Flexible FPGA design for FDTD using OpenCL

Tobias Kenter*, Jens Förstner†, Christian Plessl‡
*‡Paderborn Center for Parallel Computing and Department of Computer Science
†Department of Electrical Engineering
Paderborn University
Warburger Str. 100, 33098 Paderborn, Germany
Email: *kenter@uni-paderborn.de †jens.foerstner@uni-paderborn.de ‡christian.plessl@uni-paderborn.de

***Abstract*—Compared to classical HDL designs, generating FPGA with high-level synthesis from an OpenCL specification promises easier exploration of different design alternatives and, through ready-to-use infrastructure and common abstractions for host and memory interfaces, easier portability between different FPGA families. In this work, we evaluate the extent of this promise. To this end, we present a parameterized FDTD implementation for photonic microcavity simulations. Our design can trade-off different forms of parallelism and works for two independent OpenCL-based FPGA design flows. Hence, we can target FPGAs from different vendors and different FPGA families. We describe how we used pre-processor macros to achieve this flexibility and to work around different shortcomings of the current tools. Choosing the right design configurations, we are able to present two extremely competitive solutions for very different FPGA targets, reaching up to 172 GFLOPS sustained performance. With the portability and flexibility demonstrated, code developers not only avoid vendor lock-in, but can even make best use of real trade-offs between different architectures.**

I. Introduction

To the reconfigurable computing community, FPGAs are known for long as promising accelerators for a wide range of problems. With recent large-scale adoption of FPGAs by manufacturers and cloud providers [3], FPGAs are slowly moving towards becoming a mainstream computing product. Design flows starting from high-level specifications are an important stepping stone to deliver the required productivity for such usage. In particular synthesis from OpenCL specifications is promising, because it builds upon an established standard and promises easy portability between different FPGA families and vendors. In this work, we investigate productivity and portability of this approach in an extensive case study on photonics simulation.

Simulation codes in computational sciences are often used and refined over many years. Thus, in this domain, the utilization of widely supported and hopefully long-living standards is particularly important. The finite-difference time domain (FDTD) method used in this work is a stencil computation method for the simulation of electromagnetic fields based on Maxwell's equations. Because of the interleaved update steps of electric E-field and magnetic H-field on staggered grids, FDTD for a Maxwell solver is a bit more complex than the widely used Laplace stencil. In this work, we present a 2D FDTD implementation used for the simulation of photonic effects in microcavities.

A predecessor of this design was presented in [5], implementing deeply unrolled pipelines with Xilinx SDAccel 2016.1 on a Virtex-7 X690T-2 FPGA. In this work, we present a number of extensions and improvements towards a more flexible and more efficient design. First of all, we aimed at portability to the Intel/Altera OpenCL SDK and targeted an Arria 10 1150 GX FPGA. Along this way, we introduced a configurable design of the memory interfaces and present a design space exploration of selected alternatives. We secondly enabled splitting unrolled pipeline stages into several decoupled kernels, which is particularly important when targeting Kintex Ultrascale KU115-2 FPGAs with two to distinct so-called super-logic regions (SLR). Thirdly, making use of OpenCL vector data types, we allowed to explore trade-offs between deep pipelining and wide data paths. Finally, overall performance is also much improved by reducing the initiation interval of the pipeline from two cycles in the previous work to one cycle. With up to 10 Giga cell updates per second and more than 170 Gflop/s sustained performance, our design is to the best of our knowledge faster than any published reference on FPGA.

In the remainder of this paper, we first present some more details about the application. In Section III, we present our design goals and approach and lay out our target platforms. In Section IV, we describe the design of the compute pipeline, which is interfaced with streaming kernels for memory accesses, discussed in Section V. Section VI contains results of fully integrated designs aiming for the highest overall performance. Before concluding in Section VIII, we discuss related work in Section VII.

II. FDTD Algorithm and Microdisk Cavity

Our application solves Maxwell's equations to compute the propagation of light in a microstructure using the FDTD

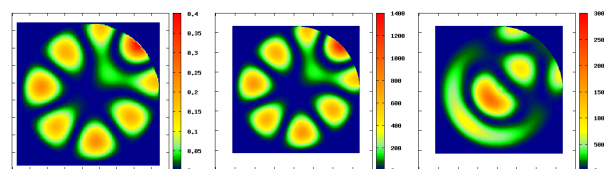

Fig. 1. Samples of simulation results: proper Whispering Gallery modes in different cavity sizes and adapted stimulus frequency (left, center) and superposition/interference pattern of modes (right).

method first introduced by Yee [9]. Listing 1 illustrates the basic update kernel of 2D FDTD, with the E-field calculated in x- and y-dimensions (`ex, ey`) and H-field in z-dimension (`hz`). Per iteration and grid point, it uses $8 + 6 = 14$ floating point operations to compute an update step. With data reuse inside each row, it uses 5 load and 3 store operations of the simulated data type, in our experiments single precision floats with 32 bit. When data from neighboring rows can be reused, the loads can be reduced to 3, but even then the application remains memory intensive.

