Energy Efficient Scientific Computing on FPGAs using OpenCL

Dennis Weller[†], Fabian Oboril[†], Dimitar Lukarski[‡], Juergen Becker[†], & Mehdi Tahoori[†]
[†]Karlsruhe Institute of Technology (KIT), [‡]PARALUTION Labs
Contact Email: {dennis.weller,fabian.oboril,mehdi.tahoori}@kit.edu

ABSTRACT

An indispensable part of our modern life is scientific computing which is used in large-scale high-performance systems as well as in low-power smart cyber-physical systems. Hence, accelerators for scientific computing need to be fast and energy efficient. Therefore, partial differential equations (PDEs), as an integral component of many scientific computing tasks, require efficient implementation. In this regard, FPGAs are well suited for data-parallel computations as they occur in PDE solvers. However, including FPGAs in the programming flow is not trivial, as hardware description languages (HDLs) have to be exploited, which requires detailed knowledge of the underlying hardware. This issue is tackled by OpenCL, which allows to write standardized code in a C-like fashion, rendering experience with HDLs unnecessary. Yet, hiding the underlying hardware from the developer makes it challenging to implement solvers that exploit the full FPGA potential. Therefore, we propose in this work a comprehensive set of generic and specific optimization techniques for PDE solvers using OpenCL that improve the FPGA performance and energy efficiency by orders of magnitude. Based on these optimizations, our study shows that, despite the high abstraction level of OpenCL, very energy efficient PDE accelerators on the FPGA fabric can be designed, making the FPGA an ideal solution for power-constrained applications.

1. INTRODUCTION

Scientific computing is an integral part of our modern life enabling for instance artificial intelligence, improved health care or accurate weather forecasts. Yet, nowadays scientific computing is not only used on large-scale high-performance servers but also in low-power domains due to the growing interest in cyber-physical systems including autonomously operating smart devices [1]. Consequently, there is a strong demand for fast but low-power accelerators.

A key component in many scientific computing domains are *Partial Differential Equations (PDEs)*. Solving PDEs efficiently is challenging and often requires numerical approaches based on the discretization of the problem space [2]. By this mean, the complex mathematical problem can be transformed into *systems of linear equations* (SLEs). These typically consist of millions of variables, and use special storage formats to avoid storing coefficients that are zero. As a result, solving these systems is computationally and storage-wise expensive [3], and a high memory bandwidth is key for high-performance solutions. However, due to irregular memory access patterns, efficient implementations w.r.t performance as well as energy demand are often challenging.

Traditionally, PDE solvers are implemented for many-core systems (CPUs), by dividing the problem space into several subdomains as well as by parallelizing the fundamental algebraic operations required by the iterative SLE solvers. In recent years also general purpose graphics processing units (GPGPUs) are gaining interest, and are now often used as accelerators, thanks to their superior performance and energy efficiency for massive data-parallel operations, such as the aforementioned algebraic operations [4].

Another group of accelerators are *Field Programmable Gate Arrays (FPGAs)*. Similar to CPUs and GPGPUs, FPGAs use the most recent manufacturing technologies, and with plenty configurable logic blocks they are well suited for data-parallel computations [5]. On top, even high-end FPGAs require less power than medium-class GPGPUs [6], and thus seem to be ideal candidates for power-constrained applications such as smart sensor systems[1]. Yet, FPGAs have been left out as accelerators for PDEs, so far. One reason is that writing efficient code for a heterogeneous platform using FPGAs has been very challenging, due to the lack of a universal programming framework. Consequently, separate codes had to be written for the host processor, for the accelerator and for the interfaces. This requires deep knowledge of the underlying FPGA architecture and use of hardware description languages, which is a major hurdle for software developers.

However, recently, both Altera and Xilinx, started to provide *OpenCL* support for their FPGAs [7, 8]. OpenCL is a framework for writing programs that execute across heterogeneous platforms [9]. By that means, OpenCL allows to write standardized C-like code for the host as well as for the accelerators, and thus relaxes the programming challenge for FPGAs. However, the increased level of abstraction makes it challenging to implement solvers at maximum performance or energy efficiency, as the developer has no direct influence on low-level characteristics such as resource

[1]E.g, a system steering an autonomous battery-powered vehicle based on sensor and camera input

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FPGA '17, February 22-24, 2017, Monterey, CA, USA
© 2017 ACM. ISBN 978-1-4503-4354-1/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3020078.3021730

usage, placement or timing constraints. Nevertheless, it was already demonstrated using OpenCL that FPGAs are very efficient platforms for DNA sequencing, FFT calculations, neural network implementations, image processing and option pricing [10–15]. While this is a great promise, the question of how well FPGAs are suited for solving PDEs efficiently using OpenCL remains unanswered.

Therefore, in this paper, *we propose a set of effective optimization strategies for fast and energy efficient OpenCL-based FPGA implementations of PDE solvers.* These include vendor-specific as well as vendor-independent techniques, data-set optimizations, algorithmic enhancements as well as data-flow and control-flow tuning to overcome the aforementioned implementation challenges for PDEs. The resulting implementations can be accessed at our project website [26]. In addition, we provide a *comprehensive study of the efficiency of FPGAs for solving PDEs using OpenCL* including a comparison with CPUs and GPGPUs. Our results of Xilinx and Altera FPGAs, show that our proposed optimization techniques improve performance and energy efficiency by orders of magnitude. As a result, very energy efficient solutions can be designed using OpenCL, despite the high abstraction level. For instance, our Altera FPGA is 2x more efficient than a quad-core processor, and 35% more efficient than a GPGPU with FPGA-like power consumption. Yet, conventional multi-core CPUs and high-performance GPGPUs still provide better performance. Nevertheless, thanks to the lower power consumption (≈ 30 Watt), FPGAs are the ideal candidates for power-constrained domains. Finally, the results also highlight fundamental differences among both manufacturers in terms of code implementation and optimization, as well as in terms of performance and efficiency.

The rest of this paper is organized as follows. The mathematical problem of our case study is introduced in Section 2, followed by a comprehensive discussion of the implementation for Xilinx and Altera in Section 3. Afterwards, the experimental results are presented and discussed in Section 4. Finally, Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 Partial Differential Equations

Solving PDEs is a core component in many scientific computing domains. Consequently, there is a diversity of PDEs. Nevertheless, solving PDEs follows typically the same main steps, which are problem discretization, transforming the PDE into systems of linear equations and finally solving these systems. Hence, findings for one PDE and one solver apply to many other cases [3].

As a case study for PDEs we use the Poisson's equation in this work. The Poisson's equation occurs in various application domains, e.g. in electrostatics and mechanical engineering. One well known problem described by this PDE is the calculation of the electric potential for a given charge distribution. The equation in two dimensions is:

$$-\Delta u(x,y) = -\left(\frac{\partial^2}{\partial^2 x} + \frac{\partial^2}{\partial^2 y}\right) u(x,y) = f(x,y), \qquad (1)$$

where *u(x,y)* is the unknown potential, *f(x,y)* is the known boundary condition and x and y are spatial coordinates. This PDE can be converted to an SLE by discretizing the continuous problem space with finite differences methods. The resulting SLE is:

$$A\,\vec{u} = \vec{b}, \qquad (2)$$

where A is the Laplace matrix, $\vec{b}$ represents the boundary condition and $\vec{u}$ is the unknown vector, which represents, in the case of electrostatics, the electrical potential. It is worth mentioning, that the Laplace matrix is sparse, containing only up to 5 non-zero entries per row for the case of a 2D problem (see Figure 1).

