Solving Maxwell's Equations with Modern C++ and SYCL: A Case Study

Ayesha Afzal[1], Christian Schmitt[1], Samer Alhaddad[2], Yevgen Grynko[2],
Jürgen Teich[1], Jens Förstner[2], and Frank Hannig[1]
[1] Hardware/Software Co-Design, Friedrich-Alexander University Erlangen-Nürnberg
[2] Department of Theoretical Electrical Engineering, Paderborn University

***Abstract*—In scientific computing, unstructured meshes are a crucial foundation for the simulation of real-world physical phenomena. Compared to regular grids, they allow resembling the computational domain with a much higher accuracy, which in turn leads to more efficient computations.**

There exists a wealth of supporting libraries and frameworks that aid programmers with the implementation of applications working on such grids, each built on top of existing parallelization technologies. However, many approaches require the programmer to introduce a different programming paradigm into their application or provide different variants of the code. SYCL is a new programming standard providing a remedy to this dilemma by building on standard C++17 with its so-called single-source approach: Programmers write standard C++ code and expose parallelism using C++17 keywords. The application is then transformed into a concrete implementation by the SYCL implementation. By encapsulating the OpenCL ecosystem, different SYCL implementations enable not only the programming of CPUs but also of heterogeneous platforms such as GPUs or other devices. For the first time, this paper showcases a SYCL-based solver for the nodal Discontinuous Galerkin method for Maxwell's equations on unstructured meshes. We compare our solution to a previous C-based implementation with respect to programmability and performance on heterogeneous platforms.

I. Introduction

The efficient implementation of unstructured mesh algorithms is challenging, and the nodal discontinuous Galerkin time domain (NDGTD) [13] is a fundamental and ubiquitous method devised to obtain a numerical solution of partial differential equations for unstructured grids in the field of science and engineering. The NDGTD method has gained increasing popularity due to high order accuracy, dealing with complicated geometries using unstructured meshes, and the ability to easily handle boundary conditions. Furthermore, it provides remarkable compactness and flexibility both regarding geometry and range of problems. Unstructured meshes have become common due to several advantages in faster grid generation without requiring much experience, more efficient computation due to the geometry-adapted discretization, and high potential for the parallelization due to efficient synchronization between nodes. The involved irregular data structures result in the efficiency of local memory access. Moreover, this scheme is local to the elements, which makes it highly suitable to be parallelized for solving large and complex problems. However, implementing such algorithms in a performance-portable fashion, i.e., being efficient on many different hardware architectures, is cumbersome.

Today's modern HPC architectures are increasingly heterogeneous, and thus require highly optimized and efficient application implementations. This fosters the algorithm developers to ponder on different parallel language implementations and possible optimizations for excellent performance portability. One plausible solution to deal with these challenges is to take advantage of a modern programming language such as C++17 in combination with appropriate abstractions to target heterogeneous and parallel hardware platforms at a much higher level of abstraction. In 2015, the Khronos Group introduced the programming paradigm SYCL to ease writing portable and efficient code. By abstracting from OpenCL [18], it promises to target a range of different devices, e.g. CPUs, GPUs, and other accelerators such as field-programmable gate arrays (FPGAs). One highlight of SYCL is its single-source approach, allowing the programmer merely to use one programming language and to not bother with boilerplate code as is the case of calling OpenCL kernels. For SYCL, the designated host language is modern C++, or to be more precise, C++17 and its corresponding Standard Template Library (STL).

The major contribution of this paper is the introduction of our approach to solving Maxwell's Equation by using a Discontinuous Galerkin time-domain solver implemented using C++17 and SYCL.

The remainder of this paper is structured as follows. We commence the discussion by introducing essential features, characteristics, and the description of how to apply the building blocks of the NDGTD algorithm, followed by an overview of our implementation for the solution of Maxwell's equations in Section IV. Section II-B briefly introduces the SYCL technology and its features that are of interest to HPC developers. After that, Section III is devoted to an overview of available relevant work in this domain. Section V contains comparisons of the programmability and performance metrics between different variants of the two implementations on heterogeneous multi-core CPU and general-purpose GPU platforms. Finally, we conclude in Section VI.

II. Theoretical Background

A. Maxwell's equations and nodal discontinuous Galerkin time domain (NDGTD)

The three-dimensional Maxwell's equations can be written as a form of the conservation law:

$$\mathbf{Q}(\mathbf{r})\frac{\partial}{\partial t}\mathbf{q}(\mathbf{r},t) + \nabla \cdot \mathbf{F}(\mathbf{q}) = 0 \tag{1}$$

The material constants are contained in the matrix $\mathbf{Q}(\mathbf{r})$. The electric and magnetic fields $\mathbf{E}$ and $\mathbf{H}$ are combined into a single six-dimensional vector $\mathbf{q}(\mathbf{r}, t)$. The flux vector $\mathbf{F}(\mathbf{q})$ is a function of $\mathbf{q}(\mathbf{r}, t)$.

In order to solve Maxwell's equations numerically, they have to be discretized in space and time. We divide the entire simulation domain Ω into K non-overlapped tetrahedrons and approximate a local polynomial solution $\mathbf{q}_h^k(\mathbf{r}, t)$ of order p in each tetrahedron or element separately (h denotes the approximation). For getting high convergence, we can increase the number of elements in the entire simulation domain by making their dimensions smaller (h-refinement). Another way for increasing the resolution is choosing a higher order p of the polynomial representation (p-refinement), and this gives a more significant number of nodes for each element, where the solution is presented.