The photonic device in our simulation is a so-called microcavity, that is a circular hole (vacuum) surrounded by perfectly conducting metal. Figure 1 illustrates different simulation results for such devices. The microcavity is a well-suited structure for a non-synthetic 2-dimensional benchmark because it is used in actual photonic devices and the solution of the problem can be analytically validated. Also, the use of different materials, metal and vacuum, requires to handle different, material-dependent stencils or stencil coefficients (`ca, cb, da, db`) for geometric domains, which complicates vectorization. Besides the basic update step from Listing 1, our actual implementation additionally injects a configurable source impulse near the disk center. Additionally, in order to get more meaningful results than the raw field strength at a given end time, it integrates the energy of the H-field over time (using 2 additional floating point operations and 2 additional memory operations).

```
for(int t=0; t<timeSteps; t++)                    1
  // Update E                                     2
  for(int x=0; x<height; x++)                     3
    for(int y=0; y<width; y++)                    4
      ex[x,y] = ca * ex[x,y] + cb *               5
                (hz[x,y] - hz[x,y-1]);            6
      ey[x,y] = ca * ey[x,y] + cb *               7
                (hz[x-1,y] - hz[x,y]);            8
  // Update H                                     9
  for(int x=0; x<height; x++)                     10
    for(int y=0; y<width; y++)                    11
      hz[x,y] = da * hz[x,y] + db *               12
                (ex[x,y+1] - ex[x,y]              13
                + ey[x,y] - ey[x+1,y]);           14
```

Listing 1. Kernel loop for 2D FDTD

III. Design Goals and Approach

As acceleration goal, we want to simulate in single precision floating point configurable grids of up to 4096 x 4096 elements, with configurable cavity sizes and a flexible stimulation source that can be precomputed on the host. The grid size is motivated from the application domain, but is also important for a fair comparison of different computing products because when grids are too small, they may fit entirely into on-chip memory resources, be it FPGA block memory or processor caches. In this section, we first present the FPGA target platforms and OpenCL-based synthesis tools used in this work and then proceed with general considerations when approaching the target application with these tool flows.

TABLE I
Overview of target platforms. *Only 1 memory bank supported in OpenCL configuration.

Board	Alpha Data ADM-PCIE-8K5	Nallatech 385A
FPGA	Xilinx Kintex Ultrascale KU115-2	Intel/Altera Arria 10 1150 GX
Note	2 regions (SLRs)	floating-point DSPs
Memory	2 x 8GB DDR4 2 x 2400MT/s	2 x 4GB DDR4 2 x 2333MT/s
Peak Bandwidth	2 x 19.2 GB/s	2 x 18.7 GB/s

A. Target Platforms and Tools

Starting from the earlier design in [5] that was developed with Xilinx SDAccel 2016.1, all Xilinx results presented in this publication are obtained with the SDx 2016.3 release that integrates the OpenCL-based SDAccel and the C/C++-based SDSoC tools. We also target the Intel/Altera SDK for OpenCL, Version 16.0.2 Build 222 that uses the same version of Quartus II as synthesis backend. Both tool flows build upon the embedded profile of the OpenCL 1.0 specification but differ by selective support for more recent OpenCL features and vendor-specific extensions, attributes and pragmas. Most notably, both vendors adopt the OpenCL 2.0 pipe feature. Intel/Altera also provides a vendor-specific *cl_altera_channels* extension. Direct, on-chip connections between IP blocks, here kernels are a well-known design feature of FPGAs. In contrast to the OpenCL pipe feature, they are most effectively instantiated statically at compile and synthesis time. Intel/Altera promotes this through the channel extension and additionally allows some standard-compliant pipe implementations, whereas Xilinx alters the pipe semantic itself. Table I gives an overview of the two boards and FPGA platforms we target in this work with the respective tool flows.