2.1.1 Conjugated Gradient Method

To solve this sparse SLE, iterative schemes like the Conjugate Gradient (CG) method are advantageous [16]. The CG scheme is one of the most powerful iterative solvers suited for symmetric, positive-definite and sparse matrices [16], which all are properties of the Laplace matrix. The CG method is also a good choice for a case study, as it has many similarities to other iterative solvers, like the multigrid method [3], as it contains many fundamental algebraic operations like basic vector operations. Hence, the results of this work, have great importance for other iterative solvers implemented using FPGA-based OpenCL platforms.

The idea of the CG algorithm is to update the current approximation of the solution by a new vector with respect to the A-orthogonal projection of the residual $r = b - Au$ [3]. The corresponding algorithm is depicted in Algorithm 1, where A is the input matrix and b is the right hand side of the system, the initial guess is given by vector u_0 and the residual is denoted by r. Moreover, the discrete L_2-norm of a vector r is given by $\|r\| := \sqrt{r^T r}$. Each iteration of the CG method gives a new approximate solution u_k where the stopping criterion is evaluated by means of the corresponding residual $r_k = b - Au_k$, which is implicitly calculated in Step 6 $r = r - \alpha A d$ (for more information see [3]).

2.2 OpenCL

Traditional methods to design FPGA-based accelerators involve register-transfer level descriptions, using hardware description languages like VHDL, Verilog or SystemC. Working with these languages is a time-consuming process, as they are akin to assembler languages and require detailed hardware programming knowledge as well as the understanding of underlying architecture for efficient use of hardware resources. To avoid such low-level programming languages, the biggest vendors of FPGAs, Altera and Xilinx, have recently released OpenCL frameworks with FPGA support [17]. Within these frameworks, it is possible to create high-level FPGA implementations without the requirement of deal-

Algorithm 1 CG algorithm

INPUT: A, b, u_0
1: $r_0 = b - Au_0$
2: $d_0 = r_0$
3: **repeat**
5: $\alpha_k = \dfrac{r_k^T r_k}{d_k^T A d_k}$
6: $r_{k+1} = r_k - \alpha_k A d_k$
7: $u_{k+1} = u_k + \alpha_k d_k$
8: $\beta_k = \dfrac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
9: $d_{k+1} = r_{k+1} + \beta_k d_k$
10: $k = k + 1$
11: **until** $\|r_{k+1}\| \leq \epsilon$
OUTPUT: u_{k+1}

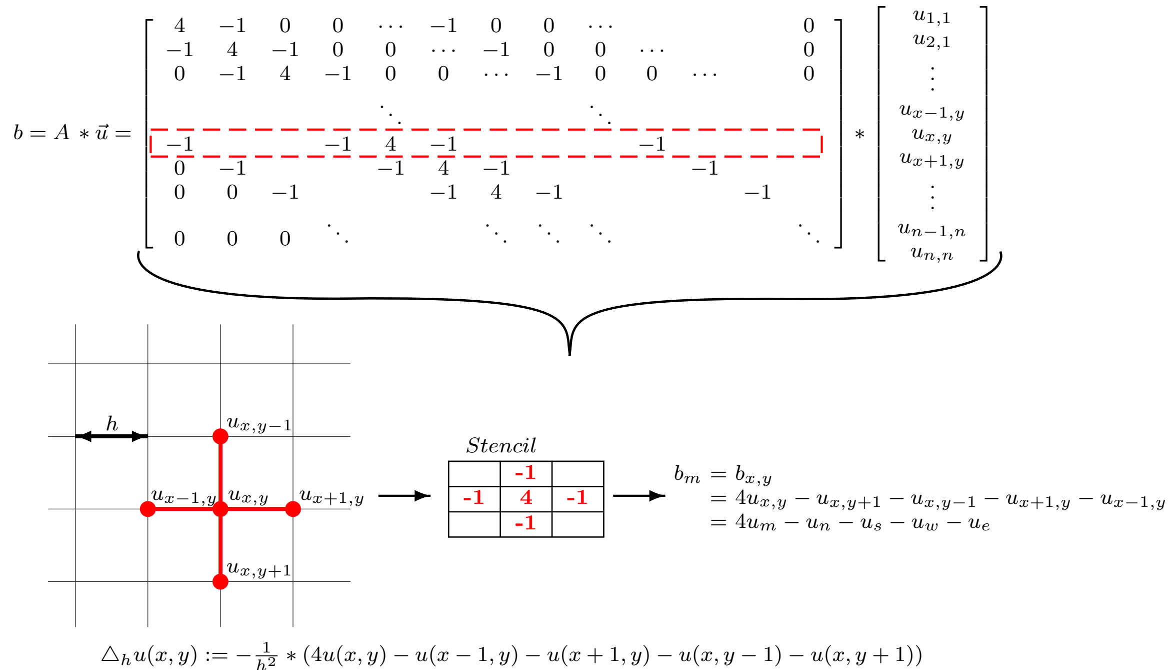

Figure 1: Laplace matrix (top) as the representation of the discretized solution for the Poisson's equation using a finite differences method. Each row is referred to the application of the discretized Laplacian operator on a grid point of the problem space (below), which takes the shape of a stencil with weights of -1 and 4.

ing with hardware description languages. Thus, investigations have been carried out, examining the potential of this new approach. As a result, in several application domains, FPGA implementations have been proposed as a mean to increase energy efficiency [10–15] (see Section 2.3 for details).

OpenCL specifies a C-like programming language, which enables the user to execute programs across heterogeneous platforms, through writing standardized code, without the need to refer to the underlying hardware. In order to implement designs, the OpenCL framework provides an application programming interface (API), which gives a host program running on a CPU platform, control over an accelerator. In the specific case of an FPGA accelerator, the host is able to load precompiled designs into the FPGA fabric, initiate data transfers and launch computations. In that regard it is important to note that software developers do not have to define the interfaces between host and accelerator, or between accelerator and memory. All interfaces as well as communication between host and accelerator are handled through the OpenCL framework. This is also a major advantage over other high-level synthesis approaches, as these focus only on the FPGA implementation. Because of that, in such high-level synthesis approaches, the software developers need to define all interfaces, and, more importantly, also need to implement all communication routines between host and accelerator manually.

Another major advantage of OpenCL is its task-based and data-based parallelism capability, potentially leading to designs with decent performance on a variety of platforms. Therefore, OpenCL partitions the original problem size into smaller junks (so called work-items), which represent single computation threads, dispatched to the accelerator and run in parallel. These work-items are grouped together into a work-group, whose size can be predefined. The size of a work-group is a design decision, and depends on the particular hardware structure (i.e. available resources) of the accelerator. Moreover, we observed that the optimal work-group size depends also on the FPGA vendors. For Xilinx, the optimal work-group size was one, whereas Altera uses work-group sizes related to the problem size. Section 3.2 discusses this issue in depth.

