FPGA acceleration for HPC supercapacitor simulations

Charles Prouveur, Matthieu Haefele, Tobias Kenter, Nils Voss

FPGA acceleration for HPC supercapacitor simulations

Charles Prouveur
charles.prouveur@cea.fr
Université Paris-Saclay, UVSQ, Inria, CNRS, CEA, Maison de la Simulation
Gif-sur-Yvette, France

Matthieu Haefele
matthieu.haefele@univ-pau.fr
Université de Pau et des Pays de l'Adour, E2S UPPA, CNRS, LMAP
Pau, France

Tobias Kenter
kenter@uni-paderborn.de
Paderborn Center for Parallel Computing and Department of Computer Science, Paderborn University
Paderborn, Germany

Nils Voss
n.voss16@imperial.ac.uk
Imperial College London
London, United Kingdom

ABSTRACT

In the search of more energy efficient computing devices that could be assembled to build future exascale systems, this study proposes a chip to chip comparison between a CPU, a GPU and a FPGA, as well as a scalability study on multiple FPGAs from two of the available vendors. The application considered here has been extracted from a production code in material science. This allows for the benchmarking of different implementations to be performed on a production test case and not just theoretical ones. The core algorithm is a matrix free conjugate gradient that computes the total electrostatic energy with an Ewald summation at each iteration.

This paper depicts the original MPI implementation of the application, details a numerical accuracy study and explains the methodology followed as well as the resulting FPGA implementation based on MaxCompiler. The FPGA implementation using 40 bits floating point number representation outperforms the CPU implementation both in terms of computing power and energy usage resulting in an energy efficiency more than 15 times better. Compared to the GPU of the same generation, the FPGA reaches 60% of the GPU performance while the ratio of the performance per watt is still better by a factor of 2. Thanks to its low average power usage, the FPGA bests both fully loaded CPU and GPU in terms of number of conjugate gradient iterations per second and per watt. Finally, an implementation using oneAPI is described as well, showcasing a new development environment for FPGA in HPC.

CCS CONCEPTS

• **Applied computing** → **Chemistry**; • **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

FPGA, Parallel computing, Super capacitors, numerical accuracy analysis, matrix-free conjugate gradient

1 INTRODUCTION

The energy requirement of the largest HPC systems is already a key issue and will become even more prominent for coming exascale systems. As of November 2022, Graphics Processing Unit (GPU) based accelerators are used in seven out of the ten largest supercomputers in the world. Their superior energy efficiency compared to general-purpose processors (CPUs) is, to a very large extent, the reason for this design choice.

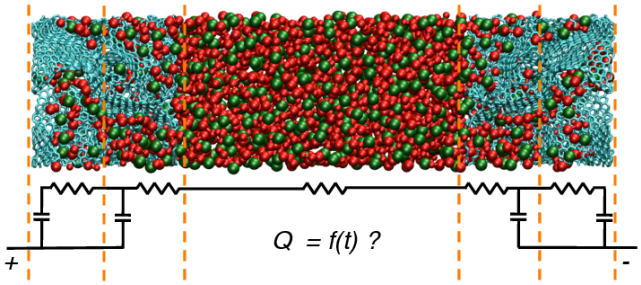

Figure 1: Supercapacitor system

Field Programmable Gate Arrays (FPGAs) are generally regarded as a middle-ground between Application-specific integrated circuits (ASICs), i.e. dedicated chips, and CPUs. This notion comes from the reconfigurable nature of these devices, making them more flexible and less expensive than ASICs (at the cost of lower area and power efficiency) and generally more power efficient than CPU and even GPU. Despite being older than GPUs, they have rarely been used in the HPC industry and have mainly been employed in embedded and low-power markets. Low programming productivity, lack of portability between devices and very long compilation times (around 15 hours for some of the builds presented here) are heavy constraints that have likely prevented their usage in the HPC context. However, the recent availability of large FPGA devices combined with more user-friendly programming frameworks like OpenCL [10], OpenACC [20], Maxeler MaxJ [2] and lately the arrival of Xilinx Vitis and Intel oneAPI [23], could open the path to their usage in coming exascale systems.

FPGAs are being tried and used in a wide array of fields and applications such as medicine coupled with neural network [18], Convolutional Neural Networks [21], financial simulation [19], numerical simulations on unstructured meshes [13], quantum dynamics [17] and molecular dynamics [11]. In the context of material science, specialized hardware dedicated to MD simulations has been created such as the Anton machine [4]. An Anton machine consists of application-specific integrated circuits (ASICs), which are very

efficient but cannot be changed once created. Recently, important progress has been made [24, 25] to also employ multi-FPGA systems for this type of MD simulations. These studies differ from our work in that system sizes and accuracy requirements allow them to focus on the asymptotically good scaling approaches with box-cell filters for short range forces and projection based 3D FFT implementations for long ranged forces. Another study [15] highlights strong scaling of all-to-all forces over multiple FPGAs, but does not cover the periodic forces needed for MD.

This work targets the Metalwalls [5] MD code that is tailored to calculate highly accurate charge densities for electrodes of supercapacitors and is more work intensive. The contributions of this article include firstly a chip to chip comparison between a FPGA, CPU and GPU of the same generation in terms of both energy and execution time for this workload. Secondly, we perform a scaling study and a portability study targeting two systems using multiple FPGAs from different vendors. To the authors knowledge, this is the first case of FPGA-accelerated supercapacitor simulation on multiple FPGAs and one of very few studies using different FPGA tool chains for relevant applications.

This paper is organized as follows. In Section two, the original CPU code and GPU reference are described as well as the computing kernels that are ported to FPGA. In Section three, the main FPGA implementation is discussed as well as the methodology and Maxeler tools used to design and optimize it. In Section four, the results are discussed both in terms of performance and energy efficiency, before presenting first scalability results to multiple FPGA. In Section five, the scope of the paper is broadened to a port with the newer oneAPI programming environment to other FPGAs and another multi-FPGA system architecture.

2 METALWALLS

2.1 Description of Metalwalls code

Metalwalls [5] is a classical molecular dynamic code created by P. Madden, currently developed and maintained by M. Salanne from Sorbonne University, Paris, France. The code purpose is to accurately simulate electrochemical systems like supercapacitors, devices able to store energy in electrostatic form. Figure 1 shows the type of system simulated at molecular scale: two carbon electrodes (blue structures) immersed in an ionic liquid (green and red spheres). By applying electric potentials Ψ_Ω and $-\Psi_\Omega$ on the two electrodes, ions migrate within the liquid and ion adsorbtion take place at the electrodes. As a result, electrodes play the role of capacitors, accumulating a charge and thus storing energy. The purpose of such numerical experiments is to evaluate how the electrodes' respective charges evolve in time for different setups. The atoms of the electrodes are represented with gaussian charges and the electrostatic potential Ψ_i felt by each electrode's atom i reads:

$$\frac{\partial U_{elec}}{\partial Q_i} = \Psi_i \tag{1}$$

with U_{elec} the electrostatic energy of the system and Q_i the charge carried by atom i. As the potential on each electrode should remain constant, we have $\Psi_i = \Psi_{\Omega\pm}$ depending on the electrode in which i is located. So at each time step, the simulation computes a new charge density on the electrodes' atoms $\{Q_i\}$. This charge density is the one that minimises the total energy:

$$U = U_{elec} - \sum_{i=1}^{N} \Psi_i Q_i, \tag{2}$$

and a matrix free conjugate gradient is used to compute this minimum.

