

Beyond PLANCK

Reproducible Research Methodology Report

Deliverable 9.4

Authors: Stratos Gerakakis, Maria Ieronymaki
Date: March 1st, 2019
Work Package: WP9 - Reproducibility in Science
DocId: pkh112-12-1.0



Revision History

Version	Authors	Date	Changes
1.0	Stratos Gerakakis Maria Ieronymaki	March 1st, 2019	Initial Version

Contents

1 Introduction	5
2 Reproducible Science	6
2.1 Reproducibility in the context of BeyondPlanck	6
2.1.1 Reproducibility Plan	6
2.2 Reproducibility in Science in General	8
2.2.1 State of the Art	8
2.2.2 Goals of the Reproducibility Utility Tool	9
2.2.3 Tool Features	9
2.2.4 Modes of Operation	10
2.2.5 Workflow of a Reproducible Procedure	11
2.2.6 Architecture	12
2.2.7 Constructing the Configuration File	13
2.2.8 Tool Status	14
2.2.9 Next Steps	14
3 GPU Usage in BeyondPlanck	15
3.1 Cholesky Decomposition Algorithms	15
3.1.1 In-place Cholesky Algorithm	16
3.1.2 Cholesky-Crout Algorithm	17
3.2 OpenCL Implementation	17
3.2.1 In-place Algorithm	18
3.2.2 Cholesky-Crout Algorithm	19
3.3 CUDA Implementation	21
3.3.1 Cholesky-Crout Algorithm	22
3.4 Input Dataset and Constraint	22
3.5 Timing Performances	23
3.5.1 Derived Dataset	23
3.5.2 Platform Specifications	23
3.5.3 Benchmark	24
3.5.3.1 Timing performances VS Cache Memory	24
3.5.3.2 Cross-platform Comparison	27
3.5.3.3 State-of-the-art: Testing Intel MKL	29
3.5.3.4 C++ implementation without for loops	29
3.6 Result Accuracy	30
3.7 Conclusion and Future Perspective	31

Applicable and Reference Documents

[AD01] pkh112-06-1.0 Deliverable 9.3: Reproducibility in Science Report

1 Introduction

As specified in the Work Package 9 definition, the work included in the WP9 can be subdivided into three main classes.

1. The first class concerns code organization and distribution. According to the Open Source philosophy of this project, all source codes will be made publicly available through a git Repository.
2. The second and most work-intensive class concerns reproducible research. In order to ensure that external users will be able to access, reuse and reproduce the codebase developed in the project, Planetek software scientists (who are not themselves cosmologists) will run the code externally, as if they were external users, and they will be in charge of developing suitable documentation. By having non-cosmologists performing this work, the end-products will be far more user-friendly.
3. Finally, with the aim to facilitate reproducibility of parallel algorithms, this WP will also investigate techniques to replace expensive computing grid calculations with low cost local or remote GPU based environment by converting suitable code into low level representation for GPU execution.

This deliverable, although titled as “Reproducible Science Report”, in reality it encompasses two major sections. One for the actual “Reproducible Science” and the other about “GPU Usage in BeyondPlanck”, both sections being part of the same WP9.

2 Reproducible Science

Our goal in WP9 is to make it as easy as possible to have reproducible BeyondPlanck results by the end of the project. For this purpose we have broken down the general concept of reproducible science in two parts:

- How reproducibility can be applied in the context of BeyondPlanck
- How can reproducibility be applied more easily in science in general

2.1 Reproducibility in the context of BeyondPlanck

For the purposes of reproducibility, BeyondPlanck, in general, is not a typical scientific project. It consists of multiple individual work-packages, each one a standalone scientific endeavor of each own, and all of them working together in synergy in order to produce a final set of results. In other words, it is a superset of a scientific piece of work, encompassing many individual pieces all working for a common goal.

From the previous research we have performed in this project, on the current status of Reproducibility in science in general, we have seen that although reproducibility is a sought after feature in many scientific pieces of work, in reality it is something that is difficult or cumbersome to achieve. Usually it is added at the end of a project without a lot of emphasis or real work behind it.

Given the complex nature of the structure of BeyondPlanck, this reproducibility effort is especially complex here too, thus the need for a separate Work Package in order to deal with it.

2.1.1 Reproducibility Plan

Our goal for providing clear, concise and detailed information for reproducing the work performed in BeyondPlanck is to develop an online guide describing and documenting each step of the workflow used to produce the final results, hosted in the official BeyondPlanck website (<https://beyondplanck.science>). This will be provided in the form of a separate section in the project website, where each step of the Gibbs Sampler loop will be individually described and all incoming and outgoing files and parameters, are fully documented.

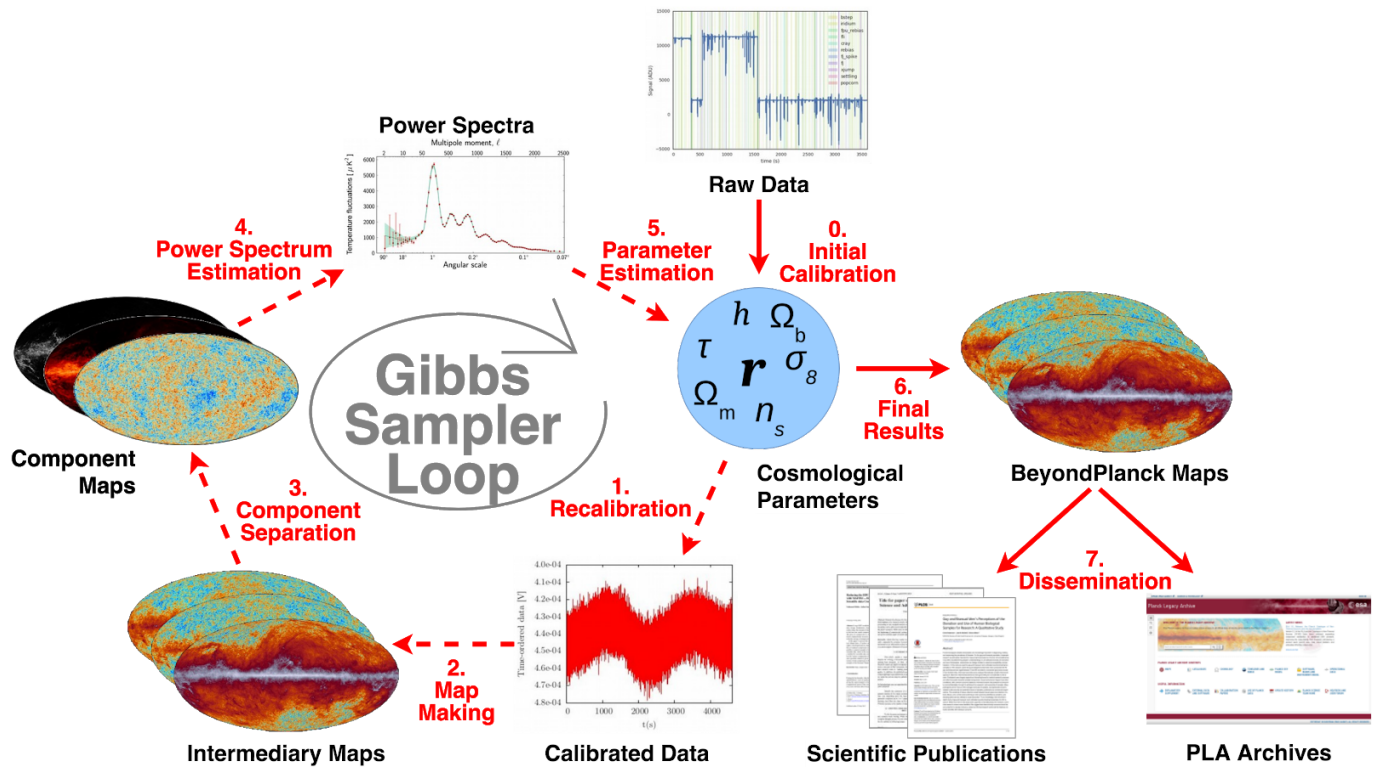


Figure 1: Major BeyondPlanck execution phases

Based on the Gibbs Sampler loop and the supporting processes at the beginning and the end of the loop, there will be 8 major phases that will be fully documented:

1. Initial raw data used by BeyondPlanck
2. Recalibration process
3. Map Making process
4. Component Separation process
5. Power Spectrum Estimation process
6. Parameter Estimation Process
7. Final Results Packaging
8. Dissemination and inclusion to PLA Archives

Each of these phases will be appropriately documented, and each one will offer:

- A detailed description of the tasks that take place within that phase
- The location of the codebase in the BeyondPlanck repository where code is executed from
- The input data required to be present at the beginning of the phase
- The configuration parameters by which the particular phase can be customized, configured and executed with
- A detailed description of the output produced.

At the current status of BeyondPlanck where the Gibbs loop is at its earliest stage, it is a little premature to go onto a full documentation effort, since many of the integration paths between components is not final yet. We are planning on starting the full documentation effort, on the second half of the BeyondPlanck project when the required components have matured and stabilized enough.

2.2 Reproducibility in Science in General

In the process of reviewing available options for tools to assist us in the task of reproducing the scientific work produced through BeyondPlanck we did an overview of the current state of the art in Reproducibility in Science. Through that process we realized two things:

1. That reproducibility in science is a very fragmented market and that there are no clear winners or de facto ways of producing reproducible science.
2. That BeyondPlanck as a project, is a very complex endeavour and difficult to fit into any of the existing reproducibility tools or services.

That led us to the following two conclusions:

1. By examining the available reproducibility tools, we feel that there is a gap that we can easily fix by providing a simple tool that will cover a big part of the needs of scientists that are looking for an easy way to make their work reproducible, thus deciding to provide this tool for Reproducibility in Science in General.
2. Although we might not be able to use the proposed tool in the totality of the BeyondPlanck, we can still validate the tool by applying it to smaller subset of it, and trying to make reproducible individual Work-packages

In the following sections we describe our plans for the reproducibility utility tool.

2.2.1 State of the Art

The latest developments on Reproducibility in Science have been examined and presented in detail pkh112-06-1.0 Deliverable 9.3 Reproducibility in Science report [AD01].

Several initiatives are taken to this direction and services have been produced. Those include web based applications, desktop applications as well as online computing resources like the ones listed below:

- Taverna (workflow management)
- Kepler-Project (GUI based scientific workflow system)
- Open Science Framework (non-profit from Center of Open Science)
- CodeOcean (cloud based computational reproducibility platform)
- Zenodo (online services for reproducibility) (also Horizon2020 funded)
- Gitlab/Github/Bitbucket (online repositories)

Despite the fact that some of them are offering impressive services, several disadvantages can be pointed out including:

- Having to comply with the imposed workflow of each service
- Not always free to use for all future work
- Enforce people who are interested in reproducing someones scientific work to also become members of that service
- No guarantees that the same services will be there tomorrow
- Vendor lock in

2.2.2 Goals of the Reproducibility Utility Tool

The following goals have been identified for the Reproducibility in Science Tool we intend on building:

- Lower the adaptation entry barrier
- Automate the reproducibility workflow as much as possible
- Make it very easy for the original author but also for the people trying to reproduce the scientific work
- Stay always current, with support for any new upcoming services or tools
- Support as many reproducibility use-cases as possible
- Reuse existing services and infrastructure to tie them all together

2.2.3 Tool Features

The intention of the tool is to be aimed at simple scientific workflows, usually the work of a small number of authors, that have a simple and clearly defined execution plan.

It will be delivered as a command line based, cross platform, single binary. The tool will be just a command line tool (think git) that does not have a user interface.

It should be able to derive most of the required information for its operation from various configuration files, or command line user input. Although the majority of the required config parameters will be coming from configuration files, the user should be able to overwrite some of them by the use of command line parameters, at execution time.

It will have a Modular Design capable of being expanded by plugins. It will be localized in different languages (already it supports English, Greek, Italian and Norwegian but it can be translated into any other language too)

Advantages:

- Being cross platform it is usable by users of all OSes
- Easy to install and update, as it is just one binary
- Being command line based, aligns with the interface provided by multiple similar tools (git, curl, docker etc)
- Being a command line tool, it can be integrated into existing execution workflows
- The tool embraces a modular design, offering support for writing extensions to integrate with third party services and tools

- Thanks to its modular design, it can accommodate new trends by easily incorporating them into the proposed workflow.
- Anyone can help, by implementing a plugin for an obscure service
- Localised to many languages in order to lower the entry barrier
- Presentation in the BeyondPlanck website
 - Documentation for each available plugin (configuration, features, usage)
 - Provide instructions for extending the tool
 - Offer troubleshooting and sample use-cases

2.2.4 Modes of Operation

The tool should support basic modes of operation. (think git add, git commit, git rebase etc)
Add, commit and rebase are different modes of the same git tool.

Initially supported modes are:

- verify
- setup
- run
- publish
- reproduce

Each mode has its own set of accompanied command line parameters, further defined in the following sections.

Verify Mode

Make a sanity check if all the required dependencies for a proper reproducible environment are installed, and check the validity of the configuration files.

Each provided plugin should offer a method called verify that is tasked to make sure that all configuration parameters, provided by the user are correctly set. The plugin itself is the only one able to make sure that it's custom configuration parameters are correct.

The verification should produce two different kind of messages: errors and warnings.

- Errors are configuration messages that prevent the plugin from properly operating and changes must be made.
- Warnings are configuration message that do not prevent the plugin from working, but they might produce unexpected or unwanted results.

As an example you can check the verify method of the HTTP input plugin, for a demonstration of how this can be done.

Setup Mode

Setup the current environment with the required input files, as defined in the reproducibility file. This is the mode where the required input files are downloaded.

This mode should internally perform a call to the 'valid' mode before proceeding with the execution. This way any potential configuration errors should be caught early one before the operation starts. The execution details for downloading files is up the implementation of each plugin.

Run Mode

Runs the current execution algorithms that manipulates input files and generates output.

Publish Mode

Once the scientific work is finalized, publish the produced results files.

Reproduce Mode

This is a mode used by interested third parties (not by the original author, although they might be able to execute it too) and is responsible for reproducing some scientific work that has been performed following the published Reproducibility in Science workflow.