Another main difference between the Altera and Xilinx OpenCL SDK is the use of optimization techniques. They are described in the next section along with implementation details of the CG algorithm. However, apart from these differences, both vendors provide the same platform model, consisting of multiple kernels and compute units (CUs). Each kernel can process data and deploys (multiple) CUs for parallel computing. The kernel design has to be determined before runtime using ahead-of-time-compilation. In contrast, OpenCL-based CPU and GPU platforms use just-in-time-compilation at runtime. This is because CPUs and GPUs have fixed architectures, whereas FPGAs are reconfigurable and their configuration is a time-consuming process (including high-level synthesis, timing analysis, place-&-route) that is not suitable for runtime compilation. In fact, the compilation of the CG solver requires several hours and an extensive amount of memory, which makes the process of optimizing the implementation very time consuming and costly.

2.3 Related Work

As OpenCL is a new alternative to include FPGAs as accelerators, there exist only few studies on the efficiency of OpenCL-based FPGA implementations. Yet, there is an increasing interest, as OpenCL promises to significantly shorten the software development time. Recently, [11] presented an OpenCL compilation framework which generated high-performance hardware for FPGAs, paving the way for further studies. Among these was [15], which presented an OpenCL-based approach to use FPGAs as energy efficient data center accelerators. In [13], it was demonstrated, how the OpenCL design of a genome sequencing algorithm, im-

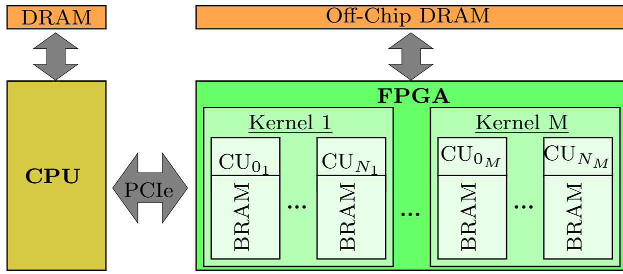

Figure 2: OpenCL platform with an FPGA accelerator: The CPU acts as host, provides the data for computation through the PCI-Express bus and controls the FPGA using the OpenCL-API. After the host has invoked the kernel execution the data is loaded from off-chip DRAM to local (BRAM) memory, and the FPGA starts processing the data.

plemented on a Xilinx FPGA, could surpass the performance and energy efficiency of CPU and GPU platforms. Further work has shown, how FPGA-based FFT accelerators can be realized [10]. The results supported the possibility of using FPGAs to achieve higher energy efficiency than GPUs under the means of OpenCL. Furthermore, [12] implemented Black-Scholes simulation for option pricing for Altera FPGAs, which were more energy efficient than comparable GPU platforms. Most recently, [14, 18] demonstrated that OpenCL also allows the efficient implementation of neural networks and sparse matrix calculations on FPGAs.

However, up to now, there is no work related to solving complete PDEs or optimizing the solver efficiency on FPGAs using OpenCL. For that reason, here we propose a comprehensive set of optimization schemes to improve the performance and energy efficiency of FPGAs for solving PDEs using OpenCL. In addition, we analyze the performance and energy efficiency of this implementation compared to traditional approaches for CPUs or GPGPUs.

3. OPTIMIZATION SCHEMES FOR FPGA-BASED PDE SOLVERS USING OPENCL

For solving the discretized Poisson's equation, the CG method is implemented on the FPGA using OpenCL. Essentially, three kernel functions are required for the complete CG algorithm (Algorithm 1), which are representative for many PDE solvers [3]:

- Scaleadd() (line: 1,6,7,9): $\vec{z} = a \cdot \vec{x} + \vec{y}$
- Dotc() (line: 5,8): $a = \vec{x}^T \cdot \vec{y}$
- LaplaceApply() (line: 1,5,6): $\vec{y} = A \cdot \vec{x}$

In a first step, each of these kernels is designed standalone, and optimized with respect to performance. As the power consumption of FPGAs is dominated by leakage power [19], performance improvements directly result in a better energy efficiency. Moreover, the performance is limited by the memory bandwidth[2]. Hence, it is necessary to achieve a throughput as close as possible to the theoretical peak bandwidth, to realize good performance and energy efficiency.

To reach the maximum memory bandwidth is very challenging, in particular on FPGAs using OpenCL (see Section 4 for details). Therefore, we propose in the following a set of optimization techniques which include data-set optimizations, algorithmic enhancements, as well as data-flow and control-flow tuning. Some of these optimizations can be applied to the kernel designs as annotations in the code, represented as specific directives, while others require code and data restructuring. Hence, the use of these techniques is a challenging task, in particular, as not all optimization schemes always result in higher but decreased performance. For instance, excessive resource utilization can lead to reduced clock speeds resulting in degraded performance.

Another optimization challenge is the integration of the three kernels into one combined design, to load them onto the FPGA via one single bitstream. This step is required, as switching between different bitstreams is inefficient due to the switching overhead of more than 400ms (for our kernel design) compared to kernel execution times that are in the range of 10-60 ms. However, the integration of all kernels into one design is not trivial, as all resources have to be shared among the kernels. This can cause timing errors or excessive resource utilization, which in turn can lead to reduced clock frequencies. As a result, it is challenging to maintain the standalone kernel performance, even if multiple kernels are integrated in one single bitstream. Thus, optimizations and design complexity have to be carefully traded-off to achieve the best performance.

In the following, our proposed optimization strategies are described. Some of them are vendor-independent approaches, while there are also a couple of vendor-dependent schemes, that are required, as both companies follow different principles for their OpenCL SDK.

3.1 Vendor-Independent Optimizations

The proposed vendor-independent optimizations apply to the Xilinx and Altera OpenCL SDKs. However, the exact implementation way may differ. In addition, some of them are supported by OpenCL constructs and specific vendor commands, while others (in particular avoidance of branches and irregular memory access) need to be applied manually.

- **Loop Unrolling**: Loop Unrolling is a very effective tool to improve the performance of loops on FPGAs with no data dependency. If a loop is unrolled N times, N loop iterations are run in parallel leading to a speedup of N-times in best case. This leads to an improved performance, but more FPGA resources are utilized. To enable this feature, one can use either the directive *__attribute__((opencl_unroll_hint(N)))* or implement the unrolling manually.
- **Loop Pipelining**: Like Loop Unrolling, this technique implements another form of data parallelism, where all sequential operations in a loop are kept busy at all times. Instead of waiting until the complete loop has finished computation, new data is requested after the current operation block in the loop was processed, as shown Figure 3. The performance is increased at the cost of additional resources required for storing intermediate results after each block. For Xilinx FPGAs Loop Pipelining can be triggered with the pragma *__attribute__((xcl_pipeline_loop))* and should be used whenever possible (see Figure 4), whereas the Altera compiler automatically employs this optimization.
- **Data Parallelism**: OpenCL enables the use of k-way single-instruction multiple-data (SIMD), which is supported by the FPGA vendors for a set of basic vector

[2]Caused by the external DDR3-DRAM memory interface of the FPGA boards. Maximum on Xilinx FPGA and Altera FPGA is 10.6 GB/s or 21.2 GB/s, respectively.

