A Stencil DSEL for Single Code Accelerated Computing with SYCL

Olivier Aumage, Denis Barthou, Alexandre Honorat

A Stencil DSEL for Single Code Accelerated Computing with SYCL

Olivier AUMAGE Denis BARTHOU Alexandre HONORAT
INRIA – LaBRI
{firstname}.{name}@inria.fr

Abstract

Stencil kernels arise in many scientific codes as the result from discretizing natural, continuous phenomenons. Many research works have designed stencil frameworks to help programmer optimize stencil kernels for performance, and to target CPUs or accelerators. However, existing stencil kernels, either library-based or language-based necessitate to write distinct source codes for accelerated kernels and for the core application, or to resort to specific keywords, pragmas or language extensions.

SYCL is a C++ based approach designed by the Khronos Group to program the core application as well as the application kernels with a single unified, C++ compliant source code. A SYCL application can then be linked with a CPU-only runtime library or processed by a SYCL-enabled compiler to automatically build an OpenCL accelerated application. Our contribution is a stencil domain specific embedded language (DSEL) which leverage SYCL together with expression template techniques to implement statically optimized stencil applications able to run on platforms equipped with OpenCL devices, while preserving the single source benefits from SYCL.

Keywords Stencil, DSEL, meta-programming, GPU, parallelization, HPC.

1. Introduction

Stencil kernels involve concurrently updating every array element with a function of itself and its close neighbourhood cells, using a regular pattern whose width, layout and dimensions characterize a given stencil. Such kernels arise in many scientific codes resulting from the discretization of natural phenomenons with local, short range interactions. While simple in principle, stencil kernels are notoriously hard to code, and even more to optimize, as the arithmetics on array indices and boundary managements are tricky and error prone. Thus many efforts have been put within the high performance computing community on designing stencil programming frameworks and languages, to hide part of this complexity.

Yet, HPC platforms now evolve towards heterogeneity, using nodes equipped with accelerators, resulting in increased complexity and reduced portability. Stencil programming frameworks have to adapt to heterogeneity but face the dual language issue. A heterogeneous stencil framework has to generate general purpose C/C++/Fortran code for the main application side and CPU kernel instances, and some accelerator code such as OpenCL for the accelerated stencil kernel instances. The SYCL [14] language has been designed by the Khronos Group consortiumto address the dual language issue in heterogeneous computing. It leverages regular C++ constructs to enable a SYCL compiler to extract accelerator kernels automatically from a unique valid C++ application source code, without the need for pragmas or keywords.

The contribution of this paper is a C++ Domain Specific Embedded Language (DSEL) based on SYCL to handle stencil computations. It builds on SYCL capabilities and expression template techniques to dramatically reduce the complexity of stencil kernel programming for heterogeneous platforms down to a few lines of code. Our DSEL has been tested with both the triSYCL [13] CPU-only SYCL library (since SYCL is pure C++, CPU-only versions can indeed be implemented as mere C++ libraries!), and with a preliminary version of Codeplay's ComputeCPP SYCL implementation for OpenCL platforms. A preliminary version of the DSEL is available on the triSYCL repository [13] inside the `tests/jacobi` folder, this version only works for triSYCL.

This paper is organized as follows. Section 2 depicts the context and discusses related works on stencil computing frameworks. Section 3 introduces our heterogeneous stencil DSEL on top of SYCL. Section 4 details the implementation and parallelization techniques we used. Section 5 concludes this paper and discusses future work.

2. Context: Computing Stencils on Heterogeneous Platforms

Stencils are widely used in scientific computing especially for image processing, and for physics simulation. Beyond scientific computing, stencils are also used in other performance demanding domains such as special effects in video games [21]. Thus, stencil kernels are key optimization targets for many performance dependent applications. However, the *art* of stencil writing requires skills in both computer science and in the considered application domain, which may be hard to find in the same programmer, or even in the same scientific team.

To reduce or even remove this impeding dual skill prerequisite, many research efforts have been put into designing stencil computing frameworks that separate the generic "computer science side" stencil execution and optimization work, from the "application specific" mathematical stencil expression, that is to implement the so called *separation of concerns*. Two fundamental approaches have been followed for doing so: compilers and libraries. A specific compiler can generate highly optimized code, but it forces the user to write the stencil code in a Domain Specific Language (DSL) which may not be practical for large applications. At the opposite, a

This is the author's version of the work, published in the following workshop:
SYCL'16, March 13, 2016, Barcelona, Spain.

library allows to write code with the same usual language as the rest of the application, still having several implementations dedicated to the desired targets. We refer to it as a Domain Specific *Embedded* Language (DSEL) since it provides a set of dedicated methods within and in compliance with the general purpose language used by the application, and as such can be compiled with any compiler available for this general language. C++ is frequently used as a general purpose language for embedding DSELs due to its ability at being specialized through templating.

Dedicated stencil libraries and compilers include the following ones, among many others. Pochoir [22] uses a stencil declaration language embedded in C++ and provides its own stencil kernel compiler to generate optimized versions, and to provide a level of error checking. Blitz++ [24] is a standalone library. By design, its optimizing capabilities are therefore less aggressive than Pochoir's compiler, but it instead offers some predefined stencil patterns (such as the laplacian operator) and a wide panel of functions to process arrays. PATUS [3] is a framework for stencil kernel generation and autotuning, which targets both CPUs and accelerators such as nVidia CUDA GPUs, in contrast to Pochoir and Blitz++. It implements a DSL for letting the programmer express the stencil expression and layout.

Some general purpose libraries can instead be used to implement stencils such as Kokkos [8]. Kokkos helps with processing arrays in parallel but does not optimize user algorithms. The map-reduce paradigm provides a mean to apply a stencil *mapped* as possible in SkePU [5], Bones [19, 20] or Java Lime [7], but they cannot be as efficient as dedicated stencil libraries.

Applying a stencil on an array and storing the results on another array is an operation with significant potential for SIMD/vector processing and more generally for accelerator and GPU processing. Exploiting the benefit of such hardware for stencil processing is challenging, especially with respect to portability. Some parameters such as workgroup size and local shared memory size depend on the hardware characteristics. An image filtering stencil C++ embedded language has been developed on the HIPA-CC [16] (Heterogeneous Image Processing Acceleration) source-to-source framework to generate optimized OpenCL/CUDA kernels. The Liszt [6] stencil compiler and tool-chain also support CPU/GPU code generation as well as distributed MPI-based code generation. The OP2 project [17] is dedicated to unstructured mesh computations and can produce OpenMP/AVX vectorized code for CPUs, and CUDA or OpenCL for GPUs. The compiler presented by Holewinski et al. [12] generates efficient stencil kernel, but it only targets GPUs. Yet, CPUs may indeed also be used aside or as fall-back for accelerators. Some stencil processing techniques can be leveraged for both CPU and accelerators, such as hierarchical overlapped tiling [27] or 3.5D blocking [18].

Finally, some frameworks can be specialized to some specific stencil *application fields*. For instance, quantum chromodynamics (QCD) simulations involve stencils in 4 dimensions, with variable coefficients depending on the direction. Heybrock et al. [11] present a method to efficiently compute QCD kernels on many-cores processors, but the computer science skills required from the programmer to adapt it to other stencil application fields limit its usability by a wider range of non specialists. The Qiral DSL is dedicated to generating an efficient QCD code [2], unfortunately for CPU only.

In contrast to the related works above, our contribution is instead to build a generic stencil processing framework on top of the recently ratified SYCL standard [14] introduced in the next Section, which leverages the unique properties of SYCL to *reconcile* the library approach and the compiled approach, to target *both* CPU and accelerator, to require a *single* source code, written in the C++ *general purpose* language.

3. Contribution: a SYCL-Based Stencil DSEL for Heterogeneous Platforms

Our contribution is to develop a stencil DSEL that can produce optimized parallel code for all target types that can be managed by SYCL. SYCL provides a common hardware interface and handles memory management. Our library specializes the code with respect to the stencil parameters (dimensions, order, pattern). This approach enables to achieve stencil kernel parallelization without the need for developing a compiler. More importantly, our approach frees the programmer from writing complicated parallel code. Indeed, he/she only needs to describe the application dependent stencil parameters and the data to be processed.

3.1 SYCL in a Nutshell

SYCL [14] is a C++ interface standardized by the Khronos Group, currently at revision 1.2. It aims to simplify parallel code writing in modern computers by overlaying the OpenCL parallelism standard. By bridging the gap between C++ and OpenCL, it enables a single application source code to target regular CPU and OpenCL enabled devices such as accelerators and GPUs. Moreover, OpenCL in itself is not easy to use for an end-user programmer, since he/she must handle many initialization and setup steps such as device discovery, kernel loading and compilation, memory management and transfers between the host and the accelerator board, and so on. A typical OpenCL application indeed starts with dozens of such preparatory calls. SYCL hides most of these steps away, and let the user focus on writing application code instead. Yet, the programmer still retains access to most OpenCL methods and can therefore reuse existing OpenCL code.

Two parallelization models are supported in SYCL. The *global* model is a flat "parallel for" loop. The *hierarchical* model defines *workgroups* and *workitems* concepts as in OpenCL, to abstract teams of cooperating hardware threads, and individual threads in such teams respectively. The use of modern C++ functionalities enable to write kernels with lambda functions, and also, for example, to have implicit barriers between two workitem method calls. These capabilities makes SYCL easy to integrate in other libraries. The common OpenCL memory spaces are addressable in SYCL: host, global, local, private and image. Data must be stored in SYCL *buffers* and can then be referenced in kernels through SYCL *accessors*. Accessors are used to associate a piece of data with one of the aforementioned memory space.

Currently two SYCL implementations exist. triSYCL [13] is a standard C++ library designed by R. Keryell which targets CPU exclusively. It is described as a "*test-bed to experiment with the provisional specification*" of SYCL. ComputeCPP from company Codeplay is a commercial SYCL implementation which targets both CPUs and OpenCL devices. In this work, we experimented with running our own stencil library using both of these SYCL implementations. As a cautionary note, since — at the time of this writing — triSYCL has an experimental status, and since the ComputeCPP version we used is a preliminary, early access version graciously provided by Codeplay, we will not expose quantitative performance results in this paper.

3.2 Stencil DSEL Description

A stencil is an elements' grid pattern which can be applied on every element in order to update its value. The stencil is mapped over the grid and consists of a reduction on the element neighbourhood values. An example of a generic 4-points stencil in two dimensions (involving the two nearest neighbours in each dimension and excluding the central point) is shown in equation 1. A, B and C arrays store the source elements, the destination elements and the coefficients, respectively. Whereas the neighbour elements are by defini-

tion close to the target element in the logical grid, they may not be close in memory, thus the DSEL first requires an *access function* to map logical neighbour indices onto array indices in memory. This is why A and B need the function g to map the element logical grid coordinates i, j onto the actual array indices.

The set of *coefficients*, provided by the array C, is needed to modulate the respective contribution of the neighbour elements on the reduction result. The function f plays the same index mapping role for C as the function g for A and B, with two extra parameters however, to give the relative position of the coefficient in the stencil pattern.

We then define two operators to link the reduction expression components together. The coefficient modulation operator $\otimes$ applies a coefficient weight on the neighbour element contributions. The reduction operator $\oplus$ then accumulates the modulated contributions from each considered neighbour element.

$$\begin{aligned} g(i,j,B) = & f(i,j,-1, \quad 0,C) \otimes g(i-1,j+0,A) \oplus \\ & f(i,j, \quad 1, \quad 0,C) \otimes g(i+1,j+0,A) \oplus \\ & f(i,j, \quad 0,-1,C) \otimes g(i+0,j-1,A) \oplus \\ & f(i,j, \quad 0, \quad 1,C) \otimes g(i+0,j+1,A) \end{aligned} \tag{1}$$

Thus in order to describe a stencil, the framework needs the following information:

- Input data, output data, and coefficients arrays;
- Access function for data arrays;
- Coefficient relative coordinates;
- Extended access function for coefficients arrays;
- Operators for linking coefficients and reductions together.

The three input, output and coefficient arrays and the coefficient relative coordinates are directly application dependent, and must therefore always be provided by the programmer. Sensible defaults can be automatically provided for the access functions (C/C++ usual row-major array layout) and for building a default stencil expression together (using operator $\times$ for applying coefficients and operator $+$ for computing the reduction by default). Such defaults may be overridden by the programmer if needed, to address more complex situations.

The following example shows how the stencil pattern of the equation 1 is created in the DSEL. It requires the only five following lines declaring the four variable coefficients and then assembling them to form a pattern (not related to any value).

```
coef_var2D<-1,0> c1; //left coefficient
coef_var2D<1,0> c2; //right coefficient
coef_var2D<0,-1> c3; //down coefficient
coef_var2D<0,1> c4; //up coefficient
auto st_pattern = c1+c2+c3+c4;
```

3.3 DSEL Design

The DSEL exposes C++ classes for the various coefficient flavors supported by the DSEL. The other classes used by the DSEL to abstract the stencil pattern (an aggregation of coefficients), the input and output information, and the whole expression resulting from applying a stencil on an input buffer and storing the results on an output buffer do not actually need to be exposed to the programmer. The `auto` type keyword and the type inference engine of modern C++ compilers are sufficient to properly determine the respective object types automatically.

Coefficients and stencil pattern Supported coefficients currently come in two flavors: *variable* coefficients such as presented in the example equation 1, and *constant* coefficients that are independent of the target element coordinates. In the first case, the only template parameter needed is the relative coordinates in the stencil pattern, $(0, 0)$ being the current targeted element. The coefficient value will be defined elsewhere in a specific buffer, thus no type is needed in the constant coefficient template parameter since it only describes a pattern and not values. However *internally* to the DSEL, the coefficient evaluation will need a type, which is set later, when a stencil is associated to the typed coefficient buffer. In the second case, the constant coefficient value need to be passed as an argument to the constructor, whereas the expression type is needed as the last template parameter (`float` by default). No coefficient buffer will be needed in this case.

```
coef_var2D<0, 0> c1; //variable
coef_fxd2D<-1, 1 /*, float*/> c2(0.2f); //constant
```

Once several coefficients have been declared, they can be aggregated together to define a stencil pattern using the specially overloaded $+$ operator. It is also possible to add several stencil patterns in a row, provided they are all of the same type, that is with the same number of dimensions and all constant or all variable coefficients.

```
/* creation of a 4 coefficient stencil*/
auto st1 = c1+c2;
auto st2 = c3+c4;
auto st = st1+st2;
```

When a stencil is created with a single coefficient, a specific method is called on it to do the transformation. This case arises when building stencils such as copies or scaling transformations.

```
coef_fxd2D<0,0> c_id {1.0f}; //is a coefficient
auto st_id = c_id.toStencil(); //is a stencil
```

Access functions Access functions correspond to the functions f and g in the equation 1. f and g abstract the data layout used in the buffers. Function g is for the input and output arrays. Function f is for variable coefficients, and is not required when using constant coefficients. It is declined in two versions, for input and output respectively. The input version may returns the element value directly, while the output version needs to return a writable, L-value compatible reference, to write the computation result into. When using the triSYCL library, these access functions can be passed as function pointers and recorded in the input and output objects.

```
/* output access function, notice the & */
float& fdl_out(int a,int b,
  accessor<float, 2, access::mode::write,
           access::target::global_buffer> acc)
             {return acc[a][b];}
/* input access function */
float  fdl_in(int a,int b,
  accessor<float, 2, access::mode::read,
           access::target::global_buffer>  acc)
              {return acc[a][b];}
/* coefficient access function */
float  fac_coef(int a,int b, int c, int d,
  accessor<float, 1, access::mode::read,
           access::target::global_buffer>  acc)
              {return acc[0];}
```

While this approach works for the C++-only triSYCL library, it is not actually compliant with the SYCL 1.2 revision which for-

bids function pointers to be used within SYCL kernels – for good reasons such that a host function pointer would not have any meaning for a kernel executed on an accelerator device. Consequently, this illegal use of function pointers in a SYCL kernel is refused by the ComputeCPP compiler. In order to still enable the programmer to specialize the various access functions, we implemented a second, SYCL 1.2 compliant approach (and inspired indeed by the SYCL language itself) using the following template based function specialization scheme. The key aspect of this approach is the empty class passed as the second template parameter. This class does not need to be defined; it only needs to be declared and solely serve the purpose of discriminating multiple function specializations from each others, just like SYCL's own kernel naming classes [14]. The DSEL then accepts the naming class as a SYCL compliant alternative everywhere it expects a function pointer.

```
class out_access; //use chosen class name
template<>
inline float& fdl_out<float, out_access>
                (...) {...}
```

Operation assembly and use To define an operation, input information must first be put together in a specific object containing the type of the element, the input buffer and associated access function. One may also specify the coefficient buffer and its associated access function when using variable coefficients. The output information is built from the type, the output buffer and its access function.

```
output_2D<float, &outBuffer, &fdl_out>
    workload_out;
input_var2D<float, &inBuffer, &coefBuffer,
            &fdl_in, &fac_coef>
    workload_in;
```

These pieces of information are assembled with the stencil to form the complete stencil expression. The programmer then only needs to declare a SYCL device queue and pass it to the DSEL for the parallel computation to be instantiated.

```
/* st is a predeclared stencil pattern */
auto op_work = workload_out << st << workload_in;
op_work.doComputation(queue);
```

The drawback of this convenient syntax is that buffers must be defined statically. However, the DSEL could also support registering buffers through object constructors instead of template parameters, at the cost of disabling some static parameters inference.

3.4 DSEL workflow

While the DSEL enables the programmer to describe stencils thanks to the dedicated interface presented in the previous section, parallelization is performed automatically in a transparent manner. Thus, the DSEL workflow summarizes as follows.

Main programmer steps The programmer declares coefficients, aggregates them into a stencil pattern, defines function access to data buffers, assembles all these pieces of information together (including actual data buffers addresses) and eventually calls the computation method. The programmer does not have to specify parallelization-related directives or structures, with the exception of the device queue. All parallelization steps are handled internally.

Automatic parallelization Multiple stencil parallelization techniques exist such as color (e.g. red-black) element partitioning or tiling. Our DSEL is based on the tiling approach. For tiled stencil parallelism to work, a ghost boundary is necessary all around each tile. This ghost region supplies a read-only input replicate of the neighbourhood elements values, for updating the target elements located within the *stencil width* distance of the tile edges. An example tile and its ghost region are shown in Fig. 1, were the centre blue cells are the tile elements and the red cells are the ghost copies of the neighbour tiles elements. Once the internal computation of each tile is complete, the ghost region of each tile is updated with the newly computed element values of the neighbour tiles.

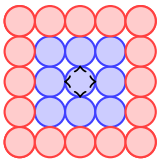

Figure 1. Stencil tile with a ghost region

The DSEL automatically obtains the ghost zone size from the stencil width. After that it computes the parallelization parameters as the size of a workgroup, considering the data type and the number of coefficients. Using these parameters, it builds a specialized parallel SYCL kernel, which can then be further optimized by the SYCL compiler, and then by the back-end OpenCL compiler in accordance with the targeted device.

4. Implementation Details

The DSEL is written in C++14 and is made of some public and private templated classes. For now only around 800 lines of code are needed to implement a basic DSEL version. The last C++14 revision is used following the triSYCL library approach. It enables useful features such as the `auto` keyword. Keyword `auto` leverages the compiler type inference capabilities to relieve the programmer from writing fastidious template types, of which our DSEL makes a large use to automatically compute stencil execution parameters. The present contribution is inspired in this meta-programming approach by Blitz++, developed by T. Veldhuizen. Veldhuizen especially introduced the *expression template* technique [15, 23] used in our DSEL to store and evaluate the stencil pattern. He also made some recommendations in his articles [25, 26], as the *active library* notion which leads to use the partial specialization technique [4].

4.1 Stencil parallelization through SYCL

The grid where the stencil is applied is partitioned in rectangular tiles to be processed potentially in parallel. Following the usual OpenCL terminology, each block is updated by a *workgroup* and each *workitem* in a workgroup updates a single element. Moreover each input element will be read as many times as there are points in the stencil pattern during the stencil computation. Therefore a device *local memory* area, shared among a workgroup, is used to temporarily cache the input elements to read. First, each workitem loads a subset of the global array of elements, corresponding to the tile elements to be updated and to the tile ghost domain, into the potentially faster local memory (depending on the actual device). Second, this local memory is accessed to compute the stencil kernel on the tile elements. The local memory enables to reuse the tile input values multiple times efficiently during the tile update, while the potentially slower global memory has been read only once for that tile.

To compute an element, the stencil kernel needs the input buffer, as well as the coefficient array when using variable coefficients. Since such coefficients may be different in every direction, no reuse is possible, and the coefficient array is left in global memory. When

using constant coefficients, such coefficients are directly inserted in the inlined stencil evaluation function.

Currently the input and output buffers must have the same size and the stencil is applied to the whole data area, writing the results at the same location in the output buffer. Thus, we assume for simplicity, for now, that the output buffer has been allocated with the same ghost domain as the input buffer. Internally the DSEL computes the workgroup sizes independently from the global input/output size. If the global size is not a multiple of the workgroup size, border tiles on the right and at the bottom of the grid are made smaller and a condition disables workitems that would get the out of bound elements to compute.

4.2 Static Parameters Computing with Meta-programming

Static parameters being computed at compile time allow highly specialized host binaries and may dramatically improve the run time when used in parallel sections. One can argue that for GPUs the kernel code goes through another device dependent compilation step at run time, beyond the SYCL compilation step, so the effort may not be necessary. However some SYCL implementation such as the CPU-only triSYCL does not involve the use of OpenCL and therefore can benefit from having the native C++ compiler perform full specialization and static parameter computing. Moreover meta-programming in C++ allows to use these static parameters to raise compilation errors. It helps the developer by detecting structural stencil errors at compile time (if the user attempts to mix fixed and variable coefficients for example), and giving a specific, accurate, and more user-friendly error message.

Static parameters are divided in two categories: those related to the stencil and those related to the parallelization technique. The first one contains for now the stencil width in each dimension direction (and thus the ghost domain width too). The second one is the workgroup size used in SYCL specific code. A distinct meta-programming algorithm is used for each category: a constructive one based on expression templates for inferring the stencil width, and a recursive one using partial specialization for computing the workgroup size.

Stencil width inference Each 2D coefficient/stencil has four static members: the maximum relative position in the stencil for each direction (two times the two dimensions). Here we will denote them by $left$, $right$, top and $bottom$ for convenience. For a coefficient with relative position (i, j) these parameters are simply equal to i for the first dimension (so for $left$ and $right$) and j for the second one (top and $bottom$). When some coefficients are "added" in order to form a stencil pattern, a binary tree is built internally using the expression template technique. This way, each node of the tree is a stencil subpattern and can statically compute the local four parameters depending on their children values. It corresponds to the minimum of the two children' values for the negative directions (i. e. $left$ and top), and to the maximum of these values for the positive directions ($right$ and $bottom$).

The four parameters are needed to compute the ghost size: the needed thickness of the ghost regions in a direction is simply the difference between the dimension maxima ($right - left$ for example). The offset for accessing the updatable (non ghost) values in the input/output buffer is then $(max(-left, 0), max(-top, 0))$. Moreover this technique handles off-center and not symmetric stencils. It also would enable to specialize code with respect to the pattern layout, in a future version of the DSEL. Indeed the pattern building order influences the tree shape. This tree shape controls the input value processing, which are traversed in the same order. For example the two trees and related stencil pattern creation code depicted in Fig. 2 would lead to different performance.

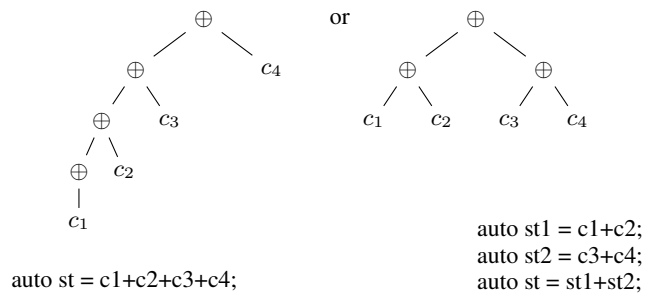

Figure 2. Two ways to create a stencil pattern from coefficients, and their related internal tree representation.

Workgroup size computation Under the assumption of some machine dependent environment variables available at compile time, the workgroup size can be statically computed w.r.t. the target machine maximum workgroup size in each dimension, and also w.r.t. the maximum local memory amount shared in a workgroup. This workgroup size corresponds to the tile size excluding the ghost elements. Notice that for now only one target is enabled in the DSEL implementation.

To do so the current naive algorithm increments the workgroup size by one element in each direction until a stop condition is reached. The stop conditions are the local memory overflow (since thanks to the stencil width the amount of accessed elements in a tile is exactly known) and the maximum workgroup size (which stop the workgroup expansion only in the related dimension). This way the workgroup can have a rectangular or square shape in two dimensions. A `static_assert` checks if the stencil width is larger than the maximum workgroup size at the algorithm beginning. This computations are performed statically using meta-programming with a tail-recursive algorithm.

4.3 Discussion

As mentioned above, we implemented two variants of the stencil DSEL API. The first variant enables the programmer to specify access function as function pointers in template parameters, but it only works for triSYCL. The second variant fully complies with the SYCL 1.2 specification (and with both triSYCL and ComputeCPP implementations), and so does not allow these function pointers, since they cannot be used with OpenCL on which SYCL is based. Although the second version is the one that would be chosen if the DSEL implementation continues, we think that the first version may offer some more benefits for the user: it requires less lines of code and is more traditional in terms of programming style.

The current DSEL prototype suffers from other restrictions as the one depicted at the end of section 3.3. Thus the buffers have to be created statically, and the benefit is to enable future versions of the DSEL to detect if the input and output buffers reference the same object. However to impose static buffers may be a too strong restriction, which may be removed in future versions, together with the static workgroup size computation using environment variables available at compile time, which restricts portability.

Finally the implementation details allow us to classify our DSEL among the two following categories: *shallow* and *deep* [9]. Actually the DSEL belongs to both, depending on which aspect we consider in a stencil. The DSEL is *shallow* regarding the coefficient operators ($\otimes$ and $\oplus$), which are not stored internally but directly inferred by the compiler (the user can directly override the $*$ and $+$ operators in the stencil elements' type class). On the opposite, the

DSEL is *deep* regarding the stencil pattern, which is stored like a tree and so defines a kind of syntax.

5. Conclusion and Future Work

The DSEL introduced in this paper allows to express with ease stencil computation, that is automatic parallelized on both CPU and GPU architectures. The stencil concise abstraction has been made possible thanks to modern C++ features and especially to metaprogramming techniques. Moreover, the parallelization is based on the new SYCL interface which offers a unified parallel programming mean for heterogeneous platforms. The benefits of such a DSEL are to ease the developer work, which then neither has to be a parallelization specialist, nor needs to use several different languages/compilers, or be restrained to a single processing unit type.

However, this DSEL is still a proof of concept at this stage, so many improvements and further experiments are needed on production release SYCL compilers/libraries instead of the preliminary version that have been used for this paper. Among our intended improvements, the priority will be put on the DSEL generalization to any n-dimensional stencil (probably using variadic templates [10]) and on extending the code specialization to both target parameters and modern tiling techniques (e.g. [1]).

Acknowledgments

We are especially grateful to the Codeplay company for providing us with a preliminary version of ComputeCPP and with swift, efficient support throughout the study. We are also very much grateful towards Ronan Keryell for implementing the triSYCL library which help kick-start this work, and for contributing valuable comments on our stencil DESL design.

References

[1] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[2] D. Barthou, O. Brand-Foissac, R. Dolbeau, G. Grosdidier, C. Eisenbeis, M. Kruse, O. Pène, K. Petrov, and C. Tadonki. Automated code generation for lattice quantum chromodynamics and beyond. *CoRR*, abs/1401.2039, 2014.

[3] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *International Parallel and Distributed Processing Symposium*, 2011.

[4] Cppreference.com. Partial template specialization, 2015. URL `http://en.cppreference.com/w/cpp/language/→ partial_specialization`.

[5] U. Dastgeer. Skeleton programming for heterogeneous gpu-based systems. Master's thesis, Linkping UniversityLinkping University, PELAB - Programming Environment Laboratory, The Institute of Technology, 2011.

[6] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[7] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM. .

[8] H. C. Edwards, C. Trott, and D. Sunderland. Kokkos tutorial, a Trilinos package for manycore performance portability, 2013. URL `https://trilinos.org/oldsite/events/trilinos_user_→ group_2013/presentations/2013-11-TUG-Kokkos-Tutorial.pdf`.

[9] J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings. September 2014. .

[10] D. Gregor and J. Jrvi. Variadic templates for c++0x, February 2008.

[11] S. Heybrock, B. Joó, D. D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig, and P. Dubey. Lattice QCD with domain decomposition on intel® xeon phi co-processors. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 69–80, 2014. .

[12] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM. .

[13] R. Keryell. triSYCL, 2015. URL `https://github.com/keryell/triSYCL`.

[14] Khronos Group. C++ single-source heterogeneous programming for OpenCL, 2015. URL `http://www.khronos.org/sycl`.

[15] A. Langer. C++ expression templates, 2003. URL `http://www.angelikalanger.com/Articles/Cuj/→ ExpressionTemplates/ExpressionTemplates.htm`.

[16] R. Membarth, F. Hannig, J. Teich, and H. Kostler. Towards domain-specific computing for stencil codes in HPC. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.

[17] G. R. Mudalige, I. Reguly, M. B. Giles, C. Bertolli, , and P. H. J. Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar)*, 2012.

[18] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5dd blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society. .

[19] C. Nugteren and H. Corporaal. Bones: An automatic skeleton-based c-to-cuda compiler for gpus. *ACM Trans. Archit. Code Optim.*, 11(4): 35:1–35:25, Dec. 2014. ISSN 1544-3566. .

[20] C. Nugteren, P. Custers, and H. Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Trans. Archit. Code Optim.*, 9(4):40:1–40:25, Jan. 2013. ISSN 1544-3566. .

[21] J. Stam. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.

[22] Y. Tang, R. Chowdhury, C.-k. Luk, and C. E. Leiserson. Coding stencil computations using the pochoir stencil-specification language, 2011.

[23] T. L. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[24] T. L. Veldhuizen. Arrays in blitz++. In D. Caromel, R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer Berlin Heidelberg, 1998. .

[25] T. L. Veldhuizen. Guaranteed optimization for domain-specific programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 307–324. Springer Berlin Heidelberg, 2004. .

[26] 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.

[27] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM. .