B. General Design Approach

When an OpenCL kernel is translated to an FPGA circuit, for every operation a dedicated functional unit is instantiated. There are two general approaches to reach a high utilization of functional units inside a loop nest. The GPU-inspired, typical OpenCL way is to process independent loop iterations as individual work items and let the scheduler at compile time or during runtime schedule these work items either in a pipelined way or to parallel instances. The OpenCL-FPGA flows support this mode, but can generate more efficient designs with the alternative approach of single work item kernels. Here, loop pipelining is performed at compile time and all elements are processed statically in loop order. Thus, explicit data reuse between neighboring spatial loop iterations is possible and no overhead is required to determine work item IDs or for synchronization between work items.

In this work, we decided to follow the more efficient single work item approach, in particular because the memory intensive nature of the FDTD algorithm requires to reuse data as effectively as possible. The first goal is thus to pipeline the inner

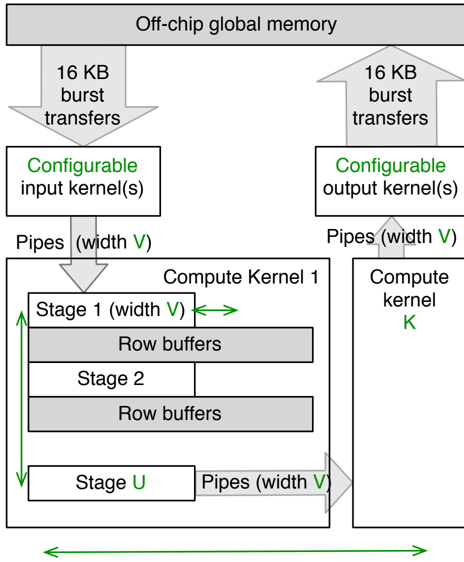

Fig. 2. Overall structure and configuration options that our design exposes.

kernel loop with a low initiation interval, ideally of one cycle. Further parallelism can then be achieved by creating deeper or wider pipelines. Replicated pipeline stages connected by row buffers can work on consecutive time iterations at the same spatial position [2], [5], [8]. A similar technique, time skewing, is also widely employed when optimizing processor cache utilization for stencil applications [4]. Wide pipelines can be created with explicit loop unrolling or by using OpenCL vector data types, combining up to 16 elements into a single variable for example of type *float16*. As there is a trade-off in our design – deep pipelines require more on-chip memory resources, wide pipelines more off-chip memory bandwidth – we decided to pursue both options and search the design space for the fastest overall designs.

Large and tightly coupled pipelines can also make it difficult for the place and route tools to achieve the desired frequency targets, particularly for the distinct super-logic regions (SLR) of the KU115-2 target. Thus, we additionally added the option to decouple the pipeline into different kernels connected via OpenCL pipes that serve as FIFO buffers. We furthermore decided to decouple the compute pipeline from the off-chip memory interfaces. While simple kernels can easily perform computations and burst memory access in lockstep, we found it difficult to achieve this for our design with variable grid size and number of time steps. Also, it turned out that the optimal interface configurations depend strongly on the considered platforms as discussed in Section V. Figure 2 illustrates the degrees of freedom that our flexible design opens up in addition to the portability between different FPGA platforms.

IV. Compute Pipeline Implementation

As outlined in the previous section, the compute kernel receives all input from off-chip global memory through pipes (or channels) and forwards its results to output pipes. We use blocking pipes that can read or write one element in each cycle and stall when an input pipe is empty or an output pipe is full. The syntax for these constructs is different between the two tool flows, but the semantic is identical. Thus, we wrapped all pipe functions into simple pre-processor macros like outlined in Listing 2 for a portable interface to the compute pipeline.

```
#if defined(__xilinx__)                                   1
  #define PIPE_READ(pipename, elem) \                     2
    read_pipe_block(pipename, &elem)                      3
#elif defined(ALTERA_CL)                                  4
  #define PIPE_READ(pipename, elem) \                     5
    elem = read_channel_altera(pipename)                  6
#endif                                                    7
```

Listing 2. Pre-processor macro to encapsulate pipe functionality.

