The Benefits of Buffer Packing

Version 1.0

Non-Confidential

Issue 01

102537_0100_01_en

The Benefits of Buffer Packing

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	21 May 2021	Non-Confidential	First release

Proprietary Notice

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. Understanding buffer packing......7

3. Basic Bandwidth Load Optimization Best Practices......8
3.1 Mesh triangle density......8
3.2 Mesh attribute precision......8
3.3 Mesh spatial locality......8
3.4 Dynamic mesh level-of-detail......9
3.5 State settings......9

4. Index Buffer Encoding......10

5. Attribute Buffer Encoding......11
5.1 Attribute interleaving......11
5.2 Interleaving for position shading......12
5.3 Buffer specialization......14
5.4 Attribute vectorization......15

6. Next steps......16

1. Overview

This guide explains how to make best use of the limited memory bandwidth available to your application on your target device and what memory bandwidth areas can be made more efficient.

This guide is useful for application developers who want to learn how to improve the 3D performance of their application.

By the end of our guide you will have a better understanding of how to improve the efficiency of your bandwidth usage by taking several simple-to-follow steps from reducing the number of vertices used on 3D models, picking the correct data types, utilizing correct LODs, and index buffer encoding.

2. Understanding buffer packing

Due to device thermal limits and energy intensive Double Data Rate (DDR) accesses, one of the biggest limitations of mobile graphics is the amount of memory bandwidth that is available for applications to use.

The use of API-visible lossy texture compression formats, such as Arm's Adaptive Scalable Texture Compression (ASTC), are key bandwidth saving techniques. But it is also important that model vertex data storage in applications is optimized.

Data storage for geometry data is commonly over 64 *bytes* per vertex, compared to 4 *bits* per texel for Ericsson Texture Compression 2 (ETC2). Therefore, any inefficiencies in buffer storage quickly accumulates into significant bandwidth consumption at the system level.

3. Basic Bandwidth Load Optimization Best Practices

Before any advanced optimizations are considered, there are a few high-level best practices that should be followed to minimize the initial bandwidth load.

3.1 Mesh triangle density

The simplest optimization that you can make is reducing the total vertex count in the 3D models. This gives a proportional reduction in vertex bandwidth. Therefore, you need to simplify all 3D models used. For mobile content, we recommend an upper limit of 250,000 input triangles per frame, which equals ~125K visible triangles after clipping and culling.

3.2 Mesh attribute precision

Adjust the data types used to store data in memory. While you can consider treating everything as 32-bit `GL_FLOAT` data types, for many use cases - such as storage of color data - this level of precision is too high.

Instead, use 16-bit `GL_HALF_FLOAT`, or packed formats such as `GL_INT_2_10_10_10_REV`, to minimize storage footprint and data fetch bandwidth. Then, order fields in memory and be ensure that you minimize any buffer space lost to padding.

Intermediate data storage that is required to store vertex shader outputs can be minimized by ensuring that outputs are stored at `mediump` precision rather than `highp` precision.

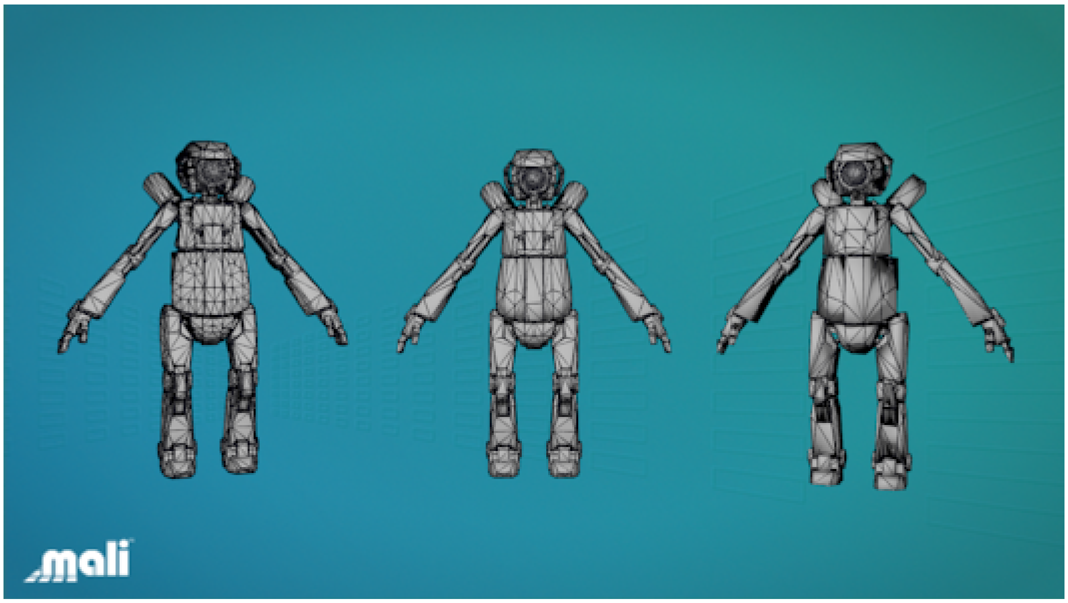
3.3 Mesh spatial locality

Modern GPU s are reliant on data caches to keep recently accessed data close to the processor, rather than constantly fetching data from main memory. The vertices for triangles that are close together on screen must be stored close together in the attribute buffer.

3.4 Dynamic mesh level-of-detail

For 3D games with highly variable scene depth, consider using a dynamic mesh level-of-detail, that selects simpler meshes as the distance between the camera and the object increases. An example of a mesh with a dynamic level-of-detail can be seen below:

Figure 3-1: Dynamic level-of-detail



When an object is only 50 pixels high, it is not necessary to use a model that uses 5000 triangles. This is because very few sample points are hit and they are therefore made completely redundant.

3.5 State settings

When reviewing meshes, check the obvious render state settings that impact mesh processing. For example, whether face culling is correctly enabled for opaque meshes.

4. Index Buffer Encoding

For most non-trivial meshes, we recommend using indexed draw calls. The additional level of indirection that indexing provides allows for vertex reuse along the seams between neighboring triangle strips. This reduces the need for physically duplicated vertices.

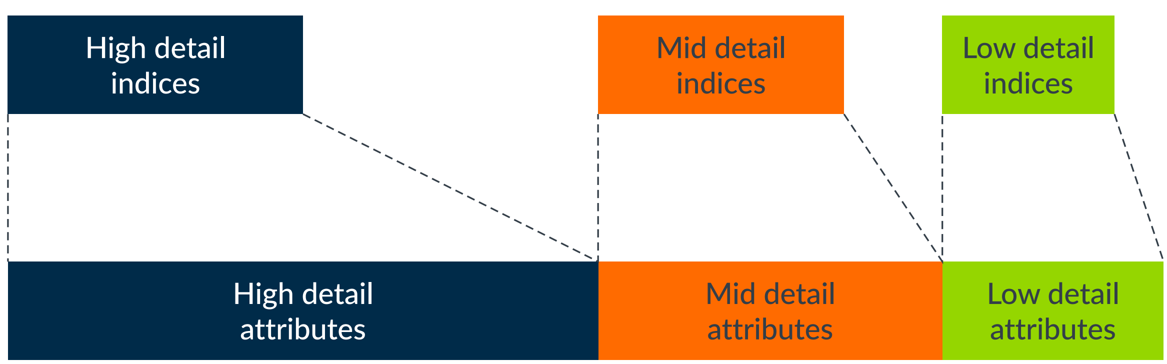
Well-structured meshes have good index locality, so nearby triangles in the model use nearby vertices in the attribute buffer, maximizing the benefits of caching. Also, index buffers should use every vertex between the min and max index used for a draw call. Any vertices that are not referenced are going to cause a loss in performance.

When implementing dynamic level-of-detail for simpler meshes, we recommend making contiguous vertex ranges for each level of detail required. While this adds an additional memory footprint for the duplicated vertices, it ensures minimal bandwidth usage due to improved cache efficiency.

The loss of cache locality from an inefficiency in vertex processing can be reduced by sparse sampling vertices from a high-detail mesh to generate a lower detail version.

The following image shows how different levels of mesh detail are partitioned, with each level of detail stored as a continuous block of vertex data in memory:

Figure 4-1: Block of vertex data



5. Attribute Buffer Encoding

Attribute buffers store the data for each vertex, allowing for the flexible storage of data that is based on a base pointer and row stride. Attribute buffer encoding is the method of how to pack multiple vertex attributes into your memory buffers.

5.1 Attribute interleaving

There are two high-level strategies for attribute storage than an application can choose to use:

1. Non-interleaved: A structure of arrays
2. Interleaved: An array of structures

Non-interleaved storage stores each individual attribute in a unique array, with data fetches for a single vertex gathered from multiple arrays.

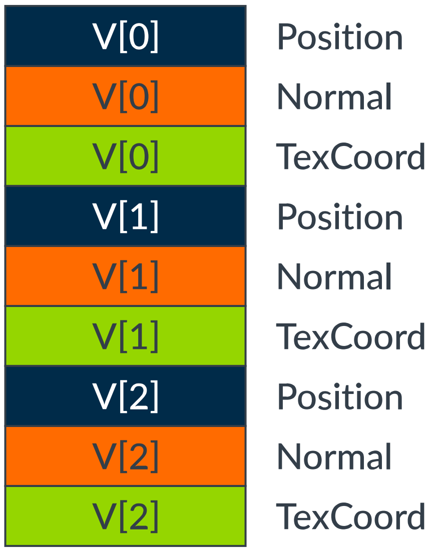
The image below shows you how attribute storage is handled in a non-interleaved buffer packing array:

Figure 5-1: Non-interleaved buffer

Interleaved storage is generally preferred over non-interleaved storage as it stores the different attributes for each vertex serially in memory. Interleaved storage also minimizes the number of unique data fetches needed for each vertex, as well as the number of redundant bytes fetched around the boundaries of the used parts of the vertex buffer.

This image shows you how attribute storage is handled in an interleaved buffer packing array:

Figure 5-2: Interleaved buffer



5.2 Interleaving for position shading

The Bifrost family of Mali GPUs comprises of the Mali-G30/50/70 series. These GPUs implement an optimized vertex processing flow that splits the vertex shader into two pieces: position shading and varying shading.

After primitive assembly, position shading is run, then primitives are put through the fixed-function clip and cull unit, before the varying shading function is run for the vertices that contribute only to visible primitives.

The diagram below shows the vertex processing flow in an IDVS geometry pipeline:

Figure 5-3: vertex processing flow in an IDVS geometry pipeline

Compared to non-interleaved buffer-packing, this approach gives two benefits:

1. Model shading costs are reduced as it does not run the varying shading for culled vertices.

2. The amount of data fetched from culled vertices can be reduced with the application helping to optimize the buffer's layout.

Let's consider the vertex data structure below:

```
c
struct vertex {
    fp32  position[4];
    fp32  xyScale[2];
    fp16  texCoord1[2];
    fp16  texCoord2[2];
    fp16  vertexColor[4];
}
```

And the corresponding vertex shader:

```
glsl
#version 300 es
precision highp float;
uniform mat4 u_mvp;
in vec4 position;
in vec2 xyScale;
in mediump vec2 texCoord1;
in mediump vec2 texCoord2;
in mediump vec4 vertexColor;
out mediump vec2 v_texCoord1;
out mediump vec2 v_texCoord2;
out mediump vec4 v_vertexColor;
void main()
{
    vec4 tmpPos = position;
    tmpPos.xy *= xyScale;
    gl_Position = u_mvp * tmpPos;
    v_texCoord1 = texCoord1;
    v_texCoord2 = texCoord2;
    v_vertexColor = vertexColor;
}
```

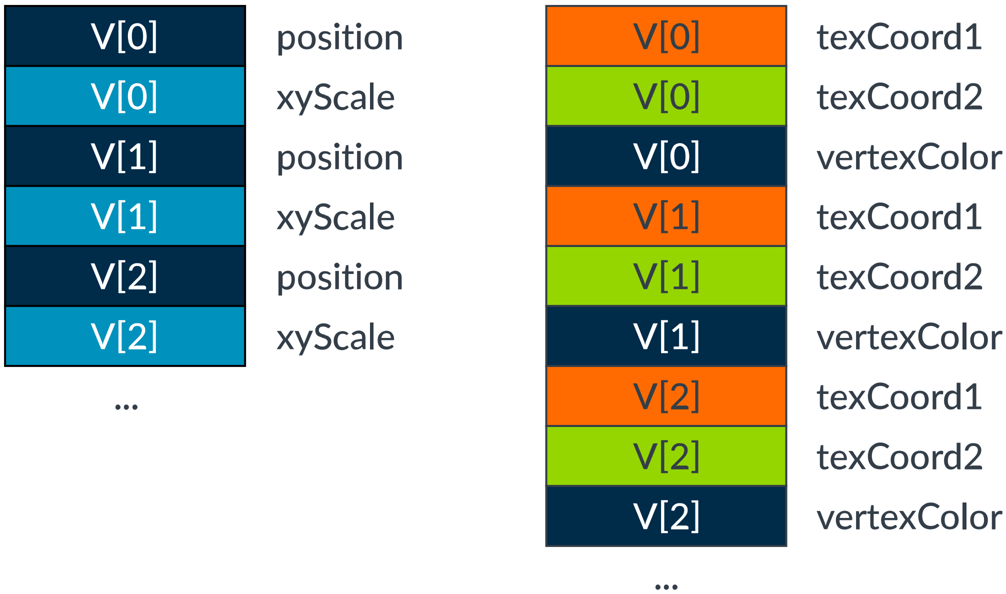
As data is always fetched from main memory as entire 64-byte cache lines, storing this using a single interleaved attribute data set in memory loads a total of 40 bytes per vertex. Even if only `position` and `xyScale` values are used, due to the vertex being culled.

Ideally, in this scenario, only 24 bytes per vertex needs to be loaded for culled vertices. This means that 40% of the read bandwidth is wasted.

To get the full benefit of split position and varying shading in Bifrost GPUs, we recommend using two interleaved data sets. The first data set should interleave all attributes required to compute `gl_Position`, and the second set should contain everything else.

Using two interleaved data sets in this way ensures that only position-related data is read from memory for culled vertices and maximizes the bandwidth savings. The two data sets can be stored as separate sub-regions inside a single buffer, or inside two separate buffers. This is shown in the image below:

Figure 5-4: Stored data in a single or separate buffers



Note

While this type of split packing can small overhead costs on older Mali GPUs, impact is minimal if other best practices, such as ensuring good spatial locality, are also followed.

5.3 Buffer specialization

It is useful to produce specialized attribute data sets for each render pass for complex geometry that is reused in multiple render passes, such as a shadow pass and a color pass.

These specialized versions should strip out the unused attributes for each pass, producing a bandwidth optimized version of the mesh for each use.

Note

For most use cases, the split data sets for position and varying shading already provide optimal position-only data sets for depth shadow map generation. Meaning that further specializations may not be necessary.

5.4 Attribute vectorization

Because all Mali GPUs are vector processors to some extent, the shader compiler can optimize memory accesses more effectively if it has guarantees that data is contiguous in memory.

In the previous example, the shader uploaded a pair of texture coordinates as two different `vec2` attributes.

```
glsl
in mediump vec2 texCoord1;
in mediump vec2 texCoord2;
out mediump vec2 v_texCoord1;
out mediump vec2 v_texCoord2;
```

This means that at compile time the compiler has no guarantee that these are contiguous in memory because the application can change buffer packing at draw time. If this version of the shader is run though the offline compiler for a Mali-T880 GPU, the vertex shader requires 10 load/store cycles to complete and the load/store unit is the critical path.

If a pair of coordinates is uploaded together as a single `vec4`, the compiler is given some guarantees that they are contiguous in memory. This allows it to perform better optimizations:

```
glsl
in mediump vec4 texCoords;
out mediump vec4 v_texCoords;
```

This version of the shader should run more quickly and use less energy as it only requires 8 load/store cycles to complete.

6. Next steps

As you can see, learning how to optimize the memory bandwidth used by your application, and tailoring that optimization for your target device, can offer significant savings when implemented correctly.

And, thankfully, these steps don't have to be overly complicated. Improvements to reduce the number of vertices used on 3D models, picking the correct data types, utilizing correct LODs, and index buffer encoding are steps you can try implementing in your games today.