GPU Compute for Mobile Devices

Tim Hartley & Johan Gronqvist, ARM

Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

Optimal OpenCL for Mali-T600 / T700

OpenCL Optimization Case Studies

ARM Introduction

- World leading semiconductor IP
  - Founded in 1990
  - 1060 processor licenses sold to more than 350 companies
  - > 10bn ARM-based chips in 2013
  - > 50bn ARM-based chips to date
- Products
  - CPUs
  - Multimedia processors
  - Interconnect
  - Physical IP
- Business model
  - Designing and licensing of IP
  - Not manufacturing or selling on chips

ARM

The Evolution of Mobile GPU Compute

ARM® MALI™
Visual Technology

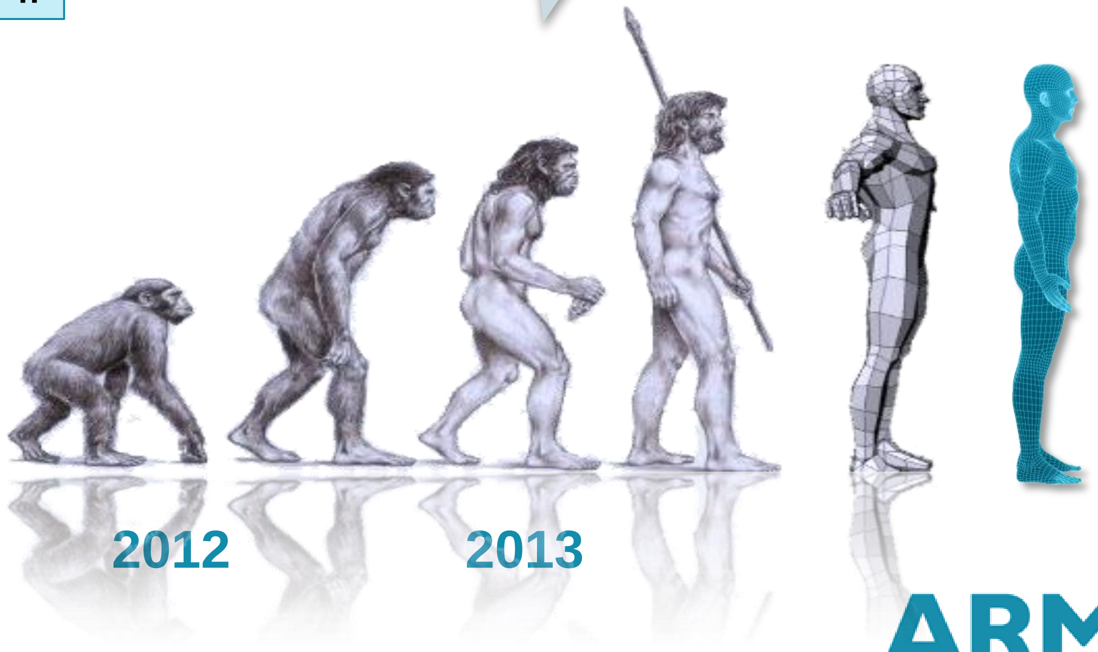
OpenGL ES 3.1 Compute Shaders
GPU Compute within OpenGL ES API

OpenCL™ Full Profile / RenderScript
Portable Heterogeneous Parallel Computation

OpenGL ES 2.0
Programmable pipeline

OpenGL® ES 1.1
Fixed pipeline

- ARM® Mali™-55 GPU
- Mali-200, Mali-300
- Mali-400 MP, Mali-450 MP
- Mali-T600 Series
- Mali-T700 Series



2007 2009 2010 2012 2013

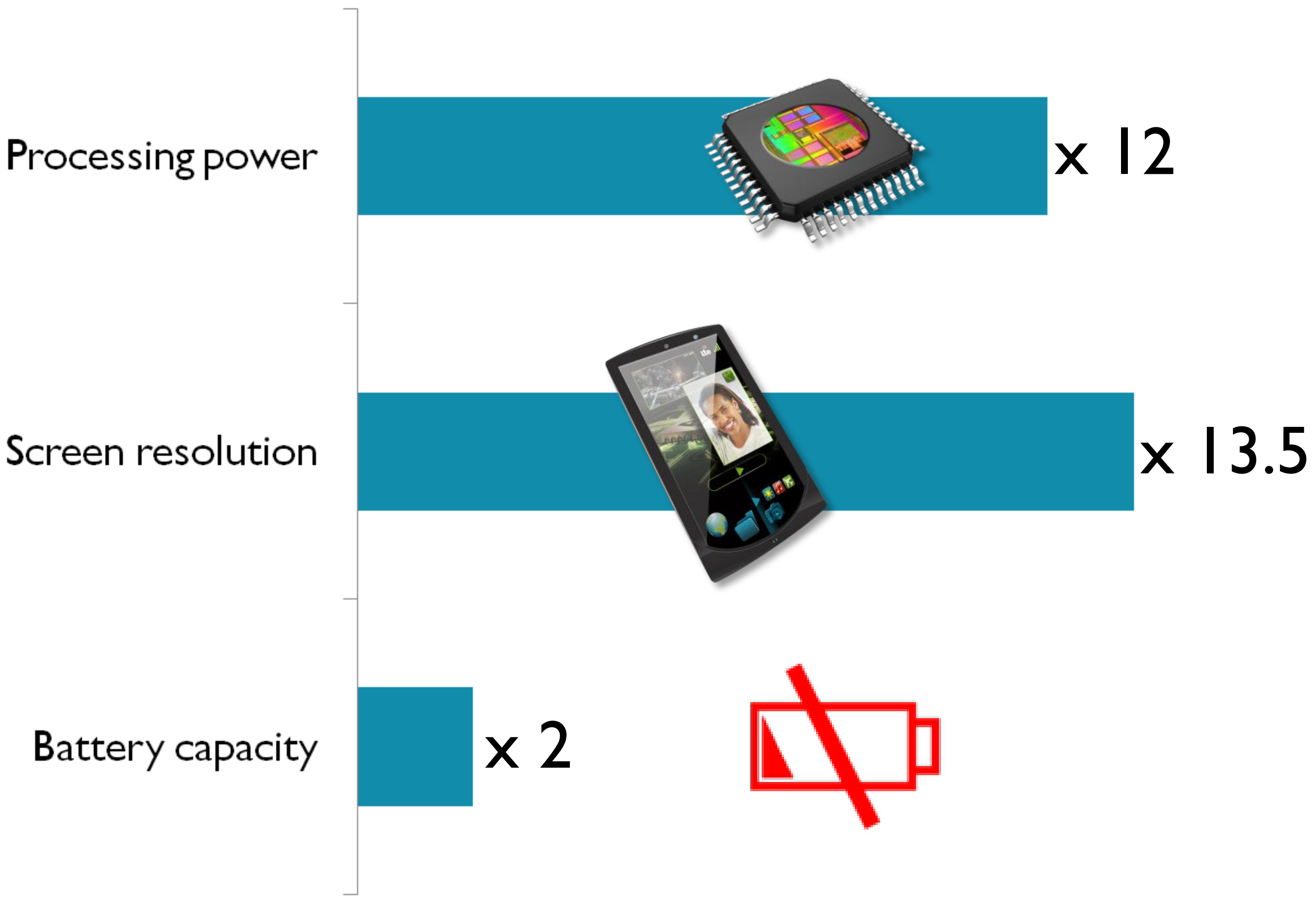
ARM

Mobile Performance: New Challenges need New Solutions

- Processing power outpacing improvements in battery performance
- Processor frequency bound by thermal limitations
- Adding duplicate cores has diminishing returns

Vital to focus on processing efficiency through heterogeneous architectures

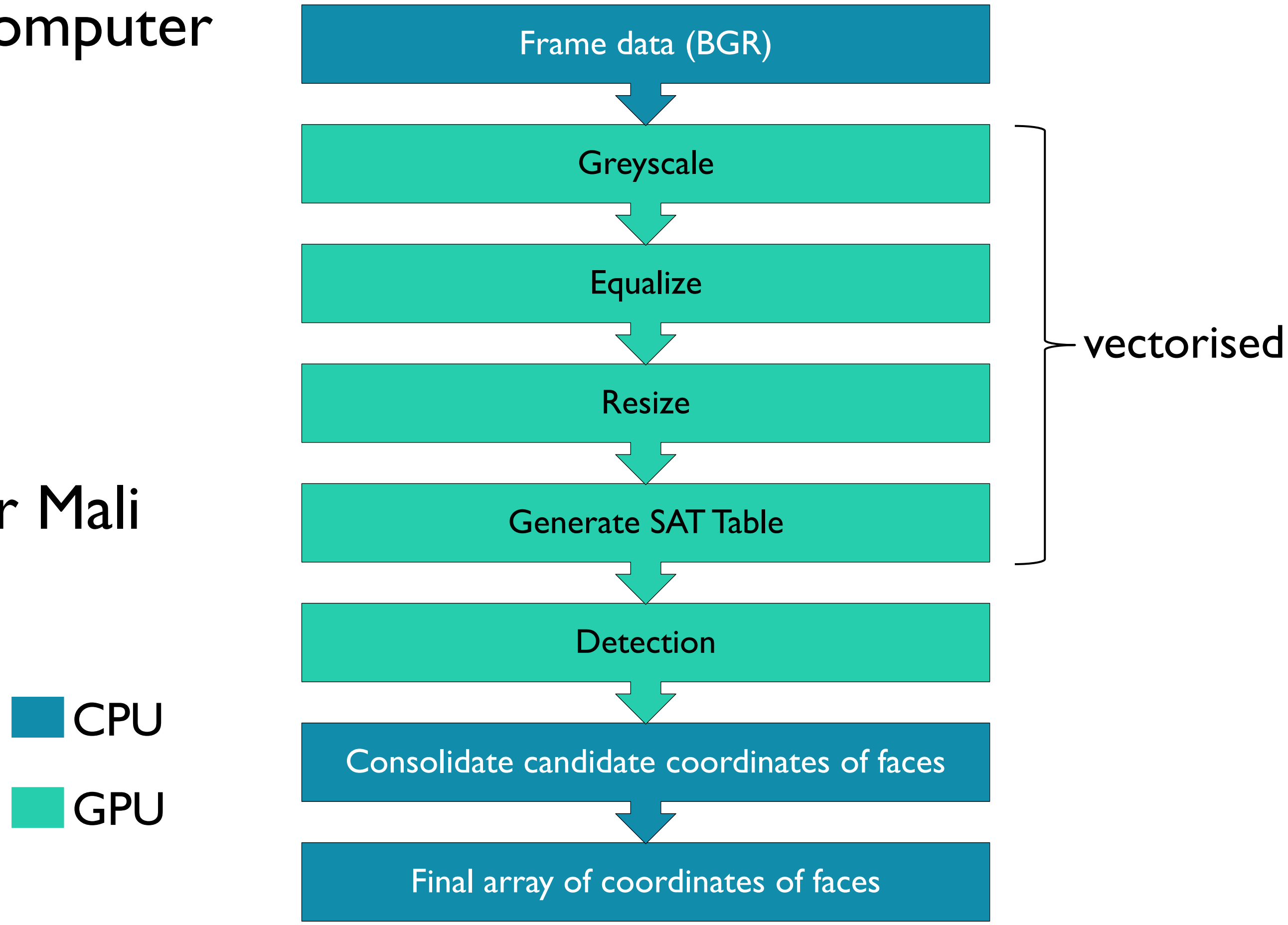
Technological Improvements since 2010



ARM

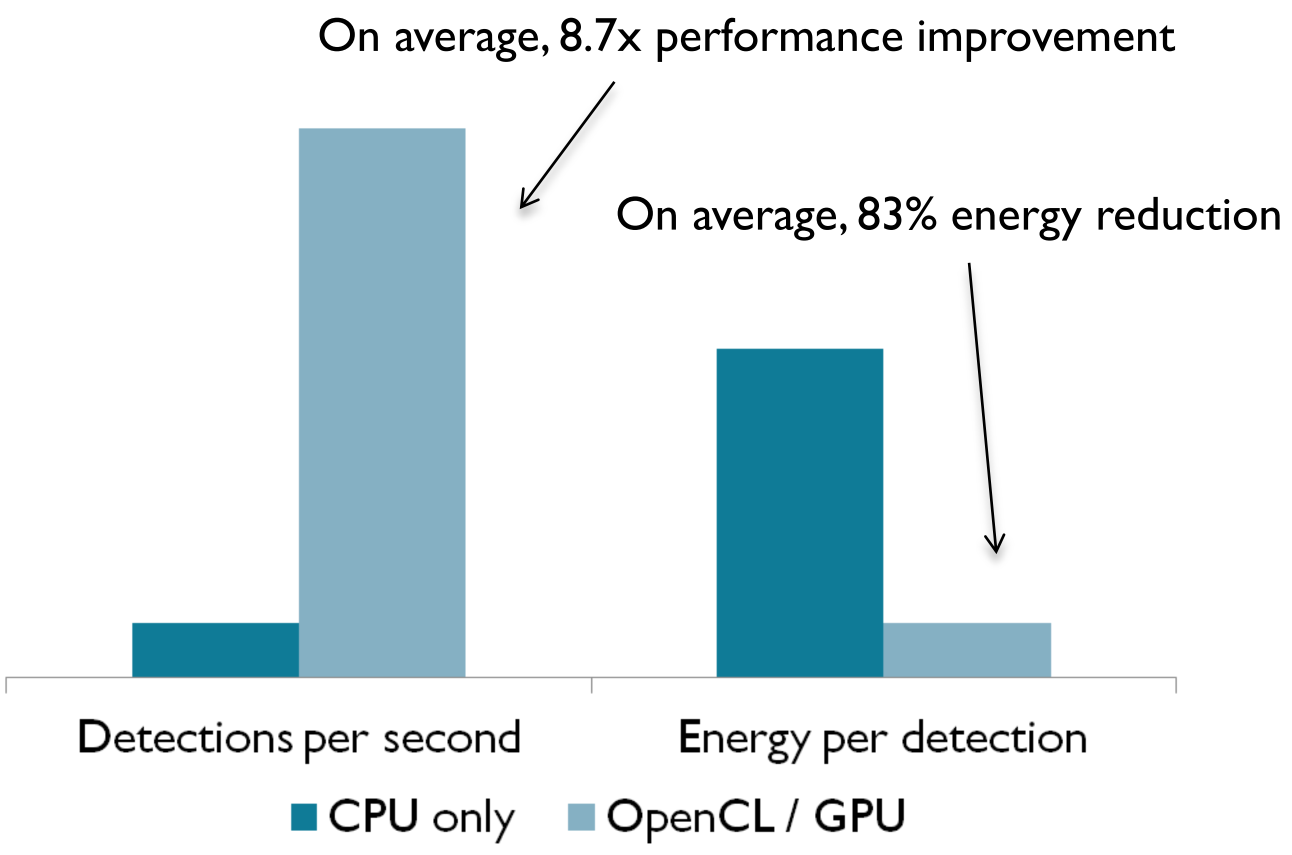
Face Detection Case Study

- Internal demo to explore possibilities of computer vision on mobile
- CPU version from OpenCV library.
  - Single threaded
  - No NEON
- OpenCL version written and optimised for Mali

