Medium support in Mesa

Neil Roberts
Overview

• What is mediump?
• What does Mesa currently do?
• The plan
• Reducing conversion operations
• Changing types of variables
• Folding conversions
• Testing
• Code
• Questions?
What is medium?
• Only in GLSL ES
• Available since the first version of GLSL ES.
• Used to tell the driver an operation in a shader can be done with lower precision.
• Some hardware can take advantage of this to trade off precision for speed.
• For example, an operation can be done with a 16-bit float:

**32-bit float**

- **sign bit**
- **exponent bits**
- **fraction bits**

Largest number approximately $3 \times 10^{38}$

Approximately 7 decimal digits of accuracy

**16-bit float**

- **sign bit**
- **exponent bits**
- **fraction bits**

Largest number 65504

Approximately 3 decimal digits of accuracy
• GLSL ES has three available precisions:
  • lowp, mediump and highp
  • The spec specifies a minimum precision for each of these.
  • highp needs 16-bit fractional part.
    • It will probably end up being a single-precision float.
  • mediump needs 10-bit fractional part.
    • This can be represented as a half float.
  • lowp has enough precision to store 8-bit colour channels.
• The precision does not affect the visible storage of a variable.
  • For example a mediump float will still be stored as 32-bit in a UBO.
  • Only operations are affected.
• The precision requirements are only a minimum.
  • Therefore a valid implementation could be to just ignore the precision and do every operation at highp.
  • This is effectively what Mesa currently does.
• The precision for a variable can be specified directly:

uniform mediump vec3 rect_color;

• Or it can be specified as a global default for each type:

precision mediump float;
uniform vec3 rect_color;
- The compiler specifies global defaults for most types except floats in the fragment shader.
- In GLSL ES 1.00 high precision support in fragment shaders is optional.
• The precision of operands to an operation determine the precision of the operation.
• Almost works like automatic float to double promotion in C.

```c
mediump float a, b;
highp float c = a * b;
```
• The precision of operands to an operation determine the precision of the operation.
• Almost works like automatic float to double promotion in C.

```plaintext
mediump float a, b;
highp float c = a * b;
```

This operation can be done in mediump

All operands are mediump.
• The precision of operands to an operation determine the precision of the operation.
• Almost works like automatic float to double promotion in C.

```c
mediump float a, b;

highp float c = a * b;
```

- Precision of result doesn’t matter
- This operation can be done in mediump
- All operands are mediump.
• Another example

```cpp
mediump float a, b;
highp float c;

mediump float r = c * (a * b);
```
• Another example

```c
mediump float a, b;
highp float c;
mediump float r = c * (a * b);
```

This operation can still be done in mediump
• Another example

```c
mediump float a, b;
highp float c;

mediump float r = c * (a * b);
```

This outer operation must be done at highp
This operation can still be done in mediump

• Corner case
  • Some things don’t have a precision, eg constants.

```c
mediump float diameter;
float circ = diameter * 3.141592;
```
• Corner case
  • Some things don’t have a precision, eg constants.

```cpp
medump float diameter;

float circ = diameter * 3.141592;
```

Constants have no precision
• Corner case
  • Some things don’t have a precision, eg constants.

```cpp
mediump float diameter;

float circ = diameter * 3.141592;
```

Precision of multiplication is mediump anyway because one of the arguments has a precision

Constants have no precision
• Extreme corner case
  • Sometimes none of the operands have a precision.

uniform bool should_pi;

mediump float result =
  float(should_pi) * 3.141592;
• Extreme corner case
  • Sometimes none of the operands have a precision.

```glsl
uniform bool should_pi;

mediump float result = float(should_pi) * 3.141592;
```

Neither operand has a precision
• Extreme corner case
  • Sometimes none of the operands have a precision.

```plaintext
uniform bool should_pi;

mediump float result = float(should_pi) * 3.141592;
```

