GPU Compute for Mobile Devices

Jonathan Kirkham

and

Tim Hartley
Introduction
Agenda

Introduction

- Heterogeneous Computing
  Compute API’s
  Thinking in Parallel
  OpenCL
  RenderScript

Mali-T600 Compute Overview

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
Heterogeneous Computing

Using a variety of different types of computational units…

- Usually a CPU and one or more units for specialist computation

- Most common example combines a CPU with the compute capabilities of a GPU
  - CPU best for general purpose program flow
  - GPU best for highly-parallel algorithms on large arrays of data

- Requires different programming models and tools
  - API’s required to pass data and control to specialised processors
  - Parallel programming requires a new way of thinking
GPGPU… What is it Good For? (1)

- Scientific community has pioneered the use of GPU’s for general purpose compute
  - GPU’s highly efficient for structured grid, dense linear algebra, particle methods
  - Also promising for unstructured grids, sparse linear algebra, spectral methods

- As such, mobile computing is likely to benefit from…
  - Image processing
  - Speech processing
  - Artificial intelligence
  - Games
  - Music
GPGPU… What is it Good For? (2)

- **Consumer entertainment**
  - HQ graphics (e.g. photorealistic ray tracing, custom render pipelines etc.)
  - Intelligent “artificial intelligence” (at last really smart opponents)
  - Novel UI (e.g. gesture, speech, eye controlled)
  - 3D spatialisation of sound effects

- **Advanced image processing**
  - Computational photography (e.g. region based focussing)
  - High dynamic range (HDR) photography
  - Augmented reality (e.g. head-up navigation)
  - Computer vision
  - Eye tracking
Physics Demo

- Spring model with 6,000 vertices

- OpenCL version:
  - 8x times faster and twice the number of vertices
  - Single digit CPU load

- Multithreaded C version:
  - 100% CPU load
Face Detection Case Study

- Initial investigation focused on face detection application accelerated using OpenCL

![Graph showing processing time vs. number of windows](image_url)
Introduction

- Heterogeneous Computing
  - Compute API's
    - Thinking in Parallel
    - OpenCL
    - RenderScript

Mali-T600 Compute Overview

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
Compute API’s

- OpenCL™
- DirectCompute
- C++ AMP
- CUDA
- Renderscript
Introduction

Heterogeneous Computing
Compute API's

– Thinking in Parallel

OpenCL
RenderScript

Mali-T600 Compute Overview

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
Thinking in Parallel

- Writing programs to run in a single thread is well-known / straightforward
  - Designing algorithms to run in serial is what we are used to doing
  - Making use of parallel processors requires a different approach

- The skill is to identify which parts of the code can be parallelised
  - Can be a complex process!
  - Often many different ways to do it
  - Some algorithms can’t be parallelised – if possible, find a different algorithm

- Example: Image Processing…
Thinking in Parallel

Traditional serial method...

```c
int x, y;

for (y = 0; y < image_height; y++)
    for (x = 0; x < image_width; x++)
    {
        // process pixel(x, y)
        dest_image[x][y] = sobel(source_image, x, y);
    }
```
Thinking in Parallel

SERIAL

Thread 0
sobel(0,0) sobel(1,0) sobel(2,0) sobel(3,0) ...

Thread 1
sobel(1,0)

Thread 2
sobel(2,0)

Thread 3
sobel(3,0)

Thread (w x h) - 2
sobel(w-2,h-1)

Thread (w x h) - 1
sobel(w-1,h-1)

PARALLEL

(Embarrassingly)
Agenda

Introduction

- Heterogeneous Computing
- Compute API's
- Thinking in Parallel
  - OpenCL
- RenderScript

Mali-T600 Compute Overview

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
OpenCL: Open Computing Language

- An open industry standard for programming a heterogeneous collection of CPUs, GPUs and other computing devices

- Specifies an API for parallel programming

- Designed for GPGPU portability
  - Via abstracted memory and execution model
  - Still scope for specific hardware optimizations
  - No need to know the GPU instruction set

- Programs executing on OpenCL devices are called *kernels*
  - Written in OpenCL C (based on C99)

- Fully supported on Mali-T600 GPU's
Writing OpenCL code

- An OpenCL application consists of two parts:
  - The CL **kernel** that does the parallel processing
    - Runs on the compute devices (the GPU cores)
    - Written in OpenCL C language
  - Application (host) side code that calls the OpenCL APIs
    - Compiles the CL kernel
    - Allocates memory buffers to pass data into and out of the kernel
    - Sets up command queues
    - Sets up dependencies between the tasks
    - Sets up the N-Dimensional Range over which the kernel executes

- Both parts need to be written correctly to get the best performance
OpenCL: Execution Model (1)

- Kernels are programs that run on OpenCL Devices
  - A single execution of a kernel is called a **work-item**
    - A work-item operates on a relatively tiny part of the overall data set
e.g. 1 or more pixels of an image
  - Work-items are grouped into **workgroups**
    - Each workgroup executes on a **Processing Element**
  - There are 1 or more Processing Elements in a **Compute Unit**
  - There are 1 or more Compute Units in an **OpenCL Device**
  - There may be more than one OpenCL Device in a system
OpenCL: Execution Model (2)
OpenCL: Execution Model (3)

- The Host creates a context that includes the following resources
  - The OpenCL device
  - The OpenCL kernels that run on the OpenCL device
    - Program Objects are built from the kernel sources
  - Memory Objects
    - A set of memory buffers visible to the host and the OpenCL Device

- The Host then issues commands into a command queue
  - Kernel execution commands
    - Execute a kernel on the processing elements of a device
  - Memory commands
    - Transfer data to, from or between memory objects
    - Map & unmap memory objects
  - Synchronization commands
OpenCL ND-Range

- ND Ranges: defining coordinates for work items
  - OpenCL kernels (work items) are set up to run in 1, 2 or 3 index space...

```c
size_t global_work_size[work_dim] = { ... };
clEnqueueNDRangeKernel( ... , work_dim, NULL, global_work_size, ... );
```

```c
#define work_dim 1
size_t global_work_size[work_dim] = { 12 };
```

```c
#define work_dim 2
size_t global_work_size[work_dim] = { 8, 8 };
```

```c
#define work_dim 3
size_t global_work_size[work_dim] = { 4, 4, 8 };
```
OpenCL workgroups (1)

- Work items of an NDRange can be divided into equally dimensioned Workgroups...

```c
size_t global_work_size[work_dim] = { ... }; // Number of work items
size_t local_work_size[work_dim] = { ... }; // Workgroup size

clEnqueueNDRangeKernel( ... , work_dim, NULL, global_work_size, local_work_size, ... );
```

```c
#define work_dim 2
size_t global_work_size[work_dim] = { 8, 8 };  
size_t local_work_size[work_dim] = { 4, 4 };  

Total number of workgroups = ( 8 x 8 ) / ( 4 x 4 ) = 4
```

