Application Note

Precisions on Arm® Mali™ GPUs

Version 1.0

Non-Confidential

Precisions on Arm® Mali™ GPUs

Release Information

The following changes have been made to this Application Note.

Document History

Date	Issue	Confidentiality	Change
20/04/2018	**A**	Non-Confidential - Published	First release

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

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

Web Address

http://www.arm.com

Contents

Precisions on Arm® Mali™ GPUs

1 Conventions and Feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

monospace denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

mono<u>space</u> denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic
denotes arguments to commands and functions where the argument is to be replaced by a specific value.

monospace bold
denotes language keywords when used outside example code.

italic	highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for Arm® processor signal names.

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- Your name and company.
- The serial number of the product.
- Details of the release you are using.
- Details of the platform you are using, such as the hardware platform, operating system type and version.
- A small standalone sample of code that reproduces the problem.
- A clear explanation of what you expected to happen, and what actually happened.
- The commands you used, including any command-line options.
- Sample output illustrating the problem.
- The version string of the tools, including the version number and build numbers.

Feedback on documentation

If you have comments on the documentation, e-mail errata@arm.com. Give:

- The title.
- The number, ARM-ECM-0442477, A.
- If viewing online, the topic names to which your comments apply.
- If viewing a PDF version of a document, the page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Arm periodically provides updates and corrections to its documentation on the Arm Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- Arm Information Center, http://infocenter.arm.com/help/index.jsp.
- Arm Technical Support Knowledge Articles, http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html.
- Arm Support and Maintenance, http://www.arm.com/support/services/support-maintenance.php.
- Arm Glossary, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

2 Preface

This Application Note is intended for developers/programmers/users who use Mali GPUs. This Application Note gives you a basic understanding of how the different levels of precision are used on Mali GPUs.

2.1 References

- https://community.arm.com/graphics/f/discussions/7334/is-there-practical-examples-of-half-float-fp16
- https://community.arm.com/graphics/b/blog/posts/benchmarking-floating-point-precision-in-mobile-gpus
- https://community.arm.com/graphics/b/blog/posts/benchmarking-floating-point-precision-in-mobile-gpus---part-ii
- https://community.arm.com/graphics/b/blog/posts/at-home-on-the-range---why-floating-point-formats-matter-in-graphics
- https://www.khronos.org/opengles/
- https://www.khronos.org/opengl/wiki/Depth_Buffer_Precision
- https://docs.unity3d.com/Manual/SL-DataTypesAndPrecision.html
- https://docs.unity3d.com/Manual/SL-ShaderPerformance.html

2.2 Terms and abbreviations

highp

high precision

mediump

medium precision

lowp

low precision

fp32

floating point 32 bits

fp16

floating point 16 bits

RTZ

A kind of rounding off algorithm**.** Round Toward Zero

RNE

A kind of rounding off algorithm. Round to Nearest Even

3 highp, mediump, lowp

The three qualifier highp, mediump and lowp are used to specify different precisions for a variable. The variable must be an integer or a floating-point scalar or a vector or matrix based on these types. The precision qualifier precedes the type in the variable declaration.

3.1 Getting Started

3.1.1 Preparation

Using floating point arithmetic is tricky. Some innocent-looking pieces of code could be the root cause of issues. How to choose just-enough precision in a shader is not an easy task. This application note provides an initial idea and basic discussion about this topic.

If you are not familiar with how floating-point works, https://community.arm.com/graphics/b/blog/posts/benchmarking-floating-point-precision-in-mobile-gpus and https://en.wikipedia.org/wiki/Single-precision_floating-point_format could be a good start.

Desktop GPUs generally always use fp32 but mobile platforms, such as Mali, can use different precisions and this can lead to considerable differences in output.

3.1.2 Benefit of lower precision

Lower precision data types provide several efficiency advantages:

1. The hardware needed for narrower arithmetic units is smaller and we need to toggle fewer transistors – and thus reduce energy consumed – per operation.
2. The hardware can often pack vectors of narrower operations together – such as issuing a pair of fp16 operations instead of a single fp32 operation – which will improve overall performance.
3. Narrow data in memory requires less storage space, so we reduce external memory bandwidth – which is always good for energy saving as DDR access is very expensive in energy usage – and will be able to fit more data concurrently into the data caches and register storage which will also help performance.

