PowerVR Low Level GLSL Optimisation

Filename	:	PowerVR Low level GLSL Optimisation
Version	:	PowerVR SDK REL_18.2@5224491a External Issue
Issue Date	:	23 Nov 2018
Author	:	Imagination Technologies Limited

Contents

List of Figures

1. Introduction

This document describes ways to optimise GLSL code for PowerVR Rogue architecture. These optimisations are low-level, and therefore can be used to get the last 10% of performance boost out from the hardware. Prior to using these techniques it is essential to make sure that the most optimal algorithms are being used, and that the GPU is well utilised.

The effectiveness of these optimisations may vary depending on the exact compiler architecture used. It is always worth checking if optimisations result in a performance improvement on the target platform. Throughout the document there are examples of USC instructions the GLSL code compiles to.

This document is based on the following:

Low-Level Thinking in High-Level Shading Languages, by Emil Persson.

Low-Level Shader Optimisation for Next-Gen and DX11, by Emil Persson.

2. Low Level Optimisations

2.1.   PowerVR Rogue USC

Generally, shader performance on PowerVR Rogue architecture GPUs depends on the number of cycles it takes to execute a shader. Depending on the configuration, the PowerVR Rogue architecture delivers a variety of options for executing multiple instructions in the USC ALU pipeline within a single cycle. It is possible to execute two F16 SOP instructions plus the F32 <-> F16 conversions, plus the mov/output/pack instruction in one cycle.

Alternatively, a developer could execute an FP32 MAD and an FP32/INT32 MAD/UNPACK instruction plus a test (conditional) instruction plus the mov/output/pack instruction in one cycle. If there is bitwise work to be done, it is possible to issue a bitwise SHIFT/COUNT, a bitwise logical operation, a bitwise shift, a test, and the mov/output/pack instructions in one cycle. It is also possible to execute a single complex operation (in other words, rcp) and a mov/output/pack instruction in one cycle.

Lastly, there is the option to execute an interpolate/sample instruction plus the usual mov/output/pack instruction in one cycle. As shown below, it is best to use all stages in one route in the pipeline below to fully utilise the ALU. Therefore it is advisable to arrange GLSL instructions that way.
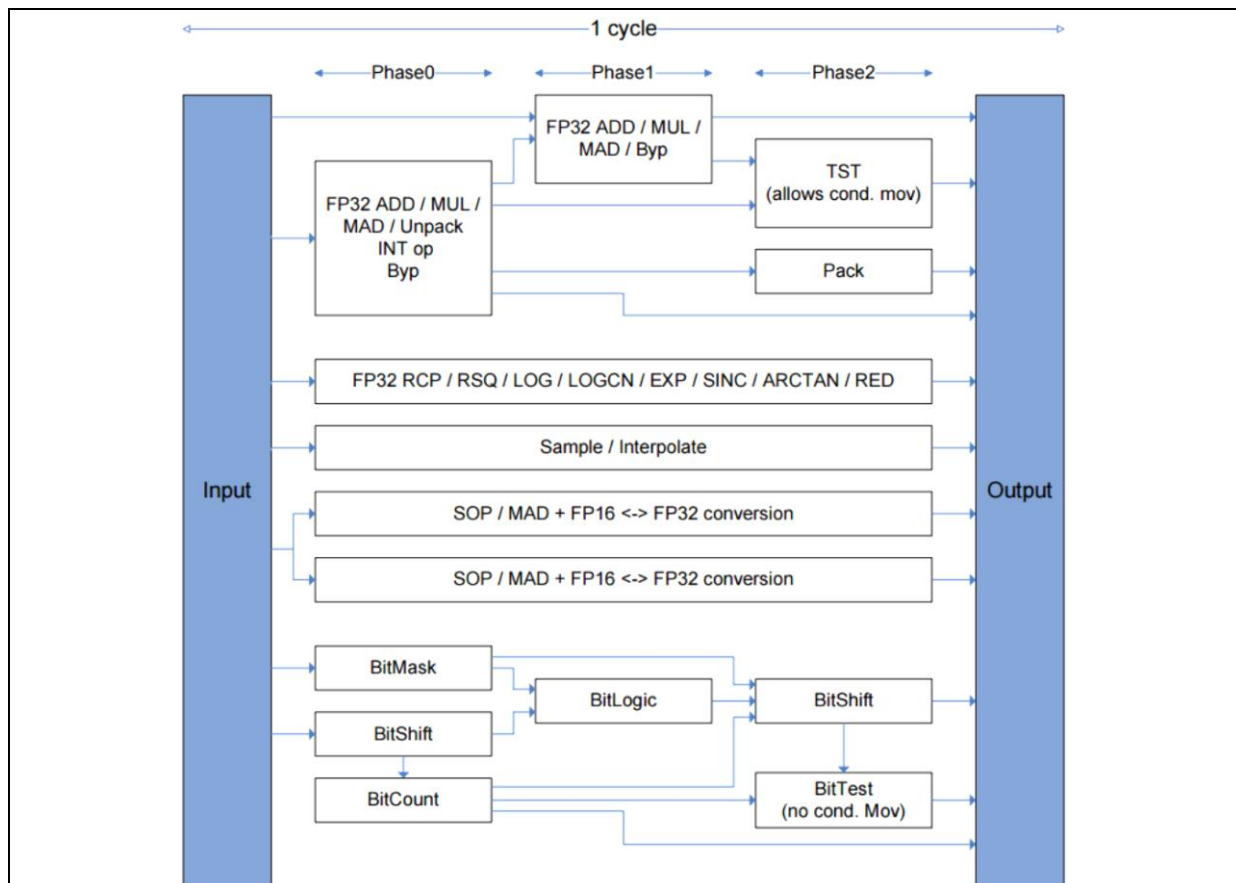

