CloudLightning
Self-Organizing & Self-Managing Heterogeneous Cloud

Work Package 2: Use Cases and Testbed

Deliverable 2.2.1: Integrated Use Cases
Title: Integrated Use Cases

Version: 1.0

Author(s): Surya Narayanan Natarajan (INTEL), Malik M. Khan (NTNU), Anne C. Elster (NTNU), Muhammad Qasim (NTNU), Hristina Palikareva (MAXELER),

Editor(s): Perumal Kuppuudaiyar (INTEL), Leonard Feehan (INTEL) Tobias Becker (MAXELER)

Reviewed by: Konstantinos Giannoutakis (CERTH)

Date: 31st, January, 2017

Dissemination Level: Public

Document Nature: Deliverable

Status: Final
## Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Revisions</th>
<th>Editor(s)</th>
<th>Date</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0</td>
<td>1</td>
<td>Perumal Kuppuudaiyar</td>
<td>November 15, 2016</td>
<td>1. Construction of the document 2. Reorganizing document’s structures</td>
</tr>
<tr>
<td>v1.0</td>
<td>2</td>
<td>Suryanarayanan Natarajan</td>
<td>December 2, 2016</td>
<td>1. Updated Abstract, 2. Introduction, 3. Raytracing use case - Draft 1.0</td>
</tr>
<tr>
<td>v1.0</td>
<td>3</td>
<td>Malik Khan, Anne C. Elster</td>
<td>December 5, 2016</td>
<td>1. Updating the section with Oil &amp; Gas use cases with first of the drafts; 2. Reorganizing some of the structures in the section 3 for describing Oil &amp; Gas use case.</td>
</tr>
<tr>
<td>v1.0</td>
<td>4</td>
<td>Hristina Palikareva</td>
<td>December 5, 2016</td>
<td>Genomics use case Draft - 1.0</td>
</tr>
<tr>
<td>v1.0</td>
<td>5</td>
<td>Suryanarayanan Natarajan</td>
<td>December 9, 2016</td>
<td>Raytracing use case - Draft 2.0</td>
</tr>
<tr>
<td>v1.0</td>
<td>6</td>
<td>Suryanarayanan Natarajan</td>
<td>January 4, 2017</td>
<td>1. Updated Relationship to other work, 2. Executive Summary, 3. Introduction, 4. Raytracing use case - Draft 3.0</td>
</tr>
<tr>
<td>v1.0</td>
<td>7</td>
<td>Suryanarayanan Natarajan</td>
<td>January 12, 2017</td>
<td>1. Abbreviation, 2. Bibliography, 3. Conclusion, 4. Raytracing use case - Draft 4.0</td>
</tr>
<tr>
<td>v1.0</td>
<td>8</td>
<td>Muhammad Qasim, Anne. C. Elster</td>
<td>January 21, 2017</td>
<td>1. Oil and Gas use case, 2. Self-Optimized libraries - Draft 4.0</td>
</tr>
<tr>
<td>v1.0</td>
<td>9</td>
<td>Hristina Palikareva</td>
<td>January 23, 2017</td>
<td>1. Genomics use case - Draft 4.0</td>
</tr>
<tr>
<td>v2.0</td>
<td>1</td>
<td>Suryanarayanan Natarajan</td>
<td>January 24, 2017</td>
<td>Spell checked and ready for review - Draft 1.0</td>
</tr>
</tbody>
</table>
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CL</td>
<td>CloudLightning</td>
</tr>
<tr>
<td>CSP</td>
<td>Cloud Service Provider</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DB</td>
<td>Data Base</td>
</tr>
<tr>
<td>DC</td>
<td>Data Center</td>
</tr>
<tr>
<td>DFE</td>
<td>Data Flow Engine</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic Acid</td>
</tr>
<tr>
<td>EAD</td>
<td>Enterprise Application Developer</td>
</tr>
<tr>
<td>EAO</td>
<td>Enterprise Application Operator</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>ID</td>
<td>Identity Document</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>MB</td>
<td>Megabyte</td>
</tr>
<tr>
<td>MIC</td>
<td>Many Integrated Core</td>
</tr>
<tr>
<td>OASIS</td>
<td>Organization for the Advancement of Structured Information Standards</td>
</tr>
<tr>
<td>OPM</td>
<td>Open Porous Media</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial Differential Equation</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SLA</td>
<td>Service-level Agreement</td>
</tr>
<tr>
<td>SDL</td>
<td>Service Description Language</td>
</tr>
<tr>
<td>SOSM</td>
<td>Self-organization Self-management</td>
</tr>
<tr>
<td>TOSCA</td>
<td>Topology and Orchestration Specification for Cloud Applications</td>
</tr>
<tr>
<td>UCC</td>
<td>University College Cork</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locater</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WP</td>
<td>Work Package</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Mark-up Language</td>
</tr>
<tr>
<td>YAML</td>
<td>Yet Another Mark-up Language</td>
</tr>
</tbody>
</table>
Relationship to Other Tasks

The Integrated Use Cases task is directly impacted by the following CL Tasks, each of which provides input to this deliverable.

- **Task T2.1 (Requirements, Opportunities and evaluation criteria):** T2.1 investigated on the proposed application domains and narrowed down the use cases to fully specified services. It analyzed the target applications to identify requirements they will place on the cloud environment.

- **Task T6.1 (Characterisation):** T6.1 investigated and characterized the proposed hardware accelerators that are used in the project to understand the maximum potential of the heterogeneous hardware and determines how well applications will adapt to them while enabling intelligent placement decisions of the service.

- **Task T5.1 (Application Description Format):** T5.1 performed state of the art analysis of Service Description Language (SDL) to create the blue-print of the services and developed a customized SDL for the heterogeneous resources used in CL.

The Integrated Use Cases task will directly impact the following CL Tasks which will serve as output of this deliverable.

- **Task T3.2 (Integration):** T3.2 will supervise the execution of the integration plan by all technical WPs in order to ensure that all of the developed components will converge to form the final solution. With respect to that, T2.2 will ensure that the workloads are ready to be deployed.

- **Task T3.3 (Evaluation Report):** T2.2 will provide lessons learned during the implementation of the use cases and the results obtained during the execution of the use cases on the testbed, to finally validate the simulation results from T7.2
Delivery model integration, T2.2 will also provide the deployment and testing strategies of all the services when executed on the testbed across CL resources.

Fig. 1 CloudLighting Deliverables Roadmap

The overview of CloudLighting Deliverables Roadmap is shown in Fig. 1.
Executive Summary

This deliverable captures the steps followed in converting the service specifications developed in D2.1.1 into completely automated deployable services that can be readily deployed in the CL heterogeneous resource pool. The previous task, identified four application use cases 1. Genomics, 2. Oil & Gas exploration, 3. Raytracing and 4. Self-optimized libraries based on the market analysis and the prior background of the contributing partners. The task has two primary goals. The first goal is to dive deeper into the implementation details of the above mentioned applications as a standalone software that can be executed in the Bare-Metal servers, Virtual Machines and the Linux Containers if applicable. The second goal is to integrate with the Service Description Language (SDL) developed in D5.1.1 which serves as the application blueprint that can be readily deployed in the testbed system. As a result, with the input from other tasks, we capture all the software dependencies required for successful working of all the chosen use cases and demonstrate the end to end functioning of the ray-tracing use case in the cloud environment (Intel testbed) where the end-user can dynamically deploy and perform rendering remotely.
# Table of contents

Revision History

Abbreviations

Relationship to Other Tasks

Executive Summary

List of figures

List of tables

1 Introduction

2 Use Case 1: Genomics Sequence Alignment

2.1 Introduction .................................................. 3

2.1.1 Challenges ................................................. 4

2.1.2 Relevance of the Genomics Sequence Alignment Application to CloudLightning ........................................ 5

2.2 Service Specification and Requirements .......................... 6

2.2.1 Application-Specific Requirements .......................... 6

2.2.2 Software Requirements ...................................... 7

2.2.3 Hardware Requirements ..................................... 8

2.3 Implementation of Standalone Software Application .......... 10
3 Use Case 2: Oil and Gas Exploration

3.1 Introduction .............................................. 26
   3.1.1 OPM and Upscaling ................................. 27

3.2 Service Specification and Requirements ................................. 29
   3.2.1 Software Requirements ................................ 29
   3.2.2 Hardware Requirements ............................... 30

3.3 Implementation of Standalone Software Application .................. 31
   3.3.1 Implementation - 1 (DUNE-based) .................. 31
   3.3.2 Implementation - 2 (PETSc-based) .................. 38

3.4 Configuring, Deploying and Testing in NTNU .......................... 45
   3.4.1 DUNE-based OPM Upscaling Application (CPU-Execution) .. 45
   3.4.2 PETSc-based OPM Upscaling Application (CPU-GPU-Execution) 49

3.5 Integration of Service Description Language .......................... 50
   3.5.1 Application Blueprint generation and validation .......... 50

4 Use Case 3: Ray Tracing ........................................... 54

4.1 Introduction ........................................... 54

4.2 Service Specification and Requirements ................................ 56
   4.2.1 Application specific requirements .................... 56
   4.2.2 Software requirements ................................ 57
   4.2.3 Hardware requirements ................................ 58

4.3 Implementation of Standalone Software Application .................. 58
# Table of contents

4.3.1 Implementation - 1 (Bare-metal version) .......................... 58
4.3.2 Implementation - 2 (Containerized version) ...................... 59
4.4 Configuring, Deploying and Testing in Intel Testbed ............... 64
  4.4.1 Intel Testbed .......................................................... 65
  4.4.2 Deployment and Testing in Intel Testbed ......................... 66
4.5 End to End use case demonstration ................................. 70
  4.5.1 Use Case scenario ...................................................... 70
  4.5.2 Use Case Flow .......................................................... 71
  4.5.3 Design and Implementation of Ray Tracing Service Web UI .... 72
  4.5.4 Demonstration ........................................................... 73
4.6 Integration of Service Description Language ........................ 75
  4.6.1 Application Blueprint generation and validation ............... 75

5 Use Case 4: Self-Optimized Libraries (BLAS and FFTs) ............... 80
  5.1 Introduction to Self-optimised Libraries .......................... 80
    5.1.1 Service Specification and Requirements ...................... 81
    5.1.2 Implementation of stand-alone software applications for BLAS
        and FFTs ................................................................. 83
    5.1.3 Deploying and testing using development systems ............ 83

6 Summary and Future Work ................................................. 85

Bibliography ........................................................................ 86

Appendix A ......................................................................... 88

Appendix B ......................................................................... 97
  B.1 Repository URLs ............................................................ 97
  B.2 Use Case 1 - Genomics software dependencies .................. 97
  B.3 Use Case 2 - Oil & Gas software dependencies .................. 97
B.4 Use Case 3 - Raytracing software dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
List of figures

1  CloudLighting Deliverables Roadmap .................................. vii
2.1 Structure of a Maxeler MAX4 DFE. ...................................... 9
2.2 A Maxeler MPC-X node connected to a CPU server through an Infini- 
    band network. .............................................................. 10
3.1 A 3D cartesian grid chosen to capture the reservoir geometry and test 
    the OPM Upscaling application. ........................................ 28
3.2 Upscaling Application Output ........................................... 46
3.3 X & Y-axis permeabilities of Oil and Water. .......................... 47
3.4 Z-axis permeabilities of Oil and Water. ............................... 47
4.1 Example: Raw material color & texture ............................... 56
4.2 Template to define color and texture of Darkstone. .................. 57
4.3 Software and hardware layers of traditional HPC .................... 60
4.4 Modified software stack proposed for containerized HPC .......... 61
4.5 Intel Testbed Physical Layout ......................................... 65
4.6 Topology, software and hardware stack of the the raytracing engine with 
    the frontend web interface ............................................ 66
4.7 Ray Tracing use case standalone testing process flow ................ 72
4.8 Architecture of embree ray tracing web app ........................ 73
4.9 The rendering job creation form of the web interface. ............ 74
4.10 Job status page ............................................................ 75
4.11 Page showing result of pre-rendering ................................ 76
4.12 Page showing link for real-time rendering .......................... 77
List of tables

2.1 Evaluation results for the genomics sequence alignment application. . . 16
Listings

2.1 Hardware and software stack ........................................ 10
2.2 Docker container recipe .............................................. 13
3.1 Setting up support software in Ubuntu 14.04 .......................... 33
3.2 Installing g++ and gfortran in Ununtu 14.04 ...................... 33
3.3 Setting up support Libraries ........................................... 33
3.4 Setting up OPM modules ............................................. 36
3.5 Setting up OS and NVIDIA-specific settings in container ........... 40
3.6 Setting up compilers and git ........................................... 40
3.7 Setting up OPM packages ............................................ 42
3.8 Setting up datasets .................................................. 44
3.9 Run Upscaling application(standalone) ............................... 45
3.10 Shell Script for the upsampling application ........................ 48
3.11 Build Docker container for Upscaling application ................... 49
3.12 Run Docker container for Upscaling application .................... 49
3.13 Pull and Run Docker container for Upscaling application ......... 49
3.14 Build Nvidia-Docker container for Upscaling application .......... 50
3.15 Run Nvidia-Docker container for Upscaling application .......... 50
3.16 Pull and Run Nvidia-Docker container for Upscaling application ...... 50
3.17 Blueprint for the upsampling of relative permeability application . 51
4.1 Hardware and software stack ......................................... 58
4.2 Dockerfile for building admin container ............................. 61
4.3 Dockerfile for building raytracing container ........................ 62
4.4 Json Script to deploy ray tracing Engine ............................. 67
4.5 Json script to deploy Ray Tracing webservice ......................... 69
4.6 Blueprint created for end to end demo of Raytracing use case ....... 77
Chapter 1

Introduction

The High Performance computing (HPC) applications requires several tool-chains such as provisioning tools, I/O clients, resource management, development tools, and a variety of scientific libraries other than the original application itself. This makes building HPC application from scratch and running them at scale a very challenging task for an ordinary HPC application end-user. A HPC expert with system administration skills is needed to manage these complex application lifecycle management task in a beowulf cluster. In addition, the diversity of heterogeneous HPC resources available in the market complicates the entire process with further requirement to manage the hardware related issues such host kernel dependencies and vendor specific device drivers, etc., in order to successfully deploy the applications. In CloudLighting, special hardware devices such as GPU, MIC and DFE are used to accelerate the performance of our use case application, for bringing these accelerators live in the compute cluster requires substantial amount of integration effort from both software and hardware perspective.

In the cloud environment, most of the application are well packaged and are completely automated. This allows the end-user can spin up their application in the cloud and use it at the earliest without any in-depth knowledge of building and porting the application. However, this is more complex when it comes to HPC applications as they have very tightly packed software stack and need expert level knowledge to build and bring the environment alive as mentioned earlier. Therefore, portability of these applications becomes the main concern that can be relieved by using either VM or container based solutions to support the application portability across multiple hardware in the Data center.
In the CloudLightning project, one of the main objectives is to isolate the role of each player involved in the system, i.e. End User, Blueprint Creator and Enterprise Cloud Operator by clearly defining their roles and creating a service-oriented architecture for the emerging heterogeneous cloud. Hence, the Blueprint Creator forms the workflow and stores the Blueprint in the blueprint and application catalogue, the Operator selects a blueprint from the blueprint catalogue and optionally edits its constraints and parameters. The Operator launches the Blueprint by: (1) requesting an appropriate solution from the CL and (2) deploying the Blueprint on the resources returned as part of that solution. The End User then interacts with the deployed blueprint to run the application and the CL system will compute the cost of utilization.

The deliverable D2.1.1 narrowed down the chosen application domains to fully specified services (DNA sequence alignment in Genomics, upscaling application from OPM in Oil &Gas Exploration, rendering application in Raytracing) and analyzed the target applications to identify requirements they will place on the cloud environment. In this deliverable, we explore the software stack requirements of these applications in detail and ways of provisioning the entire HPC system at different levels like bare-metal, packaging it in VM or containers for easy and ready deployment from the application catalogue, and also, discuss about the usability and tools required to do them for the above mentioned 3 use cases. The fourth use case, Self-Optimized Libraries, focuses on compute-intensive functions that are used in a lot of scientific applications like BLAS (Basic Linear Subprograms) and FFTs (Fast Fourier Transform functions), are typically available as efficient auto-tuned libraries with the implementation targeting multi-core CPUs/ CPU clusters/ GPUs/ MICs/ DFEs. For this deliverable, we do not, for now, provide implementation details but capture the main features of these libraries as they are part of bigger applications. We further investigate and use the conclusions of D5.1.1 Service Description Language to create blueprints of the 3 use case applications that will enable the end to end demonstration of the CL system.

