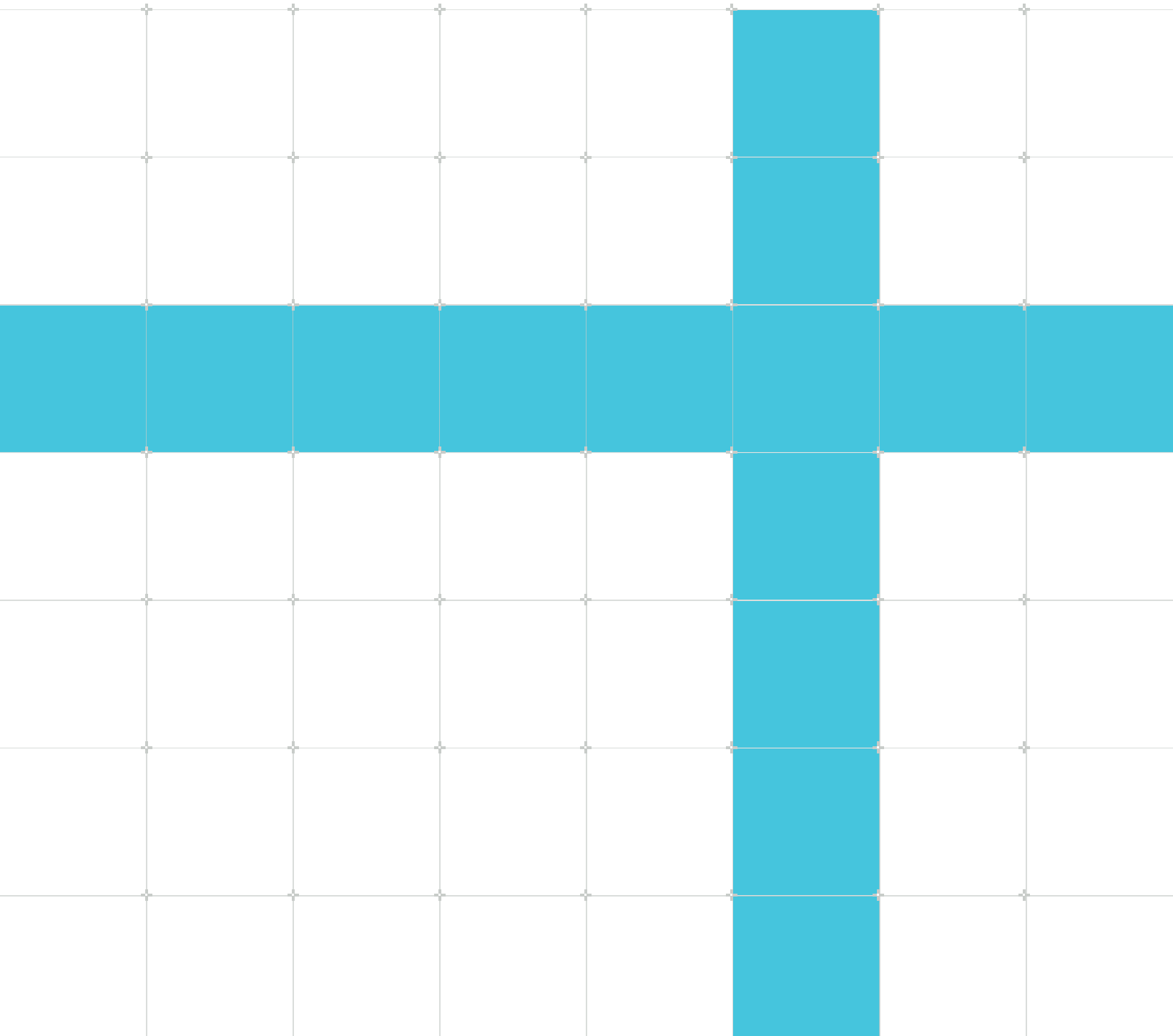
Understanding numerical precision

Version 1.0

Issue 00
102502_0100_00_en



Understanding numerical precision

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	10 July 2019	Non-Confidential	First release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

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.