In kernel code...

```c
get_global_id(0) == 5  
get_global_id(1) == 6  
get_local_id(0) == 1  
get_local_id(1) == 2

get_global_size(0) == 8  
get_global_size(1) == 8

get_group_id(0) == 1  
get_group_id(1) == 1  
get_work_dim() == 2

get_local_size(0) == 4  
get_local_size(1) == 4

get_num_groups(0) == 2  
get_num_groups(1) == 2
```
OpenCL workgroups (2)

- Work items within a workgroup have a special relationship with each other...
  - They can perform barrier operations to synchronise execution points...

```
...                                  // Some kernel code
barrier(CLK_LOCAL_MEM_FENCE);       // Wait for all kernels in
...                                  // this workgroup to catch up
...                                  // Some more kernel code

- Work items in the same workgroup have access to shared memory

- Local atomic operations are available (and must be fast)

- Workgroups are totally independent from each other
  - No barriers, no dependencies, no ordering, no coherency
  - However global atomics are available (but may be slow)
OpenCL: Memory Model

- **Private memory**
  - Private to a work-item

- **Local memory**
  - Local to a workgroup
  - Accessible by work-items in that workgroup

- **Constant memory**
  - Accessible as read-only by all work-items

- **Global memory**
  - Accessible by all work-items
C Version...

```c
unsigned char buffer[BUFFER_SIZE];

for (int i = 0; i < BUFFER_SIZE; i++)
    buffer[i] = i % 256;
```

OpenCL Version...

```c
__kernel void fill_buffer( __global unsigned char *buffer )
{
    /* Obtain the ID for this work item - we use this
to locate the corresponding position in the buffer. */
    size_t id = get_global_id(0);

    /* The kernel might be invoked on data points that are outside
    the buffer to honour global and local work space alignments for a
    particular device. Return in these cases. */
    if ( id < BUFFER_SIZE )
    {
        /* Fill the buffer location corresponding to this work item with
        the global ID (modulo 256) */
        buffer[id] = id % 256;
    }
}```
cl_command_queue queue;
cl_context context;
cl_platform_id platform = NULL;
cl_device_id device_id = NULL;
c_int err;
cl_program program;
cl_kernel kernel;
cl_mem buffer;
unsigned char *local_buf;
size_t global_work_size = BUFFER_SIZE;

err = clGetPlatformIDs( 1, &platform, NULL);
err = clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &err );
program = clCreateProgramWithSource( context, 1, &kernel_source_code, NULL, &err );
err = clBuildProgram( program, 1, &device_id, "", NULL, NULL );
kernl = clCreateKernel( program, "fill_buffer", &err );
queue = clCreateCommandQueue( context, device_id, 0, &err );
buffer = clCreateBuffer( context, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, BUFFER_SIZE, NULL, &err );
err = clSetKernelArg( kernel, 0, sizeof(cl_mem), &buffer );
err = clEnqueueNDRangeKernel( queue, kernel, 1, NULL, &global_work_size, NULL, 0, NULL, NULL );
local_buf = clEnqueueMapBuffer( queue, buffer, CL_TRUE, CL_MAP_READ, 0, BUFFER_SIZE, 0, NULL, NULL, &err );

... access buffer contents here
clEnqueueUnmapMemObject( queue, buffer, local_buf, 0, NULL, NULL );
... clean-up OpenCL objects here (kernel, program, queue etc.)
// Evaluate a Bezier patch at a (s, t) determined by the global work item.
__kernel void evaluateBezier
  (__global __write_only float4 *positions,
   __global __write_only float2 *texCoords,
   float16 matrixX, float16 matrixY, float16 matrixZ,
   int size)
{
  float s = ((float)(get_global_id(0))) / (size - 1);
  float s2 = s * s;
  float t = ((float)(get_global_id(1))) / (size - 1);
  float t2 = t * t;
  float4 S = (float4)(s2 * s, s2, s, 1);
  float4 T = (float4)(t2 * t, t2, t, 1);

  // Calculate the position of the item at (s, t).
  float4 smx = (float4)(dot(S, matrixX.lo.lo),
                        dot(S, matrixX.lo.ai),
                        dot(S, matrixX.ai.lo),
                        dot(S, matrixX.ai.ai));

  float4 smy = (float4)(dot(S, matrixY.lo.lo),
                        dot(S, matrixY.lo.ai),
                        dot(S, matrixY.ai.lo),
                        dot(S, matrixY.ai.ai));

  float4 smz = (float4)(dot(S, matrixZ.lo.lo),
                        dot(S, matrixZ.lo.ai),
                        dot(S, matrixZ.ai.lo),
                        dot(S, matrixZ.ai.ai));

  // Output the results.
  int offset = get_global_id(0) + get_global_id(1) * size;
  positions[offset] = (float4)(dot(smx, T), dot(smy, T), dot(smz, T), 1);
  texCoords[offset] = (float2)(s, t);
}
// Evaluate a Bezier patch at a (s, t) determined by the global work item.
__kernel void evaluateBezier
  (_global __write_only float4 *positions,
   _global __write_only float2 *texCoords,
   float16 matrixX, float16 matrixY, float16 matrixZ, int size)
{
  float s = ((float)(get_global_id(0))) / (size - 1);
  float s2 = s * s;
  float t = ((float)(get_global_id(1))) / (size - 1);
  float t2 = t * t;
  float4 S = (float4)(s2 * s, s2, s, 1);
  float4 T = (float4)(t2 * t, t2, t, 1);

  // Calculate the position of the item at (s, t).
  float4 smx = (float4)(dot(S, matrixX.lo.lo),
                       dot(S, matrixX.lo.hi),
                       dot(S, matrixX.hi.lo),
                       dot(S, matrixX.hi.hi));
  float4 smy = (float4)(dot(S, matrixY.lo.lo),
                       dot(S, matrixY.lo.hi),
                       dot(S, matrixY.hi.lo),
                       dot(S, matrixY.hi.hi));
  float4 smz = (float4)(dot(S, matrixZ.lo.lo),
                       dot(S, matrixZ.lo.hi),
                       dot(S, matrixZ.hi.lo),
                       dot(S, matrixZ.hi.hi));

  // Output the results.
  int offset = get_global_id(0) + get_global_id(1) * size;
  positions[offset] = (float4)(dot(smx, T), dot(smy, T), dot(smz, T), 1);
  texCoords[offset] = (float2)(s, t);
}
Introduction

Heterogeneous Computing

Compute API's

Thinking in Parallel

OpenCL

- RenderScript

Mali-T600 Compute Overview

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
RenderScript: Introduction

- An Android-hosted framework
  - Allows Android applications to run specific portions of code on a variety of heterogeneous hardware
  - Provides an API supporting high-performance, vectorised compute operations
  - Cross-platform, device-agnostic and entirely integrated with Android

- Introduced in Android 3.0 (Honeycomb)
  - Originally intended as both a 3D rendering and a compute API
  - 3D rendering API deprecated from Android 4.1 (Jelly Bean)
  - Android 4.2 the first version to support compute on a GPU
  - First RSc GPU-accelerated compute device launched in 2011
    - Nexus 10 (based on Mali-T604)
RenderScript

