Mali Offline Compiler

参考：

1. https://developer.arm.com/documentation/101863/0804/Using-Mali-Offline-Compiler/Performance-analysis/Resource-usage

2. https://developer.arm.com/documentation/101863/latest/Using-Mali-Offline-Compiler/Performance-analysis/Shader-properties

3. https://developer.arm.com/Processors/Mali-G720

Performance analysis

Mali Offline Compiler可以生成目标GPU上shader性能的静态报告，如下：

```
Configuration
=============

Hardware: Mali-T880 r2p0
Driver: Midgard r23p0-00rel0
Shader type: OpenGL ES Fragment

Main shader
===========

Work registers: 2 (100% occupancy)
Uniform registers: 2
Stack spilling: false

                             A   LS    T  Bound
Total Instruction Cycles:  6.0  1.0  0.0      A
Shortest Path Cycles:      1.7  1.0  0.0      A
Longest Path Cycles:       1.7  1.0  0.0      A

A = Arithmetic, LS = Load/Store, T = Texture

```

```
Shader properties
=================

Has uniform computation: true
```

Resource usage

这部分主要展示shader程序的硬件资源使用情况，可以看到寄存器，堆栈，共享内存以及16-bit算术单元的使用情况。

Work Registers

工作寄存器指的是每一个线程运行时开辟的读写寄存器。线程运行的并行情况（同时运行的线程数）由每个线程占据的读写寄存器数量来确定（总读写寄存器的量固定，每个线程占据的少，就可以支持更多的线程同时运行）。

减少工作寄存器的使用，可以通过减少使用32bit的数据类型，尽量使用16bit，以及简化shader程序复杂度。

Uniform registers

统一寄存器是每个运行的程序开辟的只读寄存器，用于存储统一变量和字面常量值。如果运行时使用的uniform registers超过了限制，则会回退到每个线程自己去内存中加载。会大量增加LS的耗时，同时增加带宽，降低缓存命中率。

减少uniform registers的使用，可以尽量使用16bit的数据类型，以及shader程序中减少uniform和常量的使用。

Shared storage

共享存储是一个允许一组计算线程交换数据的内存池。Arm GPU使用缓存的RAM来实现共享内存，因此性能与其他缓冲区的访问相同。只有当需要线程间共享数据才使用共享存储。

为了减少共享内存的使用量，考虑使用子组操作作为共享数据的替代方法。

Stack spilling

着色器程序把读写寄存器（work registers）使用完的情况下必须将额外的值溢出到存储在内存中每个线程的堆栈中。栈溢出非常昂贵，因为会涉及到大量的线程。

减少栈溢出的途径有：

1. 减少变量精度
2. 减少变量生命周期
3. 简化shader的复杂度

16-bit arithmetic

Arm GPU的算数运算单元每个时钟周期可以计算一个32bit的运算或者一个vec2 16bit的运算。使用16bit的数据类型可以减少寄存器的压力，同时提高运行效率和减少能耗。

GPT说最好占比是能达到50%-70%。

你可以通过以下途径最大化16-bit arithmetic的占比：

1. 使用低中精度的变量限定符
2. 在二进制的SPIR-V模块中使用RelaxedPrecision的变量限定符

但是，在position，uv，normal，depth的计算中，最好都是用高精度。

> The behaviour you are seeing is mostly a reporting artefact of how we calculate the number of 16-bit operations.
>
> We only count pure arithmetic instructions (interpolation, texturing and blending not included), of which this shader has very few. We also only count instructions that are entirely 16-bit (16-bit input, 16-bit data processing, 16-bit output). The discard instruction only has a 32-bit data path, so will include implicit widening of a 16-bit input. This has no impact on performance, but does mean it doesn't show up in the stats.

