ARM® Mali™-T600 Series GPU OpenCL

Version 2.0

Developer Guide

ARM®

DUI0538F (ID012914)

ARM Mali-T600 Series GPU OpenCL
Developer Guide

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
12 July 2012	A	Confidential	First release
07 November 2012	D	Confidential	Second release
27 February 2013	E	Non-confidential	Third release
03 December 2013	F	Non-confidential	Fourth release

Proprietary Notice

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.

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM Mali-T600 Series GPU OpenCL Developer Guide

Preface

This preface introduces the *Mali-T600 Series GPU OpenCL Developer Guide*. It contains the following sections:

About this book

This book is for Mali-T600 Series GPU OpenCL.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this book, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This guide is written for software developers with experience in C or C-like languages who want to develop OpenCL applications for Mali-T600 Series GPUs.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to OpenCL and the Mali OpenCL SDK.

Chapter 2 *Parallel Processing Concepts*

Read this for an introduction to parallel processing concepts.

Chapter 3 *OpenCL Concepts*

Read this for a description of the OpenCL concepts.

Chapter 4 *Developing an OpenCL Application*

Read this for a description of the development stages of an OpenCL application.

Chapter 5 *Execution Stages of an OpenCL Application*

Read this for a description of the execution stages of an OpenCL application.

Chapter 6 *Converting Existing Code to OpenCL*

Read this for a description of how to convert existing code to OpenCL.

Chapter 7 *Retuning Existing OpenCL Code for Mali GPUs*

Read this for a description of how to retune existing OpenCL code for the Mali-T600 Series GPUs.

Chapter 8 *Optimizing OpenCL for Mali GPUs*

Read this for a description of how to optimize OpenCL for the Mali-T600 Series GPUs.

Chapter 9 *OpenCL Optimizations List*

Read this for a list of optimizations for OpenCL on the Mali-T600 Series GPUs.

Chapter 10 *The Mali OpenCL SDK*

Read this for an introduction to the Mali OpenCL SDK.

Appendix A *OpenCL Data Types*

Read this for a list of the data types available.

Appendix B *OpenCL Built-in Functions*

Read this for a list of the OpenCL built-in functions implemented in the Mali-T600 Series GPU Linux OpenCL driver.

Appendix C *OpenCL Extensions*

Read this for a list of extensions the Mali-T600 Series GPU Linux OpenCL driver supports.

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 *ARM Glossary*, `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Typographical conventions

This book uses the following typographical 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.
`mono`space	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 >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`

Additional reading

This section lists publications by ARM and by third parties.

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

Other publications

This section lists relevant documents published by third parties:

- *OpenCL 1.1 Specification*, `www.khronos.org`

Feedback

ARM welcomes feedback on this product and its documentation.

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.
- The number, DUI0538F.
- The page numbers to which your comments apply.
- 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 GPU compute, OpenCL, the Mali-T600 Series GPU Linux OpenCL driver, and the Mali OpenCL SDK. It contains the following sections:

1.1 About GPU compute

GPU compute, or *General Purpose computing on Graphics Processing Units* (GPGPU), is the practice of using the parallel computing power of a GPU for tasks other than 3D graphics rendering.

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

GPUs are designed to execute many threads at the same time. They run compute intensive data processing tasks in parallel that contain relatively little control code. GPUs typically contain many more processing elements than application processors so can compute at a much higher rate than application processors.

OpenCL is the first open standard language to enable developers to run general purpose computing tasks on GPUs, application processors, and other types of processors.

1.2 About OpenCL

The *Open Computing Language* (OpenCL) is an open standard for writing applications to run on heterogeneous multi-processor systems. OpenCL provides a single development environment for applications that can run on different processors.

OpenCL includes a platform-independent C99-based language for writing functions called kernels that execute on OpenCL devices, and APIs that define and control the platforms.

OpenCL enables you to execute some applications faster by moving intensive data processing routines to the GPU instead of the application processor.

OpenCL makes multiprocessor applications easier to write because it manages the execution of your application across multiple application processors and GPUs.

The OpenCL language includes vectors and built in functions that enable you to easily utilize the features of accelerators.

OpenCL is an open standard developed by the *Khronos Group*, http://www.khronos.org.

1.3 About the Mali-T600 Series GPU Linux OpenCL driver

The Mali-T600 Series GPU Linux OpenCL driver is the implementation of OpenCL for the Mali-T600 Series GPUs. In this document, it is known as the *Mali OpenCL driver*.

The Mali OpenCL driver:

- Supports OpenCL version 1.1, Full Profile.
- Is binary-compatible with OpenCL 1.0 applications. This includes compatibility with the APIs deprecated in OpenCL 1.1.

Note

The Mali OpenCL driver is for the Mali-T600 Series GPUs. It does not support the Mali-300, Mali-400, or Mali-450 GPUs.

1.4 About the Mali OpenCL SDK

The Mali OpenCL SDK contains code examples and tutorials to help you understand OpenCL development.

See Chapter 10 *The Mali OpenCL SDK*.

Chapter 2
Parallel Processing Concepts

Parallel processing is the processing of computations on multiple processors simultaneously. OpenCL enables applications to use hardware resources such as GPUs to accelerate computations with parallel processing.

This chapter introduces the main concepts of parallel processing. It contains the following sections:

2.1 Types of parallelism

There are the following types of parallelism:

Data parallelism

In a data-parallel application, data is divided into data elements that can be processed in parallel. Multiple data elements are read and processed simultaneously by different processors.

The data being processed must be in data structures that can be read and written 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 sort of parallelism is very fine grained and there can be hundreds or thousands of threads active simultaneously.

OpenCL is primarily used for data parallel processing.

Task parallelism

Task parallelism is where the application is broken up into tasks and these tasks are executed in parallel. Task parallelism is also known as functional parallelism.

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

- Run a network stack that performs communication.
- Request data from external server.
- Read data from external server.
- Parse data.
- Decode video data.
- Decode audio data.
- Draw video frames.
- Play audio data.

Figure 2-1 shows parts of an application and operating system that operate simultaneously when playing an on-line video.

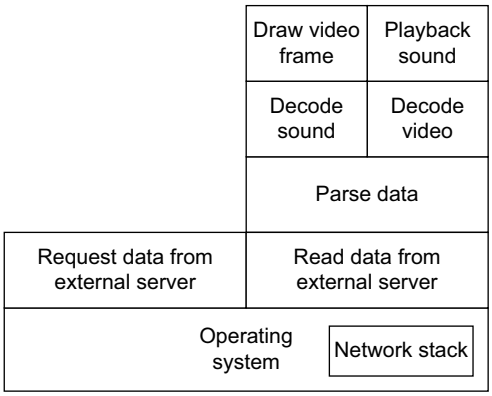

Figure 2-1 Task parallel processing

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 perform the following 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 video file.
6. Write the video file to storage.

