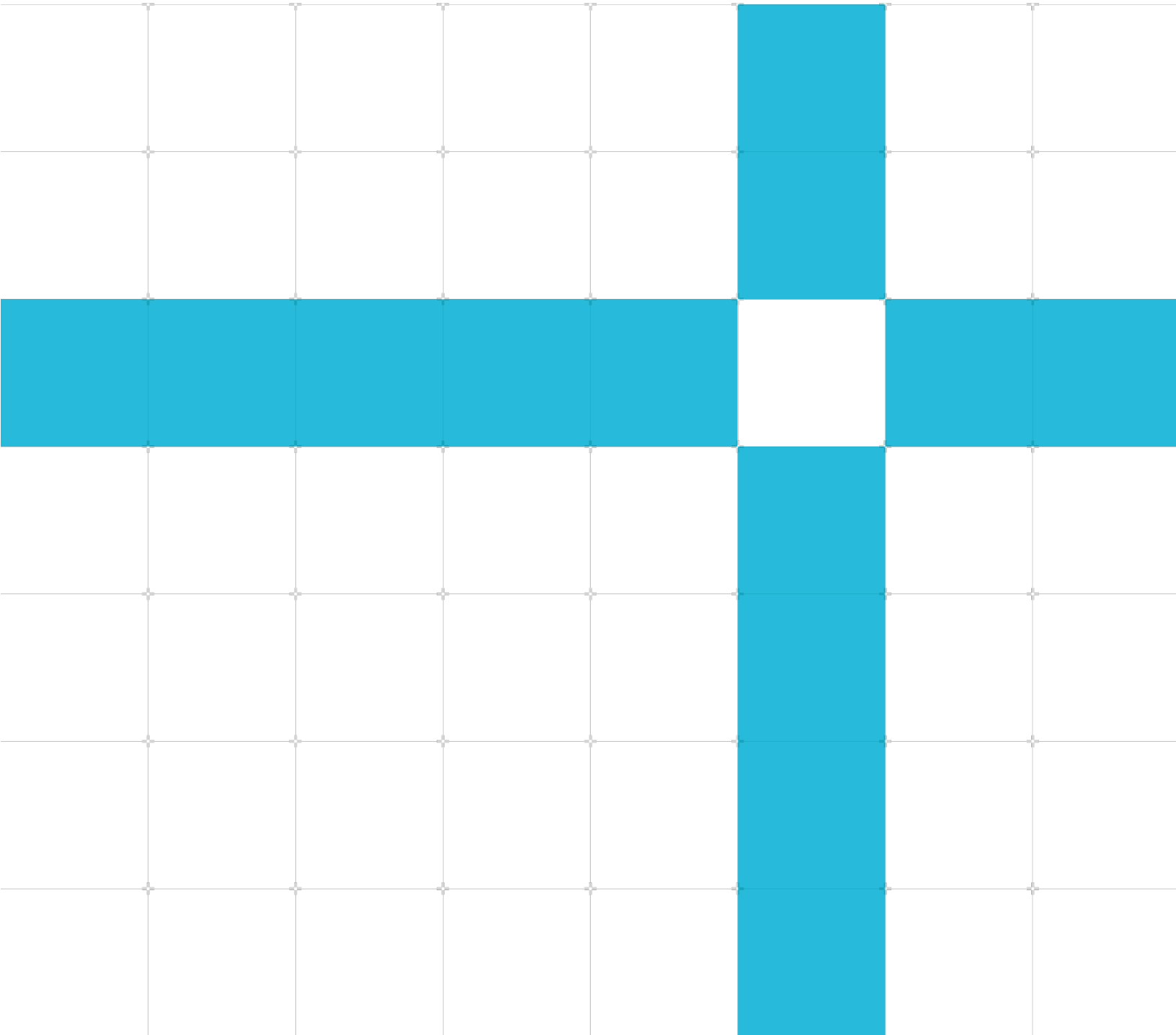
arm

Compiling with Clang for Windows on Arm

Non-Confidential

Issue 1.0
102563



Compiling with Clang for Windows on Arm

Release information

Document history

Issue	Date	Confidentiality	Change
1.0	6th August 2021	Non-Confidential	First release

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

developer.arm.com

Progressive terminology commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please email **terms@arm.com**.

