arm

Arm® Mali™ Bifrost and Valhall OpenCL

Version 4.10

Developer Guide

Arm® Mali™ Bifrost and Valhall OpenCL
Developer Guide

Copyright © 2019–2022 Arm Limited (or its affiliates). All rights reserved.

Release Information

Document history

Issue	Date	Confidentiality	Change
0100-00	15 February 2019	Non-Confidential	First release of version 1.0
0200-00	12 April 2019	Non-Confidential	First release of version 2.0
0300-00	28 June 2019	Non-Confidential	First release of version 3.0
0301-00	1 August 2019	Non-Confidential	First release of version 3.1
0302-00	27 November 2019	Non-Confidential	First release of version 3.2
0400-00	29 January 2020	Non-Confidential	First release of version 4.0
0401-00	14 August 2020	Non-Confidential	First release of version 4.1
0402-00	30 September 2020	Non-Confidential	First release of version 4.2
0403-00	11 November 2020	Non-Confidential	First release of version 4.3
0404-00	25 February 2021	Non-Confidential	First release of version 4.4
0405-00	2 July 2021	Non-Confidential	First release of version 4.5
0406-00	25 August 2021	Non-Confidential	First release of version 4.6
0407-00	15 October 2021	Non-Confidential	First release of version 4.7
0408-00	10 February 2022	Non-Confidential	First release of version 4.8
0409-00	20 May 2022	Non-Confidential	First release of version 4.9
0410-00	8 July 2022	Non-Confidential	First release of version 4.10

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more

patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at https://www.arm.com/company/policies/trademarks.

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.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

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

Previous issues of this document included language that can be offensive. We have replaced this language. See I.1 Revisions on page 111.

To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

1.1 Product revision status

The r_xp_y identifier indicates the revision status of the product described in this manual, for example, $r1p2$, where:

r_x Identifies the major revision of the product, for example, r1.

p_y Identifies the minor revision or modification status of the product, for example, p2.

1.2 Intended audience

This guide is written for software developers with experience in C or C-like languages who want to develop OpenCL on Mali™ Bifrost and Valhall GPUs.

1.3 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Convention	Use
italic	Citations.
bold	Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` <u>`underline`</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

Convention	Use
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Recommendations. Not following these recommendations might lead to system failure or damage.

Caution

Requirements for the system. Not following these requirements might result in system failure or damage.

Warning

Requirements for the system. Not following these requirements will result in system failure or damage.

Danger

An important piece of information that needs your attention.

Note

A useful tip that might make it easier, better or faster to perform a task.

Tip

A reminder of something important that relates to the information you are reading.

Remember

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

Figure 1-1: Key to timing diagram conventions
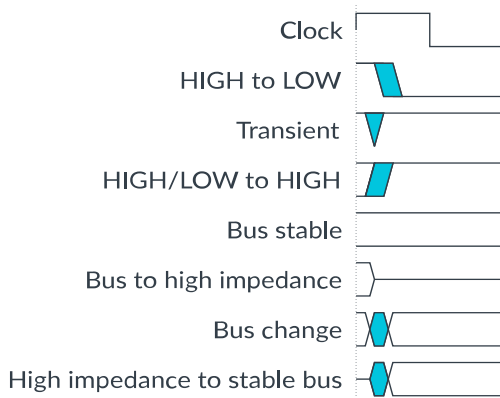
Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name, n denotes an active-LOW signal.

1.4 Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1-2: Arm publications

Document name	Document ID	Licensee only
Arm® Mali™ RenderScript Best Practices Developer Guide	101144	No
Arm® Mali™ Midgard OpenCL Developer Guide	100614	No

Table 1-3: Third-party publications

Name	Link
OpenCL 2.0 Specification	http://www.khronos.org

	Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.
Note	Adobe PDF reader products can be downloaded at http://www.adobe.com

2. Overview

This chapter introduces Mali™ GPUs, OpenCL, and the Mali™ GPU OpenCL driver.

2.1 About Arm® Mali™ GPUs

Arm® produces families of Mali™ GPUs. Bifrost and Valhall are two of the Mali™ GPU architectures.

Mali™ GPUs run data processing tasks in parallel that contain relatively little control code. Mali™ GPUs typically contain many more processing units than application processors. This enables Mali™ GPUs to compute at a higher rate than application processors without using more power.

Mali™ GPUs can have one or more shader cores.

Scalar instructions are executed in parallel so the GPU operates on multiple data elements simultaneously. You are not required to vectorize your code to do this.

2.2 About OpenCL

Open Computing Language (OpenCL) is an open standard that enables you to use the parallel processing capabilities of multiple types of processors including application processors, *Graphics Processing Units* (GPUs), and other computing devices.

OpenCL makes parallel applications easier to write, because it enables the execution of your application across multiple application processors and GPUs.

OpenCL is an open standard developed by the Khronos Group.

Related information

http://www.khronos.org

2.3 About the Mali™ GPU OpenCL driver and support

The Mali™ GPU OpenCL driver is an implementation of OpenCL for Mali™ GPUs. The Mali™ GPU OpenCL driver supports different versions of OpenCL.

The Mali™ Bifrost and Valhall drivers support the OpenCL version 3.0 full profile.

The driver is binary-compatible with OpenCL 1.0, OpenCL 1.1, OpenCL 1.2, OpenCL 2.0, and OpenCL 2.1 applications. The driver is also compatible with the APIs deprecated in OpenCL 2.0.

3. Parallel processing concepts

This chapter describes the main concepts of parallel processing.

3.1 About parallel processing

Parallel processing is the simultaneous processing of multiple computations.

Application processors are typically designed to execute a single thread as quickly as possible. This type of processing typically includes scalar operations and control code.

GPUs are designed to execute a large number of threads at the same time. Graphics applications typically require many operations that can be computed in parallel across many processors.

OpenCL enables you to use the parallel processing capabilities of GPUs or multi-core application processors.

OpenCL is an open standard language that enables developers to run general purpose computing tasks on GPUs, application processors, and other types of processors.

3.2 Types of parallelism

Data parallelism, task parallelism, and pipelines are the main types of parallelism.

3.2.1 Data parallelism

In data parallelism, data is divided into data elements that a processor can process in parallel. Several different processors simultaneously read and process different data elements.

The data must be in data structures that processors can read and write in parallel.

An example of a data parallel application is rendering three-dimensional graphics. The generated pixels are independent so the computations required to generate them can be performed in parallel. This type of parallelism is very fine-grained and can involve hundreds of thousands of active threads simultaneously.

OpenCL is primarily used for data parallel processing.

3.2.2 Task parallelism

In task parallelism, the application is broken down into smaller tasks that execute in parallel. Task parallelism is also known as functional parallelism.

An example of an application that can use task parallelism is playing a video in a web page. To display a video in a web page, your device must do several tasks:

- Run a network stack that performs communication.

- Request data from an external server.

- Read data from an external server.

- Parse data.

- Decode video data.

- Decode audio data.

- Draw video frames.

- Play audio data.

The following figure shows parts of an application and operating system that operate simultaneously when playing an on-line video.

Figure 3-1: Task parallel processing
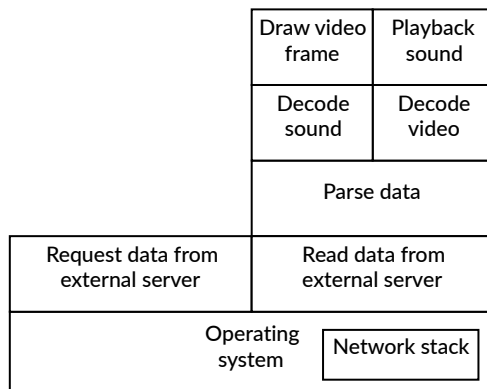
3.2.3 Pipelines

Pipelines process data in a series of stages. In a pipeline, the stages can operate simultaneously but they do not process the same data. A pipeline typically has a relatively small number of stages.

An example of a pipeline is a video recorder application that must execute these stages:

1. Capture image data from an image sensor and measure light levels.

2. Modify the image data to correct for lens effects.

3. Modify the contrast, color balance, and exposure of the image data.

4. Compress the image.

5. Add the data to the video file.

6. Write the video file to storage.

These stages must be executed in order, but they can all execute on data from different video frames at the same time.

The figure shows parts of a video capture application that can operate simultaneously as a pipeline.

Figure 3-2: Pipeline processing
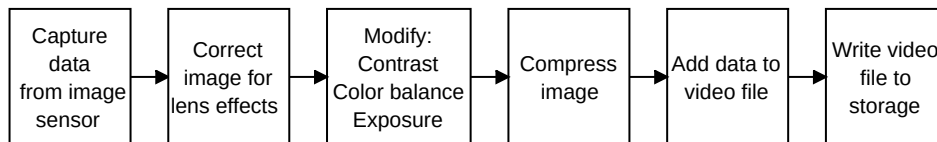
3.3 Mixing different types of parallelism

You can mix different types of parallelism in your applications.

For example, an audio synthesizer might use a combination of all three types of parallelism, in these ways:

- Task parallelism is used to compute the notes independently.

- A pipeline of audio generation and processing modules creates the sound of an individual note.

- Within the pipeline, some stages can use data parallelism to accelerate the computation of processing.

3.4 Embarrassingly parallel applications

If an application can be parallelized across a large number of processors easily, it is said to be embarrassingly parallel.

OpenCL is ideally suited for developing and executing embarrassingly parallel applications.

The following figure shows an image that is divided into many small parts. If, for example, you want to brighten the image, you can process all of these parts simultaneously.

Figure 3-3: Embarrassingly parallel processing
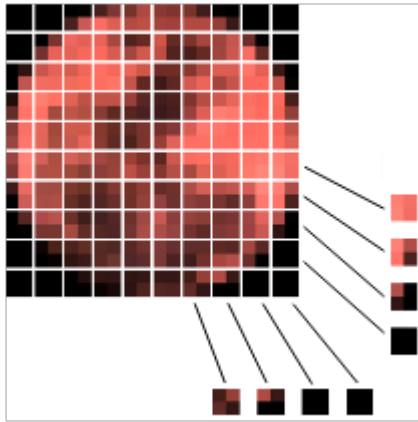
Another example of an embarrassingly parallel application is rendering three-dimensional graphics. For example, pixels are independent so they can be computed and drawn in parallel.

3.5 Limitations of parallel processing and Amdahl's law

There are limitations of parallel processing that you must consider when developing parallel applications.

For example, if your application parallelizes perfectly, executing the application on ten processors makes it run ten times faster. However, applications rarely parallelize perfectly because part of the application is serial. This serial component imposes a limit on the amount of parallelization the application can use.

Amdahl's law describes the maximum speedup that parallel processing can achieve.

The formula for Amdahl's law is shown in the following figure where the terms in the equation are:

S   Fraction of the application that is serial.
P   Fraction of the application that is parallelizable.
N   Number of processors.

Figure 3-4: Formula for Amdahl's law

$$\text{Speedup} = \frac{1}{S + \dfrac{P}{N}}$$

The following figure shows the speedup that different numbers of processors provide for applications with different serial components.

Figure 3-5: Speedup for an application with different serial components
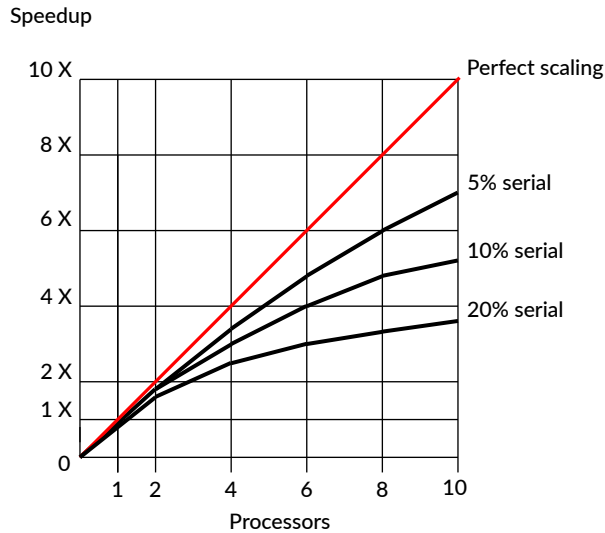
The biggest speedups are achieved with relatively small numbers of processors. However, as the number of processors rises, the per-processor gains are reduced.

You cannot avoid Amdahl's law in your application but you can reduce the impact.

For high performance with a large number of processors, the application must have a very small serial component. These sorts of applications are said to be embarrassingly parallel.

Related information

Embarrassingly parallel applications on page 18
Reducing the effect of serial computations on page 69

3.6 Concurrency

Concurrent applications have multiple operations in progress at the same time. These can operate in parallel or in serial through the use of a time sharing system.

In a concurrent application, multiple tasks attempt to share the same data. Access to this data must be managed to prevent complex problems such as race conditions, deadlocks, and livelocks.

Race conditions

A race condition occurs when two or more threads try to modify the value of one variable at the same time. In general, the final value of the computation will always produce the same value, but when a race condition occurs, the variable can get a different value that depends on the order of the writes.

Deadlocks

A deadlock occurs when two threads become blocked by each other and neither thread can make progress. This can happen when each thread obtains a lock that the other thread requires.

Livelocks

A livelock is similar to deadlock, but the threads keep running. Because of the lock, the threads can never complete their tasks.

Concurrent applications require concurrent data structures. A concurrent data structure is a data structure that enables multiple tasks to gain access to the data with no concurrency problems.

Data parallel applications use concurrent data structures. These are the sorts of data structures that you typically use in OpenCL.

OpenCL includes atomic operations to help manage interactions between threads. Atomic operations provide one thread exclusive access to a data item while it modifies it. The atomic operation enables one thread to read, modify, and write the data item with the guarantee that no other thread can modify the data item at the same time.

> **Note**
>
> OpenCL does not guarantee the order of operation of threads. Threads can start and finish in any order.

---

4. OpenCL concepts

This chapter describes the OpenCL concepts.

4.1 Using OpenCL

Open Computing Language (OpenCL) is an open standard that enables you to use the parallel processing capabilities of multiple types of processors including application processors, *Graphics Processing Units* (GPUs), and other computing devices.

OpenCL specifies an API for parallel programming that is designed for portability:

- It uses an abstracted memory and execution model.

- There is no requirement to know the application processor instruction set.

Functions executing on OpenCL devices are called kernels. These are written in a language called OpenCL C that is based on C99.

The OpenCL language includes vector types and built-in functions that enable you to use the features of accelerators. There is also scope for targeting specific architectures with optimizations.

4.2 OpenCL applications

OpenCL applications consist of two parts: application or host-side code, and OpenCL kernels.

Application, or host-side code

- Calls the OpenCL APIs.

- Compiles the OpenCL kernels.

- Allocates memory buffers to pass data into and out of the OpenCL kernels.

- Sets up command queues.

- Sets up dependencies between the tasks.

- Sets up the *N-Dimensional Range* (NDRanges) that the kernels execute over.

OpenCL kernels

- Written in OpenCL C language.

- Perform the parallel processing.

- Run on compute devices such as application processors or GPU shader cores.

You must write both of these parts correctly to get the best performance.

4.3 OpenCL execution model

The OpenCL execution model includes the host application, the context, and the operation of OpenCL kernels.

The host application

> The host application runs on the application processor. The host application manages the execution of the kernels by setting up command queues for:
>
> - Memory commands.
>
> - Kernel execution commands.
>
> - Synchronization.

The context

> The host application defines the context for the kernels. The context includes:
>
> - The kernels.
>
> - Compute devices.
>
> - Program objects.
>
> - Memory objects.

Operation of OpenCL kernels

> Kernels run on compute devices. A kernel is a block of code that is executed on a compute device in parallel with other kernels. Kernels operate in the following sequence:
>
> 1. The kernel is defined in a host application.
>
> 2. The host application submits the kernel for execution on a compute device. A compute device can be an application processor, GPU, or another type of processor.
>
> 3. When the application issues a command to submit a kernel, OpenCL creates the NDRange of work-items.
>
> 4. An instance of the kernel is created for each element in the NDRange. This enables each element to be processed independently in parallel.

4.4 OpenCL data processing

The data processed by OpenCL is in an index space of work-items.

The work-items are organized in an NDRange where:

- N is the number of dimensions.

- N can be one, two, or three.

One kernel instance is executed for each work-item in the index space.

The following figure shows NDRanges with one, two, and three dimensions.

Figure 4-1: NDRanges and work-items
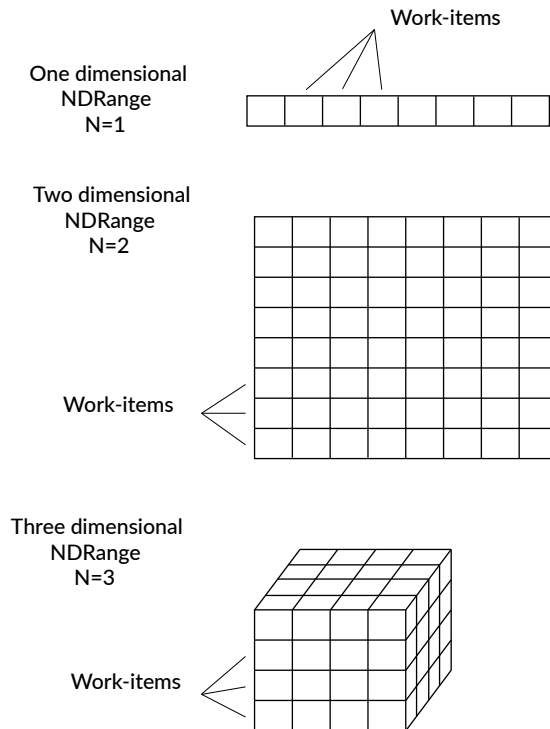
You group work-items into work-groups for processing. The following figure shows a three-dimensional NDRange that is split into 16 work-groups, each with 16 work-items.

Figure 4-2: Work-items and work-groups.
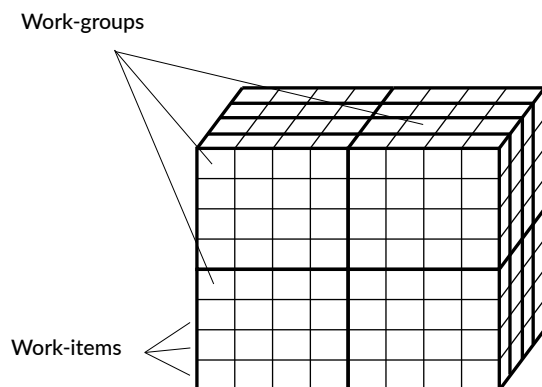
4.5 OpenCL work-groups

Work-groups have several properties, limitations and work-items:

Properties of work-groups

- Work-groups are independent of each other.