The trade-off of using a narrower type is that they can only represent a smaller range of numbers and a reduced precision, which might result in a rendering error sometimes.

3.2 Precision lost case study

Here listed some instructive examples for your reference.

3.2.1 Keep-increasing range might cause precision lost

In general, please consider carefully both the precision and the range of the numbers you are passing into your shaders, especially fragment shaders. Using too big range wastes precision, and when precision is limited, the effects can be unexpected.

Below is a very simple animated pixel shader that uses the time passed in application to pulse a red light repeatedly:

---------- Fragment Shader Code Snip Start ----------

```
uniform mediump float animation_time;

gl_FragColor = vec3(fract(animation_time), 0.0, 0.0);
```

---------- Fragment Shader Code Snip End ----------

---------- C++ code logic in application Start ----------

```
GLint location = glGetUniformLocation (myProgramObject, "animation_time");
float animation_time = 0.0f;
while (game_not_finished_yet)
{
glUniform1f(location, true, animation_time);
/* ... do some rendering here ... */
/* Now advance the time (assume 30 frames/sec) */
animation_time += 0.033f; /* 0.033s = 33ms per frame */
}
```

---------- C++ code logic in application End ----------

It looks innocent at the first glance, and it even looks fine when running the shader for the first several seconds. But just after a little thought, or after a little bit longer running, a degradation might be seen and finally leads to complete failure results within two minutes. Supposed that RTZ (rounding to zero) is used in implementation, we define fault-ratios as “delta between theoretical value and practical value divided by theoretical value” and could get an increasing fault-ratio result over time, as below,

First frame, animation_time = 0.033f (mediump),

Theoretical colour is fract(0.033f) = 0.033f,

The mathematical representation of mediump 0.033f is $(-1)^0 \times 2^{-5} \times 1.0000111001 = 0.032989501953125$,

```
Practical colour is fract (0.032989501953125) = 0.032989501953125,
Delta in Colour = 0.033 – 0.032989501953125 = 0.000010498046875
Fault Ratio = 0.000010498046875/0.33 = 0.03…%
```

Ten seconds later, animation_time = 10.033f,

```
Theoretical colour is fract(10.033f) = 0.033f,
The mathematical representation of mediump 10.033f is (-1)^0 × 2^3 × 1.0100000100 =
10.03125,
Practical colour is fract (10.03125) = 0.03125,
Delta in Colour = 0.033 – 0.03125 = 0.00175
Fault Ratio = Delta/0.033 = 5.303…%
```

One minutes later, animation_time = 60.033f,

```
Theoretical colour is fract(60.033f) = 0.033f,
60.033 => (-1)^0 × 2^5 × 1.1110000001 = 60.03125
Practical colour is fract (60.03125) = 0.03125,
Delta in Colour = 0.033 – 0.03125 = 0.00175
Fault Ratio = Delta/0.033 = 5.303…%
```

75 seconds later, animation_time = 75.033f,

```
Theoretical colour is fract(75.033f) = 0.033f,
75.033 => (-1)^0 × 2^6 × 1.0010110000 = 75
Practical colour is fract (75) = 0,
Delta in Colour = 0.033 – 0 = 0.033
Fault Ratio = Delta/0.033 = 100%
```

3.2.2 Calculation between values with big different exponents might cause precision lost

Suppose we are using a typical fp32 float, there will be 1 bit of sign, 8 bits of exponent and 23 bits of fraction. Following is a common example to lose precision. If we want to add two numbers, say sixteen million and 11.3125:

```
16000000 = (-1)^0 × 2^23 × 1.11101000010010000000000
11.31250 = (-1)^0 × 2^3 × 1.01101010000000000000000
```

To add them, we first right-shift the significand of the smaller number to make the exponents equal. In this case, we have to shift by 20 bits:

```
16000000 = (-1)^0 × 2^23 × 1.11101000010010000000000
11.31250 = (-1)^0 × 2^23 × 0.00000000000000000001011(010100...00)
```

... and then add the significands to get the result:

```
16000011 = (-1)^0 × 2^23 × 1.11101000010010000001011
```

Note that some of the bits of the smaller number (in red above) got shifted off the end of the significand and fell on the floor, so our result is off by 0.3125; this is a common way to lose precision when you're doing floating-point arithmetic. The bigger the difference in the exponents of the two numbers you're adding, the more bits you lose.