The first scalar pipeline processes a pair of E-field and H-field updates at a new position in every cycle. As can be seen in Listing 1, the H-field update at position [x,y] cannot simply take place directly together with the E-field update at the same position, because H-field update accesses values in the E-field with a positive offset (`ex[x,y+1]` and `ey[x+1,y]`). Thus, together with the E-field update at position [x,y], we perform instead the H-field update at the position of the previous outer loop iteration [x-1, y]. It uses as most recent input value the just computed `ey` value, but also values from all fields that were computed up to `width` steps earlier. These are kept locally in line buffers implemented as local arrays in OpenCL. The line buffers cause both compilers to detect false read-after-write dependencies for consecutive loop iterations, even though all accesses are at least `width` positions apart. With `__attribute__((xcl_dependence(type="inter", direction="RAW", dependent="false")))` and `#pragma ivdep` respectively, these dependencies are resolved and the compilers can lower the initiation intervals from two to one cycle. This also requires, that no line buffer is read twice per cycle, like naively happens for example to process `(ex[x,y+1] - ex[x,y])`, but can be replaced by an additional shift register. In addition to the four line buffers for the H-field update (including one for the `hz_sum` field used for result visualization), a fifth buffer is used in the E-field update step for the H-field. Since the first valid H-field update is produced only after `width` pipeline iterations, its line buffers are at the same time used to hold back the updated E-field values for the same time, so fields leave the pipeline again in lockstep and can be written back to off-chip memory.

Intermediate results do not need to be written back to off-chip memory, host memory or ultimately the file system after every time step, because we integrate an analysis in the form of energy integration into the compute pipeline. Thus, these results can instead also be forwarded into another, replicated pipeline stage. Pipeline replication can conceptually be realized through unrolling hints in the outer (time) loop of the kernel. This requires extending the row buffers into a second dimension and partitioning them in a way that each row can be accessed independently. Also, inputs need to be selected between either the previous pipeline stage or the pipe

interfaces. With such a design, we were no longer able to achieve the desired low initiation interval, which is related to a problem in the two-dimensional partitioning of the SDx tool chain. Thus, we instead resorted back to pre-processor macros to generate a configurable number of pipeline stages with individually named row buffers. With this approach, we in turn uncovered a problem in the Intel/Altera flow, that only allows execution with at most 128 named local memory objects per design, for example up to 24 pipeline stages with five row buffers each. Since the designs with the highest overall parallelism on this platform use a trade-off between deep and wide pipelining, they are not affected by this problem. Therefore, we did not invest the effort of finding a work-around for deeper pipelines with fewer, partitioned buffers.

There are two approaches to realize wider pipelines that process several spatially adjacent elements per pipeline iteration, here of the innermost (y) loop. As mentioned unrolling hints could be applied to this loop, or OpenCL native vector data types like `float16`. Given the chosen pipe interface for off-chip inputs and outputs, only the latter variant is suitable, since pipes can process an entire native data element like `float16` with each invocation, but can't process multiple smaller elements in a single cycle, which would be required to keep a wide compute pipeline fed. Thus, we use vector data types, again wrapped by some pre-processor code in order to keep the required modifications to operations and iteration spaces configurable. Both design flows can build functionally correct hardware designs from all vectorized pipelines, however for Intel/Altera the software emulation yields incorrect results for vector sizes of 8 or 16, whereas for Xilinx, starting with 2-element vectors the compiler frontend can no longer resolve the false dependencies and thus increases the initiation interval back to two.

Finally, for the mentioned separation of large monolithic compute pipelines into several smaller kernels decoupled by pipes, we again use pre-processor macros to generate individually named kernels, `compute_0, compute_1, ... compute_K`. In contrast to all the previously discussed design options (including vector data types), this one requires an explicit counter-part in the host code, discovering all available compute kernels and calling them from separate OpenCL contexts (Altera/Xilinx) or a single, asynchronous context (only Xilinx, slightly more efficient). For Xilinx SDx, besides the kernels, also the connecting pipes must be individually named, whereas Altera OpenCL also supports arrays of pipes. As pipe accesses were already encapsulated by the pre-processor macros shown in Listing 1 anyway, we just added another indirection appending an ID to each name. Before reporting on integration results for this configurable compute pipeline, we first individually explored alternatives and performance of the memory interface kernels feeding it.

V. Memory-Interface Kernels for DRAM Access

In order to achieve high off-chip memory bandwidth, burst transfers from or to many consecutive memory addresses are required. There are two general approaches to achieve this, firstly through compiler inference of bursts from loops with sufficiently clear iteration space, and secondly by using the OpenCL builtin-function `async_work_group_copy` that is functionally supported by both tool flows. In our experiments, we encountered two tool-specific limitations that caused us to switch between the two variants depending on the tool flow used, by inserting another pre-processor macro. Xilinx SDx can correctly infer the simple burst patterns we require, however in order to generate according designs the memory port bitwidth needs to match that of the employed data type, that is between 32 bit for `float` and 512 bit for `float16`. Such memory ports turned out inefficient for all but the widest data types, thus we limited the further exploration of interfaces for SDx to `async_work_group_copy`. This requires additional local on-chip buffers to read into or write from. We partitioned these buffers cyclically to allow for all data types to access 512 consecutive bits in a single cycle.