- The OpenCL driver can issue multiple work-groups for execution in parallel.

- The work-items in a work-group can communicate with each other using shared data buffers. You must synchronize access to these buffers.

Limitations between work-groups

Work-groups typically do not directly share data. They can share data using global memory.

The following are not supported across different work-groups:

- Barriers.

- Dependencies.

- Ordering.

- Coherency.

Global atomics are available but these can be slower than local atomics.

Work-items in a work-group

The work-items in a work-group can:

- Access shared memory.

- Use local atomic operations.

- Perform barrier operations to synchronize execution points.

    For example:

```
barrier(CLK_LOCAL_MEM_FENCE); // Wait for all work-items in
                              // this work-group to catch up
```

After the synchronization is complete, all writes to shared buffers are guaranteed to have been completed. It is then safe for work-items to read data written by different work-items within the same work-group.

4.6 OpenCL identifiers

There are several identifiers in OpenCL. These identifiers are the global ID, the local ID, and the work-group ID.

global ID

Every work-item has a unique *global ID* that identifies it within the index space.

work-group ID

Each work-group has a unique *work-group ID*.

local ID

Within each work-group, each work-item has a unique *local ID*.

4.7 OpenCL memory model

The OpenCL memory model contains several components and supports a number of memory types.

4.7.1 OpenCL memory model overview

The following figure shows the OpenCL memory model.

Figure 4-3: OpenCL memory model
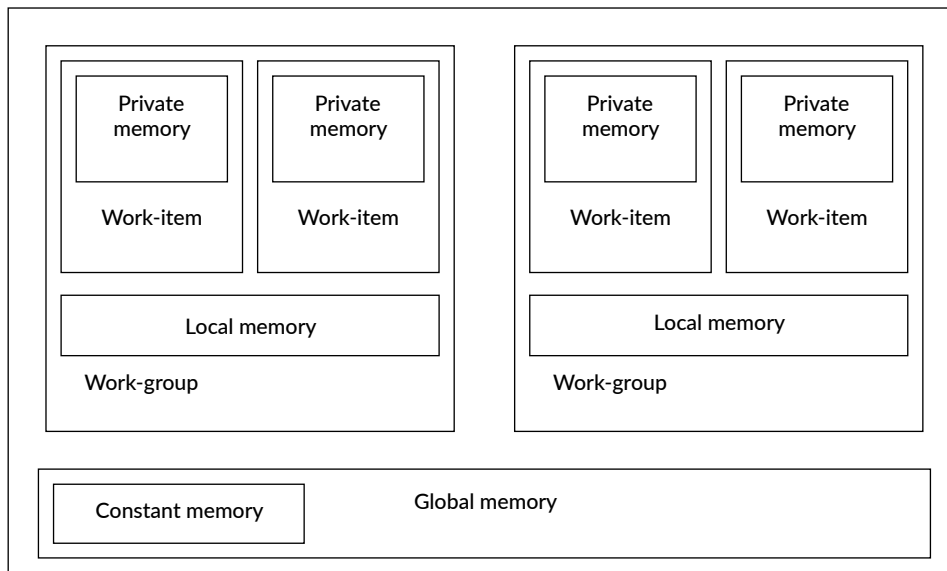
4.7.2 Memory types in OpenCL

OpenCL supports these memory types: Private memory, local memory, constant memory, and global memory.

Private memory

- Private memory is specific to a work-item.

- It is not visible to other work-items.

Local memory

- Local memory is local to a work-group.

- It is accessible by the work-items in the work-group.

- It is accessed with the `__local` keyword.

- It is consistent to all work-items in the work-group.

> **Note**
>
> Work-items execute in an undefined order. This means you cannot guarantee the order that work-items write data in. If you want a work-item to read data that are written by another work-item, you must use a barrier to ensure that they execute in the correct order.

Constant memory

- Constant memory is a memory region used for objects allocated and initialized by the host.

- It is accessible as read-only by all work-items.

Global memory

- Global memory is accessible to all work-items executing in a context.

- It is accessible to the host using `read`, `write`, and `map` commands.

- It is consistent across work-items in a single work-group.

> **Note**
>
> Work-items execute in an undefined order. This means you cannot guarantee the order that work-items write data in.
>
> If you want a work-item to read data that are written by another work-item, you must use a barrier to ensure that they execute in the correct order.

- It implements a relaxed consistency, shared memory model.

- It is accessed with the `__global` keyword.

- There is no guarantee of memory consistency between different work-groups.

4.8 Mali™ GPU OpenCL memory model

Mali™ GPUs use a different memory model compared to desktop workstation GPUs.

The main differences between desktop workstation GPUs and Mali™ GPUs are:

Desktop workstation GPUs

Traditional desktop workstation processors have their own dedicated memory.

Desktop workstation GPUs have physically separate global, local, and private memories.

Typically, a graphics card has its own memory.

Data must be copied from the application processor memory to the GPU memory and back again.

Mali™ GPUs

Mali™ GPUs have a unified memory system with the application processor.

Mali™ GPUs use global memory backed with caches in place of local or private memories.

If you allocate local or private memory, it is allocated in global memory. Moving data from global to local memory typically does not improve performance. If the data fits into the registers, moving data from global to private memory can result in improved performance.

Copying data is not required, provided it is allocated by OpenCL in the correct manner.

Each compute device, that is, shader core, has its own data caches.

Related information

Optimizing memory allocation on page 58

4.9 OpenCL concepts summary

Summary of the concepts used in OpenCL.

- OpenCL primarily uses data parallel processing.
- Computations in OpenCL are performed by pieces of code called kernels that execute on compute devices. Compute devices can be application processors or GPUs.
- The data processed by OpenCL is in an index space of work-items. The work-items are organized in an NDRange.
- One kernel instance is executed for each work-item in the index space.
- Kernel instances can execute in parallel.
- You group work-items together to form work-groups. The work-items in a work-group can communicate with each other using shared data buffers, but access to the buffers must be synchronized with barrier operations.
- Work-groups typically do not directly share data with each other. They can share data using global memory and atomic operations.
- You can issue multiple work-groups for execution in parallel.

5. Developing an OpenCL application

This chapter describes the development stages of an OpenCL application.

5.1 Software and hardware requirements for Mali™ GPU OpenCL development

You can develop an OpenCL application with any implementation so long as the implementation has the wanted features and extensions.

If you want to tune for high performance though, you require:

- A compatible OS.
- The Mali™ GPU OpenCL driver.
- A platform with a Mali™ GPU.

Estimating Mali™ GPU performance with results from a different system will produce inaccurate data.

Related information

5.2 Development stages for OpenCL

There are several stages to develop an OpenCL application. First, you must determine what you want to parallelize. Then, you must write the kernels. Finally, write infrastructure for the kernels and execute them.

You must perform the following stages to develop and OpenCL application:

Determine what you want to parallelize

The first step when deciding to use OpenCL is to look at what your application does, and identify the parts of the application that can run in parallel. This is often the hardest part of developing an OpenCL application.

Note   It is only necessary to convert the parts of an application to OpenCL where there is likely to be a benefit. Profile your application to find the most active parts and consider these parts for conversion.

Write kernels

OpenCL applications consist of a set of kernel functions. You must write the kernels that perform the computations.

If possible, partition your kernels so that the least amount of data is transferred between them. Loading large amounts of data is often the most expensive part of an operation.

Write infrastructure for kernels

OpenCL applications require infrastructure code that sets up the data and prepares the kernels for execution.

Execute the kernels

Enqueue the kernels for execution and read back the results.

Related information

6. Execution stages of an OpenCL application

This chapter describes the execution stages of an OpenCL application.

Note   This chapter is not intended as a comprehensive guide to using OpenCL.

6.1 About the execution stages

Platform setup and runtime setup are the two main parts of the OpenCL execution stages. Your OpenCL application must obtain information about your hardware, then set up the runtime environment.

6.1.1 Platform setup

Use the platform API to obtain information about your hardware, then set up the OpenCL context.

The platform API helps you to:

- Determine what OpenCL devices are available. Query to find out what OpenCL devices are available on the system using OpenCL platform layer functions.
- Set up the OpenCL context. Create and set up an OpenCL context and at least one command queues to schedule execution of your kernels.

Related information

6.1.2 Runtime setup

You can use the runtime API for many different operations.

The runtime API helps you to:

- Create a command queue.
- Compile and build your program objects. Issue commands to compile and build your source code and extract kernel objects from the compiled code.
  You must follow this sequence of commands:
  1. Create the program object by calling either:

clCreateProgramWithSource()

> Creates the program object from the kernel source code.

clCreateProgramWithBinary()

> Creates the program with a pre-compiled binary file.

2. Call the `clBuildProgram()` function to compile the program object for the specific devices on the system.

- Build a program executable.

- Create the kernel and memory objects:

1. Call the `clCreateKernel()` function for each kernel, or call the `clCreateKernelsInProgram()` function to create kernel objects for all the kernels in the OpenCL application.

2. Use the OpenCL API to allocate memory buffers. You can use the map and unmap operations to enable the application processor to access the data.

- Enqueue and execute the kernels.

Enqueue to the command queues the commands that control the sequence and synchronization of kernel execution, mapping and unmapping of memory, and manipulation of memory objects.

To execute a kernel function, you must do the following steps:

1. Call `clSetKernelArg()` for each parameter in the kernel function definition to set the kernel parameter values.

2. Determine the work-group size and the index space to use to execute the kernel.

3. Enqueue the kernel for execution in the command queue.

- Enqueue commands that make the results from the work-items available to the host.

- Clean up the unused objects.

Related information

6.2 Finding the available compute devices

To set up OpenCL, you must choose compute devices. Call clGetDeviceIDs() to query the OpenCL driver for a list of devices on the machine that support OpenCL.

You can restrict your search to a particular type of device or to any combination of device types. You must also specify the maximum number of device IDs that you want returned.

If you have two or more devices, you can schedule different NDranges for processing on the devices.

6.3 Initializing and creating OpenCL contexts

When you know the available OpenCL devices on the machine and have at least one valid device ID, you can create an OpenCL context. The context groups the devices together to enable memory objects to be shared across different compute devices.

To share work between devices, or to have interdependencies between operations submitted to more than one command queue, create a context containing all the devices you want to use in this way.

Pass the device information to the `clCreateContext()` function. For example:

```
// Create an OpenCL context

context = clCreateContext(NULL, 1, &device_id, notify_function, NULL, &err);
if (err != CL_SUCCESS)
{
        Cleanup();
        return 1;
}
```

You can optionally specify an error notification callback function when you create an OpenCL context. When you leave this parameter as a `NULL` value, the system does not register an error notification function.

To receive runtime errors for the particular OpenCL context, provide the callback function. For example:

```
//        Optionally user_data can contain contextual information
//        Implementation specific data of size cb, can be returned in private_info

void context_notify(const char *notify_message, const void *private_info,
                size_t cb, void *user_data)
{
        printf("Notification:\n\t%s\n", notify_message);
}
```

6.4 Creating a command queue

After creating your OpenCL context, use clCreateCommandQueue() to create a command queue.

OpenCL does not support the automatic distribution of work to devices. If you want to share work between devices, or have dependencies between operations enqueued on devices, then you must create the command queues in the same OpenCL context.

Example command queue:

```
// Create a command-queue on the first device available
// on the created context

commandQueue = clCreateCommandQueue(context, device, properties, errcode_ref);
if (commandQueue == NULL)
{
        Cleanup();
        return 1;
}
```

If you have multiple OpenCL devices, you must:

1. Create a command queue for each device.

2. Divide up the work.

3. Submit commands separately to each device.

6.5 Creating OpenCL program objects

Create an OpenCL program object.

The OpenCL program object encapsulates the following components:

*   Your OpenCL program source.

*   The latest successfully built program executable.

*   The build options.

*   The build log.

*   A list of devices the program is built for.

The program object is loaded with the kernel source code, then the code is compiled for the devices attached to the context. All kernel functions must be identified in the application source with the `__kernel` qualifier. OpenCL applications can also include functions that you can call from your kernel functions.

Load the OpenCL C kernel source and create an OpenCL program object from it.

To create a program object, use the `clCreateProgramWithSource()` function. For example:

```
//        Create OpenCL program
```

```
program = clCreateProgramWithSource(context, device, "<kernel source>");
if (program == NULL)
{
        Cleanup();
        return 1;
}
```

There are different options for building OpenCL programs:

- You can create a program object directly from the source code of an OpenCL application and compile it at runtime. Do this at application start-up to save compute resources while the application is running.

  If you can cache the binary between application invocations, compile the program object at platform start-up.

- To avoid compilation overhead at runtime, you can build a program object with a previously built binary.

> **Note**
>
> Applications with pre-built program objects are not portable across platforms and driver versions.

Creating a program object from a binary is a similar process to creating a program object from source code, except that you must supply the binary for each device that you want to execute the kernel on. Use the `clCreateProgramWithBinary()` function to do this.

Use the `clGetProgramInfo()` function to obtain the binary after you have generated it.

So that the binary contains the final machine code, calls to `clCreateKernel()` or `clCreateKernelsInProgram()` must be performed before calling `clGetProgramInfo()`. Building programs created using `clCreateProgramWithBinary()` on binaries obtained in this way do not require compilation.

6.6 Building a program executable

When you have created a program object, you must build a program executable from the contents of the program object. Use the clBuildProgram() function to build your executable.

Compile all kernels in the program object:

```
err = clBuildProgram(program, 1, &device_id, "", NULL, NULL);
if (err != CL_SUCCESS)
{
        Cleanup();
        return 1;
}
```

6.7 Creating kernel and memory objects

There are separate processes for creating kernel objects and memory objects. You must create the kernel objects and memory objects.

6.7.1 Creating kernel objects

Call the clCreateKernel() function to create a single kernel object, or call the clCreateKernelsInProgram() function to create kernel objects for all the kernels in the OpenCL application.

For example:

```
//        Create OpenCL kernel

kernel = clCreateKernel(program, "<kernel_name>", NULL);
if (kernel == NULL)
{
      Cleanup();
      return 1;
}
```

6.7.2 Creating memory objects

When you have created and registered your kernels, send the program data to the kernels.

Procedure
1. Package the data in a memory object.
2. Associate the memory object with the kernel.
   These are the types of memory objects:

   Buffer objects
       Simple blocks of memory.

   Image objects
       These are structures specifically for representing images. These are opaque structures. This means that you cannot see the implementation details of these structures.

To create buffer objects, use the `clCreateBuffer()` function.

To create image objects, use the `clCreateImage()` function.

6.8 Executing the kernel

There are several stages in the kernel execution. The initial stages are related to determining work-group and work-item sizes, and data dimensions. After completing the initial stages, you can enqueue and execute your kernel.

6.8.1 Determining the data dimensions

If your data is an image *x* pixels wide by *y* pixels high, it is a two-dimensional data set. If you are dealing with spatial data that involves the x, y, and z position of nodes, it is a three-dimensional data set.

The number of dimensions in the original data set does not have to be the same in OpenCL. For example, you can process a three-dimensional data set as a single dimensional data set in OpenCL.

6.8.2 Determining the optimal global work size

The global work size is the total number of work-items required for all dimensions combined.

You can change the global work size by processing multiple data items in a single work-item. The new global work size is then the original global work size divided by the number of data items processed by each work-item.

The global work size must be greater than the maximum number of threads that can run on a core, multiplied by the number of cores available to fully load the GPU. This global work size is not always achievable though, and sometimes running fewer threads may result in better performance, for example, when there is contention on the memory subsystem.

6.8.3 Determining the local work-group size

You can specify the size of the work-group that OpenCL uses when you enqueue a kernel to execute on a device. To do this, you must know the maximum work-group size permitted by the OpenCL device your work-items execute on. To find the maximum work-group size for a specific kernel, use the clGetKernelWorkGroupInfo() function and request the CL_KERNEL_WORK_GROUP_SIZE property.

If your application is not required to share data among work-items, set the `local_work_size` parameter to `NULL` when enqueuing your kernel. This enables the OpenCL driver to determine an efficient work-group size for your kernel, but this might not be the optimal work-group size.

To get the maximum work-group size in each dimension, call `clGetDeviceInfo()` with `CL_DEVICE_MAX_WORK_ITEM_SIZES`. This provides maximum sizes for the simplest kernel, and dimensions might be lower for more complex kernels. The product of the dimensions of your work-group might limit the size of the work-group.

> **Note**
>
> To get the maximum work-group size for a specific kernel, call `clGetKernelWorkGroupInfo()` with `CL_KERNEL_WORK_GROUP_SIZE`.

6.8.4 Enqueuing kernel execution

When you have identified the dimensions necessary to represent your data, the necessary work-items for each dimension, and an appropriate work-group size, enqueue the kernel for execution using `clEnqueueNDRangeKernel()`.

For example:

```
size_t globalWorkSize[1] = { ARRAY_SIZE };
size_t localWorkSize[1] = { 4 };

//   Queue the kernel up for execution across the array

errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, globalWorkSize,
 localWorkSize, 0, NULL, NULL);

if (errNum != CL_SUCCESS)
{
      printf("Error queuing kernel for execution.\n");
      Cleanup();
      return 1;
}
```

6.8.5 Executing kernels

Queuing the kernel for execution does not mean that it executes immediately. The kernel execution is put into the command queue so the device can process it later.

The call to `clEnqueueNDRangeKernel()` is not a blocking call and returns before the kernel has executed. It can sometimes return before the kernel has started executing.

It is possible to make a kernel wait for execution until previous events are finished. You can specify certain kernels wait until other specific kernels are completed before executing.

Kernels are executed in the order they are enqueued unless the property `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` is set when the command queue is created.

Kernels that are enqueued to an in-order queue automatically wait for kernels that were previously enqueued on the same queue. You are not required to write any code to synchronize them.

> **Note**
>
> Any execution failure may invalidate the queue and the context. The status of the associated event will have an error code and any further enqueue to a queue that may become invalid will return an error.

6.9 Reading the results

After your kernels have finished execution, you must make the result accessible to the host. To access the results from the kernel, use `clEnqueueMapBuffer()` to map the buffer into host memory.

