NVIDIA DOCA DPACC Compiler
# Table of contents

## Introduction
- Glossary
- Offloading Work on DPA
- DPACC Predefined Macros
- Writing DPA Applications
- Language Support
- Restrictions on DPA Code
- DPA RPC Functions
- DPA Global Functions
- Characteristics of Annotated Functions
- Handling User-defined Data Types
- Characteristics of Annotated Types
- DPA Intrinsics

## Prerequisites
- Supported Versions

## Description
- DPACC Inputs and Outputs
- DPA Program
- DPA Object
- DPA Library
- DPACC Trajectory
- Modes of Operation
Compile-and-link Mode
Compile-only Mode
Library Generation Mode

Execution

Mandatory Arguments
Commonly Used Arguments
DPA Hardware Architectures
Architecture Macros
LTO Usage Guidelines
Restrictions
Compatibility
Deprecated Features
Examples
Building Libraries
Linking with DPA Device Library
Enabling Link-time Optimizations
Including Headers
Generating Output as Source Code
DPA Compiler Usage
Compiler Driver Command-line Options
Linker Command Line Options
dpacc-extract Command Line Options
Objdump Command Line Options
This document describes DOCA DPACC compiler and instructions about DPA toolchain setup and usage.

**Introduction**

DPACC is a high-level compiler for the DPA processor which compiles code targeted for the data-path accelerator (DPA) processor into a device executable and generates a DPA program.

The DPA program is a host library with interfaces encapsulating the device executable. This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through FlexIO runtime API.

DPACC uses DPA compiler (dpa-clang) to compile code targeted for DPA. dpa-clang is part of the DPA toolchain package which is an LLVM-based cross-compiling bare-metal toolchain. It provides Clang compiler, LLD linker targeting DPA architecture, and other utilities.

**Glossary**

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>DPA as present on the BlueField DPU</td>
</tr>
<tr>
<td>Host</td>
<td>CPU that launches the device code to run on the DPA</td>
</tr>
<tr>
<td>Device function</td>
<td>Any C function that runs on the DPA device</td>
</tr>
<tr>
<td>DPA global function</td>
<td>Device function that is the point of entry when offloading any work on DPA</td>
</tr>
<tr>
<td>Host compiler</td>
<td>Compiler used to compile the code targeting the host CPU</td>
</tr>
<tr>
<td>Device compiler</td>
<td>Compiler used to compile code targeting the DPA</td>
</tr>
<tr>
<td>DPA program</td>
<td>Host library that encapsulates the DPA device executable (.elf) and host stubs which are used to access the device executable</td>
</tr>
</tbody>
</table>
Offloading Work on DPA

To invoke a DPA function from host, the following things are required:

- **DPA device code** – C programs, targeted to run on the DPA. DPA device code may contain one or more entry functions.

- **Host application code** – the corresponding host application. For more information, refer to [DPA Subsystem](#) documentation.

- **Runtime** – FlexIO or DOCA DPA library provides the runtime

The generated DPA program, when linked with a host application results in a host executable which also contains the device executable. The host application oversees loading the device executable on the device.

**DPACC Predefined Macros**

DPACC predefines the following macros:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DPA</strong></td>
<td>Defined when compiling device code file</td>
</tr>
<tr>
<td><strong>NV_DPA</strong></td>
<td>Defined to the target DPA hardware identifier macros. See <a href="#">Architecture Macros</a> for more details.</td>
</tr>
</tbody>
</table>
Writing DPA Applications

DPA device code is a C code with some restrictions and special definitions.

FlexIO or DOCA-DPA APIs provide interfaces to DPA.

Language Support

The DPA is programmed using a subset of the C11 language standard. The compiler documents any constructs that are not available. Language constructs, where available, retain their standard definitions.