- Precision of operation can come from outer expression, even the lvalue of an assignment
- Neither operand has a precision
What does Mesa currently do?
• Mesa already has code to parse the precision qualifiers and store them in the IR tree.
• These currently aren’t used for anything except to check for compile-time errors.
  • For example redeclaring a variable with a different precision.
• In desktop GL, the precision is always set to NONE.
• The precision usually doesn’t form part of the glsl_type.
• Instead it is stored out-of-band as part of the ir_variable.
enum {
    GLSL_PRECISION_NONE = 0,
    GLSL_PRECISION_HIGH,
    GLSL_PRECISION_MEDIUM,
    GLSL_PRECISION_LOW
};
class ir_variable : public ir_instruction {
  /* ... */
public:
  struct ir_variable_data {
    /* ... */
    /**
     * Precision qualifier.
     *
     * In desktop GLSL we do not care about precision qualifiers at all, in fact, the spec says that precision qualifiers are ignored.
     *
     * To make things easy, we make it so that this field is always GLSL_PRECISION_NONE on desktop shaders. This way all the variables have the same precision value and the checks we add in the compiler for this field will never break a desktop shader compile.
     */
    unsigned precision:2;
    /* ... */
  };
  /* ... */
};
• However this gets complicated for structs because members can have their own precision.

uniform block {
    mediump vec3 just_a_color;
    highp mat4 important_matrix;
} things;

• In that case the precision does end up being part of the glsl_type.
The plan
• The idea is to lower medium operations to float16 types in NIR.
• We want to lower the actual operations instead of the variables.
• This needs to be done at a high level in order to implement the spec rules.
• Work being done by Hyunjun Ko and myself and Igalia.
• Working on behalf of Google.
• Based on / inspired by patches by Topi Pohjolainen.
• Aiming specifically to make this work on the Freedreno driver.
• Most of the work is reusable for any driver though.
• Currently this is done as a pass over the IR representation.
uniform mediump float a, b;

void main()
{
    gl_FragColor.r = a / b;
}
uniform mediuim float a, b;

void main()
{
    gl_FragColor.r = a / b;
}

These two variables are mediuim
uniform mediump float a, b;

void main()
{
    gl_FragColor.r = a / b;
}

These two variables are mediump

So this division can be done at medium precision
• We only want to lower the division operation without changing the type of the variables.
• The lowering pass will add a conversion to float16 around the variable dereferences and then add a conversion back to float32 after the division.
• This minimises the modifications to the IR.
• IR tree before lowering pass

(assign (x) (var_ref gl_FragColor)
  (swiz x (swiz xxxx (expression float /
    (var_ref a)
    (var_ref b))))))
• IR tree before lowering pass

(assign (x) (var_ref gl_FragColor)
  (swiz x (swiz xxxx (expression float /
    (var_ref a)
    (var_ref b)))))

division operation
• IR tree before lowering pass

\[
\text{(assign } (x) (\text{var_ref } \text{gl_FragColor}) \text{)} \text{ (swiz } x \text{ (swiz } \text{xxxx } (\text{expression } \text{float } / (\text{var_ref } a) (\text{var_ref } b)))\text{)}\]

division operation

\text{type is 32-bit float}
• Lowering pass finds sections of the tree involving only mediump/lowp operations.
• Adds f2f16 conversion after variable derefs
• Adds f2f32 conversion at root of lowered branch
• IR tree after lowering pass

(assign  (x) (var_ref gl_FragColor)
  (expression float f162f
   (swiz x (swiz xxxx
    (expression float16_t /
     (expression float16_t f2f16
      (var_ref a))
     (expression float16_t f2f16
      (var_ref b)))))))
• IR tree after lowering pass

```
(assign (x) (var_ref gl_FragColor)
  (expression float f162f
    (swiz x (swiz xxxx
      (expression float16_t /
        (expression float16_t f2f16
          (var_ref a))
        (expression float16_t f2f16
          (var_ref b))))))
```

*each var_ref is converted to float16*
IR tree after lowering pass

(assign (x) (var_ref gl_FragColor)
  (expression float f162f
    (swiz x (swiz xxxx
      (expression float16 t /
        (expression float16_t f2f16
          (var_ref a))
        (expression float16_t f2f16
          (var_ref b))))))))

division operation is done in float16
• IR tree after lowering pass

(assign (x) (var_ref ql_FragColor) (expression float f162f (swiz x) (swiz xxxx (expression float16_t / (expression float16_t f2f16 (var_ref a)) (expression float16_t f2f16 (var_ref b))))))

