Computer Physics Communications 249 (2020) 107029

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Towards Lattice Quantum Chromodynamics on FPGA devices☆

Grzegorz Korcyl [a,*], Piotr Korcyl [b,c]

[a] Department of Information Technologies, Faculty of Physics, Astronomy and Applied Computer Science, Jagiellonian University, ul. Łojasiewicza 11, 30-348 Kraków, Poland
[b] M. Smoluchowski Institute of Physics, Faculty of Physics, Astronomy and Applied Computer Science, Jagiellonian University, ul. Łojasiewicza 11, 30-348 Kraków, Poland
[c] Institut für Theoretische Physik, Universität Regensburg, 93040 Regensburg, Germany

ARTICLE INFO

Article history:
Received 8 October 2018
Received in revised form 4 September 2019
Accepted 4 November 2019
Available online 11 November 2019

Keywords:
High performance computing
FPGA devices
Lattice QCD calculations
Computer Science and Technology

ABSTRACT

In this paper we describe a single-node, double precision Field Programmable Gate Array (FPGA) implementation of the Conjugate Gradient algorithm in the context of Lattice Quantum Chromodynamics. As a benchmark of our proposal we invert numerically the Dirac–Wilson operator on a 4-dimensional grid on three Xilinx hardware solutions: Zynq Ultrascale+ evaluation board, the Alveo U250 accelerator and the largest device available on the market, the VU13P device. In our implementation we separate software/hardware parts in such a way that the entire multiplication by the Dirac operator is performed in hardware, and the rest of the algorithm runs on the host. We find out that the FPGA implementation can offer a performance comparable with that obtained using current CPU or Intel's many core Xeon Phi accelerators. A possible multiple node FPGA-based system is discussed and we argue that power-efficient High Performance Computing (HPC) systems can be implemented using FPGA devices only.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

In the last years Field Programmable Gate Array (FPGA) devices started to pave their way into the realm of High Performance Computing (HPC), for examples see [1–3]. A well-known scientific application in this domain is Monte Carlo simulations of Quantum Chromodynamics (QCD), which are performed in the context of theoretical elementary particles physics. An iterative solver of a large sparse system of linear equations, the Dirac–Wilson operator, lies at the heart of such simulations and is the most compute-intensive kernel. In this work we describe our attempt to port the Conjugate Gradient algorithm [4] which is an example of the most naive of such iterative solvers to a single FPGA device and demonstrate that FPGA devices can compete with modern HPC solutions as far as such academic applications are concerned.

1.1. QCD Monte Carlo simulations

From an algorithmic point of view Monte Carlo simulations of Quantum Chromodynamics boil down to a numerical estimation of highly dimensional integrals. A set of complex numbers representing the values of basic degrees of freedom: gluon and quark fields, is associated respectively to each edge and point of the space–time lattice. An implementation of a Markov chain allows to construct representative candidates of possible configurations of gluon fields according to a desired probability distribution. Any function of fundamental quark and gluon variables can be evaluated on each such configuration and the estimate of the final result is obtained by taking an algebraic average over the estimates from all generated configurations. The larger is the set of available configurations the smaller are the statistical uncertainties and the more precise the final physical result is.

As long as the probability distribution depends only on gluon fields, the Markov chain update can be performed locally and efficient algorithms allow to overcome the critical slowing down reducing the computational complexity to a manageable level. Once the quark field dependence is included in the probability distribution, in each update step the determinant of the Dirac operator's matrix has to be estimated and the only applicable algorithm in such a situation is the Hybrid Monte Carlo algorithm [5,6]. Instead of the determinant it is more convenient to estimate the inverse of the Dirac operator using an appropriate iterative solver. In many practical cases 90% of computing time is spent in that solver. It is thus crucial to optimize and fully adapt that part of the code to the micro-architecture the simulations are running on.

☆ The review of this paper was arranged by David W. Walker.

* Corresponding author.
E-mail address: grzegorz.korcyl@uj.edu.pl (G. Korcyl).

https://doi.org/10.1016/j.cpc.2019.107029
0010-4655/© 2019 Elsevier B.V. All rights reserved.

1.2. Search for new architectures

In the physically most interesting cases the Dirac matrix can achieve dimensions of order $10^9 \times 10^9$ and therefore a distributed, high performance system is required to handle such a task in a reasonable amount of human time. Various HPC solutions are being used with a particular emphasis on fast interconnects as frequent local binary exchanges as well as global operations are required. On typical processors the single-node problem is memory bandwidth bound and hence a fast and large cache memory is desirable.

Since several years the HPC field witnesses a major shift in the paradigm of the micro-architecture employed, as more and more many-core processors and accelerators are being used. Modern FPGA devices are being considered as candidates for next generation of HPC solutions and some computing centers have already opened FPGA clusters to general public [7]. A dedicated pure FPGA cluster running in production mode for theoretical chemistry application was reported in [8]. It is natural to ask whether other traditional HPC applications, such as the Monte Carlo simulations of QCD, can benefit from these new computing resources. In order to answer that question we investigate the performance achieved on a FPGA device by the naive solver, the Conjugate Gradient algorithm. Its numerical complexity is directly proportional to the numerical cost of multiplying a vector by the Dirac matrix. This matrix–vector multiplication appears ultimately in any other more involved solver, so it is a valid benchmark, while keeping the setup as simple as possible.

1.3. Outline

The paper is composed as follows. We start by briefly introducing the basic ingredients of lattice QCD, the most important being the Dirac matrix which is defined in Section 2.1. For the sake of completeness we present the standard Conjugate Gradient algorithm in Section 2.2 and describe the structure of input and output data in Section 2.3. In Section 3 we provide a brief overview of the main advantages of the FPGA technology, highlighting concepts which are crucial from the high performance perspective. In Section 4 we describe previous work in this subject. In Section 5 we pay special attention to the way the part which runs on the host and the part implemented in the logic are separated. We discuss various ways of enforcing the parallelization of the main kernel function in the hardware in 6. Section 7, where we describe our methodology and the details of our runs, is devoted to benchmark results. We focus on the resource utilization (Section 7.1), problem size limits (Section 7.3) and time to solution benchmarks (Section 7.4). We end with conclusions and a discussion of further research directions in Section 9.

2. Monte Carlo simulations

2.1. Lattice Quantum Chromodynamics

Lattice QCD is defined on a four-dimensional grid of sites, typically with periodic boundary conditions, representing the discretized space–time. The basic ingredients are the *gluon (gauge) fields* described by four 3×3 complex, unitary matrices $U_\mu^{AB}(n)$, $A, B = 1, 2, 3$, $\mu = 0, \ldots, 3$ sitting on each forward edge emanating from each site of the lattice, and a *fermion (spinor) field* $\psi_\alpha^A(n)$ described by a 12 component complex vector living on the lattice sites, with $A = 1, 2, 3$ and $\alpha = 0, \ldots, 3$. The 12 components can be grouped into four sets of three, with the gauge matrices rotating each of three components independently.