Restrictions on DPA Code

- Use of C thread local storage is not allowed for any variables
- Identifiers with _dpacc prefix are reserved by the compiler. Use of such identifiers may result in an error or undefined behavior
- DPA processor does not have native floating-point support; use of floating point operations is disabled

DPA RPC Functions

A remote procedure call function is a synchronous call that triggers work in DPA and waits for its completion. These functions return a type \texttt{uint64_t} value. They are annotated with a \_dpa\_rpc\_ attribute.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DPA_MAJORM</strong></td>
<td>Defined to the major version number of DPACC</td>
</tr>
<tr>
<td><strong>DPA_MINORM</strong></td>
<td>Defined to the minor version number of DPACC</td>
</tr>
<tr>
<td><strong>DPA_PATCM</strong></td>
<td>Defined to the patch version number of DPACC</td>
</tr>
</tbody>
</table>
DPA Global Functions

A DPA global function is an event handler device function referenced from the host code. These functions do not return anything. They are annotated with a __dpa_global__ attribute.

For more information, refer to DPA Subsystem documentation.

Characteristics of Annotated Functions

- Global functions must have void return type and RPC functions must have uint64_t return type
- Annotated functions cannot accept C pointers and arrays as arguments (e.g., void my_global(int *ptr, int arr[]))
- Annotated functions cannot accept a variable number of arguments
- Inline specifier is not allowed on annotated functions

Handling User-defined Data Types

User-defined data types, when used as global function arguments, require special handling. They must be annotated with a __dpa_global__ attribute.

If the user-defined data type is typedef’d, the typedef statement must be annotated with a __dpa_global__ attribute along the data type itself.

Characteristics of Annotated Types

- They must have a copy of the definition in all translation units where they are used as global function arguments
- They cannot have pointers, variable length arrays, and flexible arrays as members
- Fixed-size arrays as C structure members are supported

- These characteristics apply recursively to any user-defined/typedef’d types that are members of an annotated type

DPACC processes all annotated functions along with annotated types and generates host and device interfaces to facilitate the function launch.

**DPA Intrinsics**

DPA features such as fences and processor-specific instructions are exposed via intrinsics by the DPA compiler. All intrinsics defined in the header file `dpaintrin.h` are guarded by the `DPA_INTRIN_VERSION_USED` macro. The current `DPA_INTRIN_VERSION` is 1.3.

Example:

```c
#define DPA_INTRIN_VERSION_USED (DPA_INTRIN_VERSION(1, 3))
#include <dpaintrin.h>
...
__dpa_thread_writeback_window();  // Fence for write barrier
```

For more information, refer to [DPA Subsystem](#) documentation.

**Prerequisites**
### Package | Instructions
--- | ---
Host compiler | Compiler specified through `hostcc` option. Both `gcc` and `clang` are supported.  

> **Note**  
> Minimum supported version for `clang` as `hostcc` is `clang 3.8.0`.

Device compiler | The default device compiler is the "DPA compiler". Installing the DPACC package also installs the DPA compiler binaries `dpa-clang`, `dpa-ar`, `dpa-nm` and `dpa-objdump`.  

> **Note**  
> `dpa-clang` is the only supported device compiler.

FlexIO SDK and C library | Available as part of the DOCA software package. DPA toolchain does not provide C library and corresponding headers. Users are expected to use the C library for DPA from the FlexIO SDK.

### Supported Versions
- DPACC version 1.8.0
- Refer to DPA Subsystem documentation for other component versions

### Description

**DPACC Inputs and Outputs**

DPACC can produce DPA programs in a single command by accepting all source files as input. DPACC also offers the flexibility of producing DPA object files or libraries from
input files.