这部分的统计只统计纯16bit的运算，不包括插值，采样，混合。纯16bit的算术运算指的是 16-bit的输入，16bit的数据处理，16bit的输出。Discard强制把16bit当成32bit处理，但是对性能不影响，因此也不会出现在统计数据里。

Performance table

这部分提供了着色器潜在的性能指标，所有的数据以时钟消耗数来展示，并且归一化到了单个core的消耗。

Total Instuction Cycles

程序中生成的所有指令的累计时钟数，不考虑控制流。

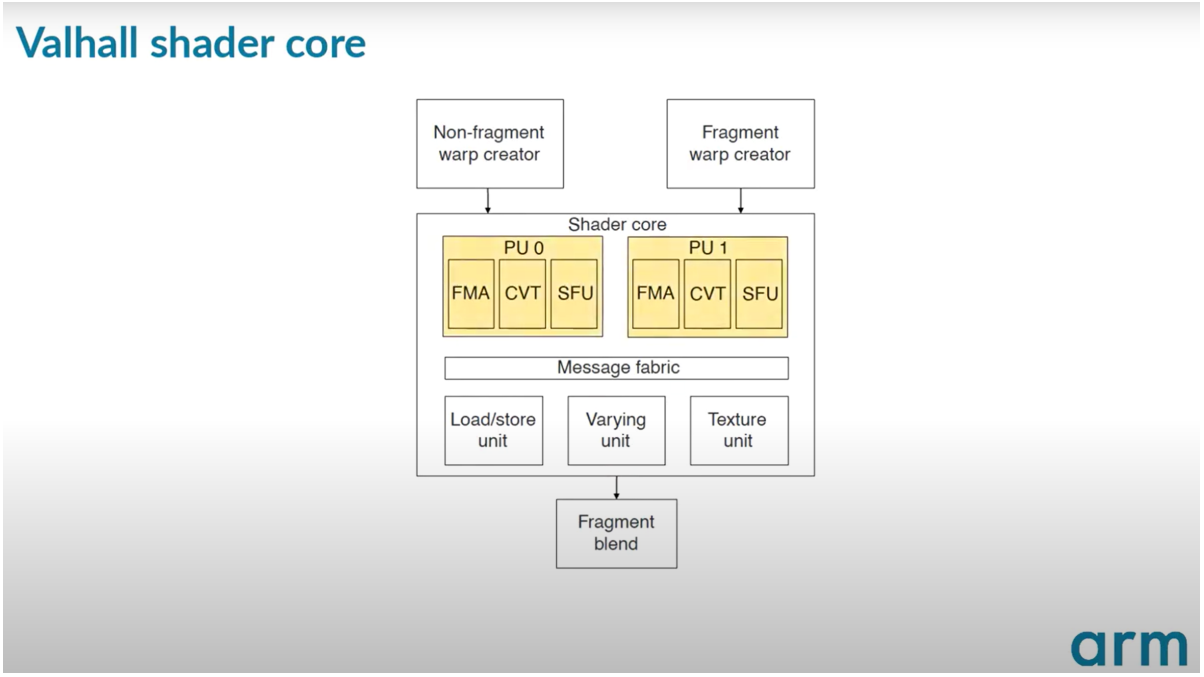
Shortest Path Cycles

程序最短控制流路径下的指令时钟数评估。

Longest Path Cycles

最长控制流路径下的指令时钟数评估。在静态分析中，不一定每次都能确保这个数据一定是最长的路径，比如一些通过通用变量控制循环迭代次数的情况，在静态分析无法确定循环次数的情况下，可能会显示"N/A"。

这部分的静态分析报告，会将不同单元分别做量化，找出最耗的那个单元就是Bounds，也就是瓶颈。



A

代表算术运算单元，包括（Valhall系列和5代之后）：

- FMA（Fused multiply accumulate）：乘法累加器，是主要的算术运算单元，用于浮点的运算，每个时钟可以执行一个32位或者两个16位的运算，大多数shader在算术上的瓶颈主要由这个的单元决定。
- CVT（Arithmetic conversion）：算术转换，例如数据类型的转换，整数加法。和FMA一样，一个时钟可以一个32位或者两个16位运算。

- SFU（Special functions unit）：特殊功能单元，比如三角函数，反三角函数，指数函数，对数函数，平方根，倒数等。因为每个单元有宽度为4的发射路径，因此一个时钟周期可以同时发出4个指令，在一个16-wide的wrap中，需要4个时钟周期才能执行完。

LS

Load/Store表示non-texture内存的读取和存储单元，包括buffer的读取，图片的读取，原子操作。（在Midgrard系列GPU中这部分管线单元还实现了插值器）。

单个cache line（64 byte）的数据可以在一个时钟周期内加载。如果所有线程束访问的数据都在同一个cache line，则数据可以在一个时钟周期内返回。