2.2.5 Workflow of a Reproducible Procedure

For authors:

- They define what are the base input files required for their work
- They define the code and execution environment that manipulates the input files
- They describe where they want to publish their results, or where they can be found (if published by third party services)

For people recreating the work they ought to just run: `recreate` to have the tool:

- download the required files
- download the code files
- execute the code and
- get the same results as the original author.

The following table is showing the available execution modes and their intended audience:

Commands	Original Author	Third Party Recreators
verify	Yes	Yes
setup	Yes	Yes
run	Yes	Yes
publish	Yes	
reproduce		Yes

Table 1: Execution Modes

2.2.6 Architecture

The Reproducibility utility tool has been designed following a modular architecture paradigm, making the tool as extensible as possible.

The tool consists of several plugins, which are responsible for implementing functionalities for third party services.

The provided plugins can be categorized according to their functionality in:

- Input plugins
- Source code Management Plugins
- Execution Plugins, and
- Output Plugins,

With each category containing several plugins.

The tool architecture and the provided plugins are depicted in the figure below:

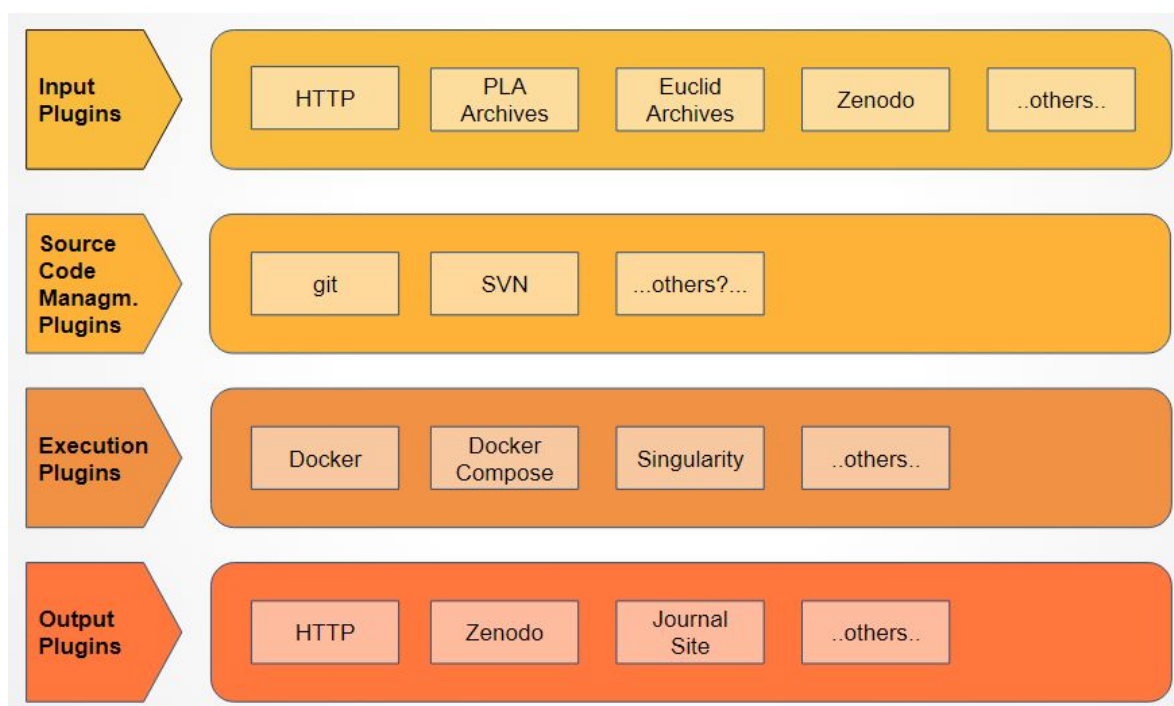


Figure 2: Reproducibility tool architecture

Upon release, the tool will support a basic integration with external services, but it should be easy to extend the tool to support any other services that might be requested or become popular in the future. Proper documentation on how to extend the tool in order to integrate with more services will also be offered together with the tool.

2.2.7 Constructing the Configuration File

The driving force behind the functionality offered by the Reproducibility tool, is the reproducible.yml file. This will be a plain text file that encapsulates all required information that makes a scientific paper reproducible.

This is the basic configuration file that defines how the specific work produced in a paper is reproducible. It is meant to encapsulate all the high level details of the reproducibility pipeline that we intend to implement.

A sample configuration file together with help notes explaining its usage is depicted below:

```
# Allows to mark the current reproducible.yml file format version
version: '0.1'

# Identify the list of input files used in this RiS workflow (Note 1)
inputs:
  service: `name_of_service`      # (Note 2)
    other_param1: other_value1    # (Note 3)
    other_param2: other_value2    # (Note 3)
  service: `another_service`      # (Note 4)

# Specify where the code can be accessed from
code:
  scm: git                        # (Note 5)
  scm_config_param1: value1       # (Note 6)
  scm_config_param2: value2

# Define the produced output artifacts that ought to be preserved (Note 7)
output:
  service: `name_of_service`      # (Note 8)
  other_param1: other_value1      # (Note 9)
  other_param2: other_value2
```

Notes:

1. "Most" of the input files, early in the RiS adoption curve, will NOT be packaged or identified by ID, so we need to address this (more likely with the inclusion of a generic HTTP plugin that just downloads urls)
2. Downloading files can be implemented by various download plugins, identified by a unique name, to be used as the service name
3. Each plugin can define its own set of configuration parameters, and must accurately document these parameters. For an example look at the HTTP input plugin
4. More than one inputs services should be configurable to allow files to be downloaded through various services. So some files can be downloaded from the HTTP input

plugin (through simple url calls) and some other files from a (hypothetical) PLA_Archives plugin, where files are downloaded from the PLANCK Archives website according to the machine interface provided by PLA.

5. Mark what kind of Source Code Management tool we are using. Initially only 'git' supported, but svn should be next. Contrary to the inputs section the scm option should only support one SCM service at a time, as there is no need to support more than one at the same time.
6. Each scm service should define and document it's own set of configuration parameters (like the HTTP input plugin has done)
7. This defines what output files are ought to be preserved by uploading to a public location and assigning a unique number.
8. Once again, this task of preserving the output artifacts will be implemented by different output plugins that can deal with the preservation process on their own.
9. Each service defines and documents its own set of configuration parameters.

2.2.8 Tool Status

We have a working version of the tool already working. We have implemented one plugin each for each of the first three workflow phases: the *HTTP* service in Input Plugins, the *git* in Source Code Management Plugins and the Execution

We are implementing two use cases:

- A very basic simple one, that is used for development purposes
- An full implementation of reproducing the BeyondPlanck work in Work Package 7 - Physical Interpretation

2.2.9 Next Steps

Continue implementing the final phase of the workflow, Uploading resulting files and assigning unique IDs to output results. Zenodo offers free persistent identifiers (DOIs) for results published on their site and is the most likely service to be integrated with.

We might also implement more plugins for all workflow phases, based on continuing collecting feedback for the provided functionality.

Finally, we will have to properly document and populate the Reproducibility in Science tool in our website.