The polynomial expression of the numerical solution in each element is given as a sum over all its nodes $\mathbf{r}_i$:

$$\mathbf{q}_h^k(\mathbf{r}, t) = \sum_{n=1}^{N} \hat{\mathbf{q}}_n^k(t)\psi_n(\mathbf{r}) = \sum_{i=1}^{N} \mathbf{q}_h^k(\mathbf{r}_i, t)L_i(\mathbf{r}) \qquad (2)$$

Here, $N = (p+1)(p+2)(p+3)/6$ is the number of nodes distributed on the element. $\psi_n(\mathbf{r})$ is a local polynomial basis, L_i is the Lagrange interpolating polynomial, and $\hat{\mathbf{q}}_n^k$ are the expansion coefficients. After multiplying the conservation law Equation (1) with $L_i(\mathbf{r})$, integrating over the local element and inserting the numerical flux through integration by parts $\mathbf{F}^*(\mathbf{q})$, we obtain:

$$\begin{aligned}\int_{\Omega^k} \Big(\mathbf{Q}(\mathbf{r})\frac{\partial}{\partial t}\mathbf{q}(\mathbf{r}, t) + \nabla \cdot \mathbf{F}(\mathbf{q})\Big)L_i(\mathbf{r})d^3r \\ = \oint_{\partial\Omega^k} \hat{\mathbf{n}} \cdot \Big(\mathbf{F}(\mathbf{q}) - \mathbf{F}^*(\mathbf{q})\Big)L_i(\mathbf{r})d^2r. \qquad (3)\end{aligned}$$

After mathematical derivation [13, 9], we obtain the semi-discrete Maxwell's equations:

$$\epsilon^k \frac{\partial \mathbf{E}^k}{\partial t} = \mathbf{D}^k \times \mathbf{H}^k + (\mathscr{M}^k)^{-1}\mathscr{F}^k \left(\frac{\Delta\mathbf{E} - \hat{n}\cdot(\hat{n}\cdot\Delta\mathbf{E}) + Z^+\hat{n}\times\Delta\mathbf{H}}{\overline{Z}}\right) \quad (4)$$

$$\mu^k \frac{\partial \mathbf{H}^k}{\partial t} = -\mathbf{D}^k \times \mathbf{E}^k + (\mathscr{M}^k)^{-1}\mathscr{F}^k \left(\frac{\Delta\mathbf{H} - \hat{n}\cdot(\hat{n}\cdot\Delta\mathbf{H}) - Y^+\hat{n}\times\Delta\mathbf{E}}{\overline{Y}}\right) \quad (5)$$

Here, we define the spatial differentiation matrix $\mathbf{D}^k$, the mass matrix $\mathscr{M}^k$, the face matrix $\mathscr{F}^k$, and the outwardly pointing normal vector to the tetrahedron surface $\hat{n}$. The impedance $Z^{\pm}$ and the conductance $Y^{\pm}$ define the material parameters of the mesh element and the neighbor one. The sums of them are $\overline{Z} = Z^+ + Z^-$ and $\overline{Y} = Y^+ + Y^-$. We use a time-explicit method to integrate over the time. The explicit low storage Runge-Kutta method (LSRK) [6, 13] turns out to be a good choice for our system in Equations (4) and (5). The LSRK is memory efficient compared to implicit methods, but we need to choose a relatively small time due to the Courant condition in order to get accuracy and numerical stability. The shortest distance between two DG-nodes determines our time-step Δt. With each iteration, the fields are calculated by Maxwell's equations. The ordinary differential equations we have in our problem take the form:

$$\frac{\partial}{\partial t}\mathbf{y}(t) = \mathbf{g}(t, \mathbf{y}).$$

The scheme of the LSRK is:

$$\begin{aligned}&\mathbf{K}_1 = \mathbf{y_n}, \\ &\left.\begin{aligned}&\mathbf{K}_2 = A_i\mathbf{K}_2 + \Delta t\mathbf{g}(t_n + c_i\Delta t, \mathbf{K}_1) \\ &\mathbf{K}_1 = \mathbf{K}_1 + B_i\mathbf{K}_2,\end{aligned}\right\} \forall i = 1...s \\ &\mathbf{y}_{n+1} = \mathbf{K}_1. \qquad (6)\end{aligned}$$

where s is the number of stages.

B. SYCL

Based on OpenCL, SYCL[1] is a programming interface designed by the Khronos Group to simplify the implementation of parallel codes for heterogeneous platforms in C++. It bridges the programmability gap between C++ and OpenCL, by providing a single-source approach, i.e., both host and device code use the same programming language and are contained in the same source file. However, due to OpenCL limitations, only a subset of C++ functionality may be used. Problematic language aspects include virtual functions, function pointers, exception handling, and run-time type information. The SYCL execution model lets the runtime system automatically manage the underlying resources and data movement between the host and the device through accessors.

Kernels are the computationally intensive code parts, and depending on their size and characteristics, application developers offload them in SYCL by using advanced C++ features such as lambda functions, OpenCL C strings, program objects, and functors. It provides a type-safe and portable programming environment to allow programs to construct a cross-platform asynchronous task graph. These capabilities make SYCL easy to integrate into other C++ libraries and middleware development. As SYCL is embedded into modern C++17, it makes use of Parallel STL algorithms by passing execution policies to them.

To implement custom parallel operations, programmers need to expose parallelism by using the C++ construct `parallel_for`. For invoking a kernel, in addition to single task and standard `parallel_for` loop, SYCL supports hierarchical parallelism by allowing work to split into work groups and work items. However, it is not necessary to re-write existing OpenCL application to make use of SYCL. Rather, the OpenCL interoperability layer may be used, meaning existing OpenCL kernel implementations may be called inside SYCL `parallel_for` statements and may automatically take advantage of the new task scheduling capabilities. To implement SYCL support for computing devices that require special code structures, SPIR-V [17]—short for Standard Portable Intermediate Representation—may be used. It is a standardized, cross-API intermediate language to represent parallel computations designed to allow the compilation chain to be split across multiple products and vendors.