Contents

1 Overview

This guide introduces the features in LLVM 12 and the associated Clang release that help developers for Arm-based devices. In particular, this guide examines how to use the native toolchain to compile for Windows on Arm (WoA). The guide uses the example of compiling the popular open-source PuTTY application for Windows on Arm.

The new LLVM toolchain variant for Windows on Arm means that developers can now develop and compile a C/C++ application on a Windows on Arm laptop with a native AArch64 LLVM toolchain. This native toolchain means that you can develop software for an Arm-based device on that device itself, rather than cross-compiling on another host or using emulation to run an x86 build of Clang.

For Windows on Arm devices, using the native toolchain is much faster than running an x86 build of Clang under emulation. Running under emulation restricts the use of modern compiler technologies like link-time optimization. This is because the 32-bit toolchain supported under emulation can only use 4GB of memory. Native binaries are AArch64 binaries and do not require emulation, speeding up the entire process for developers.

1.1 Before you begin

To follow this tutorial, you need the following hardware and software:

- A Windows on Arm device
- LLVM 12.0.0 or higher, which is available from the LLVM download page.
  The pre-built binary for Windows on Arm is LLVM-12.0.0-woa64.exe.
- Visual Studio, including the Arm build tools and the Desktop development with C++ workload. Follow the Visual Studio installation instructions in Building libraries for Windows on Arm: Install Visual Studio.
- The Microsoft C Runtime Library, version 14.00.24234.1 or later, called `vcruntime140.dll`. Microsoft distributes this runtime library as part of Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017 and 2019, available as vc_redist.arm64.exe. For more information, see this Microsoft material: The latest supported Visual C++ downloads.
- Perl, for example Strawberry Perl or similar
- A file archive utility, for example 7-Zip or similar
- A `make` utility, for example GnuWin32 or similar

2 LLVM support for Arm-based devices

LLVM 12 provides improved support for Arm-based devices over previous versions.

This improved support includes the following features:

- A native Windows on Arm toolchain
- Support for new processors
- Performance improvements to a key CPU benchmark

Let's look at each of these features in more detail.

2.1 Native Windows on Arm 64-bit toolchain

The LLVM toolchain variant for Windows on Arm means that you can now develop and compile a C/C++ application on a Windows on Arm laptop with a native AArch64 LLVM toolchain.

Compiling on Windows on Arm devices using native Clang is faster than running an x86 build of Clang through emulation. Arm and Linaro have independently confirmed that a typical compile time is twice as fast using the native toolchain.

2.2 Clang-cl support

As part of the LLVM 12 release, LLVM supports clang-cl, a compatibility layer for Microsoft Visual C++ (MSVC). This means that most developers can use clang-cl to compile their C/C++ applications on Visual Studio/MSBuild on the Windows on Arm device, without needing to change the command line.

You can use `clang-cl.exe` as a direct replacement for `cl.exe`, the MSVC compiler executable. This allows you to easily modify projects that already use MSVC to use native compilation.

2.3 Arm processor support

LLVM 12 adds support for the following processors:

- Arm Neoverse V1
- Arm Neoverse N2
- Arm Cortex-A78C
- Arm Cortex-R82
- Fujitsu A64FX

Use the Clang `-mcpu` command-line option to compile for a specific processor. When you specify the `-mcpu` option, the compiler automatically uses the appropriate architectural features for the specified processor, optimizing code performance accordingly.

