D6.1

1st Evaluation Results

2022-09-06

Work package:	WP7	
Author(s):	Manuel Arenaz	Appentra
Reviewer #1	Aggelos Ioannou	EXAPSYS
Reviewer #2	Valeria Bartsch	Fraunhofer
Dissemination Level	Public	
Nature	Report	

Date	Author(s)	Comments	Version	Status
2022-07-29	Manuel Arenaz	Initial Draft	0.1	Draft
2022-08-01	Gino Perna	LBM-CFD Section	0.2	Draft
2022-08-04	Dimitris Theodoropoulos	OOPS Section	0.3	Draft
2022-08-08	Yannick Goumaz	Robotics section	0.4	Draft
2022-08-16	Giovanni Isotton	FEM section	0.5	Draft
2022-09-06	Almut Eisenträger	MESHFREE section	0.6	Draft
2022-09-13	Elisa Thiel	MESHFREE parts	0.7	Draft

EuroHPC Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 955739. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Greece, Germany, Italy, Netherlands,Spain,Switzerland.

Executive Summary

This document is the outcome of deliverable D6.1 of the OPTIMA project. In particular, this document provides a report on the 1st performance evaluation results and the next steps based on the initial evaluation. The code of the four applications of OPTIMA (Robotic Simulation, Finite Element Methods (FEM) for Underground Analysis, computational fluid dynamics (CFD) with MESHFREE and Lattice Boltzmann Method (LBM)-CFD) has been tested on different FPGA-based platforms and it was compared with the pure CPU version of the code. The hybrid CPU-FPGA version of FEM for Underground Analysis is based on the OPTIMA Open-Source library (OOPS), which allows an easy offloading of data processing tasks such as linear algebra kernels and sparse matrix vector multiplication on FPGAs.

In general the evaluation results show a speedup even for single compute units (CUs) for some parts of the applications compared to CPUs. Major findings are:

- Some applications need hybrid systems including FPGAs and GPUs in order to speedup the whole application. This is true for the Robotics simulations which rely on image rendering on GPUs while using FPGAs to speedup the inference of Convolutional Neural Networks (CNNs) in Deep Learning applications.
- When evaluating some applications (such as FEM methods for underground analysis and OOPS) the ExaNest platform was outperformed by Alveo U280 FPGAs. This is due to the fact that Alveo U280 incorporates HBM memory, has a broader memory data width and a slightly higher clock frequency compared to the ExaNest FPGAs.
- LBM-CFD evaluation shows that some part of the application shows only small speedups and even slow-downs when comparing the Jumax platform with a CPU version with 9 threads. The data transfer between CPUs and FPGAs has been identified as a bottleneck.
- OOPS sees significant overheads related to MPI communication when running on several FPGAs.

Next steps in order to improve the performance are to move from a single CU version to multiple CU versions and to mitigate performance bottlenecks identified in the first performance evaluation.

Table of Contents

1. Terminology/Glossary

CNN	Convolutional Neural Networks
CU	Compute Unit on a FPGA
HPC	High Performance Computing
FEM	Finite Element Method
FPGA	Field Programmable Gate Array
MPI	Message Passing Interface
PCG	Preconditioned Conjugate Gradient
OpenMP	Open Multi-Processing.
OOPS	OPTIMA OPen Source
LBM	Lattice-Boltzmann Method

2. Introduction

The OPTIMA project addresses the optimization of application codes to take advantage of a set of novel programming environments for FPGA-based HPC systems. The project considers several industrial applications from robotics, automotive and oil/gas, jointly with open-source libraries that provide a set of widely used kernels. In the scope of WP6, T6.1 focuses on testing and evaluating the performance of the 1st version of all the applications implemented in WP3. This deliverable (D6.1) presents the results of this 1st evaluation on the OPTIMA FPGA-based HPC systems. To this respect, sections 3 through 6 describe the results from the four applications, section 7 includes the results from the open source libraries, while section 8 includes the evaluation of the ExaNest platform.

3. First version of Robotic Simulation

3.1 Description

Cyberbotics has created an open source software called *Webots* that simulates robots of all types, such as drones, industrial robots or autonomous cars. Webots allows developers to create a wide variety of environments and to add robots on which all kinds of sensors can be installed, like cameras, distance sensors, etc. Robot controllers can be programmed in many languages like C, C++, Python, Java or even using the Robot Operating System (ROS).

The first version aims to accelerate a Webots robot simulation which uses deep learning in its controllers on a FPGA-based system and provides an initial performance comparison with a pure CPU execution.
First, we evaluate a multi-layer perceptrons (MLP) inference running on the Jumax CPU and on the Jumax DataFlow Engines (DFEs). DFEs are processing systems developed by Maxeler and allow accelerated execution of CPU programs. They are based on FPGAs and allow Dataflow Computing. This simple MLP classifies the images of the MNIST dataset. The goal of this step is to have a first global overview of the Jumax programming process and the estimated performance gains that can be achieved using FPGAs. No simulation is involved in this step.
Moreover, we moved on with extending the performance evaluation to a more concrete application involving a simulation on Webots. The neural network structure is extended to Convolutional Neural Networks (CNNs) in order to obtain a higher complexity and a more interesting and complete implementation on Jumax. The robot application deployed consists of a self-driving car using a front camera to follow a given track. Part of the simulation content, as well as the neural network structure, is taken from a reference paper [4]. The car controller code is optimized on CPU and FPGA to measure the impact on the simulation speed.

Additional details for the 1st version of the robotic simulation can be found in D3.1.

3.2 Implementation details

The hardware acceleration was performed progressively in two steps. The first one shows the achievable performance on the JUMAX HPC system for simple multilayer perceptrons. The second step is more concrete and complex, as it accelerates the controller of a simulated autonomous car implementing a convolutional neural network. Implementation details of the 1st version of the robotic simulation can be found in D3.1.

3.3 Performance evaluation

Using Jumax, we built an FPGA implementation of the CNN and compared it against the equivalent optimized CPU version. This is an important test case, using a real-world robotics simulation, and the FPGA execution of the neural network was measured to be more than five times faster than the CPU execution. This acceleration is possible because the FPGA chip is able to compute the output of the neural network from the input image much faster by parallelizing the computation of the convolutions per layer. The use of multithreading on the CPU version allows a fair comparison, but the resulting parallelization is not as efficient

as that of FPGAs, in particular because of the memory access overheads introduced by the use of multiple threads.

As deep learning becomes omnipresent in robotics applications, it is important that the execution time bottleneck introduced by these algorithms can be overcome to allow for very fast execution of simulations. What is more, this acceleration can also be used to improve the accuracy of the simulation. The time step of the simulation can be reduced by a factor of five and still allow a real time simulation speed. Thus, the neural network could send new commands to the car five times more often. The FPGA's high performance can also contribute to other robotics applications, such as those concerning localization, navigation, object recognition, pose estimation, grasping, legged locomotion, etc.

Regarding this evaluation, as part of our contribution, it is important to also highlight the importance of hybrid systems. Unfortunately, the lack of GPUs on the Jumax HPC system at the time of this first version slows down the overall simulation rendering considerably. The current result (speed multiplied by 5) concerns only the execution of the CNN and not the complete simulation. The addition of the GPU should allow us to later obtain the predicted results (see Figure 1).

Additional details on the performance evaluation of the robotic simulation can be found in D3.1.

