This is the author's version of the work. The definitive work was published in
Proceedings of 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, April 29–May 01, 2018.

OpenCL-based FPGA Design to Accelerate the Nodal Discontinuous Galerkin Method for Unstructured Meshes

Tobias Kenter*, Gopinath Mahale*, Samer Alhaddad†,
Yevgen Grynko†, Jens Förstner†, Christian Plessl*
*Paderborn Center for Parallel Computing and
Department of Computer Science
†Department of Electrical Engineering
Paderborn University
Email: firstname.lastname@uni-paderborn.de

Christian Schmitt, Ayesha Afzal,
Frank Hannig
Department of Computer Science
Friedrich-Alexander University Erlangen-Nürnberg
Email: {christian.j.schmitt, ayesha.afzal, hannig}@fau.de

***Abstract*—The exploration of FPGAs as accelerators for scientific simulations has so far mostly been focused on small kernels of methods working on regular data structures, for example in the form of stencil computations for finite difference methods. In computational sciences, often more advanced methods are employed that promise better stability, convergence, locality and scaling. Unstructured meshes are shown to be more effective and more accurate, compared to regular grids, in representing computation domains of various shapes. Using unstructured meshes, the discontinuous Galerkin method preserves the ability to perform explicit local update operations for simulations in the time domain.**

In this work, we investigate FPGAs as target platform for an implementation of the nodal discontinuous Galerkin method to find time-domain solutions of Maxwell's equations in an unstructured mesh. When maximizing data reuse and fitting constant coefficients into suitably partitioned on-chip memory, high computational intensity allows us to implement and feed wide data paths with hundreds of floating point operators. By decoupling off-chip memory accesses from the computations, high memory bandwidth can be sustained, even for the irregular access pattern required by parts of the application.

Using the Intel/Altera OpenCL SDK for FPGAs, we present different implementation variants for different polynomial orders of the method. In different phases of the algorithm, either computational or bandwidth limits of the Arria 10 platform are almost reached, thus outperforming a highly multithreaded CPU implementation by around 2x.

I. Introduction

It has been shown that FPGAs can deliver high performance and high efficiency as accelerators in many different application domains. With the advent of FPGA architectures that can efficiently realize floating-point arithmetic, using hardened DSP blocks in addition to logic resources, the domain of floating-point heavy scientific computations became an interesting target. However, the list of applications from this domain that have been considered for FPGA acceleration so far is limited in computational patterns and complexity. In the class of stencil computations that are for example used for finite-difference time-domain (FDTD) simulations, a number of high-throughput FPGA designs have been presented [1]–[4] that rely on deep pipelining to achieve efficiency through high data reuse. Other FPGA implementations focus on common operations like dense matrix-matrix multiplication (GEMM) [5], [6] and frequency transformations like fast Fourier transform (FFT) [7], [8]. The efficiency of FPGAs for these operations builds upon buffering and forwarding or shuffling local data exactly in the pattern required by the algorithm. The highly recognized FPGA-based Anton molecular dynamics accelerators [9], [10] build upon the efficiency of such simple FFT building blocks.

In this work, we investigate the potential of another method from computational sciences, which has not been considered for FPGA acceleration so far. The discontinuous Galerkin (DG) method is state-of-the-art in many different simulation areas, particularly because it can be applied to unstructured meshes that can well represent the shapes of simulated structures and because it exploits good locality. The locality is important for large-scale HPC implementations and can even be tuned by moving to higher order variants of the method. With our implementation of the nodal discontinuous Galerkin method for solving Maxwell's equations in the time domain on unstructured grids, we put not only a novel application class into the focus of FPGA acceleration, but we also highlight for this method aspects that drive the efficiency of FPGAs. We summarize our contributions as follows.

- We investigate for the first time the suitability of the discontinuous Galerkin method on unstructured meshes for FPGA acceleration.
- We present an OpenCL-based FPGA implementation that reaches high sustained performance for several design alternatives and outperforms a vectorized and highly multithreaded CPU implementation by 2x.
- We provide practical guidance on how to interact with the utilized HLS tools to generate such designs.

The main driving factors that allow FPGAs to achieve high efficiency for this method are threefold.

- High arithmetic intensity allows for very wide datapaths that do not require to change the high-level structure of the application.

- Customized constant memory removes pressure from off-chip memory interfaces and provides fully predictable performance.
- Decoupled prefetching allows sustained high performance even for irregular indirect memory accesses.

In the rest of the paper, we first introduce and analyze the method in more detail (Section II). Based on this analysis, we present selected design aspects of our OpenCL based implementation in Section III. Section IV combines results in absolute performance metrics and compared to a CPU reference. Finally, we take a look at related work and conclude.

II. Nodal Discontinuous Galerkin Method

The nodal discontinuous Galerkin method in time domain (DGTD) [11] is used to approximate solutions for systems of partial differential equations numerically. Thus, these equations can be iteratively solved on computers using linear algebra. In contrast to related approaches, the DG method recently gained more popularity for scientific simulations because of its flexibility and accuracy. In contrast to the FDTD method [12] the DGTD method does not use an orthogonal grid, thus avoiding aliasing or staircase effect. It deals with arbitrary shapes and complex geometries [13] exploiting unstructured meshes with variable resolution.

