Building AMD ROCm from Source on a Supercomputer

Cristian Di Pietrantonio
Pawsey Supercomputing Research Centre
Perth, Western Australia
cdipietrantonio@pawsey.org.au

***Abstract*—ROCm is an open-source software development platform for GPU computing created by AMD to accompany its GPU hardware that is being increasingly adopted to build the next generation of supercomputers. We argue that the fast-pacing evolution of their software platform and the complexities of installing software on a supercomputer mandate a more flexible installation process for ROCm than the available installation methods.**

ROCm-from-source is a set of Shell scripts developed at Pawsey to configure and build ROCm in a way that is not possible with the provided with the pre-built binaries, nor with the installer script found in the repository of each ROCm project. For instance, ROCm-from-source does not require root privileges, and a custom install location, possibly on a parallel filesystem, can be specified. It can do so by building ROCm and all its dependencies from source. The challenge in this task is represented by the substantial number of projects ROCm is made of as well as their dependencies, the intricate interplay between them, and the continuous changes in the structure of the projects.

Pawsey would like to share the knowledge acquired while developing ROCm-from-source so that other supercomputing centres can benefit from it.

I. Introduction

The widespread and rapid adoption of the AMD Graphics Processing Unit (GPU) as an accelerator for computational workloads by major supercomputing centres across the globe brought changes in the software stack as well as in the hardware. With many of the supercomputers equipped with AMD GPUs (and CPUs) coming online within the next few years, attention must now be turned towards the provisioning of the companion software development platform: ROCm.

The aim of this paper is to present a new, reliable, reproducible, and convenient method of deploying the ROCm software platform on a supercomputer, as envisioned by the Pawsey Supercomputing Research Centre. Its implementation consists of Shell scripts that tackle the daunting task of building ROCm from source, which we aptly refer to as *ROCm-from-source*. The topic is of greatest importance to all the supercomputing applications teams that must make available ROCm to thousands of users in a time when said software is still rapidly evolving.

The article is structured as follows. Section II introduces the reader to GPU development platforms, and gives an overview of ROCm. Section III articulates the reasons why Pawsey decided to undergo a months-long effort of developing ROCm-from-source, and why other centres should take advantage of it. Section IV offers an overview of similar projects and explains why they have not been adopted, and why ROCm-from-source should be preferred. Section V presents ROCm-from-source, its main components and the challenges it solves. Section VI discusses deployment on a HPE Cray EX system, and section VII provides instructions on how to create ROCm containers. Section VIII is about testing and in section IX the author concludes and hints at future work.

II. Background

Graphics Processing Units are a compute hardware accelerator dominating the computing landscape because of their ability to perform a large number of operations per second in a energy-efficient manner[1]. While their compute power is impressive, their widespread use is undoubtedly due to the companion software development platform, which makes programming them relatively easy.

A. GPU software development platforms

Even though NVIDIA has set the *de facto* standard for low-level GPU programming with its the GPU development platform, CUDA, in the last couple of years AMD has risen to a direct competitor by providing valuable products at a great price. This is especially true in the HPC market where they achieved great success in delivering hardware for the most powerful supercomputer in the world (as of 2022)[2]. The Pawsey Supercomputing Research Centre, the institution the author of this paper works for, purchased a HPE Cray EX supercomputer with AMD technology too. AMD provides its own development platform, called ROCm, that heavily borrows from the well-established CUDA programming model and APIs. Clearly, design choices were made to facilitate the transition of the HPC community from what was effectively a industry monopoly to multiple competing GPU vendors. Unlike CUDA, ROCm is open source. The two main implications are that contributions from external developers are welcomed, accelerating its development, and that binaries can be independently compiled if needed.

[1]The AMD MI250X delivers around thirty times more TFLOPS than a Intel Xeon Platinum CPU, while being thirteen times more power efficient. The MI250X achieves 47.5 TFLOPS FP64, TDP: 500W. A Intel Xeon Platinum 8376 peaks at 1.5 TFLOPS FP64, TDP: 205W.

[2]https://www.top500.org/lists/top500/list/2022/11

ROCm is a very complex ecosystem of more than fifty projects comprising drivers, libraries and tools. Figure 1 captures essential parts of it. Close to the hardware level, ROCK-Kernel-Driver enables communication between the AMD GPU and the Linux kernel. The corresponding software interface executing in the user space of the operative system is ROCT-Thunk-Interface. It provides the fundamental building blocks to develop a runtime system executing on the host platform (that is, the CPU). ROCR-Runtime implements the Heterogeneous System Architecture (HSA) API enabling kernel programs submission, memory management and device synchronisation. It is equivalent to the CUDA Runtime API. Completing the low-level components of the AMD software stack, ROCm-Device-Libs implements the runtime and libraries executing on the GPU hardware.