The code is written in Fortran 90 and is parallelised with MPI. The application studied in the context of this paper is the core of Metalwalls stripped from the time evolution aspects of the physics. As can be seen on Figure 2, in the overall algorithm, it is the computation of the electrostatic potential that has the largest contribution to execution time every time step, which is why it is the focus of this study, and which is why several physics modules were taken out (slashed boxes). Given a bulk configuration, i.e. atoms' position of the ionic liquid, the application studied in the context of this paper computes the charge density on the electrodes only once.

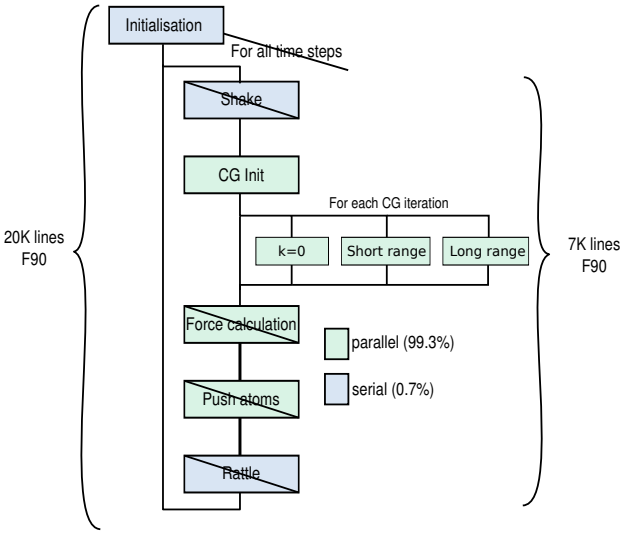

Figure 2: Molecular dynamics algorithm

2.2 Ewald summation

Ewald summation is a method for computing long-range interactions in periodic systems. It is very precise but has a high numerical cost. As electrostatic interactions are of primary importance in this context, it is used to maximize accuracy. In Metalwalls it is used for the Coulomb potential as well as the charge-dipole and the dipole-dipole interactions with 2D or 3D periodic boundary conditions. The idea is to split these calculations into a short range and a long range part. An incomplete way of looking at this algorithm but sufficient for this paper, is to consider the short range part to be computed in real space whereas the long range part is computed in Fourier space. All derivation and implementation details can be found in [16] and [6].

The distance r_c is the distance between two atoms above which their interaction is considered as long range. In this case the energy of this interaction is computed in Fourier space, otherwise it is

computed in real space. From a purely numerical point of view, each component of the total energy can be decomposed in four terms:

$$U = U_{\text{sr}} + U_{\text{lr},0} + U_{\text{lr},+} - U_{\text{self}} \quad (3)$$

where $U_{\text{sr}}, U_{\text{lr},0}, U_{\text{lr},+}$ and U_{self} are respectively the contributions for the short range part, the long range part for mode 0, the long range part for all other modes and the self interaction part.

2.3 CPU implementation

A single Conjugate Gradient (CG) iteration sums up the contributions coming from the four kernels $U_{\text{sr}}, U_{\text{lr},+}, U_{\text{lr},0}$ and U_{self}. The parallel implementation uses MPI for work distribution. In this setup, each rank has a copy of all data, only computations are distributed among ranks. Even if such a strategy does not scale in memory for a very large number of ranks, it is very efficient in this case. Thanks to a very low memory footprint, ~ 50MB for the test-case considered, which is already a large one for electrochemical systems, the application can be deployed without any memory problem on any recent compute node. This distribution saves communications and load balancing efforts between ranks when atoms move in the liquid as time evolves that would be mandatory in a domain decomposition parallelisation. The four kernels use all resources for their computations and are then executed one after the other. Contributions are then summed up on each rank. To update all ranks, a MPI reduction on the solution vector is performed once per CG iteration, i.e. all partial sums of the full vector length owned by each rank are summed up to get the final solution vector. This is the single moment that requires synchronisation and communication thanks to the "work only" distribution strategy. In term of algorithmic complexity, $U_{\text{lr},0}$ and U_{sr} kernels are in $O(N^2)$, $U_{\text{lr},+}$ kernel is in $O(N * M)$ and U_{self} kernel is in $O(N)$, with M the number of modes and N the number of atoms. In particular the reason why the U_{sr} kernel is in $O(N^2)$ here and not the $O(N)$ expected when using a box-cell list, is because of the size of the box in the production test case. The cutoff being about half of the smallest dimension (80.4 and 182.8 respectively, while the other two dimensions are 194.8 and 421.4 after rounding) prevents such an acceleration that would be possible when the cutoff is very small compared to the sizes of the box.

This CPU implementation has been extracted from the production code. It has proven its efficiency for graphene [14], with the model running for weeks on 512 cores while maintaining a parallel efficiency above 75%. As such, this code is a relevant baseline to perform comparisons with FPGA implementations running production test cases that lead to significant scientific publications in chemistry. Our implementation focuses on the conjugate gradient and the electrostatic computations done in one single time step.

2.4 GPU implementation

The GPU implementation was developped using OpenAcc. It computes all Kernels one after another once the input vectors are copied to the HBM (in the case of a P100 Nvidia GPU) with *data present*. A cache blocking strategy is then used for each kernels execution which uses *parallel loop* and *loop reduction*. An *atomic update* is used in the respective inner loop of the kernels $U_{\text{lr},0}$ and $U_{\text{lr},+}$ to ensure the validity of the computations. For the production test case, one iteration of the conjugate gradient takes 144 ms on the P100, almost 8 times faster than the execution on the full CPU used in this paper.

The reporting system of the supercomputer on which the computations were run indicates an efficiency of 91.6% of the GPU usage. 17.5% of the time was used by the $U_{\text{lr},+}$ kernel, 40.7% for the $U_{\text{lr},0}$ kernel, 41.8% for the U_{sr} kernel. This seems to confirm the memory bound aspect of our implementation on the P100 as, even though there is a factor 2 between the two kernels' algorithmic complexity, the sr kernel has twice the bandwidth requirement of the $U_{\text{lr},0}$ kernel, in this context it is logical that they would have roughly the same execution time. Furthermore, the $U_{\text{lr},+}$ kernel also represents a bigger computing time than expected based on algorithmic complexity while requiring an even bigger bandwidth.

2.5 Numerical accuracy analysis

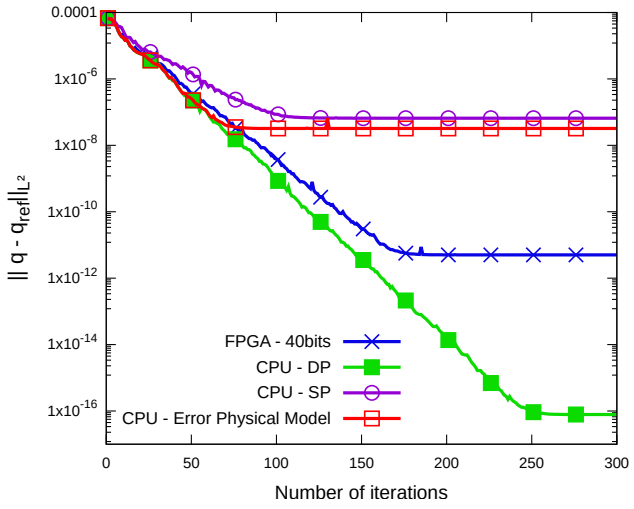

Figure 3: Error of the physical model compared to the error of the different numerical floating point representation

As a preliminary study before working on the FPGA implementation, an analysis was done to estimate the minimal numerical precision required for the computations to retain results with physical meaning. On a FPGA, one is not limited to standard IEEE 754 single or double floating point number representations as architecturally any number of bits can be used for the mantissa and exponent that represent a floating point number. The question is to what extent is it possible to lower the precision of number representation while keeping the correct physics.