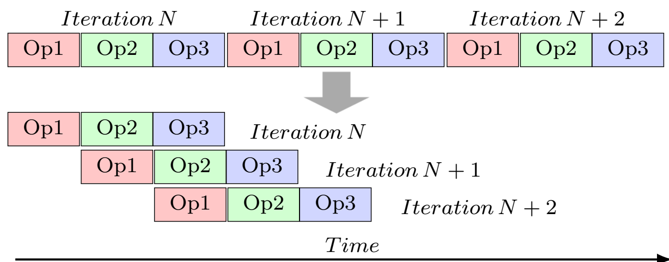

Figure 3: Loop Pipelining

operations like vector scaling. With vectorized processing, the throughput can be increased by a factor of k, which is typically 2,4,8 or 16. These k-way SIMD operations are induced by using vectorized data types. For instance, if 16-way SIMD is desired with floating point numbers, the data types have to be declared as float16. One float16 vector is the maximum data which can be transferred per clock cycle from the off-chip DDR3-DRAM memory to the on-chip local memory of the FPGA, as the related interface (for DDR3) is 512 bit wide (16 * 4 Byte (float) = 512 bit). Using vectorized processing widens the data path of the kernel as each operation is extended to process vectors instead of scalars, resulting in increased throughput but also resource utilization. Nevertheless, using float16 data types is key to achieve the best performance on FPGAs (see Figure 4 and Figure 5).

- **Replication of Compute Units**: The OpenCL framework supports the replication of compute units (CUs), which represent the implementation of a specific kernel function. Using this method, subsequent work-groups can be run in parallel instead of sequentially. As for the k-way SIMD technique, in the ideal case, the throughput can be increased by the number of replications. However, this only holds if the memory interface is not the performance bottleneck, otherwise the performance improvements are very limited. Moreover, using N CUs increases the resource utilization by more than N-times, due to additional control logic required to dispatch data to the different CUs. As a result, the number of CUs has to be carefully traded-off with resource demand. In case of PDE solvers, the *Dotc*-kernels highly benefit from using multiple CUs (see Figure 4 and Figure 5).
- **Dataflow-Driven Design**: The kernel implementations for FPGAs have to be designed under a dataflow-driven approach, using as few control structures as possible to maximize performance. These dataflow-driven designs are typically more efficient for FPGAs as control-driven approaches used for CPU implementations. The reason for this is that FPGAs do not have the sophisticated control-flow mechanisms of modern CPUs such as branch prediction or branch target buffers. As a result, the FPGA performance can be improved by 2x based on our observations, if branches/jumps are avoided. Thus, the use of control statements like "if-else" has to be restricted.
- **Regular Memory Accesses**: Due to the fact that the discretization matrix A is sparse, containing a huge number of rows and columns, sparse matrix formats like Compressed Sparse Row (CSR) are typically deployed for storage, as the traditional storage format would require terabytes of storage. Using these sparse formats leads, however, to irregular, unaligned and complex memory access patterns during the matrix vector multiplications. Since FPGAs do not have big caches these access patterns cause massive performance drops, which is a major challenge for the design of efficient PDE solvers for FPGAs. In order to avoid this undesirable behavior, we propose to implement sparse matrix vector multiplications in form of stencils. This means that indexes of the required vector elements as well as the coefficients with which these vector elements are multiplied are hard coded in the OpenCL kernel. This avoids loading the matrix elements and facilitates the access for the vector elements (for instance it allows to use float16 data types). While this increases the performance on one hand, it also decreases the flexibility of the kernel, as then only one particular stencil is supported (in our case the 2D-Laplace stencil shown in Figure 1) instead of an arbitrary matrix.

3.2 Vendor-Specific Optimization Techniques

The aforementioned optimization techniques are key to improve the performance and energy efficiency of Altera and Xilinx FPGAs. Yet, both vendors follow different design philosophies, and thus require also very specific optimization schemes on top of the vendor-independent techniques.

A first important difference between Altera and Xilinx is the usage of the concept of work-items. For GPU devices, hundreds of work-items are used to represent the execution threads, which are dispatched to different cores and run in parallel. While this concept leads to high-performance designs for the Altera OpenCL SDK, it is not efficient for the Xilinx SDK. The best performance for the latter is achieved using only one work-item per compute unit, which takes full control of the data processing and data transfers.

Another Xilinx-specific optimization is that burst transfers have to be used to transfer data between the on-chip BRAMs and the off-chip memory. The reason is that each of these transfers, controlled by the memory controller integrated in the FPGA, consists of an initiation phase and the actual transferring phase. The effort for the former phase is always constant, regardless of the size of the transferred data, producing some type of overhead. While this turned out to be no issue for Altera FPGAs, it considerably limited the performance of our Xilinx FPGA, where the initiation phase takes about 50-70 clock cycles [20]. As a result, it is very inefficient, to transfer small amount of data, as the overhead dominates. We found out that fast transfer rates can only be achieved using large burst sizes of 16KB and more, in case of the Xilinx FPGA.

As a consequence of these differences between the Altera and Xilinx OpenCL SDKs, separate kernel code has to be written for Altera FPGAs and Xilinx FPGAs, which limits the portability between both vendors. In the following, we explain our proposed kernel designs comprising all aforementioned optimization techniques for both vendors in detail. All OpenCL kernels can also be found on our website [26].

3.2.1 OpenCL Kernels for Xilinx FPGAs

For all three Xilinx kernels (*Scaleadd*, *Dotc*, *LaplaceApply*) the optimization approaches are similar with regard to burst transfers, float16 data type and loop pipelining. Therefore, we will explain them using the *Dotc*-kernel as an example.

As it can be seen in Listing 1 for the *Dotc*-kernel for Xilinx FPGAs, all computations are carried out in a work group with a single work item, as the required work-group size is set to $1 \cdot 1 \cdot 1 = 1$ (Line 2). The required data for processing needs to be loaded and stored with burst transfers, implemented as pipelined loops (Line 11,16). We observed, that without burst transfers, the performance decreases by of 100x to 1000x. As a result of the pipelined loops, in each iteration one float16 vector is loaded into local BRAMs, processed and stored. This concept is advantageous, as the external memory interface is 512 bit wide [20], which corresponds to one float16 vector[3]. Thus, the maximum possible bandwidth can be achieved, in the case of perfect pipelined loops. Consequently, the design challenge is to reach an initial interval of 1, meaning that the loop can request new data every clock cycle. However, in practice, only an interval of 3 can be reached. To compensate this drawback, multiple compute units are deployed (here: 4 CUs), to perform calculations in parallel, and thus improve the throughput.