ROCm has a compiler ecosystem to transform source code to low-level machine code. It extends LLVM to support the AMD GPU Instruction Set Architecture (ISA), named GCN. It implements OpenMP offloading, and proprietary GPU and CPU optimisations. The enhanced Clang compiler depends on ROCm-Device-Libs to produce executables able to run on AMD GPUs.

Several libraries expose GPU-accelerated implementations of popular algorithms in the science domain. Examples are Fast Fourier Transforms (rocFFT), linear algebra routines (rocBLAS), and random number generators (rocRAND).

HIP makes available to developers a high-level programming interface built on top of ROCR-Runtime, it provides wrappers around ROCm numerical libraries, and it extends the C++ language to enable programmers to write code that runs on GPU. It is designed to mimic the CUDA API to facilitate adoption and code porting. In fact, HIP can be configured and installed to be a wrapper around CUDA, enabling code portability. The vendor-agnostic HIP is complemented by the hipamd project to optimally target AMD devices. Management (`roc-smi`, `rocinfo`), debugging (`rocgdb`) and profiling (`roctracer`, `rocprof`) tools are also available.

Finally, the MIOpen library implements machine learning algorithms and primitives. It is used by frameworks such as Tensorflow and PyTorch to enable machine learning workloads on AMD hardware.

III. Motivation

Pawsey and other centres must deploy the fast-evolving ROCm platform in a timely manner for users to take advantage of latest critical improvements. It is a challenging task because of the strict requirements imposed by the administration policies of an HPC infrastructure.

ROCm documentation illustrates in details how to install pre-built binaries on workstations or small server-like environments. In these scenarios, coupling system administration with end-user applications management in one role with privileged access is feasible and common. On the other hand, tens of people distributed across different teams are needed to properly manage every aspect of running a supercomputer, with only a handful of them having access to *root privileges*. ROCm-from-source does not require administrative rights to be executed. This allows the applications team of a supercomputing centre to retain control over the deployment and maintenance of a critical user-facing component of the system.

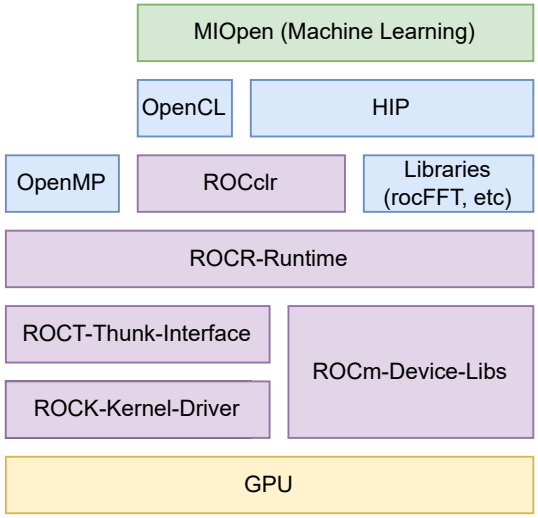

Fig. 1. A schematic and simplified representation of the ROCm software stack. At the very bottom lies the GPU hardware. The software side starts with components making up the ROCm runtime and low-level interface, coloured in purple. Various programming models are provided on top for programmers to use, complemented by highly optimised libraries. These are indicated with the blue colour in the figure. At the highest level, advanced and complex libraries such as MIOpen are found (green colour).

The fast release cycle ROCm is subject to contrasts with the slower deployment schedule of the Cray Programming Environment (CPE) on supercomputers. This means that HPE Cray sites cannot rely on vendor system-wide updates to obtain the latest ROCm version within a month or two from its release[3]. Hence, the task of updating the OS image just to include a newer ROCm version alongside the officially-provided one would fall on system admins. Unfortunately, modifying and re-deploying a HPE Cray OS image is a disruptive and time consuming task. Moreover, an additional RPM ROCm installation would consume around 10 GB of RAM when the image is loaded in memory by compute nodes.

The AMD installer places ROCm binaries within the directory `/opt/rocm` and, at the time when the author first started working on ROCm-from-source, multiple installations of ROCm were not permitted. Today AMD allows several versions of ROCm to coexist on the same system. The installation prefix, however, remains the same and cannot be changed. A customisable installation path would allow ROCm to be placed on a parallel filesystem, ideally one dedicated to software deployment. ROCm-from-source allows for it and, in doing so, it solves the memory issue mentioned previously as well as it enables a site-defined software hierarchies.

Several other reasons motivate building ROCm from source.

[3]For instance, Pawsey's Setonix is still running `cpe/22.09` featuring `rocm/5.0.2`, whereas the latest ROCm release is 5.4.3.

For instance, binaries can target the architecture they will run on; projects and features that are not enabled in the official distribution can be included.