- RenderScript language based on C99
  - Similarities with OpenCL
    - … though arguably less complex to set up
    - Similar concept of data processed in 1, 2 or 3 dimensions
    - Additional vector data types
  - RSc code is compiled partly offline via Clang and LLVM (into “bitcode”)
    - Bitcode is device-independent
    - Highly optimised at this stage
  - Bitcode bundled as part of the app
    - On-device light-weight JIT compiler converts to native code
    - Compiler also based on LLVM
    - On Mali, integration with T600 compiler to target GPU

- RenderScript programs are “single computational units”
  - Scripts cannot call functions not defined in the same unit
    - … with the exception of functions in the built-in function library
RenderScript: Offline Compiler

foo.rs → llvm-rs-cc

foo.bc → ScriptC_foo.java

Reflection

The “reflected” layer – a set of classes binding RSc with Java and compiled with other application java source files

Bundled with .apk
RenderScript: Online Compiler

- foo.bc
- ScriptC_foo.java
- Application Java source files
- APK file

libbcc

System Library .bc files

Dalvik JIT

GPU, DSP etc.

Multicore CPU
OpenCL Version...

```c
__kernel void fill_buffer( __global unsigned char *buffer )
{
    size_t id = get_global_id(0);
    buffer[id] = id % 256;
}
```

Renderscript Version...

Source code in “simple.rs”...

```c
#pragma version(1)
#pragma rs java_package_name(com.example.simple_rs)

void root( uchar *v_out, uint32_t x )
{
    *v_out = x % 256;
}
```
package com.example.simple_rs;

import android.renderscript.*;
import android.content.res.Resources;

public class ScriptC_simple extends ScriptC {
    private static final String __rs_resource_name = "simple";
    // Constructor
    public ScriptC_simple(RenderScript rs) {
        this(rs,
             rs.getApplicationContext().getResources(),
             rs.getApplicationContext().getResources().getIdentifier(__rs_resource_name, "raw",
             rs.getApplicationContext().getPackageName()));
    }

    public ScriptC_simple(RenderScript rs, Resources resources, int id) {
        super(rs, resources, id);
        __U8 = Element.U8(rs);
    }

    private Element __U8;
    private final static int mExportForEachIdx_root = 0;
    public Script.KernelID getKernelID_root() {
        return createKernelID(mExportForEachIdx_root, 2, null, null);
    }

    public void forEach_root(Allocation aout) {
        // check aout
        if (!aout.getType().getElement().isCompatible(__U8)) {
            throw new RSRuntimeException("Type mismatch with U8!");
        }
        forEach(mExportForEachIdx_root, null, aout, null);
    }
}
Partial source code to setup and call RenderScript

```java
public class simple_RS extends Activity {

    private RenderScript mRS;
    private Allocation mOutAllocation;
    private ScriptC_simple mScript;

    ...

    private void runScript() {
        mRS = RenderScript.create(this);
        mOutAllocation = Allocation.createSized(mRS, Element.U8(mRS), BUFFER_SIZE);
        mScript = new ScriptC_simple(mRS, getResources(), R.raw.simple);
        mScript.forEach_root(mOutAllocation);
    }

    ...
```
Mali-T600 Compute Overview
Agenda

Introduction

Mali-T600 Compute Overview

- OpenCL Execution Model
  - Mali OpenCL Driver
  - OpenCL Built-in Function Library
  - RenderScript Driver

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
CL execution model on Mali-T600 (1)

- **Work Item**
  - Registers, PC, SP, Private stack
  - Barriers, Local Atomics, Cached local memory

- **Work Group**

- **NDRange**
  - Global Atomics, Cached global memory
Each work-item runs as one of the threads within a core

- Every Mali-T600 thread has its own independent program counter
  - …which supports divergent threads from the same kernel
  - caused by conditional execution, variable length loops etc.
- Some other GPGPU’s use “WARP” architectures
  - These share a common program counter with a group of work-items
  - This can be highly scalable… but can be slow handling divergent threads
  - T600 effectively has a Warp size of 1
  - Up to 256 threads per core

- Every thread has its own registers
- Every thread has its own stack pointer and private stack
- Shared read-only registers are used for kernel arguments
CL Execution model on Mali-T600 (3)

- A whole work-group executes on a single core
  - Mali-T600 supports up to 256 work-items per work-group
  - OpenCL barrier operations (which synchronise threads) are handled by the hardware

- For full efficiency you need more work-groups than cores
  - To keep all of the cores fed with work
  - Most GPUs require this, so most CL applications will do this

- Local and global atomic operations are available in hardware

- All memory is cached
Agenda

Introduction

Mali-T600 Compute Overview
  OpenCL Execution Model
    Mali OpenCL Driver
  OpenCL Built-in Function Library
  RenderScript Driver

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
ARM’s OpenCL Driver

- Full profile OpenCL v1.1 in hardware and Mali-T600 driver
  - Backward compatibility support for OpenCL v1.0
  - Embedded profile is a subset of full profile
  - Image types supported in HW and driver
  - Atomic extensions (32 and 64-bit)
  - Hardware is OpenCL v1.2 ready (driver to follow)
Agenda

Introduction

Mali-T600 Compute Overview
- OpenCL Execution Model
- Mali OpenCL Driver
  - OpenCL Built-in Function Library
  - RenderScript Driver

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
CL Built-in function library (BIFL)

- CL language provides a built-in function library

- Functional groups include
  - Maths, integer, common, geometric, work-item, vector, atomic, relational, image

- Fast operation of the BIFLs is essential for CL performance
  - Every kernel calls at least one work-item function:
    - e.g. `get_global_id(0)`

- Functions like dot products are also very commonly used

- There are about 200 functions
  - But many BIFLs are overloaded with data types
    - e.g. `sin(float x), sin(double x), sin(half x)`
  - Also many BIFLs can handle all 6 vector sizes (1,2,3,4,8,16)
  - So there are actually thousands of variants
Agenda

Introduction

Mali-T600 Compute Overview

  OpenCL Execution Model
  Mali OpenCL Driver
  OpenCL Built-in Function Library
  – RenderScript Driver

Optimal OpenCL for Mali-T600

OpenCL Optimization Case Study
RenderScript Driver

- RenderScript programs run on the GPU if they can
  - with automatic fallback to the CPU if not

- Four circumstances cause a RenderScript program to run on the CPU...
  - If a Renderscript accesses a global pointer, the script cannot run on the GPU

```c
float *array;
void root(const float *in, float *out, uint32_t x)
{
    *out = *in + array[x % 5];
}
```

- Memory allocation flags - allocations need to be flagged with USAGE_SCRIPT

```c
Allocation.createTyped(mRS, typeBuilder.create(),
    typeBuilder.create(),
    MipmapControl.MIPMAP_NONE,
    Allocation.USAGE_GRAPHICS_TEXTURE);
```

- Recursive Functions
  - Any use of direct or indirect recursion within functions is incompatible with the GPU