The utilization of `async_work_group_copy` caused the Intel/Altera OpenCL tool flow to synthesize the interface kernels not as single work-item kernels, which in conjunction with a single work-item compute kernel seriously hampers performance without further modifications. Thus, we here focused on burst inference from loops. In this case, off-chip memory accesses can either go to a buffer like with `async_work_group_copy`, or be directly connected to the pipes. The latter could conceptually allow a better overlap of memory access and pipe interaction, but stalls from the pipeline can break its burst behavior. Listing 3 illustrates the parametrized generation of these alternatives for an input kernel transferring data from off-chip array `ex_global` into `ex_pipe`. The `USE_AWGC` variant uses `async_work_group_copy` (used with Xilinx design flow), `INFER_EXTRA_BURST` uses burst inference to transfer data into a local buffer, and `INFER_INLINE_BURST` (both used with Intel/Altera design flow) forwards burst reads directly into a pipe.

```
#if (USE_AWGC > 0 || INFER_EXTRA_BURST > 0)          1
  __local _VEC_DTYPE_ ex_local[BURSTSIZE];            2
#endif                                                3
#if (USE_AWGC > 0)                                    4
  async_work_group_copy(ex_local[0],                  5
    ex_global+burstPos, BURSTSIZE, 0);                6
#elif (INFER_EXTRA_BURST > 0)                         7
  for(s=0; s<BURSTSIZE; s++)                          8
    ex_local[0][s] = ex_global[burstPos+s];           9
#endif                                                10
// Write to pipe                                      11
for(s=0; s<BURSTSIZE; s++){                           12
  #if (INFER_INLINE_BURST > 0)                        13
  PIPE_WRITE(pipe_ex, ex_global[burstPos+s]);         14
  #else                                               15
  PIPE_WRITE(pipe_ex, ex_local[0][s]);                16
  #endif                                              17
}                                                     18
```

Listing 3. Code illustration of interface variants.

As final alternative, we either instantiated a single interface kernel for each separate field (as illustrated by Listing 3, or

TABLE II
OVERVIEW OF EXPLORED INTERFACE ALTERNATIVES.

	Xilinx SDx	Altera OpenCL
async_work_group_copy	used	slow (ND-range kernels)
burst inference to buffer	slow (port width)	used
inline burst inference	not tested	bit slower, high variance
separate / joint kernels	see results	see results

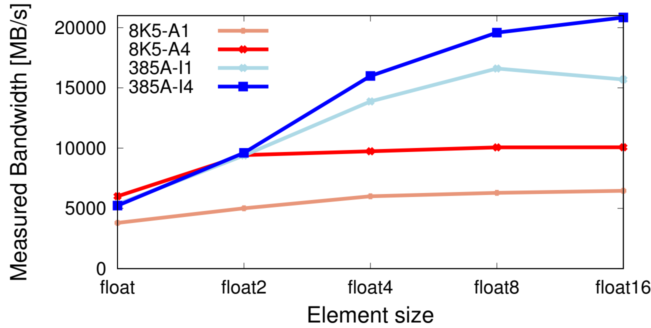

Fig. 3. Memory throughput of 16KB bursts for different target platforms, vector data types and streaming interfaces. A/I denote interface mode (asyn_work_group_copy, infer bursts), final digit denotes number of interface kernels per direction.

jointly transferred either a pair of two or all four fields within each interface kernel. With the given structure, joint interface kernels need to wait for the bursts of all fields to complete, before proceeding to the pipe loop (in Listing 3 beginning in Line 12). On the other hand, they can access memory in a more regular order and also open up slots for more compute kernels in the SDx design flow that is currently limited to a total of ten kernels.