One of the most important advantages of this method is that its mathematical approach, which depends on local operations for each element, is suitable for parallelization, allowing to solve large-scale problems on different parallel hardware platforms. In contrast to the finite element method (FEM) [14], DG can avoid expensive computations of global matrices. The DG method has been utilized in many different fields such as compressible gas dynamics [11], elastodynamics [15], acoustics [16] and electrodynamics [11], [17], [18], which is also the topic of this work.

In the remainder of this section, we first lay out the mathematical foundations for applying the DG method to Maxwell's equations, then approach it from an algorithmic perspective with emphasis on the arithmetic intensity and conclude with a brief discussion of convergence.

A. Mathematical Foundations

The fundamental equations to be solved for electrodynamics are Maxwell's equations that can be written in conservation form compactly [17] as:

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

With $\mathbf{r}$ denoting the spatial domain, $\mathbf{Q}(\mathbf{r})$ a material matrix, the vector $\mathbf{q}(\mathbf{r},t)$, consisting of the electric and magnetic fields $\mathbf{E}$ and $\mathbf{H}$ respectively and with the flux $\mathbf{F}(\mathbf{q})$. For DG, the computational domain Ω is spatially discretized and approximated by a set of K elements, typically tetrahedrons. Within each element $k \in K$ the approximated solution $\mathbf{q}_h^k(\mathbf{r},t)$ (h denotes the approximation). $\mathbf{q}$ is locally calculated and represented in so-called nodal points n that belong exclusively to each element. Depending on a chosen polynomial order, a different number of nodes is created in each element. The local approximated solution within each k-element takes the form:

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

Where N is the number of the nodes in one element, $\psi_n(\mathbf{r})$ is a local polynomial basis with its expansion coefficients $\hat{\mathbf{q}}_n^k$. L_i is a Lagrange interpolating polynomial as used in our case.

In the *discontinuous* Galerkin method, the local solutions per element do not need to be continuous at the boundary to neighboring elements. For each nodal point on the surface of an element, there exists another point with the same coordinates $\mathbf{r}$ in space on the surface of the adjacent element, and can take different field values. In order to couple the individual local solutions, for each pair of nodes on opposing surfaces of neighboring nodes, the flux is calculated through the electric and magnetic field differences $\Delta\mathbf{E} = \mathbf{E}^+ - \mathbf{E}^-, \Delta\mathbf{H} = \mathbf{H}^+ - \mathbf{H}^-$. The minus "-" denotes the local element and the plus "+" the neighbor one. Based on this, a pair of ordinary differential equations (ODEs) has been introduced [11], [18] that combines the local and global field information of the semi-discrete system:

$$\epsilon^k \frac{\partial \mathbf{E}^k}{\partial t} = \mathbf{D}^k \times \mathbf{H}^k + (\mathscr{M}^k)^{-1}\mathscr{F}^k\left(\frac{\Delta\mathbf{E} - \hat{n}\cdot(\hat{n}\cdot\Delta\mathbf{E}) + Z^+\hat{n}\times\Delta\mathbf{H}}{\overline{Z}}\right) \tag{3}$$

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

They use the mass matrix $\mathscr{M}^k$, the face matrix $\mathscr{F}^k$, the spatial differentiation matrix $\mathbf{D}^k$, and the outwardly pointing normal vector to the element face $\hat{n}$, where the flux has been calculated. The impedance $Z^\pm$ and the conductance $Y^\pm$ characterize the materials of the elements and its neighbors and are constant in each element. Their sums are $\overline{Z} = Z^+ + Z^-$ and $\overline{Y} = Y^+ + Y^-$.

In order to solve this system of ordinary differential equations, the time discretization is essential. The explicit low storage Runge Kutta scheme (LSRK) [11], [19] is applied for time integration in our work. The time step depends on the mesh quality and must not to exceed a critical limit in order to preserve numerical stability. It is determined by the smallest distance between the DG-nodal points.

B. Algorithmic Description

The computation of local solutions inside each volume element k is performed in a function that we denote as *volume kernel* and outline in Algorithm 1. For each nodal point n, it computes the right-hand side values $\mathbf{rhsE}_{k,n}$ and $\mathbf{rhsH}_{k,n}$, based on the derivatives of the respective other field ($\mathbf{H}$ and $\mathbf{E}$) at all nodal points of element k. In this representation, we use the three-dimensional $\mathbf{E}$ and $\mathbf{H}$ fields instead of the common vector term $\mathbf{q}$ in order to illustrate the alternating dependencies between the two fields. The *volume kernel* essentially calculates the time discretized first summand of Equations (3) and (4). For these calculations, the impact of other nodes is weighted by coefficients for the underlying polynomial base functions $\mathbf{D}_{n,m}$ and by specific geometry

Input: Field values $\mathbf{E}_{k,n}$, $\mathbf{H}_{k,n}$, polynomial base coefficients $\mathbf{D}_{n,m}$, geometry coefficients $\mathbf{G}_k$
Output: Local update terms $\mathbf{rhsE}_{k,n}$, $\mathbf{rhsH}_{k,n}$

```
1  foreach element k ∈ 1..K do
       // Prefetch all nodes of k first
2      foreach node n ∈ 1..N do
3          ldE = 0
4          ldH = 0
5          foreach node m ∈ 1..N do
6              ldE += D_{n,m} ⊗ E_{k,m}
7              ldH += D_{n,m} ⊗ H_{k,m}
8          end
9          rhsE_{k,n} = G_k ⊗ ldH
10         rhsH_{k,n} = G_k ⊗ ldE
11     end
12 end
```

Algorithm 1: Volume kernel

coefficients $\mathbf{G}_k$ for the concrete volume element. There is parallelism among different volume elements, but also within the computation for each element. For each iteration of the k-loop (outer loop), there is potential for data reuse, with the field values of N nodes being used for a total of N^2 calculations through the nested loops in Lines 2 and 5. Each of the N^2 polynomial base coefficients $\mathbf{D}_{n,m}$ on the other hand is only used once per iteration of the k-loop. However, they are constant for all k elements. Only a few geometry-specific coefficients $\mathbf{G}_k$ are additionally required for each specific element.

