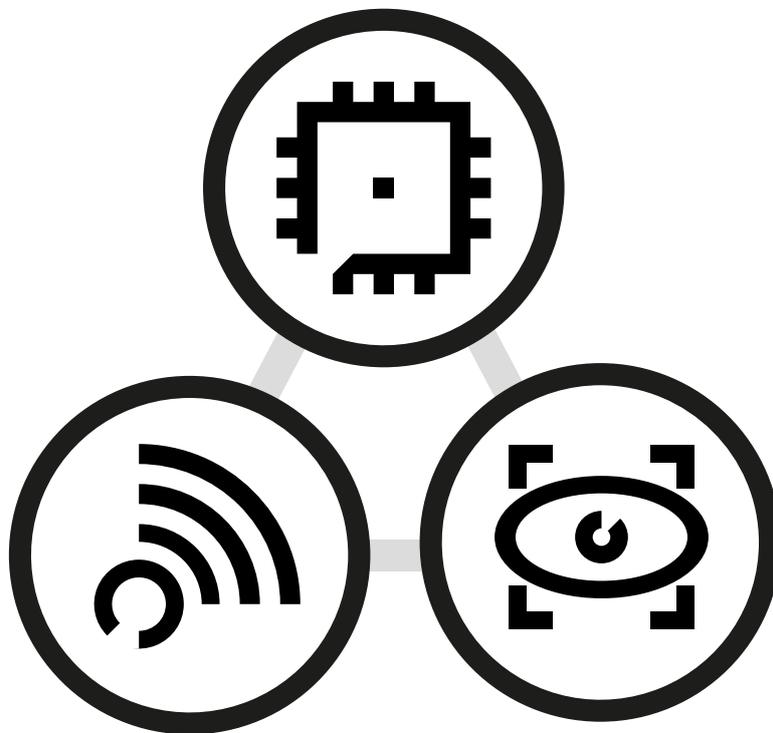


PowerVR Compute Development Recommendations

Revision: 1.0
13/03/2020
Public



Public. This publication contains proprietary information which is subject to change without notice and is supplied 'as is', without any warranty of any kind. Redistribution of this document is permitted with acknowledgement of the source.

Published: 13/03/2020-15:48

Contents

| | |
|---|-----------|
| 1. Introduction..... | 5 |
| Document Overview..... | 5 |
| Glossary..... | 5 |
| 2. A Modern System on Chip..... | 7 |
| Single Instruction, Multiple Data..... | 8 |
| Parallelism..... | 8 |
| Divergence - Dynamic Branching..... | 8 |
| Scalar or Vector Processing..... | 9 |
| Vector..... | 9 |
| Scalar..... | 9 |
| 3. Architecture Overview..... | 10 |
| Compute Data Master..... | 10 |
| Coarse Grain Scheduler..... | 11 |
| Unified Shading Cluster..... | 11 |
| Execution..... | 11 |
| Memory..... | 13 |
| USC Pair..... | 17 |
| Texture Processing Unit..... | 17 |
| L1 Mixed Cache..... | 17 |
| System Level Cache..... | 17 |
| 4. Performance Guidelines..... | 18 |
| Execute Tasks on the Most Appropriate Processor..... | 18 |
| Choose a Suitable Device..... | 19 |
| Identifying and Creating Work for the Graphics Core..... | 19 |
| Sharing the Graphics Core between Compute and Graphics Tasks..... | 20 |
| Minimise Bandwidth Usage..... | 20 |
| Avoid Redundant Copies..... | 20 |
| Group Memory Access Together..... | 23 |
| Use Shared/Local Memory Sensibly..... | 23 |
| Access Memory in Row-Major Order..... | 24 |
| Maximise Utilisation and Occupancy..... | 24 |
| Work on Large datasets..... | 24 |
| Choose an Appropriate Work-Group Size..... | 25 |
| Reduce Unified Store Usage (Private Memory)..... | 26 |
| Reduce Common Store Usage (Shared/Local Memory)..... | 26 |
| Be Aware of Padding..... | 26 |
| Maximise Instruction Throughput..... | 26 |
| Trade Precision for Speed..... | 27 |
| Tweak Work for the Graphics Core..... | 27 |
| Avoid Excessive Flow Control..... | 28 |
| Avoid Barrier Directly after Local or Constant Memory Access..... | 28 |

| | |
|---|-----------|
| Use Built-ins for Type Conversions..... | 28 |
| Measuring Peak Performance..... | 30 |
| Example Device: 500MHz G6400..... | 30 |
| Measured Throughput..... | 31 |
| Complex Operations..... | 32 |
| Optimisation Strategy Cheat Sheet..... | 34 |

1. Introduction

PowerVR Rogue is a family of Graphics Cores from Imagination Technologies designed to perform both compute and graphics tasks. Its programmable core architecture allows for extremely efficient and high-performance compute execution.

Document Overview

Elaboration on context of the document and its goals.

This document describes the recommended usage guidelines for achieving optimal performance when using compute on Imagination's PowerVR Rogue Graphics Cores. Most of the guide describes constructs and patterns that directly emerge from the PowerVR Rogue architecture. This enables the developer to make the correct decisions irrespective of the preferred approach as far as APIs and programming languages are concerned. Additionally, several specific details are included for OpenCL, OpenGL, OpenGL ES Compute, and RenderScript. These details have been written against the following API versions:

- OpenGL ES 3.1
- OpenGL 4.3
- OpenCL 1.x and 2.0
- RenderScript

This document assumes that the reader has a good working knowledge of at least one of these APIs. It is also assumed that the reader has worked through the examples of the relevant PowerVR Compute SDK, or has referred to Google's documentation for RenderScript, available [here](#).

After reading this document, readers should have a solid understanding of how compute works on Rogue Graphics Cores, as well as a good understanding of how to develop efficient and well-optimised code for these devices.

Note: This document also provides an optimisation strategy quick reference sheet in [Appendix C](#).

Glossary

A list of commonly used terms and their meanings/alternatives.

Throughout this document, extensive use is made of terminology identified in the table below.

Table 1: Common terms

| Term | Also referred to as | Description |
|-------------------------------|---|---|
| USC (Unified Shading Cluster) | Shading Cluster, Shading Unit, Execution Unit | A semi-autonomous part of the Graphics Core that can typically execute an entire work-group. Other large parts like Texture Units can be shared among USCs. |
| Core | Processor, Graphics Core | An almost completely autonomous part of the Graphics Core. Typically, a collection of USCs and possibly supporting hardware such as texture units. |
| Task | Thread Group, Warp, Wavefront | The native grouping of threads that a USC executes. Consists of 32 threads on PowerVR Rogue cores. |

Table 2: API-specific terminology

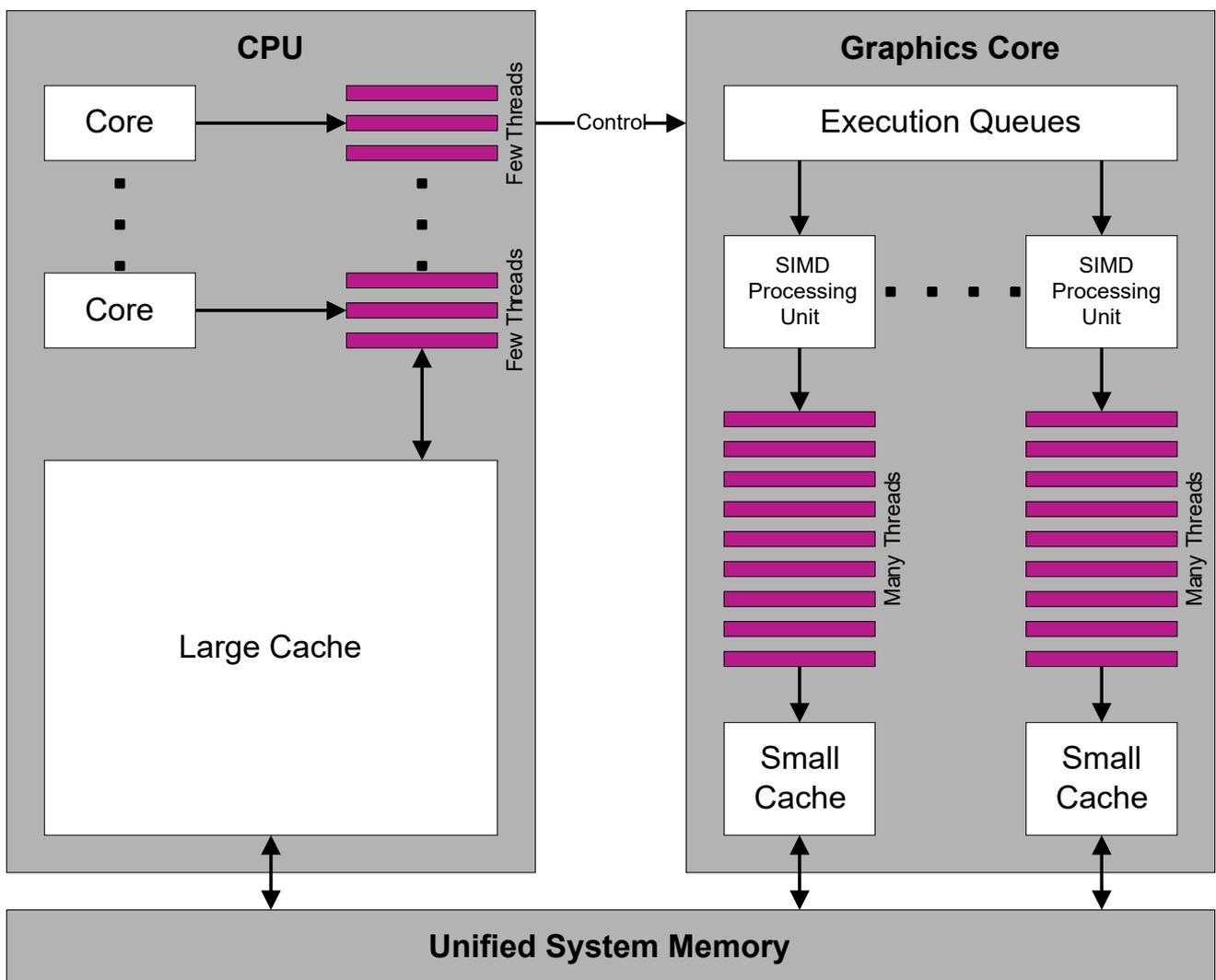
| Term | OpenGL and OpenGL ES | OpenCL | RenderScript |
|---------------------------|--|--|------------------------------|
| Kernel or shader | Compute shader | Kernel | Kernel |
| Thread | Thread, shader instance | Work-item | Kernel Invocation |
| work-group | work-group | Work-group | N/A |
| Shared memory | <code>shared variables</code> | <code>local memory</code> | N/A |
| Image | Texture, image | Image | Image |
| Constant, constant memory | <code>const / uniform variable, uniform block, uniform buffer</code> | <code>constant memory</code> | Constants |
| Private memory | Local variables, temporaries | <code>Variables, private memory</code> | Local variables, temporaries |
| Dataset | Dispatch size, dataset | <code>Global work, ND Range</code> | Dataset |