Table II summarizes the investigated interface alternatives. After sorting out the less promising alternatives, we performed detailed measurements for different pipeline widths and number of interface kernels. To this end, we integrated the interface kernels with an empty compute kernel that in every cycle tries to consume one element (of the vector data type in case of pipeline width > 1) from each input pipe and write it to the corresponding output pipe. Thus, our interface tests are constrained by the real interplay of off-chip memory bursts and internal pipe interfaces and just omit the latency of real compute pipelines. The total achieved memory throughput for selected alternatives is presented in Figure 3. For the presented experiments, we fixed the burst size to 16KB regardless of the data type and iterated 1440 times over 256KB of data. The results contain explicit bank partitioning for the 8K5 board and automatic interleaving for the 385A board. We see that four separate interface kernel always enable superior or equal throughput to joint interface kernels. For one- and two-element vector types with four interface kernels, the 8K5 and 385A

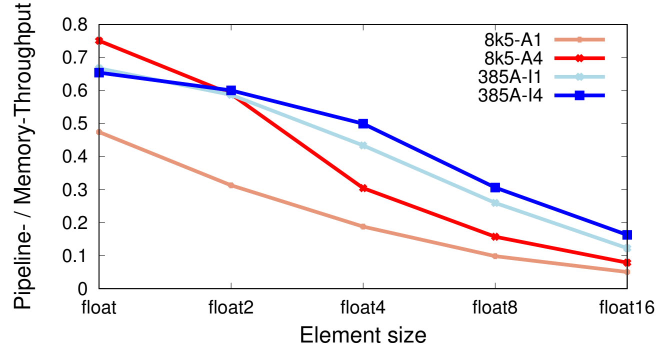

Fig. 4. Projected efficiency of memory interfaces when feeding a vectorized compute pipeline with initiation interval (II) of 1 at default target frequency.

boards with their very similar physical memory configuration perform similar, but the 5K8 board doesn't scale further up. This could either be caused by the different burst generation methods, or by limits of the common AXI bus shared by all kernels.

The goal of our interface design is not highest off-chip memory bandwidth per-se, but the ability to feed a compute pipeline with a given peak throughput as efficiently as possible. For example a scalar single-stage compute kernel at 200 MHz needs 6400 MB/s memory throughput to achieve its peak throughput of 200 Mega cell updates per second (short Mcells/s), an equally clocked 32-stage scalar compute kernel can use the same off-chip bandwidth to compute 6400 Mcells/s, whereas a 16-wide single-stage compute kernel at 250 MHz requires for full throughput of 4000 Mcells/s an off-chip memory bandwidth of 128 GB/s, which is beyond reach for our utilized boards. In Figure 4, we illustrate how efficiently compute pipelines of different widths can be utilized with the respective memory interfaces. To this end, we divide the bandwidth demands of each compute pipeline at the default OpenCL target frequencies by the measured bandwidth of the investigated interface configurations. We see that for any fixed amount of total parallelism (pipeline depth * pipeline width), deep scalar pipelines (element type `float`) can be used more efficiently than variants for wider vector data types. This is not surprising due to their higher data reuse. However, when wider pipelines are more resource efficient and thus allow more total parallelism, the efficiency of the 385A interfaces decreases slower than for the other boards.

VI. DESIGN INTEGRATION AND EVALUATION

In this section, we give an overview over a selection of designs that are the result of a design space exploration process for each of the target platforms. Following the findings of the previous section, we started by scaling scalar pipelines as deep as possible for both platforms. Both for the 8K5 and the 385A target, we initially reached a total pipeline depth of 40

stages. Synthesis details are summarized in the first columns of Tables III and IV respectively. As indicated in Section IV, currently our Intel/Altera designs with more than 24 pipeline stages can not be executed and would require modifications reducing the number of individually named buffers.

For the Xilinx Ultrascale design, we distributed the compute pipeline to two kernels with 20 stages each in the hope of achieving good utilization of both SLRs. The highest overall resource usage is in logic (CLBs) at 76%, but with 90.14% and 60.70% individual CLB utilization in the respective SLRs, imbalance in the regions seem to have prevented us from reaching deeper parallelism with two kernels. Notably, the bipartition bandwidth between the SLRs is not an issue, using only between 10% and 17% of the available connecting super long lines (SLLs). We believe that more independent synthesis, place and route attempts should be able to overcome the partitioning limitation, but instead we proceeded by splitting the pipeline to even more decoupled kernels to give the tools more freedom to find a beneficial partitioning. To that end, we derived another interface variant with two pairs of interface kernels for input and output, that each stream two fields. Its performance is indeed between that of four and of one interface kernels, thus it is most suitable when between three and six of the maximum ten kernel slots in the SDx design flow are needed for compute kernels. This way we obtained two designs with more total parallelism (see middle and right column of Table III).