Depending on the number of nodes N per element that is determined by the polynomial order p, we can calculate the number of arithmetic operations per iteration of the k-loop. With the operator $\otimes$ in Lines 6, 7 (Algorithm 1) denoting an all-to-all convolution operation on three-element vectors, 36 floating point operations are performed per iteration of the m-loop. The $\otimes$ operators in Lines 9, 10 involve two convolutions each and overall add 72 floating point operations per iteration of the n-loop. The *FLOPs* column in Table I summarizes these observations and provides concrete numbers for polynomial order $p = 3$ with $N = 20$ and $p = 4$ with $N = 35$. The next column summarizes the number of floating point operands that are unique per k-iteration and thus cause memory traffic even with perfect data reuse. The last column adds the amount of data that can be obtained by reuse of $\mathbf{E}$ and $\mathbf{H}$ or be served from constant memory ($\mathbf{D}$). The table illustrates, how the *volume kernel* obtains a high arithmetic intensity through the effective use of local memory. Calculating with 4-byte floats, we obtain over 15 operations/byte for $p = 3$ and almost 30 operations/byte for $p = 4$.

In order to connect the local solution inside an element with the rest of the simulated structure, another set of right-hand side values $\mathbf{rhsE}_{k,n}$ and $\mathbf{rhsH}_{k,n}$ is computed based on the fluxes $\Delta\mathbf{E}, \Delta\mathbf{H}$ between nodal values on the surfaces of the own element and corresponding nodal values on opposite surfaces of neighboring volume elements. We denote this step

TABLE I
ARITHMETIC INTENSITY OF VOLUME KERNEL.

	FLOPs	unique data (floats)		reused data (floats)	
	$N^2 \cdot 36$	$N \cdot 6$	($\mathbf{E}$, $\mathbf{H}$)	$N^2 \cdot 6$	($\mathbf{E}$, $\mathbf{H}$)
	$N \cdot 72$	9	($\mathbf{G}$)	$N^2 \cdot 3$	($\mathbf{D}$)
		$N \cdot 6$	($\mathbf{rhsE}/\mathbf{H}$)		
$\sum_{N=20}$	15840	249		3600	
$\sum_{N=35}$	46620	429		11025	

Input: Field values $\mathbf{E}_{k,n}$, $\mathbf{H}_{k,n}$, data layout information $localMAP_{k,f,n}$, $otherMAP_{k,f,n}$, surface geometry information $\mathbf{S}_{k,f,n}$, correlation coefficients to other element $\mathbf{LIFT}_{n,f,m}$
Output: Neighborhood update terms $\mathbf{rhsE}_{k,n}$, $\mathbf{rhsH}_{k,n}$

```
1  foreach element k ∈ 1..K do
2      foreach surface f ∈ 1..4 do
3          foreach surfacenode m ∈ 1..N_f do
4              E^- = E[localMAP_{k,f,m}]
5              H^- = H[localMAP_{k,f,m}]
6              E^+ = E[otherMAP_{k,f,m}]
7              H^+ = H[otherMAP_{k,f,m}]
8              ΔE_{f,m} = S_{k,f,m} ⊗ (E^+ − E^−)
9              ΔH_{f,m} = S_{k,f,m} ⊗ (H^+ − H^−)
10         end
11     end
12     rhsE_{k,n} = 0
13     rhsH_{k,n} = 0
14     foreach node n ∈ 1..N do
15         foreach surface f ∈ 1..4 do
16             foreach surfacenode m ∈ 1..N_f do
17                 rhsE_{k,n} += LIFT_{n,f,m} ⊗ ΔE_{f,m}
18                 rhsH_{k,n} += LIFT_{n,f,m} ⊗ ΔH_{f,m}
19             end
20         end
21     end
22 end
```

Algorithm 2: Surface kernel

as *surface kernel* and give a simplified outline in Algorithm 2. For each element k, it consists of two steps, first the calculation of flux values for every surface node (Lines 2–11), and second the calculation of expected field values for every node of the element, each based on every calculated flux value from the previous step. The *surface kernel* essentially calculates the time discretized second term of Equations (3) and (4).

Three key characteristics of the *volume kernel* show up in the *surface kernel* again. First, there is ample parallelism both among iterations of the k-loop and inside them. Inside each k-iteration, there is, however, a synchronization point between Lines 11 and 14 because all flux calculations ($\Delta\mathbf{E}, \Delta\mathbf{H}$) need to be complete before computing the first $\mathbf{rhs}$ value.

TABLE II
ARITHMETIC INTENSITY OF SURFACE KERNEL.