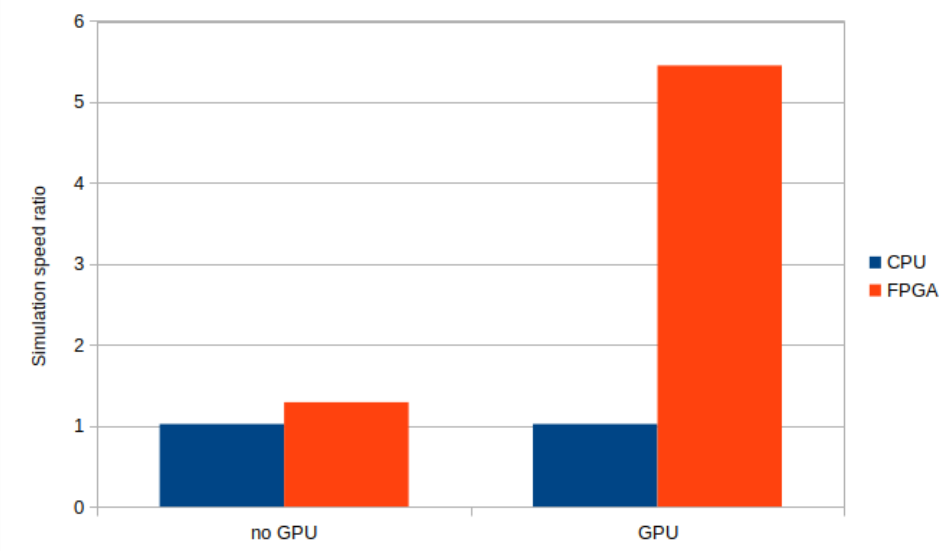


Figure 1: *Comparison of simulation speed ratios between CPU and FPGA with and without GPU*.

4. First version of Finite Element Methods for Underground Analysis

4.1 Description

M3E has developed proprietary state-of-the-art Finite Element (FE) software simulators for underground analysis, *ATLAS* and *GWS*, for geomechanics and groundwater, respectively [1]. Both ATLAS and GWS take full advantage of Chronos [2], a proprietary collection of sparse linear algebra kernels designed for HPC. The library implements best-in-class preconditioners to accelerate the convergence, and it can solve systems with hundreds of millions (or even billions) of unknowns.
Within the OPTIMA project, the main focus is on Chronos. Chronos is mainly written in C++ with a strongly object-oriented design. The Interprocessor communications are handled by CPUs through MPI directives while the fine grain parallelism is enhanced by multi-thread computation through OpenMP.
The main goal of the project is to develop a hybrid CPU-FPGA version of Chronos: the MPI communication between nodes will be preserved, and OpenCL APIs will be used for host/device communications, while for the innermost kernels, implemented in the FPGA fabric, the Optima Open Source (OOPS) library will be used.
In this first version, a hybrid CPU-FPGA version of the Preconditioner Conjugate Gradient (PCG) has been developed.

4.2 Implementation details

In the first version of Finite Element Method (FEM) for Underground Analysis we have focused on the shared memory version, while the MPI implementation will be addressed in the next developments, once the device (or accelerated) kernels are optimized. In particular, a hybrid CPU-FPGA version of the PCG has been deployed. The OpenCL APIs are used to allocate buffers on the device and make host-to-device and device-to-host copies. Regarding the kernel implementation on the device, the code relies on the OOPS library developed in the project by ICCS in D3.5.
Additional implementation details for the 1st version of the FEM for Underground Analysis can be found in D3.2.

4.3 Performance evaluation

Preliminary tests have been performed using a U280 FPGA board installed on the HACC cluster [3]. The current implementation is up to now slower than the corresponding CPU software implementation. This performance gap will be addressed in the next Work Package with support from other OPTIMA partners, namely ICCS and TSI. Additional performance details for the 1st version of the FEM for Underground Analysis can be found in D3.2.
The PCG has been implemented on four FPGAs: ZU9EG from ExaNeSt, ALveo U50, Alveo U280 and Alveo U55C. The figure below shows the numerical results in terms of execution times of 100 iterations of the PCG on a test case. Highlighted in blue is the cost of the Sparse Matrix Vector (SpMV) products. As expected, the execution time decreases significantly when multiple SpMV compute units (CUs) are utilized for the calculation. The performance of

the PCG is closely dependent on that of the SpMV products, being the most time-consuming operation in the whole iterative scheme. In terms of FPGA comparison, the results show that the embedded-based ZU9EG is significantly slower than the Alveo accelerator cards that are coupled with powerful CPUs.

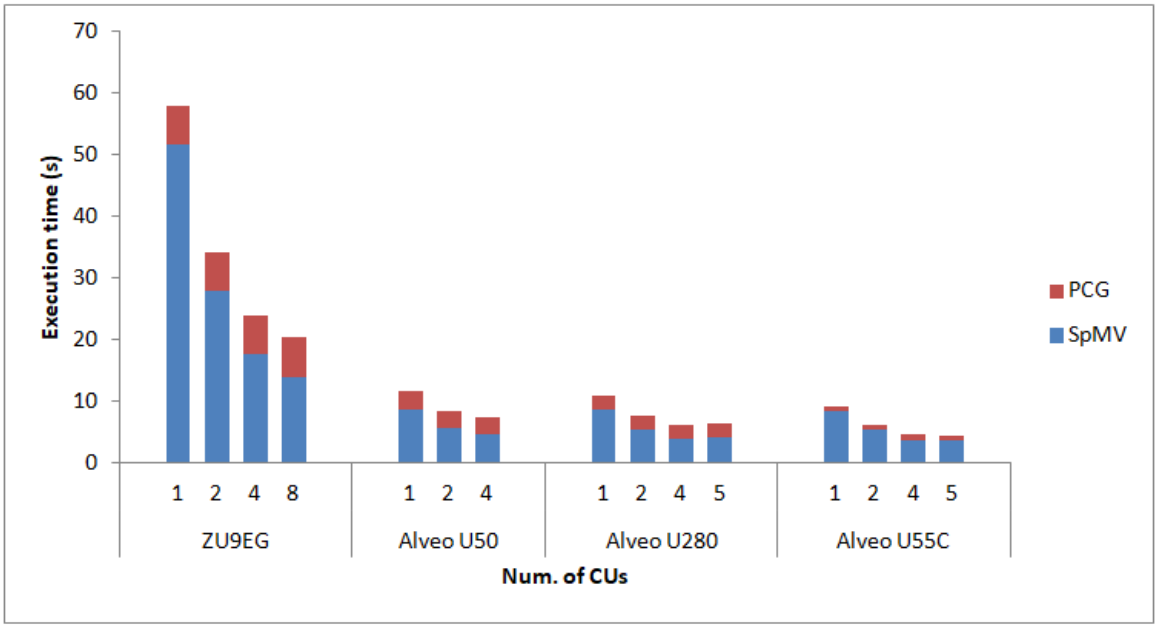


Figure 2: *Execution times of 100 iterations of PCG on different hardwares varying the number of CUs. The time of the PCG and the corresponding SPMV products are highlighted in red and blue, respectively.*

5. First version of MESHFREE

5.1 Description

The *MESHFREE* software is used for simulations of computational fluid mechanics, and for more general continuum mechanics problems in industrial contexts. Its underlying numeric methods are based on describing the computation domain as a distributed point cloud, which moves with the same speed as the simulated material. For stability of this method, it is crucial to keep the point cloud quality consistent throughout the simulation. Points are regularly removed and added where necessary, to guarantee point cloud quality and to allow for flexible refinement strategies. The point cloud organization tasks have to be run in every simulated time step and are potential hotspots in the code, depending on the precise setup being considered. Potentially millions of these points participate in the simulation and thus need to undergo organization tasks. Thispart of the code is the most suited for running on FPGAs. We ported parts of the point cloud organization to FPGA systems for accelerated computation. In this first step, the routines are run separate from the main MESHFREE code. The full interface will be considered in WP5.