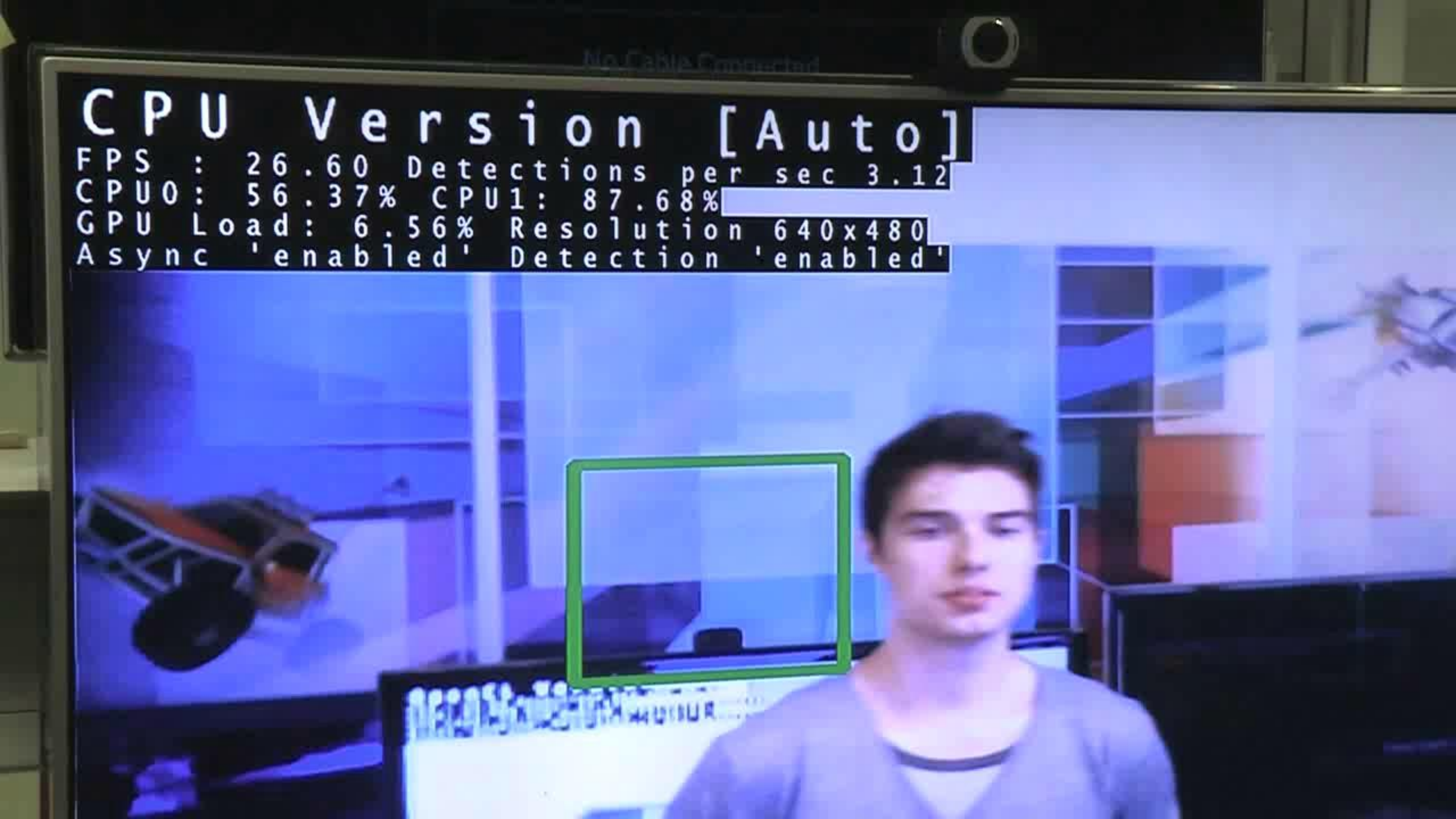


Face Detection Case Study

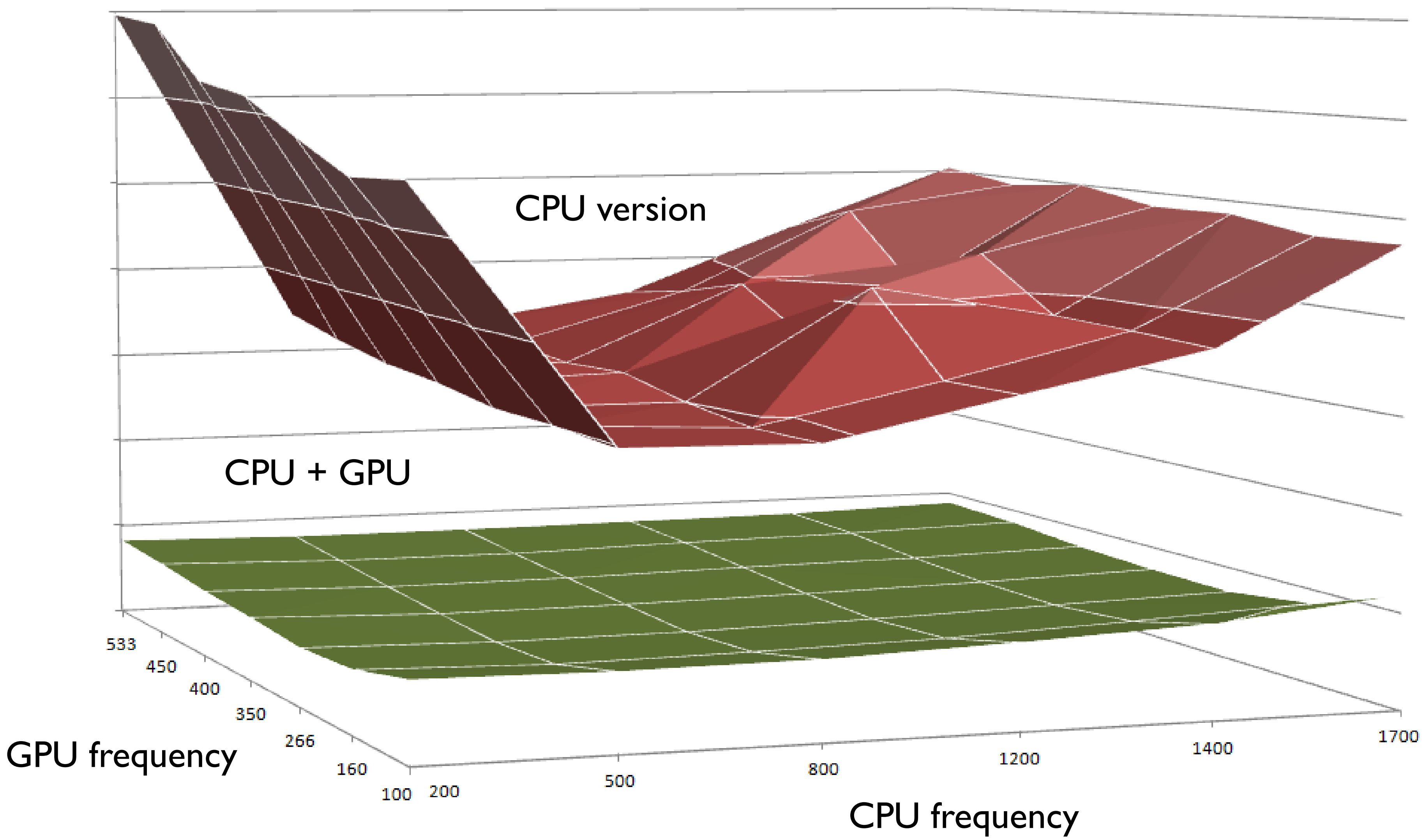
- Significant performance benefits using Mali-T600 GPU Compute
  - In terms of speed
  - …and energy
- CPU version could have been optimised more
  - Multithreaded
  - NEON
  - We would expect much better speed… but also even more power usage
  - And with the GPU implementation the CPU is free to do something else



ARM

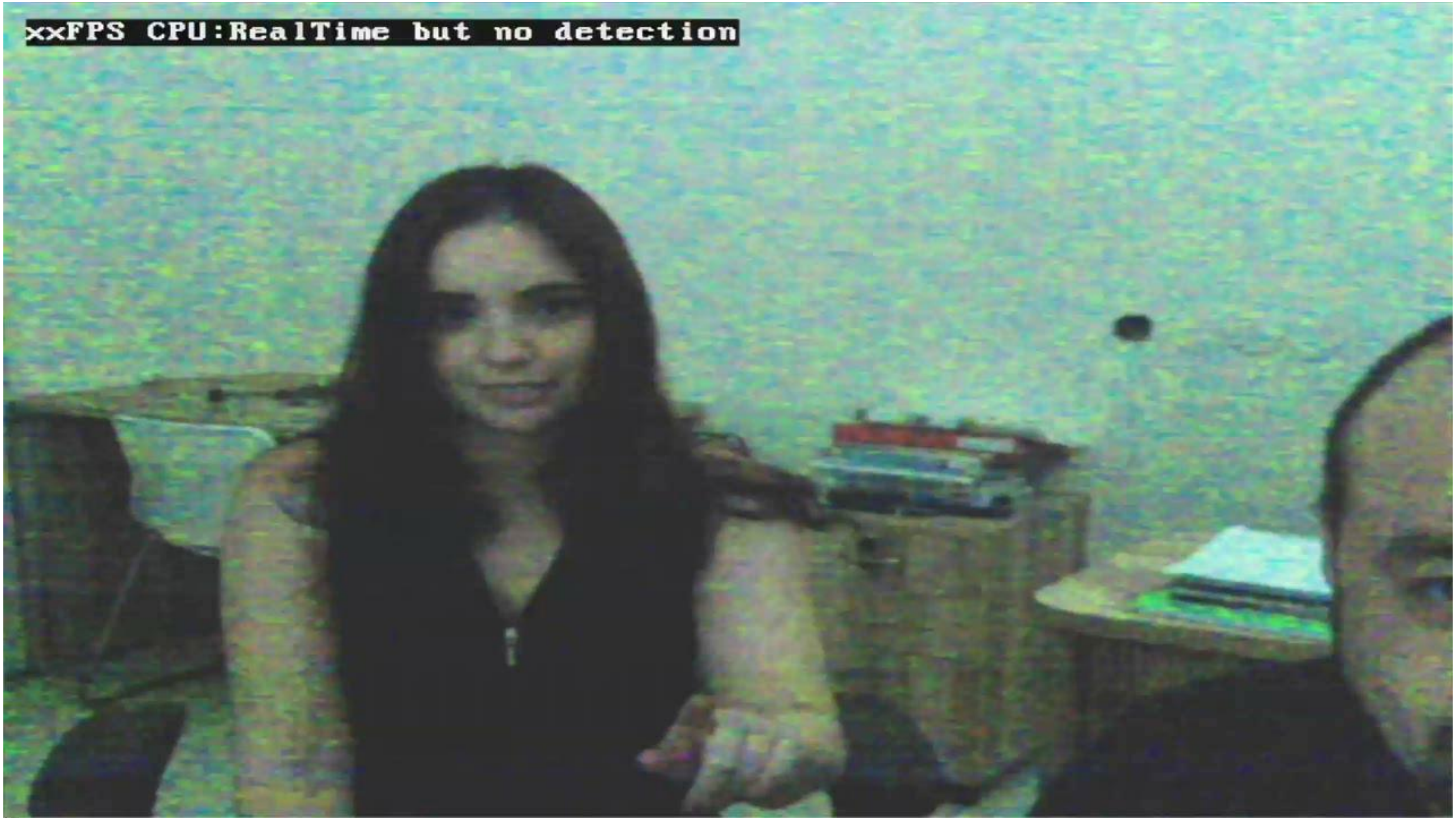

No Cable Connected
CPU Version [Auto]
FPS : 26.60 Detections per sec 3.12
CPU0: 56.37% CPU1: 87.68%
GPU Load: 6.56% Resolution 640x480
Async 'enabled' Detection 'enabled'