The first step is to determine the precision of the physical model. The parameter $\boldsymbol{r_c}$ that splits computations between the real space and the Fourier space does not impact the solution in theory. However, it does have an influence numerically.

To measure this influence, a reference solution has been generated by running the code with quadruple precision for a representative $\boldsymbol{r_c}$. The quadruple precision (floating point format of 128 bits with 16 bits exponent and sign, 112 bits mantissa) used was the Fortran $real16 = selected\_real\_kind(p = 33)$ that follows the IEEE

754-2008 standard. With this reference, errors could be computed for different precisions and different rcut values. Runs (complete conjugate gradient convergence processes) were made with the reference r_c using respectively single precision (SP), double precision (DP), and a 40 bits FPGA implementation to compute the error compared to the quadruple precision run. An average of the different errors computed for various runs with different rcuts values using double precision shows the order of the physical model numerical accuracy.

Figure 3 shows that using double precision results in data exceeding by a significant margin the physical model's precision while single precision is not good enough. Using a 40 bit floating point representation (8 bit exponent and 32 bit mantissa, down from 11 and 53 respectively in double precision) guarantees a numerical precision greater than that required by the physical model by a large enough margin. Performing arithmetic operations on a FPGA with numbers represented with less bits can result in large savings in on-chip computing resources. Saved resources on the FPGA can potentially be used to improve the performance, by increasing the level of parallelism for instance. However, reducing precision, even without damaging the physics, leads to slower convergence rates in iterative algorithms like the conjugate gradient. This means that resources savings do not translate completely into a higher performance due to the additional iterations. In our production test case, the additional iterations number varied between 25% to 50% of the DP runs' number of iteration. Since in our study the area savings by lower precision lead to a disproportionally higher increase in parallelism and throughput, this tradeoff is acceptable and will result in overall improved CG execution time.

3 FPGA IMPLEMENTATION

3.1 Maxeler hardware and software stack

The FPGA based computing solution proposed by Maxeler is a Data Flow Engine (DFE), an accelerator board composed of a FPGA and DDR memory. Eight boards of this type are plugged into a MPC-X node via PCIe. The MPC-X node is packable in a cluster and is then connected to host CPU nodes via Infiniband.

On top of this hardware, Maxeler provides a high level programming abstraction for DFE programming. One manipulates Java objects to design computing kernels with data streams, counters and accumulators, and link them automatically with FIFO buffers or add one or several memory interfaces to the DDR memory of the DFE. The execution of the java code generates the VHDL that corresponds to the design and it is given as input to the FPGA vendor toolchain in order to generate the final bitstream. This last compilation step can last for several hours, even tens of hours. Therefore it is crucial that the java code can also be directly integrated into a simulator to test the output a real FPGA would give. It allows the developer to test the correctness of the implementation on small datasets without having to run on an actual device. This enables the standard incremental development workflow: new feature implementation, testing the new implementation to verify its correctness, estimate the resource consumption and performance based on a report, and only then compile for real hardware run to make performance measurements and do production runs.

3.2 Kernels design

```
1 void k0PotCPU(double *cst,int n,double *z, double *q,
    double *V) {
2   for (int i = 0; i < n; ++i){
3     V[i] = 0;
4     for (int j = 0; j < n; ++j){
5       V[i] += f(z[i], z[j], q[j], cst, i, j); }}}
```

Figure 4: Computing kernel structure for $U_{\mathrm{lr},0}$

Figure 4 shows the structure of the original $U_{\mathrm{lr},0}$ kernel. It consists of two nested loops spanning all atoms and computing the energy of interaction for each (i,j) pair accumulated in the vector V.

On the FPGA, the computing instructions are implemented in a pipe into which the inputs are streamed in while the outputs are streamed out. In order to have an efficient design, multiple pipes are used. Pipes can be seen as vector lanes of a CPU vector unit. A dedicated vector unit is built here for our algorithm. It follows the same SIMD principle where the same operation is applied on different data during the same clock cycle.

All data is either first transferred to the DRAM of the DFE thanks to an initial function call in the application running on the host, or directly streamed from the host to the FPGA. In the first case, a second function call triggers the calculation of a single CG iteration and the first data is then streamed to the FPGA. Contiguous data are read in the DRAM and streamed into the kernel in order to fill all pipes of the kernel at each cycle (i.e. maximizing throughput). In both cases, once arriving on the chip, data is stored onto on-chip memory and starts to be used in computations as they flow through arithmetic operations (additions, multiplications) and mathematical functions (sine, cosine, erf, exponent). In this kernel, the value V[i] is computed thanks to an accumulator that will sum the contribution of the inner loop during n cycles, after which the value is sent to the CPU memory.

This design leverages fast on-chip memories (block RAMs) that can be accessed within a single clock cycle. With this implementation, using P pipes, the memory bus is used only during the first N/P cycles to load all the data onto the chip but it can become a bottleneck if the P is big enough and the FPGA workload too small. Moreover, the on-chip strategy hits a limit when the datasets are larger than the available on-chip memory, however the largest system studied so far in the context of this code still fits comfortably. The PCIe bus is minimally used here after the first N/P cycles as it transfers only one value every N/P cycles and this transfer is almost transparent as it is overlapped with the computations of the DFE. Special care has to be taken to handle the general case where N is not a multiple of P. Memory padding and validation flags in the design have to be properly set. It is worth noting that the array-of-structure layout that stores closely in memory all information about an atom is more relevant in the FPGA context. The dedicated vector units designed for our algorithm indeed require all data related to an atom be available within the same clock cycle. This feature is orthogonal to the structure-of-array layout required on CPU and GPU for best performance.

The U_{sr} and $U_{\mathrm{lr},+}$ kernels exhibits very similar structures and thus follow the same DFE implementation principle. Besides a loop

on the number of modes in the $U_{\text{lr},+}$ kernel, the major algorithmic difference is the way outputs are performed. For U_{sr} and $U_{\text{lr},0}$ kernels, the results are sent out every N/P cycles whereas with $U_{\text{lr},+}$, all outputs are done in the last N cycles. The different kernels outputs are thus not synchronized which prevents summing the outputs of these three kernels on the FPGA as it would not meet the hardware scheduling requirements. Because of this issue, the final sum is performed on the CPU along with the computation of the self potential contribution U_{self}, which is computationally negligible. Also since the data is already on the CPU at this point, it makes sense to avoid a non trivial implementation of the conjugate gradient computation on the FPGA and simply do it on the CPU. Once the residual is computed, if convergence is not reached, the updated input vector is sent back to the FPGA's DRAM and the next CG iteration is triggered. Otherwise, the algorithm is terminated.

The only drawback of the DDR based design is that it is harder to obtain the timing closure during the hardware compilation, making it harder to use all the chip resources available and harder to increase the clock frequency as well. Timing closure is the synchronization of the different components of the FPGA used in our design. It is necessary to guarantee accurate results. FPGAs primarily consist of logic, which includes LookUp Tables (LUTs) that have custom truth tables, and flipflops that are binary shift registers used for synchronizing the logic. They also contain on-chip memory such as Ultra RAM (URAM) and Block RAM (BRAM), as well as Digital Signal Processors (DSPs) that are used for computing. Achieving timing closure while maintaining performance on a FPGA requires minimizing the amount of logic used in the design, while maximizing the number of DSPs available to perform floating point operations.

The second design on the other hand is identical except it loads data directly from the host memory to the FPGA on-chip memory. Since it does not make use of the FPGA DRAM, it does not have the issue aforementioned. Although, in this design the memory bus becomes a bottleneck much faster. As shown later on, the design using direct streaming has been our best performing design.