	FLOPs		unique data	
	$4N_f \cdot 61$	(L. 8-9)	$4N_f \cdot 2$	(MAP)
	$N \cdot 4N_f \cdot 12$	(L. 17-18)	$4N_f \cdot 12$	($\mathbf{E}$, $\mathbf{H}$)
			$4N_f \cdot 5$	($\mathbf{S}$)
			$N \cdot 6$	($\mathbf{rhsE}/\mathbf{H}$)
$\sum_{N=20,N_f=10}$	12040		880	
$\sum_{N=35,N_f=15}$	28860		970	

Nevertheless, as the second similarity, the flux can be kept locally inside each k-iteration and allows for a high data reuse inside the innermost loop at Lines 17 and 18. As the third similarity, with $\mathbf{LIFT}_{n,f,m}$, again coefficients show up that are constant among all K elements. They combine a version of $\mathscr{M}$ and $\mathscr{F}$ from Equations (3) and (4) in element-local coordinates. The relative orientation of both surfaces to each other is provided through additional element-dependent coefficients $\mathbf{S}$. Table II illustrates the arithmetic intensity for this kernel, only considering memory traffic after perfect data reuse inside each k-iteration. This kernel's arithmetic is lower, reaching 3.4 operations/byte for $p = 3$ and 7.4 for $p = 4$.

An additional challenge of the *surface kernel* is that the required $\mathbf{E}$ and $\mathbf{H}$ field values are accessed indirectly, based on indices provided by two data structures, one referencing all nodes that make up the surfaces of the local element $localMAP$ and one referencing all nodes on the corresponding surfaces of neighboring elements $otherMAP$. $localMAP$ is used because nodes on an edge of an element are part of more than one surface. $otherMAP$ additionally encodes the required global topology information. The required indices are already considered in the arithmetic intensity based on Table II, but make it more challenging for a memory interface to reach its peak bandwidth.

Finally, a third kernel is required for the Runge Kutta scheme that combines the computed $\mathbf{rhsE}$ and $\mathbf{rhsH}$ with the current field values $\mathbf{E}$ and $\mathbf{H}$ and an additional residual field into new field values. The *rk kernel* operates on regular data streams and, performing much less operations than the two other kernels, has a low arithmetic intensity (0.25 operations/byte).

III. DESIGN AND IMPLEMENTATION

Before discussing the design and implementation presented in this manuscript, let us introduce the foundations that we build upon. For unstructured mesh generation, we use the open source tool Tetgen [20]. The starting point of our DG implementation was the MIDG2 project[1] by Warburton [11]. Parts of the essential kernel structure outlined in Section II-B, the calculation of analytical reference results and error metrics, as well as the complete set of coefficients for various polynomial orders are still based on this project. Also, first functional OpenCL kernels were quickly obtained from MIDG2 through its OCCA [21] kernel generation feature. Over time, however, most code has been completely redesigned into a new modern C++ reference implementation and FPGA-specific OpenCL kernel designs and host code. An open source release is in preparation. We use the Intel/Altera OpenCL SDK for FPGA version 16.0.2.222 to target an Arria 10 1150GX FPGA. The presented optimizations are based on high-level-synthesis reports, resource estimates, full synthesis runs and profiling data obtained from hardware performance counters. Tests were performed on a Nallatech 385A board that provides two memory banks of DDR4 RAM to the FPGA.

Design choices and optimizations were taken based on the algorithm structure, characteristics of the target platform and intermediate results. We present them in three categories, parallel operations, local data and global memory access.

A. Parallel Operations

In order to achieve high performance, many individual operations have to be performed in each cycle. Considering the amount of data reuse possible within the calculations for each element k, both for *volume* and *surface kernel* as outlined in Section II-B, the first decision was to exploit as much parallelism as possible within each k-iteration. For each k-iteration, given the loops structures outlined in Algorithms 1 and 2, the Intel/Altera OpenCL SDK for FPGA already creates separate instances for each of the floating point operations that show up in any of the loop nests. Thus, in correspondence to Tables I and II, dedicated circuitry for $36 + 72 = 108$ floating point operations in the *volume kernel* and $61 + 12 = 73$ in the *surface kernel* is generated in an initial design. The first challenge is, to use all of them in each cycle.

Considering the loop structures, we see that for both kernels there is one inner loop that is executed N times more often than the operations in the outer loop (or the first loop nest in case of the *surface kernel*). Thus, given that all operators of the inner loop produce a new result in each cycle, the operators of the outer loop could only be used every $1/N$ cycles, which denotes the occupancy of a functional unit. Further unrolling of the inner loops can improve the occupancy of the outer loop operators while exploiting more parallelism. For every different polynomial order p, a separate kernel variant is synthesized, with N, N_f and few other values used as compile-time constant loop bounds and array sizes. Due to resource limitations of the FPGA, unrolling the inner loop of the *volume kernel* N times and the inner loop nest of the *surface kernel* up to $4 \cdot N_f$ times is only possible for few small polynomial orders. For a parameterizable design, partial unrolling factors are used, which for every order must represent an even divider of the loop limit and are matched to fit the resource limits. The obtained unrolling factors are discussed in Section IV-A, after we discuss how all instantiated parallel operators can be fed with data in every cycle. For high occupancy, it is also important that no loop-carried dependencies prevent pipelining of the non-unrolled loops or cause any higher initiation interval (II) than 1.