The remainder of this document is organized as follows. In Section 2, we discuss about the requirements, implementation and integration with SDL for genomics use case. In Section 3, we discuss about the requirements, implementation and integration with SDL for Oil and Gas Exploration use case. In Section 4, we discuss about the requirements, implementation and integration with SDL for raytracing use case. In Section 5, we discuss about the fourth use-case, i.e. self-optimizing libraries. Finally, we summarize and discuss the future work in Section 6.
Chapter 2

Use Case 1: Genomics Sequence Alignment

In this section we introduce an implementation of Myers DNA sequence alignment algorithm which illustrates the computational challenges inherent in genomics applications. We present the motivation for selecting this application, summarize its requirements, describe technical details related to the standalone application and its dependencies, and discuss the challenges associated with integrating the application with the Service Description Language and preparing it for deployment in the CloudLightning infrastructure.

2.1 Introduction

Genetic information is encoded in a sequence of nucleotides that are chained together to form long strands. In DNA, two complementary strands are fold together to form the so-called base pairs. Human DNA contains approximately 3 billion base pairs and is organized into 23 pairs of chromosomes with each chromosome containing hundreds to thousands of genes, yielding approximately 35 thousand genes in total.

Genomics is a discipline of genetics that studies DNA information by means of DNA sequencing and various computational methods in order to analyse the structure and function of genes. DNA analysis is progressively becoming more and more beneficial to personalised medical treatments and general medical research with healthcare providers, medical institutions and pharmaceutical companies employing it in their quest for treating, managing and preventing diseases, developing new drugs, etc.
Genomics typically involves several processes organized in a pipeline: (i) DNA sequencing, (ii) sequence alignment, (iii) sequence annotation, and (iv) genetic analyses of various types.

*DNA sequencing* is a laboratory process for determining the order of nucleotides in a strand of DNA. Since DNA cannot be read in long continuous sequences, it must be first broken up into smaller strands of 100 to 1000 base pairs. These strands can be sequenced, resulting in reads, and those reads must be subsequently reassembled into the overall sequence. Traditionally, very short reads of 100 base pairs were typically produced by DNA sequencing devices, but modern equipment tends to produce longer reads. Sequences with 1000 base pairs are currently possible albeit of lower quality.

During *sequence alignment*, shorter reads of DNA are assembled back into longer sequences. This assembly involves taking multiple overlapping reads and mapping them to locations with regard to a known reference genome. The reference genome is representative of a species, and hence the mapping must take into account genetic variations and mutations as well as sequencing errors.

The resulting DNA sequence must be subsequently *annotated* before further genetic analyses can be performed. This process involves attaching biologically-relevant information to elements of the genome, e.g. identifying markers from which clinically-relevant information can be derived. Subsequent *genetic analyses* then provide biological or medical meaning of the genetic information.

### 2.1.1 Challenges

Historically, DNA sequencing has consumed the largest fraction of operational resources and costs. However, a recent dramatic fall in the cost of sequencing equipment entails that the computational cost and effort is progressively becoming a more dominant part in the genomics domain. Therefore, optimizing runtime performance and increasing computational and energy efficiency is critical for both lowering operational cost as well as for improving the throughput of the entire genomics pipeline. This will open the door for a plethora of new sophisticated genomics analyses to take place.

However, there are still many substantial computational challenges associated with various aspects of genomics applications. For example, sequence alignment of a 3 billion base-pair human genome split into short reads of 100 bases at 30× coverage requires the alignment of a total of 1 billion reads. The resulting computation can
take hundreds of hours on a single CPU core, or tens of hours on a modern multi-core processor. Moreover, subsequent genetic analyses can be significantly harder.

Further challenges arise when the genetic data of thousands or millions of individuals needs to be analysed and correlated. For example, a personalized cancer prevention or treatment plan could involve cross-referencing thousands of genes to tens of thousands reference samples which significantly increases the complexity of the task.

The genomics domain also will inevitably pose a severe Big Data challenge in the future when large amounts of raw and processed genomical data get collected and shared by healthcare professionals. The current genome repositories are already growing exponentially in size and the data storage demands are projected to be in the order of Exabytes by 2025. This will create a true Big Data challenge if any type of analytics needs to be performed with reference to the entire repository or even a representative part of it.

2.1.2 Relevance of the Genomics Sequence Alignment Application to CloudLightning

As illustrated in the previous section, DNA sequence alignment, which is a prerequisite for all types of genetic analyses to take place, is a highly computationally-intensive process in its own right. As part of CloudLightning, we implemented Myers bit-vector sequence alignment algorithm, which was described in detail in D2.1.1. Furthermore, in order to illustrate the potential benefits of exploiting heterogeneous resources in the cloud, we profiled the application and ported the computationally-complex parts of the code to Maxeler’s dataflow engines (DFEs), obtaining significant performance speed-ups.

Myers algorithm provides opportunities for multiple levels of parallelization and is hence an excellent candidate for running in a cluster and on the cloud. In a nutshell, it processes each short DNA read separately which provides the first level of potential parallelism. For each DNA read, the algorithm reduces to approximate string matching and employs dynamic-programming techniques in order to generate a scoring matrix specifying the likelihood of the read to align to each position in the reference genome. Myers algorithm exposes a second level of parallelism as it performs the computation of the scoring matrix column-wise and processes each column independently using intricate bit-level logic operations, additions and shifts, which are, in general, carried out very efficiently on DFEs.
To exploit the available levels of parallelism, our accelerated implementation of Myers sequence alignment algorithm can run on a configurable number of CPU threads and DFEs, and furthermore can do so in a multi-tenant setting, which makes it very relevant to the SOSM ideas and concepts behind CloudLightning. Furthermore, deploying the application and storing the artifact result data on the cloud would have the advantage of alleviating the Big Data challenge associated with storing the genomics data of millions of people at a centralized place.

2.2 Service Specification and Requirements

2.2.1 Application-Specific Requirements

The application requires two files as inputs to the program. The first one contains the human reference genome encoded in either multi-FASTQ\(^1\) or a proprietary binary format. The second file contains the DNA reads of an individual where each read is annotated with a set of candidate positions in the reference genome. The latter file is generated by a less precise preprocessing algorithm and is in a proprietary textual format. The output from the application is a textual file in SAM format\(^2,3\) which stores, for each DNA read and each candidate position, the scoring matrix specifying the likelihood of the read to align to each position in a configurable neighbourhood of the candidate position.

As mentioned in the previous section, the application can run on a configurable number of CPU threads and DFEs where the multi-threaded support is based upon the OpenMP API\(^4\).

The entire set of input arguments to the application are the following:

- \(\text{-r:}\) File containing the human reference genome encoded in multi-FASTQ format, or,
- \(\text{-I:}\) File containing the human reference genome encoded in a proprietary format
- \(\text{-i:}\) File containing an individual’s DNA reads encoded in a proprietary format

\(^1\)https://en.wikipedia.org/wiki/FASTQ_format
\(^2\)http://samtools.sourceforge.net/
\(^3\)https://samtools.github.io/hts-specs/SAMv1.pdf
\(^4\)http://www.openmp.org/
2.2 Service Specification and Requirements

- **-t**: Number of CPU threads
- **-d**: Number of DFEs
- **-w**: A flag specifying that the CPU part of the application needs to write the reference genome to the DDR memory of the DFE (LMem) prior to running the analysis. This part can be optionally skipped for performance purposes if the user is confident that the data has been already stored in the memory by a previous run of the application.
- **-o**: File containing the results obtained from Myers algorithm encoded in SAM format
- **-w**: A flag specifying that the CPU part of the application needs to write the reference genome to the DDR memory of the DFE (LMem) prior to running the analysis. This part can be optionally skipped for performance purposes if the user is confident that the data has been already stored in the memory by a previous run of the application.
- **-o**: File containing the results obtained from Myers algorithm encoded in SAM format

### 2.2.2 Software Requirements

The application is written in C and compiled on Linux machines using GCC. Dependencies required to build and run the application include GNU Autoconf, the OpenMP libraries (which are included in major Linux distributions), and the general purpose compression libraries zlib and bz2.

The accelerated version of the application targets Maxeler’s DFEs and consists of two parts: a DFE part and a CPU part. The DFE part performs the compute-intensive parts of the application while the CPU part acts as a host that sets up and manages the computation on the DFE and also carries out the control-intensive tasks. The compute-intensive DFE parts are described in the MaxJ programming language which is a Java-based metaprogramming approach. The compute kernels handling the data-intensive part of the application and the associated manager, which orchestrates data movement within the DFE, are written using this language. The CPU part of the application is written in C. To modify and run the application, a user needs to have
access to Maxeler’s in-house programming environment and runtime which comprises of the following components (that currently require Linux CentOS version 6.7 or higher):

MaxCompiler  Maxeler’s custom-built general-purpose programming environment for developing data-flow applications. MaxCompiler takes as input a MaxJ program and generates a DFE implementation (a .max file) that can be called from a CPU application via the SLiC interface (see below). A .max file contains a bitstream as well as relevant metadata.

SLiC  The Simple Live CPU interface is Maxeler’s application programming interface for seamless CPU-DFE integration. SLiC allows CPU applications to configure and load a number of DFEs as well as to subsequently schedule and run actions on those DFEs using simple function calls.

MaxelerOS  A software layer and run time sitting between the SLiC interface, the Linux operating system and the hardware, which manages DFE hardware and CPU-DFE interactions in a way transparent to the user.

MaxIDE  A specialised Eclipse-based integrated development environment for MaxJ and DFE design, a fast DFE software simulator and comprehensive debug provisioning tool set used during development.

MaxCompiler, SLiC, MaxelerOS and MaxIDE have been developed by Maxeler Technologies. All tools, libraries and their extensions developed within the context of CloudLightning are not open source.

In addition, both the pure C and the accelerated version of the application are considered proprietary and will be exposed only via binaries, not source code. Therefore, to run the application, the CloudLightning infrastructure will only need the MaxelerOS component.

2.2.3 Hardware Requirements

As described in D6.1.1, Maxeler’s dataflow platforms are heterogeneous systems consisting of DFEs, CPUs, networking and storage. DFEs are in-house designed accelerators that encapsulate reconfigurable high-end FPGAs at their core and are equipped with large amounts of off-chip DDR memory. The current generation of MAX4 DFE cards
Fig. 2.1 Structure of a Maxeler MAX4 DFE.

(MAIA) combine an Altera Stratix-V FPGA with 48 GB of DDR3 DRAM and their structure is depicted in Figure 2.1.

Several system architectures are available and the overall component balance can be customised at system level to better match the requirements of the user and the application.

One possible system architecture is to add one or multiple DFE PCIe card into a Maxeler high-performance CPU server, as shown in Figure 2.1.

To achieve better scalability for large-scale platforms, Maxeler has developed the concept of a pure dataflow appliance where 8 DFEs are integrated into a dense 1U industry-standard chassis. This system is called an MPC-X node (see Figure 2.2) and it can be connected to one or multiple conventional x86 CPU servers with dual Intel Xeon processors via an Infiniband network. The CPU server acts as the application host and high-performance dataflow workloads are dispatched to the MPC-X node. Inside the MPC-X, DFEs are also directly connected through MaxRing which is a local high-speed interconnect between neighbouring DFEs.

In order to be able to access and communicate with DFE cards, Infiniband and remote direct memory access (RDMA) device-specific drivers and libraries need to be installed and loaded. For the full list of requirements please refer to the Docker recipe in Section 2.3.2.

More details about Maxeler’s hardware architecture can be found in D6.1.1.
2.3 Implementation of Standalone Software Application

In this section we describe the procedures for running the genomics sequence alignment application on bare metal and from within a Docker container. The application itself provides options for running in both CPU-only mode and with DFE acceleration. In addition, we describe the recipes necessary for employing both local DFEs connected to a CPU server via PCIe, and remote DFEs hosted on an MPC-X appliance and accessible from the CPU server via Infiniband.

Both the bare-metal and the containerized versions require Linux CentOS version 6.7 or higher as well as a prebuilt dynamic library of MaxelerOS (libmaxeleros.so) version 15.2 or higher. The high-level requirements on the hardware and software stack can be summarized as follows:

Listing 2.1 Hardware and software stack

<table>
<thead>
<tr>
<th>Implementation : CPU + DFE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target CPU platform: dual Intel Xeon X5650 CPU @ 2.67GHz with 24 cores and 192GB of RAM</td>
</tr>
<tr>
<td>CPU Architecture: x86_64</td>
</tr>
<tr>
<td>CPU OperatingSystem: Linux, CentOS 6.7 or higher</td>
</tr>
<tr>
<td>Target DFE platform: local DFE cards connected via PCIe or remote MPC-X</td>
</tr>
<tr>
<td>DFE Runtime environment: MaxelerOS 15.2 or higher</td>
</tr>
<tr>
<td>Compilers required: MaxCompiler 15.2 or higher / GCC 4.4.7 or higher</td>
</tr>
</tbody>
</table>

Fig. 2.2 A Maxeler MPC-X node connected to a CPU server through an Infiniband network.
In addition, the user needs to have a prebuilt executable binary of the application called `alignReads`. The bitstream that is loaded on the DFE during the execution of the application is statically linked to this binary as is the SLiC library, and the entire communication between the CPU and the DFE is handled transparently by MaxelerOS and SLiC.

### 2.3.1 Implementation - 1 (Bare-metal version)

To recall, the CPU server side of the application acts as a host that sets up and manages the computation on the DFE. All of the following steps are performed in a **bash** console on the CPU server.

To install the required drivers and libraries (for Infiniband, RDMA, OpenMP, etc.), the user needs to run the following command:

```bash
yum install -y 
libibverbs-utils 
libibverbs-devel 
libibverbs-devel-static 
libmlx4 
libmlx5 
ibutils 
libibcm 
libibcommon 
libibmad 
libibumad 
rdma 
librdmacm-utils 
librdmacm-devel 
librdmacm 
libibumad-devel 
infiniband-diags 
perftest 
openssh-clients 
libgomp
```

---

(D2.2.1) Integrated Use Cases  Self-Organizing, Self-Managing Heterogeneous Cloud  Dissemination Level: Public
The user needs to also set several environment variables required by MaxelerOS at runtime. If \texttt{libmaxeleros.so} is located in a directory called \texttt{(libmaxeleros-parent-dir)}, then:

\begin{verbatim}
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:<libmaxeleros-parent-dir>
export LD_PRELOAD=${LD_PRELOAD}:<libmaxeleros-parent-dir>/libmaxeleros.so
\end{verbatim}

If the user targets local DFE cards connected to the CPU server via PCIe, then nothing else needs to be specified. On the other hand, if the user wants to employ remote DFE cards hosted on an MPC-X, then the user has two options. The first one is to explicitly specify the IP address of an MPC-X by setting the following environment variable:

\begin{verbatim}
export SLIC_CONF="default_engine_resource=<name-or-IP-address-of-MPC-X>"
\end{verbatim}

The second and preferred option when running multiple applications is to use the Maxeler orchestrator, which is distributed together with MaxelerOS, and let it select an MPC-X with available DFEs. In this case, as described in D6.2.1, the user needs to issue a reservation to the orchestrator specifying a name (ID) for the reservation and the desired DFE topology. Then the user needs to set the \texttt{SLIC_CONF} environment variable in the following way:

\begin{verbatim}
export SLIC_CONF="default_engine_resource=<name-of-reservation>^<IP-address-of-orchestrator>"
\end{verbatim}

After the environment variable \texttt{SLIC_CONF} is set, the user does not have to explicitly distinguish between local and remote DFEs and can then run the application as follows:

\begin{verbatim}
alignReads -I <reference-genome-file> -i <DNA-reads-file> -t 16 -d 4 -w
\end{verbatim}

The latter command runs the accelerated application using 16 threads and 4 DFEs and explicitly loads the reference genome data to the DRAM memory of each DFE prior to processing the data (for the command-line arguments please refer to Section 2.2.1). To run the application on the CPU only without employing any DFEs, simply do not specify any DFE-related parameters:

\begin{verbatim}
alignReads -I <reference-genome-file> -i <DNA-reads-file> -t 16
\end{verbatim}
In this section we focus on the steps necessary for running the genomics sequence alignment application from within a Docker container. Docker lets us package the application together with all of its dependencies into a unit ready for cloud deployment. A Docker container wraps up the application in a complete file system that contains all the required code, runtime environment, systems tools and libraries. Docker containers are lighter than virtual machines and only abstract the OS user space, while the kernel space from the host gets used. This allows us to use host hardware without device virtualization while in the same time execute the applications in a sandboxed environment.