For example:

```
local_buffer = clEnqueueMapBuffer(queue, buffer, CL_NON_BLOCKING, CL_MAP_READ, 0,
            (data_size, num_deps, &deps[0], NULL, &err);

ASSERT(CL_SUCCESS == err);
```

> **Note**
> - `clFinish()` must be called to make the buffer available.
> - The third parameter of `clEnqueueMapBuffer()` is `CL_NON_BLOCKING` in the previous example. If you change this parameter in `clEnqueueMapBuffer()` or `clFinish()` to `CL_BLOCKING`, the call becomes a blocking call and the read must be completed before `clEnqueueMapBuffer()` returns.

6.10 Cleaning up unused objects

When the application no longer requires the objects associated with the OpenCL runtime and context, you must release these resources. You can use several functions to release your OpenCL objects.

These functions decrement the reference count for the associated object:

- `clReleaseMemObject()`.
- `clReleaseKernel()`.
- `clReleaseProgram()`.
- `clReleaseCommandQueue()`.
- `clReleaseContext()`.

Ensure the reference counts owned by the application reach zero when your application no longer requires them.

7. Converting existing code to OpenCL

This chapter describes converting existing code to OpenCL.

7.1 Profiling your application

Profile your application to find the most compute intensive parts. These are the parts that might be worth porting to OpenCL.

The proportion of an application that requires high performance is often a relatively small part of the code. This is the part of the code that can make best use of OpenCL. Porting any more of the application to OpenCL is unlikely to provide a benefit.

You can use profilers, such as DS-5 Streamline, to analyze the performance of your application.

Related information

http://malideveloper.arm.com

7.2 Analyzing code for parallelization

Analyze compute-intensive code and determine the difficulty of parallelization, by checking for parallel operations, operations with few dependencies, and by analyzing different types of loops. These factors affect the difficulty of the parallelization.

7.2.1 About analyzing code for parallelization

When you have identified the most compute intensive parts of your application, analyze the code to see if you can run it in parallel.

Parallelizing code can present the following degrees of difficulty:

Straightforward

Parallelizing the code requires small modifications.

Difficult

Parallelizing the code requires complex modifications. If you are using work-items in place of loop iterations, compute variables based on the value of the global ID rather than using a loop counter.

Difficult and includes dependencies

Parallelizing the code requires complex modifications and the use of techniques to avoid dependencies. You can compute values per frame, perform computations in multiple stages, or pre-compute values to remove dependencies.

Appears to be impossible

If parallelizing the code appears to be impossible, this only means that a particular code implementation cannot be parallelized.

The purpose of code is to perform a function. There might be different algorithms that perform the same function but work in different ways. Some of these might be parallelizable.

Investigate different alternatives to the algorithms and data structures that the code uses. These might make parallelization possible.

Related information

7.2.2 Finding data parallel operations

Try to find tasks that do large numbers of operations that complete without sharing data or do not depend on the results from each other. These types of operations are data parallel, so they are ideal for OpenCL.

7.2.3 Finding operations with few dependencies

If tasks have few dependencies, it might be possible to run them in parallel. Dependencies between tasks prevent parallelization because they force tasks to be performed sequentially.

If the code has dependencies, consider:

- If there is a way to remove the dependencies.

- If it is possible to delay the dependencies so that they occur later in execution.

7.2.4 Analyze loops

Loops are good targets for parallelization because they repeat computations many times, often independently.

Consider the following types of loops:

Loops that process few elements

If the loop only processes a relatively small number of elements, it might not be appropriate for data parallel processing.

It might be better to parallelize these sorts of loops with task parallelism on one or more application processors.

Nested loops

If the loop is part of a series of nested loops and the total number of iterations is large, this loop is probably appropriate for parallel processing.

Perfect loops

Look for loops that:

- Process thousands of items.

- Have no dependencies on previous iterations.

- Access data independently in each iteration.

These types of loops are data parallel, so are ideal for OpenCL.

Simple loop parallelization

If the loop includes a variable that is incremented based on a value from the previous iteration, this is a dependency between iterations that prevents parallelization.

See if you can work out a formula that enables you to compute the value of the variable based on the main loop counter.

In OpenCL work-items are processed in parallel, not in a sequential loop. However, work-item processing acts in a similar way to a loop.

Every work-item has a unique *global id* that identifies it and you can use this value in place of a loop counter.

It is also possible to have loops within work-items, but these are independent of other work-items.

Loops that require data from previous iterations

If your loop involves dependencies based on data processed by a previous iteration, this is a more complex problem.

Can the loop be restructured to remove the dependency? If not, it might not be possible to parallelize the loop.

There are several techniques that help you deal with dependencies. See if you can use these techniques to parallelize the loop.

Non-parallelizable loops

If the loop contains dependencies that you cannot remove, investigate alternative methods of performing the computation. These might be parallelizable.

Related information

7.3 Parallel processing techniques in OpenCL

You can use different parallel processing techniques in OpenCL. These techniques include, for example, different ways of computing values, removing dependencies, software pipelining, and task parallelism.

7.3.1 Use the global ID instead of the loop counter

In OpenCL, you use kernels to perform the equivalent of loop iterations. This means that there is no loop counter to use in computations. The global ID of the work-item provides the equivalent of the loop counter. Use the global ID to perform any computations based on the loop counter.

Note

You can include loops in OpenCL kernels, but they can only iterate over the data for that work-item, not the entire NDRange.

7.3.1.1 Simplified loop example

The example shows a simple loop in C that assigns the value of the loop counter to each array element.

Loop example in C:

The following loop fills an array with numbers.

```
void SetElements(void)
{
    int loop_count;
    int my_array[4096];
    for (loop_count = 0; loop_count < 4096; loop_count++)
    {
        my_array[loop_count] = loop_count;
    }
    printf("Final count %d\n", loop_count);
}
```

This loop is parallelizable because the loop elements are all independent. There is no main loop counter `loop_count` in the OpenCL kernel, so it is replaced by the global ID.

The equivalent code in an OpenCL kernel:

```
__kernel void example(__global int * restrict my_array)
{
    int id;
    id = get_global_id(0);
    my_array[id] = id;
}
```

7.3.2 Compute values in a loop with a formula instead of using counters

If you are using work-items in place of loop iterations, compute variables based on the value of the global ID rather than using a loop counter. The global ID of the work-item provides the equivalent of the loop counter.

7.3.3 Compute values per frame

If your application requires continuous updates of data elements and there are dependencies between them, try breaking the computations into discrete units and perform one iteration per image frame displayed.

For example, the following figure shows an application that runs a continuous physics simulation of a flag.

Figure 7-1: Flag simulation

The flag is made up of a grid of nodes that are connected to the neighboring nodes. These nodes are shown in the following figure.

Figure 7-2: Flag simulation grid
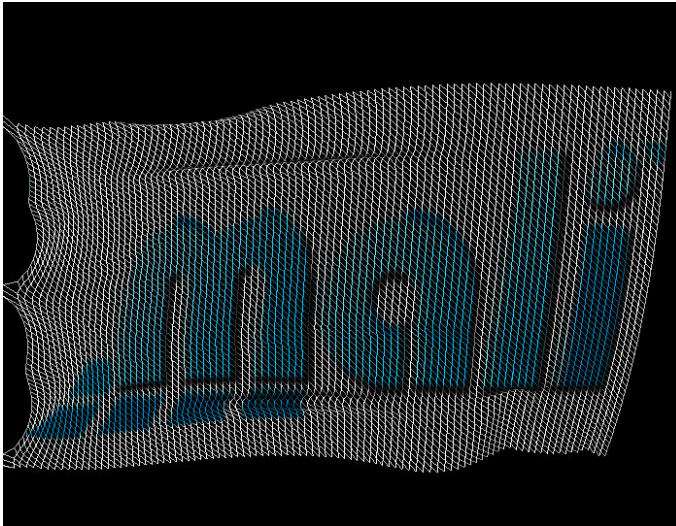
The simulation runs as a series of iterations. In one iteration, all the nodes are updated and the image is redrawn.

The following operations are performed in each iteration:

1. The node values are read from a buffer A.

2. A physics simulation computes the forces between the nodes.

3. The position and forces on the nodes are updated and stored into buffer B.

4. The flag image is drawn.

5. Buffer A and buffer B are switched.

In this case, splitting the computations into iterations also splits the dependencies. The data required for one frame is computed in the previous frame.

Some types of simulation require many iterations for relatively small movements. If this is the case, try computing multiple iterations before drawing frames.

7.3.4 Perform computations with dependencies in multiple-passes

If your application requires continuous updates of data elements and there are dependencies between them, try breaking the computations into discrete units and perform the computations in multiple stages.

This technique extends the technique that computes values per frame by splitting computations even more.

Divide the data elements into odd and even fields. This divides the dependencies so the entire computation can be performed in stages. The processing alternates between computing the odd then the even fields.

For example, this technique can be used in neural network simulation.

The individual neurons are arranged in a three-dimensional grid. Computing the state for a neuron involves reading inputs from the surrounding neurons. This means that each neuron has dependencies on the state of the surrounding neurons.

To execute the simulation, the three-dimensional grid is divided into layers and executed in the following manner:

1. The even node values are read.

2. The odd layers are computed and the results stored.

3. The odd node values are read.

4. The even layers are computed and the results stored.

Related information

7.3.5 Pre-compute values to remove dependencies

If part of your computation is serial, see if it can be removed and performed separately.

For example, the audio synthesis technique Frequency Modulation (FM) works by reading an audio waveform called the carrier. The rate the waveform is read at depends on another waveform called the modulator.

In one type of algorithm, the carrier values are read by a pointer to generate the output waveform. The position of the pointer is computed by taking the previous value and moving it by an amount determined by the value of the modulator waveform.

The position of the pointer has a dependency on the previous value and that value has a dependency on the value before it. This series of dependencies makes the algorithm difficult or impossible to parallelize.

Another approach is to consider that the pointer is moving through the carrier waveform at a fixed speed and the modulator is adding or subtracting an offset. This can be computed in parallel, but the offsets are incorrect because they do not take account of the dependencies on previous offsets.

The computation of the correct offsets is a serial process. If you pre-compute these values, the remaining computation can be parallelized. The parallel component reads from the generated offset table and uses this to read the correct value from the carrier waveform.

There is a potential problem with this example. The offset table must be recomputed every time the modulating waveform changes. This is an example of Amdahl's law. The amount of parallel computation possible is limited by the speed of the serial computation.

7.3.6 Use software pipelining

Software pipelines are a parallel processing technique that enable multiple data elements to be processed simultaneously by breaking the computation into a series of sequential stages.

Pipelines are common in both hardware and software. For example, application processors and GPUs use hardware pipelines. The graphics standard OpenGL ES is based on a virtual pipeline.

In a pipeline, a complete process is divided into a series of stages. A data element is processed in one stage and the results are then passed to the next stage.

Because of the sequential nature of a pipeline, only one stage is used at a time by a particular data element. This means that the other stages can process other data elements.

You can use software pipelines in your application to process different data elements.

For example, a game requires many different operations to happen. A game might use a similar pipeline to this:

1. The input is read from the player.

2. The game logic computes the progress of the game.

3. The scene objects are moved based on the results of the game logic.

4. The physics engine computes positions of all objects in the scene.

5. The game uses OpenGL ES to draw objects on the screen.

7.3.7 Use task parallelism

Task or functional parallelism involves dividing an application by function into different tasks.

For example, an online game can take advantage of task parallelism. To run an online game, your device performs several functions:

*   Communicates with an external server.

*   Reads player input.

*   Updates the game state.

*   Generates sound effects.

*   Plays music.

*   Updates the display.

These tasks require synchronization but are otherwise largely independent operations. This means you can execute the tasks in parallel on separate processors.

Another example of task parallelism is Digital Television (DTV). At any time the television might be performing several of the following operations:

*   Downloading a program.

- Recording a program.

- Updating the program guide.

- Displaying options.

- Reading data from media storage device.

- Playing a program.

- Decoding a video stream.

- Playing audio.

- Scaling an image to the correct size.

7.4 Using parallel processing with non-parallelizable code

If you cannot parallelize your code, it might still be possible to use parallel processing. The fact that the code cannot be parallelized only means that a specific implementation cannot be parallelized. It does not mean that the problem cannot be solved in a parallel way.

Most code is written to run on application processors that run sequentially. The code uses serial algorithms and non-concurrent data structures. Parallelizing this sort of code can be difficult or impossible.

Investigate the following approaches:

Use parallel versions of your data structures and algorithms

Many common data structures and algorithms that use them are non-concurrent. This prevents you from parallelizing the code.

There are parallel versions of many common data structures and algorithms. You might be able to use these in place of the originals to parallelize the code.

Solve the problem in a different way

Analyze what problem the code solves.

Look at the problem and investigate alternative ways of solving it. There might be alternative solutions that use algorithms and data structures that are parallelizable.

To do this, think in terms of the purpose of the code and data structures.

Typically, the aim of code is to process or transform data. It takes a certain input and produces a certain output.

Consider if the following possibilities are true:
- The data you want to process can be divided into small data elements.

- The data elements can be placed into a concurrent data structure.

- The data elements can be processed independently.

If all three possibilities are true, then you can probably solve your problem with OpenCL.

Related information

7.5 Dividing data for OpenCL

You must split data and use concurrent data structures where possible for processing by OpenCL. This section shows examples for one-, two-, and three-dimensional data.

7.5.1 About dividing data for OpenCL

Data is divided up so it can be computed in parallel with OpenCL.

The data is divided into three levels of hierarchy:

NDRange

The total number of elements in the NDRange is known as the global work size.

Work-groups

The NDRange is divided into work-groups.

Sub-groups

On devices that support OpenCL 2.1, work-groups are divided into sub-groups.

Work-items

Each work-group is divided into work-items.

Related information

7.5.2 Use concurrent data structures

OpenCL executes hundreds or thousands of individual kernel instances, so the processing and data structures must be parallelizable to that degree. This means you must use data structures that permit multiple data elements to be read and written simultaneously and independently. These are known as concurrent data structures.

Many common data structures are non-concurrent. This makes parallelizing the code difficult. For example, the following data structures are typically non-concurrent for writing data:

- Linked list.

- Hash table.

- Btree.

- Map.

This does not mean you cannot use these data structures. For example, these data structures can all be read in parallel without any issues.

Work-items can also write to these data structures, but you must be aware of the following restrictions:

- Work-items can access any data structure that is read-only.

- Work-items can write to any data structure if the work-items write to different elements.

- Work-items can write to the same element in any data structure if it is guaranteed that both work-items write the same value to the element.

  Alternatively, work-items can write different values to the same element in any data structure if it does not matter in the final output which of the values is correct. This is because either of the values might be the last to be written.

- Work-items cannot change the links in the data structure if they might impact other elements.

- Work-items can change the links in the data structure with atomic instructions if multiple atomic instructions do not access the same data.

There are parallel versions of many commonly used data structures.

7.5.3 Data division examples

You can process one-, two-, or three-dimensional data with OpenCL.

Note

The examples map the problems into the NDRanges that have the same number of dimensions. OpenCL does not require that you do this. You can for example, map a one-dimensional problem onto a two-, or three-dimensional NDRange.

7.5.3.1 One-dimensional data

An example of one-dimensional data is audio. Audio is represented as a series of samples. Changing the volume of the audio is a parallel task, because the operation is performed independently per sample.

In this case, the NDRange is the total number of samples in the audio. Each work-item can be one sample and a work-group is a collection of samples.

Audio can also be processed with vectors. If your audio samples are 16-bit, you can make a work-item represent eight samples and process eight of them at a time with vector instructions.

7.5.3.2 Two-dimensional data

An image is a natural fit for OpenCL, because you can process a 1,600 by 1,200 pixel image by mapping it onto a two-dimensional NDRange of 1,600 by 1,200. The total number of work-items is the total number of pixels in the image, that is, 1,920,000.

The NDRange is divided into work-groups where each work-group is also a two-dimensional array. The number of work-groups must divide into the NDRange exactly.

If each work-item processes a single pixel, a work-group size of 8 by 16 has the size of 128. This work-group size fits exactly into the NDRange on both the x and y axis. To process the image, you require 15,000 work-groups of 128 work-items each.

You can vectorize this example by processing all the color channels in a single vector. If the channels are 8-bit values, you can process multiple pixels in a single vector. If each vector processes four pixels, this means each work-item processes four pixels and you require four times fewer work-items to process the entire image. This means that your NDRange can be reduced to 400 by 1,200 and you only require 3,750 work-groups to process the image.

7.5.3.3 Three-dimensional data

You can use three-dimensional data to model the behavior of materials in the real world. For example, you can model the behavior of concrete for building by simulating the stresses in a three-dimensional data set.

You can use the data produced to determine the size and design of the structure you require to hold a specific load.

You can use this technique in games to model the physics of objects. When an object is broken, the physics simulation makes the process of breaking more realistic.

8. Retuning existing OpenCL code

This chapter describes how to retune existing OpenCL code so you can run it on Mali™ GPUs.

8.1 About retuning existing OpenCL code for Mali™ GPUs

OpenCL is a portable language but it is not always performance portable. This means that OpenCL applications can work on many different types of compute device but performance is not preserved. Existing OpenCL is typically tuned for specific architectures, such as desktop GPUs.

To achieve better performance with OpenCL code for Mali™ GPUs, you must retune the code:

1. Analyze the code.

2. Locate and remove optimizations for alternative compute devices.

3. Optimize the OpenCL code for Mali™ GPUs.

For the best performance, write kernels optimized for the specific target device.

Note You are not required to vectorize code for Mali™ Bifrost or Valhall GPUs. Vectorizing the code can improve performance though, depending on factors such as the content and the device. But, if the register pressure causes spilling to occur, performance is likely to reduce.

8.2 Differences between desktop-based architectures and Mali™ GPUs

There are some differences between desktop-based GPUs and Mali™ GPUs. Because of these differences, you must program the OpenCL in a different way for Mali™ GPUs.

8.2.1 About desktop-based GPU architectures

The power availability and large chip area of desktop GPUs enable them to have characteristics different to mobile GPUs.

Desktop GPUs have:

* A large chip area

* A large number of shader cores

* High-bandwidth memories.

The large power budget of desktop GPUs enables them to have these features.

Memory on desktop GPUs is organized in a hierarchy. Data is loaded from main memory into local memories. The local memories are organized in banks that are split, so there is one per thread in the thread group. Threads can access banks reserved for other threads, but when this happens accesses are serialized, reducing performance.