IV. Related Work

There are several third-party projects whose objective is to provide an alternative installation method for ROCm. They all leverage the publicly available ROCm sources but none of them works without requiring significant effort, hence motivating the work presented in this paper. The AOMP project by AMD implements scripted build of AMD LLVM and some of the ROCm projects [3]. Besides the fact that its scope is limited, the system is quite complex, making it hard to adapt and execute.

A. Spack

Spack is a software manager for supercomputers equipped with thousands of *recipes* for building from source the most common applications and libraries [6]. AMD started contributing to the development of Spack recipes for ROCm packages since Spack 0.16.0. When the author started working on ROCm-from-source, Spack support for ROCm was very experimental and not to be relied on. To date, the latest stable release of Spack is 0.19.0 and its recipes for ROCm components have vastly improved. Despite that, there are a couple of reasons to prefer ROCm-from-source to Spack now that ROCm is rapidly evolving. A site like Pawsey may stick to a version of Spack, hence its recipes, for an extended period of time (six months to one year). During this period, newer versions of ROCm may fail to build with old recipes. For instance, HIP 5.4.3 cannot be built the corresponding recipe in Spack 0.19.0. Updating the single script in ROCm-from-source taking care of installing ROCm projects is more practical than updating several Spack recipes, either in first person or downloading newer recipes (with may be incompatible with the Spack version currently used). In addition to that, a matter of convenience: ROCm projects are distributed independently, without a "meta-recipe" being provided to install all of them in one sitting. Finally, as it will be discussed later, the Cray Programming Environment may expect a ROCm deployment to follow a determined directory hierarchy, whereas Spack installs each project on its own.

B. Scripted builds

There already exist at least two long-standing third-party projects providing an alternative solution to the official deployment procedure of ROCm [1], [7]. These efforts are driven by the desire of independent developers and researchers to run machine learning frameworks, namely Tensorflow and PyTorch, on consumer-grade AMD GPU cards. All of them build the entire stack from source. The same AMD provided minimal instructions in the earlier documentation pages of ROCm on how to fetch all the necessary code repositories to be then compiled [4]. These instructions have been now removed from the newer documentation.

```
# Maintainer: Torsten Keßler <t dot kessler at posteo dot de>

pkgname=rocwmma
pkgver=5.4.1
pkgrel=1
pkgdesc='Library for accelerating mixed precision matrix multiplication'
arch=('x86_64')
url='https://rocwmma.readthedocs.io/en/latest/index.html'
license=('MIT')
depends=('hip' 'rocblas' 'openmp')
makedepends=('rocm-cmake' 'doxygen')
_git='https://github.com/ROCmSoftwarePlatform/rocWMMA'
source=("$pkgname-$pkgver.tar.gz::$_git/archive/rocm-$pkgver.tar.gz")
sha256sums=('641d2730db737edcade8da6b3f77ce85d4ad460e0902c2b688df2d51fb13f9f0')
_dirname="$(basename "$_git")-$(basename "${source[0]}" .tar.gz)"

build() {
    cmake \
      -Wno-dev \
      -B build \
      -S "$_dirname" \
      -DCMAKE_INSTALL_PREFIX=/opt/rocm \
      -DCMAKE_BUILD_TYPE=None \
      -DROCWMMA_BUILD_TESTS=OFF \
      -DROCWMMA_BUILD_SAMPLES=OFF
  cmake --build build
}

package() {
    DESTDIR="$pkgdir" cmake --install build

    install -Dm644 "$_dirname/LICENSE.md" "$pkgdir/usr/share/licenses/$pkgname/LICENSE"
}
```

Fig. 2. An example of `PKGBUILD` script from rocm-arch. As one can see, it is meant to be interpreted, not executed as is. Nonetheless, the `build` function shows the CMake parameters to use.

Started around the year 2020, rocm-arch implements a build process targeting Arch Linux, a very lightweight Linux distribution [1]. It does so by supplying `PKGBUILD` recipes for the most important ROCm components. Unfortunately, the domain-specific language those recipes are written in prevents their use in a more general setting, even more so on a supercomputer. Figure 2 shows an example PKGBUILD script. The lack of complete coverage of all ROCm projects is another problem stopping Pawsey from using this project as is.

A noteworthy effort that comes close to ROCm-from-source is hosted in the Xu Huisheng's GitHub repository *rocm-build* [7]. It targets the Ubuntu Linux distribution. In the repository one can find a comprehensive collections of Shell build scripts, one for each ROCm project; their name are prefixed with a number, establishing the order of execution. They allow for some flexibility, such as the possibility of specifying custom build and installation directories. The scripts, however, do not install binaries directly but they create `.deb` packages. These will have to be then installed using a packet manager. A script, whose purpose is to install dependencies, is also present. Not surprisingly, it delegates to the Ubuntu packet manager the retrieval and installation of each one of them. This project represents a basic starting point for an adaptation that would work on a supercomputer, but much effort would have to be dedicated before reaching that point.