The Dirac operator acting on a spinor vector is defined as [9] (for a textbook introduction see [10])

$$\begin{aligned}\psi_\alpha^A(m) &= D(m,n)_{\alpha\beta}^{AB}\psi_\beta^B(n) \\ &= \psi_\alpha^A(n) + \kappa \sum_{\mu=0}^{3} \Big[U_\mu^{AB}(n) P_{\alpha\beta}^{-\mu} \psi_\beta^B(n+\hat{\mu}) \\ &\quad + U^{\dagger,AB}(n-\hat{\mu}) P_{\alpha\beta}^{+\mu} \psi_\beta^B(n-\hat{\mu}) \Big],\end{aligned} \tag{1}$$

where

$$\kappa = \frac{1}{2}\frac{1}{m_q + 4} \tag{2}$$

and the convention that two indices are always summed over in implied. Moreover

$$P^{\pm\mu} = 1 \pm \gamma_\mu. \tag{3}$$

As implied by the notation, the $P^{\pm\mu}$ matrices act only on the α index of the spinor field, whereas the U matrices act only on the A index. The Dirac matrix is sparse as it involves only the nearest-neighbors interactions.

One usually exploits the fact that due to the specific structure of the $P^{\pm\mu}$ matrices, the two lower α components of the spinor are related to the upper two by a simple complex rescaling. Thus it is sufficient to multiply by the matrix U only the upper components, halving the number of required multiplications by the U matrix.

2.2. Conjugate gradient algorithm

The main problem is to solve a set of coupled linear equations described by the abbreviated equation

$$D_{\alpha\beta}^{AB}(n,m)\psi_B^\beta(m) = \eta_A^\alpha(n). \tag{4}$$

If D is Hermitian and positive-definite a simple iterative conjugate gradient method can be applied. The Dirac operator as defined in Eq. (1) satisfies the γ_5-hermiticity, $\gamma_5 D \gamma_5 = D^\dagger$, so one needs to solve for $DD^\dagger$ which is Hermitian and then multiply the solution by $D^\dagger$.

The conjugate gradient algorithm which we have implemented reads [4]

$\psi \leftarrow \psi_0$
$r \leftarrow \eta - D\psi$
$p \leftarrow r$
while $|r| \geq r_{min}$ **do**
  $r_{old} \leftarrow |r|$
  $\alpha \leftarrow \frac{r_{old}}{|D^\dagger p|}$
  $\psi \leftarrow \psi + \alpha p$
  $r \leftarrow r - \alpha D D^\dagger p$
  $\beta \leftarrow \frac{|r|}{r_{old}}$
  $p \leftarrow r + \beta p$
end while

All operations on the vectors r, p and ψ such as matrix–vector multiplication as well as the computation of the norm $|\cdot|$ involve a sum over the entire lattice. The number of iterations before the required residuum is reached depends on the condition number of the Dirac matrix which in turn depends on the set of all U matrices together with the quark mass m_q and some other parameters.

2.3. Input and output data

In a typical situation the set of U matrices for the entire lattice resides in the main memory (usually it is read from disk

in the initialization phase of the program). A particular physical question can be answered by choosing appropriate η vector. In the simplest situation it has a single non-zero entry set to one. The CG algorithm above returns then the vector ψ being the solution of Eq. (4). Depending on the circumstances the next computation can involve the same U matrices but a different η vector, or, more frequently, a completely new set of U and η variables.

In a multi-process implementation the lattice is divided into smaller hypercubes and each hypercube is associated to a process. Multiplication by the Dirac operator equation (1) is performed in each process independently and in parallel, while the global steps of the CG algorithm, such as the evaluation of the stopping criterion are executed by the master process. Hence, apart from send/receive operations between neighboring processes, frequent global operations such as gather and broadcast are required. Because of the nearest-neighbor structure of the Dirac operator a copy of the boundary values has to be maintained on adjacent processes. For physical reasons periodic or antiperiodic boundary conditions are imposed on the U matrices.

2.4. Single stencil calculation

The most elementary computational block is the evaluation of the single stencil, i.e. evaluation of the right hand side of Eq. (1) for a given, fixed value of the index n. From that equation, this involves eight U matrices and nine spinor fields from the neighboring lattice sites, which corresponds to $(9*24+8*18)*8 = 360*8 = 2880$ input bytes. The $U \times \psi$ matrix–vector multiplications require 1152 floating point operations for complex additions and multiplications. Adding to that the additional preparatory and final additions and multiplications, one obtains for the complete stencil 1464 floating point operations. Hence, the ratio of memory/computation calls is unfavorable, because a lot of data has to be loaded while not so many floating point operations are needed to perform the required matrix times vector multiplication. There is only a small overlap of input variables for a neighboring stencil.

3. Programmable logic devices advantages

In this section we briefly explain what are FPGA devices and highlight these of their features that are important for the main ideas presented in this paper.

FPGA devices are known in the industry and science for years but are most commonly related to specific, low-level, true real-time applications like digital signal processing, networking or interfacing between electronic components. This was due to the limited amount of primitive resources and low-abstract development methodologies that require manual designing at the level of single flip-flops known as Register-Transfer Level (RTL).

Nowadays it is no longer the case. Manufacturing process has reached 16 nm, a value similar to the one used for CPUs but first engineering samples of devices produced in 7 nm process have been presented, aimed to be publicly available by the end of the year 2018 [11]. The number of configurable logic blocks contained in a single device has quadrupled since 2012, what is incomparable to the performance increase of the CPUs during the same period of time. Not only the amount of resources but also the diversity and the complexity of hardware components that are supporting programmable logic arrays have increased. In a single package we can find more than hundred multi-gigabit transceivers (single link with rates up to 58 Gbps), more than 12 thousands Digital Signal Processing (DSP) blocks with accumulated processing bandwidth at the level of 21.1 TMACs and memory resources like distributed RAM blocks and up to 8 GB memory in a form of stacked silicon in a single 3D integrated circuit package with bandwidth of 460 GBps [12].

Development of the design for such a complex device at the RTL level becomes highly challenging. Advancement of the hardware came in pair with the introduction of high-level development environments such as Software Defined software family (SDx) and High-Level Synthesis (HLS) from Xilinx for instance [13]. Algorithmic parts of the design can now be written as C/C++ and compiled into RTL form using HLS. A set of detailed reports and versatile pragmas give the developer control over the translation process from high-level programming language into RTL.

Tremendous amount of resources and fast design methodologies yield a new and powerful component for HPC applications. Most significant change in the paradigm of how the application is being developed shifts from optimizing the algorithm to make the best use of a given hardware architecture to designing a hardware architecture that will most efficiently compute a given algorithm. Programmable logic presents a number of game-changing features that can make a strong impact on how the computing resources for HPC are being designed. The key ones are described below.

Natural parallelism is the most powerful aspect of the FPGAs. One can implement as many instances of some logic component as there are resources available. All those instances will run in parallel with respect to each other. For instance in Conjugate Gradient algorithm one can implement multiple instances of the module that performs matrix times vector multiplication, which is the most time consuming operation. This feature, when used properly, also yields well-scalable solutions. With the introduction of new devices with higher amount of resources, the overall solution can be easily accelerated by instantiation of additional modules.

Multi-gigabit transceivers are embedded within the FPGA package. Each received data word can be instantly accessed by some processing logic, without interrupts, memory transfers or engaging operating system. The latency from the transfer medium to processing module is reduced to minimum. Keeping in mind the number of transceivers in most advanced devices and natural parallelization, when data from each link can be processed in parallel, this combination can result in a very efficient infrastructure for a remote computing node in future HPC architectures. We will develop further this idea in Section 8.

