International Master of Computer Science

E5 Internship Report

LLVM/Clang integration into Buildroot

Valentín Korenblit

Tutor
Romain Naour

Supervisor
Dr. Yasmina Abdeddaïm

9th July, 2018

Acknowledgements

I would like to thank my tutor, Romain Naour, for his continuous guidance and feedback throughout this project. It would not have been possible without his help and expertise. I would also like to thank Smile for trusting and supporting me since the beginning of my internship.

Another person who made this possible is my supervisor, Yasmina Abdeddaïm. She has encouraged me and offered me a listening ear at all times, which gave me the strength to continue when I was about to give up.

Additionally, I want to thank the Buildroot community for taking the time to review my contributions and give me the necessary feedback to make the project go in the right direction.

Finally, I want to thank my parents and sister. They know how hard it is for me to be so far from them but they continue to bring me their support and motivation everyday despite the distance.

Contents

Acronyms

ABI	Application Binary Interface
API	Application Program Interface
AVX	Advanced Vector Extensions
DMA	Direct Memory Access
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GCN	Graphics Core Next
GLSL	OpenGL Shading Language
JIT	Just in Time
RISC	Reduced Instruction Set Computer
SSA	Static Single Assignment
VC4	VideoCore IV
WIP	Work in Progress

1 Introduction

Embedded systems are present in many kinds of devices, ranging from aircraft systems to consumer electronics. Thanks to the technological advances, some recent embedded sytems outperform desktop PCs of several years ago, allowing them to run a Linux system. However, while regular Linux distributions are generic and well suited for desktop systems, embedded systems present different kinds of hardware/software constraints that need to be satisfied, which normaly involves applying modifications to the operating system. Because of this, there exist build sytems like Buildroot that can generate a fully customized embedded Linux system optimized for a particular target hardware.

Two aspects that can be customized using Buildroot are the packages that will be installed on the target system and the cross-compilation toolchain that is used to build them. LLVM as a compiler infrastructure can play both roles in Buildroot: one one hand, it can be seen as a target package which provides functionalitites such as code optimization and Just In Time compilation to other packages, whereas on the other hand it opens the possibility of creating a cross-compilation toolchain that could be an alternative to Buildroot's default one, which is based on GNU tools.

This project is mainly focused on LLVM as a target package. Nevertheless, it also discusses some relevant aspects which need to be considered when building an LLVM/Clang-based cross-compilation toolchain.

1.1 Organization

This document is organized in a way that the technologies involved in the project are first introduced in order to provide the reader with the necessary information to understand the main objectives of it and interpret how software components interact with each other.

Section 3 discusses the most important aspects of Buildroot in order to provide some background information. Section 4 introduces the LLVM project from the user's point of view and then digs into its internal aspects to show how its unique design presents several advantages over GCC. As the first goal of this project is providing LLVM support for OpenGL, Section 5 discusses the Linux Graphics Stack to see which is the role of LLVM in this complex system. Section 6 contains the development of all the project itself, doing emphasis in all the problems encountered along the way. As this work was done in several iterations, this section shows the state of progress at different dates, which gives a clear view of how the project evolved and which functionalities were added on each iteration. Finally, Section 7 discusses possible future work and concludes the document.

SMILE
I.T IS OPEN

2 Smile

Smile is a French company specialized in the integration of open source solutions. Created in 1991, it has 15 agencies in 7 countries: France, Switzerland, Netherlands, Belgium, Luxembourg, Morocco and Ukraine. It offers consulting services in design, development and integration of open source solutions but also provides training sessions, technical support and hosting services. The main offers are grouped into the following categories:

- Digital
- Business Apps
- Embedded & IoT
- Outsourcing

2.1 Organizational structure

Smile is organized into autonomous entities called Business Units (BU), some offering products and services which target a specific market segment and some others that work internally for the company. Some examples of BUs are: Embedded and Connected Systems (ECS), Digital, Sales Outsourcing, Hosting, E-commerce and Recruiting, among others.

2.2 Open Source School

In February 2016, Smile launched the Open Source School (OSS) in partnership with EPSI, an engineering school. This is the first higher education school dedicated to open source technologies which offers a 3-year program (BAC+5) to train professionals to design, build and manage large open source projects.

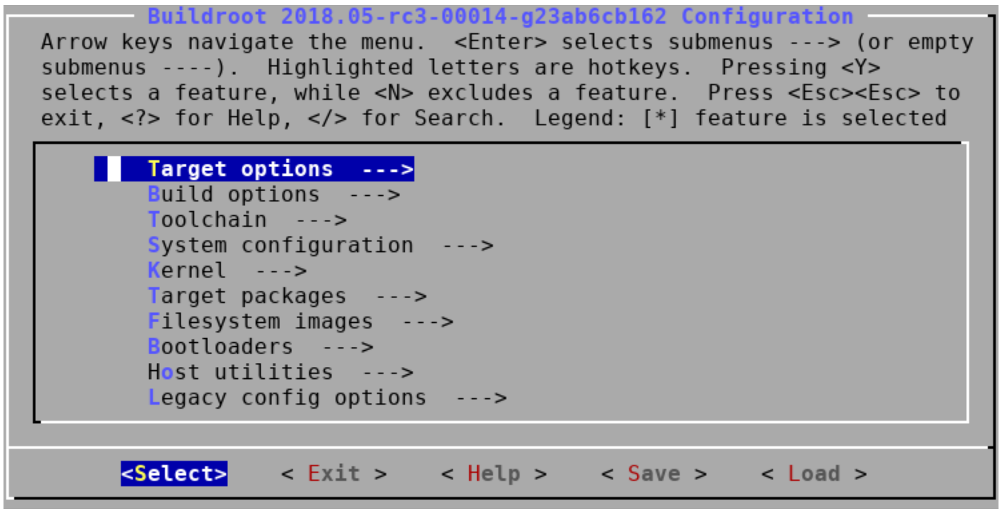
3 Buildroot

Buildroot is an open source build system that allows to build from source some or all of the following components of an Embedded Linux System:

- Cross-compilation toolchain
- Root filesystem
- Bootloader
- Kernel image

It is based on GNU Make and Kconfig, aiming to provide the user with a tool that generates a fully customizable Linux system, easily and in a few minutes. Currently, Buildroot supports many architectures, such as x86, ARM, AArch64, MIPS, PowerPC and Microblaze among others. There are more than 2200 packages available and the documentation explains in detail the process to add new ones.

Fig.1 shows the how the system can be configured by using `make menuconfig`:



```
Buildroot 2018.05-rc3-00014-g23ab6cb162 Configuration
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
selects a feature, while <N> excludes a feature.  Press <Esc><Esc> to
exit, <?> for Help, </> for Search.  Legend: [*] feature is selected

        Target options  --->
        Build options  --->
        Toolchain  --->
        System configuration  --->
        Kernel  --->
        Target packages  --->
        Filesystem images  --->
        Bootloaders  --->
        Host utilities  --->
        Legacy config options  --->

    <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 1: Buildroot

3.1 Cross-compilation toolchain

A toolchain is the set of tools that allows to compile source code into executables that can run on a target platform. A standard GNU toolchain consists of four main components:

- Binutils: set of binary utilities such as ld, as, objdump, readelf, etc.
- GNU Compiler Collection (GCC)
- C library: an API that allows user-space applications to interact with the Linux kernel
- Linux kernel headers: they contain definitions and constants needed to access the kernel directly

Buildroot uses cross-compilation, which means that the build environment is separated from the target environment. This approach has many benefits considering that the embedded target is generally slower than the host and sometimes it cannot even run a compiler. The main advantages are the following:

- Increasing productivity (host is faster than target)
- Build an application or a complete system for many platforms on the same machine
- Bootstraping a compiler on a new architecture

Buildroot offers the possibility of building an entire cross-toolchain from source or using a pre-compiled one. When choosing the first option, it is possible to customize every component of the toolchain: the C library (glibc, uclibc or musl), the version of GCC, binutils, kernel headers, and many other options.

3.2 Packages

Packages in Buildroot are stored in the `package` directory. Every package consists of at least three files:

- Config.in: contains the Kconfig code necessary to display the package and its options in the configuration tool. It is important to specify all the dependencies in this file.
- .mk file: a Makefile containing the instructions involved since the source code of the package is downloaded until it is finally installed to the target.
- .hash file: a file containing the hashes of the files downloaded by the package, such as its tarball and the license file.
- Optionally, it is possible to store patches which are applied before configuring the package.

As packages are based on different build systems, Buildroot offers several infrastructures to facilitate the integration of new packages. Among them, there are infrastructures for CMake, Autotools and Python-based packages. A generic package intrastructure is also available for packages not using any of the more common build systems.

3.3 Contributing to the project

There are four stable releases of Buildroot every year: in February, May, August and November. Each release has a tag with the following format: <year>.<month>, being 2018.05 the latest release at the time of writing this document. There is one long term support release every year, <year>.02, maintained during one year with security, bug and build fixes.

It is possible to contribute to the project by reviewing, testing and sending patches. After a patch is sent, it is discussed with the people on the mailing list, which will generally lead to doing some modifications. Finally, if the project maintainers consider that the patch is ready to be applied, it is commited to its corresponding branch.

4 LLVM

This section presents LLVM, an open source project that provides a set of low level toolchain components (assemblers, compilers, debuggers, etc.) which are designed to be compatible with existing tools typically used on Unix systems. While LLVM provides some unique capabilities and is known for some of its tools, such as the Clang compiler (C/C++/Objective-C compiler which provides a number of benefits with respect to the GCC compiler), the main thing that distinguishes LLVM from other compilers is its internal architecture.

4.1 The project

The LLVM (used to be an acronym of Low Level Virtual Machine but not anymore) project started in the year 2000 as a result of a masters thesis [1] written by Chris Lattner at the University of Illinois. This project is different from most traditional compiler projects (such as GCC) because it is not just a collection of individual programs, but rather a collection of libraries that can be used to build compilers, optimizers, JIT code generators and other compiler-related programs. LLVM is an umbrella project, which means that it has several subprojects, such as LLVM Core (main libraries), Clang, lldb, compiler-rt, libclc, and lld among others.

From the start, LLVM was conceived as an API which provides a compiler infrastructure written in C++, focusing on compile time and performance of the generated code. Thanks to this object-oriented design combined with a complete documentation, it is easy to integrate LLVM components into third-party projects, and this is in fact the key of the success achieved by this project.

Nowadays, LLVM is being used as a base platform to enable the implementation of statically and runtime compiled programming languages, such as C/C++, Java, Kotlin, Rust and Swift. This is possible because LLVM's internal structure implements many of the common structures and patterns found in most programming languages, allowing developers to focus only on the particular aspects.

Many big companies are using LLVM technology in their products, being Apple and Google the main supporters of the project. Below there are some examples:

- Apple:
  - All operating systems built with Clang
  - Xcode IDE uses Clang compiler and static analyzer by default
  - Swift uses LLVM as its compiler framework
- Google:
  - Builds Android user space and Chrome for all platforms with Clang
  - Android Renderscript compiler is based on LLVM
  - Kotlin programming language compiles directly to native code via LLVM

[1]https://llvm.org/pubs/2002-12-LattnerMSThesis.html

However, LLVM is not only being used as a traditional C/C++ toolchain but is gaining popularity in graphics. Such is the case of:

- llvmpipe (software rasterizer)
- CUDA (NVIDIA Compiler SDK based on LLVM)
- AMDGPU open source drivers
- Most of OpenCL implementations are based on Clang/LLVM

4.2 Internal aspects

Modern compiler design normally follows a three-phase approach, where the main components are: the frontend, the optimizer and the backend. Each phase is responsible for translating the input program into a different representation, making it closer to the target language. LLVM follows this approach and provides the optimizer and some backends, while frontends such as Clang live in separate projects. This sections describes each of these 3 components and highlights the main advantages of working with this model, focusing on LLVM IR (Intermediate Representation).

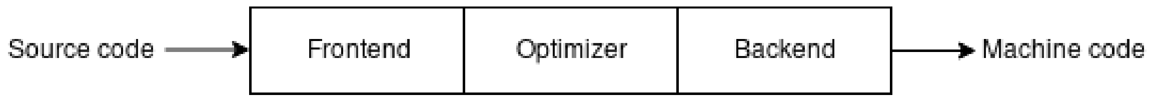


Figure 2: Three-phase compiler

4.2.1 Frontend

The frontend is the component in charge of validating the input source code, checking and diagnosing errors, and translating it in from its original language (eg. C/C++) to an intermediate representation (LLVM IR in this case) by doing lexical, syntactical and semantic analysis. Apart from doing the translation, the frontend can also perform optimizations that are language-specific.

4.2.2 LLVM IR

The LLVM IR is a complete virtual instruction set used throughout all phases of the LLVM compilation strategy, and has the main following characteristics:

- Mostly architecture-independent instruction set (RISC)
- Strongly typed
  - Single value types (eg. i8, i32, double)
  - Pointer types (eg. *i8, *i32)
  - Array types, structure types, function types, etc.
- Unlimited number of virtual registers in Static Single Assignment (SSA)
- Memory partitioned into global area, stack and heap
- Most operations are in three-address form

Intermediate Representation is the core of LLVM. It is fearly readable, as it was designed in a way that is easy for the frontends to generate but expressive enough to allow effective optimizations that produce fast code for real targets. This intermediate representation exists in three forms: a textual human-readable assembly format (.ll), an in-memory data structure and an on-disk binary "bitcode format" (.bc). LLVM provides tools to convert from from textual format to bitcode (llvm-as) and viceversa (llvm-dis). Below is an example of how LLVM IR looks like:

```c
int sum(int a, int b)
{
        return a+b;
}

int main()
{
        sum(1,2);
        return 0;
}
```

Listing 1: C example

```llvm
; ModuleID='main.c'
source_filename="main.c"
target datalayout="e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple="x86_64-buildroot-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define i32 @sum(i32, i32) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 @sum(i32 1, i32 2)
  ret i32 0
}
```

Listing 2: Equivalent code in LLVM IR

4.2.3 Optimizer

The strategy proposed by LLVM is designed to achieve high performance executables through a system of continuous optimization. Because all of the LLVM optimizations are modular (called passes), it is possible to use all of them or only a subset. There are Analysis Passes and Transformation Passes. The first ones compute some information about some IR unit (modules, functions, blocks, instructions) without mutating it and produce a result which can be queried by other passes. On the other hand, a Transformation Pass transforms a unit of IR in some way, leading to a more efficient code (also in IR). It must be noted that a transformation pass may depend on a previous analysis pass but it cannot depend on other transformation passes.

In general, the two main objectives of the optimization phase are improving the execution time of the program and reducing its code size. Every LLVM pass has a specific objective, such dead code elimination, constant propagation, combination of redundant instructions, dead argument elimination, and many others. The fact of using SSA form guarantees that each variable is defined only once, which helps a lot when performing this kind of optimizations.

The tool provided by LLVM to perform optimizations is called `opt`. It is possible to see all possible optimizations by executing `opt --help`:

```
-assumption-cache-tracker     - Assumption Cache Tracker
-atomic-expand                - Expand Atomic instructions
-barrier                      - A No-Op Barrier Pass
-basicaa                      - Basic Alias Analysis (stateless AA impl)
-basiccg                      - CallGraph Construction
-bdce                         - Bit-Tracking Dead Code Elimination
-block-freq                   - Block Frequency Analysis
-bounds-checking              - Run-time bounds checking
-branch-prob                  - Branch Probability Analysis
...
```

4.2.4 Backend

This component, also known as code generator, is responsible for translating a program in LLVM IR into optimized target-specific assembly. The main tasks carried out by the backend are register allocation, instruction selection and instruction scheduling.

Instruction selection is the process of translating LLVM IR operations into instructions available on the target architecture, taking advantage of specific hardware features that can lead to more efficient code. Register allocation involves mapping variables stored in the IR virtual registers onto real registers available in the target architecture, taking into consideration the calling convention defined in the ABI. Once these tasks and others such as memory allocation and instruction ordering are performed, the backend is ready to emit the corresponding assembly code, generating either a text file or an ELF object file as output.

4.2.5 Retargetability

The main advantage of the three-phase model adopted by LLVM is the possibility of reusing components, as the optimizer always works with LLVM IR. This eases the task of supporting new languages, as new frontends which generate LLVM IR can be developed while reusing the optimizer and backend. On the other hand, it is possible to bring support for more target architectures by writing a backend and reusing the frontend and the optimizer.

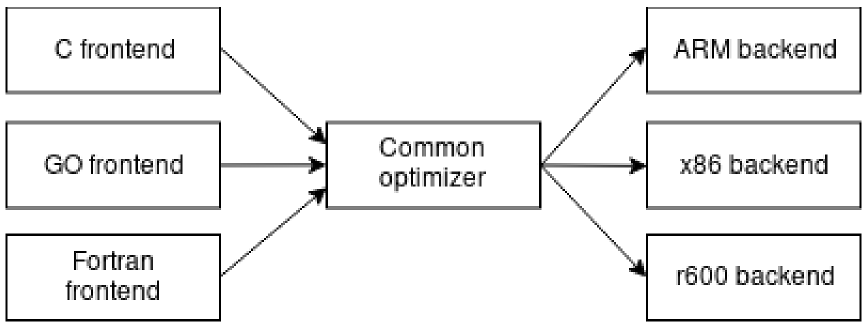


Figure 3: Retargetability

4.3 Clang

Clang is an open source compiler frontend for C/C++, Objective-C and OpenCL C for LLVM, therefore it can use LLVM's optimizer to produce efficient code. Since the start of its development in 2005, Clang has been focused on providing expressive diagnostics and an easy IDE integration. As LLVM, it is written in C++ and has a library-based architecture, which allows, for example, IDEs to use its parser to help developers with autocompletion and refactoring.

Clang was designed to offer GCC compatibility, so it accepts most GCC's command line arguments to specify the compiler options. However, GCC offers a lot of extensions to the standard language while Clang's purpose is being standard-compliant. Because of this, Clang cannot be a replacement for GCC when compiling projects that depend on GCC extensions, as is the case with Linux kernel. In this case, Linux does not build because Clang does not accept the following kinds of constructs:

- Variable length arrays inside structures
- Nested Functions
- Explicit register variables

Furthermore, Linux kernel still depends on GNU assembler and linker.

An interesting feature of Clang is that, as opposed to GCC, it can compile for multiple targets from the same binary, that is, it is a cross-compiler itself. Clang binary works as a driver, which means that it calls multiple binaries to control every phase of the compilation process, as shown in Fig.4:

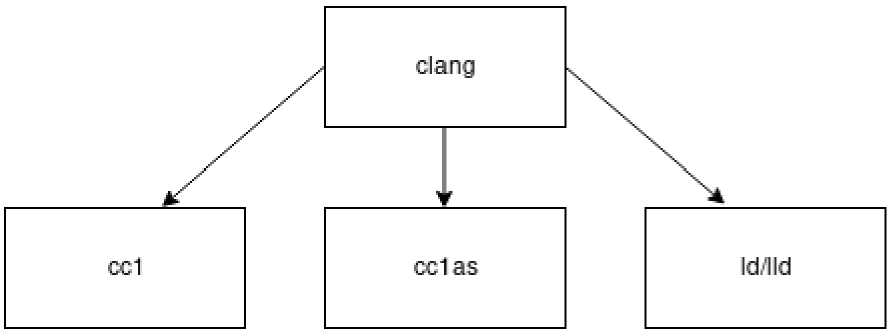


Figure 4: Clang driver

To control the target for which the code will be generated, it is necessary to specify the target triple in the command line by using the $--target =< triple >$ option. For example, `--target=armv7-linux-gnueabihf` corresponds to the following system:

- Architecture: arm
- Sub-architecture: v7
- Vendor:unknown
- OS: linux
- Environment: GNU

5 Linux graphics stack

This section intends to give an introduction to the Linux graphics stack in order to explain the role of LLVM inside this complex system comprised of many open source componentes that interact with each other. Fig. 5 shows all the components involved when 2D and 3D applications require rendering services from an AMD GPU:

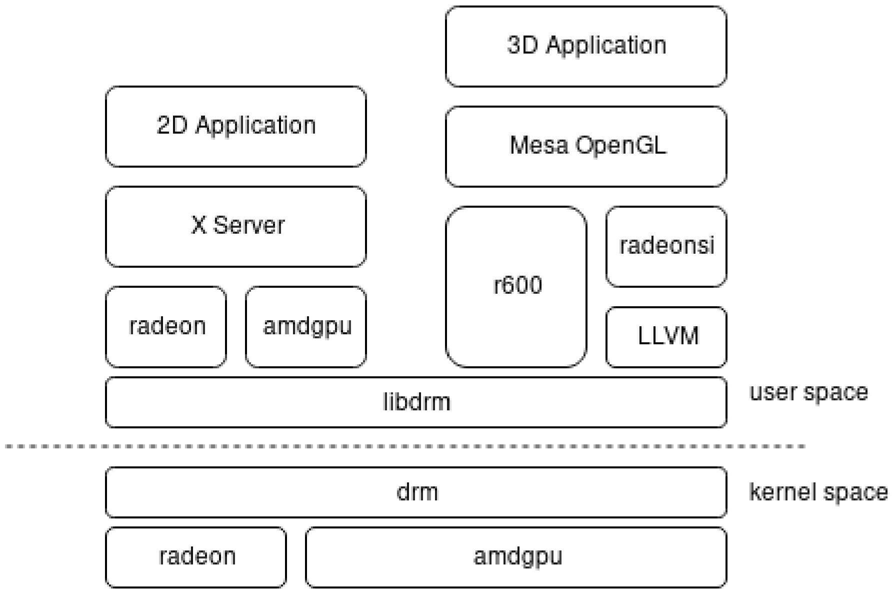


Figure 5: Typical Linux open source graphics stack for AMD GPUs

5.1 X Server

X Server is a software system that provides 2D rendering services to allow applications creating graphical user interfaces. It is based on a client-server architecture and exposes its services such as managing windows, displays and input devices through two shared libraries called Xlib and XCB. Given that X uses network client-server technology, it is not efficient when handling 3D applications due to its latency. Because of this, there exists a software system called Direct Rendering Infrastructure (DRI) which provides a faster path between applications and graphics hardware.

5.2 The DRI/DRM infrastructure

The Direct Rendering Infrastructure is a subsystem that allows applications using X Server to communicate with the graphics hardware directly. The most important component of DRI is the Direct Rendering Manager, which is a kernel module that provides multiple services:

- Initialization of GPU such as uploading firmwares or setting up DMA areas.
- Kernel Mode Setting(KMS): setting display resolution, colour depth and refresh rate.
- Multiplexing access to rendering hardware among multiple user-space applications.
- Video memory management and security.

DRM exposes all its services to user-space applications through libdrm. As most of these services are device-specific, there are different DRM drivers for each GPU, such as libDRM-intel, libDRM-radeon, libDRM-amdgpu, libDRM-nouveau, etc. This library is intended to be used by X Server Display Drivers (such as xserver-xorg-video-radeon, xserver-xorg-video-nvidia, etc.) and Mesa 3D, which provides an open source implementation of the OpenGL specification.

5.3 Mesa 3D

OpenGL is a specification that describes an API for rendering 2D and 3D graphics by exploiting the capabilities of the underlying hardware. Mesa 3D is a collection of open source user-space graphics drivers that implement a translation layer between OpenGL and the kernel-space graphics drivers and exposes the OpenGL API as libGL.so. Mesa takes advantage of the DRI/DRM infrastructure to access the hardware directly and output its graphics to a window allocated by the X server, which is done by GLX, an extension that binds OpenGL to the X Window System.

Mesa provides multiple drivers for AMD, Nvidia and Intel GPUs and also provides some software implementations of 3D rendering, useful for platforms that do not have a dedicated GPU. Mesa drivers are divided in two groups: Messa Classics and Gallium 3D. The second group is a set of utilities and common code that is shared by multiple drivers, such as nouveau (Nvidia), RadeonSI (AMD GCN) and softpipe (CPU).

As shown in Fig.6, LLVM is used by llvmpipe and RadeonSI, but it can optionally be used by r600g if OpenCL support is needed. The llvmpipe is a multithreaded software rasterizer uses LLVM to do JIT compilation of GLSL shaders. Shaders, point/line/triangle rasterization and vertex processing are implemented in LLVM IR, which is then translated to machine code. Another much more optimized software rasterizer is OpenSWR, which is developed by Intel and targets x86_64 processors with AVX or AVX2 capabilities. Both llvmpipe and OpenSWR present a much faster alternative to the classic Mesa's single-threaded softpipe.

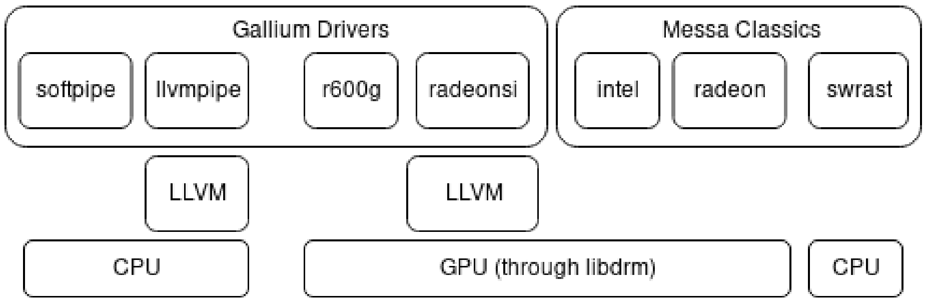


Figure 6: Mesa 3D drivers

6 LLVM/Clang integration to Buildroot

The main purpose of this internship is to integrate LLVM/Clang packages to Buildroot. These packages will activate new functionalities such as enabling Mesa 3D's llvmpipe software rasterizer (useful for systems which do not have a dedicated GPU) and providing OpenCL support for packages which are already available in Buildroot. Once LLVM is present on the system, new packages that rely on this infrastructure can be added. When this part of the project is achieved, a next step would be creating a cross-compilation toolchain based on Clang to compile Buildroot components supported by this front-end.[2]

State of the project - 2 March 2018

After some research concerning the state of the art of the LLVM project, the objectives of the internship were presented and discussed at the Buildroot Developers Meeting in Brussels[3], obtaining the following conclusions:

- LLVM itself is very useful for other packages (Mesa 3D's llvmpipe or OpenJDK's Jit compiler).
- It is questionable whether there is a need for Clang in Buildroot, as GCC is still needed and it has mostly caught up with Clang regarding performance, diagnostics and static analysis. It would be possible to build a complete userspace but some packages may break.
- LLVM does not have a stable API between major releases, so only these releases can be used.
- It could be useful to have a host-clang package that is user selectable.
- The long-term goal is to have a complete clang-based toolchain.

The first patch series aims only to activate LLVM support for Mesa 3D, and is divided into the following 3 patches:

- package/llvm: new host package
- package/llvm: enable target variant
- package/mesa3d: enable llvm support

It must be considered that with respect to the RFC series, [4] AMDGPU target support was removed and it will be added once it can be tested. Currently, the supported targets are x86, ARM and AArch64, and llvm.mk ensures that only the necessary target backends are built.

[2]Mainline Linux kernel and glibc do not yet compile with Clang
[3]https://elinux.org/Buildroot:DeveloperDaysFOSDEM2018
[4]http://lists.busybox.net/pipermail/buildroot/2017-July/196163.html

Considerations

LLVM Makefile

In order to cross-compile LLVM for the target, llvm-config and llvm-tblgen tools must first be compiled for the host. In the first patch series, a minimal version of host-llvm containing only these two tools is provided. To do this, most of the `HOST_LLVM_CONF_OPTS` are set to OFF. However, this does not avoid building LLVM libraries, which takes around one hour on a recent machine. To avoid this and build only the necessary tools:
`HOST_LLVM_MAKE_OPTS = llvm-tblgen llvm-config`

Things that need to be considered when cross-compiling LLVM:

- Path to host's llvm-tblgen: `-DLLVM_TABLEGEN`
- Specify that it is a cross-compilation: `-DCMAKE_CROSSCOMPILING`
- Default target triple: `-DLLVM_DEFAULT_TARGET_TRIPLE`
- Host triple (native code generation for the target): `-DLLVM_HOST_TRIPLE`
- Target architecture: `-DLLVM_TARGET_ARCH`
- Targets to build: `-DLLVM_TARGETS_TO_BUILD`

The result of the compilation will be one shared library containing all LLVM libraries called libLLVM.so, as `-DLLVM_BUILD_LLVM_DYLIB` is set to ON.

One important step in the process is the fact of replacing llvm-config in STAGING_DIR by its host variant. This is because llvm-config compiled for the target cannot run on the host, and this tool is needed to build applications that use LLVM libraries, as it prints the compiler flags, linker flags and object libraries needed to link against LLVM.

Mesa 3D

Currently, Mesa 3D is statically linking against LLVM libraries. When setting the option `MESA3D_CONF_OPTS += --enable-llvm-shared-libs`, the build fails because it cannot find LLVM libraries. Apparently, the problem is that llvm-config placed in STAGING_DIR is not working properly, as it provides the following output to this commands:

- `./llvm-config --shared-mode`
  `static`
- `./llvm-config --link-shared`
  `llvm-config: error: libLLVM-5.0.so is missing`
- `./llvm-config --libnames`
  `libLLVMLTO.a libLLVMPasses.a libLLVMObjCARCOpts.a...`

Even if llvm-config returns the correct lib directory, it assumes it has to use LLVM static libraries, and as the configure script from Mesa 3D calls `llvm-config --link-shared --libs` (in case `--enable-shared-libs` is activated) the build fails. Mesa's configure script clearly states that llvm-config may not give the correct output when LLVM is built as a single shared library.

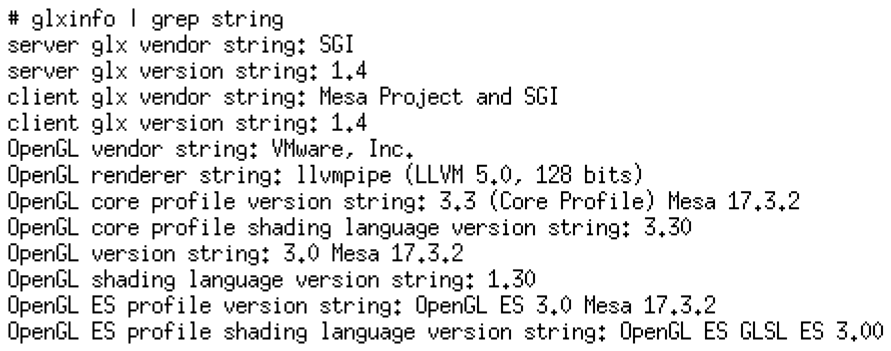
Achievements

At this date, Mesa 3D's llvmpipe was successfully tested on the following systems:

- x86_64
- ARM
- AArch64

x86_64

The tests for x86_64 were done using an AMD A4-3300M microprocessor. The built system uses a Linux kernel 4.9, X window system and works correctly with OpenGL. During this test it was possible to appreciate the better performance provided by llvmpipe with respect to softpipe.



```
# glxinfo | grep string
server glx vendor string: SGI
server glx version string: 1.4
client glx vendor string: Mesa Project and SGI
client glx version string: 1.4
OpenGL vendor string: VMware, Inc.
OpenGL renderer string: llvmpipe (LLVM 5.0, 128 bits)
OpenGL core profile version string: 3.3 (Core Profile) Mesa 17.3.2
OpenGL core profile shading language version string: 3.30
OpenGL version string: 3.0 Mesa 17.3.2
OpenGL shading language version string: 1.30
OpenGL ES profile version string: OpenGL ES 3.0 Mesa 17.3.2
OpenGL ES profile shading language version string: OpenGL ES GLSL ES 3.00
```

Figure 7: OpenGL specs

Some benchmarks were run in order to compare llvmpipe against Mesa 3D's classic softpipe software rasterizer and also against the AMD Radeon HD6480. Table 1 shows how much the LLVM code optimizer improves rendering performance:

Table 1: Results of GLMark2 and GLMark2-es2 benchmarks on x86_64

	GLMark2	GLMark2-es2
Radeon HD6480	156	156
llvmpipe	47	52
softpipe	3	3

ARM

In order to test LLVM for ARM architecture, Raspberry Pi 2 and Raspberry Pi 3 development boards were used. For the case of the Raspberry Pi 3, the 32-bit defconfig was selected. Raspberry Pi only supports OpenGL ES, so only GLMark2-es2 could be tested.

Table 2: Raspberry Pi 2 and 3 Hardware Specifications

Board	Family	SoC	CPU	GPU
RPi 2	BCM2709	BCM2836 @ 900 MHz	ARMv7 Cortex-A7 (Quad Core)	VC4
RPi 3	BCM2710	BCM2837 @ 1.2 GHz	ARMv8 Cortex-A53 (Quad Core)	VC4

When trying to execute `glmark2` the following errors are obtained:

```
Error:  GLX version >= 1.3 is required

Error:  Error:  Couldn't get GL visual config

Error:  main:  Could not initalize canvas
```

Table 3: Results of GLMark2-es2 for ARM

	GLMark2-es2
RPi2 softpipe	0
RPi2 llvmpipe	0
RPi3 (32-bit) softpipe	0
RPi3 (32-bit) llvmpipe	11

Table 3 shows an improvement in rendering when LLVM is used, and also the higher computing power of the Cortex-A53 microprocessor with respect to the Cortex-A7.

AArch64

Buildroot offers a defconfig to install a 64-bit system on the Raspberry Pi 3 (raspberrypi3_64_defconfig). There is a little improvement in rendering with respect to the 32-bit version:

Table 4: Results of GLMark2-es2 for AArch64

	GLMark2-es2
RPi3 (64-bit) softpipe	0
RPi3 (64-bit) llvmpipe	13

Considerations

- By default, the defconfigs for Raspberry Pi present in Buildroot have /dev management set to `Dynamic using devtmps only`. This must be changed to `Dynamic using devtmps + eudev` in order to allow Linux kernel to load modules dyamically, such as the VC4 device driver.
- To load VC4 device driver, assuming that the `/boot` partition has the `overlays/` directory with its dtbo files inside, the next options must be configured:
  - Add cma=256M to cmdline.txt
  - Set gpu_mem/gpu_mem_1024 to 256 in config.txt
  - Add dtoverlay=vc4-kms-v3d to config.txt

These steps allow to load the VC4 driver correctly, however it is not yet working well with X. When trying to execute any `glx` command, such as `glxinfo` or `glxgears`, it returns the following error:

```
Error:  couldn't find RGB GLX visual or fbconfig
```

Possible causes:

- Mesa is not installing libglx.so in `/usr/lib/xorg/modules/extensions/`.
- Mesa 3D package in Buildroot states that a vanilla kernel 4.5+ must be used with Gallium VC4 (defconfig uses kernel from raspberrypi's Github). However, even in this case or using Eric Anholt's kernel[5] the error persists.

Next steps

- Enable dynamic linking for Mesa 3D. This is important because when building packages that link against LLVM libraries the same problem may arise.
- For Raspberry Pi:
  - Activate glx.
  - Activate OpenGL for VC4.
- Prepare next patch series:
  - Provide an option to do a full installation of host-llvm.
  - Activate OpenCL.
  - Add Clang package (needs full host-llvm installed).
  - Add support for more targets.

[5]https://github.com/anholt/mesa/wiki/VC4-complete-Raspbian-upgrade

Update - 9 March 2018

Full host-llvm

After having investigated why Mesa 3D was not able to link dynamically against libLLVM.so, it was found that the bug in llvm-config presented in section **Mesa 3D** occurs when the option LLVM_LINK_LLVM_DYLIB is not enabled. The purpose of this option is to generate a single shared library (libLLVM.so) and dynamically link LLVM tools with it.

A priori, as llvm-tblgen and llvm-config are the only necessary tools for the host (llvm-tblgen to cross-compile LLVM for the target and llvm-config to provide linking options to packages that link against LLVM libraries), it was decided to do a minimal host-llvm installation. However, to get the correct output from llvm-config, it must be linked with libLLVM.so (host-variant), so this library must also be built. Because of this, the first approach changed and the first patch of the series (package/llvm: new host package) will provide a full installation of LLVM (tools and libraries). This approach will avoid conflicts for packages linking with LLVM libraries and will also facilitate the integration of Clang front-end, which will be provided in a future patch series.

PATCH v3

The PATCH v3 series[6] sent to the Buildroot mailing list on the 9th March contains the following 6 commits:

- [PATCH v3 1/6] package/llvm: new host package
- [PATCH v3 2/6] package/llvm: enable target variant
- [PATCH v3 3/6] package/llvm: enable AMDGPU
- [PATCH v3 4/6] package/mesa3d: enable llvm support
- [PATCH v3 5/6] package/llvm: enable ARM
- [PATCH v3 6/6] package/llvm: enable AArch64

This series provides LLVM backends for x86, ARM, AArch64 and AMDGPU (R600 to GCN). With respect to the previous version, host-llvm is entirely built because of the reasons explained above.

[6]http://lists.busybox.net/pipermail/buildroot/2018-March/215490.html

Update - 29 March 2018

New series to enable OpenCL

Once LLVM was tested working on the three more common architectures (x86, ARM and Aarch64), the next goal was activating OpenCL support. This task involved multiple steps, as there are many dependencies which need to be satisfied.

OpenCL is an API enabling general purpose computing on GPUs (GPGPU) and other devices (CPUs, DSPs, FPGAs, ASICs, etc.), being well suited for certain kinds of parallel computations, such as hash cracking (SHA, MD5, etc.), image processing and simulations.

OpenCL presents itself as a library with a simple interface:

- Standarized API headers for C and C++
- The OpenCL library (libOpenCL.so), which is a collection of types and functions which all conforming implementations must provide.

The standard is made to provide many OpenCL platforms on one system, where each platform can see various devices. Each device has certain compute characteristics (number of compute units, optimal vector size, memory limits, etc). The OpenCL standard allows to load OpenCL kernels which are pieces of C99-like code that is JIT-compiled by the OpenCL implementations (most of them rely on LLVM to work), and execute these kernels on the target hardware. Functions are provided to compile the kernels, load them, transfer data back and forth from the target devices, etc.

There are multiple open source OpenCL implementations for Linux:

- **Clover (Computing Language over Gallium)**

  It is a hardware independent OpenCL API implementation that works with Gallium Drivers (hardware dependent userspace GPU drivers) which was merged into Mesa 3D in 2012. It currently supports OpenCL 1.1 and it is close to 1.2. It has the following dependencies:

  - **libclang**: provides an OpenCL C compiler frontend and generates LLVM IR.
  - **libLLVM**: LLVM IR optimization passes and hardware dependent code generation.
  - **libclc**: implementation of the OpenCL C standard library in LLVM IR bitcode providing device builtin functions. It is linked at runtime.

- **Pocl**

  This implementation is OpenCL 1.2 standard compliant and supports some 2.0 features. The major goal of this project is to improve performance portability of OpenCL programs, reducing the need for target-dependent manual optimizations. Pocl currently supports many CPUs (x86, ARM, MIPS, PowerPC), ASPIs(TCE/TTA), NVIDIA GPUs via CUDA (experimental), HSA-supported GPUs and multiple private off-tree targets. It also works with libclang and libLLVM but it has its own Pocl Builtin Lib (instead of using libclc).

- **Beignet**

  It targets Intel GPUs (HD and Iris) starting with Ivy Bridge, and offers OpenCL 2.0 support for Skylake, Kaby Lake and Apollo Lake.

- **ROCm**

  This implementation by AMD targets ROCm (Radeon Open Compute) compatible hardware[7] (HPC/Hyperscale), providing OpenCL 1.2 API with OpenCL C 2.0. It has become open source in May 2017.

Table 5: Open source OpenCL implementations

Project	Version	Hardware
Clover	1.1	AMD
Pocl	1.2	CPU, NVIDIA[8], AMD[9], TCE/TTA
Beignet	2.0	Intel
ROCm OpenCL	1.2	AMD[10]

Because of this fragmentation concerning OpenCL implementations (without taking into account the propietary ones) there exists a program that allows multiple implementations to co-exist on the same sytem: OpenCL ICD (Installable Client Driver). It needs the following components to work:

- **libOpenCL.so (ICD loader)**: this library dispatches the OpenCL calls to OpenCL implementations.
- **/etc/OpenCL/vendors/*.icd**: these files tell the ICD loader which OpenCL implementations (ICDs) are installed on the sytem. Each file has a single line containing the name of the shared library with the implementation.
- **One or more OpenCL implementations (the ICDs)**: the shared libraries pointed by the .icd files.

[7]https://github.com/RadeonOpenCompute/ROCm
[8]Needs propietary drivers
[9]HSA compatible hardware
[10]ROCm compatible hardware

Preparation of the new series

Considering that the available system for tests has an AMD Radeon Dual Graphics GPU (integrated HD6480G + dedicated HD7450M) and that Mesa 3D is already present in Buildroot, it was decided to work with the OpenCL implementation provided by Clover. The diagram in Fig.8 shows which are the necessary components to set up the desired OpenCL environment and how they interact with each other.

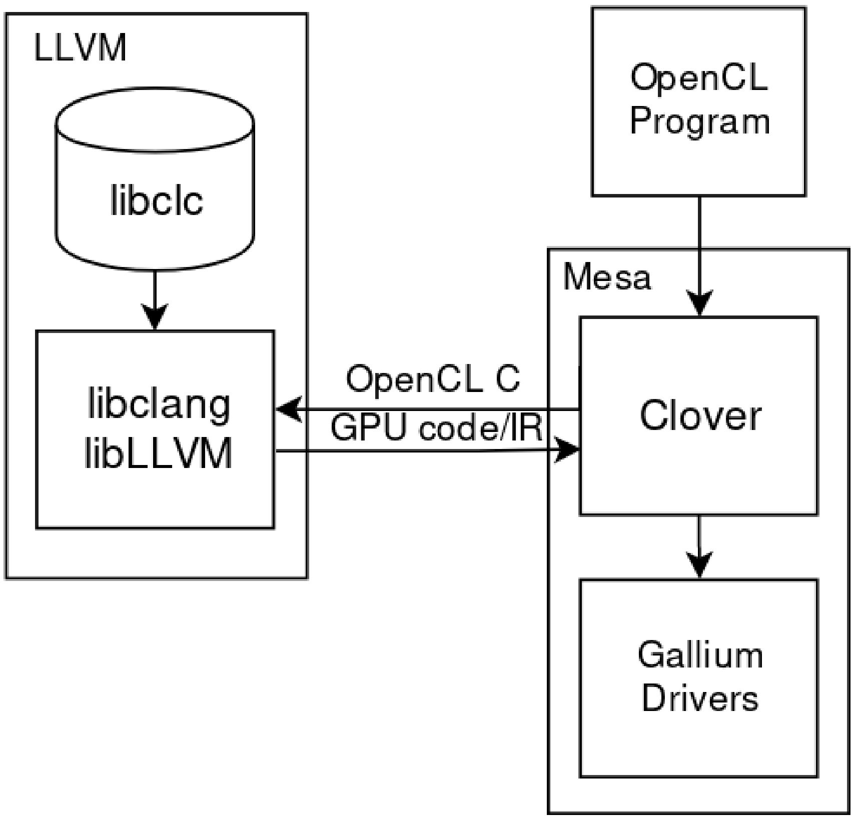


Figure 8: Clover OpenCL implementation

Clang for host

The first step was providing Clang package for the host, as it is necessary to build libclc because this library is written in OpenCL C and some functions are implemented directly in LLVM IR. Clang will transform .cl and .ll source files into LLVM IR bitcode (.bc) by calling llvm-as (the LLVM assembler).

Regarding the Makefile for building host-clang, the path to host's llvm-config must be specified. This is necessary because Clang is thought to be built as a tool inside LLVM's tree (LLVM_SOURCE_TREE/tools/clang) but Buildroot manages packages individually, so Clang's code source cannot be downloaded inside LLVM's tree.

Having Clang installed on the host is not only useful for building libclc, it also provides an alternative to GCC, which enables the possibility of creating a new toolchain based on it.

Clang for target

When trying to cross-compile Clang some problems were encountered, so it was decided to work with ARM architecture in order to make sure that a build on x86 was successful not just because of binary compatibility. The main issues were the following ones:

- **llvm-tblgen**

  When trying to cross-compile Clang, the build broke with the following error:

  ```
  llvm-tblgen: cannot execute binary file: Exec format error
  ```

  This means that llvm-tblgen from STAGING_DIR (cross-compiled) was trying to be executed on the host machine. Because of this, it was necessary to copy llvm-tblgen from host to STAGING_DIR/usr/bin. This is the same kind of problem that arised with llvm-config, which was explained before.

- **llvm-config**

  It is necessary to specify the path to llvm-config installed in STAGING_DIR:

  ```
  -DLLVM_CONFIG:FILEPATH=$(STAGING_DIR)/usr/bin/llvm-config
  ```

- **Shared libs**

  When Clang was built for the host, it generated multiple static libraries (libclangAST.a, libclangFrontend.a, libclangLex.a, etc.) and finally a shared object (libclang.so) containing all of them. However, when building for the target, it produced multiple shared libraries and finally libclang.so. This resulted in the following error when trying to use software that links against libOpenCL, which statically links with libclang (e.g, clinfo):

  ```
  $ CommandLine Error: Option 'track-memory' registered more than
  once!
  $ LLVM ERROR: inconsistency in registered CommandLine options
  ```

  The solution to this was specifying explicitely to the CMake infrastructure that shared libraries should not be built:

  ```
  CLANG_CONF_OPTS += -DBUILD_SHARED_LIBS=OFF
  ```

libclc

This library provides an implementation of the library requirements of the OpenCL C programming language, as specified by the OpenCL 1.1 specification. It is designed to be portable and extensible, as it provides generic implementations of most library requirements, allowing targets to override them at the granularity of individual functions, using LLVM intrinsics for example. It currently supports AMDGCN, R600 and NVPTX targets.

There is a particular problem with libclc: when OpenCL programs call clBuildProgram function[11] in order to compile and link a program (generally an OpenCL kernel) from source during execution, they require clc headers to be located in /usr/include/clc. This is not possible because Buildroot removes /usr/include from the target as the embedded platform is not intended to store development files, mainly because there is no compiler installed on it. But since OpenCL works with libLLVM to do code generation, a place to store clc headers must be found.

The file that adds the path to libclc headers is invocation.cpp, located at src/gallium/state_trackers/clover/llvm, inside Mesa's source tree:

```
// Add libclc generic search path
c.getHeaderSearchOpts().AddPath(LIBCLC_INCLUDEDIR,
                                clang::frontend::Angled,
                                false, false);
// Add libclc include
c.getPreprocessorOpts().Includes.push_back("clc/clc.h");
```

Listing 3: Extract from invocation.cpp

Variable LIBCLC_INCLUDEDIR is defined in Mesa's configure.ac:

```
LIBCLC_INCLUDEDIR=`$PKG_CONFIG --variable=includedir libclc`
LIBCLC_LIBEXECDIR=`$PKG_CONFIG --variable=libexecdir libclc`
```

Listing 4: Extract from configure.ac

Currently, header files are being copied to /usr/include/clc once the root filesystem is generated, but this solution only works for testing because systems generated with Buildroot must work directly after being built. The next step is to test if LIBCLC_INCLUDEDIR can be overwritten by specifying another path instead of using pkg-config.

clinfo

Clinfo is a simple command-line application that enumerates all possible (known) properties of the OpenCL platform and devices available on the system. It tries to output all possible information, including those provided by platform-specific extensions.

This application is built with a simple Makefile, so when creating the package for Buildroot it was sufficient to call the generic-package infrastructure. The main purposes of it are:

- Verifying that the OpenCL environment is set up correctly. If clinfo cannot find any platform or devices (or fails to load the OpenCL dispatcher library), chances are high no other OpenCL application will run.
- Verifying that the OpenCL development environment is set up correctly: if clinfo fails to build, chances are high that no other OpenCL application will build.
- Reporting the actual properties of the available devices.

Once installed on the target, clinfo successfully found Clover and the devices available to work with, providing the following output:

[11] https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml/clBuildProgram.html

```
Number of platforms                      1
Platform Name                            Clover
Platform Vendor                          Mesa
Platform Version                         OpenCL 1.1 Mesa 17.3.7
Platform Profile                         FULL_PROFILE
Platform Extensions                      cl_khr_icd
Platform Extensions function suffix      MESA

Platform Name                            Clover
Number of devices                        2
Device Name                              AMD SUMO (DRM 2.50.0 / 4.14.0, LLVM 5.0.1)
Device Vendor                            AMD
Device Vend                              0x1002
Device Version                           OpenCL 1.1 Mesa 17.3.7
Driver Version                           17.3.7
Device OpenCL C Version                  OpenCL C 1.1
Device Type                              GPU
Device Profile                           FULL_PROFILE
Device Available                         Yes
Compiler Available                       Yes
Max compute units                        1
Max clock frequency                      0MHz
Max work item dimensions                 3
Max work item sizes                      256x256x256
Max work group size                      256
Preferred work group size multiple       64
```

Listing 5: Output of clinfo

Piglit

Piglit is a collection of automated tests for OpenGL and OpenCL implementations. The goal of this project is to help improving the quality of open source OpenGL and OpenCL drivers by providing developers with a simple means to perform regression tests.

Once Clover was installed on the target system, it was decided to run Piglit in order to verify Mesa's OpenCL implementation conformance, taking the packaging for Buildroot from Romain Naour's series [12]:

- [PATCH v2 1/3] package/python-numpy: add host variant for piglit
- [PATCH v2 2/3] package/waffle: new package
- [PATCH v2 3/3] package/piglit: new package

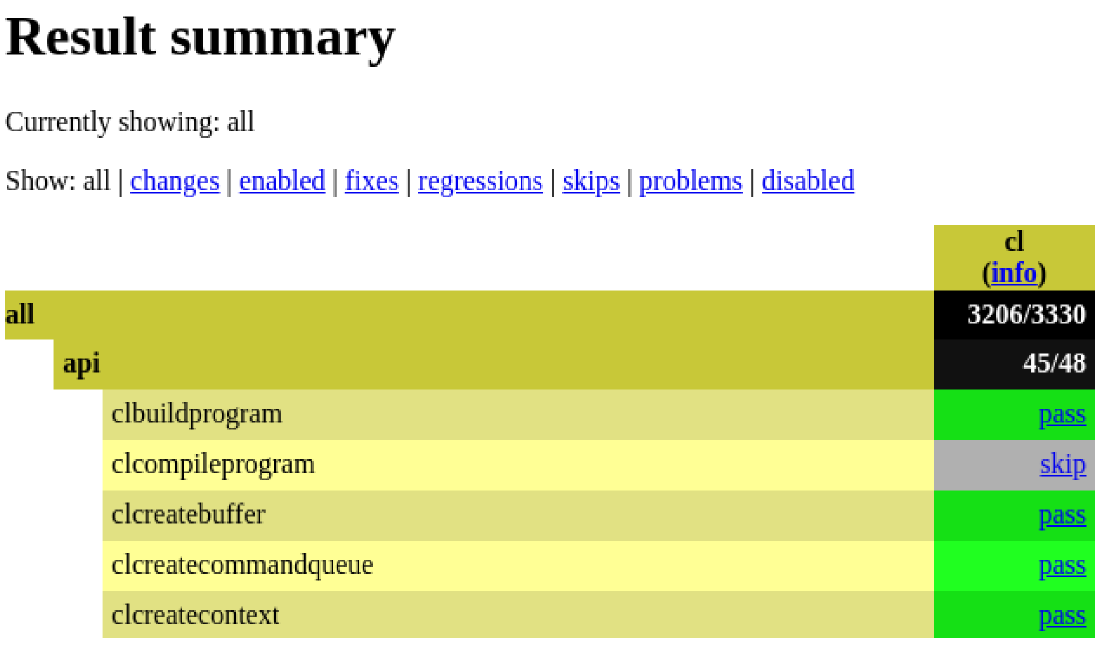
To run the OpenCL test suite, the following command must be executed:

```
piglit run tests/cl results/cl
```

The results are written in JSON format, and can be converted to HTML by running:

```
piglit summary html --overwrite summary/cl results/cl
```

[12]http://lists.busybox.net/pipermail/buildroot/2018-February/213601.html


Result summary

Currently showing: all

Show: all | changes | enabled | fixes | regressions | skips | problems | disabled

	cl (info)
all	3206/3330
api	45/48
clbuildprogram	pass
clcompileprogram	skip
clcreatebuffer	pass
clcreatecommandqueue	pass
clcreatecontext	pass

Figure 9: OpenCL Test Suite results in HTML

The results of the OpenCL test suite were the following ones:

Table 6: Piglit OpenCL Test Suite on AMD SUMO + AMD CAICOS

Total	Skip	Pass	Fail	Crash
704	94	541	60	9

Most of the tests that failed can be classified in the following categories:

- Program build with optimization options for OpenCL C 1.0/1.1+
- Global atomic operations (add, and, or, max, etc.) using a return variable
- Floating point multiply-accumulate operations
- Some builtin shuffle operations
- Global memory
- Image read/write 2D
- Tail calls
- Vector load

Some failures are due to missing hardware support for particular operations, so it would be useful to run Piglit with a more recent GPU using RadeonSI Gallium driver in order to compare the results. It would also be interesting to test with both GPUs which packages can benefit from OpenCL support using Clover.

PATCH v4

The PATCH v4 series[13] sent to the Buildroot mailing list on the 29th March contains the following 11 commits:

- [PATCH v4 1/11] package/llvm: new host package
- [PATCH v4 2/11] package/llvm: enable target variant
- [PATCH v4 3/11] package/llvm: enable AMDGPU
- [PATCH v4 4/11] package/mesa3d: enable llvm support
- [PATCH v4 5/11] package/llvm: enable ARM
- [PATCH v4 6/11] package/llvm: enable AArch64
- [PATCH v4 7/11] package/clang: new host package
- [PATCH v4 8/11] package/clang: enable target variant
- [PATCH v4 9/11] package/libclc: new package
- [PATCH v4 10/11] package/mesa3d: enable OpenCL support
- [PATCH v4 11/11] package/clinfo: new package

This series presents some improvements of the patches sent in the previous version and adds the necessary packages to enable OpenCL support for AMD GPUs: Clang and libclc. Clinfo package is also included in this series as it is a means to check whether Clover is correctly set up.

[13] http://lists.busybox.net/pipermail/buildroot/2018-March/216772.html

Update - 13 April 2018

A Buildroot hackathon gathering the core developers of the project took place during the March 31-April 2 weekend in Paris. After an extensive review of the v4 series by Romain Naour and Thomas Petazzoni, the following feedback was received:

- There is no need to have a visible Config.in.host option for host-llvm, as it is merely needed as a build dependency of the target llvm.
- Activate CCACHE, considering that Buildroot has CCACHE support and it is useful considering the size of LLVM.
- LLVM needs a toolchain with thread and C++ support.
- Some options are already passed by the CMake package infrastructure of Buildroot, so they are not necessary in llvm.mk, such as CMAKE_INSTALL_PREFIX and -G "Unix Makefiles".
- Manage LLVM_TARGETS_TO_BUILD in a more extensible way to add more back-ends.
- Support for ARM and AArch64 architectures should go directly in the first patch of the series.
- Clang binaries must be removed from the target, as there are no development files (headers) and other build tools.

Taking into account all these considerations, the next version of the series was prepared.

PATCH v5

The PATCH v5 series[14] sent to the Buildroot mailing list on the 4th April contains the following 7 commits:

- [PATCH v5 1/7] package/llvm: new package
- [PATCH v5 2/7] package/llvm: enable AMDGPU
- [PATCH v5 3/7] package/mesa3d: enable llvm support
- [PATCH v5 4/7] package/clang: new package
- [PATCH v5 5/7] package/libclc: new package
- [PATCH v5 6/7] package/mesa3d: enable OpenCL support
- [PATCH v5 7/7] package/clinfo: new package

[14] http://lists.busybox.net/pipermail/buildroot/2018-April/218023.html

Achievements

LLVM package[15] and LLVM support for Mesa 3D[16] were commited to Buildroot's master branch on the 4th April.

Bug fixing

Thanks to Buildroot autobuilders[17] it was possible to detect some bugs that were not found during development. These autobuilders are machines that select a target architecture, a toolchain, some packages randomly and try to build the configuration. There is a daily report containing the results of the autobuilders that shows which packages failed to build for a particular configuration. It is also possible to analyze the build log and the .config file so that bugs can be quickly corrected.

GCC Bug 64735

Autobuild:

http://autobuild.buildroot.net/results/ada497f6a8d20fa1a9adb2b17a138d7b726a6cdc/

Extract from build-end.log:

```
output/build/llvm-5.0.1/lib/Support/ThreadPool.cpp:14:0:
output/build/llvm-5.0.1/include/llvm/Support/ThreadPool.h: In member function
'std::shared_future<void> llvm::ThreadPool::async(Function&&, Args&& ...)':
output/build/llvm-5.0.1/include/llvm/Support/ThreadPool.h:54:75: error:
return type 'class std::shared_future<void>' is incomplete inline
std::shared_future<void> async(Function &&F, Args &&... ArgList)
```

Fix (Thomas Petazzoni):

This autotest was targeting an ARM926EJ-S processor (ARMv5 architecture). LLVM uses std::shared_future, which until gcc 7.x is not available on architectures that do not provide lock-free atomics: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=64735. Buildroot already has a BR2_TOOLCHAIN_HAS_GCC_BUG_64735 option to handle such a case, so this new dependency must be added to LLVM. It will make sure LLVM does not get built on ARMv5 platforms using a GCC older than 7.x.

Commit: http://lists.busybox.net/pipermail/buildroot/2018-April/218267.html

Shared libraries

Autobuild:

http://autobuild.buildroot.net/results/301c454c6eab802405a268f4713a574d1c366892/

Extract from build-end.log:

```
Linking CXX shared library ../../lib/libLTO.so
arm-buildroot-linux-musleabihf/bin/ld: attempted static link of dynamic object
`../../lib/libLLVM-5.0.so' collect2: error: ld returned 1 exit status
```

[15]http://lists.busybox.net/pipermail/buildroot/2018-April/218058.html

[16]http://lists.busybox.net/pipermail/buildroot/2018-April/218060.html

[17]http://autobuild.buildroot.org/

Fix:

Buildroot provides an option to build and use only static libraries on the target system. LLVM will not work in this case as it generates shared libraries. Because of this, the package should not be available if BR2_STATIC_LIBS is set.

Commit: http://lists.busybox.net/pipermail/buildroot/2018-April/218550.html

BR2_USE_WCHAR

Autobuild:

This error was detected locally.

Extract from log:

```
output/build/llvm-5.0.1/include/llvm/Support/ConvertUTF.h:203:53:
error:   std::wstring  has not been declared
bool ConvertUTF8toWide(llvm::StringRef Source, std::wstring &Result);
```

Fix:

LLVM uses std::wstring, so a toolchain with wchar support is necessary.

Commit: http://lists.busybox.net/pipermail/buildroot/2018-April/218549.html

Gallium R600 with LLVM needs libelf

Autobuild:

http://autobuild.buildroot.org/results/8845ff0f28d3273ebe884126b85cd7c4a905d81b/

Extract from log:

```
checking for EXPAT... yes
checking for RADEON... yes
configure: error: r600 requires libelf when using llvm
```

Fix:

Gallium R600 driver needs libelf when Mesa 3D is built with LLVM support. Because of this, the toolchain must use either uClibc or glibc, as musl is not currently compatible with elfutils.

Commit: http://lists.busybox.net/pipermail/buildroot/2018-April/218985.html

llvm-config's RPATH

Autobuild:

http://autobuild.buildroot.net/results/b81c12d529c66a028e2297ea5ce1d6930324fa69/

Extract from log:

```
checking for llvm-config...
output/host/x86_64-buildroot-linux-uclibc/sysroot/usr/bin/llvm-config
/output/host/x86_64-buildroot-linux-uclibc/sysroot/usr/bin/llvm-config:
error while loading shared libraries: libc.so.0: cannot open shared object
file: No such file or directory
```

Fix:

In this case, Mesa 3D failed to build because it could not correctly execute llvm-config. The problem is the following: llvm-config (host version installed in STAGING_DIR) is not being able to link correctly with the libc of the host system. This happens in the following scenario: target architecture = host architecture (normally x86_64) and target's libc different from host's libc (normally glibc).

As the RPATH of llvm-config specifies $ORIGIN/../lib (seen using readelf -d llvm-config) and the binary is located in STAGING_DIR/usr/bin, it tries to link with the libc of the target, resuting in the error displayed above.

It was found that function `llvm_setup_rpath` in AddLLVM.cmake sets this RPATH, but it just returns in case CMAKE_INSTALL_RPATH is defined. So the final solution was passing HOST_LLVM_CONF_OPTS += -DCMAKE_INSTALL_RPATH="$(HOST_DIR)/lib" in llvm.mk, so that LLVM binaries compiled for the host always link with host's libraries.

Commit: http://lists.busybox.net/pipermail/buildroot/2018-April/218938.html

Preparation of the new series

After having fixed the bugs found by the autobuilders, the OpenCL series was retaken, adding the following improvements:

- In order to cross-compile Clang, now llvm-tblgen from the host is used. llvm-tblgen is no longer copied to STAGING_DIR/usr/bin.
- libclc headers are now installed to /usr/local/include by using the –includedir option in libclc.mk. This directory is not removed by Buildroot when generating the target root filesystem.
- Some missing dependencies were propagated.

PATCH v6

The PATCH v6 series[18] sent to the Buildroot mailing list on the 11th April contains the following 4 commits:

- [PATCH v6 1/4] package/clang: new package
- [PATCH v6 2/4] package/libclc: new package
- [PATCH v6 3/4] package/mesa3d: enable OpenCL support
- [PATCH v6 4/4] package/clinfo: new package

[18]http://lists.busybox.net/pipermail/buildroot/2018-April/218849.html

Update - 23 April 2018

OpenCL for Broadcom Videocore IV

The next goal was adding OpenCL support for the Broadcom Videocore IV GPU in Buildroot. This is an interesting feature considering that this GPU is embedded in all Raspberry Pi models.

Videocore IV architecture

The VC4 has multiple instances of a special purpose floating-point shader processor, called a Quad Processor (QPU). The QPU is a 16-way SIMD processor, where each processor has two vector floating point ALUs which carry out multiply and non-multiply operations in parallel with single instruction cycle latency. Internally the QPU is a 4-way SIMD processor multiplexed 4x over four cycles

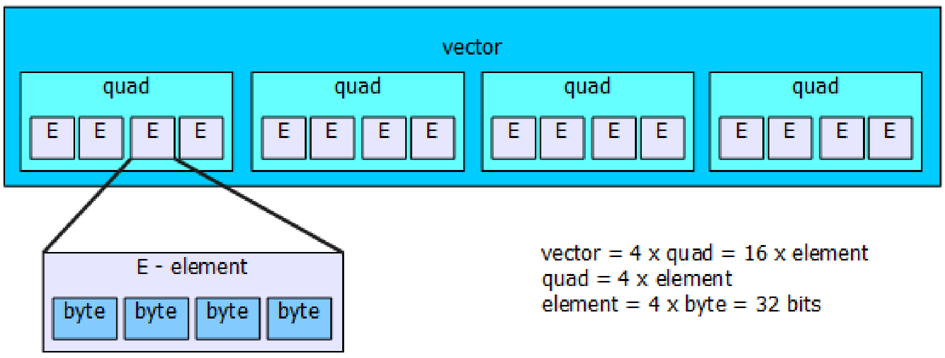

vector
quad
quad
quad
quad
E
E - element
byte
vector = 4 x quad = 16 x element
quad = 4 x element
element = 4 x byte = 32 bits

Figure 10: QPU data model

QPU is SIMD architecture (Single Instruction, Multiple Data), this means that one instruction operates on a vector of elements. When looking from the programmer's point of view, it processes a vector of 16 elements each 32-bits long. If physical structure is taken into account, a QPU processes only a 4-element vector (quad). By repeating the instruction 4 times for consecutive quads in a 16-element vector, it provides a virtual SIMD-16.

QPUs are organized into groups of up to four, termed slices, which share certain common resources: each slice shares an instruction cache, a Special Function Unit (for recip/recip-sqrt/log/exp functions), one or two Texture and Memory lookup units and Interpolation units. As the Videocore IV has 3 slices of 4 QPUs each one, it provides 12 QPUs and 3 SFUs, which makes this GPU an interesting option for solving problems that present data level parallelism. Broadcom claims a computational power of 24 GFLOPs, which comes out from the following equation:

250 MHz (Clock Rate) * 4-Way SIMD * 2 Asymmetric ALUs * 12 QPUs = 24 GFLOPs

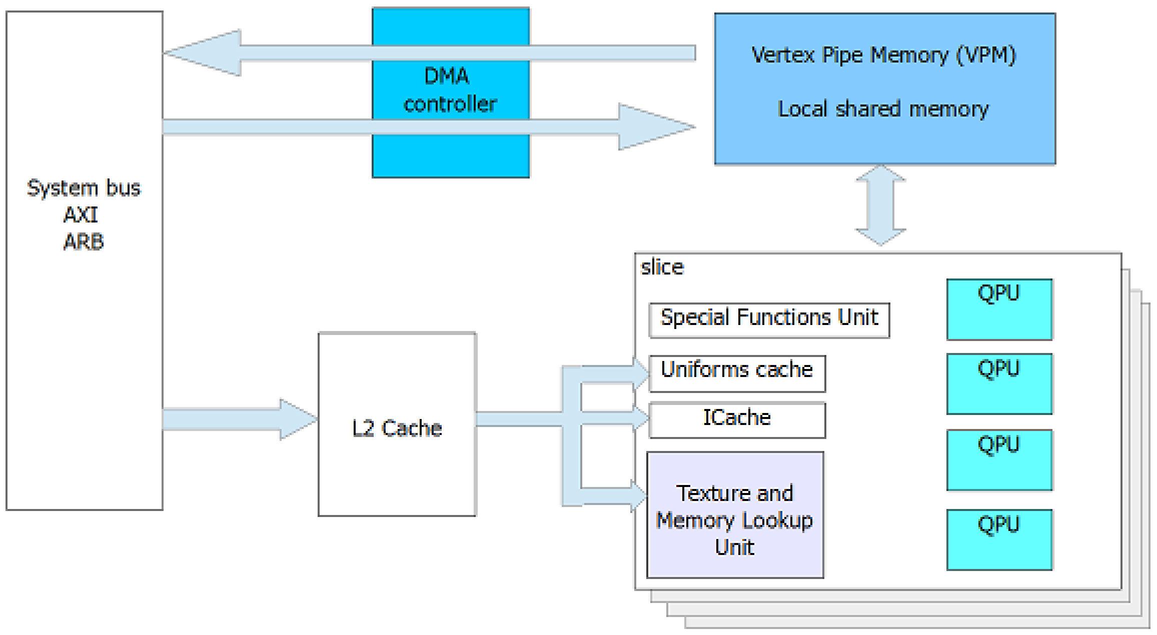


Figure 11: Simplified architecture of the Videocore IV

VC4CL

There is an open source project called VC4CL[19] which provides an implementation that supports the EMBEDDED PROFILE (trimmed version of the default FULL PROFILE) of the OpenCL 1.2 standard for the VideoCore IV GPU. This implementation consists of:

- The VC4CL OpenCL runtime library, running on the CPU to compile, run and interact with OpenCL kernels.
- The VC4C compiler, converting OpenCL kernels into machine code. This compiler also provides an implementation of the OpenCL built-in functions.
- The VC4CLStdLib, the platform-specific implementation of the OpenCL C standard library, it is linked with the kernel via VC4C.

The `cl_khr_icd` extension is supported to allow VC4CL to be found by an installable client driver loader (ICD). As explained before, it allows VC4CL to be used in parallel with other OpenCL implementations.

Not supported features:

- 64-bit data-types (long and double via `cl_khr_fp64`) are unsupported, since the Videocore IV GPU only provides 32-bit instructions.
- The `cl_khr_fp16` half floating-point type is also not supported.
- Images (WIP).
- Any application which requires a work-group with more than 12 work-items.
- Using VC4CL in combination with other applications using the VideoCore IV GPU (e.g. the VC4 Mesa driver) is untested and can cause issues on both sides.

[19]https://github.com/doe300/VC4CL/wiki

VC4CL package for Buildroot

In order to install VC4CL, VC4CLStdLib and VC4C are needed. As these projects are thought to be installed natively on top of a Raspbian distribution, some changes were made to cross-compile them under the Buildroot environment.

When instaling VC4CLStdLib, it was necessary to change the location of its header files to /usr/local/include, as they are required on runtime by the VC4C compiler and, as explained before, Buildroot removes the /usr/include directory from the target filesystem.

One particularity of this implementation is that it calls Clang binaries in order to compile OpenCL kernels on the target instead of linking with libclang, as most OpenCL implementations do. Even if the package was tested and working, this fact prevents this patch series from being sent to the Buildroot mailing list, as the mantainers do not allow a compiler to be installed on the target.

The VC4C package has the same dependencies as Clang, but also needs VC4CLStdLib and Raspberry Pi Userland, the latter providing EGL and KHR headers. Regarding VC4C's Makefile, two hooks were added: one to copy Clang binaries to /usr/bin (Clang's Makefile removes them from the target) and another one to install a precompiled header (VC4CLStdLib.h.pch) that is built during VC4C's compilation to /usr/local/include/vc4cl-stdlib, as it is a runtime dependency of VC4CL.

VC4C gives the user the possibility of choosing among three different frontends: LLVM IR Parser, SPIR-V Reader or LLVM Library:

Table 7: Frontends

Frontend	**LLVM IR Parser**	**SPIR-V Reader**	**LLVM Library**
Input formats	LLVM IR text	SPIR-V text/bin	LLVM IR text/bin
Speed	Slow	Fast	Faster
Supported LLVMs	Standard/SPIRV	SPIRV	Standard/SPIRV
Runtime Deps	Clang	SPIRV-LLVM Clang	Clang, libLLVM
Development Deps	-	SPIRV-Tools	LLVM headers
Configuration	LLVMIR_FRONTEND	SPIRV_FRONTEND	LLVMLIB_FRONTEND
Supports linking	No	Yes	No

The LLVM Library frontend was selected, as its dependencies are already packaged in Buildroot and this is also the suggested option by the creator of the project. It suffices to give the path to llvm-config installed in STAGING_DIR in vc4c.mk so that libLLVM.so can be found.

It is important to remark that CMakeLists.txt of VC4C was patched because it needs to find and execute Clang during configuration (which can only be achieved by passing the path to host's Clang) but needs the path to target's Clang for runtime kernel compilation.

The last step was packaging VC4CL. This package needs ocl-icd as a dependency, so this package was also created and VC4CL was built with ICD support, so that VC4CL.icd containing the path to libVC4CL.so was installed to /etc/OpenCL/vendors.

Testing VC4CL

To verify that the OpenCL environment was correctly set up, clinfo was installed and executed, obtaining the following output:

```
Number of platforms                       1
Platform Name                            OpenCL for the Raspberry Pi VideoCore
                                         IV GPU
Platform Vendor                          doe300
Platform Version                         OpenCL 1.2 VC4CL 0.4
Platform Profile                         EMBEDDED_PROFILE
Platform Extensions                      cl_khr_il_program cl_khr_spir
                                         cl_altera_device_temperature
                                         cl_altera_live_object_tracking
                                         cl_khr_icd
                                         cl_vc4cl_performance_counters
Platform Extensions function suffix      VC4CL
Platform Name                            OpenCL for the Raspberry Pi VideoCore
                                         IV GPU
Number of devices                        1
Device Name                              VideoCore IV GPU
Device Vendor                            Broadcom
Device Vendor ID                         0xa5c
Device Version                           OpenCL 1.2 VC4CL 0.4
Driver Version                           0.4
Device OpenCL C Version                  OpenCL C 1.2
Device Type                              GPU
Device Profile                           EMBEDDED_PROFILE
Device Available                         Yes
Compiler Available                       Yes
Linker Available                         No
Max compute units                        1
Max clock frequency                      250MHz
Device Partition                         (core)
Max number of sub-devices                0
Supported partition types                None
Max work item dimensions                 3
Max work item sizes                      12x12x12
Max work group size                      12
Preferred work group size multiple       1
```

Listing 6: Output of clinfo

Something that called the attention during the execution of clinfo was the fact that it took much more time than expected. This application calls clCreateProgramWithSource() in order to create a program object receiving the source code of an OpenCL kernel as input. This step involves calling Clang, which results in a bottleneck when using a Cortex-A7 processor running at 900Mhz (Raspberry Pi 2), considering that OpenCL programs are made to improve execution time.

```
/usr/bin/clang -cc1 -triple spir-unknown-unknown  -O3 -ffp-contract=off
-cl-std=CL1.2 -cl-kernel-arg-info -cl-single-precision-constant
-Wno-undefined-inline -Wno-unused-parameter -Wno-unused-local-typedef
-Wno-gcc-compat -x cl -S -emit-llvm-bc -o /tmp/vc4c-DRbyiL
-include-pch /usr/local/include/vc4cl-stdlib/VC4CLStdLib.h.pch
```

Listing 7: Clang invocation by clCreateProgramWithSource()

Most OpenCL programs make use of this function, but compiling kernels on the target is definitely not a viable solution. A better alternative is to compile kernels on the host and use clCreateProgramWithBinary(). This function creates a program object for a context, and loads specified binary data into the program object. For this, OpenCL kernels must be compiled to LLVM bitcode by using host-clang:

```
clang -cc1 -emit-llvm-bc -o kernel_pi.bc kernel_pi.cl
```

Listing 8: Compiling OpenCL C to LLVM bitcode

On the other side, the program must read the corresponding bitcode file and then create the program by calling clCreateProgramWithBinary:

```
FILE *fp;
char fileName[] = "./kernel_pi.bc";
size_t binary_size;
char *binary_buf;

/* Load kernel binary */
fp = fopen(fileName, "r");
if (!fp) {
        fprintf(stderr, "Could not read the kernel file: %s\n", fileName);
        exit(1);
}
binary_buf = (char *)malloc(MAX_BINARY_SIZE);
binary_size = fread(binary_buf, 1, MAX_BINARY_SIZE, fp);
fclose(fp);
...
...
clGetPlatformIDs(..
clGetDeviceIDs(...
clCreateContext(...
clCreateCommandQueue(...

