Implementation of the conjugate gradient algorithm in Lattice QCD on FPGA devices

Piotr Korcyl[*]

M. Smoluchowski Institute of Physics, Jagiellonian University, ul. Łojasiewicza 11, 30-348 Kraków, Poland
Institut für Theoretische Physik, Universität Regensburg, 93040 Regensburg, Germany
E-mail: piotr.korcyl@uj.edu.pl

Grzegorz Korcyl

Department of Information Technologies, Faculty of Physics, Astronomy and Applied Computer Science, Jagiellonian University, ul. Łojasiewicza 11, 30-348 Kraków, Poland
E-mail: grzegorz.korcyl@uj.edu.pl

Results of porting parts of the Lattice Quantum Chromodynamics code to modern FPGA devices are presented. A single-node, double precision implementation of the Conjugate Gradient algorithm is used to invert numerically the Dirac-Wilson operator on a 4-dimensional grid on a Xilinx Zynq evaluation board. The code is divided into two software/hardware parts in such a way that the entire multiplication by the Dirac operator is performed in programmable logic, and the rest of the algorithm runs on the ARM cores. Optimized data blocks are used to efficiently use data movement infrastructure allowing to reach intervals of 1 clock cycle. We show that the FPGA implementation can offer a comparable performance compared to that obtained using Intel Xeon Phi KNL.

---

[*]Speaker.

1. Introduction

Field Programmable Gate Array (FPGA) devices enjoy more and more attention from the HPC perspective in the last couple of years. Several FPGA clusters are already in operation, constructed along two different architectural philosophies: either the FPGA devices are considered as the main and only compute resource [1] or they are used as hardware accelerators attached to some CPU [2]. The advantages and disadvantages of both scenarios depend heavily on the memory access patterns in the given problem being solved. In either case, the tremendous increase of available of resources in a single FPGA device witnessed recently and new design methodologies yield a powerful candidate for HPC systems. Several features of FPGA are especially attractive: natural parallelism allows to allocate many instances of a given logical component with all of them running in parallel; multi-gigabit transceivers which are embedded within the FPGA package allow for a communication between several FPGA devices circumventing any additional hardware layer; no dependency nor overhead imposed by the operating system; and last but not least logic resources which can be dynamically reconfigured to accelerate another part of the algorithm. All this comes with a power consumption two orders of magnitude smaller than current HPC compute node processors.

In this contribution we benchmark the most compute-intensive kernel in Lattice QCD, namely the Dirac matrix multiplication on a single FPGA device. To this end we implement a simple Conjugate Gradient solver and use it to invert the Dirac matrix on a small lattice using a single Xilinx ZCU102 platform. The latter possess an 4-core ARM processor which we use to run the main program function and a programmable logic block where we implement the hardware-accelerated Dirac matrix multiplication. In these proceedings we briefly describe the conditions which must be fulfilled in order to fully pipeline the computations of all stencils. We also investigate the limitations on the possible lattice sizes that can be analyzed on a single FPGA. Along the way we briefly describe modern tools which allow the user to program solely in a high-level programming language such as c++ thus considerably simplifying the effort to program the FPGA devices.

The results discussed in this contribution were described in detail in Ref. [3]. Previous, independent efforts by other groups going in the same direction were presented at conferences in 2005 [4] and 2006 [5] by O. Callanan and recently in 2016 by T. Janson [6].

In the following section 2 we start by very briefly describing our benchmark problem, the Dirac operator and the Conjugate Gradient algorithm, then we discuss the FPGA technology, concentrating on their possible advantages in the context of HPC in section. The results of our benchmark tests are discuss in section 3, before the conclusions.

2. Implementation details

The Dirac operator matrix acting on a spinor vector is defined as [7]

$$D(m,n)_{\alpha\beta}^{AB}\psi_{\beta}^{B}(n) = (m_q+4)\psi_{\alpha}^{A}(n)+$$

$$+\frac{1}{2}\sum_{\mu=0}^{3}\left[U_{\mu}^{AB}(n)P_{\alpha\beta}^{-\mu}\psi_{\beta}^{B}(n+\hat{\mu})+U^{\dagger,AB}(n-\hat{\mu})P_{\alpha\beta}^{+\mu}\psi_{\beta}^{B}(n-\hat{\mu})\right] \quad (2.1)$$

with the convention that two indices are always summed over, and $P^{\pm\mu} = 1 \pm \gamma_{\mu}$. We exploit 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 rescaling [8], hence one can halve the number of multiplications by the U matrix.

The problem we are addressing is that of finding the solution $\psi_B^\beta(m)$ given the rght-hand side vector $\eta_A^\alpha(n)$,

$$D_{\alpha\beta}^{AB}(n,m)\psi_B^\beta(m) = \eta_A^\alpha(n). \tag{2.2}$$

The Dirac operator as defined in Eq. 2.1 satisfies the γ_5-hermiticity, $\gamma_5 D \gamma_5 = D^\dagger$, so in order to apply the CG algorithm one needs to solve for $DD^\dagger$ which is hermitian and then multiply the solution by $D^\dagger$. The simple conjugate gradient algorithm reads

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

The compute intensive part of the algorithm is the multiplication by $D^\dagger D$, out of which the most elementary computational block is the evaluation of the single stencil, i.e. the right hand side of Eq.(2.1) for a given value of the index n. This involves eight U matrices and eight spinor fields from the neighboring lattice sites, which corresponds to 1536 input bytes. The $U \times \psi$ matrix-vector multiplications require 1152 floating point operations for complex additions and multiplications.