- Debug Functions
Optimal OpenCL for Mali-T600
Introduction

Mali-T600 Compute Overview

Optimal OpenCL for Mali-T600
  - Programming Suggestions
    Tools and Support

OpenCL Optimization Case Study
Porting OpenCL code from other GPUs

- Desktop GPUs require data to be copied to local or private memory buffers
  - Otherwise their performance suffers
  - These copy operations are expensive
  - These are sometimes done in the first part of a kernel, followed by a synchronisation barrier instruction, before the actual processing begins in the second half
  - The barrier instruction is also expensive

- When running on Mali just use global memory instead
  - Thus the copy operations can be removed
  - And also any barrier instructions that wait for the copy to finish
  - Query the device flag `CL_DEVICE_HOST_UNIFIED_MEMORY` if you want to write performance portable code for Mali and desktop PC’s
    - The application can then switch whether or not it performs copying to local memory
Use Vectors

- Mali-T600 series GPUs have a vector capable GPU
- OpenCL supports explicit vector functions
  - `clGetDeviceInfo`
    - `CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR`
    - `CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT`
    - `CL_DEVICE_NATIVE_VECTOR_WIDTH_INT`
    - `CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG`
    - `CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT`
    - `CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE`
    - `CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF`
Hello OpenCL

for (int i = 0; i < arraySize; i++)
{
    output[i] =
        inputA[i] + inputB[i];
}

__kernel void kernel_name(__global int* inputA, __global int* inputB, __global int* output)
{
    int i = get_global_id(0);
    output[i] = inputA[i] + inputB[i];
}

c1EnqueueNDRangeKernel(..., kernel, ..., arraySize, ...)

i, inputA, inputB

i++, inputA, inputB

inputA, inputB

0 1 2 3 ......
for (int i = 0; i < arraySize; i++)
{
    output[i] =
        inputA[i] + inputB[i];
}

__kernel void kernel_name(__global int* inputA,
                           __global int* inputB,
                           __global int* output)
{
    int i = get_global_id(0);
    int4 a = vload4(i, inputA);
    int4 b = vload4(i, inputB);
    vstore4(a + b, i, output);
}

cEnqueueNDRangeKernel(..., kernel, ..., arraySize / 4, ...)
Creating buffers

- The application creates buffer objects that pass data to and from the kernels by calling the OpenCL API `clCreateBuffer`

- All CL memory buffers are allocated in global memory that is physically accessible by both CPU and GPU cores
  - However, only memory that is allocated by `clCreateBuffer` is mapped into both the CPU and GPU virtual memory spaces
  - Memory allocated using `malloc()`, etc, is only mapped onto the CPU

- So calling `clCreateBuffer()` with `CL_MEM_USE_HOST_PTR` and passing in a user created buffer requires the driver to create a new buffer and copy the data (identical to `CL_MEM_COPY_HOST_PTR`)
  - This copy reduces performance

- So where possible always use `CL_MEM_ALLOC_HOST_PTR`
  - This allocates memory that both CPU and GPU can use without a copy
Host data pointers

Buffers created by user (malloc) are not mapped into the GPU memory space

clCreateBuffer(CL_MEM_USE_HOST_PTR) creates a new buffer and copies the data over (but the copy operations are expensive)
Host data pointers

- Where possible don’t use CL_MEM_USE_HOST_PTR
  - Create buffers at the start of your application
  - Use CL_MEM_ALLOC_HOST_PTR instead of malloc()
  - Then you can use the buffer on both CPU host and GPU
Run Time

Where your kernel has no preference for work-group size, for maximum performance...

- either use the compiler recommended work-group size...
  
  `clGetKernelWorkgroupInfo(kernel, dev, CL_KERNEL_WORK_GROUP_SIZE, sizeof(size_t)... );`

- or use a power of 2

- You can pass NULL, but performance might not be optimal

If you want your kernel to access host memory

- use mapping operations in place of read and write operations
- mapping operations do not require copies so are faster and use less memory
**Compiler**

- **Run-time compilation isn’t free!**

- **Compile each kernel only once if possible**
  - If your kernel source is fixed, then compile the kernel during your application’s initialisation
  - If your application has an installation phase then cache the binary on a storage device for the application’s next invocation
  - Keep the resultant binary ready for when you want to run the kernel

- `clBuildProgram` only partially builds the source code
  - If the kernels in use are known at initialization time, then also call `clCreateKernel` for each kernel to initiate the finalizing compile
  - Creating the same kernels in the future will then be faster because the finalized binary is used
BIFLs

- Where possible use the built-in functions as the commonly occurring ones compile to fast hardware instructions
  - Many will target vector versions of the instructions where available

- Using “half” or “native” versions of built-in functions
  - e.g. `half_sin(x)`
    - Specification mandates a minimum of 10-bits of accuracy
  - e.g. `native_sin(x)`
    - Accuracy and input range implementation defined
  - Often not an advantage on Mali-T600… for some functions the precise versions are just as fast
Arithmetic

- Mali-T604 has a register and ALU width of 128-bits
  - Avoid writing kernels that operate on single bytes or scalar values
  - Write kernels that work on vectors of at least 128-bits.
  - Smaller data types are quicker
    - you can fit eight shorts into 128-bits compared to four integers

- Integers and floating point are supported equally quickly
  - Don’t be afraid to use the data type best suited to your algorithm

- Mali-T600 can natively support all CL data types

<table>
<thead>
<tr>
<th>16 x 8-bit chars (char16)</th>
<th>2 x 64-bit integers (long2)</th>
<th>4 x 32-bit floats (float4)</th>
<th>2 x 64-bit floats (double2)</th>
</tr>
</thead>
</table>
Register operations

- All operations can read or write any element or elements within a register
  - e.g. float4 v1, v2;
    ...
    v2.y = v1.x

- All operations can swizzle the elements in their input registers
  - e.g. float4 v1, v2;
    ...
    v2 = v1 + v1.wxzy

- These operations are mostly free, as are various data type expansion and shrinking operations
  - e.g. char -> short
Images

- Image data types are supported in hardware so use them!
  - Supports coordinate clipping, border colours, format conversion, etc
  - Bi-linear pixel read only takes a cycle
  - Happens in the texture pipeline – leaving ALU and L/S pipes free
    - If you don’t use it the texture unit turns off to save power
  - Image stores won’t use the texture unit
    - go through the L/S pipe instead

- However buffers of integer arrays can be even faster still:
  - If you don’t read off the edge of the image, and you use integer coordinates, and you don’t need format conversion then…
    - You can read and operate on 16 x 8-bit greyscale pixels at once
    - Or 4 x RGBA8888 pixels at once
Load/Store Pipeline

- The L1 and L2 caches are not as large as on desktop systems...
  - and there are a great many threads
  - If you do a load in one instruction, by the next instruction (in the same thread) the data could possibly have been evicted
  - So pull as much data into registers in a single instruction as you can
  - One instruction is always better than using several instructions!
  - And a 16-byte load or store will typically take a single cycle
Process large data sets!
- OpenCL setup overhead can limit the GPU over CPU benefit with smaller data sets