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

B. Local Data

We first consider the data that has been identified in Section II-B as either remaining constant for all elements k, or being applicable for reuse within each element. The values for constant data are included through order-specific header files and put, as part of the generated bitstream, into on-chip memory resources that are partitioned or replicated, such that every access in every unrolled loop can be served independently. There are $3N^2$ constant coefficients $\mathbf{D}$ in the *volume kernel* and $N \cdot 4N_f$ constant coefficients $\mathbf{LIFT}$ in the *surface kernel*. With full unrolling of the inner loops, $3N$ and $4N_f$ of them respectively are used in every cycle. After allocating two-dimensional arrays for them, in such a way that all parallel accesses happen in the second dimension and all sequential ones in the first dimension, the compiler puts them into block memory and creates sufficient copies to serve all parallel accesses concurrently.

Conceptually, data that allows local reuse within each element, $\mathbf{E}$ and $\mathbf{H}$ for the *volume kernel* and $\Delta\mathbf{E}, \Delta\mathbf{H}$ for the *surface kernel*, is to be treated similarly. However, in order to achieve proper pipelining in the two outer loops, during the time when one set of local data is repeatedly in use, concurrently the next set needs to be loaded or computed respectively. This requires a double-buffer pattern that allows to quickly switch between the old and the new local data set at the end of each k-iteration. For the *volume kernel*, where at full unrolling each element of the local fields is used in every cycle, we implemented this with a wide shift register by annotating a pair of array declarations with `__attribute__((register))` and implementing the shift between both arrays in a fully unrolled loop. Without this explicit formulation of the double buffer, the compiler would not properly pipeline the loop because it can not statically determine that the two phases overlap by exactly one k-iteration. For the *surface kernel*, where the local flux data requires more space and thus needs to reside in block memory, there is no such dependency problem. The local flux array is declared inside the k-loop, filled in the first loop nest and consumed in the second one, ruling out any chance of inter-k-loop dependency. Due to the pipeline depth, the compiler replicates this local array four times to allow up to four iterations of the k-loop to overlap. This replication goes on top of the replication that is required to provide sufficient ports for all read locations inside the unrolled loop.

C. Global Memory Access

When all reuse of local and constant data is exploited, efficient off-chip memory access for the remaining data is required. In OpenCL this is denoted as *global memory* and on our platform resides in two banks of DDR4 memory. Based on the analysis in Section II-B, the *rk kernel* is most bandwidth limited. With sufficiently large burst accesses into linear data structures it quickly – after some unrolling – reaches a point of peak performance that we believe to mark the peak off-chip memory bandwidth of around 20GB/s that can be reached with the Altera OpenCL SDK for FPGA on this platform. To this end, the kernel can make use of the automatic interleaving across the two memory banks offered by the tool flow. Overall, the *surface kernel* with its medium arithmetic intensity accesses more off-chip memory than the *rk kernel* and thus has most impact on the overall runtime. Therefore, we focus on the discussion of its memory interface optimization, which was performed in two main steps.

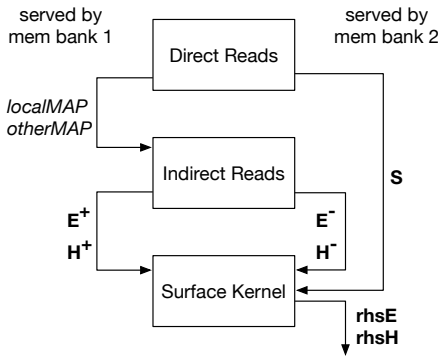

Fig. 1. Separation of direct and indirect memory accesses and separation on two memory banks.

1) We removed all global memory reads from the *surface kernel* and put them into two separate prefetching kernels (Figure 1). The first kernel performs all direct reads and forwards them into channels, a vendor-specific variant of the OpenCL pipe construct that allows direct kernel to kernel data transfers. Forming compile-time defined topologies, such channels can efficiently be implemented on FPGAs using routing resources and additional buffers as needed. The second kernel uses the MAP information from the channel to initiate the indirect and irregular memory accesses into the $\mathbf{E}$ and $\mathbf{H}$ fields. These fields are actually arranged in memory as an array of structs, such that for every such indirect access at least 6 consecutive floats, 24 bytes are read. The structs of 6 floats for field values are forwarded through another channel to the actual *surface kernel*, where they are merged with the coefficients $\mathbf{S}$ from the first kernel. We verified that both stages of indirection helped to increase memory performance and found out that the compiler automatically determined suitable depths for the channels. In particular the channel for coefficients $\mathbf{S}$ needs to be deeper than the others to compensate for the latency introduced in the indirect read kernels.

2) The irregular 24-byte reads do not profit from automatic burst-interleaving. When building a design with disabled interleaving, each read location inside the kernel can still access both memory banks, leaving it to the host code to allocate each memory object in the desired memory bank. A straightforward approach is to serve the direct memory accesses from one memory bank and the indirect ones from the other. However, in this configuration the memory bank with indirect accesses causes many more pipeline stalls than the other, as can be verified by hardware profiling. Consequently, we decided to duplicate the $\mathbf{E}$ and $\mathbf{H}$ fields and put separate copies into buffers on both memory banks. With a mix of regular

and irregular accesses served by each bank, overall highest performance is achieved. Figure 1 illustrates the decoupling with channels and separation of direct and indirect memory accesses onto both memory banks.