The process of building the Docker image very closely resembles the procedure we described in the previous section. Here we provide the recipe for building a Docker image that contains the necessary libraries and the genomics application binaries. We assume that the directory from which we run Docker contains the following directories and files:

- A directory `genomics` that contains the executable binary of the application (`alignReads`) and example genome data. This directory contains two subdirectories:
  - A directory `bin` which contains the executable `alignReads`
  - A directory `data` which contains:
    * A file with example reference genome data under `/reference-genome/`
    * A file with example DNA reads data under `/pattern-reads/`

- A directory `libmaxeleleros` that contains the dynamic MaxelerOS library `libmaxeleleros.so`

- A file `Dockerfile` that specifies the recipe for creating the Docker image

The file `Dockerfile` is instantiated as follows:

```
FROM centos:centos7

ENV APP_ROOT_DIR=/usr/src/genomics

# Install necessary packages for Infiniband, OpenMP, etc.
```

Listing 2.2 Docker container recipe
2.3 Implementation of Standalone Software Application

```bash
RUN yum install -y 
    libibverbs-utils 
    libibverbs-devel 
    libibverbs-devel-static 
    libmlx4 
    libmlx5 
    ibutils 
    libibcm 
    libibcommon 
    libibmad 
    libibumad 
    rdma 
    librdrmcm-utils 
    librdrmcm-devel 
    librdrmcm 
    librdrmcm-devel 
    infiniband-diags 
    perftest 
    openssh-clients 
    libgomp

# Add libmaxeleros.so to the image
COPY ./libmaxeleros/ /usr/lib/libmaxeleros/

# Set environment variables for MaxelerOS
ENV LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/usr/lib/libmaxeleros/
ENV LD_PRELOAD /usr/lib/libmaxeleros/libmaxeleros.so

RUN mkdir -p ${APP_ROOT_DIR}
WORKDIR ${APP_ROOT_DIR}

# Copy genomics application binary
COPY ./genomics/bin ${APP_ROOT_DIR}/bin

ENTRYPOINT ["/usr/src/genomics/bin/alignReads"]
CMD ["-I", ".\data\reference-genome\H.Sapiens.s16.dat", "-i", ".\data/ 
    pattern-reads/regions.prof", 
    "-w", 
    "-d", 
    "i", 
    "-t", 
    "16"]
```
In order to build the Docker image, the user needs to run the following command (where \texttt{max-myers-app} is the repository name of the Docker image that we create):

\begin{verbatim}
docker build -t max-myers-app .
\end{verbatim}

Similarly to the bare-metal version, before running the application from within the Docker container, users need to modify the \texttt{SLIC\_CONF} environment variable if they target DFEs hosted on an MPC-X. Please refer to the previous section for details on how to do that. The user can then run the application from the container as follows:

\begin{verbatim}
docker run -i \\                  \\
    --privileged \\
    --net=host \\
    --device=/dev/infiniband/uverbs0 \\
    --device=/dev/infiniband/rdma_cm \\
    -v \$\{PWD\}/genomics/data/reference-genome:/usr/src/genomics/data/ \\
        \rightarrow reference-genome \\
    -v \$\{PWD\}/genomics/data/pattern-reads:/usr/src/genomics/data/pattern- \\
        \rightarrow reads \\
    -e "SLIC\_CONF=\$SLIC\_CONF" \\
    -t max-myers-app [ <optional-arguments-to-application> ]
\end{verbatim}

\section{2.4 Testing at Maxeler Internal Testbed}

We have tested both the bare-metal and the containerized versions of the genomics sequence alignment application at an internal test bed at Maxeler Technologies. We used a dual Intel Xeon X5650 CPU @ 2.67GHz with 24 hyperthreaded cores and 192GB of RAM (the processors were clocked at 1.6 GHz). This CPU server node is connected to a Maxeler MPC-X 2000 which contains eight of the current generation MAX4 cards (MAIA) that combine an Altera Stratix-V FPGA with 48 GB of DDR3 DRAM each. The server and the MPC-X node are connected through two Infiniband links with FDR signaling rate. The host CPU node runs Linux CentOS 6.7 and uses gcc 4.4.7. The DFE part of the application is compiled using MaxCompiler 2015.2, and consequently, the platform uses MaxelerOS 2015.2 as runtime environment. While we are still optimizing the DFE and data-transfer part of the application (it is memory bound), preliminary
evaluation results for the bare-metal version are reported in Table 2.1 and they indicate that employing DFE acceleration yields significant speed-up in runtime performance.

Table 2.1 Evaluation results for the genomics sequence alignment application.

<table>
<thead>
<tr>
<th>Number of DFEs</th>
<th>Number of CPU threads</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU only:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2474.3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>639.4</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>339.7</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>227.4</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>253.6</td>
</tr>
<tr>
<td>CPU + DFE:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>51.4</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>24.2</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>23.9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>51.9</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>22.0</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>22.0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>51.5</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>23.7</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>24.0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>55.0</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>27.0</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>27.1</td>
</tr>
</tbody>
</table>

2.5 Integration of Service Description Language

2.5.1 Application Blueprint Generation and Validation

In this section, we describe the application-specific TOSCA service templates that describe the workflow of the genomics sequence alignment application and the dependencies/relationships between its node components. We use the TOSCA YAML service template to instantiate separate application blueprints for the CPU-only version, the DFE-accelerated version operating with local DFE cards connected via PCIe, and the DFE-accelerated version employing remote DFEs hosted in an MPC-X appliance and connected to a CPU server node via Infiniband network.
tosca_definitions_version: tosca_simple_yaml_1_0_0_wd03
description: TOSCA genomics Myers sequence alignment 2016

imports:
  - "tosca-normative-types:1.0.0.wd06-SNAPSHOT"
  - "cloudlightning-root:0.1.0-SNAPSHOT"
  - "brooklyn-types-autoimport:0.10.0-SNAPSHOT"
  - "brooklyn-types:0.1.0-SNAPSHOT"

node_types:
  cloudlightning.nodes.meta.GenomicsEngine:
    derived_from: tosca.nodes.SoftwareComponent
    description: Abstract Genomics Sequence Alignment Node
    capabilities:
      cloudlightning: cloudlightning.capabilities.SoSo
    os:
      properties:
        architecture: x86_64
        type: linux
        distribution: centos
        version: 6.7
    properties:
      num_cpu_threads:
        type: integer
        default: 16
        required: false
    input_file_location_reference_genome:
      type: string
      default: '
      required: true
    input_file_location_pattern_reads:
      type: string
      default: '
      required: true
    output_file_location_result:
      type: string
      default: ''
required: false

topology_template:
  inputs:
    cpu_threads:
      type: integer
      constraints:
        - in_range: [ 1, 32 ]

node_templates:
  genomics_engine_cpu:
    derived_from: cloudlightning.nodes.meta.GenomicsEngine
    description: Genomics Sequence Alignment CPU
    capabilities:
      cloudlightning_concrete: cloudlightning.capabilities.SoSo
    host:
      properties:
        num_cpu_threads: { get_input: cpu_threads }
    interfaces:
      Standard:
        configure: scripts/configure_genomics_Myers_CPU.sh
        start: scripts/start_genomics_Myers_CPU.sh
        create:
          implementation: scripts/create_genomics_Myers_CPU.sh
        stop: scripts/stop_genomics_Myers_CPU.sh
DFE-accelerated Version with Local DFEs

tosca_definitions_version: tosca_simple_yaml_1_0_0_wd03
description: TOSCA genomics Myers sequence alignment 2016

imports:
- "tosca-normative-types:1.0.0.wd06-SNAPSHOT"
- "cloudlightning-root:0.1.0-SNAPSHOT"
- "brooklyn-types-autoimport:0.10.0-SNAPSHOT"
- "brooklyn-types:0.1.0-SNAPSHOT"

node_types:
cloudlightning.nodes.meta.GenomicsEngine:
derived_from: tosca.nodes.SoftwareComponent
description: Abstract Genomics Sequence Alignment Node
capabilities:
cloudlightning: cloudlightning.capabilities.SoSo
os:
  properties:
    architecture: x86_64
    type: linux
distribution: centos
  version: 6.7
properties:
  num_cpu_threads:
    type: integer
    default: 16
    required: false
input_file_location_reference_genome:
  type: string
  default: '
  required: true
input_file_location_pattern_reads:
  type: string
  default: '
  required: true
output_file_location_result:
  type: string
  default: '
required: false

cloudlightning.nodes.meta.GenomicsEngineDFE:
derived_from: cloudlightning.nodes.meta.GenomicsEngine
description: Abstract Genomics Sequence-Alignment Node with DFE
   \rightarrow Acceleration
properties:
 num_dfes:
   type: integer
default: 1
required: false
target_dfe_model:
   type: string
default: 'MAIA'
required: true

topology_template:
inputs:
cpu_threads:
   type: integer
description:
   constraints:
   - in_range: [ 1, 32 ]
dfes:
   type: integer
description:
   constraints:
   - valid_values: [ 1, 2, 4, 8 ]
dfe_model:
   type: string
default: 'MAIA'
constraints:
   - valid_values: ['MAIA', 'ISCA', 'VECTIS', 'CORIA']

node_templates:
genomics_engine_local_dfes:
derived_from: cloudlightning.nodes.meta.GenomicsEngineDFE:
description: Genomics Sequence Alignment CPU Host Node with Local DFE
   \rightarrow Acceleration
2.5 Integration of Service Description Language

capabilities:
  cloudlightning_concrete: cloudlightning.capabilities.SoSo
host:
  properties:
    num_cpu_threads: { get_input: cpu_threads }
    num_dfes: { get_input: dfes }
    target_dfe_model: { get_input: dfe_model }
attributes:
  # necessary for obtaining statistics to send to the telemetry system
maxeleros_json_endpoint:
  maxeleros_json_endpoint: 'maxelerosd.sock'
interfaces:
  Standard:
    configure: scripts/configure_genomics_Myers_local_DFE.sh
    start: scripts/start_genomics_Myers_local_DFE.sh
    create:
      implementation: scripts/create_genomics_Myers_local_DFE.sh
    stop: scripts/stop_genomics_Myers_local_DFE.sh
2.5 Integration of Service Description Language

DFE-accelerated Version with Remote DFEs

tosca_definitions_version: tosca_simple_yaml_1_0_0_wd03
description: TOSCA genomics Myers sequence alignment 2016

imports:
- "tosca-normative-types:1.0.0.wd06-SNAPSHOT"
- "cloudlightning-root:0.1.0-SNAPSHOT"
- "brooklyn-types-autoimport:0.10.0-SNAPSHOT"
- "brooklyn-types:0.1.0-SNAPSHOT"

node_types:
cloudlightning.nodes.meta.GenomicsEngine:
derived_from: tosca.nodes.SoftwareComponent
description: Abstract Genomics Sequence Alignment Node
capabilities:
cloudlightning: cloudlightning.capabilities.SoSo
os:
  properties:
    architecture: x86_64
    type: linux
    distribution: centos
    version: 6.7
properties:
  num_cpu_threads:
    type: integer
    default: 16
    required: false
input_file_location_reference_genome:
  type: string
  default: ''
  required: true
input_file_location_pattern_reads:
  type: string
  default: ''
  required: true
output_file_location_result:
  type: string
  default: ''
required: false

cloudlightning.nodes.meta.GenomicsEngineDFE:
derived_from: cloudlightning.nodes.meta.GenomicsEngine
description: Abstract Genomics Sequence-Alignment Node with DFE

properties:
num_dfes:
  type: integer
default: 1
required: false
target_dfe_model:
  type: string
default: 'MAIA'
required: true

cloudlightning.nodes.meta.MPCX:
derived_from: tosca.nodes.Compute
capabilities:
  cloudlightning: cloudlightning.capabilities.SoSo
data_endpoint: tosca.capabilities.Endpoint

properties:
num_dfes:
  type: integer
default: 1
required: false
target_dfe_model:
  type: string
default: 'MAIA'
required: true

topology_template:
inputs:
cpu_threads:
  type: integer
description:
  constraints:
    - in_range: [1, 32]

dfes:
2.5 Integration of Service Description Language

type: integer
description:
constraints:
  - valid_values: [1, 2, 4, 8]
dfe_model:
type: string
default: 'MAIA'
constraints:
  - valid_values: ['MAIA', 'ISCA', 'VECTIS', 'CORIA']

node_templates:
genomics_engine_remote_dfes:
derived_from: cloudlighnting.nodes.meta.GenomicsEngineDFE:
description: Genomics Sequence Alignment CPU Host Node with Remote DFE
  ➔ Acceleration
capabilities:
cloudlightning_concrete: cloudlighnting.capabilities.SoSo
host:
  properties:
    num_cpu_threads: { get_input: cpu_threads }
attributes:
  # necessary for obtaining statistics to send to the telemetry system
maxeleros_json_endpoint:
  { concat: ["pandora://", get_attribute: [remote_dfes, ip_address ➔ ], ":18516"] }
requirements:
dfe_endpoint:
  node: remote_dfes
  relationship: remote_dfe_endpoint
  capability: Endpoint
  - network_infiniband: infiniband_network

interfaces:
Standard:
  inputs:
    mpcx_ip_address: { get_attribute: [remote_dfes, ip_address] }
  configure: scripts/configure_genomics_Myers_remote_DFE.sh
  start: scripts/start_genomics_Myers_remote_DFE.sh
  create:
    implementation: scripts/create_genomics_Myers_remote_DFE.sh
stop: scripts/stop_genomics_Myers_remote_DFE.sh

remote_dfes:
  derived_from: cloudlightning.nodes.meta.MPCX
capabilities:
  cloudlightning_concrete: cloudlightning.capabilities.SoSo
  endpoint_concrete: tosca.capabilities.Endpoint
host:
  properties:
    num_dfes: { get_input: dfes }
    target_dfe_model: { get_input: dfe_model }
requirements:
  - network_infiniband: infiniband_network

infiniband_network:
  type: tosca.nodes.network.Network

relationship_templates:
  remote_dfe_endpoint:
    type: ConnectsTo
Chapter 3

Use Case 2: Oil and Gas Exploration

Another important use case for the CloudLightning project is the Upscaling application from OPM, an open framework for porous media simulations, selected to represent an oil & gas reservoir application. Such applications are, to our knowledge, not so far offered as a cloud service.

The following sections describe our motivations for this application, the specifications and requirements, and how we provided and tested several versions that were integrated with the CL system in order to provide these cases as services.

3.1 Introduction

The oil and gas industry makes extensive use of high performance computers for generating images of the Earth’s subsurface from data collected from seismic surveys as well as simulation of multi-phase flow in porous media, typically rock formations. Within the context of the oil and gas industry, improved simulations resulting in a 1% improvement in oil and gas recovery has a billion-Euro impact.

Currently, flow simulations are typically done in-house on large dedicated workstations and/or clusters without leveraging accelerators such as GPUs. The most used flow simulator in the oil&gas industry is likely Schlumberger’s ECLIPSE simulator https://www.software.slb.com/products/eclipse or home grown effort prevalent in academia.

The contribution of our work described here is thus two-fold – both showing the usefulness of heterogeneous resources for flow simulations, as well as showing how such
an application can be integrated into a cloud-based system as a service through the self-organizing and self-managing technology developed as part of CloudLightning.

