# Implementation of the conjugate gradient algorithm for heterogeneous systems

**Salvatore Calì,**[a,*] **William Detmold,**[a] **Grzegorz Korcyl,**[b] **Piotr Korcyl**[c] **and Phiala Shanahan**[a]

[a] *Center for Theoretical Physics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

[b] *Institute of Applied Computer Science, Jagiellonian University, ul. prof. Łojasiewicza 11, 30-348 Kraków, Poland*

[c] *Institute of Theoretical Physics, Jagiellonian University, ul. prof. Łojasiewicza 11, 30-348 Kraków, Poland*

*E-mail:* calis@mit.edu, wdetmold@mit.edu, grzegorz.korcyl@uj.edu.pl, piotr.korcyl@uj.edu.pl, pshana@mit.edu

Lattice QCD calculations require significant computational effort, with the dominant fraction of resources typically spent in the numerical inversion of the Dirac operator. One of the simplest methods to solve such large and sparse linear systems is the conjugate gradient (CG) approach. In this work we present an implementation of CG that can be executed on different devices, including CPUs, GPUs, and FPGAs. This is achieved by using the SYCL/DPC++ framework, which allows the execution of the same source code on heterogeneous systems.

MIT-CTP/5348

*The 38th International Symposium on Lattice Field Theory, LATTICE2021 26th-30th July, 2021 Zoom/Gather@Massachusetts Institute of Technology*

*Speaker

## 1. Introduction

With the diversification of computing hardware, programming languages that allow software to be executed on any combination of devices, including Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), are very appealing. Such a framework gives freedom to the user to delegate the most demanding parts of a calculation to a specific accelerator, depending on the problem at hand. This mitigates the costs of different programming and optimization for each resource which is present on a supercomputer node [1]. This is the motivation for the standard SYCL/DPC++ (Data Parallel C++) (see [4] for an introduction), and in this work we explore the possibility of using this framework for lattice QCD (Quantum Chromodynamics) applications.

One of the most common problems in computational science and linear algebra is to solve systems of linear equations. For instance, in a typical lattice QCD calculation, most of the computing resources are spent in finding the solution $\psi$ of the following linear system:

$$D^{AB}_{\alpha\beta}(n,m)\psi^{B}_{\beta}(m) = \eta^{A}_{\alpha}(n), \quad A, B \in \{0,1,2\}, \quad \alpha, \beta \in \{0,1,2,3\}, \tag{1}$$

where $D$ is the Wilson-Dirac operator [5] and $\eta$ is an arbitrary source fermion field. In Eq. (1) $A, B$ are color indices, $\alpha, \beta$ are spin indices and $m, n$ represent two space-time coordinates within the lattice volume. To give a concrete example of the typical sizes, a lattice with moderately size volume $V = 48^3 \times 96$ would require the linear system (1) to be solved for a matrix of size $(V \times 12)^2 \approx 10^8 \times 10^8$. As a consequence, solving (1) using a direct approach is not feasible and iterative methods for sparse matrices are employed to drastically reduce the computational effort and the storage needs. Examples of iterative methods used in lattice QCD computations are: conjugate gradient (CG), biconjugate gradient (BiCG), and the biconjugate gradient stabilized method (BiCGSTAB). The solver performance is often improved by preconditioning, such as Algebraic Multigrid (AMG), Incomplete Cholesky factorization (IC), Jacobi method, etc. Recently deep learning techniques have been investigated to design preconditioning matrices for such systems [6, 7]. For a general introduction to iterative methods and preconditioning techniques we refer to Ref. [8].

In Ref. [9], an OpenCL implementation of the CG algorithm for Xilinx FPGAs has been presented, showing that FPGAs can provide competitive performance for the inversion of the Dirac operator (around 607 GFLOPs running in single precision on a Xilinx U280 Alveo card). For recent progress in the FPGA optimized HPCG benchmark, we also refer to Ref. [10], where the authors consider solving a simple elliptic partial differential equation discretized with a 27-point stencil on a regular 3D grid using the CG algorithm.

