# ARM® Mali™ GPU OpenCL

**Version 3.3**

## Developer Guide

ARM


ARM 100614_0303_00_en

# ARM® Mali™ GPU OpenCL

## Developer Guide



### Release Information

### Document History

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| A | 12 July 2012 | Confidential | First release |
| D | 07 November 2012 | Confidential | Second release |
| E | 27 February 2013 | Non-Confidential | Third release |
| F | 03 December 2013 | Non-Confidential | Fourth release |
| G | 13 May 2015 | Non-Confidential | First release for r6p0 |
| H | 10 August 2015 | Non-Confidential | First release for r7p0 |
| I | 01 October 2015 | Non-Confidential | First release for r8p0 |
| 0900-00 | 03 December 2015 | Non-Confidential | First release for r9p0 |
| 1000-00 | 21 January 2016 | Non-Confidential | First release for r10p0 |
| 1100-00 | 24 March 2016 | Non-Confidential | First release for r11p0 |
| 0300-00 | 21 April 2016 | Non-Confidential | Changed to version 3.0 |
| 0301-00 | 13 May 2016 | Non-Confidential | First release of version 3.1 |
| 0302-00 | 12 July 2016 | Non-Confidential | First release of version 3.2 |
| 0303-00 | 22 February 2017 | Non-Confidential | First release of version 3.3 |

### Non-Confidential Proprietary Notice

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms. 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.



ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

### Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

### Product Status

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

### Web Address

*http://www.arm.com*

# Contents
# ARM® Mali™ GPU OpenCL Developer Guide

# Preface

This preface introduces the *ARM® Mali™ GPU OpenCL Developer Guide*.

It contains the following:

# About this book

This book describes software development and optimization for OpenCL on Mali™ Midgard and Bifrost GPUs.

## Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

## Intended audience

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

## Using this book

This book is organized into the following chapters:

***Chapter 1 Introduction***
This chapter introduces Mali GPUs, OpenCL, and the Mali GPU OpenCL driver.

***Chapter 2 Parallel Processing Concepts***
This chapter describes the main concepts of parallel processing.

***Chapter 3 OpenCL Concepts***
This chapter describes the OpenCL concepts.

***Chapter 4 Developing an OpenCL Application***
This chapter describes the development stages of an OpenCL application.

***Chapter 5 Execution Stages of an OpenCL Application***
This chapter describes the execution stages of an OpenCL application.

***Chapter 6 Converting Existing Code to OpenCL***
This chapter describes converting existing code to OpenCL.

***Chapter 7 Retuning Existing OpenCL Code***
This chapter describes how to retune existing OpenCL code so you can run it on Mali GPUs.

***Chapter 8 Optimizing OpenCL for Mali GPUs***
This chapter describes the procedure to optimize OpenCL applications for Mali GPUs.

***Chapter 9 OpenCL Optimizations List***
This chapter lists several optimizations to use when writing OpenCL code for Mali GPUs.

***Chapter 10 The kernel auto-vectorizer and unroller***
This chapter describes the kernel auto-vectorizer and unroller.

***Appendix A OpenCL Data Types***
This appendix describes OpenCL data types.

***Appendix B OpenCL Built-in Functions***
This appendix lists the OpenCL built-in functions.

***Appendix C OpenCL Extensions***
This appendix describes the OpenCL extensions that the Mali GPU OpenCL driver supports.

***Appendix D Using OpenCL Extensions***
This appendix provides usage notes on specific OpenCL extensions.

### *Appendix E OpenCL 1.2*

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

### *Appendix F Revisions*

This appendix contains a list of technical changes made between releases and where they are documented in this guide.

### 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.

### Typographic conventions

*italic*

Introduces special terminology, denotes cross-references, and citations.

**bold**

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

*`monospace italic`*

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

**`monospace bold`**

Denotes language keywords when used outside example code.

`<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

Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

## Additional reading

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

**ARM publications**

See *Infocenter*, *http://infocenter.arm.com*, for access to ARM documentation.

**Other publications**

*OpenCL 1.2 Specification*, *www.khronos.org*

# Feedback

## Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

## Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *ARM Mali GPU OpenCL Developer Guide*.
- The number ARM 100614_0303_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

**Note**

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

# Chapter 1
# **Introduction**

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

It contains the following sections:

## 1.1 About ARM® Mali™ GPUs

ARM produces the following families of Mali GPUs:

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.

The arithmetic pipes in Mali Bifrost GPUs are based on quad-style vectorization. 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.

The arithmetic pipes in Mali Midgard GPUs are based on a *Single Instruction Multiple Data* (SIMD) style vectorization so instructions operate on multiple data elements simultaneously. You must explicitly vectorize your shader code for this to work.

Mali GPUs can have one or more shader cores. Each shader core contains one or more arithmetic pipes.

**Mali Bifrost GPUs**

- Mali-G71.

**Mali Midgard GPUs**

Mali Midgard GPUs include the following:

- Mali-T600 series.
- Mali-T720.
- Mali-T760.
- Mali-T820.
- Mali-T830.
- Mali-T860.
- Mali-T880.

**Mali Utgard GPUs**

——— **Note** ———

Mali Utgard GPUs do not support OpenCL.

———————————

## 1.2 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.*

## 1.3 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 driver supports the following versions:

- OpenCL version 1.2, Full Profile.
- Binary-compatibility with OpenCL 1.0 and OpenCL 1.1 applications. This includes compatibility with the APIs deprecated in OpenCL 1.2.

**Note**

The Mali GPU OpenCL driver does not support Mali Utgard GPUs.

# Chapter 2
# Parallel Processing Concepts

This chapter describes the main concepts of parallel processing.