Figure 1. PowerVR Rogue USC

2.2.   Writing expressions in MAD form

In order to take effective advantage of the USC cores, it is essential to always write math expressions in Multiply-Add form (MAD form). For example, changing the following expression to use the MAD form results in a 50% cycle cost reduction.

```
fragColor.x = (t.x + t.y) * (t.x - t.y); //2 cycles
{sop, sop, sopmov}
{sop, sop}
-->
fragColor.x = t.x * t.x + (-t.y * t.y); //1 cycle
{sop, sop}
```

2.3.   Division

It is usually beneficial to write division math in reciprocal form. Also, finishing math expressions' simplification can yield additional performance gains.

```
fragColor.x = (t.x * t.y + t.z) / t.x; //3 cycles
{sop, sop, sopmov}
{frcp}
{sop, sop}
-->
fragColor.x = t.y + t.z * (1.0 / t.x); //2 cycles
{frcp}
{sop, sop}
```

2.4.   Sign

Originally the result of `sign(x)` would be:

```
-1 if x < 0,
1 if x > 0
```

and

```
0 if x == 0
```

However, if the last case is not needed it is better to use conditional form instead of `sign()`.

```
fragColor.x = sign(t.x) * t.y; //3 cycles
{mov, pck, tstgez, mov}
{mov, pck, tstgez, mov}
{sop, sop}
-->
fragColor.x = (t.x >= 0.0 ? 1.0 : -1.0) * t.y; //2 cycles
{mov, pck, tstgez, mov}
{sop, sop}
```

2.5.   Rcp/rsqrt/sqrt

On the PowerVR Rogue architecture the reciprocal operation is directly supported by an instruction.

```
fragColor.x = 1.0 / t.x; //1 cycle
{frcp}
```

The same is true with the `inversesqrt()` function.

```
fragColor.x = inversesqrt(t.x); //1 cycle
{frsq}
```

`sqrt()` on the other hand is implemented as: `1 / (1/sqrt(x))`
This results in a two cycle cost.

```
fragColor.x = sqrt(t.x); //2 cycles
{frsq}
{frcp}
```

A commonly used alternative: `x * 1/sqrt(x)` yields the same results.

```
fragColor.x = t.x * inversesqrt(t.x); //2 cycles
{frsq}
{sop, sop}
```

The only case when it is better to use the above alternative is if the result is tested. In this case the test instructions can fit into the second instruction.

```
fragColor.x = sqrt(t.x) > 0.5 ? 0.5 : 1.0; //3 cycles
{frsq}
{frcp}
{mov, mov, pck, tstg, mov}
-->
fragColor.x = (t.x * inversesqrt(t.x)) > 0.5 ? 0.5 : 1.0; //2 cycles
{frsq}
{fmul, pck, tstg, mov}
```

2.6.   Abs/Neg/Saturate

When working on PowerVR architecture, it is essential to take advantage of modifiers such as `abs()`, `neg()` and `clamp(…, 0.0, 1.0)` - also known as `saturate()` – being free in certain cases. `abs()` and `neg()` are free if they are used on an input to an operation, in which case they are turned into a free modifier by the compiler. `saturate()` on the other hand, turns into a free modifier when used on the output of an operation.

Note that complex and sampling/interpolation instructions are exceptions to this rule. In other words, `saturate()` is not free when used on a texture sampling output, or on a complex instruction output. When these functions are not used accordingly they may introduce additional mov instructions which may inflate the cycle count of the shaders.