3.3 Compiling designs with multiple kernels

Compiling designs consists of running the full compiler toolchain for our implementation in order to obtain bitstreams that will configure the FPGA such that it implements our algorithm. There are three main parameters to set to maximize performance.

The first one is the number of pipes used in each kernel. A performance model has been developed in order to evaluate the potential bottlenecks that would be faced. As the state of the DFE is known at each clock cycle and the cost of operations is known, calculations with a spreadsheet can predict how the design is able to perform and how much of the resources of the FPGA it will use [22]. When the number of atoms N and the number of modes M is known, it is trivial to derive the number of cycles needed by each kernel. As all kernels run concurrently, maximizing performance means having a total cycle number as close as possible for every one of them in order to keep all pipelines filled. As a consequence, it is possible to find a balance for the number of pipes P for each kernel since the number of cycles is inversely proportional to it. For instance, the production test case shown here has to follow the ratio (8,4,1) for the number of pipes for kernels ($U_{\text{lr},0}, U_{\text{sr}}, U_{\text{lr},+}$).

Table 1: Resource usage of the multiple kernels designs with direct streaming relative to 1182240 LUTs, 6840 DSPs, 4320 BRAM18 & 960 URAM memory blocks on-chip.

Design Name	64 bits Design	40 bits Design	Final design
Design frequency (MHz)	200	200	300
Pipes ($U_{\text{lr},0}, U_{\text{sr}}, U_{\text{lr},+}$)	(8,4,1)	(16,8,2)	(32,16,4)
Logic (LUTs)	31.75%	27.9%	50.6 %
DSPs	33.7%	27.9%	53.8%
On-chip Mem	34.4%	20.8%	30.5%

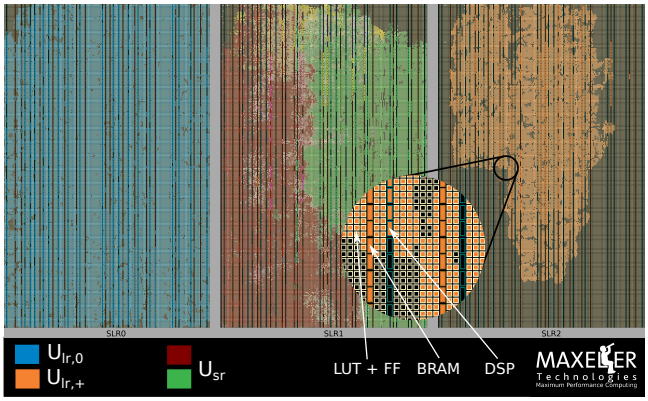

Figure 5: Multi kernel design's resources map

The second parameter to tune is the FPGA frequency which can in practice go from 100 to 350MHz for the model available in the DFE (higher frequencies are mostly feasible for designs using less area). The higher the frequency, the more computations per second. However, increasing frequency or chip usage by adding pipes makes it harder to meet timing closure, hence a balance needs to be found.

Table 1 summarizes the design evolution with the direct streaming strategy. Compilation time for the builds took between 13 hours and 32 hours each which shows again how important the design work is, as it saves a lot of compilation time by narrowing down the number of builds to test. The savings due to precision reduction enabled to double the number of pipes in each kernel for the 40 bits builds. During the initial design phase the frequency used was 200 MHz which was pushed to 300 MHz for the final optimized builds while the total number of pipes was multiplied by 4 from 13 to 52.

A final remark on the design compiled for this Xilinx hardware: this specific FPGA chip is made of three Super Logic Regions (SLRs) connected by solders which allow limited communications from one SLR to another. Since we have three separate kernels which is as many as the number of SLRs, the third design decision was to confine each kernel to its own region. This makes the compilation easier as the resources available to the compiler for each kernel are explicitly defined this way and the kernels don't have to compete for the same resources. In order for the kernels to finish their computation at the approximate same time, the number of pipes in the kernels respects the (8,4,1) ratio. An immediate consequence of the combination of this design choice and the synchronicity constraint,

is that the kernel that consumes the most resources becomes the limiting factor of the whole design. In our case it is the $U_{\mathrm{lr},0}$ kernel which fills 87% of its SLR while the two others have more resources left as can be seen in the resulting floor plan in figure 5.

As discussed, the (8,4,1) ratio is ideal for the production test case but not for all test cases. A shortcoming of trying to design all kernels on only one FPGA is that no load balancing can be done between the kernels at execution time. This issue will be addressed later with specialized designs described later in this paper.

4 RESULTS

4.1 Computing infrastructures

The Jumax machine from the Juelich supercomputing center was used for FPGA development and evaluation. It consists of two build nodes and one node for FPGA executions. The host node consists of dual socket Zen 1 AMD EPYC 7601 32-Core Processors and the Maxeler MPC-X consists of 8 MAX5C boards, comprising a Xilinx VU9P FPGA and 48 GB DDR each. CPU and GPU timings and energy have been measured on PlaFRIM[1], The CPU is an Intel Xeon Gold 6148 (skylake) running at 2.4GHz. The GPU is a NVidia P100.

4.2 Energy measurements

FPGA energy measurements were made by Maxeler on a dedicated single server node containing a single VU9P with a counter based method measuring the PCIe energy usage for the whole FPGA board. The measurement was then extrapolated to all FPGAs.

CPU and GPU energy measurements are also counter based. The EnergyScope Optmize [3] technology has been used to profile the energy requirement of these two devices. It catches the energy data provided by processors and GPUs at a fine-grained (30-1 Hertz) frequency. For Intel and AMD EPYC processors, the data is obtained from the Model-specific register (MSR) registers. For the NVIDIA GPUs the data is obtained by the nvml software package. EnergyScope Optimize aggregates the data acquired in every computational unit and in every node to deliver a single temporal reference signal. This tool has been presented at [7] and [8] and used in [1] for an application in aeronautical industry.

In the context of this paper, for each application run, job initialisation and finalisation phases are suppressed from the temporal reference signal, keeping only the application runtime. Application startup is detected by a steep increase in power usage. The remaining power values are then averaged and this mean power represents the power measure for a single run. Ten runs have been performed on three different NVidia P100 GPU running on different compute nodes and ten additional run on a single socket Intel Skylake CPU. For each architecture, the average of the mean power on all runs is used in Fig 6 as average power requirement, error bars being the standard deviation.

4.3 Test cases

The production test-case used in this paper simulates the system studied in [14] using 42508 atoms. It is a numerical experimentation of potential new supercapacitor technology that uses graphene electrodes. Large carbon planes with thickness of a single atom and pores that play the role of electrodes. The objective is to evaluate numerically the efficiency of such a system before performing real but costly experiments.

[1] https://www.plafrim.fr/

4.4 Chip to chip comparison

Figure 6 shows the execution time per iteration, the power requirement and the performance per Watt for the CPU, the FPGA and the GPU implementations. In terms of performance measured here with the time per CG iteration metric, the FPGA's performance is 5 times better than the CPU's and reaches 60% of the GPU's .

In term of power requirement, the FPGA chip needs three times less energy than the Intel skylake CPU chip to run at full load and almost four times less energy than the Nvidia P100 GPU chip. Of course, this power estimate does not take into account the needs of the CPU host node which is required to trigger the FPGA the same way we do not take into account the CPU host node for the GPU. Indeed, energy measurement were not available on the hosts, for the FPGAs or the GPUs. However, a rough estimate for the power requirement of an idle processor is around 100W. This would bring the global power requirement of the FPGA system to 150W, lowering the power gain of FPGA to 6% compared to the CPU and 47% compared to the GPU (which with the idle CPU would consume about 285W). Controlling several FPGAs with the same CPU would already reduce the impact of the CPU host power. This shows that if the FPGA technology is very efficient in term of average power, energy efficient host nodes are still required to have energy efficient FPGA based computing systems.