可以尽量利用SIMD和线程束的特性，来优化LS的使用。

例如：

- 使用vec来加载和存储数据
- 在线程束中，每个线程按顺序地址访问

V

Varying代表插值的单元。主要受插值数据量的影响，例如从Vert传递到Frag的数据。

在一个时钟周期中，每个插值单元在线程束的每个线程中可以插值32位的数据。

例如插值一个16位的vec4需要花费两个时钟周期。

T

Texturing表示纹理采样和过滤单元，主要跟贴图采样有关。

基础的性能是每个时钟周期可以执行四个双线性过滤。意味着，对于大部分格式的纹理，2x2的纹素块会在一个时钟周期内被采样。

但是对于一些不同的格式和过滤方式，会有一些差异，下图展示了详细的区别

Operation	Performance scaling
N x anisotropic filter	Up to x N
Trilinear filter	x2
3D format	x2
More than 32-bit per texel	x2
Filtered fp32	x2

例如一种4倍各向异性过滤的最坏情况，对RGBA16F使用三线性采样会被使用基础的双线性（4x2x2=16）多至少16倍的消耗。

以上不同类型的管线单元，均是并行的架构，因此可以利用好实现Latency Hiden。

Shader properties

这部分的数据主要反映shader程序在执行中会影响到性能的内容。

Has Uniform computation

对于draw call或者compute dispatch中的每个线程，如果这部分计算的结果值都是一样的，驱动会帮你优化这部分的计算，但是仍然有一定的消耗。所以尽可能的把这部分运算移动到CPU中进行。

Has side-effects

表示着色器程序有在固定管线之外的内存中有可见的副作用：

1. 存储缓冲区

2. 图像存储
3. 原子操作

有副作用的程序，没法做HSR等优化。

Has slow ray traversal

表示着色器至少使用了一次光追，会使得编译器回退到较慢的遍历行为。要避免这种情况，则必须每个rayQueryInitialize对应单个rayQueryProceed。

下面展示会造成这种情况的案例：

```
// Slow due to divergent initialization
if (cond)
    rayQueryInitialize(rq, params_1);
else
    rayQueryInitialize(rq, params_2);
```

```
// Slow due to multiple proceeds for a single initialize
rayQueryInitialize(rq, params);
if (cond)
    rayQueryProceed(rq);
rayQueryProceed(rq);
```

```
// Slow due to multiple proceeds for a single initialize
rayQueryInitialize(rq, params);
rayQueryProceed(rq);
rayQueryProceed(rq);
```

如果需要多次Proceed，则建议使用while循环

```
// Fast due to single initialize and single proceed call site
rayQueryInitialize(rq, params);
while (cond) {
    rayQueryProceed(rq);
}

```

如果是需要在某个条件下才Proceed，则需要把Initialize也一起放进条件作用域内

```
// Fast due to single initialize and proceed under the same
condition
if (cond) {
    rayQuery rq;
    rayQueryInitialize(rq, params);
    rayQueryProceed(rq);
}
```

Has slow shading rate

在某些Arm GPU上，如果着色率块的尺寸比2x2大，可能会造成可变速着色率（VRS）变慢。

- 着色率块(Shading Rate Block)：决定每个像素块内的像素将共享多少着色计算，比如2x2意味着4个像素共享一组着色计算。
- 可变速着色率(Variable Rate Shading)：通过对帧的不同区域使用不同的着色率来提高性能的一种功能

要避免在某一代GPU上出现这种情况，Arm建议使用`min()`确保着色率不会超过2。

Modifies coverage

这部分表示shader是否可以更改掩码，例如使用了discard。shader程序中如果有影响掩码值的行为，则只能使用Late ZS-test，无法使用Early-ZS的优化。