3 GPU Usage in BeyondPlanck

This section presents the approaches adopted during investigation about feasible parallel implementations of the Cholesky factorization algorithm using a heterogeneous processing environment (i.e. a system hosting both multi-core CPUs and many-core GPUs or similar HPC devices).

This study is the continuation of preliminary analysis documented in [AD01] “*Deliverable 9.3, Chapter 5, GPU for PLANCK*”, where we described our hypothesis about whether and where to introduce a GP-GPU acceleration stage in the processing pipeline of BeyondPlanck, based on our acquired knowledge of the project and the related objectives, computations and issues.

In that document we firstly provided a quick overview of GPU technologies and capabilities and then we focused on the pipeline and its components.

Further planned investigations were stimulated by active interaction with members of the consortium for determining potentially improvable “*steps*” of the pipeline. Scientific and technical discussions allowed the consortium to choose the Cholesky factorization as a relevant algorithm, deserving our attention with the aim of translating it into a form suitable for GPU (Graphics Processing Unit) and HSA (Heterogeneous System Architecture) execution.

The Cholesky decomposition has been selected as benchmarking algorithm being widely used in several linear algebra problems, and moreover due to its challenging memory-intensive nature. Different implementations and designs have been considered and compared with the state-of-the-art solutions (such as LAPACK in Intel-MKL) in terms of both performances and results.

Our investigation was possible thanks to the consortium providing us relevant reference data to apply analysis and writing/testing algorithms upon. In particular the consortium provided us a reference data matrix, typical of BeyondPlanck science problems, and a performance baseline to compare results to.

3.1 Cholesky Decomposition Algorithms

The Cholesky decomposition is a well-known method for matrix factorization widely used to solve linear systems of equations. In detail, the Cholesky decomposition of a Hermitian positive-definite matrix \mathbf{A} gives:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^*$$

where L is a lower triangular matrix and L^* denotes the conjugate transpose of L . In case of A real matrix, as in our test case, the factorization can be written as:

$$A = LL^T$$

where L^T denotes the transpose of L . Every Hermitian positive-definite matrix (and thus also every real-valued symmetric positive-definite matrix) has a unique Cholesky decomposition.

The $A = LL^T$ decomposition is sketched in Figure 3.

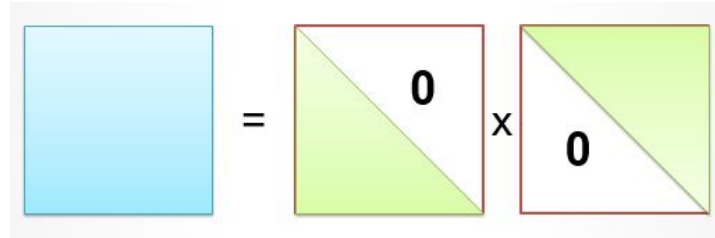


Figure 3: Visual representation of Cholesky decomposition $A=LL^T$

The following formula defines how an element (i,j) of the matrix L can be recursively computed:

$$L_{ij} = \begin{cases} 0 & \text{if } i < j \\ \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2} & \text{if } i = j \\ \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right) & \text{if } i > j \end{cases}$$

In this study we considered two different versions of the Cholesky decomposition, denoted and described as follows:

- in-place Cholesky algorithm
- Cholesky-Crout algorithm

In spite of that choice, there a lot of variants of the same algorithm, with different memory pattern and computation types.

3.1.1 In-place Cholesky Algorithm

In the current version of the algorithm all the computations are performed *in-place*, i.e. the input matrix A is progressively updated as shown in the corresponding pseudocode presented below in this section. Each element of matrix A is used as argument in

subsequent operations as soon as it is computed. Such implementation allows to halve the total memory occupation because there is no need to store the output values in an additional matrix object.

Pseudocode

```

* DO I = 1, N
*
*   SECTION A
*   A(I,I) = SQRT (A(I, I))
*   DO J = I+1, N
*       A(J,I) = A(J,I)/A(I,I)
*   END DO
*
*   SECTION B
*   DO K=I+1,N
*       DO J = K, N
*           A(J,K) = A(J,K) - A(J,I)*A(K,I)
*       END DO
*   END DO
*
* END DO

```

3.1.2 Cholesky-Crout Algorithm

An alternative implementation has been investigated using the Cholesky–Crout algorithm, which starts from the upper left corner of the matrix L and proceeds to calculate the matrix column by column.

Pseudocode

$$\begin{aligned}
 &\forall i = 1, \dots, m \\
 &l_{i,i} = \sqrt{a_{i,i} - \sum_{k=1}^{i-1} l_{i,k}^2} \\
 &\forall j = (i+1), \dots, m \\
 &l_{j,i} = \frac{1}{l_{i,i}} \left(a_{j,i} - \sum_{t=1}^{i-1} l_{j,t} l_{i,t} \right)
 \end{aligned}$$

3.2 OpenCL Implementation

OpenCL (Open Computing Language) is an open standard for cross-platform parallel programming for heterogeneous computing devices (such as CPUs, GPUs, FPGAs).

One of the most interesting and particular features of OpenCL is its high portability, which is

achieved by having different platforms as possible target computing devices. However, an OpenCL implementation requires a considerable amount of setup code, in order to detect and initialize devices, compile kernels), which yields a more complex code and higher initialization overhead.

In the following sections we describe the versions of Cholesky-Crout algorithm investigated using OpenCL.

3.2.1 In-place Algorithm

Considering the in-place algorithm, it can be noted that two different sections (named **Section A** and **Section B**, Figure 4) can be distinguished in each single iteration of the external loop.

```

DO I = 1, N
  A(I,I) = SQRT (A(I, I))
  DO J = I+1, N
    A(J,I) = A(J,I) / A(I,I)
  END DO
  DO K=I+1,N
    DO J = K, N
      A(J,K) = A(J,K) - A(J,I)*A(K,I)
    END DO
  END DO
END DO

```

Section A

Section B

Figure 4: In-place algorithm

The outermost loop iterates along matrix columns. For each column, the **Section A** is responsible for the main diagonal calculation, followed by the update of all the elements below diagonal in the current column. In particular, as shown in Figure 5, each cell of the i -th column depends only on the respective diagonal element, so every pixel in a lower-column can be computed in parallel.

On the other hand, the **Section B** updates all the remaining right-pixels below the diagonal (except the current column), using values already computed in **Section A**. The calculation performed in **Section B** can be parallelized since the elements do not depend on each other. The only sequencing rule to be guaranteed is that between section **A** and **B**, for each column.

Figure 5 shows the computation pattern adopted in the in-place Cholesky algorithm.