Programmable logic has no dependency nor overhead imposed by the operating system, which is required to execute computations on CPUs and GPUs. It is therefore also fully resistant to threats or malicious software that could degrade its performance or damage the system.

Logic resources used by a given algorithm components can be dynamically reconfigured to compute another algorithm or even released in order to reduce power usage. Although this process is rather slow (order of microseconds) and is not suitable for real-time applications, in HPC can be extremely useful for making the best use of available resources and power management.

4. Previous work

There is a continuous development of algorithms and implementations of Lattice QCD kernels for various micro-architectures and HPC solutions. The status is reviewed at the yearly Lattice conference and the most recent review can be found in [14]. Most recent progress in the implementations of state-of-the-art solvers on GPU and many-core Xeon Phi accelerators was reported in [15–17] and [18–21] respectively. Obtained performances are of the order of 300 [22]–900 [23] GFLOPs single precision for the 64 core Intel's Knights Landing architecture and

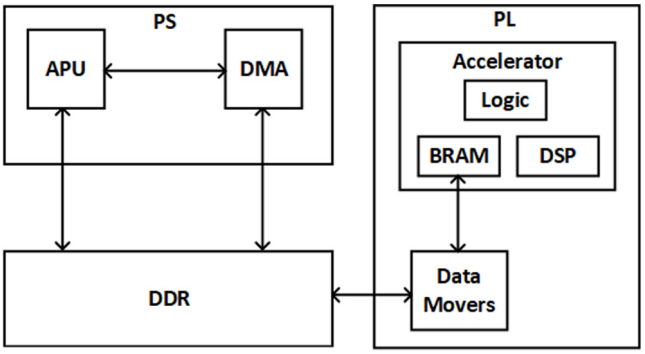

Fig. 1. Schematic view at the key components of Zynq MPSoC used to transfer data between PS and PL.

500 GFLOPs for double precision and 1.5 TFLOPs or even more for mixed precision on the NVIDIA Volta GPU [16] using multiple right hand sides optimization.

The first FPGA implementation of the conjugate gradient in the context of Lattice QCD was reported in 2005 and 2006 in conference proceedings [24] and [25] and in [26] in full detail. The Dirac operator was implemented using 2616 double precision floating point operations and the reported performance was up to 1.3 GFLOPs on the ADM-XRC-II development board featuring the Xilinx Virtex-II FPGA, XC2V6000FF1152. This research was not followed up in any way.

5. Design partitioning

It is important to carefully select which parts of the algorithm present the highest potential for hardware acceleration. Many considerations should be taken into account, therefore it is vital to understand the general mechanism of delegating computations to programmable logic.

We have chosen a Xilinx Zynq MPSoC (Multi-Processor System-on-Chip) as the device to evaluate the solution (Fig. 1). It is composed of two main functional blocks: Processing System (PS) which consists of ARM cores called Application Processing Units (APU) and its hardware entourage as well as Programmable Logic (PL) which are standard FPGA resources. Note that a similar partitioning exists for the Alveo accelerators with the CPU playing the role of the PS and the Alveo FPGA device playing the role of PL. In the following we concentrate on the former setup which we describe in detail.

The entry point of the design is the software that runs on PS. It executes the code of the algorithm and uses external DDR memory as RAM. When the code reaches a call to the hardware accelerator, the required data has to be transported from DDR to the PL. The software instructs the Direct Memory Access (DMA) controller to fetch particular dataset from the DDR and transport it to the PL through dedicated hardware interfaces. The data is routed inside the PL by the generated Data Movers to the accelerator that implements a particular software function. The computation results are returned in a reverse manner to the DDR memory and become accessible to the software on APU. Therefore, one should look for well isolated fragments that have high amount of computations on limited amount of data with no dependencies to other elements of the algorithm.

Additionally the decision about design partitioning and code structure is driven by the required dataset size and its access pattern from the accelerator: sequential or random. Requesting a data unit from the DDR by the accelerator is slow and inefficient if the entire dataset is large and multiple requests are needed from scattered addresses. For large datasets the DMA should be engaged, which after configuration, streams a particular block of data to the accelerator. Therefore the data in the DDR has to be arranged in contiguous manner and the accelerator has to contain local memory resources: Block RAMs (BRAMs) in which the received data could be stored allowing for random access to all data elements simultaneously. The cost of setting up DMA and transferring data is significant and therefore the partitioning decision should be driven by the goal of accumulating large computational blocks on blocked, non-scattered data. In that case one can profit from streamlined and parallel processing of a single data unit and minimize frequent data transport.

In the case of the CG algorithm there exists a natural candidate to be accelerated in hardware, which is the vector–matrix multiplication: the multiplication of the spinor field ψ by the Dirac operator $D^\dagger D$ in Eq. (4). In such partition the main part of the CG algorithm is executed on the APU, where the vector scalar products are evaluated and the stopping criterion is evaluated. Details of an efficient implementation of this idea are described in the next sections.

6. Software and hardware implementation

In this section we describe the details of our implementation. We start in Section 6.1 with some general remarks, then we describe the implementation of the data transfer between the DDR RAM and the logic. In Section 6.2 we describe the data partitioning within the BRAM blocks and computation parallelization in the logic in Section 6.3. In Section 6.4 we provide details on the most fundamental part of the kernel function, namely the single stencil computation. We finish this part with remarks on a way of increasing data locality in Section 6.5.

6.1. Generalities

In order to easily compare the achieved performance to the standard CPU architectures we impose double precision on all stages of the computation. This comes at a cost of requiring large logic resources to handle complex arithmetics of double precision numbers. A single addition or multiplication of two double numbers takes 14 clock cycles on the FPGA device used in this study. We note that modern GPU implementations of similar algorithms [27] use various combinations of low precision steps (single or half) in order to increase performance. The impact of this optimization should be investigated in more detail in the future but a quick inspection of the generated adder gives some insight. Three DSP hardware blocks are used in order to add two 64 bit floating point numbers. This comes from the fact that the bit-width of the inputs to the block is limited and the adder itself is limited to work on 48 bit values. Reduction of precision, so that the bit-width of the values fits the architecture of the DSP block would result in significant saving of the computing resources.

The great advantage of the FPGA implementation is the fact that we do not need to prepare data structures for contiguous memory access. Once the required data blocks are copied from the DDR memory to the memory blocks of the device data can be accessed in parallel and the matrix–vector multiplications executed simultaneously. Note that for instance the ordering of the real and imaginary parts in memory does not longer play a role, as all the appropriate reshuffling will be executed in hardware data routing, which is computationally costless. Hence no structure of arrays data layout optimization is needed. Note also that usual optimization required on modern CPU processors when the data had to be packed into vectors of the length of registers reaching 512 bits or 8 doubles, is no longer required. Vectors of arbitrary length can be processed in the hardware of a FPGA device with equal ease. In practice we work with abstract types `complex`, `su3_vector`, `su3_matrix` and `su3_spinor`:

```
struct complex {
public:
     double r,i;
     [...]
}
struct su3_vector {
public:
     complex v[3];
     [...]
}
struct su3_matrix {
public:
     complex m[9];
     [...]
}
struct su3_spinor {
public:
     su3_vector s[4];
     [...]
}
```

We experimented with a compressed data structure for the U matrix which is a unitary 3×3 complex matrix. The mathematical properties of U ensure that it is enough to store the first two rows, while the third row can be recomputed when needed [28]. Such solution alleviates the memory bandwidth limitation at the cost of requiring more compute resources. We found however, that the additional computations needed to restore the full matrix prohibit a full pipelining which is more disadvantageous than the gain from the reduced pressure on the data transfer. We defer the discussion of the impact of alternative parametrization of a single U matrix by 8 real numbers to the next publication.