8.2.2 About Mali™ GPU architectures

Mali™ GPUs use an architecture in which instructions operate on multiple data elements simultaneously.

The peak throughput depends on the hardware implementation of the Mali™ GPU type and configuration.

Mali™ GPUs can contain many identical shader cores. Each shader core supports hundreds of concurrently executing threads.

OpenCL typically only uses the arithmetic unit or execution engines and the load-store unit. The texture unit is only used for reading image data types.

In the execution engines in Mali™ Bifrost and Valhall GPUs, scalar instructions are executed in parallel so the GPU operates on multiple data elements simultaneously. You are not required to vectorize your code to do this.

8.2.3 Programming OpenCL for Mali™ GPUs

There are differences between programming OpenCL on a Mali™ GPU and a desktop GPU.

If you are targeting Mali™ GPUs, the global and local OpenCL address spaces are mapped to the same physical memory and are accelerated by L1 and L2 caches.

Related information

8.3 Retuning existing OpenCL code for Mali™ GPUs

You can optimize existing OpenCL code for Mali™ GPUs if you analyze existing code and remove the device-specific optimizations.

8.3.1 Analyze code

If you did not write the code yourself, you must analyze it to see what it does.

Try to understand the following:

- The purpose of the code.
- The way the algorithm works.

- The way the code would look like if there were no optimizations.

This analysis can act as a guide to help you remove the device-specific optimizations.

This analysis can be difficult because highly optimized code can be very complex.

8.3.2 Locate and remove device optimizations

There are optimizations for alternative compute devices that have no effect on Mali™ GPUs, or can reduce performance. To retune the OpenCL code for Mali™ GPUs, you must first remove all types of optimizations to create a non device-specific *reference implementation*.

8.3.2.1 Optimizations to remove for Mali™ Bifrost and Valhall GPUs

Remove the following types of optimizations if you are targeting Mali™ Bifrost and Valhall GPUs:

Use of local or private memory

Mali™ GPUs use caches instead of local memories. The OpenCL local and private memories are mapped into main memory. There is therefore no performance advantage using local or private memories in OpenCL code for Mali™ GPUs.

You can use local or private memories as temporary storage, but memory copies to or from the memories are an expensive operation. Using local or private memories can reduce performance in OpenCL on Mali™ GPUs.

Do not use local or private memories as a cache because this can reduce performance. The processors already contain hardware caches that perform the same job without the overhead of expensive copy operations.

Some code copies data into a local or private memory, processes it, then writes it out again. This code wastes both performance and power by performing these copies.

Barriers

Data transfers to or from local or private memories are typically synchronized with barriers. If you remove copy operations to or from these memories, also remove the associated barriers.

Cache size optimizations

Some code optimizes reads and writes to ensure data fits into cache lines. This is a useful optimization for both increasing performance and reducing power consumption. However, the code is likely to be optimized for cache line sizes that are different than those used by Mali™ GPUs.

If the code is optimized for the wrong cache line size, there might be unnecessary cache flushes and this can decrease performance.

> **Note**
>
> Mali™ GPUs have a cache line size of 64-bytes.

8.3.3 Optimize your OpenCL code for Mali™ GPUs

When you have retuned the code, performance improves. To improve performance more, you must optimize it.

Related information

9. Optimizing OpenCL for Mali™ GPUs

This chapter describes the procedure to optimize OpenCL applications for Mali™ GPUs.

9.1 The optimization process for OpenCL applications

To optimize your application, you must first identify the most computationally intensive parts of your application. In an OpenCL application that means identifying the kernels that take the most time.

To identify the most computationally intensive kernels, you must individually measure the time taken by each kernel:

Measure individual kernels

Go through your kernels one at a time and:

1. Measure the time it takes for several runs.

2. Average the results.

Note

It is important that you measure the runtime of the individual kernels to get accurate measurements.

Do a dummy run of the kernel the first time to ensure that the memory is allocated. Ensure this is outside of your timing loop.

The allocation of some buffers in certain cases is delayed until the first time they are used. This can cause the first kernel run to be slower than subsequent runs.

Select the kernels that take the most time

Select the kernels that have the longest runtime and optimize these. Optimizing any other kernels has little impact on overall performance.

Analyze the kernels

Analyze the kernels to see if they contain computationally expensive operations:

• Measure how many reads and writes there are in the kernel. For high performance, do as many computations per memory access as possible.

• For Mali™ GPUs, you can use the Offline Shader Compiler to check the balancing between the different units.

Measure individual parts of the kernel

If you cannot determine the compute intensive part of the kernel by analysis, you can isolate it by measuring different parts of the kernel individually.

You can do this by removing different code blocks and measuring the performance difference each time.

The section of code that takes the most time is the most intensive.

Apply optimizations

Consider how the most intensive section of code can be rewritten and what optimizations apply.

Apply a relevant optimization.

Check your results

Whenever you make changes to optimize your code, ensure that you measure the results so you can determine the optimization was successful. Many changes that are beneficial in one situation, might not provide any benefit, or even reduce performance under a different set of conditions.

Reiterate the process

When you have increased the performance of your code with an optimization, measure it again to find out if there are other areas you can improve performance. There are typically several areas where you can improve performance so you might need to iterate the process many times to achieve optimal performance.

9.2 Load balancing between control threads and OpenCL threads

If you can, ensure that both control threads and OpenCL threads run in parallel.

9.2.1 Do not use clFinish() for synchronization

Sometimes the application processor must access data written by OpenCL. This process must be synchronized.

You can perform the synchronization with clFinish() but Arm recommends you avoid this if possible because it serializes execution. Calls to clFinish() introduce delays because the control thread must wait until all of the jobs in the queue to complete execution. The control thread is idle while it is waiting for this process to complete.

Instead, where possible, use clWaitForEvents() or callbacks to ensure that the control thread and OpenCL can work in parallel.

9.2.2 Do not use any of the clEnqueueMap() operations with a blocking call

Use clWaitForEvents() or callbacks to ensure that the control thread and OpenCL can work in parallel.

Procedure

1. Split work into many parts.

2. For each part:
    a) Prepare the work for part x on the application processor.
    b) Submit part x OpenCL work-items to the OpenCL device.

3. For each part:
    a) Wait for part x OpenCL work-items to complete on the OpenCL device using `clWaitForEvents`.
    b) Process the results from the OpenCL device on the application processor.

9.3 Optimizing memory allocation

You can optimize memory allocation by using the correct commands.

9.3.1 About memory allocation

To avoid making the copies, use the OpenCL API to allocate memory buffers and use map and unmap operations. These operations enable both the application processor and the Mali™ GPU to access the data without any copies.

OpenCL originated in desktop systems where the application processor and the GPU have separate memories. To use OpenCL in these systems, you must allocate buffers to copy data to and from the separate memories.

Systems with Mali™ GPUs typically have a shared memory, so you are not required to copy data. However, OpenCL assumes that the memories are separate and buffer allocation involves memory copies. This is wasteful because copies take time and consume power.

The following table shows the different `cl_mem_flags` parameters in `clCreateBuffer()`.

Table 9-1: Parameters for clCreateBuffer()

Parameter	Description
`CL_MEM_ALLOC_HOST_PTR`	This is a hint to the driver indicating that the buffer is accessed on the host side. To use the buffer on the application processor side, you must map this buffer and write the data into it. This is the only method that does not involve copying data. If you must fill in an image that is processed by the GPU, this is the best way to avoid a copy.
`CL_MEM_COPY_HOST_PTR`	Copies the data pointed to by the `host_ptr` argument into memory allocated by the driver.

Parameter	Description
CL_MEM_USE_HOST_PTR	Copies the data pointed to by the host memory pointer into the buffer when the first kernel using this buffer starts running. This flag enforces memory restrictions that can reduce performance. Avoid using this if possible. When a map is executed, the memory must be copied back to the provided host pointer. This significantly increases the cost of map operations.

Arm recommends the following:

- Do not use private or local memory to improve memory read performance.

- If your kernel is memory bandwidth bound, try using a simple formula to compute variables instead of reading from memory. This saves memory bandwidth and might be faster.

- If your kernel is compute bound, try reading from memory instead of computing variables. This saves computations and might be faster.

-

> **Note**
>
> If you are using a Mali™ Bifrost or Valhall GPU in a fully coherent system, use fine-grain shared virtual memory. See F.8 Shared virtual memory on page 102.

9.3.2 Use CL_MEM_ALLOC_HOST_PTR to avoid copying memory

The Mali™ GPU can access the memory buffers created by clCreateBuffer(CL_MEM_ALLOC_HOST_PTR). This is the preferred method to allocate buffers because data copies are not required.

This method of allocating buffers is shown in the following figure.

Figure 9-1: Memory buffer created by clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)
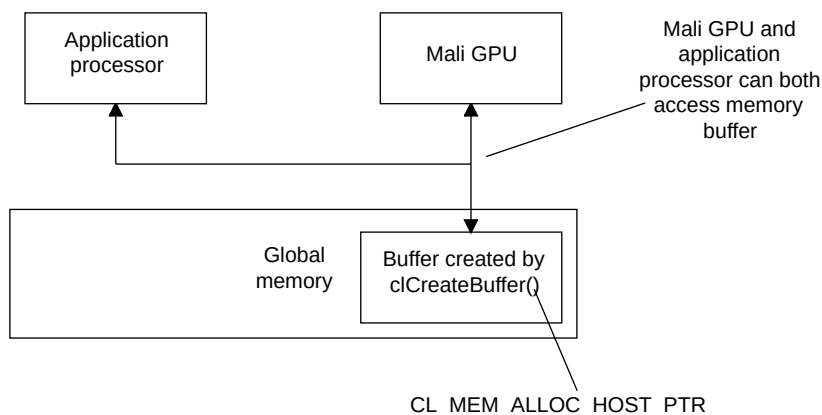
Arm recommends the following:

- You must make the initial memory allocation through the OpenCL API.

- Always use the latest pointer returned.

  If a buffer is repeatedly mapped and unmapped, the address the buffer maps into, is not guaranteed to be the same.

- > **Note**
  >
  > If you are using a Mali™ Bifrost or Valhall GPU in a fully coherent system, use fine-grain shared virtual memory. See F.8 Shared virtual memory on page 102.

9.3.3 Do not create buffers with CL_MEM_USE_HOST_PTR if possible

When a memory buffer is created using `clCreateBuffer(CL_MEM_USE_HOST_PTR)`, the driver might be required to copy the data to a separate buffer. This copy enables a kernel running on the GPU to access it. If the kernel modifies the buffer and the application maps the buffer so that it can be read, the driver copies the updated data back to the original location. The driver uses the application processor to perform these copy operations, that are computationally expensive.

This method of allocating buffers is shown in the following figure.

Figure 9-2: Memory buffer created by clCreateBuffer(CL_MEM_USE_HOST_PTR)
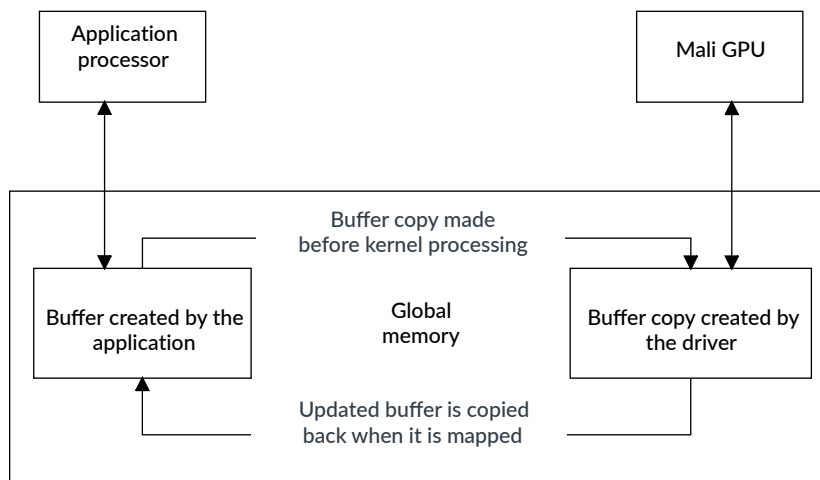
If your application can use an alternative allocation type, it can avoid these computationally expensive copy operations. For example, `CL_MEM_ALLOC_HOST_PTR`.

9.3.4 Sharing memory between I/O devices and OpenCL

For an I/O device to share memory with OpenCL, you must allocate the memory in OpenCL with CL_MEM_ALLOC_HOST_PTR.

You must allocate the memory in OpenCL with `CL_MEM_ALLOC_HOST_PTR` because it ensures that the memory pages are always mapped into physical memory.

If you allocate the memory on the application processor, the OS might not allocate physical memory to the pages until they are used for the first time. Errors occur if an I/O device attempts to use unmapped pages.

9.3.5 Sharing memory in a fully coherent system

Systems with full system coherency enable application processors and GPUs to share data easily, increasing performance.

With full system coherency, application processors and GPUs can access memory without requiring cache clean or invalidate operations on memory objects. This provides better performance than an I/O coherent system when the data is shared between application processor and GPU.

Fully coherent systems with Mali™ Bifrost or Valhall GPUs support fine-grained shared virtual memory in OpenCL 2.0 or later. See F.8 Shared virtual memory on page 102.

9.3.6 Sharing memory in an I/O coherent system

With I/O coherent allocation, the driver is not required to perform cache clean or invalidate operations on memory objects, before or after they are used on the Mali™ GPU. If you are using a memory object on both the application processor and the Mali™ GPU, this can improve performance.

If your platform is I/O coherent, you can enable I/O coherent memory allocation by passing the `CL_MEM_ALLOC_HOST_PTR` flag to `clCreateBuffer()` or `clCreateImage()`.

If you are using OpenCL 2.0 or later and your platform is I/O coherent, use shared virtual memory. See F.8 Shared virtual memory on page 102.

10. OpenCL optimizations list

This chapter lists several optimizations to use when writing OpenCL code for Mali™ GPUs.

10.1 General optimizations

Arm® recommends general optimizations such as processing large amount of data, using the correct data types, and compiling the kernels once.

Use the best processor for the job

GPUs are designed for parallel processing.

Application processors are designed for high-speed serial computations.

All applications contain sections that perform control functions and others that perform computation. For optimal performance use the best processor for the task:

- Control and serial functions are best performed on an application processor using a traditional language.
- Use OpenCL on Mali™ GPUs for the parallelizable compute functions.

Compile the kernel once at the start of your application

Ensure that you compile the kernel once at the start of your application. This can reduce the fixed overhead significantly.

Enqueue many work-items

To get maximum use of all your processor or shader cores, you must enqueue many work-items.

Process large amounts of data

You must process a relatively large amount of data to get the benefit of OpenCL. This is because of the fixed overheads of starting OpenCL tasks. The exact size of a data set where you start to see benefits depends on the processors you are running your OpenCL code on.

For example, performing simple image processing on a single 640x480 image is unlikely to be faster on a GPU, whereas processing a 1920x1080 image is more likely to be beneficial. Trying to benchmark a GPU with small images is only likely to measure the start-up time of the driver.

Do not extrapolate these results to estimate the performance of processing a larger data set. Run the benchmark on a representative size of data for your application.

Align data on 128-bit or 16-byte boundaries

Align data on 128-bit or 16-byte boundaries. This can improve the speed of loading and saving data. If you can, align data on 64-byte boundaries. This ensures data fits evenly into the cache on Mali™ GPUs.

Use the correct data types

Check each variable to see what range it requires.

Using smaller data types has several advantages:

- More operations can be performed per cycle with smaller variables.

- You can load or store more in a single cycle.

- If you store your data in smaller containers, it is more cacheable.

If accuracy is not critical, instead of an `int`, see if a `short`, `ushort`, or `char` works in its place.

For example, if you add two relatively small numbers you probably do not require an `int`. However, check in case an overflow might occur.

Only use `float` values if you require their additional range. For example, if you require very small or very large numbers.

Use the right data types

You can store image and other data as images or as buffers:

- If your algorithm can be vectorized, use buffers.

- If your algorithm requires interpolation or automatic edge clamping, use images.

Do not merge buffers as an optimization

Merging multiple buffers into a single buffer as an optimization is unlikely to provide a performance benefit.

For example, if you have two input buffers you can merge them into a single buffer and use offsets to compute addresses of data. However, this means that every kernel must perform the offset calculations.

It is better to use two buffers and pass the addresses to the kernel as a pair of kernel arguments.

Use asynchronous operations

If possible, use asynchronous operations between the control threads and OpenCL threads. For example:

- Do not make the application processor wait for results.

- Ensure that the application processor has other operations to process before it requires results from the OpenCL thread.

- Ensure that the application processor does not interact with OpenCL kernels when they are executing.

Avoid application processor and GPU interactions in the middle of processing

Enqueue all the kernels first, and call `clFinish()` at the end if possible.

Call `clFlush()` after one or more `clEnqueueNDRange()` calls, and call `clFinish()` before checking the final result.

Avoid blocking calls in the submission thread

Avoid `clFinish()` or `clWaitForEvent()` or any other blocking calls in the submission thread.

If possible, wait for an asynchronous callback if you want to check the result while computations are in progress.

Try double buffering, if you are using blocking operations in your submission thread.

Batching kernels submission

From version r17p0 onwards, the OpenCL driver batches kernels that are flushed together for submission to the hardware. Batching kernels can significantly reduce the runtime overheads and cache maintenance costs. For example, this reduction is useful when the application is accessing multiple sub-buffers created from a buffer imported using `clImportMemoryARM` in separate kernels.

The application should flush kernels in groups as large as possible. When the GPU is idle though, reaching optimal performance requires the application to flush an initial batch of kernels early so that the GPU execution overlaps the queuing of further kernels.

Related information

Converting existing code to OpenCL on page 40

10.2 Kernel optimizations

Arm recommends some kernel optimizations such as experimenting with the work-group size and shape, minimizing thread convergence.

Query the possible workgroup sizes that can be used to execute a kernel on the device

For example:

```
clGetKernelWorkgroupInfo(kernel, dev, CL_KERNEL_WORK_GROUP_SIZE,
  sizeof(size_t)... );
```

In typical applications, the workgroup size should be at least as large as the warp size and ideally a multiple of the warp size

If you are using a barrier, a smaller workgroup size is better.

When you are selecting a workgroup size, consider the memory access pattern of the data.

Finding the best workgroup size can be counter-intuitive, so test different options to see what one is fastest.

If you are not sure what workgroup size is best, define local_work_size as NULL

The driver picks the workgroup size it thinks as best.