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview....6

2. Numerical Precision....7

3. Shader precision....8

4. The pros and cons of lower precision....9

5. Debugging....11

6. Mitigating loss of precision....12

7. Next steps....14

1. Overview

This guide explores the different levels of numerical precision available for a GPU. It explains the advantages of using narrower data types, and when you might consider using the higher precision types available instead.

This guide is designed for application developers who have an understanding of developing shaders and want to increase their knowledge and optimize their shader performance further.

When you have finished this guide, you will understand both 16 floating-point values and 32 floating-point values, shader precision, rounding values, the pros and cons of lower precision, texture coordinates, and how to mitigate the loss of precision.

2. Numerical Precision

Data-plane processing is about processing a large amount of data and computation as efficiently as possible. Using algorithms in narrow data types through the processing path is one of the main tricks used improve efficiency.

For most graphics algorithms, the eventual color outputs from a render are likely to be RGBA8 surfaces that only store 8 bits per color channel. Therefore, it is not necessary to use the full FP32 floating-point value.

A FP16 value is a viable alternative for color computation. In many compute problem domains, it is possible to go even further towards smaller data types.

3. Shader precision

The OpenGL ES and Vulkan graphics standards have their own definitions of data types. Variables are declared using a precision qualifier that defines the minimum precision that the implementation can use.

However, an implementation can substitute a more precise variable if required. As shown in the table below, the specification for the OpenGL ES shader language for floating-point values states that:

Declaration	Range	Magnitude	Precision
`lowp`	[-2, 2]	[2^-8, 2]	Absolute 2^-8
`mediump`	[-2^14, 2^14]	[2^-14, 2^14]	Relative 2^-10
`highp`	[-2^62, 2^62]	[2^-62, 2^62]	Relative 2^-16

`lowp` float values can be stored using a 10-bit fixed point, `mediump` float values can be stored using a 16-bit floating point, and `highp` float values can be stored using a 32-bit floating point. However, the resulting output depends on the underlying GPU.

Mali GPUs do not distinguish between `lowp` and `mediump` variables. This means that both are mapped to 16-bit data types, and `highp` variables are mapped to 32-bit data types.


Note

The older Mali-400 series GPUs, which are based on the Utgard architecture, do not support `highp` processing in fragment shaders. Therefore, all variables will be treated at 16-bit variables when using Mali-400 series GPUs.

Rounding modes

In addition to the floating-point precision differences, consider how floating-point values are rounded if a result cannot be exactly represented. There are broadly three main categories of floating-point rounding available:

1. Round to nearest.
2. Round toward 0.
3. Round away from 0.

Round to nearest gives the least additional error, up to 0.5 ULP, compared to up to 1ULP for round toward 0 and round away from 0. However, round to nearest is slightly more expensive to support in hardware. As with numerical precision, the OpenGL ES graphics specifications does not tightly define the implementation.

Mali GPUs default to rounding to nearest, although this can be overridden by the application in some versions of OpenCL.

4. The pros and cons of lower precision

Lower precision data types provide a variety of efficiency advantages:

- The hardware needed for narrower arithmetic units is smaller, and fewer transistors need to be toggled. This means that each operation uses less energy.
- Overall performance can be improved by packing vectors of narrower operations together. For example, it is possible to issue a pair of FP16 operations instead of a single FP32 operation.
- Narrow data in memory requires less storage space and reduces the need for expensive external DDR memory, allowing more data to fit into both the data caches and register storage concurrently, improving performance.

However, the trade-off is that narrower data types can only represent a smaller range of numbers. Such as:

- Both floating-point types and integer types suffer from a reduced dynamic range. For example, an FP32 float can store a value with a range of up to 2^62. Compare this to an FP16 float that can only store a value with a range of up to 2^14.
- Floating-point types also suffer from reduced precision inside any given dynamic range. FP32 values provide 24 fractional bits, and an FP16 float provides 11 fractional bits.

Therefore, we recommend that you use narrow types, except when they provide insufficient precision and would result in a rendering error.

In general, any graphics application should be using a mixture of FP types. There are many cases where an FP16 is fine, but there are also cases where this is insufficient, and an FP32 should be used instead.

Because FP16 values offer twice as much energy efficiency and performance as FP32 values, FP16 values should be used in applications when possible. This means that it is easier to consider the cases in which using `mediump` is insufficient.

Vertex positions

To ensure output position accuracy and stability of the vertex position in the vertex shader, we recommend that you use `highp`. Always use `highp` for input positions, transform matrices, and for any distance-based computation for lighting.

Texture coordinates

Textures are addressed with a UV coordinate between 0 and 1. Using an FP16 coordinate gives 11 fractional bits, with an accuracy of 1 part in 2048. This means it is unable to accurately address common texture sizes such as 1440p (2560x1440 pixel) renders, even when using `GL_NEAREST` filtering.

Many games use smaller textures than this, such as 512x512 pixels or 1024x1024 pixels. But, many games also use `GL_LINEAR` for filtering.

For smooth linear interpolation during filtering and stable addressing at a sub-texel accuracy, we recommend that you use at least 16 sub-texel indices. This means that even for some smaller textures, FP16 is insufficient.

For both reasons, we recommend using `highp` varying input variables in the fragment shaders for texture coordinates, to ensure FP32 interpolation precision. However, if the texture is less than, or equal to, 2048 texels wide in each dimension, it is not normally required to store texture coordinate inputs or outputs for the vertex shader at a higher precision.

Storing data in input attribute buffers as `GL_HALF_FLOAT` and writing `mediump` outputs from vertex shaders minimizes the memory bandwidth needed to store coordinates in memory, and loading them as `highp` inputs in fragment shaders gives the high precision interpolation.

Note: The Mali-400 series of GPUs do not support highp operations in the fragment shader math units. However, the 400-series GPUs do include a higher precision path between the varying interpolator and the texture sampling unit. To avoid losing this additional precision for texture coordinates, the interpolated varying value must pass directly into the `texture2D()` call. Any arithmetic computation on the coordinate before use results in a drop to FP16 precision, and a subsequent drop in sample position accuracy.

Depth samplers

Most modern content will use 24-bit unsigned normalized integers or 32-bit float depth buffers. To sample data from these textures without losing data precision, the texture sampler must be a `highp` sampler.

-bit per channel texture formats

OpenGL ES 3.0 introduces 32-bit per channel textures, for both floating-point and integer data types. Given their wider data width, using anything other than a `highp` sampler would result in data truncation.

While 32-bit texture channels are available, we do not recommend using them due to their high memory bandwidth and energy efficiency costs.

-bit render targets

OpenGL ES 3.0 and OpenGL ES 3.2 introduces 32-bit per channel output framebuffer attachments, both for integer data, in ES 3.0, and floating-point data, in ES 3.2 data. Given their wider data width, using anything other than a `highp` computation and output in the shader program results in data truncation.

While 32-bit channel output framebuffers are available, we do not recommend using them due to their high memory bandwidth and energy efficiency costs.

5. Debugging

Debugging precision issues can be difficult, because it is hard to know precisely which computation is overflowing. Some trial-and-error debugging techniques are required to identify the culprit.

Forcing highp in shaders

For draw calls that use `highp` precision, forcing every variable in the shaders is the quickest way to confirm their precision and to see if the issues are resolved. If they are resolved, you can reintroduce lower precision until you find the computations which are at fault. Mitigation strategies can then be implemented.

Increasing depth precision

With any geometry Z-fighting problems, it is always worth reviewing the precision of your depth buffers, to ensure that you have sufficient depth precision to avoid quantization issues with coplanar geometry.

You can also determine if depth precision is a likely cause of rendering issues, by swapping to a 32-bit floating-point depth buffer, `DEPTH_COMPONENT_32F`.

Note: We recommend avoiding 16-bit depth buffers, as they are prone to precision problems.

6. Mitigating loss of precision

The basic idea of floating-point numbers is that the location of the fractional bits which are stored changes, or floats, is based on the magnitude of the number you are trying to represent. The level of accuracy that you can store reduces as the magnitude of the stored number increases.

For many types of shader arithmetic, accuracy of small numbers is important. Examples include: unorm color outputs, UV texture coordinates, and components in unit length vectors in which all values are between 0.0 and 1.0.

Preserving accuracy of the numbers in this output range is important. Therefore, we will now discuss how you can use mathematical construction to reduce the errors introduced by precision limitations.

Avoid large magnitudes

Avoid creating numbers with a large magnitude that will be turned in to a small number in mathematical operations. For example, consider the expression:

```
glsl
float opA = 100.00;
float opB =   0.01;
float tmp = (a + b)
float result = tmp - a;
```

When executed at FP32 precision, this expression gives the expected answer 100.01. But, when executed at FP16 precision, this expression gives the answer 99.989.

This happens because of the large difference in magnitudes of the original inputs. This means that the intermediate value of `tmp` lacks enough accuracy to store the fractional part of 100.01, and so only contains the value 100. However, the smaller value `tmp - a` can be stored, meaning that the errors do not cancel out.

To avoid losing accuracy, construct equations that preserve intermediate values, so they are as close as possible to the final magnitude. For example, if passing in a rotation from the application into `sin()` or `cos()`, we know that the useful part of the function can be found between [0, 2(PI)). Any values that are higher than this are just repeated rotations larger than 360 degrees, and are visually indistinguishable from a smaller rotation.

So rather than passing in an ever-increasing value from the application, wrap the rotation on the CPU to the range [0, 2(PI)), in turn, preserving as much precision as possible in the useful range.

For this example, if the rotation is not wrapped to a small range on the CPU, then the object eventually ceases rotating. The magnitude of the number becomes so large that adding in a small incremental rotation does not do anything. This is because the small increment is below the accuracy threshold of the stored number.

This happens quickly with FP16 numbers, but it also happens eventually with FP32 numbers.

Exploit symmetrical functions

The sign-bit is always stored in a floating-point number. For many types of periodic mathematical functions, this can be used to improve accuracy because the magnitude of the numbers that need to be stored can be reduced.

For example, a rotation of +270 degrees is the same as a rotation of -90 degrees. So, for inputs into `sin()` and `cos()`, it is preferable to use values in the range [-(PI), +(PI)) instead of [0, 2(PI)). This is because the -PI to +PI range halves the maximum magnitude, therefore preserving one bit of accuracy which the latter values would lose.

Exploit built-in functions

Built-in functions in the shader libraries are often backed by hardware that preserves more precision than the equivalent function that is implemented in shader code arithmetic.

An example of this is the `Fused Multiply Accumulate` operation. This operation is very common in compute applications:

```
glsl
float r = (a * b) + c;
```

If this operation is implemented as separate multiply and add operations, the result of (a * b) is rounded to fit into a `tmp` float. The result of `tmp + c` is rounded again, so that two sets of rounding errors are introduced.

When using a hardware fused multiply accumulate operation, only the final result needs to be rounded to the output precision. This removes the intermediate rounding result, and the error that it introduces.

Minimize memory size

Double Data Rate (DDR) memory bandwidth requires lots of power, so when reviewing shaders and narrowing precision, remember also to narrow any associated vertex attributes stored in memory.

Support for `GL_HALF_FLOAT` attributes is a core feature in OpenGL ES 3.0. If you are using OpenGL ES 2.0, remember that all Mali GPUs support the `[OES_vertex_half_float][VHF]` extension.

OpenGL ES OES Vertex Half-Float Information

A caveat of using lower numerical precision is that, In general, lower precision is better. However, the cost of type conversion may not be free. Therefore, try to minimize the number of casts needed in shader code by loading data at a suitable precision level.

7. Next steps

When it comes to improving the efficiency of using shaders, knowing when it is best to use either 16fp or 32fp values, shader precision, rounding values, the pros and cons of lower precision, texture coordinates, and how to mitigate the loss of precision are all key aspects in your efforts to optimize your shader performance on Mali GPUs.