DPA object files contain both host stub objects (DPACC-generated interfaces) and device objects. These DPA object files can later be given to DPACC as input to produce the DPA library.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Option Name</th>
<th>Default Output File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile input device code files to DPA object files</td>
<td>--compile or -c</td>
<td>.dpa.o appended to the name of each input source file</td>
</tr>
<tr>
<td>Compile and link the input device code files/DPA object files, and produce a DPA program</td>
<td>No specific option</td>
<td>No default name, output file name must be specified</td>
</tr>
<tr>
<td>Compile and build DPA library from input device code files/DPA object files</td>
<td>--gen-libs or -gen-libs</td>
<td>No default name, output library name must be specified</td>
</tr>
</tbody>
</table>

DPACC can accept the following file types as input:

<table>
<thead>
<tr>
<th>Input File Extension</th>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c</td>
<td>C source file</td>
<td>DPA device code</td>
</tr>
<tr>
<td>.dpa.o</td>
<td>DPA object file</td>
<td>Object file generated by DPACC, containing both host and device objects</td>
</tr>
<tr>
<td>.a</td>
<td>DPA object archive</td>
<td>An archive of DPA object files. User can generate this archive from DPACC-generated DPA objects.</td>
</tr>
</tbody>
</table>

Based on the mode of operations, DPACC can generate the following output files:

<table>
<thead>
<tr>
<th>Output File Type</th>
<th>Input Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPA object file</td>
<td>C source files</td>
</tr>
<tr>
<td>DPA program</td>
<td>C source files, DPA object files, and/or DPA object archives</td>
</tr>
</tbody>
</table>
The following provides the commands to generate different kinds of supported output file types for each input file type:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>DPACC Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>C source file</td>
<td>DPA program</td>
<td>dpacc -hostcc=gcc in.c -o libprog.a</td>
</tr>
<tr>
<td></td>
<td>DPA object</td>
<td>dpacc -hostcc=gcc in.c -c</td>
</tr>
<tr>
<td></td>
<td>DPA library</td>
<td>dpacc -hostcc=gcc in.c -o lib&lt;name&gt; -gen-libs</td>
</tr>
<tr>
<td>DPA object</td>
<td>DPA program</td>
<td>dpacc -hostcc=gcc in.dpa.o -o libprog.a</td>
</tr>
<tr>
<td></td>
<td>DPA library</td>
<td>dpacc -hostcc=gcc in.dpa.o -o lib&lt;name&gt; -gen-libs</td>
</tr>
<tr>
<td>DPA object archive</td>
<td>DPA program</td>
<td>dpacc -hostcc=gcc in.a -o libprog.a</td>
</tr>
<tr>
<td></td>
<td>DPA library</td>
<td>dpacc -hostcc=gcc in.a -o lib&lt;name&gt; -gen-libs</td>
</tr>
</tbody>
</table>

**DPA Program**

DPACC produces a DPA program in compile-and-link mode. A DPA program is a host library which contains:

- DPACC-generated host stubs which facilitate invoking a DPA global function from the host application
- Device executable, generated by DPACC by compiling input DPA device code

DPA program library must be linked with the host application that contains appropriate runtime APIs to load the device executable onto DPA memory.

**DPA Object**

DPACC produces DPA object files in compile-only mode. A DPA object is an object file for the host machine. In a DPA object, the device object generated by compiling the input
device code file is placed inside a specific section of the generated host stubs object. This process is repeated for each input file.

**DPA Library**

A DPA library is a collection of two individual libraries:

- DPA device library – contains device objects generated from input files
- DPA host library – contains host interface objects corresponding to the device objects in DPA device library

The DPA device library is consumed by DPACC during DPA-program generation and the DPA host library can optionally be linked with other host code and be distributed as the host library. Both libraries are generated as static archives.
DPACC Trajectory

The following diagram illustrates DPACC compile-and-link mode trajectory.
Modes of Operation

Compile-and-link Mode

This is a one-step mode that accepts C source files or DPA object files and produces the DPA program. Specifying the output library name is mandatory in this mode.

Example commands:

```
$ dpacc in1.c in2.c -o myLib1.a -hostcc=gcc  # Takes C sources to produce myLib1.a library
```
Compile-only Mode

This mode accepts C source code and produces .dpa.o object files. These files can be given to DPACC to produce the DPA program. The mode is invoked by the --compile or -c option.