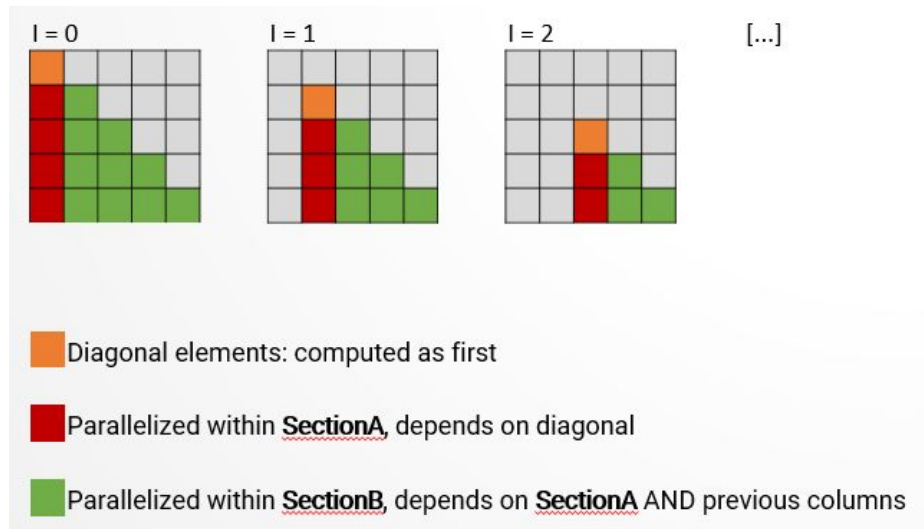


Figure 5: Computation pattern

Taking into account for these constraints, two different kernel implementations have been tested in OpenCL:

- Multi-thread + multiple kernel implementation
- Mono-thread + single kernel implementation

Multi-thread + multiple kernels

In the current version, the host code iterates through all of the columns, enqueueing two different kernels for **Section A** and **Section B**.

The number of threads instantiated for each columns varies in **Section A** and **Section B**.

Mono-thread + single kernel

For the implementation discussed here, the host code does not iterate through the columns, but a single thread is enqueued only ones. In this case, the kernel code is responsible for loops at each levels executing each iteration sequentially: the actual *not-dependence* between Section A and Section B is not exploited, and parallelization is not applied.

3.2.2 Cholesky-Crout Algorithm

Reviewing the Cholesky-Crout algorithm, the parallelization can be performed row-wise: for each column, each element on its lower-diagonal (not on the main diagonal) is independent of each other, and therefore parallelizable. However, the columns are not independent of each other. There are, therefore, two synchronization points:

1. a column j can only be calculated once all of column $j-1$ has been calculated;
2. the elements within a column (except the main diagonal) can only calculate their values after the corresponding main diagonal value has been computed (see Figure 6 and Figure 7).

$$\forall i = 1, \dots, m$$

$$l_{i,i} = \sqrt{a_{i,i} - \sum_{k=1}^{i-1} l_{i,k}^2}$$

Diagonal element computed as first

$$\forall j = (i+1), \dots, m$$

$$l_{j,i} = \frac{1}{l_{i,i}} \left(a_{j,i} - \sum_{\iota=1}^{i-1} l_{j,\iota} l_{i,\iota} \right)$$

Figure 6: Cholesky-Crout algorithm

```

3  _kernel void cholesky_crout(
4  _global double *A,
5  _global double *L,
6  const unsigned int N,
7  const unsigned int col)
8  {
9
10     const int g_id = get_global_id(0);
11     int row = col + g_id;
12     int k;
13     double sum = 0.0;
14     double value;
15     double sum_d = 0.0;
16
17     if (g_id == 0)
18     {
19         for (k = 0; k < col; k++)
20         {
21             sum_d += L[col * N + k] * L[col * N + k];
22         }
23
24         L[col * N + col] = sqrt(A[col * N + col] - sum_d);
25     }
26     else
27     {
28         for (k = 0; k < col; k++)
29         {
30             sum += L[row * N + k] * L[col * N + k];
31             sum_d += L[col * N + k] * L[col * N + k];
32         }
33
34         value = sqrt(A[col * N + col] - sum_d);
35         L[row * N + col] = (1.0 / value * (A[row * N + col] - sum));
36     }
37 }

```

Get the current thread identifier and use it as index in the matrix

Item on the diagonal

Figure 7: Cholesky-Crout algorithm implementation

Figure 8 clearly shows the pattern of the Cholesky-Crout algorithm, in which each element of a column depend only on its row and the row of the main diagonal intersecting this column. In detail, green elements are the active elements for each iteration, which directly depend on the yellow elements, indirectly on the orange ones, and do not depend on the grey ones.

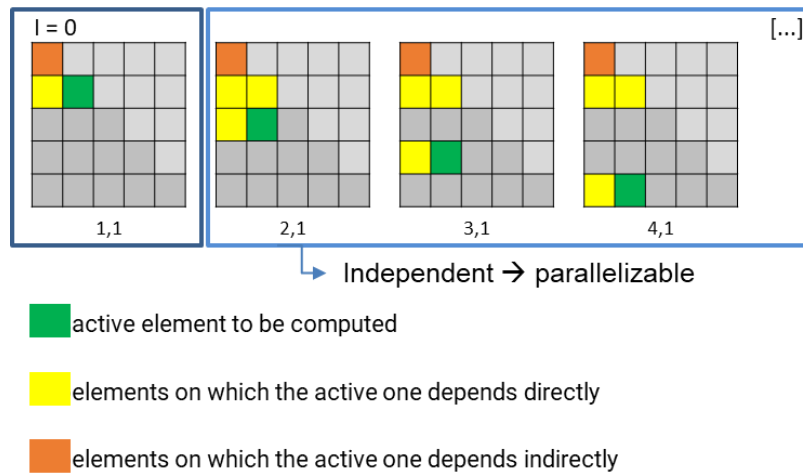


Figure 8: Cholesky-Crout algorithm pattern

Taking into account for these constraints, two different kernel implementations have been tested in OpenCL:

- Multi-thread implementation
- Mono-thread implementation

Multi-thread + single kernel

For the implementation discussed here, the host code iterates through all of the columns, enqueueing kernels to process the current column.

In particular, this implementation uses a single kernel to calculate each value of the column by enqueueing, at each j -th column iteration, $N-j$ work-items (i.e. $N-j$ threads). Since there is no synchronization between the main diagonal element and the others, all items are required to recalculate that value locally. Although this may have impact on performance, it is necessary for the algorithm to be correct. Also, since the same kernel is used for both the main diagonal and the other elements (which have very different algorithms for calculating their value), there's the need for an additional conditional statement on the start of the kernel, which in turn may also result in loss of performance.

Mono-thread + single kernel

For the implementation discussed here, the host code does not iterate through the columns, but a single thread is enqueued only ones. In this case, the kernel code is responsible for loops at each levels and *de facto* no parallelization is applied.

3.3 CUDA Implementation

In this section, we describe our implementations of the Cholesky decomposition in GPU using CUDA.

CUDA is a computing platform and API created by NVIDIA for general-purpose computing on GPUs. CUDA code must be compiled with its own LLVM-based C/C++ compiler, and has the limitation of only being able to be executed on CUDA-enabled GPUs.

The CUDA workflow is very similar to that of OpenCL: the host code enqueues different CUDA kernels to execute on the GPU, and kernel code can only handle device memory, while host code can only handle host memory, with CUDA API calls for memory transfer. Each CUDA kernel execution is called a *thread*, and the threads are divided into individual *blocks*. The total number of blocks and the maximum number of threads per block are limited by hardware.

3.3.1 Cholesky-Crout Algorithm

Multi-thread + single kernel

The base algorithm tested in CUDA was the Cholesky-Crout in multi-thread + single kernel version.

The overall strategy used for CUDA implementation is very similar to the one used in OpenCL. First, the input matrix of the appropriate size is loaded to the host memory, and then copied to the GPU memory. Then, a host loop goes through all columns enqueueing the kernel responsible for the actual matrix decomposition.

3.4 Input Dataset and Constraint

The considered input dataset consist of a symmetric positive-definite matrix with the following features:

N x N	2688 x 2688
Data Format	double-precision floating point (64 bit)
Memory Footprint	~57.803 MB

Table 2: Input dataset features

All computation were performed in double-precision.

The input matrix used in our test has been provided by the consortium and reflects a realistic use case in CMB field of study.

Additional tests were executed for several matrix sizes in order to measure the algorithm performances over different data loads.

3.5 Timing Performances

In this section, we evaluate the results of the aforementioned Cholesky decomposition algorithms using OpenCL, in terms of timing performances.

It is important to stress that it is possible to run the same OpenCL code with different so-called **compute devices**, that is, different platforms in which the code will be executed, and this platform can either be a CPU or a GPU.

Several tests were performed varying the execution platforms and the input data size.

3.5.1 Derived Dataset

Several subsets of the initial test dataset have been created in order to evaluate the execution time as function of the input data size. In particular, the behavior in term of execution time with respect to the available cache memory has been analyzed.

The auxiliary test datasets have been characterized by the number N of rows (and columns) which the matrix is composed of.

Each subset is composed of **N -by- N** double values extracted from $(row,col)=(0,0)$ of the initial matrix.

3.5.2 Platform Specifications

CPUs

Processor	Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz	Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz	Intel(R) Core(TM) i7 CPU 3770 @ 3.40GHz
Cores	4	16	4
Logical Processor	8	32	8
L1 cache x core	32.768 KB	32.768 KB	32.768 KB
L2 cache x core	262.144 KB	262.144 KB	262.144 KB
L3 cache x core	8388.608 KB	20971.520 KB	8388.608 KB
Total Cache	~9 MB	~25 MB	~9 MB

Table 3: CPUs Platform specifications

GPUs

Model	AMD Radeon(TM) RX 460 Graphics	NVIDIA GeForce GTX 1050	NVIDIA Quadro K600	NVIDIA Quadro K2000	NVIDIA Tesla K20c
Compute Units	14	n.a.	n.a.	n.a.	n.a.
Stream Processors	896	640	192	384	2496
Graphics clock	1200 MHz	1354 MHz	876 MHz	954 MHz	706 MHz
On Board Memory	4096 MB GDDR5	2048 MB GDDR5	1024 MB DDR3	2048 MB GDDR5	5120 MB GDDR5

Table 4: GPUs Platform specifications

3.5.3 Benchmark

3.5.3.1 Timing performances VS Cache Memory

The following two different devices, in terms of OpenCL compiler, have been considered:

- Intel® CPU Runtime for OpenCL™ Applications / Intel(R) Core(TM) i7 CPU 860@ 2.80GHz
- Experimental OpenCL 2.1 CPU Only Platform / Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz

Kernel Comparison

Algorithm	Kernel version	Average processing time [s]	
		Intel® CPU Runtime for OpenCL™ Applications	Experimental OpenCL 2.1 CPU Only Platform
In-place	Multithread + Multiple kernel	66.77102975	68.16231025
	Monothread + Single kernel	225.0516145	226.436251
Cholesky-Crout	Multithread + Single kernel	5.735215	4.931208
	Monothread + Single kernel	7.782861	8.3914775

Table 5: Processing time comparison

From now on, the implementation of Cholesky-Crout algorithm in multi-thread version will be

used and the execution platform will be set to **Experimental OpenCL 2.1 CPU Only Platform / Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz.**

L3 cache : 8MB

In the following section, the trend of processing time as function of input matrix size has been analyzed. In detail, we consider matrices of several sizes NxN (i.e. different data memory footprint) related to the each cache memory level.

	N	Matrix SIZE [MB]	OpenCL on Intel CPU	GNU g++ sequential
			PROCESSING TIME [us]	PROCESSING TIME [us]
L3 cache	500	2.00	68002	195313
	750	4.50	121982	651563
	1000	8.00	197998	1522500
	1250	12.50	298969	3018750
	1500	18.00	462982	5268750
	1750	24.50	612960	8750000
	2000	32.00	1917908	12465620
	2250	40.50	3341984	18828120
	2500	50.00	4115964	25217190
	2688	57.80	5382962	30612500

Table 6: L3 cache comparison

Figure 9 shows the mean execution time elapsed for Cholesky factorization of NxN matrix; the trend shows a break-point around N=1750 (i.e. 24MB, approximately three-times the available L3 cache).

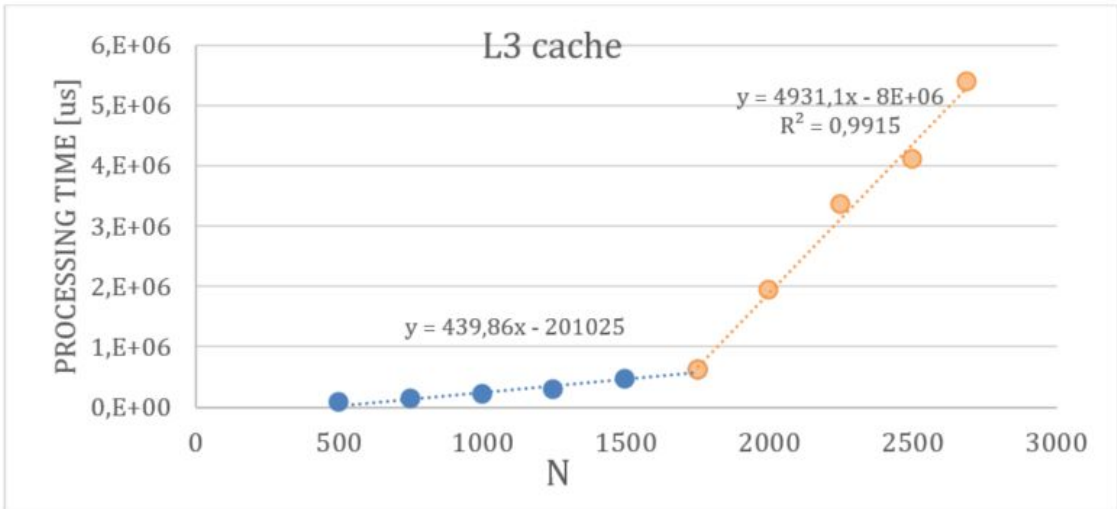


Figure 9: Mean execution time elapsed for Cholesky factorization of NxN matrix

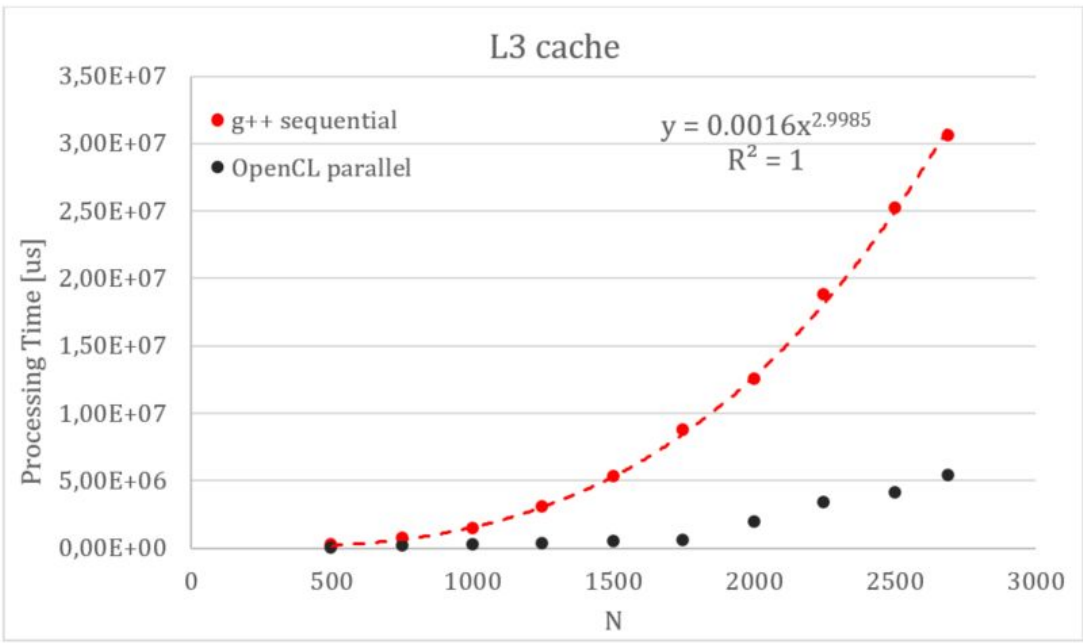


Figure 10: OpenCL parallel and serial Cholesky-Crout comparison

Comparison between OpenCL parallel implementation (black) and serial Cholesky-Crout version (red) in terms of execution time are reported in Figure 10. The same comparison has been repeated for input matrices having memory occupation comparable to L1 and L2 caches.

L1 cache : 128 KB

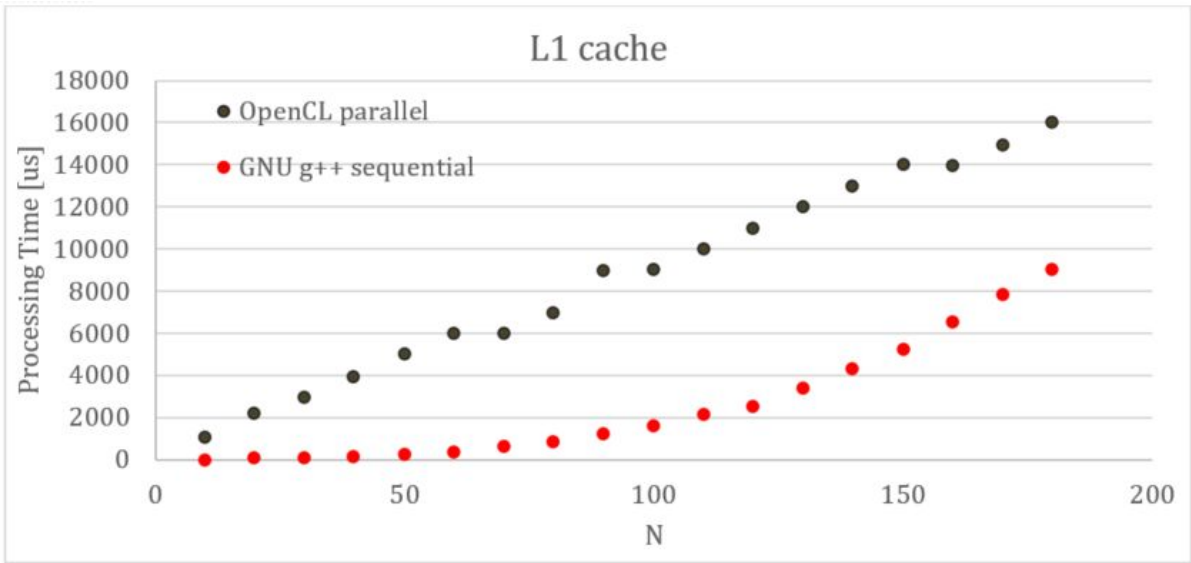


Figure 11

Figure 11 shows that the serial version of the algorithm gives systematically better

performances with respect to the parallelized version, in cases of input matrix smaller than the available L1 cache.

L2 cache : 1024 KB

On the contrary, taking into account inputs with memory occupation comparable to L2 cache, it seems that the OpenCL parallelized implementation offers better performance than the sequential version starting from $N \sim 270$ (~ 600 KB), as presented in Figure 12.

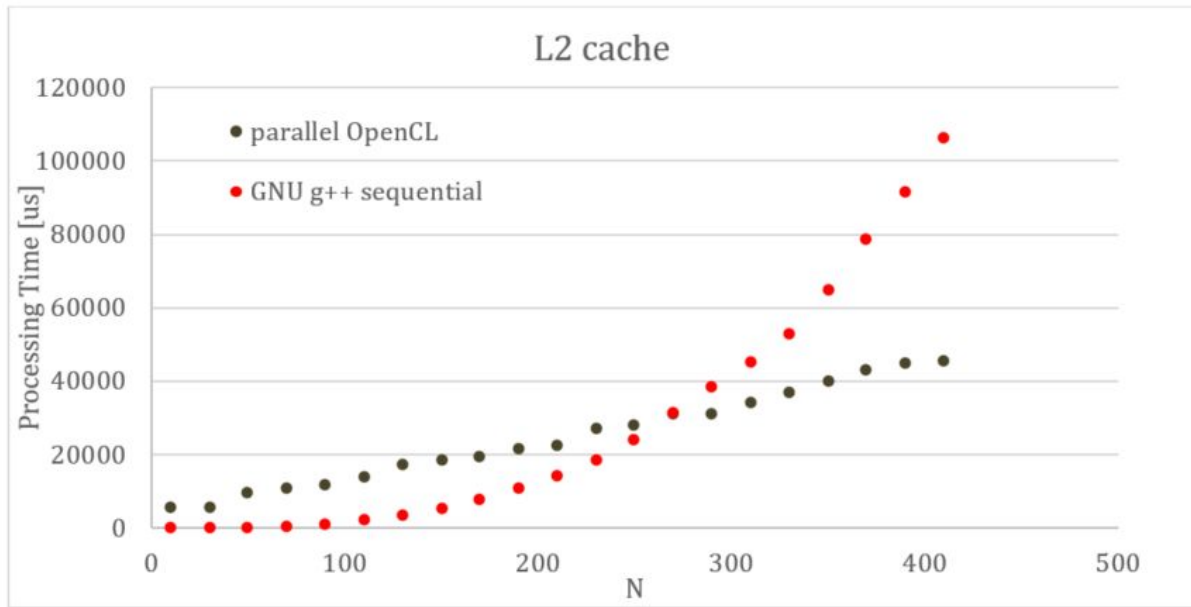





Figure 12

3.5.3.2 Cross-platform Comparison

OpenCL implementation

Results achieved running the OpenCL implementation of Cholesky decomposition are reported in the table below, w.r.t. to four different target platforms. The considered testing devices consist of six GPU and three CPUs with different technical specification in terms of number of cores/threads and clock rate.

Algorithm + Kernel	Average processing time [ms]				
	Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz	Intel(R) Core(TM) i7 CPU 3770 @ 3.40GHz	Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz	AMD Radeon(TM) RX 460 Graphics	NVIDIA GeForce GTX 1050
	CPU 4core, 8thread	CPU 4core, 8thread	CPU 16core, 32thread	GPU (64 bit floating point)	GPU (64 bit floating point)
Cholesky-Crout Multi-thread + Single kernel	~5000	~2000	~700	~8000	~4500



Many core CPUGPUGPU

Table 7

The obtained benchmark shows that the current implementation and input matrix size offer better performances on CPU than GPU. In particular, parallelization on CPU had a good speedup experimenting a very powerful device, characterized by many cores and large cache.

CUDA implementation

Additional analyses have been performed testing the CUDA implementation of Cholesky-Crout algorithm on the available NVIDIA GPU devices. As shown in the table below, the considered CUDA implementation provides a sensible improvement in timing for the most powerful device, for which the matrix decomposition is executed in ~1.3 seconds, including the data transfer between host and device.

Algorithm + Kernel	Average processing time [ms]			
	NVIDIA GeForce GTX 1050	NVIDIA Quadro K600	NVIDIA Quadro K2000	NVIDIA Tesla K20c
	GPU (64 bit floating point)	GPU (64 bit floating point)	GPU (64 bit floating point)	GPU (64 bit floating point)
Cholesky-Crout Multi-thread + Single kernel	~1300	~9900	~5000	~1900

Table 8

Despite this, considering both the OpenCL and CUDA implementations over the tested platforms, the best performance is offered by the OpenCL version running on many-core CPU.

3.5.3.3 State-of-the-art: Testing Intel MKL

In order to benchmark CPU optimizations of the Cholesky decomposition, we wrote a simple program that makes the appropriate external calls to LAPACK routines and measured its elapsed time.

```
26
27     pkcore::STimeValue tv0, tv1;
28     pkcore::STimeDesc td;
29
30     pkcore::TimeGetValue(&tv0);
31
32
33     for (size_t i = 0; i < nc; ++i)
34     {
35         if (0 != LAPACKE_dpotrf(LAPACK_ROW_MAJOR, 'L', n, reinterpret_cast<double*>(data), n)) {
36             std::cout << "ERROR !" << std::endl;
37             return -1;
38         }
39     }
40
41     {
42         pkcore::TimeGetValue(&tv1);
43         fprintf(stdout, "ELAPSED: %zu microseconds.\n",
44             , pkcore::TimeGetElapsed(&tv0,&tv1,&td) / nc);
45     }
```

Figure 13

The testing program read the input matrix as contiguous-double array and perform the external call to LAPACK routine *dpotrf*, corresponding to Cholesky decomposition, as reported in Figure 13. It's worth noting that the calculation is also done in-place, so the input matrix is also the output matrix. The table below shows the results obtained for the tested platforms.

	Intel(R) Core(TM) i7 CPU 3770 @ 3.40GHz CPU 4core, 8thread
Intel MKL - LAPACKE	~128ms

Table 9

3.5.3.4 C++ implementation without for loops

An alternative approach has been tested in order to deal with the evident memory-intensive nature of the considered Cholesky algorithm. In detail, the OpenCL parallelization has been replace by a pure C++ sequential implementation in which the formulation of each output value is explicitly expressed, so completely avoiding the use of multiple nested loops in the code.

This approach is not feasible considering the standard test dataset size, then as been tested using a small 3x3 matrix, resulting in a total computation time of ~3us. The target platform for this benchmark is the following :

Intel(R) Core(TM) i7 CPU 3770 @ 3.40GHz

Figure 14 shows the sample code used for this test pointing out instructions for reading from memory, the actual computations and instructions for writing to memory.

```

138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159

```

```

T a00 = pkacc::Pixe12Any<P,T>(*(&a_0 ));
T a10 = pkacc::Pixe12Any<P,T>(*(&a_1 ));
T a11 = pkacc::Pixe12Any<P,T>(*(&a_1 + 1));
T a20 = pkacc::Pixe12Any<P,T>(*(&a_2 ));
T a21 = pkacc::Pixe12Any<P,T>(*(&a_2 + 1));
T a22 = pkacc::Pixe12Any<P,T>(*(&a_2 + 2));

```

```

T 100 = std::sqrt(a00);
T 110 = a10 / 100;
T 111 = std::sqrt(a11 - (110 * 110));
T 120 = a20 / 100;
T 121 = (a21 - (120 * 110)) / 111;
T 122 = std::sqrt(a22 - (120 * 120) - (121 * 121));

```

```

*(l_0 ) = pkacc::Any2Pixe1<P,T>(100);
*(l_1 ) = pkacc::Any2Pixe1<P,T>(110);
*(l_1 + 1) = pkacc::Any2Pixe1<P,T>(111);
*(l_2 ) = pkacc::Any2Pixe1<P,T>(120);
*(l_2 + 1) = pkacc::Any2Pixe1<P,T>(121);
*(l_2 + 2) = pkacc::Any2Pixe1<P,T>(122);

```

Figure 14

The experiment has been performed considering limited test cases and platform environment, but opens up promising results and will be further investigated in the near future.

3.6 Result Accuracy

The following procedure has been used to evaluate the accuracy of the decomposition results for each of the analyzed algorithms.

Given the input symmetric positive-definite matrix **A**:

1. perform Cholesky decomposition to obtain the lower triangular matrix **L** such as

$$\mathbf{A} = \mathbf{L} * \mathbf{L}^T$$

2. Reconstruct the matrix **A** starting from the resulting **L**:

$$\mathbf{A}_{est} = \mathbf{L} * \mathbf{L}^T$$

3. Evaluate the cumulated difference between **A** and its estimation **A_{est}** as:

$$\varepsilon = \sum_{i,j=(0,0)}^{(N,N)} |A[i][j] - A_{est}[i][j]|$$

Algorithm	ϵ
LAPACK dpotrf routine [Fortran 90]	3.63234 E(-10)
Cholesky In-place [OpenCL]	3.62461 E(-10)
Cholesky-Crout [OpenCL]	2.48818 E(-10)
Cholesky-Crout [CUDA]	2.62335 E(-10)

3.7 Conclusion and Future Perspective

In this work we present concrete benchmark results obtained for different platforms, implementations and parallelization techniques of Cholesky decomposition algorithm.

Our experiments showed that parallelization on CPU seems to have a better performance than on GPU for this particular kind of algorithm, due to the intrinsic memory-intensive nature of the algorithm and memory transfer over-head. In addition, our implementation does not improve performances offered by well-established library for linear algebra problems, such as LAPACK (considering both the LAPACKE C interfaces provided by Intel MKL, and the native Fortran90 LAPACK routines).

Despite this, the following steps can be explored in the near future:

- Use NVIDIA devices with CUDA, also testing optimized mathematical libraries as cuSOLVER (on-going analysis)
- Test existing libraries (MAGMA, PLASMA)
- Explore AMD GPU-Open ROCm Platform
- Explore serial block algorithm + GPU Cholesky decomposition in each block
- Stress block mode by employing 3x3 computations and then proceed with parallelizable operations.
- Improve cache friendly-ness by implementing row packing