5.2 Implementation details

Implementation details for the first version of MESHFREE can be found in D3.3.

5.3 Performance evaluation

Due to interfacing over multiple different high level languages – Fortran to C to maxj and back – the initial steps of the implementation were more complex and a lot of work for defining the interface was needed. This, unfortunately, led to delays in the work package and resulted in the first implementation being much smaller than intended. The implementation is currently running on the Maxeler simulation and development environment only and thus performance results in table 1 are preliminary.

Table 1: summary of running times for seek holes

Reference method	CPU [ms]	CPU+FPGA [ms]	Speedup
seekHoles	20-30	40-60	0.5

As a next step the FPGA code will be generated and the performance evaluation will be repeated on the hardware.

6. First version of LBM-CFD

6.1 Description

Enginsoft (ES) is using a modified version of *LBM-CFD* code in order to develop custom approaches taylored on vertical solutions for its business. The code is based on an open-source solver for LBM equations for Fluid-Dynamics. The code allows to very efficiently solve a wide variety of problems that ES normally parametrizes in order to let users perform sensitivity and optimization design and take advantage of HPC systems. The LBM approach permits to efficiently solve problems without having to deal with differential equations, allowing to speed up the solution when this method is applicable. The core of the solver is coded in C with the preprocessor coded in C,C++ and Python.

In the LBM approach, both space and time are discretized so that only certain discrete velocity vectors are allowed. For this first version, we have used the so-called D2Q9 lattice, in which there are two dimensions and just nine allowed displacement vectors. One of the allowed displacements is zero, while the other eight take a molecule from its current site into one of the eight nearest sites in the square lattice, either horizontally, vertically, or at a 45-degree diagonal.

The initial work within the project has been to understand the timing of the mathematical kernel in order to initially focus on the most important parts.
The kernel has two important stages where calculations are involved (described here briefly in macro-areas):

- Collisions. For each lattice, do the following:
    - From the nine microscopic densities n_i, compute the macroscopic density ρ and velocity components u_x and u_y.

- From these three macroscopic variables, use equation

$$D(\vec{v}) \longrightarrow w_i \left[1 + \frac{3\,\vec{e}_i \cdot \vec{u}}{c} + \frac{9}{2}\left(\frac{\vec{e}_i \cdot \vec{u}}{c}\right)^2 - \frac{3}{2}\frac{|\vec{u}|^2}{c^2}\right].$$

  to compute the equilibrium number densities n_i^{eq}
- Update each of the nine number densities according to the previous step until the difference between steps becomes negligible

- Streaming. Move all the moving molecules into adjacent or diagonal lattice sites by copying the appropriate n_i values.

The problem is generally CPU, i.e. computationally, bound. Simulations are also done in the time domain, producing a time history of the flow velocities, temperatures and mass flows.

6.2 Implementation details

In this section we describe the main work for implementing a hybrid CPU-FPGA version of the solver mainly based on an SMP (openMP) code.

By instrumenting the kernel with run-time performance collecting libraries and running a number of different simulations using D2Q9 lattice, the following timing schedule has been extracted:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.39     11.51    11.51 1168582500     0.00     0.00  computeEquilibrium
 31.68     22.43    10.92    10000     1.09     1.09  propagate
 23.09     30.39     7.96 123870000     0.00     0.00  bgk
  6.09     32.49     2.10 126067500     0.00     0.00  computeMacros
  1.96     33.16     0.68  5960000     0.00     0.00  splitEqNeq
  1.76     33.77     0.60  5960000     0.00     0.00  regularizedF
```

Figure 3: *timing framework results of a sample CFD problem*

Double and single precision versions show similar results in percentage. By analyzing the number of calls per simulation, as seen in Figure 3, we decided that ***ComputeEquilibrium***, ***bgk[1] and computeMacros*** methods should be the computational tasks to focus on.
In order to better optimize the procedure it has been decided to buffer all the calculations to be performed on each cell (bgk & computeEquilibrium) in a vector and then perform one single call to the FPGA accelerated code in order to process the entire domain.

BGK

BGK is a term to be calculated for each cell in the system (as listed in the following picture), which in turn calls Computeqeuilibrim (see next picture):

[1] Bhatnagar Gross and Krook (BGK) model for relaxation to equilibrium. Bhatnagar, P. L.; Gross, E. P.; Krook, M. (1954-05-01). "A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems". *Physical Review*. **94** (3): 511–525. Bibcode:1954PhRv...94..511B. doi:10.1103/PhysRev.94.511. ISSN 0031-899X.

```
232  //bgk collision term
233  void bgk(double* fPop, void* selfData) {
234      double omega = *((double*)selfData);
235      double rho, ux, uy;
236      computeMacros(fPop, &rho, &ux, &uy);
237      double uSqr = ux*ux+uy*uy;
238      int iPop;
239      for(iPop=0; iPop<9; ++iPop) {
240          fPop[iPop] *= (1-omega);
241          fPop[iPop] += omega * computeEquilibrium (
242                                  iPop, rho, ux, uy, uSqr );
243      }
244  }
```

Figure 4: *bgk Method serial version*

computeEquilibrim

The computeEquilibrium method computes the local equilibrium in a cell along the designed direction:

```
221
222  //compute local equilibrium from rho and u
223  double computeEquilibrium(int iPop, double rho,
224                            double ux, double uy, double uSqr)
225  {
226      double c_u = c[iPop][0]*ux + c[iPop][1]*uy;
227      return rho * t[iPop] * (
228                  1. + 3.*c_u + 4.5*c_u*c_u - 1.5*uSqr
229              );
230  }
```

Figure 5: *computeEquilibrium Method, original CPU version*

It has been decided to merge the two parts and write them in the corresponding maxj Java language to be then processed by the Maxeler environment.
The corresponding implementation has been developed into the following piece of code (Maxeler Environment) that corresponds to the previous two methods (Java code to be then translated on FPGA by the Maxeller compiler):

```
1  package bgkdfe;
2
3  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
4  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
5  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
6  import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
7  import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;
8
9  class bgkDFEKernel extends Kernel {
10         bgkDFEKernel(KernelParameters parameters) {
11
12     super(parameters);
13
14     DFEVectorType<DFEVar> nodeType = new DFEVectorType<DFEVar>(dfeFloat(11, 53), 10);
15     DFEVectorType<DFEVar> procnodeType = new DFEVectorType<DFEVar>(dfeFloat(11, 53), 9);
16
17     // Input
18     DFEVector<DFEVar> inVector = io.input("inVector", nodeType);
19
20
21         DFEVar omega = inVector[9];
22
23         //computeMacros
24
25         DFEVar upperLine = inVector[2] + inVector[5] + inVector[6];
26         DFEVar mediumLine = inVector[0] + inVector[1] + inVector[3];
27         DFEVar lowerLine = inVector[4] + inVector[7] + inVector[8];
28         DFEVar rho = upperLine + mediumLine + lowerLine;
29         DFEVar ux = (inVector[1] + inVector[5] + inVector[8] - inVector[3] - inVector[6] - inVector[7]) / rho;
30         DFEVar uy = (upperLine - lowerLine) / rho;
31
32         DFEVar uSqr = ux*ux + uy*uy;
33
34         DFEVector<DFEVar> r1 = procnodeType.newInstance(this);
35         DFEVector<DFEVar> r2 = procnodeType.newInstance(this);
36         DFEVector<DFEVar> result = nodeType.newInstance(this);
37         DFEVector<DFEVar> c_u = procnodeType.newInstance(this);
38
39         int c[][] = { {0,0}, {1,0}, {0,1}, {-1,0}, {0,-1},
40               {1,1}, {-1,1}, {-1,-1}, {1,-1} };
41     double t[] = { 4./9., 1./9., 1./9., 1./9., 1./9., 1./36., 1./36., 1./36., 1./36. };
42
43         //bgk