Feed the beast!
- The ALU’s work at their most efficient when running lots of compute
  - Don’t be afraid to use a high density of vector calculations in your kernels

Avoid writing kernels that use a large numbers of variables
- Reduces the available registers
  - and therefore the maximum workgroup size reduces
- Sometimes better to re-compute a value than store in a variable

Avoid prime number work size dimensions
- Cannot select an efficient workgroup size with a prime number of work items
- Ideally workgroup size should be a multiple of 4
Agenda

Introduction

Mali-T600 Compute Overview

Optimal OpenCL for Mali-T600
  Programming Suggestions
    – Tools and Support

OpenCL Optimization Case Study
OpenCL Tools and Support

- ARM OpenCL SDK available for download at malideveloper.com
  - Several OpenCL samples and benchmarks

- Debugging
  - Notoriously difficult to do with parallel programming
    - Serial programming paradigms don’t apply
  - DS-5 Streamline compatible with OpenCL
    - Raw instrumentation output also available
  - Mali Graphics Debugger
    - Logs OpenGL ES and OpenCL API calls
    - Download from malideveloper.com
  - OpenCL `printf` function will be available in OpenCL 1.2
OpenCL Optimization Case Study
Laplace filters are typically used in image processing
  - ... often used for edge detection or image sharpening

This case study will go through a number of stages...
  - Demonstrating a variety of optimisation techniques
  - and showing the change in performance at each stage

Our example will process and output 24-bit images
  - and we’ll measure performance across a range of image sizes

But first, a couple of image samples showing the effect of the filter we are using...
OpenCL Laplace Case Study
OpenCL Laplace Case Study
OpenCL Laplace Case Study

-1  -1  -1  -1
-1   9  -1  -1
-1  -1  -1  -1

The Laplace operator is applied to the image using a 3x3 kernel. The kernel is applied to each pixel in the image, and the new pixel value is calculated as the sum of the product of the kernel values and the corresponding pixel values in the original image. The result is an image that appears to be smoother and less detailed than the original.
OpenCL Laplace Case Study

image “stride” = width x 3
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height)
{
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind = 3 * (x + w * y);
        pdst[ind] = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind = 3 * (x + w * y);

    unsigned char blue = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind = 3 * (x + w * (y + 1));
    pdst[ind] = blue;
    pdst[ind + 1] = green;
    pdst[ind + 2] = red;
}
OpenCL Laplace Case Study

```c
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height)
{
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind = 3 * (x + w * y);
        pdst[ind]     = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind = 3 * (x + w * y);

             psrc[ind+3*(2+w)] - psrc[ind+3*2*w] - psrc[ind+3*(1+2*w)] - psrc[ind+3*(2+2*w)];
    gColor = gColor - psrc[ind+1] - psrc[ind+4] - psrc[ind+7] - psrc[ind+3*w+1] + psrc[ind+3*(1+w)+1] * 9 -
             psrc[ind+3*(2+w)+1] - psrc[ind+3*2*w+1] - psrc[ind+3*(1+2*w)+1] - psrc[ind+3*(2+2*w)+1];
             psrc[ind+3*(2+w)+2] - psrc[ind+3*2*w+2] - psrc[ind+3*(1+2*w)+2] - psrc[ind+3*(2+2*w)+2];

    unsigned char blue  = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red   = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind = 3 * (x + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;
    pdst[ind + 2] = red;
}
```
OpenCL Laplace Case Study

```c
#define MAX(a,b) (((a)>(b)?(a):(b))
#define MIN(a,b) (((a)<(b)?(a):(b))

__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height) {
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary) {
        ind = 3 * (x + w * y);
        pdst[ind] = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind = 3 * (x + w * y);


    unsigned char blue = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind = 3 * (x + 1 + w * (y + 1));
    pdst[ind] = blue;
    pdst[ind + 1] = green;
    pdst[ind + 2] = red;
}
```

Boundary checking… ideally we don’t want to calculate for values at the right and bottom edges. (But this might not be the best place to handle this.)
The main calculation... we need to perform this for the red, green and blue color components...
OpenCL Laplace Case Study

```c
#define MAX(a,b) ((a)>(b)?(a):(b))
#define MIN(a,b) ((a)<(b)?(a):(b))

__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height)
{
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = 0;
    int xBoundary = w - 2;
    int yBoundary = h - 2;

    if (x >= xBoundary || y >= yBoundary)
    {
        ind = 3 * (x + w * y);
        pdst[ind] = psrc[ind];
        pdst[ind + 1] = psrc[ind + 1];
        pdst[ind + 2] = psrc[ind + 2];
        return;
    }

    int bColor = 0, gColor = 0, rColor = 0;
    ind = 3 * (x + w * y);


    unsigned char blue = (unsigned char)MAX(MIN(bColor, 255), 0);
    unsigned char green = (unsigned char)MAX(MIN(gColor, 255), 0);
    unsigned char red = (unsigned char)MAX(MIN(rColor, 255), 0);
    ind = 3 * (x + 1 + w * (y + 1));
    pdst[ind] = blue;
    pdst[ind + 1] = green;
    pdst[ind + 2] = red;
}
```

Finally we clamp the results to make sure they lie between 0 and 255... and then write out to the destination...
OpenCL Laplace Case Study

## Results

<table>
<thead>
<tr>
<th>Image</th>
<th>Pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td>768 x 432</td>
<td>331,776</td>
</tr>
<tr>
<td>2560 x 1600</td>
<td>4,096,000</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>4,194,304</td>
</tr>
<tr>
<td>5760 x 3240</td>
<td>18,662,400</td>
</tr>
<tr>
<td>7680 x 4320</td>
<td>33,177,600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mali T604 @ 533MHz</td>
<td>x0.5</td>
</tr>
<tr>
<td>Single A15 @ 1.7GHz</td>
<td>x0.7</td>
</tr>
</tbody>
</table>
OpenCL Laplace Case Study

Use the offline compiler mali_clcc to analyse the kernel

```
psg@psg-mali:~/laplace# mali_clcc -v laplace.cl
Entry point: __llvm2lir_entry_math
8 work registers used, 8 uniform registers used

Pipelines: A / L / T / Overall
Number of instruction words emitted: 54 +31 + 0 = 85
Number of cycles for shortest code path: 3 / 4 / 0 = 4 (L bound)
Number of cycles for longest code path: 25.5 /28 / 0 = 28 (L bound)
Note: The cycle counts do not include possible stalls due to cache misses.
```
### Optimisation 1

- Replace the data fetch (= `psrc[index]`) with `vloadN`
  - Each `vload16` can load 5 pixels at a time (at 3 bytes-per-pixel)
  - This load should complete in a single cycle

- Perform the Laplace calculation as a vector calculation
  - Then Mali works on all 5 pixels at once

- Replace the data store (pdst[index] = ) with `vstoreN`
  - Allows us to write out multiple values at a time
  - Need to be careful to only output 15 bytes (3 pixels)

- As we’ll be running 5 times fewer work items, we’ll need to update the `globalWorkSize` values...

```c
globalWorkSize[0] = image_height;
globalWorkSize[1] = (image_width / 5);
```
From processing 1 pixel…

…to processing 5 pixels…

But we would like to load this data in a way that allows us to efficiently calculate the results in a single vector calculation…
OpenCL Laplace Case Study

3 x overlapping, 16-byte reads from row1 (vload16)...

