Deliverable D3.2

1st Version of Finite Element Methods
2022-06-30

Work package:	WP3 Development of 1st Version of Applications	
Author(s):	Giovanni Isotton	M3E
Reviewer #1	ES	Gino Perna
Reviewer #2	CYB	
Dissemination Level	Public	
Nature	Report	

Confidentiality
This document may contain proprietary material of certain OPTIMA contractors. The commercial use of any information contained in this document may require a license from the proprietor of that information.

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

The deliverable provides details of the first accelerated version of the M3E application. For all the kernels that were accelerated, the first part describes the algorithms while in the second part the focus is on interfacing the OOPS library to the application. At this stage the focus is on a shared-memory implementation. The deliverables closes with a comparison between the CPU software implementation and the accelerated version on the XACC machine.

Table of Contents

List of Abbreviations

HPC	High Performance Computing
FPGA	Field Programmable Gate Array
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
MPI	Message Passing Interface
PCG	Preconditioned Conjugate Gradient
OpenMP	Open Multi-Processing.
OOPS	OPTIMA OPen Source
BLAS	Basic Linear Algebra Subprograms

1. Introduction

M3E has developed proprietary state-of-the-art Finite Element (FE) software simulators for underground analysis [1]: ATLAS and GWS, for geomechanics and groundwater, respectively.

When it comes to the solution of extreme-size FE models, it is well-known that, for large models, the most time-consuming task (up to 95% of the total computational time) consists of solving the arising linear systems of equations. 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 focus is laid 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.

Within the OPTIMA project, we will develop the CPU-version of Chronos in order to create a CPU-FPGA version. In particular, the MPI communication between nodes will be preserved, using OpenCL APIs for host/device communications. For the innermost kernels, OOPS library will be used [ref to D3.5].

2. Definition of tasks for hardware acceleration

Within the OPTIMA project, M3E main goal is the acceleration of the solution of the linear system of equations:

$$Ax = b$$

where A is a symmetric positive definite matrix, x is the unknown solution and b is a given right-hand side.

To solve this system, both ATLAS and GWS use an iterative method that constructs, from an arbitrary initial solution x_0, a succession of vectors (x_1, x_2, ..., x_k) converging to the exact solution x. In particular, the Preconditioner Conjugate Gradient (PCG) is used.

In this work package we focused on the PCG preconditioned with Jacobi that is based on the following kernels:

- *SpMV* : Sparse Matrix by Vector Product
- *daxpy* : double-precision daxpy operation
- *dxpay* : double-precision dxpay operation
- *ddot* : double-precision dot product
- *dnrm2* : double-precision norm-2
- *applyJ* : Jacobi preconditioner application

2.1 Sparse Matrix by Vector product (SpMV)

The Sparse Matrix by Vector Product (*SpMV*) is the most expensive operation in any iterative method. The *SpMV* kernel computes the resulting vector Z of the product between a sparse matrix A and a vector $Y$of length N.

Suppose that the sparse matrix A is stored in Compressed Sparse Row (CSR) format and its entries are stored with the aid of three arrays:

- $coef$: double-precision array containing the values of the entries of A
- ja : integer array containing the column indices of the entries of A
- iat : integer array containing the pointers to the beginning of each row in the arrays $coef$ and ja

The algorithm can be written as:

$$Z_i = \sum_{j=iat_i}^{iat_{i+1}-1} coef_j * Y_{ja_j} \text{ for } i = 0, N-1$$

The C++ interface for the *SpMV* kernel is:

- const int N : [in] array size

- const int NT : [in] number of entries of the matrix
- const double *$coef$: [in] double-precision pointer to $coef$ array
- const int *ja : [in] integer pointer to ja array
- const int *iat : [in] integer pointer to iat array
- const double *Y : [in] double-precision pointer to Y array
- double *Z : [out] double-precision pointer to Z array

2.2 Level-1 BLAS

Below are the BLAS Level-1 kernels used by the PCG.