49         for(int iPop = 0; iPop < 9; ++iPop){
50                r1[iPop] <== inVector[iPop] * ( 1 - omega );
51                //debug.simPrintf("%f\t", r1[iPop]);
52                c_u[iPop] <== c[iPop][0] * ux + c[iPop][1] * uy;
53                r2[iPop] <== omega * rho * t[iPop] * (1. + 3. * c_u[iPop] + 9./2. * c_u[iPop] * c_u[iPop] - 3./2. * uSqr);
54                //debug.simPrintf("Rho: %f\t t: %f\t c_u: %f\t uSqr: %f\t", rho, t[iPop], c_u[iPop], uSqr);
55                result[iPop] <== (r1[iPop] + r2[iPop]);
56                //debug.simPrintf("%f\t", result[iPop]);
57         }
58         //debug.simPrintf("\n");
59         result[9] <== omega;
60
```

Figure 6: BGK & *computeEquilibrium Method rearranged for FPGA (maxj code)*[2]

[2] In particular lines 52 & 53 correspond to the method in Figure 5, while the remaining part (49-58) refers to the method in Figure 4. The rest is initialization.

The main attention has been paid particularly on trying to buffer the coefficients in order to maximize the amount of calculations done in one call to the FPGA. This has been the real problem due to a complete refactoring of the code into the following:

```
117     // apply collision step to all lattice nodes
118   void collide(Simulation* sim) {
119       int iX, iY, iPop, count;
120       count = 0;
121       for (iX=1; iX<=sim->lx; ++iX) {
122           for (iY=1; iY<=sim->ly; ++iY) {
123               if(sim->lattice[iX][iY].isbgk){
124                   for(iPop=0; iPop<9; ++iPop){
125                       sim->indfe[10*count+iPop] = sim->lattice[iX][iY].fPop[iPop];
126                   }
127                   sim->indfe[10*count+iPop] = *((double*) sim->lattice[iX][iY].dynamics->selfData);
128                   ++count;
129               }
130               else {
131                   collideNode(&(sim->lattice[iX][iY]));
132               }
133           }
134       }
135
136       bgkDFE(count, sim->indfe, count*10*sizeof(double), sim->outdfe, count*10*sizeof(double));          //Sli-C function to compute on dfe
137
138       count = 0;
139       for (iX=1; iX<=sim->lx; ++iX) {
140           for (iY=1; iY<=sim->ly; ++iY) {
141               if(sim->lattice[iX][iY].isbgk){
142                   for(iPop=0; iPop<9; ++iPop){
143                       sim->lattice[iX][iY].fPop[iPop] = sim->outdfe[10*count+iPop];
144                   }
145                   ++count;
146                 }
147             }
148         }
149   }
150
```

Figure 7: Collide changes (buffered memory) to implement just one single call to FPGA bgk for the entire domain (bgkDFE)

It can be seen that the code is divided into three main blocks: the buffering with initialization into a vector (10 times lattice count) (lines 118-134) that is then processed at once (line 136) and then de-serialized into the original matrix (lines 138-149).

The buffering and de-serializing parts of the collide method have become the most time-consuming part of the code (see next section), leaving the bgk accelerated method, for the overall domain, with a very tiny percentage of the total execution time.
In the 2nd part of the project we will focus on parallelizing this part in order to take advantage of multiple FPGAs and MPI threads.

6.3 Performance evaluation

In this phase tests have been performed on the JUMAX platform at Juelich by running the single thread application on the host along with FPGAs, and comparing it with the openMP CPU version with 9 threads (nine is due to D2Q9 lattice pattern used).
The problem solved was a reference problem from our database, while similar results have been achieved with other examples.

Results are summarized in the following table:

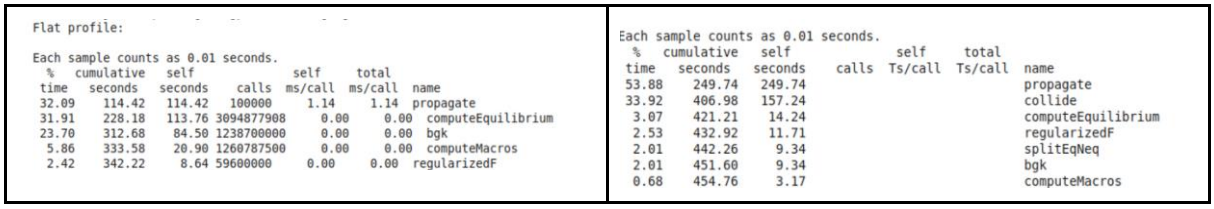
Table 2: summary of running times for 1000 time steps

Reference method	CPU [ms]	CPU+FPGA [ms]	Speedup
computeEquilibrium	11510	7947	1.44
Bgk (serial)	7960	1673	4.75
Bgk (openMP)	1245	1673	0.744

The table summarizes, in this first version, timing ratios achieved with the implementation. It should be noted that:

- The run for BGK method on FPGA was optimized for the specific lattice (it was the same for serial and openMP part on the FPGA) which shows performance bottlenecks in the memory transfer path.
- BGK has been constrained to the particular 9 thread implementation (in the openMP test) in order to compare it with the FPGA counterpart. The number 9 comes from the specific pattern D2Q9 used in the tests, allowing the parallel tasks to be compared.

The timing difference can be summarized in the following picture (100k time steps) showing effects on BGK & computeEquilibrium FPGA acceleration and buffering bottleneck on collide method:



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 32.09    114.42   114.42   100000     1.14     1.14  propagate
 31.91    228.18   113.76 3094877908     0.00     0.00  computeEquilibrium
 23.70    312.68    84.50 1238700000     0.00     0.00  bgk
  5.86    333.58    20.90 1260787500     0.00     0.00  computeMacros
  2.42    342.22     8.64 59600000     0.00     0.00  regularizedF
```

