WHITE PAPER

# OpenCL™ on Mali FAQs

November 2013

**Addressing the common questions that occur when using OpenCL on an ARM® Mali™-T6xx or -MaliT7xx platform**

By Karthik Hariharakrishnan, Anthony Barbier, Hedley Francis

## Contents

# FAQs

## 1. Where can I learn about OpenCL on Mali?

a) For general information on developing on Mali please visit http://malideveloper.arm.com.

b) For an SDK of samples and a framework for developing OpenCL 1.1 on Mali please see the Mali OpenCL SDK at: http://malideveloper.arm.com/develop-for-mali/sdks/mali-opencl-sdk/.

c) For documentation on general OpenCL development, as well as special considerations for developing on Mali, please see the Mali OpenCL Developer Guide at: http://malideveloper.arm.com/develop-for-mali/documentation/developer-guides/mali-t600-series-gpu-opencl-developer-guide/

## 2. What are some of the factors affecting CL Kernel performance?

A compute program (CL kernel) typically consists of a mix of A and LS instruction words. Achieving high performance on the Mali-T604 involves:

- Using a sufficient number of active threads to hide the execution latency of instructions (pipeline depth). The maximum number of threads (256) stems from the fact that each thread can use 4 registers from a register bank that contains 1024 registers. The larger the number of registers used by the kernel, the fewer the concurrent threads. So if a kernel uses 8 registers, only a maximum of 128 threads can run in parallel. If there are enough threads to hide latency there should be no performance implication of using more registers. Developers should experiment with this to find which suits their application.

- Using vector operations in kernel code to allow for straightforward mapping to vector instructions by the compiler. Please look later for a simple example on before / after vectorization for a simple kernel.

- Having a balance between A and LS instruction words. Without cache misses, the ratio of 2:1 of A-words to LS-words is optimal; with cache misses, a higher ratio is desirable. For example, a kernel consisting of 15 A-words and 7 LS-words is still likely to be bound by the LS-pipe.

## 3. Can I use memory allocated by *malloc* inside the GPU?

Allocations made using malloc will not be visible to the GPU. As a result, the `cl_mem_flags` parameter in `clCreateBuffer` is interpreted as follows:

| | |
|---|---|
| `CL_MEM_COPY_HOST_PTR` | Copies the content of the CPU pointer into the buffer specified in clCreateBuffer |
| `CL_MEM_USE_HOST_PTR` | Copies the content of the host pointer into the buffer when the first kernel using this buffer starts running. This flag enforces some other memory restrictions which will slow down things. So if possible, it's better to avoid using it. |
| `CL_MEM_ALLOC_HOST_PTR` | Hint to the driver indicating that the buffer will be accessed on the host side. To use the buffer on the CPU side, the user needs to map this buffer and write the data in it. So given that they can write directly to a mapped buffer, this method is the only one which provides a zero copy solution. So if you have to fill in an image that needs to be worked on by the GPU, this is the best way to avoid a copy. |

## 4. Why can't memory allocated by malloc be shared?

When we ask the GPU driver to allocate memory, the driver makes sure that the pages associated with that allocation are pinned and as a result will always be available to the GPU. This is not the same with memory allocated with malloc. As a result, when the GPU wants to access this memory it can cause side-effects if the pages are not available in the RAM. This is one of the main reasons we advise people to use `ALLOC_HOST_POINTER`.

## 5. What is the architecture of the ARM Mali-T604 GPU?

The ARM Mali-T604 GPU is comprised of four identical cores, each supporting up to 256 concurrently executing (active) threads. Each core has two arithmetic (A) pipelines, one load-store (LS) pipeline and one texture (T) pipeline. Thus, the peak throughput of each core is two A instruction words, one LS instruction word and one T instruction per cycle (although T instructions are only used for image format loads).

The ARM Mali-T604 (and others of the Mali-T6xx processor family) use VLIW (Very Long Instruction Word) architecture - i.e most instruction words contain multiple instructions. In addition, the ARM Mali-T604 is a SIMD (Single Instruction Multiple Data) architecture, so that most instructions operate on multiple data elements packed in 128-bit vector registers.