As SYCL is merely a language specification, different implementations are available. Notably, there are ComputeCPP [7] by

[1]http://www.khronos.org/sycl/

Codeplay, a commercial implementation of SYCL 1.2 that offers support for AMD and Intel CPUs and GPUs. An open-source implementation also targeting CPUs is triSYCL [16]. Support for non-single-source OpenCL interoperability is partly done, and support for SPIR and SPIR-V is in development. For the CPU parallelization back end, OpenMP is used. triSYCL's development is mainly funded by FPGA vendor Xilinx, hinting at the future support of such devices via SPIR. Another open-source SYCL implementation is sycl-gtx [29], originating from the University of Ljubljana. It targets OpenCL 1.2 compatible devices, e.g., CPUs and GPUs.

III. Related work

The solution of Maxwell's equations for unstructured meshes using the NDGTD is applied in various scientific fields for large-scale simulations. The review [5] describes how to exploit the NDGTD in nanophotonics with modeling sources, total-field/scattered-field technique, anisotropic material, absorbing boundary conditions (PML), dispersive media and how to deal with curvilinear elements. An example of the light scattering properties, by randomly irregular particles with wavelength-scale surface roughness, is given in [10]. It shows results consistent with the laboratory measurements of real samples. Furthermore, the results demonstrated by the NDGTD are very close to experiments regarding reproducing the second harmonic generation [9, 19] from Split-Ring resonator arrays based on their nonlinear optical response and from plasmonic gap antennas, which have strong resonant properties. Examples from the area of the electromagnetics where the NDGTD is used to know how a sent plane wave is scattered by an aircraft, and to understand the exposure of the head tissues radiated by a local source are given in [8]. In [15], the authors describe their work on porting and optimizing the very same application as presented in this manuscript to FPGAs.

a) Frameworks: In the past, a large variety of libraries for high-performance computing are developed such as `hypre` [1], `DUNE` [3], `FEniCS` [20], and `PETSc` [2]. The `hypre` software package is a collection of pre-conditioners and solvers. One part of the highly scalable hypre for unstructured grids and general matrices is the `BoomerAMG` [12]. `DUNE` is a general software framework for the solution of PDEs. However, both `hypre` and `DUNE` do not provide support for GPU accelerators. The finite element method library `FEniCS` uses a Python-embedded DSL, called Unified Form Language (UFL), and its `FEniCS` Form Compiler (FFC) develops a lot of routines and data structures while focusing on the finite element method for the solution of differential equations. `PETSc` is a suite of data structures and routines for C, C++, Fortran, and Python, which provides shared-memory and distributed-memory parallelization via pthreads and MPI, as well as supports GPU accelerators.

In addition to that, there is a finite element `DUNE` lattice interface based `Kaskade` [28] toolbox. It has been developed for solving of practical PDE problems of up to three-dimensional space in an object-oriented C++ programming language for a verity of fields. However, for the sake of simplicity, its usability is deliberately limited to very large problems with avoiding the parallelization for the distributed memory.

b) DSLs: Formerly, several high-level languages, tools, and frameworks have been developed that use domain-specific abstractions to shield the programmer from low-level details in diverse application domains. The noteworthy examples for DSLs include `ExaSlang` [23, 24] and `Julia` [4] with corresponding compilers customized towards the description of mesh-based PDEs solvers. `ExaStencils` uses Geometric Multigrid on Scala-based compiler with the language called `ExaSlang` and `Julia` builds on a just-in-time (JIT) compiler. It provides an interface to calling Python and C functions, and deals with the multiple dispatch features for the distributed parallel execution. Further, `Firedrake` [22] is an automated tool-chain in a domain-specific Python embedded language that uses `PETSc`.

IV. Maxwell's Equations Solvers

In this section, we introduce the algorithmic structure, and C and SYCL programming languages based implementations applied to unstructured meshes to highlight foremost aspects of Maxwell's equations solvers for scientific simulations. For this particular case study, the time domain solution of Maxwell's equations for three-dimensional systems with nodal discontinuous Galerkin (NDG) method is abbreviated as C-based Maxwell's (CBM) solver[2] by Warburton [13] and SYCL-based Maxwell's (SBM) solver.

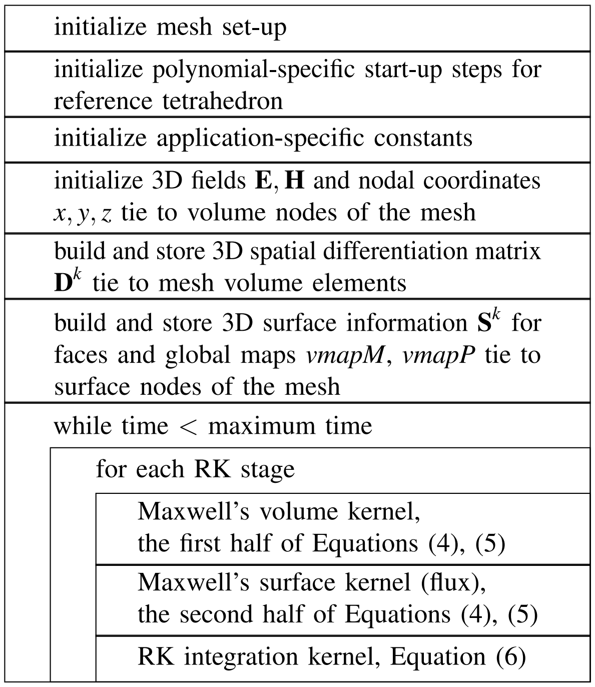
A. Structure

This section of simulation structure is added to get an idea of the entire flow of Maxwell's solvers. Our case study of Maxwell's equations solvers incorporate a lot of initialization steps, which include a contribution from the mesh set-up steps, the polynomial-specific start-up processes for reference tetrahedron, the initial conditions applied to the electromagnetic fields, and the application constants.