2.2.1 daxpy

The *daxpy* kernel computes the so-called daxpy operation with two double-precision arrays X and Y of length N. The algorithm can be written as:

$$Y_i = Y_i + da * X_i \text{ for } i = 0, N-1$$

The C++ interface for the *daxpy* kernel is:

- const int N : [in] array size
- const double da : [in] double-precision scaling factor
- const double *X : [in] double-precision pointer to X array
- double *Y : [inout] double-precision pointer to Y array

2.2.2 dxpay

The *dxpay* kernel computes the so-called dxpay operation with two double-precision arrays X and Y of length N. The algorithm can be written as:

$$Y_i = da * Y_i + X_i \text{ for } i = 0, N-1$$

The C++ interface for the *dxpay* kernel is:

- const int N : [in] array size
- const double da : [in] double-precision scaling factor
- const double *X : [in] double-precision pointer to X array
- double *Y : [inout] double-precision pointer to Y array

2.2.3 ddot

The *ddot* kernel computes the double-precision dot product between two arrays X and Y of length N. The algorithm can be written as:

$$ddot = \sum_{i=0}^{N-1} X_i * Y_i$$

The C++ interface for the *ddot* kernel is:

- const int N : [in] array size
- const double *X : [in] double-precision pointer to X array
- const double *Y : [in] double-precision pointer to Y array
- double *ddot* : [out] double-precision *dot* product

2.2.4 dnrm2

The *dnrm2* kernel computes the double-precision norm-2 of an array X of length N. The algorithm can be written as:

$$dnrm2 = \sqrt{\sum_{i=0}^{N-1} X_i * X_i}$$

The C++ interface for the *dnrm2* kernel is:

- const int N : [in] array size
- const double *X : [in] double-precision pointer to X array
- double *dnrm2* : [out] double-precision norm-2 of X

2.3 applyJ

The Jacobi preconditioner J is the cheaper (it requires the less number of flops for both computation and application to a vector) preconditioner for the PCG. J is a diagonal matrix where each term is the inverse of the diagonal term of A:

$$J_{ii} = \frac{1}{A_{ii}} \text{ for } i = 0, N-1$$

The matrix J can be stored in a 1D double-precision array without using the CSR format.

The *applyJ* kernel computes the application of J to a vector Y, the resulting vector Z is a simple scaling of Y. The algorithm can be written as:

$$Z_i = J_i * Y_i \text{ for } i = 0, N-1$$

The C++ interface for the *applyJ* kernel is:

- const int N : [in] array size
- const double *J : [in] double-precision pointer to J preconditioner
- const double *Y : [in] double-precision pointer to Y array
- double *Z : [out] double-precision pointer to Z array

3. Implementation details

In this section we describe the main implementation details of the hybrid CPU-FPGA version of the PCG.

At this stage of the project we have focused on the shared memory version, the MPI implementation will be addressed in the next developments once the device (or accelerated) kernels are optimized.

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.

3.1 OpenCL interface

During the set-up of PCG, before the start of the iterative scheme, the OpenCL APIs are used to copy from host to device the matrix, preconditioner and right-hand side. Moreover, some scratch buffers are allocated on the device. The set-up stage is shown in Figure 1 and Figure 2.

```
// Store matrix, rhs, preconditioner and solution array on the device
cl_int err;
OCL_CHECK(err, cl::Buffer d_iat(program_interface.context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
                                (nrows+1)*sizeof(int), (void *)iat, &err));

OCL_CHECK(err, cl::Buffer d_ja(program_interface.context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
                               nterm*sizeof(int), (void *)ja, &err));

OCL_CHECK(err, cl::Buffer d_coef(program_interface.context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
                                 nterm*sizeof(double), (void *)coef, &err));

OCL_CHECK(err, cl::Buffer d_rhs(program_interface.context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
                                nrows*sizeof(double), (void *)rhs, &err));

OCL_CHECK(err, cl::Buffer d_Jac(program_interface.context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
                                nrows*sizeof(double), (void *)Jac, &err));

OCL_CHECK(err, cl::Buffer d_x(program_interface.context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
                              nrows*sizeof(double), (void *)x, &err));

// Copies Host to Device
OCL_CHECK(err, err = program_interface.q.enqueueMigrateMemObjects({d_iat}, 0));
OCL_CHECK(err, err = program_interface.q.enqueueMigrateMemObjects({d_ja}, 0));
OCL_CHECK(err, err = program_interface.q.enqueueMigrateMemObjects({d_coef}, 0));
OCL_CHECK(err, err = program_interface.q.enqueueMigrateMemObjects({d_rhs}, 0));
OCL_CHECK(err, err = program_interface.q.enqueueMigrateMemObjects({d_Jac}, 0));
```