The idea of this project is to explore the possibility that a single-source code can be used on different architectures for lattice QCD applications. Therefore, in this work we consider a single-node DPC++ implementation of the Conjugate Gradient algorithm applied to the Wilson-Dirac

[1]In this direction, it is worth mentioning that there exist programming models like *Kokkos* and *Raja*, which are C++ abstraction layers for performance portable parallel execution. Using specific features of these models, an algorithm can be mapped onto existing parallel programming languages and frameworks, like CUDA, OpenMP [1], HIP [2], and SYCL/DPC++ [3, 4].

operator. This implementation is executed on different devices (CPUs, GPUs and FPGAs) and we test the performances in each case.

## 2. Numerical details

For these calculations, we explore lattice volumes ranging from $4^4$ up to $14^4$ and focus on the inversion of the standard Wilson-Dirac operator. The latter is known to satisfy the $\gamma_5$-hermiticity, $\gamma_5 D \gamma_5 = D^\dagger$, so to use the CG algorithm we first solve for $DD^\dagger$ (which is hermitian by construction) and then we multiply the solution by $D^\dagger$. In this context, the standard CG algorithm reads

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

where $\psi_0$ represents the initial guess. To access the elements of the Wilson-Dirac operator, we store the sparse matrix $D$ in the so-called coordinate format, i.e. $D$ is stored using three arrays: one array contains the value of the non-zero elements and the other two arrays the corresponding row and column indices. Such an approach is not optimal and does not take advantage of the diagonal structure of D, but it is easy to implement and it is used as a starting point of this exploratory study. Using the coordinate format, a pseudo-code for the sparse-matrix vector multiplication needed in CG reads

```
for $k = 1$, $k$++, while $N_{\text{NZ}}$ do
    out[Row[k]] = out[Row[k]] + Val[k]*in[Col[k]];
end for,
```

where $N_{\text{NZ}}$ is the number of non-zero elements and *in* and *out* are the input and output vectors. Row, Col and Val are the three arrays used to store the sparse matrix (respectively the row and column indices and the corresponding value).

SYCL/DPC++ offers several ways to invoke a kernel and in this work we mainly use the so-called *ND-range kernels*. This approach allows a kernel function to be invoked on each iteration of the task; moreover, the programmer has full control of the parallelism and has the possibility to split the global size of the problem into a desired number of smaller blocks, called *workgroups*. Each workgroup can be interpreted as a 1, 2, or 3 dimensional block of threads and contains a set of work items that are mapped, e.g. to a core in a GPU. The workgroup size determines the occupancy of the compute units and to achieve optimal performances a tuning of the workgroup size is needed. Using ND-range kernels, we have implemented in SYCL/DPC++ all the main operations that appear in the CG algorithm: the sparse-matrix vector multiplication (which represents the most demanding part in terms of computational cost), the dot product, and vector additions/differences.

The code has been tested on three different types of hardware:

- CPUs: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz
- GPUs: Nvidia GeForce RTX 2080 Ti, Nvidia A100-PCIE-40GB
- FPGAs: Intel Arria 10, Intel Stratix 10

and in the next section, we discuss the performance that we have obtained on these devices.

## 3. Results

We split this section into two parts: the first details the results obtained on CPUs and GPUs and the latter contains some preliminary results and considerations related to SYCL/DPC++ codes on FPGA hardware.