Due to the low interface efficiency for wider pipelines and to the increased initiation intervals caused by the compiler frontend, wider designs were not very promising. Initial tests also seem to indicate that wide pipelines don't help much to reduce logic utilization. At this point it is unclear to us, whether the free DSP resources could be put to better use in floating point applications like ours. There are some spare BRAM resources that would be very helpful when aiming for larger row buffers or burst interface buffers.

Running two of these designs – the third one sadly hangs at first kernel invocation when running on actual hardware – we measure the effective throughput in MCells/s. With more than 5 GCells/s, we were able to more than double the 2 GCells/s our previous publication [5] on Virtex-7 and approach the throughput of the simpler Altera Stratix V design from [8]. Measured performance is very consistent over different grid sizes. Due to timing problems, the design tools reduced the clock frequency of the final designs below the target of 250 MHz. Thus, the measured results are below the extrapolation of multiplying the isolated interface throughput from the previous section with the total pipeline depth. On the other hand, with an ideal memory interface, the compute pipelines could also achieve more throughput, as we see from multiplying the clock frequency with the total parallelism.

The scalar Altera Arria 10 design (Table IV) is limited by on-chip memory resources usage 96% of the available RAM blocks. To mitigate this pressure, wider, but less deep pipelines that thus require less row buffers are helpful. Additional overall parallelism needs to make up for the less efficient interfaces for wider pipelines analyzed in the previous section. With pipelines of `float4` elements, we were able to more than double the total amount of parallelism. The shift from deep to wide Altera designs visibly shifts the resource pressure from on-chip memory to DSP blocks, up to the point where 86% of the 1518 variable-precision DSP blocks are utilized. Logic utilization is still relatively relaxed, probably because of the extended functionality of the DSP blocks.

The measured performance of the fully functional vectorized design reaches 10.7 GCells/s, higher than the simpler reference from [8]. Measured performance with a 4-wide and 20-deep design even slightly surpasses the projection based on the isolated interface measurements. This is possible because the compute pipeline latency can decouple the points in time when input and output kernels initiate their burst transfers. At 16 floating point operations per cell update, our fastest design sustains 172 GFLOPS, to the best of our knowledge the highest throughput published for a similar application on a single FPGA. Without off-chip memory limitations, the raw performance of the compute pipeline itself could even reach 283 GFLOPS.

TABLE III
SELECTED DESIGNS ON 8K5 WITH KINTEX ULTRASCALE KU115-2. FIRST AND SECOND DESIGN SUCCESSFULLY TESTED, THIRD DESIGN HANGS AT HARDWARE EXECUTION, BUT IS FUNCTIONAL IN SIMULATION.

total parallelism	40	45	48
configuration (kernels x stages x vector)	2 x 20 x 1	3 x 15 x 1	6 x 8 x 1
interface kernels (each dir.)	4	2	2
clock frequency [MHz]	160	170	150
CLB usage (SLR1, SLR2)	90%, 61%	77%, 94%	84%, 97%
BRAM usage	51%	57%	63%
DSP usage	24%	27%	29%
throughput MCells/s			
measured	**4961**	**5393**	-
extrapolated from interface	7480	7110	7584
theoretical pipeline peak	6400	7650	7200

TABLE IV
SELECTED DESIGNS ON 385A WITH ARRIA 10 1150 GX. TWO FIRST DESIGNS NOT RUN DUE TO LIMITATION TO EXECUTE DESIGNS WITH MORE THAN 128 NAMED MEMORY BLOCKS, THIRD AND FOURTH DESIGN FULLY FUNCTIONAL (USED TO PRODUCE SAMPLE RESULTS FOR SECTION II.)

total par.	40	64	80	96
configuration (KxSxV)	1x40x1	1x32x2	1x20x4	1x24x4
i.f. kernels	4	4	4	4
freq. [MHz]	191	202	208	185
ALM usage	38%	34%	34%	37%
RAM usage	96%	88%	66%	74%
DSP usage	37%	60%	72%	86%
throughput				
measured	-	-	**10183**	**10722**
i.f. x par.	6520	9606	9993	11991
theor. peak	7640	12928	16640	17706

VII. Related Work

As most closely related work with regard to the design and to performance numbers, Waidyasooriya and Hariyama [8] report of two very similar OpenCL-based 2D FDTD designs on Altera Stratix V GX. The faster design from [8] completes a test setup of 1024 x 1024 grid and 15360 simulation steps in 1.69 seconds, which corresponds to 8.876 Gcells/s. Our own previous work [5] targeted Xilinx Virtex-7 and reached around 2 Gcell/s on grids up to 4096 x 4096. Both designs [5], [8] use only unrolling in time. In [5], we present 36 pipeline stages, but only with an initiation interval of two. Targeting the smaller grid and thus requiring 4x smaller row buffers, [8] reaches a pipeline depth of 44 with initiation interval of one. It performs only the basic field updates and lacks enhanced evaluation options like the integration of H-field energy over time. Such limited access to intermediate results is a typical shortcoming of libraries that can force computational scientists to write or modify code themselves. With an OpenCL-based design, such modifications are now also within reach for FPGA targets.