Face Detection Relative Energy Usage

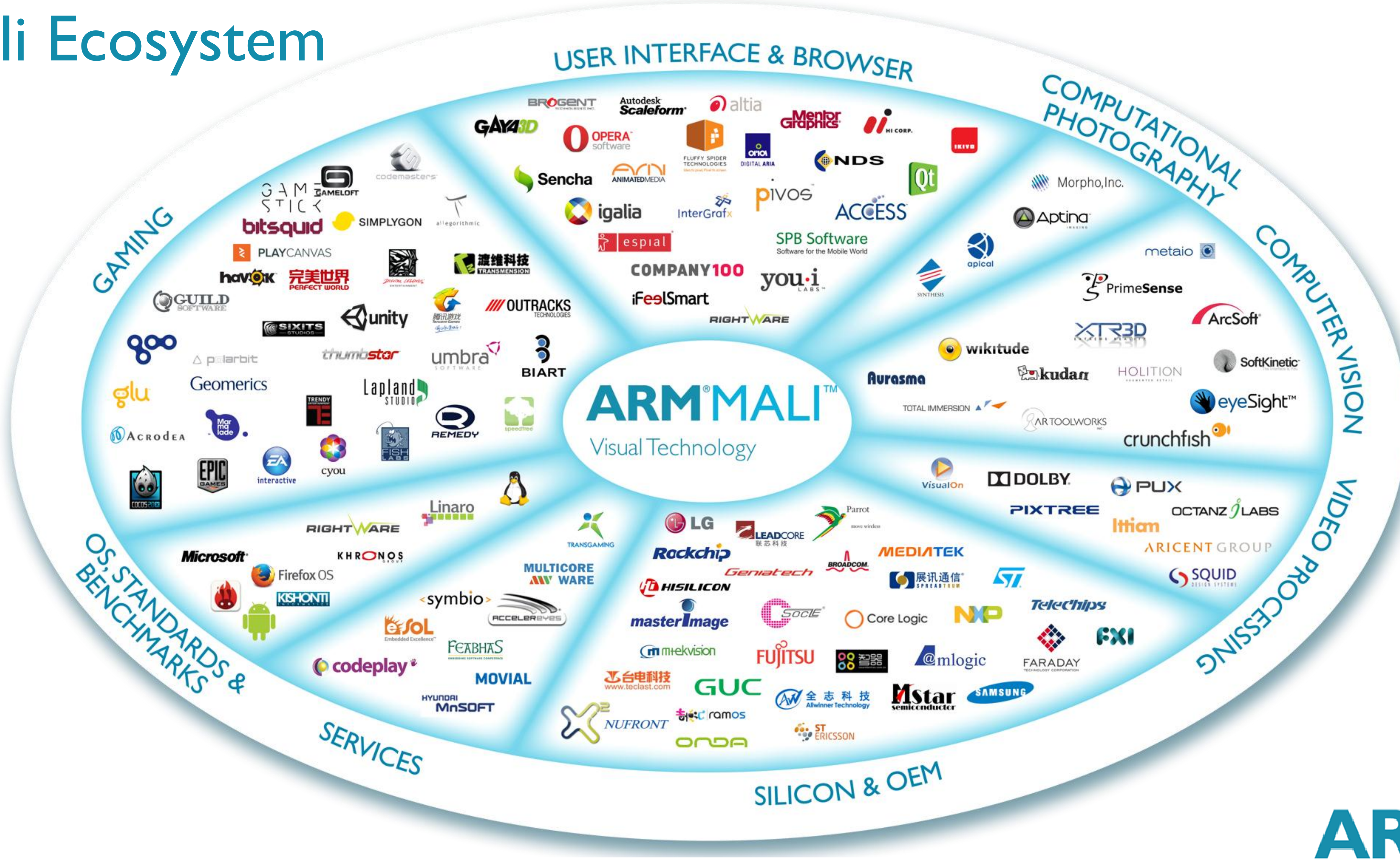


- CPU + GPU always more efficient than CPU only
- CPU + GPU on average ~5x more efficient

ARM


xxFPS CPU:RealTime but no detection

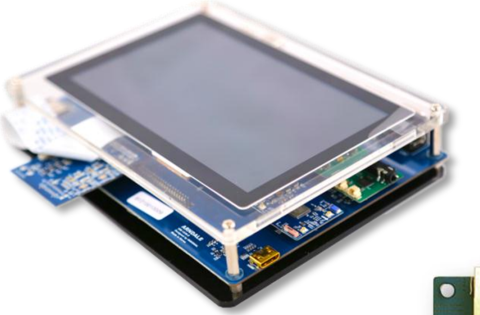
Mali Ecosystem



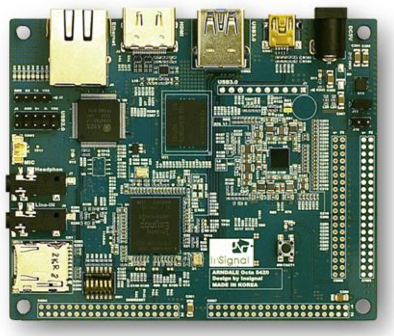
ARM

GPU Compute on Mali

- Full profile OpenCL conformance since late 2012
- OpenCL devices: Arndale platforms, Samsung Chromebook
  - http://malideveloper.arm.com/develop-for-mali/development-platforms/insignal-arndale-octa-board/
  - http://malideveloper.arm.com/develop-for-mali/development-platforms/samsung-arndale-board/
  - http://malideveloper.arm.com/develop-for-mali/development-platforms/samsung-chromebook/
  - Including full guide for running OpenCL 1.1



- Other devices:
  - Google Nexus 10: first GPU-accelerated RenderScript device
  - Samsung Galaxy S5
- All based on Mali-T6xx



Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

- OpenCL Execution Model
- Mali Drivers

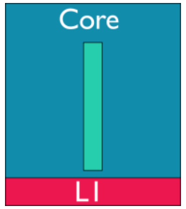
Optimal OpenCL for Mali-T600 / T700

OpenCL Optimization Case Studies

CL Execution model on Mali-T600 (1)

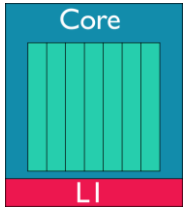
Work item

Registers, PC, SP, Private stack

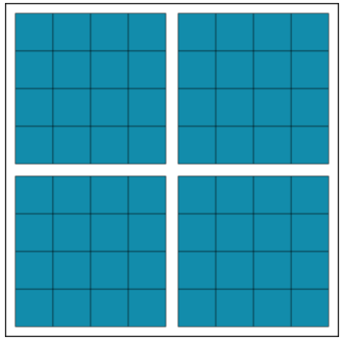


Work Group

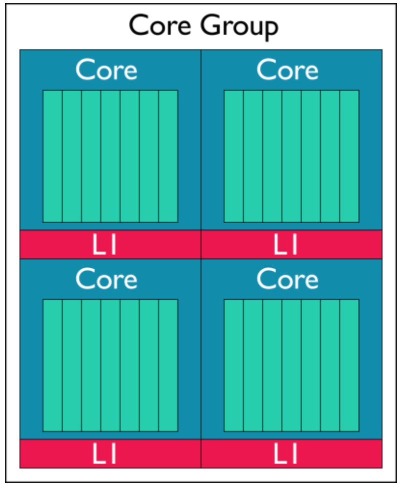
Barriers, Local Atomics, Cached local memory



ND Range



Global Atomics, Cached global memory



ARM

CL Execution model on Mali-T600 (2)

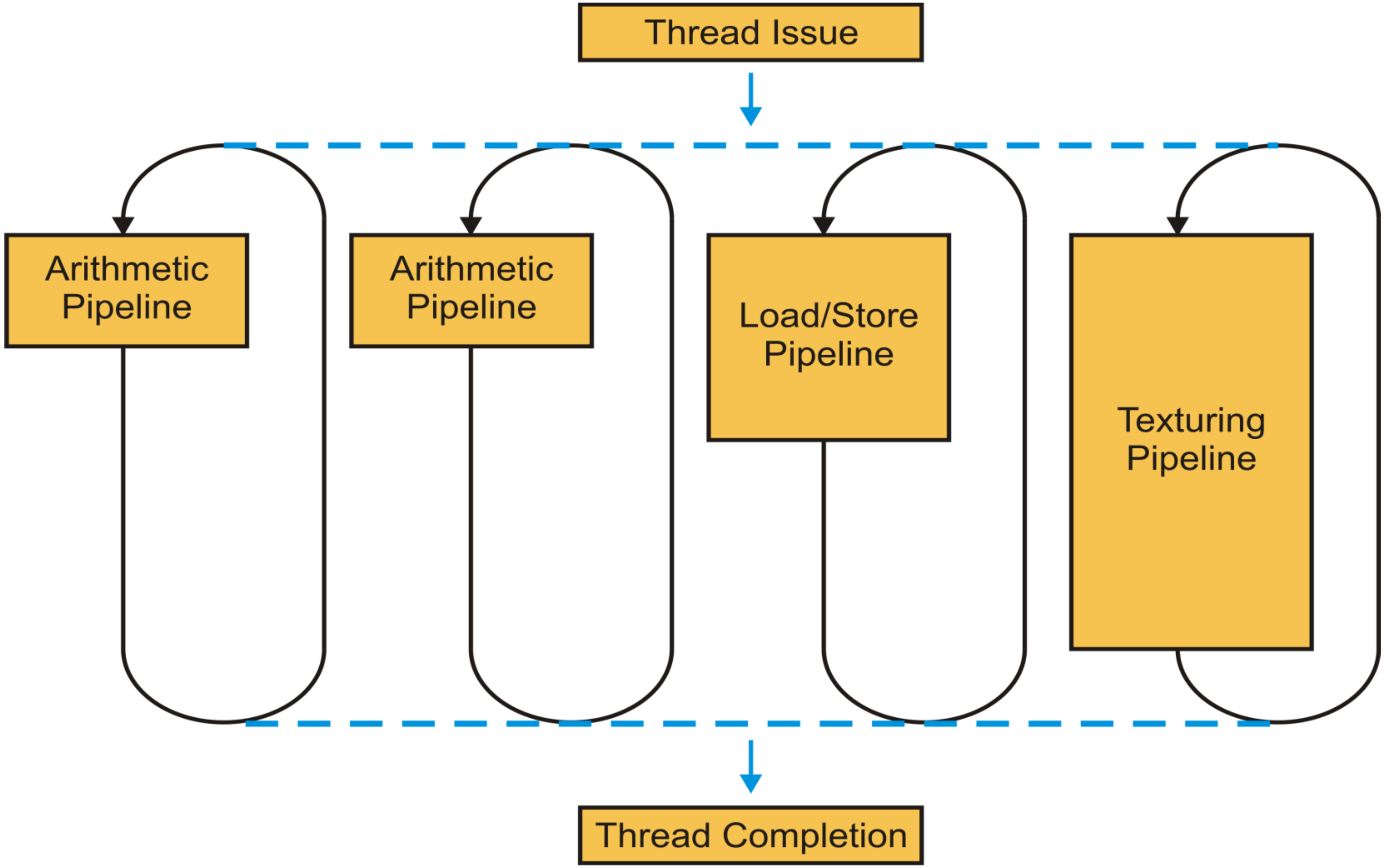
- Each work-item runs as one of the threads within a core
  - Every Mali-T600 thread has its own independent program counter
  - …which supports divergent threads from the same kernel
  - caused by conditional execution, variable length loops etc.
  - Some other GPGPU's use "WARP" architectures
  - These 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

ARM

Inside a Core



$$T = \max(A_0, A_1, LS, Tex)$$

Inside each ALU

- Each ALU has a number of hardware compute blocks:

Dot product (4 x muls, 3 x adds)	7 flops
Vector add	4 flops
Vector mul	4 flops
Scalar add	1 flop
Scalar mul	1 flop
	= 17 flops / cycle / ALU / core (FP32)

- Theoretical peak vs Realistic peak performance

- Capable of 5 FP64 flops

ARM

Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

- OpenCL Execution Model
- Mali Drivers

Optimal OpenCL for Mali-T600 / T700

OpenCL Optimization Case Studies

ARM's OpenCL Driver

- Full profile OpenCL v1.1 in hardware and Mali-T600 / T700 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)
  - printf implemented as an extension to v1.1 driver

ARM

A Note about RenderScript

- The GPU-Compute API on Android™
- Similar architecture to OpenCL
  - Based on C99
- Transparent device selection
  - The driver manages and selects devices
- Transparent memory management
  - Copying managed by the driver, based on allocation flags
- Higher level than OpenCL
  - Less explicit control over details

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

```
float *array;

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

```
rs_allocation array;

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

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

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

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

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

Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

Optimal OpenCL for Mali-T600 / T700

- Programming Suggestions
- Optimising with DS-5 Streamline and HW Counters
- Optimising: Two Examples
- General Advice