The start-up steps load the polynomial order based data for the reference tetrahedron, which is straight-sided. This polynomial-specific data includes the local reference coordinates $\mathbf{r}$ with their respective derivatives, the LIFT matrix $(\mathscr{M}^k)^{-1}\mathscr{F}^k$, and the local index mapping data structure for indirect nodal accesses through indices. This local index mapping is required in NDG, as nodal points lying on an edge belong to adjacent faces. All coordinates of the mesh elements must be transformed into this reference element in order to apply all operations. After that, the tetrahedrons are back-transformed into their origin and global mapping *vmapM*, *vmapP* is incorporated. Moreover, the initialization of problem-specific constants leads to the loading of Runge-Kutta coefficients A, B, polynomial order p, final time t, time-step size Δt, and nodal tolerance, etc. for the NDG algorithm.

The basic structure of Maxwell's equations solvers is outlined in Figure 1. The time-step processes include the explicit time discretization of the system of computationally intensive Maxwell's surface and volume equations using Runge-Kutta integration.

[2]https://github.com/tcew/MIDG2



```
initialize mesh set-up
initialize polynomial-specific start-up steps for reference tetrahedron
initialize application-specific constants
initialize 3D fields $\mathbf{E}, \mathbf{H}$ and nodal coordinates $x, y, z$ tie to volume nodes of the mesh
build and store 3D spatial differentiation matrix $\mathbf{D}^k$ tie to mesh volume elements
build and store 3D surface information $\mathbf{S}^k$ for faces and global maps vmapM, vmapP tie to surface nodes of the mesh
while time < maximum time
    for each RK stage
        Maxwell's volume kernel, the first half of Equations (4), (5)
        Maxwell's surface kernel (flux), the second half of Equations (4), (5)
        RK integration kernel, Equation (6)
```

Fig. 1. Basic program structure of Maxwell's equations solvers

B. *Implementation of C-based Maxwell's (CBM) solver*

The provided application uses the open source `Tetgen` [25] tool as unstructured mesh generator and the external `ParMETIS` [14] library for the partitioning of the grid. For all development processes, the CBM implementation initializes data using double C arrays in the source code.

The implementation creates the mesh data structure and utilizes separate storage of data for the volume elements, the volume nodes and the surface nodal points of the mesh. In order to introduce this split data storage in the preprocessing, the surfaces or faces of tetrahedrons are defined inside the structure. Therefore, we can make extra calculations for the nodes laying on each surface of the tetrahedron and make operations on the subset of the available number of faces. After initialization processes, the construction of three-dimensional spatial differentiation matrix $\mathbf{D}^k$ (i.e., the geometrical coefficients) and the surface information $\mathbf{S}^k$ (i.e., the normal coordinates with their direction $\hat{n}$ and the edge length) has been performed.

For every two neighboring nodes lying on adjacent faces of vicinal elements, there exist same coordinates $\mathbf{r}$ in space which can take different field values. Therefore, the three-dimensional electromagnetic fields differences $\Delta\mathbf{E}$ and $\Delta\mathbf{H}$ between every two neighboring surface nodes have to be computed, by exploiting those above two global index maps *vmapM*, *vmapP*. In order to address lack of local solution uniqueness, the numerical flux $\mathbf{F}^*$ is calculated through the combination of the electromagnetic field differences $\Delta\mathbf{E}$, $\Delta\mathbf{H}$ and the surface information $\mathbf{S}^k$. This numerical flux $\mathbf{F}^*$ together with LIFT matrix $(\mathscr{M}^k)^{-1}\mathscr{F}^k$ computes the second half of the time discrete right-hand side of Maxwell's Equations (4) and (5) on each surface node, called Maxwell's surface kernel.

However, the time-step process for Maxwell's volume kernel computes first half of time discrete right-hand side of Maxwell's Equations (4) and (5) inside each volume tetrahedron element. This is performed by utilizing the spatial differentiation matrix $\mathbf{D}^k$ and the initial values defined for electromagnetic fields $\mathbf{E}$ and $\mathbf{H}$. Thus, their sum forms the arithmetic-intensive system of ordinary differential Equations (4) and (5). After that, for each time-step size Δt, the computed right-hand side of Maxwell's equations, initial fields values $\mathbf{E}, \mathbf{H}$, and Runge-Kutta coefficients A, B lead to the updated local numerical solution for fields. This is done via explicit time integration using a fourth order, five stages Runge-Kutta scheme as described in Equation (6).

C. *Implementation of SYCL-based Maxwell's (SBM) solver*

The C-based Maxwell's (CBM) solver has been ported to C++17 and SYCL and uses high-level abstractions in terms of specialized data structures, operators and other features exploiting modern C++ features.

1) Modern C++ features: This section presents a case study of the time domain solution for three-dimensional Maxwell's equations, while establishing high-level abstractions and introducing modern C++ programming language features, transformations, and optimizations. Therefore, as a consequence, we hind away the computations in function calls and macros, which makes it more semantic, structured and easy to understand with the cleaner division between application initialization steps and the infrastructure.

In the following, we introduce the high-level C++ programming features and the semantic data structures of appropriate operators to get convenient access to the computations in the SBM solver. This abstracted implementation is later used as a good test bed for applying new and state-of-the-art approaches such as the SYCL programming paradigm.

a) Custom data types: The tetrahedron volume element accesses the three-dimensional coordinates of the vertices on a local tetrahedron element. To do that, we use custom data types that represent with vectors and matrices without the need for index accesses. Consequently, we can access individual components of vectors, and on the other hand, we can also perform computations on all three components at a time, which provides a new possibility for vectorization. Furthermore, the utilization of three-dimensional functions, for example, flux, curl, Euclidean norm, dot and cross products, and determinant, etc., in larger expressions of Maxwell's equations makes the implementation intuitive for the programmer, and enables compiler-applied optimizations such as vectorization.