These stages must be performed in order but they can be all be operating on data from a different video frame at the same time.

Figure 2-2 shows parts of an application that can operate simultaneously as a pipeline playing a video from the internet.

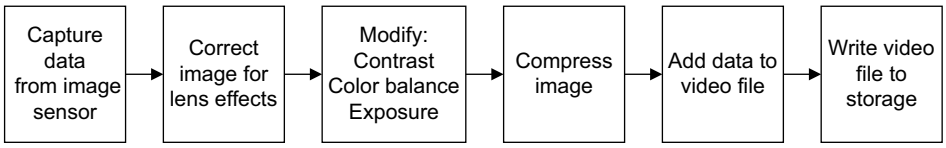

Figure 2-2 Pipeline processing

2.2 Concurrency

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

In a concurrent application multiple tasks are typically trying to share the same data. Access to this data must be managed carefully otherwise there can be complex problems such as:

Race conditions
: A race condition occurs when two or more threads try to modify the value of one variable at the same time. The final value of the variable should always be the same, but when a race condition occurs the variable can get a different value depending on the order of the writes.

Deadlocks
: A deadlock occurs when two threads become blocked by each other and neither thread can progress with their operations. This can happen when the threads each obtain a lock that the other thread requires.

Live locks
: Live locks are similar to a deadlocks but the threads keep running. However, because of the lock the threads can never complete their task.

A concurrent data structure is a data structure that can be accessed by multiple tasks without causing concurrency problems.

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

2.3 Limitations of parallel processing

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 10 processors makes it run 10 times faster.

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 speedup you can get from parallel processing. The formula for Amdahl's law is shown in Figure 2-3 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 2-3 Formula for Amdahl's law

Figure 2-4 shows the speedup that different numbers of processors provide for applications with different serial components.

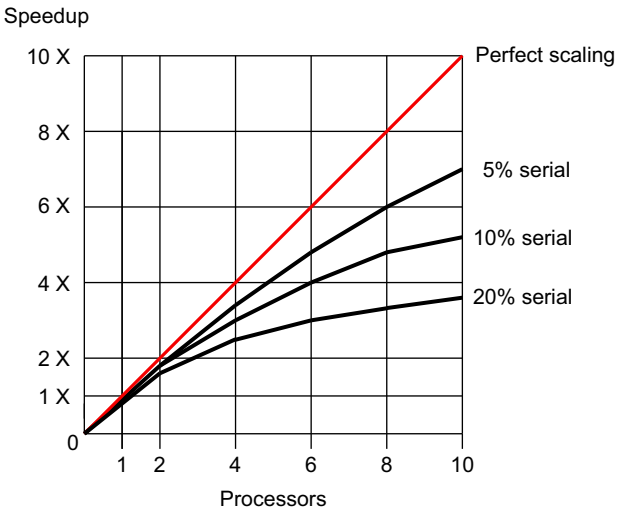

Figure 2-4 Speedup for application with different serial components

The biggest speedups are achieved with relatively small numbers of processors. However, as the number of processors rises the gains reduce.

You cannot avoid Amdahl's law in your application but you can reduce the impact. See *Reducing the effect of serial computations* on page 9-13.

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*. See *Embarrassingly parallel applications* on page 2-6.

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.

An example of an embarrassingly parallel application is rendering three dimensional graphics. The pixels are completely independent so they can be computed and drawn in parallel.

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

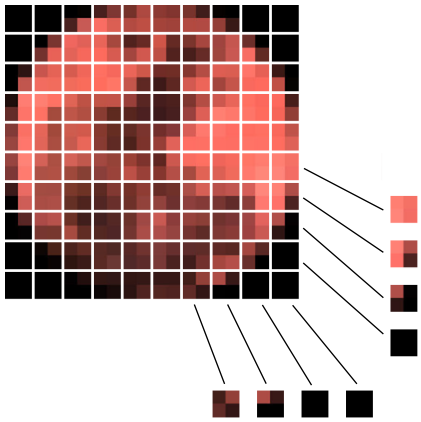

Figure 2-5 Embarrassingly parallel processing

Figure 2-5 shows an image divided into many small parts. These parts can all be processed simultaneously.

2.5 Mixing different types of parallelism

You can mix these types of parallelism in your applications. For example, an audio synthesizer might use a combination of all three types of parallelism:

- Task parallelism computes 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.

Chapter 3
OpenCL Concepts

This chapter describes the OpenCL concepts. It contains the following sections:

3.1 About OpenCL

OpenCL is an open standard that enables you to use the parallel processing capabilities of multiple types of processors including application processors, 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 or GPU instruction set.
- There is scope for specific hardware optimizations.

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

The Mali-T600 Series GPUs support OpenCL 1.1, Full Profile.

3.2 OpenCL applications

OpenCL applications consist of the following parts:

Application, or host, side code

- Calls the OpenCL APIs.
- Compiles the CL 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 NDRanges that the kernels execute over.

OpenCL kernels

- Written in OpenCL C language.
- Perform the parallel processing.
- Runs on the compute devices such as 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:

- Kernels that run on compute devices.
- A host application that runs on the application processor.

The host application

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

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. A 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

This section describes OpenCL data processing. It contains the following sections:

- *Work-items and the NDRange*.
- *OpenCL work-groups* on page 3-6.
- *Identifiers in OpenCL* on page 3-7.

3.4.1 Work-items and the NDRange

The data processed by OpenCL is in an *index space* of *work-items*. The work-items are organized in an *N-Dimensional Range* (NDRange) 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.

Figure 3-1 shows NDRanges with one, two and three dimensions.

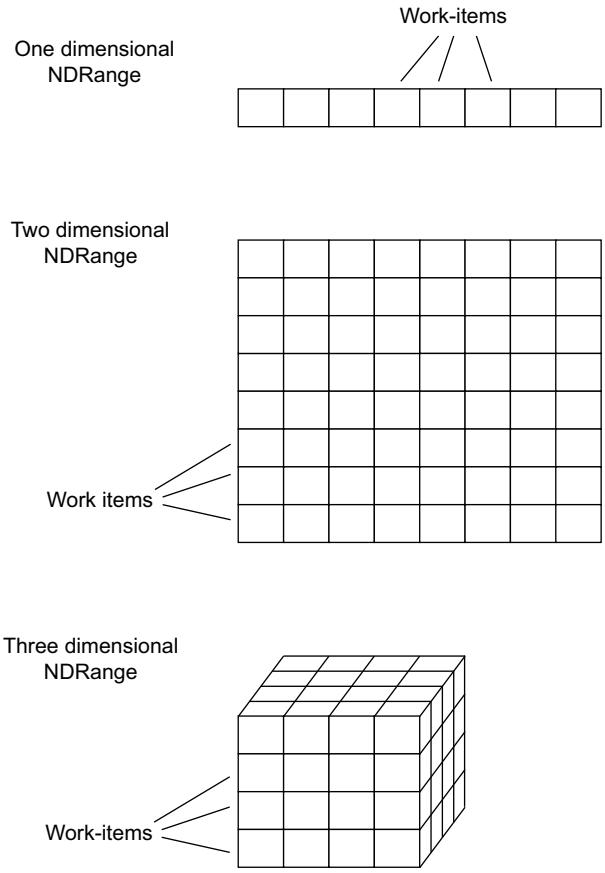

Figure 3-1 NDRanges and work-items

You group work-items into work-groups for processing. Figure 3-2 on page 3-6 shows a three dimensional NDRange that is split into 16 work-groups each with 16 work-items.