The following table specifies the command-line option for each of the new target processors:

Table 1 Clang command-line options for targeting new processors

Processor	Clang command-line option
Arm Neoverse V1	`-mcpu=neoverse-v1`
Arm Neoverse N2	`-mcpu=neoverse-n2`
Arm Cortex-A78C	`-mcpu=cortex-a78c`
Arm Cortex-R82	`-mcpu=cortex-r82`
Fujitsu A64FX	`-mcpu=a64fx`

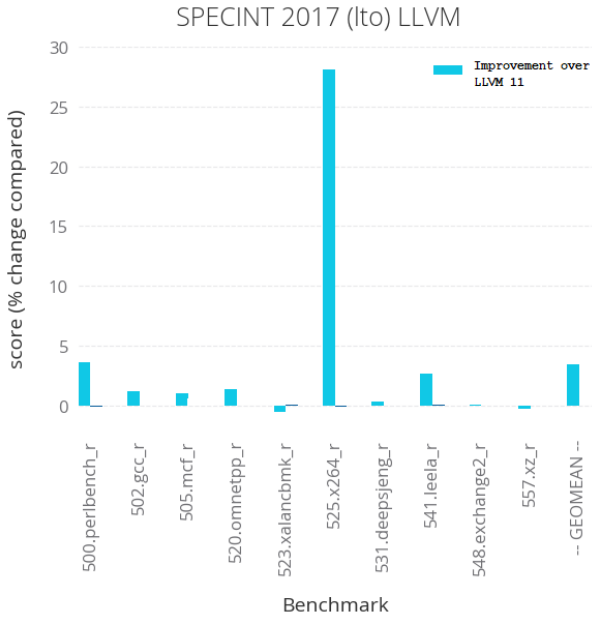
2.4 SPEC CPU 2017 CPU benchmark improvements

LLVM 12 adds new generic vectorization optimizations that improve performance on the SPEC CPU 2017 Integer 525.x264_r benchmark. On Arm Neoverse N1 hardware, there is a 25% improvement for this individual benchmark, and an overall 2% improvement in the SPEC CPU 2017 INT score.

Other optimizations in LLVM 12 that contribute to improved benchmark scores include the following:

- LLVM identifies SAD pattern and combines UDADDV instructions to generate vector addition operations.
- LLVM now supports epilogue vectorization.

The following chart shows the performance improvement of LLVM 12 compared to LLVM 11 over several different benchmarks:



2.5 Out-of-line atomics for LSE deployment

Armv8.1-A introduced AArch64 Large System Extensions (LSE), which provide more efficient atomic instructions for large multiprocessor systems.

LLVM 12 adds the new Clang option `-moutline-atomics`, which detects at runtime whether the processor supports LSE. If LSE is supported, the compiler uses the new atomic instructions if possible. For processors without LSE support, the compiler uses Armv8.0-A LL/SC loops. This behavior mirrors similar support that is available within the GNU family of projects.

2.6 Improved support for SVE and SVE2 intrinsics

LLVM 11 was the first LLVM release to add vector-length agnostic Scalable Vector Extensions (SVE) intrinsics support. LLVM 12 adds vector-length specific ACLE support and improves vector-length-agnostic support.

2.7 SVE code-generation infrastructure

LLVM 12 introduces the ability to vectorize certain loops using width-agnostic SVE auto-vectorization.

For example, consider the following loop, which is adapted from the set of loops in the updated Test Suite for Vectorising Compilers, TSVC_2:

```
void s000(double * __restrict a, double * __restrict b) {
  unsigned LEN_1D = 1024;
#pragma clang loop vectorize_width(2, scalable) interleave(disable) unroll(disable)
  for (int i = 0; i < LEN_1D; i++) {
    a[i] = b[i] + 1;
  }
}
```

The `#pragma clang loop vectorize_width(2, scalable)` code enables SVE auto-vectorization. This code tells LLVM to attempt to vectorize the loop using a scalable vectorization width of two lanes.

You can learn more about width-agnostic SVE auto-vectorization in Arm's recent Linaro Connect presentation: SVE and SVE2 in LLVM.

2.8 MVE optimizations

M-Profile Vector Extension (MVE) is an extension of the Armv8.1-M architecture. MVE is designed to give a significant performance improvement for machine learning and digital signal processing workloads on processors for embedded devices, such as Arm Cortex-M55.

LLVM 12 includes optimizations to vectorization and code quality for MVE, leading to significant improvements to both performance and code size for a range of workloads. In particular, LLVM can now fully utilize the capabilities of MVE to allow tail-predicated vectorization of reduction loops.