## 6. What are some of the differences between Desktop based architectures and Mali?

In several respects, programming for the ARM Mali-T6XX GPU embedded on a System-on-Chip (SoC) is easier than programming for desktop class GPUs:

- The `global` and `local` OpenCL address spaces get mapped to the same physical memory (the system memory – usually DDR3), backed by L1 and L2 caches that are transparent to the programmer. This often removes the need for explicit data copying and associated barrier synchronization.

- All threads have individual program counters. This means that branch divergence (if statements) is less of an issue than for warp-based architectures.

## 7. Can you give us an example for vectorizing a kernel?

We will look at a 3 × 3 Sobel filter that computes discretized gradients of the luminosity of an image:

- The input is an *H × W* array of unsigned 8-bit integers (`uchar's`), containing the luminosity values.
- The output is two H × W arrays of signed 8-bit integers (`char's`), containing the discretized gradient values along the horizontal and vertical directions. The gradient values are obtained by convolving the input image with the Sobel masks shown in the following Figure and dividing the results by 8 (i.e. shifting right by 3).

We assume that the results are computed for (H − 2) × (W − 2) inner pixels of the output images, leaving pixels on the (1 pixel wide) border intact. The sobel filter operates on a 3 x 3 area of the image.   Following are the masks for horizontal and vertical axes:

| | | |
|---|---|---|
| -1 | X | 1 |
| -2 | X | 2 |
| -1 | X | 1 |

***Horizontal (dx) mask***

| | | |
|---|---|---|
| 1 | 2 | 1 |
| X | X | X |
| -1 | -2 | -1 |

***Vertical (dy) mask***

### Naïve Implementation

This following shows a naïve implementation of a CL kernel that does the above for an image using scalar operations. The following computes one output pixel (dx and dy) per work item. The computation is performed using 16-bit integer arithmetic to avoid overflow. The kernel itself is very straightforward.

```
__kernel void sobel_char(__global const uchar* restrict inputImage,
                         const int width,
                         __global char* restrict outputImageDX,
__global char* restrict outputImageDY)
{
       /*
       * A single value is computed per work-item.
       */
    const int column = get_global_id(0);
    const int row = get_global_id(1);

    const int offset = row * width + column;

    /* First row of input. */
    uchar leftLoad = inputImage[offset + 0];
    uchar middleLoad = inputImage[offset + 1];
    uchar rightLoad = inputImage[offset + 2];

    short leftData = convert_short(leftLoad);
    short middleData = convert_short(middleLoad);
    short rightData = convert_short(rightLoad);

    short dx = rightData - leftData; //compute (-1 x 1)
    short dy = rightData + leftData + middleData * (short)2; // compute (1 2 1)

    /* Second row of input */
    leftLoad = inputImage[offset + width * 1 + 0];
    rightLoad = inputImage[offset + width * 1 + 2];

    leftData = convert_short(leftLoad);
    rightData = convert_short(rightLoad);

    dx += (rightData - leftData) * (short)2;

    /* Third row of input. */
    leftLoad = inputImage[offset + width * 2 + 0];
    middleLoad = inputImage[offset + width * 2 + 1];
    rightLoad = inputImage[offset + width * 2 + 2];

    leftData = convert_short(leftLoad);
    middleData = convert_short(middleLoad);
    rightData = convert_short(rightLoad);

    dx += rightData - leftData;
    dy -= rightData + leftData + middleData * (short)2;

    /* Store the results */
    outputImageDX[offset + width + 1] = convert_char(dx >> 3);
    outputImageDY[offset + width + 1] = convert_char(dy >> 3);
}
```