Figure 1: Copies from host to device during the set-up stage

```
// allocate PCG scratches on the host
double *rhs_nrm  = (double*) OOPS_malloc( sizeof(double) ) ;
double *res0_nrm = (double*) OOPS_malloc( sizeof(double) ) ;
double *res_nrm  = (double*) OOPS_malloc( sizeof(double) ) ;
double *resr_nrm = (double*) OOPS_malloc( sizeof(double) ) ;
double *beta     = (double*) OOPS_malloc( sizeof(double) ) ;
double *alpha    = (double*) OOPS_malloc( sizeof(double) ) ;
double *ptap     = (double*) OOPS_malloc( sizeof(double) ) ;

// allocate PCG scratches on the device
OCL_CHECK(err, cl::Buffer d_res(program_interface.context, CL_MEM_READ_WRITE,
                                nrows*sizeof(double), NULL, &err));

OCL_CHECK(err, cl::Buffer d_pxr(program_interface.context, CL_MEM_READ_WRITE,
                                nrows*sizeof(double), NULL, &err));

OCL_CHECK(err, cl::Buffer d_pnew(program_interface.context, CL_MEM_READ_WRITE,
                                 nrows*sizeof(double), NULL, &err));

OCL_CHECK(err, cl::Buffer d_Axp(program_interface.context, CL_MEM_READ_WRITE,
                                nrows*sizeof(double), NULL, &err));

OCL_CHECK(err, cl::Buffer d_scr(program_interface.context, CL_MEM_READ_WRITE,
                                nrows*sizeof(double), NULL, &err));
```

Figure 2: Scratch buffers allocated on the device during the set-up stage.

Finally, for each of the OOPS kernels used in the PCG, a specific interface has been developed to set the arguments and instantiate the execution on the device. An example is provided in Figure 3.

```
void OOPS_daxpy(const int N, cl::Buffer &d_X, const int incX, cl::Buffer &d_Y, const int incY,
                const double da){

   cl_int err;

   // get the kernel
   cl::Kernel krnl;
   OCL_CHECK(err, krnl= cl::Kernel (program_interface.program,"krnl_daxpy", &err));

   // Set the Kernel Arguments
   int narg = 0;
   OCL_CHECK(err, err = krnl.setArg(narg++, N));
   OCL_CHECK(err, err = krnl.setArg(narg++, d_X));
   OCL_CHECK(err, err = krnl.setArg(narg++, incX));
   OCL_CHECK(err, err = krnl.setArg(narg++, d_Y));
   OCL_CHECK(err, err = krnl.setArg(narg++, incY));
   OCL_CHECK(err, err = krnl.setArg(narg++, da));

   // Launch the Kernel
   OCL_CHECK(err, err = program_interface.q.enqueueTask(krnl));

   // Synchronize
   program_interface.q.finish();

}
```

Figure 3: Interface for OOPS kernels.

3.2 OOPS Level-1 BLAS

Please, refer to deliverable 3.5 for a detailed description of the BLAS kernels.

3.3 Sparse Matrix by Vector product

Please, refer to deliverable 3.5 for a detailed description of the SpMV kernel.

4. Preliminary numerical results on XACC prototype

Preliminary tests have been performed using a U280 FPGA board installed on the XACC cluster [3]. For the Sparse Matrix-Vector product, the table below shows a comparison between the OOPS implementation with 16 cores (OOPS_16c) and the software implementation with one core (SW_01c) of the Intel Xeon Gold 6234 processor at 3.30GHz:

	TIME [ms]		
Matrix Name	Cubo_35199	Cubo_246389	Emilia_923
OOPS_16c	12.6	77.9	99.9
SW_01c	4.53	36.3	48.1

The current OOPS implementation is slower than the corresponding CPU software implementation. This performance gap will be addressed in the next Work Package with support from ICCS and TSI, other OPTIMA partners.

5. Concluding remarks

OOPS library has successfully interfaced with the basic linear solver kernels. In the next Work Package, the focus will be on the SpMV kernel. Once the format for matrix storage and its interface are defined the development will proceed on porting the application to distributed memory using MPI directives and multiple FPGA boards.

6. 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] XACC Web Page.
https://systems.ethz.ch/research/data-processing-on-modern-hardware/hacc.html

[4] OPTIMA project Deliverable 3.5 : 1st version of OOPS library