Cattaneoet et al. [1] investigate data reuse strategies for simple 2D and 3D stencil calculations on FPGA. Their approach is more generic than the highly application-specific optimizations and configurable target platform-specific optimizations we present. Absolute performance numbers in GFLOPS are however an order of magnitude below our results. Sano et al. [6] pipeline stencil computations over nine FPGAs and this way obtain performance of up to 260 GFLOPS. With our approach of splitting a deep compute pipeline into separate kernels coupled through pipes, scaling over multiple devices would be an interesting next step. We already cross the FPGA family and design tool boundaries at the specification level. A particular challenge could be to try to connect different target FPGAs via pipes or – easier, but less exciting – by calls from a common host code with MPI scaling.

Vasiljevic et al. [7] propose a library of OpenCL memory streaming components that can be used to decouple compute kernels from off-chip memory interface. In contrast to the interface kernels employed in our work, this library does not focus on portability, but on support for three different memory access patterns.

VIII. Conclusion

We have presented a flexible FDTD design for microcavity simulations that can be parametrized for different OpenCL tool flows and FPGA targets. It allows to trade-off different resource or platform limitations and overall achieves highly competitive performance. To achieve this flexibility, we used some straight-forward pre-processor macros and some that are tedious to maintain. These drawbacks, as well as some tool-induced performance limitations may easily be mitigated by future tool releases. Overall we conclude that the OpenCL-based FPGA design path can already produce fully functional designs with promising performance. Since typical optimizations for FPGA targets seem to be well suited for FPGAs from different vendors or device families of the same vendor, OpenCL allows a comparable platform independence inside the FPGA domain as inside the GPU domain, where investments into development of long-living simulation codes often hinge on such portability. When selecting the best platform for a specific problem, the FPGA domain offers interesting trade-offs. We partially covered logic, DSP, and on-chip memory resources, as well as off-chip memory bandwidths. Speed grade and power consumption of employed FPGAs, as well as off-chip memory technologies and sizes can offer further alternatives.

In future work, we would like to explore such platform alternatives like FPGA boards with hybrid memory cube (HMC) or cache-coherent coupling of FPGAs to CPU host memory. As the presented designs fit for grid sizes of up to 4096 width present a compromise between flexibility and highest efficiency, we also intend to study in more detail the trade-offs between fixing simulation parameters at synthesis time and leaving them open until program runtime.

Acknowledgment

This work was partially funded 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), by the German Federal Ministry of Education and Research (BMBF) within the collaborative research project "HighPerMeshes" (011H|160054) and supported by Xilinx under the Xilinx University Program (XUP).

References

[1] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio. On how to accelerate iterative stencil loops: A scalable streaming-based approach. *ACM Transactions on Architecture and Code Optimizations (TACO)*, 12(4):53:1–53:26, Dec. 2015.

[2] H. Giefers, C. Plessl, and J. Förstner. Accelerating finite difference time domain simulations with reconfigurable dataflow computers. *SIGARCH Computer Architecture News*, 41(5):65–70, June 2014.

[3] N. Hemsoth and T. P. Morgan. *FPGA Frontiers – New Applications in Reconfigurable Computing*. Next Platform Press, 2017.

[4] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proc. Workshop on Memory System Performance and Correctness (MSPC)*, pages 51–60, New York, Oct. 2006. ACM.

[5] T. Kenter and C. Plessl. Microdisk cavity FDTD simulation on FPGA using OpenCL. In *Proc. Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC), held in conjuction with Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[6] 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)*, 25(3):695–705, Mar. 2014.

[7] J. Vasiljevic, R. Wittig, P. Schumacher, J. Fifield, F. M. Vallina, H. Styles, and P. Chow. OpenCL library of stream memory components targeting FPGAs. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 104–111, Dec. 2015.

[8] 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)*, pages 1–6, June 2016.

[9] K. Yee. Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation*, 14:302–307, May 1966.