### 3.1 CPUs and GPUs

Before showing the main results of this investigation, we highlight that on GPU hardware, in order to obtain the optimal performance, an initial tuning of the workgroup size is needed. For example, in Figure 1 we report a study of the performance of the whole CG algorithm for a lattice volume $14^4$, keeping the number of compute units fixed and varying the workgroup size for the Nvidia A100-PCIE-40GB card. In this case we find an optimal workgroup size of 256. A similar

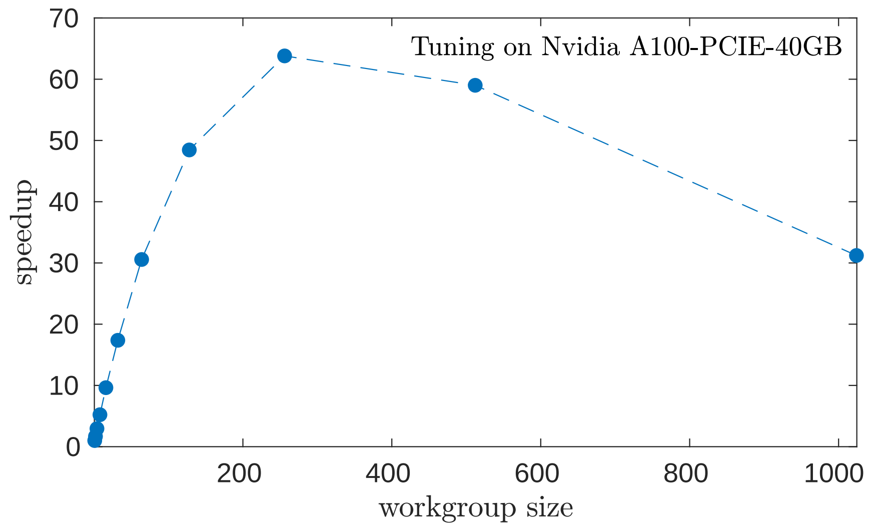

**Figure 1:** Tuning of the optimal workgroup size on the Nvidia card A100-PCIE-40GB, performed for our implementation of the CG algorithm in SYCL/DPC++.

study has been performed on Nvidia GeForce RTX 2080 Ti, for which we find it is more optimal to use a workgroup size of 96. Once this tuning has been performed, we look at the speedup on CPUs and GPUs as a function of the compute units, as shown in Figure 2. We observe in general a better

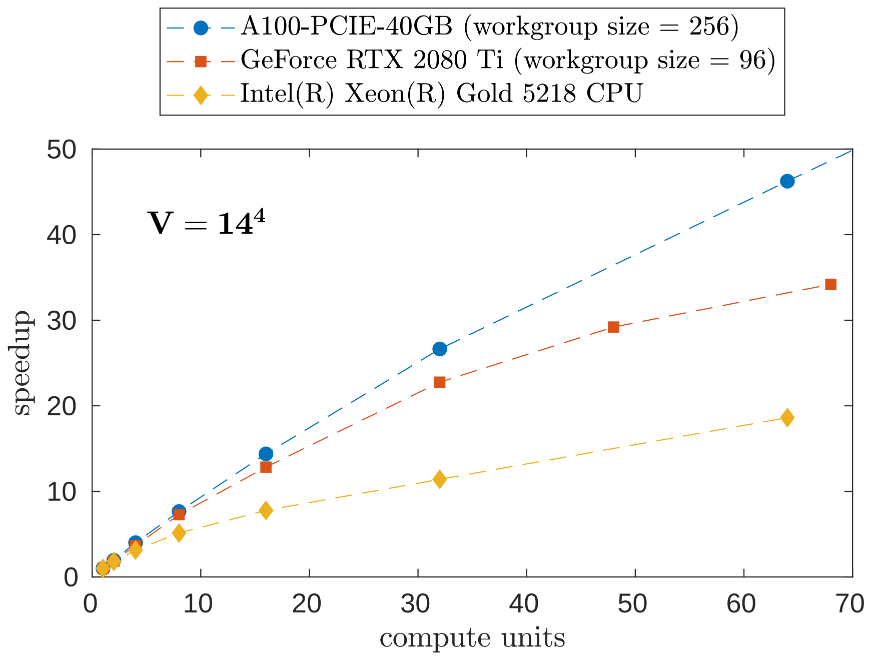

**Figure 2:** Speedup study of our implementation of the CG algorithm in SYCL/DPC++, for a volume $V = 14^4$ and with the Dirac operator stored in coordinate format. The reference point is the execution time obtained using a single compute unit.

scaling on NVIDIA GPUs and in particular for Nvidia A100-PCIE-40GB, than for the GeForce RTX 2080 Ti and the CPU architecture.