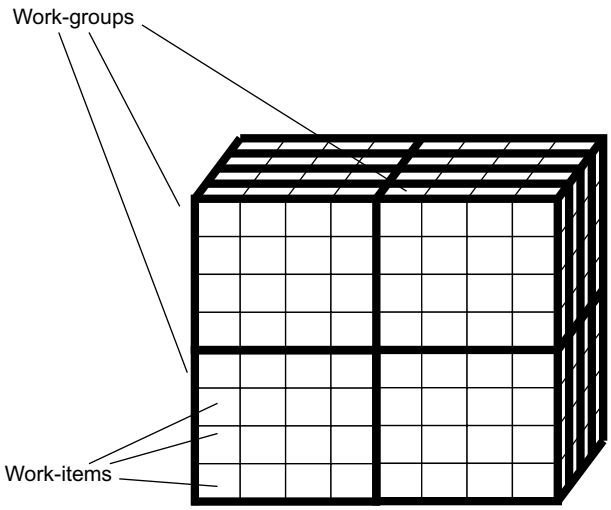

Figure 3-2 Work-items and work-groups.

3.4.2 OpenCL work-groups

Work-groups have a number of properties and limitations:

Properties of work-groups

- Work-groups are independent of each other.
- You 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 of 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 do the following:

- Perform barrier operations to synchronize execution points.

  For example:

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

- Use local atomic operations.
- Access shared memory.

3.4.3 Identifiers in OpenCL

There are a number of identifiers in OpenCL:

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

local ID Within each work-group, each work-item has a unique *local ID* that identifies it within its work-group.

work-group ID

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

3.5 The OpenCL memory model

The OpenCL memory model contains a number of components. Figure 3-3 shows the OpenCL memory model.

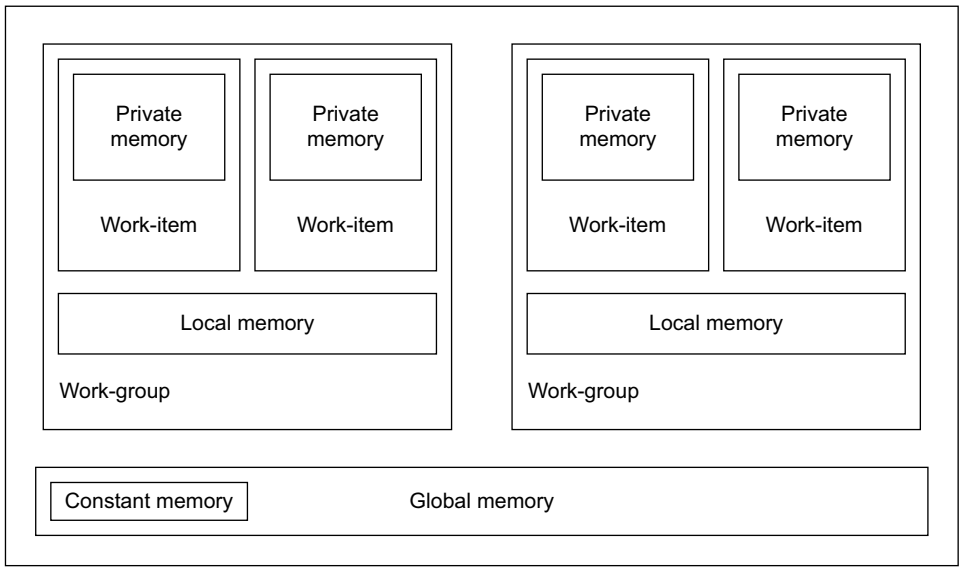

Figure 3-3 OpenCL memory model

Private memory

- Private memory is specific to a work-item.
- It is not visible to other work-items.

Local memory

- Local to a work-group.
- Accessible by the work-items in the work-group.
- Consistent to all work-items in the work-group.
- Accessed with the `__local` keyword.

Constant memory

- A memory region used for objects allocated and initialized by the host.
- Accessible as read-only by all work-items.

Global memory

- Accessible to all work-items executing in a context,
- Accessible to the host using `read`, `write`, and `map` commands.
- Consistent across work-items in a single work-group.
- Implements a relaxed consistency, shared memory model.
- There are no guarantee of memory consistency between different work-groups.
- Accessed with the `__global` keyword.

3.6 The Mali GPU memory model

The Mali GPU has a different memory model to desktop workstations:

Desktop Traditional desktop workstations have physically separate global, local and private memories.

Typically a graphics-card has its own local memory.

Data must be copied to the local memory and back again.

Mali GPU Mali GPUs have a unified memory system.

Local and private memory is physically global memory.

Moving data from global to local or private memory typically does not improve performance.

The traditional copying of data is not required.

Each compute device, that is the shader cores, have their own data caches.

3.7 OpenCL concepts summary

OpenCL primarily uses data parallel processing. OpenCL uses the following terminology:

- Computations in OpenCL are performed by pieces of code called *kernels* that execute on *compute device*s. Compute devices can be application processors, GPUs, or other types of processors.
- 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.
- Kernels instances execute in parallel.
- Work-items group 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 synchronised.
- Work-groups typically do not directly share data with each other. They can share data using global memory.
- 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 required for OpenCL development

To develop OpenCL applications for Mali GPUs, you require:

- A platform with a Mali-T600 Series GPU.
- An implementation of OpenCL for the Mali-T600 Series GPU.

You can develop on other hardware platforms with implementations of OpenCL but you cannot use them to estimate performance on a Mali-T600 Series GPU.

Implementations of OpenCL are available for a number of operating systems.

For a list of available hardware and the OpenCL drivers for the Mali-T600 Series GPUs see *Mali developer center*, www.malideveloper.arm.com.

4.2 Development stages

These are the stages for developing an 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. See *Analyzing code for parallelization* on page 6-3.

Note

Only 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 consists of a set of kernel functions. You must write the kernels that perform the computations.

Write infrastructure for kernels

OpenCL applications require infrastructure to set-up and run the kernels.