A schematic view of the modern FPGA device is shown on the left panel of figure 1. As already mentioned it contains a processing part (PS) which in our case is equipped with an ARM processor, and the PL part with the programmable logic. The figure also shows the connections to the external DDR memory. One of the key decisions from the programming point of view is the software partitioning between the PS and PL parts. 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 equation 2.2. In such scenario the main part of the CG algorithm is executed on the PS part. After each iteration the new residuum vector is returned to the DDR where the ARM processor evaluates the vector scalar products and the stopping criterion.

In order to easily compare the performance with the standard CPU architectures we use exclusively double precision numbers. This induces a considerable increase in the cost, because large logic resources are required to handle 64 bit arithmetics. A single addition or multiplication takes 14 clock cycles on the FPGA device used in this study. Modern GPU implementations of similar algorithms [9] use various combinations of low precision steps (single or half) in order to increase performance. FPGA devices offer a much broader spectrum of available fix point and floating point arithmetics which can be tuned to optimize the performance.
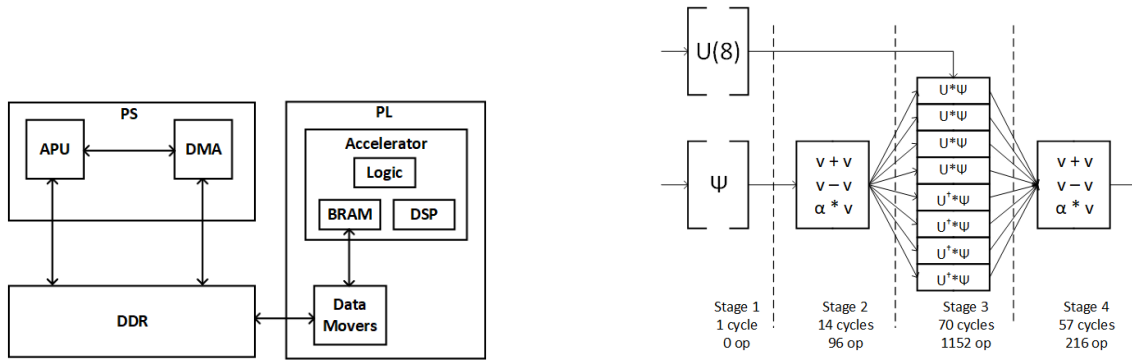
Figure 1: Left panel: schematic view at the key components of Zynq MPSoC used to transfer data between PS and PL. Right panel: computation sequence of the stencil solver

One of the great advantages of the programmable logic implementation is that one can use abstract types in the accelerated function without any loss of performance. Once the required data blocks are copied from the DDR memory to the memory blocks of the device, data can be accessed in parallel. In particular the ordering of the real and imaginary parts in memory does not longer play a role, as all the appropriate reshufflings will be executed in hardware data routing. In practice this allow us to work entirely with abstract types `complex`, `su3_vector`, `su3_matrix` and `su3_spinor`, the latter being simply

```
struct su3_spinor {
public:
        su3_vector s[4];
}
```

However, 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. In the computation of a single stencil one needs eight different *U* matrices and we store them separately with the help of HLS pragmas such as `HLS ARRAY_PARTITION`. Although this requires duplicating the amount of stored data, the gain is considerable as thanks to that the stencil evaluation can be fully pipelined. Full pipelining means achieving initiation interval of one clock cycle, i.e. the hardware block can accept new input data at each clock cycle. The fixed structure of our computations, i.e. execution of exactly the same computations for each lattice site, allows for fully a pipelined Dirac matrix multiplication, if only enough logic resources are available on the device. Once the data is stored appropriately the compiler parallelizes and pipelines the computations as shown on the right panel of figure 1 by the schematic view of the logical data movement during a single stencil evaluation. The evaluation can be naturally divided into 5 stages.

In the first stage all the necessary data is copied from the BRAM memory blocks to local registers (declared with the `ARRAY_PARTITION complete` attribute). The process requires only one clock cycle to collect all needed data. Later, in stage 2 linear combinations of input data, 8 additions and 8 subtractions of `su3_vector` type, are evaluated. They are all performed in

Table 1: Resource consumption versus parallel execution.

latency	interval	bram	dsp	ff $[10^6]$	lut $[10^6]$	uram
142	1	508	6960	1.58	0.99	696
151	2	448	4320	0.97	0.64	696
151	2	428	3480	0.83	0.57	696
162	4	412	1740	0.47	0.35	696
250	120	1388	546	0.13	0.09	–
		(76%)	(21%)	(24%)	(34%)	acc.
		(95%)	(21%)	(31%)	(40%)	full

parallel, taking 14 clock cycles. The most compute intensive stage 3 involves `su3_matrix` and `su3_vector` multiplications. In total 1152 operations on `double` are performed. Complete parallelization allows to execute them in a 5-layer operation cascade taking in total $5*14 = 70$ cycles. Finally, at stage 4 all contributions are added up to the final result. Because of the dependencies between consecutive partial results this creates 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. The kernel is fully pipelined: i.e. it can accept new input data at each clock cycle and produce the results with latency of 142 cycles.