To represent faces of a tetrahedron, we introduced another custom data type, which greatly simplifies the initial mesh set-up by providing appropriate comparison operators. Further, some memory is saved by storing of normal vectors $\mathbf{n}$ once for each face instead of storing it for all the surface nodal points.

b) Kernel fusion: In our SYCL-based Maxwell's (SBM) solver, kernel fusion is applied. The fusion of Maxwell's kernels can be easily performed since the computations on volume nodes are independent of the surface nodes. As a consequence, it reduces extra buffer creation to store intermediate results of surface and volume kernels, and also reduces extra global memory accesses of type read or write. However, there is a dependency between Maxwell's kernels and Runge-Kutta integration kernel as described previously in Section IV-B. Maxwell's kernel takes initial values of electromagnetic fields, whereas the RK kernel updates the field values. Thus, in order to fuse both Maxwell's and RK kernels, we use a double buffering technique to swap between initial and updated values of fields for each RK stage.

2) SYCL features: SYCL introduces the concept of queues, buffers, accessors and the command groups, which follows significant code restructuring without requiring any significant code additions.

For data storage, we set up a number of buffer objects with corresponding sizes and types. Inside the queue scope, we declare some accessors to make this data available to the kernels. In case memory transfers are necessary, they will be handled automatically by SYCL. However, the storage of data is separated from the data accesses in SYCL.

Listing 1. Command group handler with the definition of accessors and parallel for construct

```
buffer<real3_t, 2> b_E(range<2>{elems, 3});
 [...]
queue.submit([&] (handler &cgh) {
auto a_E = b_E.get_access<access::mode::read>(
    cgh);
 [...]
cgh.parallel_for_work_group<class
    fused_kernels>(nd_range<>(range<>(elems),
    range<>(grpsize)), [=](group<> grp){
   grp.parallel_for_work_item([=](nd_item<1> k
       ){
     [...]
});});});
```

An abstraction in SYCL is the introduction of the command group `cgh` concept via the `handler` object (see Listing 1) compare to the respective OpenCL platform paradigm. The command group includes all the necessary OpenCL commands required to execute host data correctly, and this group of commands is then submitted to the queue. The `handler` object provides a link between the queue and a whole group of commands. SYCL-specific accessors are always declared inside the command group construct and are templated by the mode of access as read, write or both to provide a fine control on complex memory hierarchy. Before the execution of kernels, the accessors implicitly build up a high-level task graph with all the data dependencies captured by the SYCL run-time to determine a correct and efficient schedule. It can be fully asynchronous scheduled to execute kernels independently of the host code until a programmer particularly requests access to the host data. On the other hand, SYCL also allows the user to create customized execution environments to handle all the resources manually. However, we are keeping our SBM solver simpler when implementing it in an implicitly parallel way and letting the SYCL implementation do it automatically.

In this SBM solver, plain "for loops" of kernels are invoked inside command groups using a `parallel_for` construct to enqueue a kernel. This `parallel_for` construct makes the language implicitly parallel and takes two arguments. The first argument is the kernel execution range of `parallel_for` loop that implicitly goes over all mesh elements. The second is the execution of kernel defined in a C++-friendly lambda expression. Kernels are also given a name defined as a class. We do not need to provide any concrete memory layout or indexing anymore and it is managed by SYCL that how to parallelize or schedule more efficiently. We pass an `index` object in the kernel parameters, which provides all the indices information around the current work-item organized in a one-dimensional iteration space to access the mesh elements. These extracted index coordinates provide the flexibility to the user with adding an easy write to the accessors and on the other hand, also allow to get data access to other fields.

Besides this standard level of parallelism, SYCL enables a new addition to the OpenCL model via "hierarchical parallel for" construct. We also applied the hierarchical parallelism for invoking all three Maxwell's and Runge-Kutta kernels using `parallel_for_work_group` and `parallel_for_work_item` constructs. It introduces the concepts of work group and work item as in OpenCL, however, frees the programmer from painful work-group synchronization and provides the coarse grain work group level of parallelism for better mapping to CPU-like architectures.

SYCL also provides a specific OpenCL interoperability mode for the heterogeneous computing backend. For this, we built higher-level SYCL kernels from plain OpenCL kernels with help of Boost.Compute interoperability mode of triSYCL implementation. The equivalent OpenCL objects can be retrieved from the SYCL objects at any point in a SYCL application. Inside the queue, `cgh.set_args()` allows a direct synergy of higher level C++ single-source SYCL realm with the OpenCL world to use existing libraries with no overhead. The SYCL kernel arguments are set with a single variadic function call and relieve the programmer from managing the explicit memory transfers between host and devices.

V. Evaluation

In the following, we evaluate those above mentioned SYCL-based solvers and compare them to the original implementation [13] for a variant of three-dimensional unstructured grids on CPU and GPU architectures.

First of all, we provide some details on the experimental set-up and methodology, which are necessary to cover both relevant cases. After that, besides an in-depth analysis of the programmer efficiency, we also discuss the convergence, software complexity and run-time performance for each of the presented solvers.

A. Experimental set-up and methodology

We compare the existing C-based Maxwell's (CBM) solver with our new SYCL-based implementations. As SYCL implementation, we selected triSYCL [16]. Both variants of the solver's implementation are evaluated to automate the process for a flexible number of cores and applied to a variety of unstructured grids with different mesh sizes, such as 72, 1052, 1978, 5235 and 10 420 mesh elements. They are also analyzed for various polynomial order p, e.g., linear, quadratic, cubic, quartic, quintic, sextic, septic, and so on.