OpenCL Optimization Case Studies

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

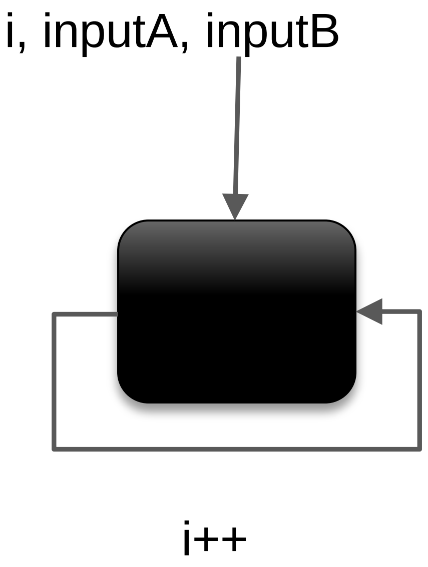
- When running on Mali just use global memory instead
  - Thus the copy operations can be removed
  - And also any barrier instructions that wait for the copy to finish
  - Query the device flag `CL_DEVICE_HOST_UNIFIED_MEMORY` if you want to write performance portable code for Mali and desktop PC's
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

Use Vectors

- Mali-T600 and T700 series GPUs have a vector capable GPU
- Mali prefers 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

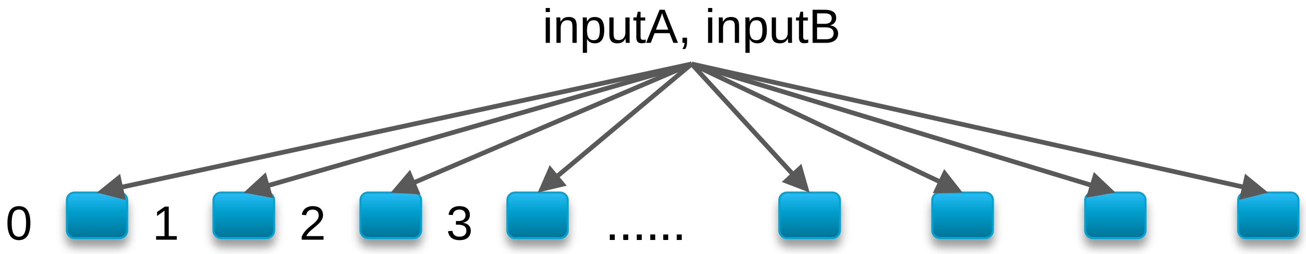
Hello OpenCL

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



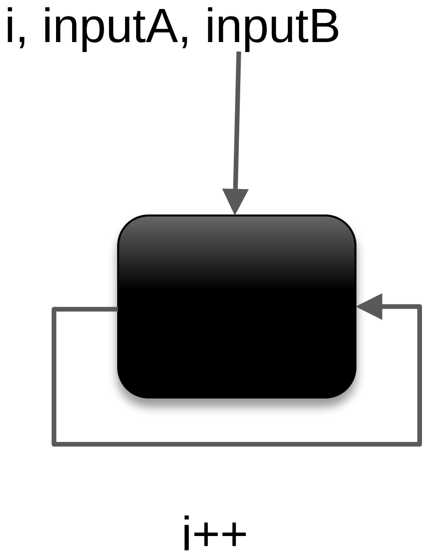

```
__kernel void kernel_name(__global int* inputA,
                          __global int* inputB,
                          __global int* output)
{
    int  i = get_global_id(0);
    output[i] = inputA[i] + inputB[i];
}

clEnqueueNDRangeKernel(..., kernel, ..., arraySize, ...)
```



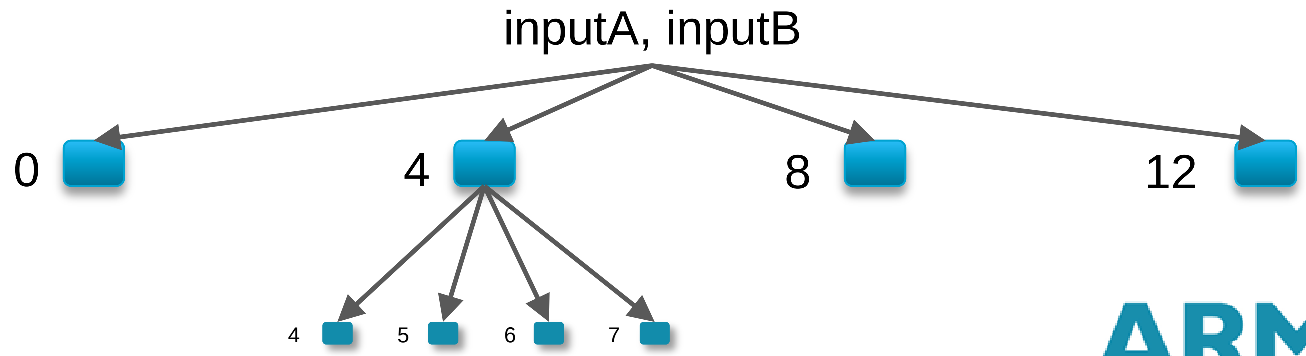
Hello OpenCL Vectors

```
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);
}

clEnqueueNDRangeKernel(..., kernel, ..., arraySize / 4, ...)
```



Creating buffers

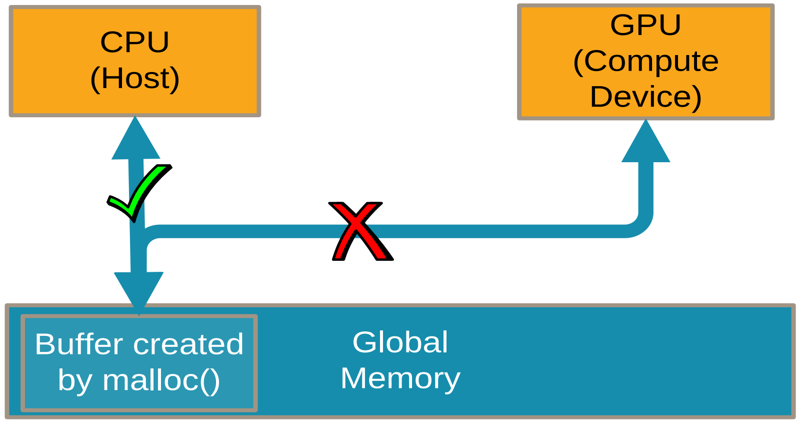
- The application creates buffer objects that pass data to and from the kernels by calling the OpenCL API **`clCreateBuffer()`**

- 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

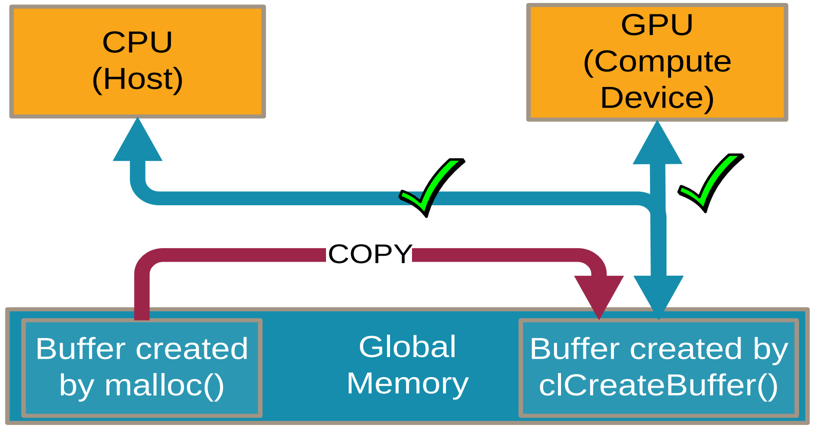
- So calling **`clCreateBuffer()`** with **`CL_MEM_USE_HOST_PTR`** and passing in a user created buffer requires the driver to create a new buffer and copy the data (identical to **`CL_MEM_COPY_HOST_PTR`**)
  - This copy reduces performance

- 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

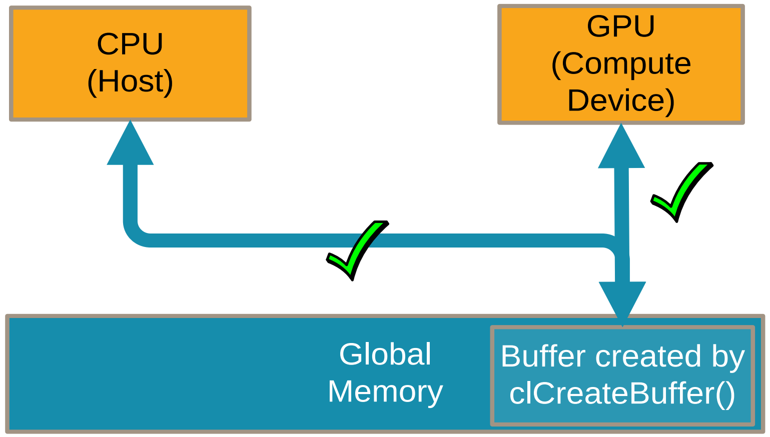


Buffers created by user (**`malloc`**) are not mapped into the GPU memory space



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

Host data pointers



`clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)`
creates a buffer visible by both GPU and CPU

- 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

ARM

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 large multiple of 4
  - 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

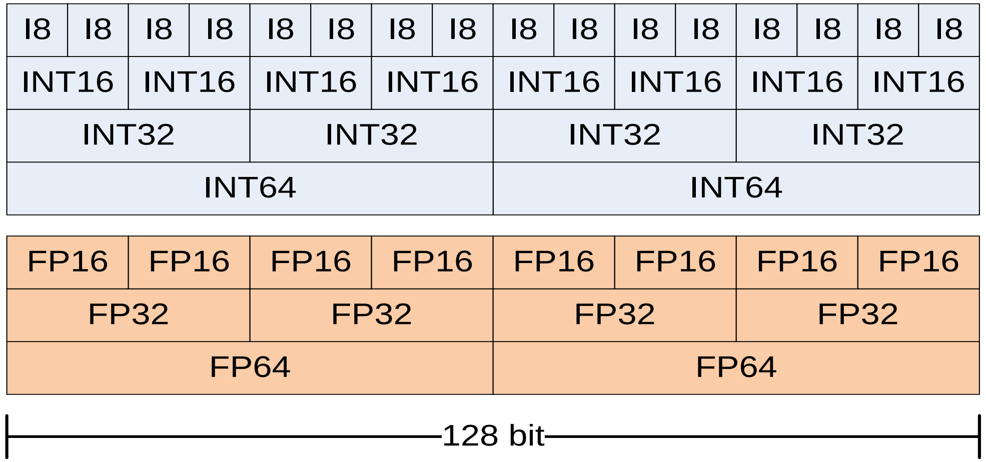
- Using "`half`" or "`native`" versions of built-in functions
  - e.g. **`half_sin(x)`**
  - Specification mandates a minimum of 10-bits of accuracy
  - e.g. **`native_sin(x)`**
  - Accuracy and input range implementation defined

  - Not always an advantage on Mali-T600 / T700… for some functions the precise versions are just as fast

Arithmetic

- Mali-T600 / T700 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 / T700 can natively support all CL data types
- VLIW: Several operations per instruction word
  - Some operations are free

```
16 x 8-bit chars (char16)
2 x 64-bit integers (long2)
4 x 32-bit floats (float4)
2 x 64-bit floats (double2)
```

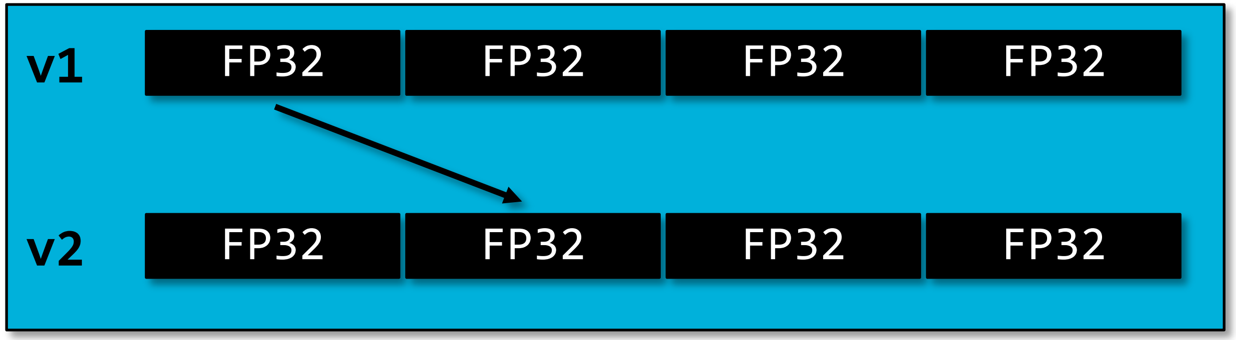


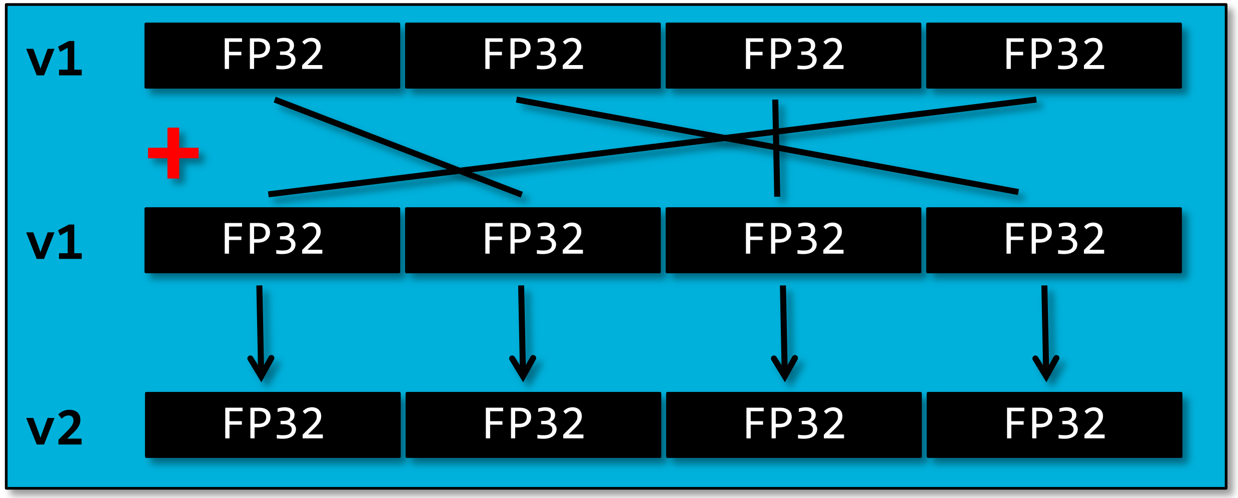
Register operations

- 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 datacould 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 (assuming no cache misses)