Now, let's look at one of the more optimal ways of rewriting this kernel (that has almost 9x improvement in performance). The following kernel performs 2 char16 load operations for the leftmost and rightmost vectors, and reconstructs the middle vector by swizzle operations.

```
/**
* \brief Sobel filter kernel function.
*
* We compute a char16 per work-item, but load 2 char16.
*/
__kernel void sobel_char16_swz(__global const uchar* restrict inputImage, const int width,
                                                  __global char* restrict outputImageDX, __global char*
restrict outputImageDY)
{
    /* Each kernel calculates 16 output pixels in the same row. */
    const int column = get_global_id(0) * 16;
    const int row = get_global_id(1) * 1;

    const int offset = row * width + column;
    /*
     * First row of input.
     * We load the necessary 18 pixels by using two 16-component
     * loads, and the conversion step selects the components it needs.
     */
    uchar16 leftLoad = vload16(0, inputImage + (offset + 0));

    short16 leftData = convert_short16(leftLoad);
    short16 dx = -leftData;
    short16 dy = leftData;

    uchar16 rightLoad = vload16(0, inputImage + (offset + 2));
    short16 rightData = convert_short16(rightLoad);
    short16 middleData = convert_short16((uchar16)(leftLoad.s12345678,
                                                  rightLoad.s789abcde));

    dx += rightData;
    dy += rightData + middleData * (short)2;

    /* Second row of input */
    leftLoad = vload16(0, inputImage + (offset + width * 1 + 0));
    leftData = convert_short16(leftLoad);
    dx -= leftData*(short)2;

    rightLoad = vload16(0, inputImage + (offset + width * 1 + 2));
    rightData = convert_short16(rightLoad);

    dx += rightData * (short)2;

    /* Handle third row of input and store the result */
    leftLoad = vload16(0, inputImage + (offset + width * 2 + 0));
    leftData = convert_short16(leftLoad);

    dx -= leftData;
    dy -= leftData;

    rightLoad = vload16(0, inputImage + (offset + width * 2 + 2));
    rightData = convert_short16(rightLoad);

    dx += rightData;
    vstore16(convert_char16(dx >> 3), 0, outputImageDX + (offset + width + 1));

       middleData = convert_short16((uchar16)(leftLoad.s12345678, rightLoad.s789abcde));
    dy -= rightData + middleData * (short)2;

    vstore16(convert_char16(dy >> 3), 0, outputImageDY + (offset + width + 1));
}
```

## 8. Load sharing - What do I need to know?

In general it is always a good thing to ensure that both the CPU and GPU run in parallel. Some things that you should try to avoid are:

- Using `clFinish` for synchronization – Often you will need the CPU to access data that has been written by the GPU. One way to do this is using `clFinish`. But this will introduce delays as the call to `clFinish` will have to wait till the GPU Job completes. During that time, the CPU remains idle. This scenario will serialize execution and should be avoided if possible.

- Using any of the `EnqueueMap` operations with a blocking call. Where possible you should consider using `clWaitForEvents` or callbacks to ensure that the CPU and GPU can work in parallel. So something along the lines of the following would work well:
  - Split work into many parts (1, 2, …, n)
  - For i = 1 to n
    - Do CPU processing for part i
    - Submit GPU Jobs for part i
  - For i = 1 to n
    - Wait for GPU job for part i to complete (using clWaitForEvents)
    - Do more work on the CPU for part i

## 9. Things to do or avoid?

- Remember that conversions from int to float are not without cost, and that the execution time for conversions from char to float and back again corresponds to several arithmetic operations .

- Use vload/vstore instructions where possible – Remember that the registers are 128 bit wide. So if possible we can operate on 16 chars at the same time.

## 10. Which extensions are currently supported?

Use `clGetDeviceInfo()` to get a list of the supported extensions.