It is also beneficial to use `clamp(…, 0.0, 1.0)` instead of `min(…, 1.0)` and `max(…, 0.0)`. This changes a test instruction into a saturate modifier.

```
fragColor.x = abs(t.x * t.y); //2 cycles
{sop, sop}
{mov, mov, mov}
-->
fragColor.x = abs(t.x) * abs(t.y); //1 cycle
{sop, sop}
```

```
fragColor.x = -dot(t.xyz, t.yzx); //3 cycles
{sop, sop, sopmov}
{sop, sop}
{mov, mov, mov}
-->
fragColor.x = dot(-t.xyz, t.yzx); //2 cycles
{sop, sop, sopmov}
{sop, sop}
```

```
fragColor.x = 1.0 - clamp(t.x, 0.0, 1.0); //2 cycles
{sop, sop, sopmov}
{sop, sop}
-->
fragColor.x = clamp(1.0 - t.x, 0.0, 1.0); //1 cycle
{sop, sop}
```

```
fragColor.x = min(dot(t, t), 1.0) > 0.5 ? t.x : t.y; //5 cycles
{sop, sop, sopmov}
{sop, sop}
{mov, fmad, tstg, mov}
{mov, mov, pck, tstg, mov}
{mov, mov, tstz, mov}
-->
fragColor.x = clamp(dot(t, t), 0.0, 1.0) > 0.5 ? t.x : t.y; //4 cycles
{sop, sop, sopmov}
{sop, sop}
{fmad, mov, pck, tstg, mov}
{mov, mov, tstz, mov}
```

However, the developer should be wary of complex functions, as they are translated into multiple operations and therefore in this case it matters where the modifiers are placed.

For example, `normalize()` is decomposed into:

```
vec3 normalize( vec3 v )
{
    return v * inverssqrt( dot( v, v ) );
}
```

In this case it is best to negate one of the inputs of the final multiplication rather than the inputs in all cases, or create a temporary negated input:

```
fragColor.xyz = -normalize(t.xyz); //6 cycles
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, mov}
-->
fragColor.xyz = normalize(-t.xyz); //7 cycles
{mov, mov, mov}
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, mov}
```

3. Transcendental functions

3.1. Exp/Log

On the PowerVR Rogue architecture the `2^n` operation is directly supported by an instruction.

```
fragColor.x = exp2(t.x); //1 cycle
{fexp}
```

The same is true with the `log2()` function.

```
fragColor.x = log2(t.x); //1 cycle
{flog}
```

`exp` is implemented as:

```
float exp2( float x )
{
  return exp2(x * 1.442695); //2 cycles
  {sop, sop}
  {fexp}
}
```

`log` is implemented as:

```
float log2( float x )
{
  return log2(x * 0.693147); //2 cycles
  {sop, sop}
  {flog}
}
```

`pow(x, y)` is implemented as:

```
float pow( float x, float y )
{
  return exp2(log2(x) * y); //3 cycles
  {flog}
  {sop, sop}
  {fexp}
}
```

3.2. Sin/Cos/Sinh/Cosh

`sin`, `cos`, `sinh`, and `cosh` on PowerVR architecture have a reasonably low cost of four cycles.

This is broken down as two cycles of reduction plus `fsinc` plus one conditional.

```
fragColor.x = sin(t.x); //4 cycles
{fred}
{fred}
{fsinc}
{fmul, mov} //+conditional
```

```
fragColor.x = cos(t.x); //4 cycles
{fred}
{fred}
{fsinc}
{fmul, mov} //+conditional
```

```
fragColor.x = cosh(t.x); //3 cycles
{fmul, fmul, mov, mov}
{fexp}
{sop, sop}
```

```
fragColor.x = sinh(t.x); //3 cycles
{fmul, fmul, mov, mov}
{fexp}
{sop, sop}
```

3.3. Asin/Acos/Atan/Degrees/Radians

If the math expressions' simplifications are completed, then these functions are usually not needed. Therefore they do not map to the hardware exactly. This means that these functions have very high cost, and should be avoided at all times.

asin() costs a massive 67 cycles.

```
fragColor.x = asin(t.x); //67 cycles
//USC code omitted due to length
```

acos() costs a massive 79 cycles.

```
fragColor.x = acos(t.x); //79 cycles
//USC code omitted due to length
```

atan() is still costly, but it could be used if needed.

```
fragColor.x = atan(t.x); //12 cycles (lots of conditionals)
//USC code omitted due to length
```