3.2.3 High level Power operation might lead to precision lost

Suppose we are using medium precision (fp16), there will be 5 bits of exponent and 11 bits of significand. If we want to get the fourth power of a small fp16 value, say 0.34375f,

$$0.34375 = (-1)^0 \times 2^{-2} \times 1.0110000000$$

$$\text{Power}\ (0.34375, 4) = (-1)^0 \times 2^{-7} * 1.1100100110(001)$$

Here, the higher power value you used in the algorithm, the more precision lost in the result.

3.2.4 Lower precision is usually insufficient when making texture lookups

Suppose we are using medium precision (fp16), there will be 5 bits of exponent and 11 bits of significand.

11 bits of significand means that the minimum distinguishable value is 1/2048. It means that fp16 is not enough to sample textures whose size is larger than 2048.

Another thing to consider for texturing is the need for sub-texel accuracy when using GL_LINEAR filtering (or similar). In detail, when you are filtering using GL_LINEAR filtering your sample point is falling somewhere between two texels. If you want stable filtering between those values, you want a minimum number of “steps” you can address to provide a smooth gradient between the contributing texel values. Hence, you probably want 8 sub-texel stops, which reduces this to only being useful for addressing textures which are only 256 (=2048/8) texels wide.

Finally, if you have any kind of UV wrapping that repeatedly tiles a texture over a surface, then that makes this even worse. E.g. the coordinate range may be 0-4 rather than 0-1 if you want to repeat a texture 4 times, which reduce the 256 to 64 (=256/4). In such case, fp16 is not enough to sampling textures whose width or height is larger than 64.

3.3 What happened next after precision lost

The easiest way is just to drop the red bits in last section (bits exceed of precision limit) on the floor; in the numeric business, that's called round-toward-zero (RTZ) or truncation. It is equivalent to pretending the red bits are all zero, even if they aren't.

The other way is rounding to nearest. Instead of dropping the red bits, we can round them up or down to whichever 24-bit significand value is closer. GPUs that perform RNE rounding (a kind of rounding to nearest rule), such as Arm's Mali-T604, could produce more accurate results and lower error(half the error of RTZ).

Different rule to handle the precision lost could lead to different rendering results.

If any of the intermediate results in your algorithm has such precision lost, and the final rendering result is sensitive to lower significant bits which might have been lost, unexpected observable issue might occur.

3.4 General rule to choose precision

It is important to understand how to select “just-enough” precisions for your calculations.

1. For anything related with position (vertex positions, uniform matrices for position transform, distance computation for lighting, etc), we’d generally recommend using highp/fp32.
2. For texture coordinates, there are two independent things here --- the precision of data stored in memory and the precision inside the shader program. It might usually be OK with mediump for storage in memory (e.g. if you know there is no texture repeated or mirrored and the input texture is only 1024x1024, which is common in games), but you need highp varyings in shaders for interpolation when you start to need the sub-texture addressing for filtering. If general, we recommend using highp varying input variables in the fragment shaders for texture coordinates, ensuring fp32 interpolation precision.
3. For texture samplers, most modern content will use 24-bit uniform integer or 32-bit float depth buffers. To sample data from these textures without losing data precision the texture sampler must be a highp sampler. Besides, OpenGL ES 3.0 introduces the concept of general purpose 32-bit per channel textures, both for floating point and integer data types. Given their wider data width, using anything other than a highp sampler would result in data truncation.
4. For anything related with colour, or intermediate values which will turn into a colour at some point (such as normal value for lighting), then fp16 is probably enough.
5. Sometimes, choosing the right precision is not an easy job, even for very good programmer. Make sure your application is fully tested under “different GPU implementations” could be a safeguard to avoid precision issues.

3.5 Suggestion on Mali GPUs

OpenGL ES 2.0/3.0 spec has defined the minimum precision requirement for highp, mediump, and lowp. Different implementation could have their own precisions based on it. Here listed the output of API--glGetShaderPrecisionFormat on Mali GPUs.