IV. Results

Based on the described design and implementation, we compiled and synthesized designs for the Nallatech 385A PCIe board with Arria 10 1150GX FPGA. We first report on the raw design parameters and performance metrics before comparing end-to-end performance with a CPU reference and evaluating the convergence of the implementation.

A. Design Variants, Parallelism and Throughput

We summarize parameters and results for designs with different polynomial orders p in Table III. Each design provides the full DG kernel functionality, that is it combines a *volume kernel*, a *surface kernel* and an *rk kernel*, but in the table we highlight only the two interesting kernels. For them, we we first report the achieved unrolling factor for the inner loop, and based on this the number of floating point operations that could ideally be performed per cycle. For example after unrolling the inner *volume kernel* loop for $p = 3$ by 20, there are 20 instances of the 36 floating point operations of this loop (cf. Table I), along with the 72 operations of the outer loop, enabling together 792 operations per cycle (Table III, third column). The following column depicts measured throughput in GFLOPs/s for this kernel on a mesh containing 10000 elements (K). Based on this measured value, we compute the occupancy, that is the percentage of cycles that each floating point unit is occupied. The final column per kernel displays the measured effective memory bandwidth for this kernel. After the per-kernel details, the synthesis results for the common three-kernel designs are presented.

The kernels for orders $p = 3$ and 4 perform full unrolling on the *volume kernel*, whereas for orders $p = 5$ and 6 as well as for all *surface kernels* partial unrolling is employed. Due to more area efficient local memory access, for these the loop over all 4 surfaces is always unrolled first before proceeding partially unrolling for the N_f surface nodes. We see that in spite of the irregular memory accesses of the *surface kernel*, more than 16 GB/s or 80% of the realistically achievable memory bandwidth is sustained. As the arithmetic intensity increases for higher orders, less bandwidth is needed to achieve higher occupancy and a higher compute throughput of the *surface kernel* data paths is achieved, up to 100 GFLOPs/s for $p = 6$. The *volume kernels* for orders $p = 3$ to 5 also reach or exceed 100 GFLOPs/s compute throughput, peaking at 164 GFLOPs/s for $p = 4$.

For higher orders, the parallelism in the *volume kernel* was decreased, partially in favor of more resources for the *surface kernel*. Several design alternatives with higher amounts of unrolling were explored and stay within the resource limits. Besides some designs that could not be successfully routed, others work correctly, but fail to deliver better overall performance due to reduced clock frequencies, sometimes in combination with the bandwidth limitations. For example for $p = 3$, a design with fully $(4 \cdot 10)$ unrolled *surface kernel* runs at 188MHz, reaches the same bandwidth in the *surface kernel* than the less unrolled variant from Table III, but half its occupancy, while the *volume kernel* is linearly slower with the clock frequency. Overall, the results show that the FPGA platform can achieve high sustained performance for different design points.

B. Performance on CPU

We compare the performance of our FPGA design with an optimized, yet platform-portable application for the same computations on CPUs. We use a C++ application with sufficient explicit parallel constructs built using OpenMP. With OpenMP, parallelism in the outermost loop, the k-loop in Algorithms 1 and 2 is exploited, which is in contrast the the FPGA implementation that uses k-loop parallelism only as additional dimension for pipelining. This C++ version serves three purposes: First, we used it to learn about the application, its data dependencies and access patterns, and potential performance bottlenecks. Secondly, we use it to verify our FPGA design regarding correctness. Finally, it now acts as a baseline for the performance comparison with our FPGA design.

To get an impression of the effectiveness of the parallelization efforts on CPU, we perform measurements on a compute node of the OCuLUS cluster[2] with two Intel Xeon E5-2670 CPUs, each featuring 8 physical cores operating at 2.60GHz. Based on performance characteristics of its typical workloads, OCuLUS has hyperthreading disabled. We compiled the application using the Intel C++ compiler (icc version 17.0.2) using flags `-std=c++14 -O3 -xHost -fopenmp`. We consider the following variants of this application to get a clear understanding of effects of vectorization and multi-threading.

- single-threaded with vectorization explicitly disabled
- single-threaded with automatic and annotation-based vectorization enabled
- multi-threaded and vectorized

For the non-vectorized variant, we explicitly disabled vectorization by using `-no-vec` and removed vectorization hints that were annotated in the source code. The number of threads was controlled with the `OMP_NUM_THREADS` parameter.

Through vectorization, we noticed a performance gain of 28% and 40% for orders $p = 3$ and $p = 4$ respectively. Figure 2 shows the additional speedups on top of that obtained through multithreading for 10000 mesh elements. Regardless of the numerical order, the multi-threaded variant exhibits an approximately linear scaling with the number of threads, reaching around 10x speedup with 16 threads. This suggests that the application's performance is overall computationally bound, rather than being restricted by the system's memory bandwidth. On the other hand, we do not see perfect 16x scaling with 16 threads, which can be attributed to communication and synchronization between the threads.

[2]https://pc2.uni-paderborn.de/hpc-services/available-systems/oculus/

TABLE III

UNROLLING, NUMBER OF PARALLEL OPERATIONS, MEASURED PERFORMANCE, CALCULATED UTILIZATION OF FLOATING POINT OPERATORS, MEASURED BANDWIDTH AND REPORTED CLOCK FREQUENCY OF KERNEL VARIANTS FOR DIFFERENT ORDERS.