Chapter 5
Execution Stages of an OpenCL Application

This chapter describes the execution stages of an OpenCL application. It contains the following sections:

Note

This chapter provides an overview of the execution stages of an OpenCL application. It is not intended as a comprehensive lesson in OpenCL.

5.1 About the execution stages

This section describes a high level view of the OpenCL execution stages. Your OpenCL application must obtain information about your hardware then setup the runtime environment. The following sections describe these steps:

- *Platform setup*.
- *Runtime setup*.

5.1.1 Platform setup

You use the platform API to:

- Determine what OpenCL devices are available.

  Query to find out what OpenCL devices are available on the system using OpenCL platform layer functions. See *Finding the available compute devices* on page 5-4.

- Set up the OpenCL context.

  Create and set up a OpenCL context and one or more command queues to schedule execution of your kernels. See *Initializing and creating OpenCL contexts* on page 5-5.

5.1.2 Runtime setup

You use the runtime API to:

- Create a command queue.

  See *Creating a command queue* on page 5-6.

- Compile and build your program objects.

  Issue commands to compile and build your source code and extracts kernel objects from the compiled code.

  You must follow this sequence of commands:

  1. The program object is created either by calling `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.

  See *Creating OpenCL program objects* on page 5-7.

- Build a program executable.

  See *Building a program executable* on page 5-8.

- 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 both the application processor and the Mali GPU to access the data.

  See *Creating kernel and memory objects* on page 5-9.

- 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:

1. Call `clSetKernelArg()` for each parameter in the kernel function definition to set the kernel parameter values.
2. Determine the work-group size and index space to use to execute the kernel.
3. Enqueue the kernel for execution in the command queue.

See *Executing the kernel* on page 5-10.

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

  Reading the results on page 5-12.

- Clean up.

  Cleaning up on page 5-13.

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.

5.3 Initializing and creating OpenCL contexts

After 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 devices together to enable memory objects to be shared across different compute devices.

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 creating an OpenCL context. Leaving this parameter as a `NULL` value results in no error notification being registered.

Providing a callback function can be useful if you want to receive runtime errors for the particular OpenCL context. 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. For example:

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

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

If you have multiple OpenCL devices, such as an application processor and a GPU, 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

The program object encapsulates:

- 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 and then the code is compiled on 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 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 startup to save compute resources while the application is running.
- To avoid compilation at runtime, you can build a program object with a previously built binary.

  Note

  Applications with pre-built program objects are not portable.

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

After 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 == NULL)
{
        Cleanup();
        return 1;
}
```

5.7 Creating kernel and memory objects

This section describes creating kernel and memory objects. It contains the following sections:

- *Creating kernel objects*.
- *Creating memory objects*.

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

After you have created and registered your kernels, send the program data to the kernels:

1. Package the data in a memory object.
2. Associate the memory object with the kernel.

There are two 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, that is, you cannot see the implementation details of these structures.

To create buffer objects, use the `clCreateBuffer()` function. To create image objects, use the `clCreateImage2D()` or `clCreateImage3D()` functions.

5.8 Executing the kernel

This section describes the stages in executing the kernel. It contains the following sections:

- *Determining the data dimensions*.
- *Determining the optimal global work size*.
- *Determining the local work-group size*.
- *Enqueuing kernel execution* on page 5-11.
- *Executing kernels* on page 5-11.

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. You can for example, 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 worksize 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 the most efficient work-group size for your kernel.

To get the maximum possible 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

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. That is, 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 for later processing by the device. The call to `clEngueueNDRangeKernel()` 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_FALSE, CL_MAP_READ, 0,
               ( sizeof(unsigned char) * buffer_size), num_deps, deps[1], NULL, &err );

ASSERT( CL_SUCCESS == err );
```

Note

This call does not guarantee to make the buffer available until you call `clFinish()`.

If you change the third parameter of `clFinish()` or `clEnqueueBuffer()`, `CL_FALSE` to `CL_TRUE`, the call becomes a blocking call and the read is completed before `clEnqueueMapBuffer()` returns.

5.10 Cleaning up

When the application no longer requires the various objects associated with the OpenCL runtime and context, you must free these resources. Use the following 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 section describes converting existing code to OpenCL. It contains the following sections:

6.1 Profile 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 typically a relatively small part of the code. This is the part of the code that can probably 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. You can download DS-5 from the *Mali developer web site*, http://www.malideveloper.arm.com

6.2 Analyzing code for parallelization

This section describes how to analyze compute intensive code for parallelization. It contains the following sections:

- *About analyzing code for parallelization*.
- *Look for data parallel operations*.
- *Look for operations with few dependencies*
- *Analyze loops* on page 6-4.

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 be the following:

Straight forward

Parallelizing the code requires small modifications. See *Use the global ID instead of the loop counter* on page 6-6.

Difficult Parallelizing the code requires complex modifications. See *Compute values in a loop with a formula instead of using counters* on page 6-7.

Difficult and includes dependencies

Parallelizing the code requires complex modifications and the use of techniques to avoid dependencies. See the following sections:

- *Compute values per frame* on page 6-7.
- *Perform computations with dependencies in multiple-passes* on page 6-8.
- *Pre-compute values to remove dependencies* on page 6-8.

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 a different ways. Some of these might be parallelizable.

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

See *Using parallel processing with non-parallelizable code* on page 6-11.

6.2.2 Look for data parallel operations

Look for tasks that do large numbers of operations that:

- Complete without sharing data.
- Do not depend on the results from each other.

These types of operations are data parallel so are ideal for OpenCL.

6.2.3 Look for operations with few dependencies

If tasks have few dependencies, it might be possible to run them in parallel.

Dependencies between tasks prevent parallelization because it forces tasks to be performed sequentially. If the code has dependencies, consider:

- Is there a way to remove the dependencies?
- Can you delay the dependencies to later in execution?

6.2.4 Analyze loops

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

Loops that process a small number of 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.
- Accesses 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. See *Use the global ID instead of the loop counter* on page 6-6.

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 a number of techniques that help you deal with dependencies. See if you can use these techniques to parallelize the loop. See *Parallel processing techniques in OpenCL* on page 6-6.

Non-parallelizable loops

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

See *Using parallel processing with non-parallelizable code* on page 6-11.

6.3 Parallel processing techniques in OpenCL

This section describes parallel processing techniques you can use in OpenCL. It contains the following sections:

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 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.