The user can explicitly provide the output object file name using the --output-file or -o option.

Example commands:

```
$ dpacc in3.dpa.o in4.dpa.o -o myLib2.a -hostcc=gcc  # Takes DPA object files to produce myLib2.a library
$ dpacc in1.c in3.dpa.o -o myLib3.a -hostcc=gcc     # Takes C source and DPA object to produce myLib3.a library
```

Library Generation Mode

This mode accepts C source files or DPA object files and produces the DPA program. Specifying the output DPA library name is mandatory in this mode.

Example commands:

```
$ dpacc -c input1.c -hostcc=gcc               # Produces input1.dpa.o
$ dpacc -c input3.c input4.c -hostcc=gcc    # Produces input3.dpa.o and input4.dpa.o
$ dpacc -c input2.c -o myObj.dpa.o -hostcc=gcc # Produces myObj.dpa.o

$ dpacc in1.c in2.c -o libdummy1 -hostcc=gcc -gen-libs  # Takes C sources to produce libdummy1_host.a and libdummy_device.a archives
$ dpacc in3.dpa.o in4.dpa.o -o libdummy2 -hostcc=gcc -gen-libs # Takes DPA object files to produce libdummy2_host.a and libdummy2_device.a archives
$ dpacc in1.c in3.dpa.o -o outdir/libdummy3 -hostcc=gcc -gen-libs # Takes C source and DPA object to produce outdir/libdummy3_host.a and outdir/libdummy3_device.a archives
```
Execution

To execute DOCA DPACC compiler:

Usage: dpacc <list-of-input-files> -hostcc=<path> [other options]

Helper Flags:
- -h, --help        Print help information about DPACC
- -V, --version     Print DPACC version information
- -v, --verbose     List the compilation commands generated by this invocation while also
executing every command in verbose mode
- -dryrun, --dryrun  Only list the compilation commands generated by DPACC, without
executing them
- -keep, --keep      Keep all intermediate files that are generated during internal compilation
steps in the current directory
- -keep-dir, --keep-dir  Keep all intermediate files that are generated during internal compilation
steps in the given directory
- -optf, --options-file <file>,...  Include command line options from the specified file

Mandatory Arguments

<table>
<thead>
<tr>
<th>Flag</th>
<th>DPACC Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of one or more</td>
<td>All</td>
<td>List of C source files or DPA object file names. Specifying at least one input file is mandatory. A file with an unknown extension is treated as a DPA object file.</td>
</tr>
<tr>
<td>input files</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-hostcc, --hostcc</td>
<td>All</td>
<td>Specify the host compiler. This is typically the native compiler present on the host system.</td>
</tr>
<tr>
<td>&lt;path&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note

The host compiler used to link the host application with the DPA program must be
<table>
<thead>
<tr>
<th>Flag</th>
<th>DPACC Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o, --output-file &lt;file&gt;</td>
<td>Compile-and-link/library generation</td>
<td>Specify name and location of the output file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>link-compatible with the hostcc compiler provided here.</td>
</tr>
</tbody>
</table>