In order to allow the compiler to take advantage of the natural parallelism of FPGA devices it is crucial to store data in PL in such a way that independent data is located in separate BRAM blocks. This limitation corresponds to the fact that in a single PL clock cycle only one memory element can be read from the BRAM block (hardware interface limitation). In the computation of a single stencil one needs eight different U matrices, ideally they should be stored in eight separate BRAM blocks. The set of eight U matrices has a natural partitioning according to the four values of the μ index. These four sets are fully independent. We further duplicate each of these arrays so that the forward and backward matrices, i.e. $n + \hat{\mu}$ and n arguments on Eq. (1) can be accessed simultaneously. We observe that without this improvement the interval of a single stencil evaluation increases dramatically because the computations await for consecutive reads of data from BRAM memory. We list the details of the BRAM blocks allocation in Section 6.3.

Apart from natural parallelism, FPGA devices offer pipelined processing. The full pipelining in computational block means achieving Initiation Interval (II) of one clock cycle, i.e. the hardware block can accept new input data at each clock cycle. The fixed structure of our computations, namely execution of exactly the same computations for each lattice site, allows for a fully pipelined Dirac matrix multiplication, if only enough logic resources are available on the device.

This is exactly what we observe in Section 7: the evaluation board which we used for testing does not possess enough memory blocks and hence II for the single stencil calculation block is much longer than one clock cycle. On the other hand, compilation on larger devices equipped with additional Ultra RAM memory (URAM large, embedded memory blocks) shows that full pipelining is possible. Details of the single stencil implementation are described in Section 6.4.

Hence, an efficient implementation of the CG algorithm on a single FPGA device is limited by the memory type and capacity available on the device. In the next section we describe the benchmarks obtained on the evaluation board where lack of large memory blocks dramatically limit the achievable performance. On contrary, the performance increases three orders of magnitude in larger devices with more memory.

6.2. Data transmission

The key issue in achieving high performance is the ability of quickly transferring data from the host memory to the accelerator. In the case of the FPGA devices data transfer may or may not be initiated at each call to the accelerated function.

When the U matrices do not change between consecutive iterations of the solver they can be loaded to the PL only once at the first function call and then kept in PL during the entire execution of the algorithm. This happens when the entire lattice fits into the memory blocks within the logic or when multiple devices are used and the lattice can be divided into blocks which do fit in the memory blocks. In the case when the entire lattice does not fit in the memory blocks within the logic it needs to be reloaded in order to enable the computation of all stencils. In either case, fast data transfer is crucial in achieving the shortest time to solution.

Several ingredients are needed to achieve fast data transfer. They include memory allocation on PS side and selection of data transfer mechanism to PL. In case the computations delegated to hardware are carefully isolated and do not require PS to interact before the final result is calculated, the dataset should be streamed from the DDR to PL at the beginning and back at the end. The fastest streaming can be achieved by assuring that consecutive physical addresses of DDR hold consecutive array elements and no coherency check mechanisms are required. Those two requirements are fulfilled by allocating arrays on the PS side using `sds_alloc_non_cacheable(x)` command. It has to be accompanied by a proper generation of the Data Movers in the PL, therefore the compiler has to be instructed how the data in the input arguments is organized and accessed. The attribute `PHYSICAL_CONTIGUOUS` of pragma's `SDS data` informs that the data is continuously ordered on the physical level and the attribute `SEQUENTIAL` states that the PL logic will access elements consecutively. Those instructions will cause the compiler to infer DMASIMPLE interface type, which allows the fastest streaming of large memory blocks between DDR and PL. In our case we declare the U matrices in the following way

```
#pragma SDS data mem_attribute( U_x_i_in:PHYSICAL_CONTIGUOUS, ... )
#pragma SDS data access_pattern( U_x_i_in:SEQUENTIAL, ... )
```

and call the accelerated function providing them as arguments

```
void multBatch(double U_x_i_in[vol], ...
```

One has to take care that all transfer channels between the DDR and PL are used. The evaluation board Xilinx ZU9EG has 8 data channels, whereas the hardware accelerated function has in principle only 6 input arguments: ψ_{in}, ψ_{out}, U_x, U_y, U_z and U_t. Therefore just for that reason we separated the real and imaginary parts of our input data and sent them to the logic, thus utilizing the full bandwidth.

6.3. Memory blocks allocation

In order to achieve full pipelining the data transferred from the DDR has to be stored in the BRAM blocks in an appropriate way. All calls to memory in the accelerated function have to be inspected. The crucial point here is that one has to make sure that computations which operate on different data and therefore can be executed in parallel have access to the required memory blocks. One has to ensure that all data needed at a given clock cycle reside in different BRAM blocks or registers so that they can be read simultaneously.

A single BRAM has a fixed data width and capacity, defined by the hardware architecture of the device family. For instance Zynq Ultrascale+ has blocks with 36 kBits capacity and width of 72 bit data words. When storing 64 bit values of the type `double` 8 bits are left unused and inaccessible per entry. Similarly, the remaining capacity cannot be shared with other arrays. Therefore it is important to properly arrange memory resources with the provided mechanisms that allow to instruct the HLS compiler to generate advanced BRAM structures, while letting the developer use standard `array` abstracts. Pragma `HLS ARRAY_PARTITION` is the simplest way to receive more hardware interfaces to access multiple array elements in parallel by dividing one large array into many smaller with the content distributed to more BRAM blocks, each with its individual interface. Complete partitioning means that all N elements of a given dimension of an array will be distributed into N separate memory instances (BRAMs or registers), each accessible at the same time.

Similar mechanism is employed by `HLS ARRAY_RESHAPE` pragma. Data can be distributed to multiple memory resources but additional vertical control is possible, which allows for a better use of the fixed 72 bits data width. Let us consider a 2-dimensional array of integers (32 bit), where one dimension has size 2. Complete partition of this dimension will generate two separate arrays, housed in two BRAMs. This will result in parallel access however also in a waste of resources because in each array only 32 bits out of 72 bits available will be used by a single element. Complete reshaping of the same array will result in a better arrangement of the elements, by reordering them vertically, so first all available bits of the memory block are used and then additional BRAM is used. In our example, two elements of 32 bits can be packed into one 72 bit data word and therefore only one BRAM is required while still preserving access to both elements at the same time. (See Fig. 2.)