Result is converted back to float32 before storing in var
Reducing conversion operations
• This will end up generating a lot of conversion operations.
• Worse:

```cpp
precision mediump float;
uniform mediump float a;

void main()
{
    float scaled = a / 5.0;
    gl_FragColor.r = scaled + 0.5;
}
```
• This will end up generating a lot of conversion operations.
• Worse:

```cpp
precision mediump float;
uniform mediump float a;

void main()
{
    float scaled = a / 5.0;  // operation will be done in mediump then converted back to float32 to store in the variable

    gl_FragColor.r = scaled + 0.5;
}
```
• This will end up generating a lot of conversion operations.
• Worse:

    precision mediump float;
    uniform mediump float a;

    void main()
    {
        float scaled = a / 5.0;
        gl_FragColor.r = scaled + 0.5;
    }

then the result will be immediately converted back to float16 for this operation
• Resulting NIR

    vec1 32 ssa_1 = deref_var &a (uniform float)
    vec1 32 ssa_2 = intrinsic load_deref (ssa_1)
    vec1 16 ssa_3 = f2f16 ssa_2
    vec1 16 ssa_6 = fdiv ssa_3, ssa_20
    vec1 32 ssa_7 = f2f32 ssa_6
    vec1 16 ssa_8 = f2f16 ssa_7
    vec1 32 ssa_9 = f2f32 ssa_8
    vec1 16 ssa_10 = f2f16 ssa_9
    vec1 16 ssa_13 = fadd ssa_10, ssa_22
• Resulting NIR

```
vec1 32 ssa_1 = deref_var &a (uniform float)
vec1 32 ssa_2 = intrinsic load_deref (ssa_1)
vec1 16 ssa_3 = f2f16 ssa_2
vec1 16 ssa_6 = fdiv ssa_3, ssa_20
vec1 32 ssa_7 = f2f32 ssa_6
vec1 16 ssa_8 = f2f16 ssa_7
vec1 32 ssa_9 = f2f32 ssa_8
vec1 16 ssa_10 = f2f16 ssa_9
vec1 16 ssa_13 = fadd ssa_10, ssa_22
```

Lots of redundant conversions!
• There is a NIR optimisation to remove redundant conversions
• Only enabled for GLES because converting f32→f16→f32 is not lossless
Changing types of variables
• Normally we don’t want to change the type of variables
• For example, this would break uniforms because they are visible to the app
• Sometimes we can do it anyway though depending on the hardware
• On Freedreno, we can change the type of the fragment outputs if they are mediump.
• `gl_FragColor` is declared as mediump by default
• The variable type is not user-visible so it won’t break the app.
• This removes a conversion.
• We have a specific pass for Freedreno to do this.
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 16 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_4 = load_const (0x00000001 /* 0.000001 */)
vec1 16 ssa_5 = intrinsic load_uniform (ssa_4) (0, 0, 0)
vec1 16 ssa_7 = frcp ssa_5
vec1 16 ssa_8 = fmul ssa_2, ssa_7
vec1 32 ssa_9 = f2f32 ssa_8
vec4 32 ssa_10 = vec4 ssa_9, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_10, ssa_1) (0, 15, 0, 160)
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 16 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_4 = load_const (0x00000001 /* 0.000000 */)
vec1 16 ssa_5 = intrinsic load_uniform (ssa_4) (0, 0, 0)
vec1 16 ssa_7 = frcp ssa_5
vec1 16 ssa_8 = fmul ssa_2, ssa_7
vec1 32 ssa_9 = f2f32 ssa_8 removes this conversion
vec4 32 ssa_10 = vec4 ssa_9, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_10, ssa_1) (0, 15, 0, 144)
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 16 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_4 = load_const (0x00000001 /* 0.000000 */)
vec1 16 ssa_5 = intrinsic load_uniform (ssa_4) (0, 0, 0)
vec1 16 ssa_7 = frcp ssa_5
vec1 16 ssa_8 = fmul ssa_2, ssa_7
vec4 16 ssa_10 = vec4 ssa_8, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_10, ssa_1) (0, 15, 0, 160)

use 16-bit output directly
Folding conversions
• Consider this simple fragment shader

uniform highp float a, b;

void main()
{
    gl_FragColor.r = a / b;
}
Consider this simple fragment shader:

```glsl
uniform highp float a, b;
void main()
{
    gl_FragColor.r = a / b;
}
```

The operation is using `highp`.
• Consider this simple fragment shader

```c
uniform highp float a, b;
void main()
{
    gl_FragColor.r = a / b;
}
```