	Volume Kernel					Surface Kernel					Synthesis Results (all Kernels)			
p	unroll	FLOPs /cycle	GFLOPs /s	occu-pancy	GB/s	unroll	FLOPs /cycle	GFLOPs /s	occu-pancy	GB/s	Freq [MHz]	Logic [%]	DSP [%]	RAM [%]
3	20	792	103	64.4%	6.1	$4 \cdot 5$	301	35	57.6%	16.3	202	32	42	42
4	35	1332	164	65.8%	6.1	$4 \cdot 5$	301	42	74.6%	12.3	187	35	60	46
5	14	576	99	85.5%	2.3	$4 \cdot 7$	397	60	75.2%	11.6	201	37	39	61
6	12	504	74	83.4%	1.2	$4 \cdot 14$	613	102	94.5%	13.4	176	38	48	72

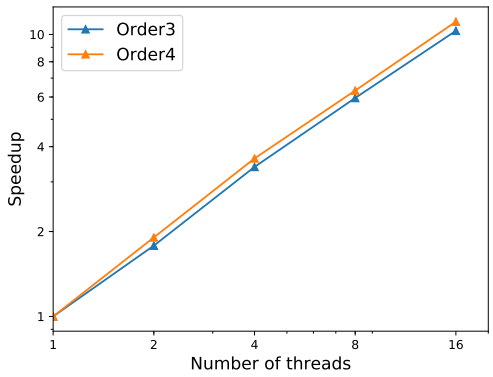

Fig. 2. Thread scaling of CPU reference for mesh with 10000 elements.

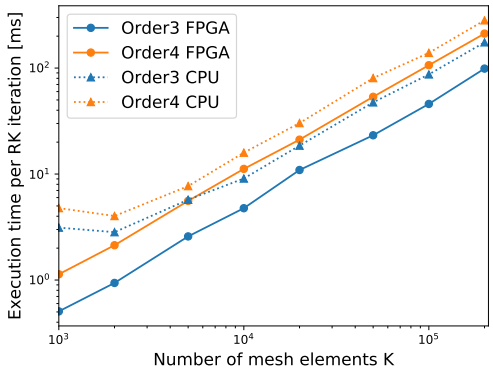

Fig. 3. Execution time on FPGA vs. 16-thread CPU reference.

In Figure 3, we compare for orders $p = 3$ and $p = 4$ the application performance of the FPGA to the vectorized, 16-thread CPU reference. To this end, we now measure the time to compute all three kernels in a single Runge-Kutta iteration. Due to dependencies, not all kernels can execute concurrently, but there is some optimization potential through overlapping. For the presented results, the CPU reference partially overlaps the execution of the *volume* and *surface kernels* in order to increase data locality. The FPGA version for order $p = 3$ in turn overlaps the execution of *volume kernel* and *rk kernel*, thus sharing bandwidth and putting the allocated compute resources to better use. A similar optimization for order $p = 4$ caused routing problems for the presented unrolling factors, thus the presented results are based on the sequential execution of all three kernels.

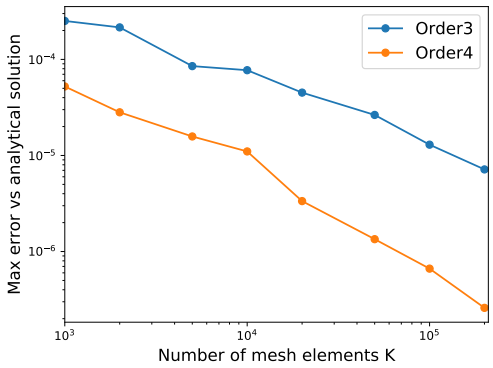

Fig. 4. Result convergence for different mesh sizes (h-refinement) and orders 3 and 4 (p-refinement).

The FPGA consistently outperforms the 16-thread CPU reference by 2x for order $p = 3$ and by around 1.5x for order $p = 4$, which can partially be attributed to the missed optimization potential of overlapping kernels. Notably, the speedups are much higher than that (up to 5x) for small meshes, where the CPU can not fully amortize for the thread management overhead. It has to be noted however, that even for those meshes, the 16-thread variant is the fastest on CPU.

Finally, with Figure 4, we verify the convergence rate [17] of our FPGA designs for orders $p = 3$ and $p = 4$ towards the given analytical solution. As desired, the result converges both with increased mesh size (*h-refinement*) and with higher polynomial order (*p-refinement*).

V. RELATED WORK

To the best of our knowledge, this is the first realization of the DG method on FPGAs. The MIDG2 DG implementation builds upon OCCA, which through OpenCL conceptually can target FPGAs [21]. Analysis of DG implementations on GPUs shows potential and challenges [22]. For polynomial orders up to $p = 6$, an implementation calculating a single nodal point per thread achieves less data reuse than our design, but profiting from an order of magnitude more effective memory bandwidth reaches higher absolute performance of between 500 and 1000 GFLOPs/s. For yet higher orders, it pays of to express the calculations as general dense matrix-matrix multiplication problem (SGEMM).