In our case we declare the storage of the U matrices on the PL side as (we separate the real and imaginary parts for the reasons of transmission performance, see Section 6.2)

```
double U_x_i[2][vol][9];
double U_x_r[2][vol][9];
```

and we enforce on these matrices the following properties

```
#pragma HLS array_partition variable=U_x_i complete dim=1
#pragma HLS array_reshape variable=U_x_i complete dim=3
#pragma HLS resource variable=U_x_i core=XPM_MEMORY latency=1 uram
```

which ensures that the two copies of the array are stored separately (complete partitioning in the first argument) and the elements are reordered on the fly according to the third argument, i.e. color indices. The last pragma enforces the variable to be placed in the large URAM memory banks.

We have designed a PL memory infrastructure in a way that all required elements are accessible at a given clock cycle by the computing kernel. Set of reports generated by the HLS tool allows to identify bottlenecks in memory access that stalls computation flow. Using both memory management pragmas on selected arrays and applying data duplication we have achieved II = 1 on devices that contain URAM blocks.

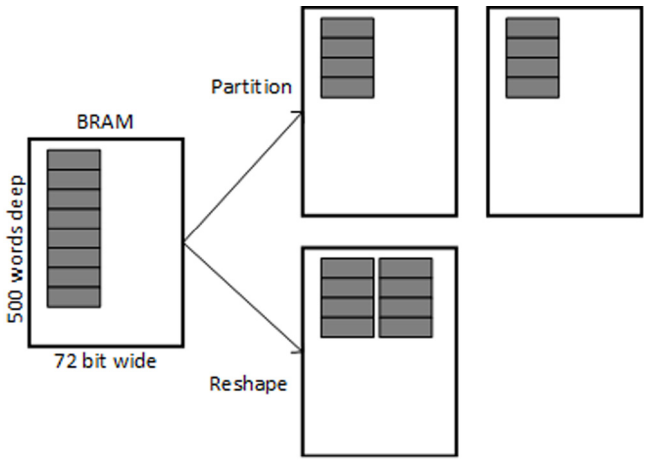

Fig. 2. Various schemes of array data arrangement within BRAMs.

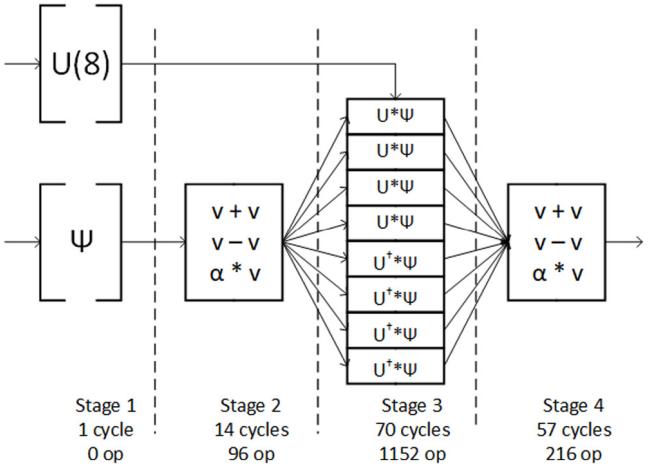

Fig. 3. Computation sequence of the stencil solver.

6.4. Stencil evaluation implementation

Single stencil evaluation is implemented as a separate hardware function composed out of 4 main processing stages (Fig. 3). We note that addition and multiplication of two double type variables consumes 14 clock cycles on the VU13P device. This number can differ for different devices and depends on other compile-time constraints.

First (Stage 1) we copy all the necessary data from the BRAM memory blocks to local registers which we declare with the `ARRAY_PARTITION complete` attribute. The process requires only one clock cycle to collect all needed data. It is important to separate all variables so that they can be accessed simultaneously, however we can stick to our abstract types such as the `su3_matrix` type,

```
su3_matrix link[8];
#pragma HLS ARRAY_PARTITION complete variable=link dim=1

su3_vector u[8];
#pragma HLS ARRAY_PARTITION complete variable=u dim=1
```

Table 1
Resource consumption versus parallel execution. Abbreviations in the column names have the following meaning: BRAM — Block RAM, DSP — Digital Signal Processing slice, FF — Flip-Flop unit, LUT — Look-up Table unit, URAM — Ultra RAM. The percentages assigned for the ZU9EG device and Alveo refer to the fraction of the total resources used for the project. The last column indicates whether the resources were consumed by the kernel only or by the full solution.

Device	Latency	Interval	BRAM	DSP	FF [10^6]	LUT [10^6]	URAM	Compiled unit
XCVU13P	142	1	508	6960	1.58	0.99	696	Kernel
XCVU13P	151	2	448	4320	0.97	0.64	696	Kernel
XCVU13P	151	2	428	3480	0.83	0.57	696	Kernel
XCVU13P	162	4	412	1740	0.47	0.35	696	Kernel
ALVEO U250	138	1	612 (30%)	8328 (72%)	1.13 (41%)	0.74 (55%)	696 (54%)	Kernel
ALVEO U250	138	1	774 (39%)	8332 (72%)	1.28 (46%)	0.84 (62%)	696 (54%)	Full
ZU9EG	250	120	1388 (76%)	546 (21%)	0.13 (24%)	0.09 (34%)	–	Kernel
ZU9EG	250	120	1735 (95%)	546 (21%)	0.17 (31%)	0.11 (40%)	–	Full

In Stage 2 we prepare linear combinations of input data. It requires 8 additions and 8 subtractions of `su3_vector` type, that is 96 basic operations on `double`. They are all performed in parallel, therefore the Stage 2 takes 14 clock cycles to provide the result.

Most operations are accumulated in Stage 3, where `su3_matrix` and `su3_vector` multiplications are performed. Decomposing to basic data type, they require 1152 operations on `double`. The matrix times vector operation is executed in 3 steps: first all nine complex by complex multiplications are evaluated in 28 clock cycles which correspond to a multiplication and an addition of doubles. Then the summation of three complex numbers is performed in 28 cycles corresponding to two double additions. Finally, a rescaling by κ requires another 14 cycles, corresponding to a single multiplication of doubles. In total a 5-layer operation cascade is generated and the latency to compute the output `su3_vector` takes $28+28+14 = 70$ cycles. Note that this is a minimal number of cycles required to analogously perform the multiplication of two `su3_vector`'s, which means that the entire matrix structure is hidden by parallelization. Note also that once the matrix U is in the registers we use it to perform, in parallel, the multiplication of two `su3_vector`, which belong to a single spinor $\psi(n)$. To evaluate the single stencil, 8 such multiplications are required. By using the completely partitioned arrays and instructing the HLS compiler that there are no dependencies between those arrays, we can unroll this loop completely given the required amount of resources and the following instructions are executed in parallel

```
for (int i = 0; i < 4; i++) {
#pragma HLS dependence variable=link inter false
[...]
 dagger_matrix_times_vector(link[i], u[i], w[i], uu[i], ww[i]);
}

for (int i = 4; i < 8; i++) {
#pragma HLS dependence variable=link inter false
[...]
 matrix_times_vector(link[i], u[i], w[i], uu[i], ww[i]);
}
```

Finally (Stage 4) we scale and add up all contributions to the final result. Because of the dependencies between consecutive partial results, we had to create a 4-layer operation cascade, which in total takes $57 = (4 * 14) + 1$ clock cycles, 4 additions plus one data copy.