While degrees and radians take only one cycle, they can usually be avoided if only radians are used.

```
fragColor.x = degrees(t.x); //1 cycle
{sop, sop}
```

```
fragColor.x = radians(t.x); //1 cycle
{sop, sop}
```

4. Intrinsic functions

4.1. Vector*Matrix

The vector * matrix multiplication has quite a reasonable cost, despite the number of calculations that need to happen. Optimisations such as taking advantage of knowing that w is 1 however, do not reduce the cost.

```
fragColor = t * m1; //4x4 matrix, 8 cycles
{mov}
{wdf}
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}

fragColor.xyz = t.xyz * m2; //3x3 matrix, 4 cycles
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop}
```

4.2. Mixed Scalar/Vector math

`normalise()`/`length()`/`distance()`/`reflect()` functions usually contain a lot of function calls inside them such as `dot()`. A developer can take advantage of knowing how these functions are implemented.

For example, if it is known that two operations have a shared subexpression, then it is possible to reduce the cycle count. However, that only happens if the input order allows it.

```
fragColor.x = length(t-v); //7 cycles
fragColor.y = distance(v, t);
{sopmad, sopmad, sopmad, sopmad}
{sop, sop, sopmov}
{sopmad, sopmad, sopmad, sopmad}
{sop, sop, sopmov}
{sop, sop}
{frsq}
{frcp}
-->
fragColor.x = length(t-v); //9 cycles
fragColor.y = distance(t, v);
{mov}
{wdf}
{sopmad, sopmad, sopmad, sopmad}
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{frsq}
{frcp}
{mov}
```

Manually expanding these complex instructions can sometimes help the compiler optimise the code.

```
fragColor.xyz = normalize(t.xyz); //6 cycles
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, mov}
-->
fragColor.xyz = inversesqrt(dot(t.xyz, t.xyz)) * t.xyz; //5 cycles
{sop, sop, sopmov}
{sop, sop}
{frsq}
{sop, sop}
{sop, sop}
```

Also, in expanded form it is possible to take advantage of grouping vector and scalar instructions together.

```
fragColor.xyz = 50.0 * normalize(t.xyz); //7 cycles
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, fmul, mov, mov}
{sop, sop}
-->
fragColor.xyz = (50.0 * inversesqrt(dot(t.xyz, t.xyz))) * t.xyz; //6 cycles
{sop, sop, sopmov}
{sop, sop}
{frsq}
{sop, sop, sopmov}
{sop, sop}
{sop, sop}
```

Below is a list of what the complex instructions can be expanded to.

`cross()` can be expanded to:

```
vec3 cross( vec3 a, vec3 b )
{
  return vec3( a.y * b.z - b.y * a.z,
               a.z * b.x - b.z * a.x,
               a.x * b.y - b.y * a.y );
}
```

`distance()` can be expanded to:

```
float distance( vec3 a, vec3 b )
{
  vec3 tmp = a - b;
  return sqrt(dot(tmp, tmp));
}
```

`dot()` can be expanded to:

```
float dot( vec3 a, vec3 b )
{
  return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

`faceforward()` can be expanded to:

```
vec3 faceforward( vec3 n, vec3 I, vec3 Nref )
{
  if( dot(Nref, I) < 0 )
  {
    return n;
  }
  else
  {
    return -n:
  }
}
```

`length()` can be expanded to:

```
float length( vec3 v )
{
  return sqrt(dot(v, v));
}
```

`normalize()` can be expanded to:

```
vec3 normalize( vec3 v )
{
  return v / sqrt(dot(v, v));
}
```

`reflect()` can be expanded to:

```
vec3 reflect( vec3 N, vec3 I )
{
  return I - 2.0 * dot(N, I) * N;
}
```

`refract()` can be expanded to:

```
vec3 refract( vec3 n, vec3 I, float eta )
{
  float k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I));
    if (k < 0.0)
      return 0.0;
    else
      return eta * I - (eta * dot(N, I) + sqrt(k)) * N;
}
```

```
vec3 faceforward( vec3 n, vec3 I, vec3 Nref )
```

4.3.   Operation grouping

Generally it is beneficial to group scalar and vector operations separately. This way the compiler can pack more operations into a single cycle.