ARM

Miscellaneous

- 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 to Mali GPUs

Mali-T600 / T700 Compute Overview

Optimal OpenCL for Mali-T600 / T700

- Programming Suggestions
- Optimising with DS-5 Streamline and HW Counters
- Optimising: Two Examples
- General Advice

OpenCL Optimization Case Studies

Hardware Counters

- Counters per core
  - Active Cycles
  - Pipe activity
  - L1 cache
- Counters per Core Group
  - L2 caches
- Counters for the GPU
  - Active cycles
- Accessed through Streamline™
  - Timeline of all hardware counters, and more
  - Explore the execution of the full application
  - Zoom in on details

Streamline

CPU Activity
User
System
100%
51.55% avg.
1.38% avg.
GPU Vertex
Activity
100.00% avg.
Mali Arithmetic Pipe
A instructions
600 K
321,186
376,030
Mali Core Cycles
Compute cycles
1 M
417,980
Mali Core Threads
Compute threads
150 K
66,803
Mali Job Manager Cycles
GPU cycles
1 M
417,990
Mali L2 Cache
External read beats
External write beats
L2 read hits
L2 write hits
600 K
36,142
8,333
13,818
18,209
Mali Load/Store Cache
Read hits
Read misses
Write hits
Write misses
600 K
27,370
49,823
8,154
4,226
Mali Load/Store Pipe
LS instruction issues
LS instructions
1.1 M
178,916
131,759

Memories

- Only one programmer controlled memory
  - Many transparent caches
- Memory copying takes time
  - It can easily dominate over kernel execution time
- Use appropriate memory allocation schemes
- Avoid synchronization points
  - Cache maintenance has a cost as well
- Streamline to the rescue
  - Visualize when kernels are executed
  - Many features not covered here

Hiding Pipeline Latency

- Needs enough threads
  - Limited by register usage
- When there are issues
  - Few instructions issued per cycle
  - Spilling of values to memory
- Symptoms
  - Low Max Local Workgroup Size in OpenCL
  - Few instructions issued per cycle in limiting pipe
- Remedy
  - Smaller types → More values per register
  - Splitting kernels

Pipeline Utilization

- Prefer vector operations
  - More components per operation
- Prefer small types
  - More components in 128 bits
- Balance work between the pipes
  - Do less – with the pipe that limits performance

$$T = \max(A_0, A_1, LS, Tex)$$

ARM

Finding the Bottlenecks

- Host application or Kernel execution
  - Avoid memory copying
  - Avoid cache flushes

- Which pipe is important?
  - Operations in other pipes incur little or no runtime cost

- Saving operations or saving registers
  - How much register pressure can we handle, and still hide the latencies?

- How well are we using the caches
  - Are instructions spinning around the LS pipe waiting for data?

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 v1.2 **`printf`** function implemented as an extension in v1.1

ARM

Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

Optimal OpenCL for Mali-T600 / T700

- Programming Suggestions
- Optimising with DS-5 Streamline and HW Counters
- Optimising: Two Examples
- General Advice

OpenCL Optimization Case Studies

The Limiting Pipe

- Three hardware counters
  - Cycles active (#C)
  - Number of A instructions (#A)
  - Number of LS instructions (#LS)
- The goal
  - Similar values for #A and #LS → Both pipes used
  - Max(#A, #LS) similar to #C → Limiting pipe used every cycle
- Example:
  - #LS / #A = 5
  - #LS / #A = 1, #C up by < 10%

$$\bar{y} = a\bar{x} + \bar{y}$$

$$\bar{y} = 0.05a\bar{x} + 0.05a\bar{x} + \ldots + 0.05a\bar{x} + \bar{y}$$

Cache Utilization

- The Load/Store pipe hides latency
  - Many threads active
- Not always successful
  - Insufficient parallelism
  - Bad cache utilization
  - Failing threads will be reissued
- Reissue is a sign of cache-misses
  - Instruction words issued
  - Instruction words completed
- Example
  - Inter-thread stride for memory accesses

Execution Order

- Kernel saxpy
  - Load from x
  - Load from y
  - Compute
  - Store to y

$$\bar{y} = a\bar{x} + \bar{y}$$

- Execution order
  - Threads 1 through N load from x
  - Threads 1 through N load from y
  - Threads 1 through N compute
  - Threads 1 through N store to y

- How many bytes should we load per thread?

ARM

A Single Instruction Word

- We should have one load instruction word
  - The next bytes will be picked up by the next thread
- Loading less is bad
  - Does not utilize the SIMD operations
- Loading more is bad
  - The next bytes will be loaded after all other threads have loaded their first
- Saxpy with different strides
  - 128 bits: 4.5 issues per instruction
  - 256 bits: 5.5 issues per instruction
  - 64 bytes: 9.3 issues per instruction

Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

Optimal OpenCL for Mali-T600 / T700

- Programming Suggestions
- Optimising with DS-5 Streamline and HW Counters
- Optimising: Two Examples
- General Advice

OpenCL Optimization Case Studies

Know your bottleneck

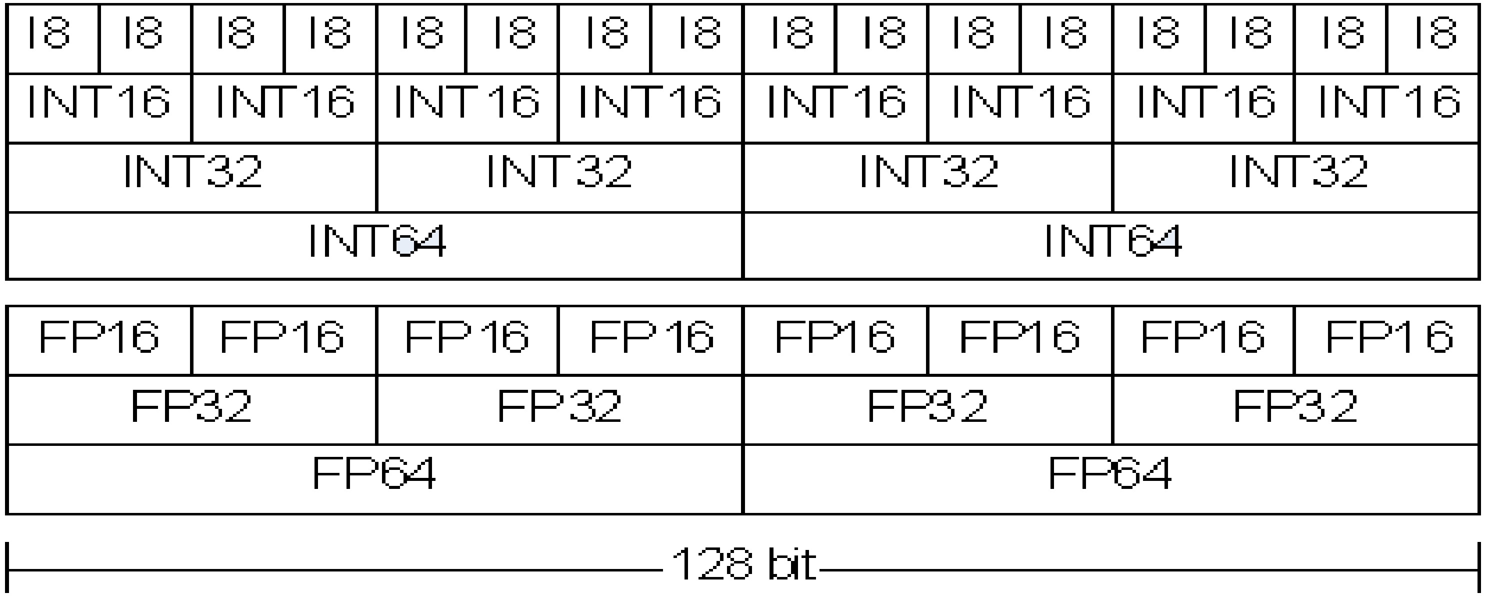
- Use vector operations
- If you are bandwidth-limited, merge kernels
  - Avoid reloading data
- If you are register-limited, split kernels
  - Easier for the compiler to do a good job
- If you are Load-Store-limited, do less load-store
  - Compute complex expressions instead of using lookup-tables
- If you are Arithmetic-limited, do less arithmetic
  - Tabulate functions
  - Use polynomial approximations instead of special functions

Synchronization between threads

- Two options in OpenCL
  - Barriers inside a work-group
  - Atomics between work-groups
- We like atomics to ensure data consistency
  - But preferably on the same core
- Barriers can be useful to improve cache utilization
  - Limit divergence between threads
  - Keeping jobs small serves the same purpose
- We see examples of large jobs with many barriers
  - We often prefer small jobs with dependencies

Vectorize your operations

- More components per operation
  - For basic arithmetic and memory operations
  - Square roots, trigonometry and atomics are scalar
- Fewer registers used
  - The compiler will only do part of the job



ARM

Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

Optimal OpenCL for Mali-T600 / T700

OpenCL Optimization Case Studies

- Laplace
- SGEMM

OpenCL Laplace Case Study

- Laplace filters are typically used in image processing
  - … often used for edge detection or image sharpening
  - and can be part of a computer vision filter chain
- This case study will go through a number of stages…
  - demonstrating a variety of optimization 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 images samples showing the effect of the filter we are using…