And the same for rows 2 and 3...
The five pixels can then be computed as follows…
OpenCL Laplace Case Study

```c
__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height) {
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = x * 5 * 3 + w * y * 3;

    uchar16 row1a_ = vload16(0, psrc + ind);
    uchar16 row1b_ = vload16(0, psrc + ind + 3);
    uchar16 row1c_ = vload16(0, psrc + ind + 6);
    uchar16 row2a_ = vload16(0, psrc + ind + (w * 3));
    uchar16 row2b_ = vload16(0, psrc + ind + (w * 3) + 3);
    uchar16 row2c_ = vload16(0, psrc + ind + (w * 3) + 6);
    uchar16 row3a_ = vload16(0, psrc + ind + (w * 6));
    uchar16 row3b_ = vload16(0, psrc + ind + (w * 6) + 3);
    uchar16 row3c_ = vload16(0, psrc + ind + (w * 6) + 6);

    int16 row1a = convert_int16(row1a_);
    int16 row1b = convert_int16(row1b_);
    int16 row1c = convert_int16(row1c_);
    int16 row2a = convert_int16(row2a_);
    int16 row2b = convert_int16(row2b_);
    int16 row2c = convert_int16(row2c_);
    int16 row3a = convert_int16(row3a_);
    int16 row3b = convert_int16(row3b_);
    int16 row3c = convert_int16(row3c_);

    int16 res = (int)0 - row1a - row1b - row1c - row2a - row2b * (int)9 - row2c - row3a - row3b - row3c;
    res = clamp(res, (int16)0, (int16)255);
    uchar16 res_row = convert_uchar16(res);

    vstore8(res_row.s01234567, 0, pdst + ind);
    vstore4(res_row.s89ab, 0, pdst + ind + 8);
    vstore2(res_row.scd, 0, pdst + ind + 12);
    pdst[ind + 14] = res_row.se;
}
```

Parameter 3 now refers to the width of the image / 5.

3 overlapping 16-byte reads for each of the 3 rows (5 pixels-worth in each read)

Convert each 16-byte uchar vector to int16 vectors
OpenCL Laplace Case Study

_kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height) 
{
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = x * 5 * 3 + w * y * 3;

    uchar16 row1a_ = vload16(0, psrc + ind);
    uchar16 row1b_ = vload16(0, psrc + ind + 3);
    uchar16 row1c_ = vload16(0, psrc + ind + 6);
    uchar16 row2a_ = vload16(0, psrc + ind + (w * 3));
    uchar16 row2b_ = vload16(0, psrc + ind + (w * 3) + 3);
    uchar16 row2c_ = vload16(0, psrc + ind + (w * 3) + 6);
    uchar16 row3a_ = vload16(0, psrc + ind + (w * 6));
    uchar16 row3b_ = vload16(0, psrc + ind + (w * 6) + 3);
    uchar16 row3c_ = vload16(0, psrc + ind + (w * 6) + 6);

    int16 row1a = convert_int16(row1a_);
    int16 row1b = convert_int16(row1b_);
    int16 row1c = convert_int16(row1c_);
    int16 row2a = convert_int16(row2a_);
    int16 row2b = convert_int16(row2b_);
    int16 row2c = convert_int16(row2c_);
    int16 row3a = convert_int16(row3a_);
    int16 row3b = convert_int16(row3b_);
    int16 row3c = convert_int16(row3c_);

    int16 res = (int)0 - row1a - row1b - row1c - row2a - row2b * (int)9 - row2c - row3a - row3b - row3c;
    res = clamp(res, (int16)0, (int16)255);
    uchar16 res_row = convert uchar16(res);

    vstore8(res_row.s01234567, 0, pdst + ind);
    vstore4(res_row.s89ab, 0, pdst + ind + 8);
    vstore2(res_row.scd, 0, pdst + ind + 12);
    pdst[ind + 14] = res_row.se;
}
## OpenCL Laplace Case Study

### Optimisation 1 Results

<table>
<thead>
<tr>
<th>Image</th>
<th>Pixels</th>
<th>Original</th>
<th>Opt 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>768 x 432</td>
<td>331,776</td>
<td>x1</td>
<td>x1.4</td>
</tr>
<tr>
<td>2560 x 1600</td>
<td>4,096,000</td>
<td>x1</td>
<td>x4.5</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>4,194,304</td>
<td>x1</td>
<td>x1.7</td>
</tr>
<tr>
<td>5760 x 3240</td>
<td>18,662,400</td>
<td>x1</td>
<td>x6.0</td>
</tr>
<tr>
<td>7680 x 4320</td>
<td>33,177,600</td>
<td>x1</td>
<td>x6.2</td>
</tr>
</tbody>
</table>

- **Work registers:** 8
- **ALU cycles:** 25.5
- **L/S cycles:** 28
- **Opt 1:**
  - x1.4
  - x4.5
  - x1.7
  - x6.0
  - x6.2
OpenCL Laplace Case Study

- **Optimisation 2**

  - We can reduce the number of loads
    - by synthesizing the middle vector row from the left and right rows…

row 1a: \( p_1, p_2, p_3, p_4, p_5 \)
row 1b: \( p_2, p_3, p_4, p_5, p_6 \)
row 1c: \( p_3, p_4, p_5, p_6, p_7 \)

becomes…

row 1a: \( p_1, p_2, p_3, p_4, p_5 \)
row 1c: \( p_3, p_4, p_5, p_6, p_7 \)

row 1b \( \leftarrow \) row1(p2, p3, p4, p5) + row2(p6)
We can reduce the number of loads by synthesizing the middle vector row from the left and right rows…

```c
uchar16 row1a_ = vload16(0, psrc + ind);
uchar16 row1b_ = vload16(0, psrc + ind + 3);
uchar16 row1c_ = vload16(0, psrc + ind + 6);
uchar16 row2a_ = vload16(0, psrc + ind + (w * 3));
uchar16 row2b_ = vload16(0, psrc + ind + (w * 3) + 3);
uchar16 row2c_ = vload16(0, psrc + ind + (w * 3) + 6);
uchar16 row3a_ = vload16(0, psrc + ind + (w * 6));
uchar16 row3b_ = vload16(0, psrc + ind + (w * 6) + 3);
uchar16 row3c_ = vload16(0, psrc + ind + (w * 6) + 6);
```

becomes…

```c
uchar16 row1a_ = vload16(0, psrc + ind);
uchar16 row1c_ = vload16(0, psrc + ind + 6);
uchar16 row1b_ = (uchar16)(row1a_.s3456789a, row1c_.s56789abc);
uchar16 row2a_ = vload16(0, psrc + ind + (w * 3));
uchar16 row2c_ = vload16(0, psrc + ind + (w * 3) + 6);
uchar16 row2b_ = (uchar16)(row2a_.s3456789a, row2c_.s56789abc);
uchar16 row3a_ = vload16(0, psrc + ind + (w * 6));
uchar16 row3b_ = (uchar16)(row3a_.s3456789a, row3c_.s56789abc);
uchar16 row3c_ = vload16(0, psrc + ind + (w * 6) + 6);
```
## Optimisation 2 Results