For evaluation and comparison, run-time measurements are taken using a high-resolution timer. Further, we use open source

C and C++ Code Counter `CCCC` [26] tool for the programming complexity analysis of original CBM and abstracted SBM solvers. It allows the measurement of software product metrics, such as, number of modules (NOM), lines of code (LoC) and McCabe's cyclomatic complexity, etc., and generates a report organized as `HTML` document. In addition to that, Halstead's complexity metric is also computed statically from both variants of code.

To test both implemented solvers, we use two Intel CPUs, i.e., Intel Xeon E5-2630 v2 at 2.60 GHz frequency and Intel Skylake i7-6700 at a clock speed of 3.4 GHz and an NVIDIA GeForce GTX 745 GPU. Intel Xeon E5-2630 v2 has a microarchitecture of Ivy Bridge-EP (12 logical cores per socket), which exhibits a theoretical maximum achievable single core double precision (DP) performance of 20.8 giga floating point operations per second (GFLOPS). However, Intel Skylake i7-6700 (8 logical cores per socket) can perform 16 double precision (DP) floating point operations (FLOPs) per cycle, and thus theoretically has a 54.4 GFLOPS performance per core. As the platform compiler, we select CLANG 5.0.1.

B. Convergence

We verified the correctness of SBM implementations by comparing resulted convergence rates. To this end, a local updated field value of Maxwell's solvers (i.e., y component of electric field E_y) is tested against the given exact analytical E_y value (Figure 2). Depending on a chosen polynomial order p, additional unstructured nodal grid points N, suitable for interpolation, can be carefully created to ensure a high-order accuracy of Maxwell's solvers. To ensure that interpolations on nodal tetrahedra *well behave* for Maxwell's solvers, we imply minimizing an approximation to the Lebesgue constant, while using non-equidistant construction for optimal Legendre-Gauss-Lobatto (LGL) points on the reference tetrahedron with a Wrap and Blend method [27].

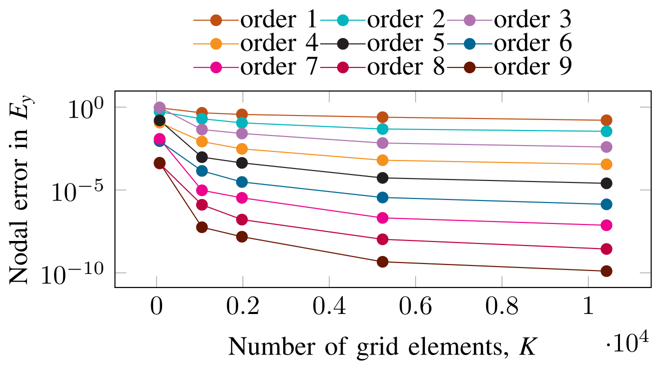


Fig. 2. Consequences of different polynomial order p (p refinement) and grid elements K (h refinement) in terms of convergence for Maxwell's equations solvers at the final time t of 0.75 seconds

C. Software complexity metrics

In the following, we investigate the development efforts for both code variants, i.e., CBM and abstracted SBM solver, using quantitative software metrics. These metrics, in particular, measure the program complexity, and at the same time, development efforts and cost for parallelizing, tuning and porting in software engineering.

a) Lines of code (LoC): LoC is one of the most widely used and easily quantifiable metrics. To get suitable and comparable results, we consider only LoC excluding empty lines, comments, and certain statements like header inclusions. The original CBM solver presented in Section IV-B has 1443 LoC both for host C code and for all three kernels, whereas the SBM solver takes only 323 LoC. This reduction in application complexity is due to a series of simplifications provided by SYCL, such as the reduced need for boilerplate code, custom data types and consequent STL implementation, and in-line kernels.

However, the LoC metric does not take into account different complexity of code lines. Thus, it is only a rough indicator of real effort.

b) Halstead complexity metric: The Halstead complexity metric [11] focuses on data streams by using fine-grained operands and operators and model a quantitative measure of the program module's length and the programming effort directly from source code to compare high-level parallel programming. It is based on interpreting the source code as a sequence of unique and total number of operators and operands (n_1, n_2, N_1, N_2), respectively. The identifier, type name, type specifier and constant are all counted as operands, whereas, storage class specifiers, type qualifiers and reserved words are all treated as operators. It is useful during development to assess code quality of Maxwell's equations solvers in both C and SYCL languages. For this purpose, we investigate following four Halstead's measures to compare both CBM and SBM solvers.

- The Halstead's length ($N = N_1 + N_2$) is the sum of the total number of operators and operands in the program.
- The Halstead's volume ($V = N{*}log_2(n_1{+}n_2)$) describes the size of the implementation of an algorithm, and therefore, is less sensitive to code layout than the LoC measures.
- The Halstead's level ($L = (2/n_1) * (n_2/N_2)$) defined as the inverse of the error-proneness or difficulty level D to write or understand a program.
- The Halstead's effort to implement ($E = V/L$) or understand a program is proportional the volume V and to the difficulty level D of the program.

TABLE I
PROGRAMMING COMPLEXITY METRICS FOR CBM AND SBM SOLVERS

Complexity metrics	CBM solver	SBM solver	CBM kernels	SBM kernels
LoC	1 443	323	191	95
M	210	40	19	11
N	26 807	4 840	1 918	1 437
V	80 094	40 148	13 455	10 107
L	0.008938	0.009654	0.011182	0.012582
E	8 960 266	4 158 392	1 203 245	803 318

Table I compares both approaches in terms of data streams of operands and operators for all four Halstead's measures.

A low Halstead's length and volume quantitatively indicate a smaller size of implementation for the SBM solver. Further, the high Halstead's level of the SBM solver showcases that it tends to contain fewer errors compared to the C-based Maxwell's (CBM) solver, which uses the same operands many times in the program, and is more prone to errors. The smaller Halstead's effort quantitatively indicates that this abstracted SYCL case study is easier to implement and understand.

c) McCabe's cyclomatic complexity metric: The cyclomatic complexity metric [21] provides a quantitative measurement based on the control flow graph of the program as:

$$M = \text{edges count} - \text{nodes count} + 2 * \text{connected parts count}$$

This applies that the McCabe's cyclomatic complexity number M would be one if the source code has only a single path and does not contain any control flow statements.