V. ROCm-from-source

This paper introduces ROCm-from-source, a build system for ROCm written entirely in Shell. It freely available on GitHub at the following link: `https://github.com/`

PawseySC/rocm-from-source. While Pawsey uses it to deploy ROCm on a HPE Cray EX system, ROCm-from-source is designed to run on many Linux distributions with only a minimal set of dependencies. It can therefore be easily adopted by other institutions looking for a manageable way of providing the AMD GPU development platform and runtime to its users.

Using ROCm-from-source is easy. Listing 1 demonstrates how Pawsey staff deployed the latest ROCm on the GPU partition of Setonix for early adopters to try.

```
git clone --branch rocm5.4.3rev0 \
https://github.com/PawseySC/rocm-from-source.git
export ROOT_INSTALL_DIR=\
  /software/setonix/2022.11/rocm
./rocm-from-source/install_rocm.sh
```

Listing 1. Executing ROCm-from-source only requires a few command lines.

Each release of ROCm-from-source is tagged with rocm{X}.{Y}.{Z}rev{R}, where

- X, Y, Z are the major, minor and patch numbers of a ROCm release. That is, they indicate which version of ROCm will be built.
- R is the revision number of the build scripts for a given ROCm version. It is increased each time an improvement is done for the same ROCm release.

There are a few other parameters that can be set to customise the ROCm build and installation, well documented within the *driver script*.

A. The driver script

The install_rocm.sh script orchestrates the execution of various other helper scripts to install ROCm. Before that happens, sensible defaults for build and installation parameters are set. If run on an HPE Cray supercomputer, the installation prefix is the only information required from the user.

Other important input variables are the following ones.

- GFX_ARCHS lists GPU architectures ROCm device code will be compiled for. The default value is gfx90a, the one powering the MI200, MI210, and MI250X GPUs.
- ROCM_DEPS_INSTALL_DIR specifies the location where to install ROCm dependencies. Because they are likely not to change from one ROCm version to the next, placing them in a dedicated location enables their reuse, hence reducing compilation times and number of files.
- MODULEFILE_DIR sets the location where ROCm module files are installed.

The script is also configured to use all the CPU cores available, and to skip the build of already-installed packages in case of re-execution.

Next, the utils.sh file containing utility Shell functions used throughout ROCm-from-source scripts is sourced. These are the topic of the next subsection.

ROCm-from-source can be executed on a variety of Linux environments. A supported use case is the installation on a Ubuntu machine. This is useful for development purposes and, importantly, to build ROCm containers. The script will test whether the installation is taking place on a supercomputer or on a Ubuntu system. In the former case, the module command is used to load the compiler and Python modules. In the latter, it is assumed the script is run as the root user and the apt command will be used to retrieve build dependencies required in such scenario.

The build process is then started by sourcing, in order, the following files:

- set_env.sh to set common Linux environment variables, such as PATH and LD_LIBRARY_PATH, as well as ROCm specific ones like ROCM_PATH;
- install_build_deps.sh to download and install in a temporary location build dependencies like CMake;
- install_rocm_deps.sh to install ROCm dependencies; and,
- install_rocm_projects.sh to finally build and install all the ROCm projects.

The following sections will explore each of these steps in detail.

B. Shell functions

To simplify scripts, avoid programming mistakes, and ease future development, repetitive sequences of Shell commands have been wrapped in convenient high-level Shell functions. For instance, a typical command sequence to download and unpack source code is wget <url-to-tar>, tar xf <tar-file>, and cd <src>. Such sequence is replaced with a custom wget_untar_cd <url> command.

```
wget_untar_cd () {
    url=$1
    tarfile=${url##*/}
    folder=${tarfile%.tar.gz}
    if [ -z ${BUILD_FOLDER+x} ]; then
        BUILD_FOLDER=".";
    fi
    cd ${BUILD_FOLDER}
    [ -e ${tarfile} ] || \
      run_command wget "${url}"
    [ -e ${folder} ] || \
      run_command tar xf "${tarfile}"
    cd "$folder"
}
```

Listing 2. The implementation of the wget_untar_cd <url> command takes care of downloading a tarball, extracting the compressed folder, and changing the current work directory. Note that if the tarball or folder is already present, the function changes directory and nothing more.

Similarly, we abstract compilation sequences like the idiomatic configure, make, and make install. Eventually, ROCm-from-source can build and install from source with a single command like configure_install <url> and cmake_install <folder> <args>.

Within these high-level commands, logic to handle multiple attempts at building the same source code is easily implemented. For instance, if the installation process gets interrupted unexpectedly, one would want to restart right before

the interruption happened, and not to rebuild everything. On the other hand, when debugging a compilation error in a new ROCm version, starting from a clean state is a good idea.