2. A Modern System on Chip

A quick explanation into the background of Rogue architecture.

As shown in the figure below, a modern System on Chip (SoC) often integrates both a CPU and a Graphics Core. The CPU is optimised for processing sequential, heavily-branched datasets that require low memory latency, dedicating transistors to control and data caches. The Graphics Core is optimised for repetitive processing of large, unbranched datasets, such as in 3D rendering. Transistors are dedicated to arithmetic logic units rather than data caches and flow control.

Figure 1: Abstract heterogeneous architecture



Single Instruction, Multiple Data

Typical CPUs are optimised to execute large, heavily-branched tasks on multiple pieces of data at a time. A thread running on a CPU is often unique and is executed on its own, largely independent of all other threads. Any given processing element will process just a single thread. Typical numbers of threads for a specific program on a CPU are commonly one to eight, up to a few tens at any period of time.

Graphics Cores work on the principle that the exact same piece of code will be executed on numerous multiple threads, often numbering into the millions, to handle the large screen resolutions of modern devices. These operations differ only in input and normally follow the exact same steps, instruction for instruction. To do this efficiently, each processor executes the same instruction on multiple threads concurrently, in a form of Single Instruction, Multiple Data (SIMD) processing.

This should not be confused with vector processing, which is another form of SIMD. SIMD processors may either be scalar; operating on one element at a time, or vector; operating on multiple elements at a time.

Parallelism

The main advantage of this type of architecture is that significant numbers of threads can be run in parallel for a correctly structured application. This type of parallelism is also done with extremely high efficiency and effectiveness. SIMD architectures are usually capable of running many orders of magnitude more threads at once than a typical CPU.

Operating on large coherent datasets is something they perform exceptionally well at, which is why they are traditionally used for graphical and image processing. Algorithms that function independently on such data are well suited to this sort of processor.

Divergence - Dynamic Branching

With each thread executing on one processor core, thread divergence must be handled differently to how it works on a typical CPU. A CPU is made with the assumption of a sequential pipe that will be branching, looping, doubling back, and otherwise following an extremely complex flow control in a very deep call stack. Modern CPUs are heavily optimised for this kind of work, with sophisticated branch prediction and other features aiming to optimise this kind of program.

A typical Graphics Core is optimised with the assumption that each "batch" of threads will execute the exact same code, instruction for instruction (lockstep), without any difference whatsoever. Additionally, it is expected but not required that the input will only be slightly different (coherent).

Optimising the design for these assumptions leads to the aforementioned efficiency for these cases. The downside to this is that when branches are encountered (`if`, `while`, `dynamic for`), each thread cannot just execute a different path. The typical implementation is to have the processing core execute all paths of a branch in every thread and just disable all threads that have not taken the path, unless none of the paths are used. The actual mechanisms for this may be different on different architectures, but the net effect is the same.

In terms of computation, the cost of executing any branching code is the cost of executing all branches (unless none of threads hit the branch) plus the branching instructions themselves. It is therefore necessary to be careful of excessive branching when executing on a SIMD processor. There are exceptions to this: one of the most common and notable cases is conditionally loading or storing data from main memory, where it is always worthwhile to branch. Usually, the bandwidth saved vastly outperforms the branch cost.

Scalar or Vector Processing

Modern processing architectures can either operate on a single value per processing unit (scalar) or multiple values per processing unit (vector).

Vector

Vector processing can be very efficient, especially for colour and vertex manipulation, as it can work on multiple values at the same time, rather than just one. Traditional rendering operations are therefore well suited to this type of architecture, as calculations often operate on three or four elements at once.

The main drawback of vector architectures is that if scalar values or vectors smaller than what the processor expects are used, the additional processing element width is wasted. The most common vector width is four, which means that a shader or kernel operating on three component vectors will operate these instructions with 75% efficiency. Having a shader that does work on scalars in the worst case may take this number down to as low as 25%. This wastes energy, as parts of the processor are doing useless work. It is possible to optimise for this by vectorising code, but this introduces additional programmer burden.

Scalar

Scalar processors tend to be more flexible in terms of what operations can be performed per hardware cycle, as there is no need to fill the additional processing width with data. Whilst vector architectures could potentially work on more values in the same silicon area, the actual number of useful results per clock will usually be higher in scalar architectures for non-vectorised code. Therefore, for compute tasks, it is a lot easier to work with scalar architectures, as there is no need to vectorise. Scalar architectures tend to be better suited to general purpose processing and more advanced rendering techniques. PowerVR Rogue cores are all scalar architectures.

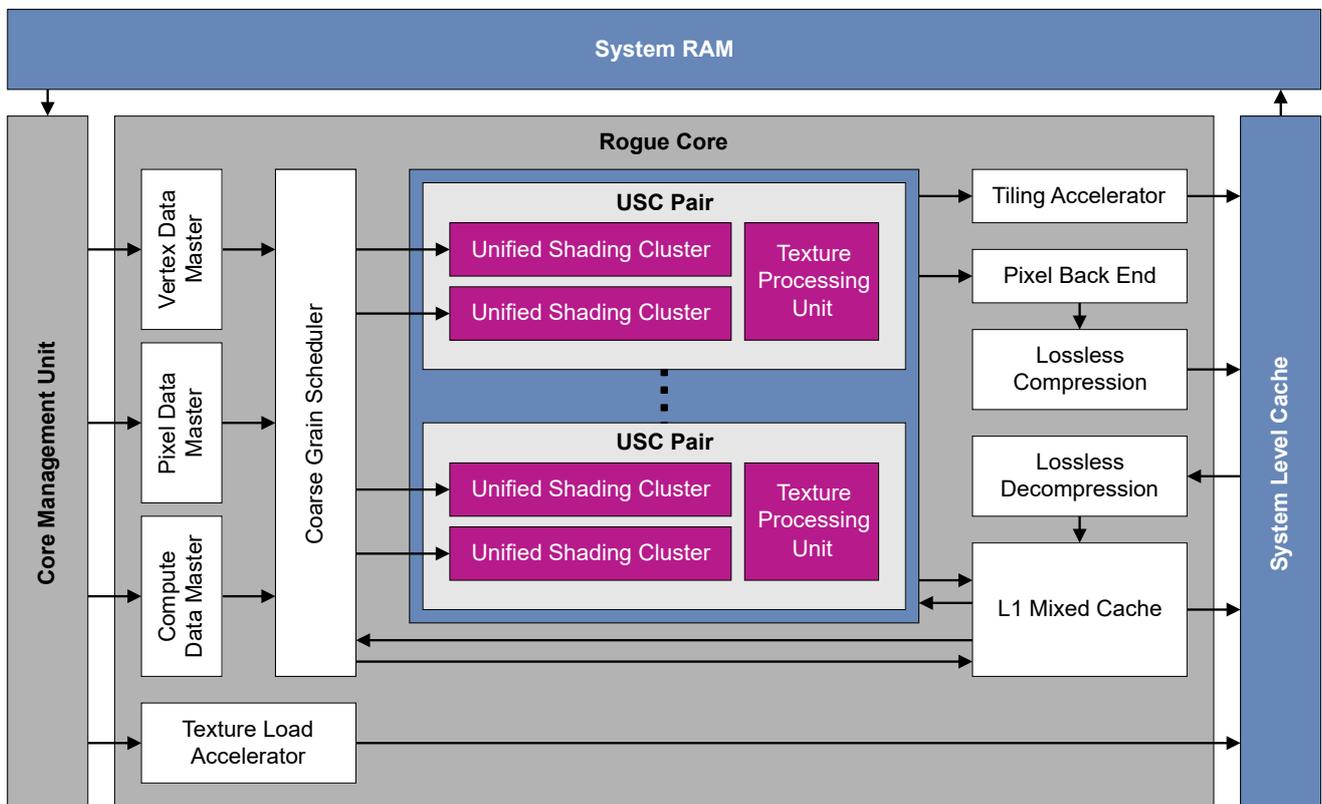
3. Architecture Overview

Detailed explanation on PowerVR Rogue architecture.