### Commonly Used Arguments

**Tip**

Use `--help` option for a list of all supported options.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-app-name, --app-name &lt;name&gt;</td>
<td>Specify DPA application name for the DPA program. This option is required if multiple DPA programs are part of a host application because each DPA application must have a unique name. Default name is __dpa_a_out.</td>
</tr>
<tr>
<td>-mcpu=&lt;target_cpu&gt;</td>
<td>Specify the target DPA hardware for code generation. See DPA Hardware Architectures for more details. Supported values: <code>nv-dpa-bf3</code>, <code>nv-dpa-cx7</code></td>
</tr>
<tr>
<td>-flto</td>
<td>Enable link-time optimization (LTO) for device code. Specify this option during compilation along with an optimization level in devicecc-options.</td>
</tr>
<tr>
<td>-devicecc-options, --devicecc-options &lt;options&gt;,...</td>
<td>Specify the list of options to pass to the device compiler.</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>-devicelink-options, --devicelink-options &lt;options&gt;,...</td>
<td>Specify the list of options to pass during device linking stage.</td>
</tr>
<tr>
<td>-device-libs, --device-libs '-L&lt;path&gt; -l&lt;name&gt;',...</td>
<td>Specify the list of device libraries including their names (in -l) and their paths (in -L). FlexIO libraries are linked by default.</td>
</tr>
<tr>
<td>-I, --common-include-path &lt;path&gt;,...</td>
<td>Specify include search paths common to host and device code compilation. FlexIO headers paths are included by DPACC by default.</td>
</tr>
<tr>
<td>-o, --output-file &lt;file&gt;</td>
<td>Specify name and location of the output file.</td>
</tr>
<tr>
<td></td>
<td>• Compile-only mode – name of the output DPA object file. If not specified, .dpa.o is generated for each .c file.</td>
</tr>
<tr>
<td></td>
<td>• Compiler-and-link mode – name of the output DPA program. This is a mandatory option in compiler-and-link mode.</td>
</tr>
<tr>
<td></td>
<td>• Library generation mode – name of the output library. This is a mandatory option for this mode. Output files &lt;name&gt;_device.a and &lt;name&gt;_host.a are generated.</td>
</tr>
<tr>
<td>-hostcc-options, --hostcc-options &lt;options&gt;,...</td>
<td>Specify the list of options to pass to the host compiler.</td>
</tr>
<tr>
<td>-gen-libs, --gen-libs</td>
<td>Generate a DPA library from input files</td>
</tr>
<tr>
<td>-ldoca_dpa, --ldoca_dpa</td>
<td>Link with DOCA-DPA libraries</td>
</tr>
</tbody>
</table>

**Note**

Using machine dependent options (e.g., -mcpu, -march, -mabi) through -devicecc-options to influence compiler code generation is not supported.
DPA Hardware Architectures

The following table mentions the DPA architectures, the associated values supported in the compiler through the `-mcpu` option, and the macros defined by the compiler to identify these architectures.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Value</th>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectX-7</td>
<td>nv-dpa-cx7</td>
<td>__NV_DPA_CX7</td>
</tr>
<tr>
<td>BlueField-3</td>
<td>nv-dpa-bf3</td>
<td>__NV_DPA_BF3</td>
</tr>
</tbody>
</table>

Since ConnectX-7 and BlueField-3 share the same DPA hardware, `nv-dpa-cx7` is treated as an alias of `nv-dpa-bf3` by the compiler.

Architecture Macros
As described in section "DPA Hardware Architectures", the compiler defines identifier macros for each version of DPA hardware. Each identifier macro has a unique integer value which is strictly greater than that of macros for older DPA CPU models. Known aliases such as BlueField-3 DPA and ConnectX-7 DPA share the same integer value. The macro \_NV\_DPA is defined to the value of current compilation target. This can be used to write device code specific to a DPA hardware generation as shown in the following:

```c
#if __NV_DPA == __NV_DPA_BF3
// Code for Bluefield-3 here
#elif __NV_DPA > __NV_DPA_BF3
// Code for devices after Bluefield-3 here
#endif
```

**Note**

The ordering established by the value of the hardware version identifier macros does not imply an ordering of features supported by hardware. It is the user responsibility to ensure that features used in the code which are specific for a DPA version are actually supported on the hardware.

**LTO Usage Guidelines**

**Restrictions**

- Only the default linker script is supported with LTO
- Using options -fPIC/-fpic/-shared/-mcm=large through -devicecc-options is not supported when LTO is enabled
- Fat objects containing both LLVM bitcode and ELF representation are not supported
- Thin LTO is not supported
Compatibility

During compilation, LLVM generates the object as bitcode IR (intermediate representation) when LTO is enabled instead of ELF representation. The bitcode IR generated by the DPA compiler is only guaranteed to be compatible within the same version of DPACC. All objects involved in link-time optimization (enabled with -flto) must be built with the same version of DPACC.