Note
The performance might not be optimal.

If you want to set the local work size, set the reqd_work_group_size qualifier to kernel functions

This provides the driver with information at compile time for register use and sizing jobs to fit properly on shader cores.

Experiment with work-group size

If you can, experiment with different sizes to see if any give a performance advantage. Sizes that are a multiple of two are more likely to perform better.

If your kernel has no preference for the work-group size, you can pass `NULL` to the local work size argument of the `clEnqueueNDRangeKernel()`.

Experiment with work-group shape

The shape of the work-group can affect the performance of your application. For example, a 32 by 4 work-group might be the optimal size and shape.

Experiment with different shapes and sizes to find the best combination for your application.

Check for synchronization requirements

Some kernels require work-groups for synchronization of the work-items within the work-group with barriers. These typically require a specific work-group size.

In cases where synchronization between work-items is not required, the choice of the size of the work-groups depends on the most efficient size for the device.

You can pass in `NULL` to enable OpenCL to pick an efficient size.

Consider combining multiple kernels

If you have multiple kernels that work in a sequence, consider combining them into a single kernel. If you combine kernels, be careful of dependencies between them.

However, do not combine the kernels if there are widening data dependencies.

For example:

- If there are two kernels, A and B.

- Kernel B takes an input produced by kernel A.

- If kernel A is merged with kernel B to form kernel C, you can only input to kernel C constant data, plus data from what was previously input to kernel A.

- Kernel C cannot use the output from kernel A n-1, because it is not guaranteed that kernel A n-1 has been executed. This is because the order of execution of work-items is not guaranteed.

Typically this means that the coordinate systems for kernel A and kernel B are the same.

Note

If combining kernels requires a barrier, it is probably better to keep them separate.

Avoid splitting kernels

Avoid splitting kernels. If you are required to split a kernel, split it into as few kernels as possible.

> **Note**
> - Splitting a kernel can sometimes be beneficial if it enables you to remove a barrier.
> - Splitting a kernel can be useful if your kernel suffers from register pressure.

Check if your kernels are small

If your kernels are small, use data with a single dimension and ensure the work-group size is a power of two.

Use a sufficient number of concurrent threads

Use a sufficient number of concurrent threads to hide the execution latency of instructions.

The number of concurrent threads that the shader core executes depends on the number of active registers your kernel uses. The higher the number of registers, the smaller the number of concurrent threads.

The number of registers used is determined by the compiler based on the complexity of the kernel, and how many live variables the kernel has at one time.

To reduce the number of registers:

- Try reducing the number of live variables in your kernel.
- Use a large NDRange, so there are many work-items.

Experiment with this to find what suits your application. You can use the Offline Complier to produce statistics for your kernels to assist with this.

Optimize the memory access pattern of your application

Use data structures with linear access and high locality. These improve cacheability and therefore performance.

10.3 Code optimizations

Arm® recommends some code optimizations such as using built-in functions or experimenting with your data to increase algorithm performance.

Use vector loads and saves

To load as much data as possible in a single operation, use vector loads. These enable you to load 128 bits at a time. Do the same for saving data.

For example, if you are loading char values, use the built-in function `vload16()` to load 16 bytes at a time.

Do not try to load more than 128 bits in a single load. This can reduce performance.

Perform as many operations per load as possible

Operations that perform multiple computations per element of data loaded are typically good for programming in OpenCL:

- Try to reuse data already loaded.

- Use as many arithmetic instructions as possible per load.

Avoid conversions to or from float and int

Conversions to or from `float` and `int` are relatively expensive so avoid them if possible.

Experiment to see how fast you can get your algorithm to execute

There are many variables that determine how well an application performs. Some of the interactions between variables can be very complex and it is difficult to predict how they impact performance.

Experiment with your OpenCL kernels to see how fast they can run:

Data types

Use the smallest data types for your calculation as possible.

For example, if your data does not exceed 16 bits do not use 32-bit types.

Load store types

Try changing the amount of data processed per work-item.

Data arrangement

Change the data arrangement to make maximum use of the processor caches.

Maximize data loaded

Always load as much data in a single operation as possible. Use 128-bit wide vector loads to load as many data items as possible per load.

Use shift instead of a divide

If you are dividing by a power of two, use a shift instead of a divide.

Note
- This only applies to integers.

- This only works for powers of two.

- Divide and shift use different methods of rounding negative numbers.

Use vector loads and saves for scalar data

Use vector load `VLOAD` instructions on arrays of data even if you do not process the data as vectors. This enables you to load multiple data elements with a single instruction. A vector load of 128-bits takes the same amount of time as loading a single character. Multiple loads of single characters are likely to cause cache thrashing and this reduces performance. Do the same for saving data.

Use _sat() functions instead of min() or max()

`_sat()` functions automatically take the maximum or minimum values if the values are too high or too low for representation. You are not required to add additional `min()` or `max()` code.

Avoid writing kernels that use many live variables

Using too many live variables can affect performance and limit the number of concurrently executing threads per core.

Do not calculate constants in kernels

- Use defines for constants.

- If the values are only known at runtime, calculate them in the host application and pass them as arguments to the kernel.

  For example, `height-1`.

Use the offline compiler to produce statistics

Use the offline compiler to produce statistics for your kernels and check the ratio between arithmetic instructions and loads.

Note
For more information about the offline compiler, see https://developer.arm.com/tools-and-software/graphics-and-gaming/graphics-development-tools/mali-offline-compiler/docs/101086/latest/usage/opencl.

Use the built-in functions

Many of the built-in functions are implemented as fast hardware instructions, use these for high performance.

Use the cache carefully

- The amount of cache space available per thread is low so you must use it with care.

- Use the minimum data size possible.

- Use data access patterns to maximize spatial locality.

- Use data access patterns to maximize temporal locality.

Use large sequential reads and writes

General Purpose computations on a GPU can make very heavy use of external memory. Using large sequential reads and writes significantly improves memory performance.

Related information

Kernel auto-vectorizer and unroller on page 73
OpenCL built-in functions on page 80
half_ and native_ math functions on page 80

10.4 Execution optimizations

Arm recommends some execution optimizations such as optimizing communication code to reduce latency.

Arm recommends that:

- If you are building from source, cache binaries on the storage device.

- If your application contains excessive kernels that may not be invoked, you can reduce the compiling time by deferring the actual compiled kernels until the `clCreateKernel()` function is called. Use these options to configure OpenCL runtime compiler:

    fdeferred-compilation

    Defers kernel compilation until `clCreateKernel()` is called.

    fno-deferred-compilation

    Executes all kernels compiled at `clBuildProgram()` or `clLinkProgram()` functions.

    It is the default compiler option. The compiler spawns multiple threads to compile kernels in parallel. The compiler uses the `nproc` variable to spawn a thread per processor, assuming enough kernels need compiling.

    Use the following option to control the maximum number of threads spawned.

    ```
    -j <number>
    ```

- If you use callbacks to prompt the processor to continue processing data resulting from a kernel execution, ensure the callbacks being set before you flush the queue. Otherwise, the callbacks might occur a larger batch of work, later than the callbacks might have completed the actual work.

10.5 Reducing the effect of serial computations

You can reduce the impact of serial components in your application by reducing and optimizing the computations.

Use memory mapping instead of memory copies to transfer data.

Optimize communication code.

To reduce latency, optimize the communication code that sends and receives data.

Keep messages small.

Reduce communication overhead by sending only the data that is required.

Use power of two sized memory blocks for communication.

Ensure the sizes of memory blocks used for communication are a power of two. This makes the data more cacheable.

Send more data in a smaller number of transfers.

Compute values instead of reading them from memory.

A simple computation is likely to be faster than reading from memory.

Do serial computations on the application processors.

> Application processors are optimized for low latency tasks.

Use clEnqueueFillBuffer() to fill buffers.

> The Mali™ OpenCL driver contains an optimized implementation of `clEnqueueFillBuffer()`. Use in place of manually implementing a buffer fill in your application.

Use clEnqueueFillImage() to fill images.

> The Mali™ OpenCL driver contains an optimized implementation of `clEnqueueFillImage()`. Use this in place of manually implementing an image fill in your application.

10.6 Mali™ Bifrost and Valhall GPU specific optimizations

Arm recommends some Mali™ Bifrost and Valhall GPU specific optimizations.

> **Note**
>
> Only use these optimizations if you are specifically targeting a Mali™ Bifrost or Valhall GPU.

Ensure that the threads all take the same branch direction in if-statements and loops

> In Mali™ Bifrost and Valhall GPUs, groups of adjacent threads are arranged together. Scalar instructions in adjacent threads are executed in parallel so the GPU operates on multiple data elements simultaneously. The scalars execute in threads and these must operate in lock-step. If your shaders contain branches, such as if statements or loops, the branches in adjacent threads can go different ways. The arithmetic unit cannot execute both sides of the branch at the same time. The two operations are split and the processing speed is reduced. To avoid this performance reduction, try to ensure that adjacent threads all branch the same way.
>
> The following table shows the number of adjacent threads for each Mali™ Bifrost and Valhall GPU.

Table 10-1: Adjacent threads

GPU	Adjacent threads
Mali™-G31	4
Mali™-G51	4
Mali™-G52	8
Mali™-G57	16
Mali™-G71	4
Mali™-G72	4
Mali™-G76	8
Mali™-G77	16
Mali™-G78	16

Avoid excessive register usage

Every thread has 64 32-bit working registers. A 64-bit variable uses two adjacent 32-bit registers for its 64-bit data.

If a thread requires more than 64 registers, the compiler might start storing register data in memory. This reduces performance and the available bandwidth. This is especially bad if your shader is already load-store bound.

Vectorize 8-bit and 16-bit operations

For 16-bit operations, use 2-component vectors to get full performance. For basic arithmetic operations, fp16 version is twice as fast as fp32 version.

For 8-bit types, such as `char`, use four-component vectors for best performance.

Do not vectorize 32-bit operations

Mali™ Bifrost and Valhall GPUs use scalars so you are not required to vectorize 32-bit operations. 32-bit scalar and vector arithmetic operations have same performance.

Use 128-bit load or store operations

128-bit load or store operations make the more efficient use of the internal buses.

Load and store operations are faster if all threads in a quad load from the same cache-line

If all threads in a quad load from the same cache-line, the arithmetic unit only sends one request to the load-store unit to load the 512-bit data.

For example, this example is fast because consecutive threads load consecutive 128-bit vectors from memory:

```
global float4 * input_array;
float4 v = input_array[get_global_id(0)];
```

This second version is slower, because the four threads with adjacent global `ids` load data from different cache lines.

```
global float4 * input_array;
float4 v = input_array[4*get_global_id(0)];
```

One cache line is 512-bits.

Note

Use 32-bit arithmetic in place of 64-bit if possible

64-bit arithmetic operates at half the speed of 32-bit arithmetic.

Use fine-grained shared virtual memory

If your system supports it, using the shared virtual memory feature in OpenCL 2.0 provides cache-coherent memory. This reduces the requirement for manually synchronizing memories and increases performance. See F.8 Shared virtual memory on page 102.

Try to get a good balance of usage of the execution engines and load-store units

If one unit is overused, this can limit the overall performance of the application the GPU is executing. For example, the load-store unit is overused, try computing values rather than

loading them. If the execution engine is overused, try loading values instead of computing them.

11. Kernel auto-vectorizer and unroller

This chapter describes the kernel auto-vectorizer and unroller.

11.1 About the kernel auto-vectorizer and unroller

The OpenCL compiler includes a kernel auto-vectorizer and a kernel unroller. You must manually enable these features.

The kernel auto-vectorizer takes existing code and transforms it into vector code.

The unroller merges work-items by unrolling the bodies of the kernels.

If these operations are possible, they can provide substantial performance gains.

For Bifrost and Valhall GPUs, you manually enable these features by passing the kernel transformations command-line options to the compiler, see:

- 11.2.1 Kernel auto-vectorizer command and parameters on page 74.
- 11.3.1 Kernel unroller command and parameters on page 75.
- 11.4 The dimension interchange transformation on page 75.

There are several options to control the auto-vectorizer and unroller. The following table shows the basic options.

Table 11-1: Kernel auto-vectorizer and unroller options

Option	Description
no option	Kernel unroller and vectorizer enabled, with conservative heuristics.
`-fno-kernel-vectorizer`	Disable the kernel vectorizer.
`-fno-kernel-unroller`	Disable the kernel unroller.
`-fkernel-vectorizer`	Enable aggressive heuristics for the kernel vectorizer.
`-fkernel-unroller`	Enable aggressive heuristics for the kernel unroller.

Note

The kernel auto-vectorizer performs a code transformation. For the transformation to be possible, several conditions must be met:

- The enqueued NDRange must be a multiple of the vectorization factor.
- Barriers are not permitted in the kernel.
- Thread-divergent code is not permitted in the kernel.
- Global offsets are not permitted in the enqueued NDRange.

11.2 Kernel auto-vectorizer options

You can optionally use the dimension and factor parameters to control the behavior of the auto-vectorizer.

11.2.1 Kernel auto-vectorizer command and parameters

The format of the kernel auto-vectorizer command is:

```
-fkernel-vectorizer= <dimension><factor>
```

The parameters are:

dimension   This selects the dimension along which to vectorize.

factor   This is the number of neighboring work-items that are merged to vectorize.

This must be one of the values 2, 4, 8, or 16. Other values are invalid.

The vectorizer works by merging consecutive work-items. The number of work-items enqueued is reduced by the vectorization factor.

For example, in a one-dimensional NDRange, work-items have the local-IDs 0, 1, 2, 3, 4, 5...

Vectorizing by a factor of four merges work-items in groups of four. First work-items 0, 1, 2, and 3, then work-items 4, 5, 6, and 7 going upwards in groups of four until the end of the NDRange.

In a two-dimensional NDRange, the work-items have local-IDs such as (0,0), (0,1), (0,2)..., (1,0), (1,1), (1,2)... where (x,y) is showing (global_id(0), global_id(1)).

The vectorizer can vectorize along dimension 0 and merge work-items (0,0), (1,0)...

Alternatively it can vectorize along dimension 1 and merge work-items (0,0), (0,1)...

11.2.2 Kernel auto-vectorizer command examples

Examples of auto-vectorizer commands.

The following table shows examples of auto-vectorizer commands.

Table 11-2: Kernel auto-vectorizer command examples

Example	Description
`-fkernel-vectorizer`	Enable the vectorizer, use heuristics for both dimension and factor.
`-fkernel-vectorizer=x4`	Enable the vectorizer, use dimension 0, use factor 4.
`-fkernel-vectorizer=z2`	Enable the vectorizer, use dimension 2, use factor 2.
`-fkernel-vectorizer=x`	Enable the vectorizer, use heuristics for the factor, use dimension 0.
`-fkernel-vectorizer=2`	Enable the vectorizer, use heuristics for the dimension, use factor 2.

11.3 Kernel unroller options

You can optionally use additional parameters to control the behavior of the kernel unroller.

11.3.1 Kernel unroller command and parameters

The format of the kernel unroller command is:

```
-fkernel-unroller= <dimension><factor>
```

The parameters are:

dimension   This selects the dimension along which to unroll.
factor   This is the number of neighboring work-items that are merged.

The performance gain from unrolling depends on your kernel and the unrolling factor, so experiment to see what suits your kernel. It is typically best to keep the unroll factor at eight or below.

11.3.2 Kernel unroller command examples

Examples of kernel unroller commands.

The following table shows examples of kernel unroller commands.

Table 11-3: Kernel unroller command examples

Example	Description
`-fkernel-unroller`	Enable the unroller, use heuristics for both dimension and factor.
`-fkernel-unroller=x4`	Enable the unroller, use dimension 0, use factor 4.
`-fkernel-unroller=z2`	Enable the unroller, use dimension 2, use factor 2.
`-fkernel-unroller=x`	Enable the unroller, use heuristics for the factor, use dimension 0.
`-fkernel-unroller=2`	Enable the unroller, use heuristics for the dimension, use factor 2.

11.4 The dimension interchange transformation

The dimension interchange transformation swaps the dimensions of a work-group. This transformation can improve cache locality and improve performance.

Dimension interchange is applied to kernels with the following annotation:

- ```
  __attribute__ ((annotate("interchange")))
  ```

This interchanges dimensions `0` and `1`.

- ```
  __attribute__ ((annotate("interchange<dim0><dim1>")))
  ```

This interchanges dimensions dim0 and dim1, where `<dim0>` and `<dim1>` can be `0`, `1` or `2`.

You can disable dimension interchange with the following option:

```
-fno-dim-interchange
```

There are no parameters.

Appendix A OpenCL data types

This appendix describes OpenCL data types.

A.1 About OpenCL data types

This appendix lists the data types available in OpenCL. Most of these types are all natively supported by the Mali™ GPU hardware.

The OpenCL types are used in OpenCL C. The API types are equivalents for use in your application. Use these to ensure that the correct data is used, and it is aligned on 128-bit or 16-byte boundaries.

Up to 32-bits per chunk can work as vectors on Mali™ Bifrost and Valhall GPUs. This means you can use `char`, `short`, and `half` in vectors.

Converting between vector types has a low performance cost on Mali™ GPUs. For example, converting a vector of 8-bit values to 16-bit values:

```
ushort8 a;  uchar8 b;
a = convert_ushort8(b);
```

A.2 OpenCL data type lists

List of OpenCL data types organized by type.

A.2.1 Built-in scalar data types

List of built-in scalar data types.

Table A-1: Built-in scalar data types

Types for OpenCL kernels	Types for application	Description
`bool`	-	true (1) or false (0)
`char`	`cl_char`	8-bit signed
`unsigned char, uchar`	`cl_uchar`	8-bit unsigned
`short`	`cl_short`	16-bit signed
`unsigned short, ushort`	`cl_ushort`	16-bit unsigned
`int`	`cl_int`	32-bit signed
`unsigned int, uint`	`cl_uint`	32-bit unsigned
`long`	`cl_long`	64-bit signed
`unsigned long, ulong`	`cl_ulong`	64-bit unsigned

Types for OpenCL kernels	Types for application	Description
float	cl_float	32-bit float
half	cl_half	16-bit float
size_t	-	unsigned integer, with size matching CL_DEVICE_ADDRESS_BITS
ptrdiff_t	-	unsigned integer, with size matching CL_DEVICE_ADDRESS_BITS
intptr_t	-	signed integer, with size matching CL_DEVICE_ADDRESS_BITS
uintptr_t	-	unsigned integer, with size matching CL_DEVICE_ADDRESS_BITS
void	void	void

> **Note**
> You can query CL_DEVICE_ADDRESS_BITS with clGetDeviceInfo(). The value returned might be different for 32-bit and 64-bit host applications, even on the same Mali™ GPU.