### 3.1.1 OPM and Upscaling

For this use case, it was important to select an application that is both as open as possible (preferably with an open source license), but at the same time actively used in the oil & gas industry. Unfortunately, most codes used in this domain are, like Schlumberger’s ECLIPSE, indeed proprietary. However, Statoil together with SINTEF, Fraunhofer, and others, have been pushing the development of The Open Porous Media (OPM) project, which provides a set of open-source applications centered on the simulation of flow and transport of fluids in porous media. Like most newer reservoir simulation codes, the project now provides some parallelizations, but does not take advantage of accelerators such as GPU.

Upscaling is a fundamental process to describe the reservoir flow and is common to all the reservoir simulators. The process is an averaging procedure in which the static and dynamic characteristics of a fine-scale model are approximated for a coarse-scale model. A typical 3D geological models may contain a detailed description of a reservoir in order of 10 million data points, so preferably upscaling would desired to be avoided but the computational requirements may become impractical. The upscaled model is often sufficient to provide the effective properties of the reservoir. The relatively less expensive and simpler upscaling of volumetric properties such as porosity and saturation is ignored in the scope of this work since they are simply given by the bulk and pore volume weighted arithmetic average respectively. However, permeability is a nonadditive property and no such simple averaging method exists. The approximation of permeability from good to poor, depends both on the averaging procedure as well the computational capability. These facts make upscaling a very good candidate to test the efficiency of GPU accelerators.

As mentioned, ECLIPSE is a widely used reservoir simulator. The OPM applications read and write standard ECLIPSE data files making OPM a friendly alternative option for the oil & gas industry and academia for reservoir simulations. OPM, like ECLIPSE has support for several different grid types[1]. The most important grid types are the corner-point grid and the unstructured grid. The corner point grids are also know as pillar grids, since they describe a 3D geometry in terms of pillars/coordinate lines running from top to bottom of the model. This typically matches the topology of
reservoirs well. In general, the 3D cell geometry of a corner-point grid is defined by eight corners and bi-linear planes as surfaces. However, since the corner of the neighbouring grid blocks are specified independently, the grid formats is able to handle faults and gaps better than classical cartesian grids.

The OPM upscaling application uses input files that include the upscaling model file, and the data of the rock samples. A general OPM/ECLIPSE data file is comprised of eight sections (Some of the sections are optional). Following is a brief introduction of sections that are of interest for upscaling application.

- **RUNSPEC**: Includes a description of the run, such as grid size, table sizes, number of wells, which phases to include an so forth.

- **GRID**: Defines the grid dimensions and shape, shown in Figure 3.1. Additionally, it also includes petrophysics (porosity, permeability).

- **PROPS**: Fluid and rock properties (relative permeabilities, PVT table)

- **SOLUTION**: Description of how the model is to be initialized

The Upscaling application within OPM was selected as the target for our use case since it is one of the most complete and tested applications within OPM, and an application we already had some experience with from a previous project with Statoil.

The upscaling application is currently run from the command-line interface and can output the results both in a text or standard output. Once the input of the rocks
and upscaling model files are specified through command line, the implementation calculates the upscaled permeability though calls appropriate PDE (Partial Differential Equation) solver libraries.

The solvers that the Upscaling application is currently using is based on DUNE (Distributed and Unified Numerics Environment)[2]. DUNE is an open source modular toolbox for solving large systems of equations with grid-based methods especially targeting reservoir simulations and OPM. The primary goal of DUNE is to provide implementation of common solver methods based on Finite Elements (FE), Finite Volumes (FV), and also Finite Differences (FD). The DUNE library is developed primarily at Fraunhofer with oil & gas applications in mind, and have, like OPM, also been developed with support from Statoil.

Since we want to offer both a wider set of solvers as well as codes that may run on GPUs, we elected to redevelop the Upscaling codes datastructures and interfaces so that it would work with not only the DUNE solver interfaces it was originally tailored for, but so that it also could work with PETSc and other libraries that uses the PETSc interface.

Initial work was done by a previous NTNU master student, now working for Statoil on the OPM project as a pilot project during the proposal writing period of CloudLightning, but the new interface, which rely heavily on C++ templating and other features, were at the time only tested for a multicore CPUs. We have since added the GPU support, as we want to take advantage of heterogeneous resources.

3.2 Service Specification and Requirements

The upscaling application should be provided as a standalone implementation to run on bare-metal, in Virtual Machine or in containers. These implementations should be provided as separate services to the cloud system, where the requirements for these versions are different, mostly from the software stack.

3.2.1 Software Requirements

The application has certain requirements for its standard version currently in use by Statoil. We have also added a set of additional requirements related to the modified implementations at NTNU.
The application is currently available on Linux. We used **Ubuntu 14.04 LTS** for the build versions of the application. The general requirements for the standard version installed from source include **DUNE**, **superlu3**, **suitesparse**, **BOOST**, **ERT libraries** and packages like **git**, **cmake** and **util-linux**. The compilers required on the system are **g++** and **gfortran**.

Additionally, in order for the GPU implementation to work, the foremost requirement is that the relevant GPU drivers for the specific GPUs need to be present. As mentioned, our GPU implementation leverage the Portable, Extensible Toolkit for Scientific Computation (PETSc)\[^3\] solver libraries, which also provide some GPU support based on Vienna OpenCL. To support the software stack for the upscaling application, one would thus need the PETSc and ViennaCL\[^4\] libraries with OpenCL for the GPUs available. For NVIDIA GPUs, the OpenCL support is provided with the drivers, so the needed files are already installed.

For both application versions (Dune-based CPU only and PETSc-based CPU-GPU version), the containerized solutions developed are docker based. Docker engine\[^5\], version 1.12.3 has been used to build the containerized solutions for CPU-based application version. For the CPU-GPU version, nvidia’s docker solution nvidia-docker version 1.0.0 RC 3 is used which has support for docker engine version 1.12.

### 3.2.2 Hardware Requirements

As mentioned, the upscaling application is part of the OPM project and thus the Hardware requirements are general to all applications in the project. For the CPU version, the application depends on CPUs, either Intel or AMD supporting at least SSE instructions. After we developed the integration with PETSc, the application may now also utilize NVIDIA or AMD GPUs with OpenCL support. These GPUs are accelerators to the host CPUs and connected via PCIe for communication. Specific version of GPU device drivers (made available by the vendor) matching the given GPU hardware, will be needed to run on the host servers in order to make the GPUs usable. In our case, since our test bed used NVIDIA GPUs, we used the device drivers for those specific models of GPU devices provided by the vendor.

NVIDIA provides an online tool to select the best suited device drivers, given the operating system and the actual GPU device model. It has been found important to have the correct device driver installed, not only just for employing GPUs for efficient application execution, but also for management and monitoring of the co-processing...
resource itself. If the correct version of the device drivers are not installed on the system for a particular GPU, it might happen that some of the monitoring tools may not be able to report metrics that are critical to CL system.

3.3 Implementation of Standalone Software Application

The Upscaling application requires a set of inputs which have been kept constant in number and which are identical for both the standard version and the version with GPU support during testing. The current interface of the application uses the command-line. The application is part of the opm-upscaling package from the OPM project.

The following sub-sections provides details about the various implementations we provide, which use either DUNE or PETSc as the PDE solvers, and thus the specific routine titles are either DUNE-based or PETSc-based. The implementation details also include the details of how to run the application in different runtime environments.

3.3.1 Implementation - 1 (DUNE-based)

The current standard implementation of the OPM project applications uses the DUNE library for solving the partial-differential equations (PDE’s) with grid-based methods. DUNE separates data structures from the algorithms and relies heavily on generic programming techniques and static polymorphism to achieve high-performance and flexible codes. DUNE is the established numerical engine in the upstream OPM project.

Implementation Environments – Bare Metal

The standard implementation of the upscaling application is built from source, from the git repositories for the OPM project. The process of setting up the process may not be required for the users of the application, provided, it is setup on a subset of the machines forming part of the cloud resources. On a fresh operating system install, the upscaling application is dependent on several additional packages and support software. These includes:

- the compilers i.e. g++ and gfortran
- tools i.e. git, cmake, doxygen and latex
• libraries i.e. superlu, suitesparse, Boost[6], DUNE (dune geometry, dune
  gid, dune istl, dune common), BLAS, Eigen (version 3) and ERT

Getting the above packages to interact is challenging since a lot of care has to be
taken that the appropriate versions are installed. Therefore, we generally would warn
against using a bare metal implementation for applications that rely on such a complex
set of libraries and versions.

However, we were still able to test this application on a freshly installed operating
systems using bare metal. Following the setup of the operating system (we recommend
a clean install) and the aforementioned software environments, the OPM project is
setup with the OPM packages cloned from the OPM respository.

The OPM packages have to be built in a specified order. The number and order of
the packages is important w.r.t the current or archived branches of the project, given
the active development of the project. For this reason, we confine ourselves to fixed
package versions, that can be deterministic-ally built and used, for the purposes of this
project. However, the DUNE version of the use case application has also been built
and tested, at times, from the contemporary current branch of the project.

As mentioned, both an archived version and a recent master branch of the published
OPM library were made available to run on bare-metal and both are included in
documentation as separate scripts. However, we are focusing on using the archived
branch to avoid the script braking and other issues related to working with actively
updated packages.

We do not provide GPU-based versions on bare metal. However, as described
later, docker containers and especially one provided by NVIDIA offers near bare-metal
performance.

**Implementation Environment – Virtual Machine**

The DUNE-based use-case application for the virtual-machines (VM), is essentially the
same solution as the one discussed in the previous section. However, the complexity
comes from successfully setting up the VMs for the application to run on them.
Consequently, the details for the preparation of the environment in the virtual machines
is provided in a concise script, which may run after a particular VM is executed.

The pre-requisites for the script is simply a virtual machine with fresh install of, in
our tested case, Ubuntu 14.04 LTS. The script when run, takes over the responsibility
of adding the additional packages on the fresh system, to prepare it to accept the cloning and install from the OPM project repositories.

These packages include general packages like build-essential, automake, cmake, cmake-data, git and util-linux. For the compilers one would need to have g++ and gfortran. The set of same libraries as described by the previous sub-section, however the specific packages can be found in the scripts shared.

Listing 3.1 Setting up support software in Ubuntu 14.04.

```bash
#Universe Package Add Repo
sudo apt-add-repository "deb http://archive.ubuntu.com/ubuntu/$(lsb_release -sc) universe"

sudo apt-get install -y libdpkg-perl=1.17.5ubuntu5
sudo apt-get update
sudo apt-get autoremove
sudo apt-get clean

sudo apt-get install -y build-essential g++ gfortran

For the compilers one would need to have g++ and gfortran as shown in 3.2.

Listing 3.2 Installing g++ and gfortran in Ununtu 14.04

```bash
sudo apt-get install -y build-essential g++ gfortran
```

The set of same libraries as described by the previous sub-section, however the specific packages can be found in the scripts code in 3.3.

Listing 3.3 Setting up support Libraries.

```bash
#Basic libraries necessary for both DUNE and OPM
sudo apt-get update
sudo apt-get install -y automake

sudo apt-get install -y libsuperlu3-dev libsuitesparse-dev #libjsonrpc++-
  dev libboost1.54-all-dev
```
# Installing Boost

wget http://sourceforge.net/projects/boost/files/boost/1.55.0/boost_1_55_0.tar.bz2

# Extract archive

tar xvfo boost_1_55_0.tar.bz2

# Change directory

cd boost_1_55_0

# Run the script which prepares Boost's build process

sudo ./bootstrap.sh --with-libraries=atomic,chrono,context,coroutine,
   date_time,exception,filesystem,graph,graph_parallel,iostreams,locale,
   log,math,mpi,program_options,random,regex,serialization,signals,
   system,test,thread,timer,wave

# prereqs for libboostiostreams

sudo apt-get install libbz2-dev

# Compile the project

sudo ./b2 toolset=gcc cxxflags=-std=gnu++0x

# Add the Boost libraries path to the default Ubuntu library search path

cd ..

# Inform user that Boost 1.55 was successfully installed

echo "Boost 1.55 was successfully installed."

export BOOST_ROOT=~/.opm/boost_1_55_0

export BOOST_LIBRARYDIR=~/.opm/boost_1_55_0/stage/lib

export BOOST_INCLUDEDIR=~/.opm/boost_1_55_0/boost

#Ubuntu add-apt-repo dependencies

sudo apt-get install -y python-software-properties

# Necessary Backports

sudo add-apt-repository -y ppa:opm/ppa sudo apt-get update

sudo apt-get install -y libtinyxml1-dev

sudo apt-get install -y software-properties-common

sudo add-apt-repository -y ppa:opm/ppa

sudo apt-get update

sudo apt-get install -y libopenblas-dev libdune-common-dev libdune-geometry-dev libdune-grid-dev libdune-istl-dev libsuitesparse-dev

#Install necessary OpenMPI

sudo apt-get install -y software-properties-common
sudo apt-cache search opm-simulators
sudo apt-get install -y openmpi-bin
sudo apt-get install -y libopm-simulators-bin

#INSTALL ERT
if [ -d "$directory" ]
then
  echo "$directory already exists."
else
  git clone https://github.com/Ensembles/ert.git -b release/2015.04
  echo "$directory downloaded."
fi
cd ert/
directory="Build"
if [ -d "$directory" ]
then
  echo "$directory already exists."
else
  mkdir build/
  echo "$directory created."
fi
cd build/
#continue installing ERT
make
make install

##INSTALL DUNE support packages

sudo add-apt-repository ppa:opm/ppa -y
sudo apt-get update
sudo apt-get install -y libdune-cornerpoint1
sudo apt-get install -y libdune-localfunctions-dev
3.3 Implementation of Standalone Software Application

However, it is important to note that the system needs to be setup so that it includes to the universal repository for the Ubuntu 14.04 LTS, which is the chosen OS for the VMs. Other Linux versions may not work with the all the other libraries and packages.

After setting up the environment, the OPM project is installed from source with a set of core packages like

- `opm-common`,
- `opm-parser`,
- `opm-output`,
- `opm-core`,
- `opm-upscaling`.

Code in Listing 3.4 shows the way to setup the OPM modules and install them system wide.

Listing 3.4 Setting up OPM modules.