- 掩码： 用于决定哪些像素需要被渲染

NOTE：在api层面的行为，比如开启 alpha-to-coverage，也可以影响掩码，但是这里不考虑这种情况。

Use late ZS test

这部分展示shader是否有存在强制使用late ZS test的情况，譬如说在frag中写深度。这种情况会使得HSR失效，显著损失性能。

在明确知道行为安全的情况下，可以使用`early_fragment_tests`强制开启Early ZS。

NOTE：api层面的行为这里不考虑，比如关闭深度写入。

Read color buffer

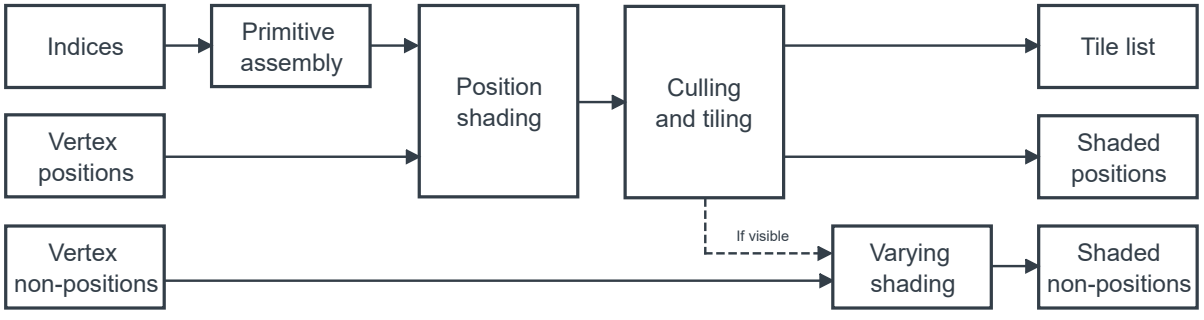
是否在shader程序中存在读取color buffer的行为，例如使用`gl_LastFragColorARM`来读取当前像素的颜色值。有这种行为会直接被认为是半透材质，不会使用HSR技术。

Vertex shader variants

从Bifrost系列以来，Arm GPU的顶点着色器均采样索引驱动架构（Index-Driven Vertex Shading，IDVS）。

在这个架构中，顶点着色器会被编译成两个二进制部分：

- position shader：只计算position的输出
- varying shader：计算剩下的非position的输出



position shader会每个顶点都执行，而varying shader只有在图元经过可见性剔除之后才会执行。

推荐的顶点属性流

基于以上架构的考虑，如果要达到最佳的带宽和性能利用率，必须调整顶点属性的内存布局来匹配Arm GPU对于顶点着色器的分阶段处理。

第一阶段的position计算，会用于图元的剔除。第二阶段的其余非position属性的计算，只会在可见图元的顶点中进行。

Arm建议将所有位置相关的输入属性，交错存储在一个内存范围，与非位置属性分开。这样可以确保在postion shading阶段只从DRAM中加载位置相关的属性，最小化缓存污染。

NOTE: 在Midgard系列的GPU中，不支持这部分优化。

下面展示了OpenGL ES中的顶点属性流报告

```
Recommended attribute streams
=============================

Position attributes
  - inPos (location=dynamic)

Non-position attributes
  - inTexCoord (location=1)
```

inPos / inTextCoord 均是具体的属性位置，与shader中的一致，括号显示的是绑定的位置。例如Unity中某个属性后面绑定的是：TEXCOORD0。

对于Vulkan，可能会显示为如下：

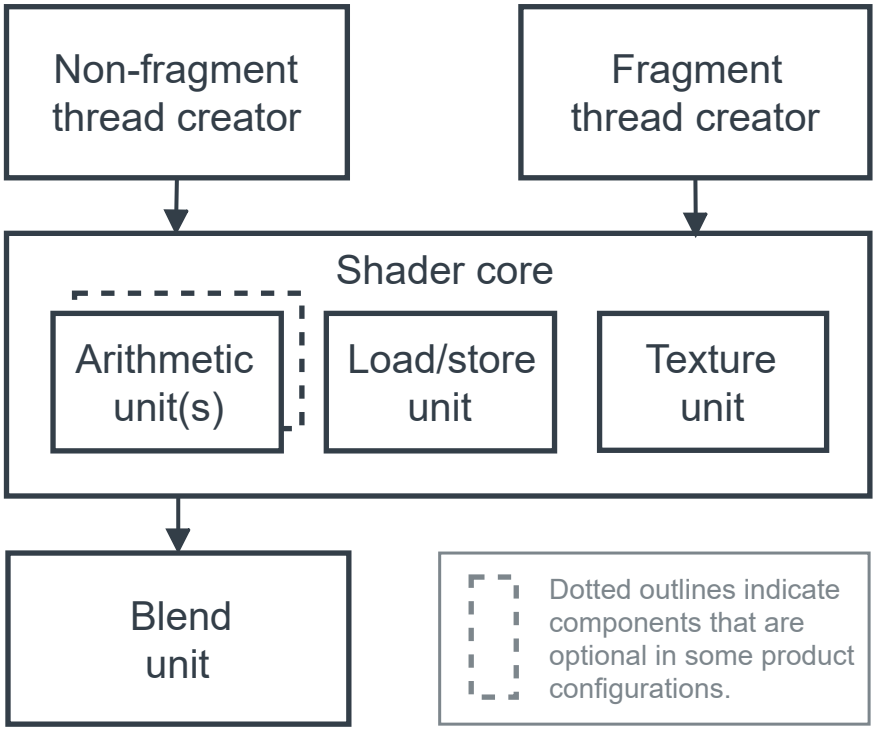
```
Position attributes
  - OpVariable %17 'offset' (location=2)
  - OpVariable %64 (location=0)
```

这部分通常只需要在定义Attributes的时候，显示的指定号属性的布局即可，例如：

```
struct VertexInput {
    float3 position : POSITION;
    float3 normal : NORMAL;
    float2 uv : TEXCOORD0;
};
```

GPU architectture

Midgard

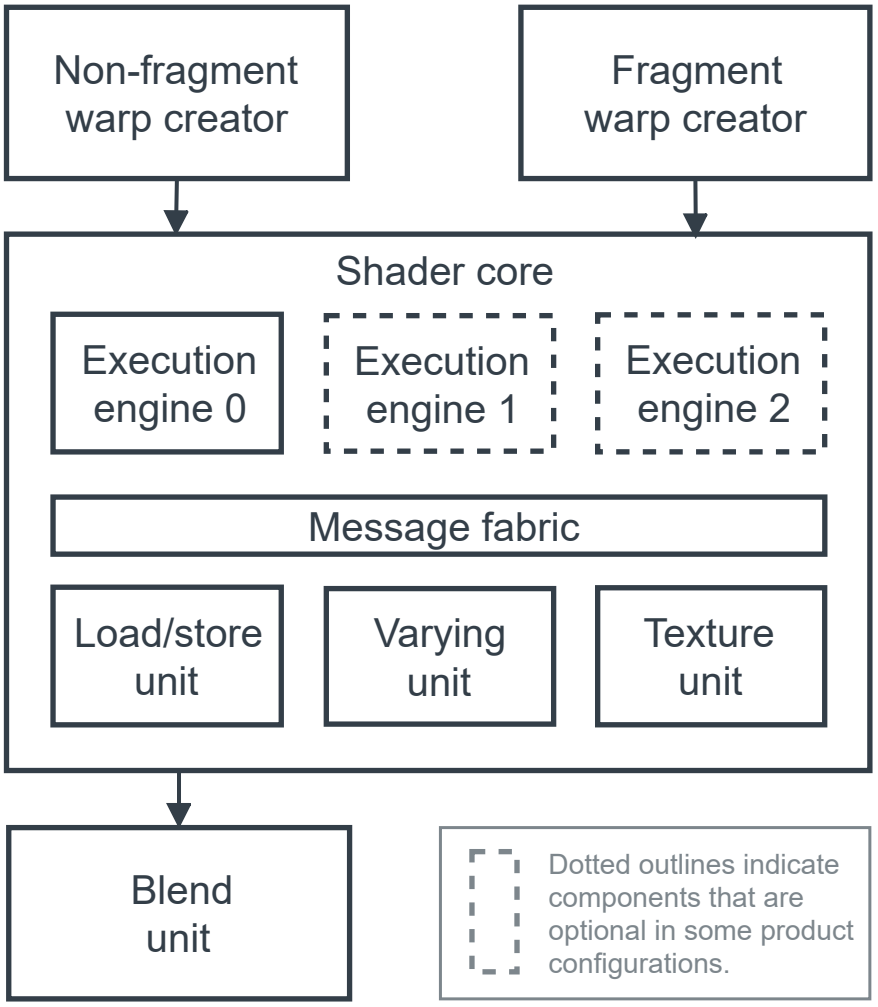


寄存器使用数的断点：

- 0-4 registers：满线程
- 5-8 regiters：50%
- 8-16 registers：25%

越多的线程数，可以保持GPU忙碌，建议Fragment维持0-4，其他stage维持5-8。

Bifrost



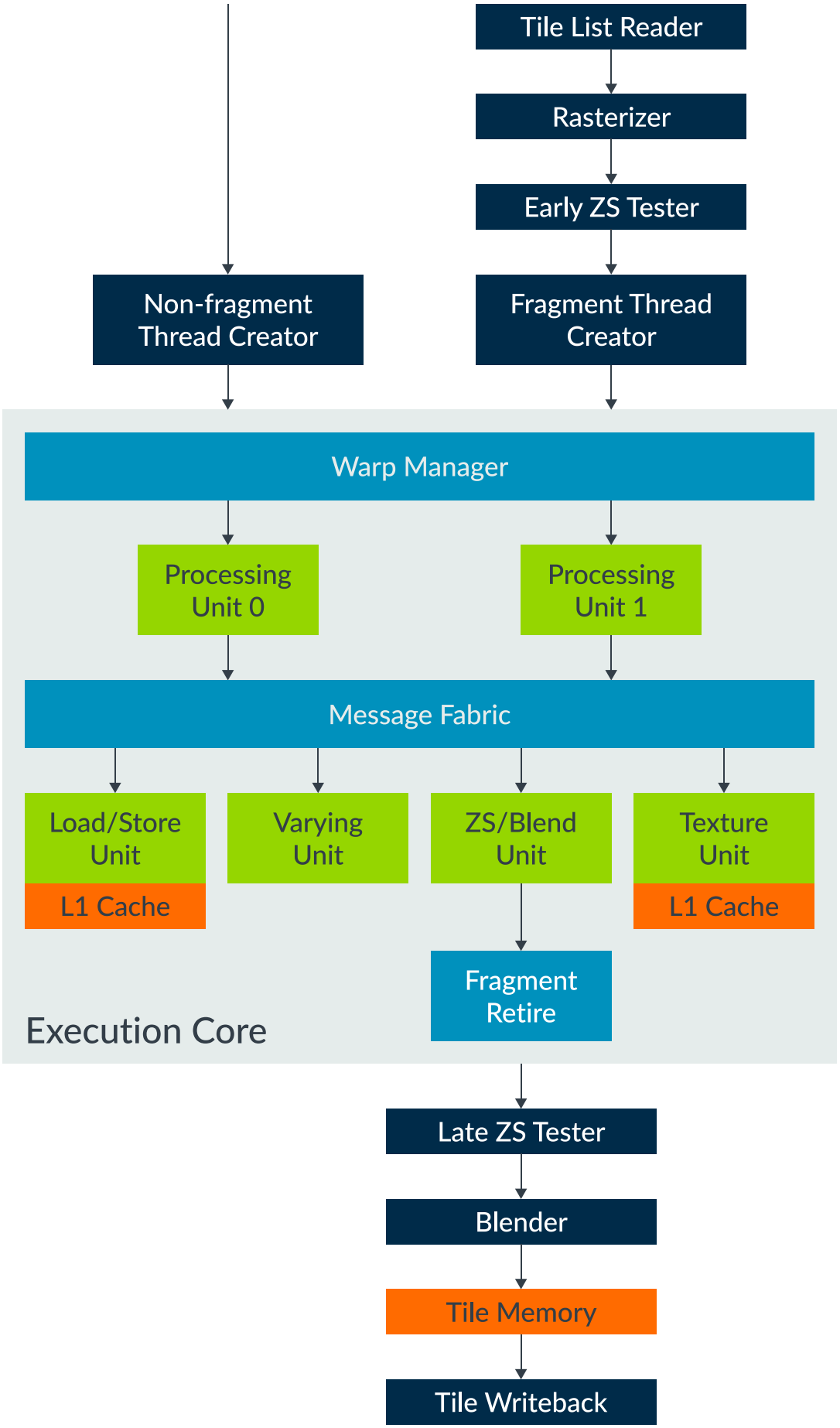
相比Midgard，Bifrost系列把Varying跟LS单元分离出来成为独立的单元。

寄存器使用数的断点：

- 0-32 registers：满线程
- 33-64 registers：50%

建议优化的目标是fragment使用0-32个寄存器。

Valhall



寄存器的断点和Bifrost一样。