	325.000	214.000	68.930	189.000	33.759
2.097.152	605.000	588.000	398.000	104.000	378.000	36.770
4.194.304	1.119.000	1.086.000	750.000	211.000	771.000	49.715
8.388.608	2.166.000	2.114.000	1.442.000	321.000	1.444.000	77.291
16.777.216	4.222.000	4.131.000	2.862.000	639.000	2.910.000	128.000
33.554.432	—	—	—	1.244.000	5.775.000	224.000
67.108.864	—	—	—	2.353.000	11.660.000	382.000
134.217.728	—	—	—	4.634.000	23.692.000	—
268.435.456	—	—	—	9.683.000	47.941.000	—

4 CONCLUSIONS

This paper described the main features and design principles of the FDBB project (`https://gitlab.com/mmoelle1/FDBB`) and provided a brief performance analysis. From the results obtained from a synthetic micro-benchmark we conclude that the additional abstraction layer introduced by FDBB does not cause any performance penalty. Preliminary timings for the more realistic mini-app support our claim that it is possible to write generic codes for heterogeneous HPC systems without sacrificing efficiency. It is, however, necessary to implement mechanisms to choose the optimal back-end for the platform at hand since the performance of the different back-ends can differ by orders of magnitudes.

ACKNOWLEDGEMENTS

The author would like to thank Denis Demidov, Peter Gottschling, Klaus Iglberger, Karl Rupp, and Conrad Sanderson for their support on integrating the different linear al-

gebra back-ends into FDBB. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 678727.

REFERENCES

[1] Basu, P., Williams, S., Straalen, B. Van, Oliker, L., Colella, Ph., and Hall, M. Compiler-Based Code Generation and Autotuning for Geometric Multigrid on GPU-Accelerated Supercomputers, *Parallel Computing (PARCO)* (2017). DOI: 10.1016/j.parco.2017.04.002.

[2] Bell, I. H., Wronski, J., Quoilin, S., and Lemort, V. Pure and Pseudo-pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp. *Industrial & Engineering Chemistry Research* (2014) **53**(6):2498–2508. DOI: 10.1021/ie4033999.

[3] Christen, M., Schenk, O. and Burkhart, H. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. *Parallel & Distributed Processing Symposium (IPDPS)*, 2011 IEEE International. DOI: 10.1109/IPDPS.2011.70.

[4] Demidov, D., Ahnert, K. Rupp, K. and Gottschling, P. Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries. *SIAM Journal on Scientific Computing* (2013) **35**(5):C453-C472. DOI: 10.1137/120903683.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison

[6] Guennebaud, G., Jacob, B. et al. *Eigen v3.* (2010), `http://eigen.tuxfamily.org`.

[7] Iglberger, K., Hager, G., Treibig, J. and Rüde, U. Expression Templates Revisited: A Performance Analysis of Current Methodologies. *SIAM Journal on Scientific Computing* (2012) **34**(2):C42–C69. DOI: 10.1137/110830125.

[8] Möller, M. and Jaeschke, A. *FDBB: Fluid Dynamics Building Blocks.* (2018), Retrieved from `https://gitlab.com/mmoelle1/FDBB`.

[9] Möller, M. *UETLI: Unified Expression Template Library Interface.* (2018), Retrieved from `https://gitlab.com/mmoelle1/uetli`.

[10] Pell, O. and Averbukh, V. Maximum Performance Computing with Dataflow Engines. *Computing in Science & Engineering* (2012) **14**(4):98–103. DOI: 10.1109/MCSE.2012.78.

[11] Rupp, K., Tillet, Ph., Rudolf, F., Weinbub, J., Morhammer, A., Grasser, T., Jüngel, A., and Selberherr, S. ViennaCL - Linear Algebra Library for Multi- and Many-Core Architectures. *SIAM Journal on Scientific Computing* (2016) **38**:412–439. DOI: 10.1137/15M1026419.

[12] Sanderson, C. and Curtin, R. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* (2016) **1**(2):26. DOI: 10.21105/joss.00026

[13] Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, James and Melonakos, J. (2015). ArrayFire - A high performance software library for parallel computing with an easy-to-use API. Atlanta: AccelerEyes. Retrieved from `https://github.com/arrayfire/arrayfire`

[14] Gottschling, P. and Lumsdaine, A. The Matrix Template Library 4. Retrieved from `www.osl.iu.edu/research/mtl/mtl4/`.