<table>
<thead>
<tr>
<th>Image</th>
<th>Pixels</th>
<th>Original</th>
<th>Opt 1</th>
<th>Opt 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>768 x 432</td>
<td>331,776</td>
<td>x1</td>
<td>x1.4</td>
<td>x1.4</td>
</tr>
<tr>
<td>2560 x 1600</td>
<td>4,096,000</td>
<td>x1</td>
<td>x4.5</td>
<td>x4.5</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>4,194,304</td>
<td>x1</td>
<td>x1.7</td>
<td>x2.0</td>
</tr>
<tr>
<td>5760 x 3240</td>
<td>18,662,400</td>
<td>x1</td>
<td>x6.0</td>
<td>x6.0</td>
</tr>
<tr>
<td>7680 x 4320</td>
<td>33,177,600</td>
<td>x1</td>
<td>x6.2</td>
<td>x6.3</td>
</tr>
</tbody>
</table>

| Work registers: | 8 | 8+ |
| ALU cycles:     | 25.5 | 22.5 |
| L/S cycles:     | 28 | 13 |

- Vectorize
- Synth. loads
OpenCL Laplace Case Study

- **Optimisation 3**
  - Use `short16` instead of `int16`
    - smaller register use allows for a larger `CL_KERNEL_WORK_GROUP_SIZE` available for kernel execution

```c
int16 row1a = convert_int16(row1a_);
int16 row1b = convert_int16(row1b_);
int16 row1c = convert_int16(row1c_);
int16 row2a = convert_int16(row2a_);
int16 row2b = convert_int16(row2b_);
int16 row2c = convert_int16(row2c_);
int16 row3a = convert_int16(row3a_);
int16 row3b = convert_int16(row3b_);
int16 row3c = convert_int16(row3c_);

int16 res = (int)0 - row1a - row1b - row1c - row2a - row2b * (int)9 - row2c - row3a - row3b - row3c;
res = clamp(res, (int16)0, (int16)255);
uchar16 res_row = convert_uchar16(res);
```

becomes...

```c
short16 row1a = convert_short16(row1a_);
short16 row1b = convert_short16(row1b_);
short16 row1c = convert_short16(row1c_);
short16 row2a = convert_short16(row2a_);
short16 row2b = convert_short16(row2b_);
short16 row2c = convert_short16(row2c_);
short16 row3a = convert_short16(row3a_);
short16 row3b = convert_short16(row3b_);
short16 row3c = convert_short16(row3c_);

short16 res = (short)0 - row1a - row1b - row1c - row2a - row2b * (short)9 - row2c - row3a - row3b - row3c;
res = clamp(res, (short16)0, (short16)255);
uchar16 res_row = convert_uchar16(res);
```
## OpenCL Laplace Case Study

### Optimisation 3 Results

<table>
<thead>
<tr>
<th>Image</th>
<th>Pixels</th>
<th>Original</th>
<th>Opt 1</th>
<th>Opt 2</th>
<th>Opt 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>768 x 432</td>
<td>331,776</td>
<td>x1</td>
<td>x1.4</td>
<td>x1.4</td>
<td>x1.5</td>
</tr>
<tr>
<td>2560 x 1600</td>
<td>4,096,000</td>
<td>x1</td>
<td>x4.5</td>
<td>x4.5</td>
<td>x6.2</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>4,194,304</td>
<td>x1</td>
<td>x1.7</td>
<td>x2.0</td>
<td>x1.9</td>
</tr>
<tr>
<td>5760 x 3240</td>
<td>18,662,400</td>
<td>x1</td>
<td>x6.0</td>
<td>x6.0</td>
<td>x8.5</td>
</tr>
<tr>
<td>7680 x 4320</td>
<td>33,177,600</td>
<td>x1</td>
<td>x6.2</td>
<td>x6.3</td>
<td>x9.0</td>
</tr>
</tbody>
</table>

| Work registers: | 8 | 8+ | 8 | 7 |
| ALU cycles:     | 25.5 | 22.5 | 24.5 | 13.5 |
| L/S cycles:     | 28 | 13 | 8 | 9 |
OpenCL Laplace Case Study

- **Optimisation 4**
  - Try 4-pixels per work-item rather than 5
  - With some image sizes perhaps the driver can optimize more efficiently when 4 pixels are being calculated

```c
__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height)
{
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = x * 5 * 3 + w * y * 3;
    ...
}
```

becomes...

```c
__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int width, int height)
{
    int y = get_global_id(0);
    int x = get_global_id(1);
    int w = width;
    int h = height;
    int ind = x * 4 * 3 + w * y * 3;
    ...
}
```
OpenCL Laplace Case Study

- Optimisation 4
  - And our date write out becomes simpler...

```c
... vstore8(res_row.s01234567, 0, pdst + ind); vstore4(res_row.s89ab, 0, pdst + ind + 8); vstore2(res_row.scd, 0, pdst + ind + 12); pdst[ind + 14] = res_row.se;
```

becomes...

```c
... vstore8(res_row.s01234567, 0, pdst + ind); vstore4(res_row.s89ab, 0, pdst + ind + 8);
```

- ...and we need to adjust the setup code to adjust the work-item count.
### Optimisation 4 Results

<table>
<thead>
<tr>
<th>Image</th>
<th>Pixels</th>
<th>Original</th>
<th>Opt 1</th>
<th>Opt 2</th>
<th>Opt 3</th>
<th>Opt 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>768 x 432</td>
<td>331,776</td>
<td>x1</td>
<td>x1.4</td>
<td>x1.4</td>
<td>x1.5</td>
<td>x1.6</td>
</tr>
<tr>
<td>2560 x 1600</td>
<td>4,096,000</td>
<td>x1</td>
<td>x4.5</td>
<td>x4.5</td>
<td>x6.2</td>
<td>x5.2</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>4,194,304</td>
<td>x1</td>
<td>x1.7</td>
<td>x2.0</td>
<td>x1.9</td>
<td>x5.3</td>
</tr>
<tr>
<td>5760 x 3240</td>
<td>18,662,400</td>
<td>x1</td>
<td>x6.0</td>
<td>x6.0</td>
<td>x8.5</td>
<td>x7.2</td>
</tr>
<tr>
<td>7680 x 4320</td>
<td>33,177,600</td>
<td>x1</td>
<td>x6.2</td>
<td>x6.3</td>
<td>x9.0</td>
<td>x7.5</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Work registers</td>
<td></td>
<td></td>
<td>8</td>
<td>8+</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>ALU cycles</td>
<td></td>
<td></td>
<td>25.5</td>
<td>22.5</td>
<td>24.5</td>
<td>13.5</td>
</tr>
<tr>
<td>L/S cycles</td>
<td></td>
<td></td>
<td>28</td>
<td>13</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