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 53.88    249.74   249.74                             propagate
 33.92    406.98   157.24                             collide
  3.07    421.21    14.24                             computeEquilibrium
  2.53    432.92    11.71                             regularizedF
  2.01    442.26     9.34                             splitEqNeq
  2.01    451.60     9.34                             bgk
  0.68    454.76     3.17                             computeMacros
```

Figure 8: *Timing difference: left original code, right Jumax version*

There is a huge performance improvement to the methods implemented on FPGA with a similar performance bottleneck in collide mainly due to memory transfer of data buffers back and forth from/to the FPGA. All tests are performed in DP arithmetic.
The problem is time dependent, and the propagate method of the code (another 30% of runtime in CPU version) is relying on this. New optimizations should be done by researching the buffering type to optimize communications with the FPGAs and the possibility of parallelizing stripes of lattices.

7. First version of Open Source Libraries (OOPS)

7.1 Description

Table 3 - Kernels supported by the OOPS library.

BLAS	L1	ROT, ROTM, SWAP, SCAL, COPY, AXPY, DOT, DSDOT, NRM2, ASUM, AMIN, AMAX
	L2	GEMV, GBMV, SYMV, SBMV, SPMV, TRMV, TBMV, TPMV, TRSV, TBSV, TPSV

	L3	GEMM, SYMM, TRMM, TRSM
Sparse Linear Algebra	SpMV	
General Computer-Aided Engineering (CAE) solvers	PETSc	Preconditioners: Jacobi Direct solvers: LU factorization / decomposition Krylov: CG

The *OOPS library* set exposes a set of high-level function prototypes for accelerating applications executed on the host processor. These function prototypes allow developers to easily offload data processing for common tasks, such as linear algebra and sparse matrix-vector multiplication, onto reconfigurable hardware. Table 3 lists all kernels that will be supported by the OOPS library set.

Section 7.2 provides an overview of how all kernels are implemented on the FPGA, whereas section 7.3 discusses performance results. It should be noted that D3.5 provides an elaborated description of each OOPS kernel implementation, as well as its performance when compared to other available software versions.

7.2 Implementation details

As described in D3.5, communication between the host processor and each OOPS kernel is done as follows:

1. On the host side, each kernel uses the Xilinx Runtime (XRT) system to allocate memory space on the FPGA card, as well as copying input data to the allocated memory space on the FPGA side.
2. On the FPGA side, each kernel IP leverages (when possible) burst memory access for efficient data transfer between the host and FPGA memory. Internally, each kernel employs dedicated HLS directives that enable hardware optimizations during data processing (e.g. fully pipelined hardware, loop-unrolling, etc), as well as overlapping between processing and writing results back to memory.
3. Once finished, on the host side, the software on the host side copies back results to the host memory.

```
float asum(hls::stream< v_dt>& Xin, int N) {
  unsigned int vSize = ((N - 1) / VDATA_SIZE) + 1;
  float result[VDATA_SIZE];
  float final_result=0;
  #pragma HLS ARRAY_PARTITION result dim=1 complete
  v_dt temp;
  int count =0;
  init_asum:
  for(int i=0;i<VDATA_SIZE;i++){
    #pragma HLS unroll
    result[i]=0;
```

```
  }
  excecute_asum:
  for (int i = 1; i < vSize; i++) {
    #pragma HLS pipeline II=1
    temp=Xin.read();
    for(int j=0;j<VDATA_SIZE;j++){
      #pragma HLS unroll
      count++;
      if (count>N)
        temp_result[j] +=0;
      else
        result[j] +=abs_float(temp.data[j]);
    }
  }
  excecute_final_of_sum:
  for (int i=0;i<VDATA_SIZE;i++){
    #pragma HLS pipeline II=1
    final_result+=result[i];
  }
  return final_result;
}
```

The OOPS library applies dedicated optimization techniques for faster data access from the memory, and parallel processing. For example, in all BLAS L1 kernels, the implementation is based on different functions for memory access and data processing that overlap, using the DATAFLOW directive. Moreover, code iterations with a for-loop have been annotated as fully pipelined structures that can be unrolled, using the PIPELINE, ARRAY_PARTITION and UNROLL directives respectively. The code snippet above, provides an example of how the asum BLAS L1 kernel (returns the sum of the absolute values in a vector) data processing function is implemented.

```
loop_over_cols:
   for (int j=0;j<N;j+=VDATA_SIZE) {
          #pragma HLS dataflow
          wide_read_x(Xup1, Xup2, Xlow1, Xlow2, X, j/VDATA_SIZE);

          read_y(Yup1_in,Yupper1,M/4);
          read_y(Yup2_in,Yupper2,M/4);
          read_y(Ylow1_in,Ylower1,M/4);
          read_y(Ylow2_in,Ylower2,M/4);

          wide_read_matrix( Aupper1, Aup1, M/4, N/VDATA_SIZE, j/VDATA_SIZE);
          wide_read_matrix( Aupper2, Aup2, M/4, N/VDATA_SIZE, j/VDATA_SIZE);
          wide_read_matrix( Alower1, Alow1, M/4, N/VDATA_SIZE, j/VDATA_SIZE);
          wide_read_matrix( Alower2, Alow2, M/4, N/VDATA_SIZE, j/VDATA_SIZE);

          gemv(Aup1,Yup1_temp,Xup1,alpha,M/4);
          gemv(Aup2,Yup2_temp,Xup2,alpha,M/4);
          gemv(Alow1,Ylow1_temp,Xlow1,alpha,M/4);
```

```
        gemv(Alow2,Ylow2_temp,Xlow2,alpha,M/4);

        accum(Yup1_in,Yup1_temp,Yup1_out,beta,j/VDATA_SIZE,M/4);
        accum(Yup2_in,Yup2_temp,Yup2_out,beta,j/VDATA_SIZE,M/4);
        accum(Ylow1_in,Ylow1_temp,Ylow1_out,beta,j/VDATA_SIZE,M/4);
        accum(Ylow2_in,Ylow2_temp,Ylow2_out,beta,j/VDATA_SIZE,M/4);