The collections of Shell functions in the `utils.sh` file makes ROCm-from-source reliable and easy to maintain. It also makes ROCm-from-source approachable by new users and contributors. The author argues that the described approach is almost necessary considering the large number of projects that must be built and installed.

C. Build environment, compiler and linker options

The build environment must be under tight control for the installation process to be reproducible and reliable. The are a variety of software packages, possibly multiple versions of each, present on an HPE Cray system. The default installation of ROCm is one of them. ROCm-from-source must ensure that only libraries built as it proceeds are linked in the successive steps, not the ones in other pre-existing ROCm deployments. To pick up the right libraries at compilation time, `set_env.sh` prefixes and exports the `LIBRARY_PATH` and `LD_LIBRARY_PATH` variables with the final installation path of ROCm projects. The `RPATH` field of the generated libraries and executables is also populated with the same paths to achieve the same behaviour during runtime. To do so, the `-rpath` linker option is used within the `LDFLAGS` environment variable.

Only the `$ROCM_PATH` prefix has to be added to the search path for the ROCm binaries to be found. This is because ROCm is migrating from deploying each project within its own sub-directory to placing all libraries and executables under the same filesystem location. For instance, HIP is not installed anymore in `$ROCM_PATH/hip` but it can be found in `$ROCM_PATH`. The old directory is still created for compatibility reasons, but it contains placeholder headers and symlinks to new file locations. Warnings are generated when header files in the old location are included.

There exist ROCm environment variables that are needed both at build time and runtime. These are `HIP_PATH`, `HSA_PATH`, `HIP_CLANG_PATH`, `ROCM_PATH`, and `HIP_RUNTIME`. The first three point to the locations of the respective ROCm components and are notably used within the `hipcc` compiler wrapper. The `ROCM_PATH` variable is used by `clang` to find the device libraries, and has a corresponding command line argument: `--rocm-path`. The last variable, `HIP_RUNTIME`, seems to accept only `"ROCclr"` as value.

The script relies on `PrgEnv-gnu`'s `gcc` to compile ROCm dependencies and the AMD fork of LLVM. Once `clang` has been compiled, it is set to be compiler for ROCm projects. The `gfortran` compiler is used to compile Fortran code in rocBLAS and other numerical libraries. The author tried to use HPE Cray compiler wrappers for C/C++ instead of directly executing `gcc` but they resulted in link errors. Compilation with the `PrgEnv-cray` module has not yet been attempted, but the author does not see any potential issue given that the underlying compiler is Clang.

ROCm projects use CMake to define their build process. Many of them depend on HIP and resolve such a dependency by leveraging CMake's built-in support for HIP. This feature is found in CMake version 3.21 or newer. If no suitable version of CMake is found in the system, one will be compiled from source.

D. ROCm dependencies

ROCm depends on several software libraries that may not be present on an HPE Cray system, or that are installed without all the required components. Examples are `libX11`, `libdrm`, `elfutils`, and `gettext`. For some projects, dependencies were listed in the project description. For others, they were discovered during the build process through CMake configuration errors and inspecting `CMakeLists.txt` files. Discovering and installing all dependencies from source was one of the most time-consuming step of the development of ROCm-from-source. Considerable effort went also towards identifying environment variables that interfere with configure scripts. An example is the `LDFLAGS` variable causing compilation errors in `elfutils`, or the `CPLUS_INCLUDE_PATH` interfering with the Boost build.

Interestingly, the rocprofiler project requires the closed-source aqlprofiler library, developed by AMD. Only its binary is distributed. In ROCm-from-source, the binary is downloaded and placed in the appropriate `lib64` directory. There have been discussions [8] in the community on whether the source code could be provided. Some managed to modify the code of rocprofiler to remove the dependency altogether.

E. ROCm projects

ROCm is open source and thus all projects are available on GitHub. One particular repository, `https://github.com/RadeonOpenCompute/ROCm`, acts as index of all the others. The `repo` command can download and checkout many repositories in parallel. As a first step towards compiling ROCm, our script downloads all the indexed repositories by providing the `repo` command with the index repository. Instructions to do so were present on a now-old documentation page [4]. There are a couple of projects missing from the index, as a result of refactoring processes, such as hipRAND and Flang. Those are downloaded separately.

In the initial attempts at compiling ROCm the author tried to download and compile one project at a time but would eventually hit compilation errors. It turned out that many ROCm projects, at least in the release version `4.5.0`, relied on the presence of the other projects within the same parent directory. Relative paths to access each other files were commonly seen in source code and `CMakeLists.txt` files. While project configurations have improved, the use of the `repo` tool is the expected way of retrieving ROCm source code.