For the other two Xilinx kernels (*Scaleadd*, *LaplaceApply*) the optimization approaches are very similar. However, the *LaplaceApply*-kernel poses some special challenges. First, the implementation of 16-way SIMD operations is not trivial. This is due to the fact, that despite of using the 2D-Laplace stencil, still irregular memory accesses exist (only five non-continuous vector elements are accessed per operation as shown in Figure 1). To resolve this issue, the stencil (i.e., $4m - s - n - e - w$) is implemented with the help of scalar products on float16-data as follows:

$$
\begin{aligned}
out &= \vec{m} \cdot (0, \ldots, 0, -1, 4, -1, 0, \ldots, 0) \\
out &+= \vec{n} \cdot (0, \ldots, 0, 0, -1, 0, 0, \ldots, 0) \\
out &+= \vec{s} \cdot (0, \ldots, 0, 0, -1, 0, 0, \ldots, 0),
\end{aligned}
$$

where $\vec{m}$, $\vec{n}$ and $\vec{s}$ are float16 input vectors for the different rows required by the stencil routine (see Figure 1). Then, each vector $\vec{m}$, $\vec{n}$ and $\vec{s}$ is processed multiple times using different right-hand vectors, which differ in the stencil position, i.e. in the places with the non-zero-elements to compute all output elements. Furthermore, to minimize data traffic, multiple rows are loaded at once to maximize reuse of data in the local memory. The second challenge of the *LaplaceApply*-kernel is that the boundary elements of the grid require a special treatment, which leads to many if-else statements in the kernel code. To eliminate these branches, the input data is modified by adding an additional row of zeros below the bottom row to the grid as well as one row of zeros the top of the grid. By that means, the extra handling of the bottom and top row is avoided.

As a result, this implementation is very demanding in terms of resource utilization. In particular, the available BRAMs can limit the performance, as it determines the size of burst transfers as well as the amount of data that can be reused locally for the stencil computations.

[3]float16 = 16 floats = 16 * 4 Byte = 16 * 32 Bit = 512 bit

Listing 1: OpenCL *Dotc*-kernel for Xilinx [26]

```
#define BSIZE 2048 //to load 128KB in each burst
__kernel __attribute__((reqd_work_group_size(1,1,1)))
void kernel_dotc(__global const float16 *x, __global
    const float16 *y, float out, const int nloops) {
  int gid = get_group_id(0); // use of 4 CUs
  float temp = 0.0f;
  __local float16 x_loc[BSIZE]; // local BRAMs
  __local float16 y_loc[BSIZE];

  for (int loop = 0; loop < nloops; loop++){
    __attribute__((xcl_pipeline_loop))
    for (int bid=0; bid<BSIZE; bid++) {
      //load data to local memory
      x_loc[bid] = x[bid+(loop+gid*nloops)*BSIZE];
    }
    __attribute__((xcl_pipeline_loop))
    for (int bid=0; bid<BSIZE; bid++) {
        //load data to local memory
        y_loc[bid] = y[bid+(loop+gid*nloops)*BSIZE];
        //Carry out dot-product and accumulate
        temp += dot(x_loc[bid],y_loc[bid]);
    }
  }
  out[gid] = temp;
}
```

3.2.2 OpenCL Kernels for Altera FPGAs

Using the Altera OpenCL SDK the concept of work-items is used differently, with multiple work-items per CU, resulting in completely different kernel designs as for Xilinx FPGAs. In general, the main idea behind our Altera kernels is that each work-item only processes one float16 element. To explain this further, the *Dotc*-kernel is used as example.

As it can be seen Listing 2, each work item (identified by gid in Line 6), processes only one float16 pair (Line 16), and stores the intermediate result in a local buffer. Thus, a reduction in a log-2-manner has to be carried out (Line 18-22), to obtain the final result. In comparison, the reduction is not required for the Xilinx kernel, which consists of only one work item which accumulates the intermediate results in each clock cycle (Line 20). Moreover, as it can be seen in the code, burst transfers are not required as explained before, while loop unrolling is performed (Line 14).

For the other kernels (*Scaleadd*, *LaplaceApply*) a similar implementation strategy is used. However, for the *LaplaceApply*-kernel, again some tricks are required to enable the usage of SIMD instructions, which, however, are different from

Listing 2: OpenCL *Dotc*-kernel for Altera [26]

```
#define BLOCK_SIZE 64
__attribute__((num_compute_units(4))) // use 4 CUs
// work-group with 64 elements
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void cdot(__global const float16 *x, __global
     const float16 *y, __global float *out, int size) {
  int gid = get_global_id(0);
  int lid = get_local_id(0);
  int group = get_group_id(0);
  int nl= get_local_size(0);
  __local float intermed[BLOCK_SIZE];

  // compute block-wise dot-products
  intermed[lid] = 0;
  #pragma unroll
  for (int i = 0; i<16; i++)
    intermed[lid] += x[gid][i]*y[gid][i];
  // perform a binary-tree reduction
  barrier(CLK_LOCAL_MEM_FENCE);
  for (int i = (min(BLOCK_SIZE,nl))/2; i > 0; i /= 2) {
    if (lid < i) intermed[lid] += intermed[lid + i];
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  //write result back
  if (lid == 0) out[group] = intermed[lid];
}
```

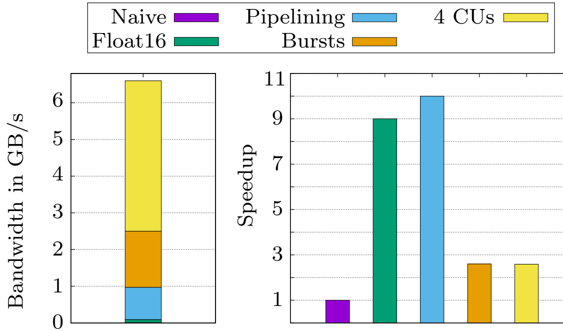

Figure 4: Performance optimization for the Xilinx *Dotc*-kernel. The optimization techniques are applied on top of each other in the given sequence.

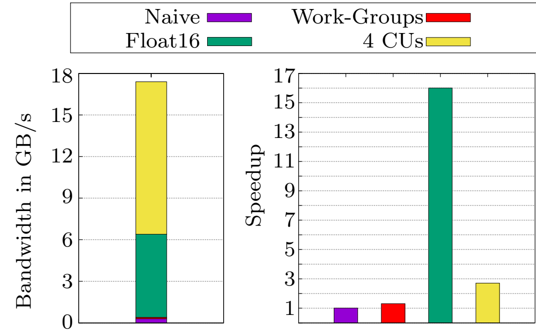

Figure 5: Performance optimization for the Altera *Dotc*-kernel. The optimization techniques are applied on top of each other in the given sequence.

those proposed for the Xilinx kernel. Instead of using scalar products, simpler vector-scaling routines are employed in combination with a clever data partitioning. First, to eliminate branches for the boundary elements, these are computed in separate kernels, which only compute the boundary elements. The runtime of these kernels is negligible in comparison to the main *LaplaceApply*-kernel, which computes the results for all points within the grid (see Figure 1). Hence, these kernels should be designed to consume as less resources as possible, to not induce restrictions for the other kernels. Second, all computations within the main kernel are performed using SIMD instructions. Therefore, the rows $\vec{n}$ and $\vec{s}$ are loaded as float16 vectors, while $\vec{m}$ contains now 18 float elements to include the left and right neighbors. Then, the output vector (again float16) is calculated as

$$\vec{out} = 4 \cdot \vec{m}[1:16] - \vec{s} - \vec{n} - \vec{m}[0:15] - \vec{m}[2:17].$$

Third, to minimize data transfer and maximize data reuse within the main *LaplaceApply*-kernel, each work-item calculates four float16 output vectors, namely two neighboring float16 packages for two adjacent rows. Processing more than four packages was not possible for our Altera FPGA due to resource limitations (available logic elements).