The higher M number for C-based Maxwell's (CBM) solver (Table I) quantitatively showcases the complexity of this original code compared to our new SBM solver. It measures the higher number of test cases that would have to be written to execute all paths in a CBM solver. Evidently, the implementation of our SBM solver has a comparatively very low number of basic paths, which makes it easy to understand, and therefore, has a lower probability of containing errors.

D. Run-time performance

The abstracted SYCL-based implementation is tested using triSYCL [16], which under the hood does parallelization via OpenMP. Results are compared for both the original C-based Maxwell's solver and the SYCL implementations. The run-time comparison for both solvers on an NVIDIA GPU and two different Intel x86 execution platforms is shown in Figures 3 and 4. The handling of data and memory in the SYCL port is greatly simplified by using the SYCL-provided data structures. However, in case of using triSYCL as the SYCL implementation, these abstractions also come with a price, as memory accesses need to go through a hierarchy of function calls and to wrap data structures until the real memory location is revealed and used.

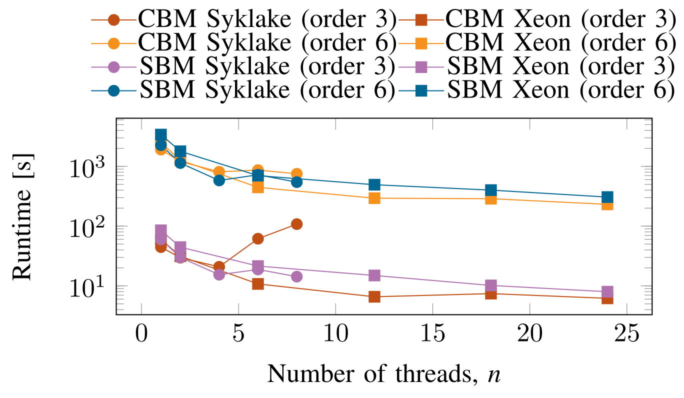


Fig. 3. Run-time comparison of both Maxwell's solvers for cubic and sextic polynomial orders with 1978 mesh elements on both Intel Syklake i7-6700 CPU and Intel Xeon E5-2630 v2 CPU platforms

The original CBM solver uses external `ParMETIS` library for mesh partitioning which results in better performance. Additionally, the CBM solver has been implemented using MPI, i.e., a very explicit communication scheme where programmers need to handle data transfers between the different processes themselves. Current SYCL implementations do not provide distributed-memory parallelism, so no explicit communication statements and no data transfers can be specified, greatly reducing the source of potential errors, but, of course, also restricting the application to single nodes. While theoretically it would be possible to implement a hybrid parallism concept such as MPI+SYCL, this would violate the ideas and concepts behind SYCL. Rather, a better route is to employ a SYCL implementation that internally handles data transfers and exposes a partitional global address space (PGAS) to the user. Basically, SYCL already provides a task graph model from which data dependencies could be derived and resolved accordingly.

As a consequence of the removal of MPI, all references to `ParMETIS` and mesh partitioning have also been removed from the SYCL port, meaning that adjacent mesh elements are no longer stored at nearby memory locations. Since the memory layout of all important data structures, e.g., of the electromagnetic fields, follow the memory layout of the mesh, this results in memory accesses with potentially larger distances between each other, effectively reducing the number of cache hits and thus performance. The reason for removal of all references to MPI and mesh partitioning `ParMETIS` library is that we are planning to introduce SYCL-matching programming model, i.e., task-based like GPI for distributed-memory parallelization.

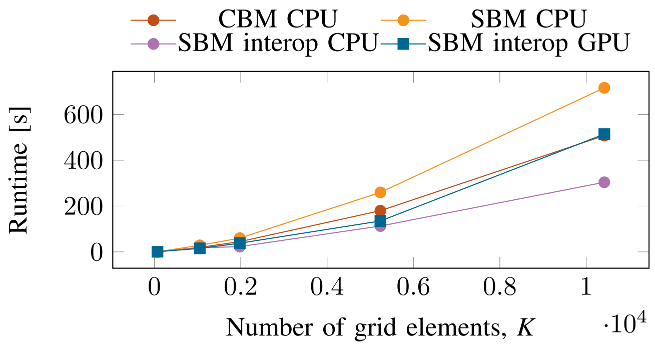


Fig. 4. Run-time comparison of different implementations of SBM solvers with CBM solver for cubic polynomial and K grid elements on heterogeneous Intel Syklake i7-6700 CPU and NVIDIA GeForce GTX 745 GPU platforms

Our finding for these Maxwell's solvers is that the interoperability mode of the open-source triSYCL performs better in comparison with other mentioned solvers for both heterogeneous CPU and GPU platforms (Figure 4). Compared to pure SYCL version, SYCL kernels are built from OpenCL kernels for this interoperability mode. However, data storage and accesses still done via SYCL buffers and accessors and OpenCL kernels are still scheduled asynchronously by the SYCL run-time according to the data parallel task graph model. The SBM interoperability version outperforms the CBM with 49 % decrease in run-time for order 3 and 1978 mesh elements. The reason for better performance of the interoperability mode is that it encounters less overhead for the memory accesses compare to pure triSYCL implemenation which is quite immature at this stage. Further, the multi-core CPU version can handle the unoptimized random data accesses a way better than general-purpose GPU version. Moreover, with switching on the hierarchical level of parallelism for different work group sizes, we observe some level of performance portability for our triSYCL implementation. The results show that our new SYCL-based case study incorporates a compact algorithmic description and, thus, is less error-prone. Furthermore, it has a certain potential for performance portability with satisfactory execution speed for the unstructured meshes.