OpenCL Laplace Case Study

Original

OpenCL Laplace Case Study

Filtered

OpenCL Laplace Case Study

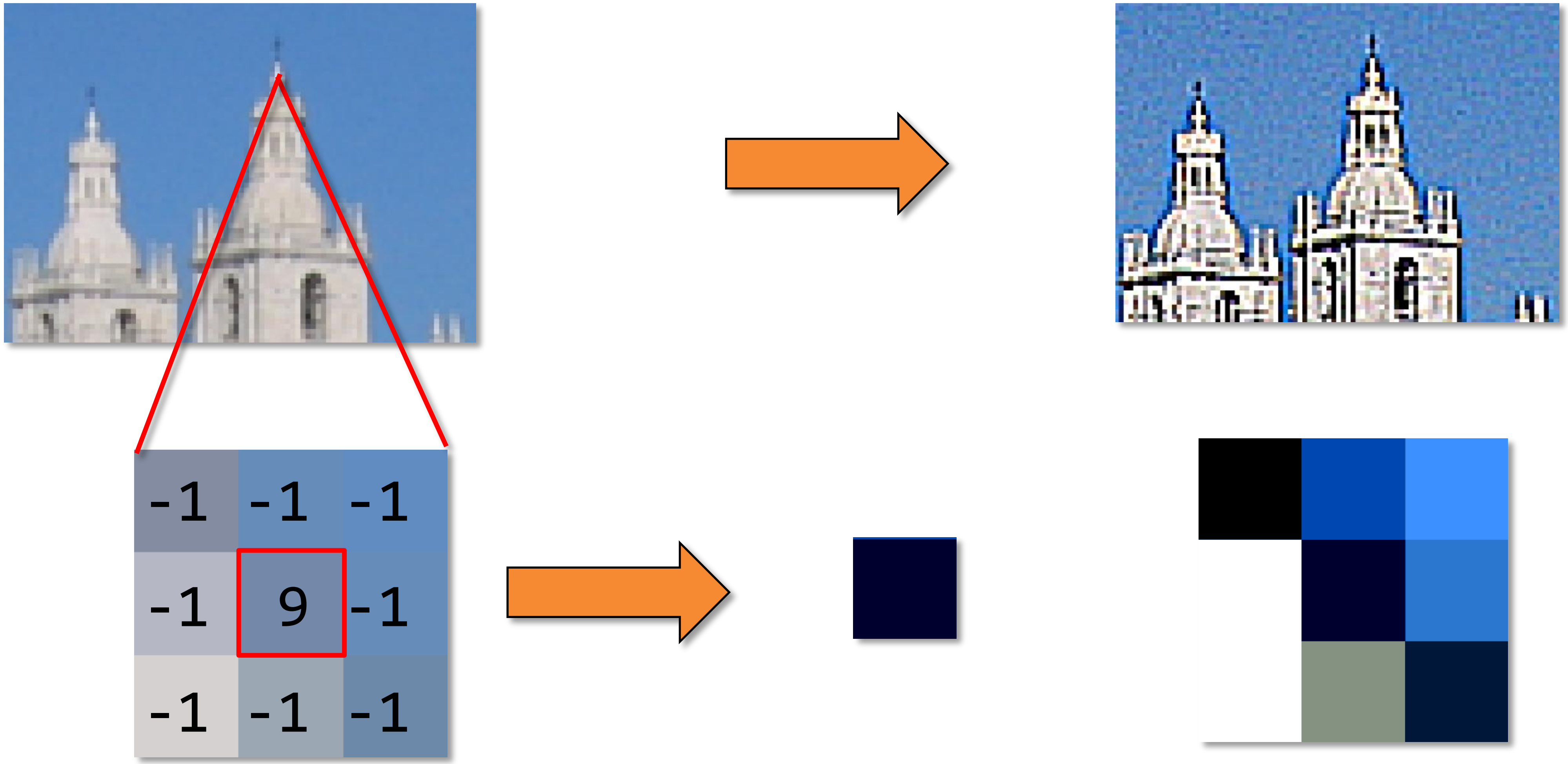
Original

ARM

OpenCL Laplace Case Study

Filtered

OpenCL Laplace Case Study



OpenCL Laplace Case Study

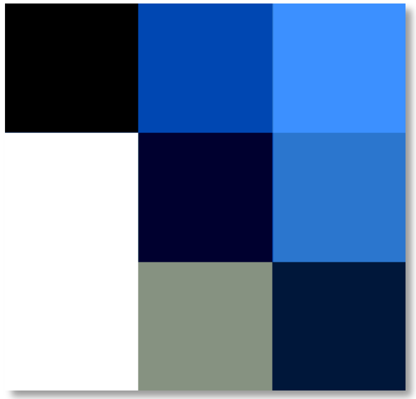
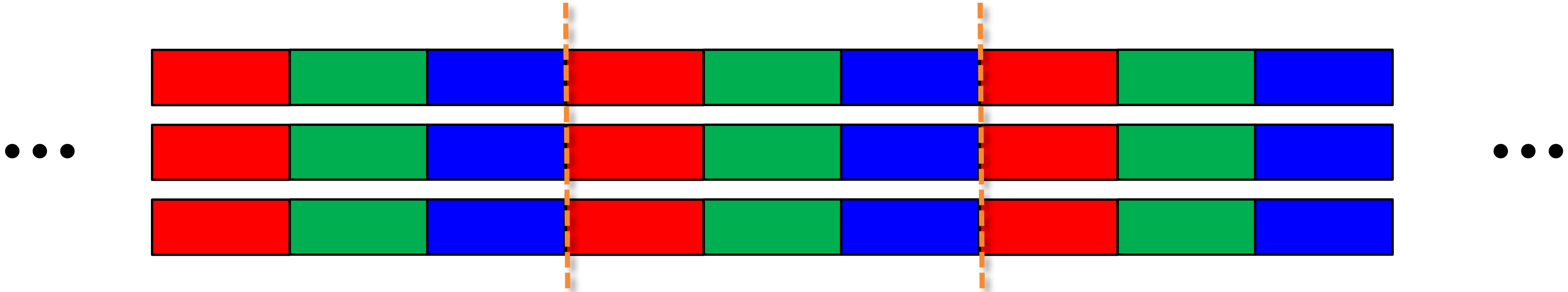


image “stride” = width x 3



OpenCL Laplace Case Study

```
#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);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –
             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];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –
             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 + 1 + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;
    pdst[ind + 2] = red;
}
```


ARM

OpenCL Laplace Case Study

```
#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);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 -
             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];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 -
             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 + 1 + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;
    pdst[ind + 2] = red;
}
```

Destination buffer   Source buffer   Image width   Image height

OpenCL Laplace Case Study

```
#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);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –
             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];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –
             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 + 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.)

ARM

OpenCL Laplace Case Study

```
#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);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –
             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];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –
             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 + 1 + w * (y + 1));
    pdst[ind]     = blue;
    pdst[ind + 1] = green;
    pdst[ind + 2] = red;
}
```

The main calculation… we need to perform this for the red, green and blue color components…

ARM

OpenCL Laplace Case Study

```
#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);

    bColor = bColor - psrc[ind] - psrc[ind+3] - psrc[ind+6] - psrc[ind+3*w] + psrc[ind+3*(1+w)] * 9 –
             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];
    rColor = rColor - psrc[ind+2] - psrc[ind+5] - psrc[ind+8] - psrc[ind+3*w+2] + psrc[ind+3*(1+w)+2] * 9 –
             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 + 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

Image	Pixels	Time (s)
768 x 432	331,776	0.0107
2560 x 1600	4,096,000	0.0850
2048 x 2048	4,194,304	0.0865
5760 x 3240	18,662,400	0.382
7680 x 4320	33,177,600	0.680

Mali T604 @ 533MHz

CPU
0.0229 x0.5
0.125 x0.7
0.128 x0.7
0.572 x0.7
1.02 x0.7

Single A15 @ 1.7GHz

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.
```


ARM

OpenCL Laplace Case Study: **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)

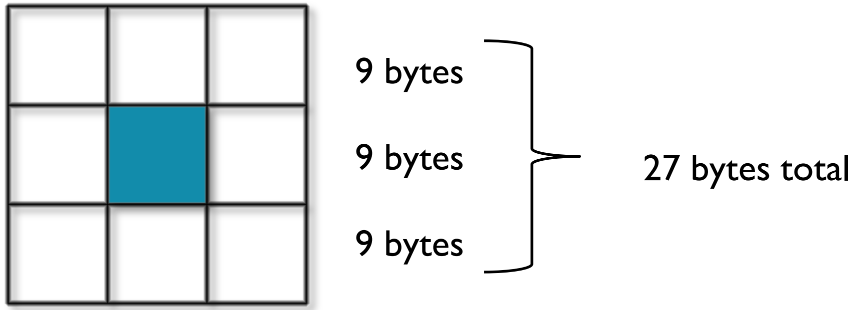
- As we'll be running 5 times fewer work items, we'll need to update the `globalWorkSize` values…

```
globalWorkSize[0] = image_height;
globalWorkSize[1] = (image_width / 5);
```

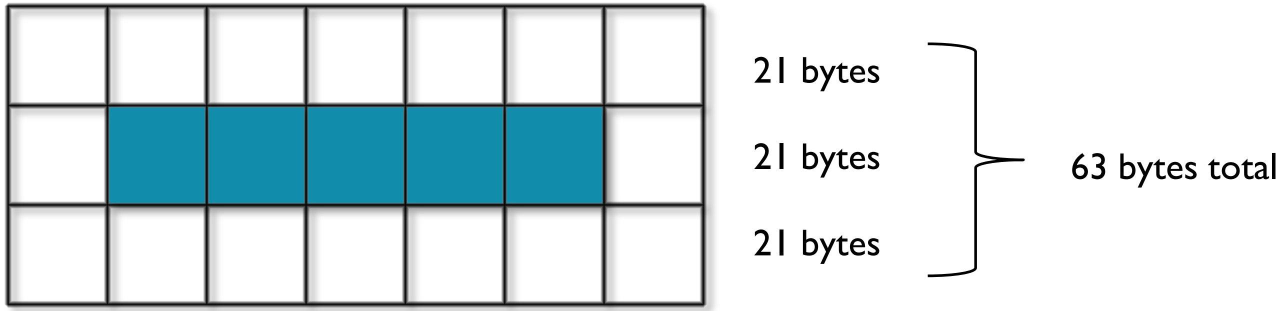

ARM

OpenCL Laplace Case Study

From processing 1 pixel…



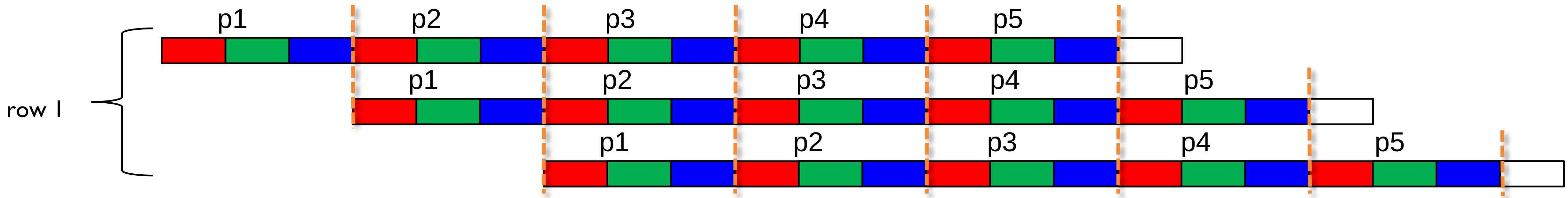
…to processing 5 pixels…



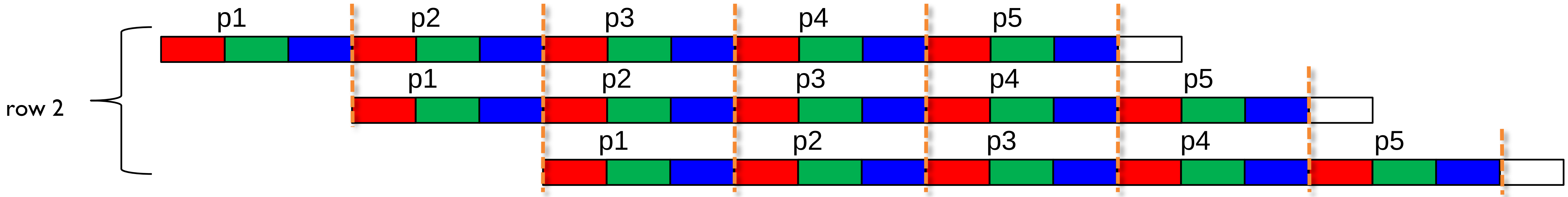
But we would like to load this data in a way that allows us to efficiently calculate the results in a single vector calculation…

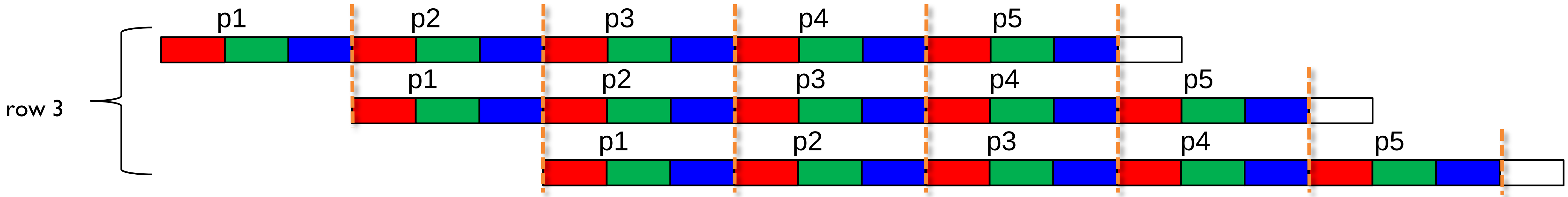
OpenCL Laplace Case Study

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



And the same for rows 2 and 3…

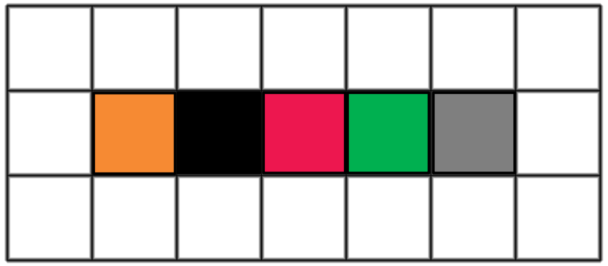


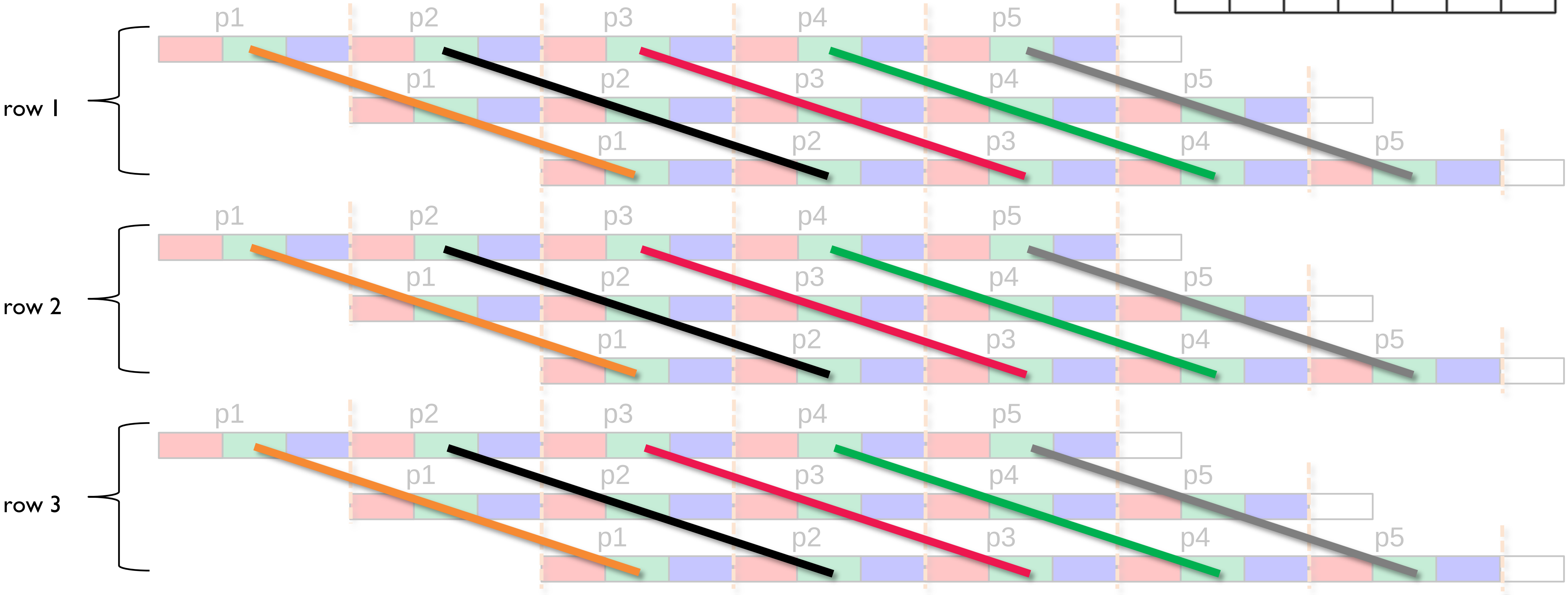


ARM

OpenCL Laplace Case Study

The five pixels can then be computed as follows…





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;
}
```