        write_y(Yup1_out,Yupper1,M/4);
        write_y(Yup2_out,Yupper2,M/4);
        write_y(Ylow1_out,Ylower1,M/4);
        write_y(Ylow2_out,Ylower2,M/4);
}
```

As expected, higher level functions, like the BLAS L2 and L3 routines are implemented by reusing the ones from L1. For example, the code snippet above provides the L2 gemv HLS code (calculates a scalar-matrix-vector product, and then adds the result to a scalar-vector product), which adopts a column-based approach. An outer for-loop with step VDATA_SIZE (vector size) forms a hardware pipeline that fetches the two vectors first (wide_read_x, read_y), reads the matrix (wide_read_matrix), calculates the scalar-matrix-vector result, accumulates it to the product of the second vector-scalar (accum), and finally writes the output to the memory (write_y). Further details on the implementation code for each kernel can be found in D3.5.

7.3 Performance evaluation

Performance evaluation of the first version of the OOPS library was done using non-optimized code, as well as the OpenBLAS [5], a fine-tuned software library for algebraic calculations. This section provides a subset of our evaluation results; full performance evaluation is available in D3.5.

Overall, the first version of the OOPS library focused on the development of finely-tuned compute units (CUs) for each kernel, where each CU occupies only a fraction of the available logic (less than 5%) and memory bandwidth. Nevertheless, as shown in Section 8.2 and D3.5, in many cases, results show that even single-CU kernels can outperform software-optimized routines, like the OpenBLAS.

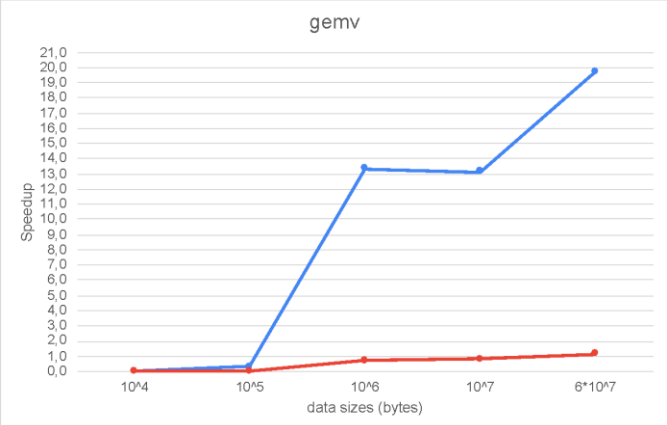


Figure 9: *GEMV performance against non-optimized code (blue) and OpenBLAS (red).*

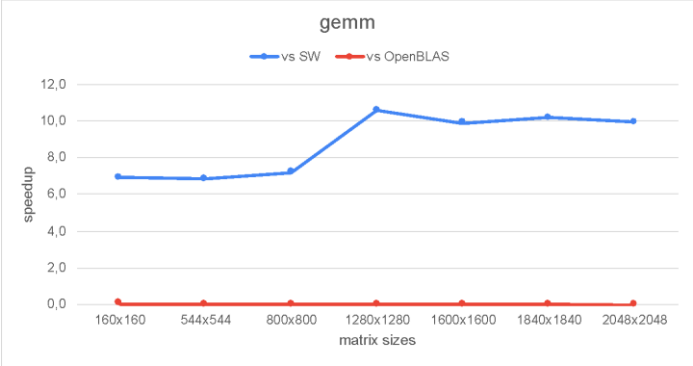


Figure 10: *GEMM performance against non-optimized code (blue) and OpenBLAS (red).*

Table 4 - Resource utilization of the gemv and gemm kernels

Kernel	Registers (%)	LUTs (%)	BRAMs (%)	DSPs (%)
gemv	4.73	4.06	2.83	5.81
gemm	1.02	1.17	1.09	0.76

There are cases though, where the current single-CU implementation of an OOPS kernel processes data slower than its counterpart OpenBLAS routine. For example, Figures 8 and 9 compare the gemv and gemm kernels respectively, against non-optimized software (almost 20x and 11x faster respectively) and OpenBLAS (in gemv performance is approximately 15% faster for large matrix sizes, whereas in gemm OpenBLAS is faster). However, as shown in Table 4 and described in D3.5, a single CU of both kernels occupies very few resources, leaving room for multi-CU implementations that will fully utilize the available HBM bandwidth. This approach is expected to drastically improve performance that will enable faster data processing than OpenBLAS, as soon as the final version of the OOPS library is ready.

8. ExaNeSt platform evaluation

8.1 Description

In this section we provide the initial evaluation results from the implementation of different kernels, derived from the OOPS library, on the ExaNeSt platform. The testbed platform involves a single baseboard that incorporates four custom-built Quad-FPGA Daughter Boards (QFDBs) with four Zynq Ultrascale+ MPSoCs each, designated as F1, F2, F3 and F4. As a result, there are 16 available FPGAs on the baseboard. We do not utilize the F1 FPGA of each QFDB for kernel acceleration, due to issues with the complexity of the platform design as well as the firmware. The F1 FPGA is the network FPGA and incorporates all the required logic to enable the other 3 FPGAs to communicate with the outside world (e.g. other QFDBs). There is work in progress to allow the F1 FPGAs to also be able to incorporate accelerating kernels. As a result, the number of available FPGAs for acceleration is reduced to 12.

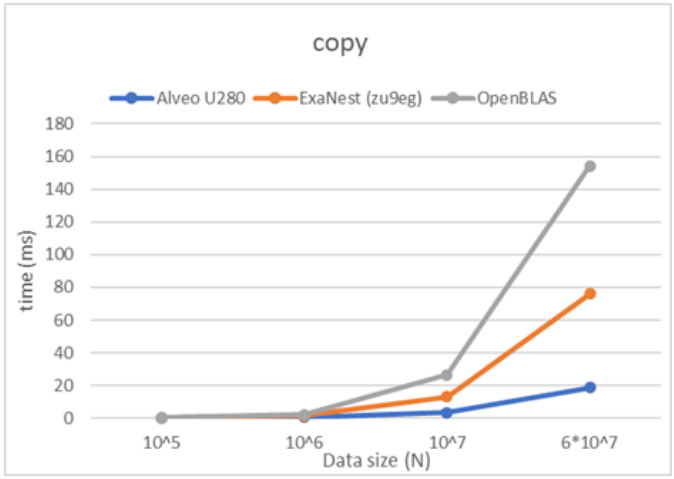
8.2 ExaNeSt platform FPGA evaluation

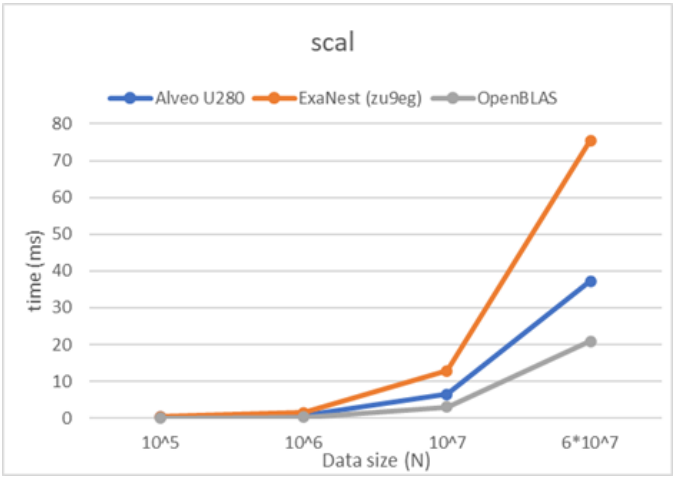