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

Loop example in C:

The following loop fills an array with numbers.

```
void SetElements(void)
{
int loop_count;
int my_array[4096];

for (loop_count = 0; loop_count < 4096; loop_count++)
    {
    my_array[loop_count] = loop_count;
    }

printf("Total %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 using than 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 image shown in Figure 6-1 is of 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 are shown in Figure 6-2.

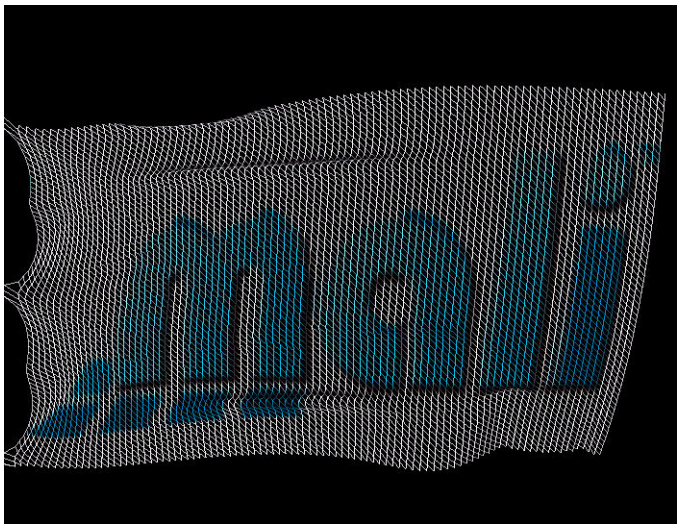

Figure 6-2 Flag simulation grid

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 described in *Compute values per frame* on page 6-7 by breaking up computations 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 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.

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 is dependent on another waveform called the modulator.

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 re-computed 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 standards 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 a 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 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. Input read from player.
2. Game logic computes the progress of the game.
3. Scene objects moved based on the results of the game logic.
4. Physics engine computes positions of all objects in the scene.
5. Game uses OpenGL ES to draw objects on screen.

6.3.7 Use task parallelism

Task or functional parallelism involves breaking an application up 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:

- Communicate with an external server.
- Read player input.
- Update the game state.
- Generate sound effects.
- Play music.
- Update the display.

These tasks require synchronisation 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 program.

- Recording program.
- Updating program guide.
- Displaying options.
- Reading from media storage device.
- Playing program.
- Decoding video stream.
- Playing audio.
- Scaling image to correct size.

6.4 Using parallel processing with non-parallelizable code

If you cannot parallelize your code there is still the possibility that you can use parallel processing.

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.

The fact the code cannot be parallelized only means this specific implementation cannot be parallelized. It does not mean the problem cannot be solved in a parallel way.

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.

See *Use concurrent data structures* on page 6-12.

Solve the problem in a different way

Take a step back and think about 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.

- Can the data you want to process be broken up into small data elements?
- Can these data elements be placed into a concurrent data structure?
- Can you process the data elements independently?

If the answer to these are yes, then you can probably solve your problem with OpenCL.

6.5 Dividing data for OpenCL

This section describes dividing data for processing with OpenCL. It contains the following sections:

- *About dividing data for OpenCL*.
- *Use concurrent data structures*.
- *Data division examples*.

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 the following hierarchy of levels:

- At the highest level the data is divided into an NDRange. The total number of elements in the NDRange is known as the global work size.
- The NDRange is divided into work-groups.
- Each work-group is divided into work-items.

See Chapter 3 *OpenCL Concepts*.

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 be all be read in parallel without any issues.

Work-items can also write to these data structures but you must be aware of a number of restrictions:

- Work-items can access any data structure that is read-only.
- Work-items can write to any data structure providing the work-items write to different elements.
- 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 provided that 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

The following are examples of data in different dimensions that you can process with OpenCL:

Note

These examples map the problems into the NDRanges with 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 1600 by 1200 pixel image by mapping it onto an two dimensional NDRange of 1600 by 1200.

The total number of work-items is the total number of pixels in the image, that is, 1920000.

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 15000 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 4 pixels, this means each work-item processes 4 pixels and you require 4 time fewer work-items to process the entire image. This means your NDRange can be reduced to 400 by 1200 and you only require 3750 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 concrete 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 for Mali GPUs

This chapter describes how to retune existing OpenCL code for 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 can work on many different types of compute device but performance is not preserved.

Existing OpenCL is typically tuned for specific architectures such as desktop GPUs. To achieve better performance on Mali GPUs you must retune the code for the Mali GPUs.

The procedure to convert OpenCL code to run optimally on Mali GPUs is:

1. Analyze the code.
2. Locate and remove optimizations for alternative compute devices.
3. Vectorize the code.
4. Optimize the code for the Mali GPU.

7.2 Differences between desktop based architectures and Mali GPUs

This section describes the differences between desktop based GPU and Mali GPUs. It contains the following sections:

- *About desktop based GPU architectures*.
- *About the architecture of the Mali-T600 Series GPUs*.
- *Programming a Mali-T600 Series GPU* on page 7-4.

7.2.1 About desktop based GPU architectures

Desktop GPUs have:

- Large chip area.
- A large numbers of shader cores.
- Very high bandwidth memories.

These are possible because desktop GPUs have a large power budget.

Desktop GPUs have shader architectures that put threads into *thread groups*. These are known as *warps* or *wavefronts*.

This mechanism means 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 the architecture of the Mali-T600 Series GPUs

The Mali-T600 Series GPUs contains one to eight identical shader cores. Each shader core supports up to 256 concurrently executing threads.

Each shader core contains:

- Two or four 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 peak throughput of each shader core is two arithmetic instruction words and one load-store instruction word per cycle.

The Mali-T600 Series GPUs use a VLIW (*Very Long Instruction Word*) architecture. Each instruction word contain multiple operations. The Mali-T600 Series GPUs also use SIMD (*Single Instruction Multiple Data*), 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. The next instruction from the program executes after the completion of the previous instruction.

7.2.3 Programming a Mali-T600 Series GPU

In some respects, programming a Mali-T600 Series GPU is easier than programming a desktop GPU:

- On a Mali GPU the global and local OpenCL address spaces are mapped to the same physical memory and are backed by L1 and L2 caches. This means 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 issue. This is not the case for warp or wavefront based architectures.

  Note

  In OpenCL each work-item maps to a thread.

7.3 Procedure for retuning existing OpenCL code for Mali GPUs

This section describes the procedure for retuning existing OpenCL code for Mali GPUs. It contains the following sections:

- *Analyze code*.
- *Locate and remove device optimizations*.
- *Optimizing your OpenCL code for Mali GPUs* on page 7-6.

7.3.1 Analyze code

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