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

ARM

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;
}
```

Perform the Laplace calculation on all five pixels at once
Then clamp the values between 0 and 255 (using the BIFL!)

Convert back to uchar16… and then write 5 pixels to destination buffer

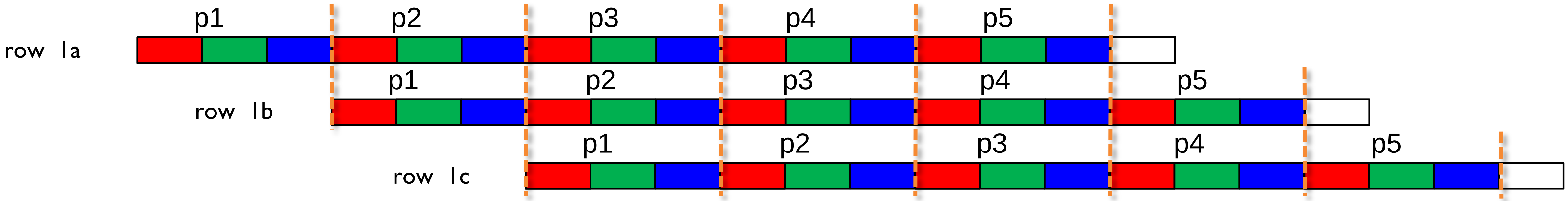
OpenCL Laplace Case Study

- Vectorization Results

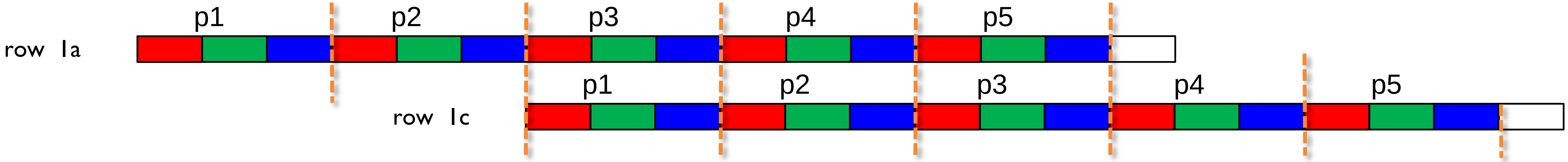
Image	Pixels	Original	Opt I
768 x 432	331,776	0.0107	x1.4
2560 x 1600	4,096,000	0.0850	x4.5
2048 x 2048	4,194,304	0.0865	x1.7
5760 x 3240	18,662,400	0.382	x6.0
7680 x 4320	33,177,600	0.680	x6.2
	Work registers:	8	8+
	ALU cycles:	25.5	22.5
	L/S cycles:	28	13

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…



becomes…



row 1b ⟵ row1(p2, p3, p4, p5) + row2(p6)

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…

```
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…

```
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 row3c_    = vload16(0, psrc + ind + (w * 6) + 6);
uchar16 row3b_    = (uchar16)(row3a_.s3456789a, row3c_.s56789abc);
```

OpenCL Laplace Case Study

- Synthesize Loads Results

Image	Pixels	Original	Opt 1 (Vectorize)	Opt 2 (Synth. loads)
768 x 432	331,776	0.0107	x1.4	x1.4
2560 x 1600	4,096,000	0.0850	x4.5	x4.5
2048 x 2048	4,194,304	0.0865	x1.7	x2.0
5760 x 3240	18,662,400	0.382	x6.0	x6.0
7680 x 4320	33,177,600	0.680	x6.2	x6.3
	Work registers:	8	8+	8
	ALU cycles:	25.5	22.5	24.5
	L/S cycles:	28	13	8

ARM

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

```
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…

```
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

- Using Short Ints Results

			Vectorize	Synth. loads	Shorts
Image	Pixels	Original	Opt 1	Opt 2	Opt 3
768 x 432	331,776	0.0107	x1.4	x1.4	x1.5
2560 x 1600	4,096,000	0.0850	x4.5	x4.5	x6.2
2048 x 2048	4,194,304	0.0865	x1.7	x2.0	x1.9
5760 x 3240	18,662,400	0.382	x6.0	x6.0	x8.5
7680 x 4320	33,177,600	0.680	x6.2	x6.3	x9.0
	Work registers:	8	8+	8	7
	ALU cycles:	25.5	22.5	24.5	13.5
	L/S cycles:	28	13	8	9

ARM

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

```
__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…

```
__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

- And our date write out becomes simpler…

```
...
    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…

```
...
    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.

OpenCL Laplace Case Study

- Computing 4 Pixels Results

Image	Pixels	Original	Opt 1 (Vectorize)	Opt 2 (Synth. loads)	Opt 3 (Shorts)	Opt 4 (4 Pixels)
768 x 432	331,776	0.0107	x1.4	x1.4	x1.5	x1.6
2560 x 1600	4,096,000	0.0850	x4.5	x4.5	x6.2	x5.2
2048 x 2048	4,194,304	0.0865	x1.7	x2.0	x1.9	x5.3
5760 x 3240	18,662,400	0.382	x6.0	x6.0	x8.5	x7.2
7680 x 4320	33,177,600	0.680	x6.2	x6.3	x9.0	x7.5
	Work registers:	8	8+	8	7	6
	ALU cycles:	25.5	22.5	24.5	13.5	14
	L/S cycles:	28	13	8	9	6

ARM

OpenCL Laplace Case Study: **Optimisation 5**

- How about 8 pixels per work-item?

ARM

OpenCL Laplace Case Study

```
__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_1  + b_z_1  * (short)9 - c_z_1;

    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_2  - b_z_2  - c_z_2;

    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);
}
```


ARM

OpenCL Laplace Case Study

- Computing 8 Pixels: Results

Image	Pixels	Original	Opt 1 (Vectorize)	Opt 2 (Synth. loads)	Opt 3 (Shorts)	Opt 4 (4 Pixels)	Opt 5 (8 Pixels)
768 x 432	331,776	0.0107	x1.4	x1.4	x1.5	x1.6	x1.2
2560 x 1600	4,096,000	0.0850	x4.5	x4.5	x6.2	x5.2	x5.6
2048 x 2048	4,194,304	0.0865	x1.7	x2.0	x1.9	x5.3	x5.8
5760 x 3240	18,662,400	0.382	x6.0	x6.0	x8.5	x7.2	x8.4
7680 x 4320	33,177,600	0.680	x6.2	x6.3	x9.0	x7.5	x9.1
	Work registers:	8	8+	8	7	6	8+
	ALU cycles:	25.5	22.5	24.5	13.5	14	24
	L/S cycles:	28	13	8	9	6	11

OpenCL Laplace Case Study: **Summary**

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

Agenda

Introduction to Mali GPUs

Mali-T600 / T700 Compute Overview

Optimal OpenCL for Mali-T600 / T700

OpenCL Optimization Case Studies

- Laplace
- SGEMM

SGEMM: Preface

- Question from a developer sent to malidevelopers@arm.com...
  - Running SGEMM on 1024x1024 matrices on a Chromebook (Dual A15, Mali-T604)
  - Takes ~3s on the CPU
  - Takes ~84s using OpenCL on the GPU

- Initial analysis from ARM Developer Relations engineers…
  - Error found in the DVFS implementation of the device used
  - Working around this reduced the time to ~12s
  - Further analysis showed how susceptible SGEMM is to workgroup size
  - And some analysis showed benefits in pre-transposing matrix on the CPU
  - With some experimentation in LWS, time reduced to ~2.5s on GPU

ARM

SGEMM: The task

- Input: Matrices A, B, C (assumed to be nxn square matrices) and constants alpha, beta
- Task: $C = \alpha AB + \beta C$
- In terms of matrix elements: $C_{ij} = \alpha \sum_{k=0}^{N-1} A_{ik} B_{kj} + \beta C_{ij}$
- Naive implementation:

```
__kernel void sgemm(__global float *A, __global float *B, __global float *C,
                    float alpha, float beta, int n)
{
  float sum = 0.0;
  for (int k=0; k<n; k++) { sum += A[i*n+k]*B[k*n+j]; }
  C[i*n+j] = alpha*sum + beta*C[i*n+j];
}
```

Transposition

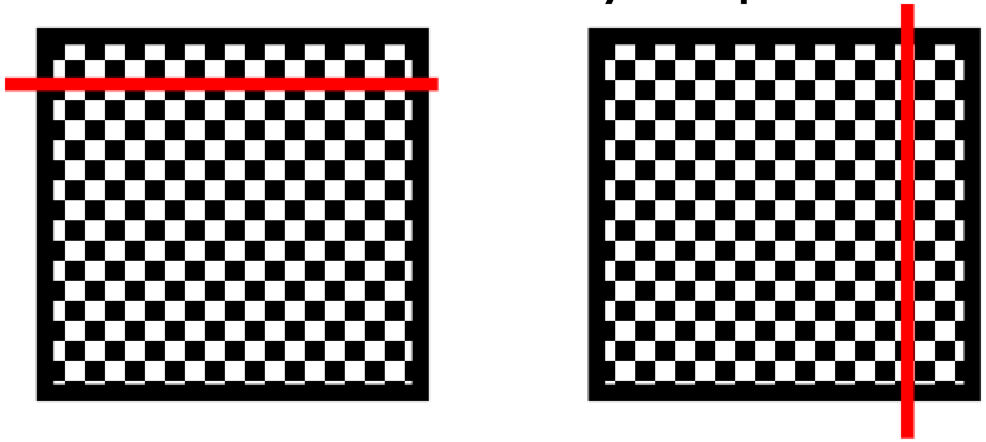
- We could transpose B before the computation, and implement the kernel

$$C_{ij} = \alpha \sum_{k=0}^{N-1} A_{ik} (B^T)_{jk} + \beta C_{ij}$$

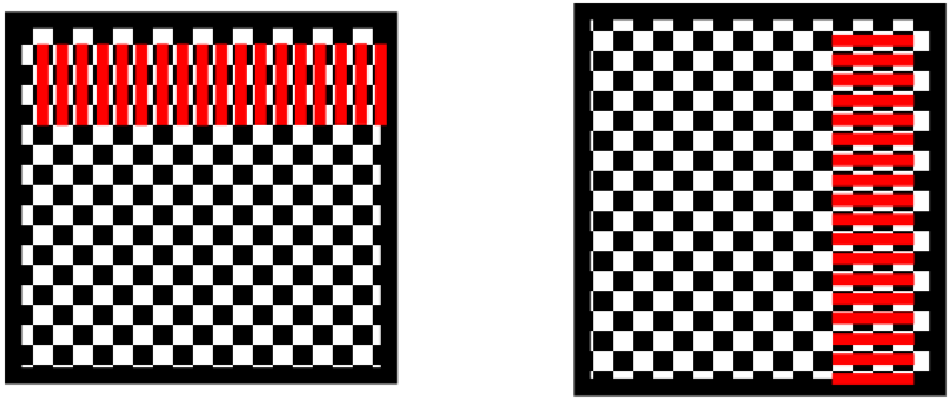
- We now have two kernels
  - One kernel for the transposition
  - One kernel for the matrix multiplication
  - Runtime is dominated by the multiplication
- On the Midgard architecture, there generally an advantage to adding a transposition.
- [List advantages of transposition]