3.3 Impact of the Optimization Techniques

To highlight the potential of our proposed optimization schemes, we implemented the *Dotc*-kernels using different optimization strategies. As it can be seen from the corresponding results in Figure 4 for our Xilinx FPGA, a speedup of 620x can be achieved compared to the naive design using all techniques. Moreover, it is also noticeable that all schemes are necessary to obtain the best performance, as using SIMD instructions improves the baseline performance by 9x, loop pipelining adds another 10x, and using memory bursts as well as multiple compute units brings another 2.5x each. In this regard it is important to note that the sequence matters with which the optimization techniques are applied. In other words, using a different sequence of these optimization schemes will change the speedup values of each approach, however, the final combined value will remain the same. In addition it is worth to note that the results clearly show that the theoretical optimal speedup is often not achievable (e.g. 16x for float16, or 4x with 4 CUs). This is due to dependencies in the control and data flow as well as due to timing adjustments during the compilation process, which lead to different clock frequencies for different designs containing different optimization levels.

The performance of the *Dotc*-kernel for the Altera FPGA also shows significant improvements using the proposed optimization schemes. Yet, the overall speedup is just 59x. The reason, why the Altera version benefits less than the Xilinx implementation is that the Altera OpenCL compiler already optimizes the naive implementation using for instance pipelining. Consequently, the naive kernel achieves a bandwidth of 300 MByte/s, 30x more than the Xilinx version. Again, a very effective measure is to use multiple CUs and float16. In addition, it is very important to use appropriate sizing for the work-group sizes, as only in this case float16 and multiple CUs can be as effective as illustrated. For instance, with a work-group size of 512 instead of 64 (see Listing 2) the performance drops from 17.7 GB/s to 14.6 GB/s.

4. RESULTS

4.1 Hardware Configuration

To evaluate the performance and efficiency of FPGAs, we use Xilinx (ADM-PCIE-7V3) and Altera (Terasic DE5-NET) PCIe-Boards and compare them against CPUs and GPGPUs. In this regard, we selected a low-power GPGPU using conventional DDR3-RAM (Intel HD Graphics) as well as a high-performance model equipped with very fast GDDR5-RAM (Nvidia GTX 980) to have a comprehensive analysis. The used platforms are listed in Table 1. For the CPU platform, OpenMP was deployed, which is an implementation of multithreading, capable of assigning computation threads to different processors to achieve parallelism and consequently increasing throughput. The Intel HD Graphics was accessed through the OpenCL framework, same as for the FPGAs (Xilinx SDAccel 2016.1 and Altera Quartus 15.1). In contrast, for the GTX 980, CUDA (v7.5) was used, which is, akin to OpenCL, a parallel computing platform. For the GPGPUs and the CPU, we used the Paralution library [21], which provides very efficient CG implementations. The implementations of the OpenCL kernels for the FPGA accel-

Table 1: Listing of platforms used for benchmarking of the CG method

	Platform	Specification	API	Theoretical Memory Bandwidth	Technology
CPU	Intel Core i5-4590	4 cores @ 3.3GHz	OpenMP	21.2 GB/s (dual-DDR3-1333)	22nm
GPU	Intel HD Graphics 4600	20 exec. units @ 1.15 GHz	OpenCL	21.2 GB/s (dual-DDR3-1333)	22nm
	Nvidia GeForce GTX 980	2048 CUDA Cores @ 1.1 GHz	CUDA	224 GB/s (256-bit GDDR5 @3.5 GHz)	28nm
FPGA	Xilinx Virtex 7	XC7VX690T @ 200 MHz	OpenCL	10.6 GB/s (DDR3-1333)	28nm
	Altera Stratix V GX	5SGXEA7 @ 300 MHz	OpenCL	21.2 GB/s (dual-DDR3-1333)	28nm

Table 2: **Standalone kernel performance in GB/s.**

	Naive		Optimized		Speedup	
	Xilinx	Altera	Xilinx	Altera	Xilinx	Altera
ScaleAdd	0.06	3.6	9.1	17.5	1492x	5x
Dotc	0.01	0.3	6.6	17.7	623x	59x
LaplaceApply	0.03	0.01	6.6	11.5	2000x	1150x

erators are published under [26]. All these devices are run on the same hardware platform, which uses an Intel Core i5-4590 as host CPU. More details about the specifications can be found in [22, 23]. All measurements were carried out with a vector size of 2^{26} (≈ 67 million), which is a reasonable problem size for discretized PDEs.

4.2 Performance Measurements

The performance of all kernels within the CG solver is limited by the available memory bandwidth, which itself is constraint by the maximum transfer rate of the employed memory devices. Thus, as a comparable measure, the ratio of achieved bandwidth to maximum bandwidth was used and expressed in percentage.

Using the aforementioned optimization methods, the performance of the Xilinx standalone kernels surpassed 6 GB/s, while more than 11 GB/s were achieved on the Altera board. In this regard, Table 2 lists the speedup between naive and optimized kernel designs. As it can be seen from the data, our proposed optimization strategies have a significant effect and boost the performance by orders of magnitude. However, neither for Altera nor for Xilinx, these standalone kernels could be combined into one single bitstream due to resource and timing constraints. Yet, having a single bitstream is essential to enable high performance, as switching between different bitstreams is inefficient (please see Section 3 for more details). Therefore, the designs of the *LaplaceApply*-Kernel had to be simplified. In case of Xilinx the compute units were reduced from 3 to 1, as otherwise not enough BRAMs were available for the other kernels. For Altera the kernel was modified to process only one float16 package instead of four (see Section 3.2.2), as otherwise the resource demand is that high (more than 80% of all logic elements), that the clock frequency drops from 300 MHz to less than 250 MHz which causes performance penalties for the other kernels and results in an overall worse performance.

Table 3 contains the throughput of each kernel (as part of a single bitstream in case of the FPGA platforms) in comparison between the platforms (see Table 1 for details). The Xilinx kernels are inferior to the Altera kernels, due to the fact that the available bandwidth is smaller (single-channel vs. dual-channel[4]), and some optimization strategies had to be discarded for the *LaplaceApply*-kernel, as just discussed. The Altera FPGA delivers a performance in the same range as the Intel Core and the Intel Graphics platforms. However, as expected, it could not beat the Nvidia GTX 980, which was superior in all kernel computations.

For all platforms, the *LaplaceApply*-kernel had the lowest throughput because of the irregular memory accesses. In contrast, the *Scaleadd*-kernel achieved the best performance due to its simplicity, except for the Nvidia and Altera platforms. A possible explanation, why these platforms achieve a better performance executing the *Dotc*-kernel is that this kernel requires only read operations, while the *Scaleadd*-kernel uses read and write memory accesses. In addition, it is obvious that the FPGAs have more problems with the *LaplaceApply*-kernels compared to the other platforms. This is due to the fact that these kernels require more control-flow operations as well as less regular memory accesses. CPUs and GPGPUs can handle these challenges much better due to their sophisticated microarchitectures and caches.