PowerVR's Rogue architecture is a programmable architecture capable of executing general purpose computations as well as rendering graphics.

As shown in the following figure, Rogue provides different hardware for transferring vertex, pixel, and compute data between memory and the Graphics Core, and programmable processors for performing the actual operations. Unlike the Series5 architecture, Rogue has a dedicated path for compute tasks via the Compute Data Master (CDM), with the programmable arithmetic handled by Unified Shading Clusters (USCs).

Figure 2: Rogue architecture and data flow overview



Compute Data Master

Any compute jobs that have been dispatched by the host will need to be first translated into individual tasks for the Graphics Core to consume. The job of the Compute Data Master (CDM) is to take input from the host and generate these tasks.

Coarse Grain Scheduler

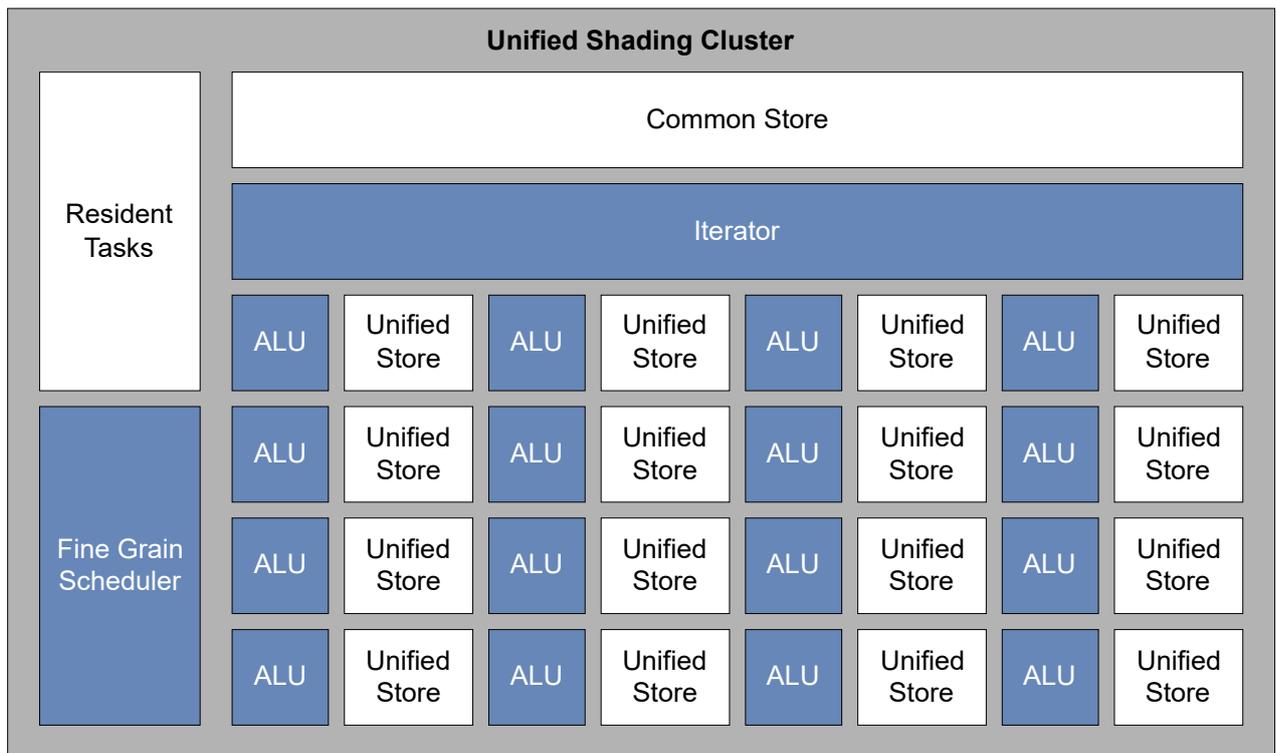
Once tasks have been generated by the CDM, they have to be scheduled for execution. The Coarse Grain Scheduler (CGS) takes the generated tasks and schedules tasks across the available Unified Shading Clusters (USCs) on a broad scale.

Unified Shading Cluster

Details on the Unified Shading Cluster component.

USCs are the programmable cores at the heart of the Rogue's computing power and are where the code in compute kernels is executed as illustrated below. The USC is a scalar SIMD processor, as described in the [A Modern System on Chip](#) section. Each USC normally has 16 ALUs.

Figure 3: Rogue Unified Shading Cluster



Execution

To execute the code within a compute kernel, the compiler has to translate the developer's intentions into something the USC can understand. It is equally useful

for the kernel writer to understand the mapping between hardware and API to understand how to get the best performance.

Threads

Each work item of OpenCL, thread of OpenGL ES, or kernel invocation for RenderScript is handled as a thread of instructions in the hardware. These threads are executed together, in groups of up to 32, known as a task. In any given cycle, 16 threads - one per ALU Pipe, will execute the same instruction. This means that half of a task runs in lockstep, with the other half staggered so that it executes one cycle behind.

Tasks

Tasks are the grouping used by Rogue hardware for SIMD execution of threads, with each task treated as a unit of work. Each task is executed by all ALU Pipes, one at a time, with half of the threads executing in lockstep in one cycle and the other half executing in the next cycle.

In RenderScript, the number of tasks used by a given execution is managed by the RenderScript compiler. If the global work size is a multiple of 32, the compiler will be able to separate threads to complete tasks, where every thread in a task is actively doing useful work. Any other number of items will leave threads vacant in some tasks, with tasks filling until all kernel instances are attached to a thread. The actual number of tasks and vacant threads per task will vary.

For OpenCL and OpenGL, while work-groups do not map directly to tasks, their size dictates task allocation and is arguably very important. Work-groups are split into tasks according to their size, as detailed in the following points:

- Work-group sizes that are a multiple of 32 allow each task generated to be fully used. Every thread in the task is actively doing useful work, with multiple tasks created until all work items are attached to a thread.
- Work-group sizes of four, eight, and 16 typically execute with multiple work-groups per task – eight, four, or two respectively. However, if there is barrier synchronisation in the kernel and the work-group size was not specified at compile-time, tasks will execute one work-group per task. Unused threads in the task will be left vacant, causing a dramatic drop in utilisation.

Note: The maximum work-group size of an execution is 512. Higher sizes are unsupported.

Scheduling

The Fine Grained Scheduler (FGS) is responsible for scheduling and descheduling work on a per-task basis from a pool of resident tasks. The size of this task pool will vary from device to device, but can typically be expected to be 48 or higher. The more tasks that are in this pool, the better any latency can be hidden.

One very important aspect of these tasks is that it is zero-cost for the USC. Tasks that need to wait for any reason are switched out and a ready-to-execute task is scheduled in its place immediately. This means that having large datasets scheduled at the same time generally allows greater efficiency without incurring

scheduling overhead. This is unlike, for example, CPU threads where scheduling cost is considerable.

Data Dependencies

When a data transfer dependency is encountered in a kernel, a task will need to wait for the data to be transferred before it can continue execution. The FGS will schedule out this task at no cost and schedule in another if one is available, until the task is ready to resume.

Data dependencies are not just data reads and writes but occur when the result of a read or write is required by another instruction. Any memory fences or barriers that occur within a kernel or implicit fences caused by using the result of a read will cause a dependency.

Memory

Memory space inside the USC comprises largely of two banks of registers. These are: the Common Store, which is shared across the entire USC; and the Unified Stores, which are allocated per four ALU Pipelines.

Unified Stores

The Unified Stores are four small banks of registers in the USC, with each shared equally between four ALU Pipes. Any local/temporary variables used in a kernel or OpenCL `private` memory will be allocated into this space.

Each Unified Store consists of several 128-bit registers, with 1280 registers available in each. The actual number of registers available to a thread will vary in practice depending on a number of factors. Generally, using a maximum of 10 registers per thread allows the minimum recommended occupancy of 512 threads per USC.

These 128-bit registers are effectively accessed by a kernel as if they were a vector of four 32-bit registers, so a kernel can use up to 40 32-bit values to hit the recommended utilisation. If a given kernel requires more than this, either the number of resident tasks goes down or the number of threads executed per task does, so that each thread gets more register space. Imagination's compiler only reduces utilisation, as high occupancy is important to hide any latency in the threads.

For very excessive memory allocations above this threshold, utilisation may get to a point where it will not go any lower, and allocations will start spilling to main memory. This has high bandwidth and latency costs, though most applications will never hit this limit. The actual point that this happens is under software control and will vary from device to device.

Note: The compiler will try to reduce Unified Store allocations where possible, though it must balance this with the instruction count use as well. For OpenCL, the exact amount of register usage can be determined with a disassembling compiler, which Imagination can provide under NDA.

Common Store

The Common Store is shared between all ALU Pipes in a given USC and is used for any data that is coherent between threads. Any shared memory (`shared` in OpenGL, `local` in OpenCL) is stored here. Constant memory allocations (`const` or `uniform` in OpenGL, `constant` in OpenCL) that have known sizes at compile time are also stored here. It is also used to store object handles such as image, texture, and sampler states, or pointers to main memory.

Data in the Common Store is allocated across four banks, each representing a column of data that is 128 bits wide, as shown below. Data transfers can be served by multiple banks at once and an entire row of data can be read in a single cycle or written in four cycles. Up to 16 threads in a task can each fetch a 32-bit value in one cycle if they fetch consecutive data from the same row, shown in green below.