 Deprecated Features

- The -ldpa option which links with DOCA-DPA libraries is deprecated and will be removed in future releases. Use the option -ldoca_dpa instead.

Examples

This section provides some common use cases of DPACC and showcases the dpacc command.

Building Libraries

This example shows how to build DPA libraries using DPACC. Libraries for DPA typically contain two archives, one for the host and one for the device.

```
dpacc input.c -hostcc=gcc -o lib<name> -gen-libs -hostcc-options="-fPIC"
```

This command generates the output files lib<name>_host.a and lib<name>_device.a.

The host stub archive can be linked with other host code to generate a shared/static host library.

- Generating a static host library:
Generating a shared host library:

```
ar x lib<name>_host.a               # Extract objects to generate *.o
ar cr lib<name>.a <*src.host.o> *.o  # Generate final static archive with all objects
```

```
gcc -shared -o lib<name>.so <*src.host.o> -Wl,-whole-archive -l<name>_host -Wl,-no-whole-archive  # Link the generated archive to build a shared library
```

**Linking with DPA Device Library**

The DPA device library generated by DPACC using `-gen-libs` as part of a DPA library can be consumed by DPACC using the `-device-libs` option.

```
dpacc input.c -hostcc=gcc -o libInput.a -device-libs="-L <path-to-library> -l<libName>"
```

**Enabling Link-time Optimizations**

Link-time optimizations can be enabled using `-flto` along with an optimization level specified for device compilation.

```
dpacc input1.c -hostcc=gcc -c -flto -devicecc-options="-O2" dpacc input2.c -hostcc=gcc -c -flto -devicecc-options="-O2" dpacc input1.dpa.o input2.dpa.o -hostcc=gcc -o libInput.a
```

**Including Headers**

This example includes headers for device compilation using `devicecc-options` and host compilation using `hostcc-options`. You may also specify headers for any compilation on both the host and device side using the `-I` option.
DPACC provides an option, `-src-output`, to generate the output as host source code. This source can be compiled by the host compiler to generate functionally equivalent output which DPACC would have generated directly.

This example shows how to build various outputs of DPACC as source using this option and how to compile the generated source.

**DPA-program Source**

Generate DPA-program source by passing the following option to DPACC:

```
dpacc input.c -hostcc=gcc -o libfoo.c -src-output
```

Compile the generated source using host compiler to generate an object and build an archive with this object. A macro `__DPACC_SRC_TARGET__` must be defined when building this object to remove code which is unnecessary when building from source:

```
$ gcc libfoo.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-parameter
            -Wno-return-type -Wno-implicit-function-declaration -D__DPACC_SRC_TARGET__
$ ar cr libfoo.a libfoo.o
```

**DPA-library Source**

Generate DPA-library source by passing the following option to DPACC:

```
dpacc input.c -hostcc=gcc -o foo -gen-libs -src-output
```
This generates the device archive `libfoo_device.a` and host code files `libfoo.lib.c` and `input.dpa.c`.

The host archive of DPA-library is generated by compiling these sources and building an archive. The `__DPACC_SRC_TARGET__` macro must be defined in this instance to remove unnecessary code:

$$\texttt{gcc libfoo.lib.c input.dpa.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-parameter -Wno-return-type -Wno-implicit-function-declaration -D__DPACC_SRC_TARGET__}$$

$$\texttt{ar cr libfoo_host.a libfoo.lib.o input.dpa.o}$$

**DPA-object Source**

Generate DPA-object source by passing the following option to DPACC:

```
dpacc input.c -hostcc=gcc -c -src-output
```

This generates a single file, `input.dpa.c`.

Compile the host file to generate an object:

```
gcc input.dpa.c -c -I /opt/mellanox/flexio/include -Wno-attributes -Wno-pedantic -Wno-unused-parameter -Wno-return-type -Wno-implicit-function-declaration
```

**DPA Compiler Usage**

dpa-clang is a compiler driver for accessing the Clang/LLVM compiler, assembler, and linker which accepts C code files or object files and generates an output according to different usage modes.