It contains the following sections:

## 2.1 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.

## 2.2 Types of parallelism

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

This section contains the following subsections:



### 2.2.1 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.

### 2.2.2 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.

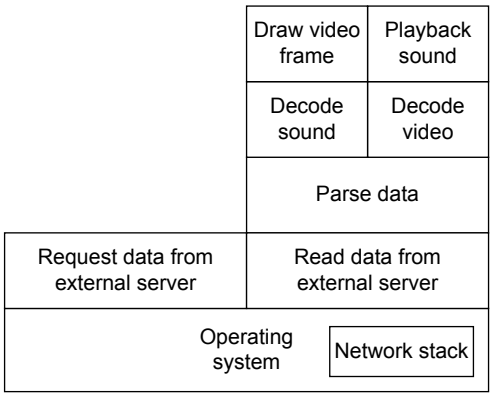


**Figure 2-1 Task parallel processing**

### 2.2.3 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.

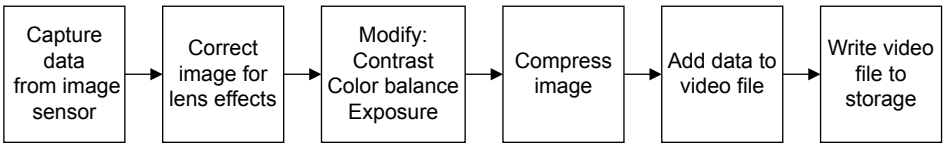


**Figure 2-2 Pipeline processing**

## 2.3 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.

## 2.4 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.

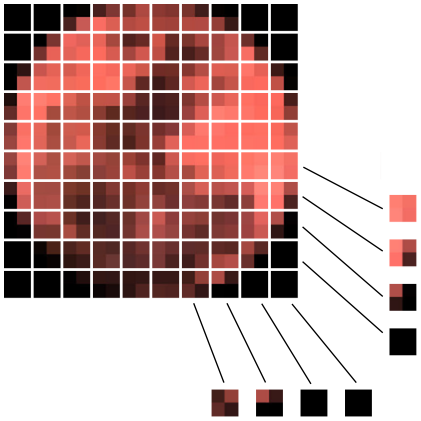

**Figure 2-3 Embarrassingly parallel processing**

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.

## 2.5 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.

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

**Figure 2-4 Formula for Amdahl's law**

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

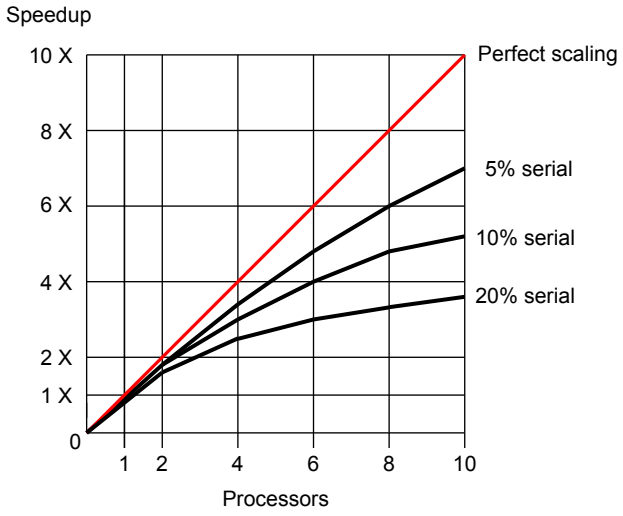


**Figure 2-5 Speedup for an application with different serial components**

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 concepts

*2.4 Embarrassingly parallel applications* on page 2-20.
*9.5 Reducing the effect of serial computations* on page 9-84.

## 2.6 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.

# Chapter 3
# OpenCL Concepts

This chapter describes the OpenCL concepts.

It contains the following sections:

## 3.1 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.

The Midgard OpenCL driver supports OpenCL 1.2, Full Profile.

## 3.2 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.

## 3.3 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.

## 3.4 OpenCL data processing

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

The work-items are organized in an NRange where:

- N is the number of dimensions minus one.
- N can be zero, one, or two.

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

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

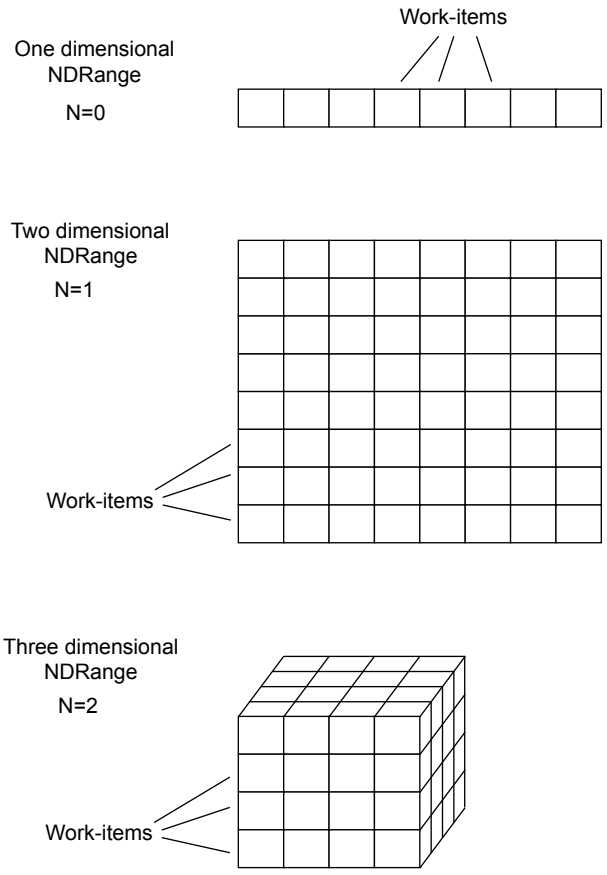


**Figure 3-1 NDRanges and work-items**

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.