Overall, the kernel requires 142 clock cycles and a total of 1464 basic operations to compute the final result since the reception of the input data. It is important to emphasize that, when the kernel is fully pipelined it can accept new input data at each clock cycle and produce the results with latency of 142 cycles, which means there is no dead time in the computation flow, except for startup iteration. Assuming the input data is delivered at requested speed and the clock frequency is 500 MHz the kernel itself is capable to achieve 676 GFLOPs performance. All computations have been decomposed into basic operations and those possible to parallelize have been implemented so. The only factor preventing further optimization is the critical path created by the dependency scheme between partial results.

The implementation is fully scalable. If enough compute resources are available on the device, a second instance of the hardware accelerated kernel can be instantiated. Because the computations for each lattice site are completely independent, the set of lattice sites could thus be divided into two parts and each of them could be associated to one of the kernels, reducing the total computation time by a factor of two.

6.5. Data locality

In order to increase locality of the implementation we further divided the local lattice into two blocks, by splitting into two halves the data from one dimension. The reason for this optimization is that we gain from the fact that two blocks are completely independent and can be analyzed fully in parallel if only the device has enough resources. Moreover, higher clock frequencies are possible because we increase locality of the hardware implementation by reducing the number of calls to distant memory blocks. Greater locality yields much more flexibility to the compiler which can optimize the code better. This optimization requires to store twice a copy of the boundary data for each block separately, which however is negligible compared to the optimization described above when the entire copy of all U matrices is needed to allow parallel access to the data.

7. Benchmarks results

In this section we describe the timings gathered for the CG algorithm. First we comment on the obtained compilation logs focusing on the resource utilization summaries, then we discuss timings and FLOP performances for various hardware configurations.

7.1. Compilation details

In Table 1 we report on the amount of resources used for the compilation of the entire project on various devices. We specify the total number of particular architectural blocks: Block RAM, Digital Signal Processing slices, Flip-Flop units, Look-Up Table units and Ultra RAM needed to compile either the accelerated kernel only or the entire solution containing also the data moving infrastructure.

The first four rows are generated for Xilinx VU13P, at the moment the FPGA with highest amount of resources. We have

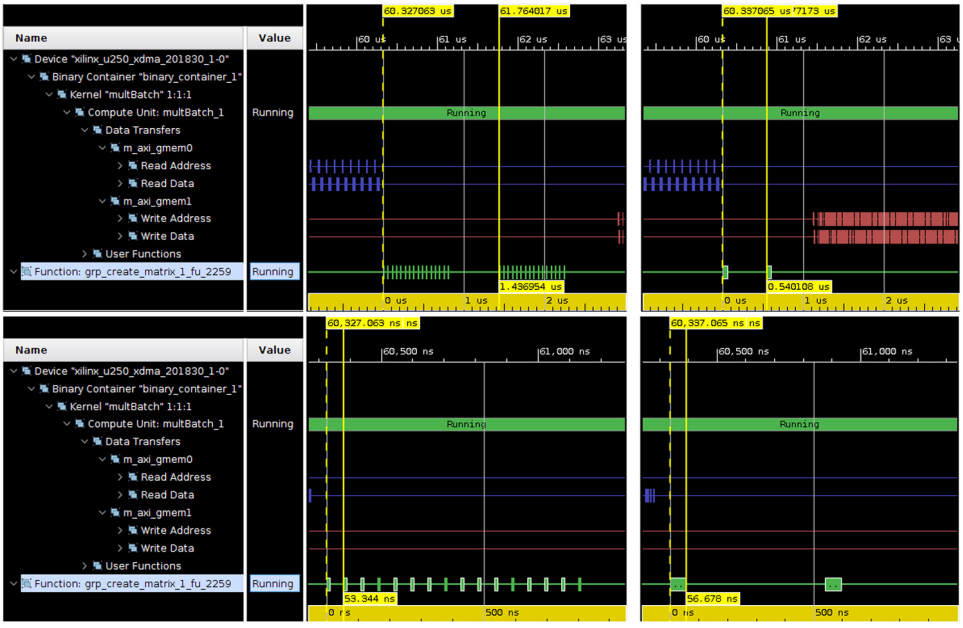

Fig. 4. Emulation timeline of the kernel. The left column shows the case of II = 16 and the right column shows the kernel achieving initiation interval of 1 clock cycle, both obtained on the ALVEO U250 device. Blue signals correspond to reception of the input datasets, red to the transmission of the output results and lower green signal shows an execution of one the kernel components. The set of input data was reduced to 32 sites in order to improve readability of the figure. Figure is a screenshot of the QEMU capture.

Table 2
Resource availability on different devices discussed in the text. Column names are the same as in Table 1. Data from Xilinx datasheets: [29,30].

Device	BRAM	DSP	FF [10^6]	LUT [10^6]	URAM
ZU9EG	1824	2520	0.55	0.27	–
ALVEO U250	2000	11508	2.75	1.34	1280
VU13P	5376	12288	3.46	1.73	1280

successfully compiled the design and analyzed the reports in order to provide prospects of how our kernel would perform on high-end devices. Consecutive two entries are results from Alveo U250 accelerator board for which we have compiled the design and emulated it with Quick EMUlator (QEMU — part of Xilinx programming suite). Last entries show the results of the design compiled and tested on hardware using Xilinx ZU9EG. For the reference, in Table 2 we present available resources on each of these devices.

The compilation process can be controlled by a set of parameters as `#define` in the include files and arguments in `#pragma` statements. One can control

- the problem size by setting the dimensions of the lattice, which also fixes the sizes of generated arrays ($\psi(n)$, $U_\mu(n)$ as well as nearest-neighbor tables),
- the partitioning/reshape factors of the data arrays,
- number of instances of computing kernels by HLS `allocation instances`
- parallelization factors with HLS `unroll factor`

Using those parameters one can find a balance between resource usage and achieved Iteration Interval.

For the following results we used HLS version 2018.2. There is a noticeable change between this and the previous version of the compilation suite which enforces the tool to produce results with lower II rather than minimize resource usage.

From the resource usage collected in Table 1 one can derive many conclusions. The most important one is that it is possible to generate a kernel capable to compute single stencil within 142 cycles and maintain II = 1 allowing fully pipelined designs. Example of emulation report of such kernel is shown in Fig. 4. Middle-range devices do not have enough internal memory resources in order to properly distribute array elements for the kernel to compute new results at each clock cycle. Therefore, the memory usage is crucial while achieving II = 1. Both Alveo U250 and VU13P have enough resources to accommodate fully pipelined kernel instance, although a difference in consumption can be noted. This comes from the fact that the design for Alveo was compiled with clock frequency 300 MHz and for VU13P it was 500 MHz. In such case, the compiler optimization process has selected to use more FF and LUT for arithmetic operations at the expense of DSP. On ZU9EG 76% of memory is used (the rest has to be reserved for Data Movers) but only 21% of DSP blocks, which means that most of the computing resources are left unused due to lack of parallel memory interfaces and duplicated data units.

The application also requires large datasets to be held in internal memories. Devices that contain URAMs have a great advantage over devices with a number of BRAMS. This is because access to a single large memory allows achieving better timing and avoids long signal propagation paths in generated routing.

One can also notice lack of improvement when the unroll factor is not consistent with the loop trip count. Additional resources reserved by subsequent instances of the kernel are not fully used because the flow control logic has to wait until uneven kernels finish.