Initially we evaluated the performance of a single MPSoC by utilizing the kernels of the OOPS library. The results of the software (CPU) execution, as well as the execution on the Alveo U280 accelerator card, where the initial implementation of the kernels was conducted, are derived from Deliverable 3.5. For the comparison with software, the OpenBLAS library was chosen, as it is highly optimized. We have implemented the L1 routines of the BLAS library on the ExaNeSt platform, which required some small modifications to the OOPS library, mainly concerning the function interfaces. Table 5 lists the functions already implemented. There is work in progress on the implementation of the L2 routines (already implemented on Alveo U280) on the ExaNeSt FPGAs

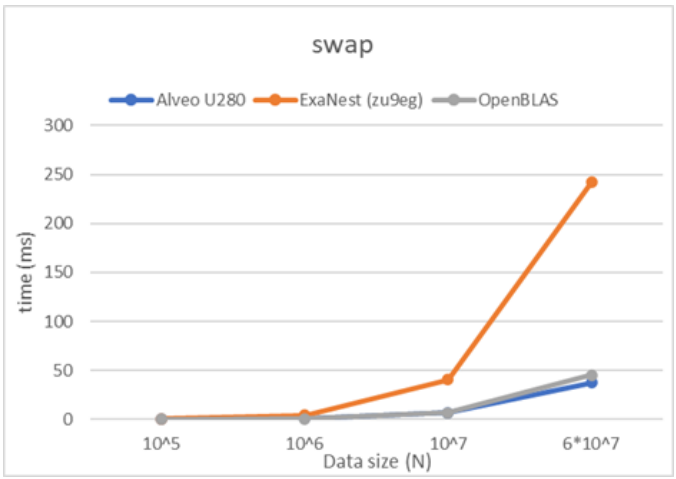
Table 5 - Kernels of the OOPS library implemented on the ExaNeSt platform FPGAs

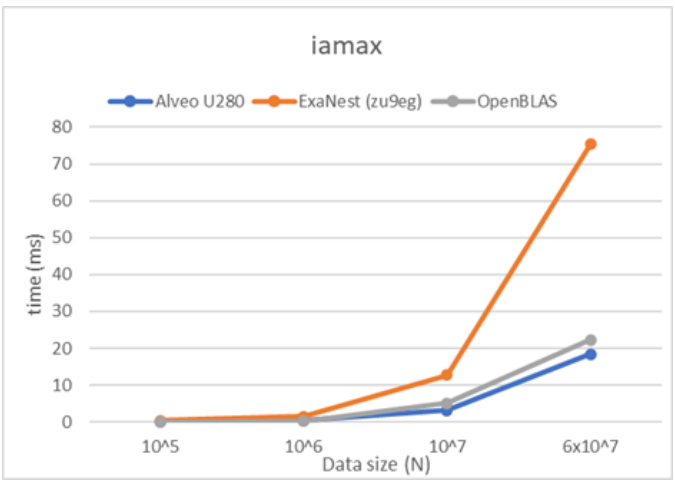
BLAS	L1	COPY, SCAL, SWAP, IAMAX, ASUM, NRM2, DOT, DDOT, AXPY, XPAY, ROT, ROTM

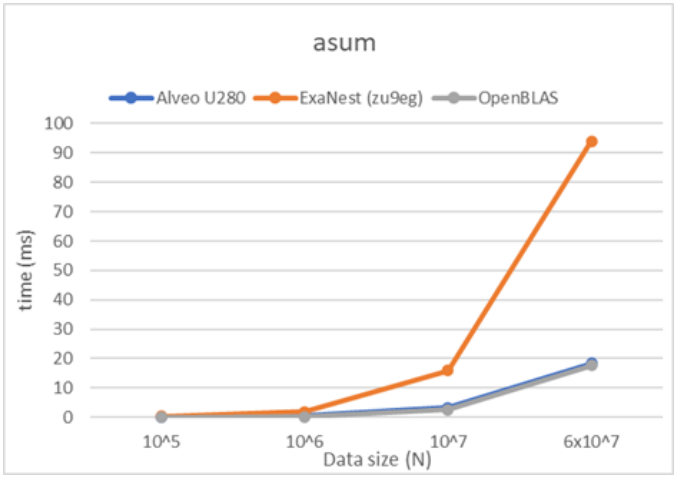
In the following Figures the results presented, concern the execution on a single FPGA of the ExaNeSt platform, on the Alveo U280 card and the single threaded OpenBLAS.

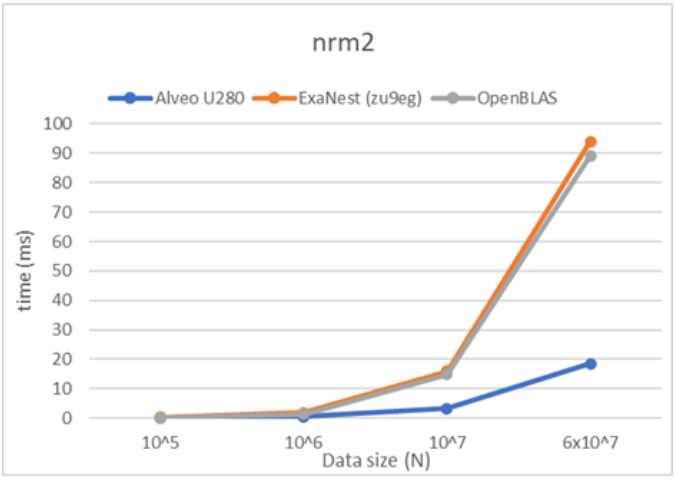

copy
Alveo U280
ExaNest (zu9eg)
OpenBLAS
time (ms)
180
160
140
120
100
80
60
40
20
0
10^5
10^6
10^7
6*10^7
Data size (N)


scal
Alveo U280
ExaNest (zu9eg)
OpenBLAS
time (ms)
80
70
60
50
40
30
20
10
0
10^5
10^6
10^7
6*10^7
Data size (N)


swap
Alveo U280
ExaNest (zu9eg)
OpenBLAS
time (ms)
300
250
200
150
100
50
0
10^5
10^6
10^7
6*10^7
Data size (N)


iamax
Alveo U280
ExaNest (zu9eg)
OpenBLAS
time (ms)
80
70
60
50
40
30
20
10
0
10^5
10^6
10^7
6x10^7
Data size (N)


asum
Alveo U280
ExaNest (zu9eg)
OpenBLAS
time (ms)
100
90
80
70
60
50
40
30
20
10
0
10^5
10^6
10^7
6x10^7
Data size (N)


nrm2
Alveo U280
ExaNest (zu9eg)
OpenBLAS
time (ms)
100
90
80
70
60
50
40
30
20
10
0
10^5
10^6
10^7
6x10^7
Data size (N)

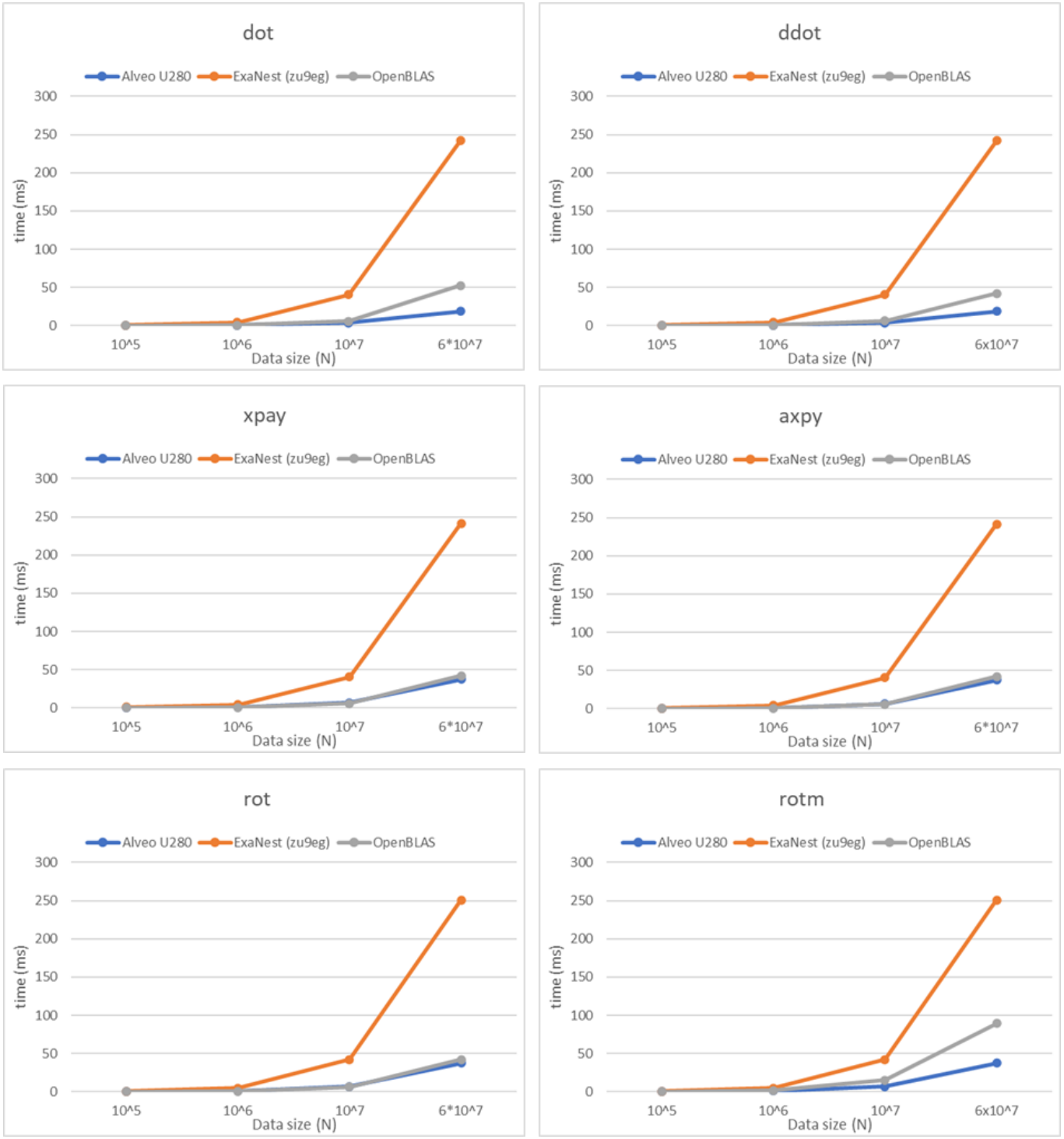


Figure 11 – Execution time comparison of the ExaNeSt FPGA, the Alveo U280 and the single thread OpenBLAS running on software.

The current results show that the execution of a single kernel on the ExaNeSt MPSoC is up to 5 times slower than the Alveo U280 and the software. This is mainly affected by three factors. (a)The Alveo U280 incorporates HBM memory that offers significantly higher bandwidth than the DDR of the ExaNeSt. (b)The memory data width of the Alveo card is 512 bits, while for the ExaNeSt FPGA it is 128 bits. (c) The kernels that are implemented on the

Alveo card operate at a higher clock frequency (300MHz for the Alveo vs. 200MHz for the ExaNeSt FPGAs). Also, another minor cause of slowdown is related to the weak ARM Cortex-A53 host processor of the ExaNeSt MPSoC. On the other hand, the execution of a single kernel on the Alveo U280 accelerator card is slightly faster than the single threaded software, which is very promising, as each of these kernels requires very limited resources, and multiple compute units can be placed within the FPGA to increase performance.

8.3 MPI evaluation

Communication between the nodes constitutes another major task that increases execution time of an application executed on the ExaNeSt platform. The MPI framework[3] is utilized for the distribution of data and computation to the nodes of the ExaNeSt platform. In order to evaluate the communication overheads, we implemented a simple vector addition kernel on the FPGA. The master thread of the software distributes the data using non-blocking I/O and gathers the results using blocking I/O. The total execution time of the experiments conducted on a single baseboard of the ExaNeSt platform are shown in Table 6. The execution time includes the communication overheads of the MPI, the host/FPGA communication as well as the computation. We need to note that the simple vector addition kernel was chosen in order to mainly evaluate the MPI communication overhead.

Table 6 – Total execution time of distributed execution on the ExaNeSt platform using MPI.

SSH | RSH (seconds)	**Number of elements (in millions)**															
Task allocation (divisor)	**1**		**2**		**4**		**8**		**16**		**32**		**64**		**128**	
1 (single FPGA)	0.67	0.45	0.67	0.47	0.74	0.54	0.89	0.71	1.21	1.01	1.86	1.62	3.04	2.84	N/A	N/A
3 (one task per FPGA of the same QFDB)	0.83	0.57	0.81	0.59	0.88	0.65	1.07	0.85	1.37	1.17	2.17	1.83	3.66	3.19	6.87	6.01
4 (one task per QFDB)	0.83	0.57	0.77	0.57	0.91	0.64	1.21	0.91	1.37	1.24	2.11	2.01	3.24	2.93	6.59	5.58