```bash
modules='opm-parser opm-material opm-core dune-cornerpoint opm-autodiff opm-porsol opm-upscaling'
echo "===============================
git clone https://github.com/OPM/opm-common.git -b release/2015.10
cd opm-common
mkdir build/
cd build/
cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE ../
make
sudo make install
cd .....
gitmodulelink='git clone https://github.com/OPM/'
gitext='.git -b release/2015.04'
for module in ${modules}; do
  if [ -d "${module}" ]; then
    echo "${directory} directory already exists."
  else
    ${gitmodulelink}${module}${gitext}
echo "inside..."
```
3.3 Implementation of Standalone Software Application

```bash
mkdir ${module}/build
cd ${module}/build
cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE ../
nice make -j 4
fi
cd ../../
done
```

Additionally, there are packages which would change depending on which archived or current branch of the project is used for the build. Some of the latest changes in the project have combined a few packages or renamed some as well. This is a serious challenge when working with large and complex software that is actively in use and is updated regularly. Lastly, the input data set for the Upscaling application is also cloned and made part of `opm-upscaling` as a separate directory.

The `upscaling_relperm` application can be run from the bin directory present inside the build directory from inside the `opm-upscaling` directory. The input dataset is present in the `UpscalingDatasets` directory at the same level. Actual command to execute the standalone application are also provided with the script, so that if desired the script itself may run the test application with the datasets provided. The results can be stored in a file or provided on the standard terminal output.

**Implementation Environments (Containerization)**

The third version of the packaged instance of the upscaling use-case application is done with preparation of a containerized version. The DUNE-based version is packaged in a Docker container which is developed through a Dockerfile. In addition to the environment setup and the setup of the use case application, is the use of docker framework. This introduces a constraint of having the hosts setup to support the docker framework.

In the Dockerfile for the application, the steps are the same as the setup of for the virtual machines (VMs). However, to run the container, the whole point is for it to be a solution that is packaged and ready to use. The idea is that the containers are pre-assembled, and thus can be downloaded and used. The compressed size of the container is about 1GB for the DUNE-based version. The `pull` command from the docker framework is used to run the container.
Once the container is executed successfully, the state of the container with root login, is the same as a virtual machine when the script runs and finishes successfully.

3.3.2 Implementation - 2 (PETSc-based)

The implementation for the integration of the PETSc library for the GPU support, was done with the 2015.04 branch of the OPM project on github. This will remain our reference point when referring to the base case for the implementation done for the GPUs.

Implementation Environments – Key Points for Setup

The standard implementation of the upscaling application which employs GPUs to enhance performance of the linear solvers involved is also built from source like the CPU-only version. However, the source is not available at the standard git repositories for the OPM project. The standard repositories are augmented with the additional files which provide the interface to the PETSc library.

The PETSc-based GPU implementation is especially sensitive to different installations and library versions that may be present on a Ubuntu-based system. It is for this reason, we have confined ourselves to only providing the GPU-version as a containerized solution. It is, however, possible to run the setup process on a freshly installed Ubuntu 14.04 operating system and have a bare-metal version as well.

The NVIDIA-docker solution provides an environment that offers near bare-metal performance and thus is an appropriate solution for the GPU-version of the application as a service. On a fresh operating system install (highly recommended), the environment that the upscaling application should have several packages and support software. These includes:

- the compilers i.e. g++ and gfortran
- tools i.e. docker engine, nvidia-docker, git, cmake, doxygen and latex
- libraries i.e. PETSc, ViennaCL, superlu, suitesparse, Boost, BLAS, Eigen (version 3), MPI and ERT
- system software i.e. GPU device drivers and OpenCL support for NVIDIA GPUs
Getting the above packages to interact is challenging since a lot of care has to be taken that the appropriate versions are installed. We therefore generally would warn against using a bare metal implementation for applications that rely on such a complex set of libraries and versions.

However, we were still able to test this application on a freshly installed operating systems using bare metal. Following the setup of the operating system (we recommend a clean install) and the aforementioned software environments, the OPM project is setup with the OPM packages cloned from the OPM repository. The initial process is similar to the DUNE-based CPU-only version.

The standard OPM packages are augmented with additional files which are hosted separately to the standard repositories. Afterwards, the specific build order has to be followed as was done with the DUNE version of the application. The number and order of the packages is important w.r.t the current or archived branches of the project, given the active development of the project. For this reason, we confine ourselves to fixed package versions, that can be deterministic-ally built and used, for the purposes of this project. It is also important for the PETSc-based version because of the in-house development of the PETSc support, which may not be available fully for different branches of the OPM project from the standard repository. PETSC-based version of the application is build and tested with the 2015.04 branch-based packages only.

Implementation Environments (Containerization)

The PETSc-based use-case application solution is provided as a containerized solution due to its sensitive interoperability nature, with OS, drivers and library versions. It is based on essentially the same steps as discussed in the previous section. However, the complexity comes from successfully setting up the environment in a container for the application to run successfully. Consequently, the details for the preparation of the environment in the container is provided in a concise script, which runs on a system supporting docker containers.

The pre-requisites for the solution is an instance of Ubuntu 14.04 LTS. Additionally, the system should be setup to support docker engine and nvidia-docker solution. The code in Listing 3.5 sets up nvidia’s OpenCL support, libraries and executable directories. The same CUDA software with same version needs to be installed at the host as well. The application can then be either run by building a docker container from a Dockerfile or by fetching and running an already built and hosted nvidia-docker container. In either
case, the containerized solution bares the responsibility of providing the additional packages for the application, to prepare it and perform the cloning, patching and install from the OPM project repositories and PETSc-specific support file repositories.

Listing 3.5 Setting up OS and NVIDIA-specific settings in container.

```
FROM ubuntu:14.04
MAINTAINER MALIK M. KHAN <malikmk@idi.ntnu.no>

ENV CUDA_VERSION 7.5
LABEL com.nvidia.cuda.version="7.5"


RUN apt-get install -y --no-install-recommends --force-yes "cuda-toolkit-7.5\n  "

RUN echo "/usr/local/cuda/lib" >> /etc/ld.so.conf.d/cuda.conf & & \n  echo "/usr/local/cuda/lib64" >> /etc/ld.so.conf.d/cuda.conf & & \n  ldconfig

ENV PATH /usr/local/cuda/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64:${LD_LIBRARY_PATH}
```

These packages include general packages like `build-essential`, `automake`, `cmake`, `cmake-data`, `git` and `util-linux`. For the compilers one would need to have `g++` and `gfortran` as shown in Listing 3.6. The set of same libraries as described by the previous sub-section, however the specific packages can be found in the scripts shared.

Listing 3.6 Setting up compilers and git.

```
#General Packages
RUN apt-get update \n```
apt-get install -y sudo \
apt-get install -y apt-utils \
apt-get install -y vim \
sudo apt-get install -y git \
sudo apt-get install -y gcc \
sudo apt-get install -y g++ \
sudo apt-get install -y gfortran \
sudo apt-get install -y libc-dev build-essential cmake cmake-data util-linux \
sudo apt-get install -y software-properties-common \
sudo apt-get update

It is important to note that the system needs to be setup so that it includes to the universal repository for the Ubuntu 14.04 LTS. Other Linux versions may not work with the all the other libraries and packages.

After setting up the environment, the OPM project is installed from source with a set of core packages. These are:

- opm-common,
- opm-parser,
- opm-core,
- opm-material,
- opm-grid,
- dune-cornerpoint,
- opm-autodiff,
- opm-porsol,
- opm-upscaling.

The opm-common is the only repository which does not have a 2015.04 branch available, and thus the next branch 2015.10 is used instead. Code in Listing 3.7 sets up the packages (including update for PETSc support).
Listing 3.7 Setting up OPM packages.

```
# Download OPM packages

WORKDIR /home/opm
RUN git clone https://github.com/OPM/opm-data.git

RUN git clone https://github.com/OPM/opm-common.git -b release/2015.10
WORKDIR /home/opm/opm-common
RUN mkdir build;
WORKDIR /home/opm/opm-common/build
RUN cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

WORKDIR /home/opm
RUN git clone https://github.com/OPM/opm-parser.git -b release/2015.04
WORKDIR /home/opm/opm-parser
RUN mkdir build;
WORKDIR /home/opm/opm-parser/build
RUN cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

WORKDIR /home/opm
RUN git clone https://github.com/OPM/opm-core.git -b release/2015.04
WORKDIR /home/opm/opm-core
RUN sudo git remote add malikmk https://github.com/malikmk/opm-core
RUN git remote -v
RUN git fetch malikmk
RUN git merge malikmk/petsc-support

#WORKDIR /home/opm/opm-core
WORKDIR /home/opm/opm-core/build
RUN cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

WORKDIR /home/opm
RUN git clone https://github.com/OPM/opm-material.git -b release/2015.04
WORKDIR /home/opm/opm-material
RUN mkdir build;
```
WORKDIR /home/opm/opm-material/build
RUN cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

WORKDIR /home/opm
RUN git clone https://github.com/OPM/dune-cornerpoint.git -b release/2015.04
WORKDIR /home/opm/dune-cornerpoint
RUN mkdir build;
WORKDIR /home/opm/dune-cornerpoint/build
RUN cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

WORKDIR /home/opm
RUN git clone https://github.com/OPM/opm-autodiff.git -b release/2015.04
WORKDIR /home/opm/opm-autodiff
RUN mkdir build;
WORKDIR /home/opm/opm-autodiff/build
RUN cmake -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

WORKDIR /home/opm/opm-core
WORKDIR /home/opm
RUN git clone https://github.com/OPM/opm-porsol.git -b release/2015.04
WORKDIR /home/opm/opm-porsol
RUN sudo git remote add malikmk https://github.com/malikmk/opm-porsol
RUN git remote -v
RUN git fetch malikmk
RUN git merge malikmk/petsc-support
RUN mkdir build;
WORKDIR /home/opm/opm-porsol/build
RUN cmake -DPETSC_ROOT=/home/opm/petsc/ -DPETSC_INCLUDE_DIR=/home/opm/petsc/arch-linux2-c-opt/include -DPETSC_LIBRARY=/home/opm/petsc/arch-linux2-c-opt/lib/libpetsc.so -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

WORKDIR /home/opm
RUN git clone https://github.com/OPM/opm-upscaling.git -b release/2015.04
WORKDIR /home/opm/opm-upscaling
RUN sudo git remote add malikmk https://github.com/malikmk/opm-upscaling

(D2.2.1) Integrated Use Cases  Self-Organizing, Self-Managing Heterogeneous Cloud  Dissemination Level: Public
RUN git remote -v
RUN git fetch malikmk
RUN git merge malikmk/petsc-support
WORKDIR /home/opm/opm-upscaling
RUN mkdir build;
WORKDIR /home/opm/opm-upscaling
WORKDIR /home/opm/opm-upscaling/build
RUN cmake -DPETSC_ROOT=/home/opm/petsc/ -DPETSC_INCLUDE_DIR=/home/opm/petsc/arch-linux2-c-opt/include -DPETSC_LIBRARY=/home/opm/petsc/arch-linux2-c-opt/lib/libpetsc.so -DUSE_MPI=ON -DCMAKE_BUILD_TYPE=RELEASE .. ;
RUN nice make -j 4; make install;

For certain packages, the PETSc-specific support files are also patched after fetching the packages from the standard repositories. These packages include:

- opm-core,
- opm-porsol,
- opm-upscaling.

Listing 3.7 shows the patching of these packages as well as installing OPM with them.

Listing 3.8 Setting up datasets.

WORKDIR /home/opm/opm-upscaling
RUN git clone https://github.com/malikmk/UpscalingDataSets.git; cd UpscalingDataSets; tar -xzvf model.grdecl.tar.gz;
WORKDIR /home/opm/opm-upscaling/UpscalingDataSets
RUN git pull;

It is worth noting that opm-data package is also cloned and provided. It includes data sets for different applications in the OPM project, which may allow some other applications in the project to be used as well. However, these applications are not part of our use case and thus are not discussed further. Lastly, the input dataset for the Upscaling application is also cloned (shown in Listing 3.8) and made part of opm-upscaling as a separate directory.

The upscaling_relperm application can be run from the bin directory present inside the build directory from inside the opm-upscaling directory. The input dataset
is present in the **UpscalingDatasets** directory at the same level. Actual command to execute the standalone application are also provided with the script, so that if desired the script itself may run the test application with the datasets provided as shown in Listing 3.9. The results can be stored in a file or provided on the standard terminal output. Listing 3.9 provides an example of how to execute the upscaling application with one of the data sets (model, rocks) provided. This example works with one model file and four files containing data about the rocks.

Listing 3.9 Run Upscaling application (standalone).
```
WORKDIR /home/opm/opm-upscaling/
#RUN Upscaling Application, standalone, using GPUs.
./build/bin/upscale_relperm --points <iterations> ./UpscalingDataSets/model.
  grdecl ./UpscalingDataSets/rock{1..4}.txt
```

**Upscaling Output:** The sample output from a single run of an upscaling use case application is shown in Figure 3.2. The application output shows the details about the input files first and then reports on the timings and other details about the execution.

Figure 3.3 shows water and oil permeabilities in X- and Y-directions vs saturation of water. Oil flows better when most of the rock content is oil and vice versa for water. Figure 3.4 shows the comparison of the flow of water and oil in Z-direction. Z-direction permeability are significantly lower than the X- and Y-directions which is not surprising since the model has many layers.

### 3.4 Configuring, Deploying and Testing in NTNU

NTNU has provisioned two clusters for its part of the test bed. They are the SMP cluster from Numascale currently running **Ubuntu 14.04 LTS** and a CPU-GPU cluster currently running an instance of Rocks cluster distribution. The configuration and testing details for the upscaling application are provided based on the target platform.

#### 3.4.1 DUNE-based OPM Upscaling Application (CPU-Execution)

The standard implementation of the upscaling application is built from source, from the git repositories for the OPM project. The process of setting up the application
### Results from upscaling relative permeability.

- **MPI version**: [Image]
- **Finished**: Wed Dec 7 17:09:56 2016
- **Hostname**: hpclab-numascale

#### Eclipse file: ./Files/model.grdecl
- **cells**: 44408
- **Pore volume**: 0.0274506
- **volume**: 0.18065
- **Porosity**: 0.272733

- **Stone 1**: ./Files/rock1.txt (41 points)
- **Stone 2**: ./Files/rock2.txt (41 points)
- **Stone 3**: ./Files/rock3.txt (41 points)
- **Stone 4**: ./Files/rock4.txt (41 points)
- **fJunctionCurve**: 4

#### Timings:
- **Tesselation**: 5.88286 secs
- **Upscaling**: 1924.17 secs (wallclock time)
- **216.316 secs pr. saturation point**
- **MPI-nodes**: 4
- **Speedup**: 3.88502, efficiency: 0.971254

#### Options used:
- **Boundary conditions**: Fixed (no-flow)
- **points**: 30
- **maxPermContrast**: 3e7
- **minPerm**: 1e-12
- **minPorc**: 0.0001
- **surfaceTension**: 11 dynes/cm
- **gravity**: 0

#### Single phase permeability
- **|Kxx| Kxy Kxz| = 38.603 0 0
- **|Kyx| Kyy Kyz| = 0 32.3875 0
- **|Kzx| Kzy Kzz| = 0 0 1.22136

---

#### Relative permeability

<table>
<thead>
<tr>
<th>Pc (Pa)</th>
<th>Sw</th>
<th>Krxx</th>
<th>Kryx</th>
<th>Krzz</th>
<th>Kxox</th>
<th>Kzyy</th>
<th>Kzzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.674e+05</td>
<td>0.3783</td>
<td>0.008</td>
<td>0.000</td>
<td>0.000</td>
<td>0.9998</td>
<td>0.9998</td>
<td>0.9999</td>
</tr>
<tr>
<td>7.520e+04</td>
<td>0.3945</td>
<td>3.612e-06</td>
<td>0.626e-06</td>
<td>1.382e-06</td>
<td>0.9784</td>
<td>0.9026</td>
<td>0.8279</td>
</tr>
<tr>
<td>3.886e+04</td>
<td>0.4187</td>
<td>7.757e-05</td>
<td>0.0001511</td>
<td>1.977e-05</td>
<td>0.9547</td>
<td>0.9822</td>
<td>0.3198</td>
</tr>
<tr>
<td>9700.0</td>
<td>0.4269</td>
<td>6.0003704</td>
<td>0.00084275</td>
<td>0.8000340</td>
<td>0.0270</td>
<td>0.0532</td>
<td>0.081155</td>
</tr>
<tr>
<td>5016.0</td>
<td>0.4401</td>
<td>0.0005408</td>
<td>0.0005141</td>
<td>0.0005170</td>
<td>0.0417</td>
<td>0.0712</td>
<td>0.001067</td>
</tr>
<tr>
<td>3524.0</td>
<td>0.4593</td>
<td>0.0000153</td>
<td>0.0001778</td>
<td>0.01438</td>
<td>0.7473</td>
<td>0.7742</td>
<td>0.001060</td>
</tr>
</tbody>
</table>

---

Fig. 3.2 Upscaling Application Output
Fig. 3.3 X & Y-axis permeabilities of Oil and Water.

Fig. 3.4 Z-axis permeabilities of Oil and Water.
begins with the operating system itself. For the application, a clean install of Ubuntu 14.04 LTS is recommended. That is because the application is tested with the said OS extensively to be included as CL use case. It may, however, be possible to build the application on subsequent releases of the Ubuntu OS.

On a fresh operating system install, the environment that the upscaling application should have several packages and support software. These includes:

- the compilers i.e. `g++` and `gfortran`
- tools i.e. `git`, `cmake`, `doxygen` and `latex`
- libraries i.e. `superlu`, `suitesparse`, `Boost`, `DUNE` (dune geometry, dune gid, dune istl, dune common), `BLAS`, `Eigen` (version 3) and `ERT`

Getting the above packages to interact is challenging since a lot of care has to be taken that the appropriate versions are installed. We therefore generally would warn against using a bare metal implementation for applications that rely on such a complex set of libraries and versions.

However, if required, the installation script installs the required software environment and subsequently the OPM packages, software stack environment for OPM, and the application itself, on a freshly-installed OS. A simple command on the command-line would suffice, as shown in Listing 3.10.

```
Listing 3.10 Shell Script for the upscaling application.
{
root@root:~$ . ./opm-upscaling-dune-install.sh
}
```

The scripts made available can both setup an archived version or a recent master branch of the published OPM library. However, we are focusing on using the archived branch to avoid the script braking and other issues related to working with actively updated packages.