### Performance Analysis
- **Image Size**:
  - 768 x 432
  - 2560 x 1600
  - 2048 x 2048
  - 5760 x 3240
  - 7680 x 4320
- **Optimisation Results**:
  - **Opt 1**: 1.4x to 1.6x
  - **Opt 2**: 4.5x to 5.3x
  - **Opt 3**: 1.5x to 6.3x
  - **Opt 4**: 5.2x to 7.5x
- **Work Registers**:
  - 8 to 8+
  - 8 to 7
- **ALU Cycles**:
  - 25.5 to 13.5
- **L/S Cycles**:
  - 28 to 9
OpenCL Laplace Case Study

- Optimisation 5
  - How about 8 pixels per work-item?
__kernel void math(__global unsigned char *pdst, __global unsigned char *psrc, int w, int h)
{
    const int y = get_global_id(0);
    const int x = get_global_id(1) * 8;
    int ind = (x + w * y) * 3;
    short16 acc_xy;
    short8 acc_z;

    uchar16 l_0 = vload16(0, psrc + ind);
    uchar16 r_0 = vload16(0, psrc + ind + 14);
    short16 a_xy_0 = convert_short16((uchar16)(l_0.s0123456789abcdef));
    short8 a_z_0 = convert_short8((uchar8)(r_0.s23456789));
    short16 b_xy_0 = convert_short16((uchar16)(l_0.s3456789a, l_0.sbcde, r_0.s1234));
    short8 b_z_0 = convert_short8((uchar8)(r_0.s56789abc));
    short16 c_xy_0 = convert_short16((uchar16)(l_0.s6789abcd, r_0.s01234567));
    short8 c_z_0 = convert_short8((uchar8)(r_0.s89abcdef));
    acc_xy = -a_xy_0 - b_xy_0 - c_xy_0;
    acc_z = -a_z_0 - b_z_0 - c_z_0;

    uchar16 l_1 = vload16(0, psrc + ind + (w * 3));
    uchar16 r_1 = vload16(0, psrc + ind + (w * 3) + 14);
    short16 a_xy_1 = convert_short16((uchar16)(l_1.s0123456789abcdef));
    short8 a_z_1 = convert_short8((uchar8)(r_1.s23456789));
    short16 b_xy_1 = convert_short16((uchar16)(l_1.s3456789a, l_0.sbcde, r_0.s1234));
    short8 b_z_1 = convert_short8((uchar8)(r_1.s56789abc));
    short16 c_xy_1 = convert_short16((uchar16)(l_1.s6789abcd, r_0.s01234567));
    short8 c_z_1 = convert_short8((uchar8)(r_1.s89abcdef));
    acc_xy = -a_xy_1 + b_xy_1 * (short)9 - c_xy_1;
    acc_z += -a_z_0 - b_z_0 - c_z_0;

    uchar16 l_2 = vload16(0, psrc + ind + (w * 6));
    uchar16 r_2 = vload16(0, psrc + ind + (w * 6) + 14);
    short16 a_xy_2 = convert_short16((uchar16)(l_2.s0123456789abcdef));
    short8 a_z_2 = convert_short8((uchar8)(r_2.s23456789));
    short16 b_xy_2 = convert_short16((uchar16)(l_2.s3456789a, l_0.sbcde, r_0.s1234));
    short8 b_z_2 = convert_short8((uchar8)(r_2.s56789abc));
    short16 c_xy_2 = convert_short16((uchar16)(l_2.s6789abcd, r_0.s01234567));
    short8 c_z_2 = convert_short8((uchar8)(r_2.s89abcdef));
    acc_xy += -a_xy_2 - b_xy_2 - c_xy_2;
    acc_z += -a_z_0 - b_z_0 - c_z_0;

    short16 res_xy = clamp(acc_xy, (short16)0, (short16)255);
    short8 res_z = clamp(acc_z, (short8)0, (short8)255);

    vstore16(convert uchar16(res_xy), 0, pdst + ind);
    vstore8(convert uchar8(res_z), 0, pdst + ind + 16);
}
## OpenCL Laplace Case Study

### Optimization 5 Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>768 x 432</td>
<td>331,776</td>
<td>x1</td>
<td>x1.4</td>
<td>x1.4</td>
<td>x1.5</td>
<td>x1.6</td>
<td>x1.2</td>
</tr>
<tr>
<td>2560 x 1600</td>
<td>4,096,000</td>
<td>x1</td>
<td>x4.5</td>
<td>x4.5</td>
<td>x6.2</td>
<td>x5.2</td>
<td>x5.6</td>
</tr>
<tr>
<td>2048 x 2048</td>
<td>4,194,304</td>
<td>x1</td>
<td>x1.7</td>
<td>x2.0</td>
<td>x1.9</td>
<td>x5.3</td>
<td>x5.8</td>
</tr>
<tr>
<td>5760 x 3240</td>
<td>18,662,400</td>
<td>x1</td>
<td>x6.0</td>
<td>x6.0</td>
<td>x8.5</td>
<td>x7.2</td>
<td>x8.4</td>
</tr>
<tr>
<td>7680 x 4320</td>
<td>33,177,600</td>
<td>x1</td>
<td>x6.2</td>
<td>x6.3</td>
<td>x9.0</td>
<td>x7.5</td>
<td>x9.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>8+</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>8+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work registers:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU cycles:</td>
<td>25.5</td>
<td>22.5</td>
<td>24.5</td>
<td>13.5</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td>L/S cycles:</td>
<td>28</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>
OpenCL Laplace Case Study

- **Summary**
  - **Original version:** Scalar code
  - **Optimisation 1:** Vectorize
    - Process 5 pixels per work-item
    - Vector loads (vloadn) and vector stores (vstorn)
    - Much better use of the GPU ALU: Up to \textbf{x6.2} performance increase
  - **Optimisation 2:** Synthesised loads
    - Reduce the number of loads by synthesising values
    - Performance increase: up to \textbf{x6.3} over original
  - **Optimisation 3:** Replace int16 with short16
    - Reduces the kernel register count
    - Performance increase: up to \textbf{x9.0} over original
  - **Optimisation 4:** Try 4 pixels per work-item rather than 5
    - Performance increase: up to \textbf{x7.5} over original
      - but it depends on the image size
  - **Optimisation 5:** Try 8 pixels per work-item
    - Performance increase: up to \textbf{x9.1} over original… but a mixed bag.
Jonathan Kirkham
& Tim Hartley

jonathan.kirkham@arm.com
tim.hartley@arm.com
ARM TechCon™
Where Intelligence Connects