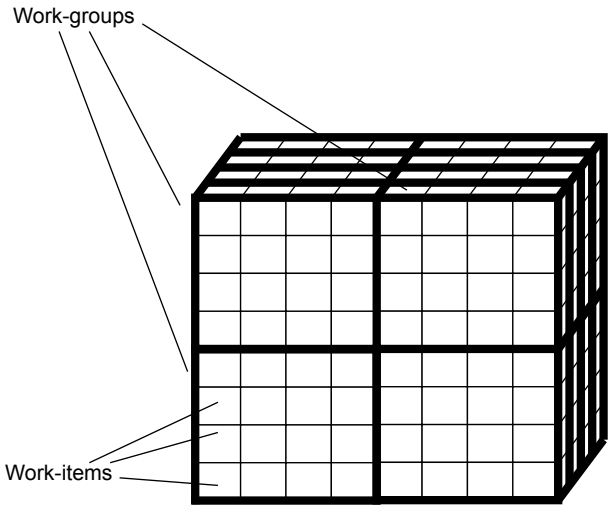


**Figure 3-2 Work-items and work-groups.**

## 3.5 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.

## 3.6 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*.

## 3.7 The OpenCL memory model

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

This section contains the following subsections:



### 3.7.1 OpenCL memory model overview

The following figure shows the OpenCL memory model.

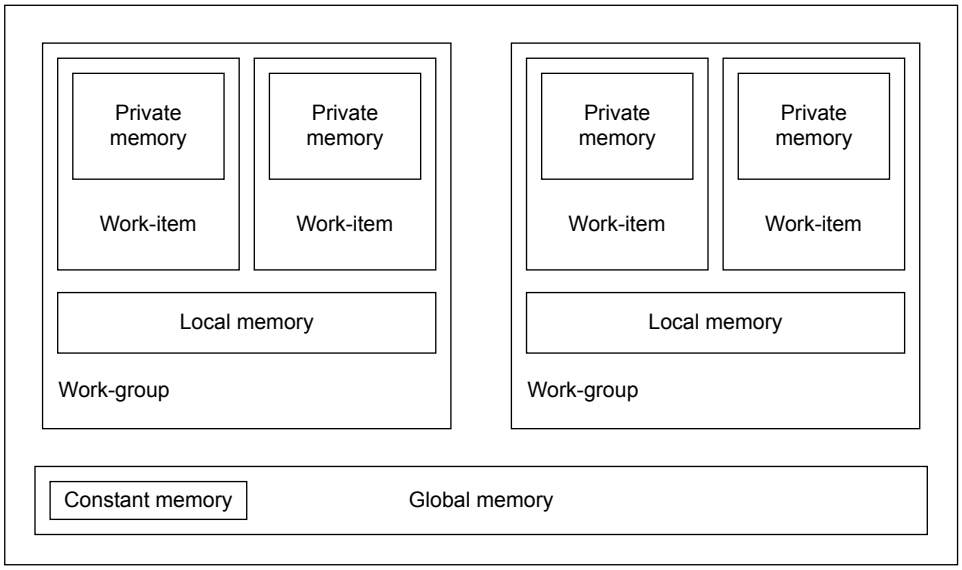


**Figure 3-3 OpenCL memory model**

### 3.7.2 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.

## 3.8 The 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 or private memory typically does not improve 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 concepts

*8.3 Memory allocation* on page 8-71.

## 3.9 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.

# Chapter 4
# Developing an OpenCL Application

This chapter describes the development stages of an OpenCL application.

It contains the following sections:

## 4.1 Software and hardware requirements for Mali GPU OpenCL development

Implementations of OpenCL are available for several operating systems. You can develop on other hardware platforms with implementations of OpenCL.

To develop OpenCL applications for Mali GPUs, you require:

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

**Note**

The Mali GPU must be a Mali Midgard or Bifrost GPU.

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

### Related concepts

*1.1 About ARM® Mali™ GPUs* on page 1-12.

## 4.2 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 concepts

*6.2 Analyzing code for parallelization* on page 6-53.

# Chapter 5
# 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 lesson in OpenCL.

It contains the following sections:

## 5.1 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.

This section contains the following subsections:

- *5.1.1 Platform setup* on page 5-39.
- *5.1.2 Runtime setup* on page 5-39.

### 5.1.1 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 concepts

*5.2 Finding the available compute devices* on page 5-41.

*5.3 Initializing and creating OpenCL contexts* on page 5-42.

### 5.1.2 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()` or `clCreateProgramWithBinary()`:

     **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 concepts

*5.4 Creating a command queue* on page 5-43.
*5.5 Creating OpenCL program objects* on page 5-44.
*5.7 Creating kernel and memory objects* on page 5-46.
*5.8 Executing the kernel* on page 5-47.
*5.9 Reading the results* on page 5-49.
*5.10 Cleaning up unused objects* on page 5-50.

### Related references

*5.6 Building a program executable* on page 5-45.

## 5.2 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.

If your Mali GPU has two shader core groups, the OpenCL driver treats each core group as a separate OpenCL device, each with its own `cl_device_id`. This means you can form separate queues and execute them independently of each other. Alternatively, you can choose one core group for OpenCL and leave the other core group for other GPU tasks.

The OpenCL driver returns the device arrays in core group ID order.

**Note**

Some of the Midgard devices are asymmetric. One core group might contain four shader cores while another core group might contain two shader cores.

## 5.3 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 on 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.

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);
}
```

## 5.4 Creating a command queue

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

OpenCL 1.2 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.

## 5.5 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.

## 5.6 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;
}
```

## 5.7 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.

This section contains the following subsections:



### 5.7.1 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;
}
```

### 5.7.2 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 2D or 3D 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.

## 5.8 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.

This section contains the following subsections:



### 5.8.1 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.

### 5.8.2 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 large if you want to ensure high performance. Typically, the number is several thousand, but the ideal number depends on the number of shader cores in your device.

### 5.8.3 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 is 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 total work-group size, call `clGetKernelWorkGroupInfo()` with `CL_KERNEL_WORK_GROUP_SIZE`. If the maximum work-group size for a kernel is lower than 128, performance is reduced. If this is the case, try simplifying the kernel.

The work-group size for each dimension must divide evenly into the total data-size for that dimension. This means that the x size of the work-group must divide evenly into the x size of the total data. If this requirement means padding the work-group with extra work-items, ensure the additional work-items return immediately and do no work.

### 5.8.4 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;
}
```

### 5.8.5 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.

## 5.9 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,
               (sizeof(buffer_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.