A.2.2 Built-in vector data types

List of built-in vector data types where *n* = 2, 3, 4, 8, or 16.

Table A-2: Built-in vector data types

OpenCL Type	API type for application	Description
char*n*	cl_char*n*	8-bit signed
uchar*n*	cl_uchar*n*	8-bit unsigned
short*n*	cl_short*n*	16-bit signed
ushort*n*	cl_ushort*n*	16-bit unsigned
int*n*	cl_int*n*	32-bit signed
uint*n*	cl_uint*n*	32-bit unsigned
long*n*	cl_long*n*	64-bit signed
ulong*n*	cl_ulong*n*	64-bit unsigned
float*n*	cl_float*n*	32-bit float

A.2.3 Other built-in data types

List of other built-in data types.

Table A-3: Other built-in data types

OpenCL Type	Description
image2d_t	2D image handle
image3d_t	3D image handle
image2d_array_t	2D image array
image1d_t	1D image handle

OpenCL Type	Description
`image1d_buffer_t`	1D image created from buffer
`image1d_array_t`	1D image array
`sampler_t`	Sampler handle
`event_t`	Event handle

A.2.4 Reserved data types

List of reserved data types. Do not use these in your OpenCL kernel code.

Table A-4: Reserved data types

OpenCL Type	Description
`bool`n	Boolean vector.
`half`n	16-bit float, vector.
`quad`, `quad`n	128-bit float, scalar, and vector.
`complex half`, `complex half`n	Complex 16-bit float, scalar, and vector.
`imaginary half`, `imaginary half`n	Imaginary 16-bit complex, scalar, and vector.
`complex float`, `complex float`n,	Complex 32-bit float, scalar, and vector.
`imaginary float`, `imaginary float`n	Imaginary 32-bit float, scalar, and vector.
`complex double`, `complex double`n	Complex 64-bit float, scalar, and vector.
`imaginary double`, `imaginary double`n	Imaginary 64-bit float, scalar, and vector.
`complex quad`, `complex quad`n	Complex 128-bit float, scalar, and vector.
`imaginary quad`, `imaginary quad`n	Imaginary 128-bit float, scalar, and vector.
`float`nx`m	$n*m$ matrix of 32-bit floats.
`double`nx`m	$n*m$ matrix of 64-bit floats.
`long double`, `long double`n	64-bit - 128-bit float, scalar, and vector.
`long long`, `long long`nb	128-bit signed int, scalar, and vector.
`unsigned long long`, `ulong long`, `ulonglong`n	128-bit unsigned int, scalar, and vector.

> **Note**
>
> The `half` and `half` vector data types can be used with the `cl_khr_fp16` extension.

Appendix B OpenCL built-in functions

This appendix lists the OpenCL half_ and native_ math and the synchronization built-in functions.

B.1 half_ and native_ math functions

List of half_ and native_ math functions. The half_ and native_ variants of the math functions are provided for portability.

Table B-1: half_ and native_ math functions

half_ functions	native_ functions
`half_cos()`	`native_cos()`
`half_divide()`	`native_divide()`
`half_exp()`	`native_exp()`
`half_exp2()`	`native_exp2()`
`half_exp10()`	`native_exp10()`
`half_log()`	`native_log()`
`half_log2()`	`native_log2()`
`half_log10()`	`native_log10()`
`half_powr()`	`native_powr()`
`half_recip()`	`native_recip()`
`half_rsqrt()`	`native_rsqrt()`
`half_sin()`	`native_sin()`
`half_sqrt()`	`native_sqrt()`
`half_tan()`	`native_tan()`

Mali™ GPUs implement most of the full precision variants of the `half_` and `native_` math functions at full speed so you are not required to use the `half_` and `native_` functions.

Note

On Mali™ GPUs, the following functions are faster than the full precision versions:

- `native_sin()`.

- `native_cos()`.

- `native_tan()`.

- `native_divide()`.

- `native_exp()`.

- `native_sqrt()`.

- `half_sqrt()`.

B.2 Synchronization functions

List of synchronization functions.

The `barrier()` function has no speed rating because it must wait for multiple work-items to complete. The time determines the length of time the function takes in your application. This also depends on several factors such as:

- The number of work-items in the work-groups being synchronized.

- How much the work-items diverge.

Synchronization functions

Synchronization functions include:

- `barrier()`

- `mem_fence()`

- `read_mem_fence()`

- `write_mem_fence()`

Note

Arm® recommends that you avoid using barriers, especially in small kernels.

Appendix C OpenCL extensions

This appendix describes the OpenCL extensions that the Mali™ GPU OpenCL driver supports.

C.1 OpenCL extensions supported by the Mali™ GPU OpenCL driver

The Mali™ GPU OpenCL driver supports several extensions on Mali™ GPUs.

The supported extensions are the following:

- `cl_khr_byte_addressable_store`
- `cl_khr_create_command_queue`
- `cl_khr_egl_image`
- `cl_khr_extended_versioning`
- `cl_khr_fp16`
- `cl_khr_global_int32_base_atomics`
- `cl_khr_global_int32_extended_atomics`
- `cl_khr_icd`
- `cl_khr_image2d_from_buffer`
- `cl_khr_int64_base_atomics`
- `cl_khr_int64_extended_atomics`
- `cl_khr_local_int32_base_atomics`
- `cl_khr_local_int32_extended_atomics`
- `cl_khr_priority_hints`
- `cl_khr_3d_image_writes`
- `cl_khr_device_uuid`
- `cl_khr_semaphore`
- `cl_khr_external_semaphore`
- `cl_khr_external_semaphore_sync_fd`
- `cl_khr_integer_dot_product`
- `cl_khr_external_memory`
- `cl_khr_external_memory_dma_buf`

The Mali™ GPU OpenCL driver also supports the following optional Arm® extensions:

- `cl_arm_core_id`

- `cl_arm_controlled_kernel_termination`

- `cl_arm_import_memory`

- `cl_arm_import_memory_android_hardware_buffer`

- `cl_arm_import_memory_host`

- `cl_arm_import_memory_dma_buf`

- `cl_arm_import_memory_protected`

- `cl_arm_job_slot_selection` (on devices that support work submission via job slots)

- `cl_arm_non_uniform_work_group_size`

- `cl_arm_printf`

- `cl_arm_scheduling_controls`

- `cl_arm_protected_memory_allocation`

- `cl_ext_cxx_for_opencl`

- `cl_ext_yuv_images`

- `cl_ext_image_from_buffer`

- `cl_ext_image_requirements_info`

- `cl_ext_image_tiling_control`

The following extensions are only supported on Mali™ Bifrost and Valhall GPUs under OpenCL 2.0 or later:

- `cl_khr_depth_images`

- `cl_khr_il_program`

- `cl_khr_subgroups` (supported on all Valhall GPUs and Bifrost GPUs from Mali™-G76)

- `cl_khr_subgroup_extended_types` (supported on all Valhall GPUs and Bifrost GPUs from Mali™-G76)

The following extensions are only supported on GPUs that support the underlying ISA instructions:

- `cl_arm_integer_int8`

- `cl_arm_integer_dot_product_accumulate_int8`

- `cl_arm_integer_dot_product_accumulate_int16`

- `cl_arm_integer_dot_product_accumulate_saturate_int8` (supported on all Valhall GPUs from Mali™-G77)

Related information
http://www.khronos.org

Appendix D Using OpenCL extensions

This appendix provides usage notes on specific OpenCL extensions.

D.1 Inter-operation with EGL

The DDK supports the use of EGL images for sharing data between different Khronos APIs, such as OpenGL and OpenCL.

D.1.1 EGL images

The `EGL_KHR_image_base` EGL extension provides the basic mechanism for sharing EGL images.

The `EGL_KHR_image_base` EGL extension defines two entry points for creating and destroying EGL images:

```
EGLImageKHR eglCreateImageKHR(
                    EGLDisplay dpy,
                    EGLContext ctx,
                    EGLenum target,
                    EGLClientBuffer buffer,
                    const EGLint *attrib_list)

EGLBoolean eglDestroyImageKHR(
                    EGLDisplay dpy,
                    EGLImageKHR image)
```

The `eglCreateImageKHR` call returns an opaque handle that the Khronos APIs use for referencing EGL images. Nothing in the extension specification precludes an EGL image from being the storage for the content that is to be shared, but the actual role of EGL images in the DDK is to serve only as references to memory allocations made by a low level API. There are two low level APIs for memory allocation that the DDK actively supports:

- Gralloc on Android, which is covered in `ANDROID_image_native_buffer` subsection.

- `dma_buf` on Linux, covered in `EGL_EXT_image_dma_buf_import` subsection.

The availability of the memory allocation methods depends on the platform, the `EGL_KHR_image_base` extension relies on additional extensions to define platform-specific values for the `target` and `buffer` parameters on the `eglCreateImageKHR` call.

The `EGL_KHR_image_base` extension only states that the `target` parameter is a unique number identifying the content source. The `buffer` parameter is an extension-specific handle that is cast to `EGLClientBuffer`, a void pointer on the official Khronos EGL headers.

D.1.1.1 Preserving EGL images for portability

Applications needing the EGL image contents to be preserved must set `EGL_IMAGE_PRESERVED_KHR` to `EGL_TRUE` rather than relying on the default value.

The only attribute defined by the `EGL_KHR_image_base` extension is `EGL_IMAGE_PRESERVED_KHR`, which is a boolean defining whether the image contents are undefined after the `eglCreateImageKHR` call returns. The default value for `EGL_IMAGE_PRESERVED_KHR` is `EGL_FALSE`, meaning that undefined contents are acceptable unless the application explicitly sets the value to `EGL_TRUE`.

Failing to set `EGL_IMAGE_PRESERVED_KHR` to `EGL_TRUE` can lead to applications relying on undefined behavior, that is subject to change between Mali™ DDK releases.

D.1.1.2 OpenCL support for EGL images

There are two OpenCL mechanisms for supporting data sharing with other APIs from the Khronos ecosystem.

The two OpenCL mechanisms for supporting data sharing with other APIs from the Khronos ecosystem are:

cl_khr_egl_image

> Mali™ DDK supports this API for importing EGL images.

cl_khr_gl_sharing

> Mali™ DDK does not support this API for importing OpenGL or OpenGL ES objects such as vertex buffers, index buffers, textures, or render buffer objects. This API does not add any substantial benefit to the EGL image mechanism and is tightly coupled to the OpenGL semantics.

Because customers can implement all their relevant use cases with the `cl_khr_egl_image` path, cl_khr_gl_sharing extension is unsupported on the Mali™ DDK since the r6p0-01rel0 release.

The supported `cl_khr_egl_image` OpenCL extension defines an entry point for creating `cl_mem` objects out of EGL images:

```
cl_mem clCreateFromEGLImageKHR (
    cl_context context,
    CLeglDisplayKHR display,
    CLeglImageKHR image,
    cl_mem_flags flags,
    const cl_egl_image_properties_khr * properties,
    cl_int * errcode_ret)
```

The `image` parameter is an EGL image handle that is returned by `eglCreateImageKHR`. The flags are a subset of the flags that are accepted by `clCreateBuffer`, where the accepted flags are only `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE`. The list of `properties` is present on the extension for allowing future additions, but no such properties currently exist.

D.1.1.3 Synchronization when using the cl_khr_egl_image extension

Applications are responsible for flushing the work when one Khronos API consumes the output from another.

For example, if an application produces an OpenGL ES render that OpenCL has to consume, then it is the responsibility of the application to issue a `glFinish` before the OpenCL access happens.

The same application responsibility is also required when OpenCL outputs data that OpenGL is going to use. For example, an OpenGL ES rendering application that consumes some OpenCL output needs to ensure that it calls `clFinish` or `clWaitForEvents` to guarantee that the data is available for consumption before using OpenGL ES to access it.

In addition to flushing, there are GPU architectures that require the EGL images that are consumed or produced by OpenCL to be transferred between different device memories. That kind of architecture is common on discrete graphics cards for desktop PCs. The Mali™ implementations, on the other hand, uses the same main memory as the application processor, rather than a dedicated memory. Therefore there is no requirement to transfer data between the application processor and any dedicated GPU memory.

The OpenCL API delegates to the applications to choose the times at which the transfers happen by explicitly signaling when the accesses start and end.

The start and end of the EGL image accesses by OpenCL applications must be signaled by enqueuing `clEnqueueAcquireEGLObjectsKHR` and `clEnqueueReleaseEGLObjectsKHR` commands on an OpenCL command queue before and after a kernel accesses the EGL image data. An OpenCL kernel that uses an EGL image that it has not previously acquired gets an error at kernel enqueue time, for ensuring portability and compliance with the OpenCL standard, even on Mali™ implementations.

D.1.1.4 EGL images limitation

It is not possible to query an EGL image for its format or dimensions through the EGL API. The Mali™ OpenCL driver, on the other hand, allows querying for the format and dimensions of an OpenCL image created out of an EGL image.

However, the reported OpenCL format is only meaningful for formats that have a mapping between the EGL image source format and a format defined in the OpenCL specification.

A second limitation is that OpenCL images created out of EGL images cannot be memory mapped by means of `clEnqueueImageMap` and `clEnqueueUnmapMemObject`. The Khronos specifications do not explicitly say whether the memory mapping must be supported for this kind of OpenCL images. The DDK does not support memory mapping.

D.1.2 ANDROID_image_native_buffer

The Android graphics stack builds on top of the Gralloc memory allocation library. Processes can share the handles that reference the Gralloc allocations. This interprocess shareability is key for supporting common use cases such as a camera driver sharing data with a rendering application.

SurfaceFlinger, the window composition service, can run on its own process and consume data from rendering applications because the Gralloc allocations are accessible across processes, making Android a very modular system.

A Gralloc allocation consists only of a chunk of memory and lacks any internal state describing what is stored on the buffer or any hint on how the buffer is used.

The next level of abstraction in the Android graphics stack is the user interface (UI) library, where an `ANativeWindowBuffer` struct is defined by putting together a Gralloc buffer with a description of its content and usage.

The `ANativeBuffer` struct is wrapped around a `GraphicBuffer` object that adds some convenient methods to the plain struct.

The application must create a Gralloc buffer by instantiating a GraphicBuffer, as in the following code example:

```
GraphicBuffer* graphicBuffer =
    new GraphicBuffer(
        width,
        height,
        (PixelFormat)pixel_format,
        GraphicBuffer::USAGE_HW_TEXTURE |
            GraphicBuffer::USAGE_HW_RENDER |
            GraphicBuffer::USAGE_SW_WRITE_RARELY |
            GraphicBuffer::USAGE_SW_READ_RARELY);
```

A pointer to a GraphicBuffer, the wrapper for an `ANativeWindowBuffer`, that is created in that way, can be used for specifying the image storage that an EGL image references to. In order to allow the creation of an EGL image out of an `ANativeWindowBuffer` pointer, Google requires that all Android vendors support the `ANDROID_image_native_buffer` extension, sitting on top of `EGL_KHR_image_base`. The `ANDROID_image_native_buffer` extension defines values for the `eglCreateImageKHR` parameters that must be supported on Android. When the target parameter is `EGL_NATIVE_BUFFER_ANDROID` then the buffer parameter can be an `ANativeWindowBuffer` pointer cast to `EGLClientBuffer`. To create an EGL image for the GraphicBuffer instance from the previous example, invoke `eglCreateImageKHR` as:

```
EGLImageKHR eglImage = eglCreateImageKHR(
                display,
                contenxt,
                EGL_NATIVE_BUFFER_ANDROID,
                graphicBuffer->getNativeBuffer(),
                NULL);
```

The resulting EGL image can be used in OpenGL and OpenCL.

D.1.2.1 OpenCL supported formats

Shareable buffer formats that are supported for OpenCL.

The list of `ANativeWindowBuffer` formats that are supported for sharing data between OpenGL and OpenCL are as follows:

- HAL_PIXEL_FORMAT_RGBA_8888
- HAL_PIXEL_FORMAT_BGRA_8888
- HAL_PIXEL_FORMAT_RGB_565
- HAL_PIXEL_FORMAT_RGBX_8888
- HAL_PIXEL_FORMAT_YV12

OpenCL image writing is only allowed for:

- HAL_PIXEL_FORMAT_RGBA_8888
- HAL_PIXEL_FORMAT_BGRA_8888
- HAL_PIXEL_FORMAT_RGB_565
- HAL_PIXEL_FORMAT_RGBX_8888

The result of writing to any format not listed here is undefined.

As a courtesy to OpenCL developers, the Mali™ OpenCL driver returns informative error codes on the `clCreateFromEGLImageKHR` calls:

CL_IMGE_FORMAT_NOT_SUPPORTED
> If the format is not supported by the OpenCL driver.

CL_INVALID_OPERATION
> If the format is known but the flags are invalid, because the `CL_MEM_WRITE_ONLY` or `CL_MEM_READ_WRITE` flags are specified for a format for which writing is not supported.

D.1.3 EGL_EXT_image_dma_buf_import

The DDK supports the `EGL_EXT_image_dma_buf_import` extension on all the Linux variants and also on Android. This extension allows applications to import images allocated through the `dma_buf` low level API.

The extension defines acceptable values for the target and buffer parameters from the `eglCreateImageKHR` call:

target
> Target must be `EGL_LINUX_DMA_BUF_EXT`.

buffer
> Buffer must be `NULL`.

Additional properties also defined by the extension are used for telling the `dma_buf` file descriptors, format, width, and height.

D.2 The cl_arm_printf extension

The OpenCL extension `cl_arm_printf` enables you to use the `printf()` function in your kernels.

> **Note**
>
> the `printf()` function is included in OpenCL 1.2.

D.2.1 About the cl_arm_printf extension

The implementation of the cl_arm_printf extension uses a callback function that delivers the output data to the host from the device. You must write this callback function.

You must register the callback function as a property of the context when the OpenCL context is created. It is called when an OpenCL kernel completes. If the callback is not registered, the `printf()` output is still produced but it is not available.

Messages are stored atomically and complete in the output buffer. The buffer is implemented as a circular buffer so if the output is longer than the buffer size, the output wraps around on itself. In this case, only the last part of the output is available.