In the beginning the author's plan was to compile HIP, followed by the numerical libraries and, finally, supporting tools like `rocgdb`. However, the order in which projects had to be compiled was not fully documented. A reverse engineering approach was followed during which, starting

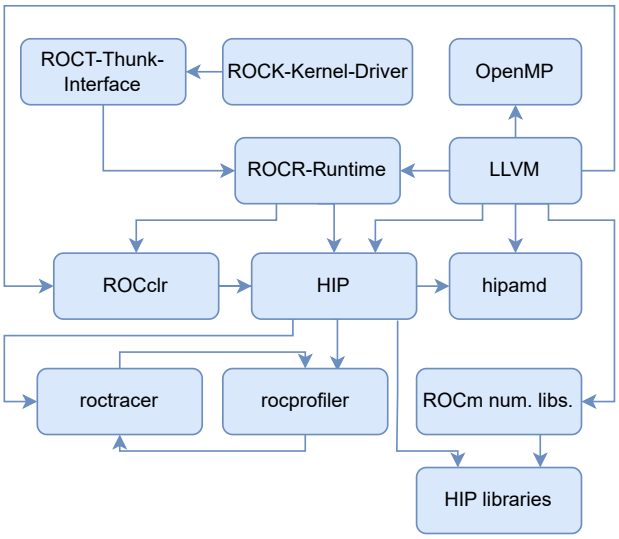

Fig. 3. A simplified diagram showing direct build dependencies among principal ROCm projects.

from a project, the author worked recursively his way towards all the required dependencies. The discovery of the *AOMP* project somewhat helped speeding up the process. It is a scripted build of AMD's LLVM that represents another source of information regarding the expected configuration of some projects [3]. Figure 3 shows a simplified dependency graph relating major ROCm projects.

The build and installation of each ROCm project is handled by CMake. With the `cmake_install` Shell function doing the heavy lifting, such that each project can be compiled with a single command line, what is left for the author to determine are the correct CMake options. When ROCm-from-source started, not every ROCm project documented CMake build instructions. The reason still holds today. Most users install the software using provided installer scripts, including the developers themselves. Hence, the author scanned those scripts to find the right combinations of build parameters. Given the complexity of those scripts, sometimes direct help from AMD developers was crucial [5]. One of the ROCm developers reached out for feedback on the ROCm installation process. The outcome of the exchange was better documentation around the CMake options for each project.

The HIP project makes an interesting case of why building ROCm from source is not easy. There exist two repositories implementing HIP: *HIP*, also referred to as *common HIP*, and *hipamd*. The former is a platform-agnostic implementation: it can be compiled for either ROCm or CUDA. The latter extends the former to provide a specialised implementation for AMD GPUs. In addition, HIP depends on ROCclr, Radeon Open Compute Common Language Runtime, which enables portability across Windows and Linux. What projects to compile and in which order is not easily understood. For instance, ROCclr documentation shows how to compile the project on its own [2]. However, HIP documentation implies ROCclr is now built during the HIP build process, with only the path to its source code to be provided.

The AMD fork of LLVM supplies the compiler infrastructure within ROCm. Several LLVM projects are required, all are installed under the `$ROCM_PATH/llvm` prefix. The Clang project results in a compiler that is used by the `hipcc` wrapper script to compile HIP code. The device-side runtime and libraries are implemented in the ROCM-Device-Libs external project of LLVM. Once compiled, device libraries are present under the `$ROCM_PATH/llvm/amdgcn/bitcode` directory. The `hipcc` compiler driver expects to find `$ROCM_PATH/amdgcn/bitcode` instead, so ROCm-from-source makes sure the required symlink is created. AMD forked the "Classic Flang" project to provide a Fortran compiler with optimisations for its hardware, hence ROCm-from-source avoids building the *in-tree* LLVM Flang. LLVM implements OpenMP offloading (computations to GPUs) within the `libomptarget` library. In AMD pre-built ROCm 4.5.0, Clang was not compiled with this feature enabled. At the time, ROCm-from-source offered a way of enabling and testing OpenMP offloading. In ROCm 5.0 and later, OpenMP offloading is available by default.

A trial and error process, combined with the guidance provided by AOMP build scripts, gave the author knowledge on how to compile the LLVM stack. The ROCm-Device-Libs project is better built together with LLVM, specifying it as external project in the CMake command line. The OpenMP runtime, on the other hand, has to be compiled separately. It depends on ROCR-Runtime, which in turns requires Clang and the device libraries to be already built. Hence, the `LLVM_ENABLE_RUNTIMES` CMake option is not to be used when building LLVM. Finally, Classic Flang requires building three different projects, in the following order: the `libpgmath` dependency, Flang, and the Flang runtime.

F. Patches and bug fixes

Minor bugs in build configurations and source code are routinely found in ROCm projects. Most likely due to the rapid development process, these issues have to be fixed for ROCm to be compiled successfully. Importantly, thanks to ROCm being open source, the author contributes to the affected projects with GitHub *pull requests* to patch the upstream code, or opened GitHub *issues*.