Figure 4: Good (16 values per cycle) in green, and worst (one value per cycle) in red, shown on Common Store access.

| | Bank 0 | | | | Bank 1 | | | | Bank 2 | | | | Bank 3 | | | |
|---------------|--------|---|---|---|--------|---|---|---|--------|---|----|----|--------|----|----|----|
| 32-bit offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | | | | | | | | | | | | | | | | |
| 16 | | | | | | | | | | | | | | | | |
| 32 | | | | | | | | | | | | | | | | |
| 48 | | | | | | | | | | | | | | | | |
| 64 | | | | | | | | | | | | | | | | |
| 80 | | | | | | | | | | | | | | | | |
| 96 | | | | | | | | | | | | | | | | |
| 112 | | | | | | | | | | | | | | | | |
| 128 | | | | | | | | | | | | | | | | |

However, if more than one thread tries to access data from the same column as another thread, as shown in red above, it will cause a bank clash as a bank can only serve one such request per cycle. Instead, it will serialise these fetches and it will take one cycle per row to access the same amount of data, slowing down the pipeline. To avoid bank clashes, all data in the common store should be accessed across rows. Maximum throughput is therefore 512 bits/cycle: four 128-bit registers, consisting of four 32-bit values for example, from different banks each cycle.

For instance, imagine a kernel that uses a block of shared memory as a linear array of `float`. In the best case, if the stride between each kernel's access was exactly 32-bit, with each thread fetching a subsequent `float` variable, all 16 threads would be able to fetch one value each in one cycle. The worst case scenario is that if the stride between each kernel was 512-bit, with kernels trying to access every 16th

float in the array, the shader would need 16 cycles, one for each value, to complete the fetch. Formally, we could express the rules as:

- Four banks of 128-bit registers
- Each read can fetch an entire 128-bit register
- Each 128-bit register can hold up to four 32-bit values
- Each read takes one cycle
- Registers from the same bank cannot be read from in the same cycle

Some examples are shown below:

```
//OpenGL compute
shared float common[SIZE];

int gid = get_global_id(0);

//Best scenario : 32-bit stride(one float)
//100% throughput (512 bit/cycle), no clashes. One 32-bit value read/thread in one cycle.
float temp = common[gid];

//Worst scenario: 512 bit (16 float or four vec4) stride
//6.25% throughput(32 bit/cycle), 16-way clash. One 32-bit value read/thread in 16 cycles
float temp = common[gid * 16];

//Normal scenario : 128-bit stride (four float or one vec4), requesting 128 bits/thread
//100% throughput (512 bit/cycle), four-way clash for 16 threads
//throughput. Four 32-bit values read/thread in four cycles
vec4 temp = vec4(common[gid*4],common[gid*4+1], common[gid*4+2],common [gid*4+3]);

//OpenCL
local float common[SIZE];
int gid = gl_InvocationId.x;

//Best scenario : 32-bit stride (one float)
//100% throughput (512 bit/cycle), no clashes. One 32-bit value read/thread in one cycle.
float temp = common[gid];

//Worst scenario: 512 bit stride (16 float or four float4)
//6.25% throughput(32 bit/cycle), 16-way clash. One 32-bit value read/thread in 16 cycles
float temp = common[gid * 16];

//Normal scenario : 128-bit stride (four floats or one float4), requesting 128 bits/thread
//100% throughput (512 bit/cycle), four-way clash for 16 threads because we have exceeded max
//throughput. Four 32-bit values read/thread in four cycles
float4 temp = (float4 )(common[gid*4],common[gid*4+1], common[gid*4+2],common [gid*4+3]);
```

The rule to avoid these scenarios is simple. At any one time, threads should read consecutive values at the same time without any gaps. The most common way one could accidentally enter that scenario would be having blocks of data per thread, instead of interleaved data.

A good way to iterate an array that contains multiple values for each thread would be:

```
float x = index * stride_per_index + thread_id
```

Conversely, the following is very likely to cause multi-way bank clashes and waste bandwidth:

```
float x = thread_id * stride_per_thread + index
```

System Memory Access

System memory is the Random Access Memory (RAM) on the device outside of the Graphics Core, used by all system components. This is the type of memory that is referred to when discussing "memory bandwidth" in general. As many different subsystems need to access this memory, accesses have to be scheduled between them, increasing the latency of memory fetches.

OpenCL memory marked as `global`, OpenCL `constant` memory with a size not known at compile time, images, OpenGL Shader Storage Buffer Objects, and RenderScript data in Allocators are stored in System Memory.

It is important to note that the System Memory budget is limited and shared between all resources. The performance of many applications will be limited by this resource, making it a prime target for optimisation.

Rogue hardware can perform burst transfers on data transfers to and from system memory. This allows multiple bytes of data to be transferred in one go without waiting for other devices. The more bytes that can be transferred in one burst, the lower the total latency of all memory accesses.

System Memory transactions are cached. A cache line in the L1 or System Level Cache is 128 bytes and maps to a 64-byte aligned segment in system memory. Transactions larger than 128 bytes are broken down into multiple cache-line requests that are issued independently. A cache line request is serviced at either the throughput of the L1 or L2 cache, in case of a cache hit, or otherwise at the throughput of system memory.

The hardware resolves System Memory accesses in row order for a given task's threads, so accessing data in row order will increase cache locality. Consider the instance of a task containing 16 threads that all perform an 8-byte memory transfer, and the transfers resolve to accesses that all fall within the same aligned 128-byte cache line. The access latency in that case would be the sum of the time taken to service a single cache miss, plus the time to service a further 15 cache hits.

This memory can be read and written in transactions of one, two, four, or multiples of four bytes, and must be aligned to on-chip memory appropriately. One-byte transactions are byte-aligned, two-byte transactions must be 2-byte aligned, and all other transactions must be 4-byte aligned. For best performance, data should generally be accessed from the kernel in this way. The compiler will split an unaligned transfer into multiple aligned transfers, which will be slower.

If a kernel performs all of its reads and writes at the same location in a kernel, it is possible for the compiler to combine multiple reads and writes into a single burst transaction. A burst transfer among the Unified Store and Global memory can transfer a maximum of 64 bytes at once. This includes any reads or writes within the shader or kernel between the system memory and temporaries or OpenCL private memory. In order to perform these burst transfers, global memory must be aligned to four bytes, which normally includes all data types or vectors equal to or larger than 32 bits. Examples of such are `int`, `int2`, `int4`, `float`, `float2`, `float4`, `double`, `short2`, `char4`, but not `char`, `char2`, `bool`, `short`.

Burst transfers between System Memory and the Common Store can transfer different amounts of data if the following conditions are met. In the case of OpenCL, this depends on the size of the work-group.

- Transfer memory between `local` and `global` memory in any direction
- A work-group's size is greater than or equal to 17 and specified at compile time
- The OpenCL function `async_work_group_copy` is used

Then, a different burst transfer to the common store can be executed, which provides a maximum of 4096 bytes across the work-group with a single instruction. In the typical cases of preloading data into shared memory, this path should always be preferred. Otherwise, burst transfers to the Common Store (shared memory) can still be used efficiently, but not as much as the above method.

USC Pair

Each USC pair contains two USCs and a Texture Processing Unit (TPU) to allow the most efficient balance between texture access requirements and compute density. The USC is the main processing element in the Rogue architecture.

Texture Processing Unit

The TPU is a specialised piece of hardware used to accelerate access to images and textures from within kernel code. It handles image reads directly with its own cache to ensure that image data transfers are as fast as possible. This hardware exists to better handle the typically huge size of images and the specialised access required by most image processing algorithms.

L1 Mixed Cache

The L1 Mixed Cache is the main cache used by the Rogue architecture and all data transfers to and from memory go through here first. If this cache cannot serve a data fetch request, the request is passed to the System Level Cache.

System Level Cache

The System Level Cache interacts directly with the System RAM and is the last chance for a data fetch to hit a cache. Data fetches that miss this cache are fetched from System RAM.

4. Performance Guidelines

Recommendations to optimise application performance on PowerVR Rogue.

PowerVR Rogue Graphics Cores are capable of efficiently executing a range of parallel processing algorithms. To achieve the best performance, it is important to follow the guidelines throughout this section as closely as possible. These guidelines are based on four main strategies:

- *Execute Tasks on the Most Appropriate Processor*
- *Minimise Bandwidth Usage*
- *Maximise Utilisation and Occupancy*
- *Maximise Instruction Throughput*

For all these strategies, it is always a good idea to profile applications to find out where the bottlenecks in processing are. This will work as a guide as to where to target optimisation efforts and can be used to determine the results of any given optimisation. The PowerVR SDK provides PVRTune as a real-time hardware profiler which supports the visualisation of how the Graphics Core is being used.

Table 3: The methods outlined and their API appropriacy

| Method | Applicable to | | | |
|---|---------------|--------|-----------|--------------|
| | OpenCL | OpenGL | OpenGL ES | RenderScript |
| <i>Create Allocations with Correct USAGE Flags</i> | # | # | # | # |
| <i>Create Shared Memory Objects with CL_ALLOC_HOST_PTR</i> | # | # | # | # |
| <i>Create Buffers with the Correct Usage Hints</i> | # | # | # | # |
| <i>Use Zero-Copy Paths if Possible</i> | # | # | # | # |
| <i>Use Shared/Local Memory Sensibly</i> | # | # | # | # |
| <i>Avoid Barrier Directly after Local or Constant Memory Access</i> | # | # | # | # |
| <i>Use Built-ins for Type Conversions</i> | # | # | # | # |

Execute Tasks on the Most Appropriate Processor

Explaining how correctly assigning tasks improves performance and how to do so.