## 5.10 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 for all OpenCL objects reach zero when your application no longer requires them. You can obtain the reference count by querying the object. For example, by calling `clGetMemObjectInfo()`.

# Chapter 6
# Converting Existing Code to OpenCL

This chapter describes converting existing code to OpenCL.

It contains the following sections:

## 6.1 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 to profile your application.

### Related information

*http://malideveloper.arm.com.*

## 6.2 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.

This section contains the following subsections:



### 6.2.1 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 concepts


*6.3.1 Use the global ID instead of the loop counter* on page 6-55.

*6.3.2 Compute values in a loop with a formula instead of using counters* on page 6-55.

*6.3.3 Compute values per frame* on page 6-56.

*6.3.4 Perform computations with dependencies in multiple-passes* on page 6-57.

*6.3.5 Pre-compute values to remove dependencies* on page 6-57.

*6.4 Using parallel processing with non-parallelizable code* on page 6-59.


### 6.2.2 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.

### 6.2.3 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.

### 6.2.4 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 concepts

*6.3 Parallel processing techniques in OpenCL* on page 6-55.
*6.3.1 Use the global ID instead of the loop counter* on page 6-55.
*6.4 Using parallel processing with non-parallelizable code* on page 6-59.

## 6.3 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.

This section contains the following subsections:



### 6.3.1 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.

#### Simplified loop example

Example showing 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;
}
```

### 6.3.2 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.

### 6.3.3 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 6-1 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.

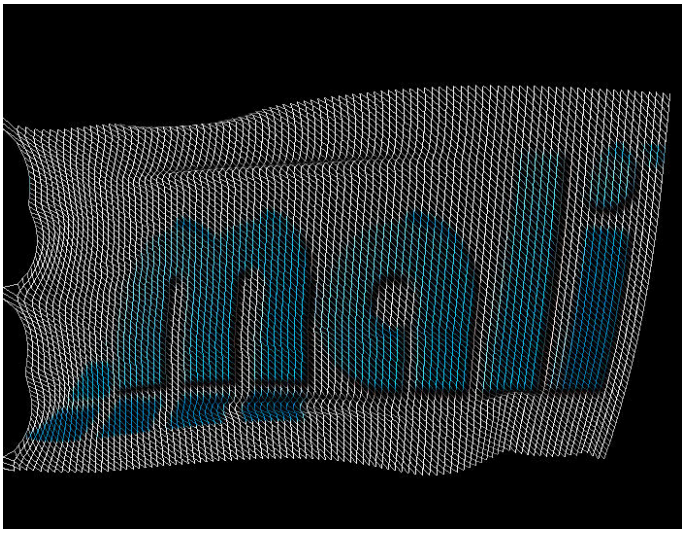

**Figure 6-2 Flag simulation grid**

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.

### 6.3.4 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 concepts

*6.3.3 Compute values per frame* on page 6-56.

### 6.3.5 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.

### 6.3.6 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.

### 6.3.7 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.

## 6.4 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 concepts

*6.5.2 Use concurrent data structures* on page 6-60.

## 6.5 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.

This section contains the following subsections:



### 6.5.1 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.

**Work-items**
: Each work-group is divided into work-items.

#### Related references

*Chapter 3 OpenCL Concepts* on page 3-23.

### 6.5.2 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.

### 6.5.3 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.

### 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 8 samples and process 8 of them at a time with vector instructions.

### 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.

### 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.

# Chapter 7
# Retuning Existing OpenCL Code

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

It contains the following sections:

## 7.1 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.

This section contains the following subsection:

- *7.1.1 Converting the existing OpenCL code for Mali™ GPUs* on page 7-63.

### 7.1.1 Converting the existing OpenCL code for Mali™ 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**

For best performance on Mali Midgard GPUs, you must vectorize your code.

You are not required to vectorize code for Mali Bifrost GPUs, but vectorizing your code does not reduce its performance.

## 7.2 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.

This section contains the following subsections:



### 7.2.1 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.

Desktop GPUs have shader architectures that put threads into *thread groups*. These are known as *warps* or *wavefronts*. This mechanism means that the threads must operate in lock-step. If they do not, for example, if there is a branch in the code and threads take different directions, the threads are said to be *divergent*.

When threads are divergent, the two operations are split and must be computed twice. This halves the processing speed.

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.

### 7.2.2 About Mali GPU architectures

Mali GPUs use a SIMD architecture. Instructions operate on multiple data elements simultaneously.

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

The Mali GPUs contain 1 to 16 identical shader cores. Each shader core supports up to 384 concurrently executing threads.

Each shader core contains:

- One to three arithmetic pipelines.
- One load-store pipeline.
- One texture pipeline.

**Note**

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

The Mali GPUs use a VLIW (*Very Long Instruction Word*) architecture. Each instruction word contains multiple operations. The Mali GPUs also use SIMD, so that most arithmetic instructions operate on multiple data elements simultaneously.

Each thread uses only one of the arithmetic or load-store execution pipes at any point in time. Two instructions from the same thread execute in sequence.

### 7.2.3 Programming OpenCL for Mali GPUs

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

On a Mali GPU:

- The global and local OpenCL address spaces are mapped to the same physical memory and are accelerated by L1 and L2 caches. This means that you are not required to use explicit data copies or implement the associated barrier synchronization.
- All threads have individual program counters. This means that branch divergence is not a major problem. Branch divergence is a major issue for warp or wavefront-based architectures.

  **Note**

  In OpenCL, each work-item typically maps to a single thread on a Mali GPU.

- Use the kernel auto-vectorizer.

  **Note**

  This feature is not available for Bifrost GPUs.

#### Related references

*Chapter 10 The kernel auto-vectorizer and unroller* on page 10-86.

## 7.3 Procedure for 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.

This section contains the following subsections:



### 7.3.1 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.

### 7.3.2 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*.

These optimizations are the following:

**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 very 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.

———————————

**Use of scalars**

Mali Bifrost GPUs use scalars.

Mali Midgard GPUs use scalars and 128-bit vectors.

**Modifications for memory bank conflicts**

Some GPUs include per-warp memory banks. If the code includes optimizations to avoid conflicts in these memory banks, remove them.

**Optimizations for divergent threads, warps, or wavefronts**

Some GPU architectures group work-items together into what are called warps or wavefronts. All the work-items in a warp must proceed in lock-step together in these architectures and this means branches can perform badly.

Threads on Mali Midgard GPUs are independent and can diverge without any performance impact. If your code contains optimizations or workarounds for divergent threads in warps or wavefronts, remove them.

——— **Note** ———

Mali Midgard GPUs do not use warps or wavefronts.

———————————

——— **Note** ———

This optimization only applies to Mali Midgard GPUs.

———————————

### 7.3.3 Optimize your OpenCL code for Mali GPUs

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

#### Related references

*Chapter 8 Optimizing OpenCL for Mali GPUs* on page 8-68.

# Chapter 8
# Optimizing OpenCL for Mali GPUs

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

It contains the following sections:

## 8.1 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 run times 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 run-time 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 pipelines.

  ——— **Note** ———

  Compiler output using `-v` is not available for Bifrost GPUs.

  ———————————

**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. Consider how this code can be rewritten.

## 8.2 Load balancing between control threads and OpenCL threads

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

This section contains the following subsections:



### 8.2.1 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.

### 8.2.2 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.

## 8.3 Memory allocation

You can optimize memory allocation by using the correct commands.

This section contains the following subsections:



### 8.3.1 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 8-1 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 contents of the `host_ptr` argument into memory allocated by the driver. |
| `CL_MEM_USE_HOST_PTR` | Copies the content of 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.<br>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.

### 8.3.2 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.

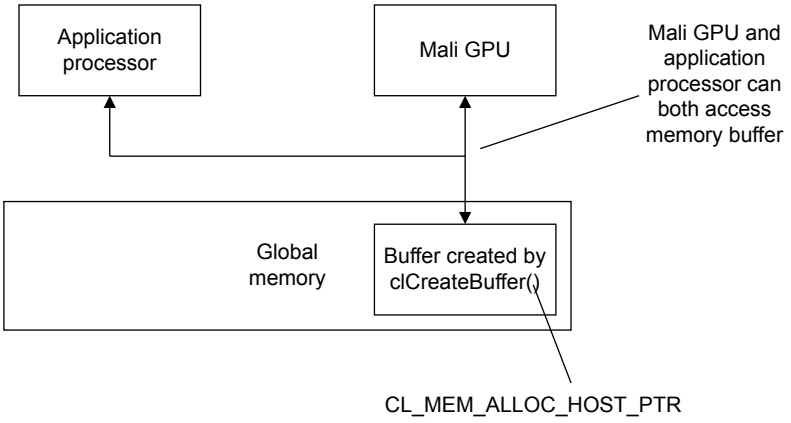


**Figure 8-1 Memory buffer created by clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)**

**Note**

- 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.

### 8.3.3 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.

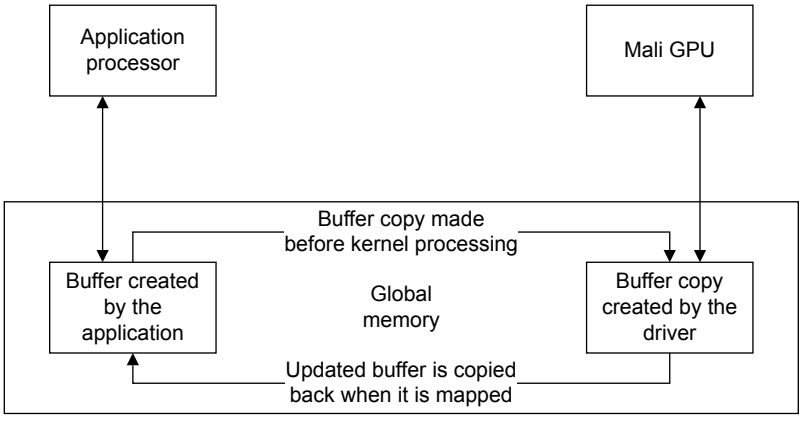


**Figure 8-2 Memory buffer created by clCreateBuffer(CL_MEM_USE_HOST_PTR)**

If your application can use an alternative allocation type, it can avoid these computationally expensive copy operations. For example, `CL_MEM_ALLOC_HOST_PTR`.

### 8.3.4 Do not allocate memory buffers created with malloc() for OpenCL applications

The Mali GPU cannot access the memory buffers created by `malloc()` because they are not mapped into the address space of the Mali GPU.

The inaccessible memory buffer is shown in the following figure.

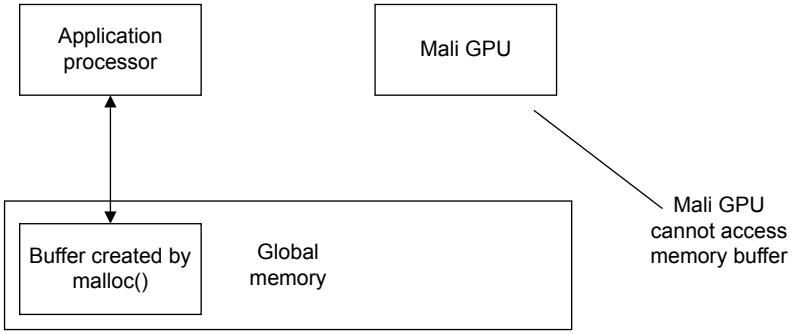


**Figure 8-3 Memory buffer created by malloc()**

### 8.3.5 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.

### 8.3.6 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()`.

# Chapter 9
# OpenCL Optimizations List

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

It contains the following sections:

## 9.1 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. For example, in a four shader-core Mali GPU system, enqueue 1024 or more 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.

## Related references

*Chapter 6 Converting Existing Code to OpenCL* on page 6-51.

## 9.2 Kernel optimizations

ARM recommends some kernel optimizations such as experimenting with the work-group size and shape, minimizing thread convergence, and using a workgroup size of 128 or higher.

**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)... );
```

**For best performance, use a workgroup size that is between 4 and 64 inclusive, and a multiple of 4**
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 the global work size is not divisible by 4, use padding at the edges or use a non-uniform workgroup size**
To ensure the global work size is divisible by 4, add a few more dummy threads.
Alternatively you can let the application processor compute the edges.
You can use a non-uniform workgroup size, but this does not guarantee better performance than the other options.