You can configure the size of the buffer. The default size is 1MB.

Related information

http://www.khronos.org

D.2.2 cl_arm_printf example

The example code shows the cl_arm_printf extension in use. It shows how to use the buffer-size property and the callback property that is required to get output.

The example code prints *Hello, World!* on the console:

```
#include <stdio.h>
#include <CL/cl.h>
#include <CL/cl_ext.h>
const char *opencl =
    "__kernel void hello()\n"
    "{\n"
    " printf(\"Hello, World!\\n\");\n"
    "}\n";

void callback(const char *buffer, size_t length, size_t final, void *user_data)
{
    fwrite(buffer, 1, length, stdout);
```

```
}

int main()
{
    cl_platform_id platform;
    cl_device_id device;
    cl_context context;
    cl_context_properties context_properties[] =
    {
        CL_CONTEXT_PLATFORM, 0,
        CL_PRINTF_CALLBACK_ARM, (cl_context_properties)callback,
        CL_PRINTF_BUFFERSIZE_ARM, 0x1000,
        0
    };
    cl_command_queue queue;
    cl_program program;
    cl_kernel kernel;

    clGetPlatformIDs(1, &platform, NULL);
    context_properties[1] = (cl_context_properties)platform;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    context = clCreateContext(context_properties, 1, &device, NULL, NULL, NULL);
    queue = clCreateCommandQueue(context, device, 0, NULL);

    program = clCreateProgramWithSource(context, 1, &opencl, NULL, NULL);
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    kernel = clCreateKernel(program, "hello", NULL);
    clEnqueueTask(queue, kernel, 0, NULL, NULL);

    clFinish(queue);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    return 0;
}
```

D.3 The cl_arm_import_memory extensions

The `cl_arm_import_memory` extensions enable you to directly import memory into OpenCL using the `clImportMemoryARM()` function.

The following extensions enable OpenCL kernels to access imported memory buffers and process images created from imported buffers:

- cl_arm_import_memory

- cl_arm_import_memory_android_hardware_buffer

- cl_arm_import_memory_dma_buf

- cl_arm_import_memory_host

- cl_arm_import_memory_protected

The Mali™ Driver Development Kit includes tests that demonstrate how you can use these features.

> **Note**
>
> For more information about these extensions, see the Khronos extension specifications at https://www.khronos.org/.

D.4 The cl_arm_job_slot_selection extension

The `cl_arm_job_slot_selection` extension enables applications to select the job slot to use for work submission to the GPU.

The job slot is selected at command queue creation time via the `CL_QUEUE_JOB_SLOT_ARM` property. Applications can use the `CL_DEVICE_JOB_SLOTS_ARM` device info query to get a bitmask of allowed job slots.

One possible use-case for this extension could be to reduce the latency of submitting work to the GPU when the GPU is shared between multiple workloads. For example, context switching between two workloads requires the work scheduled on the GPU to yield to a soft-stop request before scheduling the new work, which in turn requires all the threads running on the GPU to complete execution. This can take a long time with threads running compute work and lead to an unacceptable latency before other workloads begin execution on the GPU. To improve this behavior, is it possible to partition the GPU into multiple groups of shader cores that can each be targeted by different job slots. Beginning execution on a given partition does not require the work on other partitions to complete, therefore reducing the latency of starting the new work.

The Mali kernel module exposes a `core_mask sysfs` file that allows allocation of shader cores to specific job slots, effectively creating partitions in the GPU. For example, for a typical GPU that has 12 cores and three job slots, the default masks for all job slots would likely be `0xFFF`, meaning that all cores are allocated to all job slots. This allocation can be changed by writing a space-separated list of core masks for each of the job slots to the core_mask file. This creates two partitions, one allocating four cores to job slots 0 and 1, and another allocating eight cores to job slot 2, as follows:

```
echo '0xF00 0xF00 0x0FF' > /sys/path/to/mali/core_mask
```

An OpenCL application can then use the extension to send compute work to job slot 2, targeting the partition with eight cores. Any other work on the GPU that must start execution quickly can target job slot 0 and or 1 without needing the work resident on the GPU partition linked to job slot 2 to soft-stop.

Power management strategies have to be considered when partitioning the GPU. Attempting to submit work to a job slot that does not have any online cores will result in a GPU fault.

> **Note**
>
> For more information about this extension, see the Khronos extension specifications at https://www.khronos.org/.

D.5 The cl_ext_cxx_for_opencl extension

The `cl_ext_cxx_for_opencl` extension enables the driver to compile kernel code in C++ for OpenCL mode.

For more details, see https://www.khronos.org/registry/OpenCL/extensions/ext/cl_ext_cxx_for_opencl.html.

This language mode can be enabled using the `-cl-std=CLC++` flag accepted by both the online compiler and the `mali_clcc` offline compiler.

The following example shows C++ for OpenCL kernel code:

```
template<class T>
T add(T x, T y)
{
  return x + y;
}

__kernel void k_int(__global int * a, __global int * b)
{
  auto index = get_global_id(0);
  a[index] = add(b[index], b[index + 1]);
}

__kernel void k_float(__global float * a, __global float * b)
{
  auto index = get_global_id(0);
  a[index] = add(b[index], b[index + 1]);
}
```

The C++ for OpenCL language is documented in: https://www.khronos.org/opencl/assets/CXX_for_OpenCL.pdf.

Note

For more information about the extensions, see the Khronos extension specifications at https://www.khronos.org/.

D.5.1 Limitation of the current implementation of cl_ext_cxx_for_opencl

C++ for OpenCL support is in the experimental phase and there are a few known limitations.

Missing diagnostics

Mali compiler fails to diagnose some invalid types used in kernel arguments and objects that are created in the local address space.

Therefore, it is not recommended to use non-POD type in kernel arguments or member references in classes that are created in the local address space.

No automatic support for program scope variables with non-trivial constructors or destructors

Program scope variables with non-trivial constructors or destructors, for example global constructors or global destructors, are not supported automatically with the `cl_ext_cxx_for_opencl` extension. A manual workaround for constructors is needed in the application host side, which is explained in the *Clang User Manual*, see https://clang.llvm.org/docs/UsersManual.html#constructing-and-destroying-global-objects.

The following example shows an implementation of the host helper function demonstrating how to query the constructor stub kernel name from a single translation unit.

```
// Helper function returns name of ctor stub if found in the program or empty
string otherwise.
std::string get_ctor_kernel_name_to_enqueue(cl_program program, size_t
name_size) {
  std::string kernel_names(name_size);
  cl_err err = clGetProgramInfo(program, CL_PROGRAM_KERNEL_NAMES, name_size,
&kernel_names[0], nullptr);

  // Handling of err code is omitted.

  // Global ctor stub name is as follows _GLOBAL__sub_I_<translation unit
name>.
  auto ctor_name_pos = kernel_names.find("_GLOBAL__sub_I_");
  auto end_pos       = std::string::npos;
  if (ctor_name_pos != std::string::npos)
  {
     end_pos = kernel_names.find(";", ctor_name_pos);
     end_pos = end_pos != std::string::npos ? end_pos - ctor_name_pos :
std::string::npos;
  }
  return kernel_names.substr(ctor_name_pos, end_pos);
}
```

When a program consists of multiple translation units, each unit containing a global constructor is linked to a combined module. The application code has to make sure to retrieve names of all constructor kernels and enqueue them in a sequence.

Note

There is no automatic support nor manual support of non-trivial constructors or destructors for static variables inside functions.

D.6 The cl_arm_controlled_kernel_termination extension

The `cl_arm_controlled_kernel_termination` extension allows an OpenCL kernel to exit or abort the execution of the kernel code, by using the `arm_terminate_kernel` function on the GPU, before the OpenCL kernel naturally comes to the end of the execution.

There are two termination types:

ARM_TERMINATION_SUCCESS

This type means successful termination of an OpenCL kernel. Any subsequent enqueued work still runs.

This termination type is intended to reduce the running time of the kernel once a certain result has been achieved. It does not have to be a success about what the kernel is intended to do. For example, a face recognition algorithm may exit with success and report a result of false for face detection.

ARM_TERMINATION_FAILURE

This type means termination failure of an OpenCL kernel. Any subsequent work does not run.

This termination type is intended for a situation where something is wrong in the kernel code. For example, validating inputs to a kernel or detecting code that is not expected to be reached, such as a default in a switch. By detecting these failures, debugging the kernel code becomes easier, rather than continuing to execute the kernel code. Because it stops the execution at the point of failure, instead of letting the code behave weird or crash at a later stage.

D.7 The cl_khr_suggested_local_work_size extension

The `cl_khr_suggested_local_work_size` extension allows an application to query the local work size that the OpenCL runtime would choose when enqueuing a kernel without explicitly specifying the local work size.

This extension can be useful if an application enqueues the same kernel multiple times with same global work size and global offsets, but wants the runtime to select a local work size. The size that the runtime chooses as a local work size can be queried once and then specified explicitly when the application enqueues the kernel each time. It avoids the overhead of the runtime calculating the local work size for each enqueue operation.

For more information about this extension, see https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_Ext.html#cl_khr_suggested_local_work_size.

Appendix E OpenCL 1.2

This appendix describes some of the important changes to the Mali™ OpenCL driver in OpenCL 1.2.

E.1 OpenCL 1.2 compiler options

OpenCL 1.2 adds options for offline and online compilation.

The following options are added in OpenCL 1.2:

Online compilation

In OpenCL 1.2, you can compile and link separate OpenCL files during online compilation. Any compiler flags you specify for compilation are used for frontend and middle-level optimizations. These flags are discarded during backend compilation and linking. You can also specify a limited list of compiler flags during the linking phase. These flags are forwarded to the compiler backend after linking.

You can supply different flags during the compilation of different modules because they only affect the frontend and mid-level transformation of separate modules. Later in the build process, the commonly linked module overrides these flags with the flags passed during the linking phase, to the overall linking program. It is safe to mix modules that are compiled separately with different various options, because only a limited set of linking flags are applied to the overall program.

The full set of flags can only affect early compilation steps. For example, if `-cl-opt-disable` is passed, it only disables the early optimization phases. During the linking phase, the `-cl-opt-disable` option is ignored and the backend optimizes the module. `-cl-opt-disable` is ignored because it is not a permitted link-time option.

Offline compilation

For customers with access to the `mali_clcc` offline compiler, in OpenCL 1.2, the compilation and linking steps are not available separately on the command line. Compilation and linking occur in one stage, with the source files you specify on the command line.

You can specify several build options together with the source files. These flags are applied to all files and all compilation phases from the frontend to the backend, to produce the final binary.

For example:

```
mali_clcc -cl-opt-disable file1.cl file2.cl -o prog.bin
```

E.2 OpenCL 1.2 compiler parameters

OpenCL 1.2 adds a number of compiler parameters.

OpenCL 1.2 includes the following compiler parameters:

-create-library.

The compiler creates a library of compiled binaries.

-enable-link-options.

This enables you to modify the behavior of a library you create with `-create-library`.

-cl-kernel-arg-info.

This enables the compiler to store information about the arguments of kernels, in the program executable.

E.3 OpenCL 1.2 functions

The following API functions are added in OpenCL 1.2.

OpenCL includes the following API functions:

clEnqueueFillBuffer()

Arm recommends you use this function in place of writing your own.

clEnqueueFillImage()

Arm recommends you use this function in place of writing your own.

clCreateImage()

This includes support for 1D and 2D image arrays.

> **Note** This function deprecates all previous image creation functions.

---

clLinkProgram()

Using this typically does not provide much performance benefit in the Mali™ OpenCL driver.

clCompileProgram()

Using this typically does not provide much performance benefit in the Mali™ OpenCL driver.

clEnqueueMarkerWithWaitList()

clEnqueueBarrierWithWaitList()

clEnqueueMigrateMemObjects()

The Mali™ OpenCL driver supports the memory object migration API `clEnqueueMigrateMemObjects()`, but this does not provide any benefit because Mali™ GPUs use a unified memory architecture.

OpenCL 1.2 includes the following built-in function:

printf()

clUnloadPlatformCompiler()

Using this function releases accumulated resources in the OpenCL compiler, which can free up some memory at the cost of taking more time in the subsequent compiler operations.

Note

The flag `CL_MAP_WRITE_INVALIDATE_REGION` has no effect in the Mali™ OpenCL driver.

E.4 Functions deprecated in OpenCL 1.2

Several functions are deprecated in OpenCL 1.2 but are still available in the Mali™ OpenCL driver.

The deprecated functions are:

- `clEnqueueMarker()`
- `clEnqueueBarrier()`
- `clEnqueueWaitForEvents()`
- `clCreateImage2D()`
- `clCreateImage3D()`
- `clUnloadCompiler()`
- `clGetExtensionFunctionAddress()`

Appendix F OpenCL 2.0

This appendix describes the important changes to the Mali™ OpenCL driver in OpenCL 2.0.

F.1 About OpenCL 2.0

OpenCL 2.0 adds several new high-level features.

OpenCL 2.0 is a backwards compatible version of OpenCL that adds several new high-level features as well as adding smaller changes to some of the existing features. Several of the new features provide better performance for specific use-cases.

New features include:

- Shared virtual memory.
- Program scope variables.
- C11 atomics.
- Generic address space.
- Pipes.
- Device execution.

F.2 OpenCL 2.0 functions

The following API and built-in functions are added in OpenCL 2.0.

For detailed descriptions of these functions, see the OpenCL 2.0 documentation at the Khronos Group at https://www.khronos.org/.

F.2.1 OpenCL 2.0 API functions

Several new API functions are added in OpenCL 2.0.

The following functions support `CommandQueues` and `Samplers`:

Table F-1: CommandQueue and Sampler functions

Function name	Description
`clCreateCommandQueueWithProperties()`	Variant of `clCreateCommandQueue()` that offers a wider range of properties.
`clCreateSamplerWithProperties()`	Variant of `clCreateSampler()` that offers a wider range of properties.

The following functions support OpenCL pipes:

- `clCreatePipe()`
- `clGetPipeInfo()`

The following functions support Shared Virtual Memory:

- `clSVMAlloc()`
- `clSVMFree()`
- `clSetKernelArgSVMPointer()`
- `clSetKernelExecInfo()`
- `clEnqueueSVMFree()`
- `clEnqueueSVMMemcpy()`
- `clEnqueueSVMMemFill()`
- `clEnqueueSVMMap()`
- `clEnqueueSVMUnmap()`

F.2.2 OpenCL 2.0 built-in functions

Several new built-in functions are added in OpenCL 2.0.

OpenCL 2.0 adds many functions for atomic operations, compatible with the C11 standard.

Atomic functions are useful for both when multiple threads on the GPU access the same data, and when GPU and CPU use shared data via SVM in fine-grained mode.

The following table shows examples of the main types of atomic functions:

Table F-2: Atomic operation functions

Function name	Description
`atomic_fetch_add()`	Add a value to an object, return the value of object before the addition.
`atomic_fetch_sub()`	Subtract a value from an object, return the value before the subtraction.
`atomic_load()`	Load, that is, fetch the value of the atomic object.
`atomic_store()`	Store a new value in an atomic object.
`atomic_compare_exchange_strong()`	Compare the value of an atomic object to an expected value, and replace with the desired if a match. Returns true 1 if the value was replaced, or false 0 when the expected value is not matching - expected value is updated to match the current value of the atomic object.
`atomic_compare_exchange_weak()`	Same as previously, but can sometimes fail even if the object matches the desired value.
`atomic_flag_test_and_set()`	Atomically fetch the current flag value, set it to true (1) and return the value before it was set.

Device execution functions support creating new work, creating, setting, and destroying events. They also support constructing ndrange sets.

The following functions support device execution:

- `create_user_event()`

- `retain_event()`

- `release_event()`

- `set_user_event_status()`

- `is_valid_event()`

- `enqueue_kernel()`

- `ndrange_1D(), ndrange_2D(), ndrange_3D()`

Functions for pipes enable the CL kernel to read, write individual or multiple packets, and check the current status of a pipe.

The following functions support pipes:

- `read_pipe()`

- `write_pipe()`

- `reserve_read_pipe()`

- `reserve_write_pipe()`

- `commit_read_pipe()`

- `commit_write_pipe()`

- `is_valid_reserve_id()`

- `work_group_reserve_read_pipe()`

- `work_group_reserve_write_pipe()`

- `work_group_commit_write_pipe()`

- `work_group_commit_read_pipe()`

- `get_pipe_num_packets()`

- `get_pipe_max_packets()`

F.3 OpenCL 2.0 compiler options

The following compiler options are added in OpenCL 2.0.

OpenCL 2.0 adds support for the following options:

Table F-3: OpenCL 2.0 compiler options

Name	Description
`-cl-uniform-work-group-size`	Force the driver to accept only uniform work-group sizes.
`-g`	Provides extra debug information.

OpenCL 2.0 also supports all options from the previous versions of OpenCL.

F.4 Program scope variables

In OpenCL 2.0, you can use global variables, known as program scope variables, to store data shared between different kernels and multiple invocations of the same kernel.

Program scope variables are analogous to global variables in C or C++ programming, with similar benefits and drawbacks. The lifetime of the global variables corresponds to the lifetime of the program that defines the variables. The use of global variables in OpenCL is transparent to the host-side code.

The host code must enable CL2.0 when compiling with `-cl-std=CL2.0`, when building the OpenCL program. For example:

```
cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, &err);
err = clBuildProgram(program, 0, NULL, "-cl-std=CL2.0", NULL, NULL);
```

The following code sample shows how to use a global variable:

```
__global int my_global;

__kernel void kernel_one(int x)
{
 my_global = x;
}

__kernel void kernel_two(__global int *the_answer)
{
 *the_answer = my_global;
}
```

For more information, see the *OpenCL C specification V2.0*, section 6.5 Address space qualifiers.

F.5 Functions deprecated in OpenCL 2.0

Several functions are deprecated in OpenCL 2.0 but are still available in the Mali™ OpenCL driver.

The deprecated functions are:

- `clCreateCommandQueue()`
- `clCreateSampler()`
- `clEnqueueTask()`
- The `clGetDeviceInfo()` param_name of `CL_DEVICE_HOST_UNIFIED_MEMORY`
- The `clGetDeviceInfo()` param_name of `CL_DEVICE_QUEUE_PROPERTIES`
- The `clGetImageInfo()` param_name of `CL_IMAGE_BUFFER`

F.6 OpenCL 2.0 extensions

One extension is added in OpenCL 2.0.