If the hardware platform includes both a CPU and Graphics Core, an application should be structured so that it exposes both serial and parallel tasks. Serial tasks are

most efficiently executed on a CPU, whereas parallel tasks are good candidates for executing on a Graphics Core.

To maximise system performance, tasks should be running on the CPU and Graphics Core at the same time. For example, using the CPU to prepare the next batch of data while the Graphics Core processes the current batch. The Graphics Core is often a more power efficient processor if the algorithms used can be expressed as a highly parallel task. If the aim is to reduce power consumption, it becomes important to consider whether an algorithm can be expressed as a parallel task for execution on the Graphics Core, letting the CPU idle for this time.

Choose a Suitable Device

Choosing a device varies depending on the API. The developer should consider the work that is to be split between the CPU and Graphics Core as work that would be split into tasks suited for sequential or parallel execution. Tasks for the CPU can be implemented in any language supported by the platform, such as C++ or Java.

In OpenGL and OpenGL ES, tasks for the Graphics Core are normally dispatched as Compute Shaders. There are some cases where suitable alternatives exist and can be successfully implemented. For example, using fragment shaders for image processing when displaying directly to screen or implementing time-evolving physical systems with vertex shaders and Transform Feedback. This document in general assumes compute shaders, as they are by far the most flexible of the three with the biggest area of application.

For OpenCL, the Installable Client Driver (ICD) enables multiple OpenCL drivers to coexist under the same system. If a hardware platform contains OpenCL drivers for other devices such as a CPU or DSP, an application can select between these devices at run-time.

RenderScript dynamically chooses whether a task can run on CPU or Graphics Core at runtime, depending on the workloads of each system and the suitability of the kernel for execution on each. Thus, a developer cannot explicitly choose Graphics Core execution, but by using RenderScript and noting any device restrictions, they give RenderScript the choice to do so.

Identifying and Creating Work for the Graphics Core

The USCs rely on hardware multithreading to maximise use of their ALUs. As described in the [Architecture Overview](#) section, the USC can execute instructions from a pool of resident threads, switching between them with zero-cost.

Keep the USCs Busy

The most common reason a thread is not ready to execute is that its next instruction is a pending memory access, which can span hundreds of cycles of latency. If other threads are ready to execute during this entire period, the memory latency is completely hidden. This results in full use of the USC, eliminating the cost of the memory transfer.

If all threads are waiting for memory accesses, the USC will idle, reducing its efficiency. In other words, the application is bandwidth limited. This usually happens if the ratio of arithmetic to memory instructions in a kernel is low or there are not

enough threads available to hide the latency. Other system operations could also be reducing the bandwidth available.

If an application is proving to be bandwidth limited, it is worth trying to reduce the number of memory fetches in each kernel or favouring run-time calculations over lookup tables. More work items can also be provided to help mask this. Tools such as PVRTune can be used to help detect when this happens.

Avoid Overly Short Kernels with Small datasets

For a given kernel execution, the first few threads often run with reduced efficiency due to the time taken to allocate work to the USCs. The last few executions also often run with reduced efficiency as the pool of resident tasks has emptied. The more work that is done per execution, the lower the percentage of this cost. This can be achieved by:

- Using somewhat longer kernels, which can mask out the latency of the scheduling/descheduling
- Working on larger datasets, which create more threads per execution that can all be scheduled together. Typically, this should be in the order of several thousands to millions of data points.

If an algorithm has a small kernel length or works on a small dataset, it may be difficult to achieve high USC efficiency. In this case, the algorithm may be more suited for execution on a CPU.

Sharing the Graphics Core between Compute and Graphics Tasks

A long-running kernel can starve system components that rely on 3D graphics processing, such as a user interface. In this situation, a long-running kernel can be split into multiple kernels executed one after another. This typically requires the first kernel to write some data to global memory and the second to read this data back, which will introduce additional bandwidth usage.

Minimise Bandwidth Usage

Explaining how reducing memory access can improve performance and how to do so.

If an application does not need to access memory or move data around, then it should not do so. Several changes can allow an application to avoid this depending on the algorithm used. Changes such as doing calculations at run-time instead of using a lookup or reducing the number of memory fetches for an image filter can help greatly. Other optimisations can be performed for Rogue Graphics Cores as explained over the following sections.

Avoid Redundant Copies

One of the first steps in reducing memory throughput for an application is to ensure that hardware components using the same memory all access the same data without any intermediate copying required. Examples of hardware components include the CPU, Graphics Core, camera interfaces, and video decoders.

Create Allocations with Correct `USAGE` Flags

Rogue devices are capable of allocating memory and then mapping the allocation to a CPU pointer but cannot map arbitrary host-allocated memory into the Graphics Core. In RenderScript, device memory models are treated semi-opaquely with no explicit indication to the developer whether memory is allocated for the CPU or Graphics Core. Memory spaces are defined for Allocations via "`USAGE`" flags however, with each usage being treated as a potentially separate backing store.

Allocations should be explicitly synchronised across usages each time they are used in a different context to ensure that everything has the same view of memory. Synchronisations can cause copies to occur when memory is allocated in different memory spaces and so should only be used when necessary. If an algorithm can be designed to avoid synchronisations at any point, it should do so.

Create Shared Memory Objects with `CL_ALLOC_HOST_PTR`

In OpenCL, host and device memory models are specified independently from each other, with interaction performed by either explicitly copying data or by sharing regions of a memory object. Rogue devices are capable of allocating memory and then mapping the allocation to a CPU pointer but cannot map arbitrary host-allocated memory into the Graphics Core. Therefore, memory objects that are to be shared should be created using the flag `CL_ALLOC_HOST_PTR`. They must also be accessed using mapping functions `clEnqueueMapBuffer` or `clEnqueueMapImage`, and `clEnqueueUnmapMemObject`.

If an object is created using the flag `CL_USE_HOST_PTR`, prior to enqueueing a kernel, the driver will create a copy of it that can be mapped into the Graphics Core address space. This copy occurs every time data is transferred between the host and device and will increase system bandwidth demand.

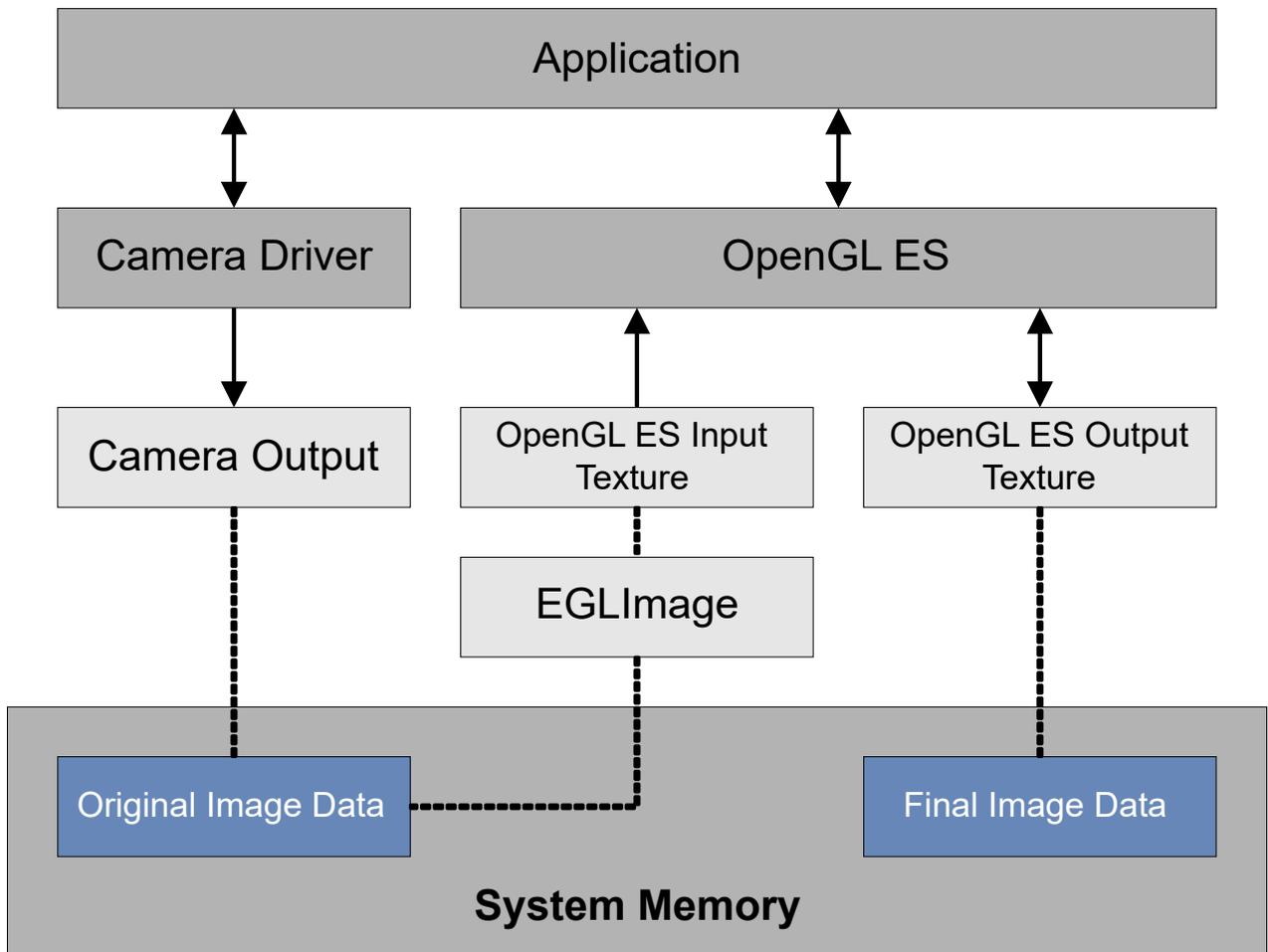
Create Buffers with the Correct Usage Hints

In OpenGL and OpenGL ES, buffer allocation is handled by the driver semi-opaquely with usage hints during buffer allocation. The same recommendations apply as with OpenGL and OpenGL ES and it is therefore necessary to use the hints that are correct for the developer's application. Mapping solutions (`glMapBufferRange`) are preferred to explicit uploads (`glBufferSubData`) when the data is going to be changed frequently.