Try to understand the following:

- What is the purpose of the code?
- How does the algorithm work?
- What would the code look like if there were no optimizations?

The answers to these questions can act as a guide to help you remove the device specific optimizations.

These questions can be difficult to answer 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 reduce performance.

To retune the code for a Mali GPU you must first remove all of the following types of optimizations to create a non device-specific *reference implementation*;

Use of local or private memory

Mali GPUs use caches instead of local memories. OpenCL local and private memories are mapped into main memory. There is therefore no performance advantage using local or private memories on a Mali GPU.

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 on Mali GPUs.

If your code copies data into a local or private memory, processes it, then writes it out again the copies both waste performance and power.

There are circumstances when copying does not waste performance. For example, if data is processed during a copy to local or private memory and used by a single work-group.

In this case the data can only be used by a single work-group. If you want the data to be accessed by multiple work-groups do not do any copies and keep the data in global memory.

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 then those used by a Mali GPU.

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

The Mali-T600 Series GPUs use a 64 byte line size. Try retuning the code to this line size.

Use of scalars

Some GPUs work with scalars whereas Mali GPUs can also use vectors. Vectors process multiple elements simultaneously enabling higher data throughput.

Optimizations for 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.

Mali GPUs do not use warps or wavefronts so remove any optimizations for them.

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

Threads on a Mali GPU 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.

7.3.3 Optimizing your OpenCL code for Mali GPUs

To optimize the code for a Mali GPU see Chapter 8 *Optimizing OpenCL for Mali GPUs*.

Chapter 8
Optimizing OpenCL for Mali GPUs

This chapter describes the procedure to optimize applications for OpenCL for the Mali-T600 Series GPUs. It contains the following sections:

8.1 The optimization process for OpenCL applications

This section describes the steps to take to optimize an OpenCL application. It contains the following sections:

- *Measure individual kernels*.
- *Select the kernels that take the most time*.
- *Analyze the kernels*.
- *Measure individual parts of the kernel*.

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:

8.1.1 Measure individual kernels

Go through your kernels one at a time and:

1. Measure the time of a number of runs.
2. Average the results.

Note

It is important that you measure the time of kernels on their own to get accurate measurements.

Do a dummy run of the kernel the first time to ensure 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.

8.1.2 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.

8.1.3 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.
- Use the Off-line Shader Compiler to check the balancing between the different pipelines.

8.1.4 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 the application processor and the Mali GPU

If you can, ensure that both the application processor and Mali GPU run in parallel:

Do not use clFinish() for synchronization

Sometimes the application processor must access data written by the Mali GPU.

You can do this is with `clFinish()` but this introduces delays because calls to `clFinish()` wait until the Mali GPU job completes. During that time, the application processor is idle.

Avoid this if possible because it serializes execution.

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

Where possible, use `clWaitForEvents()` or callbacks to ensure that the application processor and Mali GPU can work in parallel.

Something similar the following works well:

1. Split work into many parts.
2. For each part:
   a. Do application processor processing for part `X`.
   b. Submit the OpenCL work-items for part `X`.
3. For each part:
   a. Wait for OpenCL work-items for part `X` to complete using `clWaitForEvents`.
   b. Do more work on the application processor.

8.3 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 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.

Chapter 9
OpenCL Optimizations List

This chapter lists a number of optimizations to use when writing OpenCL for the Mali-T600 series GPUs. It contains the following sections:

9.1 General optimizations

ARM recommends the following:

Use the best processor for the job

GPUs are designed for parallel processing. Application processors are designed for high speed serial computations and can also perform parallel computation in a cluster configuration.

All applications contain sections that perform control functions and others that perform computation.

- Use OpenCL for the parallelizable compute functions.
- Control and serial functions are best performed on your application processor.

See Chapter 6 *Converting Existing Code to OpenCL*.

Compile the kernel once at the start of your application

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

Enqueue a large number of work-items

To get maximum use of all the Mali GPU shader cores you must enqueue a large number of work-items. For a Mali-T604 this is a minimum of 4096 work-items.

If you can perform your computation with fewer shader cores you can save power by enqueueing fewer work-items.

Process large amounts of data

You must be processing 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 data set where you start to see benefits depends on the processor you are running your OpenCL code on.

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.

Use the correct data types

Check each variable to see what range it requires.

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.
- An advantage of using smaller variables is more operations can be performed per cycle.

Use the right image data type

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.

Use asynchronous operations

If possible, use asynchronous operations between the Mali GPU and the application processor. For example:

- Do not make the application processor wait for results.
- Ensure the application processor has other operations to process before it requires results from the Mali GPU.
- Ensure the application processor does not interact with OpenCL kernels when they are executing.

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 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.

9.2 Memory optimizations

This section describes memory optimizations. It contains the following sections:

9.2.1 About memory optimizations

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 the memories are separate and buffer allocation involves memory copies. This is wasteful because copies take time and consume power.

To avoid 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.

Table 9-1 shows the different `cl_mem_flags` parameters in `clCreateBuffer()`,

Table 9-1 Parameters for clCreateBuffer()

Parameter	Description
`CL_MEM_ALLOC_HOST_PTR`	This is a hint to the driver indicating that the buffer is accessed on the host side. To use the buffer on the application processor side, you must map this buffer and write the data into it. This is the only method that does not involve coping 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. 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.

9.2.2 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 Figure 9-1.

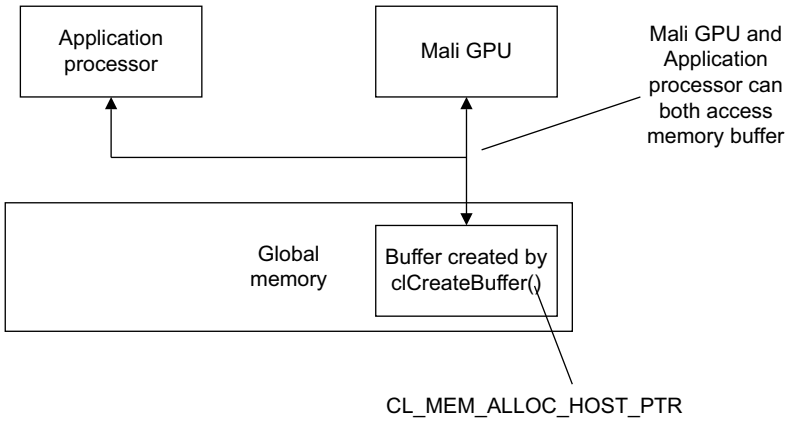

Figure 9-1 Memory buffer created by `clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)`

Note