⚠️ **Note**
Invoking the compiler, assembler, or linker directly may lead to unexpected errors.

Refer to the following resources for more detailed information on Clang:

- *Clang Compiler User’s Manual*
- *Clang command line argument reference*
- *Target-dependent compilation options*

**Compiler Driver Command-line Options**

dpa-clang <list-of-input-files> [other-options]

**Linker Command Line Options**

LLD is the default linker provided in the DPA toolchain. Linker-related options are passed to through the compiler driver.

dpa-clang -Wl,<linker-option>

For more information, please refer to the [LLD command line reference](#).

**dpacc-extract Command Line Options**

dpacc-extract is a tool for extracting a device executable out of a DPA program or a host executable containing DPA program(s).

To execute dpacc-extract:
Usage: dpacc-extract <input-file> -o=<output-file> [other options]

Helper Flags:

- `--output-file` Specify name of the output file
- `--app-name <name>` Specify name of the DPA application to extract
- `--help` Print help information about dpacc-extract
- `--version` Print dpacc-extract version information
- `--options-file <file>`,... Include command line options from the specified file

Mandatory arguments:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input file</td>
<td>DPA program or host executable containing DPA program. Specifying one input file is mandatory.</td>
</tr>
<tr>
<td>-o, --output-file &lt;file&gt;</td>
<td>Specify name and location of the output device executable.</td>
</tr>
<tr>
<td>-app-name, --app-name &lt;name&gt;</td>
<td>Specify name of the DPA application to extract. Mandatory if input file has multiple DPA apps.</td>
</tr>
</tbody>
</table>

**Objdump Command Line Options**

The dpa-objdump utility prints the contents of object files and final linked images named on the command line.

For more information, please refer to the Objdump command line reference.

Commonly used dpa-objdump options:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--mcpu=nv-dpa-bf3</td>
<td>Option to choose micro-architecture for DPA processor. nv-dpa-bf3 is the default CPU for dpa-objdump.</td>
</tr>
</tbody>
</table>

**Archiver Command Line Options**
dpa-ar is a Unix ar-compatible archiver.

For more information, please refer to the Archiver command line reference.

**NM Tool Command Line Options**

The dpa-nm utility lists the names of symbols from object files and archives.

For more information, please refer to the NM tool command line reference.

**Common Compiler Options**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--mcpu=nv-dpa-bf3</td>
<td>Option to choose micro-architecture and ABI for DPA processor. nv-dpa-bf3 is the default CPU for the compiler.</td>
</tr>
<tr>
<td>-mrelax/-mno-relax</td>
<td>Option to enable/disable linker relaxations.</td>
</tr>
<tr>
<td>-I &lt;dir&gt;</td>
<td>Option to include header files present in &lt;dir&gt;.</td>
</tr>
</tbody>
</table>

**Common Linker Options**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Wl,-L &lt;path-to-library&gt; -Wl,-l&lt;library-name&gt;</td>
<td>Option to link against libraries</td>
</tr>
</tbody>
</table>

**Note**

Linker options are provided through the compiler driver dpa-clang.
Note

The LLD linker script is honored in addition to the default configuration rather than replacing the whole configuration like in GNU ld. Hence, additional options may be required to override some default behaviors.

Debugging Options

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fdebug-macro</td>
<td>Option to emit macro debugging information. This option enables macro-debugging similar to GCC option -g3.</td>
</tr>
</tbody>
</table>

Miscellaneous Notes

- Objects produced by LLD are not compatible with those generated by any other linker.
- The default debugging standard of the DPA compiler is DWARFv5. GDB versions <10.1 have issues processing some DWARFv5 features. Use the option -devicecc-options="-gdwarf-4" with DPACC to debug with GDB versions <10.1.