Use Zero-Copy Paths if Possible

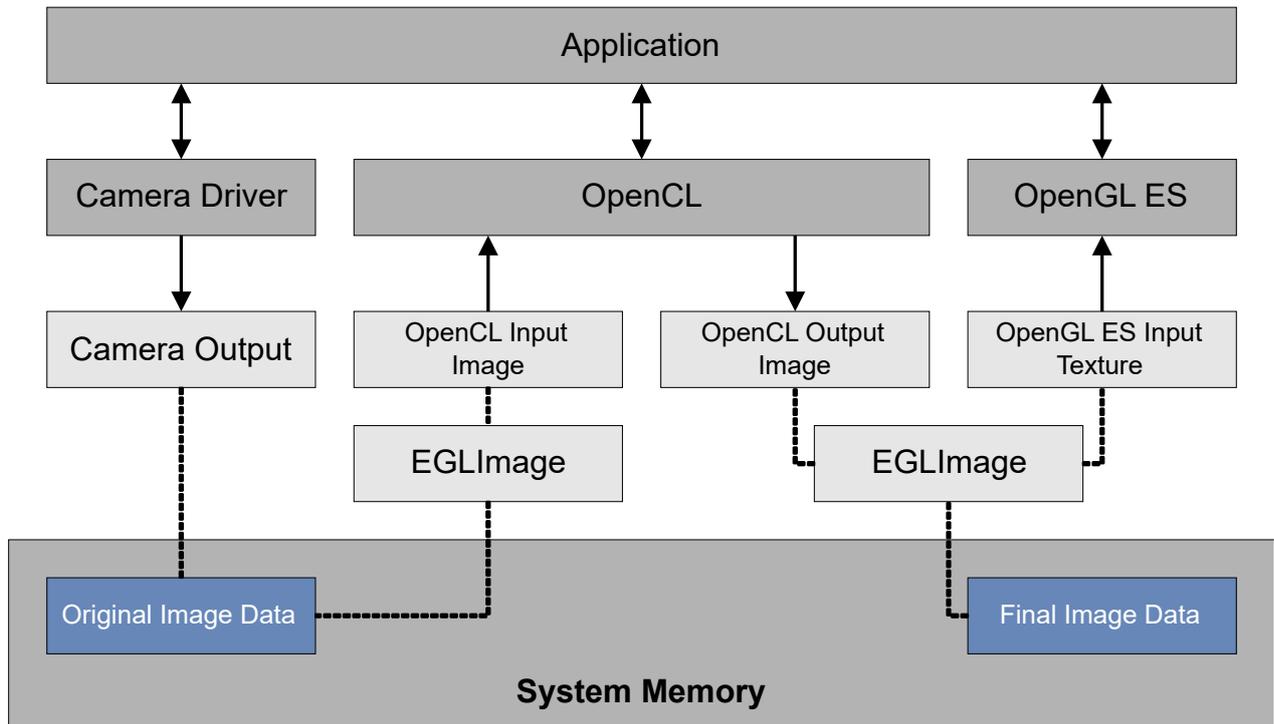
This specific technique is of particular note to interop tasks. When processing input data from an external source such as a camera module, a zero-copy path can be enabled between the camera data and the API image used by the Graphics Core. This is achieved by using `EGLImages`. This typically means directly rendering an image - see the example below for communicating an image from the Camera module.

Figure 5: Zero-copy from a camera to OpenCL and from OpenCL to OpenGL ES



OpenGL and OpenGL ES: If the output image is to be used outside the graphics pipeline, it can also be shared with an `EglImage`. If the data is not required outside the graphics pipeline, it may be worth it to consider directly using a fragment shader for rendering directly to the framebuffer instead of using Image Load/Store.

OpenCL: A zero-copy path can also be enabled between the OpenCL output image and the OpenGL ES input texture used for rendering by using `EGLImages`, as illustrated below. An example of how to do this (OpenCL Example) is included in the PowerVR SDK.

Figure 6: Zero-copy from a camera using `EGLImage`

Group Memory Access Together

The compiler uses several heuristics and can identify memory access patterns in a kernel that can be combined into burst transfers for read or write operations. To allow this, memory accesses should be grouped together as closely as possible to be as easy to identify as possible. Generally, putting reads at the start of a kernel and writes at the end allows for the best efficiency. Accesses to larger data types, such as vectors, also compile into single transfers wherever possible. This means that loading a single `float4` is preferred over four separate float values.

Use Shared/Local Memory Sensibly

Unlike Series5, Rogue devices have a good amount of shared memory available to make good use of work-groups. Kernels should generally pre-fetch data into shared memory if it is going to be accessed from multiple kernels. This is effectively a software-controlled cache, augmenting the hardware-controlled caches on the device.

Typically, the way to make good use of this is to evaluate how many accesses are going to be needed across work-items, then cache these values into shared memory at the start of a kernel. This work should ideally be done with explicit commands where available, such as OpenCL `async_work_group_copy`. Or the work can be split between available work items as evenly as possible, with no two threads caching

the same values. Barriers should be used to synchronise all work-items as late as possible, but before using the values. Then, each kernel should proceed as usual, using these cached values.

Access Memory in Row-Major Order

To make best use of the hardware's memory architecture, all memory should be accessed in row-major order across threads. This applies to all memory banks in the architecture, though for subtly different reasons as described in [Memory](#) section. Generally, memory can be considered as linear, even when allocated as an n-dimensional array. This means that two contiguous values in the last dimension are physically next to each other in memory and can often be accessed together in a single transfer. This is much more important when working on mobile devices with limited bandwidth on large data sets.

Note: This applies to most kinds of memory and buffers, but specifically does not apply to texture or image objects. These are optimised for cache efficiency based on n-dimensional access and dedicated hardware handles this. In these cases, it is important to access nearby pixels to ensure cache coherency. For example, a typical two-dimensional texture should generally be accessed in square/neighbouring pixels starting from (0, 0) and increasing. See the [Texture Processing Unit](#) section for more details.

Maximise Utilisation and Occupancy

Explaining how to optimise the amount of work done and the benefits.

Utilisation is the efficiency of execution for each task with regards to the number of ALU Pipes being used at any given time. If a given task does not have a full 32 threads performing useful work, then available processing power is being wasted and utilisation is low. Developers should always strive to keep hardware busy.

Occupancy refers to the number of tasks that are queued and ready for execution in a USC. The FGS can swap tasks seamlessly if a task is already resident, which hides any latency that would otherwise be caused by stalls in a task that is currently running. The [Execution](#) section gives more details on how the hardware manages tasks and threads.

Work on Large datasets

The most important part of ensuring high utilisation and occupancy is to ensure there are enough data points to make best use of the hardware. Datasets larger than about 512 items per USC on the device typically provide enough work to maintain high utilisation and occupancy, with larger numbers of items increasing efficiency further.

Aim for a dataset Size that is a Multiple of 32

A dataset with a size that is a multiple of 32 is the best opportunity for full utilisation. In the case of sufficiently large datasets, the difference in utilisation should be negligible. However, care needs to be taken when doing this to ensure that either filler data does not affect the final result or with code in the kernel. For OpenCL, a kernel's global work size can be set to a padded size, whilst the actual

dataset is kept at its original size. In this case, the kernel must be careful to avoid out of bounds memory accesses.

Choose an Appropriate Work-Group Size

The number of ALU Pipes active in any given cycle depends on the number of threads assigned to each task. A task consists of 32 threads and a few things affect whether these are all used fully or not. In OpenCL and OpenGL ES Compute, a developer has precise control over this by choosing a work-group size, so it should be chosen carefully.

Aim for a Work-Group Size of 32

Ideally, a work-group size of 32 is perfect for execution on Rogue hardware, as this exactly matches the size of a task. This ideal situation is not always possible, depending on the size of the total dataset actually being worked on and the local problem space. Suggested work-group sizes are listed below, in order of decreasing efficiency:

- 32
- Multiples of 32, up to and including 512 - which is practically ideal
- 16, eight, or four, specified at compile time - which is only recommended if no alternative exists
- Any other size, though this should be avoided - will have several threads per work-group idle
- Two, though again, this is ideally avoided - half of the threads will be idle

For a discussion on exactly how the compiler handles work-group sizes, see the section concerning [Execution](#).

Tell the Compiler the Work-Group Size

Applications should always tell the compiler how large a work-group a kernel expects to work on.

For OpenGL and OpenGL ES this means using `layout(local_size_x=X, local_size_y=Y, local_size_z=Z)`.