**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. The driver usually selects the work group size as 64.

——— **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()`.

**Use a work-group size of 128 or 256 if possible**
The maximum work-group size is typically 256, but this is not possible for all kernels and the driver suggests another size. A work-group size of 64 is the smallest size guaranteed to be available for all kernels.

If possible, use a work-group size of 128 or 256. These make optimal use of the Mali GPU hardware. If the maximum work-group size is below 128, your kernel might be too complex.

**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.

**Check for inter-thread communication**

Use `clGetKernelWorkGroupInfo()` to check if the device can execute a kernel that requires a minimum of inter-thread communication. If the device cannot execute the kernel, the algorithm must be implemented as a multi-pass algorithm. This involves enqueuing multiple kernels.

**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.

———————————

**Minimize thread divergence**

It is beneficial to minimize thread divergence, this improves data locality for caching.
To minimize thread divergence, avoid the following:

- Variable length loops.
- Asymmetric conditional blocks.

**Ensure that the kernels exit at the same time**

Branches are computationally cheap on Mali GPUs. This means you can use loops in kernels without any performance impact.

Your kernels can include different code segments but try to ensure the kernels exit at the same time.

A workaround to this is to use a *bucket algorithm*.

**Make your kernel code as simple as possible**

This assists the auto-vectorization process.

Using loops and branches might make auto-vectorization more difficult.

**Use vector operations in kernel code**

Use vector operations in kernel code to help the compiler to map them to vector instructions.

**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 off-line compiler to produce statistics for your kernels to assist with this.

**Optimize the memory access pattern of your application**

## 9.3 Code optimizations

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

**Vectorize your code**

Mali Midgard GPUs perform computation with vectors. These enable you to perform multiple operations per instruction.

Vectorizing your code makes the best use of the Mali GPU hardware so ensure that you vectorize your code for maximum performance.

Mali Midgard GPUs contain 128-bit wide vector registers.

——— **Note** ———

The Midgard compiler can auto-vectorize some scalar code.

———————————

**Vectorize incrementally**

Vectorize in incremental steps. For example, start processing one pixel at a time, then two, then four.

**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.

**Avoid processing single values**

Avoid writing kernels that operate on single bytes or other small values. Write kernels that work on vectors.

**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 the precise versions of built-in functions**

Use the precise versions of built-in functions.

Usually, the `half_` or `native_` versions of built-in functions provide no extra performance. The following functions are exceptions:

- `native_sin()`.
- `native_cos()`.
- `native_tan()`.
- `native_divide()`.
- `native_exp()`.
- `native_sqrt()`.
- `half_sqrt()`.

**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**

Avoid writing kernels that use many live variables. Using too many live variables can affect performance and limits the maximum work-group size.

**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 `mali_clcc` offline compiler built from the DDK tree to produce statistics for your kernels and check the ratio between arithmetic instructions and loads.

**Note**

Statistics from the offline compiler using `-v` are not available for Bifrost GPUs.

**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 references

*Chapter 10 The kernel auto-vectorizer and unroller* on page 10-86.
*Appendix B OpenCL Built-in Functions* on page Appx-B-96.
*B.3 half_ and native_ math functions* on page Appx-B-100.

## 9.4 Execution optimizations

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

ARM also recommends that:

- If you are building from source, cache binaries on the storage device.
- If you know the kernels that you are using when your application initializes, call `clCreateKernelsInProgram()` to initiate the final compile as soon as possible.

  Doing this ensures that when you use kernels in the future, they start faster because the existing finalized binary is used.
- If you use callbacks to prompt the processor to continue processing data resulting from the execution of a kernel, ensure that the callbacks are set before you flush the queue.

  If you do not do this, the callbacks might occur at the end of a larger batch of work, later than they might have based on actual completion of work.

## 9.5 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.

## 9.6 Mali™ Bifrost GPU specific optimizations

ARM recommends some Mali Bifrost GPU specific optimizations.

**Note**

Only use these optimizations if you are specifically targeting a Mali Bifrost GPU.

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

In Mali Bifrost GPUs, groups of four adjacent threads are arranged together as quads. If your shaders contain branches, such as if statements or loops, the branches in quads can go different ways. This reduces performance because the arithmetic unit cannot execute both sides of the branch at the same time.

Try to ensure that the threads in quads all branch the same way.

**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 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 pipeline 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 `id`s load data from different cache lines.

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

**Note**

One cache line is 512-bits.

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

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

**Try to get a good balance of usage of the arithmetic pipe, Load-store unit, and texture unit**

# Chapter 10
# The kernel auto-vectorizer and unroller

This chapter describes the kernel auto-vectorizer and unroller.

**Note**

- The kernel auto-vectorizer and unroller are enabled by default for Midgard GPUs.
- This feature is not available for Bifrost GPUs.

It contains the following sections:

## 10.1 About the kernel auto-vectorizer and unroller

The OpenCL compiler includes a kernel auto-vectorizer and a kernel unroller:

- 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.

These operations can provide substantial performance gains if they are possible.

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

**Table 10-1 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.

## 10.2 Kernel auto-vectorizer options

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

This section contains the following subsections:



### 10.2.1 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)...

### 10.2.2 Kernel auto-vectorizer command examples

Table showing examples of auto-vectorizer commands.

**Table 10-2 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 |

## 10.3 Kernel unroller options

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

This section contains the following subsections:



### 10.3.1 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.

### 10.3.2 Kernel unroller command examples

Table showing examples of kernel unroller commands.