VI. Conclusion

In this paper, we have presented a stable, accurate and efficient implementation to solve Maxwell's equations on an unstructured mesh by using modern C++ features and SYCL, an abstraction layer for single-source parallel programming of heterogeneous devices built on top of the concepts and cross-platform portability of OpenCL. Although SYCL status is still in the phase of gathering feedback from C++ community, this case study showcases that how the SYCL-based modern C++ implementation frees the developer from writing complicated parallel codes, achieves user-friendly behavior, high productivity, and throughput of algorithmic development for the domain of unstructured grids on latest heterogeneous platforms.

Acknowledgments

This work has been supported by the German Federal Ministry for Eduction and Research (BMBF) within the collaborative research project "HighPerMeshes" (011H|160054).

References

[1] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. Yang. Scaling hypre's multigrid solvers to 100,000 cores. In *High-Performance Scientific Computing: Algorithms and Applications*, pages 261–279. Springer, 2012. DOI: 10.1007/978-1-4471-2437-5_13.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern Software Tools for Scientific Computing*, pages 163–202. Birkhäuser, 1997. DOI: 10.1007/978-1-4612-1986-6_8.

[3] P. Bastian et al. A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. *Computing*, 82(2):103–119, 2008.

[4] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *The Computing Research Repository (CoRR)*, Sept. 24, 2012.

[5] K. Busch, M. König, and J. Niegemann. Discontinuous Galerkin methods in nanophotonics. *Laser & Photonics Reviews*, 5(6):773–809, 2011.

[6] M. H. Carpenter and C. A. Kennedy. Fourth-order 2N-storage Runge-Kutta schemes. Technical report NASA-TM-109112, NASA Langley Research Center, Hampton, VA, USA, June 1994.

[7] ComputeCpp. ComputeCpp – Accelerate complex C++ applications on heterogeneous compute systems using open standards. Mar. 2017. URL: https://www.codeplay.com/products/computesuite/computecpp.

[8] V. Dolean, H. Fahs, L. Fezoui, and S. Lanteri. Locally implicit discontinuous Galerkin method for time domain electromagnetics. *Journal of Computational Physics*, 229(2):512–526, 2010.

[9] Y. Grynko and J. Förstner. Simulation of second harmonic generation from photonic nanostructures using the discontinuous Galerkin time domain method. In *Recent Trends in Computational Photonics*, pages 261–284. Springer, 2017.

[10] Y. Grynko, Y. Shkuratov, and J. Förstner. Light scattering by irregular particles much larger than the wavelength with wavelength-scale surface roughness. *Optics Letters*, 41(15):3491–3494, 2016.

[11] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977. ISBN: 0-444-00205-7.

[12] V. E. Henson and U. M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002. DOI: 10.1016/S0168-9274(01)00115-5.

[13] J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: Algorithms, analysis, and applications*. Springer Science & Business Media, 2008. ISBN: 978-0-387-72065-4.

[14] G. Karypis, K. Schloegel, and V. Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Department of Computer Science, University of Minnesota*, 1997.

[15] T. Kenter et al. OpenCL-based FPGA design to accelerate the nodal Discontinuous Galerkin method for unstructured meshes. In *Proc. IEEE Int'l Symposium on Field-Programmable Custom Computing Machines*. ACM, Apr. 29–May 1, 2018.

[16] R. Keryell. triSYCL: An open source implementation of OpenCL SYCL from Khronos Group. Apr. 2014. URL: https://github.com/triSYCL/triSYCL.

[17] J. Kessenich, B. Ouriel, and R. Krisch. SPIR-V specification provisional (version 1.1, revision 3, unified). Jan. 2, 2018. URL: https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf.

[18] Khronos OpenCL Working Group. The OpenCL specification, version: 1.2, document revision: 19. Nov. 2012. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[19] S. Linden, F. B. Niesler, J. Förstner, Y. Grynko, T. Meier, and M. Wegener. Collective effects in second-harmonic generation from split-ring-resonator arrays. *Physical Review Letters*, 109(1):015502, 2012.

[20] A. Logg, K.-A. Mardal, and G. Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012. ISBN: 978-3-642-23098-1.

[21] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2(4):308–320, 1976.

[22] F. Rathgeber et al. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):24, 2017.

[23] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. ExaSlang: A domain-specific language for highly scalable multigrid solvers. In *Proc. Int'l Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 42–51. IEEE Computer Society, Nov. 17, 2014. DOI: 10.1109/WOLFHPC.2014.11.

[24] C. Schmitt et al. Systems of Partial Differential Equations in ExaSlang. In *Software for Exascale Computing – SPPEXA 2013 –2015*. Volume 113, Lecture Notes in Computational Science and Engineering. Springer, Aug. 2016. ISBN: 978-3-319-40526-1. DOI: 10.1007/978-3-319-40528-5.

[25] H. Si. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software*, 41(2):11, 2015.

[26] C. Thirumalai, P. A. Reddy, and Y. J. Kishore. Evaluating Software Metrics of Gaming Applications using Code Counter Tool for C and C++ (CCCC). In *IEEE Int'l Conf. on Electronics, Communication and Aerospace Technology*, volume 2, pages 180–184, 2017.

[27] T. Warburton. An explicit construction of interpolation nodes on the simplex. *Journal of engineering mathematics*, 56(3):247–262, 2006.

[28] M. Weiser, A. Schiela, and S. Götschel. Kaskade7 Finite Element Toolbox. URL: http://www.zib.de/projects/kaskade7-finite-element-toolbox.

[29] P. Žužek. ProGTX: Implementation of the SYCL specification. 2016. URL: https://github.com/ProGTX/sycl-gtx.