`gl_FragColor` will be converted to a 16-bit output.
• This can generate an IR3 disassembly like this:

```
mov.f32f32 r0.x, c0.y
(rpt5)nop
rcp r0.x, r0.x
(ss)mul.f r0.x, c0.x, r0.x
(rpt2)nop
cov.f32f16 hr0.x, r0.x
```
• This can generate an IR3 disassembly like this:

mov.f32f32 r0.x, c0.y  
(rpt5)nop  
rcp r0.x, r0.x  
(ss)mul.f r0.x, c0.x  
(rpt2)nop  
cov.f32f16 hr0.x, r0.x

32-bit float registers for the multiplication
• This can generate an IR3 disassembly like this:

```assembly
mov.f32f32 r0.x, c0.y
(rpt5)nop
rcp r0.x, r0.x
(ss)mul.f r0.x, c0.x, r0.x
(rpt2)nop
cov.f32f16 hr0.x r0.x
```

result is converted to half-float for output
• This last conversion shouldn’t be necessary.
• Adreno allows the destination register to have a different size from the source registers.
• We can fold the conversion directly into the multiplication.
• We have added a pass on the NIR that does this folding.
• It requires changes the NIR validation to allow the dest to have a different size.
• Only enabled for Freedreno.
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 32 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_3 = load_const (0x00000001 /* 0.000000 */)
vec1 32 ssa_4 = intrinsic load_uniform (ssa_3) (0, 0, 0)
vec1 32 ssa_5 = frcp ssa_4
vec1 32 ssa_6 = fmul ssa_2, ssa_5
vec1 16 ssa_7 = f2f16 ssa_6
vec4 16 ssa_8 = vec4 ssa_7, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_8, ssa_1) (0, 15, 0, 0)
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 32 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_3 = load_const (0x00000001 /* 0.000000 */)
vec1 32 ssa_4 = intrinsic load_uniform (ssa_3) (0, 0, 0)
vec1 32 ssa_5 = frcp ssa_4
vec1 32 ssa_6 = fmul ssa_2, ssa_5
vec1 16 ssa_7 = f2f16 ssa_6 \text{ remove this conversion }
vec4 16 ssa_8 = vec4 ssa_7, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_8, ssa_1) (0, 15, 0, 144)
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 32 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_3 = load_const (0x00000001 /* 0.000000 */)
vec1 32 ssa_4 = intrinsic load_uniform (ssa_3) (0, 0, 0)
vec1 32 ssa_5 = frcp ssa_4
vec1 16 ssa_6 = fmul ssa_2, ssa_5

vec4 16 ssa_8 = vec4 ssa_6, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_8, ssa_1) (0, 15, 0, 144)
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 32 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_3 = load_const (0x00000001 /* 0.000000 */)
vec1 32 ssa_4 = intrinsic load_uniform (ssa_3) (0, 0, 0)
vec1 32 ssa_5 = frcp ssa_4
vec1 16 ssa_6 = fmul ssa_2, ssa_5
vec4 16 ssa_8 = vec4 ssa_6, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_8, ssa_1) (0, 15, 0, 144)
vec1 32 ssa_1 = load_const (0x00000000 /* 0.000000 */)
vec1 32 ssa_2 = intrinsic load_uniform (ssa_1) (0, 0, 0)
vec1 32 ssa_3 = load_const (0x00000001 /* 0.000000 */)
vec1 32 ssa_4 = intrinsic load_uniform (ssa_3) (0, 0, 0)
vec1 32 ssa_5 = frcp ssa_4
vec1 16 ssa_6 = fmul ssa_2, ssa_5

vec4 16 ssa_8 = vec4 ssa_6, ssa_0.y, ssa_0.z, ssa_0.w
intrinsic store_output (ssa_8, ssa_1) (0, 15, 0, 144)
Testing
• We are writing Piglit tests that use mediump
• Most of them check that the result is less accurate than if it was done at highp
• That way we catch regressions where we break the lowering
• These tests couldn’t be merged into Piglit proper because not lowering would be valid behaviour.
Code
• The code is at gitlab.freedesktop.org/zzoon on the mediump branch
• There are also merge requests (1043, 1044, 1045).
• Piglit tests are at: https://github.com/Igalia/piglit/
  • branch nroberts/wip/mediump-tests
Questions?