		Midgard/Bifrost			Utgard		
		Floating range	Floating Precison	Inter Range	Floating range	Floating Precison	Inter Range
Vertex Shader	highp	-2^127 ~ 2^127	2^-23	-2^31 ~ 2^30	-2^127 ~ 2^127	2^-23	-2^24 ~ 2^24
	mediump	-2^15 ~ 2^15	2^-10	-2^15 ~ 2^14	-2^127 ~ 2^127	2^-23	-2^24 ~ 2^24
	lowp	-2^15 ~ 2^15	2^-10	-2^15 ~ 2^14	-2^127 ~ 2^127	2^-23	-2^24 ~ 2^24
Fragment Shader	highp	-2^127 ~ 2^127	2^-23	-2^31 ~ 2^30	0-0	0	0-0
	mediump	-2^15 ~ 2^15	2^-10	-2^15 ~ 2^14	-2^15 ~ 2^15	2^-10	-2^11 ~ 2^11
	lowp	-2^15 ~ 2^15	2^-10	-2^15 ~ 2^14	-2^15 ~ 2^15	2^-10	-2^11 ~ 2^11

Note that: It could be reasonably sure that the reported precision of this API is correct for the basic arithmetic instructions. But it doesn't capture all details, such as rounding modes and interpolation precision. It is also possible that implementations could convert medium to highp in some cases. So, it is not guaranteed that implementations with same precisions must generate the same rendering result.

List several hints here on which you need to pay more attention when developing applications on Mali Platform.

1. Mali GPUs do not distinguish between lowp and mediump variables, so both are mapped to 16-bit data types, and highp variables are mapped to 32-bit data types. However, the older Mali-400 series GPUs – based on the Utgard architecture – do not support highp processing in fragment shaders, so all variables will be treated at 16-bit variables except texture-coordinate and varying-load which are fp24 (Using varying to make texture lookups directly can get a fp24 precision. However, using temp register to do so is only fp16 precision). It is the reason why some highp needed algorithm could generate observable defect in rendering result on Utgard. Modifying the algorithm to less depending on precision could be necessary here.
2. On Midgard/Bifrost GPUs, suggest using highp for texture coordinates varyings both in VS and FS, unless highp could lead to obvious side effect (e.g. obvious bandwidth increasing due to complicated rendering mesh). Mali GPUs pay more attention on bandwidth/power saving rather than precisions. If mediump is specified for a texture coordinates varying either in VS or FS, implementation could decide whether to upgrade mediump to highp. Staying at mediump might save bandwidth and power but take the risk of getting an unexpected rendering result on some not-well-programmed applications. On the contrast, upgrading to highp might bring a waste on bandwidth and power (depends on the mesh complexity). It is a trade-off, and Mali decide to pay more attention to bandwidth and power saving. This might be one reason why some special precision issues only exist on Mali GPUs.

4 Depth buffer precision

About depth buffer precision, please refer to https://www.khronos.org/opengl/wiki/Depth_Buffer_Precision for detailed explanation. We just list the brief conclusion from it.

4.1 zNear and zFar

You may have configured your zNear and zFar clipping planes in a way that severely limits your depth buffer precision. Generally, this is caused by a zNear clipping plane value that's too close to 0.0. As the zNear clipping plane is set increasingly closer to 0.0, the effective precision of the depth buffer decreases dramatically. Moving the zFar clipping plane further away from the eye always has a negative impact on depth buffer precision, but it's not one as dramatic as moving the zNear clipping plane.

In short, push the zNear clipping plane out, and pull the zFar plane in as much as possible.

4.2 Pay more attention to polygons near zFar

The perspective divide, by its nature, causes more Z precision close to the front of the view volume than near the back. As a result, polygons near zFar plan is more likely to bleed through the nearby polygons in front of them.

4.3 glPolygonOffset and glDepthRange

For coplanar primitives, round-off errors or differences in rasterization typically create "Z fighting". Using glPolygonOffset could generally be an easier choice, although glDepthRange would be another option.

For astronomically large scene, typical approach is using multi-pass rendering to divide the scene objects to regions which don't interfere with each other in Z, then render regions separately and combine the rendering results of each region together in the final pass.