10 (one task per FPGA per QFDB)	0.93	0.59	0.94	0.61	1.01	0.67	1.14	0.98	1.41	1.27	2.13	1.99	3.04	2.71	5.53	5.96

We utilized both SSH (blue) and RSH (green) protocols in our experiments presented in Table 5. The main reason is that the SSH protocol performance is lower, as shown in the results, for three main reasons: (a) it requires handshaking between the nodes, which is computationally expensive, but is executed only once in every session, (b) the data traffic is encrypted, which requires processing power for encryption and decryption, (c) there is a slight increase in network traffic. These operations are executed by the ARM Cortex-A53 processor, which is an embedded processor with low processing capabilities. On the other hand, the simpler RSH protocol does not use encryption which is not safe on a public network. As the ExaNeSt platform operates on a private network, the RSH protocol could be utilized in order to speed up the communication process.

[3] In the present evaluation MPI is acting over TCP

The execution time results show significant overheads related to communication. Especially for small data sizes, there is a significant increase in execution time as the number of MPI ranks (threads) increase. In the largest data sizes, a very slight decrease in the execution time is observed, as the number of threads increases. These results show that a significant amount of calculation is required in order to hide the MPI communication overheads.

9. Conclusions

Deliverable D6.1 presents the results of the 1st evaluation phase of the OPTIMA application codes on the OPTIMA FPGA-based HPC systems, providing the initial validation and verification of the results, as well as targeted suggestions for the improvement of the applications during the second half of the project.

Webots has ported CNNs for inference on the Jumax platform and compared it to an optizimed CPU version. This part of the code runs 5 times faster on Jumax than on a CPU. However in order to speedup the whole application a hybrid cluster with GPUs for image rendering.

FEM Methods for Underground Analysis have identified their linear algebra kernel based on Chronos as a performance bottleneck of their application. A hybrid CPU-FPGA version has been developed based on OOPS. Comparing different FPGAs, Alveo U55C has been identified to give the highest speedups compared to Alveo U280, Alveo U50 and ZU9EG.

Meshfree has ported a distance calculation kernel for seeking holes in the pointcloud to the Maxeler simulation and development platform. The data transfer between CPUs and FPGAs is identified as the main bottleneck of the current implementation.

LBM-CFD has ported kernels on Jumax and compared them against a CPU version with 9 OpenMP threads. Speedups in some parts of the code are counterbalanced by slowdowns in other parts. The data transfer between CPUs and FPGAs has been identified as the main bottleneck.

Regarding the evaluation of the ExaNeSt platform, the results from the execution of different kernels with the **OOPS library** show that the FPGAs that are the building blocks of the ExaNeSt platform are underperforming compared to newer FPGAs that incorporate HBM memory. In fact, a single Alveo U280 is able to execute the same work five times faster than a single Zynq Ultrascale+ MPSoC of the ExaNeSt platform. Also, the experiments with MPI (on ExaNeSt) show significant communication overheads. The applications targeting the ExaNeSt platform are required to provide significant computation tasks to each rank in order to hide these communication overheads.

10. References

[1] Janna C., Isotton G., Frigo M., Spiezia N. and Tosatto O., ATLAS Web page. https://www.m3eweb.it/atlas, 2020.

[2] Janna C., Isotton G. and Frigo M., Chronos Web page. https://www.m3eweb.it/chronos, 2020.

[3] HACC Web Page. https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html

[4] Forson E., Towards Data Science. Teaching Cars To Drive Using Deep Learning — Steering Angle Prediction. Sep 16, 1997. https://towardsdatascience.com/teaching-cars-to-drive-using-deep-learning-steering-angle-prediction-5773154608f2

[5] http://www.openblas.net/