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.

3. Benchmarks results

We compiled our implementation for two systems: Xilinx ZU9EG. This is the device, which we had available, and Xilinx XCVU13P which we do not have physical access to. We used HLS version 2018.2. We report on the number of resources used for the compilation of the entire project in table 1. The first four entries are generated for the larger Xilinx XCVU13P device and differ by the initiation interval shown in column 2. The latter can be controlled by HLS pragmas limiting the number of instantiated blocks. In the last row of the table we separate resource usage gathered from the Xilinx ZU9EG. The resource usage shows that it is possible to generate a fully pipelined kernel capable of computing single stencil within 142 cycles and maintaining initiation interval of 1 clock cycle. Smaller devices do not have enough internal memory resources to properly distribute array elements for the kernel to compute new results at each clock cycle. Therefore, the memory usage is crucial while achieving full pipelining. 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.

Table 2: Performance in GFLOPs.

Interval	Clock frequency [MHz]	Performance [GFLOPs]
1	500	676
2	500	351
4	500	179
120	150	1.82

The maximal problem size which was run on the Xilinx ZU9EG device was $V = 6^3 \times 8$ because of the limited memory resources available on that device. Problem sizes which were compiled for the Xilinx XCVU13P device and fit into that device memory range from $V = 8^4$ to $V = 8^3 \times 12$. The consumption of compute resources does not change as the problem size increases, only the amount of used memory blocks increases.

The timings are collected from execution and compilation logs by inspecting the trace data captured during algorithm execution in hardware. 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 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,

$$\text{performance} = V \times f \times \nu / (V \times \delta + \tau) \tag{3.1}$$

In table 2 we report on the obtained performances in GFLOPs. Maximum performance can be achieved with the design that has the lowest initiation interval. That is possible on the XCVU13P device with has enough memory and DSP block 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. The performance drops linearly with the amount of fully utilized kernels.

Due to limited memory resources we have achieved only 1.8 GFLOPs on the available ZU09EG. Not only high number of clock cycles for initiation interval but also lower clock frequency (Virtex Ultrascale+ is the high-end devices family and contains URAMs) are the reason for slower design.

When the kernels have to wait until all data is transported from the DDR to PL, 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 ZCU09EG 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. Another solution is to use a direct connection between a DDR memory die and the logic as in Ref.[6], where the data is streamed into the programmable logic block.

4. Conclusion

In this contribution we have presented our implementation of the Conjugate Gradient algorithm as used Lattice QCD for FPGA devices. Our hardware accelerated kernel is fully parallelized and pipelined. We benchmarked our solution on a relatively small Xilinx evaluation board. Compilation details for larger devices suggest that the obtained performance can comparable with the one obtained on modern CPU units. We conclude therefore that FPGA devices can be considered as a viable solution for HPC systems.

Acknowledgment

References

[1] 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, and P. Panek, *Aruz — large-scale, massively parallel fpga-based analyzer of real complex systems*, *Computer Physics Communications* **232** (2018) 22 – 34.

[2] T. Kenter, *Reconfigurable Accelerators in the World of General-Purpose Computing*. Universität Paderborn, 2016.

[3] G. Korcyl and P. Korcyl, *Towards Lattice Quantum Chromodynamics on FPGA devices*, arXiv:1810.04201.

[4] O. Callanan, D. Gregg, A. Nisbet, and M. Peardon, *High performance scientific computing using fpgas with ieee floating point and logarithmic arithmetic for lattice qcd*, in *2006 International Conference on Field Programmable Logic and Applications*, pp. 1–6, Aug, 2006.

[5] O. Callanan, A. Nisbet, E. Özer, J. Sexton, and D. Gregg, *FPGA implementation of a lattice quantum chromodynamics algorithm using logarithmic arithmetic*, in *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA*, 2005.

[6] T. Janson and U. Kebschull, *Highly parallel lattice qcd wilson dirac operator with fpgas*, in *PARCO*, 2017.

[7] C. Gattringer and C. B. Lang, *Quantum Chromodynamics on the Lattice*. Springer, Berlin, Heidelberg, 3 ed., 2010.

[8] C. Tadonki, *Wilson-dirac operator revisited on multicore vector computers*, 09, 2016.

[9] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, *Solving Lattice QCD systems of equations using mixed precision solvers on GPUs*, *Comput. Phys. Commun.* **181** (2010) 1517–1528, [arXiv:0911.3191].