For OpenCL this means using `__attribute__((reqd_work_group_size(X, Y, Z)))`.

By knowing the size, the compiler can better allocate on-chip resources, such as shared memory, and perform targeted optimisations at compile time. Deferring the choice until the kernel is enqueued or letting the runtime choose misses out on these opportunities.

In the case where a kernel is intended for use with multiple different work-group sizes, compiling the kernel multiple times with different work-group sizes is usually a much better option than deferring the choice to runtime. The performance gains can be quite significant in some cases and is usually worth the additional application complexity.

For work-group sizes of 16, eight, or four where barrier synchronisation is used, full utilisation can only be achieved if this attribute is specified.

Reduce Unified Store Usage (Private Memory)

The USC's Unified Stores are shared between all work-items that are resident on a USC at any given time and are used to store any temporary variables or OpenCL `private` memory allocations, and in the kernel.

The compiler prioritises high occupancy over high utilisation and will reduce the utilisation of each task based on private memory only. Having at least 512 threads resident per USC provides the best opportunity to hide any data fetch latency whilst maximising utilisation. This equates to a maximum of 40 scalar values used by the kernel.

If a kernel uses less private memory than this, more tasks can be made resident on most cores. The compiler will attempt to optimise unified store private memory use where possible, reducing the final amount. Under NDA, Imagination can provide a compiler which gives exact values for private register usage.

Reduce Common Store Usage (Shared/Local Memory)

`Shared` (OpenCL `local`) memory is extremely useful as it can act as a software-controlled caching mechanism, thus reducing bandwidth requirements in an application, as discussed in [Group Memory Access Together](#). A lot of optimisations usually start by caching into this store.

However, there are limits to how much can be used before affecting occupancy. As with Unified memory, resident work-groups share local memory on a USC. The more work-groups that can fit within the maximum available local memory, the greater the opportunity the core has to hide latencies in kernel execution.

Be Aware of Padding

To improve data access speed, all data types less than 32-bits are padded to 32-bits in both the Unified and Common Stores. This means that a `char` or `short` takes up 32-bits of register space.

Note: When querying the amount of `local` memory available to an OpenCL device, the number returned is how many 32-bit values can be stored, which applies equally to `float`, `int`, or `char` values.

For some kernels, this padding means that register use may start going beyond best performance limits, as discussed previously, whereas it would not if they were not padded. In these cases, it may be useful to manually pack or unpack values from one or more 32-bit types. This allows values to be stored for their lifetime as smaller values and only being expanded when needed. There is a computation cost associated with this, so care should be taken.

Maximise Instruction Throughput

Explaining how to improve the rate of processing.

To get the best performance, the hardware needs to do as many calculations as it can in as short a time as possible. Ensuring the right algorithm is used is the most important thing to do here, as detailed in [Identifying and Creating Work for](#)

the Graphics Core. Once the correct algorithm is chosen, several optimisations can further improve performance, as detailed below.

Trade Precision for Speed

Unless specifically needed for a particular scenario, precision should liberally be traded for speed. This minimises the use of arithmetic instructions with low throughput, as long as care is taken to ensure there is enough precision to produce acceptable results.

For OpenGL and OpenGL ES, consider using `mediump` floats whenever practical. These may have optimised paths by requiring less precision, resulting in some reasonable gains in some common instructions.

For OpenCL kernels, always use the `-cl-fast-relaxed-math` build option, as this enables aggressive compiler optimisations for floating-point arithmetic which the standard otherwise disallows. This flag will often result in good performance gains at the cost of very little precision. `native_*` math built-ins should also be used, as they are much faster than their un-prefixed variants for the cost of some arithmetic precision.

For RenderScript, enable the pragma operation `#pragma rs_fp_relaxed` to enable some optimisations or `#pragma rs_fp_imprecise` to allow the compiler to perform more aggressive optimisations.

Tweak Work for the Graphics Core

The PowerVR Graphics Core has an impressive arithmetic throughput and fully supports integer and bitwise. Nevertheless, it is still beneficial to avoid instructions that are excessively long or complicated. It is important to remember that all Graphics Cores are optimised for floating point calculations and may achieve better throughputs with them.

The integer path is reasonably fast, especially in combinations of arithmetic (except division) bit-shifts. When tests are needed, it may be possible to process several of those per cycle.

That said, it will normally be faster still to work with floating point. Ideally, the numbers should be `float` in the first place. For kernels with a lot of arithmetic on comparatively few values per kernel, it may be beneficial to even turn integer values into `float` and back to integers after calculating. If in doubt, use a profiling compiler as these kinds of gains will be reflected in the cycle count of the kernel.

Integer division should always be avoided if possible. Even the worst case scenario of casting to `float`, dividing, and casting back to an integer will be a lot faster than true integer division. The caveat of that is the reduced range of input; not all integers can be represented by `float`, hence the results will be accurate for a specific range. The integer numbers that are exactly representable by a 32-bit float are up to roughly 16.7M. However, the reciprocal involved may cost extra accuracy, so be wary with numbers even in the low millions.

Avoid Excessive Flow Control

Any instruction that changes what code is executed based on some condition is considered to be flow control. Any flow control statements (`if`, `?`, `switch`, `do`, `while`, `for`) in a kernel may cause dynamic branching as discussed in the [Divergence - Dynamic Branching](#) section.

PowerVR Rogue is very flexible and efficient with typical branching, but a branch is always a branch and inherently inefficient in a parallel environment. Branching should be avoided where practical, either with compile-time conditionals using the pre-processor or by replacing them with an arithmetic statement instead. Note that when conditionals are used to decide whether to access memory, the bandwidth savings will normally far outweigh the cost of the branch.

Also note that when very few instructions are present in any path of a branch statement, predicates will often be used instead, which results in fewer instructions used overall. Whether this happens or not can be seen when using a disassembling compiler, such as PVRShaderEditor.

As an alternative to using actual branches, it is often possible to use built-in functions such as `clamp`, `min`, `max`, or even casting a `bool` to a `float` or `int` value and performing arithmetic. Replacing branches with arithmetic can be tricky, but in some cases can result in decent performance boosts when done well, as these do not cause any kind of divergence among threads.

Avoid Barrier Directly after Local or Constant Memory Access

The hardware may have a latency of up to two cycles for calculating an array index and using it to fetch shared memory from the Common Store (see the [Memory](#) section). The compiler is generally able to hide this latency by rearranging instructions to carry out arithmetic during this time. If a barrier is present immediately after an access, as is often the case, then the compiler typically cannot do this. To allow the compiler to hide the latency, as much arithmetic that does not depend on the result of the access as possible should be performed before the barrier is inserted.

Use Built-ins for Type Conversions

While generally not very expensive and sometimes can be completely hidden, the compiler may sometimes require some additional instructions to convert between different data types. When suitable, explicit pack/unpack or conversion functions should generally be preferred to manual operations.

OpenCL

OpenCL provides a very clear and complete suite of type conversion functions. When they are present, developers should use the `convert_type()` functions. When conversions are required, follow the conventions:

- For conversions from `int` to `float`, round to zero (`rtz`) is fastest.
- For conversions from `uint` to `float`, round to zero (`rtz`) and round to negative infinity (`rtn`) are fastest. Round to positive infinity (`rtp`) is relatively fast, and round to nearest even (`rte`) is slow.

Also useful are the `as_type()` functions, which are analogous to reinterpret casts in C++. These allow a kernel write to access two types that are conceptually the same size (e.g., `int` and `char4`) to be interpreted interchangeably. Types that are not actually the same size on-chip have a higher cost than those that do, as additional instructions are required to extract the correct bits. For example, a `char4` uses 128-bits of register space, whereas `as_type()` treats it as 32-bits. Packing instructions need to be used to compensate, for example, as an `int4` treated as `float4` would typically be a NO-OP.

OpenGL and OpenGL ES

For OpenGL and OpenGL ES, the relevant conversions are a suite of `packXXXXXX` or `unpackXXXXXX`, and generally let developers pack and unpack floating point values relevant to the graphics pipeline. Consult the specific API version for available instructions.

Measuring Peak Performance

Explaining how to evaluate performance.

Throughput, in Operations-Per-Second, is one of the most important factors for performance. The theoretical maximum throughput for an arithmetic operation can be expressed as the following equation:

Each USC has 16 ALU Pipes that run in parallel. Rogue USCs are optimised to maximise the number of scalar floating-point operations that can be processed on every cycle. Each ALU Pipe can use one of the following paths at any time:

- The main unit which can execute up to two instructions in parallel, one of which must be a 32-bit float, the other either a 32-bit float or any integer. The supported native operations include multiply-and-add, multiply, and addition. This unit can also perform packing and test operations on the results of these and a final move operation.
- A 16-bit float Sum-of-Products unit, which can execute two operations of the form $(a * b) OP (c * d)$, where OP is either add, subtract, min, or max.
- A bitwise unit. The supported native operations include logic operations, logical shifts and arithmetic shifts. In one cycle it is possible to perform up to two shifts with a logical operation in between.
- A 32-bit complex floating-point unit that can execute a single operation. The supported native operations include reciprocal, logarithmic, exponent, (cardinal) sine operations, and arc tangent. The complex unit may require more than a single instruction to execute these - for example, requiring range reduction before executing sines. In general, these are many times more efficient than a full software implementation.

The compiler implements other operations as routines on top of these, consuming multiple cycles.