/* Create kernel program from the kernel binary */
program = clCreateProgramWithBinary(context, 1, &device_id,
                                    (const size_t *)&binary_size,
                                    (const unsigned char **)&binary_buf,
                                    &binary_status, &error);

error = clBuildProgram(program,...
...
...
```

Listing 9: Example using clCreateProgramWithBinary

This approach drastically improves the performance and would be the suitable solution when running OpenCL programs on embedded platforms. Furthermore, it does not need major changes in the code, but compiled kernels must be shipped together with the executable file.

In order to test some real OpenCL application, the EasyCL project was added to Buildroot, allowing to get an idea about what kind of functionalities work correctly. Many tests failed with the following error:

```
64-bit operations are not supported by the VideoCore IV architecture,
further compilation may fail!
```

The test suite was aborted when running `reduce_multipleworkgroups_ints_noscratch` because of an invalid index:

```
terminate called after throwing an instance of 'std::out_of_range'
```

The final results:

```
[  FAILED  ] testscalars.test1 (75455 ms)
[       OK ] testintarray.main (68581 ms)
[       OK ] testfloatwrapper.main (71079 ms)
[       OK ] testfloatwrapper.singlecopytodevice (1 ms)
[       OK ] testfloatwrapper.doublecopytodevice (1 ms)
[       OK ] testqueues.main (69716 ms)
[       OK ] testqueues.defaultqueue (69969 ms)
[       OK ] testclarray.main (71562 ms)
[       OK ] testfloatwrapperconst.main (70102 ms)
[       OK ] testintwrapper.main (69920 ms)
[       OK ] test_scenario_te42kyfo.main (67494 ms)
[       OK ] testfloatarray.main (67650 ms)
[       OK ] testeasycl.main (68985 ms)
[       OK ] testeasycl.power2helper (0 ms)
[       OK ] testinout.main (67367 ms)
[  FAILED  ] testlocal.uselocal (73568 ms)
[  FAILED  ] testlocal.notUselocal (73460 ms)
[  FAILED  ] testlocal.globalreduce (73644 ms)
[  FAILED  ] testlocal.localreduce (193205 ms)
[  FAILED  ] testlocal.reduceviascratch_multipleworkgroups (192997 ms)
[  FAILED  ] testlocal.reduceviascratch_multipleworkgroups_ints (194560 ms)
```

Listing 10: EasyCL tests on VC4CL

It is interesting to see the results of the same test suite executed on AMD Radeon Dual Graphics GPU (integrated HD6480G + dedicated HD7450M) using Clover platform. First of all, the program finishes correctly, it is not aborted with 'std::out_of_range' exception as with VC4CL and the results show that most of the tests pass:

```
[  FAILED  ] testscalars.test1 (1794 ms)
[       OK ] testintarray.main (1582 ms)
[       OK ] testfloatwrapper.main (1575 ms)
[       OK ] testfloatwrapper.singlecopytodevice (1 ms)
[       OK ] testfloatwrapper.doublecopytodevice (1 ms)
[       OK ] testqueues.main (1561 ms)
[       OK ] testqueues.defaultqueue (1581 ms)
[       OK ] testclarray.main (1564 ms)
[       OK ] testfloatwrapperconst.main (1568 ms)
[       OK ] testintwrapper.main (1563 ms)
[       OK ] test_scenario_te42kyfo.main (1518 ms)
[       OK ] testfloatarray.main (1507 ms)
[       OK ] testeasycl.main (1572 ms)
[       OK ] testeasycl.power2helper (0 ms)
[       OK ] testinout.main (1478 ms)
[       OK ] testlocal.uselocal (1742 ms)
[       OK ] testlocal.notUselocal (1649 ms)
[  FAILED  ] testlocal.globalreduce (3833 ms)
[       OK ] testlocal.localreduce (2210 ms)
[       OK ] testlocal.reduceviascratch_multipleworkgroups (1708 ms)
[       OK ] testlocal.reduceviascratch_multipleworkgroups_ints (1793 ms)
[  FAILED  ] testlocal.reduce_multipleworkgroups_ints_noscratch (1714 ms)
[       OK ] testdefines.simple (1505 ms)
[       OK ] testbuildlog.main (1390 ms)
[       OK ] testnewinstantiations.createForFirstGpu (1612 ms)
[       OK ] testnewinstantiations.createForIndexedGpu (1590 ms)
[       OK ] testnewinstantiations.createForIndexedDevice (3212 ms)
[       OK ] testnewinstantiations.createForPlatformDeviceIndexes (1590 ms)
[       OK ] testnewinstantiations.createForFirstGpuOtherwiseCpu (1601 ms)
[  FAILED  ] testucharwrapper.main (1593 ms)
[       OK ] testkernelstore.main (1530 ms)
[       OK ] testkernelstore.cl_deletes (6115 ms)
[       OK ] testdirtywrapper.main (1610 ms)
[       OK ] testDeviceInfo.basic (0 ms)
[       OK ] testDeviceInfo.gpus (0 ms)
[       OK ] testLuaTemplater.basicsubstitution1 (1 ms)
[       OK ] testLuaTemplater.basicsubstitution1b (0 ms)
[       OK ] testLuaTemplater.basicsubstitution (1 ms)
[       OK ] testLuaTemplater.startofsection (1 ms)
[       OK ] testLuaTemplater.endofsection (0 ms)
[       OK ] testLuaTemplater.loop (1 ms)
[       OK ] testLuaTemplater.nestedloop (1 ms)
[       OK ] testLuaTemplater.foreachloop (1 ms)
[       OK ] testLuaTemplater.codesection (0 ms)
[       OK ] testLuaTemplater.codingerror (1 ms)
[       OK ] testLuaTemplater.include (1 ms)
[       OK ] testTemplatedKernel.basic (3055 ms)
[       OK ] testTemplatedKernel.withbuilderror (1585 ms)
[       OK ] testTemplatedKernel.withtemplateerror (2 ms)
[       OK ] testTemplatedKernel.withbuilderrorintargs (1321 ms)
[       OK ] testTemplatedKernel.withargserror (1544 ms)
[       OK ] testTemplatedKernel.basic2 (15166 ms)
[       OK ] testTemplatedKernel.foreach (1513 ms)
[       OK ] testTemplatedKernel.forrange (1531 ms)
[       OK ] testTemplatedKernel.forrange2 (1514 ms)
[       OK ] testStructs.main (1692 ms)
[       OK ] testprofiling.basic (6476 ms)
[       OK ] testprofiling.noprofiling (1503 ms)
[       OK ] testcopybuffer.main (1 ms)
[       OK ] testcopybuffer.withoffset (2 ms)
[       OK ] testcopybuffer.throwsifnotondevice (1 ms)
[       OK ] teststatefultimer.basic (1942 ms)
[       OK ] teststatefultimer.notiming (1872 ms)
```

Listing 11: EasyCL tests on Clover (AMD SUMO + AMD CAICOS)

Update - 14 May 2018

Achievements

Clang package[20] was commited to Buildroot's master branch on the 28th April. After that date, two patches were sent in order to have a cleaner version of the package. As the objective is installing only libclang.so, the first patch removes unnecessary files from the target, more specifically the following ones:

- Binaries in:
  - /usr/bin
  - /usr/libexec
- Directories:
  - /usr/lib/clang
  - /usr/share/clang
  - /usr/share/opt-viewer
  - /usr/share/scan-build
  - /usr/share/scan-view
- Manual
  - /usr/share/man/man1/scan-build.1

The second patch serves to link libclang.so dynamically against libLLVM.so, because at the start libclang was linking against LLVM static libraries (libLLVMOption.a, libLLVMMCParser.a, libLLVMProfileData.a, etc.), producing duplicated code. As Clang is an LLVM tool, it was necessary to set LLVM_LINK_LLVM_DYLIB to ON in Clang's Makefile.

Improved OpenCL series

With respect to the v6 series, the following changes were introduced:

- libclc headers are now installed to /usr/share instead of /usr/local/include. Given that clc headers are being installed to a non-standard location, it was necessary to specify this path in Mesa's configure.ac. Otherwise, pkg-config outputs the absolute path to these headers located in STAGING_DIR, which causes a runtime error when calling clBuildProgram.
- libclc dependencies on target llvm were removed, as host-clang is the only build dependency.
- OpenCL support for RadeonSI was added.

[20]http://lists.busybox.net/pipermail/buildroot/2018-April/219824.html

PATCH v7

The PATCH v7 series[21] sent to the Buildroot mailing list on the 4^{th} May contains the following 3 commits:

- [PATCH v7 1/3] package/libclc: new package
- [PATCH v7 2/3] package/mesa3d: enable OpenCL support
- [PATCH v7 3/3] package/clinfo: new package

[21]http://lists.busybox.net/pipermail/buildroot/2018-May/220772.html

Update - 15 June 2018

LLVM/Clang version 5.0.2

LLVM and Clang version 5.0.2 were released on the 16th May. Both releases are API and ABI compatible with 5.0.0 and 5.0.1 and include mitigations for CVE-2017-5715[22] (Spectre Variant 2) for X86 and MIPS. Due to the importance of these bufgixes, both patches were applied to Buildroot's master branch:

LLVM bumped to version 5.0.2:

http://lists.busybox.net/pipermail/buildroot/2018-May/221643.html

Clang bumped to version 5.0.2:

http://lists.busybox.net/pipermail/buildroot/2018-May/221642.html

Bug fixing

Fix for host-llvm when built with GCC 8

Autobuild:

http://autobuild.buildroot.net/results/824c70e982d8ec7e518cf4db058767df42db6b04

Extract from log:

```
output/build/host-llvm-5.0.1/include/llvm/ExecutionEngine/
Orc/OrcRemoteTargetClient.h:722:26: error: could not convert
'((llvm::orc::remote::OrcRemoteTargetClient<ChannelT>*)this)->
callB<llvm::orc::remote::OrcRemoteTargetRPCAPI::ReadMem>(Src, Size)'
from 'Expected<vector<unsigned char,allocator<unsigned char>>>'
to 'Expected<vector<char,allocator<char>>>'
return callB<ReadMem>(Src, Size);
```

Fix:

GCC 8.0.1 detects the type mismatch between char and unsigned char and causes the compilation to fail. Clang and earlier versions of GCC don't detect the issue. This bug was already known [23] and has been fixed upstream in LLVM 6

Commit: http://lists.busybox.net/pipermail/buildroot/2018-May/221648.html

Fix host-clang binaries

Autobuild:

This error was detected locally when trying to build libclc

Extract from log:

[22] https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)
[23] https://bugzilla.redhat.com/show_bug.cgi?id=1540620

```
CommandLine Error: Option 'x86-use-base-pointer' registered more than once!
LLVM ERROR: inconsistency in registered CommandLine options
```

Fix:

Clang binaries are tools, and given that DLLVM_LINK_LLVM_DYLIB is set, they are linked against libLLVM.so. The problem is that binaries are also linking against some LLVM static libraries, which results in the error shown above. However, it is not the same case for libclang, which is also a tool but links only against libLLVM.so. To fix this problem, LLVM_DYLIB_COMPONENTS=all must be added in Clang's Makefile so that binaries only link against libLLVM.so and the double symbol definition is eliminated.

Commit: http://lists.busybox.net/pipermail/buildroot/2018-June/222682.html

Buildroot 2018.05

Buildroot's stable version 2018.05 was released on the 1st June, containing both LLVM and Clang packages: http://lists.busybox.net/pipermail/buildroot/2018-June/222697.html

7 Conclusions and future work

This document describes the whole process of integrating a new package to Buildroot, detailing every necessary step to meet the requirements which allow the package to be merged into the project. This involves finding all the correct dependencies (toolchain properties or packages) and writing a Makefile adjusting the corresponding build options. Thanks to the tests and reviews done by the community, it was possible to advance rapidly and send improved patch series as soon as possible.

Currently, LLVM 5.0.2, Clang 5.0.2 and LLVM support for Mesa 3D are available in Buildroot 2018.05. The update of these packages to version 6.0.0 has been done by another contributor and will be available in the next stable release. The activity on the mailing list shows an interest of Buildroot users and contributors in LLVM. Such is the case of a contributor who is creating a package Chromium browser, which relies on Clang to be built. This contribution also adds `lld`, the system linker from the LLVM project that provides an alternative to GNU `ld` and claims to be much faster than the latter one.

Regarding future work, the most immediate goal is to get OpenCL support for AMD GPUs merged into Buildroot. The next step will be to add more packages that rely on LLVM/Clang and OpenCL. On the other hand, the fact of creating a toolchain based on LLVM/Clang is still being discussed on the mailing and is a topic that requires an agreement from the core developers of the project.