4.3 Power Measurements & Energy Efficiency

In addition to the performance measurements, also the power consumption was measured in order to determine the energy efficiency for each kernel and platform. Two methods were used for this purpose: a power meter pluggable in outlets, and Intel CPU registers for the CPU power consumption [24]. Using these tools, the total power consumption can be divided into the following components:

- System idle: Contains the power consumption of the host computer with its CPU, the DRAM and all other peripheries excluding the FPGA (GPU) board and without running any computations.
- FPGA (GPU) idle: Is related to the power consumption of the FPGA board (GPU) plugged into the host computer when no computations are running. This is essentially the difference of the idle system power with and without accelerator.
- CPU computation: Is the power consumption of the CPU cores which are exclusively used for the CG computation tasks. This value is obtained by reading out the corresponding power monitor registers inside the CPU and by subtracting the CPU power consumption during idle (included in System Idle).
- FPGA (GPU) computation: Is the power consumption which is required to carry out the CG computation in addition to the FPGA (GPU) idle power consumption. The data is obtained by measuring the total system power while computations are running, and then the idle and CPU consumption is subtracted.
- Host memory: This is the power consumed by the required host memory to execute the CG solver. Therefore, the total system power is measured with four memory modules and then with only one module to determine the consumption per module. This is then used to obtain the total memory power consumption.

With the help of this classification we are able to break down the power consumption and estimate only the power

[4]Please note that this restriction does not apply to all Xilinx PCIe boards. For instance, the ADM-PCIE-KU3 supports also dual-channel memory access under OpenCL which should result in 2x of the performance of the ADM-PCIE-7V3 used in this study

Table 3: Throughput of the optimized kernels for each platform in GB/s and as percentage compared to the peak memory transfer rate. Moreover, the execution time for one iteration of the CG algorithm is given.

Vendor	Xilinx	Altera	Intel Core i5	Intel Graphics	Nvidia
Scaleadd	9.1 GB/s (85%)	17.5 GB/s (83%)	16.6 GB/s (78%)	17.9 GB/s (84%)	165 GB/s (67%)
Dotc	6.6 GB/s (62%)	17.7 GB/s (83%)	16.5 GB/s (77%)	16.5 GB/s (77%)	175 GB/s (73%)
LaplaceApply	2.8 GB/s (27%)	8.1 GB/s (38%)	11.2 GB/s (52%)	15.1 GB/s (71%)	143 GB/s (59%)
CG Runtime	710 ms	300ms	250ms	280ms	25ms

Table 4: Accelerator power consumption for the kernels of each platform in Watt. *A + B means: A is consumed by the accelerator incl. its memory, while the host CPU consumes B to manage the accelerator

Vendor	Xilinx	Altera	Intel Core i5	Intel Graphics	Nvidia
Scaleadd	35.3 + 18.7*	28.3 + 0.7	65 + 0	45.0 + 0.1	148.6 + 12.9
Dotc	35.8 + 18.7	27.5 + 0.7	66 + 0	45.4 + 0.1	153.6 + 12.9
LaplaceApply	37.8 + 18.7	27.8 + 0.7	71 + 0	45.6 + 0.1	230.1 + 12.9
CG	32.8 + 18.7	27.8 + 0.7	66 + 0	41.2 + 0.1	148.5 + 12.9

consumed by the CG computations as highlighted for the case of the *ScaleAdd*-kernel in Figure 6. The corresponding results for all kernels are presented in Table 4, which shows several interesting facts:

1. The FPGAs have the lowest power consumption (less than 30 Watt for the Altera board) making them ideal candidates for power-constrained applications. In comparison, the Intel Graphics performing similar to the Altera FPGA requires 45 Watt (including 15 Watt for its memory) and the quad-core processor has a 2x higher power consumption than the Altera FPGA. The most power-hungry platform is the Nvidia GTX 980, requires about 170 to 230 Watt for the kernels.
2. The consumption of the FPGA platforms is almost independent of the executed kernel, while there are significant differences for the CPU and Nvidia GTX 980. The reason for this behavior is that the power consumption of the FPGAs is dominated by leakage power, whereas dynamic power consumption is mostly responsible for the CPU and GPU power consumption.
3. Another interesting observation is that in case of the Xilinx platform the host CPU consumes a considerable amount of power (18.7 Watt), while this does not happen when the Altera board is used (0.7 Watt). This is because the host continuously polls the accelerator about the status of computation using the Xilinx SDK resulting in 100% load on two (out of four) CPU cores. In contrast, using the Altera FPGA for acceleration, almost no additional CPU power is required, as interrupts are used instead of polling, and this is more efficient in the case of long kernel runtimes[5]. Also Nvidia's CUDA solution uses polling, yet the CPU load is less, resulting in less power demand for the host CPU compared to the Xilinx solution.

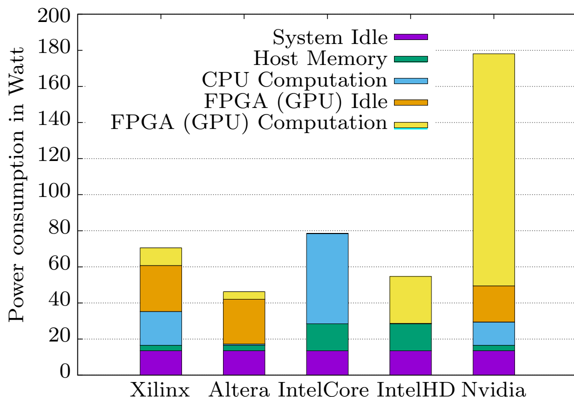

Figure 6: Power consumption of *Scaleadd*-kernel (Note: PCIe accelerators require less host memory as they come with their own memory)

As a result, the Altera platform has in this scenario the best energy efficiency of all platforms using conventional DDR3 memory (see Table 5), achieving a throughput of up to 630 MB/Joule and requiring only 8.5 Joule per CG iteration. This is more than 2x less than the quad-core CPU and 35% less than the Intel Graphics. Hence, if power consumption is constrained, for instance because of cooling issues that limit heat dissipation as in many mobile and IoT platforms, an OpenCL-based FPGA solution is preferable over a low-performance GPGPU considering energy efficiency as well as raw power consumption.

The Xilinx platform has a worse energy efficiency, which however is mostly due to the fact that our Xilinx FPGA uses a single-channel memory interface resulting in only half the memory bandwidth compared to the Altera platform, and that polling is employed by the Xilinx SDK. Only the Nvidia platform has an even better energy efficiency than the Altera FPGA. Yet, it also has memory that offers a 10x higher bandwidth compared to the Altera platform. Consequently, the results of the Altera FPGA are very good.