For example, here is the list of the extensions supported in version r3p0b of the driver (codenamed Dysprosium Beta):

```
"cl_khr_int64_base_atomics",

"cl_khr_int64_extended_atomics",

"cl_khr_global_int32_base_atomics",

"cl_khr_global_int32_extended_atomics",

"cl_khr_local_int32_base_atomics",

"cl_khr_local_int32_extended_atomics",

"cl_khr_byte_addressable_store",

"cl_arm_core_id",

"cl_khr_gl_sharing",

"cl_khr_egl_event",

"cl_khr_egl_image"
```

## 11. Why is the first execution of my kernel longer than the following ones?

This is because the allocation of some buffers in certain cases is delayed until the first time they are used:

If you call `clCreateBuffer` using `CL_MEM_COPY_HOST_PTR` then the buffer will be allocated immediately, otherwise it will be allocated when you first map or use the buffer in a kernel.

So if you are not happy with the overhead being part of the first kernel execution then you can try to map/unmap the buffer just after creating it: this will force the driver to allocate it.

What usually happens:

```
clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)  /* doesn't do anything */

clEnqueueNDRangeKernel() /* Allocate the buffer + execute the kernel */

clEnqueueNDRangeKernel() /* execute the kernel */

...
```

This will move the overhead to `clCreateBuffer()` (but will take more time as it will also have to copy the content of the host pointer passed to the function).

```
clCreateBuffer(CL_MEM_COPY_HOST_PTR)  /* Allocate the buffer + memcpy */
clEnqueueNDRangeKernel() /* execute the kernel */
clEnqueueNDRangeKernel() /* execute the kernel */
...
```

Will move the overhead to clEnqueueMapBuffer:

```
clCreateBuffer(CL_MEM_ALLOC_HOST_PTR)  /* doesn't do anything */
clEnqueueMapBuffer() /* allocate the buffer */
clEnqueueNDRangeKernel() /* execute the kernel */
clEnqueueNDRangeKernel() /* execute the kernel */
```

## 12. What is the minimum number of threads I need to ask for so that the driver starts all of the cores?

You need to ask for the `max_local_workgroup_size` x `task_queue_depth` x `number_of_cores`, which on an ARM Mali-T604 is: 256 x 4 x 4 = 4096

## 13. On some other architectures it's recommended to use floats – why is that not the case on Mali?

Most other architectures have a scalar architecture: it means each instruction operates on a single element (an int, a float, etc) and because the GPU is mainly optimized for graphics, which involves a lot of floating point operations, floats are faster on these architectures.

Mali GPUs have a vector architecture: it means instruction operates on a vector of values. Each register is 128bits wide, which means you can operate in a single instruction on a vector of 4x32bits (int, floats) or 8x16bits (shorts, half-floats), or 16x8bits (char,uchar).

So, for example, if you are doing some image processing (input / output being 8 bits) then using fixed point arithmetic with shorts might be accurate enough, in which case you will be able to process 8 elements per instruction instead of 4 if you were using floats.

So, to summarize: which type you use on Mali doesn't matter (ints and floats are equally fast). What matters is how many elements of this type can you fit in a 128 bits wide register (you can fit twice as many shorts as floats or ints).

## 14. Does the driver support half floats?

Not yet, this will come with the r4p0 version of the driver.

## 15. How do I achieve 68 GFLOPS/sec with a quad-core ARM Mali-T604 at 533 MHz?

Each ALU is capable of doing 17ops on 32-bit floats (not 4x32-bit floats, some of the 17 are made from 4 wide vector ops though, but not all).

Also, if you are using shorts, e.g. i16, then you can do twice as many operations for the vector ops.

So flops are made up from:

+7: dot product (4 Muls, 3 adds)

+1: scalar add

+4: vec4 add

+4: vec4 multiply

+1: scalar multiply

All added together makes 17.

17 ALU ops on 4x32-bit floats multiplied by 2 ALUs multiplied by 4 cores on a T604 = 68GFlops for FP32 operations.

## 16. In which DDK release will the CL/GLES interop be available?

This will be supported in the r4p0 version of the driver.

## 17. What is the cycle penalty associated with branch operations?

Branches are free if they can be merged with an arithmetic instruction.

However evaluating the condition might be expensive if the condition is complex.

For example a range such as

```
if x>(width-1) or y>(height-1) then branch
```

will take more than one cycle to evaluate the expression.