We also study the performances of the kernels in terms of GFlops/s, with special focus on the most demanding part, i.e., the sparse-matrix vector multiplication. As an approximation of the maximal theoretical performance that can be achieved on the hardware considered here, we compare the actual performances with the expectations of the naive roofline model [11]. In this context, the maximal attainable performance $P$ (Flops/s) is given by

$$P = \min(\pi, \beta \times I), \tag{2}$$

where $\pi$ is the peak performance, $\beta$ (Bytes/s) is the peak bandwidth and $I$ (Flops/Bytes) is the operational intensity. Operations like sparse-matrix vector multiplications, dot products, and vector additions have low operational intensities and in these cases the kernel is said to be *memory-bound* ($I < \pi/\beta$).

In Figure 3, we show the performances that we obtain for the sparse-matrix vector multiplication, along with the theoretical limit of the roofline model. As we can see from the figure, the maximum performance we can reach is around 65 GFlops/s on Nvidia A100-PCIE-40GB and, in general, for all the devices shown we observe that the actual performances are around 60% – 70% of the theoretical ones.

### 3.2 FPGAs

The same SYCL/DPC++ code, tested on CPUs and GPUs as discussed above, has also been compiled and executed for Intel FPGAs. We find that the same source code is able to run on FPGA

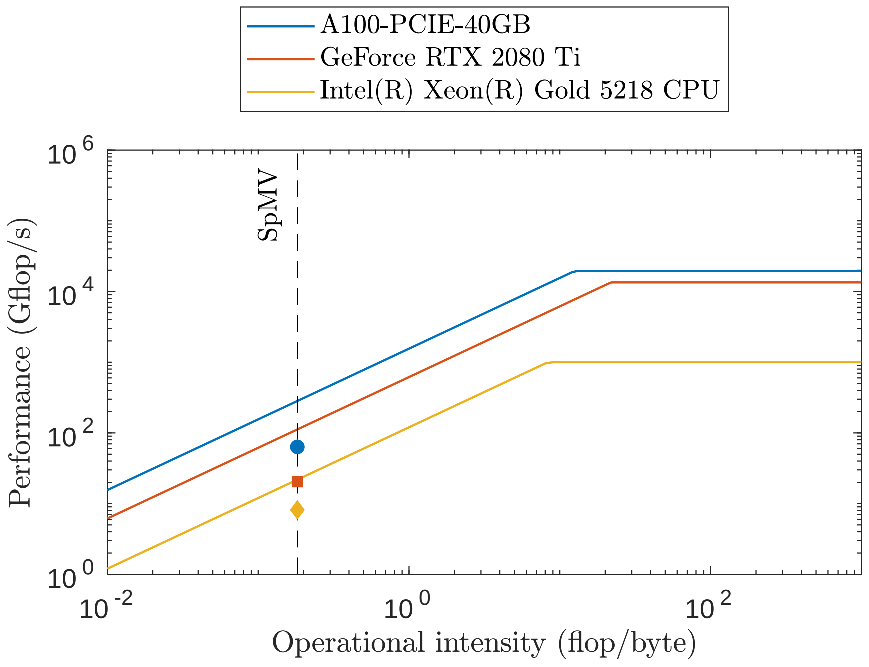

**Figure 3:** Performance study of the sparse-matrix vector multiplication on different devices (points) and comparison with the naive roofline model (solid lines).

hardware, as we expect from the SYCL/DPC++ framework. However, we experience portability problems and using the code as it is leads to much worse performance on the FPGAs, typically below 1 GFlops/s. Therefore, we are currently far from realizing the idea of a single source code, running with good performance on all devices, and we are testing alternative kernels for execution on FPGAs.

One idea is to consider single-task kernels, implementing loop unrolling mechanisms as described in the Intel oneAPI Github repository `https://github.com/oneapi-src/oneAPI-samples`. For kernels implementing simple functions, like vector addition, this approach (which consists of combining single-task kernels and the compiler directive "#pragma unroll") seems very promising, giving rise to a significant performance improvement. However, we still face performance issues with the sparse-matrix vector multiplication kernel, probably because of the frequent memory accesses of the algorithm when the Dirac operator is stored in coordinate format. In the future, it could be interesting to test such kernels for a more suitable representation of the Dirac operator.

## 4. Conclusions and outlook

In these proceedings we have explored for the first time a SYCL/DPC++ implementation of the CG algorithm for the Wilson-Dirac operator. This framework allows a single-source application to be executed on different architectures, and we have tested our software on CPUs, GPUs, and FPGAs. Using the so-called ND-range parallel kernels and the Dirac matrix stored in coordinate format, we have seen that it is possible to obtain acceptable performances on CPUs and GPUs (around 65 GFlops/s on Nvidia A100-PCIE-40GB), that can be further improved designing algorithms more suitable for the Dirac operator. The same code also runs on FPGA hardware, but we observe worse performances, generally below 1 GFlops/s. Therefore, although the idea of having a single-source code to solve the Dirac equation running on different architectures is very appealing, at the moment we are far from this goal. More investigations are needed and in the near future we plan to consider different kernels and matrix representations that can speed up the execution times on FPGAs.

## 5. Acknowledgments

This work was supported by the Foundation for Polish Science grant no. TEAM/2017-4/39, by the Polish Ministry for Science and Higher Education grant no. 7150/E-338/M/2018, and by the Priority Research Area Digiworld under the program Excellence Initiative – Research University at the Jagiellonian University as well as by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics, under grant contract numbers DE-SC0011090 and DE-SC0021006, and by the Carl G and Shirley Sontheimer Research Fund at MIT. WD is also supported by the SciDAC4 award DE-SC0018121. We gratefully acknowledge INTEL for providing access to FPGA hardware through the development sandbox *Intel DevCloud*.

## References

[1] OpenMP Architecture Review Board, “The OpenMP API specification for parallel programming, http://www.openmp.org.”

[2] “HIP: C++ Heterogeneous-Compute Interface for Portability, https://rocmdocs.amd.com.”

[3] Khronos Group, “SYCL , https://www.khronos.org/sycl/.”

[4] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*, Apress (2020).

[5] K.G. Wilson, *Confinement of Quarks*, *Phys. Rev. D* **10** (1974) 2445.

[6] B. Xiao, P. Shanahan, D. Hackett, S. Calì and Y. Lin, *Neural Network Preconditioning for U(1) Wilson-type Dirac Operators*, *Poster presented by Brian Xiao at Lattice 2021, https://indico.cern.ch/event/1006302/contributions/4380639/ [Paper in preparation]* .

[7] J. Sappl, L. Seiler, M. Harders and W. Rauch, *Deep learning of preconditioners for conjugate gradient solvers in urban water related problems*, *CoRR* **abs/1906.06925** (2019) [1906.06925].

[8] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Other Titles in Applied Mathematics, SIAM, second ed. (2003), 10.1137/1.9780898718003.

[9] G. Korcyl and P. Korcyl, *Optimized implementation of the conjugate gradient algorithm for FPGA-based platforms using the Dirac-Wilson operator as an example*, 2001.05218.

[10] A. Zeni, K. O’Brien, M. Blott and M.D. Santambrogio, *Optimized implementation of the hpcg benchmark on reconfigurable hardware*, in *European Conference on Parallel Processing*, pp. 616–630, Springer, 2021.

[11] S. Williams, A. Waterman and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, *Commun. ACM* **52** (2009) 65.