7.2. Timing analysis

Fig. 4 shows a sample of results of the emulation of our kernel on the Alveo U250 device. In order to increase the readability of the figure we have decreased the problem size to $V = 2^3 \times 4 = 32$ lattice sites. The left part of the figure shows which channels were monitored during the emulation. On the horizontal axis of the right part of the figure emulated time is shown in micro or nano seconds. The most interesting signal marked in green in the

Table 3
Resource consumption versus problem size. Column names are the same as in Table 1. All lattice sizes fit in the largest available VU13P device.

L	T	BRAM	DSP	FF [10^6]	LUT [10^6]	URAM
6	8	508	6960	1.58	0.99	696
8	8	508	6960	1.58	0.99	888
8	10	508	6960	1.58	0.99	888
8	12	508	6960	1.58	0.99	1080

lower part of the screenshots is the function *grp_create_matrix* which is a component of our kernel. The corresponding timeline on the rights shows at which clock cycle that particular function is executed. The remaining signals, blue and red, show the data transfer to and from the kernel. The right part of the figure has two columns: the left column shows results for an emulation where the initiation interval for the kernel was set to 16 clock cycles, whereas the right column corresponds to an initiation interval of 1 clock cycle. Because, as we described above in Section 6.5, we divide the analyzed lattice into two parts, the signal has two parts, each consisting of 16 calls to the kernel. The upper row shows the timeline on a larger time scale providing an overview of the entire execution of the program. When II = 1 the calls form a continuous band as at every clock cycle an execution of that function is initiated. The lower row shows consecutive calls to the kernel in more detail. One can see that when II = 16 the delay between the execution of the *grp_create_matrix* is 53.344 ns which corresponds to exactly 16 clock cycles at 300 MHz. Correspondingly, in the case of II = 1, 16 calls to that function take 56.678 ns which is equal to 16 clock cycles + 1 additional clock cycle. Although for the timeline we had to chose a particular function, the same conclusions must remain true for the entire kernel, because each kernel call contains exactly one call to the *grp_create_matrix* function.

7.3. Problem size limits

In this section we briefly comment on the maximal problem size that fits in the currently available FPGA devices using our implementation. The problem size is given by the dimensions of the four-dimensional lattice where the gauge and spinor fields are defined. We parametrize them with two integers L and T, L being the number of lattice sites in three spatial directions and T being the extend in the time direction. The total number of lattice sites is given by $V = L^3 \times T$ and is equal to the number of the calls to the hardware accelerated single stencil computation kernel.

In Table 3 we report on the dependence of FPGA resources used as a function of the problem size. The first entry corresponds to the maximal problem size which was run on the Xilinx ZU9EG device. The limited memory resources available on that device allow to use a lattice of size $V = 6^3 \times 8$. We remind that the lattice is divided into two sublattices, $6^3 \times 4$ each in order to increase data locality. Subsequent entries correspond to problem sizes which were compiled for the Xilinx VU13P device and fit into that device memory. We notice that the consumption of compute resources does not change as the problem size increases, only the amount of used memory blocks increases.

7.4. Performance benchmarks

7.4.1. Methodology

The timings are collected from on-hardware execution, full system emulations and compilation logs. We count the number of cycles actually needed for the execution of particular steps of the calculation (reading data in from DDR, evaluating all stencils, writing data out to DDR). Assuming that the input data does not need to be loaded from the DDR memory for each consecutive call of the accelerated function, the duration of the computations can be calculated knowing the degree of parallelization and the following parameters: the interval δ and latency τ of the compute kernel, the clock frequency ν, the number of lattice sites V to be processed and the number of FLOPs per lattice site f,

$$\text{performance} = V \times f \times \nu/(V \times \delta + \tau) \tag{5}$$

In order to estimate the number of FLOPS per lattice site f, we added appropriate counters for all arithmetic operations of the basic types (`int`, `double` and `complex`) used in our implementation and explicitly counted all floating point operations, obtaining 1464 FLOPs per lattice site.

7.4.2. Results

In Table 4 we report on the obtained performances in GFLOPs. The numbers come from the inspection of the trace data captured during algorithm emulation or execution in hardware and application of Eq. (5). Results for the VU13P device have been extrapolated from compilation logs.

Maximum performance can be achieved with the design that has the lowest II. That is possible on the VU13P and Alveo U250 devices which have enough memory and DSP blocks available. Assuming no data transfer is required and the clock frequency is at the level of 500 MHz (kernel timing closure obtained) the generated computing logic reaches 676 GFLOPs. Correspondingly for the emulated kernel on the Alveo device at 300 MHz we obtained the performance of 405 GFLOPs. The performance scales linearly with the amount of fully utilized kernels.

Due to limited memory resources we have achieved only 1.8 GFLOPs on the available ZU9EG. Not only high number of clock cycles for II but also lower clock frequency (Virtex Ultrascale+ is the high-end devices family and contains URAMs) are the reason for slower design.

As it was explained in Section 4, the kernels have to wait until all data is transported from the DDR to PL. Including the time required by this process, we achieved 1.3 GFLOPs on the available platform. The ratio between with and without data transport is strongly dependent on the hardware infrastructure available on the FPGA chip itself and the hardware platform it is mounted on. Our ZU9EG device has 4 hardware interfaces between DDR and PL. Using Virtex Ultrascale+ devices, one can profit from tens of embedded multigigabit transceivers to stream data into the device. Although it will require to design a proprietary data distribution system, we expect to achieve much better ratio than in the case of the proof-of-concept platform.

8. Prospects for multi-node systems

It is clear from the results described in previous sections that practically interesting problem sizes can be dealt with only using a multi-node system. It also follows that FPGA devices cannot be used as simple accelerators because the data transfer between DDR memory and the logic severely limits the achieved performance. Therefore the only viable solution is to connect the hardware logic blocks directly together bypassing the control of the host processing unit. This can be achieved using the built in transceivers and a simple send/receive infrastructure implemented directly in the logic.

The initial problem would be then divided into blocks each of which would be processed by a single FPGA device. The data would be loaded once at the beginning and kept in the logic between consecutive iterations of the algorithm. The interconnect infrastructure would be used to exchange boundaries between adjacent blocks. In such a setup the calculation of the norm of the residue vector could also be executed in the logic and only

Table 4
Achieved performance in GFLOPs.

Device	Initiation interval	Clock frequency [MHz]	Performance [GFLOPs]
VU13P	1	500	676
ALVEO U250	1	300	405
ZU9EG	120	150	1.82

the norm would be returned to the control host processing unit, where the stopping criterion would be evaluated.

Our preliminary results point that an HPC solution based entirely on FPGA devices could outperform current installations at a considerably smaller electric power consumption.

9. Conclusion

In this paper we have presented our implementation of the Conjugate Gradient algorithm as used in the Monte Carlo simulations of Quantum Chromodynamics. We described various optimizations for the specific micro-architecture of the FPGA devices. We benchmarked our solution and found out that the obtained performance is comparable with the one obtained on modern CPU units. Our computing kernel is fully pipelined and can achieve 676 GFLOPs with double precision. We conclude therefore that FPGA devices can be (taking into account their development pace — should be) considered as a viable solution for HPC systems as far as the specific application considered in this paper is concerned. Scalability of this solution still has to be demonstrated and benchmarked, we have however pointed out a direction which seems to be particularly encouraging.