The number of iterations per second and per watt metric is similar to an energy to solution criterion and it combines the two previous aspects. Considering this metric the comparison to CPU leads up to a factor 16 for our test case, a huge gain in energy efficiency for the same computation. Compared to the GPU, the FPGA is performing better by a factor over 2. Finally our measured computing time with the FPGA is the same as the theoretical computing time given by the number of cycles divided by the frequency. The fact that both numbers are the same attests to the efficiency of the current implementation and shows the good predictability of the FPGA computing time. It also shows the usefullness of the performance model to assess our results.

4.5 Performance with multiple chips

The parallelization on multiple FPGAs was based on the following strategy: computations were divided on the CPU so that each FPGA

Table 2: Resource usage of the single kernel designs relative to 1182240 LUTs, 6840 DSPs, 4320 BRAM18 memory blocks on-chip.

Design Name	Design $U_{\mathrm{lr},0}$	Design U_{sr}	Design $U_{\mathrm{lr},+}$
Design frequency (MHz)	300	300	300
Total number of pipes	96	48	42
Logic (LUTs)	52.9%	67.5%	63.2%
DSPs	87.2%	55.4%	83.5%
On-chip Mem	27.2%	25.8%	38.4%

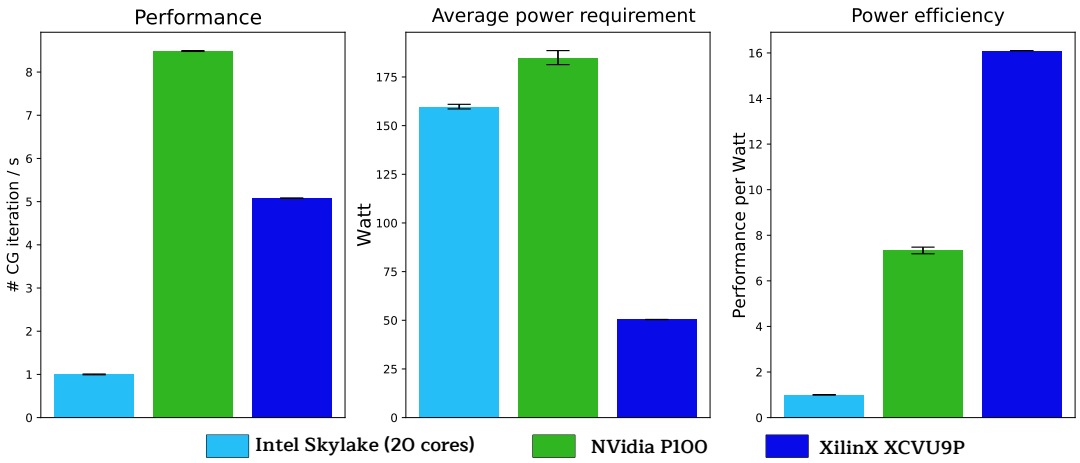

Figure 6: Execution time, power requirement and performance per Watt comparison for CPU, FPGA and GPU

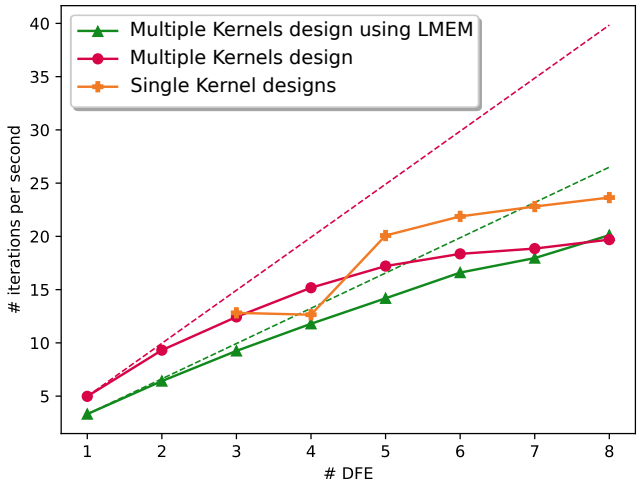

Figure 7: Speed up of the different designs

would compute a part of the solution vector similar to the MPI implementation. These parts would then be sent back to the CPU to be summed into the final solution vector, ending the iteration. If a new iteration was needed because the residue was not small enough, the new solution vector would be sent back to the FPGAs. Two designs were tested initially: the first approach used the multiple kernels design described in the previous section. The work is divided across MPI ranks, one per FPGA, and the computations are accelerated on the FPGAs. At Each iteration, the results are synchronized thanks to one MPI_ALL_REDUCE() call once the outputs from the FPGAs have been collected. This approach requires only one FPGA design. As can be seen on figure 7, we observe better performance as the number of FPGAs increases but because of the communications increase, the scaling is poor. Indeed the total bandwidth for the connection between the CPU node and the FPGA node is only up to 4 Gb/s which is quickly congested as the number of FPGAs is increased.

The other approach, more resource effective, was to specialize FPGAs for a single kernel in order to better fill the chips. The same kernel was implemented in each SLR, making them completely synchronous. Table 2 shows how much of the chips resources (in particular the number of DSPs) were used for the single kernel designs. Of course since three separate designs were required for this approach to work, it took a longer time to compile and test compared to the multiple kernels design.

Another difference compared to the first approach is that the workload was balanced on the number of FPGAs assigned for each design. The load balancing was done on the CPU by choosing how many FPGAs are assigned to which of the 3 specialized designs. In order to find the most efficient balance, the times of all combinations were computed. Starting at three FPGAs, it is trivially one FPGA per design. But for four FPGAs, the extra FPGA is not taken advantage of. Indeed the designs $U_{\mathrm{lr},0}$ and U_{sr} need the same number of FPGAs in our case while the $U_{\mathrm{lr},+}$ is much faster than them and does not require more FPGAs. As a consequence, the extra FPGA accelerates one of the first two designs but the other one takes as much time as before. The total time is even higher than with three FPGAs because of the additional communications overhead. With five FPGAs though, we can assign 2 FPGAs each to the first two designs, and observe a performance twice as good as with three FPGAs. Unfortunately with six and more FPGAs the communications overhead is such that the scaling observed is far from the theoretical scaling.

Both approaches are bottlenecked by the PCIe Gen2 x8 connection between the CPU node and the FPGA nodes. In order to remedy to this issue, using the FPGA's DRAM to save on bandwidth was included in the design even though it meant decreasing the number of pipes by 33% and the frequency (from 300 MHz to 260 MHz) in order to achieve timing closure. This results immediately in a worse single chip performance but gives much better scaling. Although, it only manages to overcome the multiple kernels design performance that does not use the FPGA DRAM at 8 FPGAs. As a note, PCIe Gen3 x16 can achieve four times greater bandwidth, up to 16GBs.

Had the system been using that technology, we can assume that the results would have been able to scale much better with all designs.