2.9 Debug support for SVE and SVE2

LLDB now has full support for debugging SVE and SVE2 applications, including dynamic size update of SVE registers.

3 Compiling PuTTY natively on WoA with Clang

This section of the guide describes how to use Clang to compile and build an application for Windows on Arm. The example application used in this tutorial is PuTTY, an open-source SSH and telnet client. We use PuTTY as an example because it is well known, widely used, and freely available.

LLVM 12 includes support for native compilation on Windows for Arm (WoA). This support means we can compile the application natively on the WoA device itself, rather than cross-compiling on a different machine.

To compile PuTTY on a Windows on Arm device:

1. Start a Windows command prompt with **Start** > **cmd.**
2. Create a folder to use for the build, for example `C:\putty`, and move into that folder:
   ```
   mkdir C:\putty
   cd C:\putty
   ```
3. Download the Putty source archive and save it in your build folder.

   The Putty source archive is the file `putty-src.zip`, labeled as **Windows source archive** in the **Source code** section of the downloads page.

The downloads page also provides pre-compiled Windows on Arm binaries. Because the aim of this tutorial is to demonstrate native compilation, we want to download the Windows source files rather than a pre-compiled binary.

4. Extract the Putty source archive `putty-src.zip` into the build folder, using 7-Zip or a similar file archive utility.

   To use 7-Zip, right-click the zip file in the build folder, and click **7-Zip** > **Extract Here**.
5. Run the Perl script `mkfiles.pl` to automatically generate the makefiles and folders used by the build process:
   ```
   perl mkfiles.pl
   ```
   The script generates makefiles for several different compilers. The makefile that we will use is `Makefile.clangcl`, which uses the Clang compiler.
6. Run the Visual Studio batch file to configure Command Prompt for Developers, to automatically configure your environment to compile for Arm-based targets:
   ```
   cmd /k ""C:\Program Files (x86)\Microsoft Visual
   Studio\2019\Community\VC\Auxiliary\Build\vcvarsall.bat"" x64_arm64
   ```
7. Move to the `windows` subdirectory in the build folder:
   ```
   cd windows
   ```
8. Use the GnuWin32 `make` utility, or similar, to build the PuTTY application, as shown by the following command:
   ```
   C:\GnuWin32\bin\make.exe Platform=arm64  -f  Makefile.clangcl all
   ```

The following options control the build process:

- `Platform=arm64` specifies that the build target is a 64-bit Arm-based system.
- `-f Makefile.clangcl` specifies that the build process uses the Clang compiler.
- `all` directs the build process to compile all components of the PuTTY application.

When the build process completes, there will be a new executable `putty.exe` in the `windows` subdirectory in the build folder. This is the natively compiled PuTTY application. Double-click it to run and check that it works on your Windows on Arm device.

4 Related information

Here are some resources related to material in this guide:

- Clang documentation
- Compiling on Arm for Arm: New native LLVM toolchain on Windows on Arm
- GnuWin32
- The LLVM Compiler Infrastructure
- LLVM download page
- Microsoft resources:
  - Install Visual Studio
  - Microsoft Visual Studio documentation
  - The latest supported Visual C++ downloads
  - Visual Studio on Arm-powered devices
  - visualstudio.microsoft.com
  - Windows 10 on Arm documentation
- PuTTY: a free SSH and Telnet client
- Strawberry Perl
- What is new in LLVM12 for Arm?
- Windows on Arm portal on developer.arm.com

5 Next steps

This guide introduced the new Clang features which help to support Arm devices, and showed you how to use Clang to natively compile the PuTTY application for a Windows on Arm device.

As we have seen, compiling applications natively for Windows on Arm with Clang is not difficult. If you are compiling an existing application, most of the work is likely to be investigating how the current build process works. When you understand the build process, adapting it for Windows on Arm is straightforward.

You can apply the same process and techniques to build your own application for Windows on Arm.

As a next step, you could try building a different application for Windows on Arm. You could try another open-source application, or create a library of your own.

You can also learn more about Windows on Arm by visiting our Windows on Arm portal .