Mediump support in Mesa

Neil Roberts

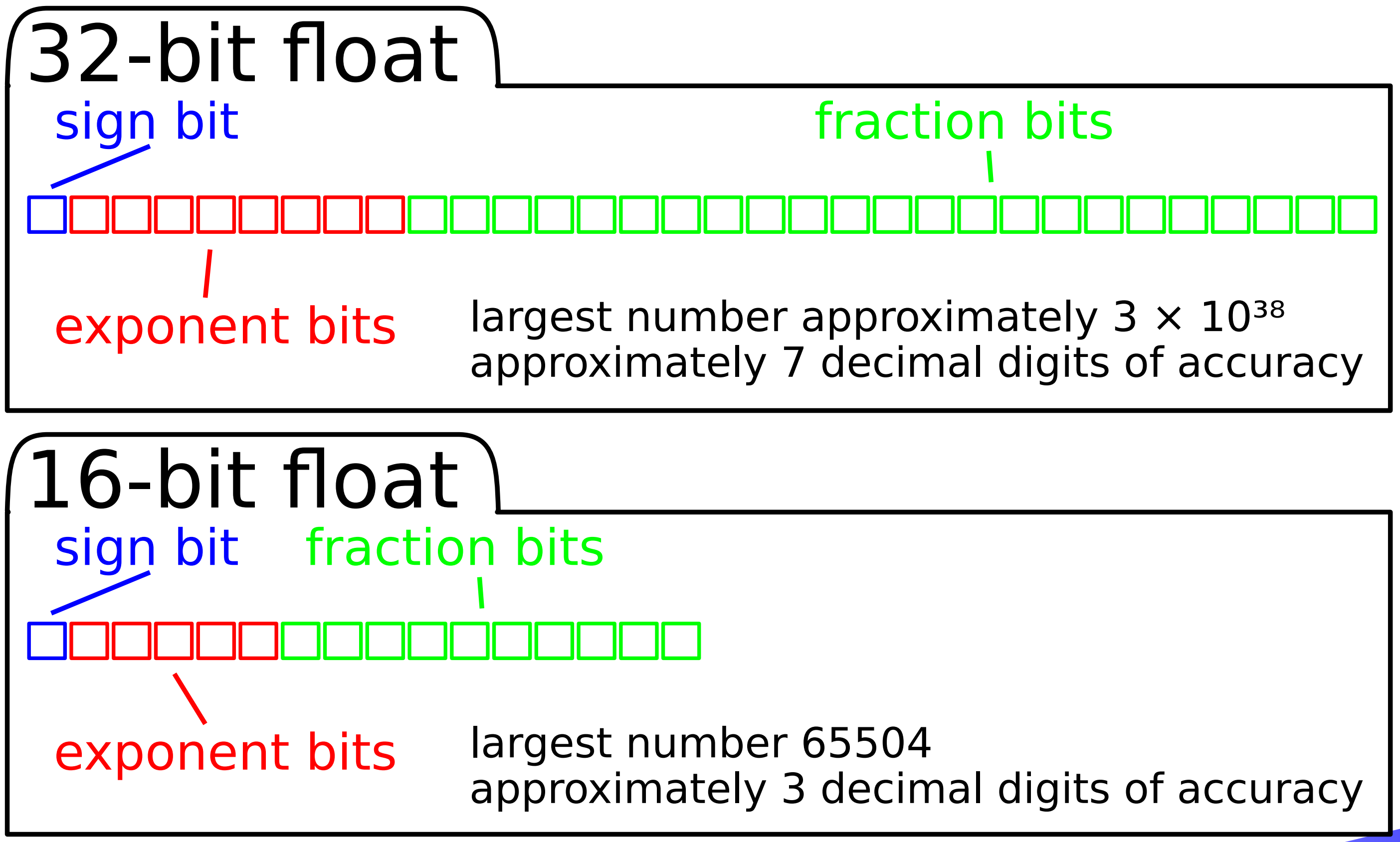
Overview

- Introduction
- History
- Examples
- Current status
- Future
- Questions

What is mediump?

- Only in GLSL ES
- Available since the first version of OpenGL 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:



- 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.

```
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.

```
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.

```
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

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

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

- Another example

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

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

This operation can still
be done in mediump

- Another example

```
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.

```
mediump float diameter;

float circ = diameter * 3.141592;
```

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

```
mediump float diameter;

float circ = diameter * 3.141592;
```

Constants have no precision

- Corner case
  - Some things don't have a precision, eg constants.

```
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.

```
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.

```
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

How does it work?

Mesa already keeps track of the precision of variables.

We added some extra code to handle this in more corner cases.
(default precision of struct members, return values of functions).

The idea is to lower mediump 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.

Currently this is done as a pass over the IR representation.

```
uniform mediump float a, b;

void main()
{
    gl_FragColor.rgba = vec4(a / b);
}
```

These two variables are mediump

```
uniform mediump float a, b;

void main()
{
    gl_FragColor.rgba = vec4(a / b);
}
```

These two variables are mediump

```
uniform mediump float a, b;

void main()
{
    gl_FragColor.rgba = vec4(a / b);
}
```

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.

Questions?