- You must make the initial memory allocation through the OpenCL API.
- If OpenCL calls are repeatedly interleaved with application processor activity, the pointers that access buffers on the CPU might change.
- Always use the latest pointer returned. This is because the pointer returned is not guaranteed to be the same value every time you call the function on a particular buffer.

9.2.3 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 memory space of the Mali GPU. This is shown in Figure 9-2.

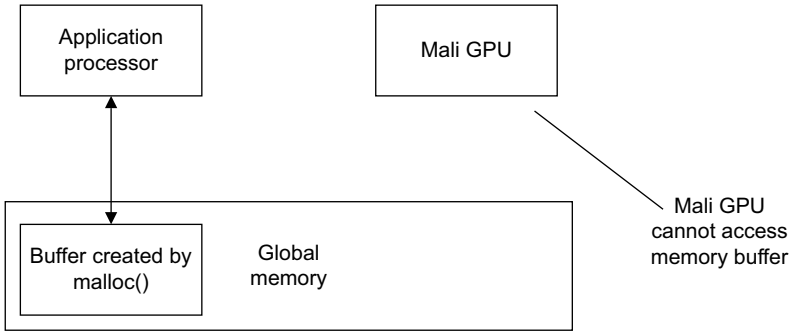

Figure 9-2 Memory buffer created by malloc()

9.2.4 Do not create buffers with `CL_MEM_USE_HOST_PTR` if possible

The Mali GPU can access the memory buffers created by `clCreateBuffer(CL_MEM_USE_HOST_PTR)` but buffers created this way must have data copied into them by the application processor. These copy operations are computationally expensive so it is best to avoid this method of allocating buffers if possible. This method of allocating buffers is shown in Figure 9-3.

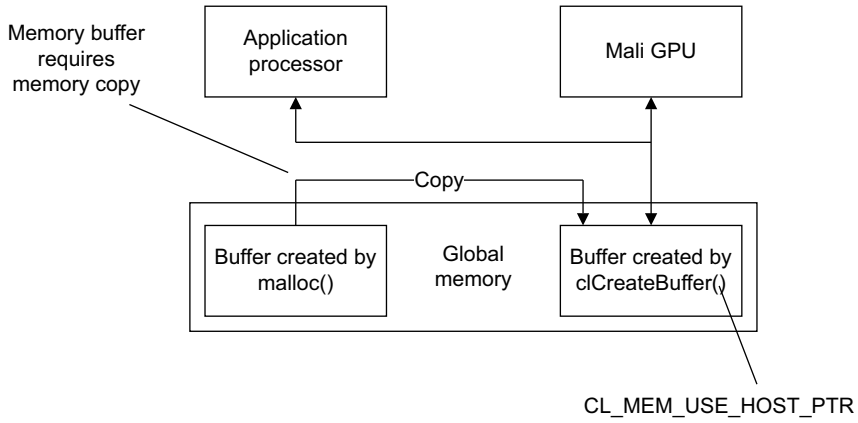

Figure 9-3 Memory buffer created by `clCreateBuffer(CL_MEM_USE_HOST_PTR)`

9.3 Kernel optimizations

ARM recommends the following:

- If your kernel has no preference for the work-group size, pass `NULL` to the local work size argument of the `clEnqueueNDRangeKernel()`.
- If possible, use work-group sizes that are a power of two. These are more efficient on the Mali-T600 Series GPUs. 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 hardware in the Mali-T600 Series GPUs. If the maximum work-group size is below 128, your kernel might be too complex.
- Some kernels require work-groups for synchronisation of the work-items within the work-group with barriers. These typically require a specific work-group size.

  In cases where synchronisation between work-items are not required, the choice of the size of the work-groups depends on the most efficient size for the device. Pass in `NULL` to enable OpenCL to pick the optimal size.
- 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.
- 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 you have kernels A and B. Kernel B takes an input produced by kernel A.

  If a kernel A is merged with a kernel B to form kernel C then C can only input any constant data plus the output from the part that was previously kernel A.

  Kernel C cannot use the output from kernel A n-1, because it is not guaranteed that 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 A and B are the same.
- Avoid splitting kernels. If you are required to split a kernel, split it into as few kernels as possible.
- If your kernels are small, use data with a single dimension and ensure the work-group size is a power of two.

Use vector operations in kernel code

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

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 a large number of work-items.
- Make sure the variables are large, for example, use vectors.

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.

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

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

9.4 Code optimizations

ARM recommends the following:

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`.

Make your kernel code as simple as possible

Make your kernel code as simple as possible. This assists the auto-vectorization process.

Vectorize your code

Mali GPUs compute 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 you vectorize your code for maximum performance.

The shader cores in the Mali-T600 Series GPUs contain 128-bit wide vector registers. Vectorize the algorithms in your kernels to make best use of the Mali GPU hardware.

Vectorize incrementally

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

Use vector loads and saves

Use vector loads to load as much data as possible in a single operation. 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.

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.

Do 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 re-use data already loaded.
- Use as many arithmetic instructions as possible per load.

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

Use the built-in functions

Many of the built-in functions are implemented as fast hardware instructions. See Appendix B *OpenCL Built-in Functions* for a list of built-in functions with relative speed ratings.

Use the precise versions of built-in functions

Use the precise versions of built-in functions.

In most cases 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()`.

See *half_ and native_ math functions* on page B-4.

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.

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 works for powers of two.
- Divide and shift use different methods of rounding negative numbers.

Avoid conversions from `float` to `int`

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

Avoid processing single values

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

Avoid writing kernels that use a large number of live variables

Avoid writing kernels that use a large number of live variables. Using too many live variables can impact performance and limits the maximum workgroup size.

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. You can fit eight 16-bit words into a 128-bit wide vector but only four 32-bit words.

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 Mali GPU caches.

Maximise 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.

9.5 Execution optimizations

ARM recommends the following:

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

9.6 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 the communication code that sends and receives data to reduce latency.
- Keep messages small. Reduce communication overhead by sending only the data that is absolutely required.
- Ensure the size of memory blocks used for communication are a power of 2. This makes the data more cacheable.
- If possible, 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. These are optimized for low latency tasks.

Chapter 10
The Mali OpenCL SDK

The Mali OpenCL SDK includes the following tutorials to help you understand OpenCL development:

Hello World Tutorial

This tutorial provides a basic introduction to OpenCL and vectorization.

Template Tutorial

This tutorial provides an OpenCL template that you can use as a starting point to develop an OpenCL application.

Memory Optimizations

The Memory Optimizations directory contains a Data Sharing tutorial that demonstrates efficient sharing of memory between a Mali-T600 Series GPU and an application processor.

Sobel Filter Tutorial

This tutorial demonstrates the use of the Sobel image filter. This is a simple convolution filter used primarily for edge detection algorithms.

FIR Float Filter Tutorial

This tutorial demonstrates the use of a floating point *Finite Input Response* (FIR) image filter. You can use this for pixelization or noise reduction.

Mandelbrot Tutorial

This tutorial demonstrates the use of calculating the Mandelbrot set to produce fractal patterns.

SGEMM Tutorial

This tutorial demonstrates the use of *Single-Precision General Matrix Multiplication* (SGEMM) in OpenCL.

The Mali OpenCL SDK is available from the *Mali developer center*, http://www.malideveloper.arm.com

Appendix A
OpenCL Data Types

This appendix lists the data types available in OpenCL. These types are all natively supported on Mali GPUs.

The OpenCL types are used in OpenCL C. The API types are equivalents for use in your application. Use these to ensure 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, an add of two 256-bit vectors takes twice as long as an add of two 128-bit vectors. You can use vector sizes less than 128-bit 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 no performance cost on a Mali GPU. For example, converting a vector of 8-bit values to 16-bit values:

```
ushort8 a;  uchar8 b;
a = convert_ushort16(b);
```

Table A-1 shows 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`	-	32-bit or 64-bit unsigned integer
`ptrdiff_t`	-	32-bit or 64-bit unsigned integer
`intptr_t`	-	signed integer
`uintptr_t`	-	unsigned integer
`void`	`void`	void

Table A-2 shows 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

Table A-3 shows 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
`sampler_t`	sampler handle
`event_t`	event handle

Table A-4 shows 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
`double`, `double`*n*	64-bit float, vector
`half`*n*	16-bit, vector
`quad`, `quad`*n*	128-bit float, vector
`complex half`, `complex half`*n*, `imaginary half`, `imaginary half`*n*	16-bit complex, vector
`complex float`, `complex float`*n*, `imaginary float`, `imaginary float`*n*	32-bit complex, vector
`complex double`, `complex double`*n*, `imaginary double`, `imaginary double`*n*	64-bit complex, vector
`complex quad`, `complex quad`*n*, `imaginary quad`, `imaginary quad`*n*	128-bit complex, 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, vector
`long long`, `long long`*nb*	128-bit signed
`unsigned long long`, `ulong long`, `ulonglong`*n*	128-bit unsigned

Appendix B
OpenCL Built-in Functions

This appendix lists the OpenCL built-in functions. It contains the following sections:

The functions listed have a relative speed rating. The ratings are from A to C, where A is the fastest.

Note

Ratings for memory accesses are separate from arithmetic operations. An A rated memory operation might be equivalent to a C rated arithmetic operation.

B.1 Work-item functions

Table B-1 lists the work-item functions.

Table B-1 Work-item functions

Function	Speed
`get_work_dim()`	A
`get_global_size()`	A
`get_global_id()`	A
`get_local_size()`	A
`get_local_id()`	A
`get_num_groups()`	A
`get_group_id()`	A
`get_global_offset()`	A

B.2 Math functions

Table B-2 lists the math functions.

Table B-2 Math functions

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

B.3 half_ and native_ math functions

Typically, on most architectures there is a trade-off between accuracy and speed. The Mali-T600 Series GPUs implement the full precision variants of the math functions at full speed so you are not required to make this trade-off.

The `half_` and `native_` variants of the math functions are provided for portability. See *Math functions* on page B-3.

Table B-3 lists the `half_` and `native_` math functions.

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()`

Note

In most cases 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()`.

B.4 Integer functions

Table B-4 lists the integer functions.

Table B-4 Integer functions

Function	Speed
abs()	A
abs_diff()	A
add_sat()	A
hadd()	A
rhadd()	A
clz()	A
max()	A
min()	A
sub_sat()	A
mad24(), identical to 32-bit multiply accumulate	A
mul24(), identical to 32-bit multiplies	A
clamp()	B
mad_hi()	B
mul_hi()	B
mad_sat()	B
rotate()	B
upsample()	B

B.5 Common functions

Table B-5 lists the common functions.

Table B-5 Common functions

Function	Speed
`max()`	A
`min()`	A
`step()`	A
`clamp()`	B
`degrees()`	B
`mix()`	B
`radians()`	B
`smoothstep()`	B
`sign()`	B

B.6 Geometric functions

Table B-6 lists the geometric functions.

Table B-6 Geometric functions

Function	Speed
`dot()`	A
`normalize()`	B
`fast_distance()`	B
`fast_length()`	B
`fast_normalize()`	B
`cross()`	B
`distance()`	B
`length()`	B

B.7 Relational functions

Table B-7 lists the relational functions.

Table B-7 Relational functions

Function	Speed
`any()`	A
`all()`	A
`bitselect()`	A
`select()`	A
`isequal()`	A
`isnotequal()`	A
`isgreater()`	A
`isgreaterequal()`	A
`isless()`	A
`islessequal()`	A
`islessgreater()`	A
`isfinite()`	B
`isinf()`	B
`isnan()`	B
`isnormal()`	B
`isordered()`	B
`isunordered()`	B
`signbit()`	B

B.8 Vector data load and store functions

Table B-8 lists the vector data load and store functions. These are all speed A.

Table B-8 Vector data load and store functions

Function	Speed
`vload()`	A
`vstore()`	A
`vload_half()`	A
`vstore_half()`	A
`vloada_half()`	A
`vstorea_half()`	A

B.9 Synchronization

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 a number of factors such as:

- The number of work-items in the work-groups being synchronized,
- How much the work-items diverge.

Table B-9 lists the synchronization functions.

Table B-9 Synchronization functions

Function	Speed
`barrier()`	-
`mem_fence()`	A
`read_mem_fence()`	A
`write_mem_fence()`	A

Note

ARM recommends you do not use barriers, especially in small kernels.

B.10 Asynchronous copy functions

Table B-10 lists the 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

Table B-11 lists the atomic functions.

Table B-11 Atomic Functions

Function	Speed
`atomic_add()`	B
`atomic_sub()`	B
`atomic_xchg()`	B
`atomic_inc()`	B
`atomic_dec()`	B
`atomic_cmpxchg()`	B
`atomic_min()`	B
`atomic_max()`	B
`atomic_and()`	B
`atomic_or()`	B
`atomic_xor()`	B

B.12 Miscellaneous vector functions

Table B-12 lists the miscellaneous vector functions.

Table B-12 Miscellaneous vector functions

Function	Speed
`vec_step()`	A
`shuffle()`	A
`shuffle2()`	B

B.13 Image read and write functions

Table B-13 lists the image read and write functions.

Table B-13 Image read and write functions

Function	Speed
read_imagef()	A
read_imagei()	A
read_imageui()	A
write_imagef()	A
write_imagei()	A
write_imageui()	A
get_image_width()	B
get_image_height()	B
get_image_depth()	B
get_image_channel_data_type()	B
get_image_channel_order()	B
get_image_dim()	B

Appendix C
OpenCL Extensions

The Mali OpenCL driver supports the following extensions:

- `cl_khr_byte_addressable_store`.
- `egl_khr_cl_event`.
- `cl_khr_egl_event`.
- `cl_khr_egl_image`.
- `cl_khr_global_int32_base_atomics`.
- `cl_khr_global_int32_extended_atomics`.
- `cl_khr_int64_base_atomics`.
- `cl_khr_int64_extended_atomics`.
- `cl_khr_local_int32_base_atomics`.
- `cl_khr_local_int32_extended_atomics`.