Technically related to our work is the implementation of a discretized Euler method for unstructured meshes on FPGA [23]. Based on a much simpler numerical method, it goes beyond our current work in two regards. First, the authors use a mesh node renumbering technique that is presented in more detail in [24]. Optimizing the mesh in this way allows to keep neighboring cells close to each user and thus fully reuse them through a specialized cache in the on-chip memory. Second, the authors manually pipeline and optimize their arithmetic unit with 71 floating point operations and thus achieve 325MHz on Virtex-6. With 3 instances of this arithmetic unit, they achieve 69 GFLOPs/s for the single kernel. In comparison, our work utilizes the higher amount of resources of the Arria 10 platform to generate wider datapaths for more kernels.

VI. Conclusion

In this work, we have ported a state-of-the art simulation method from computational sciences for the first time to FPGAs and found it to be well-suited. We have introduced an OpenCL-based implementation that can be adapted to generate design variants for different polynomial orders. We have presented FPGA and tool-flow specific optimizations that allow to utilize the potential of the target platform and achieve high sustained performance. Furthermore, we have outlined conceptual differences and performance improvements compared to a highly multi-threaded CPU reference.

By applying mesh reordering techniques and customized memory window implementations, further efficiency may be gained. As the DG method is also suitable for large-scale HPC, multi-FPGA or MPI parallelization provide other promising directions for further research.

Acknowledgment

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) within the collaborative research project "HighPerMeshes" (011H|160054) and by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901) and within the project "PerficienCC - Performance and Efficiency in HPC with Custom Computing" (PL 595/2-1).

References

[1] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 3, pp. 695–705, Mar. 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.51

[2] H. M. Waidyasooriya and M. Hariyama, "FPGA-based deep-pipelined architecture for FDTD acceleration using OpenCL," in *Proc. IEEE/ACIS Int. Conf. on Computer and Information Science (ICIS)*, June 2016, pp. 1–6.

[3] T. Kenter, J. Förstner, and C. Plessl, "Flexible FPGA design for FDTD using OpenCL," in *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–7.

[4] H. Giefers, C. Plessl, and J. Förstner, "Accelerating finite difference time domain simulations with reconfigurable dataflow computers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 65–70, Jun. 2014.

[5] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64bitt floating-point FPGA matrix multiplication," in *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 86–95. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046204

[6] H. Giefers, R. Polig, and C. Hagleitner, "Measuring and modeling the power consumption of energy-efficient FPGA coprocessors for GEMM and FFT," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 307–323, Dec 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-1057-6

[7] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, pp. 15:1–15:33, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2159542.2159547

[8] J. Sheng, B. Humphries, H. Zhang, and M. C. Herbordt, "Design of 3D FFTs with FPGA clusters," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.

[9] D. E. Shaw *et al.*, "Anton, a special-purpose machine for molecular dynamics simulation," *Communications of the ACM*, vol. 51, no. 7, pp. 91–97, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1364782.1364802

[10] ——, "Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2014, pp. 41–53. [Online]. Available: https://doi.org/10.1109/SC.2014.9

[11] J. S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Science & Business Media, 2007.

[12] A. Taflove and S. Hagness, *Computational Electrodynamics: The Finite-difference Time-domain Method*, ser. Artech House antennas and propagation library. Artech House, 2005.

[13] Y. Grynko, Y. Shkuratov, and J. Förstner, "Light scattering by irregular particles much larger than the wavelength with wavelength-scale surface roughness," *Optics letters*, vol. 41, no. 15, pp. 3491–3494, 2016.

[14] P. Monk, *Finite Element Methods for Maxwell's Equations*, ser. Numerical Analysis and Scientific Computing. Clarendon Press, 2003. [Online]. Available: https://books.google.de/books?id=zI7Y1jT9pCwC

[15] M. Dumbser and M. Käser, "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes II. the three-dimensional isotropic case," *Geophysical Journal International*, vol. 167, no. 1, pp. 319–336, 2006.

[16] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas, "A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media," *Journal of Computational Physics*, vol. 229, no. 24, pp. 9373–9396, 2010.

[17] K. Busch, M. Koenig, and J. Niegemann, "Discontinuous Galerkin methods in nanophotonics," *Laser & Photonics Reviews*, vol. 5, no. 6, pp. 773–809, 2011.

[18] 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*. Springer, 2017, pp. 261–284.

[19] M. H. Carpenter and C. A. Kennedy, "Fourth-order 2N-storage Runge-Kutta schemes," NASA Langley Research Center, Hampton, VA, USA, Tech. Rep. NASA-TM-109112, Jun. 1994.

[20] S. Hang, "A quality tetrahedral mesh generator and a 3 d delaunay triangulator," 2005.

[21] D. S. Medina, A. St.-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," *CoRR*, vol. abs/1403.0968, 2014. [Online]. Available: http://arxiv.org/abs/1403.0968

[22] A. Modave, A. St-Cyr, and T. Warburton, "GPU performance analysis of a nodal discontinuous galerkin method for acoustic and elastic models," *Computers & Geosciences*, vol. 91, pp. 64 – 76, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0098300416300668

[23] Z. Nagy, C. Nemes, A. Hiba, A. Kiss, Á. Csík, and P. Szolgay, "FPGA based acceleration of computational fluid flow simulation on unstructured mesh geometry," in *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 128–135.

[24] A. Hiba, Z. Nagy, and M. Ruszinko, "Memory access optimization for computations on unstructured meshes," in *2012 13th Int. Workshop on Cellular Nanoscale Networks and their Applications*, 2012, pp. 1–5.