5 INTEL ONEAPI AND STRATIX 10 VARIANT

Considering the results in the previous sections or other case studies that show promising performance and efficiency for some HPC applications on a specific combination of FPGA hardware and development tools, a crucial question comes up about the portability of designs or insights to other FPGA targets. In this section, we investigate such a port to the Intel oneAPI development environment, which is an implementation and extension of the SYCL standard and to the Intel Stratix 10 FPGA architecture. As unified programming model for Intel CPUs, GPUs and FPGAs and conceptually also open to other vendors, oneAPI has the potential to become a more lasting and widely adopted tool flow than Maxeler MaxJ. In contrast to the MaxJ language, which describes the dataflow *architecture* on the FPGA in terms of components like streams or memory blocks along with their behavior, oneAPI and SYCL code builds upon the C++ syntax to specify the *functionality* of the FPGA design. This on the one hand improves the accessibility of this approach for developers from the HPC domain, who are often familiar with C/C++ programming. But on the other hand it creates a larger gap or ambiguity between the code and the final architecture (see also [12]); ultimately with both tool flows, the designs running on the FPGA are actually quite similar in order to exploit the potential of the hardware.

Targeting the Stratix 10 FPGAs in Bittware 520N cards hosted in *Noctua 2* at Paderborn Center for Parallel Computing (PC2), we want to confirm at the same time, if a different system architecture can overcome the observed scalability limits. We build upon the insights from the Maxeler/Xilinx design experience, and directly aim for three individual single kernel designs to be distributed and load balanced via a corresponding MPI host implementation.

5.1 oneAPI kernel designs

The core of the FPGA kernels implemented with oneAPI preserves the nested loop structure of the reference code as depicted in Fig. 4. The compiler automatically tries to pipeline the loops in a way that in each cycle one iteration of the inner loop starts (denoted as *Initiation Interval*, here *II 1*) and the inputs for the respective iteration (in Fig. 4 defined by a pair of `(i,j)` indices) are fed into a specialized datapath for the respective function (in Fig. 4, 6 inputs for the force calculation function `f`). Once the pipeline is filled, every cycle one result is emitted at the output port of the datapath. Up to here, this is the same structure as generated with MaxJ, with the main difference in the resource consumption and latency of the arithmetic function blocks selected by the respective tools.

For the accumulation, the developer has for the first time to manually adapt the code in order to maintain a pipeline with II 1 by implementing the shift register pattern as outlined in the Intel documentation [9]. The second transformation for all kernels is to enable on-chip memory buffers for the data that is reused through multiple iterations of the outer loop. Declaring arrays within the kernel scope with a compile-time constant size (here an upper limit of supported atoms) achieves this. In a third step, parallelism is added via an unrolling annotation for the inner loop. For the designs presented here, this works well only for powers-of-two, whereas for other factors, additional effort is required to support the compiler finding the right banking scheme for local memory. Reconsidering the accumulation in this context, it is most resource efficient to first perform a parallel (and latency insensitive) reduction over partial sums within a parallel block, before feeding the result into a single shift register. Figure 8 summarizes the updated structure for the $U_{\mathrm{lr},0}$ kernel in contrast to the reference baseline in Figure 4.

For the U_{sr} and $U_{\mathrm{lr},+}$ kernels, the accumulation was outsourced to a separate accumulator kernel with the same structure as the compute kernel by receiving respectively sending the partial block sum (`v_blk` in Fig. 8) via a SYCL *pipe*, which gets translated into a simple FIFO on FPGA. For the U_{sr} kernel, this resolves an II bottleneck introduced by the compiler for the index calculations along with a subsequent condition (`i != jb+jj`) in the force calculation. For the $U_{\mathrm{lr},+}$, it helps to relax a dependency on intermediate results per atom and thus allows to also fully pipeline the outer loop, processing one `(i,j)` pair per pipe per cycle once data is available on chip, like all other oneAPI and Maxeler designs presented here.

5.2 Compiling and analyzing oneAPI kernels

The three kernels along with their respective host interface are compiled into separate shared library objects, each containing one corresponding bitstream. With the first synthesis results, an accuracy and convergence problem showed up that was not visible in emulation. We identified the hardware implementation of the error function `erf(double arg)` and `erfc(double arg)`, used in the $U_{\mathrm{lr},0}$ and U_{sr} kernels respectively, to contain an approximation that is not sufficiently accurate for the application requirements.

```
1  void k0PotOneAPI(double *cst,int n,double *z,double *q,
       double *V) {
2    double Vl[n_max], zl[n_max], ql[n_max];
3    // loop to fill local memory skipped here
4    for (int i = 0; i < n; ++i){
5      double s_reg[II_CYCLES] = {0};// shift register
6      for (int jb = 0; jb < n; jb+=U_BLOCK){
7        double v_blk = 0.0;
8        #pragma unroll
9        for (int jj = 0; jj < n; jj+=U_BLOCK)
10         v_blk += f(zl[i],zl[jb+jj],ql[jb+jj], cst,i,j);
11       // here hiding the latency of accumulation
12       s_reg[II_CYCLES-1] = v_blk + s_reg[0];
13       // parallel shift left operation
14       #pragma unroll
15       for(uchar s=0; s<II_CYCLES-1; s++)
16           s_reg[s] = s_reg[s+1];
17     }
18     double v_sum = 0.0;
19     // reduction of partial sums from shift register
20     #pragma unroll
21     for(uchar s=0; s<II_CYCLES-1; ++s)
22         v_sum += s_reg[s];
23     Vl[i] = v_sum;
24   }
25   // loop to write back Vl skipped here
26 }
```

Figure 8: Outline of structural changes to kernel $U_{\mathrm{lr},0}$ for oneAPI.

Deviations of individual `erf()`-calls from the C standard library reference reached up to $1x10^{-4}$. To fix the functionality, we referred to manually calling an accurate implementation of `erf()`, which uses the same polynomial approximation as employed in the glibc[2] library and originally developed at SunPro, a Sun Microsystems, Inc. business. We performed some area optimizations by first selecting the coefficients based on one of multiple possible input intervals and then reusing the same arithmetic resources for the floating point operations. With this, we are able to synthesize fully functional double precision designs of all kernels, however the area overheads are still large.

Table 3: Resource usage of oneAPI single kernel designs relative to 705500 ALMs, 4713 DSPs, 9094 block RAMs available for kernels. Effect of unrolling here also denoted as pipes. In brackets for $U_{\mathrm{lr},0}$ and U_{sr} estimates with efficient but inaccurate erf approximation.

Design Name	$oneU_{\mathrm{lr},0}$	$oneU_{\mathrm{sr}}$	$oneU_{\mathrm{lr},+}$
Frequency [MHz]	400.00	396.67	322.50
Number of pipes	8	4	8
Logic [ALMs]	62.5% (30.1%)	69.4% (33.2%)	51.3%
DSPs	35.9% (14.2%)	27.3% (13.5%)	39.1%
On-chip Mem	14.5% (14.9%)	27.9% (24.2%)	44.8%

Table 3 summarizes the parallelism, resource consumption and clock frequencies of kernels synthesized with oneAPI. When comparing these results to Table 2, we need to briefly touch upon the different metrics reported here. Both the Xilinx VU9P targeted by Maxeler and the Intel Stratix 10 targeted by oneAPI group multiple LUTs together with twice more FFs into blocks denoted as CLBs for Xilinx (8 LUTs per CLB) and as ALMs (2 LUTs per ALM) by Intel. In the reports summarized here, the Maxeler tool output reports the actual used LUTs (and FFs), whereas the Intel Quartus output reports any ALM as utilized in which at least one LUT or FF is utilized. Also, the numbers in Table 2 include infrastructure components like PCIe (DDR controllers are not used and optimized away in the non-LMEM designs), whereas for Table 3, only the kernel region of the FPGA is considered, because a fixed shell with PCIe and DDR controllers blocks the remaining 20%-25% of the resources.