Acknowledgments

The Authors thank K.K. for many valuable discussions.

This work was in part supported by Deutsche Forschungsgemeinschaft under Grant No. SFB/TRR 55 and by the polish NCN Grant No. UMO-2016/21/B/ ST2/01492, by the Foundation for Polish Science Grant No. TEAM/2017-4/39 and by the Polish Ministry for Science and Higher Education Grant No. 7150/E-338/M/2018.

The project could be realized thanks to the support from Xilinx University Program and their donations.

Appendix. Conventions

Dirac matrices are given by

$$\gamma_0 = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}, \gamma_1 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix},$$
$$\gamma_2 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}, \gamma_3 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}. \quad \text{(A.1)}$$

Additionally, we have

$$\gamma_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}. \quad \text{(A.2)}$$

References

[1] URL http://www.hpctoday.com/state-of-the-art/when-are-fpgas-the-right-choice-to-improve-hpc-performance/.
[2] URL http://www.top500.org/news/intel-launches-fpga-accelerator-aimed-at-hpc-and-hpda-applications/.
[3] URL https://www.nextplatform.com/2018/04/04/another-step-toward-fpgas-in-supercomputing/.
[4] M. Hestenes, E. Stiefel, J. Res. Natl. Bur. Stand. 49 (1952) 409.
[5] S. Duane, A. Kennedy, B.J. Pendleton, D. Roweth, Phys. Lett. B 195 (2) (1987) 216–222, http://dx.doi.org/10.1016/0370-2693(87)91197-X, URL http://www.sciencedirect.com/science/article/pii/037026938791197X.
[6] C. Urbach, K. Jansen, A. Shindler, U. Wenger, Comput. Phys. Comm. 174 (2006) 87–98, http://dx.doi.org/10.1016/j.cpc.2005.08.006, arXiv:hep-lat/0506011.
[7] URL https://pc2.uni-paderborn.de/hpc-services/available-systems/fpga-research-clusters/.
[8] R. Kiełbik, K. Hałagan, W. Zatorski, J. Jung, J. Ulański, A. Napieralski, K. Rudnicki, P. Amrozik, G. Jabłoński, D. Stożek, P. Polanowski, Z. Mudza, J. Kupis, P. Panek, Comput. Phys. Comm. 232 (2018) 22–34, http://dx.doi.org/10.1016/j.cpc.2018.06.010, URL http://www.sciencedirect.com/science/article/pii/S0010465518302182.
[9] K.G. Wilson, Phys. Rev. D 10 (1974) 2445–2459, http://dx.doi.org/10.1103/PhysRevD.10.2445, URL https://link.aps.org/doi/10.1103/PhysRevD.10.2445.
[10] C. Gattringer, C.B. Lang, Quantum Chromodynamics on the Lattice, third ed., Springer, Berlin, Heidelberg, 2010.
[11] T. Sims, Xilinx unveils revolutionary adaptable computing product category, Xilinx inc., 2018.
[12] M. Wissolik, D. Zacher, A. Torza, B. Day, Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance, Xilinx inc., 2017.
[13] UltraFast High-Level Productivity Design Methodology Guide, Xilinx inc, 2018.
[14] A. Rago, Proceedings, 35th International Symposium on Lattice Field Theory, Lattice 2017: Granada, Spain, June 18-24, 2017, in: EPJ Web Conf., 175, 2018, p. 01021, http://dx.doi.org/10.1051/epjconf/201817501021, arXiv:1711.01182.
[15] B. Joó, R.G. Edwards, F.T. Winter, Exascale Scientific Applications: Scalability and Performance Portability, CRC Press, 2017, p. 345.
[16] M. Clark, A. Strelchenko, A. Vaquero, M. Wagner, E. Weinberg, Comput. Phys. Comm. 233 (2018) 29–40.
[17] R.C. Brower, E. Weinberg, M. Clark, A. Strelchenko, Phys. Rev. D 97 (11) (2018) 114513.
[18] S. Heybrock, B. Joo, D.D. Kalamkar, M. Smelyanskiy, K. Vaidyanathan, T. Wettig, P. Dubey, The International Conference for High Performance Computing, Networking, Storage, and Analysis: SC14: HPC Matters, SC, New Orleans, LA, USA, November 16-21, 2014, 2014, http://dx.doi.org/10.1109/SC.2014.11, arXiv:1412.2629.
[19] S. Heybrock, M. Rottmann, P. Georg, T. Wettig, Proceedings, 33rd International Symposium on Lattice Field Theory, Lattice 2015: Kobe, Japan, July 14-18, 2015, LATTICE2015, PoS, 2016, p. 036, http://dx.doi.org/10.22323/1.251.0036, arXiv:1512.04506.
[20] P. Georg, D. Richtmann, T. Wettig, Proceedings, 35th International Symposium on Lattice Field Theory, Lattice 2017: Granada, Spain, June 18-24, 2017, in: EPJ Web Conf., 175, 2018, p. 02007, http://dx.doi.org/10.1051/epjconf/201817502007, arXiv:1710.07041.
[21] I. Kanamori, H. Matsufuru, 5th International Workshop on Legacy HPC Application Migration: International Symposium on Computing and Networking, CANDAR 2017, LHAM 2017, Aomori, Japan, November 19-22, 2017, 2017, arXiv:1712.01505.
[22] S. Durr, Proceedings, 35th International Symposium on Lattice Field Theory, Lattice 2017: Granada, Spain, June 18-24, 2017, in: EPJ Web Conf., 175, 2018, p. 02001, http://dx.doi.org/10.1051/epjconf/201817502001, arXiv:1709.01828.
[23] P.A. Boyle, G. Cossu, A. Yamaguchi, A. Portelli, Proceedings, 33rd International Symposium on Lattice Field Theory, Lattice 2015: Kobe, Japan, July 14-18, 2015, LATTICE2015, PoS, 2016, p. 023, http://dx.doi.org/10.22323/1.251.0023.
[24] O. Callanan, A. Nisbet, E. Özer, J. Sexton, D. Gregg, 19th International Parallel and Distributed Processing Symposium, IPDPS 2005, CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA, 2005, http://dx.doi.org/10.1109/IPDPS.2005.228, URL https://doi.org/10.1109/IPDPS.2005.228.
[25] O. Callanan, D. Gregg, A. Nisbet, M. Peardon, 2006 International Conference on Field Programmable Logic and Applications, 2006, pp. 1–6, http://dx.doi.org/10.1109/FPL.2006.311191.

[26] O. Callanan, High Performance Scientific Computing using FPGAs for Lattice QCD (Ph.D. thesis), Trinity College (Dublin, Ireland), School of Computer Science and Statistics, 2007, p. 161.
[27] M.A. Clark, R. Babich, K. Barros, R.C. Brower, C. Rebbi, Comput. Phys. Comm. 181 (2010) 1517–1528, http://dx.doi.org/10.1016/j.cpc.2010.05.002, arXiv:0911.3191.
[28] P. De Forcrand, D. Lellouch, C. Roiesnel, J. Comput. Phys. 59 (1985) 324–330, http://dx.doi.org/10.1016/0021-9991(85)90149-4.
[29] URL https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.
[30] URL https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf#VUSP.