In this regard it is worth noting that the energy efficiency and performance of the FPGA solutions can be massively improved if faster memory is used instead of conventional DDR-memory. By that means, FPGAs could even become a possible rival for high-performance GPGPUs. For instance Altera's new Stratix 10 generation supports "High Bandwidth Memory" (HBM) [25], which offers transfer rates upto 1 TB/s. Consequently, the energy efficiency can increase by 10x and more. In addition, the recent FPGAs also include integrated multi-core ARM-like processors, which can be exploited for computations like the reduction inside *Dotc*-kernel which do not perform that well on the FPGA. Nevertheless, compared to conventional CPUs, even the state-of-the-art FPGA solutions can offer a better energy efficiency (please note as before that the Xilinx solution has a poor energy efficiency due to the low performance caused by the single-channel memory interface, whereas all other solutions use multiple memory channels).

[5] Please note that for very short kernel execution times in the order of a few μs polling offers a performance advantage over interrupts. However, as our kernels require several ms for the computation, this advantage is negligible.

Table 5: Energy efficiency of each platform and kernel in MB/J and the resulting energy for one CG iteration.

Vendor	Xilinx	Altera	Intel Core i5	Intel Graphics	Nvidia
Scaleadd	169	604	256	438	1025
Dotc	105	628	252	400	1052
LaplaceApply	50	285	158	364	587
Joule/iteration	33.9	8.5	20.2	11.5	5.1

4.4 Utilization

One important parameter for the OpenCL designs on FPGAs is the resource utilization. A design is not configurable if resource constraints cannot be met. As shown in Table 6, the utilization of the Altera FPGA is about 50% for the logic elements. For the Xilinx FPGA, the utilization is in the same range, apart from the BRAMs, which are required for the burst transfers. Thus, efficient designs on the Xilinx FPGA have to consider this by distributing the number of BRAMs to the three kernel in an optimal way.

4.5 Portability of OpenCL Kernels

OpenCL enables the user to write code once and deploy it on different devices without modifying the code. In this work, this point has been examined by porting OpenCL programs between Xilinx SDK and Altera SDK. Our investigations clearly show that none of the used kernels can be ported efficiently without making significant modifications in the kernel code. This is due to the fact that each optimization technique applied, on the one hand increases performance, but on the other hand decreases the flexibility to reuse the code on other platforms. And second, a problem arises when porting kernels designed for CPU- or GPU-based platforms to FPGA-based platforms. This is because there exist different compilation policies. On CPU- and GPU-based platforms the kernel code is compiled at runtime and thus each kernel can use all resources, while FPGA-based platforms force the user to carry out offline compilations and share resources among all kernels. Therefore, utilization of resources must be determined before runtime which affects the way the kernel code is designed. Thus, portability of OpenCL code is only possible to a limited extent.

5. CONCLUSION

Scientific computing is of great importance for our modern life standard enabling high quality health care, ever improving artificial intelligence and smart cyber-physical systems. Consequently, fast and energy efficient approaches for scientific computing are required, in particular for power-constrained application domains including for instance smart sensor systems. This requires appropriate hardware accelerators, especially for solving partial differential equations (PDEs) which are an essential element of many scientific computing tasks. In this paper, we evaluated the advantages of using FPGA-based accelerators for solving PDEs using OpenCL. OpenCL allows to perform all necessary implementations in a C-like language rendering knowledge of the underlying hardware as well as hardware description languages unnecessary. This, however, makes it also very challenging to exploit the full FPGA potential and design fast yet energy efficient solvers. To tackle this challenge we proposed a comprehensive set of optimization techniques including data-set optimizations, algorithmic enhancements as well as data-flow and control-flow tuning schemes that improve performance by orders of magnitude, and thus enable designers to take full advantage from FPGAs. Yet, our comparison of Altera and Xilinx FPGA implementations show that efficient OpenCL code requires fundamentally different optimization approaches for Altera and Xilinx. As a byproduct, portability of OpenCL designs between Altera and Xilinx, is not given and does not conserve high efficiency. Nevertheless, the comparison with CPU- and GPGPU-platforms also shows that FPGAs are more energy efficient than conventional CPUs in solving PDEs and for power-constrained systems FPGAs deliver competitive performance to GPUs and offer even a better energy efficiency. Hence, FPGAs are the ideal solution for power-constrained PDE accelerators.

Table 6: Utilization of FPGA resources for CG

Vendor	LUT	Registers	BRAMs	DSPs
Altera	50%	25%	26%	31%
Xilinx	35%	15%	65%	18%

6. ACKNOWLEDGEMENT

The authors would like to thank Vinay Singh and Herve Ratinger from Xilinx, Nico Trost from PARALUTION Labs as well as Peter Figuli and Leonard Masing from KIT for their valuable inputs for this work.

7. REFERENCES

[1] D. P. Möller, *Guide to Computing Fundamentals in Cyber-Physical Systems: Concepts, Design Methods, and Applications.* Springer, 2016.
[2] W. Schiesser, *The Numerical Method of Lines: Integration of Partial Differential Equations.* Elsevier Science, 2012.
[3] Y. Saad, *Iterative methods for sparse linear systems.* Siam, 2003.
[4] J. Bolz *et al.*, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *ACM TOG*, 2003.
[5] Altera, "A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next-Generation System Requirements," 2015.
[6] Nvidia, "Tesla K8 GPU Active Accelerator ," 2014.
[7] "Altera OpenCL SDK," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.
[8] "SDAccel - Xilinx OpenCL SDK," http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[9] J. E. Stone *et al.*, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, pp. 66–73, 2010.
[10] J. Andrade *et al.*, "From OpenCL to Gates: The FFT," in *Global Conference on Signal and Information Processing*, Dec 2013, pp. 1238–1241.
[11] T. S. Czajkowski *et al.*, "From opencl to high-performance hardware on FPGAS," in *Proceedings of FPL*, 2012.
[12] D. P. Singh *et al.*, "Harnessing the Power of FPGAs Using Altera's OpenCL Compiler," in *Proceedings of FPGA*, 2013.
[13] A. Sirasao *et al.*, "FPGA Based OpenCL Acceleration of Genome Sequencing Software," *System*, p. 11.
[14] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of FPGA*, 2016.
[15] J. Cong *et al.*, "Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper," in *Proceedings of ISLPED*, 2016, pp. 154–155.
[16] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," p. 1, 1994.
[17] F. Richter-Gottfried *et al.*, "Opencl 2.0 for fpgas using oclacc," *arXiv preprint arXiv:1508.07977*, 2015.
[18] H. Giefers *et al.*, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA," in *Proceedings of ISPASS*, April 2016, pp. 46–56.
[19] F. Li *et al.*, "Architecture Evaluation for Power-efficient FPGAs," in *Proceedings of FPGA*, 2003, pp. 175–184.
[20] Xilinx, "SDAccel Development Environment Methodology Guide: Performance Optimization," 2016.
[21] D. Lukarski, "Paralution-library for iterative sparse methods," 2015.
[22] "7 Series FPGAs Overview ," www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdfXC.
[23] Altera, "Altera Stratix-V family overview table," 2015.
[24] F. Oboril *et al.*, "High-resolution online power monitoring for modern microprocessors," in *Proceedings of DATE*, 2015.
[25] Altera, "Enabling Next-Generation Platforms Using Altera's 3D System-in-Package Technology," 2015.
[26] http://cdnc.itec.kit.edu/OpenCLFPGA.php.