When actually analyzing the oneAPI results in Table 3 in comparison to Table 2, we see a small advantage in clock frequency, but a huge gap of 12x ($U_{\mathrm{lr},0}$ and U_{sr} kernels) and 5.25x ($U_{\mathrm{lr},0}$ kernel) in achieved parallelism. Firstly, regarding the clock frequency, oneAPI FPGA compilation and synthesis tools by default aim for an optimistic target clock frequency of 480MHz. After placement and routing, the tools automatically reduce the clock according to the identified latency of the critical path to a working configuration. Due to the current limitation of the designs to power-of-two parallelism, the presented designs do not approach the resource limits of the FPGA, which makes it easier to reach high clock frequencies, but overall leaves some performance potential unused. Looking further at the gap in parallelism, a factor of around 2 can directly be attributed to the manual `erf()` implementation in the first two kernels. Numbers in brackets in Table 3 indicate resource consumption with the inaccurate `erf()` function block provided by the oneAPI library. A function block that is just a bit more resource intensive, yet accurate, which is present in the Maxeler compiler, could reduce resource consumption roughly by half and accordingly enable higher parallelism.

[2]https://www.gnu.org/software/libc/sources.html

Table 4: [LUTs, DSPs] usage of basic arithmetic functions based on oneAPI reports and Maxeler documentation.

Tool/Target	oneAPI/Stratix 10		Maxeler/VU9P	
Data Type	double	ap_float<10,35>	double	float<8,32>
Multiplication	[3253, 14]	[504, 1]	[132 , 7]	[427, 0]
Addition	[3484, 18]	[1696, 5]	[582 , 3]	[95, 4]
Division	[6034, 50]	[1864, 21]	[3135, 0]	[1247, 0]
exp()	[5504, 20]	[2218, 9]	[1290, 38]	[555, 16]
sqrt()	[2613, 28]	[1029, 6.5]	[1653, 0]	[673, 0]
fused sin/cos	[7548, 43]	[7548, 43]?		
sin()			[1261, 28]	[771, 10]

After considering the unused resources and the impact of the `erf()` implementation, there is a parallelism gap of roughly 4–5x that still needs to be investigated. It needs to be noted, that a considerable amount of parallelism was added to the Maxeler designs only after the transition to 40 bit floating point arithmetic. With the `ap_float` type, oneAPI also provides a custom precision floating point implementation, however, the closest available type with at least that much precision contains 46 bits (10 bits exponent, 35 bits mantissa). In Table 4, we summarize the resource consumption for the floating point operations and functions frequently showing up within the three kernels. These numbers can not be considered as apples to apples comparison, since on the one hand the underlying architectures of LUTs and DSPs of the two FPGA families differs and on the other hand because of how these numbers were obtained. The oneAPI numbers are extracted from report estimates, where some of them are evidently adapted to the pipeline context they are embedded in and thus might differ in other designs or the final implementation. The Maxeler numbers are taken from their documentation, except for the erf() function, for which they were estimated back from synthesis results. This said about the limited comparability, there seems to be a clear trend pointing to either a more efficient floating point library provided by Maxeler, or the VU9P FPGA architecture being more suitable to combine logic and DSP resources into double precision and 40 bit floating point arithmetic. Many operations of the ap_float implementation also show promising improvements over the double precision baseline. However the sin/cos function block seems not to be adapted and in the full designs with a double precision host interface, frequent cast operations cause additional area overheads, such that so far we have not been able to achieve a higher degree of parallelism with any ap_float design. With more optimizations or upcoming oneAPI tool releases, it should be possible to shrink the parallelism gap.

5.3 Experiments

With the double precision oneAPI kernel designs, the production test case (Section 4.3) was executed on the FPGA partition of the

Noctua 2 system at PC2. Its nodes with Bittware 520N cards containing Stratix 10 GX 2800 FPGAs combine dual socket AMD Milan Epyc 7713 CPUs with two of the FPGA cards, each communicating with PCIe Gen3 x8 (physically x16, but not supported by the FPGA shell). The experiment was conducted such that each MPI rank was exclusively controlling one FPGA for the computations.

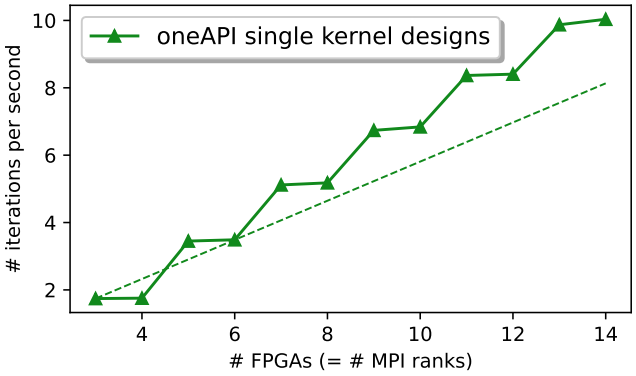

Figure 9: Multi-FPGA scaling with oneAPI

Figure 9 shows the performance in CG iterations per second from 3 (minimum required to run the single kernel designs) to 14 ranks. As was to be expected from the limited parallelism within each FPGA design, the performance does not match the Maxeler results, with 3 ranks reaching only about half the performance of the single-FPGA multiple kernel reference (Figure 6). However, the parallel speedups are perfectly matching the expectations based on per-kernel throughput, here displayed as superlinear speedup, since in the setup with 3 FPGAs that is taken as scaling baseline, the $U_{\mathrm{lr},0}$ kernel is underutilized and only catches up towards full occupancy. Once more and more U_{sr} and $U_{\mathrm{lr},+}$ instances are added, big performance jumps show up at odd FPGA counts, when the next pair of U_{sr} and $U_{\mathrm{lr},+}$ kernels is complete. These scaling results are promising, but higher performance per kernel needs to be achieved before coming to clear conclusions at which point communication might limit further scalability.

6 CONCLUSION

This paper compares the original CPU and GPU implementations of a matrix free conjugate gradient that minimises the total energy of a realistic electrochemical system with FPGA implementations. The first set of FPGA implementations were developed using the Maxeler software environment to be used on Xilinx hardware. A numerical accuracy study has enabled the usage of an intermediate floating point number representation using 40 bits, lying between the standard single and double IEEE754 representations, without impacting the physical results of the algorithm. The comparisons have been performed with a production test case of a size of 42508 atoms. Time and counter based energy measurements have been performed for all CPU, GPU and FPGA. The production test reveals that the FPGA is faster than the CPU by a factor of 5. The FPGA requires also almost 3 times less electrical power to deliver the same results. Combining these two features leads to an impressive factor of 16 when considering the number of iterations per second and per Watt. Compared to a GPU of the same generation, the FPGA only achieved 60% of its performance but the ratio of the performance per watt is nonetheless better by a factor of 2. The main limitation to scalability on this FPGA platform is similar to the one faced by GPU accelerated computing infrastructures: the interconnect between the host and the accelerator. It is even stronger in this case as the PCIe 2.0 interconnect has to be shared between eight FPGAs.

Considering the strong results of the chip to chip comparison, and despite the early scalability issues, the FPGA technology is, in our opinion, a clear candidate to be part of large scale HPC systems up to the exascale class. However, to this end the ecosystem of HPC capable FPGA accelerator hardware and development tools has to mature further, reducing frictions in development, portability and practical usage of this technology. In this sense, the Intel oneAPI development flow, that we utilized for the final part of this study, is a promising counterpart to the Maxeler MaxJ approach used for the main body of this work. While the performance of the Intel port is not fully competitive yet due to challenges with the floating point libraries, it has turned out that many of the high-level design experiences and decisions could be transferred between these quite radically different FPGA tools, whereas other optimization patterns had to be completely adapted to the different tool.