```
fragColor.xyz = t.xyz * t.x * t.y * t.wzx * t.z * t.w; //7 cycles
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop}
-->
fragColor.xyz = (t.x * t.y * t.z * t.w) * (t.xyz * t.wzx); //4 cycles
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{sop, sop}
```

5. FP16 overview

5.1.   FP16 precision and conversions

FP16 pipeline works well when reduced precision is sufficient. However, it is advisable to always check whether the optimisations resulted in precision artefacts. When 16 bit float precision hardware is available, and shaders use mediump, then 16<->32 bit conversion is free using modifiers. This is because the USC ALU pipeline includes it.

When shaders do not use the 16 bit instructions or the hardware does not contain a 16 bit float pipeline (such as with early Rogue hardware), then the instructions just run on the regular 32 bit pipeline. As a result, no conversions happen.

5.2.   FP16 SOP/MAD operation

SOP stands for Sum of Products .The FP16 SOP/MAD pipeline is one of the strongest points of the PowerVR ALU pipeline. If used correctly, it enables developers to pack more operations into a single cycle. This may result in increased performance and reduced power consumption.

The single cycle FP16 SOP/MAD operation can be described by the following pseudo code:

```
//Inputs:
a, b, c, d = any of {S0, S1, S2, S3, S4, S5}.

z = min(s1, 1 - s0)

e, f, g, h = any of {S0, S1, S2, S3, S4, S5} or z.

//Inputs only for the MAD pipeline:
v, w, x, y = any of {S0, S1, S2, S3, S4, S5}.

//Operations to be performed
jop = any of {add, sub, min, max, rsub, mad}
kop = any of {add, sub, min, max, rsub}

//Either use the MAD or the SOP pipeline
if (jop == mad)
{
  //two mad operations performed in parallel
  W0.e0 = a*e+v
  W1.e0 = b*f+x
  W0.e1 = c*g+w
  W1.e1 = d*h+y
}
else
{
  //multiply the SOP inputs and perform the desired operation on the result
  //performed in parallel
  j = (a * e) jop (b * f)
  k = (c * g) kop (d * h)

  //convert result to FP32
  //or keep the results as is
  if (rfmt(1) = 1)
  {
        w1 = toF32(k)
        w0 = toF32(j)
  }
  else if (rfmt(0) = 1) then
  {
        w0[31:16] = one of {j, a, b}
        w0[15:0]  = one of {k, c, d}
  }
  else
  {
        w0[31:16] = one of {k, c, d}
        w0[15:0]  = one of {j, a, b}
  }
}
```

It is also possible to apply various modifiers – such as `abs()`, `negate()`, `clamp()`, `oneminus()` - to the inputs and clamp to the outputs. See the next section for how to fully exploit the FP16 SOP/MAD pipeline.

5.3.   Exploiting the SOP/MAD FP16 pipeline

The PowerVR Rogue architecture has a powerful FP16 pipeline optimised for common graphics operations. This section describes how to take advantage of this. It is important to note that converting the inputs to FP16 and then converting the output to FP32 is free.

With SOP/MAD there are a number of options. In one cycle, it is possible to execute two SOPs or two MADs or one MAD + one SOP. Alternatively, it is possible to execute four FP16 MADs in one cycle.

Executing four MADs in one cycle:

```
fragColor.x = t.x * t.y + t.z;
fragColor.y = t.y * t.z + t.w;
fragColor.z = t.z * t.w + t.x;
fragColor.w = t.w * t.x + t.y;
{sopmad, sopmad, sopmad, sopmad}
```

SOP with a choice of an operation between the result of the multiplies:

```
fragColor.z = t.y * t.z OP t.w * t.x;
fragColor.w = t.x * t.y OP t.z * t.w;
```

where OP can be either an addition, a subtraction, a `min()` or a `max()`:

```
fragColor.z = t.y * t.z + t.w * t.x;
fragColor.z = t.y * t.z - t.w * t.x;
fragColor.z = min(t.y * t.z, t.w * t.x);
fragColor.z = max(t.y * t.z, t.w * t.x);
```

A developer can also apply either a negate, an `abs()` or a `clamp()` (`saturate`) to all the inputs.

```
fragColor.z = -t.y * abs(t.z) + clamp(t.w, 0.0, 1.0) * -t.x;
```

Finally, it is also possible to apply a `clamp()` (`saturate`) to the end result:

```
fragColor.z = clamp(t.y * t.z OP t.w * t.x, 0.0, 1.0);
fragColor.z = clamp(t.y * t.z + t.w, 0.0, 1.0);
```

After applying all this knowledge, we can show off the power of this pipeline by using everything in one cycle:

```
//1 cycle
mediump vec4 fp16 = t;
highp vec4 res;
res.x = clamp(min(-fp16.y * abs(fp16.z), clamp(fp16.w, 0.0, 1.0) * abs(fp16.x)), 0.0, 1.0);
res.y = clamp(abs(fp16.w) * -fp16.z + clamp(fp16.x, 0.0, 1.0), 0.0, 1.0);
fragColor = res;
{sop, sop}
```

6. Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp