SYCL Code Generation for Multigrid Methods

Stefan Groth, Christian Schmitt, Jürgen Teich, and Frank Hannig
Hardware/Software Co-Design, Department of Computer Science
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany

ABSTRACT

Multigrid methods are fast and scalable numerical solvers for partial differential equations (PDEs) that possess a large design space for implementing their algorithmic components. Code generation approaches allow formulating multigrid methods on a higher level of abstraction that can then be used to define a problem- and hardware-specific solution. Since these problems have considerable implementation variability, it is crucial to define a general mapping of core components in multigrid methods to the target software. With SYCL there exists a high-level C++ abstraction layer that is capable of targeting a multitude of possible architectures. We contribute a general way to map multigrid components to SYCL functionality and provide a performance evaluation for specific algorithmic components.

CCS CONCEPTS

• **Computing methodologies** → *Multiscale systems*; • **Software and its engineering** → *Parallel programming languages*.

KEYWORDS

Code generation, SYCL, Multigrid methods

ACM Reference Format:
Stefan Groth, Christian Schmitt, Jürgen Teich, and Frank Hannig. 2019. SYCL Code Generation for Multigrid Methods. In *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES '19), May 27–28, 2019, Sankt Goar, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3323439.3323984

1 INTRODUCTION

Solving PDEs is a crucial task in engineering and the natural sciences. Often, an analytical solution to the PDE is not possible or feasible. As a consequence, the transformation of the PDE into a discretized systems of equations and numerical solution is done, to which multigrid methods provide a fast and scalable approach.

Multigrid methods possess a large design space regarding their algorithimic components. One solution to remedy this is code generation, which allows users to specify an algorithm in an abstract notation, and have a tool to create a problem-specific and platform-specific implementation automatically. For this purpose, ExaStencils [1] provides a multi-layer domain-specific language that is capable of generating C++ code from increasingly more abstract problem definitions. When looking at the implementation, the design space magnifies: Code may be executed on CPUs, but other platforms with different vector extension instruction sets exist. Depending on the vendor, libraries to program their respective devices include CUDA, OpenCL, or ROCm. This makes code portability a significant challenge in the engineering of numerical software, since they are often executed on large computing clusters that employ embedded systems to reduce energy consumption [2]. Ensuring that source code is executable on each possible architecture requires additional abstraction compared to standard parallel programming libraries.

A solution for this is SYCL, which is a C++ abstraction layer for writing parallel programs on heterogeneous architectures. It allows the implementation of code for device and host code in a single-source multiple compiler-passes (SMCP) design. Single-source means that the kernel code can be written in the same file as the host code which allows using most of the C++ standard as well as compiler features such as type checking or other static analysis. With this approach, source code containing kernels is compiled once for the device and once for the host, thus generating the appropriate communication and algorithm logic for the target architecture [3]. A SYCL application consists of explicitly parallel command groups, which combine a set of necessary data accessors with a kernel, and a program running on the host device. The kernel is associated with an iteration space, and its body is executed in parallel for each point in this range on the device. Depending on the necessary data accessors, the SYCL runtime may schedule the execution of kernels in parallel depending on their overlapping accessors. This way, a task-graph is built during program execution that ensures correct execution order between kernels.

SYCL provides an interface for programming devices without explicit need to use an architecture-specific API. A wide range of platforms can already be targeted by the SYCL back ends currently in development. Codeplay's ComputeCpp is an implementation of the SYCL standard that supports CPUs and OpenCL devices and provides experimental functionality for Nvidia GPUs and SPIR-V devices. TriSYCL provides an implementation of the SYCL standard, which allows parallelizing code with OpenMP and additionally targets FPGAs by Xilinx.

The contributions of this paper are:

- A general approach to realize the recursive, non-linear multigrid formulation for different schedules as a SYCL application.
- An evaluation of two compute-intensive components of a standard multigrid application with the triSYCL[1] and ComputeCpp[2] SYCL back end compared to a CPU-only implementation.

Next, we introduce the basics of Multigrid methods in Section 2 followed by a mapping strategy to SYCL components in Section 3, an evaluation in Section 4 and related works as well as a conclusion in Section 5.

[1]https://github.com/triSYCL/triSYCL
[2]https://www.codeplay.com/products/computesuite/computecpp

2 ALGORITHMIC COMPONENTS OF MULTIGRID METHODS

This section introduces the general algorithmic components of the recursive, non-linear multigrid formulation. For an in-depth explanation, the respective literature is recommended [4]. We implemented the general principles described here in the ExaStencils code generator [1] to produce C++ code that parallelizes the application of stencils with SYCL. The essential idea of multigrid methods is based on two principles [4]:

(1) Standard iterative solvers can reduce the high-frequency component of an error term in a few iteration steps significantly. This behavior is called the smoothing property.
(2) It is possible to sufficiently approximate a discretized problem with fewer discretization points as long as its error term is smooth. Furthermore, by restricting the problem to fewer discretization points, it is possible to transform the low-frequency components of an error term to high-frequency components to efficiently apply the smoother again.

These principles combine to an iterative algorithm that recursively traverses a hierarchy of D-dimensional grids G_k that exist within the computational domain $\Omega \subset \mathbb{R}^D$. These grids $G_k, 1 \leq k \leq K$ possess an increasing amount of discretization points compared to their predecessors. For example, G_{k+1} doubles its amount of discretization points in each dimension compared to G_k if the standard coarsening strategy is employed. Let the problem on G_k be denoted

$$L_k \cdot u_k(x) = f_k(x)$$

The solution $u(x)$, the right-hand side $f(x)$, and the residual $r(x)$ are functions that depend on the grid positions $x \in \Omega$. We denote them as u, f and r from this point on to not obfuscate the algorithm.

Algorithm 1 shows a recursive formulation of one multigrid cycle. Note that multigrid methods are very flexible regarding their particular composition and thus, each step depicted in the algorithm has multiple alternative implementations. On the coarsest grid $k = 1$, the problem is either solved directly by algorithms such as the conjugate gradient method, GMRES, LU factorization, or by many smoothing operations. A suitable smoothing method $\mathtt{S}(u, L, f, i)$ applies pre- and post-smoothing. It improves the initial guess u and does so in i iterations. General examples include the (red-black) Gauss-Seidel or Jacobi methods, and many other problem-specific smoothers exist. Then, a new residual approximation r_k is calculated and restricted with an interpolation operation $R : r_k \longrightarrow r_{k-1}$. Again, we may consider different approaches, such as considering only direct neighboring values or also considering diagonal relationships. The resulting vector with fewer discretization points is used to approximate the solution on a coarser grid. This procedure is then recursively executed. The coarser solutions v_{k-1} are prolongated with the help of an additional interpolation method $P : v_{k-1} \longrightarrow v_k$ and used to correct the residual on the finer grids to return to the original amount of discretization points.

Depending on γ, the algorithm results in different schedules, which have better convergence rates depending on the problem. Selecting $\gamma = 1$ yields a so-called V-cycle, which traverses each grid size twice, while other options include the W-cycle by setting $\gamma = 2$. They both are depicted in Figure 1, but we may also modify the cycle further, for example, start at the coarsest grid.

```
function MG( $u_k, L_k, f_k, \gamma, i_1, i_2$ ):
    if $k = 1$ then
        solve $L_k \cdot u_k = f_k$                      // coarsest grid
    else
        $\bar{u}_k = \mathtt{S}(u_k, L_k, f_k, i_1)$     // pre-smoothing
        $r_k = f_k - L_k \cdot \bar{u}_k$                // calculate residual
        $r_{k-1} = R \cdot r_k$                          // restriction
        for 1 to $\gamma$ do
            // recursion
            $u_{k-1} = \mathtt{MG}( v_{k-1}, L_{k-1}, r_{k-1}, \gamma, i_1, i_2)$
        end
        $v_k = P \cdot v_{k-1}$                          // prolongation
        $\tilde{u}_k = \bar{u}_k + v_k$                  // correction
        $u_k = \mathtt{S}(\tilde{u}_k, L_k, f_k, i_2)$   // post-smoothing
    end
    return $u_k$
end
```

Algorithm 1: Recursive formulation of a multigrid cycle. Executing this on the finest level l in the j-th iteration results in a better solution for the original problem $u_l^{(j+1)}$.

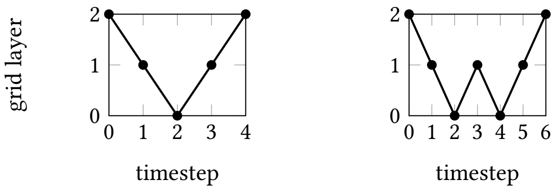

Figure 1: The V-Cycle and W-Cycle for two grid layers and their respective timesteps.

$$\frac{1}{h_k^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} \cdot u_k(x, y)$$

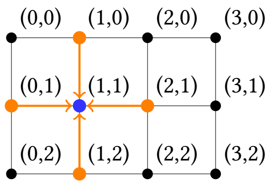

Figure 2: The five-point approximation of Poisson's equation and a graphical representation of a stencil in a 2D equidistant grid. The blue central node's value is computed from the values of its orange neighbors.

The restriction, prolongation, pre- and post-smoothing operators in the multigrid formulation can be realized through stencil computations, which are the primary target for parallelization. Stencils are usually defined on regular grids, i.e., equidistantly placed values in a fixed space, and describe update rules of a center node depending on a local arrangement. For example, Figure 2 depicts the two-dimensional discrete Poisson's equation with eliminated Dirichlet boundary, a classical model problem, where $h_k = \frac{1}{N_k}$ and $N_k \in \mathbb{N}$ is the total number of grid points on layer k, as well as a graphical despcription of a stencil computation.

The residual $r = f - L \cdot \bar{u}$ is a measure of how much an approximation $\bar{u}$ fulfills the original problem $L \cdot u = f$. It is normed to define a criterion for convergence of the multigrid algorithm because the error function $e = \bar{u} - u$ is not available. The multigrid cycle MG described in Algorithm 1 is executed until the normed residual $\|r\|$ is small enough.

3 REALIZING MULTIGRID WITH SYCL

To evaluate SYCL in a multigrid context, a general mapping strategy for multigrid components to SYCL functionality is introduced in this section. While the formalisms presented in Section 2 are explained in a general way, adapting multigrid methods to difficult problems does not change these components but instead uses them to express more complex mathematical concepts.

Aside from targeting a multitude of architectures, one of SYCL's main advantage is its task graph model expressed through accessors to buffers in a specific kernel. This task graph abstracts data flow and removes the need to place inter-kernel synchronization points within an application explicitly. Each stencil operation over the entire grid is realized as a one-dimensional command group where the grid size of the respective multigrid layer defines its iteration domain. Since stencil codes are regular for each considered point in the grid and do not change for a specific operation during the multigrid algorithm, we can take advantage of the code generation approach and directly resolve the concrete neighborhood weighting given by the stencil in the kernel.

However, mapping the stencil codes for smoothing still requires two kernels. While Jacobi smoothers theoretically expose full parallelism, handling each point in the grid simultaneously, they still have to be implemented in a way such that no intra-kernel data race exists. Here, SYCL allows for a higher level of abstraction than normal OpenCL barrier synchronization: So-called parallel-for-work-groups allow defining more than one kernel in the same command group and implicitly ensure that all points in a kernels' iteration domain finish before the next kernel starts. We model Jacobi smoothers as such a parallel-for-work-group and separate all read from write accesses. This approach simplifies the synchronization process compared to barrier synchronization or a double buffering scheme.

Gauss-Seidel iterations may employ coloring schemes that execute smoothing on subsets of grid points in a specific order. For example, red-black coloring handles all points with even indices first and the rest afterward. These coloring schemes are handled by executing the same smoothing operation twice with different indexing checks, thus also requiring two kernels.

Furthermore, enforcing boundary conditions (BCs) requires two additional kernels per dimension to calculate the corresponding values in the boundary regions.

These considerations imply that it is necessary to implement a parallel kernel for restriction, prolongation, correction as well as two kernels for smoothing and $2 \cdot D$ kernels per grid function for each multigrid layer, leading to a total of

$$(K - 1)(4 + 6 \cdot D) \tag{1}$$

necessary kernels to implement for this general multigrid formulation. This formula shows the advantage of employing a code generation approach paired with SYCL compared to standard parallel programming:

(1) Through code generation, it becomes unnecessary to implement a large number of kernels manually.
(2) SYCL implicitly handles data synchronization, which becomes infeasible to do during generation time with increasingly complex multigrid methods.

To reduce the total amount of buffer allocations necessary, we decided to declare each buffer at a globally accessible namespace and allocate them in an initialization function before the actual multigrid algorithm. The functions r, f and u require a buffer per multigrid layer. Note that the prolongation and correction can be combined to one kernel, therefore saving a buffer for v, since v_{k-1} is the same as u_{k-1}. The functions themselves are realized as one-dimensional SYCL buffers and their grid values are ordered row-major. For example, in a square two-dimensional grid of size H^2, an entry at position (x, y) is located at buffer entry $x + H \cdot y$.

The space allocated inside a buffer does not only depend on the total amount of grid points, but depending on the applied BCs it might be necessary to also designate space for so-called ghost points in the respective buffer. Depending on the maximum stencil size used inside the multigrid algorithm each grid is extended such that it is not possible to access out-of-bounds coordinates by applying a stencil. This approach is preferable to defining boundary treatments inside the kernels because it would lead to branch divergence in a parallel dispatch, which is undesired in GPU programming [5].

Calculating the normed residual entails the generation of a reduction procedure that extracts a singular value from a buffer. We introduce global buffers that are reused for each reduction to reduce the amount of total data allocations necessary. They temporarily store mappings before the actual reduction. For example, it is necessary to calculate the square of each entry of r for the L_2-norm. The reduction itself is in-place and employs standard optimization techniques that can be found in [6].

4 EVALUATION

The execution times (ETs) reported in this section were measured on an Intel Xeon CPU and an Nvidia Tesla K20c GPU. We compare Codeplay's ComputeCpp targeting the GPU, triSYCL's OpenMP parallelization on the CPU, and a sequential implementation.

Smoothing is the most time-consuming part of a typical multigrid application. Figure 3 shows the speedup of using ComputeCpp and triSYCL compared to a sequential implementation of a red-black Gauss-Seidel smoother generated by the ExaStencils code generator [1]. While triSYCL and ComputeCpp do not provide a significant speedup for buffer sizes 10^5 and 10^6, the potential for SYCL parallelization is shown by the increasing speedup for the largest buffer size when using ComputeCpp. Note that using SYCL is slower than the sequential implementation for buffer sizes smaller than 10^5. This performance impact means that it becomes crucial to determine which multigrid layers one should parallelize with the current SYCL implementations.

We also found that calculating the normed residual and reporting it back to the host is another performance-critical component in our generated code, especially when considering that this calculation is a relatively small part of a standard multigrid algorithm. Figure 4 shows a comparison of execution times for the implemented reduction targeting the GPU with ComputeCPP compared

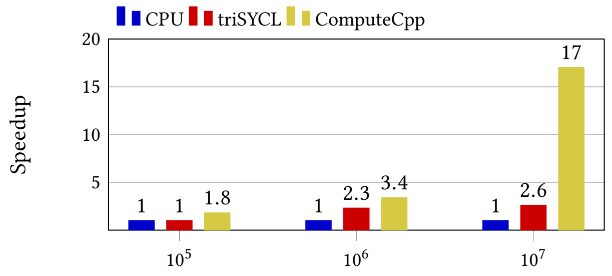

Figure 3: The speedup provided by employing SYCL for red-black Gauss-Seidel smoothing for buffer sizes 10^5 to 10^7.

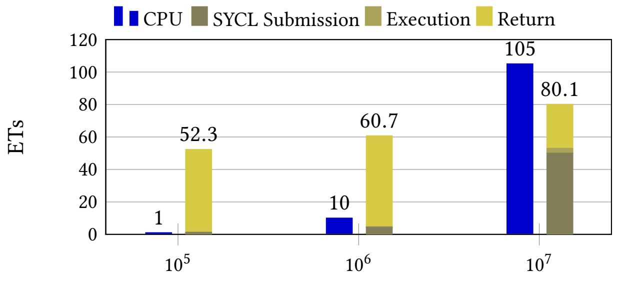

Figure 4: ETs for `std::acummulate` on the CPU compared to a reduction on the GPU with ComputeCpp for buffer sizes 10^5 to 10^7.

to an `std::accumulate` call on the CPU. The latter is split into the amount of time spent between kernel submission and start of execution, time spent within the kernel, and amount of time spent until the result is reported back to the host. This relation indicates that communication overhead takes up a significant amount of time while the actual kernel execution is almost negligible. It is worth noting that the reduction using triSYCL is not listed in this figure because the resulting ETs are 0.4, 4.5 and 44.7 seconds for buffer sizes 10^5 to 10^7, respectively.

A reason for these ETs is the fact that the implementation of parallel-for-work-groups in the triSYCL back end is slow. Additionally, a standard reduction implementation decreases the amount of considered buffer entries by a factor of $2 \cdot W$ in one step, where W is the maximum number of work-items. Since W is significantly smaller for CPUs compared to GPUs, this also impacts performance. Other reduction strategies might be necessary for embedded devices that possess a different programming model. Although the SYCL parallel STL[3] provides a `reduce` function, an optimal implementation heavily depends on the target architecture, which raises the question why there is no built-in structure in the SYCL specification for such a standard technique.

5 RELATED WORKS AND CONCLUSION

Trigkas [7] evaluated a prototypical implementation of the SYCL standard called Syclone. They ported six applications from three different benchmarks to SYCL, OpenCL, and OpenMP to compare execution times. Of particular interest to this paper are their results of the "27stencil" application from the EPCC OpenACC benchmark suite[4], which applies a 27-point stencil in a 3D neighborhood for fields of size 10^6 to 10^7. They measured execution times on an Intel Xeon and Intel Xeon Phi and found that SYCL requires 211 seconds to run on the most extensive dataset while a sequential execution only takes 61 seconds. They further suspect that these poor execution times are tied to data transfers because the SYCL runtime handles them instead of the programmer.

Silva et al. [8] also evaluated the "27stencil" benchmark to evaluate the performance of triSYCL compared to an OpenMP and OpenCL implementation. They found that their OpenCL implementation is 2.35 times faster and the OpenMP implementation 2.22 times faster than their SYCL implementation. They also state that the triSYCL implementation is competitive with OpenCL in terms of ETs, while requiring significantly fewer lines of code.

The SYCL standard itself provides a straight-forward approach to implement algorithms that can target a variety of heterogeneous architectures that exposes a high degree of parallelism without having to state synchronization points in the resulting program explicitly. Therefore, SYCL is well suited for code generation when back end performance improves further.

ACKNOWLEDGMENTS

This work was partially supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 "Software for Exascale Computing" in project ExaStencils under contract number TE 163/17, and by the German Federal Ministry for Education and Research (BMBF) within project "HighPerMeshes" (01IH16005B).

REFERENCES

[1] C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, H. Köstler, U. Rüde, and C. Lengauer, "Systems of partial differential equations in ExaSlang", in *Software for Exascale Computing – SPPEXA 2013–2015*, ser. Lecture Notes in Computational Science and Engineering, vol. 113, Springer, Aug. 2016, ISBN: 978-3-319-40526-1. DOI: 10.1007/978-3-319-40528-5.

[2] D. Jensen and A. Rodrigues, "Embedded systems and exascale computing", *Computing in Science Engineering*, vol. 12, no. 6, pp. 20–29, Nov. 2010, ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.95.

[3] R. Keryell, M. Rovatsou, and L. Howes, "SYCL integrates OpenCL devices with modern C++, Version 1.2.1", The Khronos Group, SYCL Specification, Jul. 19, 2018. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf.

[4] U. Trottenberg, C. W. Oosterlee, and A. Schüller, *Multigrid*. Academic Press, Dec. 2000, 631 pp., ISBN: 978-0-12-701070-0.

[5] Nvidia, "OpenCL Best Practice Guide", Feb. 2011.

[6] J. Nickolls, I. Buck, and M. Garland, "Scalable parallel programming", in *2008 IEEE Hot Chips 20 Symposium (HCS)*, Aug. 2008, pp. 40–53. DOI: 10.1109/HOTCHIPS.2008.7476525.

[7] A. Trigkas, "Investigation of the OpenCL SYCL programming model", Master's thesis, The University of Edinburgh, UK, Aug. 22, 2014.

[8] H. Cardoso da Silva, F. Pisani, and E. Borin, "A comparative study of SYCL, OpenCL, and OpenMP", in *International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, (Los Angeles, CA, USA), Oct. 26–28, 2016, pp. 61–66. DOI: 10.1109/SBAC-PADW.2016.19.

[3] https://github.com/KhronosGroup/SyclParallelSTL

[4] https://github.com/EPCCed/epcc-openacc-benchmarks