**Docker-Container solution**

The DUNE-based version of the application can be either run by building a docker container from a Dockerfile or by fetching and running an already built and hosted
docker container, in installed docker engine. In either case, the containerized solution
bears the responsibility of providing the software environment for the application,
to prepare it and perform the cloning, patching and install from the OPM project
repositories and PETSc-specific support file repositories.

The code in Listing 3.11 shows how to build a container with the DUNE-based
upsampling application, if Dockerfile is available, and is present in the working directory.

Listing 3.11 Build Docker container for Upscaling application.

```
sudo docker build -t upscaling_cpu .
```

The code in Listing 3.11 will create a docker container named `upscaling_cpu` and can
be verified by the command `docker images`.

Listing 3.12 Run Docker container for Upscaling application.

```
sudo docker run -t -i upscaling_cpu /bin/bash
```

To run the container, the code in Listing 3.12 may be used. We have already prepared
the containers that can be used as is. To pull the already prepared container from the
private repository, the code in Listing 3.13 should be used.

Listing 3.13 Pull and Run Docker container for Upscaling application.

```
sudo docker run -t -i malikmk/upscaling-cpu /bin/bash
```

3.4.2 PETSc-based OPM Upscaling Application (CPU-GPU-Execution)

The PETSc-based use-case application solution is provided as a containerized solution
due to its sensitive interoperability nature, with OS, drivers and library versions. It is
based on essentially the same steps as discussed in the previous section. However, the
complexity comes from successfully setting up the environment in a container for the
application to run successfully. Consequently, the details for the preparation of the
environment in the container is provided in a concise script, which runs on a system
supporting docker containers.
The pre-requisites for the solution is an instance of Ubuntu 14.04 LTS. Additionally, the system should be setup to support docker engine and nvidia-docker solution. The application can then be either run by building a docker container from a Dockerfile or by fetching and running an already built and hosted nvidia-docker container. In either case, the containerized solution bares the responsibility of providing the additional packages for the application, to prepare it and perform the cloning, patching and install from the OPM project repositories and PETSc-specific support file repositories.

The code in Listing 3.14 shows how to build a container with the upscaling application, if Dockerfile is available and present in the working directory.

```
Listing 3.14 Build Nvidia-Docker container for Upscaling application.

```

```
sudo nvidia-docker build -t upscaling_cpugpu .
```

The code in Listing 3.14 will create a docker container named `upscaling_cpugpu` and can be verified by the command `nvidia-docker images`.

```
Listing 3.15 Run Nvidia-Docker container for Upscaling application.

```

```
sudo nvidia-docker run -t -i upscaling_cpugpu /bin/bash
```

To run the container, the code in Listing 3.15 may be used. We have already prepared the containers that can be used as is. To pull these already prepared container from the private repository, the code in Listing 3.16 should be used.

```
Listing 3.16 Pull and Run Nvidia-Docker container for Upscaling application.

```

```
sudo nvidia-docker run -t -i malikmk/vcl_mpi /bin/bash
```

### 3.5 Integration of Service Description Language

#### 3.5.1 Application Blueprint generation and validation

Service Description Language (SDL) provides a mechanism to deploy applications as services in the cloud environments. SDL developed for CloudLightning is to facilitate
application life-cycle management for the user and the resource management for the cloud service provider to ensure the distinct function for both. After careful analysis in D5.1.1, the CL-SDL specs are built on top of TOSCA \[7\] augmented with additional features for HPC. Blueprints (BP) are the manifestations of the SDL templates. The BP syntax is based on Brooklyn YAML. The Blueprint in Listing 3.17 represents the one created for the upscaling application as one of the use cases for CL.

Listing 3.17 Blueprint for the upscaling of relative permeability application.

tosca_definitions_version: tosca_simple_yaml_1_0_0_wd03
description: TOSCA Upscaling 2016
template_name: cloudlightning-upscaling2016
template_version: 0.1. 0-SNAPSHOT
template_author: MalikKhan

imports:
  - "tosca-normative-types:1.0.0.wd06~SNAPSHOT"
  - "cloudlightning-root:0.1.0~SNAPSHOT"
  - "brooklyn-types-autoimport:0.10.0~SNAPSHOT"
  - "brooklyn-types:0.1.0~SNAPSHOT"

node_types:
  cloudlightning.nodes.meta.UpscalingEngine:
    derived_from: tosca.nodes.SoftwareComponent
tags:
    icon: images/upscaling_ntnu.png
description: Abstract Upscaling Node
capabilities:
  cloudlightning: cloudlightning.capabilities.SoSo
properties:
  cloudlightning.endpoint:
    type: string
    default: ''
    required: false
  cloudlightning.endpoint.type:
    type: string
    default: ''
    required: false
cloudlightning.nodes.GPUUpscalingEngine:
derived_from: cloudlightning.nodes.meta.UpscalingEngine
tags:
    icon: images/upscaling_ntnu.png
description: OPM based upscaling of relative permeability on a CPU-GPU node
capabilities:
    cloudlightning_concrete: cloudlightning.capabilities.SoSo
attributes:
    upscaling_endpoint: { concat: [ "ssh://", get_attribute: [ HOST, public_ip_address (SOSM) ], ":8222" ] }
interfaces:
    Standard:
        start: scripts/start_gpu_upscaling.sh
        create:
            implementation: scripts/create_gpu_upscaling.sh
        stop: scripts/stop_upscaling_gpu.sh

cloudlightning.nodes.CPUUpscalingEngine:
derived_from: cloudlightning.nodes.meta.UpscalingEngine
tags:
    icon: images/upscaling_ntnu.png
description: OPM based upscaling of relative permeability on a CPU-only node
capabilities:
    cloudlightning_concrete: cloudlightning.capabilities.SoSo
attributes:
    upscaling_endpoint: { concat: [ "ssh://", get_attribute: [ HOST, public_ip_address ], ":8222" ] }
interfaces:
    Standard:
        start: scripts/start_cpu_upscaling.sh
        create:
            implementation: scripts/create_cpu_upscaling.sh
        stop: scripts/stop_cpu_upscaling.sh
3.5 Integration of Service Description Language

cloudlightning.nodes.UpscalingWebservice:
derived_from: tosca.nodes.SoftwareComponent
tags:
  icon: images/600px-Gnome-web-browser.svg.png
attributes:
  ui_endpoint: { concat: [ "http://", get_attribute: [ HOST,
                   → public_ip_address ], ":8081" ] }
cadvisor_endpoint: { concat: [ "http://", get_attribute: [ HOST,
                    → public_ip_address ], ":8080" ] }
interfaces:
  Standard:
    start: scripts/start_upscaling_webservice.sh
    create: scripts/create_upscaling_webservice.sh
    stop: scripts/stop_upscaling_webservice.sh
Chapter 4

Use Case 3: Ray Tracing

4.1 Introduction

Ray tracing comes under a broader category of image rendering that uses the laws of physics to generate more photo-realistic graphics from 2D or 3D models by means of computer programs like Monte Carlo methods. Usually the input to a ray tracer engine will have a well defined data structures as inputs like geometry or the structure of the object, viewpoint of the camera, texture of the object, lighting of the environment etc.

The ray tracing engine then renders by solving the rendering equations to produce the graphical image as output. In computer graphics, rendering is basically used by architects to conceptualize their ideas, gamers to display graphics, animation artists to create movies and many more areas where visual effects has a major role. Rendering can be classified into 2 forms: 1. Pre-rendering, where, the rendering is done in a batch mode or off-line mode typically in the animation industry and images are rendered for every scene and finally stitched into a movie, 2. Real-time rendering, where, the rendering is done in a very interactive way typically in video games.

Currently, HPC-cloud convergence is considered nascent. The concept offers interesting benefits for many stakeholders such as providers of Cloud IaaS, hardware vendors, ISV, HPC application integrator and the end users. One example of taking advantage of the cloud is related to computer games. There are companies like OnLive and Gaikai that offers gaming as a service that runs on servers in the cloud. These are embarrassingly parallel applications that will benefit from the heterogeneous hardware resources investigated in CloudLightning. The back-end server in the cloud processes
user interactions from the game client, and sends back a compressed, rendered image of the game to the user.

There are many advantages of using a cloud based rendering service for the end user, application developer and infrastructure provider. These are: 1. the end user does not need to worry about the installation and patching up of the game to use it immediately, also does not bother about the hardware requirements like high performing servers and huge disk space etc, 2. Application developer can be care-free about the piracy issues as the game cannot be copied from the client side. Also, the application developer can easily release a demo version of the new games to get the user experience, 3. Infrastructure provider can achieve proper utilization of their servers by providing easy deployment and usability to the developer.

Working of ray tracing and the algorithms for forward, backward and hybrid ray tracing are explored in D.2.1.1 (Use Case Requirements Report). For the development and deployment of a standalone implementation, we have used Embree [8] which is an open source ray tracing library that are optimized to run on Intel Xeon and many integrated core architectures such as Intel Xeon Phi with support for SSE, AVX, AVX2, and AVX512.

Embree is a collection of high-performance ray tracing kernels, developed at Intel. It mainly targets the graphics application engineers in improving the performance of their application by leveraging the optimized ray tracing kernels of Embree and the APIs exposed by the kernel. It supports runtime code selection which makes the kernel compatible to run across multiple generations of Intel processors with different instruction set. Embree is released as Open Source under the Apache 2.0 license. Already, there are couple of rendering software that use Embree as their back-end rendering kernel Eg: OSPRay [9], Corona Renderer [10], FluidRay RT [11], SimLab Composer [12], Brighter 3D [13] etc.

In this deliverable, we have investigated and documented on how to build and execute Embree in bare-metal and containerized modes across Xeon and Xeon Phi processors and in the cloud testbed built in Intel.
4.2 Service Specification and Requirements

4.2.1 Application specific requirements

Embree takes in wavefront format object file (*.OBJ) as input to perform rendering on it. OBJ is a geometry definition file format that represents 3D geometry alone. OBJ file comes along with Material Template Library file (*.MTL). MTL files carries the material properties of the objects like texture, light reflection properties, surface shades etc and are referenced internally in the .OBJ file. Few examples of the raw material color and texture are shown in Fig. 4.1 and the template to compose darkstone example is shown in Fig. 4.2. More details about the file format is explained in the website (http://www.fileformat.info/format/wavefrontobj/egff.htm). In our case, we use a free and open source 3D model creation suite called Blender [14]. Also, there are lot of websites from where the object files can be downloaded.

![Fig. 4.1 Example: Raw material color & texture](image)

Once the object file is ready, next job is to choose what kind of rendering operations has to be performed to obtain the output computer generated graphics like viewer, steam viewer, path tracer etc. All the operations have common flag options to set up the rendering requirements as mentioned below for the pre-rendering mode.

- **-vp**: Camera position in terms of x,y,z axis
4.2 Service Specification and Requirements

Snippet of material file

```
newmtl TowerStaggeredTiles
Ns 96.078431
Ks 1.000000 1.000000 1.000000
Ka 0.064176 0.053301 0.046244
Kd 0.500000 0.500000 0.500000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 2
map_Kd ppm/DarkStone.ppm
```

Fig. 4.2 Template to define color and texture of Darkstone.

- **-vi**: Camera look-at point
- **-vu**: Camera up vector
- **-fov**: Vertical field view
- **-thread**: Number of parallel threads

The navigation in the real-time rendering mode follows the camera orbit model, where the camera revolves around the current center of interest. With the left mouse button you can rotate around the center of interest (the point initially set with -vi). Holding Control pressed while clicking the left mouse button rotates the camera around its location. **-size** flag can be used to set initial window size for real-time rendering, **-fullscreen** to set the full screen mode.

### 4.2.2 Software requirements

Embree supports Windows (32 bit and 64 bit), Linux (64 bit) and Mac OS X (64 bit). The code compiles with the Intel Compiler, GCC, Clang and the Microsoft Compiler. Using the Intel Compiler improves performance by approximately 10%. Performance also varies across different operating systems, with Linux typically performing best as it supports transparently transitioning to 2MB pages. The instructions for building
and installing from source code or installation using a precompiled binaries can be found in https://embree.github.io/downloads.html. Parallelization model used: Intel® Threading Building Blocks (TBB). Additional software required to build embree: Intel parallel studio (for intel compiler), cmake, freeglut, libXmu.

4.2.3 Hardware requirements

Embree is optimized for Intel CPUs supporting SSE, AVX, AVX2, and AVX-512 instructions, and requires at least a CPU with support for SSE2. Ray tracing has been too slow to consider for real-time until recently industry started using specialized accelerators like GPUs/Xeon Phis to overcome it by exploiting the embarrassingly parallel nature of the application. In our case, we use Xeon Phi many-core processor to gain performance. Xeon Phi, Knights Corner(KNC) is a co-processor that usually (at the point of time writing this document) exists with the host processor (Xeon) connected via PCIe for communication. In order to use the card, device specific drivers needs to be loaded to bootup the card to ready state.

4.3 Implementation of Standalone Software Application

4.3.1 Implementation - 1 (Bare-metal version)

In this section, we describe the fine grained details and the complete procedure to run the application on the bare-metal with an OS. Also, we capture the software and hardware stack details for each of our implementation. The software and hardware stack we used for the standalone implementation is mentioned in Listing 4.2.

Listing 4.1 Hardware and software stack

<table>
<thead>
<tr>
<th>Implementation</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target platform</td>
<td>Intel® Xeon(R) CPU E5-2680 v2 @ 2.80GHz</td>
</tr>
<tr>
<td>Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>OperatingSystem</td>
<td>Linux, CentOS 7.0</td>
</tr>
<tr>
<td>Compiler required</td>
<td>Intel Compiler 15.0.2 / GCC 4.8.2</td>
</tr>
<tr>
<td>Software package</td>
<td>Clang 3.4.2 and Visual Studio 12 2013</td>
</tr>
</tbody>
</table>
Embree Source code or precompiled executable can be downloaded from https://embree.github.io/downloads.html. For using pre-compiled binaries follow this procedure:

```
wget https://github.com/embree/embree/releases/download/v2.8.0/embree-2.8.0.x86_64.linux.tar.gz
bash tar xzf embree-2.8.0.x86_64.linux.tar.gz
source embree-2.8.0.x86_64.linux/embree-vars.sh
```

Instruction to compile and setup Embree ray tracing environment on bare metal server (https://embree.github.io/downloads.html). Use the following commands to install the dependencies using yum:

```
sudo yum install cmake.x86_64
dsudo yum install tbb.x86_64 tbb-devel.x86_64
dsudo yum install freeglut.x86_64 freeglut-devel.x86_64
dsudo yum install libXmu.x86_64 libXi.x86_64
dsudo yum install libXmu-devel.x86_64 libXi-devel.x86_64
```

#1

Create a build directory inside the Embree root directory and execute

```
→ ccmake .. inside this build directory.
```

```
mkdir build
cd build
cmake
cmake -DCMAKE_CXX_COMPILER=icpc -DCMAKE_C_COMPILER=icc ..
```

#2

```
make
```

#1. Ensure that Intel parallel studio is installed and it works before this step. #2. Running ccmake will open a dialog where you can perform various configurations from the visible options. After having configured Embree, press c to configure and g to generate a Makefile and leave the configuration.

### 4.3.2 Implementation - 2 (Containerized version)

Container solutions are mainly investigated for this use case instead of full-fledged virtual machines (VM) due to some technical limitations posed by KVM kernels in virtualizing and accessing the Xeon Phi co-processor from VMs. However, there are
some workaround available to provision the MIC co-processors using VMs by patching [15] the virtual machine kernel which indeed considered as out of scope for this task. Moreover, Container based computing is revolutionizing [16], [17], [18] the application development and deployment in many aspects such as portability, performance and less hypervisor overhead. Hence, ray tracing use cases are built using Docker based container solution.

In this section, we provide instructions to download and deploy prebuilt Docker images for quick deployment. We first present how the containerization works in traditional HPC clusters and then explore the modification of the traditional HPC software stack that results in an simpler way of accessing the accelerators for the application developers and the end users. Fig. 4.3 shows the whole HPC stack and how they are tightly coupled. The main disadvantage with this model is, at a time only one container can access the hardware. The user or the application developer should be aware of the hardware library or device driver dependencies for example MPSS in the case of Xeon Phi. We overcame this issue by creating an admin and application container separately as shown in Fig. 4.4 which will isolate the application developer or end user from bothering about how to access the hardware.

![Fig. 4.3 Software and hardware layers of traditional HPC](image)

We have built application container to deploy Embree in Xeon and admin, application containers to deploy Embree targeting Xeon Phi coprocessors. The automated way to build an application container image is by using a Dockerfile. Dockerfile contains all the scripts to download and install the whole software stack and dependent libraries inside the container. The Docker file is basically required to build the application from scratch and is mentioned below. Also, the application and the containers are availa-
Fig. 4.4 Modified software stack proposed for containerized HPC

ble in the bitbucket repository https://bitbucket.org/cloudlightning/t2.2_raytracing_integrated_usecase/src

Admin container Dockerfile

This Dockerfile is used to build an Xeon Phi Coprocessor host image based on Centos. For Xeon Phi scenario we have built an admin containers to manage kernel dependencies, drivers and networking, etc.

Listing 4.2 Dockerfile for building admin container

FROM centos:7
MAINTAINER Perumal K "pkudaiyar@gmail.com"
ADD container-files /
RUN yum -y install openssh-server openssh-clients
RUN yum update -y && yum install -y epel-release && yum install -y iproute
  ± python-setuptools hostname inotify-tools yum-utils which && yum
  -> install -y net-tools wget pciutils bridge-utils sudo && yum clean all
  ± & easy_install supervisor
RUN mkdir /usr/local/MIC_STACKS/
COPY mpss-3.7-linux.tar.gz /
RUN tar -xvf /mpss-3.7-linux.tar.gz -C /usr/local/MIC_STACKS/
RUN mv /usr/local/MIC_STACKS/mpss-3.7-linux/mpss-3.7 /usr/local/MIC_STACKS/
RUN rm /mpss-3.7-linux.tar.gz
4.3 Implementation of Standalone Software Application

WORKDIR /usr/local/MIC_STACKS/mpss-3.7
RUN yum -y install *.rpm
RUN yum clean all
ADD sshd.conf /etc/supervisord.d/sshd.conf
ADD mpssd.conf /etc/supervisord.d/mpssd.conf
EXPOSE 22 9111
RUN echo 'root:secret' | chpasswd
RUN mkdir -p /var/run/sshd ; chmod -rx /var/run/sshd
RUN ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key
RUN sed -ri '/#PermitRootLogin yes/PermitRootLogin yes/g' /etc/ssh/sshd_config
RUN sed -ri '/UsePAM yes/UsePAM yes/g' /etc/ssh/sshd_config
RUN sed -ri '/UsePAM no/UsePAM no/g' /etc/ssh/sshd_config
RUN yum install -y xorg-x11-xauth xorg-x11-fonts-* xorg-x11-utils
RUN echo X11Forwarding yes >> /etc/ssh/sshd_config
RUN echo X11DisplayOffset 10 >> /etc/ssh/sshd_config
WORKDIR /
ADD mic_setup.sh /
RUN chmod +x mic_setup.sh
RUN chmod +x /config/bootstrap.sh
VOLUME ["/data"]
ENTRYPOINT ["/config/bootstrap.sh"]

Application container Dockerfile

Same Docker image can be used for both CPU and Phi scenarios except an additional volume mapping has to be done in the second case to passthrough PCIe for Xeon Phi from host to Docker container. Follow the below specified instruction to run example ray tracing applications using Docker container on CPU and Xeon Phi environment.

Listing 4.3 Dockerfile for building raytracing container

FROM centos:7
RUN echo 'root:secret' | chpasswd
RUN yum install -y openssh-server || true
RUN yum install -y openssh-clients
RUN yum install -y unzip
RUN yum install -y gimp
RUN yum install -y Mesa* freeglut.x86_64 freeglut-devel.x86_64 libXmu.
\[x86_64 libXi.x86_64 libXmu-devel.x86_64 tbb.x86_64 wget pciutils
\]libXi-devel.x86_64 || true
RUN yum install -y xorg-x11-xauth xorg-x11-fonts-* xorg-x11-utils
RUN mkdir /embree
WORKDIR /embree
RUN wget https://github.com/embree/embree/releases/download/v2.8.0/embree
\[-2.8.0.x86_64.rpm.tar.gz
RUN tar xzf embree-2.8.0.x86_64.rpm.tar.gz
RUN rpm --install embree-lib-2.8.0-1.x86_64.rpm
RUN rpm --install embree-devel-2.8.0-1.x86_64.rpm
RUN rpm --install embree-examples-2.8.0-1.x86_64.rpm
ADD rpm/*.rpm /embree/
RUN rpm --install libscif*.rpm
RUN rpm --install mpss-coi-*.rpm
RUN rm embree-2.8.0.x86_64.rpm.tar.gz
RUN rm *.rpm
RUN wget https://github.com/embree/embree/releases/download/v2.8.0/embree
\[-2.8.0.x86_64.linux.tar.gz
RUN tar xzf embree-2.8.0.x86_64.linux.tar.gz
RUN rm embree-2.8.0.x86_64.linux.tar.gz
RUN mkdir -p /var/run/sshd ; chmod -rx /var/run/sshd
RUN ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key
RUN sed -ri '/^PermitRootLogin/ s/yes/no/' /etc/ssh/sshd_config
RUN sed -ri '/^UsePAM/ s/yes/no/' /etc/ssh/sshd_config
RUN export SINK_LD_LIBRARY_PATH=/usr/local:$SINK_LD_LIBRARY_PATH
RUN echo X11Forwarding yes >> /etc/ssh/sshd_config
RUN echo X11DisplayOffset 10 >> /etc/ssh/sshd_config
RUN echo "source/embree/embree-2.8.0.x86_64.linux/embree-vars.sh" >> /root
\imizer.sbin/ssh" >> /root
\imizer.sbin/ssh"
EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]

On the other hand, the prebuilt application container can be downloaded from the
Instruction to deploy the embree ray tracing application manually:

**Prerequisites**

- Linux Environment with Docker engine install, version > 1.7
- Standard x86 - 64bit CPU.
- Standard display

Docker Image: Our custom build Docker Image can be downloaded from the [https://hub.docker.com/r/pkudaiyar/mic_app_embree/](https://hub.docker.com/r/pkudaiyar/mic_app_embree/) using following command

```
Docker pull pkudaiyar/mic_app_embree:1
```

Executing Docker command:

a. With Local Display:

```
Docker run -it -e DISPLAY -v HOME/.Xauthority:/root/.Xauthority -v /tmp/.
→ X11-unix:/tmp/.X11-unix --net=host pkudaiyar/mic_app_embree:1 bash
```

b. With remote client ssh X11 tunnelling: Install and start Xming X server for windows (https://sourceforge.net/projects/xming/)

At remote server run:

```
Docker run -d -it -p 9000:22 pkudaiyar/mic_app_embree:1 bash
```

Make SSH connection (ssh -p 9000 root@<Docker-ip>) to running container by enabling X11 forwarding (password: secret)

```
ssh -p 9000 root@<Docker-ip>
```

## 4.4 Configuring, Deploying and Testing in Intel Testbed

Intel has provisioned a development testbed environment equipped with CPU and MIC co-processors, the hardware architecture and software stack of this test environment are described in the deliverable D6.2.1 (Infrastructure support for handling the heterogeneous resources). Also, we have captured the end to end cloud implementation of
raytracing application with a front end webGUI using which the End User can perform rendering.

4.4.1 Intel Testbed

Intel’s CloudLightning test environment has been equipped with both CPU and Xeon Phi co-processors as shown in the Fig. 4.5. There are six bare metal Xeon servers of varied CPU configurations wired together using high speed 10Gig network switch for building a compute cluster. Among those there are two nodes which have been installed with Xeon Phi co-processors and one with NVIDIA GPU. It uses High Availability / High Throughput (HA/HT) distributed CoreOS operating system with Mesos as the resource manager.

![Fig. 4.5 Intel Testbed Physical Layout](image-url)
4.4 Configuring, Deploying and Testing in Intel Testbed

4.4.2 Deployment and Testing in Intel Testbed

The Embree rendering engine is an experimental renderer rather than a fully featured one, as such it lacks a friendly end-user gui. To fully flesh out the end to end cloud implementation of the use case a simple web based user interface was designed and implemented. The overall topology of the test environment where the application will be deployed for testing, its software stack composition and the targeted hardware in Intel testbed are shown in Figure 4.6.

Fig. 4.6 Topology, software and hardware stack of the the raytracing engine with the frontend web interface

Download and installation procedure

Scripts for the installation and configuration of Intel Embree (https://embree.github.io/) kernels are included in the Docker build scripts known as ‘Dockerfile’. These Docker files can be downloaded from the following bitbucket repository hosted by the CloudLightning project consortium. Repository URL: https://bitbucket.org/cloudlightning/t2.2_raytracing_integrated_usecase

Cloning the source code:

```bash
git clone https://<user_id>@bitbucket.org/cloudlightning/t2.2_raytracing_integrated_usecase.git
```
There are two Docker project folders inside this project. They are:

1. **Mic_app_embree_2**: This folder contains a Dockerfile and several RPM files consisting of libraries and drivers needed for Xeon Phi co-processor card. Embree ray tracer container image can be built using standard Docker build command as shown below.

   Navigate to mic_app_embree_folder_2 and enter the following commands.

   ```bash
   Docker build -t <repository>/<image_name>:<version>
   ```

   Alternatively, users can also download the pre-built container images available in the Dockerhub repository using the following command.

   ```bash
   Docker pull pkudaiyar/mic_app_embree:1
   ```

2. **Ray_tracing_web_app**: This folder consists of source codes and a Dockerfile needed for building a ray tracing web application (front end) container. Navigate to this folder and run the following Docker container from the command

   ```bash
   Docker build -t <repository>/<image_name>:<version>
   ```

   Alternatively, users can also download the prebuilt container images available in the Dockerhub repository using the following command.

   ```bash
   Docker pull pkudaiyar/embree_raytracer_webgui:2
   ```

**Deployment procedure**

Marathon framework provides web UI and also REST API with JSON scripts to deploy the application on the MESOS managed infrastructure. We used REST API method which requires JSON scripts in order to deploy the Ray Tracing application which comprises of Ray Tracing engine (Listing 4.4) and the Ray Tracing web GUI (Listing 4.5). Examples of these scripts as shown in below

Listing 4.4 Json Script to deploy ray tracing Engine

```json
{
  "id": "/cloudlightning/rtengine0001",
  "cpus": 0.1,
}
```
"mem": 512,
"constraints": [
    
    "mic",
"CLUSTER",
"available"
]

"container": {
    "type": "Docker",
    "Docker": {
        "image": "pkudaiyar/ray_tracing_app_embree:1",
        "privileged": true,
        "network": "BRIDGE",
        "portMappings": [
            
            "containerPort": 22,
            "hostPort": 0,
            "servicePort": 10022,
            "protocol": "tcp"
        ]
    },
    "parameters": [
        
        "key": "hostname",
        "value": "embree_renderer.weave.local"
    ]
}

"volumes": [
    
    "containerPath": "/sys/class/mic/mic0",
    "hostPath": "/sys/class/mic/mic0",
    "mode": "RO"
]


Listing 4.5 Json script to deploy Ray Tracing webservice

```json
{
    "id": "/cloudlightning/rtwebservice0001",
    "cpus": 0.1,
    "mem": 512,
    "constraints": [
        ["hostname",
        "CLUSTER",
        "10.2.0.34"
        ],
    ],
    "cmd": "service mongodb restart & /usr/bin/python /src/webserver.py & /usr/local/bin/npm start",
    "container": {
        "type": "Docker",
        "Docker": {
            "image": "pkudaiyar/embree_raytracer_webgui:1",
            "forcePullImage": true,
            "network": "BRIDGE",
            "portMappings": [
                {
                    "containerPort": 9393,
                    "hostPort": 0,
                    "servicePort": 19393,
                    "protocol": "tcp"
                },
                {
                    "containerPort": 3005,
                    "hostPort": 0,
                    "servicePort": 19393,
                    "protocol": "tcp"
                }
            ]
        }
    }
}
```
4.5 End to End use case demonstration

4.5.1 Use Case scenario

The user can enter a rendering job configuration via a web form. A job (rendering job source file) consists of an .OBJ format file (including any relevant material file and...
images) compressed as a zip archive which is uploaded via the web-form, or provided via a valid web url. In addition to this, details such as camera position, camera target, output frame size, lighting and job type are specified. Job type can be either pre-rendering (interactive) or real-time rendering (batch). Interactive jobs when submitted will generate a SSH command that the user can copy-paste to execute and view the job in interactive mode. A batch job will produce a rendered image, or series of images based on the .OBJ file and the submitted parameters. The batch job may be a long running task, for example when rendering a series of images to produce an animation, in this case the images may only be downloaded when the job is completed. When a job or jobs have been submitted, the user can keep track of all jobs by viewing the job management page. This provides a list of all submitted jobs and their status (running or completed). For interactive jobs the user can access the ssh command to view the job that was originally submitted. For batch jobs, if completed, the user can choose to download the image or images as a zip archive. From the job management page, the user can also delete jobs that are no longer required. Front-end web pages for submitting jobs, monitoring the progress and getting back the results are shown in the Appendix A.

4.5.2 Use Case Flow

Fig. 4.7 illustrates the deployment process flow for the end to end demonstration of the raytracing use case in the Intel testbed starting from

1. Posting both services to marathon i.e submitting the Raytracing JSON and webUI JSON using REST API.

2. Marathon/Mesos framework schedules and places both the containers in the available resource.

3. Service instances are created on the allocated resource and IP address of the containers are retrieved for further access.

4. Access ray tracing service end point and SSH X11 connection for remote display for interactive ray tracing.
4.5 End to End use case demonstration

4.5.3 Design and Implementation of Ray Tracing Service Web UI

The system comprises two major components as shown in Fig. 4.8. The Embree container and the Web UI container. Embree container mainly contains the ray tracing engine and the supporting libraries needed to communicate with Xeon Phi. The user interface web application is implemented in Node.js. User actions are communicated through a Python based communication service using REST based URLs. The Python web service reads and writes job data from the local MongoDB database and also communicates with the Embree container using SSH for command execution and SCP for job-file and image transfer.

When a job is submitted, the zipped obj file is temporarily uploaded inside the UI container, the communication service sends the file to the Embree container via SCP, it then remotely executes SSH commands to unzip the job file. For interactive rendering the location of the file is returned to the UI container, which then uses location to generate the interactive rendering command for the user. For a batch job, the uploaded file is sent to the Embree binary file along with the requested job parameters and it produces the image or images. The images are output to the same location as the uploaded obj file within the Embree container. When the user attempts to download the job images, the images are zipped and copied via SCP back to the UI container and are downloaded by the users browser.
These containers can be deployed in separate locations, by a deployment script or blueprint which must notify the Web UI container of the IP Address and SSH port of the Embree container once it is successfully initialized. The Web UI subsequently communicates with the Embree container via the SSH protocol.

Fig. 4.8 Architecture of embree ray tracing web app

4.5.4 Demonstration

Architecture mentioned above was used in the back end and in this section we present the screenshots of the end to end demonstration. Once the containers are deployed and executed, we can access the web UI to create a new rendering job. Figure 4.9 shows the web UI where the ray tracing parameters can be filled to initiate the job.
Fig. 4.9 The rendering job creation form of the web interface.

Fig. 4.10 shows job status page where user has to select the nature of the rendering job in the drop down on the left. Selected job details to the right. Image shows an in progress batch job with status 3/14 indicating it has completed rendering of 3 out of 14 frames in the job. The download and mpeg generation buttons are disabled until all job frames are completed. A preview image has not yet been generated for this job.

Once the job has completed rendering, they can be downloaded by the end user. Fig. 4.11 shows a completed batch job, the preview image has been generated and an mpeg comprised of all the batch frames is being generated and downloaded.

Suppose if the user chose interactive mode i.e. real-time rendering, a job command dialog with the generated ssh command to connect interactively to the render job will appear as shown in Figure 4.12 and the user can copy paste it to his terminal to continue with the process.
4.6 Integration of Service Description Language

4.6.1 Application Blueprint generation and validation

Service Description Language (SDL) defines a template to specify and to compose services into applications for deployment on cloud environments. They include service descriptions, deployment specifications, and the specific resources required to run each service. Moreover, in CloudLightning the details that a SDL should capture will be immense as it targets a heterogeneous infrastructure. SDL developed for CloudLightning
should facilitate application lifecycle management for the user and the resource management for the cloud service provider to ensure the distinct function for both. Two SDL standards were explored: 1. Cloud Application Management for Platforms (CAMP), 2. Topology and Orchestration Specification for Cloud Applications (TOSCA). Both of these SDLs are well analyzed in the D5.1.1 deliverable and finally CL-SDL specifications are built on top of TOSCA and new features required to express HPC application and heterogeneous resource are added to it. Blueprint is the actual implementation of these templates that are used to represent specific application requirements, constraints and metrics by the Enterprise Application Developer. In particular, the blueprint
syntax finalized for CL is based on Brooklyn YAML implementation that is compliant with TOSCA specifications. The deployment of the blueprint can be done from the Apache Brooklyn web console; by copying the contents of the blueprint into the YAML dialogue box or by using ‘CURL’ tool on the command-line. A language for describing CloudLightning Blueprint was presented in WP5 D5.1.1 Service Description Format and with the inputs from the ray-tracing use case, the blueprints were created to deploy and manage the application in the cloud as shown in Listing 4.6

Listing 4.6 Blueprint created for end to end demo of Raytracing use case
```json
{
  ray_tracing_web_service:
    type: Docker_container
    id: "<filled-by-SOSO or Gateway service>"
    cpus: 0.1
    mem: 512
    instances: 1
    constraints:
      - hostname
      - CLUSTER
      - <filled-by-SOSO>
    properties:
      image: pkudaiyar/raytrace_web_service:1
      privileged: false
      network: BRIDGE
      portMappings:
        - containerPort: 9090
          hostPort: <filled-by-SOSO or left 0to decide by RM>
          protocol: tcp
      parameters:
        - key: hostname
          value: <filled-by-so or gateway service>
      volumes:
        - containerPath: "<filled-by-soso>"
          hostPath: "<filled-by-soso>"
          mode: RW
      labels:
        name: ray_tracing_webservice
  ray_tracing_engine:
    type: Dockerdoc_container
    id: "<filled-by-soso or gateway service>"
    cpus: 0.1
    mem: 512
    instances: 1
    constraints:
```
```yaml
- hostname
- CLUSTER
  - <filled-by-SOSO>
properties:
  image: pkudaiyar/mic_app_embree:1
  privileged: true
  network: BRIDGE
  portMappings:
    - containerPort: 22
      hostPort: 0
      protocol: tcp
  parameters:
    - key: hostname
      value: <filled-by-soso or gateway service>
  volumes:
    - containerPath: "<filled-by-soso>"
      hostPath: "<filled-by-soso>"
      mode: RO
  labels:
    name: embree_cpu_mic_instance
```
Chapter 5

Use Case 4: Self-Optimized Libraries (BLAS and FFTs)

BLAS (Basic Linear Subprograms), including matrix-vector and matrix-matrix multiplication, as well as FFTs (Fast Fourier Transform functions) are compute-intensive functions that are used in a lot of scientific applications and thus typically available as efficient libraries, several of which are self-optimized. In the following sections, we describe the main features of these libraries, and their main implementation on multi-core CPU, CPU clusters, GPU, MICs and FPGAs.

For this use case, since BLAS and FFTs are offered as optimized library functions on all the platforms, we for now, provide a review of BLAS and FFTs as they are offered as library function, often called by several applications.

This Use Case is, form a cloud perspective, actually very similar to the Upscaling Use Case from Oil & Gas described in Chapter 3. The Upscaling application interfaces with a given solver in DUNE and PETSc much the same way BLAS and FFT libraries would be used from a given application. Also, we have considered this use case a more “nice to have” than the other three, which were emphasized throughout our proposal. For this deliverable, we thus do not, for now, provide implementation details.

5.1 Introduction to Self-optimised Libraries

Computer architectures are becoming more and more complex with several layers of memory including several levels of cache, some shared among several cores; optimizing
code for a given platform is becoming very hard. Add multiple processors per system, and one can understand the attractiveness of using adaptive techniques on core libraries such as BLAS, as well as FFTs.

Vendors typically provide hand tuned versions of the core BLAS functions, such as general matrix-matrix multiplications, but recent efforts include auto-tuned library, such as Atlas (http://math-atlas.sourceforge.net/), a BLAS library and FFTW (http://www.fftw.org/) (FFTs). These libraries assemble their overall library functions (optimising for cache levels etc) through auto-tuning based on empirical testing of a given hardware system.

Both FFTW and cudaFFT are also vector-parallelised (e.g. use SIMD instructions), and the actual codes set up with a planner for a given size input, etc. Since these functions are both compute-intensive and used in many scientific applications, they are provided on many different platforms, including multi-core CPUs, many-core systems such as Intel’s Xeon Phi as well as GPUs. As mentioned in the project proposal, these libraries are thus a very interesting target as a use-case for our self-organised self-managed cloud as tasks that use these libraries can be scheduled on various heterogeneous platforms within a cloud environment.

The self-optimized version of these libraries that are targeting a given heterogeneous hardware, are vendor-specific. For this use case, we, for now, limit ourselves to identifying these libraries for our targeted platforms (Multi-core CPU, GPU, MIC and DFEs) and focusing on the vanilla DGEMM (Double precision GEneral Matrix Matrix multiplication) and FFT.

5.1.1 Service Specification and Requirements

The BLAS and FFTs that are self-optimized should generally be pre-compiled for the given targeted hardware as the optimizations may take significant time (but are worth it given how much they are used).

The optimizations also involves running a planner that would optimized the respective functions with respect to input size.

For a description of DGEMM (on Intel CPUs), see: https://software.intel.com/en-us/node/684733
Software requirements

The BLAS and FFT libraries are, as mentioned, typically provided by each vendor, but are also provided by others – e.g. the open source Atlas BLAS and FFTW libraries. They are easily callable from Fortran, C and C++ under Linux on CPUs and with special set-ups on MICs and GPUs. BLAS and FFTs are also provided on DFE FPGA-based Maxeler systems, but since these systems are 'software-on-a-chip', the setup is a bit more complex.

Software requirements – BLAS

The BLAS are available (non-optimized) as free CPU software via http://www.netlib.org/blas/ and as self-optimized open software for CPUs via the ATLAS (Automatically Tuned Linear Algebra Software) from http://math-atlas.sourceforge.net/

For Intel CPUs, Intel provides MKL (Math Kernal Library) that includes BLAS that are self-optimized. As stated on https://software.intel.com/en-us/intel-mkl, MKL:

- Includes highly optimized, threaded and vectorized functions tailored to maximize performance on each Intel processor family
- Utilizes de facto standard C and Fortran APIs for compatibility with BLAS, LAPACK and FFTW functions from other math libraries
- Is available with both free community-supported and paid support licenses

MKL includes support for MICs (Intel Phis) through set-up of environment variables that lets MKL spread the work-load between the CPU and MIC. Details on how to run MKL for MIC can be found at https://software.intel.com/en-us/articles/performance-tips-of-using-intel-mkl-on-intel-xeon-phi-coprocessor

For GPUs, NVIDIA provide cuBLAS as part of CUDA, their C-based annotated programming environment.

Maxeler provides customized codes that do DGEMM (matrix-matrix multiplication, the most central BLAS routine. Some detail are available at http://maxeler.github.io/maxpower/maxjdoc/index.html?maxpower/blas/l3/TileAccumulator.html
5.1 Introduction to Self-optimised Libraries

Software requirements – FFT

FFTW (Fastest Fourier Transform in the West) is an open source C subroutine library from MIT that provide auto-tuned FFT routines. As is stated at http://www.fftw.org/, FFTW Version 3.3 introduced support for the AVX x86 extensions, a distributed-memory implementation on top of MPI, and a Fortran 2003 API.

On GPUs, NVIDIA offers their auto-tuned cuFFT which works much like FFTW which uses a planner for auto-tuning.

Maxeler also offers FFT routines on their FPGA systems, but again it is more complex. It can currently be found on their application gallery page at http://appgallery.maxeler.com/#/ together with lots of other interesting FPGA-taylored applications.

Hardware requirements

As mentioned, the great thing about BLAS and FFT libraries, is that they are available on most, if not just about all, HPC platforms. One can either run them only on the host CPUs or run them on the accelerators. In Intel’s case, the MKL also provides automatic load-balancing between the CPU and MIC, so the power of both can be utilized.

5.1.2 Implementation of stand-alone software applications for BLAS and FFTs

BLAS and FFTs are offered as library function, much in the way that the Uscaling application on the Oil & Gas use case, called a given solver in DUNE and PETSc. The applications utilizing pre-compiled self-optimized BLAS and FFTs can thus be made available on most platforms (CPU, MIC, GPU) inside containers.

5.1.3 Deploying and testing using development systems

Since the BLAS and FFT libraries are available on all our main platforms, they may provide useful feedback to the simulations of the whole SOSM system. However, as mentioned earlier, this Use Case is actually very similar to the Oil & Gas use case, from a cloud perspective. For this reason the other three Use cases were emphasized in the proposal as well as for this deliverable. We thus do not, for now, provided
implementation details for this use case. Should one want to consider developing test-applications based on these libraries, some of them could be based on the benchmarking work done for D2.1.1.
Chapter 6

Summary and Future Work

In this deliverable, we initially analyzed the inputs and outputs of this task. Inputs are mainly from the task T2.1 where the applications for the use case were identified and their functional requirements were explored. In this deliverable, we took a step further to implement these use cases in multiple ways across the heterogeneous environment as proposed in CL project. The application software and hardware requirements are captured and are implemented as a standalone application in the bare-metal. Later, in order to reduce the complexity of building the monolithic applications and to follow the cloud-based approach, we packaged these applications in the VMs/containers. The procedure to build them from scratch or to download the VM/container from the CL repository are mentioned in detail. List of repositories and their software licensing requirements are mentioned in Appendix B.

This deliverable also captures the possible blueprint description for the use case applications by taking the inputs of Service Description Language defined in D5.1.1 to support the final integration and demonstration of the CL system.

Further, once the CL official testbed is configured and becomes operational at NTNU premises as per schedule (to finish by M25 by task T2.3), the use case applications will be executed by integrating with CL system. Application characteristics will be captured and analyzed. Depending on the results, the existing implementation may require some additional tuning through multiple iteration in order to achieve the desired goal. After successful evaluation of use cases and the satisfactory results, traces of application runtime characteristics will be gathered and fed into WP7 as an input for the large scale simulation.
Bibliography


Appendix A

This appendix covers the screenshots of the different front-end web portal pages constructed to demonstrate the end to end function of the raytracing use case where the user logs into the portal using his userID and password to create a new job or to access the already executed job.
Welcome to Ray Tracing and 3D Image Rendering Portal
Powered by CloudLightning

Select Your Rendering Job Option

Create New
Open Existing
Welcome to Ray Tracing and 3D Image Rendering Portal
Powered by CloudLightning

Step 1: Model File
Please upload your model file or specify the web URL

a. Browse to local file system
b. Type web URL of your model file

Browse
Reset
Upload

Getting Started Status Results

Ray Tracing Service

http://cloudlightning.eu/services/raytracing/session/64581354545802563336363

Search
Welcome to Ray Tracing and 3D Image Rendering Portal
Powered by CloudLightning

Step 1: Model File
Please upload your model file or specify the web URL

a. C:\Users\pkudaiyar\3dmodels\church.obj.zip

b. Type web URL of your model file

Model file received successfully
Welcome to Ray Tracing and 3D Image Rendering Portal
Powered by CloudLightning

Step 2: Ray Tracing Mode
Please choose from following options

- Interactive real time rendering (needs X11 clients)
- Batch / Offline rendering
Welcome to Ray Tracing and 3D Image Rendering Portal
Powered by CloudLightning

Step 3: Ray Tracing Parameters
Please configure the rendering parameters

a. Ambient Light (RGB, default: 1,1,1)
   - Value: 1,1,1

b. Frame Size (Width X Height, default: 768 x 512)
   - Value: 768 x 512

c. Initial Camera Position (X,Y,Z Co-Ordinates)
   - Value: Z0,10,10

d. Camera Look-at-Point (X,Y,Z Co-Ordinates)
   - Value: Z0,10,10

Start
Stop
Ray Tracing Service

Getting Started | Status | Results

Ray Tracing Job Status

Job Queue

<table>
<thead>
<tr>
<th>Job Name</th>
<th>Job ID</th>
<th>Rendering Mode</th>
<th>Submitted Time</th>
<th>Start Time</th>
<th>Finish Time</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Church</td>
<td>CLRT3D8599489F859I8958R</td>
<td>Real Time</td>
<td>8 June 2016 10:25:34</td>
<td>8 June 2016 10:25:38</td>
<td>unknown</td>
<td>Running</td>
</tr>
</tbody>
</table>

Job Logs: CLRT3D8599489F859I8958R
Ray Tracing Service

Job Name: 3D Church

Job ID: CLRT3D8599489FJF859J8958R

Download
Clean

Download Completed Jobs

Completed Jobs

Job Name
3D Church
Movie-1
Game-1

Download Completed Jobs

Getting Started  Status  Results
Appendix B

This appendix consolidates and presents the repository URLs for easy access and the license used by different libraries.

B.1 Repository URLs

<table>
<thead>
<tr>
<th>Use Case</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genomics</td>
<td><a href="https://bitbucket.org/cloudlightning/t2.2_genomics_sequence_alignment_use_case">https://bitbucket.org/cloudlightning/t2.2_genomics_sequence_alignment_use_case</a></td>
</tr>
<tr>
<td>Oil &amp; Gas</td>
<td><a href="https://bitbucket.org/cloudlightning/t2.2-oil-and-gas_upscaling_app_use_case">https://bitbucket.org/cloudlightning/t2.2-oil-and-gas_upscaling_app_use_case</a></td>
</tr>
<tr>
<td>Raytracing</td>
<td><a href="https://bitbucket.org/cloudlightning/t2.2_raytracing_integrated_usecase">https://bitbucket.org/cloudlightning/t2.2_raytracing_integrated_usecase</a></td>
</tr>
</tbody>
</table>

B.2 Use Case 1 - Genomics software dependencies

The genomics sequence-alignment application and the software stack necessary for running DFE-ported applications (MaxCompiler, MaxelerOS, SLiC, MaxIDE and related utilities) are proprietary to Maxeler Technologies and will not be released as open source. For use in the project, the relevant software and libraries are provided to project partners. For a list of dependencies, please see Section 2.2.

B.3 Use Case 2 - Oil & Gas software dependencies
### B.4 Use Case 3 - Raytracing software dependencies

<table>
<thead>
<tr>
<th>Technologies used</th>
<th>Licensed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc, g++, gfortran, ERT</td>
<td>GNU GPL3</td>
</tr>
<tr>
<td>Boost Library</td>
<td>BSL</td>
</tr>
<tr>
<td>Dune-Packages</td>
<td>GNU GPL2</td>
</tr>
<tr>
<td>SuperLU</td>
<td>GNU FDL</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>GNU LGPL V3</td>
</tr>
<tr>
<td>PETSc</td>
<td>BSD</td>
</tr>
<tr>
<td>Docker, TOSCA</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>Nvidia-docker, nvcc</td>
<td>Nvidia</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technologies used</th>
<th>Licensed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap, notify.js, spin.js, jade</td>
<td>MIT</td>
</tr>
<tr>
<td>jquery</td>
<td>jQuery License</td>
</tr>
<tr>
<td>pymongo</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>paramiko</td>
<td>LGPL</td>
</tr>
<tr>
<td>python-imaging-library</td>
<td>PIL License</td>
</tr>
<tr>
<td>nodejs</td>
<td>MIT</td>
</tr>
<tr>
<td>Embree</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>MPSS</td>
<td>GPL</td>
</tr>
</tbody>
</table>