7 ACKNOWLEDGMENTS

This work has been funded by the EU H2020 project EXA2PRO (801015). The authors gratefully acknowledge the access to the PRACE-3IP PCP pilot system at Jülich Supercomputing Centre, which has been partially been funded by the European Union FP7 programme under grant agreement no. RI-312763 (PRACE-3IP), as well as the access to the Piz Daint system through the preparatory access request (2010PA5523) approved by CSCS, Switzerland and the access to CEA-TGCC (Irene/Joliot-Curie) under the project tgcc0076 at maison de la simulation .

Experiments presented in this paper for CPU and GPU were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr). Also, the authors gratefully acknowledge the funding of this project by computing time provided by the Paderborn Center for Parallel Computing (PC2). Finally, the authors gratefully acknowledge that part of this work was funded by ERC 757858 ATMO, https://doi.org/10.3030/757858.

REFERENCES

[1] Emmanuel Agullo, Marek Felšöci, Amina Guermouche, Hervé Mathieu, Guillaume Sylvand, and Bastien Tagliaro. 2022. *Study of the processor and memory power consumption of coupled sparse/dense solvers.* Research Report RR-9463. Inria de l'université de Bordeaux. 17 pages. https://hal.inria.fr/hal-03589695

[2] Rob Dimond, Sébastien RacaniÃÎre, and Oliver Pell. 2011. Accelerating large-scale HPC Applications using FPGAs. In *20th IEEE Symposium on Computer Arithmetic.* https://doi.org/10.1109/ARITH.2011.34

[3] EnergyScope. 2022. The EnergyScope general information site. https://www.linkedin.com/company/energy-scope/. [Online; accessed 8-December-2022].

[4] D. E. Shaw et al. 2014. Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 41–53. https://doi.org/10.1109/SC.2014.9

[5] Marin-Laflèche et al. 2020. MetalWalls: A classical molecular dynamics software dedicated to the simulation of electrochemical systems. *Journal of Open Source Software* 53, 5 (2020), 178–183. https://doi.org/10.21105/joss.02373

[6] T. R. Gingrich and M. Wilson. 2010. On the Ewald summation of Gaussian charges for the simulation of metallic surfaces. *Chem. Phys. Lett.* 500 (2010), 178–183. https://doi.org/10.1016/j.cplett.2010.10.010

[7] Hervé MATHIEU. 2021. A tool to measure the energy profile of HPC&AI applications. https://jcad2021.sciencesconf.org/data/Herve_Mathieu_energy_scope.pdf. [Online; accessed 8-December-2022].

[8] Hervé MATHIEU. 2022. To respond to the energy emergency, in particular with EnergyScope. https://jsmcia2022.sciencesconf.org/data/program/20221021_energyscope_jsmcia2022.pdf. [Online; accessed 8-December-2022].

[9] Intel. 2022. FPGA Optimization Guide for Intel oneAPI Toolkits (Rev. 13). https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi-dpcpp-fpga-optimization-guide.pdf.

[10] Z. Jin and H. Finkel. 2018. Evaluating an OpenCL FPGA Platform for HPC: a Case Study with the HACCmk Kernel. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2018.8547586

[11] Server Kasap and Khaled Benkrid. 2011. A high performance implementation for Molecular Dynamics simulations on a FPGA supercomputer. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 375–382. https://doi.org/10.1109/AHS.2011.5963962

[12] Tobias Kenter. 2019. Invited Tutorial: OpenCL design flows for Intel and Xilinx FPGAs: Using common design patterns and dealing with vendor-specific differences. In *Proc. Int. Workshop on FPGAs for Software Programmers (FSP), collocated with Int. Conf. on Field Programmable Logic and Applications (FPL)*.

[13] Tobias Kenter, Adesh Shambhu, Sara Faghih-Naini, and Vadym Aizinger. 2021. Algorithm-Hardware Co-design of a Discontinuous Galerkin Shallow-Water Model for a Dataflow Architecture on FPGA. In *Proc. Platform for Advanced Scientific Computing Conf. (PASC)*. 11. https://doi.org/10.1145/3468267.3470617

[14] Trinidad Méndez-Morales, Nidhal Ganfoud, Zhujie Li, Matthieu Haefele, Benjamin Rotenberg, and Mathieu Salanne. 2019. Performance of microporous carbon electrodes for supercapacitors: Comparing graphene with disordered materials. *Energy Storage Materials* 17 (2019), 88 – 92. https://doi.org/10.1016/j.ensm.2018.11.022

[15] Johannes Menzel, Christian Plessl, and Tobias Kenter. 2021. The Strong Scaling Advantage of FPGAs in HPC for N-Body Simulations. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 1 (November 2021), 30 pages. https://doi.org/10.1145/3491235

[16] S. K. Reed, O. J. Lanning, and P. A. Madden. 2007. Electrochemical interface between an ionic liquid and a model metallic electrode. *J. Chem. Phys.* 126 (2007). https://doi.org/10.1063/1.2464084

[17] José M. Rodríguez-Borbón, Amin Kalantar, Sharma S. R. K. C. Yamijala, M. Belén Oviedo, Walid Najjar, and Bryan M. Wong. 2020. Field Programmable Gate Arrays for Enhancing the Speed and Energy Efficiency of Quantum Dynamics Simulations. *Journal of Chemical Theory and Computation* 16, 4 (2020), 2085–2098. https://doi.org/10.1021/acs.jctc.9b01284 PMID: 32216276.

[18] Ahmed Sanaullah, Chen Yang, Yuri Alexeev, Kazutomo Yoshii, and Martin C. Herbordt. 2018. Real-time data analysis for medical diagnosis using FPGA-accelerated neural networks. *BMC Bioinformatics* 19, 1 (jan 2018). https://doi.org/10.1186/s12859-018-2505-7

[19] X. Tian and K. Benkrid. 2010. High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs.GPU. *ACM Trans. Reconfigurable Technol. Syst.* 3, 4 (november 2010).

[20] Ryuta Tsunashima, Ryohei Kobayashi, Norihisa Fujita, Taisuke Boku, Seyong Lee, Jeffrey S. Vetter, Hitoshi Murai, and Masahiro Nakao. 2020. OpenACC unified programming environment for GPU and FPGA multi-hybrid acceleration. In *13th International Symposium on High-Level Parallel Programming and Applications*. 114–133.

[21] Nils Voss, Marco Bacis, Oskar Mencer, Georgi Gaydadjiev, and Wayne Luk. 2017. Convolutional Neural Networks on Dataflow Engines. In *IEEE International Conference on Computer Design*.

[22] Nils Voss, Bastiaan Kwaadgras, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. 2021. On Predictable Reconfigurable System Design. 18, 2 (feb 2021), 28 pages. https://doi.org/10.1145/3436995

[23] Yong Wang, Yongfa Zhou, Qi Scott Wang, Yang Wang, Qing Xu, Chen Wang, Bo Peng, Zhaojun Zhu, Katayama Takuya, and Dylan Wang. 2021. Developing medical ultrasound beamforming application on GPU and FPGA using oneAPI. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 360–370. https://doi.org/10.1109/IPDPSW52791.2021.00064

[24] Chunshu Wu, Sahan Bandara, Tong Geng, Vipin Sachdeva, Woody Sherman, and Martin Herbordt. 2021. System-Level Modeling of GPU/FPGA Clusters for Molecular Dynamics Simulations. In *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC49654.2021.9622838

[25] Chunshu Wu, Tong Geng, Sahan Bandara, Chen Yang, Vipin Sachdeva, Woody Sherman, and Martin Herbordt. 2021. Upgrade of FPGA Range-Limited Molecular Dynamics to Handle Hundreds of Processors. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 142–151. https://doi.org/10.1109/FCCM51124.2021.00024