**Table 10-3 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 |

## 10.4 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.

It contains the following sections:

## 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.

Vector sizes of 128 bits are optimal. Vector sizes greater than 128 bits are broken into 128-bit parts and operated on separately. For example, adding two 256-bit vectors takes twice as long as adding two 128-bit vectors. You can use vector sizes less than 128 bits without issue.

The disadvantage of using vectors greater than 128 bits is that they can increase code size. Increased code size uses more instruction cache space and this can reduce performance.

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.

This section contains the following subsections:



### 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 |
| `float` | `cl_float` | 32-bit float |
| `half` | `cl_half` | 16-bit float, for storage only |
| `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 |
| `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 |

**Table A-4 Reserved data types (continued)**

| OpenCL Type | Description |
|---|---|
| `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`*n*x*m* | *n***m* matrix of 32-bit floats |
| `double`*n*x*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.
- The `double` and `double` vector data types can be used with the `cl_khr_fp64` extension on Mali Midgard GPUs. This extension is not available on Bifrost GPUs.

# Appendix B
# OpenCL Built-in Functions

This appendix lists the OpenCL built-in functions.

It contains the following sections:

## B.1 Work-item functions

List of work-item functions.

**Table B-1 Work-item functions**

| Function |
|---|
| `get_work_dim()` |
| `get_global_size()` |
| `get_global_id()` |
| `get_local_size()` |
| `get_local_id()` |
| `get_num_groups()` |
| `get_group_id()` |
| `get_global_offset()` |

## B.2 Math functions

List of math functions.

**Table B-2 Math functions**

| Function | Function | Function |
|---|---|---|
| fabs() | acos() | acosh() |
| ceil() | acospi() | asinh() |
| fdim() | asin() | atanh() |
| fmax() | asinpi() | copysign() |
| fmin() | atan() | erfc() |
| mad() | atan2() | erf() |
| maxmag() | atanpi() | fmod() |
| minmag() | atan2pi() | fract() |
| rint() | cbrt() | frexp() |
| round() | cos() | hypot() |
| trunc() | cosh() | ilogb() |
| - | cospi() | ldexp() |
| - | exp() | lgamma() |
| - | exp2() | lgamma_r() |
| - | exp10() | log() |
| - | expml() | log10() |
| - | floor() | log1p() |
| - | fma() | logb() |
| - | log2() | modf() |
| - | pow() | nan() |
| - | pown() | nextafter() |
| - | powr() | remainder() |
| - | rsqrt() | remquo() |
| - | sin() | rootn() |
| - | sincos() | sinh() |
| - | sinpi() | tan() |
| - | sqrt() | tanh() |
| - | - | tanpi() |
| - | - | tgamma() |

**Note**

The *ulp* error of `lgamma()` is *16ulp* unless the correctly rounded result is less than one. If the correctly rounded result is less than one, `lgamma()` is also less than one. The error of `lgamma_r()` is same as `lgamma()`.

`lgamma()` is logarithmic, so if the correctly rounded result is small, the precision of the result is not important.

## B.3 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-3 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()`.

### Related references

*B.2 Math functions* on page Appx-B-98.

## B.4 Integer functions

List of integer functions.

**Table B-4 Integer functions**

| Function | Notes |
|---|---|
| `abs()` | |
| `abs_diff()` | |
| `add_sat()` | |
| `hadd()` | |
| `rhadd()` | |
| `clz()` | |
| `max()` | |
| `min()` | |
| `sub_sat()` | |
| `mad24()` | Identical to 32-bit multiply accumulate |
| `mul24()` | Identical to 32-bit multiplies |
| `clamp()` | |
| `mad_hi()` | |
| `mul_hi()` | |
| `mad_sat()` | |
| `rotate()` | |
| `upsample()` | |
| `popcount()` | |

## B.5 Common functions

List of common functions.

**Table B-5 Common functions**

| Function |
|---|
| max() |
| min() |
| step() |
| clamp() |
| degrees() |
| mix() |
| radians() |
| smoothstep() |
| sign() |

## B.6 Geometric functions

List of geometric functions.

**Table B-6 Geometric functions**

| Function |
|---|
| dot() |
| normalize() |
| fast_distance() |
| fast_length() |
| fast_normalize() |
| cross() |
| distance() |
| length() |

## B.7 Relational functions

List of relational functions.

**Table B-7 Relational functions**

| Function |
|---|
| any() |
| all() |
| bitselect() |
| select() |
| isequal() |
| isnotequal() |
| isgreater() |
| isgreaterequal() |
| isless() |
| islessequal() |
| islessgreater() |
| isfinite() |
| isinf() |
| isnan() |
| isnormal() |
| isordered() |
| isunordered() |
| signbit() |

## B.8 Vector data load and store functions

List of vector data load and store functions.

**Table B-8 Vector data load and store functions**

| Function |
|---|
| `vload()` |
| `vstore()` |
| `vload_half()` |
| `vstore_half()` |
| `vloada_half()` |
| `vstorea_half()` |

## B.9 Synchronization

List of synchronization functions.

The `barrier()` function has no speed rating because it must wait for multiple work-items to complete. The time this takes 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.

**Table B-9 Synchronization functions**

| Function |
|---|
| `barrier()` |
| `mem_fence()` |
| `read_mem_fence()` |
| `write_mem_fence()` |

**Note**

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

## B.10 Asynchronous copy functions

List of asynchronous copy functions. These have no speed rating because the copy speed depends on the size of the data copied.

**Table B-10 Asynchronous copy functions**

| Function |
|---|
| `async_work_group_copy()` |
| `async_work_group_strided_copy()` |
| `wait_group_events()` |
| `prefetch()` |

## B.11 Atomic functions

List of atomic functions.