Example Device: 500MHz G6400

The table below shows the theoretical rates of throughput achievable for a 500MHz G6400 device, with four USCs, for various simple operations.

| Data type | Operation | Operations per instruction | Cycles per instruction | Theoretical throughput of 0.5GHz, 4xUSD G6400 |
|--------------|------------------|----------------------------|------------------------|--|
| 16-bit float | Sum-Of-Products | 6 | 1 | $(0.5 \times 4 \times 16 \times 6) \div 1 = \mathbf{192 \text{ GFLOPS}}$ |
| float | Multiply-and-Add | 4 | 1 | $(0.5 \times 4 \times 16 \times 4) \div 1 = \mathbf{128 \text{ GFLOPS}}$ |
| float | Multiply | 2 | 1 | $(0.5 \times 4 \times 16 \times 2) \div 1 = \mathbf{64 \text{ GFLOPS}}$ |

| Data type | Operation | Operations per instruction | Cycles per instruction | Theoretical throughput of 0.5GHz, 4xUSD G6400 |
|-----------|---------------------|----------------------------|------------------------|--|
| float | Add | 2 | 1 | $(0.5 \times 4 \times 16 \times 2) \div 1 = \mathbf{64 \text{ GFLOPS}}$ |
| float | Divide ^A | 1 | 4 | $(0.5 \times 4 \times 16 \times 1) \div 4 = \mathbf{8 \text{ GFLOPS}}$ |
| float | Divide ^B | 1 | 2 | $(0.5 \times 4 \times 16 \times 1) \div 2 = \mathbf{16 \text{ GFLOPS}}$ |
| int | Multiply-and-Add | 2 | 1 | $(0.5 \times 4 \times 16 \times 2) \div 1 = \mathbf{64 \text{ GILOPS}}$ |
| int | Multiply | 1 | 1 | $(0.5 \times 4 \times 16 \times 1) \div 1 = \mathbf{32 \text{ GILOPS}}$ |
| int | Add | 1 | 1 | $(0.5 \times 4 \times 16 \times 1) \div 1 = \mathbf{32 \text{ GILOPS}}$ |
| int | Divide | 1 | 30 | $(0.5 \times 4 \times 16 \times 1) \div 30 = \mathbf{1.07 \text{ GILOPS}}$ |

Note: By default, the compiler implements `float` division as two range reductions, followed by reciprocal and multiplication instructions, requiring four cycles.

Note: For OpenGL GLSL shaders, OpenCL kernels with `-cl-finite-math-only` or `-cl-fast-relaxed-math`, or RenderScript kernels with `#pragma rs_fp_imprecise`, the compiler omits the range reduction, requiring only two cycles.

Measured Throughput

Actual measured peak throughput is likely to be less than the theoretical maximum, due to system overheads such as dynamic clock scaling, system bandwidth, rendering the user interface and driver/hardware setup costs. Actual kernels are also unlikely to generate only arithmetic operations and may include control-flow instructions and memory accesses.

Complex Operations

A list of compute functions and their performance cost.

The table below lists the costs for standard single-precision, floating-point functions. For OpenCL, it assumes the `-cl-fast-relaxed-math` compilation flag.

A low cost represents less than 30 hardware instructions, and a high cost represents more than 100 instructions. 30 to 100 instructions is considered a medium cost.

| Function | Cost |
|----------------------------------|--------|
| <code>x+y</code> | Low |
| <code>x*y</code> | Low |
| <code>x/y</code> | Low |
| <code>1/x</code> | Low |
| <code>rsqrt(x), 1/sqrt(x)</code> | Low |
| <code>sqrt(x)</code> | Low |
| <code>cbrt(x)</code> | Low |
| <code>hypot(x)</code> | Medium |
| <code>exp(x)</code> | Medium |
| <code>exp2(x)</code> | Low |
| <code>exp10(x)</code> | Medium |
| <code>expm1(x)</code> | Medium |
| <code>log(x)</code> | Low |
| <code>log2(x)</code> | Low |
| <code>log10(x)</code> | Low |
| <code>log1p(x)</code> | High |
| <code>sin(x)</code> | Low |
| <code>cos(x)</code> | Low |
| <code>tan(x)</code> | High |
| <code>sincos(x, cptr)</code> | Medium |
| <code>asin(x)</code> | Medium |
| <code>acos(x)</code> | Medium |
| <code>asin(x)</code> | Medium |
| <code>atan(x)</code> | Low |
| <code>atan2(y, x)</code> | Medium |
| <code>sinh(x)</code> | Medium |
| <code>cosh(x)</code> | Medium |
| <code>tanh(x)</code> | Medium |
| <code>asinh(x)</code> | High |
| <code>acosh(x)</code> | High |
| <code>atanh(x)</code> | High |
| <code>pow(x, y)</code> | High |
| <code>fma(x, y, z)</code> | Low |
| <code>frexp(x, exp)</code> | Low |
| <code>ldexp(x, exp)</code> | Medium |

| Function | Cost |
|--------------------|--------|
| logb(x) | Low |
| ilogb(x) | Low |
| fmod(x, y) | Medium |
| remainder(x, y) | High |
| remquo(x, y, iptr) | High |
| modf(x, iptr) | Low |
| fdim(x, y) | Low |
| trunc(x) | Low |
| round(x) | Low |
| rint(x) | Low |
| ceil(x) | Low |
| floor(x) | Low |
| asinpi(x) | Medium |
| acospi(x) | Medium |
| atan2pi(x, y) | Medium |
| atanpi(x) | Low |
| copysign(x, y) | Low |
| sinpi(x) | Medium |
| cospi(x) | Medium |
| tanpi(x) | Medium |
| fabs(x) | Low |
| fmax(x, y) | Low |
| fmin(x, y) | Low |
| fract(x, ptr) | Low |
| hypot(x, y) | Medium |
| ldexp(x, y) | Medium |
| mad(x, y, z) | Low |
| maxmag(x, y) | Low |
| minmag(x, y) | Low |
| nan(x) | Low |
| nextafter(x, y) | Low |
| powr(x, y) | High |
| pown(x, y) | High |
| rootn(x, y) | High |
| signbit(x) | Low |

Optimisation Strategy Cheat Sheet

A brief list of abbreviated points on how to improve performance.

Choose the correct device for the task

- Graphics Core: Large datasets, simple flow, coherent data, large numbers of computations in general
- CPU: complex flow control and branching, one to a few threads of execution, little parallelism, one-shot tasks

Optimise the algorithm for convergence

- Avoid algorithm with complex flow control or threads in a work-group that are idle or exit early
- Define and execute the kernel for maximum data coherency of nearby items
- If having flow control, try to maximise the probability that all threads in a work-group will take the same path

If possible, define a work-group size that is compile-time known and a multiple of 32

- Occupancy, register pressure, and many aggressive optimisations benefit from a work-group size that is known at compiletime and is a multiple of 32
- Hitting both of these conditions maximises benefits but work singularly as well. Compile-time work-group size enables several optimisations, while a work-group that is a multiple of 32 maximises occupancy

Use a large dataset

- Provide enough work for the Graphics Core to schedule so that latency can be hidden
- A few thousands to a few millions should be the target
- Only consider less than a few thousands if the kernel is sufficiently long enough to warrant setting up

Minimise bandwidth / maximise utilisation

- Use shared memory to minimise bandwidth use
- Use a healthy amount of arithmetic instructions interspersed with memory read/writes to hide latency
- Schedule a lot of work simultaneously to assist the scheduler to hide memory fetch latency
- Consider using arithmetic instead of using lookup tables

Maximise occupancy

- Try to use temporary store (unified store) and shared memory (common store) sensibly so as to not hurt occupancy. Registers are not infinite. Up to ~40 32-bit registers per work-item should be sufficient.
- Minimise necessary temporaries.
- Use a healthy amount of arithmetic instructions interspersed with memory read/writes to hide this latency. Consider using arithmetic versus lookups when bandwidth is limited.

Balance minimising bandwidth use with maximising occupancy

- work-group size, shared memory use, and private memory use must all be tweaked together
- Bandwidth should be the first target of optimisation in most applications
- However, too much shared or private memory use may force the compiler to reduce utilisation

Trade precision for speed wherever possible

- Consider using `half` instead of `float` (OpenCL), `mediump` instead of `highp` (OpenGL ES)

Access shared memory sequentially on 2D accesses, in row-major order

- Access sequential elements from kernel instances to maximise shared memory access speed and minimise bank conflicts.
- Do not use stride when accessing values. Stride access in most cases directly increases the number of conflicts. Maximum shared memory speed is achieved by accessing sequential elements.

Access raw global memory linearly and textures in square patterns

- Accessing memory linearly helps cache locality and burst transfers
- Texture hardware is optimised for spatial locality of pixels

Balance the length of the kernel by aiming for a few tens to a few hundreds of hardware instructions

- If the kernel is not long enough, the cost of setting up might end up being a big part of overall execution time.
- Excessively long kernels, with thousands of instructions or long loops, are generally inefficient. They can also starve other system components, notably Graphics.

Use shared objects with API Interoperability to avoid expensive redundant copies

- See what options and extensions are available to the combination of APIs being used, such as `EglImage`
- Using a shared image or buffer and avoiding a round-trip to the CPU will almost always prove a big gain

Use burst transfers

- Help the compiler identify burst transfers by grouping reads and writes together, making them easy to identify. Use explicit block-copy instructions wherever possible, such as the OpenCL `async_work_group_copy`.

Try to hide common-store latency

- If possible, try to have a few unrelated calculations between accesses to shared memory and their corresponding barrier syncs