The following extension is added in OpenCL 2.0:

`cl_khr_depth_images`

Enables support for depth and depth-stencil images.

F.7 OpenCL 2.0 optimizations

OpenCL 2.0 includes several features that can improve performance over OpenCL 1.2.

OpenCL 2.0 includes the following features that you can use to optimize your code:

Shared virtual memory

On a fully coherent platform, shared virtual memory reduces the requirement to call map and unmap API functions, when a memory region is used on both the GPU and the application processor. See F.8 Shared virtual memory on page 102.

Read-Write images

This enables the same kernel to both read from and write to a single image, that when used correctly, can improve cache efficiency and reduce memory usage.

Generic Address space

This enables code to be written once, and it works in any address space.

sRGB images

If the OpenCL kernel is reading from an sRGB image, it is not required to be translated to RGB before it can be used, the read_image call converts to standard RGB as part of the read operation.

Program scope variables

In some circumstances, program scope variables can be useful to avoid passing data from the host program to multiple kernels. For example, if a kernel is calculating a histogram, storing that in a buffer, the host program then passes the same buffer to another kernel that does some other part of the work, using the histogram, and the histogram is never used on the host, then a plausible solution is to make the histogram into a global variable in the program. Both kernels must be part of the same program for this to work correctly. As always, using global variables does have some drawbacks, particularly when it comes to understanding what variables can be modified by what parts of the code.

Pipes and device execution

Arm recommends that you avoid using the OpenCL pipes and device execution functionality. See F.9 OpenCL 2.0 pipes and device execution on page 105.

F.8 Shared virtual memory

Shared Virtual Memory (SVM) is a feature of OpenCL 2.0 that enables the same virtual memory address range to be used on both the GPU and the application processor.

There are two types of SVM:

Fine-grained
> This is available when your platform supports full coherency.

Coarse-grained
> This is for non-coherent or IO-coherent platforms.

SVM has the following advantages:

- It has lower overhead than the traditional `cl_buffer` interface.

- SVM is easier to use in the host program because it is only a pointer to data. With full coherency, you can use the memory without the overhead of calls to map and unmap functions. It is also possible to use atomic operations on both the GPU and application processor side to update data that is shared between the two architectures.

  The map and unmap functions are still required with coarse-grained SVM.

- It is easier to share work-loads between the GPU and application processor, because the address of the memory is the same in both GPU and application processor.

  This enables you to build data structures that naturally use pointers such as linked lists or binary trees, on the host application processor, and the GPU can traverse these without having to translate the pointer values.

The following code-fragments are examples that show the difference between using a pointer and using a CL buffer.

> **Note**
> This code only illustrates the difference, between the use of SVM buffer and CL buffer, it is not a complete example.

The first example shows the traditional approach of sharing data with the `cl_buffer` interface:

```
/* Create and prepare buffer content */

size_t buffer_size = 100 * 1024;
cl_buffer *buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, buffer_size, NULL,
 &err);
cl_event buffer_event;
void *buffer_map_ptr = clEnqueueMapBuffer(queue, CL_NON_BLOCKING, CL_MAP_WRITE,
 buffer, 0, buffer_size, 0, NULL, &buffer_event, &err);
... some other code, perhaps ...
clWaitForEvent(1, &buffer_event);
... use buffer_map_ptr to fill the content ...
clEnqueueUnmapBuffer(queue, buffer, 0, NULL, &buffer_event);

/* Now do some actual CL kernel work. */
```

```
cl_event kernel_event;
clSetKernelArg(kernel, 0, &buffer);
clEnqueueNDRangeKernel(queue, kernel, NULL, work_dim, global_size, local_size, 1,
 &buffer_event, &kernel_event);
... Ideally do some other work here ...
clWaitForEvent(1, &kernel_event);

/* make use of the new buffer content */
buffer_map_ptr = clEnqueueMapBuffer(queue, CL_NON_BLOCKING, CL_MAP_WRITE, buffer, 0,
 buffer_size, 0, NULL, &buffer_event, &err);
... some other code, perhaps ...
clWaitForEvent(1, &buffer_event);
... use buffer_map_ptr to get data out of the buffer ...
clEnqueueUnmapBuffer(queue, buffer, 0, NULL, &buffer_event);
```

In a coherent system, you can use SVM to write code like the following:

```
/* Create and prepare buffer content */
size_t buffer_size = 100 * 1024;
void *buffer = clSvmAlloc(context, CL_MEM_READ_WRITE, buffer_size, 0);
... use buffer to fill the content ...

/* Now do some actual CL kernel work. */
clSetKernelArgSVMPointer(kernel, 0, buffer);
cl_event kernel_event;
clEnqueueNDRangeKernel(queue, kernel, NULL, work_dim, global_size, local_size, 0,
 NULL, &kernel_event);
... Ideally do some other work here ...
clWaitForEvent(1, &kernel_event);

/* make use of the new buffer content */
... use buffer to get the data stored by the kernel ...
clSVMFree(queue, buffer);
```

In a non-coherent system, it is still possible to use SVM, but you must use `map()` and `unmap()` calls to ensure that the view of the memory content is up to date on the application processor and the GPU.

```
/* Create and prepare buffer content */
size_t buffer_size = 100 * 1024;
void *buffer = clSvmAlloc(context, CL_MEM_READ_WRITE, buffer_size, 0);
clEnqueueSVMMap(queue, CL_NON_BLOCKING, CL_MAP_WRITE, buffer, buffer_size, 0, NULL,
 &buffer_event);
clWaitForEvent(1, &buffer_event);

... use buffer to fill the content ...
clEnqueueSVMUnmap(queue, buffer, 0, NULL, &buffer_event)
/* Now do some actual CL kernel work. */
clSetKernelArgSVMPointer(kernel, 0, buffer);
cl_event kernel_event;
clEnqueueNDRangeKernel(queue, kernel, NULL, work_dim, global_size, local_size, 1,
 &buffer_event, &kernel_event);
... Ideally do some other work here ...
clEnqueueSVMMap(queue, CL_NON_BLOCKING, CL_MAP_WRITE, buffer, buffer_size, 0, NULL,
 &buffer_event);
clWaitForEvent(1, &buffer_event);

/* make use of the new buffer content */
... use buffer to get to the data stored by the kernel ...

clEnqueueSVMUnmap(queue, buffer, 0, NULL, &buffer_event)
clWaitForEvent(1, &buffer_event);
clSVMFree(queue, buffer);
```

> **Note**
>
> You can use `map()` and `unmap()` calls in a coherent system, but there is still some overhead from the API functions.

F.9 OpenCL 2.0 pipes and device execution

OpenCL 2.0 adds pipes and device execution features.

OpenCL 2.0 adds the following features:

- Device execution.
- Pipes.

> **Note**
>
> Arm recommends that you avoid using the OpenCL pipes and device execution functionality. These are not tested beyond conformance and are not hardware accelerated.

Appendix G OpenCL 2.1

This appendix describes the important changes to the Mali™ OpenCL driver in OpenCL 2.1.

G.1 About OpenCL 2.1

OpenCL 2.1 adds several new high-level features.

OpenCL 2.1 is backwards compatible with previous versions of OpenCL. This version of OpenCL is supported on all Valhall GPUs and Bifrost GPUs from Mali™-G76.

New features include:

- Support for Intermediate Language (IL) programs.
- Device and host timer.
- Priority hints.
- Subgroup support.

For detailed information about OpenCL 2.1, see the Khronos specification: https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf.

G.2 OpenCL 2.1 functions

The following API and built-in functions are added in OpenCL 2.1.

For detailed descriptions of these functions, see the OpenCL 2.1 documentation at the Khronos Group at https://www.khronos.org/.

G.2.1 OpenCL 2.1 API functions

Several new API functions are added in OpenCL 2.1.

The new functions are:

- `clCloneKernel`
- `clCreateProgramWithIL`
- `clEnqueueSVMMigrateMem`
- `clGetDeviceAndHostTimer`
- `clGetHostTimer`
- `clGetKernelSubGroupInfo`
- `clSetDefaultDeviceCommandQueue`

G.2.2 OpenCL 2.1 built-in functions

Several new built-in functions are added in OpenCL 2.1.

The new functions are:

- `get_enqueued_num_sub_groups`
- `get_kernel_max_sub_group_size_for_ndrange`
- `get_kernel_sub_group_count_for_ndrange`
- `get_max_sub_group_size`
- `get_num_sub_groups`
- `get_sub_group_size`
- `get_sub_group_local_id`
- `get_sub_group_id`
- `sub_group_all`
- `sub_group_any`
- `sub_group_barrier`
- `sub_group_broadcast`
- `sub_group_commit_read_pipe`
- `sub_group_commit_write_pipe`
- `sub_group_reduce_<op>`
- `sub_group_reserve_read_pipe`
- `sub_group_reserve_write_pipe`
- `sub_group_scan_exclusive_<op>`
- `sub_group_scan_inclusive_<op>`

G.3 Intermediate language programs

The Intermediate Language (IL) is partly compiled code and enables developers to distribute OpenCL kernel code without distributing the CL source code. IL code is also portable between different OpenCL platforms.

IL code uses the SPIR-V format to store the intermediate form where the compiler has parsed the code and produced a form of "virtual machine code". This "virtual machine code" can be further processed to make real machine code for a specific platform.

To compile the SPIR-V, use a general purpose OpenCL to SPIR-V compiler. The compiled SPIR-V is stored as a binary file or an array of bytes in the OpenCL application code to pass to the `clCreateProgramWithIL` function.

Compiling

Compiling OpenCL to SPIR-V can be done in two ways, either using Clang and `llvm-spirv`:

```
clang -Xclang -finclude-default-header -cl-std=CL2.0 --target=spir[64]-unknown-
unknown
-emit-llvm -c -O0 -o <file.bc> <file.cl>
llvm-spirv -o <file.spv> <file.bc>
```

Or for customers with access to the `mali_clcc` offline compiler, by using `mali_clcc`:

```
mali_clcc -emit=spir-v file.cl
```

Optionally, use -b32 or -b64 to specify a 32-bit or 64-bit version of the code. Set the -b32 or -b64 option to the same as the OpenCL application code, so a 32-bit application uses 32-bit OpenCL code.

Note

Clang and `llvm-spirv` are available from https://github.com/KhronosGroup/SPIRV-LLVM-Translator.

G.4 Device and host timer functions

The `clGetHostTimer` and `clGetDeviceAndHostTimer` functions are recommended for measuring the time in code that relates to OpenCL.

The Arm® implementation guarantees that both functions match the GPU performance counters in the underlying time-source which ensures a consistent scale for time measurement.

G.5 Queue priority hints

In OpenCL 2.1, you can assign priority to a command queue.

The `CL_KHR_PRIORITY_HINTS` extension allows you to assign priority to a command queue. Use the `CL_QUEUE_PRIORITY_KHR` property when creating a command queue with the function `clCreateCommandQueueWithProperties`.

The supported priorities are:

- `CL_QUEUE_PRIORITY_HIGH_KHR`

- `CL_QUEUE_PRIORITY_MED_KHR`

- `CL_QUEUE_PRIORITY_LOW_KHR`

The priority affects the order work is taken from the queues, with high priority work taken first. Once enqueued work has been taken from the queue, it continues until completion, even if higher priority work is enqueued before the lower priority work completes.

Appendix H OpenCL 3.0

This appendix describes the important changes to the Mali™ OpenCL driver in OpenCL 3.0.

H.1 About OpenCL 3.0

OpenCL 3.0 adds several new high-level features.

OpenCL 3.0 is not guaranteed to be backwards compatible with previous versions of OpenCL. OpenCL 3.0 is supported on all Valhall GPUs and Bifrost GPUs.

For detailed information about OpenCL 3.0, see the Khronos specification: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

H.2 OpenCL 3.0 functions

Several new functions are added in OpenCL 3.0.

clSetContextDestructorCallback()
    Allows you to register a callback to be called when the context is destroyed.

clCreateBufferWithProperties()
    Allows the creation of buffers with properties.

clCreateImageWithPropertie()
    Allows the creation of images with properties.

Other changes
clSetProgramReleaseCallback()
    Deprecated in OpenCL 3.0.

clEnqueueSVMMigrateMem()
    Added with the CL_COMMAND_SVM_MIGRATE_MEM event command type.

Appendix I   Revisions

This appendix contains a list of technical changes made between releases and where they are documented in this guide.

I.1 Revisions

This appendix describes the technical changes between releases of this book.

Table I-1: Issue 0100-00

Change	Location
First release for version 1.0.	-

Table I-2: Differences between issue 0100-00 and 0200-00

Change	Location
Added batching kernels submission information.	10.1 General optimizations on page 62
Added Valhall GPU information.	Various

Table I-3: Differences between issue 0200-00 and 0300-00

Change	Location
Added a table showing the number of adjacent threads for the different GPUs.	10.6 Mali Bifrost and Valhall GPU specific optimizations on page 70
Added support for OpenCL 2.1.	Various
Corrected the specified NDRanges.	4.4 OpenCL data processing on page 23

Table I-4: Differences between issue 0300-00 and 0301-00

Change	Location
Added the `cl_arm_job_slot_selection` extension.	Various

Table I-5: Differences between issue 0301-00 and 0302-00

Change	Location
Added Mali™-G57 to the adjacent threads table.	10.6 Mali Bifrost and Valhall GPU specific optimizations on page 70
Added the `cl_arm_import_memory_protected` extension.	Various

Table I-6: Differences between issue 0302-00 and 0400-00

Change	Location
Added subgroup information for devices that support OpenCL 2.1.	7.5.1 About dividing data for OpenCL on page 49
Added the extension: `cl_arm_integer_dot_product_accumulate_saturate_int8`.	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82

Change	Location
Added the `cl_arm_import_memory_android_hardware_buffer` and `cl_khr_extended_versioning` extensions.	D.3 The cl_arm_import_memory extensions on page 90
	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82
Added that a call to `clCreateKernel` is necessary for `clGetProgramInfo` to return the machine code.	6.5 Creating OpenCL program objects on page 34
Changed the terms arithmetic pipeline, load-store pipeline, and texture pipeline to arithmetic unit, load-store unit, and texture unit.	8.2.2 About Mali GPU architectures on page 53
	9.1 The optimization process for OpenCL applications on page 56
	10.6 Mali Bifrost and Valhall GPU specific optimizations on page 70
Clarified the reference count information.	6.10 Cleaning up unused objects on page 39
Clarified the requirements for high performance tuning.	5.1 Software and hardware requirements for Mali GPU OpenCL development on page 29
Clarified the vectorizing code information.	8.1 About retuning existing OpenCL code for Mali GPUs on page 52
Removed redundant information.	6.8.3 Determining the local work-group size on page 37
	B. OpenCL built-in functions on page 80
Removed the section of not allocating memory buffers created with malloc() for OpenCL applications.	-
Removed the arithmetic pipe information.	2.1 About Arm Mali GPUs on page 15
Removed the explicit mention of OpenCL version 1.2 as a supported version.	2.3 About the Mali GPU OpenCL driver and support on page 15
Updated the section of enqueue many work-items.	10.1 General optimizations on page 62
Updated the image objects description.	6.7.2 Creating memory objects on page 36
Updated the kernel optimizations section.	10.2 Kernel optimizations on page 64
Updated the private memory information.	4.8 Mali GPU OpenCL memory model on page 27

Table I-7: Differences between issue 0401-00 and 0400-00

Change	Location
Added Mali™-G78 to the adjacent threads table.	10.6 Mali Bifrost and Valhall GPU specific optimizations on page 70

Table I-8: Differences between issue 0400-00 and 0402-00

Change	Location
Added `cl_ext_cxx_for_opencl`, `cl_khr_device_uuid`, and `cl_arm_scheduling_controls`.	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82
Newly added.	D.5 The cl_ext_cxx_for_opencl extension on page 92
Added `clUnloadPlatformCompiler()`.	E.3 OpenCL 1.2 functions on page 96
Added a note for `cl_arm_thread_limit_hint`.	10.2 Kernel optimizations on page 64

Table I-9: Differences between issue 0402-00 and 0403-00

Change	Location
Added a note.	6.8.5 Executing kernels on page 38

Table I-10: Differences between issue 0403-00 and 0404-00

Change	Location
Corrected the description for `half` build-in scalar data type.	A.2.1 Built-in scalar data types on page 77
Added the extension of `cl_arm_controlled_kernel_termination`.	D.6 The cl_arm_controlled_kernel_termination extension on page 93

Table I-11: Differences between issue 0404-00 and 0405-00

Change	Location
Removal the content about `cl_arm_thread_limit_hint`.	6.8.2 Determining the optimal global work size on page 37, 10.2 Kernel optimizations on page 64, C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82
Added OpenCL 3.0.	H. OpenCL 3.0 on page 110
Updated content for OpenCL 3.0.	2.3 About the Mali GPU OpenCL driver and support on page 15
Updated the links for `cl_ext_cxx_for_opencl`.	D.5 The cl_ext_cxx_for_opencl extension on page 92
Added the extension of `cl_khr_subgroup_extended_types`.	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82

Table I-12: Differences between issue 0405-00 and 0406-00

Change	Location
Added the extension of `cl_khr_suggested_local_work_size`.	D.7 The cl_khr_suggested_local_work_size extension on page 94
Updated the content about compilation deferring.	10.4 Execution optimizations on page 68

Table I-13: Differences between issue 0406-00 and 0407-00

Change	Location
Added the extensions: • `cl_khr_semaphore` • `cl_khr_external_semaphore` • `cl_khr_external_semaphore_sync_fd` • `cl_khr_integer_dot_product`	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82

Table I-14: Differences between issue 0407-00 and 0408-00

Change	Location
Added the extensions of `cl_khr_external_memory`, `cl_arm_controlled_kernel_termination`, `cl_arm_protected_memory_allocation`, and `cl_khr_external_memory_dma_buf`.	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82
Updated the content of `fno-deferred-compilation`.	10.4 Execution optimizations on page 68

Table I-15: Differences between issue 0408-00 and 0409-00

Change	Location
Added the extensions: `cl_ext_image_from_buffer`, `cl_ext_image_requirements_info`, and `cl_ext_image_tiling_control`. Adjusted the sequence of `cl_ext_cxx_for_opencl` in the list.	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82

Table I-16: Differences between issue 0409-00 and 0410-00

Change	Location
Added the extension of `cl_ext_yuv_images`.	C.1 OpenCL extensions supported by the Mali GPU OpenCL driver on page 82