An interesting example is represented by the roctracer and rocprofiler projects in ROCm 5.3.0. There seems to be a circular dependency between the two, whereby roctracer needs rocprofiler headers, and rocprofiler depends on roctracer being built and its header files being installed. The quick fix adopted to make roctracer compile is manually copying rocprofiler header files within the roctracer project directory.

The rocBLAS build process installs Tensile, a dependency, within a Python virtual environment. Because previously installed Python dependencies are placed in a custom filesystem location, they are not included in the newly created environment. This causes Tensile not to work properly. The solution adopted is to install Tensile in the same custom

location, avoiding virtual environments altogether. A patch to the `CMakeLists.txt` file of rocBLAS implements the fix using a couple of command lines.

The build configuration of the ROCm projects is such that CMake looks for dependencies in the default ROCm installation path, which is `/opt/rocm`. This behaviour cannot be easily changed through a CMake parameter because the path is hardcoded into the `CMakeLists.txt` file. A `sed` command is then used to replace `/opt/rocm` with the `$ROCM_INSTALL_DIR` value.

Other patches address wrong type casting in roctracer and hipamd codes, wrong installation of HIPIFY binaries and HIP CMake config files, and the removal of problematic flags in the compilation of LLVM OpenMP. The need for these patches are reviewed at each new ROCm release, and possibly new ones are generated.

VI. Deployment on HPE Cray EX

The author identified two aspects of deploying ROCm on a HPE Cray EX system. The build and installation process is one; the other is integrating it with other applications and tools.

ROCm-from-source was designed to minimise the number of dependencies coming from the external environment. Hence, deploying ROCm on an HPE Cray system is conceptually no different, nor harder, than installing any other package from source. Unfortunately, the compiler wrappers available in the Cray Programming Environment disrupt the linking stage for some of the ROCm projects and dependencies. Fortunately, the author does not see any advantage in using them, given that Clang will be used to compile ROCm projects in any case.

Pawsey-built ROCm complements the one already present on the HPE Cray OS image. It is currently installed as part of the software stack based on the GNU Programming Environment, like all other software. ROCm libraries are integrated into the Spack software manager as *external packages*. In this way, they are available to satisfy dependency requirements during GPU-enabled software installations.

Pawsey staff have started exploring the way Cray Programming Environments leverage ROCm to offload computations to AMD GPUs. The test case is a Fortran code that implements matrix multiplication and uses OpenACC. The author attempted to compile and run the program with both the preinstalled ROCm 5.0.2 and our custom build, and with the Cray and GNU programming environments. The GNU environment, `gfortran` version 12 in particular, seems to compile the code and link the executable to ROCm libraries. However, computation still happens on the CPU, indicating that perhaps one must wait the support for the gfx90a ISA in GCC 13.

The code compiles and runs on GPU as expected when using the Cray Programming Environment with the HPE Cray-installed `rocm/5.0.2`. One must remember to also load the `craype-accel-amd-gfx90a` module, otherwise GPU offloading directives are quietly ignored. Warnings and errors arises during the compilation with ROCm 5.4.3 built from source, as shown in listing 1.

```
Warning: Cannot find all neccessary
path for loaded rocm version!!!
lld: error: undefined symbol:
 __ockl_get_num_groups
>>> referenced by [...] cce-openmp__llc.amdgpu
```

Listing 1. Warning and error messages when compiling with custom ROCm suggests missing libraries or incompatible ROCm version.

Without having access to internal documentation or source code of the Cray Fortran compiler, we can only take a guess on what those messages mean. The warning suggests the installation directory tree does not look like it should according to the CPE. This may be due to a project not being installed or to a difference in the installation layout between the two ROCm versions. The linking error might be a consequence of the first warning.

What can also be inferred by listing 1 is that OpenACC directives are translated to OpenMP ones. Further investigations running the compiler in verbose mode confirms it: the `cce_omp_offload_linker` executable, part of the LLVM build by HPE Cray, is called.

VII. Build ROCm containers

ROCm-from-source is well suited for building custom ROCm containers. Scripts are written in the Shell scripting language, and the build process relies on external dependencies only minimally. Hence, building a container is as easy as executing the main script from within a Dockerfile or a Singularity definition file.

```
%post
apt-get -y update
export ROOT_INSTALL_DIR=/opt/rocm
export CPATH=/usr/include/python3.9:$CPATH
/rocm-from-source/install_rocm.sh
rm -rf /rocm-from-source
```

Listing 2. Excerpt from a Singularity definition file where ROCm-from-source is used to build a ROCm container.

The advantage of building your own ROCm container is again found in more control over installed software, new features enabled within ROCm projects, and possibility of reducing the container size by choosing a lightweight operative system image.