**Table B-11 Atomic functions**

| Function |
|---|
| `atomic_add()` |
| `atomic_sub()` |
| `atomic_xchg()` |
| `atomic_inc()` |
| `atomic_dec()` |
| `atomic_cmpxchg()` |
| `atomic_min()` |
| `atomic_max()` |
| `atomic_and()` |
| `atomic_or()` |
| `atomic_xor()` |

## B.12 Miscellaneous vector functions

List of miscellaneous vector functions.

**Table B-12 Miscellaneous vector functions**

| Function |
|---|
| `vec_step()` |
| `shuffle()` |
| `shuffle2()` |

## B.13 Image read and write functions

List of image read and write functions.

**Table B-13 Image read and write functions**

| Function |
|---|
| `read_imagef()` |
| `read_imagei()` |
| `read_imageui()` |
| `write_imagef()` |
| `write_imagei()` |
| `write_imageui()` |
| `get_image_width()` |
| `get_image_height()` |
| `get_image_depth()` |
| `get_image_channel_data_type()` |
| `get_image_channel_order()` |
| `get_image_dim()` |

# Appendix C
# OpenCL Extensions

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

It contains the following section:

## 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_local_int32_base_atomics`.
- `cl_khr_local_int32_extended_atomics`.
- `cl_khr_global_int32_base_atomics`.
- `cl_khr_global_int32_extended_atomics`.
- `cl_khr_byte_addressable_store`.
- `cl_khr_int64_base_atomics`.
- `cl_khr_int64_extended_atomics`.
- `cl_khr_icd`.
- `cl_khr_egl_image`.
- `cl_khr_fp16`.
- `cl_khr_fp64`.
- `cl_khr_3d_image_writes`.
- `cl_khr_image2d_from_buffer`.

**Note**

The `cl_khr_fp64` extension only works on Mali Midgard GPUs.

The Mali GPU OpenCL driver on Midgard GPUs also supports the following optional ARM extensions:

- `cl_arm_core_id`.
- `cl_arm_printf`.
- `cl_arm_thread_limit_hint`.
- `cl_arm_import_memory`.
- `cl_arm_import_memory_dma_buf`.
- `cl_arm_non_uniform_work_group_size`.

The Mali GPU OpenCL driver on Bifrost GPUs also supports the optional ARM extension, `cl_arm_shared_virtual_memory`.

### Related information

*The Khronos Group.*

# Appendix D
# Using OpenCL Extensions

This appendix provides usage notes on specific OpenCL extensions.

It contains the following sections:

## D.1 Inter-operation with EGL

You can use extensions for OpenCL inter-operation with EGL.

The OpenCL extension `cl_khr_egl_image` enables you to share OpenCL images or buffers with EGL.

**Note**

If you are using `cl_khr_egl_image`, you can achieve higher performance if you create read-only images with `clCreateFromEGLImageKHR()`.

## 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.

This section contains the following subsections:

- *D.2.1 About the cl_arm_printf extension* on page Appx-D-115.
- *D.2.2 cl_arm_printf example* on page Appx-D-115.

### 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;
}
```

# Appendix E
# **OpenCL 1.2**

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

It contains the following sections:

## 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**

In OpenCL 1.2, for offline compilation 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()**

**Note**

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

## E.4 Functions deprecated in OpenCL 1.2

These functions are deprecated in OpenCL 1.2 but are still available in the Mali OpenCL driver:

**Table E-1 Functions deprecated in OpenCL 1.2**

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

## E.5 OpenCL 1.2 extensions

OpenCL 1.2 adds the `cl_arm_shared_virtual_memory` extension.

This section contains the following subsection:



### E.5.1 The cl_arm_shared_virtual_memory extension

The OpenCL `cl_arm_shared_virtual_memory` extension provides support for most of the OpenCL 2.0 *Shared Virtual Memory* (SVM) features in OpenCL 1.2.

SVM provides benefits the following benefits compared to OpenCL buffer memory objects:

- Shared virtual addresses between the application processor and the Mali GPU enable them to directly share complex data structures that contain pointers. For example, linked list and tree structures.
- Your code can make fewer OpenCL 1.2 API calls.
- Your code does not require systematic cache maintenance overhead with fine-grained support.

Platforms that support full coherency between a Mali Bifrost GPU and an application processor can use fine-grained SVM allocations and atomic operations. For example, platforms using an ARM CoreLink™ CCI-550 interconnect.

For more information, see the `cl_arm_shared_virtual_memory` extension specification and the OpenCL 2.0 specification.

**Note**

This extension is only available for Mali Bifrost GPUs.

#### Related information

*http://www.khronos.org.*

# Appendix F
# **Revisions**

This appendix contains a list of technical changes made between releases and where they are documented in this guide.

It contains the following section:

- *F.1 Revisions* on page Appx-F-124.

## F.1 Revisions

This describes the technical changes between released issues of this guide.

**Table F-1 Issue 0302_00**

| Change | Location | Affects |
|---|---|---|
| Addition of various optimizations | *Chapter 9 OpenCL Optimizations List* on page 9-74 | All Mali GPUs |
| Addition of Mali Bifrost GPU specific optimizations section | *9.6 Mali™ Bifrost GPU specific optimizations* on page 9-85 | Mali Bifrost GPUs |
| Addition of extension `cl_khr_image2d_from_buffer` | *C.1 OpenCL extensions supported by the Mali™ GPU OpenCL driver* on page Appx-C-112 | All Mali GPUs |

**Table F-2 Issue 0303_00**

| Change | Location | Affects |
|---|---|---|
| Removed extensions:`cl_khr_gl_sharing`<br>`cl_khr_egl_event`<br>`egl_khr_cl_event` | *C.1 OpenCL extensions supported by the Mali™ GPU OpenCL driver* on page Appx-C-112 | All Mali GPUs |
| Removed section on OpenCL inter-operation with OpenGL ES | *D.1 Inter-operation with EGL* on page Appx-D-114 | All Mali GPUs |
| Removed functions:`clCreateFromGLTexture2D()`<br>`clCreateFromGLTexture3D()` | *E.4 Functions deprecated in OpenCL 1.2* on page Appx-E-121 | All Mali GPUs |