Execution order, without transposition

- In program order, we have a very simple access pattern



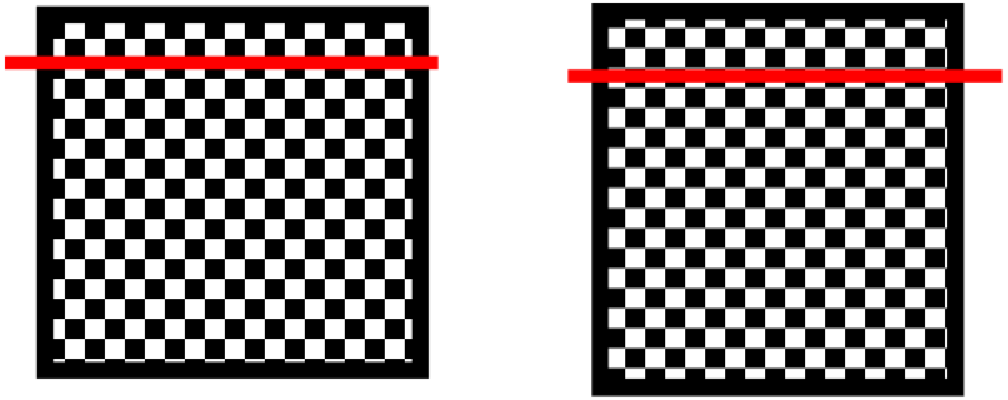
- Taking the threads in a workgroup into account, it becomes slightly less simple



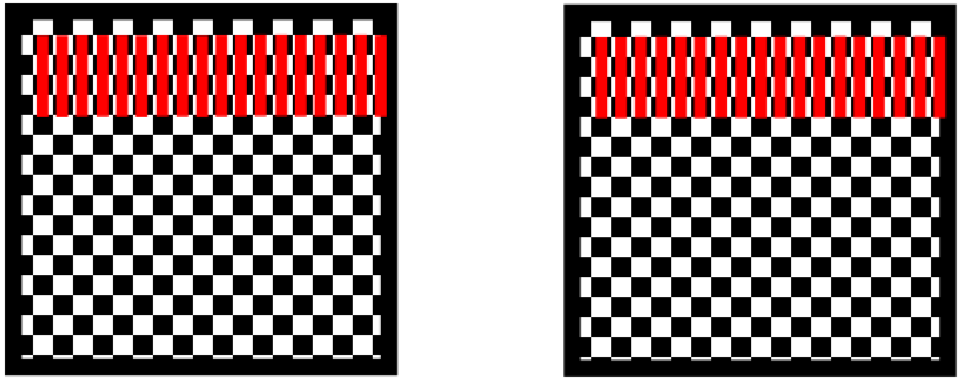
ARM

With transposition

- In program order, we always have sequential loads from memory.



- Taking the threads in a workgroup into account, we switch between different cache lines



Register Blocking

- New view: A, B and C are block matrices with block-sizes
  Δ I x Δ K, Δ K x Δ J and Δ I x Δ J
- Same equation, different multiplication operation

$$C_{ij} = \alpha \sum_{k=0}^{N-1} A_{ik} \bullet B_{kj} + \beta C_{ij}$$

- The number of elements that need to be loaded into registers shows that we do not care about deltaK, and we want deltaI similar to deltaJ

$$N^3 \left(\frac{1}{\Delta I} + \frac{1}{\Delta J} \right)$$

ARM

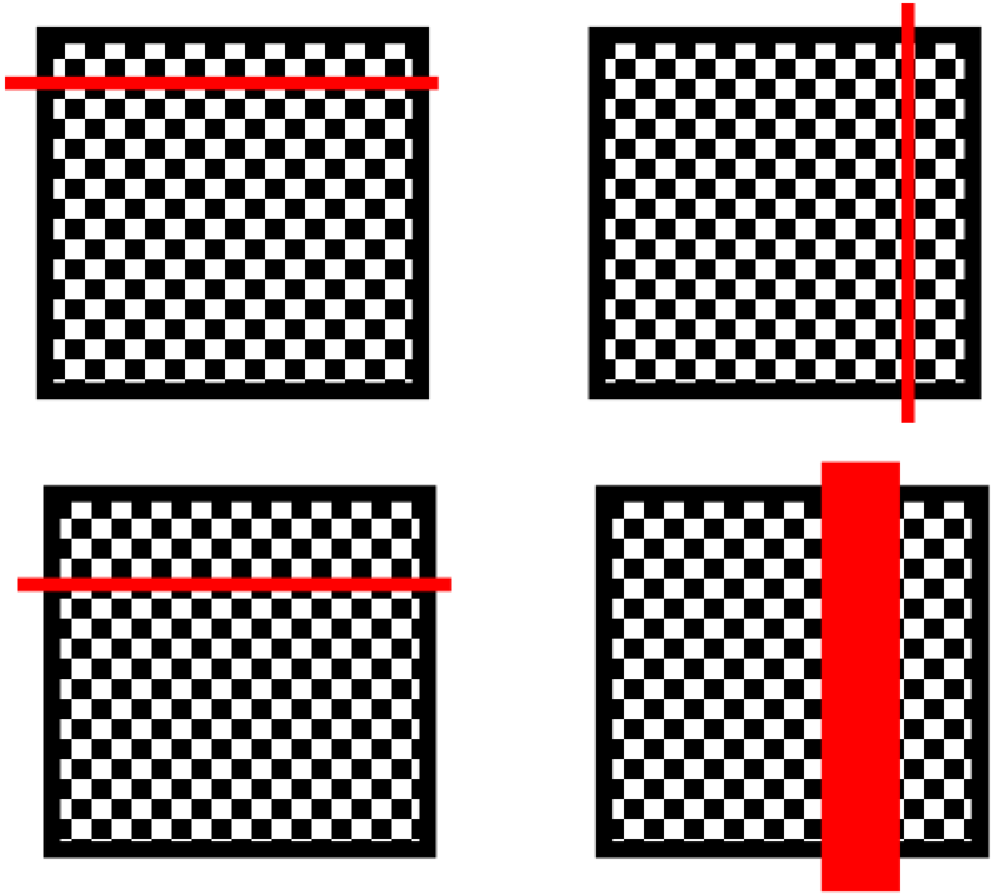
Vectorisation

- The "inner" matrix multiplication multiplies two small matrices. We want to implement this matrix multiplication using vector operations.
- We prefer operations on 4-component vectors.
- Without transposition, this requires ΔK and ΔJ to be multiples of 4, but with transposition this only requires ΔK to be a multiple of 4.
- Due to the finite number of registers, we choose (ΔI, ΔJ, ΔK) equal to (1, 4, 4) and (2, 2, 4) without and with transposition, respectively.
- We saw that similar ΔI and ΔJ are better, and here find an advantage for the transposition.
- Other schemes with more complex rearrangements than transposition are also possible.

Blocked implementation

- `for (k=0; k<n; k++) sum += b[i, k] * b[k,j];`
- Scalar multiplication
- 2 elements loaded per multiplication

- ```
  for (k=0; k<n/4 k++) {
       sum += a[i, k].x * b[k, j] +
              a[i, k].y * b[k+1, j] +
              a[i, k].z * b[k+2, j] +
              a[i, k].w * b[k+3, j]; }
  ```
- Using 4 vector multiplication
- 20 elements read per 16 multiplications


  ```

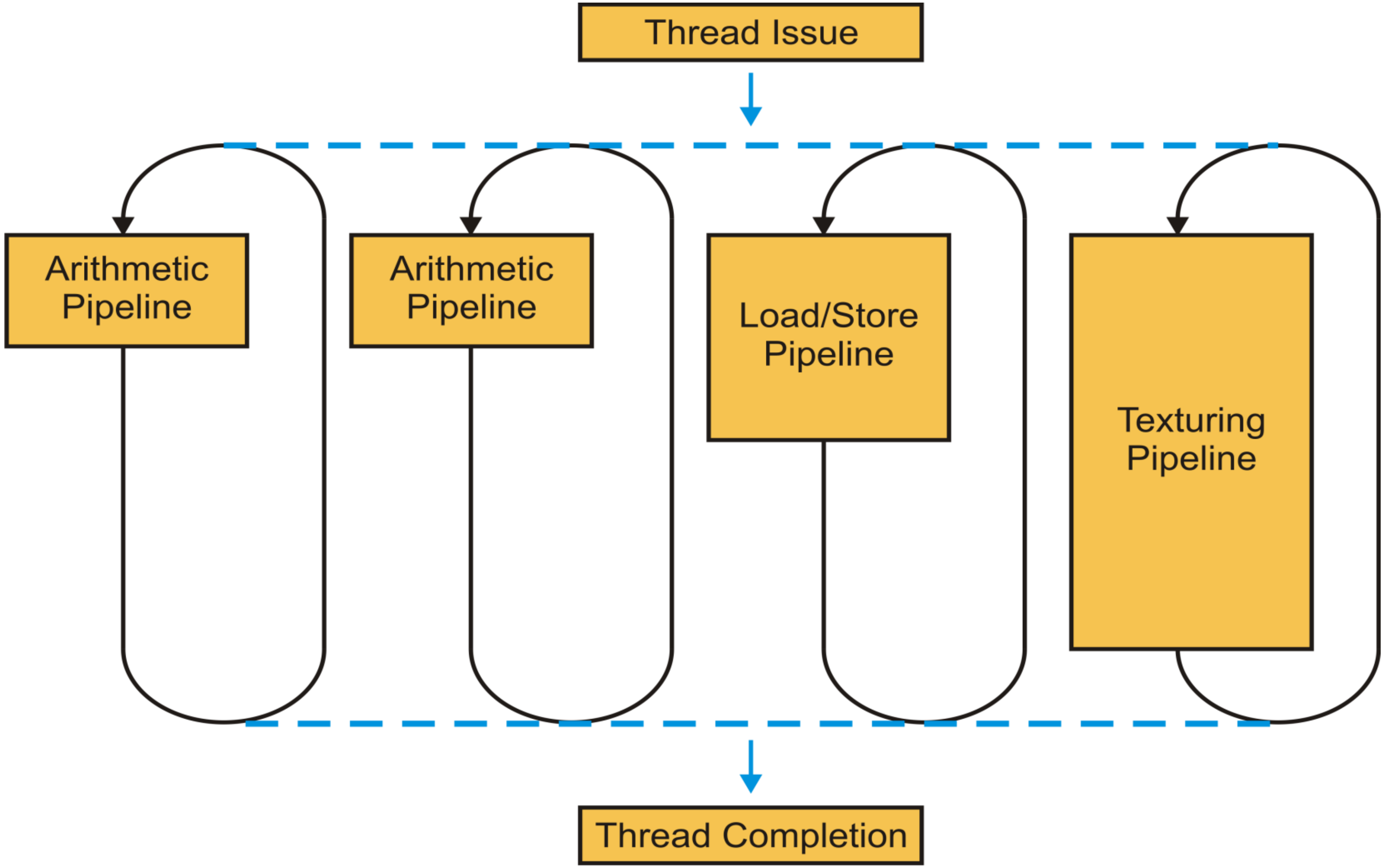
Cache utilization

- We can compute the number of cache-lines that a workgroup has to load while executing..
- We reuse cache-lines during every sequence of 4 iterations, and we therefore compute the number of L1 cache lines needed by one workgroup for 4 iterations.

Workgroup size (dim 2)	1	2	4	8	16	32	64	128
L1 fraction	2.0	1.0	0.52	0.28	0.19	0.19	0.28	0.52
Li fraction (transposed)	1.0	0.52	0.28	0.19	0.19	0.28	0.52	1.0

- If all threads execute in the order they were started, there is no problem as long as we are below 100%.
- In reality, threads diverge

Inside a Core



$$T = \max(A_0, A_1, LS, Tex)$$

Thread divergence

- Threads execute independently and have independent PC values
- Divergence in PC values due to cache misses and behaviour at various queues
- One workgroup will work on several iterations at once
- Several workgroups will be simultaneously active (for large enough matrices)
- This increases cache usage
- Lower estimated cache usage without thread divergence is a buffer against performance degradation due to thread divergence

ARM

Cache blocking

- We need to handle thread divergence for large matrices
- We introduce another level of blocking, considering the matrices to consist of larger blocks
- We pause the loop at the end of every block, waiting for the remaining threads to finish.
- This delays all threads at workgroup switch, and therefore has a cost.
- It ensures that all threads active on the GPU work on a small dataset, allowing better cache utilization.
- A trade-off that is needed for larger matrices.

Implementation

- We wait every dk iterations of the inner loop
-

```
for (uint k = 0; k < nv4; k += dk)
{
    for (uint kk = k; kk < k + dk; kk += 1)
    {
        // Inner loop body
    }
    // Wait for all work-items to finish the current tile.
    barrier(CLK_GLOBAL_MEM_FENCE);
}
```

Barriers

- At a barrier, all threads in the workgroup enter the texture pipe and wait until all threads have arrived.
- Then they exit from the pipe, one thread at a time.
- In many cases relating to correctness, barriers can be avoided and replaced by implicit barriers at job-switch or by explicit synchronization using atomics.
- For performance, we have seen that barriers can be useful to counter thread divergence.

Transposition revisited

- In sequential execution, transposition minimizes cache misses.
- On a parallel architecture, this is less clear, however
- It allows us to use better register blocking, for a good trade-off between less loads and more vector operations.
- It decreases the L1 cache usage (for our preferred workgroup sizes), allowing us to cope better with thread divergence.
- Transposition allows us to keep looking at the same page of memory for a longer time, which is beneficial for the MMU.

GPU Compute for Mobile Devices

Tim Hartley & Johan Gronqvist, ARM