Within this space, Pawsey looks forward to provide its users with Tensorflow and PyTorch containers. Currently, the Tensorflow container distributed by AMD lacks several required Python packages and so it is unable to run out of the box. GPU-enabled OpenFOAM containers represent another use case. OpenFOAM is a Computational Fluid Dynamics (CFD) software that can be built in many different ways, and containers are the preferred way for Pawsey to support it.

VIII. Testing

ROCm installations produced by ROCm-from-source were tested on a variety of systems at Pawsey. These include AMD

provided high-end servers featuring MI100 GPU cards, and the Test and Development System (TDS) for Setonix. A fresh installation takes less than two hours using sixty-four CPU cores and, with the currently enabled projects, occupies around 7 GB of disk space. This figure does not include machine learning libraries, as explained in the *Future work* section. The author verified, by manual inspection, that all major components were indeed installed. Simple HIP applications and benchmarks were compiled and run successfully. OpenMP offloading for the C++ language works correctly, but the Flang offloading capabilities will only be present starting from LLVM 16.

All was done in preparation for Setonix, the Pawsey supercomputer currently being brought to production. Unfortunately, delays in the commissioning process meant that source builds of ROCm could not be thoroughly evaluated with production codes and workloads. Moreover, experiments that involve the help of a system administrator, like a comparison with a RPM installation, could not be performed due to commissioning-related tasks having higher precedence. As discussed in previous sections, integration with the Cray environment is still a work in progress. However, some preliminary testing on Setonix has been already done. The ROCm Validation Suite executed successfully, and results align with the ones of the preinstalled ROCm 5.0.2. One significant issue yet to be solved came up. The AMD GPU driver is too old for ROCm 5.4.3 and some components might not work, like `rocgdb`.

Very recently, Pawsey has granted selected research groups access to the GPU partition of Setonix. They have available the latest ROCm, which was built with ROCm-from-source, and will be using it in the incoming weeks to develop highly scalable software as part of the Pawsey Center for Extreme-scale Readiness (PaCER) program.

IX. Conclusions and future work

This paper introduced a source build process for AMD ROCm that targets installations on a supercomputer. While doing so, the author gave an overview of the main components of ROCm, discussed complexities of said software and the difficulties encountered while compiling it. Researchers will only start testing the deployment during the incoming weeks, but early results are encouraging and Pawsey already provides a source build of ROCm alongside the one installed on the HPE Cray operative system image.

All the ROCm components needed for high performance computing, namely HIP and numerical libraries, have been taken care of. The work is not finished yet, however, because other projects must be compiled. Machine learning libraries like MIOpen were also included in the build process in early versions of ROCm-from-source. As the focus moved to fixing issues around LLVM and numerical libraries, the author decided to pause efforts on components of secondary importance to Pawsey. In the near future, the work on machine learning libraries will resume as Pawsey seeks to provide users with optimised Tensorflow and PyTorch containers.

Future effort will be dedicated to installing ROCm outside of a Cray Programming Environment. We would then provide ROCm as an additional programming environment. Enabling and testing the OpenCL runtime is another future objective of great importance. A few research groups and third-party applications use OpenCL as their framework of choice to offload computations to GPUs.

To support systems whose compute nodes do not have Internet access, ROCm-from-source could be rewritten to work with source tarballs downloaded beforehand. At the moment, source code for each ROCm dependency is retrieved right before its compilation. Finally, keeping up with new ROCm releases will be an ongoing concern for the author. In particular, new projects may be added to ROCm, patches may need to be revisited or removed altogether, new bugs fixed.

References

[1] Torsten Keßler Akash Patel. ROCm for Arch Linux. https://github.com/rocm-arch/rocm-arch (accessed: March 13th, 2023), January 2019.

[2] AMD. ROCclr - Radeon Open Compute Common Language Runtime. https://github.com/ROCm-Developer-Tools/ROCclr (accessed March 29th, 2023).

[3] AMD. The AOMP repository. https://github.com/ROCm-Developer-Tools/aomp (accessed: March 28th, 2023).

[4] AMD. Getting ROCm source code. https://sep5.readthedocs.io/en/latest/Installation_Guide/Installation-Guide.html#getting-rocm-source-code (accessed: March 13th, 2023), January 2014.

[5] Cristian Di Pietrantonio. [Bug]: wrong include statement? . https://github.com/ROCmSoftwarePlatform/rocBLAS/issues/1255 (accessed: March 28th, 2023).

[6] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. The spack package manager: bringing order to hpc software chaos. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[7] Xu Huisheng. rocm-build. https://github.com/xuhuisheng/rocm-build (accessed: March 13th, 2023), December 2022.

[8] littlewu2508. Possiblity of providing the source code of libhsa-amd-aqlprofile64. https://github.com/RadeonOpenCompute/ROCm/issues/1781 (acccessed: April 14th, 2023).